1. A) Definition of ISA: Instruction set Architecture is the interface between the software and hardware. ISA defines the instructions, data types, registers, addressing modes which are visible to the programmer.

Seven dimensions of ISA

1. Class of ISA
2. Memory Addressing
3. Addressing modes
4. Type and size of operands
5. Operations
6. Control flow instruction
7. Encoding an ISA

Class of ISA

1. There are two important classes of GPR based ISA.
2. Register-memory ISA such as 80*86: Here one input operand (i.e. data or value) is in register and other input operand in memory and after ALU operation result is stored in register.
3. Load-store ISA such as MIPS: Here one input operand is in register and other input operand is in register and result goes to register. To transfer the value to memory we need to use load store instructions.
4. The 80*86 has16 general purpose registers and 16 floating point registers.
5. MIPS has 32 general purpose registers and 32 floating-point register.

Memory Addressing

1. The 80*86 and MIPS use byte addressing for accessing memory operands.
2. Byte addressing refers to architectures where data can be accessed 8 bits (1 Byte) at a time. In some machines; accesses to objects larger than a byte must be aligned.
3. For MIPS the data is aligned and for 8086 data is misaligned.
4. An access to an object of size s bytes at byte address A is aligned if A mod s = 0. Misalignment causes hardware complications.

Addressing modes

1. Addressing modes specify the address of a memory object. An addressing mode specifies how to calculate the effective memory address of an operand by using information held in

registers and/or constants contained within a machine instruction or elsewhere.

2. MIPS addressing modes are Register, Immediate (for constants), and Displacement, where a constant offset is added to a register to form the memory address. The 80x86 supports those three plus three variations of displacement: no register (absolute or direct addressing mode), two registers (based indexed with displacement), two registers where one register is multiplied by the size of the operand in bytes (based scaled index and displacement)

Type and Size of Operands

1. Like most of the ISAs the 8086 and MIPS support operand size of 8-bit (ASCII Character), 16-bit (Unicode character or half word), 32-bit (Integer or word), 64-bit (Long or double word) and floating point in 32-bit (Single precision) and 64-bit (Double precision).
2. The 80x86 also supports 80-bit floating point (extended double precision).

Operations

1. The general categories of operations are data transfer, arithmetic, logical, control, and floating point operations.
2. The instructions for data transfer are LB,LW,SB,SW,SD etc.
3. The instructions for arithmetic operations are DADD,DADDI,DMUL,DDIV etc.
4. The instructions for control flow operations are BEQ,BNEZ etc.

Control Flow Instructions

1. Virtually all ISAs including 8086 and MIPS support conditional branches, unconditional jumps, procedure calls, and returns.
2. There are some small differences. MIPS conditional branches (BE,BNE, etc.) test the contents of registers, while the 80x86 branches (JE,JNE, etc.) test condition code bits set as side effects of arithmetic/logic operations.
3. MIPS procedure call (JAL) places the return address in a register, while the 80x86 call (CALLF) places the return address on a stack in memory.

Encoding an ISA

1. Encoding means converting the instruction into binary form. There are two types of encoding 1) fixed length encoding 2) variable length encoding.
2. The MIPS has fixed length encoding and all instructions are 32-bit long.

3. While 8086 has Variable length encoding and instructions are ranging from 1-18 bytes, so a program compiled for the 80x86 is usually smaller than the same program compiled for MIPS.

1. B) The sum of the failure rates is

$$\text{Failure rate}_{\text{system}} = 10 \times \frac{1}{1{,}000{,}000} + \frac{1}{500{,}000} + \frac{1}{200{,}000} + \frac{1}{200{,}000} + \frac{1}{1{,}000{,}000}$$

$$= \frac{10 + 2 - 5 + 5 + 1}{1{,}000{,}000 \text{ hours}} = \frac{23}{1{,}000{,}000} = \frac{23{,}000}{1{,}000{,}000{,}000 \text{ hours}}$$

or 23,000 FIT. The MTTF for the system is just the inverse of the failure rate:

$$\text{MTTF}_{\text{system}} = \frac{1}{\text{Failure rate}_{\text{system}}} = \frac{1{,}000{,}000{,}000 \text{ hours}}{23{,}000} = 43{,}500 \text{ hours}$$

or just under 5 years.

1. C) CPU time for a program can then be expressed two ways:

CPU time = CPU clock cycles for a program * Clock cycle time

CPU time = CPU clock cycles for a program /Clock rate

If we know the number of clock cycles and the instruction count, we can calculate the average number of *clock cycles per instruction* (CPI). Because it is easier to work with, and because we will deal with simple processors in this chapter, we use CPI. Designers sometimes also use *instructions per clock* (IPC), which is the inverse of CPI.

CPI is computed as
CPI =CPU clock cycles for a program /Instruction count(IC)

By transposing instruction count in the above formula, clock cycles can be defined as IC × CPI. This allows us to use CPI in the execution time formula:
CPU time = Instruction count(IC) × Cycles per instruction × Clock cycle time

$$\text{CPU clock cycles} = \sum_{i=1}^{n} \text{IC}_i \times \text{CPI}_i$$

where IC_i represents number of times instruction i is executed in a program and CPI_i represents the average number of clocks per instruction for instruction i. This form can be used to express CPU time as

$$\text{CPU time} = \left(\sum_{i=1}^{n} IC_i \times CPI_i \right) \times \text{Clock cycle time}$$

and overall CPI as

$$CPI = \frac{\sum_{i=1}^{n} IC_i \times CPI_i}{\text{Instruction count}} = \sum_{i=1}^{n} \frac{IC_i}{\text{Instruction count}} \times CPI_i$$

The latter form of the CPI calculation uses each individual CPI_i and the fraction of occurrences of that instruction in a program (i.e., $IC_i \div$ Instruction count).

