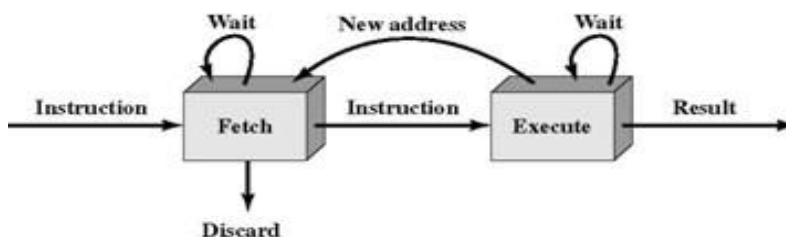


UNIT-3❖ INSTRUCTION PIPELINING

As computer systems evolve, greater performance can be achieved by taking advantage of improvements in technology, such as faster circuitry, use of multiple registers rather than a single accumulator, and the use of a cache memory. Another organizational approach is instruction pipelining in which new inputs are accepted at one end before previously accepted inputs appear as outputs at the other end.



3.1(a) Simplified View



3.1(b) Expanded View

3.1 Two-Stage Instruction Pipeline

Figure 3.1a depicts this approach. The pipeline has two independent stages. The first stage fetches an instruction and buffers it. When the second stage is free, the first stage passes it the buffered instruction. While the second stage is executing the instruction, the first stage takes advantage of any unused memory cycles to fetch and buffer the next instruction. This is called *instruction prefetch* or *fetch overlap*.

This process will speed up instruction execution only if the fetch and execute stages were of equal duration, the instruction cycle time would be halved. However, if we look more closely at this pipeline (Figure 3.1b), we will see that this doubling of execution rate is unlikely for 3 reasons:

1 The execution time will generally be longer than the fetch time. Thus, the fetch stage may have to wait for some time before it can empty its buffer.

2 A conditional branch instruction makes the address of the next instruction to be fetched unknown. Thus, the fetch stage must wait until it receives the next instruction address from the execute stage. The execute stage may then have to wait while the next instruction is fetched.

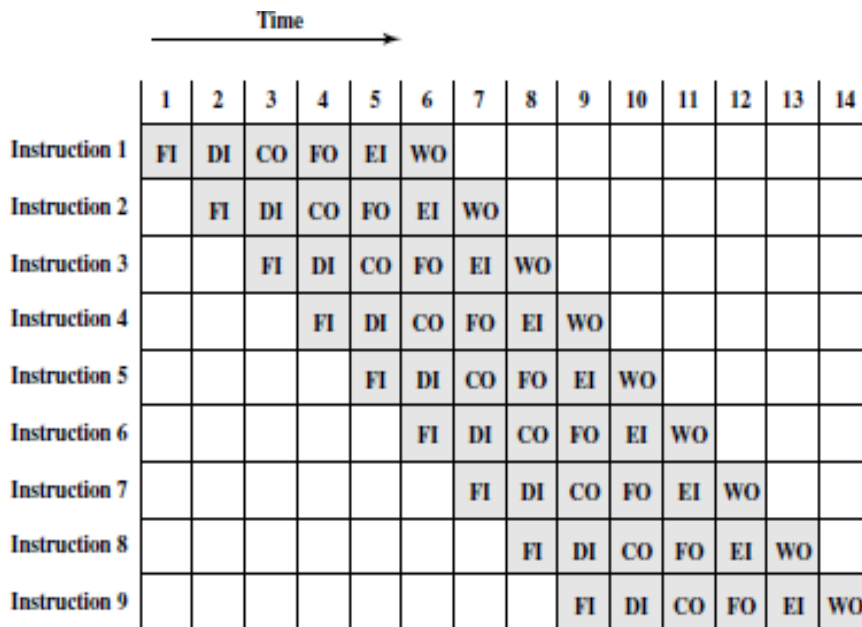
3 When a conditional branch instruction is passed on from the fetch to the execute stage, the fetch stage fetches the next instruction in memory after the branch instruction. Then, if the branch is not taken, no time is lost. If the branch is taken, the fetched instruction must be discarded and a new instruction fetched.

To gain further speedup, the pipeline must have more stages. Let us consider the following decomposition of the instruction processing.

1. **Fetch instruction (FI):** Read the next expected instruction into a buffer.
2. **Decode instruction (DI):** Determine the opcode and the operand specifiers.

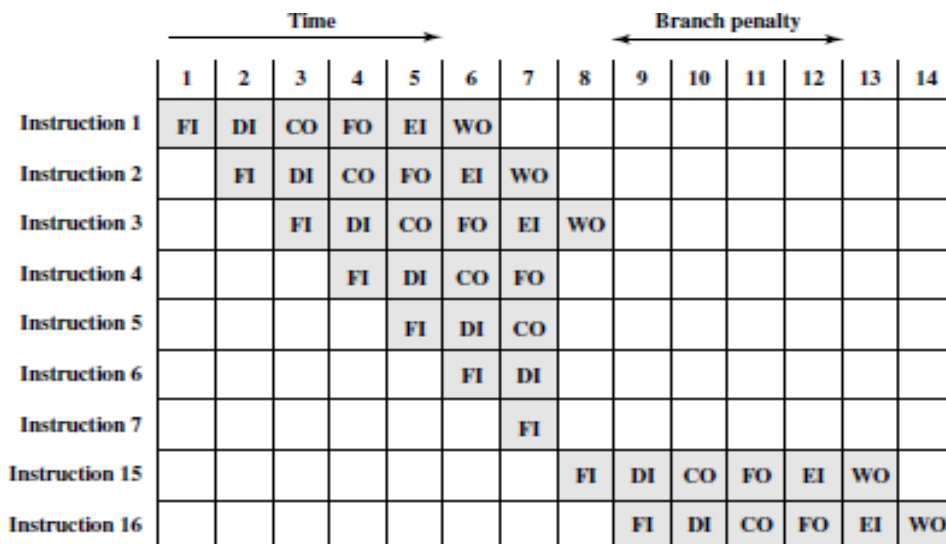
3. **Calculate operands (CO):** Calculate the effective address of each source operand. This may involve displacement, register indirect, indirect, or other forms of address calculation.
4. **Fetch operands (FO):** Fetch each operand from memory.
5. **Execute instruction (EI):** Perform the indicated operation and store the result, if any, in the specified destination operand location.
6. **Write operand (WO):** Store the result in memory.

Figure 3.2 shows that a six-stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.



3.2 Timing Diagram for Instruction Pipeline Operation

FO and WO stages involve a memory access. If the six stages are not of equal duration, there will be some waiting involved at various pipeline stages. Another difficulty is the conditional branch instruction, which can invalidate several instruction fetches. A similar unpredictable event is an interrupt.

