The IBM 6x86 and 6x86L Microprocessor Architectural Overview

Application Note

Superscalar, Superpipelined x86 Compatible CPU

- **SUPERSCALAR, SUPERPIPELINED ARCHITECTURE**
  - **Dual 7-stage integer pipelines**
  - **High performance on-chip FPU**
  - **100 MHz and greater operating frequency**
- **X86 INSTRUCTION SET COMPATIBLE**
  - **Runs Windows, DOS, UNIX, Novell and others**
- **OPTIMUM PERFORMANCE WITHOUT RECOMPILATION**
  - **Intelligent instruction dispatch**
  - **Out-of-order instruction completion**
  - **Register renaming**
  - **Data forwarding**
  - **Branch prediction**
  - **Speculative execution**

Introduction

The superscalar, superpipelined IBM 6x86 and 6x86L processor architecture provides next generation performance to IBM PC-compatible software. Because the IBM 6x86 and 6x86L processor is fully compatible with the 486 instruction set, it is capable of executing a wide range of existing and future operation systems and applications including Windows, DOS, UNIX, Windows NT, Novell, OS/2, and Solaris.

The IBM 6x86 and 6x86L processor achieves unsurpassed performance levels through the use of super-pipelined integer units and an on-chip floating point unit. The superpipelined architecture reduces timing constraints and allows the IBM 6x86 and 6x86L processor to operate at core frequencies of 100 MHz and above. Additionally, the IBM 6x86 and 6x86L processor integer and floating point units are optimized for maximum instruction throughput by using advanced architectural techniques including register renaming, out-of-order completion, data forwarding, branch prediction, and speculative execution. These design innovations eliminate many data dependencies and resource conflicts that otherwise would degrade performance of existing non-optimized software programs.

Overview

The IBM 6x86 and 6x86L processor architecture achieves performance by incorporating both superscalar and superpipelined features. The **superscalar** architecture enables the IBM 6x86 and 6x86L microprocessor to execute multiple instructions in parallel. Traditionally, the disadvantage of a superscalar architecture is that the circuit complexity prohibits high frequency of operation. In contrast, the IBM 6x86 and 6x86L processor architecture divides the most complex stages of operation into simpler sub-stages. This technique is referred to as **superpipelining** and allows the superscalar IBM 6x86 and 6x86L processor architecture to operate at very high core frequencies (100 MHz and above).

The IBM 6x86 and 6x86L processor architecture consists of five major functional blocks as shown in the high-level block diagram:

- Integer Unit (IU)
- Floating Point Unit (FPU)
- Cache
- Memory Management Unit (MMU)
- Bus Interface Unit (BIU)

The IU, FPU and Cache are discussed in more detail in the following sections.

