The Cyrix M1 Architecture

Overview

Cyrix M1 architectural feature comparison

Feature	Cyrix M1	Intel Pentium	Alpha 21164	PowerPC 604
x86 instruction set	✔	✔		
Superscalar	✔	✔	✔	✔
Multiple integer units	✔	✔	✔	✔
Superpipelined	✔		✔	
Register renaming	✔			✔
General purpose registers	32	8	32	32
Data forwarding	✔			✔
Branch prediction	✔	✔	✔	✔
Speculative execution	✔			✔
Out-of-order completion	✔		✔	✔
Cache size	16 KByte	16 KByte	16 Kbyte + 96 KByte L2	32 KByte
Cache architecture	Unified + ILC	Harvard	Harvard	Harvard
FPU	✔	✔	✔	✔

The Cyrix M1 architecture is a superscalar, superpipelined x86 processor architecture operating at very high clock rates. The architecture's sophisticated dependency- and conflict-reduction schemes are implemented in hardware, allowing it to deliver performance increases of roughly 2.5 times that of the 486 architecture operating at an identical clock rate, and a gain of 30%–50% over the Pentium at an identical clock rate when running today's applications. This architectural advantage, coupled with the core clock rates of 100 MHz and better, yield up to five times the performance of a 486-50, and up to two times that of a current Pentium processor. The M1 architecture provides more than 12 times the performance of a typical RISC-based architecture operating in "compatibility" mode, the mode required for the RISC architecture to run existing x86 software.

The Cyrix M1 architecture includes five basic elements: Integer Unit, Floating Point Unit, Cache Unit, Memory Management Unit, and Bus Control Unit.

Integer Unit and Floating Point Unit

The M1 is a superscalar, superpipelined architecture that utilizes two seven-stage integer pipelines, the x-pipe and the y-pipe. Each pipeline contains a prefetch stage, two decode stages (ID1, ID2), two address-calculation stages (AC1, AC2), an execute stage (EX), and a write-back stage (WB).

The M1 architecture also contains a single, 64-bit, enhanced x87-compatible, floating point pipeline. The FPU is enhanced by a four-instruction queue and four independent, 64-bit write buffers.

Cache Unit and Memory Management Unit

The M1 architecture contains a 16-KByte, on-chip, 4-way set associative, unified instruction/data cache as the primary data cache and secondary instruction cache, and a 256-byte, fully set associative, instruction line cache that is the primary instruction cache. The unified cache is dual-ported to allow for two simultaneous fetches, reads, writes, or combinations of any two.

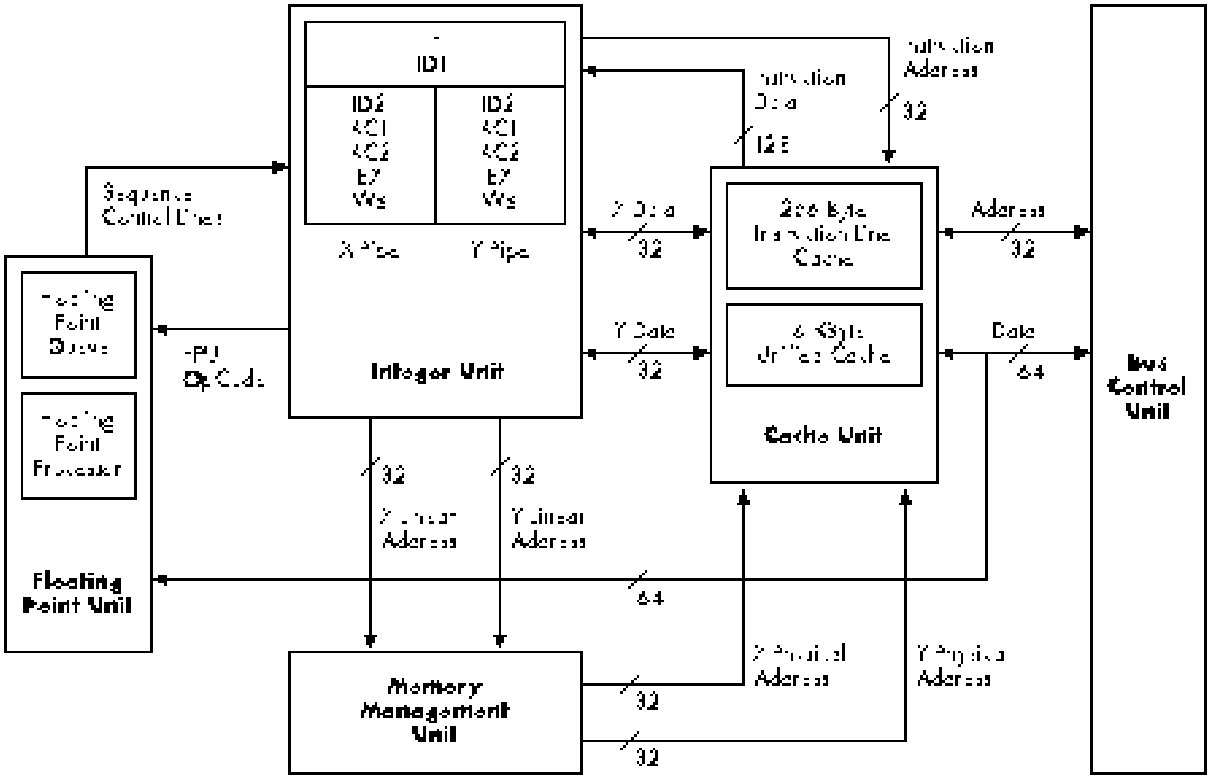
The M1 architecture memory management unit includes two paging mechanisms, the traditional x86 architecture mechanism, and a M1-unique variable-sized paging mechanism. The variable-sized paging mechanism allows software to map address regions between 4 KBytes and 4 GBytes in size. The use of large, contiguous memories significantly increases performance in applications that make heavy use of RAM, such as video-intensive graphics applications and desktop publishing applications.