2. A) There are three classes of hazards

1) Structural Hazards

2) Data Hazards

3) Control Hazards

Structural Hazards

- Structural Hazards arise due to resource conflicts.
- Structural hazards also arise when some functional unit is not fully pipelined.
- Figure shows a structural hazard between load instruction and instruction 3. Both load instruction and instruction 3 want to access one memory port at the same time and hence there is a resource conflict.

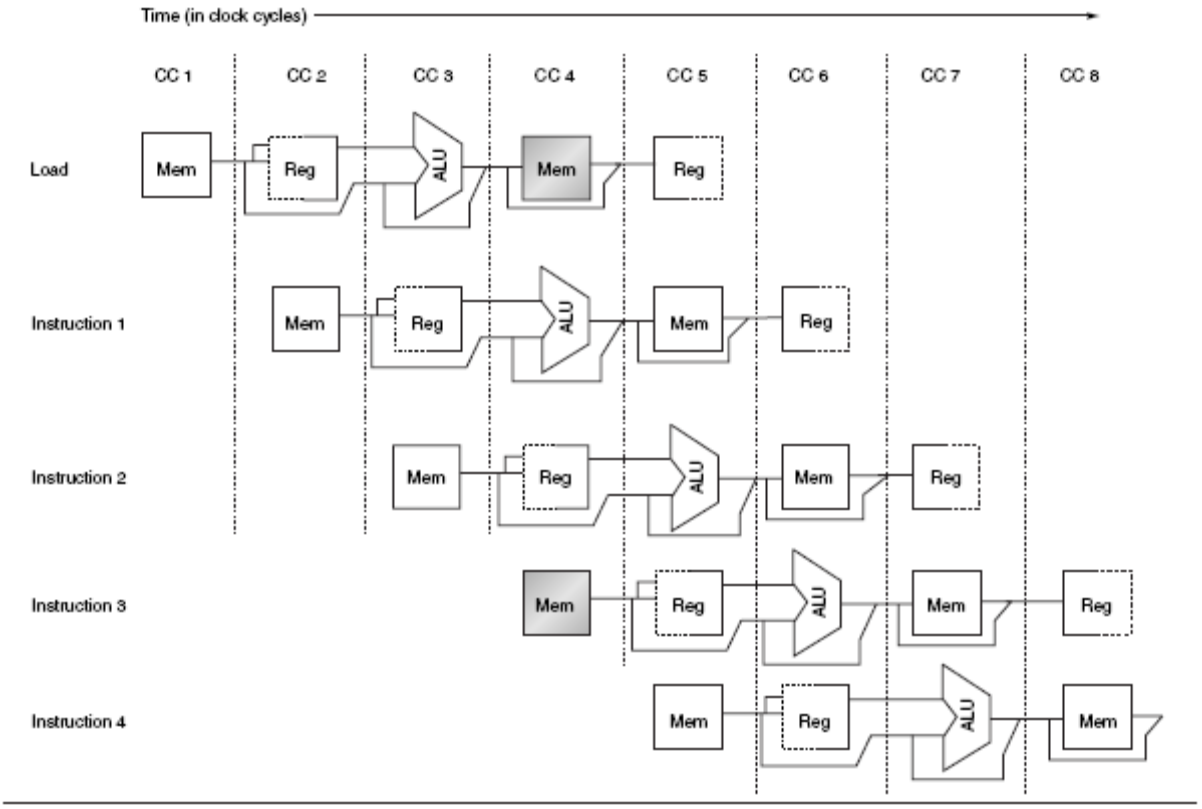


Figure A.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this example the load instruction uses the memory for a data access at the same time instruction 3 wants to fetch an instruction from memory.

Data Hazard

- Data hazard occurs when one instruction is dependent on the result produced by the other instruction.
- Data hazards occur when the pipeline changes the order of read/write accesses to operands so that the order differs from the order seen by sequentially executing instructions on an unpipelined processor. Consider the pipelined execution of these instructions:

  DADD R1,R2,R3
  DSUB R4,R1,R5
  AND R6,R1,R7
  OR R8,R1,R9
  XOR R10,R1,R11

- All the instructions after the DADD use the result of the DADD instruction i.e. R1. As shown in Figure A.6, the DADD instruction writes the value of R1 in the WB pipe stage, but the DSUB instruction reads the value during its ID stage. This problem is called a *data hazard*.

