PIPELINING:

PIPELINING:

Introduction Pipeline

hazards

Implementation of pipeline

What makes pipelining hard to implement?

Pipelining: Basic and Intermediate concepts

Pipeline is an implementation technique that exploits parallelism among the instructions in a sequential instruction stream. Pipeline allows to overlapping the execution of multiple instructions. A Pipeline is like an assembly line each step or pipeline stage completes a part of an instructions. Each stage of the pipeline will be operating an a separate instruction. Instructions enter at one end progress through the stage and exit at the other end. If the stages are perfectly balance.
(assuming ideal conditions), then the time per instruction on the pipeline processor is given by the ratio:

Time per instruction on unpipelined machine/ Number of Pipeline stages

Under these conditions, the speedup from pipelining is equal to the number of stage pipeline. In practice, the pipeline stages are not perfectly balanced and pipeline does involve some overhead. Therefore, the speedup will be always then practically less than the number of stages of the pipeline. Pipeline yields a reduction in the average execution time per instruction. If the processor is assumed to take one (long) clock cycle per instruction, then pipelining decrease the clock cycle time. If the processor is assumed to take multiple CPI, then pipelining will aid to reduce the CPI.

A Simple implementation of a RISC instruction set
Instruction set of implementation in RISC takes at most 5 cycles without pipelining. The 5 clock cycles are:

1. Instruction fetch (IF) cycle:

Send the content of program count (PC) to memory and fetch the current instruction from memory to update the PC.

New PC ← [PC] + 4; Since each instruction is 4 bytes

2. Instruction decode / Register fetch cycle (ID):

Decode the instruction and access the register file. Decoding is done in parallel with reading registers, which is possible because the register specifies are at a fixed location in a RISC architecture. This corresponds to fixed field decoding. In addition it involves:

- Perform equality test on the register as they are read for a possible branch.
- Sign-extend the offset field of the instruction in case it is needed.
- Compute the possible branch target address.

3. Execution / Effective address Cycle (EXE)

The ALU operates on the operands prepared in the previous cycle and performs one of the following function defending on the instruction type.

* Memory reference: Effective address ← [Base Register] + offset

* Register- Register ALU instruction: ALU performs the operation specified in the instruction using the values read from the register file.
* Register- Immediate ALU instruction: ALU performs the operation specified in the instruction using the first value read from the register file and that sign extended immediate.

4. Memory access (MEM)

For a load instruction, using effective address the memory is read. For a store instruction memory writes the data from the 2nd register read using effective address.

5. Write back cycle (WB)

Write the result in to the register file, whether it comes from memory system (for a LOAD instruction) or from the ALU.

Five stage Pipeline for a RISC processor

Each instruction taken at most 5 clock cycles for the execution

* Instruction fetch cycle (IF)
* Instruction decode / register fetch cycle (ID)
* Execution / Effective address cycle (EX)
* Memory access (MEM)
* Write back cycle (WB)

The execution of the instruction comprising of the above subtask can be pipelined. Each of the clock cycles from the previous section becomes a pipe stage – a cycle in the pipeline. A new instruction can be started on each clock cycle which results in the execution pattern shown figure 2.1. Though each instruction takes 5 clock cycles to complete, during each clock cycle the hardware will initiate a new instruction and will be executing some part of the five different instructions as illustrated in figure 2.1.

Instruction #	Clock number								
	1	2	3	4	5	6	7	8	9
Instruction i	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				IF	ID	EXE	MEM	WB	
Instruction I+4					IF	ID	EXE	MEM	WB

Figure 2.1 Simple RISC Pipeline. On each clock cycle another instruction fetched

Each stage of the pipeline must be independent of the other stages. Also, two different operations can't be performed with the same data path resource on the same clock. For example, a single ALU cannot be used to compute the effective address and perform a subtract operation during the same clock cycle. An adder is to be provided in the stage 1 to compute new PC value and an ALU in the stage 3 to perform the arithmetic indicatedin the instruction (See figure 2.2). Conflict should not arise out of overlap of instructions using pipeline. In other words, functional unit of each stage need to be independent of other functional unit. There are three observations due to which the risk of conflict is reduced.

- Separate Instruction and data memories at the level of L1 cache eliminates a conflict for a single memory that would arise between instruction fetch and data access.
- Register file is accessed during two stages namely ID stage WB. Hardware should allow to perform maximum two reads one write every clock cycle.
- To start a new instruction every cycle, it is necessary to increment and store the PC every cycle.

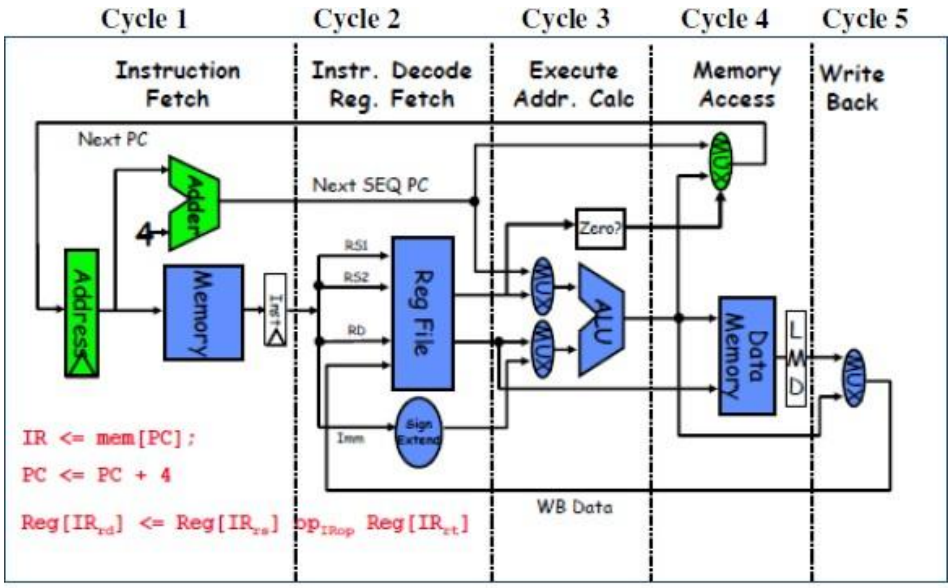


Figure 2.2 Diagram indicating the cycle and functional unit of each stage.

Buffers or registers are introduced between successive stages of the pipeline so that at the end of a clock cycle the results from one stage are stored into a register (see figure 2.3). During the next clock cycle, the next stage will use the content of these buffers as input. Figure 2.4 visualizes the pipeline activity.

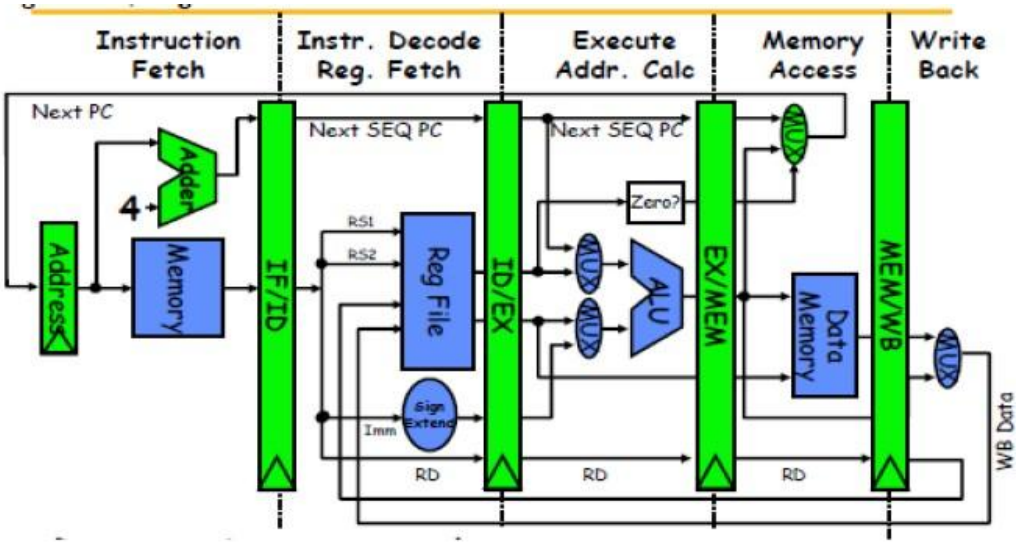


Figure 2.3 Functional units of 5 stage Pipeline. IF/ID is a buffer between IF and ID stage.

Basic Performance issues in Pipelining

Pipelining increases the CPU instruction throughput but, it does not reduce the executiontime of an individual instruction. In fact, the pipelining increases the execution time of each instruction due to overhead in the control of the pipeline. Pipeline overhead arises from the combination of register delays and clock skew. Imbalance among the pipe stages reduces the performance since the clock can run no faster than the time needed for the slowest pipeline stage.

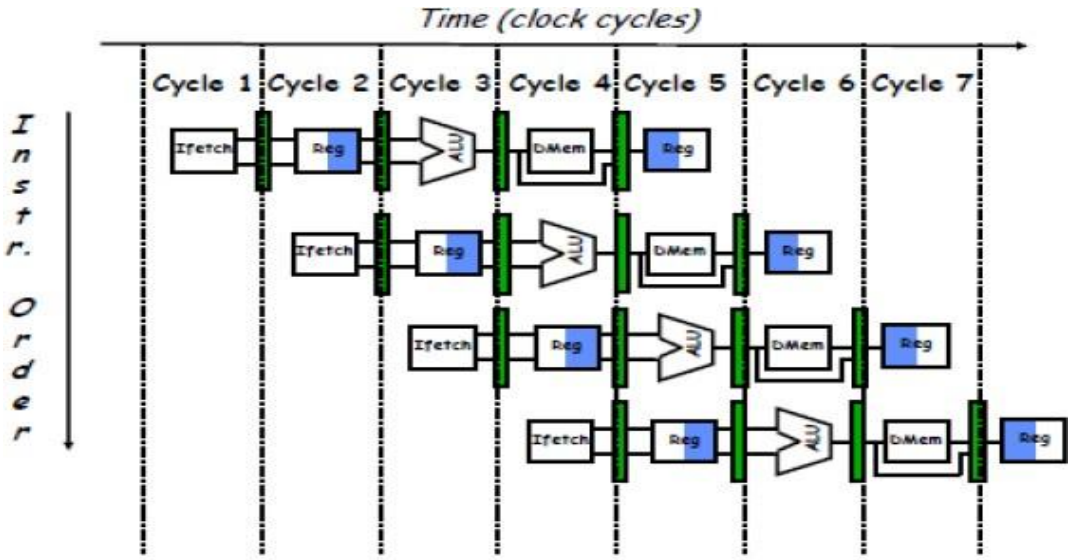


Figure 2.4 Pipeline activity

Pipeline Hazards

Hazards may cause the pipeline to stall. When an instruction is stalled, all the instructions issued later than the stalled instructions are also stalled. Instructions issued earlier than the stalled instructions will continue in a normal way. No new instructions are fetched during the stall. Hazard is situation that prevents the next instruction in the instruction stream fromk executing during its designated clock cycle. Hazards will reduce the pipeline performance.

Performance with Pipeline stall

A stall causes the pipeline performance to degrade from ideal performance. Performance improvement from pipelining is obtained from:

$$\textbf{Speedup} = \frac{\text{Average instruction time un-pipelined}}{\text{Average instruction time pipelined}}$$

$$\textbf{Speedup} = \frac{\text{CPI unpipelined} * \text{Clock cycle unpipelined}}{\text{CPI pipelined} * \text{Clock cycle pipelined}}$$

$$\textbf{CPI pipelined} = \text{Ideal CPI} + \text{Pipeline stall clock cycles per instruction}$$

$$\textbf{CPI pipelined} = 1 + \text{Pipeline stall clock cycles per instruction}$$

Assume that,
i) cycle time overhead of pipeline is ignored
ii) stages are balanced With
theses assumptions

$$\text{Clock cycle unpipelined} = \text{clock cycle pipelined}$$
$$\text{Therefore, Speedup} = \frac{\text{CPI unpipelined}}{\text{CPI pipelined}}$$

$$\text{Speedup} = \frac{\text{CPI unpipelined}}{1+\text{Pipeline stall cycles per instruction}}$$

If all the instructions take the same number of cycles and is equal to the number of pipeline stages or depth of the pipeline, then,

$$\text{CPI unpipelined} = \text{Pipeline depth}$$

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1+\text{Pipeline stall cycles per instruction}}$$

If there are no pipeline stalls,
Pipeline stall cycles per instruction = zero
Therefore,
Speedup = Depth of the pipeline.