Bus Interface Unit

The M1 architecture bus interface unit provides the signal lines and device timing required to implement the architecture in a system. The bus interface unit is logically isolated, to provide for wide flexibility in system implementation.

The Cyrix M1 Architecture

The Cyrix M1 architecture includes five basic elements: integer unit, floating point unit, cache unit, memory management unit and bus control unit.



Integer Unit and Floating Point Unit (FPU)

Overview

The M1 is a superscalar, superpipelined processor architecture. *Superscalar* means that the M1 architecture contains two separate instruction pipelines capable of executing instructions in parallel. *Superpipelined* means that the instruction pipelines themselves are divided into seven processing stages, allowing higher clock rates with a given process technology.

Because of the degree of sophistication in the M1 architecture superpipelining scheme, an M1 register access and an M1 cache access require the same amount of time — one clock — to complete. Other processor architectures, such as the Pentium, typically require two or more clocks to complete a cache access.

In x86 programs, branch instructions occur an average of every four to six instructions. Branch instructions change the sequential ordering of the instruction stream, that may result in pipeline stalls as the processor calculates, retrieves, and decodes the new instruction stream. The M1 architecture provides two mechanisms to reduce the performance impact and latency of branch instructions: *branch prediction* and *speculative execution*.

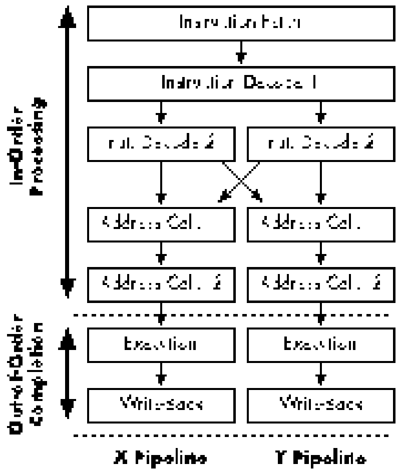
Cyrix M1 architectural feature comparison

Feature	Cyrix M1	Intel Pentium	Alpha 21164	PowerPC 604
Superscalar	✔	✔	✔	✔
Multiple integer units	✔	✔	✔	✔
Superpipelined	✔		✔	
Branch prediction	✔	✔	✔	✔
Speculative execution	✔			✔
Out-of-order completion	✔		✔	✔
FPU	✔	✔	✔	✔

Integer Unit

The integer unit includes two seven-stage integer pipelines, the x-pipe and the y-pipe. Each pipeline contains an instruction fetch stage (IF), two instruction decode stages (ID1, ID2), two address calculation stages (AC1, AC2), an execute stage (EX), and a write-back stage (WB).

Integer unit logic diagram



Instruction Fetch (IF)

The instruction fetch (IF) stage prefetches up to 16 bytes of instructions per clock. Branch predictions are performed during this cycle to fetch instructions from the predicted path.

Branch Prediction. The M1 architecture uses branch prediction to select an instruction path for a branch instruction. The selection — or prediction — is based on the target address and history information in the branch target buffer (BTB). In the IF pipeline stage, the instruction stream is checked for branch instructions. If a branch instruction is found in the stream, the processor accesses the BTB. The type of BTB data checked depends on whether the branch instruction was conditional or unconditional.

Unconditional Branches. Unconditional branches are changes in program flow that always occur — they are not dependent upon the fulfillment of certain conditions or the state of certain program elements. Unconditional branches result in a BTB check for the branch instruction's target address.

If the check results in a BTB hit — a target address associated with the branch instruction is found in the BTB — the processor predicts that the branch will be taken at the address in the BTB, and begins fetching instructions from that address. Execution begins based on the new instruction stream.

Conditional Branches. Conditional branches are changes in program flow that depend upon the fulfillment of certain conditions or the state of certain program elements. Conditional branches result in a BTB check for the branch instruction's target address. If the check results in a BTB hit — a target address associated with the branch instruction has been found — the processor checks the history bits associated with the BTB address to determine if the branch should be "taken" or "not taken."

When the history bits predict that the branch will be taken, the processor begins fetching instructions from the BTB address. Speculative execution begins, based on the new instruction stream.