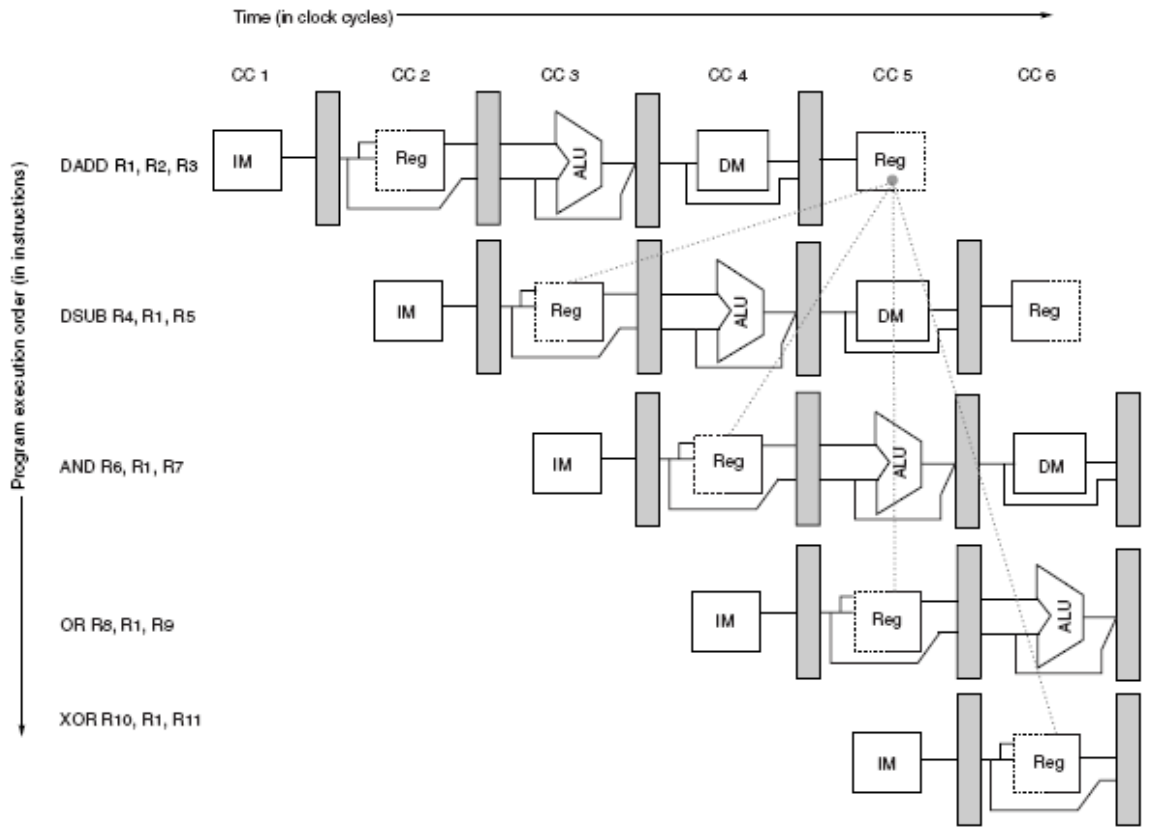


Figure A.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is not written until after those instructions read it.

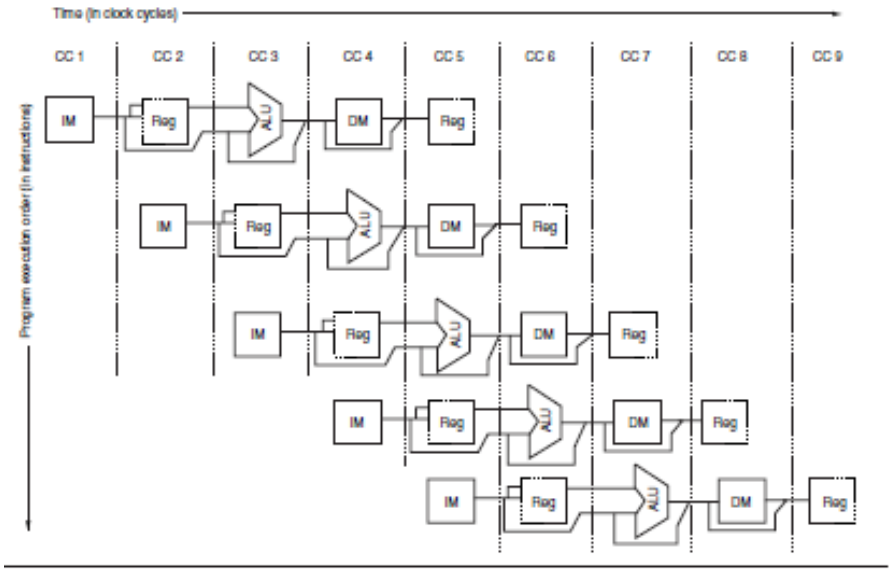
2. B) Classic Five stage pipeline for RISC processor

- Every instruction in the RISC can be implemented in 5 clock cycles. The five clock cycles are IF, ID, EX, MEM and WB.
- Although each instruction will take 5 clock cycles for its execution the hardware will initiate a new instruction in every clock cycle and will be executing some part of different instructions.
- The basic RISC pipeline is shown below. On each clock cycle another instruction is fetched and begins its 5-cycle execution.

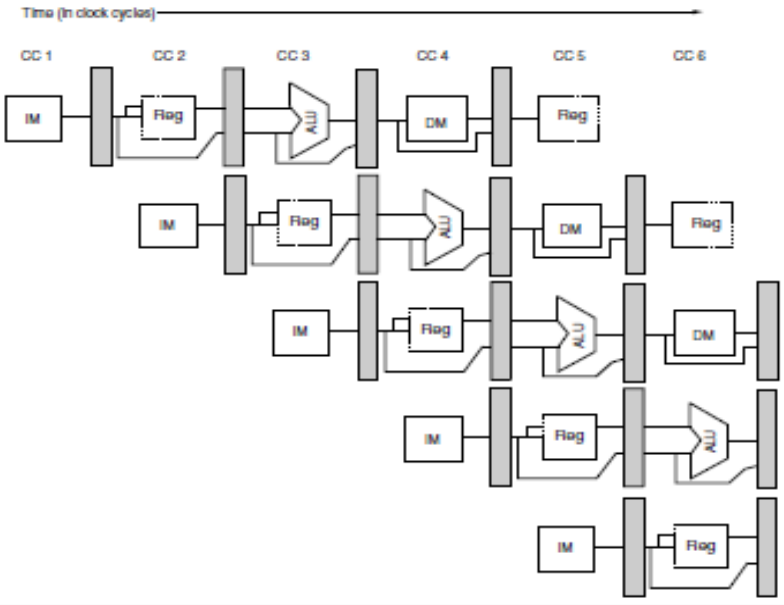
	Clock number								
Instruction number	1	2	3	4	5	6	7	8	9
Instruction *i*	IF	ID	EX	MEM	WB				
Instruction *i* + 1		IF	ID	EX	MEM	WB			
Instruction *i* + 2			IF	ID	EX	MEM	WB		
Instruction *i* + 3				IF	ID	EX	MEM	WB	
Instruction *i* + 4					IF	ID	EX	MEM	WB