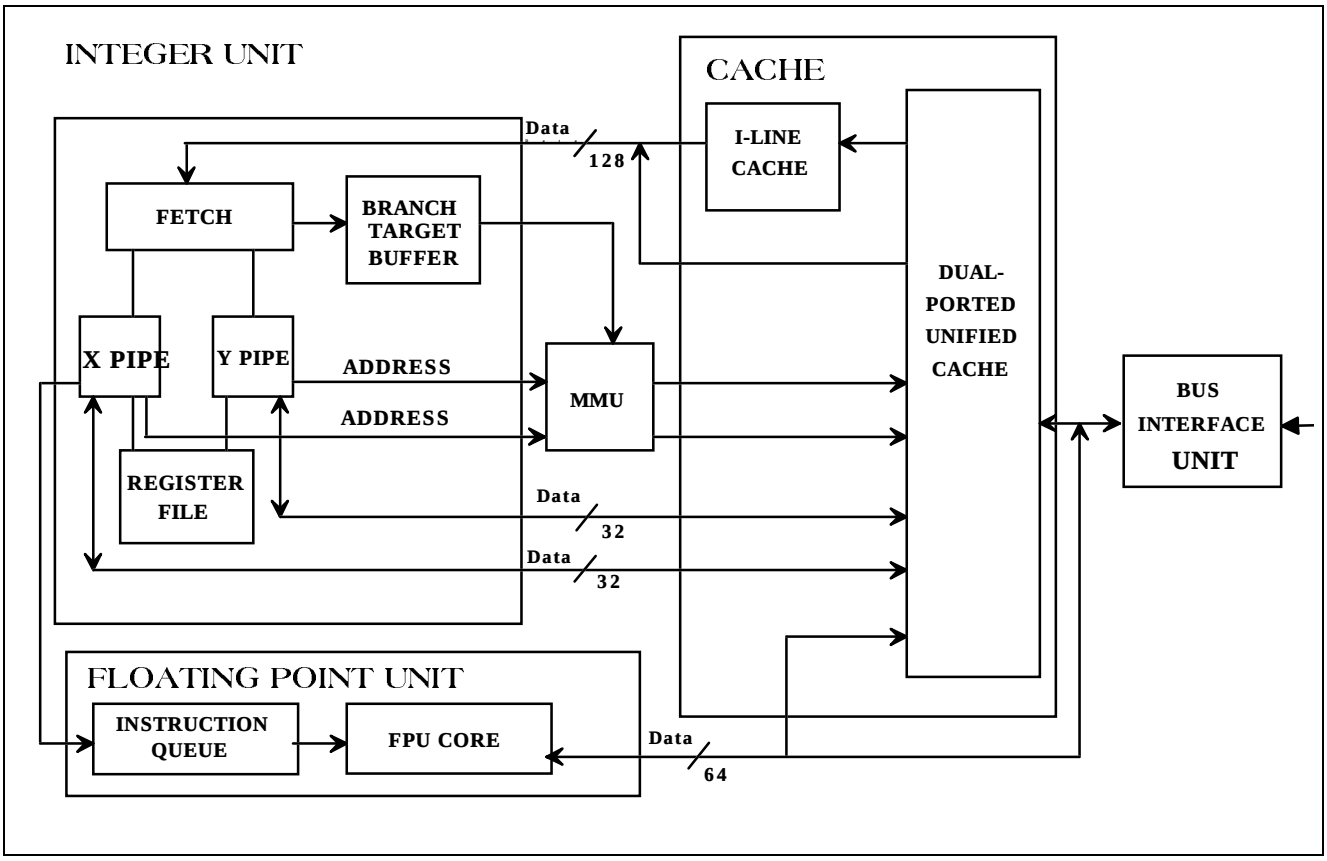

Figure 1. IBM 6x86 and 6x86L Microprocessor Architecture Overview

Interger Unit

Pipeline Description

The IBM 6x86 and 6x86L processor integer unit contains dual 7-stage integer pipelines, referred to as the X and Y pipelines, that provide parallel instruction execution capability. The 7 pipeline stages are:

- Prefetch (PF)
- Instruction Decode 1 (ID1)
- Instruction Decode 2 (ID2)
- Address Calculation 1 (AC1)
- Address Calculation 2 (AC2)
- Execute (EX)
- Write-back (WB)

.