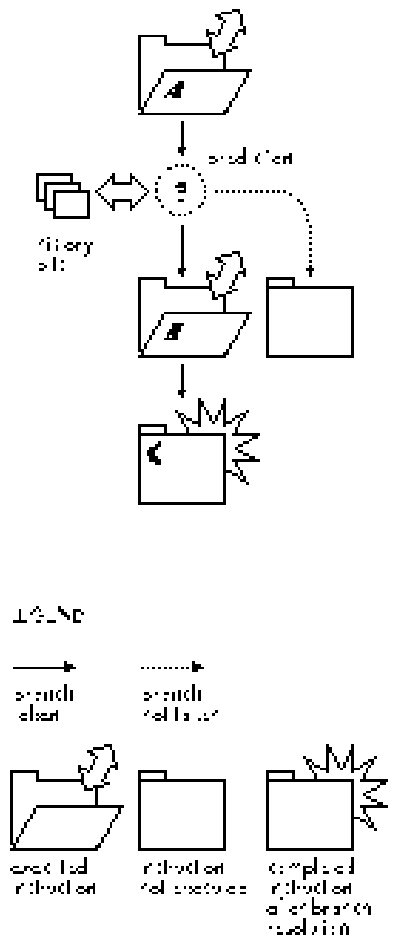
When the history bits predict that the branch will not be taken, speculative execution continues along the sequential instruction stream, the "not taken" branch.

If the BTB check results in a miss — the address is not found — the processor predicts that the branch will not be taken and thus, the stream does not branch.

The decision to fetch instructions from either the taken branch address or the not-taken branch address is based on a 4-stage prediction algorithm. Branch prediction accuracy in the M1 architecture is approximately 90%.

Return Instructions. Return (RET) instructions are branch instructions with dynamic target addresses. As a result, the M1 architecture caches RET target addresses in a return stack, rather than in the BTB. The RET address is pushed on the stack during a CALL instruction and popped during the corresponding RET instruction.

Branch prediction flow
The M1 architecture selects an instruction path for a branch instruction.



Instruction Decode (ID1, ID2)

The instruction decode stage determines the instruction length (ID1), decodes the instruction, and determines the optimal pipeline in which to execute the instruction (ID2).

Intelligent Instruction Dispatch. The M1 architecture uses intelligent instruction dispatch to select the destination pipeline — either the x-pipe or the y-pipe — for each instruction decoded. For the most commonly used instructions in the x86 instruction set there are no constraints on pipeline selection, so the selection is made to optimize the parallel use of the pipelines. However, there are certain conditions where the architecture imposes selection restraints on instruction processing:

X-Pipe-Only Instructions. The following instructions may only be processed in the x-pipeline:

- Branch
- Floating point
- Exclusive

There are some additional parameters for the dispatch of these instruction types, as follows:

Branch and Floating Point Instructions. Branch and floating point instructions may be paired with an instruction in the y-pipeline.

Exclusive Instructions. Exclusive instructions may not be paired with an instruction in the y-pipeline. However, hardware in both pipelines is used to accelerate completion of these instructions.

The following M1 instruction types are exclusive.

- Protected mode segment loads
- Control, debug, and test register accesses
- String instructions
- Multiply and divide instructions
- I/O port accesses
- Push/Pop All (PUSHA and POPA)
- Task switches

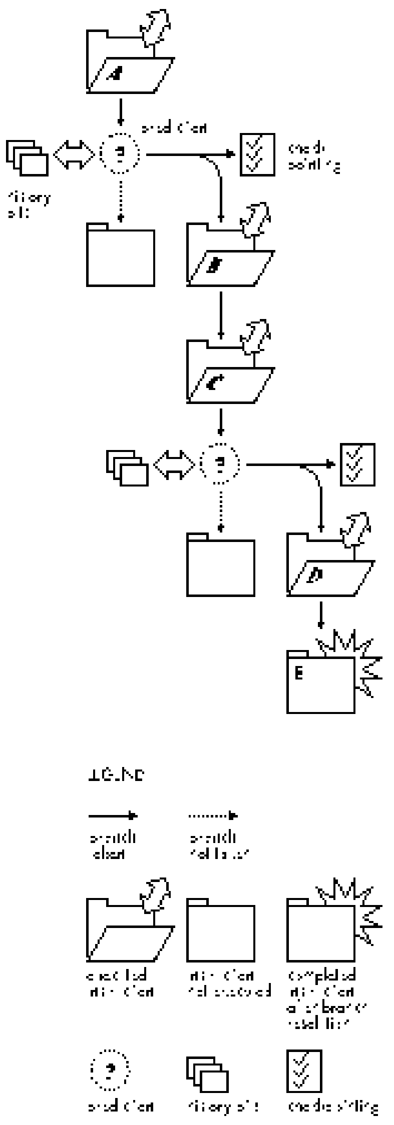
Address Calculation (AC1, AC2)

The address calculation stage calculates up to two effective addresses per clock cycle, performs register renaming, and makes scoreboard checks (AC1). The second address calculation stage (AC2) accesses the translation lookaside buffer (TLB), cache, and register file, as well as performing segmentation and paging checks. The superpipelined architecture of the address calculation stage allows ALU instructions that use an operand from the cache to complete in a single clock.

