Application Note 106
Software Customization for the 6x86™ Family

Applicable to the Entire Line of 6x86™ Cyrix® Processors.
App Note Rev 1.7.

APPLICATION NOTE 106

Software Customization for the 6x86 Family

1. Introduction

This document provides guidance in writing effective software for the Cyrix 6x86™ and 6x86MX™ processors. Differences between 6x86 and 6x86MX CPUs are listed in Appendix A. The Cyrix Software Developer web site (www.cyrix.com) provides current information and code examples on topics such as CPU Detection and Cache Line Locking.

1.1. No Instruction-Pair Optimization Needed

It is important to point out, that the Cyrix 6x86 and 6x86MX require no instruction-pair optimization as does the Pentium® CPU. The reason is that the execution pipelines in the 6x86/6x86MX are more balanced than those in the Pentium. Both legacy 16-bit code, and Pentium optimized 32-bit code pass through the Cyrix execution pipelines with the same efficiency.

2. 6x86 Family Coding Suggestions

The suggestions in this section will help both the programmer to produce higher performance software when using the 6x86 and the 6x86MX processors.

2.1. Remove Address Generation Interlocks

Separate Address Generation Interlocks (AGIs) by 2 cycles (2 to 4 instructions).

For example, avoid:

```
add cx,bx
add dx,[cx]
```

In this example *cx* in the second instruction cannot be used until the *cx* in the first instruction is finished updating. This coding will produce a two clock bubble which results in a four instruction penalty.

Another common example to avoid:

```
mov eax,[eax]
mov eax,[eax]
mov eax,[eax]
cmp eax,whatever
```

2.2. Don’t Move Variables Into Registers For Speed

It has been a common practice to place frequently used variables into the CPU general purpose registers (EAX, EBX, ECX, EDX). This was done because register access was the far quicker than cache access. This practice is not needed for 6x86 architecture CPUs. The 6x86 L1 cache access time is the same as register access time. This also reduces “register pressure” limitations in the compiler.

2.3. Avoiding RISC-like Instruction Coding, Complex is Better

For the CPUs that have pairing constraints, optimization of code consists of partly of turning CISC instructions into RISC-like instruction equivalents. The RISC equivalents increase code size that end up taking more space in the cache.

The 6x86 and 6x86MX are designed to accelerate complex x86 instructions and do not have pair optimization constraints. For this reason, it is recommended not to break complex instructions into RISC equivalents for 6x86 family of CPUs.

The instruction below takes only one clock cycle to execute.

```
add [mem], eax
```

If this instruction is broken into three RISC-like instructions three clock cycles are required:

```
mov ebx, [mem]
add ebx, eax
mov [mem], ebx
```

As another example, a loop instruction takes only one clock cycle:

```
loop foo
```

Using the following RISC-like coding takes two clock cycles:

```
dcc ecx
jle foo
```

2.4. RAW Dependencies

Some Read-After-Write (RAW) dependencies can cause a stall.

For example avoid:

```
add   [mem],bx
add   cx,[mem]
```

The second add instruction stalls because the memory access [mem] must wait for the first instruction to complete updating.

2.5. Don't Make Calls Without a Matching RET.

Making a CALL, without a return, will result in branch miss predictions. This will hinder speculative execution.

For example avoid:

```
pushoffset Main_PGM
jmp SubRoutine

Main_PGM proc
          ...
Main_PGM endp

SubRoutine_PGM proc
          ...
ret
SubRoutine endp
```

Instead call the subroutine.

```
call SubRountine_PGM
```

rather than:

```
push offset main_PGM
jmp SubRoutine
```

2.6. Don’t Optimize for the FPU Like a Pentium

The 6x86 FPU is not pipelined. It can only execute one FPU instruction at a time. This is typically an issue when hand-coding FPU instructions in assembly language. Most compilers do not make highly optimized FPU code. Use 486 FPU optimization compiler switches.

The FXCH instruction is not paired with other FPU instructions on the 6x86. The 6x86 FXCH instruction takes three clocks and two clocks on the 6x86MX.

2.7. Mix Integer and FPU Instructions

The 6x86 CPU will execute integer instructions and FPU instructions at the same time.

2.8. Mixing 16 And 32 Bit Code

There is no penalty for mixing 16 and 32 bit code so there is no penalty for one prefix.

