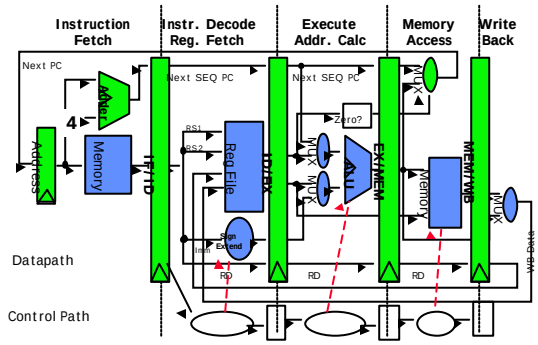
CS252
Graduate Computer Architecture
Lecture 2

Pipelining, Caching, and Benchmarks

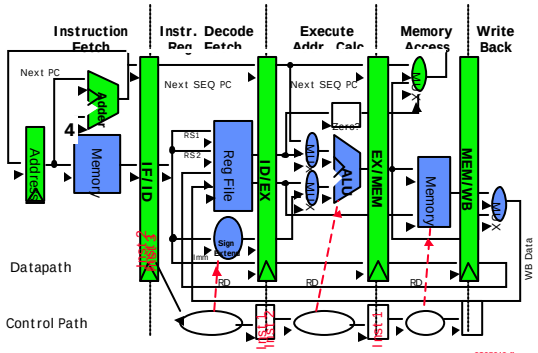
January 24, 2002
Prof. David E Culler
Computer Science 252
Spring 2002

©University of California, Berkeley

5 Steps of MIPS Datapath

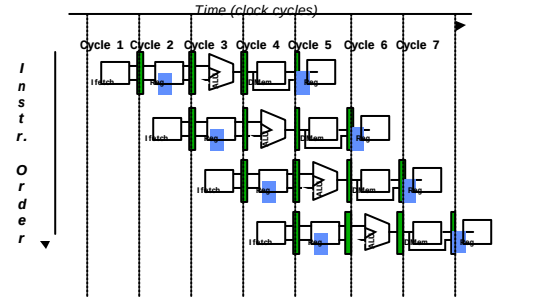


5 Steps of MIPS Datapath



Review: Visualizing Pipelining

Figure 3.3, Page 133 , CA:AQA 2e

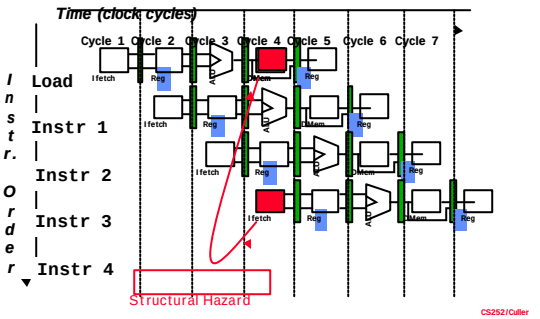


Limits to pipelining

- **Hazards:** circumstances that would cause incorrect execution if next instruction were launched
  - Structural hazards: Attempting to use the same hardware to do two different things at the same time
  - Data hazards: Instruction depends on result of prior instruction still in the pipeline
  - Control hazards: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps).

Example: One Memory Port/Structural Hazard

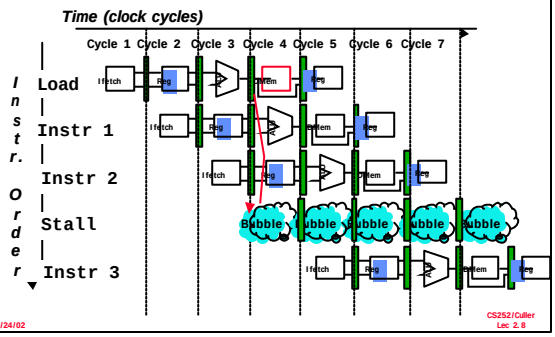
Figure 3.6, Page 142 , CA:AQA 2e



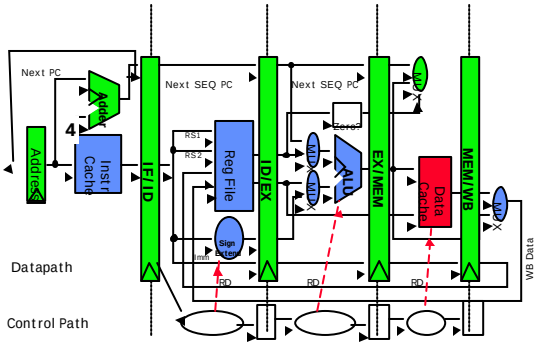
Resolving structural hazards

- Defn: attempt to use same hardware for two different things at the same time
- Solution 1: Wait
  - ⇒must detect the hazard
  - ⇒must have mechanism to stall
- Solution 2: Throw more hardware at the problem