Execution (EX)

The execution stage performs ALU operations. Speculative execution and out-of-order completion take place during this stage and the WB stage.

Speculative execution
By following the instruction stream of a predicted branch, speculative execution eliminates pipeline stalls that would result from waiting on the resolution of the predicted branch instruction.



Speculative Execution. The M1 architecture uses speculative execution to continuously execute instructions in the x- and y-pipelines following a branch instruction or floating point operation. By following the instruction stream of a predicted branch, speculative execution eliminates pipeline stalls that would result from waiting on the resolution of the predicted branch instruction.

Checkpointing and Speculation Level. Once a branch instruction has been predicted, the processor checkpoints the machine state — the state of registers, flags, and the processor environment — and increments the speculation level counter. With checkpointing complete, the processor fetches the instruction stream from the predicted target address and begins executing the stream as if the branch had been correctly predicted.

The M1 architecture is capable of up to four levels of speculation — combinations of branch predictions and floating point operations — active at any one time.

Once a predicted branch instruction is resolved, the processor decrements the speculation level counter and then acts on the instruction stream, depending on the outcome of the prediction.

Correctly Predicted Branches. Correctly predicted branches result in the processor clearing the checkpoints for that branch and continuing execution of the current stream.

Mispredicted Branches. Mispredicted branches result in the processor clearing the pipeline and fetching the correct instruction stream from the actual target address. The machine state is restored to the checkpoint values in a single clock, and the processor resumes execution on the correct instruction stream in the ID1 stage.

Cache/Memory Writes. For compatibility, writes that result from speculatively-executed instructions are prohibited from updating the cache or external memory until the originating branch instruction is resolved.

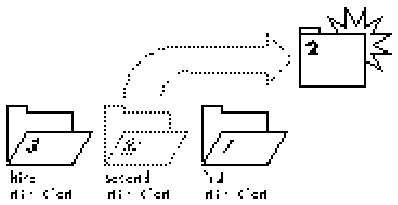
Constraints. Speculative execution continues in the M1 architecture until one of four conditions occurs:

A branch instruction or floating point operation is decoded when

1. the speculation level counter is already at four or
2. an exception or fault occurs or
3. the write buffers are full, or
4. an attempt is made to modify a non-checkpointed resource, such as the system registers.

Out-of-order completion

Current and subsequent instructions in the EX stage of the non-stalled pipeline may be completed without waiting for the instruction in the stalled pipeline to complete.



Out-of-Order Completion. The M1 architecture uses out-of-order completion to enable instructions in one pipeline to complete without waiting for an instruction in the other pipeline to complete, regardless of the order in which the two instructions were issued.

Out-of-order completion occurs in the execution (EX) and write-back (WB) stages of the pipeline. Out-of-order completion occurs whenever the following conditions are met:

1. an instruction in one pipeline is ready to complete before an instruction in the other pipeline,
2. the instruction in the other pipeline is the “first” instruction,
3. the “first” instruction requires multiple clock cycles to complete.

These conditions usually result from the “first” instruction’s being stalled while waiting for a memory access to complete.

With out-of-order completion, the “ready-to-complete” instruction, as well as any subsequent instructions in the EX stage of the non-stalled pipeline, may be completed without waiting for the “first” instruction to complete.

x86 Program Compatibility. The rules governing inter-instruction dependencies and program order ensure that software compatibility is maintained, while still achieving significant performance increases.

When there are inter-instruction dependencies that might cause the stall in one pipe to stall the other, the M1 architecture includes a number of data dependency removal schemes to enable the non-stalled pipe to continue executing.

The M1 architecture always provides instructions to the EX stage in program order, and allows instructions to complete out-of-order only from that point on. In conjunction with the restrictions governing exclusive instructions, this limitation ensures that exceptions occur in program order and that writes resulting from instructions completed out-of-order are issued to the cache or external bus in program order.

Write-Back (WB)

The write-back stage writes to the register file and write buffers and updates the machine state.

Floating Point Unit (FPU)

The M1 architecture integral FPU is a 64-bit, enhanced x87-compatible device. The FPU provides a four-instruction queue and a set of four independent, 64-bit write buffers, allowing up to four FPU instructions to be outstanding while the integer unit continues to execute instructions in parallel.

Cache and Memory Management Unit (MMU)

Cyrix M1 architectural feature comparison

Feature	Cyrix M1	Intel Pentium	Alpha 21164	PowerPC 604
Cache size	16 KBytes	16 KBytes	16 Kbytes + 96 Kbytes L2	32 KBytes
Cache architecture	Unified+ ILC	Harvard	Harvard	Harvard

Overview

The M1 architecture cache is a 16-Kbyte, on-chip, dual-ported, 4-way set associative, unified instruction/data cache, providing the primary data cache and secondary instruction cache. The instruction line cache is a 256-byte, fully associative, cache providing the primary instruction cache.