Types of hazard

Three types hazards are:

1. Structural hazard
2. Data Hazard
3. Control Hazard

Structural hazard

Structural hazard arise from resource conflicts, when the hardware cannot support all possible combination of instructions simultaneously in overlapped execution. If some combination of instructions cannot be accommodated because of resource conflicts, the processor is said to have structural hazard. Structural hazard will arise when some functional unit is not fully pipelined or when some resource has not been duplicated enough to allow all combination of instructions in the pipeline to execute. For example, if memory is shared for data and instruction as a result, when an instruction contains data memory reference, it will conflict with the instruction reference for a later instruction (as shown in figure 2.5a). This will cause hazard and pipeline stalls for 1 clock cycle.

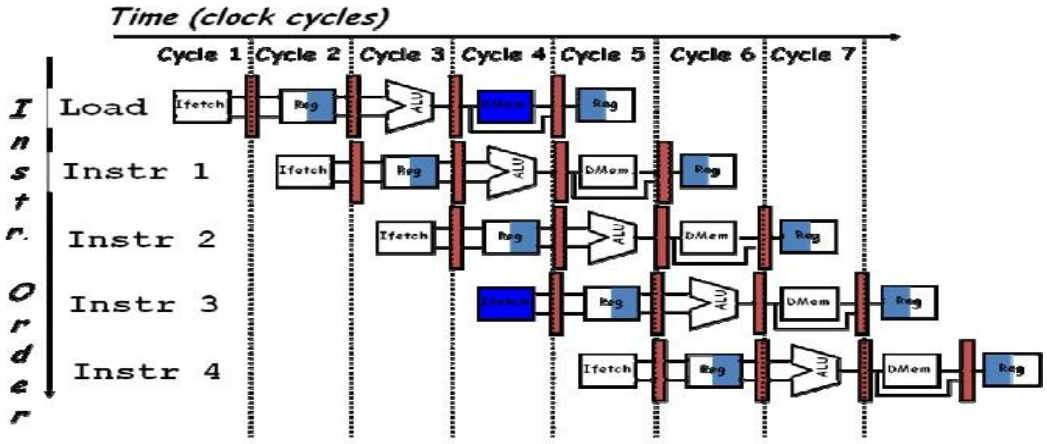
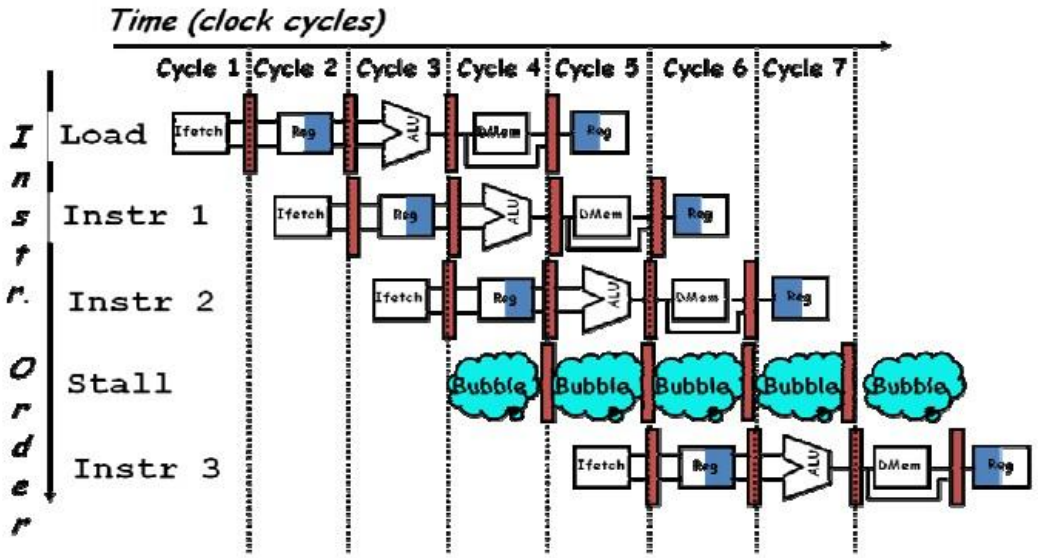


Figure 2.5a Load Instruction and instruction 3 are accessing memory in clock cycle4



Instruction #	Clock number								
	1	2	3	4	5	6	7	8	9
Load Instruction	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				Stall	IF	ID	EXE	MEM	WB
Instruction I+4						IF	ID	EXE	MEM

Figure 2.5b A Bubble is inserted in clock cycle 4

Pipeline stall is commonly called Pipeline bubble or just simply bubble

Data Hazard

Consider the pipelined execution of the following instruction sequence (Timing diagram shown in figure 2.6)

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

DADD instruction produces the value of R1 in WB stage (Clock cycle 5) but the DSUB instruction reads the value during its ID stage (clock cycle 3). This problem is called Data Hazard. DSUB may read the wrong value if precautions are not taken. AND instruction will read the register during clock cycle 4 and will receive the wrong results. The XOR instruction operates properly, because its register read occurs in clock cycle 6 after DADD writes in clock cycle 5. The OR instruction also operates without incurring a hazard because the register file reads are performed in the second half of the cycle whereas the writes are performed in the first half of the cycle.

Minimizing data hazard by Forwarding

The DADD instruction will produce the value of R! at the end of clock cycle 3. DSUB instruction requires this value only during the clock cycle 4. If the result can be moved from the pipeline register where the DADD store it to the point (input of LAU) where DSUB needs it, then the need for a stall can be avoided. Using a simple hardware technique called Data Forwarding or Bypassing or short circuiting, data can be made available from the output of the ALU to the point where it is required (input of LAU) at the beginning of immediate next clock cycle.

Forwarding works as follows:

i) The output of ALU from EX/MEM and MEM/WB pipeline register is always feedback to the ALU inputs.

ii) If the Forwarding hardware detects that the previous ALU output serves as the source for the current ALU operations, control logic selects the forwarded result

as the input rather than the value read from the register file. Forwarded results are required not only from the immediate previous instruction, but also from an instruction that started 2 cycles earlier. The result of ith instruction Is required to be forwarded to (i+2)th instruction also. Forwarding can be generalized to include passing a result directly to the functional unit that requires it.

Data Hazard requiring stalls

```
LD      R1, 0(R2)
DADD R3, R1, R4
AND     R5, R1, R6
OR      R7, R1, R8
```

The pipelined data path for these instructions is shown in the timing diagram (figure 2.7)

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
LD R1, 0(R2)	IF	ID	EXE	MEM	WB				
DADD R3,R1,R4		IF	ID	EXE	MEM	WB			
AND R5, R1, R6			IF	ID	EXE	MEM	WB		
OR R7, R1, R8				IF	ID	EXE	MEM	WB	
LD R1, 0(R2)	IF	ID	EXE	MEM	WB				
DADD R3,R1,R4		IF	ID	Stall	EXE	MEM	WB		
AND R5, R1, R6			IF	Stall	ID	EXE	MEM	WB	
OR R7, R1, R8				Stall	IF	ID	EXE	MEM	WB

Figure 2.7 In the top half, we can see why stall is needed. In the second half, stall created to solve the problem.

The LD instruction gets the data from the memory at the end of cycle 4. even with forwarding technique, the data from LD instruction can be made available earliest during clock cycle 5. DADD instruction requires the result of LD instruction at the beginning of clock cycle 5. DADD instruction requires the result of LD instruction at the beginning of clock cycle 4. This demands data forwarding of clock cycle 4. This demands data forwarding in negative time which is not possible. Hence, the situation calls for a pipeline stall.Result from the LD instruction can be forwarded from the pipeline register to the and instruction which begins at 2 clock cycles later after the LD instruction. The load instruction has a delay or latency that cannot be eliminated by forwarding alone. It is necessary to stall pipeline by 1 clock cycle. A hardware called Pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared. The pipeline interlock helps to preserve the correct execution pattern by introducing a stall or bubble. The CPI for the stalled instruction increases by the length of the stall. Figure 2.7 shows the pipeline before and after the stall. Stall causes the DADD to move 1 clock cycle later in time. Forwarding to the AND instruction now goes through the register file or forwarding is not required for the OR instruction. No instruction is started during the clock cycle 4.

Control Hazard

When a branch is executed, it may or may not change the content of PC. If a branch is taken, the content of PC is changed to target address. If a branch is taken, the content of PC is not changed

The simple way of dealing with the branches is to redo the fetch of the instruction following a branch. The first IF cycle is essentially a stall, because, it never performs useful work. One stall cycle for every branch will yield a performance loss 10% to 30% depending on the branch frequency

Reducing the Brach Penalties

There are many methods for dealing with the pipeline stalls caused by branch delay

1. Freeze or Flush the pipeline, holding or deleting any instructions after the ranch until the branch destination is known. It is a simple scheme and branch penalty is fixed and cannot be reduced by software

2. Treat every branch as not taken, simply allowing the hardware to continue as if the branch were not to executed. Care must be taken not to change the processor
state until the branch outcome is known.

Instructions were fetched as if the branch were a normal instruction. If the branch is taken, it is necessary to turn the fetched instruction in to a no-of instruction and restart the fetch at the target address. Figure 2.8 shows the timing diagram of both the situations.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Untaken Branch	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	ID	EXE	MEM	WB			
Instruction I+2			IF	ID	EXE	MEM	WB		
Instruction I+3				IF	ID	EXE	MEM	WB	
Instruction I+4					IF	ID	EXE	MEM	WB
Taken Branch	IF	ID	EXE	MEM	WB				
Instruction I+1		IF	Idle	Idle	Idle	Idle	Idle		
Branch Target			IF	ID	EXE	MEM	WB		
Branch Target+1				IF	ID	EXE	MEM	WB	
Branch Target+2					IF	ID	EXE	MEM	WB

Figure 2.8 The predicted-not-taken scheme and the pipeline sequence when the branch is untaken (top) and taken (bottom).

3. Treat every branch as taken: As soon as the branch is decoded and target Address is computed, begin fetching and executing at the target if the branch target is known before branch outcome, then this scheme gets advantage.

For both predicated taken or predicated not taken scheme, the compiler can improve performance by organizing the code so that the most frequent path matches the hardware choice.

4. Delayed branch technique is commonly used in early RISC processors.

In a delayed branch, the execution cycle with a branch delay of one is Branch instruction
Sequential successor-1
Branch target if taken

The sequential successor is in the branch delay slot and it is executed irrespective of whether or not the branch is taken. The pipeline behavior with a branch delay is shown in Figure 2.9. Processor with delayed branch, normally have a single instruction delay.
Compiler has to make the successor instructions valid and useful there are three ways in

which the to delay slot can be filled by the compiler.

Instruction	Clock number								
	1	2	3	4	5	6	7	8	9
Untaken Branch	IF	ID	EXE	MEM	WB				
Branch delay Instruction (i+1)		IF	ID	EXE	MEM	WB			
Instruction (i+2)			IF	ID	EXE	MEM	WB		
Instruction (i+3)				IF	ID	EXE	MEM	WB	
Instruction (i+4)					IF	ID	EXE	MEM	WB
Taken Branch	IF	ID	EXE	MEM	WB				
Branch delay Instruction (i+1)		IF	ID	EXE	MEM	WB			
Branch Target			IF	ID	EXE	MEM	WB		
Branch Target+1				IF	ID	EXE	MEM	WB	
Branch Target+2					IF	ID	EXE	MEM	WB

Figure 2.9 Timing diagram of the pipeline to show the behavior of a delayed branch is the same whether or not the branch is taken.

The limitations on delayed branch arise from

i) Restrictions on the instructions that are scheduled in to delay slots.

ii) Ability to predict at compiler time whether a branch is likely to be taken or not taken.

The delay slot can be filled from choosing an instruction

a) From before the branch instruction

b) From the target address

c) From fall- through path.