Detecting and Resolving Structural Hazard



Eliminating Structural Hazards at Design Time

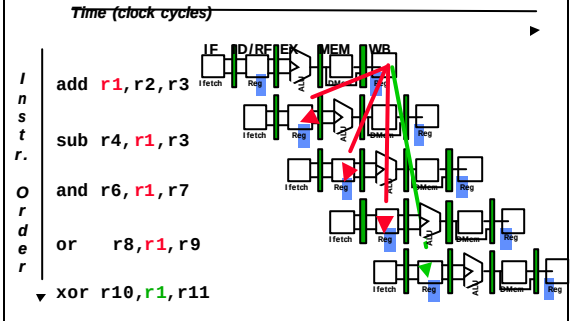


Role of Instruction Set Design in Structural Hazard Resolution

- Simple to determine the sequence of resources used by an instruction
  - opcode tells it all
- Uniformity in the resource usage
- Compare MIPS to IA32?
- MIPS approach => all instructions flow through same 5-stage pipeling

Data Hazards

Figure 3.9, page 147 , CA:AQA 2e



Three Generic Data Hazards

- Read After Write (RAW)
  Instr$_J$ tries to read operand before Instr$_I$ writes it

```
I: add r1,r2,r3
J: sub r4,r1,r3
```

- Caused by a "Data Dependence" (in compiler nomenclature). This hazard results from an actual need for communication.

Three Generic Data Hazards

- **Write After Read (WAR)**
  Instr$_J$ writes operand ***before*** Instr$_I$ reads it

```
I: sub r4,r1,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Reads are always in stage 2, and
  - Writes are always in stage 5

Three Generic Data Hazards

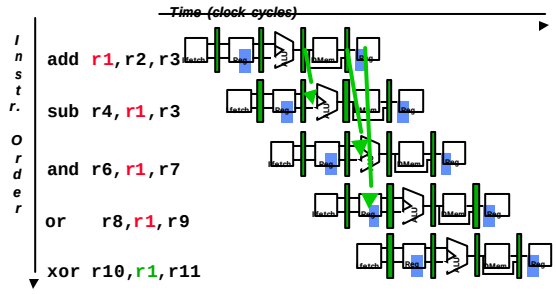
- **Write After Write (WAW)**
  Instr$_J$ writes operand ***before*** Instr$_I$ writes it.