2.8.1 Prefix Issues

There is a one clock penalty for two to six prefix's (with the following caveats):

A one clock penalty also occurs if an instruction has a(n):

- Address size override prefix with a displacement in the address.
- Operand size override prefix and immediate operand.
- Decode length of an instruction is greater than valid length of the instruction queue

Only the first instruction is decoded when:

- The second instruction has more than one prefix
- Length of the first instruction more than six bytes

- The first instruction is six bytes long and the second has a prefix
- The length of the first and second instruction together is greater than the valid instruction queue length.
- The last byte of instruction one is 0F and the second has no prefix
- The first instruction has taken a predicted change of flow.

2.9. Branch and FPU Optimization

The CPU will speculatively execute up to four FPU or jump (JMP or Jcc) instructions at any one time. The fifth FPU instruction or jump will cause a stall until one of the earlier jump or FPU instructions reaches completion.

2.9.1 No Penalty When Using Partial Registers

The 6x86 core has no problem with mixing code that uses 8, 16 or 32 parts of the same register on successive instructions.

For example:

```
mov   bh,   [mem1]
add   ebx,  [mem2]
```

does not cause a stall condition due to using register BH then EBX.

2.9.2 Write Gathering For Video Memory or Other Memory Mapped I/O

The 6x86 and 6x86MX processors have the ability to be programmed on a region by region basis for optimization of different memory types. The Data books and BIOS writers guides detail all the available options.

Application Note 103 *6x86MX BIOS Writers Guide* defines Region 7 for memory above physical memory. Typically this is where the frame buffer resides. Write combining is suggested to be enabled for Region 7. If Region 7 has write combining enabled, video memory in this region will be optimized.

2.10. Self-Modifying Code Should Be Avoided

Self-modifying code can have significant negative performance impact due to the need to flush CPU state information to keep caches and internal information coherent once the CPU has detected that code has been modified. While interesting, most of the time, self modifying coding will be slower than other programming techniques.

Self-modifying code is detected when a write occurs to a location that uses a 256 byte instruction line buffer (ILB). The ILB can be thought of as a prefetch buffer. Self-modifying code detection can also be detected when data and code lie on the same cache line and data is changed.

An indication that self modifying code is being used is when a CS: override is used in assembly language code.

2.11. Exclusive Instructions Use Both Pipes

These exclusive instructions execute in the X pipe only and use resources in the Y pipe. No instruction is paired. This will have a negative performance impact. The next instructions that follow will enter the pipes once the exclusive instruction leaves the AC1 stage.

The instructions are:

AAM, ARPL, BOUND, CALL, CLI, CLTS, DIV, ENTER, HLT, IDIV, IMUL, IN, INS, INT, INT0, INT1, INT3, INVD, INVLPG, IRET, Jump Indirect, Jump Interseg, LAXR, LEAVE, LGDT, LXS, LLDT, LMSW, LSL, LTR, MOVseg/sr, MOVS, MUL, OUT, OUTS, POPA, POPF, POP es/ss/ds, POP fs/gs, PUSHA, PUSHF, RET, SGDT, SIDT, SLDT, SMSW, STI, CLI, STD, CLD, STR, VERR, VERW, WAIT, WBINVD, LDS, LES, LGS, LGS, LSS, JCXZ, LOOP, CMOPXCHG, BSWAPCMPS, SCAS, XLAT.

Most of these instructions are typically used by operating system code.

2.12. Some Instructions Only Go Down the X Pipe

The following instructions always go down the X pipe:

Jcc, JMP, CALL, SETcc, and FPU instructions.

Other instructions are paired and sent down the Y pipe.

2.13. Unified Cache Architecture Issues

The Cyrix 6x86/6x86MX CPUs use a unified cache design. This means there is only one primary or L1 cache for both data and code. This typically provides a higher hit rate than the Harvard (split data and code) cache design used in the Pentium. But there exists the possibility for large data operations that data will fill the entire cache and causing code misses. This issue can be addressed by writing critical code to fit in the instruction line buffer.

On the 6x86MX there is another way to address this issue by locking-down critical code in the cache. See “Cache Line Locking to Aid Real Time Software” on page 21 for more information on Cache Line Locking on the 6x86MX.

The unified cache is effectively dual ported. It is possible to access two data items in the same clock if there is no bank conflict.

Two simultaneous misaligned word accesses will result in an extra clock delay in the AC2 stage.

A three clock penalty results from a write followed by a read to the same memory location.