The principle of scheduling the branch delay is shown in fig 2.10

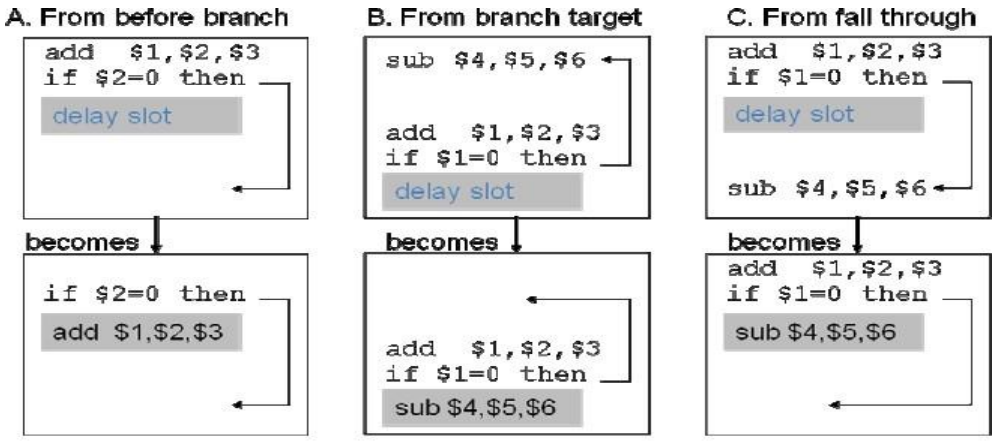


Figure 2.10 Scheduling the Branch delay

What makes pipelining hard to implements?

Dealing with exceptions: Overlapping of instructions makes it more difficult to know whether an instruction can safely change the state of the CPU. In a pipelined CPU, an instruction execution extends over several clock cycles. When this instruction is in execution, the other instruction may raise exception that may force the CPU to abort the instruction in the pipeline before they complete

Types of exceptions:

The term exception is used to cover the terms interrupt, fault and exception.
I/O device request, page fault, Invoking an OS service from a user program, Integer arithmetic overflow, memory protection overflow, Hardware malfunctions, Power failure etc. are the different classes of exception. Individual events have important characteristics that determine what action is needed corresponding to that exception.

i) Synchronous versus Asynchronous

If the event occurs at the same place every time the program is executed with the same data and memory allocation, the event is asynchronous. Asynchronous events are caused by devices external to the CPU and memory such events are handled after the completion of the current instruction.

ii) User requested versus coerced:
User requested exceptions are predictable and can always be handled after the current instruction has completed. Coerced exceptions are caused by some hardware event that is not under the control of the user program. Coerced exceptions are harder to implement because they are not predictable

iii) User maskable versus user non maskable :

If an event can be masked by a user task, it is user maskable. Otherwise it is user non maskable.

iv) Within versus between instructions:
Exception that occur within instruction are usually synchronous, since the instruction triggers the exception. It is harder to implement exceptions that occur withininstructions than those between instructions, since the instruction must be
stopped and restarted. Asynchronous exceptions that occurs within instructions arise from catastrophic situations and always causes program termination.

v) Resume versus terminate:
If the program's execution continues after the interrupt, it is a resuming event otherwise if is terminating event. It is easier implement exceptions that terminate execution. 29

Stopping and restarting execution:

The most difficult exception have 2 properties:

1. Exception that occur within instructions
2. They must be restartable

For example, a page fault must be restartable and requires the intervention of OS. Thus pipeline must be safely shutdown, so that the instruction can be restarted in the correct state. If the restarted instruction is not a branch, then we will continue to fetch the sequential successors and begin their execution in the normal fashion. 11) Restarting is usually implemented by saving the PC of the instruction at which to restart. Pipeline control can take the following steps to save the pipeline state safely.

i) Force a trap instruction in to the pipeline on the next IF

ii) Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in pipeline. This prevents any state changes for instructions that will not be completed before the exception is handled.

iii) After the exception – handling routine receives control, it immediately saves the PC of the faulting instruction. This value will be used to return from the exception later.

NOTE:

1. with pipelining multiple exceptions may occur in the same clock cycle because there are multiple instructions in execution.

2 Handling the exception becomes still more complicated when the instructions are allowed to execute in out of order fashion.

Pipeline implementation

Every MIPS instruction can be implemented in 5 clock cycle

1. Instruction fetch cycles.(IF)

IR ← Mem [PC]

NPC ← PC+ 4

Operation: send out the [PC] and fetch the instruction from memory in to the Instruction Register (IR). Increment PC by 4 to address the next sequential instruction.

2. Instruction decode / Register fetch cycle (ID)

A ← Regs [rs]

B ← Regs [rt]

Imm ← sign – extended immediate field of IR;

Operation: decode the instruction and access that register file to read the registers (rs and rt). File to read the register (rs and rt). A & B are the temporary registers. Operands are kept ready for use in the next cycle.

Decoding is done in concurrent with reading register. MIPS ISA has fixed length Instructions. Hence, these fields are at fixed locations.

3. Execution/ Effective address cycle (EX)

One of the following operations are performed depending on the instruction type.

* Memory reference:

: ALU output ← A+ Imm;

Operation: ALU adds the operands to compute the effective address and places the result in to the register ALU output.

- Register – Register ALU instruction:

ALU output ← A _*func* B;

Operation: The ALU performs the operation specified by the function code on the value taken from content of register A and register B.

*. Register- Immediate ALU instruction:

ALU output ← A *Op* Imm ;

Operation: the content of register A and register Imm are operated (function Op) and result is placed in temporary register ALU output.

*. Branch:

ALU output ← NPC + (Imm << 2)
Cond ← (A == O)

MEMORY HIERARCHY:

Introduction

Cache performance Cache

Optimizations

Virtual memory.

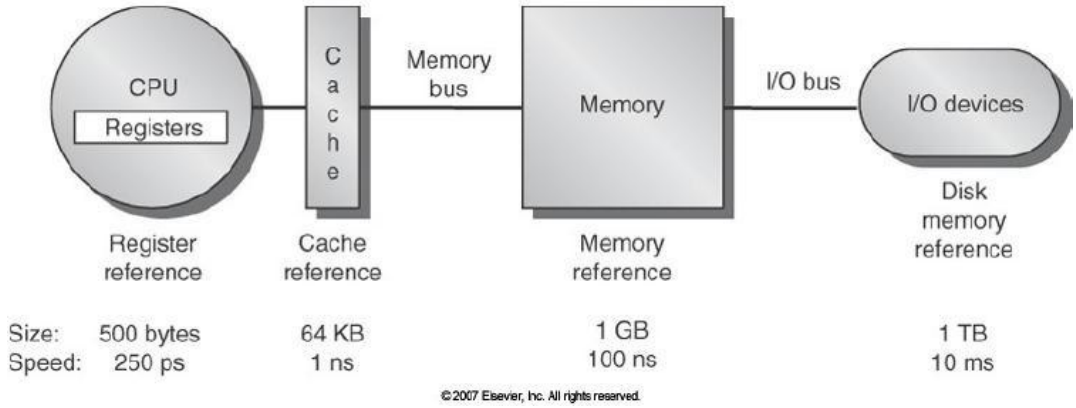
MEMORY HIERARCHY

- Unlimited amount of fast memory
  - Economical solution is memory hierarchy
  - Locality
  - Cost performance

Principle of locality

  - most programs do not access all code or data uniformly.

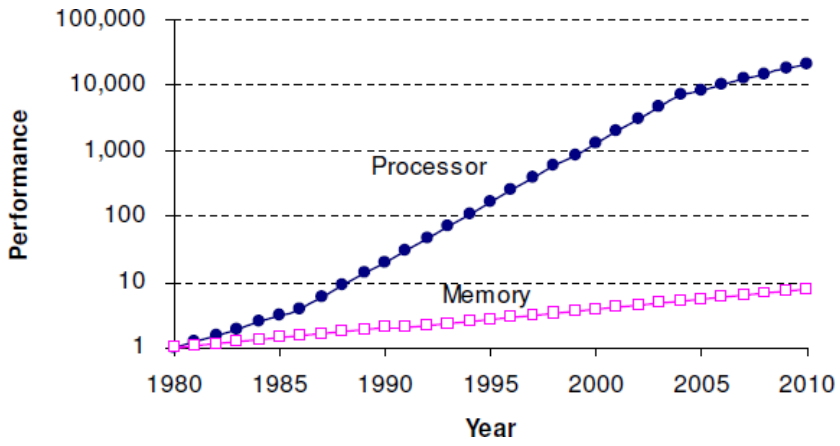
- Locality occurs
  - Time (Temporal locality)
  - Space (spatial locality)
- Guidelines
  - Smaller hardware can be made faster
  - Different speed and sizes



Goal is provide a memory system with cost per byte than the next lower level

- Each level maps addresses from a slower, larger memory to a smaller but faster memory higher in the hierarchy.
  - Address mapping
  - Address checking.
- Hence protection scheme for address for scrutinizing addresses are also part of the memory hierarchy.

Why More on Memory Hierarchy?



- The importance of memory hierarchy has increased with advances in performance of processors.

- Prototype
  - `– When a word is not found in cache
- Fetched from memory and placed in cache with the address tag.
- Multiple words(block) is fetched for moved for efficiency reasons.
  - key design

- Set associative
  - Set is a group of block in the cache.
  - Block is first mapped on to set.
    - » Find mapping
    - » Searching the set Chosen by

    the address of the data:

(Block address) MOD(Number of sets in cache)

- n-block in a set
  - Cache replacement is called n-way set associative.

Cache data

- Cache read.
- Cache write.

Write through: update cache and writes through to update memory. Both strategies

- Use write buffer.

this allows the cache to proceed as soon as the data is placed in the buffer rather than wait the full latency to write the data into memory.

Metric

used to measure the benefits is miss rate

No of access that miss

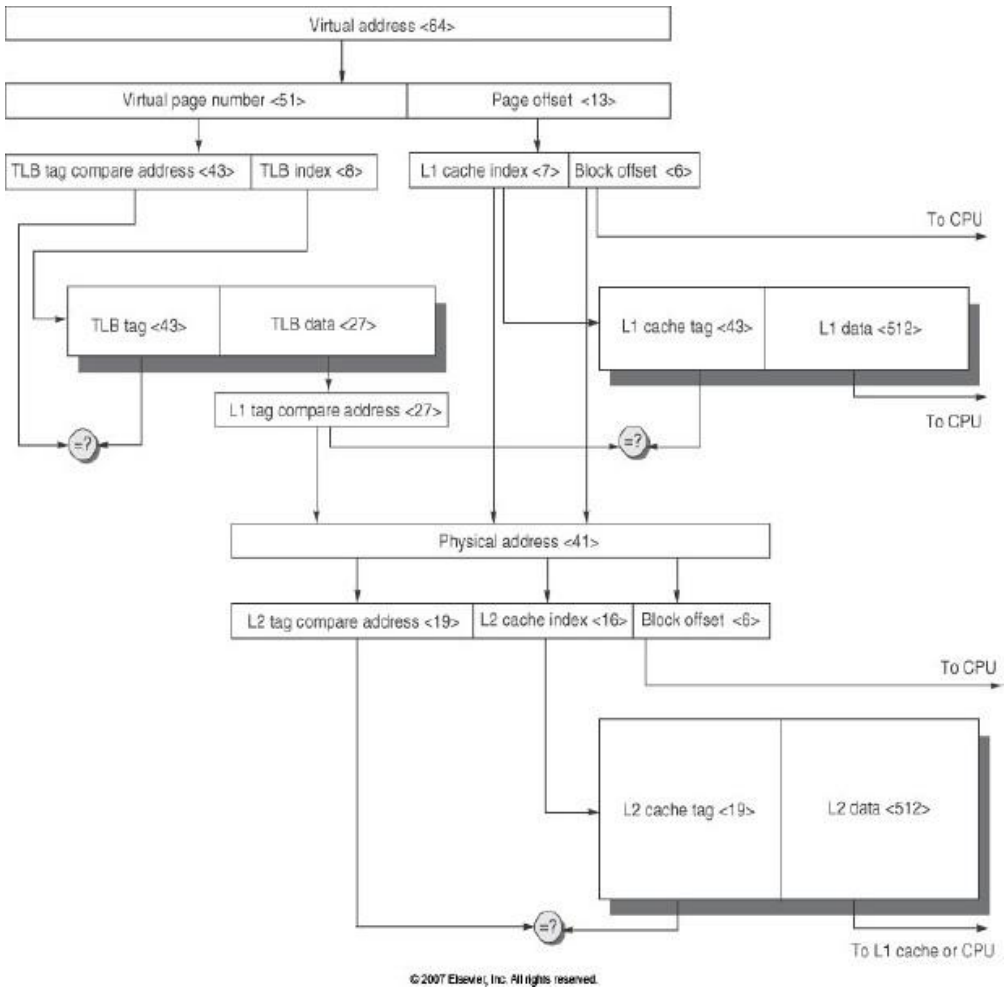
No of accesses

Write back: updates the copy in the cache.