```
I: sub r1,r4,r3
J: add r1,r2,r3
K: mul r6,r1,r7
```

- Called an "output dependence" by compiler writers This also results from the reuse of name "r1".
- Can't happen in MIPS 5 stage pipeline because:
  - All instructions take 5 stages, and
  - Writes are always in stage 5
- Will see WAR and WAW in later more complicated pipes

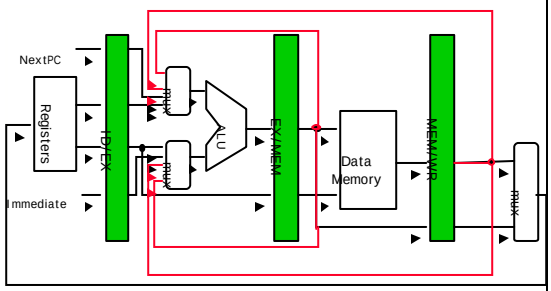
Forwarding to Avoid Data Hazard

Figure 3.10, Page 149 , CA:AQA 2e



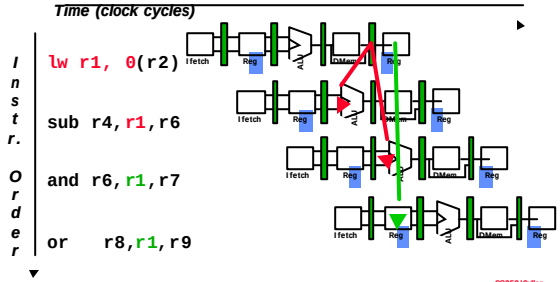
HW Change for Forwarding

Figure 3.20, Page 161, CA:AQA 2e



Data Hazard Even with Forwarding

Figure 3.12, Page 153 , CA:AQA 2e

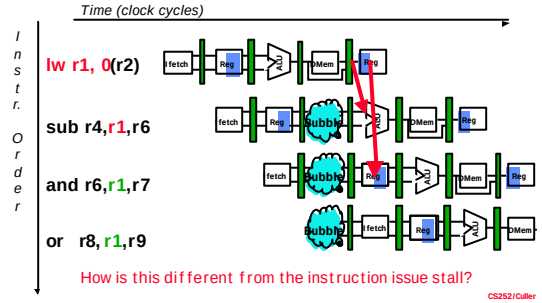


Resolving this load hazard

- Adding hardware? ... not
- Detection?
- Compilation techniques?

- What is the cost of load delays?

Resolving the Load Data Hazard



How is this different from the instruction issue stall?

Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

d = e – f;

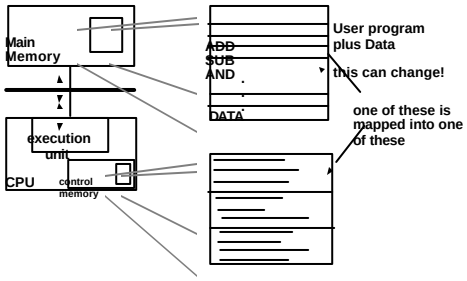
assuming a, b, c, d ,e, and f in memory.

Slow code:		Fast code:	
LW	Rb,b	LW	Rb,b
LW	Rc,c	LW	Rc,c
ADD	Ra,Rb,Rc	LW	Re,e
SW	a,Ra	ADD	Ra,Rb,Rc
LW	Re,e	LW	Rf,f
LW	Rf,f	SW	a,Ra
SUB	Rd,Re,Rf	SUB	Rd,Re,Rf
SW	d,Rd	SW	d,Rd

Instruction Set Connection

- What is exposed about this organizational hazard in the instruction set?
- k cycle delay?
  - bad, CPI is not part of ISA
- k instruction slot delay
  - load should not be followed by use of the value in the next k instructions
- Nothing, but code can reduce run-time delays
- MIPS did the transformation in the assembler

Historical Perspective: Microprogramming

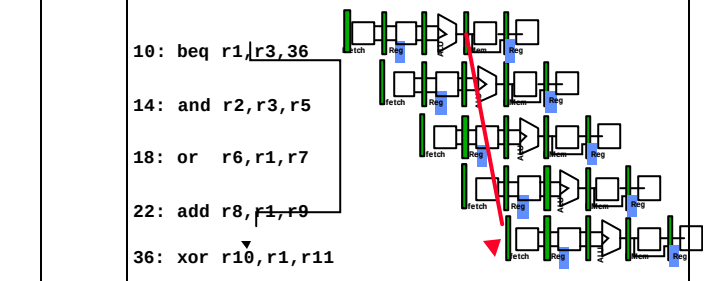


Supported complex instructions a sequence of simple micro-inst (RTs)
Pipelined micro-instruction processing, but very limited view.