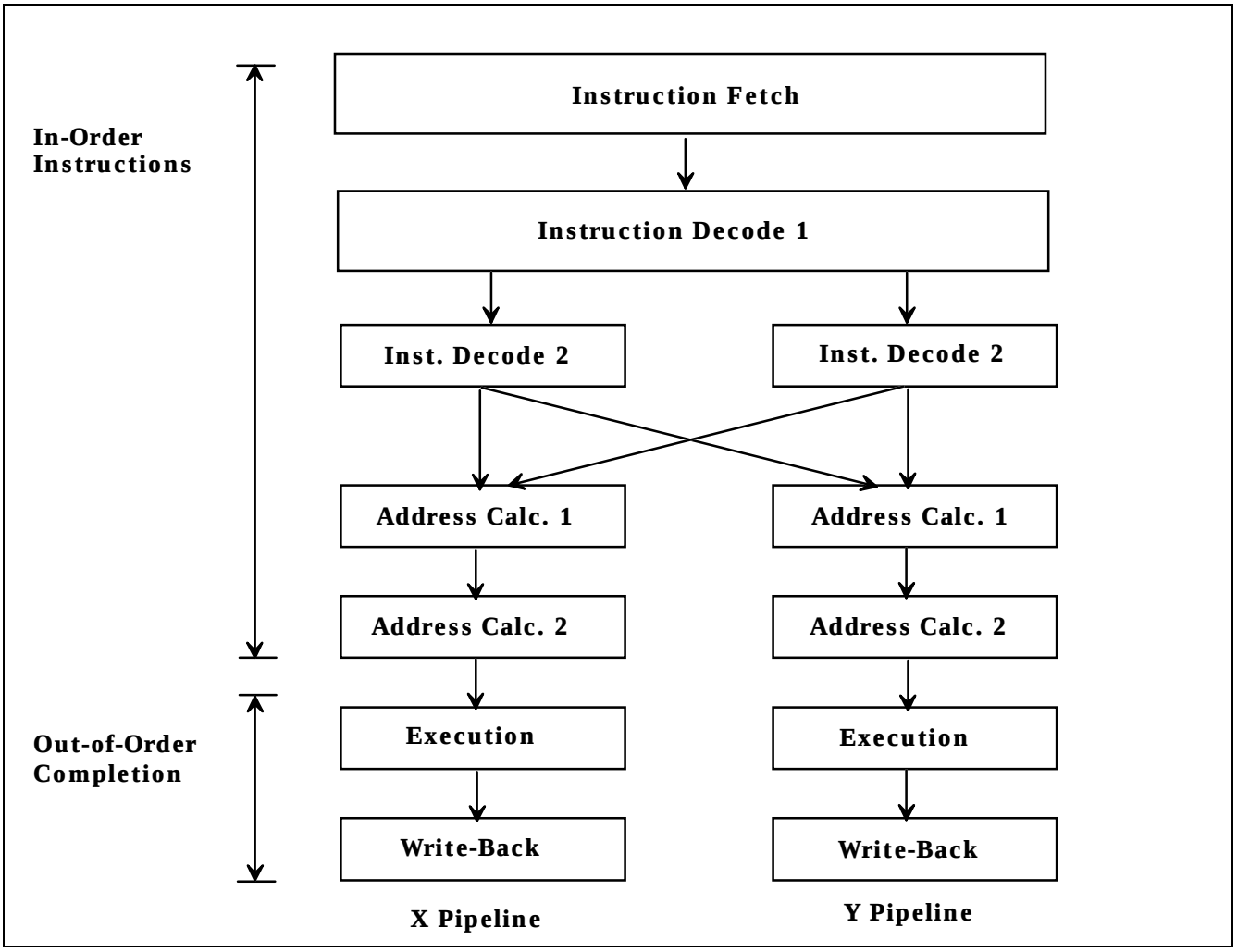

Figure 2. Integer Unit Pipelines

The **Prefetch** (PF) stage is common to both the X and Y pipe. During this stage, 16 bytes of code are fetched per core clock from the memory subsystem. Additionally, the code stream is checked to identify the presence of instructions that modify the normal sequential execution of the program. These instructions are referred to as branch instructions. Two types of branch instructions exist: (1) unconditional branches that always modify the instruction flow, and (2) conditional branches that modify the instruction flow based on a variable. If either type of branch instruction is detected, the branch prediction logic provides the predicted target address for the instruction. The prefetch stage then begins fetching at the predicted address.