Simple RISC pipeline. On each clock cycle, another instruction is fetched and begins its 5-cycle execution. If an instruction is started every clock cycle, the performance will be up to five times that of a processor that is not pipelined. The names for the stages in the pipeline are the same as those used for the cycles in the unpipelined implementation: IF = instruction fetch, ID = instruction decode, EX = execution, MEM = memory access, and WB = write back.

- But the overlapped execution of multiple instructions inside the pipeline should not introduce new type of conflicts.
- Hence to avoid conflicts three methods are used in a pipeline.
- First we separate the instruction and data memory which we would typically implement with separate instruction and data caches. The use of separate caches eliminates a conflict for a single memory that would arise between IF(Instruction Fetch) stage and memory access(MEM) stage.
- Second the register file is used in two stages one for reading in ID and one for writing in WB. Thus to handle reading and writing to same register perform the register write in first half of the clock cycle and register read in the second half of the clock cycle. It is indicated by solid and dashed lines in the figure given below.
- Figure shows the pipeline and abbreviation IM is used for instruction memory and DM for data memory and CC for clock cycle.



- The PC should be updated to start the execution of the new instruction in every clock cycle. PC points to the address of the next instruction to be fetched for execution. But for branches the PC is not changed until ID stage and it causes problem.
- Also pipeline registers are present between the successive stages of the pipeline so that all the results from a given stage are stored into register that is used as the input to the next stage on the next clock cycle.
- For example: The result of the ALU operation is computed during the EX stage, but not actually stored until WB, it arrives there by passing through two pipeline registers. The pipeline registers are named as IF/ID, ID/EX, EX/MEM and MEM/WB.



A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers prevent interference between two different instructions in adjacent stages in the pipeline. The registers also play the critical role of carrying data for a given instruction from one stage to the other. The edge-triggered property of registers—that is, that the values change instantaneously on a clock edge—is critical. Otherwise, the data from one instruction could interfere with the execution of another!

3.A. Loop Unrolling

Increases the amount of ILP and improves the scheduling.

In case of loop unrolling the loop body is replicated multiple times and loop termination code is adjusted at the end. If loop is unrolled for 4 iterations, then the unrolled loop would contain the loop body of 4 iterations and loop termination code at the end.

loop body(1 st iteration)

loop body(2 nd iteration)

loop body(3 rd iteration)

loop body(4 th iteration)

loop termination code.

Hence when the loop is unrolled for four iterations there are four copies of loop body and loop termination code is adjusted at the end as shown below.

Loop	L.D	F0,0(R1)	
	ADD.D	F4,F0,F2	
	S.D	F4,0(R1)	
	L.D	F6,-8(R1)	
	ADD.D	F8,F6,F2	
	S.D	F8,-8(R1)	
	L.D	F10,-16(R1)	
	ADD.D	F12,F10,F2	
	S.D	F12, -16(R1)	
	L.D ADD.D S.D DADDUI BNE	F14,-24(R1) F16,F14,F2 F16,-24(R1) R1,R1,#-32 R1,R2,LOOP	

But in loop unrolling if same set of registers are used it complicates the scheduling process. Hence different set of registers are used for each iteration. Each loop body requires 6 clock cycles and loop termination code requires 3 clock cycles.Total clock cycles= (6*4) + 3 = 27 Thus scheduling unrolled loop significantly reduces the number of clock cycles and increases ILP. Loop unrolling is very useful when loop iterations are independent. Also load and store of different iterations can be easily interchanged if loop iterations are independent.There is shortage of registers due to aggressive loop unrolling and scheduling.

There are three different types of limits to the gains that can be achieved by loop unrolling: a decrease in the amount of overhead amortized with each unroll, code size limitations, and compiler limitations. Let's consider the question ofloop overhead first. When we unrolled the loop four times, it generated sufficient parallelism among the instructions that the loop could be

scheduled with no stall cycles. In fact, in 14 clock cycles, only 2 cycles were loop overhead: the DADDUI, which maintains the index value, and the BNE, which terminates the loop. If the loop is unrolled eight times, the overhead is reduced from 1/2 cycle per original iteration to 1/4.

A second limit to unrolling is the growth in code size that results. For larger loops, the code size growth may be a concern particularly if it causes an increase in the instruction cache miss rate.

Another factor often more important than code size is the potential shortfall in registers that is created by aggressive unrolling and scheduling. This secondary effect that results from instruction scheduling in large code segments is called register pressure. It arises because scheduling code to increase ILP causes the number of live values to increase. After aggressive instruction scheduling, it may not be possible to allocate all the live values to registers. The transformed code, while theoretically faster, may lose some or all of its advantage because it generates a shortage of registers.