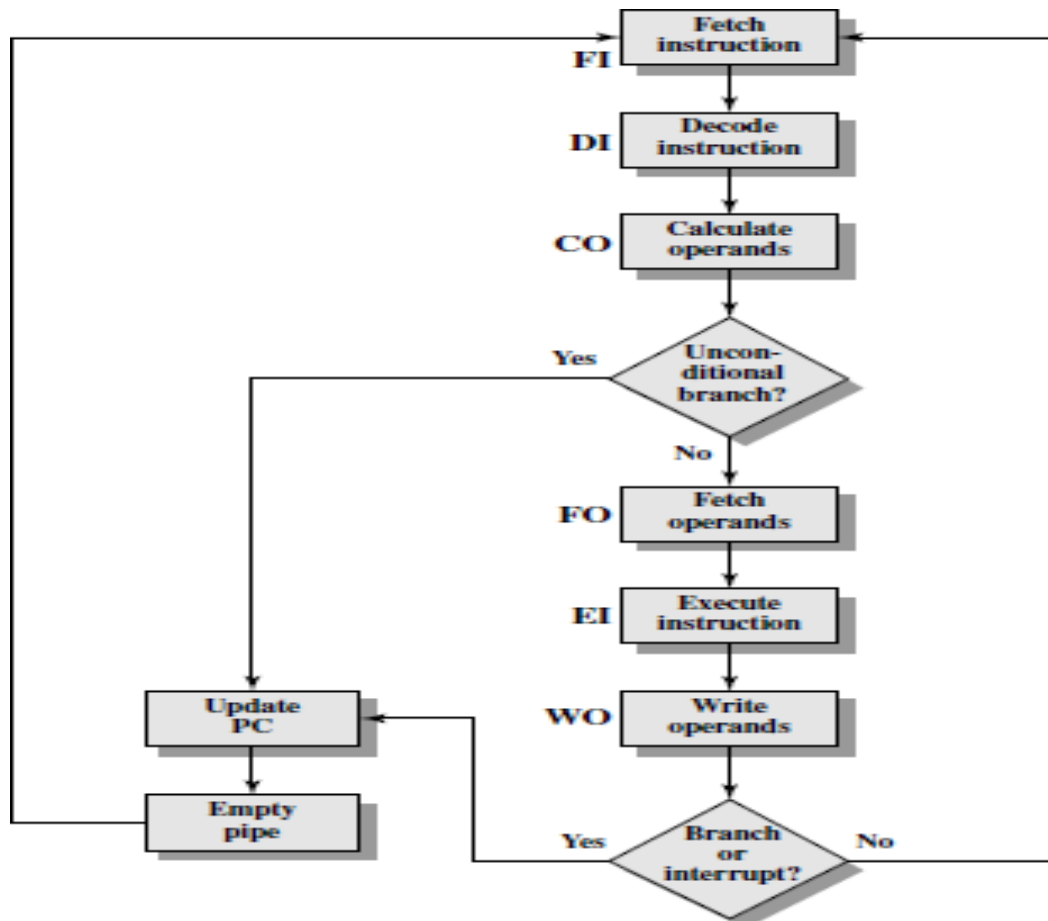


3.3 Timing Diagram for Instruction Pipeline Operation with interrupts

Figure 3.3 illustrates the effects of the conditional branch, using the same program as Figure 3.2. Assume that instruction 3 is a conditional branch to instruction 15. Until the instruction is executed, there is no way of knowing which instruction will come next. The pipeline, in this example, simply loads the next instruction in sequence (instruction 4) and proceeds.

In Figure 3.2, the branch is not taken. In Figure 3.3, the branch is taken. This is not determined until the end of time unit 7. At this point, the pipeline must be cleared of instructions that are not useful. During time unit 8, instruction 15 enters the pipeline.

No instructions complete during time units 9 through 12; this is the performance penalty incurred because we could not anticipate the branch. Figure 3.4 indicates the logic needed for pipelining to account for branches and interrupts.



3.4 Six-stage CPU Instruction Pipeline

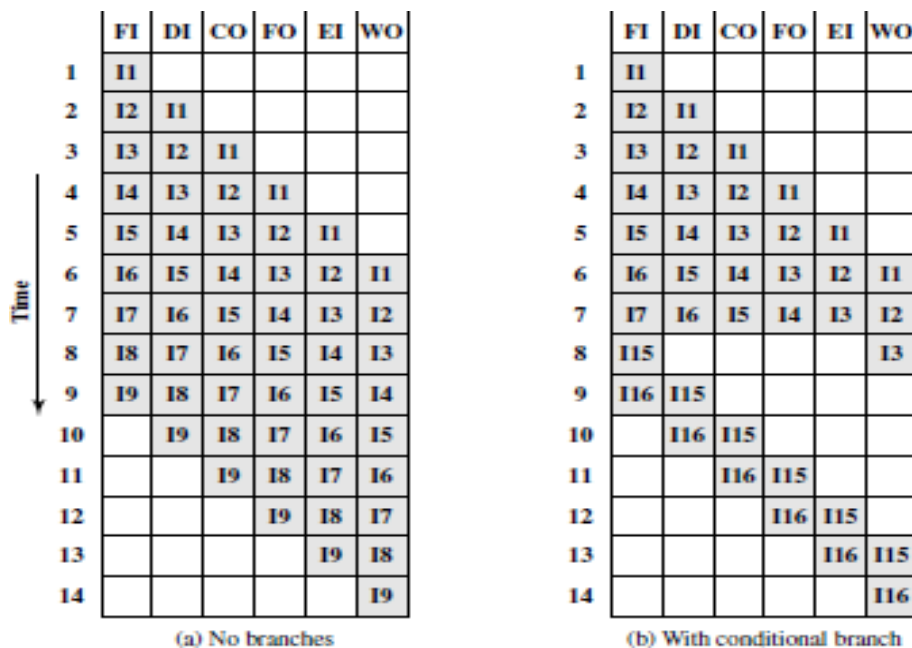
Figure 3.5 shows same sequence of events, with time progressing vertically down the figure, and each row showing the state of the pipeline at a given point in time. In Figure 3.5a (which corresponds to Figure 3.2), the pipeline is full at time 6, with 6 different instructions in various stages of execution, and remains full through time 9; we assume that instruction I9 is the last instruction to be executed. In Figure 3.5b, (which corresponds to Figure 3.3), the pipeline is full at times 6 and 7. At time 7, instruction 3 is in the execute stage and executes a branch to instruction 15. At this point, instructions I4 through I7 are flushed from the pipeline, so that at time 8, only two instructions are in the pipeline, I3 and I15.

For high-performance in pipelining designer must still consider about :

1 At each stage of the pipeline, there is some overhead involved in moving data from buffer to buffer and in performing various preparation and delivery functions. This overhead can appreciably lengthen the total execution time of a single instruction.

2 The amount of control logic required to handle memory and register dependencies and to optimize the use of the pipeline increases enormously with the number of stages. This can lead to a situation where the logic controlling the gating between stages is more complex than the stages being controlled.

3 Latching delay: It takes time for pipeline buffers to operate and this adds to instruction cycle time.



3.5 An Alternative Pipeline depiction

Pipelining Performance

Measures of pipeline performance and relative speedup:

The cycle time t of an instruction pipeline is the time needed to advance a set of instructions one stage through the pipeline; each column in Figures 3.2 and 3.3 represents one cycle time.