- Causes of high miss rates

– Three Cs model sorts all misses into three categories

- Compulsory: every first access cannot be in cache
  – Compulsory misses are those that occur if there is an infinite cache
- Capacity: cache cannot contain all that blocks that are needed for the program.
  – As blocks are being discarded and later retrieved.
- Conflict: block placement strategy is not fully associative
  – Block miss if blocks map to its set.



Miss rate can be a misleading measure for several reasons

So, misses per instruction can be used per memory reference

$$\frac{\text{Misses}}{\text{Instruction}} = \frac{\text{Miss rate X Memory accesses}}{\text{Instruction count}}$$

$$= \text{Miss rate X } \frac{\text{Mem accesses}}{\text{Instruction}}$$

Cache Optimizations

Six basic cache optimizations

1. Larger block size to reduce miss rate:
- To reduce miss rate through spatial locality.
- Increase block size.
- Larger block size reduce compulsory misses.
- But they increase the miss penalty.

2. Bigger caches to reduce miss rate:
- capacity misses can be reduced by increasing the cache capacity.
- Increases larger hit time for larger cache memory and higher cost and power.

3. Higher associativity to reduce miss rate:
- Increase in associativity reduces conflict misses.

4. Multilevel caches to reduce penalty:
- Introduces additional level cache
- Between original cache and memory.
- L1- original cache

L2- added cache.
L1 cache: - small enough
- speed matches with clock cycle time. L2
cache: - large enough
- capture many access that would go to main memory. Average
access time can be redefined as
Hit timeL1+ Miss rate L1 X (Hit time L2 + Miss rate L2 X Miss penalty L2)

5. Giving priority to read misses over writes to reduce miss penalty:
- write buffer is a good place to implement this optimization.
- write buffer creates hazards: read after write hazard.

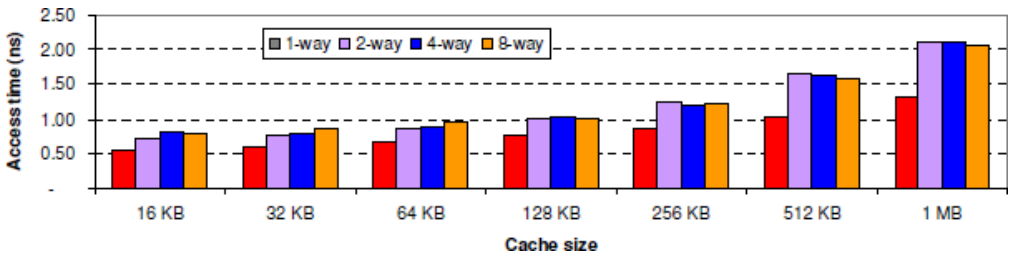
6. Avoiding address translation during indexing of the cache to reduce hit time:
- Caches must cope with the translation of a virtual address from the processor to a physical address to access memory.
- common optimization is to use the page offset.
- part that is identical in both virtual and physical addresses- to index the cache.

Advanced Cache Optimizations

- Reducing hit time
  - Small and simple caches
  - Way prediction
  - Trace caches
- Increasing cache bandwidth
  - Pipelined caches
  - Multibanked caches
  - Nonblocking caches
- Reducing Miss Penalty
  - Critical word first
  - Merging write buffers
- Reducing Miss Rate
  - Compiler optimizations
- Reducing miss penalty or miss rate via parallelism
  - Hardware prefetching
- Compiler prefetching
-

First Optimization : Small and Simple Caches

- Index tag memory and then compare takes time
- _ Small cache can help hit time since smaller memory takes less time to index
  - E.g., L1 caches same size for 3 generations of AMD microprocessors: K6, Athlon, and Opteron
  - Also L2 cache small enough to fit on chip with the processor avoids time penalty of going off chip
- Simple _ direct mapping
  - Can overlap tag check with data transmission since no choice
- Access time estimate for 90 nm using CACTI model 4.0
- Median ratios of access time relative to the direct-mapped caches are 1.32, 1.39, and 1.43 for 2-way, 4-way, and 8-way caches



Second Optimization: Way Prediction

- How to combine fast hit time of Direct Mapped and have the lower conflict misses of 2-way SA cache?

• 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

– Miss _ 1st check other blocks for matches in next clock cycle

• Accuracy » 85%

• Drawback: CPU pipeline is hard if hit takes 1 or 2 cycles

- Used for instruction caches vs. data caches

Third optimization: Trace Cache

• Find more instruction level parallelism?
How to avoid translation from x86 to microops?

• Trace cache in Pentium 4

1. Dynamic traces of the executed instructions vs. static sequences of instructions as determined by layout in memory

– Built-in branch predictor

2. Cache the micro-ops vs. x86 instructions

– Decode/translate from x86 to micro-ops on trace cache miss

+ 1. _ better utilize long blocks (don't exit in middle of block, don't enter at label in middle of block)

- 1. _ complicated address mapping since addresses no longer aligned to powerof-2 multiples of word size

- 1. _ instructions may appear multiple times in multiple dynamic traces due to different branch outcomes

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 mispredicted 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

– requires F/E bits on registers or out-of-order execution

– requires multi-bank memories

• "hit under miss" reduces the effective miss penalty by working during miss vs. ignoring CPU requests

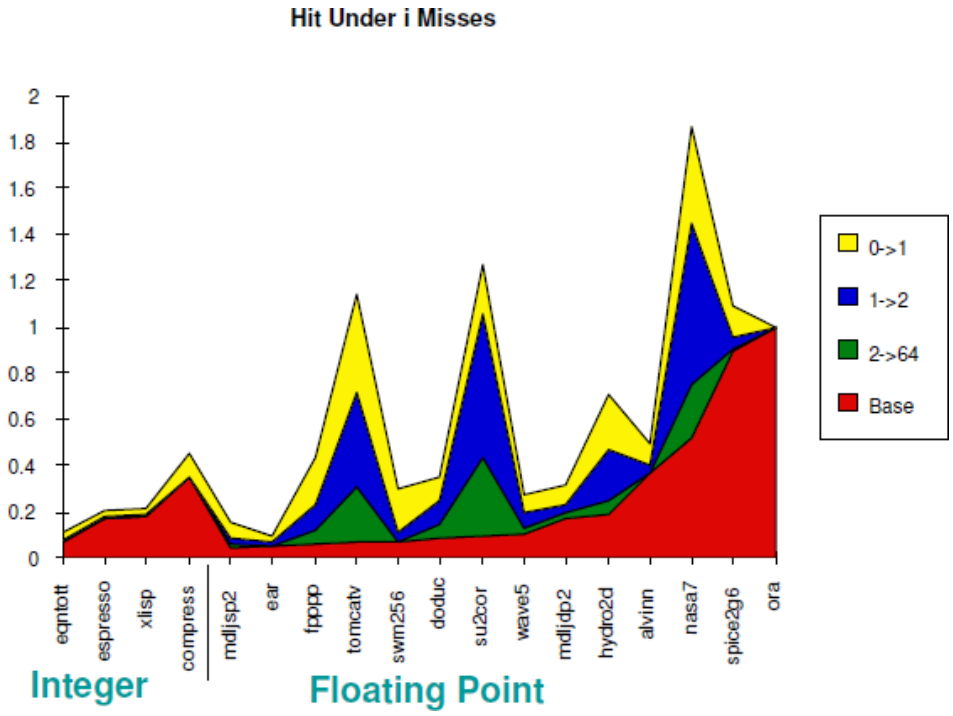
• "hit under multiple miss" or "miss under miss" may further lower the effective miss penalty by overlapping multiple misses

– Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses

– Requires multiple memory banks (otherwise cannot support)

– Pentium Pro allows 4 outstanding memory misses

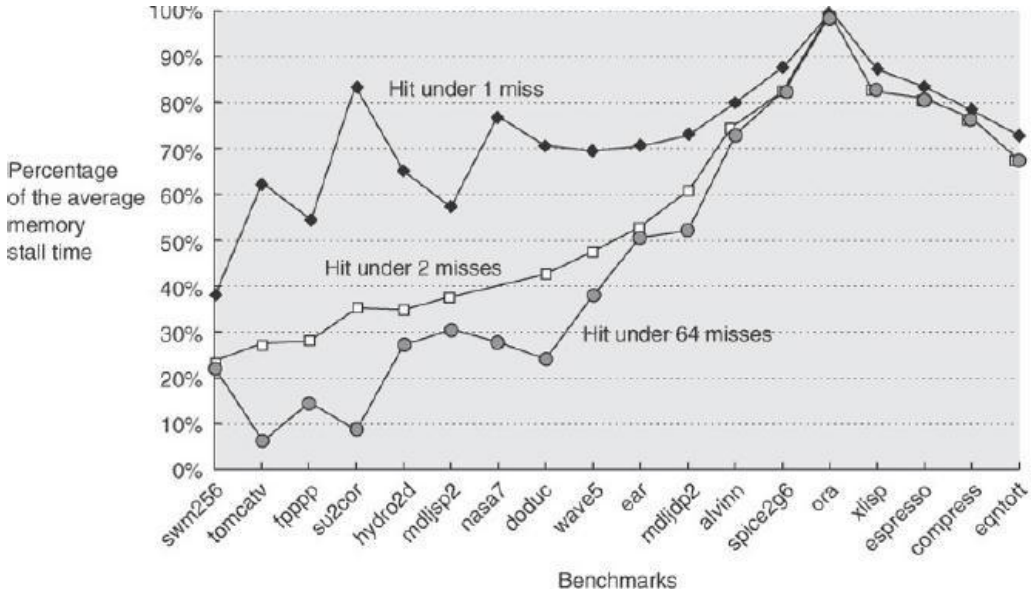
Value of Hit Under Miss for SPEC



• FP programs on average: AMAT= 0.68 -> 0.52 -> 0.34 -> 0.26

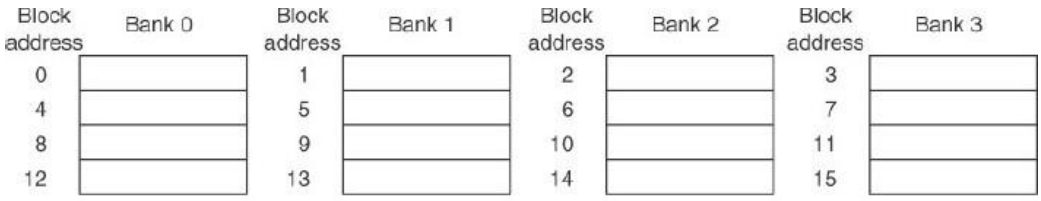
• Int programs on average: AMAT= 0.24 -> 0.20 -> 0.19 -> 0.19

• 8 KB Data Cache, Direct Mapped, 32B block, 16 cycle miss, SPEC 92



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
  - E.g.,T1 ("Niagara") L2 has 4 banks
- Banking works best when accesses naturally spread themselves across banks _ mapping of addresses to banks affects behavior of memory system
- Simple mapping that works well is "sequential interleaving"
  - Spread block addresses sequentially across banks
  - E,g, if there 4 banks, Bank 0 has all blocks whose address modulo 4 is 0; bank 1 has all blocks whose address modulo 4 is 1; …



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

- Don't wait for full block before restarting CPU
- Early restart—As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Spatial locality _ tend to want next sequential word, so not clear size of benefit of just early restart