3. B Dynamic Branch Prediction

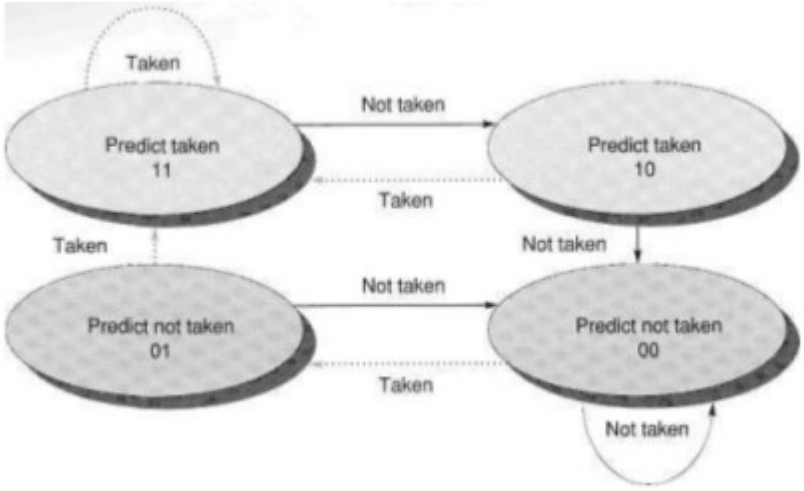
The simplest dynamic branch-prediction scheme is a branch-prediction buffer or branch history table. A branch-prediction buffer is a small memory indexed by the lower portion of the address of the branch instruction. The memory contains a bit that says whether the branch was recently taken or not.

The prediction is a hint that is assumed to be correct, and fetching begins in the predicted direction. If the hint turns out to be wrong, the prediction bit is inverted and stored back.

This simple 1-bit prediction scheme has a performance shortcoming: Even if a branch is almost always taken, we will likely predict incorrectly twice, rather than once, when it is not taken, since the mis-prediction causes the prediction bit to be flipped.

To remedy this weakness, 2-bit prediction schemes are often used. In a 2-bit scheme, a prediction must miss twice before it is changed. Figure shows the finite-state processor for a 2-bit prediction scheme. A branch-prediction buffer can be implemented as a small, special "cache" accessed with the instruction address during the IF pipe stage, or as a pair of bits attached to each block in the instruction cache and fetched with the instruction. If the instruction is decoded as a

branch and if the branch is predicted as taken, fetching begins from the target as soon as the PC is known. Otherwise, sequential fetching and executing continue.



3. C Co-relating Predictor and Tournament Predictor

Branch predictors that use the behavior of other branches to make a prediction are called correlating predictors or two-level predictors. Existing correlating predictors add information about the behavior of the most recent branches to decide how to predict a given branch. For example, a (1,2) predictor uses the behavior of the last branch to choose from among a pair of 2-bit branch predictors in predicting a particular branch. In the general case an (m,n) predictor uses the behavior of the last m branches to choose from 2 m branch predictors, each of which is an n-bit predictor for a single branch. The attraction of this type of correlating branch predictor is that it can yield higher prediction rates than the 2-bit scheme and requires only a trivial amount of additional hardware.

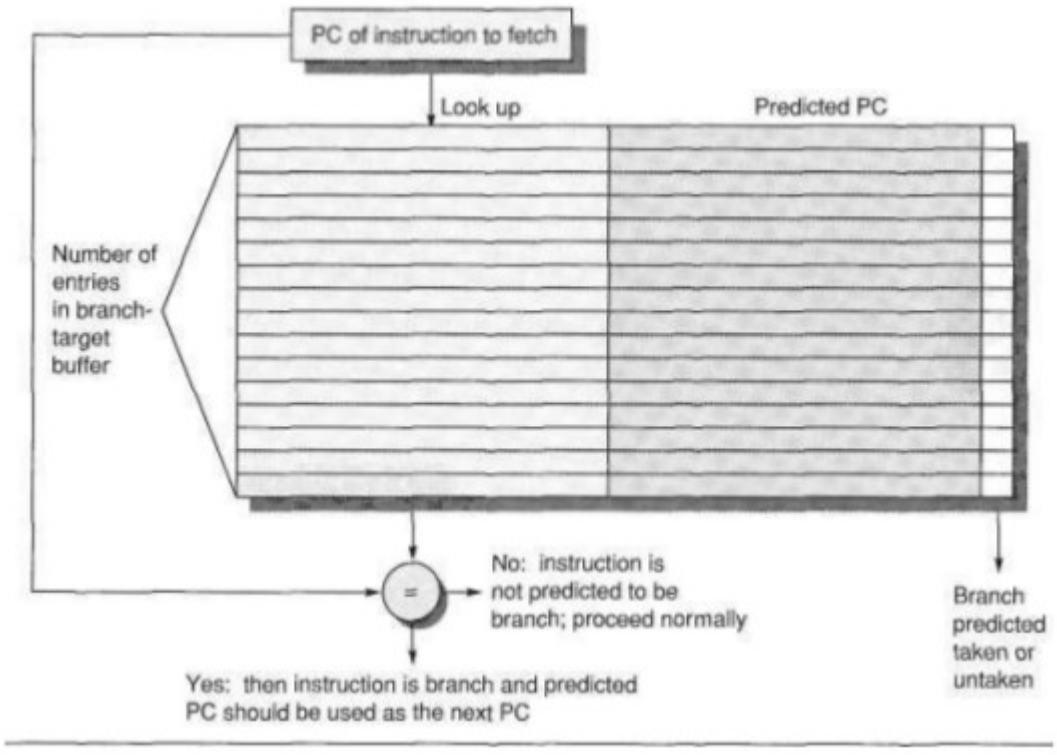
Tournament predictors take this insight to the next level, by using multiple predictors, usually one based on global information and one based on local information, and combining them with a selector. Tournament predictors can achieve both better accuracy at medium sizes (8K-32K bits) and also make use of very large numbers of prediction bits effectively. Existing tournament predictors use a 2-bit saturating counter per branch to choose among two different predictors based on which predictor (local, global, or even some mix) was most effective in recent predictions. As

in a simple 2-bit predictor, the saturating counter requires two mis-predictions before changing the identity of the preferred predictor.