The cycle time can be determined as

$$t = \max [t_i] + d = t_m + d \quad 1 \dots i \dots k$$

where

t_i = time delay of the circuitry in the i th stage of the pipeline

t_m = maximum stage delay (delay through stage which experiences the largest delay)

k = number of stages in the instruction pipeline

d = time delay of a latch, needed to advance signals and data from one stage to the next

In general, the time delay d is equivalent to a clock pulse and $t_m \ll d$. Now suppose that n instructions are processed, with no branches. Let $T_{k,n}$ be the total time required for a pipeline with k stages to execute n instructions. Then

$$T_{k,n} = [k + (n - 1)]t$$

A total of k cycles are required to complete the execution of the first instruction, and the remaining $n - 1$ instructions require $n - 1$ cycles. This equation is easily verified from Figures 3.1. The ninth instruction completes at time cycle 14:

$$14 = [6 + (9 - 1)]t$$

Now consider a processor with equivalent functions but no pipeline, and assume that the instruction cycle time is kt . The speedup factor for the instruction pipeline compared to execution without the pipeline is defined as

$$S_k = \frac{T_{1,n}}{T_{k,n}} = \frac{nk\tau}{[k + (n - 1)]\tau} = \frac{nk}{k + (n - 1)}$$

❖ **Pipeline Hazards**

A **pipeline hazard** occurs when the pipeline, or some portion of the pipeline, must stall because conditions do not permit continued execution. Such a pipeline stall is also referred to as a *pipeline bubble*. There are three types of hazards: resource, data, and control.

RESOURCE HAZARDS A resource hazard occurs when two (or more) instructions that are already in the pipeline need the same resource. The result is that the instructions must be executed in serial rather than parallel for a portion of the pipeline. A resource hazard is sometime referred to as a *structural hazard*.

Let us consider a simple example of a resource hazard. Assume a simplified five-stage pipeline, in which each stage takes one clock cycle. In Figure 3.6a which a new instruction enters the pipeline each clock cycle. Now assume that main memory has a single port and that all instruction fetches and data reads and writes must be performed one at a time. In this case, an operand read to or write from memory cannot be performed in parallel with an instruction fetch. This is illustrated in Figure 3.6b, which assumes that the source operand for instruction I1 is in memory, rather than a register. Therefore, the fetch instruction stage of the pipeline must idle for one cycle before beginning the instruction fetch for instruction I3. The figure assumes that all other operands are in registers.

		Clock cycle								
		1	2	3	4	5	6	7	8	9
Instruction	I1	FI	DI	FO	EI	WO				
	I2		FI	DI	FO	EI	WO			
	I3			FI	DI	FO	EI	WO		
	I4				FI	DI	FO	EI	WO	

(a) Five-stage pipeline, ideal case

		Clock cycle								
		1	2	3	4	5	6	7	8	9
Instruction	I1	FI	DI	FO	EI	WO				
	I2		FI	DI	FO	EI	WO			
	I3			Idle	FI	DI	FO	EI	WO	
	I4					FI	DI	FO	EI	WO

(b) I1 source operand in memory

3.6 Example of Resource Hazard

DATA HAZARDS A data hazard occurs when two instructions in a program are to be executed in sequence and both access a particular memory or register operand. If the two instructions are executed in strict sequence, no problem occurs but if the instructions are executed in a pipeline, then the operand value is to be updated in such a way as to produce a different result than would occur only with strict sequential execution of instructions. The program produces an incorrect result because of the use of pipelining.

As an example, consider the following x86 machine instruction sequence:

ADD EAX, EBX /* EAX = EAX + EBX

SUB ECX, EAX /* ECX = ECX - EAX

The first instruction adds the contents of the 32-bit registers EAX and EBX and stores the result in EAX. The second instruction subtracts the contents of EAX from ECX and stores the result in ECX.

Figure 3.7 shows the pipeline behaviour. The ADD instruction does not update register EAX until the end of stage 5, which occurs at clock cycle 5. But the SUB instruction needs that value at the beginning of its stage 2, which occurs at clock cycle 4. To maintain correct operation, the pipeline must stall for two clocks cycles. Thus, in the absence of special hardware and specific avoidance algorithms, such a data hazard results in inefficient pipeline usage. There are three types of data hazards;

	Clock cycle									
	1	2	3	4	5	6	7	8	9	10
ADD EAX, EBX	FI	DI	FO	EI	WO					
SUB ECX, EAX		FI	DI	Idle		FO	EI	WO		
B3			FI			DI	FO	EI	WO	
B4						FI	DI	FO	EI	WO

3.7 Example of Resource Hazard

- **Read after write (RAW), or true dependency:** A hazard occurs if the read takes place before the write operation is complete.
- **Write after read (RAW), or antidependency:** A hazard occurs if the write operation completes before the read operation takes place.
- **Write after write (RAW), or output dependency:** Two instructions both write to the same location. A hazard occurs if the write operations take place in the reverse order of the intended sequence. The example of Figure 3.7 is a RAW hazard.

CONTROL HAZARDS A control hazard, also known as a *branch hazard*, occurs when the pipeline makes the wrong decision on a branch prediction and therefore brings instructions into the pipeline that must subsequently be discarded.

❖ **Dealing with Branches**

Until the instruction is actually executed, it is impossible to determine whether the branch will be taken or not. A variety of approaches have been taken for dealing with conditional branches:

- Multiple streams
- Prefetch branch target
- Loop buffer
- Branch prediction
- Delayed branch

MULTIPLE STREAMS A simple pipeline suffers a penalty for a branch instruction because it must choose one of two instructions to fetch next and may make the wrong choice.

A brute-force approach is to replicate the initial portions of the pipeline and allow the pipeline to fetch both instructions, making use of two streams. There are two problems with this approach:

- With multiple pipelines there are contention delays for access to the registers and to memory.
- Additional branch instructions may enter the pipeline (either stream) before the original branch decision is resolved. Each such instruction needs an additional stream.

Examples of machines with two or more pipeline streams are the IBM 370/168 and the IBM 3033.

PREFETCH BRANCH TARGET When a conditional branch is recognized, the target of the branch is prefetched, in addition to the instruction following the branch. This target is then saved until the branch instruction is executed. If the branch is taken, the target has already been prefetched.

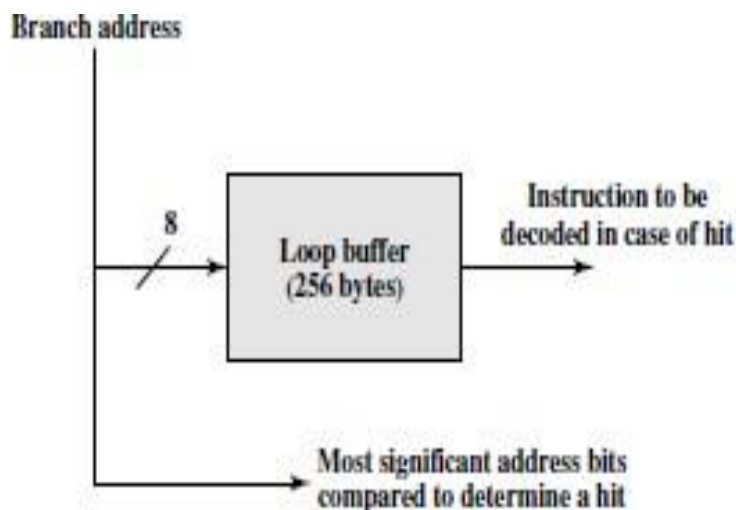
Example- The IBM 360/91 uses this approach.