• 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

– Long blocks more popular today _ Critical Word 1st Widely used

Eight optimization: Merging Write Buffer to Reduce Miss Penalty-

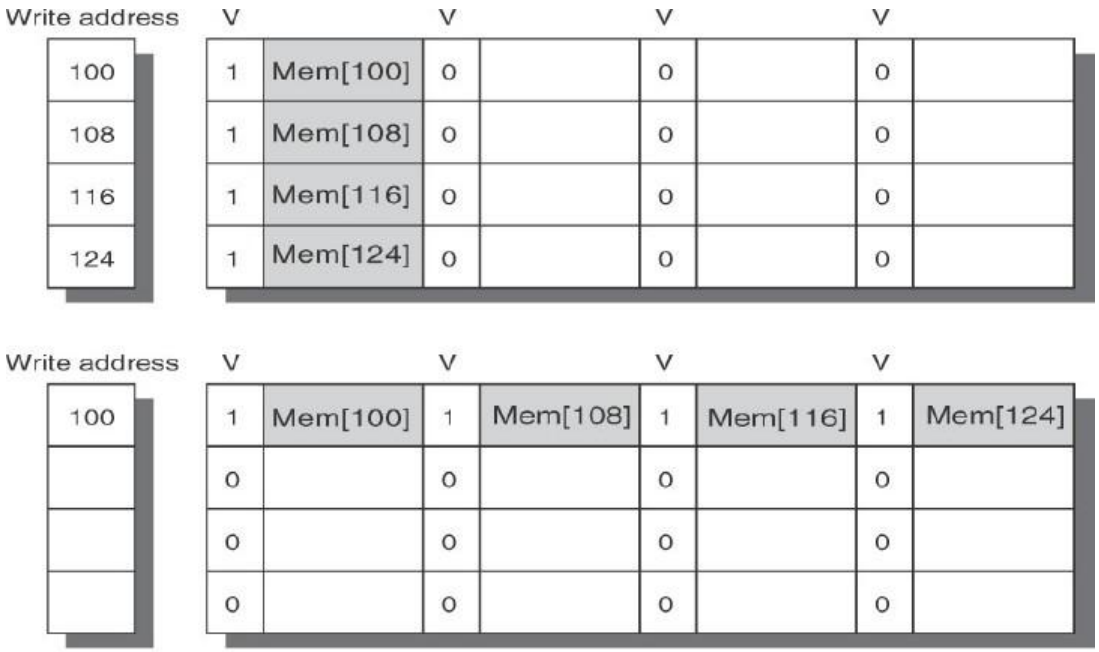
• Write buffer to allow processor to continue while waiting to write to memory

• 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



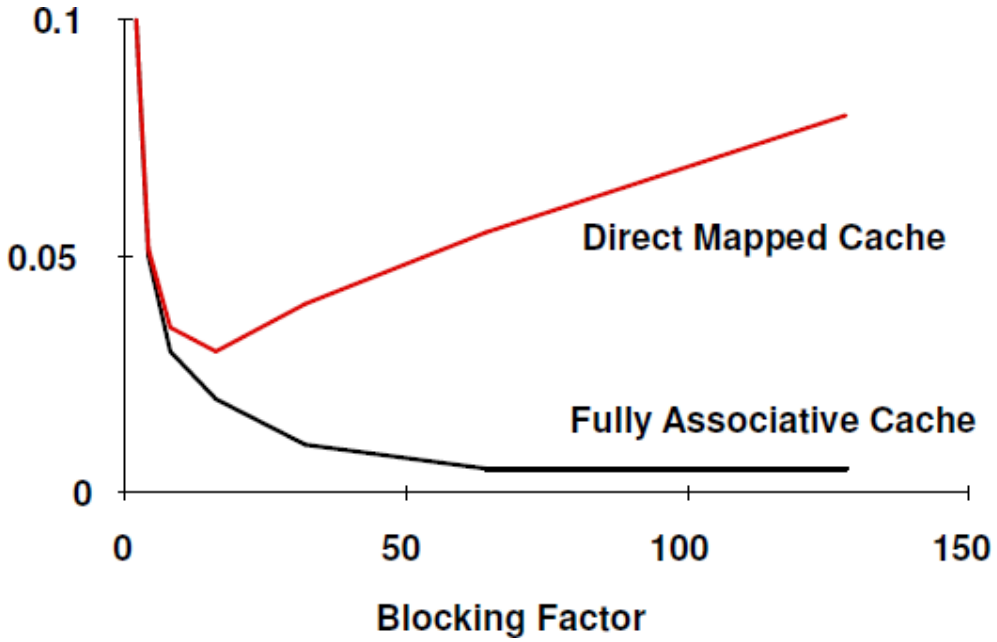
Ninth optimization: Reducing Misses by Compiler Optimizations

- McFarling [1989] reduced caches misses by 75% on 8KB direct mapped cache, 4 byte blocks in software
- 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

Merging Arrays Example

```
/* Before: 2 sequential arrays */ int
val[SIZE];
int key[SIZE];
/* After: 1 array of stuctures */ struct
merge {
int val;
int key;
};
struct merge merged_array[SIZE];
```

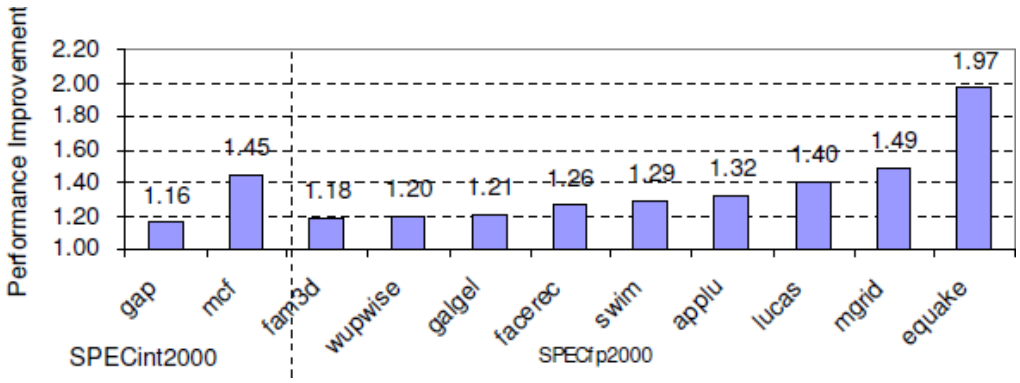
Reducing conflicts between val & key; improve spatial locality



- Conflict misses in caches not FA vs. Blocking size
  - – Lam et al [1991] a blocking factor of 24 had a fifth the misses vs. 48 despite both fit in cache

Tenth optimization Reducing Misses by Hardware Prefetching of Instructions & Data

- Prefetching relies on having extra memory bandwidth that can be used without penalty
- 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



Eleventh optimization: Reducing Misses by Software Prefetching Data

- Data Prefetch
  - – Load data into register (HP PA-RISC loads)
  - – Cache Prefetch: load into cache (MIPS IV, PowerPC, SPARC v. 9)
  - – Special prefetching instructions cannot cause faults; a form of speculative execution
- Issuing Prefetch Instructions takes time
  - – Is cost of prefetch issues < savings in reduced misses?
  - – Higher superscalar reduces difficulty of issue bandwidth

The techniques to improve hit time, bandwidth, miss penalty and miss rate generally affect the other components of the average memory access equation as well as the complexity of the memory hierarchy.

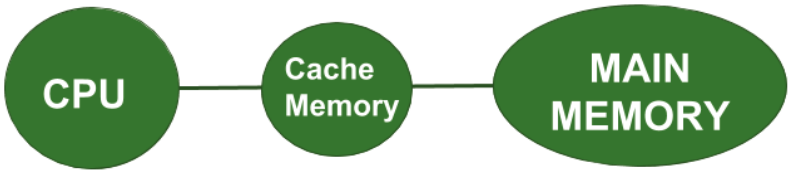
MEMORY

Multilevel Cache Organisation
Cache is a random access memory used by the CPU to reduce the average time taken to access memory.
Multilevel Caches is one of the techniques to improve Cache Performance by reducing the "MISS PENALTY". Miss Penalty refers to the extra time required to bring the data into cache from the Main memory whenever there is a "miss" in cache .
For clear understanding let us consider an example where CPU requires 10 Memory References for accessing the desired information and consider this scenario in the following 3 cases of System design :

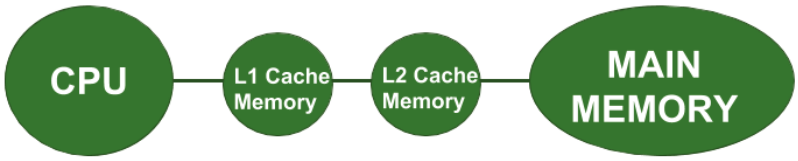
Case 1 : System Design without Cache Memory

Here the CPU directly communicates with the main memory and no caches are involved.
In this case, the CPU needs to access the main memory 10 times to access the desired information.

Case 2 : System Design with Cache Memory



Here the CPU at first checks whether the desired data is present in the Cache Memory or not i.e. whether there is a "hit" in cache or "miss" in cache. Suppose there are 3 miss in Cache Memory then the Main Memory will be accessed only 3 times. We can see that here the miss penalty is reduced because the Main Memory is accessed a lesser number of times than that in the previous case.
Case 3 : System Design with Multilevel Cache Memory



Here the Cache performance is optimized further by introducing multilevel Caches. As shown in the above figure, we are considering 2 level Cache Design. Suppose there are 3 miss in the L1 Cache Memory and out of these 3 misses there are 2 miss in the L2 Cache Memory then the Main Memory will be accessed only 2 times. It is clear that here the Miss Penalty is reduced considerably than that in the previous case thereby improving the Performance of Cache Memory.
NOTE :
We can observe from the above 3 cases that we are trying to decrease the number of Main Memory References and thus decreasing the Miss Penalty in order to improve the overall System Performance. Also, it is important to note that in the

Multilevel Cache Design, L1 Cache is attached to the CPU and it is small in size but fast. Although, L2 Cache is attached to the Primary Cache i.e. L1 Cache and it is larger in size and slower but still faster than the Main Memory.

Effective Access Time = Hit rate * Cache access time
+ Miss rate * Lower level access time

Average access Time For Multilevel Cache:(T_{avg})
$T_{avg} = H_1 * C_1 + (1 – H_1) * (H_2 * C_2 +(1 – H_2) *M)$

where
H1 is the Hit rate in the L1 caches.
H2 is the Hit rate in the L2 cache.
C1 is the Time to access information in the L1 caches.
C2 is the Miss penalty to transfer information from the L2 cache to an L1 cache.
M is the Miss penalty to transfer information from the main memory to the L2 cache.

Example:
Find the Average memory access time for a processor with a 2 ns clock cycle time, a miss rate of 0.04 misses per instruction, a miss penalty of 25 clock cycles, and a cache access time (including hit detection) of 1 clock cycle. Also, assume that the read and write miss penalties are the same and ignore other write stalls.
Solution:
Average Memory access time(AMAT)= Hit Time + Miss Rate * Miss Penalty.

Hit Time = 1 clock cycle (Hit time = Hit rate * access time) but here Hit time is directly given so,

Miss rate = 0.04

Miss Penalty= 25 clock cycle (this is time taken by the above level of memory after the hit)

so, AMAT= 1 + 0.04 * 25
AMAT= 2 clock cycle

according to question 1 clock cycle = 2 ns

AMAT = 4ns

Cache coherence

In computer architecture, cache coherence is the uniformity of shared resource data that ends up stored in multiple local caches. When clients in a system maintain caches of a common memory resource, problems may arise with incoherent data, which is particularly the case with CPUs in a multiprocessing system.

In the illustration on the right, consider both the clients have a cached copy of a particular memory block from a previous read. Suppose the client on the bottom updates/changes that memory block, the client on the top could be left with an invalid cache of memory without any notification of the change. Cache coherence is intended to manage such conflicts by maintaining a coherent view of the data values in multiple caches.

In a shared memory multiprocessor system with a separate cache memory for each processor, it is possible to have many copies of shared data: one copy in the main memory and one in the local cache of each processor that requested it. When one of the copies of data is changed, the other copies must reflect that change. Cache coherence is the discipline which ensures that the changes in the values of shared operands (data) are propagated throughout the system in a timely fashion.

The following are the requirements for cache coherence:

Write Propagation
Changes to the data in any cache must be propagated to other copies (of that cache line) in the peer caches.
Transaction Serialization
Reads/Writes to a single memory location must be seen by all processors in the same order.
Theoretically, coherence can be performed at the load/store granularity. However, in practice it is generally performed at the granularity of cache blocks.

Coherence defines the behavior of reads and writes to a single address location.[2]

One type of data occurring simultaneously in different cache memory is called cache coherence or in some systems known as global memory.