4.A Branch Target Buffer

A branch-prediction cache that stores the predicted address for the next instruction after a branch is called a branch-target buffer or branch-target cache.

Because a branch-target buffer predicts the next instruction address and will send it out before decoding the instruction, we must know whether the fetched instruction is predicted as a taken branch. If the PC of the fetched instruction matches a PC in the prediction buffer, then the corresponding predicted PC is used as the next PC. The hardware for this branch-target buffer is essentially identical to the hardware for a cache.



If a matching entry is found in the branch-target buffer, fetching begins immediately at the predicted PC. Note that unlike a branch-prediction buffer, the predictive entry must be matched to this instruction because the predicted PC will be sent out before it is known whether this instruction is even a branch. If the processor did not check whether the entry matched this PC, then the wrong PC would be sent out for instructions that were not branches, resulting in a slower processor. We only need to store the predicted-taken branches in the branch-target buffer, since an untaken branch should simply fetch the next sequential instruction, as if it were not a branch.

4. B Speculation

One of the significant advantages of speculation is its ability to uncover events that would otherwise stall the pipeline early, such as cache misses. This potential advantage, however, comes with a significant potential disadvantage. Speculation is not free: it takes time and energy, and the recovery of incorrect speculation further reduces performance. In addition, to support the higher instruction execution rate needed to benefit from speculation, the processor must have additional resources, which take silicon area and power. Finally, if speculation causes an exceptional event to occur, such as a cache or TLB miss, the potential for significant performance loss increases, if that event would not have occurred without speculation. To maintain most of the advantage, while minimizing the disadvantages, most pipelines with

speculation will allow only low-cost exceptional events (such as a first-level cache miss) to be handled in speculative mode. If an expensive exceptional event occurs, such as a second-level cache miss or a translation lookaside buffer (TLB) miss, the processor will wait until the instruction causing the event is no longer speculative before handling the event. Although this may slightly degrade the performance of some programs, it avoids significant performance losses in others, especially those that suffer from a high frequency of such events coupled with less-than-excellent branch prediction.

4. C Value Prediction

One technique for increasing the amount of ILP available in a program is value prediction. Value prediction attempts to predict the value that will be produced by an instruction. Obviously, since most instructions produce a different value every time they are executed (or at least a different value from a set of values), value prediction can have only limited success. There are, however, certain instructions for which it is easier to predict the resulting value—for example, loads that load from a constant pool, or that load a value that changes infrequently. In addition, when an instruction produces a value chosen from a small set of potential values, it may be possible to predict the resulting value by correlating it without an instance.

Value prediction is useful if it significantly increases the amount of available ILP. This possibility is most likely when a value is used as the source of a chain of dependent computations, such as a load. Because value prediction is used to enhance speculations and incorrect speculation has detrimental performance impact, the accuracy of the prediction is critical.

5. A Amdahl's Law is

$$\text{Speedup} = \frac{1}{\frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}} + (1 - \text{Fraction}_{\text{enhanced}})}$$

$$80 = \frac{1}{\frac{\text{Fraction}_{\text{parallel}}}{100} + (1 - \text{Fraction}_{\text{parallel}})}$$

Simplifying this equation yields

$$0.8 \times \text{Fraction}_{\text{parallel}} + 80 \times (1 - \text{Fraction}_{\text{parallel}}) = 1$$

$$80 - 79.2 \times \text{Fraction}_{\text{parallel}} = 1$$

$$\text{Fraction}_{\text{parallel}} = \frac{80 - 1}{79.2}$$

$$\text{Fraction}_{\text{parallel}} = 0.9975$$

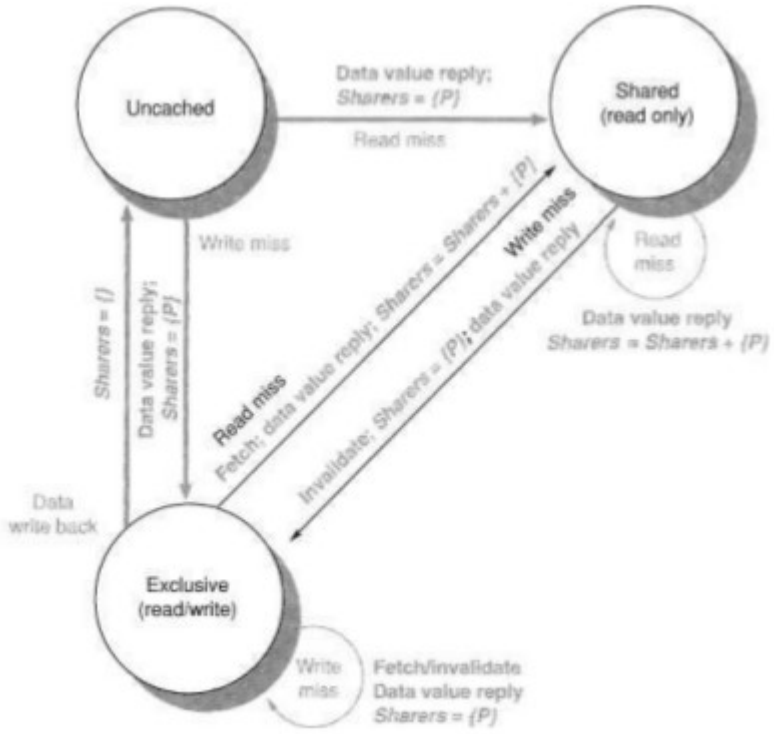
Thus, to achieve a speedup of 80 with 100 processors, only 0.25% of original computation can be sequential.