LOOP BUFFER A loop buffer is a small, very-high-speed memory maintained by the instruction fetch stage of the pipeline and containing the n most recently fetched instructions, in sequence. If a branch is to be taken, the hardware first checks whether the branch target is within the buffer. If so, the next instruction is fetched from the buffer.

The loop buffer has three benefits:

1. With the use of prefetching, the loop buffer will contain some instruction sequentially ahead of the current instruction fetch address.
2. If a branch occurs to a target just a few locations ahead of the address of the branch instruction, the target will already be in the buffer.
3. This strategy is particularly well suited to dealing with loops, or iterations; hence the name *loop buffer*. If the loop buffer is large enough to contain all the instructions in a loop, then those instructions need to be fetched from memory only once, for the first iteration. For subsequent iterations, all the needed instructions are already in the buffer.

Figure 3.8 gives an example of a loop buffer



3.8 Loop Buffer

BRANCH PREDICTION Various techniques can be used to predict whether a branch will be taken. Among the more common are the following:

- Predict never taken
- Predict always taken
- Predict by opcode
- Taken/not taken switch
- Branch history table

The first three approaches are static: they do not depend on the execution history up to the time of the conditional branch instruction. The latter two approaches are dynamic: They depend on the execution history.

The first two approaches are the simplest. These either always assume that the branch will not be taken or continue to fetch instructions in sequence, or they always assume that the branch will be taken and always fetch from the branch target. The predict-never-taken approach is the most popular of all the branch prediction methods.

DELAYED BRANCH It is possible to improve pipeline performance by automatically rearranging instructions within a program, so that branch instructions occur later than actually desired.

❖ 8086 Processor Family

The x86 organization has evolved dramatically over the years.

Register Organization

The register organization includes the following types of registers (Table 3.1):

(a) Integer Unit in 32-bit Mode			
Type	Number	Length (bits)	Purpose
General	8	32	General-purpose user registers
Segment	6	16	Contain segment selectors
EFLAGS	1	32	Status and control bits
Instruction Pointer	1	32	Instruction pointer

(b) Integer Unit in 64-bit Mode			
Type	Number	Length (bits)	Purpose
General	16	32	General-purpose user registers
Segment	6	16	Contain segment selectors
RFLAGS	1	64	Status and control bits
Instruction Pointer	1	64	Instruction pointer

(c) Floating-Point Unit			
Type	Number	Length (bits)	Purpose
Numeric	8	80	Hold floating-point numbers
Control	1	16	Control bits
Status	1	16	Status bits
Tag Word	1	16	Specifies contents of numeric registers
Instruction Pointer	1	48	Points to instruction interrupted by exception
Data Pointer	1	48	Points to operand interrupted by exception

Table 3.1: X86 Processor Registers

- **General:** There are eight 32-bit general-purpose registers. These may be used for all types of x86 instructions; and some of these registers also serve special purposes. For example, string instructions use the contents of the ECX, ESI, and EDI registers as operands without having to reference these registers explicitly in the instruction.

- **Segment:** The six 16-bit segment registers contain segment selectors, which index into segment tables, as discussed in Chapter 8. The code segment (CS) register references the segment containing the instruction being executed. The stack segment (SS) register references the segment containing a user-visible stack. The remaining segment registers (DS, ES, FS, GS) enable the user to reference up to four separate data segments at a time.

- **Flags:** The 32-bit EFLAGS register contains condition codes and various mode bits. In 64-bit mode, this register is extended to 64 bits and referred to as RFLAGS. In the current architecture definition, the upper 32 bits of RFLAGS are unused.

- **Instruction pointer:** Contains the address of the current instruction. There are also registers specifically devoted to the floating-point unit:

- **Numeric:** Each register holds an extended-precision 80-bit floating-point number. There are eight registers that function as a stack, with push and pop operations available in the instruction set.

- **Control:** The 16-bit control register contains bits that control the operation of the floating-point unit, including the type of rounding control; single, double, or extended precision; and bits to enable or disable various exception conditions.

- **Status:** The 16-bit status register contains bits that reflect the current state of the floating-point unit, including a 3-bit pointer to the top of the stack; condition codes reporting the outcome of the last operation; and exception flags.

- **Tag word:** This 16-bit register contains a 2-bit tag for each floating-point numeric register, which indicates the nature of the contents of the corresponding register. The four possible values are valid, zero, special (NaN, infinity, denormalized), and empty.

EFLAGS REGISTER The EFLAGS register (Figure 3.9) indicates the condition of the processor and helps to control its operation. It includes the six condition codes (carry, parity, auxiliary, zero, sign, overflow), which report the results of an integer operation. In addition, there are bits in the register that may be referred to as control bits:

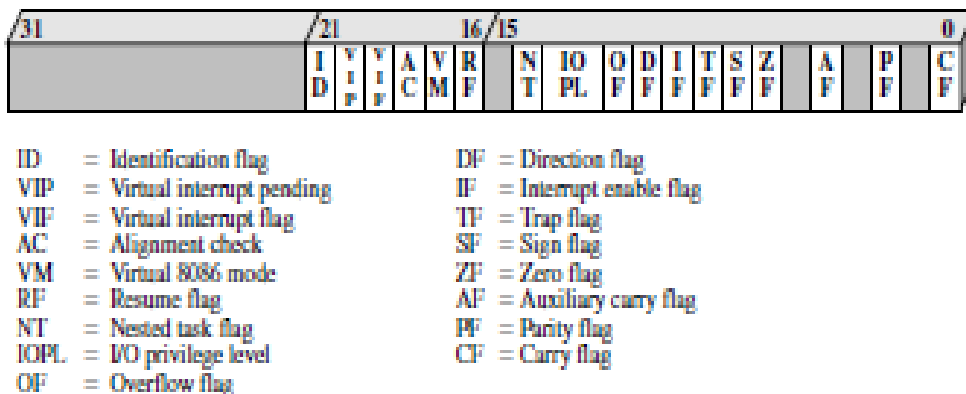


Figure 3.9 PentiumII EFLAGS Register

- **Trap flag (TF):** When set, causes an interrupt after the execution of each instruction. This is used for debugging.

- **Interrupt enable flag (IF):** When set, the processor will recognize external interrupts.

- **Direction flag (DF):** Determines whether string processing instructions increment or decrement the 16-bit half-registers SI and DI (for 16-bit operations) or the 32-bit registers ESI and EDI (for 32-bit operations).

- **I/O privilege flag (IOPL):** When set, causes the processor to generate an exception on all accesses to I/O devices during protected-mode operation.