In a multiprocessor system, consider that more than one processor has cached a copy of the memory location X. The following conditions are necessary to achieve cache coherence:[4]

In a read made by a processor P to a location X that follows a write by the same processor P to X, with no writes to X by another processor occurring between the write and the read instructions made by P, X must always return the value written by P.
In a read made by a processor P1 to location X that follows a write by another processor P2 to X, with no other writes to X made by any processor occurring between the two accesses and with the read and write being sufficiently separated, X must always return the value written by P2. This condition defines the concept of coherent view of memory. Propagating the writes to the shared memory location ensures that all the caches have a coherent view of the memory. If processor P1 reads the old value of X, even after the write by P2, we can say that the memory is incoherent.
The above conditions satisfy the Write Propagation criteria required for cache coherence. However, they are not sufficient as they do not satisfy the Transaction Serialization condition. To illustrate this better, consider the following example:

A multi-processor system consists of four processors - P1, P2, P3 and P4, all containing cached copies of a shared variable S whose initial value is 0. Processor P1 changes the value of S (in its cached copy) to 10 following which processor P2 changes the value of S in its own cached copy to 20. If we ensure only write propagation, then P3 and P4 will certainly see the changes made to S by P1 and P2. However, P3 may see the change made by P1 after seeing the change made by P2 and hence return 10 on a read to S. P4 on the other hand may see changes made by P1 and P2 in the order in which they are made and hence return 20 on a read to S. The processors P3 and P4 now have an incoherent view of the memory.

Therefore, in order to satisfy Transaction Serialization, and hence achieve Cache Coherence, the following condition along with the previous two mentioned in this section must be met:

Writes to the same location must be sequenced. In other words, if location X received two different values A and B, in this order, from any two processors, the processors can never read location X as B and then read it as A. The location X must be seen with values A and B in that order.
The alternative definition of a coherent system is via the definition of sequential consistency memory model: "the cache coherent system must appear to execute all threads' loads and stores to a single memory location in a total order that respects the program order of each thread
.Thus, the only difference between the cache coherent system and sequentially consistent system is in the number of address locations the definition talks about (single memory location for a cache coherent system, and all memory locations for a sequentially consistent system).

Another definition is: "a multiprocessor is cache consistent if all writes to the same memory location are performed in some sequential order".

Rarely, but especially in algorithms, coherence can instead refer to the locality of reference. Multiple copies of same data can exist in different cache simultaneously and if processors are allowed to update their own copies freely, an inconsistent view of memory can result.

UNIT 5

Virtual memory-

Hardware support for address translation,
page fault handling.
Translation look aside buffer.
Hardware-software interface.

Virtual memory

In computing, virtual memory (also virtual storage) is a memory management technique that provides an "idealized abstraction of the storage resources that are actually available on a given machine" which "creates the illusion to users of a very large (main) memory".

The computer's operating system, using a combination of hardware and software, maps memory addresses used by a program, called virtual addresses, into physical addresses in computer memory. Main storage, as seen by a process or task, appears as a contiguous address space or collection of contiguous segments. The operating system manages virtual address spaces and the assignment of real memory to virtual memory. Address translation hardware in the CPU, often referred to as a memory management unit (MMU), automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities to provide a virtual address space that can exceed the capacity of real memory and thus reference more memory than is physically present in the computer.

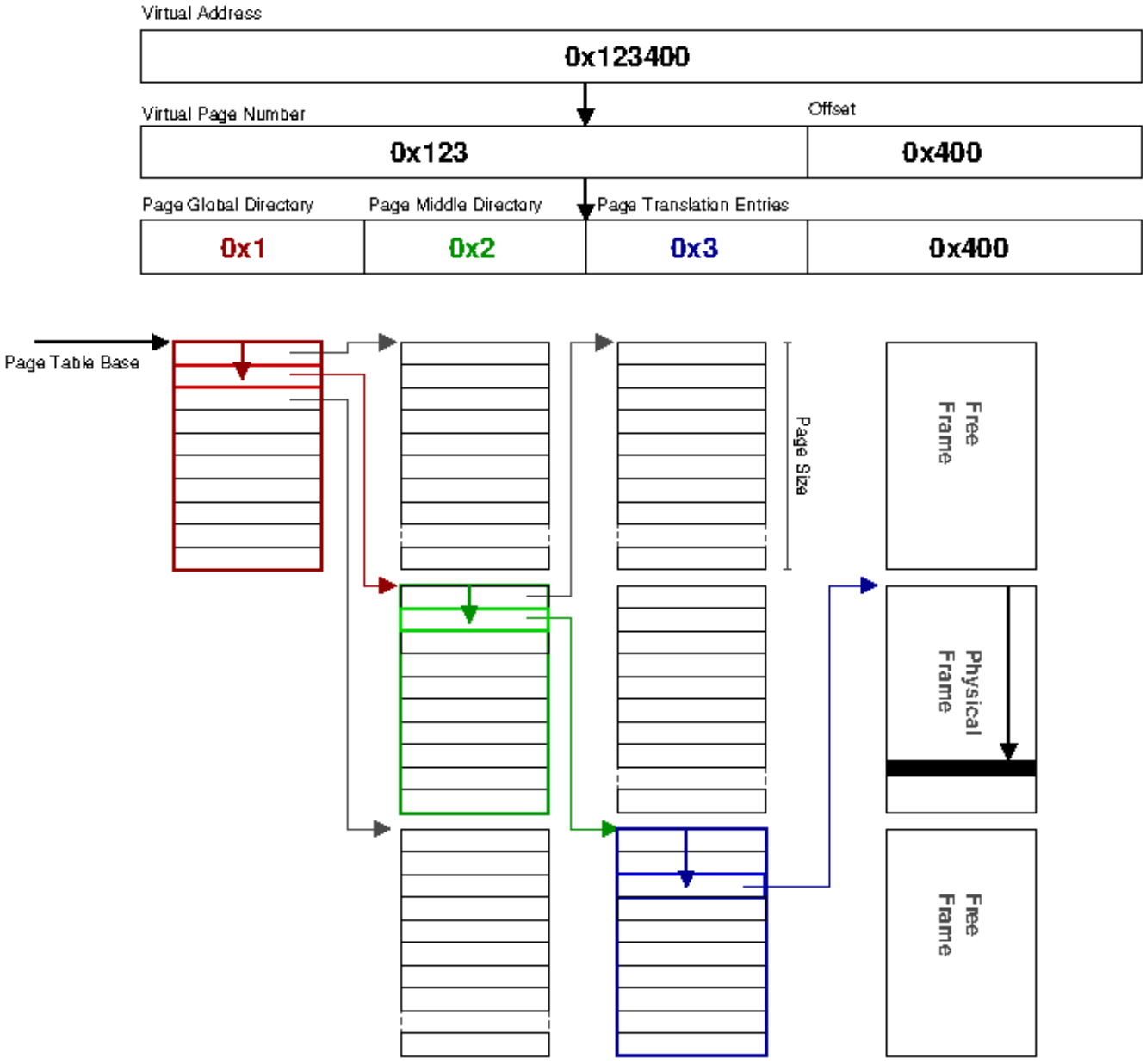
The primary benefits of virtual memory include freeing applications from having to manage a shared memory space, increased security due to memory isolation, and being able to conceptually use more memory than might be physically available, using the technique of paging.

Hardware support for virtual memory

As covered in the section called “The TLB”, the processor hardware provides a lookup-table that links virtual addresses to physical addresses. Each processor architecture defines different ways to manage the TLB with various advantages and disadvantages.
The part of the processor that deals with virtual memory is generally referred to as the Memory Management Unit or MMU
x86-64
XXX
Itanium
The Itanium MMU provides many interesting features for the operating system to work with virtual memory.
Address spaces
the section called “Flushing the TLB” introduced the concept of the address-space ID to reduce the overheads of flushing the TLB when context switching. However, programmers often use threads to allow execution contexts to share an address space. Each thread has the same ASID and hence shares TLB entries, leading to increased performance. However, a single ASID prevents the TLB from enforcing protection; sharing becomes an "all or

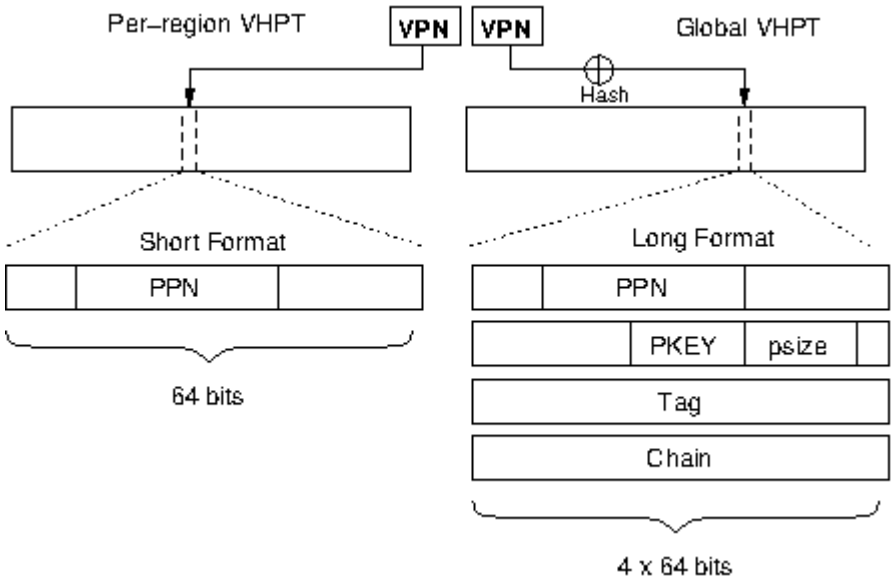
nothing" approach. To share even a few bytes, threads must forgo all protection from each other (see also the section called “Protection”).

Figure 6.7. Illustration Itanium regions and protection keys



The Itanium MMU considers these problems and provides the ability to share an address space (and hence translation entries) at a much lower granularity whilst still maintaining protection within the hardware. The Itanium divides the 64-bit address space up into 8 regions, as illustrated in Figure 6.7, “Illustration Itanium regions and protection keys”. Each process has eight 24-bit region registers as part of its state, which each hold a region ID (RID) for each of the eight regions of the process address space. TLB translations are tagged with the RID and thus will only match if the process also holds this RID, as illustrated in Figure 6.8, “Illustration of Itanium TLB translation”.

Figure 6.8. Illustration of Itanium TLB translation



Further to this, the top three bits (the region bits) are not considered in virtual address translation. Therefore, if two processes share a RID (i.e., hold the same value in one of their region registers) then they have an aliased view of that region. For example, if process-A holds RID 0x100 in region-register 3 and process-B holds the same RID 0x100 in region-register 5 then process-A, region 3 is aliased to process-B, region 5. This limited sharing means both processes receive the benefits of shared TLB entries without having to grant access to their entire address space.

Protection Keys

To allow for even finer grained sharing, each TLB entry on the Itanium is also tagged with a protection key. Each process has an additional number of protection key registers under operating-system control.

When a series of pages is to be shared (e.g., code for a shared system library), each page is tagged with a unique key and the OS grants any processes allowed to access the pages that key. When a page is referenced the TLB will check the key associated with the translation entry against the keys the process holds in its protection key registers, allowing the access if the key is present or otherwise raising a protection fault to the operating system. The key can also enforce permissions; for example, one process may have a key which grants write permissions and another may have a read-only key. This allows for sharing of translation entries in a much wider range of situations with granularity right down to a single-page level, leading to large potential improvements in TLB performance.