5. B The states could be the following:

Shared—One or more processors have the block cached, and the value in memory is up to date (as well as in all the caches).

Uncached—No processor has a copy of the cache block.

Modified—Exactly one processor has a copy of the cache block, and it has written the block, so the memory copy is out of date. The processor is called the owner of the block.



When a block is in the uncached state, the copy in memory is the current value, so the only possible requests for that block are

• Read miss—The requesting processor is sent the requested data from memory, and the requestor is made the only sharing node. The state of the block is made shared.

• Write miss—The requesting processor is sent the value and becomes the sharing node. The block is made exclusive to indicate that the only valid copy is cached. Sharers indicates the identity of the owner.

When the block is in the shared state, the memory value is up to date, so the same two requests can occur:

• Read miss—The requesting processor is sent the requested data from memory, and the requesting processor is added to the sharing set.

• Write miss—The requesting processor is sent the value. All processors in the set Sharers are sent invalidate messages, and the Sharers set is to contain the identity of the requesting processor. The state of the block is made exclusive.

When the block is in the exclusive state, the current value of the block is held in the cache of the processor identified by the set Sharers (the owner), so there are three possible directory requests:
• Read miss—The owner processor is sent a data fetch message, which causes the state of the block in the owner's cache to transition to shared and causes the owner to send the data to the directory, where it is written to memory and sent back to the requesting processor. The identity of the requesting processor is added to the set Sharers, which still contains the identity of the processor that was the owner (since it still has a readable copy).
• Data write back—The owner processor is replacing the block and therefore must write it back. This write back makes the memory copy up to date (the home directory essentially becomes the owner), the block is now uncached, and the Sharers set is empty.
• Write miss—The block has a new owner. A message is sent to the old owner, causing the cache to invalidate the block and send the value to the directory, from which it is sent to the requesting processor, which becomes the new owner. Sharers is set to the identity of the new owner, and the state of the block remains exclusive.

6. A Six basic Cache optimization techniques

First Optimization: Larger Block Size to Reduce Miss Rate

The simplest way to reduce miss rate is to increase the block size. Larger block sizes will reduce also compulsory misses.

Larger blocks take advantage of spatial locality.

Second Optimization: Larger Caches to Reduce Miss Rate

The obvious drawback is potentially longer hit time and higher cost and power. This technique has been especially popular in off-chip caches.

Third Optimization: Higher Associativity to Reduce Miss Rate

Increasing block size reduces miss rate while increasing miss penalty, and greater associativity can come at the cost of increased hit time.

The second observation, called the 2:1 cache rule of thumb, is that a direct- mapped cache of size JV has about the same miss rate as a two-way set-associative cache of size N/2.

Fourth Optimization: Multilevel Caches to Reduce Miss Penalty

The first-level cache can be small enough to match the clock cycle time of the fast processor. Yet the second-level cache can be large enough to capture many accesses that would go to main memory, thereby lessening the effective miss penalty.

Using the subscripts LI and L2 to

refer, respectively, to a first-level and a second-level cache, the original formula is

Average memory access time = Hit time L i + Miss rate L1 x Miss penalty L1

and

Miss penalty L1 = Hit time L2 + Miss rate L2 x Miss penalty L2

so

Average memory access time = Hit time L1 + Miss rate L1

x (Hit time L2 + Miss rate L2 x Miss penalty L2)

Fifth Optimization: Giving Priority to Read Misses overwrites to Reduce Miss Penalty

The simplest way out of this dilemma is for the read miss to wait until the write buffer is empty. The alternative is to check the contents of the write buffer on a read miss, and if there are no conflicts and the memory system is available, let the read miss continue. Virtually all desktop and server processors use the latter approach, giving reads priority over writes.

Sixth Optimization: Avoiding Address Translation during

Indexing of the Cache to Reduce Hit Time

Full virtual addressing for both indices and tags eliminates address translation time from a cache hit.

One alternative to get the best of both virtual and physical caches is to use part of the page offset —the part that is identical in both virtual and physical addresses—to index the cache. At the same time as the cache is being read using that index, the virtual part of the address is translated, and the tag match uses physical addresses.

6. B Three types of cache miss

Compulsory—The very first access to a block cannot be in the cache, so the block must be brought into the cache. These are also called cold-start misses or first-reference misses.

• Capacity—If the cache cannot contain all the blocks needed during execution of a program, capacity misses (in addition to compulsory misses) will occur because of blocks being discarded and later retrieved.
• Conflict—If the block placement strategy is set associative or direct mapped, conflict misses (in addition to compulsory and capacity misses) will occur because a block may be discarded and later retrieved if too many blocks map to its set. These misses are also called collision misses. The idea is that hits in a fully associative cache that become misses in an w-way set-associative cache are due to more than n requests on some popular sets.