- **Resume flag (RF):** Allows the programmer to disable debug exceptions so that the instruction can be restarted after a debug exception without immediately causing another debug exception.

- **Alignment check (AC):** Activates if a word or doubleword is addressed on a nonword or nondoubleword boundary.

- **Identification flag (ID):** If this bit can be set and cleared, then this processor supports the processorID instruction. This instruction provides information about the vendor, family, and model.

- **Nested Task (NT) flag** indicates that the current task is nested within another task in protected-mode operation.

- **Virtual Mode (VM) bit** allows the programmer to enable or disable virtual 8086 mode, which determines whether the processor runs as an 8086 machine.

- **Virtual Interrupt Flag (VIF) and Virtual Interrupt Pending (VIP) flag** are used in a multitasking environment

CONTROL REGISTERS The x86 employs four control registers (register CR1 is unused) to control various aspects of processor operation. All of the registers except CR0 are either 32 bits or 64 bits long. The flags are as follows:

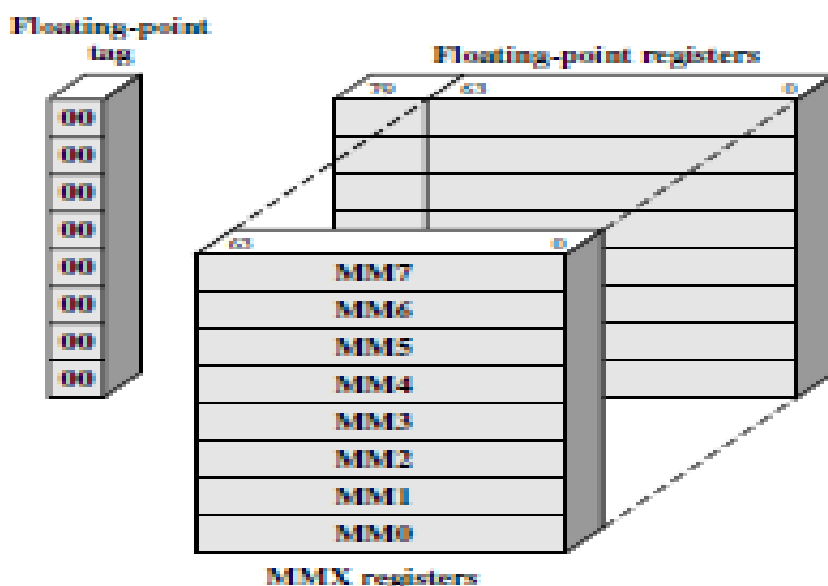
- **Protection Enable (PE):** Enable/disable protected mode of operation.

- **Monitor Coprocessor (MP):** Only of interest when running programs from earlier machines on the x86; it relates to the presence of an arithmetic coprocessor.

- **Emulation (EM):** Set when the processor does not have a floating-point unit, and causes an interrupt when an attempt is made to execute floating-point instructions.
- **Task Switched (TS):** Indicates that the processor has switched tasks.
- **Extension Type (ET):** Not used on the Pentium and later machines; used to indicate support of math coprocessor instructions on earlier machines.
- **Numeric Error (NE):** Enables the standard mechanism for reporting floating-point errors on external bus lines.
- **Write Protect (WP):** When this bit is clear, read-only user-level pages can be written by a supervisor process. This feature is useful for supporting process creation in some operating systems.
- **Alignment Mask (AM):** Enables/disables alignment checking.
- **Not Write Through (NW):** Selects mode of operation of the data cache. When this bit is set, the data cache is inhibited from cache write-through operations.
- **Cache Disable (CD):** Enables/disables the internal cache fill mechanism.
- **Paging (PG):** Enables/disables paging.
 1. When paging is enabled, the CR2 and CR3 registers are valid.
 2. The CR2 register holds the 32-bit linear address of the last page accessed before a page fault interrupt.
 3. The leftmost 20 bits of CR3 hold the 20 most significant bits of the base address of the page directory; the remainder of the address contains zeros. The page-level cache disable (PCD) enables or disables the external cache, and the page-level writes transparent (PWT) bit controls write through in the external cache.
 4. Nine additional control bits are defined in CR4:
 - **Virtual-8086 Mode Extension (VME):** Enables virtual interrupt flag in virtual-8086 mode.
 - **Protected-mode Virtual Interrupts (PVI):** Enables virtual interrupt flag in protected mode.
 - **Time Stamp Disable (TSD):** Disables the read from time stamp counter (RDTSC) instruction, which is used for debugging purposes.
 - **Debugging Extensions (DE):** Enables I/O breakpoints; this allows the processor to interrupt on I/O reads and writes.
 - **Page Size Extensions (PSE):** Enables large page sizes (2 or 4-MByte pages) when set; restricts pages to 4 KBytes when clear.
 - **Physical Address Extension (PAE):** Enables address lines A35 through A32 whenever a special new addressing mode, controlled by the PSE, is enabled.
 - **Machine Check Enable (MCE):** Enables the machine check interrupt, which occurs when a data parity error occurs during a read bus cycle or when a bus cycle is not successfully complete
 - **Page Global Enable (PGE):** Enables the use of global pages. When PGE = 1 and a task switch is performed, all of the TLB entries are flushed with the exception of those marked global.
 - **Performance Counter Enable(PCE):** Enables the Execution of the RDPMC (read performance counter) instruction at any privilege level.

MMX REGISTERS The MMX instructions make use of 3-bit register address fields, so that eight MMX registers are supported. (Figure 3.11). The existing floating-point registers are used to store MMX values. Specifically, the low-order 64 bits (mantissa) of each floating-point register are used to form the eight MMX registers. Some key characteristics of the MMX use of these registers are as follows:

- Recall that the floating-point registers are treated as a stack for floating-point operations. For MMX operations, these same registers are accessed directly.
- The first time that an MMX instruction is executed after any floating-point operations, the FP tag word is marked valid. This reflects the change from stack operation to direct register addressing.
- The EMMS (Empty MMX State) instruction sets bits of the FP tag word to indicate that all registers are empty. It is important that the programmer insert this instruction at the end of an MMX code block so that subsequent floating-point operations function properly.
- When a value is written to an MMX register, bits [79:64] of the corresponding FP register (sign and exponent bits) are set to all ones. This sets the value in the FP register to NaN (not a number) or infinity when viewed as a floating-point value. This ensures that an MMX data value will not look like a valid floating-point value.



3.11 Mapping of MMX Registers to Floating Point Registers

Interrupt Processing

Interrupt processing within a processor is a facility provided to support the operating system. It allows an application program to be suspended, in order that a variety of interrupt conditions can be serviced and later resumed.

INTERRUPTS AND EXCEPTIONS An *interrupt* is generated by a signal from hardware, and it may occur at random times during the execution of a program. An *exception* is generated from software, and it is provoked by the execution of an instruction. There are two sources of interrupts and two sources of exceptions:

1. Interrupts

- **Maskable interrupts:** Received on the processor's INTR pin. The processor does not recognize a maskable interrupt unless the interrupt enable flag (IF) is set.
- **Nonmaskable interrupts:** Received on the processor's NMI pin. Recognition of such interrupts cannot be prevented.