Itanium Hardware Page-Table Walker

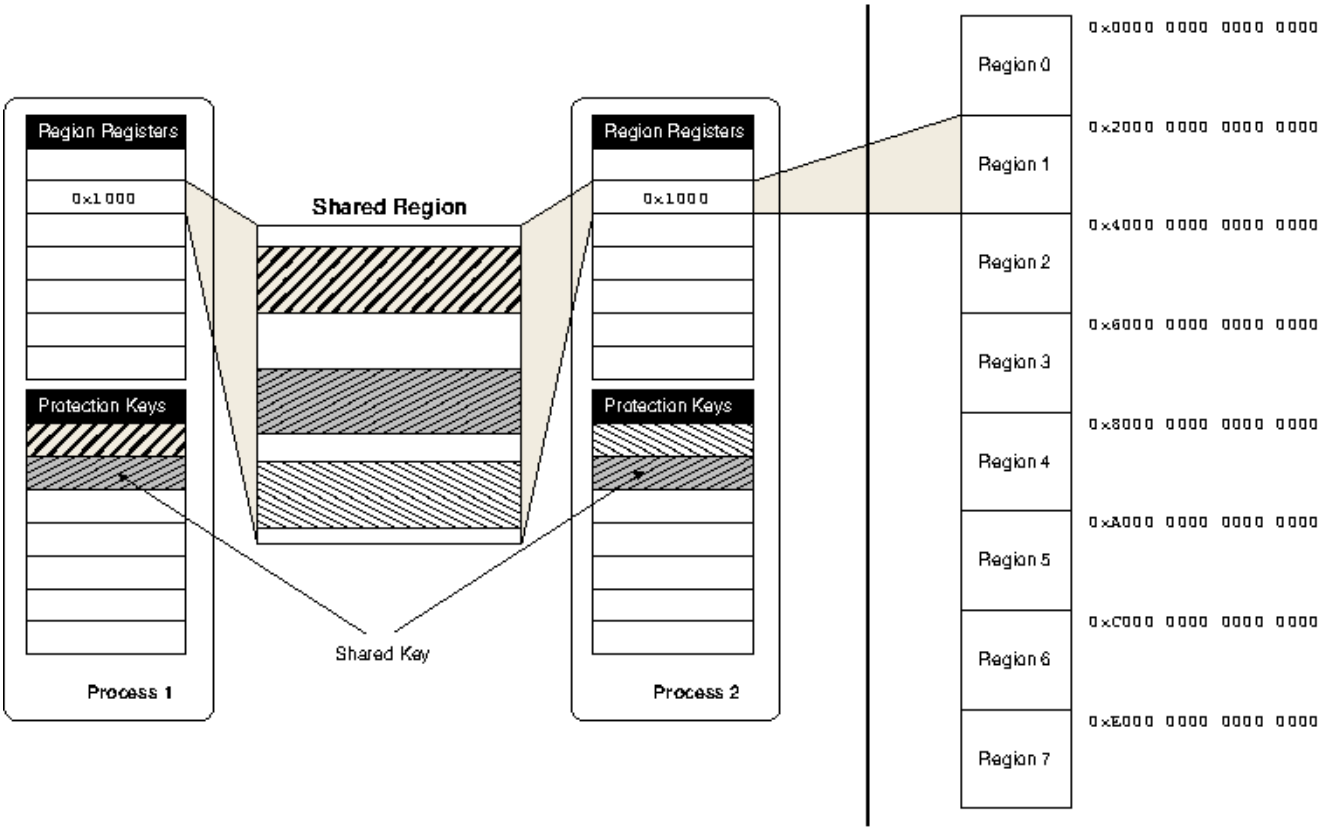
Switching context to the OS when resolving a TLB miss adds significant overhead to the fault processing path. To combat this, Itanium allows the option of using built-in hardware to read the page-table and automatically load virtual-to-physical translations into the TLB. The hardware page-table walker (HPW) avoids the expensive transition to the OS, but requires translations to be in a fixed format suitable for the hardware to understand.

The Itanium HPW is referred to in Intel's documentation as the virtually hashed page-table walker or VHPT walker, for reasons which should become clear. Itanium gives developers the option of two mutually exclusive HPW implementations; one based on a virtual linear page-table and the other based on a hash table.

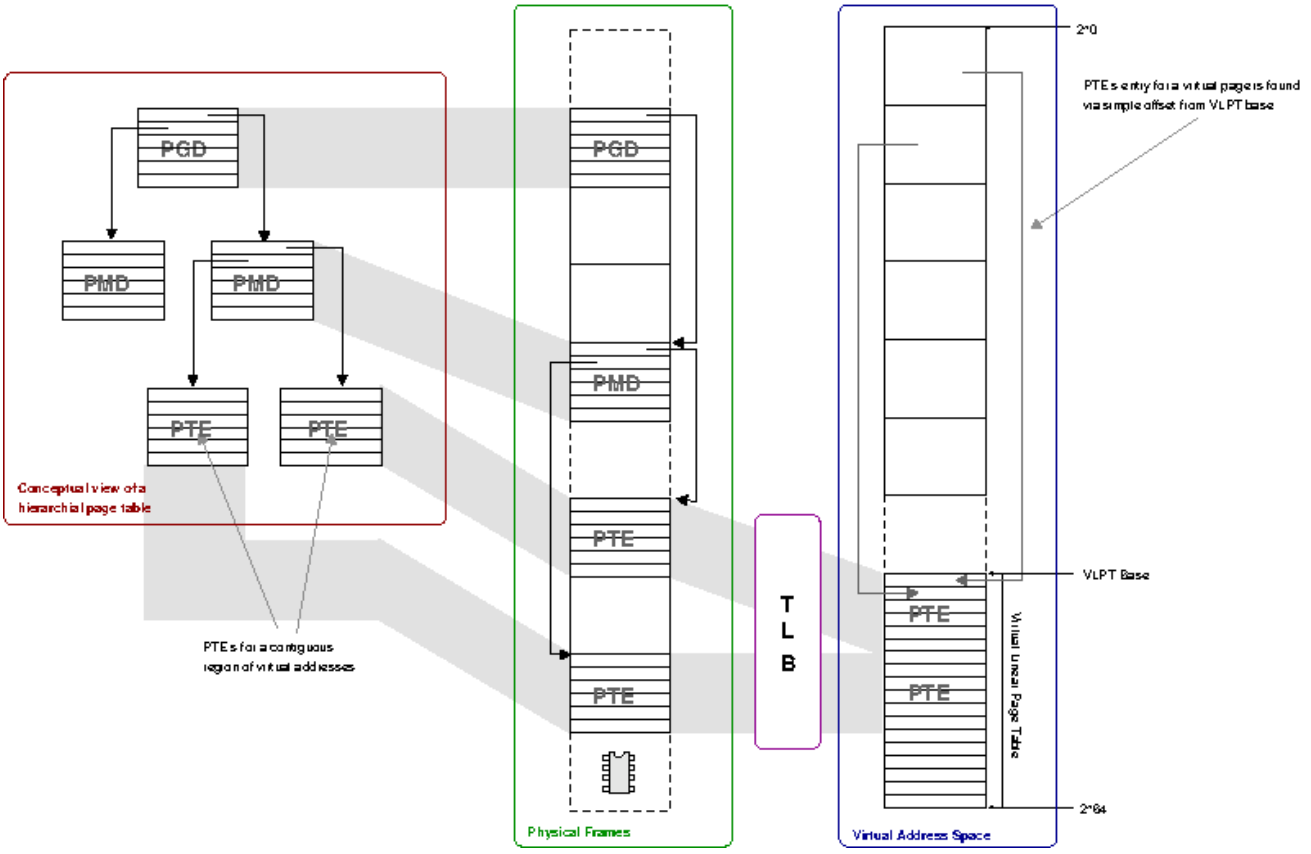
It should be noted it is possible to operate with no hardware page-table walker; in this case each TLB miss is resolved by the OS and the processor becomes a software-loaded architecture. However, the performance impact of disabling the HPW is so considerable it is very unlikely any benefit could be gained from doing so

Virtual Linear Page-Table

The virtual linear page-table implementation is referred to in documentation as the short format virtually hashed page-table (SF-VHPT). It is the default HPW model used by Linux on Itanium.
The usual solution is a multi-level or hierarchical page-table, where the bits comprising the virtual page number are used as an index into intermediate levels of the page-table (see the section called "Three Level Page Table"). Empty regions of the virtual address space simply do not exist in the hierarchical page-table. Compared to a linear page-table, for the (realistic) case of a tightly-clustered and sparsely-filled address space, relatively little space is wasted in overheads. The major disadvantage is the multiple memory references required for lookup.
Figure 6.9. Illustration of a hierarchical page-table

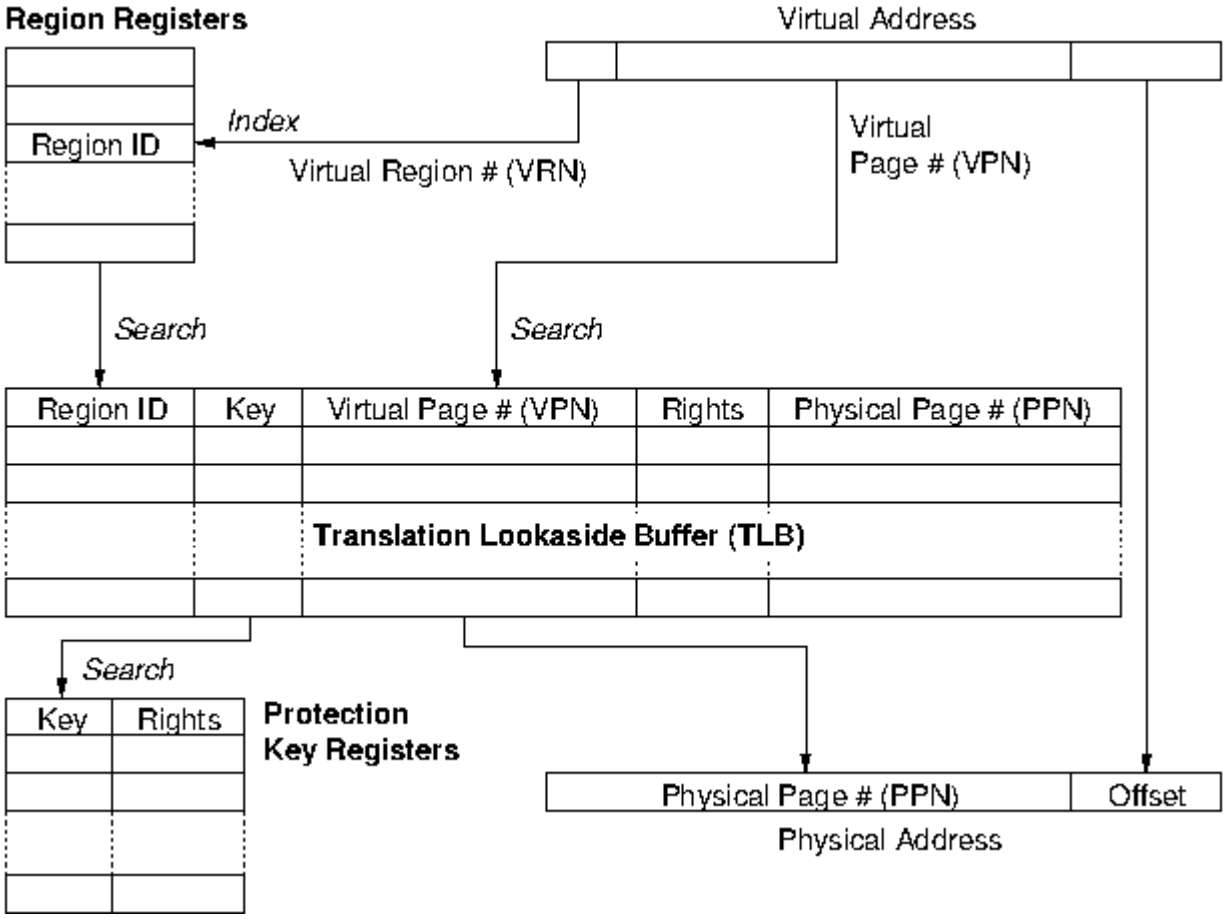


With a 64-bit address space, even a 512~GiB linear table identified in the section called "Virtual Address Translation" takes only 0.003% of the 16-exabytes available. Thus a virtual linear page-table (VLPT) can be created in a contiguous area of virtual address space.
Just as for a physically linear page-table, on a TLB miss the hardware uses the virtual page number to offset from the page-table base. If this entry is valid, the translation is read and inserted directly into the TLB. However, with a VLPT the address of the translation entry is itself a virtual address and thus there is the possibility that the virtual page which it resides in is not present in the TLB. In this case a nested fault is raised to the operating system. The software must then correct this fault by mapping the page holding the translation entry into the VLPT.
Figure 6.10. Itanium short-format VHPT implementation



This process can be made quite straight forward if the operating system keeps a hierarchical page-table. The leaf page of a hierarchical page-table holds translation entries for a virtually contiguous region of addresses and can thus be mapped by the TLB to create the VLPT as described in Figure 6.10, “Itanium short-format VHPT implementation”.

Figure 6.11. Itanium PTE entry formats



The major advantage of a VLPT occurs when an application makes repeated or contiguous accesses to memory. Consider that for a walk of virtually contiguous memory, the first fault will map a page full of translation entries into the virtual linear page-table. A subsequent access to the next virtual page will require the next translation entry to be loaded into the TLB, which is now available in the VLPT and thus loaded very quickly and without invoking the operating system. Overall, this will be an advantage if the cost of the initial nested fault is amortised over subsequent HPW hits.