The **Instruction Decode** stage is superpipelined and consists of two sub-stages ID1 and ID2. The ID1 stage evaluates the code stream provided by the prefetch stage and determines the number of instruction bytes for up to two instructions per clock. The ID2 stage then decodes the two instructions and selects either the X or Y pipeline for further execution. A load balancing algorithm is used for pipeline selection. This algorithm determines which pipeline is least likely to delay instruction completion due to interactions with previously dispatched instructions.

The **Address Calculation** stage is also superpipelined and consists of the two sub-stages AC1 and AC2. If the current instructions require memory operands, the AC1 stages calculate up to two linear memory addresses per clock (one per pipeline) and AC2 then performs the associated memory management functions and cache accesses. For register operands, register renaming occurs during AC1 and AC2 then accesses the register file. Additionally, floating point instructions aredispatched to the FPU during the AC2 stage. All instructions are kept in program order up to and during the AC1 and AC2 stages.

The **Execute** (EX) stage actually performs the instruction operation using the operands provided by the address calculation stage. The operation results are written to the register file and write buffers during the **Write-Back** (WB) stage. Once instructions have entered the EX stage, instructions in one pipeline may complete independently of the second pipeline. In other words, instructions may complete in a different order than they were dispatched. This is referred to as out-of-order completion. However, any resulting bus cycles are always issued in program order.

Optimized Pipeline Utilization

The IBM 6x86 and 6x86L processor architecture optimizes parallel use of the X and Y pipelines by allowing the majority of instructions to be dispatched in pairs, and by allowing the two pipelines to operate in a relatively independent fashion. These techniques maximize performance by reducing the number of clocks in which pipeline stages are idle.

Instruction Dispatch

The IBM 6x86 and 6x86L processor architecture enforces very few instruction pairing constraints. The most commonly used instructions in the X86 instruction set may be dispatched in pairs (from ID2) to either pipeline, regardless of dependencies that may exist between the two instructions. However, there are three categories of instructions that must be dispatched only in the X pipeline: (1) branch instructions, (2) floating point instructions, and (3) exclusive instructions.

The first two X-pipe-only instruction types, branch and floating point, may be paired with another instruction in the Y pipeline. Exclusive instructions may not be paired. Instructions are classified as exclusive if they may fault in the EX pipe stage and are typically instructions that require multiple memory accesses. Although exclusive instructions may not be paired, hardware from both pipelines is used to accelerate instruction completion. The IBM 6x86 and 6x86L processor exclusive instruction types are listed below:

- 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)
- Task switches

Out-Of-Order Completion

Out-of-order completion occurs in the EX and WB stages when an instruction in one pipeline completes prior to a previously dispatched instruction in the adjacent pipeline that requires multiple clocks to complete. This type of processing is primarily used when an instruction in one pipeline is stalled waiting for a memory access to complete. Under this condition, the current and subsequent instructions in the EX stage of the adjacent pipe can be completed without waiting for the pending access to complete, assuming no inter-instruction dependencies.

The IBM 6x86 and 6x86L processor architecture always supplies instructions in program order to the EX stage, and allows instructions to complete out of order only from that point on. In conjunction with exclusive instructions, this ensures that exceptions occur in program order. Also, writes resulting from instructions completed out of order are always issued to the cache or external bus in program order. Thus, X86 software compatibility is maintained.

Data Dependency Removal

The IBM 6x86 and 6x86L microprocessor incorporates key architectural features that eliminate idle pipeline stages resulting from inter-instruction data dependencies. A combination of register renaming, data forwarding and data bypassing techniques are used to eliminate write-after-write (WAW), write-after-read (WAR) and read-after-write (RAW) data dependencies.

Register Renaming

The IBM 6x86 and 6x86L processor architecture contains 32 physical general purpose registers. These 32 registers are mapped, or renamed, to any one of the 8 logical general purpose registers defined by the X86 architecture (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP). This renaming is controlled entirely by on-chip hardware and is therefore transparent to software.