2.14. Code Branch Alignment

Align branch targets to eight byte boundaries. The CPU will fetch 16 bytes on 8 byte boundaries.

2.15. Data Alignment

Don't span a data item across eight-byte boundaries.

2.16. Branch Miss Predictions Should Be Avoided

Extra clock-cycle delays occur when a CPU’s branch prediction predicts incorrectly. These delays typically result from branch repairs that require using additional resources. With register renaming, branch repair has a penalty of one clock.

Code should be designed to flow top to bottom with fewer loops. The trade off is code size.

Loops are initially predicted taken.

2.17. SMM

Cyrix SMM implementation is different than other vendors in terms of what is saved and restored automatically on entrance and exit of SMI interrupt. The Cyrix design minimized the overhead for entry and exit of the handler. This minimal CPU state save and restore resulted in fast entry and exit of the SMI handler. This implementation permits not only faster power management decisions but also allows for virtualization of peripherals (i.e., MediaGX).

The Cyrix SMM implementation is fully software configurable. This permits for using TSRs, or device drivers to act as SMM handlers.

For a detailed discussion refer to Application Note 107 *6x86MX SMM Design Guide.*

3. 6x86 Unique Features

3.1. CPUID and Returned Feature Bits

The 6x86 and 6x86L implement the CPUID Instruction, however for compatibility with previous CPUs the CPUID instruction is not enabled by BIOS. See Cyrix ID Application Note 112 on the Cyrix Developers web page for detailed information. The home web page is www.cyrix.com.

The CPUID instruction execution, with EAX=1, loads the feature bits into the EDX register. For the early 6x86 devices, the only feature bit that is enabled is bit 0 for the FPU.

The 6x86L supports an FPU, Debug Extensions (also known as I/O breakpoints) and the Compare Exchange 8 Byte instruction. The 6x86L can be identified by DIR0 values 2h. The 6x86L also adds support for the instruction MOV to and from CR4.

3.2. Cache Organization

The 6x86 contains a dual ported 16K unified cache with 512 lines and 32 bytes per line.

3.3. TLB Organization

The TLB on the 6x86 is a 128-entry direct-mapped TLB, backed up by an 8-entry fully associative victim TLB. Both TLBs are dual ported. This allows both integer pipes to access TLB entries simultaneously. The value of 136 should be used in algorithms that make decisions about flushing individual entries with the INVLPG instruction or all the entries with a MOV to CR3 instruction.

4. 6x86MX Unique Features

4.1. CPUID Bits

The 6x86MX is initialized with CPUID enabled. CPUID will return the following information for the 6x86MX CPU:

FEATURE	DATA
Vender ID String	"CyrixInstead"
CPUID Levels Supported	1
Family	6
Model	0
Stepping	TBD
Feature Flag Value	0x0080A135

FEATURE FLAGS SET

FEATURE FLAG SET	FEATURE
FPU Present	6x86MX contains an enhanced FPU.
I/O Breakpoints	I/O cycles can be trapped, controllable by CR4:DE.
Time Stamp Counter Supported	RDTSC instruction is supported, controllable by CR4:TSD.
RDMSR and WRMSR Instructions Present	Model Specific Registers are supported.
CMPXCGH8B Instruction Supported	Compare exchange eight byte instruction supported.
PTE Global Bit Support	PTE TLB will not be flushed when CR3 is written.
CMOV and FCMOV Instructions Supported	Conditional move instructions supported.
MMX™ Instructions Supported	Multi Media Instruction Extensions supported.

4.2. MMX™ Instructions and Optimizations

All instructions execute in one clock except: Packed Multiply, Packed Multiply-and-Add, and Dword Mov from an MMX™ register to a x86 core register. A new MMX™ instruction can be issued during each clock.

4.2.1 Fast FPU/MMX™ Switching

Certain processors, such as the Pentium™ with MMX, take as many as 50 or 60 machine cycles to switch between FPU and MMX™ instruction execution. The 6x86MX does not have this limitation. It executes the EMMS instruction in one clock.

4.2.2 Extended MMX™ Instructions

Cyrix has added instructions to its implementation of the Intel MMX™ Architecture in order to facilitate writing of multimedia applications. All of the added instructions follow the SIMD (single instruction, multiple data) format. Many of the instructions add flexibility to the MMX™ architecture by allowing both source operands of an instruction to be preserved, while the result goes to a separate register that is derived from the input registers.