The major drawback is that the VLPT now requires TLB entries which causes an increase on TLB pressure. Since each address space requires its own page table the overheads become greater as the system becomes more active. However, any increase in TLB capacity misses should be more than regained in lower refill costs from the efficient hardware walker. Note that a pathological case could skip over page_size ÷ translation_size entries, causing repeated nested faults, but this is a very unlikely access pattern.

The hardware walker expects translation entries in a specific format as illustrated on the left of Figure 6.11, “Itanium PTE entry formats”. The VLPT requires translations in the so-called 8-byte short format. If the operating system is to use its page-table as backing for the VLPT (as in Figure 6.10, “Itanium short-format VHPT implementation”) it must use this translation format. The architecture describes a limited number of bits in this format as ignored and thus available for use by software, but significant modification is not possible.

A linear page-table is premised on the idea of a fixed page size. Multiple page-size support is problematic since it means the translation for a given virtual page is no longer at a constant offset. To combat this, each of the 8-regions of the address space (Figure 6.7, “Illustration Itanium regions and protection keys”) has a separate VLPT which only maps addresses for that region. A default page-size can be given for each region (indeed, with Linux HugeTLB, discussed below, one region is dedicated to larger pages). However, page sizes can not be mixed within a region.

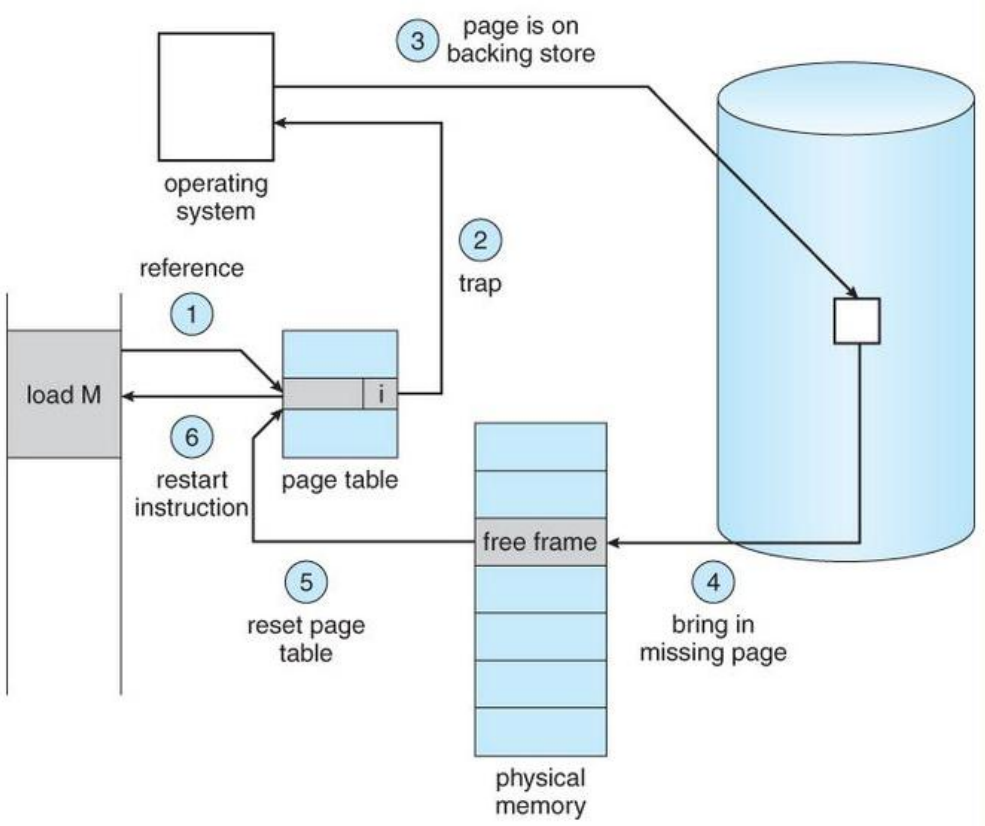
Virtual Hash Table

Using TLB entries in an effort to reduce TLB refill costs, as done with the SF-VHPT, may or may not be an

effective trade-off. Itanium also implements a hashed page-table with the potential to lower TLB overheads. In this scheme, the processor hashes a virtual address to find an offset into a contiguous table.
The previously described physically linear page-table can be considered a hash page-table with a perfect hash function which will never produce a collision. However, as explained, this requires an impractical trade-off of huge areas of contiguous physical memory. However, constraining the memory requirements of the page table raises the possibility of collisions when two virtual addresses hash to the same offset. Colliding translations require a chain pointer to build a linked-list of alternative possible entries. To distinguish which entry in the linked-list is the correct one requires a tag derived from the incoming virtual address.
The extra information required for each translation entry gives rise to the moniker long-format~VHPT (LF-VHPT). Translation entries grow to 32-bytes as illustrated on the right hand side of Figure 6.11, “Itanium PTE entry formats”.
The main advantage of this approach is the global hash table can be pinned with a single TLB entry. Since all processes share the table it should scale better than the SF-VHPT, where each process requires increasing numbers of TLB entries for VLPT pages. However, the larger entries are less cache friendly; consider we can fit four 8-byte short-format entries for every 32-byte long-format entry. The very large caches on the Itanium processor may help mitigate this impact, however.
One advantage of the SF-VHPT is that the operating system can keep translations in a hierarchical page-table and, as long as the hardware translation format is maintained, can map leaf pages directly to the VLPT. With the LF-VHPT the OS must either use the hash table as the primary source of translation entries or otherwise keep the hash table as a cache of its own translation information. Keeping the LF-VHPT hash table as a cache is somewhat sub-optimal because of increased overheads on time critical fault paths, however advantages are gained from the table requiring only a single TLB entry.

Page Fault Handling in Operating System

A page fault occurs when a program attempts to access data or code that is in its address space, but is not currently located in the system RAM. So when page fault occurs then following sequence of events happens :



- The computer hardware traps to the kernel and program counter (PC) is saved on the stack. Current instruction state information is saved in CPU registers.
- An assembly program is started to save the general registers and other volatile information to keep the OS from destroying it.
- Operating system finds that a page fault has occurred and tries to find out which virtual page is needed. Some times hardware register contains this required information. If not, the operating system must retrieve PC, fetch instruction and find out what it was doing when the fault occurred.
- Once virtual address caused page fault is known, system checks to see if address is valid and checks if there is no protection access problem.
- If the virtual address is valid, the system checks to see if a page frame is free. If no frames are free, the page replacement algorithm is run to remove a page.
- If frame selected is dirty, page is scheduled for transfer to disk, context switch takes place, fault process is suspended and another process is made to run until disk transfer is completed.
- As soon as page frame is clean, operating system looks up disk address where needed page is, schedules disk operation to bring it in.
- When disk interrupt indicates page has arrived, page tables are updated to reflect its position, and frame marked as being in normal state.

- Faulting instruction is backed up to state it had when it began and PC is reset. Faulting is scheduled, operating system returns to routine that called it.
- Assembly Routine reloads register and other state information, returns to user space to continue execution.

Translation Lookaside Buffer (TLB) in Paging

In Operating System (Memory Management Technique : Paging), for each process page table will be created, which will contain Page Table Entry (PTE). This PTE will contain information like frame number (The address of main memory where we want to refer), and some other useful bits (e.g., valid/invalid bit, dirty bit, protection bit etc). This page table entry (PTE) will tell where in the main memory the actual page is residing.

Now the question is where to place the page table, such that overall access time (or reference time) will be less.

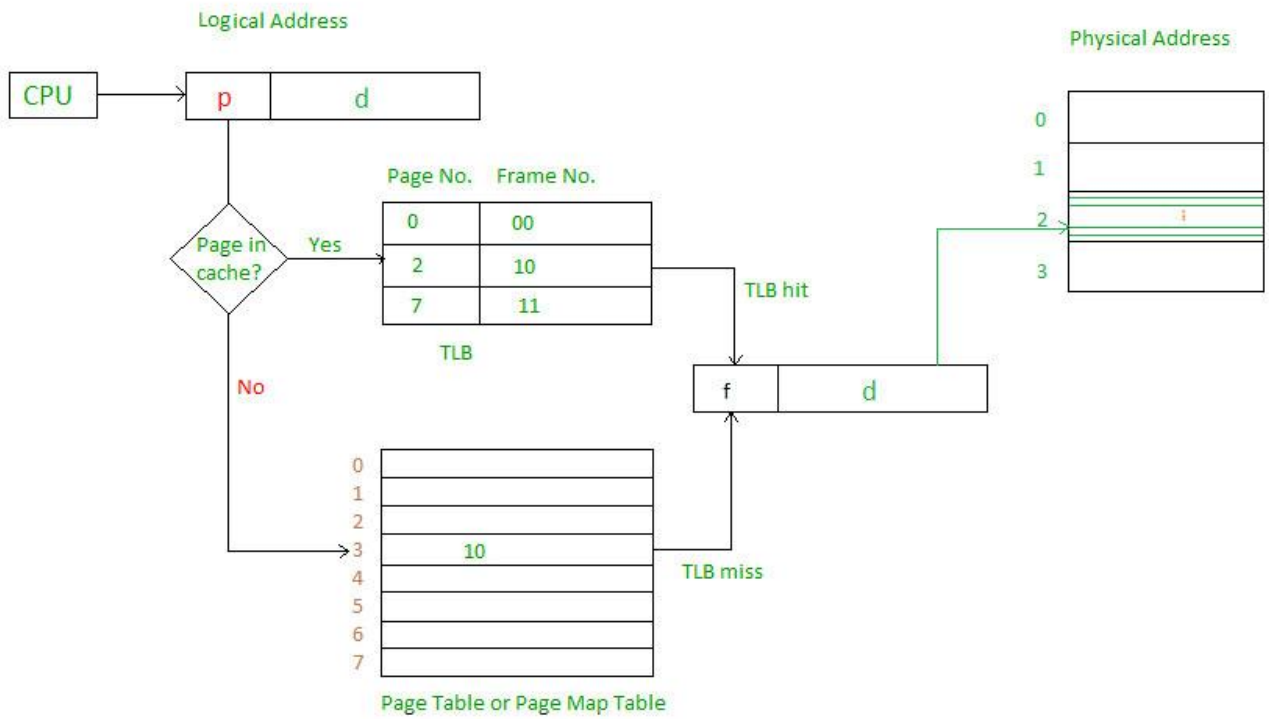
The problem initially was to fast access the main memory content based on address generated by CPU (i.e logical/virtual address). Initially, some people thought of using registers to store page table, as they are high-speed memory so access time will be less.

The idea used here is, place the page table entries in registers, for each request generated from CPU (virtual address), it will be matched to the appropriate page number of the page table, which will now tell where in the main memory that corresponding page resides. Everything seems right here, but the problem is register size is small (in practical, it can accommodate maximum of 0.5k to 1k page table entries) and process size may be big hence the required page table will also be big (lets say this page table contains 1M entries), so registers may not hold all the PTE's of Page table. So this is not a practical approach.

To overcome this size issue, the entire page table was kept in main memory. but the problem here is two main memory references are required:

1. To find the frame number
2. To go to the address specified by frame number

To overcome this problem a high-speed cache is set up for page table entries called a Translation Lookaside Buffer (TLB). Translation Lookaside Buffer (TLB) is nothing but a special cache used to keep track of recently used transactions. TLB contains page table entries that have been most recently used. Given a virtual address, the processor examines the TLB if a page table entry is present (TLB hit), the frame number is retrieved and the real address is formed. If a page table entry is not found in the TLB (TLB miss), the page number is used to index the process page table. TLB first checks if the page is already in main memory, if not in main memory a page fault is issued then the TLB is updated to include the new page entry.



Steps in TLB hit:

1. CPU generates virtual address.
2. It is checked in TLB (present).
3. Corresponding frame number is retrieved, which now tells where in the main memory page lies.

Steps in Page miss:

1. CPU generates virtual address.
2. It is checked in TLB (not present).
3. Now the page number is matched to page table residing in main memory (assuming page table contains all PTE).
4. Corresponding frame number is retrieved, which now tells where in the main memory page lies.
5. The TLB is updated with new PTE (if space is not there, one of the replacement technique comes into picture i.e either FIFO, LRU or MFU etc).

Effective memory access time(EMAT) : TLB is used to reduce effective memory access time as it is a high speed associative cache.

EMAT = h*(c+m) + (1-h)*(c+2m)

where, h = hit ratio of TLB

m = Memory access time

c = TLB access time