2. Exceptions

- **Processor-detected exceptions:** Results when the processor encounters an error while attempting to execute an instruction.
- **Programmed exceptions:** These are instructions that generate an exception (e.g., INTO, INT3, INT, and BOUND).

INTERRUPT VECTOR TABLE Interrupt processing on the x86 uses the interrupt vector table. Every type of interrupt is assigned a number, and this number is used to index into the interrupt vector table. This table contains 256 32-bit interrupt vectors, which is the address (segment and offset) of the interrupt service routine for that interrupt number.

INTERRUPT HANDLING When an interrupt occurs and is recognized by the processor, a sequence of events takes place:

- 1 If the transfer involves a change of privilege level, then the current stack segment register and the current extended stack pointer (ESP) register are pushed onto the stack.
- 2 The current value of the EFLAGS register is pushed onto the stack.
- 3 Both the interrupt (IF) and trap (TF) flags are cleared. This disables INTR interrupts and the trap or single-step feature.
- 4 The current code segment (CS) pointer and the current instruction pointer (IP or EIP) are pushed onto the stack.
- 5 If the interrupt is accompanied by an error code, then the error code is pushed onto the stack.
- 6 The interrupt vector contents are fetched and loaded into the CS and IP or EIP registers. Execution continues from the interrupt service routine.

❖ **REDUCED INSTRUCTION SET COMPUTERS: Instruction Execution Characteristics, Large Register Files and RISC Architecture**

❖ **Instruction Execution Characteristics**

The *semantic gap* is the difference between the operations provided in HLLs and those provided in computer architecture. . Designers responded with architectures intended to close this gap. Key features include large instruction sets, dozens of addressing modes, and various HLL statements implemented in hardware. Such complex instruction sets are intended to

- Ease the task of the compiler writer.
- Improve execution efficiency, because complex sequences of operations can be implemented in microcode.
- Provide support for even more complex and sophisticated HLLs.

The results of these studies inspired some researchers to look for a different approach: namely, to make the architecture that supports the HLL simpler, rather than more complex To understand the line of reasoning of the RISC advocates, we begin with a brief review of instruction execution characteristics. The aspects of computation of interest are as follows:

- **Operations performed:** These determine the functions to be performed by the processor and its interaction with memory.
- **Operands used:** The types of operands and the frequency of their use determine the memory organization for storing them and the addressing modes for accessing them.
- **Execution sequencing:** This determines the control and pipeline organization.

Implications

A number of groups have looked at results such as those just reported and have concluded that the attempt to make the instruction set architecture close to HLLs is not the most effective design strategy. Rather, the

HLLs can best be supported by optimizing performance of the most time-consuming features of typical HLL programs.

- Generalizing from the work of a number of researchers, three elements emerge that, by and large, characterize RISC architectures.
 - First, use a large number of registers or use a compiler to optimize register usage. This is intended to optimize operand referencing. The studies just discussed show that there are several references per HLL instruction and that there is a high proportion of move (assignment) statements. This suggests that performance can be improved by reducing memory references at the expense of more register references.
 - Second, careful attention needs to be paid to the design of instruction pipelines. Because of the high proportion of conditional branch and procedure call instructions, a straightforward instruction pipeline will be inefficient. This manifests itself as a high proportion of instructions that are prefetched but never executed.
 - Finally, a simplified (reduced) instruction set is indicated. This point is not as obvious as the others, but should become clearer in the ensuing discussion.

❖ Use of Large Register Files:

The reason that register storage is indicated is that it is the fastest available storage device, faster than both main memory and cache. The register file will allow the most frequently accessed operands to be kept in registers and to minimize register-memory operations.

Two basic approaches are possible, one based on software and the other on hardware.

The software approach is to rely on the compiler to maximize register usage. The compiler will attempt to allocate registers to those variables that will be used the most in a given time period. This approach requires the use of sophisticated program-analysis algorithms.

The hardware approach is simply to use more registers so that more variables can be held in registers for longer periods of time. Here we will discuss the hardware approach.

Register Windows

On the face of it, the use of a large set of registers should decrease the need to access memory.

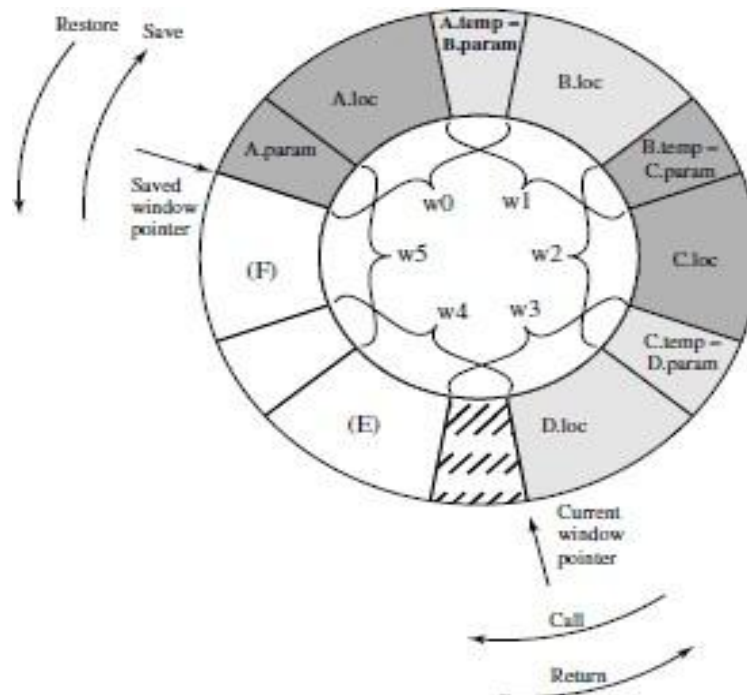
Because most operand references are to local scalars, the obvious approach is to store these in registers, and few registers reserved for global variables. The problem is that the definition of *local* changes with each procedure call and return, operations that occur frequently

The solution is based on two other results reported. First, a typical procedure employs only a few passed parameters and local variables. Second, the depth of procedure activation fluctuates within a relatively narrow range.

The concept is illustrated in Figure 3.12. At any time, only one window of registers is visible and is addressable as if it were the only set of registers (e.g., addresses 0 through N -1). The window is divided into three fixed-size areas. **Parameter registers** hold parameters passed down from the procedure that called the current procedure and hold results to be passed back up. **Local registers** are used for local variables, as assigned by the compiler. **Temporary registers** are used to exchange parameters and results with the next lower level (procedure called by current procedure). The temporary registers at one level are physically the same as the parameter registers at the next lower level. This overlap permits parameters to be passed without the actual movement of data. Keep in mind that, except for the overlap, the registers at two different levels are physically distinct. That is, the parameter and local registers at level J are disjoint from the local and temporary registers at level J + 1.