4.2.3 Detecting Extended MMX™ Instructions

1) Check for Family 6 Model 0 - Extended MMX™ supported

2) Future CPU's can be checked by the Extended Feature Flag

CPUID Extended Flag[24] - Extended MMX supported

See the Cyrix Developer web site for the current extended CPUID description.

Cyrix Extended Instructions To MMX™ Instruction Set

MMX™ INSTRUCTIONS	OPCODE	OPERATION AND CLOCK COUNT	
PADDSIW *Packed Add Signed Word with Saturation Using Implied Destination* MMX Register plus MMX Register to Implied Register Memory plus MMX Register to Implied Register	 0F51 [11 mm1 mm2] 0F51 [mod mm r/m]	Sum signed packed word from MMX register/ memory ---> signed packed word in MMX register, saturate, and write result ---> implied register	 1 1
PAVEB *Packed Average Byte* MMX Register 2 with MMX Register 1 Memory with MMX Register	 0F50 [11 mm1 mm2] 0F50 [mod mm r/m]	Average packed byte from the MMX register/memory with packed byte in the MMX register. Result is placed in the MMX register.	 1 1
PDISTIB *Packed Distance and Accumulate with Implied Register* Memory, MMX Register to Implied Register	 0F54 [mod mm r/m]	Find absolute value of difference between packed byte in memory and packed byte in the MMX register. Using unsigned saturation, accumulate with value in implied destination register.	 2
PMACHRIW *Packed Multiply and Accumulate with Rounding* Memory to MMX Register	 0F5E[mod mm r/m]	Multiply the packed word in the MMX register by the packed word in memory. Sum the 32-bit results pairwise. Accumulate the result with the packed signed word in the implied destination register.	 2
PMAGW *Packed Magnitude* MMX Register 2 to MMX Register 1 Memory to MMX Register	 0F52 [11 mm1 mm2] 0F52 [mod mm r/m]	Set the destination equal ---> the packed word with the largest magnitude, between the packed word in the MMX register/memory and the MMX register.	 2 2
PMULHRIW *Packed Multiply High with Rounding, Implied Destination* MMX Register 2 to MMX Register1 Memory to MMX Register	 0F5D [11 mm1 mm2] 0F5D [mod mm r/m]	Packed multiply high with rounding and store bits 30 - 15 in implied register.	 2 2
PMULHRW *Packed Multiply High with Rounding* MMX Register 2 to MMX Register1 Memory to MMX Register	 0F59 [11 mm1 mm2] 0F59 [mod mm r/m]	Multiply the signed packed word in the MMX register/memory with the signed packed word in the MMX register. Round with 1/2 bit 15, and store bits 30 - 15 of result in the MMX register.	 2 2
PMVGEZB *Packed Conditional Move If Greater Than or Equal to Zero* Memory to MMX Register	 0F5C [mod mm r/m]	Conditionally move packed byte from memory ---> packed byte in the MMX register if packed byte in implied MMX register is greater than or equal ---> zero.	 1
PMVLZB *Packed Conditional Move If Less Than Zero* Memory to MMX Register	 0F5B [mod mm r/m]	Conditionally move packed byte from memory ---> packed byte in the MMX register if packed byte in implied MMX register is less than zero.	 1
PMVNZB *Packed Conditional Move If Not Zero* Memory to MMX Register	 0F5A [mod mm r/m]	Conditionally move packed byte from memory ---> packed byte in the MMX register if packed byte in implied MMX register is not zero.	 1
PMVZB *Packed Conditional Move If Zero* Memory to MMX Register	 0F58 [mod mm r/m]	Conditionally move packed byte from memory ---> packed byte in the MMX register if packed byte in implied the MMX register is zero.	 1
PSUBSIW *Packed Subtracted with Saturation Using Implied Destination* MMX Register 2 to MMX Register1 Memory to MMX Register	 0F55 [11 mm1 mm2] 0F55 [mod mm r/m]	Subtract signed packed word in the MMX register/ memory from signed packed word in the MMX register, saturate, and write result ---> implied register.	 1 1

4.2.4 Implied Registers

Implied registers provide a third register for Cyrix Extended MMX instructions. The implied register is used as a destination register for results, so that the source register's contents are not overwritten.