The M1 architecture memory management unit (MMU) provides an enhanced x86-compatible page-mapping mechanism, and a M1-unique page-mapping mechanism to improve performance.

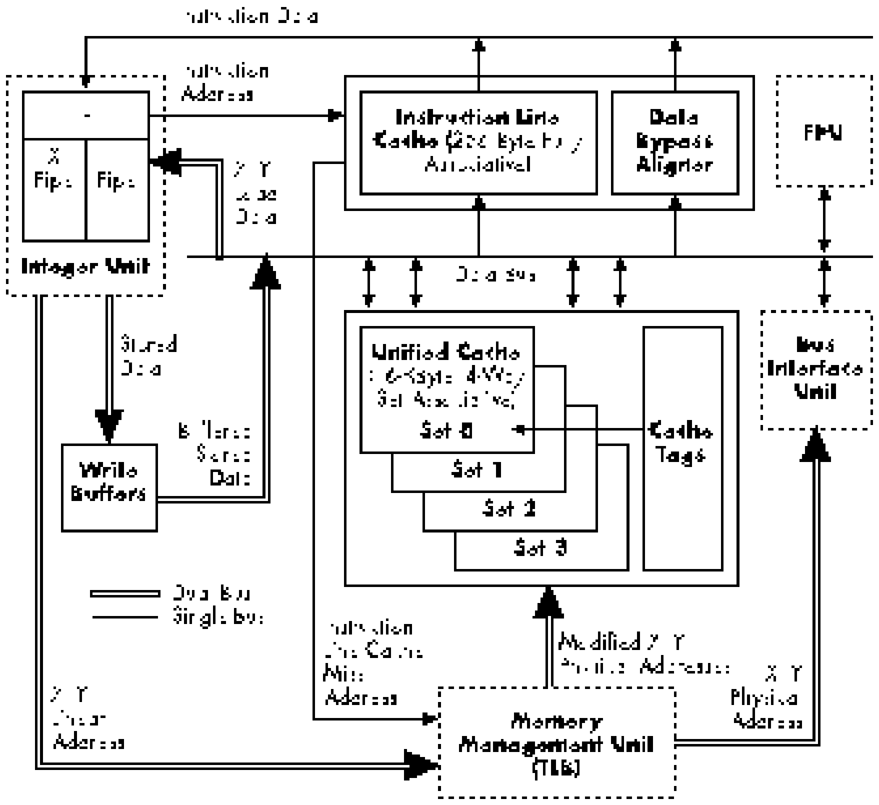
Cache Unit

The M1 architecture cache is an innovative design combining a unified instruction/data cache capable of storing data and instructions in any ratio, with a separate instruction line cache.

By combining the unified cache and the instruction line cache into a single architectural cache scheme, the M1 architecture provides a higher hit rate over a wider range of applications than other cache architectures of the same size and achieves high bandwidth.

Unified Cache. The 4-way set associative unified cache can store up to 16 KBytes of code and data, with each of the 512 cache lines holding 32 bytes.

Cache unit logic diagram



Dual-Porting. The unified cache is dual-ported to allow for two simultaneous code fetches, reads (x- or y-pipe, FPU), writes (x- or y-pipe, FPU), or combinations of any two of these operations.

Instruction Line Cache. The instruction line cache is a 256-byte fully associative instruction cache.

The instruction line cache is filled from the unified cache. Code fetches from the integer unit which hit in the line cache do not access the unified cache. For instruction line cache misses, the 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 algorithm for cache line replacements. To ensure safety for 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 line cache, the appropriate line is invalidated.

Memory Management Unit (MMU)

The MMU translates integer unit linear addresses into physical addresses for use by the cache unit and the bus interface unit. The MMU includes a translation lookaside buffer (TLB), a victim translation lookaside buffer (Victim TLB), and a directory table entry (DTE) cache.

Translation Lookaside Buffer (TLB). The M1 architecture TLB is a 128-line, direct-mapped cache for the most recently used page table entries (PTE).