6. **C Four memory hierarchy questions for Virtual Memory**

Q1: Where Can a Block Be Placed in Main Memory?
Operating systems allow blocks to be placed anywhere in main memory.
Q2: How Is a Block Found If It Is in Main Memory?
Both paging and segmentation rely on a data structure that is indexed by the page or segment number. This data structure contains the physical address of the block. For segmentation, the offset is added to the segment's physical address to obtain the final physical address. For paging, the offset is simply concatenated to this physical page address.
This data structure, containing the physical page addresses, usually takes the
form of a page table. Indexed by the virtual page number, the size of the table is
the number of pages in the virtual address space. Given a 32-bit virtual address,
4 KB pages, and 4 bytes per Page Table Entry (PTE), the size of the page table
would be (2 32 /2 12) X 2 2 = 2 22 or 4 MB.
Q3: Which Block Should Be Replaced on a Virtual Memory Miss?
All operating systems try to replace the least-recently used (LRU) block because if the past predicts the future, that is the one less likely to be needed.
To help the operating system estimate LRU, many processors provide a use bit or reference bit, which is logically set whenever a page is accessed.
Q4: What Happens on a Write?
The level below main memory contains rotating magnetic disks that take millions of clock cycles to access. Because of the great discrepancy in access time, no one has yet built a virtual memory

operating system that writes through main memory to disk on every store by the processor. (This remark should not be interpreted as an opportunity to become famous by being the first to build one!) Thus, the write strategy is always write back.

7. A Eleven Advanced Cache optimization

First Optimization : Small and Simple Caches

Small cache can help hit time since smaller memory takes less time to index

Simple _ direct mapping and can overlap tag check with data transmission since no choice

Second Optimization: Way Prediction

Way prediction: keep extra bits in cache to predict the “way,” or block within the set, of next cache access. Multiplexer is set early to select desired block, only 1 tag comparison performed that clock cycle in parallel with reading the cache data

Third optimization: Trace Cache

Dynamic traces of the executed instructions vs. static sequences of instructions as determined by layout in memory. Cache the micro-ops vs. x86 instructions

Fourth optimization: pipelined cache access to increase bandwidth

Pipeline cache access to maintain bandwidth, but higher latency

• Instruction cache access pipeline stages:

1: Pentium

2: Pentium Pro through Pentium III

4: Pentium 4

- _ greater penalty on mis-predicted branches
- _ more clock cycles between the issue of the load and the use of the data

Fifth optimization: Increasing Cache Bandwidth Non-Blocking Caches

Non-blocking cache or lockup-free cache allow data cache to continue to supply

cache hits during a miss. “hit under miss” reduces the effective miss penalty by working during miss vs. ignoring CPU requests

“hit under multiple miss” or “miss under miss” may further lower the effective

miss penalty by overlapping multiple misses

Sixth optimization: Increasing Cache Bandwidth via Multiple Banks

Rather than treat the cache as a single monolithic block, divide into independent banks that can support simultaneous accesses. Simple mapping that works well is “sequential interleaving”

Seventh optimization :Reduce Miss Penalty: Early Restart and Critical Word First

Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution

Critical Word First—Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block

Eight optimization: Merging Write Buffer to Reduce Miss Penalty

Write buffer to allow processor to continue while waiting to write to memory

• If buffer contains modified blocks, the addresses can be checked to see if address of new data matches the address of a valid write buffer entry

• If so, new data are combined with that entry

• Increases block size of write for write-through cache of writes to sequential words, bytes since multiword writes more efficient to memory

• The Sun T1 (Niagara) processor, among many others, uses write merging

Ninth optimization: Reducing Misses by Compiler Optimizations

Instructions

Reorder procedures in memory so as to reduce conflict misses

Profiling to look at conflicts (using tools they developed)

Data

Merging Arrays: improve spatial locality by single array of compound elements vs. 2 arrays

Loop Interchange: change nesting of loops to access data in order stored in memory

Loop Fusion: Combine 2 independent loops that have same looping and some variables overlap

Blocking: Improve temporal locality by accessing “blocks” of data repeatedly vs. going down whole columns or rows.

Tenth optimization Reducing Misses by Hardware Prefetching of Instructions & Data

• Instruction Prefetching

Typically, CPU fetches 2 blocks on a miss: the requested block and the next consecutive block.

Requested block is placed in instruction cache when it returns, and prefetched block is placed into instruction stream buffer

Data Prefetching

Pentium 4 can prefetch data into L2 cache from up to 8 streams from 8 different 4 KB pages

Prefetching invoked if 2 successive L2 cache misses to a page, if distance between those cache blocks is < 256 bytes

7. B. Protection via Virtual memory and machines

Virtual memory

Processes can be protected from one another by having their own page tables, each pointing to distinct pages of memory.

Rings added to the processor protection structure expand memory access protection from two levels (user and kernel) to many more.

Protect Processes from each other. Page based virtual memory including TLB which caches page table entries –Example: Segmentation and paging in 80x86

Processes share hardware without interfering with each other.Provide User Process and Kernel Process

System call to transfer to supervisor mode. Return like normal subroutine to user mode

Mechanism to limit memory access

Memory protection

•Virtual Memory

–Restriction on each page entry in page table

–Read, write, execute privileges

–Only OS can update page table

–TLB entries also have protection field

•Bugs in OS

–Lead to compromising security

–Bugs likely due to huge size of OS code