For example, the IDCT (Inverse Discrete Cosine Transform) algorithm, used in MPEG video decode, has several places where two vector inputs are used in two separate calculations. In one calculation, the two vectors may be added, and in the second one of the vectors are subtracted from the other. In order to accomplish this algorithm using the basic MMX instructions, one of the vectors must be copied in order to preserve its original value before the first computation. This is because the MMX instructions all destroy the contents of one of the source registers by using the same register as the destination.

Several of the Cyrix-added MMX instructions get around this problem by having an *implied destination* register, which is derived from the first source register. This way, the contents of both source vectors is preserved without having to make a copy of either one. A few of the instructions use an implied register as another source, so that the first register in the instruction is still the destination.

The implied register is calculated from the first source, according to the following table:

IMPLIED REGISTER PAIRS

FIRST SOURCE REGISTER	IMPLIED REGISTER
mm0	mm1
mm1	mm0
mm2	mm3
mm3	mm2
mm4	mm5
mm5	mm4
mm6	mm7
mm7	mm6

As implied from the table, the source and destination registers are in pairs, where the pairs are determined by changing the least significant bit of the binary representation of the register number.

4.2.5 Implied Instruction Examples

The PADDSIW instruction performs the same function as the basic MMX PADDSW instruction, except that it preserves the contents of both input vectors. If one of the vectors of interest is in register mm1 and the other is in register mm2, the instruction would look like this:

```
PADDSIW mm1, mm2   ;result in mm0
```

and the result would end up in register mm0. The instruction could also be written as:

```
PADDSIW mm2, mm1   ;result in mm3
```

and the result would end up in register mm3. In this particular instruction, the second input can also be a memory operand, but the implied register stays the same, so

```
PADDSIW mm1, [si]  ;result in mm0
```

puts its result in register mm0.

Caution is required for programming with these instructions in order for them to have the desired effect. For example,

```
PADDSIW mm1, mm0   ;result in mm0
```

will put its result in register mm0, thus losing the original input value. The instruction written this way is exactly equivalent to

```
PADDSW mm0, mm1
```

A few of the instructions that use an implied register still use the first register in the instruction as the destination. These instructions are the packed conditional move commands PMVZB, PMVZNB, PMVLZB, and PMVGEZB. Note that the mnemonics for these instructions do not have the “I” for “implied destination” in them, so there should be no ambiguity about where the result goes. In the case of the packed conditional move instructions, the packed values from the source are moved as packed values to the destination register, depending upon the packed values in the implied register. They are three-input instructions.

These instructions are beneficial for numerous algorithms, for example:

Implied destination instructions: PADDSIW, PSUBSIW, PMULHRIW

These are used for preserving the first src operand. Using them properly can overcome the short comings of two operand MMX instructions.

Fixed-point mode higher accuracy multiply with rounding: PMULHRW, PMACHRIW, and PMULHRIW provide higher accuracy multiplication. The result is a 1.15 instead of Intel 2.14 format. This is important in digital filters, video/audio/speech coding where improved accuracy is needed.

4.2.6 Examples of Extended MMX Instructions Applications

Average Instruction: PAVEB

PAVEB averages two MMX registers in byte partitions. This is useful in algorithms such as motion compensation.

Magnitude Instruction: PMAGW

This is useful in signal scaling by finding the largest magnitude among a number of samples.

Distance instruction: PDISIB

PDISIB calculates with an implied dst = SUM (a-b). The instruction works on byte a partition. This is appropriate for motion estimation which is part of many video compression algorithms.

Conditional Move Instructions: PMVGEZB, PMVLZB, PMVNZB, PMVZB

This group of instructions uses byte partition. These instructions can be used for image manipulation, such as the chroma keying algorithm.

4.2.7 Enabling Extended MMX Instructions

The Extended MMX instructions are disabled by default. These instructions are enabled by setting Bit 0 of CCR7 (index 0xeb) to 1. To do this will require ring 0 access which for some operating systems will mean a ring 0 device driver.

4.3. SMM Enhancements

SMM has been enhanced to support reentrancy or nesting of SMI's. This will facilitate servicing an SMM interrupt within an SMM context. Code and Data are cached on the 6x86MX to enhance performance.

See Application Note 107 *6x86MX SMM Design Guide* for details.

4.4. Cache Organization

The 6x86MX contains a dual ported 64K write-back unified cache. The cache is organized as a 4-way set associative cache with 2048 lines and 32 bytes per line.