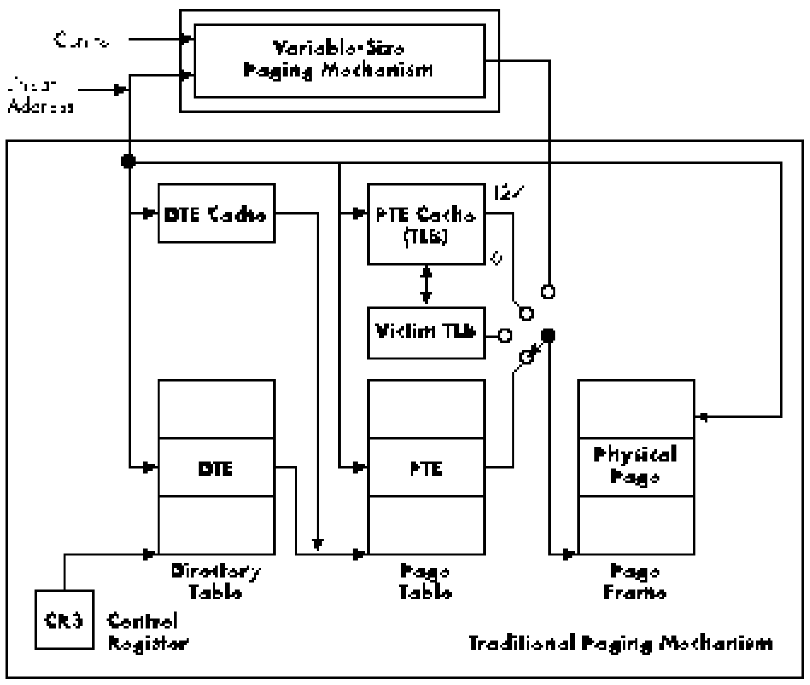
Victim Translation Lookaside Buffer (Victim TLB). The Victim TLB is an eight-entry, fully associative cache storing the PTEs displaced from the main TLB due to a miss. If a PTE access occurs while the PTE is stored in the Victim TLB, the Victim PTE is swapped with a PTE in the main TLB. This swapping has the effect of selectively increasing TLB associativity. The Victim TLB is updated on an “oldest-entry” basis.

Directory Table Entry (DTE) Cache. The DTE cache is a four-entry, fully associative cache storing the most recent DTE accesses. With the DTE cache, only a single memory access to the page table is required when there is a page table entry (PTE) miss followed by a DTE hit,.

M1-Unique Variable-Size Page Mechanism. The M1-unique variable-sized paging mechanism allows software to map address regions between 4 KBytes and 4 GBytes in size. The use of large, contiguous memories significantly increases performance in applications which make heavy RAM demands, such as video-intensive graphics applications and desktop publishing applications.

The large contiguous memories allowed by the variable-sized paging mechanism also help avoid the TLB thrashing associated with certain operating systems and applications.

Memory management unit (MMU)



Register Renaming and Data Forwarding

Cyrix M1 architectural feature comparison

Feature	Cyrix M1	Intel Pentium	Alpha 21164	PowerPC 604
Register renaming	✔			✔
General purpose registers	32	8	32	32
Data forwarding	✔			✔

Overview

In traditional x86 architectures, instructions execute in a single pipeline in the order in which they were issued — the instruction process is *serial*. In superscalar architectures, instructions execute in either of two pipelines, and may complete in a different order from that in which they were issued — the instruction process is *parallel*.

Because superscalar architectures use two parallel pipelines, instructions are issued in pairs, one to each pipeline. This provides the potential for very dramatic performance increases over traditional architectures, but existing software is often coded in such a way that it cannot take full advantage of both pipelines.

The limiting factor in the case of superscalar architectures is that one of the paired instructions may require the data from the other instruction. As both instructions are executing at the same time, the instruction that requires the data to complete stalls while it waits for the other instruction to write the data it needs to the appropriate register. This is what is known as a *read-after-write (RAW) dependency*, and the stall it causes is called a *serialization stall*.

There are two ways to minimize the effects of RAW dependencies in superscalar architectures: *recompilation*, and *architectural design solutions*.

Recompilation — requiring all existing software to be "optimized" for the new architecture — is the solution adopted by those architectures that are merely intragenerational improvements of traditional architectures. Recompilation works by reducing the number of dependencies generated by code serialization. While somewhat effective, recompilation is enormously expensive, for everyone other than the processor developer. For the application developer and, more importantly, for the end-user — who must replace or upgrade a significant software investment in order to achieve the promised performance — recompilation is expensive, frustrating, and time-consuming. This approach doesn't deliver the promised performance from the huge installed base of x86 compatible applications, though it reduces development time and expenses for the processor developer.

The more comprehensive solution is to remove the dependencies resulting from code serialization by building a better architecture. This is the approach taken by the M1 architecture. While this is more complicated for Cyrix, it has the advantage of delivering the promised performance today — by taking advantage of the installed base of x86 applications. This dramatically reduces the real cost of acquiring high-performance technology to the end-user, as it eliminates the need to replace existing applications with "new" or "optimized" versions of the same program.

The M1 architecture implements three architectural enhancements to eliminate or minimize the impact of data dependencies:

- Dynamic register renaming,
- Data forwarding,
- Data bypassing.

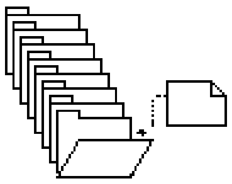
Register Renaming

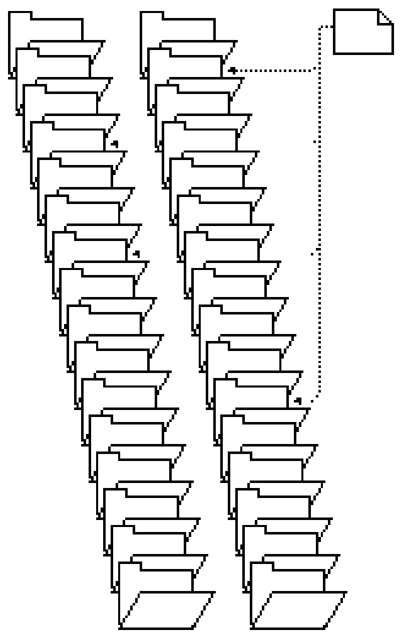
In traditional x86 architectures, the processor has access to 8 general-purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP. All of the processor's register operations are confined to these eight physical registers. In superscalar architectures based on the x86 model, this situation results in a performance degradation, as the advantages of having two pipelines are offset by the inadequacy of the register set.