Each time a write to a logical register occurs, a new physical register is assigned to the logical register. This prevents overwriting the previous data in the logical register and thus eliminates write-after-write (WAW) and write-after-read (WAR) dependencies as illustrated in the following examples.

WAR Dependency Removal Example

Assume the following instructions are executing simultaneously in the X and Y pipelines:

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

X PIPE **Y PIPE**

(1) BX <- AX (2) AX <- AX + CX

A WAR dependency exists with the AX register because the Y pipe must wait for the X pipe to read AX before the add instruction in the Y pipe updates the value of AX. This causes the Y pipe to stall in an architecture where register renaming is not used and out-of-order completion is allowed.

In the IBM 6x86 and 6x86L microprocessor, physical registers are substituted for the logical registers. The operations are completed in parallel with no Y pipeline stall as shown below.

Initial assignments: AX = reg0
BX = reg1
CX = reg2

X PIPE **Y PIPE**

(1) reg3 <- reg0 (2) reg4 <- reg0 + reg2

Final assignments: AX = reg4
BX = reg3
CX = reg2

WAW Dependency Removal Example

Assume the following instructions are executing simultaneously in the X and Y pipelines:

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

X PIPE **Y PIPE**

(1) AX <- mem (2) AX <- AX + BX

The X pipe issues a memory access. The Y pipe is waiting for the same memory data as the X pipe to be used in the ADD calculation. Using data forwarding (see Data Forwarding), the memory operand is available to both pipelines at the same time. A WAW dependency is created with AX because the Y pipe must wait for the X pipe to update AX before the Y pipe can write the result of the ADD instruction to AX. This causes the Y pipe to stall in an architecture where register renaming is not used. Using register renaming, the IBM 6x86 and 6x86L processor substitutes physical registers for the logical registers. The operations are completed in parallel with no Y pipeline stall as shown below:

Initial assignments: AX = reg0 BX = reg1

X PIPE Y PIPE

(1) reg2 <- mem (2) reg3 <- mem + reg1

Final assignments: AX = reg3
BX = reg1

Data Forwarding

In addition to register renaming, the IBM 6x86 and 6x86L processor architecture incorporates a technique called Data Forwarding that is used to eliminate read-after-write register and memory dependencies. Data forwarding allows pairs of instructions with RAW register dependency to execute simultaneously, thus eliminating pipeline stalls. The IBM 6x86 and 6x86L processor architecture implements two types of data forwarding: (1) operand forwarding, and (2) result forwarding.

Operand forwarding occurs when a MOV instruction is used to load data into a register or memory location. The register or memory location is then used in a subsequent instruction as an operand creating a RAW dependency on the operand register or memory location. Using operand forwarding, the load data is immediately made available to the subsequent instruction without waiting for the completion of the MOV instruction. Operand forwarding is illustrated in the following example.

Operand Forwarding Example

Assume the following instructions are executing simultaneously in the X and Y pipelines:

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

X PIPE Y PIPE

(1) AX <- (mem) (2) BX <- AX + BX

A RAW dependency exists with AX because the Y pipe must wait for the X pipe to load the memory data into AX before AX register can be used as an operand in the Y pipe's add instruction. The IBM 6x86 and 6x86L microprocessor uses operand forwarding to eliminate the RAW dependency and the associated Y pipe stall. Thus, the two instructions complete in parallel instead of serially as illustrated below.

Initial assignments: AX = reg0
BX = reg1

X PIPE Y PIPE

(1) reg2 <- [mem] (2) reg3 <- [(mem] + reg1

Final Assignments: AX = reg2
BX = reg3

Result forwarding occurs when an instruction stores the results of its operation in either a register or memory location. A subsequent MOV instruction is then used to transfer the result data to a second location creating a RAW dependency on the result data location. Using result forwarding, the result data is transferred to the second location without waiting for the first instruction to complete its store. Result forwarding is illustrated in the following example.