4.5. TLB Organization

The 6x86MX TLB is organized into two levels. The first level is 16 entry direct. The second level is 384 entry 6 way. The value 400 should be used in algorithms that make decisions about flushing individual entries with the INVLPG instruction or all the entries with a MOV to CR3 instruction.

4.6. Performance Monitors

See the 6x86MX Data book for a detailed listing and discussion of performance monitors.

Note that the 6x86MX contains performance monitors that are not provided by from other CPU manufactures. When applicable the 6x86MX performance monitors overlap with the Intel® Pentium® P55C CPU. It is recommended that if performance monitors are used that the usage be conditioned by the CPU vendor and CPU Family and Model.

Due to the architectural differences between the Cyrix 6x86MX and the Intel P55C the counters may have different meanings with relation to TLB architecture, Cache architecture, and pipe naming.

The following Event Counter Types are different between the 6x86MX and Intel P55C:

EVENT COUNTER TYPE DIFFERENCES

NUMBER(HEX)	6X86MX DESCRIPTION
07	External Inquires
08	External Inquires that hit
0D	L2 TLB Code Misses
10	Reserved
11	Reserved
17	Instructions Executed in the Y pipe
20	Reserved
21	Reserved
2A	Reserved
2C	Reserved
2E	Reserved
30	Reserved
31	MMX Instruction Data Reads
32	Reserved
35	Reserved
36	Reserved
40	L2 TLB Misses (Code or Data)
41	L1 TLB Data Miss
42	L1 TLB Code Miss
43	L1 TLB Miss (Code or Data)
44	TLB Flushes
45	TLB Page Invalidates
46	TLB Page Invalidates that hit
48	Instructions Decoded

4.7. Determining Clock Multiplier

The clock multiplier of the 6x86MX can determined by reading the bottom 4 bits of DIR0. The register index for DIR0 is FEh.

DIR0[3:0]	CLOCK MULTIPLIER
0	1x
1	2
2	2.5x
3	3x
4	3.5x
5	4x
6	4.5x
7	5x

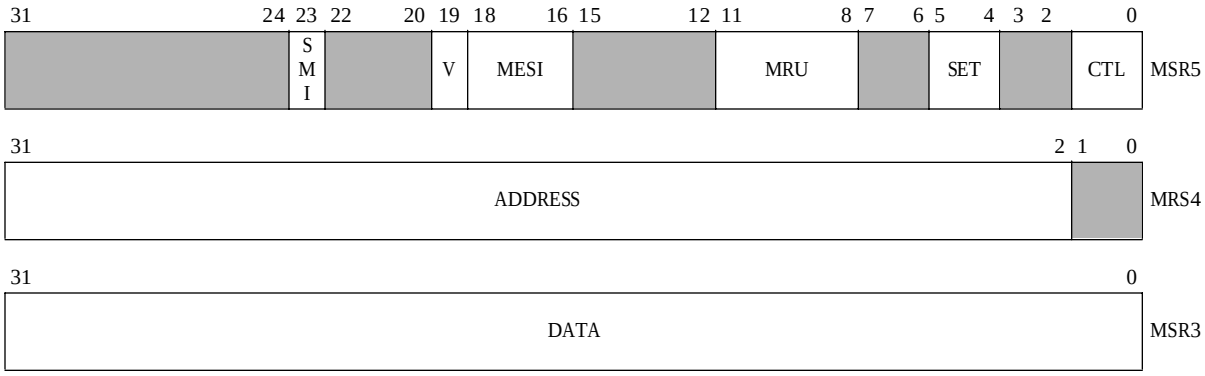
4.8. Cache Line Locking to Aid Real Time Software

The 6x86MX adds the unique ability to lock down lines in the primary or L1 cache of the CPU. This is valuable for Real Time applications. By locking down code or data in the L1 cache it guarantees that the information is kept in the L1 cache until software unlocks the area and the L1 cache LRU replaces the data. The only negative impact is the loss of the cache lines for normal cache operation.

Items locked down stay coherent only with that CPU, but are not guaranteed coherent with main memory.

4.9. Cache Line Locking Operations

The Cache Line locking feature is controlled by using MSR3, MSR4, and MSR5. When a line is unlocked, the line is marked invalid to avoid write backs to non-existent memory.