With a huge installed base of applications designed to run on the x86 model architecture — including the use of those eight general-purpose (GP) registers — the solution to removing this performance obstacle involves more than simply adding more registers. An architecture which intends to protect the existing installed base while still providing a substantial performance increase must make the same registers available to the processor, even when those registers are already in use.

To accomplish this, the Cyrix M1 architecture implements dynamic register renaming on a set of 32 physical registers. Any one of the 32 physical registers can be used as any one of the eight x86 GP registers. In this way, the processor's access to the GP register it needs is not limited by the x86 model of registers; (any GP register can be dynamically assigned to any of 32 physical registers the processor requires). As register renaming is an architectural — not a software — enhancement, it is transparent to the applications running on the processor, and therefore allows those applications to enjoy the full benefit of the superscalar architecture, without being specially recompiled.

Without register renaming
All the processor's register operation are limited to these eight physical registers.





Using register renaming
Any GP register can be dynamically assigned to any of 32 physical registers the processor requires.

Write-After-Read (WAR) Dependency Removal. A WAR dependency exists when an instruction in one pipeline must wait for the instruction in the other pipeline to read a register value before it can write to the same register. A WAR dependency results in a stall in the "waiting" instruction pipeline. With dynamic register renaming, the M1 eliminates the WAR dependency.

The instructions

(1) MOV	BX,AX
(2) ADD	AX,CX

are executed in parallel,

X PIPE	Y PIPE
(1) BX << AX	(2) AX << AX+CX

resulting in a WAR dependency on the AX register in the Y pipeline.

The WAR dependency is resolved in the M1 by mapping the three GP registers involved into four physical registers.

Where the initial mapping is
AX=reg0
BX=reg1
CX=reg2
the M1 executes in parallel,

X PIPE	Y PIPE
(1) reg3 << reg0	(2) reg4 << reg0+reg2

resulting in the removal of the WAR dependency on the AX register in the Y pipeline.
The final register mapping is
AX=reg4
BX=reg3
CX=reg2.

Write-After-Write (WAW) Dependency Removal. A WAW dependency exists in any situation where an instruction in one pipeline must wait for the instruction in the other pipeline to complete a register before it can write to the same register. In the M1, which utilizes data forwarding to remove memory dependencies, a WAW dependency results in a stall in the "waiting" instruction pipeline. With dynamic register renaming, the M1 eliminates the WAW dependency.

The instructions

(1) MOV	AX,[mem]
(2) ADD	AX,BX

are executed in parallel using data forwarding,

X PIPE	Y PIPE
(1) AX << [mem]	(2) AX <<[mem]+BX

resulting in a WAW dependency on the AX register in the Y pipeline.

The WAW dependency is resolved in the M1 by mapping the two GP registers involved into four physical registers.

Where the initial mapping is

AX=reg0
BX=reg1,

the M1 executes in parallel,

X PIPE	Y PIPE
(1) reg2 << [mem]	(2) reg3 << [mem]+reg1

resulting in the removal of the WAW dependency on the AX register in the Y pipeline.

The final register mapping is

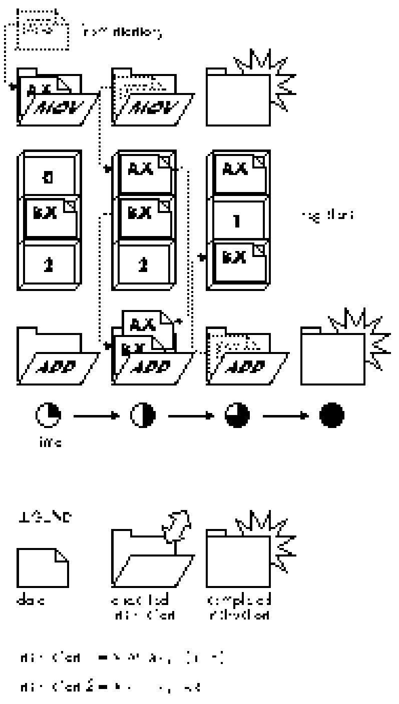
AX=reg3
BX=reg1.

Data Forwarding

Data forwarding, when used in conjunction with register renaming, removes data dependencies between instructions that normally require serialization. By forwarding the operand or result from the "leading" instruction to the "following" instruction, data forwarding allows the instructions to execute in parallel, eliminating serialization stalls and removing most RAW dependencies.