3.12 Overlapping Register windows



3.13 Circular Buffer Organization of Overlapping Windows

The circular organization is shown in Figure 3.13, which depicts a circular buffer of six windows. The buffer is filled to a depth of 4 (A called B; B called C; C called D) with procedure D active. The current-window pointer (CWP) points to the window of the currently active procedure. Register references by a machine instruction are offset by this pointer to determine the actual physical register.

The saved-window pointer (SWP) identifies the window most recently saved in memory. If procedure D now calls procedure E, arguments for E are placed in D's temporary registers (the overlap between w3 and w4) and the CWP is advanced by one window. If procedure E then makes a call to procedure F, the call cannot be made with the current status of the buffer. This is because F's window overlaps A's window. If F begins to load its temporary registers, preparatory to a call, it will overwrite the parameter registers of A (A.in).

Thus, when CWP is incremented (modulo 6) so that it becomes equal to SWP, an interrupt occurs, and A's window is saved. Only the first two portions (A.in and A.loc) need be saved. Then, the SWP is incremented and the call to F proceeds. A similar interrupt can occur on returns. For example, subsequent to the activation of F, when B returns to A, CWP is decremented and becomes equal to SWP. This causes an interrupt that results in the restoration of A's window.

Global Variables

The window scheme does not address the need to store global variables, those accessed by more than one procedure. Two options suggest themselves.

First, variables declared as global in an HLL can be assigned memory locations by the compiler, and

all machine instructions that reference these variables will use memory-reference operands.

An alternative is to incorporate a set of global registers in the processor. These registers would be fixed in number and available to all procedures. There is an increased hardware burden to accommodate the split in register addressing. In addition, the compiler must decide which global variables should be assigned to registers.

Large Register File versus Cache

The register file, organized into windows, acts as a small, fast buffer for holding a subset of all variables that are likely to be used the most heavily. From this point of view, the register file acts much like a cache memory, although a much faster memory. The question therefore arises as to whether it would be simpler and better to use a cache and a small traditional register file.

Table 3.2 compares characteristics of the two approaches.

Large Register File	Cache
All local scalars	Recently-used local scalars
Individual variables	Blocks of memory
Compiler-assigned global variables	Recently-used global variables
Save/Restore based on procedure nesting depth	Save/Restore based on cache replacement algorithm
Register addressing	Memory addressing

Table 3.2 Characteristics of large register files and cache organization

- The window-based register file holds all the local scalar variables (except in the rare case of window overflow) of the most recent $N - 1$ procedure activations. The cache holds a selection of recently used scalar variables. The register file should save time, because all local scalar variables are retained.

- The cache may make more efficient use of space, because it is reacting to the situation dynamically.

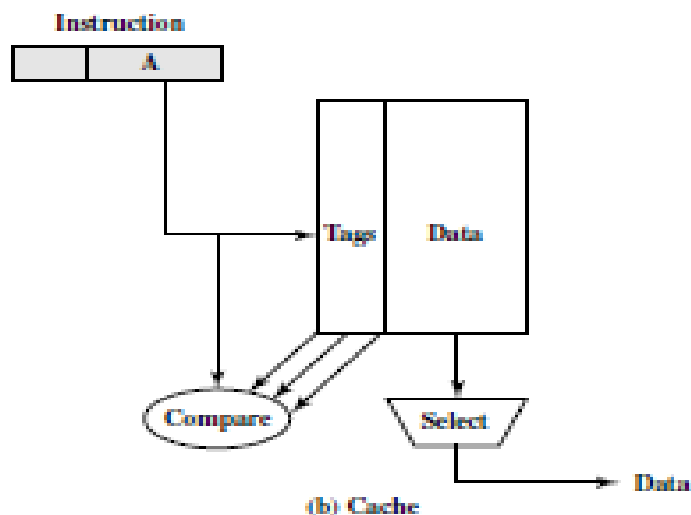
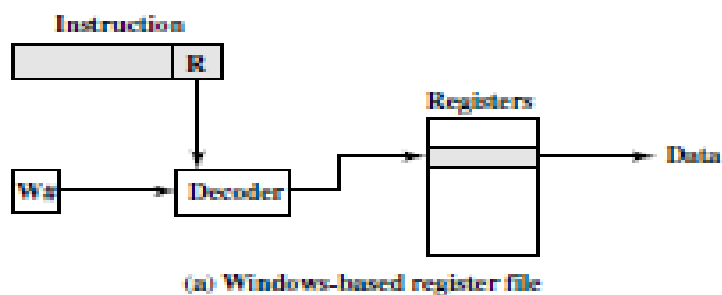
A register file may make inefficient use of space, because not all procedures will need the full window space allotted to them.

- The cache suffers from another sort of inefficiency: Data are read into the cache in blocks. Whereas the register file contains only those variables in use, the cache reads in a block of data, some or much of which will not be used.

- The cache is capable of handling global as well as local variables. There are usually many global scalars, but only a few of them are heavily used. A cache will dynamically discover these variables and hold them. If the window-based register file is supplemented with global registers, it too can hold some global scalars. However, it is difficult for a compiler to determine which globals will be heavily used.

- With the register file, the movement of data between registers and memory is determined by the procedure nesting depth. Most cache memories are set associative with a small set size.

- Figure 3.14 illustrates the difference. To reference a local scalar in a window-based register file, a “virtual” register number and a window number are used. These can pass through a relatively simple decoder to select one of the physical registers. To reference a memory location in cache, a full-width memory address must be generated. The complexity of this operation depends on the addressing mode. In a set associative cache, a portion of the address is used to read a number of words and tags equal to the set size. Another portion of the address is compared with the tags, and one of the words that were read is selected. It should be clear that even if the cache is as fast as the register file, the access time will be considerably longer. Thus, from the point of view of performance, the window-based register file is superior for local scalars. Further performance improvement could be achieved by the addition of a cache for instructions only.



3.14 Referencing a scalar

❖ **Reduced Instruction Set Computer:**

Why CISC

CISC has richer instruction sets, which include a larger number of instructions and more complex instructions. Two principal reasons have motivated this trend: a desire to simplify compilers and a desire to improve performance.

The first of the reasons cited, compiler simplification, seems obvious. The task of the compiler writer is to generate a sequence of machine instructions for each HLL statement. If there are machine instructions that resemble HLL statements, this task is simplified.

This reasoning has been disputed by the RISC researchers. They have found that complex machine instructions are often hard to exploit because the compiler must find those cases that exactly fit the construct. The task of optimizing the generated code to minimize code size, reduce instruction execution count, and enhance pipelining is much more difficult with a complex instruction set.

The other major reason cited is the expectation that a CISC will yield smaller, faster programs. Let us examine both aspects of this assertion: that program will be smaller and that they will execute faster. There are two advantages to smaller programs. First, because the program takes up less memory, there is a savings in that resource. Second, in a paging environment, smaller programs occupy fewer pages, reducing page faults.

The problem with this line of reasoning is that it is far from certain that a CISC program will be smaller than a corresponding RISC program. Thus it is far from clear that a trend to increasingly complex instruction sets is appropriate. This has led a number of groups to pursue the opposite path.