Result Forwarding Example

Assume the following instructions are executing simultaneously in the X and Y pipelines:

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

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

A RAW dependency exists with AX because the Y pipe must wait for the X pipe to store the result of the add instruction in AX before the AX register can be stored in the desired memory location. The 6x86 and 6x86L microprocessor uses result forwarding to eliminate the RAW dependency and associated Y pipe stalls. Thus, the two instructions complete in parallel instead of serially as illustrated below.

Initial assignments: AX = reg0
BX = reg1

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

Final assignments: AX = reg2
BX = reg1

Data Bypassing

The IBM 6x86 and 6x86L processor architecture incorporates a third technique, referred to as data bypassing, to reduce the performance penalty associated with read-after-write memory dependencies. Data bypassing occurs when a memory location used to store the result of an operation is then used as an operand in a subsequent instruction. This creates a RAW dependency with the memory location. Using data bypassing, the result of the first instruction is immediately available for use as an operand in the subsequent instruction without waiting for completion of the memory write. Data bypassing is illustrated in the following example.

Data Bypassing Example

Assume the following instructions are executing simultaneously in the X and Y pipelines:

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

X PIPE Y PIPE

(1) mem <- mem + AX (2) BX <- BX - mem

A RAW dependency exists with [mem] because the Y pipe must wait for the X pipe to store the result of the ADD instruction in[mem] before [mem] can be used as an operand in the Y pipe's subtract instruction. Using data bypassing, the Y pipe must still wait for the X pipe to complete an add operation before performing the subtract instruction. However, data bypassing minimizes the number of clocks that the Y pipe is stalled by passing the result of the add instruction directly to the Y pipe without waiting for the result to be written to and then read from memory as shown below.

Initial assignments: AX = reg0
BX = reg1

X PIPE Y PIPE

(1) mem <- mem + reg0 (2) reg2 <- reg1 - (mem + reg0)

Final assignments: AX = reg0
BX = reg2

Branch Control

Branch instructions occur, on average, every four to six instructions in x86 compatible programs. Branch instructions typically change the normal sequential flow of the program. This can cause pipeline stages to stall waiting for the CPU to calculate, retrieve and decode the new instruction stream. The IBM 6x86 and 6x86L processor architecture minimizes the performance impact of latency of branch instructions through the use of branch prediction and speculative execution.

Branch Prediction

The IBM 6x86 and 6x86L processor architecture uses a Branch Target Buffer (BTB) to store branch target addresses and branch prediction information. During the prefetch stage, the instruction stream is checked for the presence of branch instructions. If an unconditional branch instruction is encountered, the processor access the BTB to check for the branch instruction's target address. If the branch instruction hits in the BTB, the processor begins fetching at the target address specified by the BTB.

In the 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 hits in the BTB, the IBM 6x86 and 6x86L processor begins fetching instructions at the predicted target address. If the conditional branch misses in the BTB, the IBM 6x86 and 6x86L processor predicts 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 4-state branch prediction algorithm that achieves approximately 90% prediction accuracy.

Once prefetched, a branch instruction is decoded and then dispatched to the X pipeline only. The branch instruction proceeds through the X pipeline and is then resolved in either the EX stage or the WB stage. The 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 setting the condition codes is executed in parallel with the branch, the branch instruction is resolved in the WB stage.

Correctly predicted branch instructions execute in a single clock. If resolution of a branch indicates that a misprediction has occurred, the IBM 6x86 and 6x86L processor flushes the pipeline and fetches from the correct target address. The correct instruction stream is then fed directly from the cache to the ID1 stage. The resulting misprediction latency is 4 or 5 cycles depending on whether the branch is resolved in EX or WB, respectively.

Since the target address of return (RET) instructions is dynamic rather than static, the microprocessor caches target addresses for RET instructions in a 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. Using the return stack, RET instructions execute in a single clock.

Speculative Execution

The IBM 6x86 and 6x86L processor architecture is capable of speculative execution following a predicted branch or floating point instruction. 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 in parallel with integer instructions.