Operand Forwarding. When a processor executes a MOV instruction from memory to a register in one pipeline, while the instruction in the other pipeline requires the data being moved, a read-after-write (RAW) dependency is created on the data being moved — the operand — in the second pipeline. The M1 eliminates this dependency by forwarding the operand to the second pipeline without waiting for the MOV instruction to complete.

Without data forwarding
When a processor executes a MOV instruction in one pipeline, while the instruction in the other pipeline requires the data being moved, a read-after-write (RAW) dependency is created on the data being moved — the operand — in the second pipeline:



The instructions

(1) MOV	AX,[mem]
(2) ADD	BX,AX

are executed in parallel,

X PIPE	Y PIPE
(1) AX << [mem]	(2) BX << AX+BX

resulting in a RAW dependency on the AX register in the Y pipeline.

The RAW dependency is resolved in the M1 using data forwarding from memory to the Y pipeline.

Where the initial mapping is

AX=reg0
BX=reg1

the two instructions are executed in parallel using data forwarding,

X PIPE	Y PIPE
(1) reg2 << [mem]	(2) reg3 << [mem]+reg1

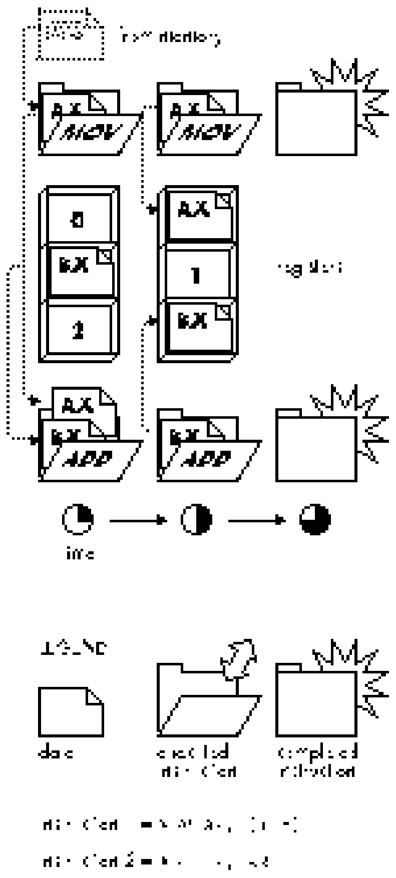
resulting in the removal of the RAW dependency on the AX register in the Y pipeline.

The final register mapping is

AX=reg2
BX=reg3.

Using data forwarding

The M1 eliminates this dependency by forwarding the operand to the second pipeline without waiting for the MOV instruction to complete:



Result Forwarding. When a processor executes a MOV instruction from a register to memory on the results of an instruction in the other pipeline, a read-after-write (RAW) dependency is created on the data being moved — the result — in the first pipeline. The M1 eliminates this dependency by forwarding the result to the first pipeline without waiting for the instruction in the first pipeline to complete the store of its result.

The instructions

(1) ADD	AX,BX
(2) MOV	[mem],AX

are executed in parallel,

X PIPE	Y PIPE
(1) AX << AX+BX	(2) [mem] << AX

resulting in a RAW dependency in the Y pipeline on the AX register.

The RAW dependency is resolved in the M1 using data forwarding from the X pipeline to memory.

Where the initial mapping is

AX=reg0
BX=reg1

the two instructions are then executed in parallel using data forwarding,

X PIPE	Y PIPE
(1) reg2 << reg0+reg1	(2) [mem] << reg0+reg1

resulting in the removal of the RAW dependency on the AX register in the Y pipeline.

The final register mapping is

AX=reg2
BX=reg1.

Data Bypassing

Data bypassing allows a memory location or register operand to be passed directly to the next instruction, without waiting for the location or register to be updated, further reducing the penalty of RAW dependencies that cannot be resolved using data forwarding. Although many processors perform data bypassing for register operands, the M1 also allows bypassing for memory operands.

When a processor executes an instruction in one pipeline, while the instruction in the other pipeline requires the result of that instruction as its operand, a read-after-write (RAW) dependency is created on the operand in the second pipeline. When the dependency occurs for a memory location, the M1 minimizes the delays resulting from this dependency by passing the result of the first pipeline instruction to the second, eliminating the read cycle from memory of the result of the first instruction.

The instructions

(1) ADD	[mem],AX
(2) SUB	BX,[mem]

are executed,

X PIPE	Y PIPE
(1) [mem] << [mem]+AX	(2) BX << BX-[mem]

resulting in a RAW dependency on the memory operand in the Y pipeline.

The RAW dependency is minimized in the M1 by passing the result of the X pipeline instruction directly to the Y pipeline instruction.

Where the initial mapping is

AX=reg0
BX=reg1

the instructions are executed in parallel using memory bypassing,

X PIPE
(1) [mem] << [mem]+reg0
Y PIPE
(2) reg2 << reg1-{[mem]+reg0}

resulting in the minimalization of the RAW dependency on memory in the Y pipeline.

The final register mapping is

AX=reg0
BX=reg2.

Cyrix Corporation
P.O. Box 850118
Richardson, TX 75085-0118
(214) 968-8388
Fax:214-699-9857