Characteristics of Reduced Instruction Set Architectures

Although a variety of different approaches to reduced instruction set architecture have been taken, certain characteristics are common to all of them:

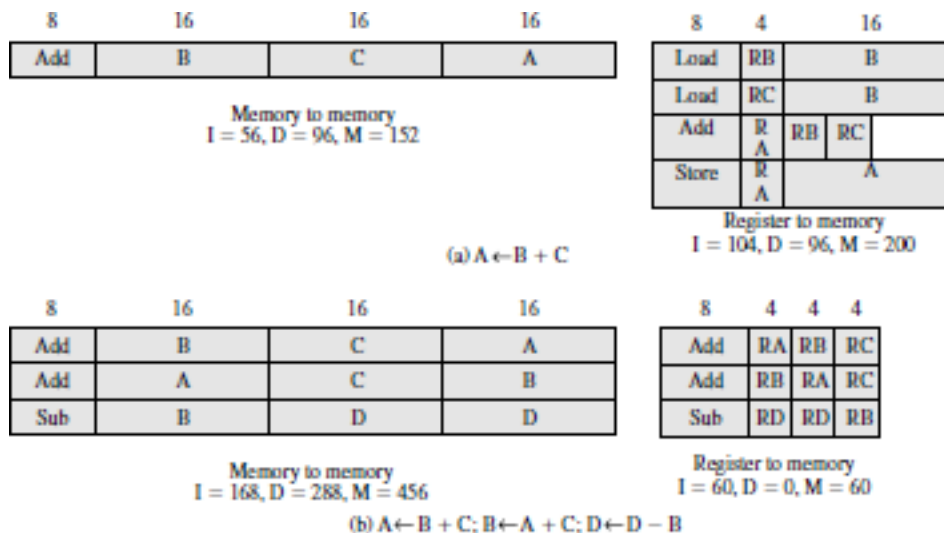
- One instruction per cycle

- Register-to-register operations
- Simple addressing modes
- Simple instruction formats

One machine instruction per machine cycle. A *machine cycle* is defined to be the time it takes to fetch two operands from registers, perform an ALU operation, and store the result in a register. Thus, RISC machine instructions should be no more complicated than, and execute about as fast as, microinstructions on CISC machines. With simple, one-cycle instructions, there is little or no need for microcode; the machine instructions can be hardwired. Such instructions should execute faster than comparable machine instructions on other machines, because it is not necessary to access a microprogram control store during instruction execution.

A second characteristic is that most operations should be **register to register**, with only simple LOAD and STORE operations accessing memory. This design feature simplifies the instruction set and therefore the control unit. For example, a RISC instruction set may include only one or two ADD instructions (e.g., integer add, add with carry); the VAX has 25 different ADD instructions.

This emphasis on register-to-register operations is notable for RISC designs. Contemporary CISC machines provide such instructions but also include memory-to-memory and mixed register/memory operations. Figure 13.5a illustrates the approach taken, Figure 13.5b may be a fairer comparison.



I = number of bytes occupied by executed instructions
 D = number of bytes occupied by data
 M = total memory traffic = I + D

3.15 Two Comparisons of Register-to-Register and Register-to-Memory References

A third characteristic is the use of **simple addressing modes**. Almost all RISC instructions use simple register addressing. Several additional modes, such as displacement and PC-relative, may be included.

A final common characteristic is the use of **simple instruction formats**. Generally, only one or a few formats are used. Instruction length is fixed and aligned on word boundaries

These characteristics can be assessed to determine the potential performance benefits of the RISC approach.

First, more effective optimizing compilers can be developed

A second point, already noted, is that most instructions generated by a compiler are relatively simple anyway. It would seem reasonable that a control unit built specifically for those instructions and using little or no microcode could execute them faster than a comparable CISC.

A third point relates to the use of instruction pipelining. RISC researchers feel that the instruction pipelining technique can be applied much more effectively with a reduced instruction set.

A final, and somewhat less significant, point is that RISC processors are more responsive to interrupts because interrupts are checked between rather elementary operations.

CISC versus RISC Characteristics

After the initial enthusiasm for RISC machines, there has been a growing realization that (1) RISC designs may benefit from the inclusion of some CISC features and that (2) CISC designs may benefit from the inclusion of some RISC features. The result is that the more recent RISC designs, notably the PowerPC, are no longer “pure” RISC and the more recent CISC designs, notably the Pentium II and later Pentium models, do incorporate some RISC characteristics.

Table 3.3 lists a number of processors and compares them across a number of characteristics. For purposes of this comparison, the following are considered typical of a classic RISC:

1. A single instruction size.
2. That size is typically 4 bytes.
3. A small number of data addressing modes, typically less than five. This parameter is difficult to pin down. In the table, register and literal modes are not counted and different formats with different offset sizes are counted separately.
4. No indirect addressing that requires you to make one memory access to get the address of another operand in memory.
5. No operations that combine load/store with arithmetic (e.g., add from memory, add to memory).
6. No more than one memory-addressed operand per instruction.
7. Does not support arbitrary alignment of data for load/store operations.
8. Maximum number of uses of the memory management unit (MMU) for a data address in an instruction.
9. Number of bits for integer register specifier equal to five or more. This means that at least 32 integer registers can be explicitly referenced at a time.
10. Number of bits for floating-point register specifier equal to four or more. This means that at least 16 floating-point registers can be explicitly referenced at a time.

Items 1 through 3 are an indication of instruction decode complexity. Items 4 through 8 suggest the ease or difficulty of pipelining, especially in the presence of virtual memory requirements. Items 9 and 10 are related to the ability to take good advantage of compilers.

In the table, the first eight processors are clearly RISC architectures, the next five are clearly CISC, and the last two are processors often thought of as RISC that in fact have many CISC characteristics.

Processor	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic	Max number of memory operands	Unaligned addressing allowed	Max Number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
AMD29000	1	4	1	no	no	1	no	1	8	3 ^a
MIPS R2000	1	4	1	no	no	1	no	1	5	4
SPARC	1	4	2	no	no	1	no	1	5	4
MC88000	1	4	3	no	no	1	no	1	5	4
HP PA	1	4	10 ^a	no	no	1	no	1	5	4
IBM RT/PC	2 ^a	4	1	no	no	1	no	1	4 ^a	3 ^a
IBM RS/6000	1	4	4	no	no	1	yes	1	5	5
Intel i860	1	4	4	no	no	1	no	1	5	4
IBM 3090	4	8	2 ^b	no ^b	yes	2	yes	4	4	2
Intel 80486	12	12	15	no ^b	yes	2	yes	4	3	3
NSC 32016	21	21	23	yes	yes	2	yes	4	3	3
MC68040	11	22	44	yes	yes	2	yes	8	4	3
VAX	56	56	22	yes	yes	6	yes	24	4	0
Clipper	4 ^a	8 ^a	9 ^a	no	no	1	0	2	4 ^a	3 ^a
Intel 80960	2 ^a	8 ^a	9 ^a	no	no	1	yes ^a	—	5	3 ^a

^aRISC that does not conform to this characteristic.

^bCISC that does not conform to this characteristic.

Table 3.3: Characteristics of some Processors