CACHE TEST REGISTER BIT DEFINITIONS

REGISTER NAME	FIELD NAME	RANGE	DESCRIPTION
MSR5	SMI	23	SMI Address Bit. Selects separate/cacheable SMI code/data space
	V, MESI	19 - 16	Valid, MESI Bits* If = 1000, Modified If = 1001, Shared If = 1010, Exclusive If = 0011, Invalid If = 1100, Locked Valid If = 0111, Locked Invalid Else = Undefined
	MRU	11 - 8	Used to determine the Least Recently Used (LRU) line.
	SET	5 - 4	Cache Set. Selects one of four cache sets to perform operation on.
	CTL	1 - 0	Control field If = 00: flush cache without invalidate If = 01: write cache If = 10: read cache If = 11: no cache or test register modification
MSR4	ADDRESS	31 - 2	Physical Address
MSR3	DATA	31 - 0	Data written or read during a cache test.

*Note: All 32 bytes should contain valid data before a line is marked as valid.

MSR5 CACHE CONTROL OPERATIONS:

ACTION	ECX	EDX	EAX	OPERATION
Read/ Write	03h	-----	Cache Data	Data to/from MSR3
Write	04h	Address Upper 32 Bits	Address Lower 32 Bits	Data at EDX:EAX->MSR4
Read	04h	Address Upper 32 Bits	Address Lower 32 Bit	MSR4 -> EDX:EAX
Write	05h	-----	Data	Function_MSR5 (EAX)
Read	05h	-----	Data	Read MSR5 -> EAX

Lock information is kept in the MESI bits.

4.10. Cache Modification Actions Effects on Locked Lines

CACHE MODIFICATION ACTION EFFECTS ON LOCKED LINES

ACTION	EFFECT ON LOCK BITS
Power on Reset	Cleared
Reset	Cleared
Warm Reset	Unaffected
Flush	Unaffected
WBINVD	Unaffected
INVD	Unaffected

4.11. Cache Line Locking Guide Lines

Cache line locking is more effective when the following suggestions are followed:

1) Don't lock all ways or sets of the cache. Inspect the cache before locking. Avoid locking set 3 so it will be available for normal cache operation.

2) Do not allocate an address twice in a cache block. The results will be catastrophic.

Check for an address already being locked.

See “Cache Line Locking Operations” on page 22 for additional information.

4.12. Cache Line Locking Example Code

As time permits, Cache Line Locking examples will be placed on the Cyrix Software Developer web page.

Appendix A.
Summary of Differences Between 6x86 and 6x86MX CPUs

FEATURE	6x86	6x86MX
Cache	16K Size 4-way 512 lines 32 bytes per line	64K Size 4-way 2048 lines 32 bytes per line
TLB	128/8	16/384
BTB	256	512
Lockable Cache	no	yes
Time Stamp Counter	no	yes
Performance Counters	no	yes
Global PTE	no	yes
CPUID enabled	no	yes
MMX	no	yes
Extended MMX	no	yes
Nestable SMI support	no	yes
CMOV	no	yes
SMI code/data cacheable	no	yes
Prefetch Queue Depth	64 bytes	64 bytes

Appendix B. Web Page for Software Vendor Support

For more information, help, or to contact Software Vendor Support:

```
http://www.cyrix.com/developers/software/isv.htm
```

Appendix C.
6x86™ and 6x86MX™ Technical Documents.

- 6x86 Data Book
- 6x86 BIOS Writer's Guide
- 6x86 SMM Programmer's guide
- 6x86MX Data Book
- 6x86MX BIOS Writer's Guide
- 6x86MX SMM Programmer's Guide
- SMM Programmer's Guide.
- Application Note 101 Board Design and Bus Differences
- Application Note 102 Signal and Bus Description
- Application Note 103 BIOS Writer's Guide
- Application Note 104 Fan Voltage Regulator and Chipset Guide
- Application Note 105 Thermal Considerations
- Application Note 106 CPU Optimization
- Application Note 107 SMM Design Guide
- Application Note 108 Cyrix MMX Extension
- Application Note 112 Cyrix CPU Detection Guide

Printed in the United States of America

Order Number: 94xxx-xx

Cyrix Corporation

2703 North Central Expressway

Richardson, Texas 75080

United States of America

May 22, 1998 10:19 am
C:\!!!devices\appnotes\106ap.fm5

Rev 1.7 Added Prefetch Queue to Differences Table Appendix A, Page 25
Rev 1.6 Changed TRx to MSRx