The processor architecture is capable of up to four levels of speculation (i.e. combination of four conditional branches and/or floating point operations). After generating the prefetch 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 processor generates the correct prefetch 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 (segment registers, system flags).

Floating Point Unit

The IBM 6x86 and 6x86L processor Floating Point Unit (FPU) interfaces to the integer unit and the cache via a 64-bit bus. The processor FPU is x87 instruction set compatible and adheres to the IEEE-754 standard. Because most applications contain FPU instructions inter-mixed with integer instructions, the IBM 6x86 and 6x86L processor FPU architecture achieves high performance by completing integer and FPU operations in parallel.

FPU Parallel Execution

The IBM 6x86 and 6x86L processor executes integer instructions in parallel with FPU instructions. Additionally, integer instructions may complete out-of-order with respect to the FPU instructions. The processor 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 for use by the FPU. If no exceptions are detected, the IBM 6x86 and 6x86L processor checkpoints the state of the CPU and, during AC2, dispatches the floating point instruction to the FPU instruction queue. The processor can then complete any subsequent integer instructions speculatively and out-of-order relative to the FPU instruction and relative to any potential FPU exceptions that may occur.

As additional FPU instructions enter the pipeline, the IBM 6x86 and 6x86L processor dispatches up to four FPU instructions to the FPU instruction queue. The IBM 6x86 and 6x86L processor continues executing speculatively and out-of-order, relative to the FPU queue, unless the microprocessor encounters one of the conditions that causes speculative execution to halt. As the FPU completes instructions, the speculation level decreases and the checkpointed resources are available for reuse in subsequent operations. The IBM 6x86 and 6x86L processor FPU also uses a set of 4 write buffers to prevent stalls due to speculative writes.

Cache

The IBM 6x86 and 6x86L microprocessor contains an on-chip dual-ported unified instruction/data cache and a fully associative 256-byte instruction line (I-Line) cache that store the contents of the most commonly used memory locations. The unified cache functions as the primary data cache and the secondary instruction cache. The I-Line cache functions as the primary instruction cache.

The instruction line cache is filled from the unified cache by the prefetching unit. Prefetches that hit in the I-Line cache do not access the unified cache. However, if an I-Line cache miss occurs, the prefetching unit reads from the unified cache and fills the I-Line cache simultaneously. The I-Line cache uses a pseudo-LRU algorithm for cache line replacements. To ensure proper operation in the case of self-modifying code, any writes to the unified cache are checked against the contents of the I-Line cache. If a hit occurs in the I-Line cache, the appropriate line is invalidated.

The unified cache is dual-ported to allow any two of the following operations to occur in parallel:

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

Cache Architectural Benefits

The IBM 6x86 and 6x86L processor two-level cache architecture has two important advantages: (1) high hit rate, and (2) high bandwidth. The high hit rate is achieved because the unified architecture of the cache allows large portions of the cache to be allocated for code or data storage if required. The same situation could easily overflow a cache architecture of comparable size with separate instruction and data caches. Additionally, the unified cache is 4-way set associative resulting in an even better hit rate.

The second advantage of the IBM 6x86 and 6x86L processor cache architecture is high bandwidth. High bandwidth is typically only achieved by implementing separate instruction and data caches. The IBM 6x86 and 6x86L processor cache achieves bandwidth to the unified cache by implementing the instruction line cache as the primary instruction cache. Using the I-Line cache, data references and only instruction fetches that miss in the I-Line cache access the unified cache directly. Additionally, the unified cache is dual-ported allowing two accesses to occur in parallel.

Summary

The IBM 6x86 and 6x86L processor architecture is a high performance x86-compatible processor architecture that provides next generation performance to both existing non-optimized PC-compatible software as well as future 32-bit applications. By combining a sophisticated architecture with unique advanced features, the IBM 6x86 and 6x86L processor achieves high performance without the need for software recompilation.