Could not reorganize macroinstructions to enable pipelining

Administration

- Tuesday is Stack vs GPR Debate
  - Christine Chevalier — Dan Adkins
  - Yury Markovskiy — Mukund Seshadri
  - Yatish Patel — Manikandan Narayanan
  - Rachael Rubin — Hayley Iben
- Think about address size, code density, performance, compilation techniques, design complexity, program characteristics
- Prereq quiz afterwards
- Please register (form on page)

Control Hazard on Branches => Three Stage Stall

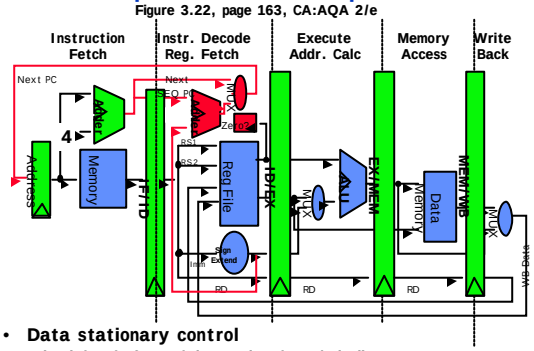


Example: Branch Stall Impact

- If 30% branch, Stall 3 cycles significant
- Two part solution:
  - Determine branch taken or not sooner, AND
  - Compute taken branch address earlier
- MIPS branch tests if register = 0 or ¹ 0
- MIPS Solution:
  - Move Zero test to ID/RF stage
  - Adder to calculate new PC in ID/RF stage
  - 1 clock cycle penalty for branch versus 3

Pipelined MIPS Datapath

Figure 3.22, page 163, CA:AQA 2/e



- Data stationary control
  - local decode for each instruction phase / pipeline stage

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear

#2: Predict Branch Not Taken
- Execute successor instructions in sequence
- "Squash" instructions in pipeline if branch actually taken
- Advantage of late pipeline state update
- 47% MIPS branches not taken on average
- PC+4 already calculated, so use it to get next instruction

#3: Predict Branch Taken
- 53% MIPS branches taken on average
- But haven't calculated branch target address in MIPS
  - » MIPS still incurs 1 cycle branch penalty
  - » Other machines: branch target known before outcome

Four Branch Hazard Alternatives

#4: Delayed Branch
- Define branch to take place AFTER a following instruction

```
branch instruction
  sequential successor1
  sequential successor2
  ........
  sequential successorn
........
branch target if taken
```

Branch delay of length *n*

- 1 slot delay allows proper decision and branch target address in 5 stage pipeline
- MIPS uses this

Delayed Branch

- Where to get instructions to fill branch delay slot?
  - Before branch instruction
  - From the target address: only valuable when branch taken
  - From fall through: only valuable when branch not taken
  - Canceling branches allow more slots to be filled
- Compiler effectiveness for single branch delay slot:
  - Fills about 60% of branch delay slots
  - About 80% of instructions executed in branch delay slots useful in computation
  - About 50% (60% x 80%) of slots usefully filled
- Delayed Branch downside: 7-8 stage pipelines, multiple instructions issued per clock (superscalar)

Recall: Speed Up Equation for Pipelining

$$CPI_{pipelined} = \text{Ideal CPI} + \text{Average Stall cycles per Inst}$$

$$\text{Speedup} = \frac{\text{Ideal CPI} \times \text{Pipeline depth}}{\text{Ideal CPI} + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipeline d}}}{\text{Cycle Time}_{\text{pipelined}}}$$

For simple RISC pipeline, CPI = 1:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipeline d}}}{\text{Cycle Time}_{\text{pipelined}}}$$

Example: Evaluating Branch Alternatives

$$\text{Pipeline speedup} = \frac{\text{Pipeline depth}}{1 + \text{Branch frequency} \times \text{Branch penalty}}$$

Assume:

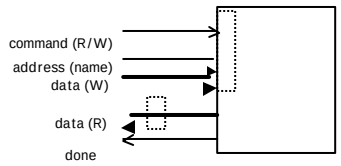
Conditional & Unconditional = 14%, 65% change PC

Scheduling scheme	*Branch penalty*	*CPI*	*speedup v. stall*
Stall pipeline	3	1.42	1.0
Predict taken	1	1.14	1.26
Predict not taken	1	1.09	1.29
Delayed branch	0.5	1.07	1.31

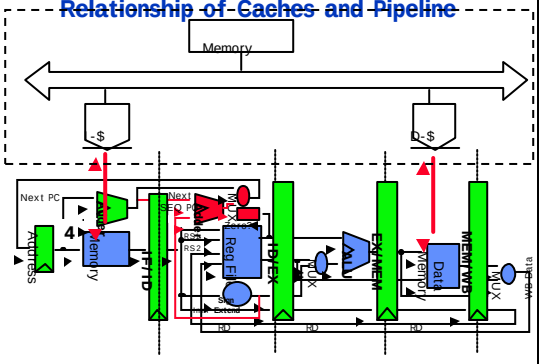
Questions?

The Memory Abstraction

- Association of <name, value> pairs
  - typically named as byte addresses
  - often values aligned on multiples of size
- Sequence of Reads and Writes
- Write binds a value to an address
- Read of addr returns most recently written value bound to that address



Relationship of Caches and Pipeline



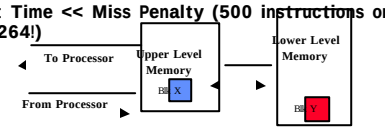
Example: Dual-port vs. Single-port

- Machine A: Dual ported memory
- Machine B: Single ported memory, but its pipelined implementation has a 1.05 times faster clock rate
- Ideal CPI = 1 for both
- Loads are 40% of instructions executed
  - Speedup(enhancement) = Time w/o enhancement / Time w/
  - Speedup(B) = Time(A) / Time(B)
    = CPI(A)xCT(A) / CPI(B)xCT(B)
    = 1 / (1.4 x 1/1.05) = 0.75

Machine A is 1.33 times faster

Memory Hierarchy: Terminology

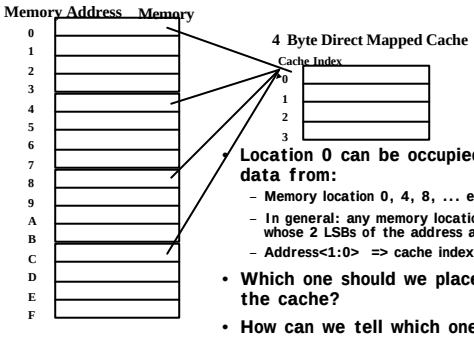
- Hit: data appears in some block in the upper level (example: Block X)
  - Hit Rate: the fraction of memory access found in the upper level
  - Hit Time: Time to access the upper level which consists of RAM access time + Time to determine hit/miss
- Miss: data needs to be retrieve from a block in the lower level (Block Y)
  - Miss Rate = 1 - (Hit Rate)
  - Miss Penalty: Time to replace a block in the upper level + Time to deliver the block the processor
- Hit Time << Miss Penalty (500 instructions on 21264!)



4 Questions for Memory Hierarchy

- Q1: Where can a block be placed in the upper level? *(Block placement)*
- Q2: How is a block found if it is in the upper level? *(Block identification)*
- Q3: Which block should be replaced on a miss? *(Block replacement)*
- Q4: What happens on a write? *(Write strategy)*

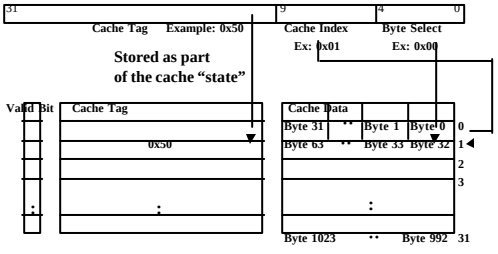
Simplest Cache: Direct Mapped



- Location 0 can be occupied by data from:
  - Memory location 0, 4, 8, ... etc.
  - In general: any memory location whose 2 LSBs of the address are 0s
  - Address<1:0> => cache index
- Which one should we place in the cache?
- How can we tell which one is in the cache?

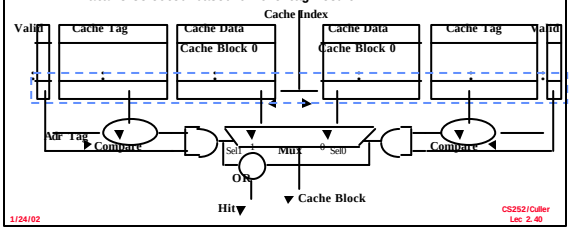
1 KB Direct Mapped Cache, 32B blocks

- For a 2 ** N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2 ** M)



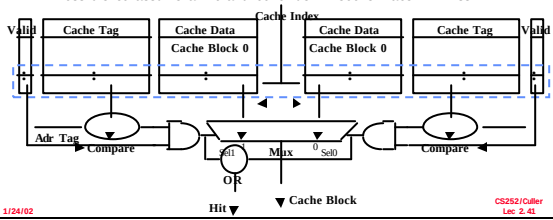
Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operates in parallel (N typically 2 to 4)
- Example: Two-way set associative cache
  - Cache Index selects a "set" from the cache
  - The two tags in the set are compared in parallel
  - Data is selected based on the tag result



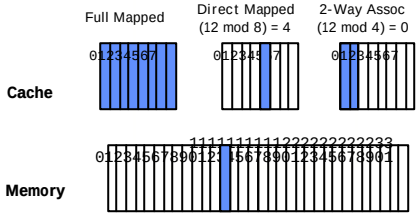
Disadvantage of Set Associative Cache

- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.



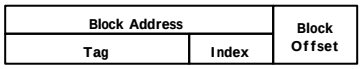
Q1: Where can a block be placed in the upper level?

- Block 12 placed in 8 block cache:
  - Fully associative, direct mapped, 2-way set associative
  - S.A. Mapping = Block Number Modulo Number Sets



Q2: How is a block found if it is in the upper level?

- Tag on each block
  - No need to check index or block offset
- Increasing associativity shrinks index, → expands tag →



Q3: Which block should be replaced on a miss?

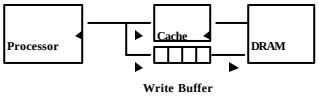
- Easy for Direct Mapped
- Set Associative or Fully Associative:
  - Random
  - LRU (Least Recently Used)

Assoc:	2-way		4-way		8-way	
Size	LRU	Ran	LRU	Ran	LRU	Ran
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Q4: What happens on a write?

- ***Write through***—The information is written to both the block in the cache and to the block in the lower-level memory.
- ***Write back***—The information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.
  - is block clean or dirty?
- Pros and Cons of each?
  - WT: read misses cannot result in writes
  - WB: no repeated writes to same location
- WT always combined with write buffers so that don't wait for lower level memory

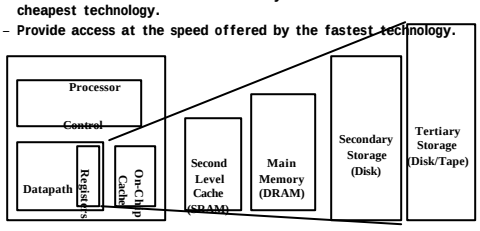
Write Buffer for Write Through



- A Write Buffer is needed between the Cache and Memory
  - Processor: writes data into the cache and the write buffer
  - Memory controller: write contents of the buffer to memory
- Write buffer is just a FIFO:
  - Typical number of entries: 4
  - Works fine if: Store frequency (w.r.t. time) << 1 / DRAM write cycle
- Memory system design:
  - Store frequency (w.r.t. time) -> 1 / DRAM write cycle
  - Write buffer saturation

A Modern Memory Hierarchy

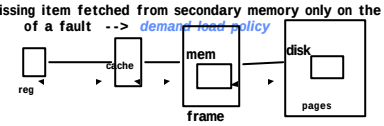
- By taking advantage of the principle of locality:
  - Present the user with as much memory as is available in the cheapest technology.
  - Provide access at the speed offered by the fastest technology.



Speed (ns):	1s	10s	100s	10,000,000s (10s ms)	10,000,000,000s (10s sec)
Size (bytes):	100s	Ks	Ms	Gs	Ts

Basic Issues in VM System Design

- size of information blocks that are transferred from secondary to main storage (M)
- block of information brought into M, and M is full, then some region of M must be released to make room for the new block --> *replacement policy*
- which region of M is to hold the new block --> *placement policy*
- missing item fetched from secondary memory only on the occurrence of a fault --> *demand load policy*



Paging Organization

virtual and physical address space partitioned into blocks of equal size

page frames

pages

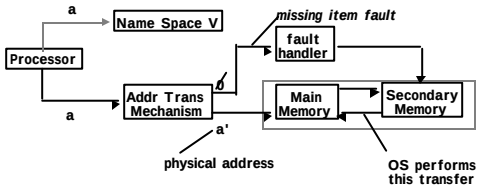
Address Map

V = {0, 1, . . . , n - 1} virtual address space $n > m$
M = {0, 1, . . . , m - 1} physical address space

MAP: V --> M U {∅} address mapping function

MAP(a) = a' if data at virtual address a is present in physical address a' and a' in M

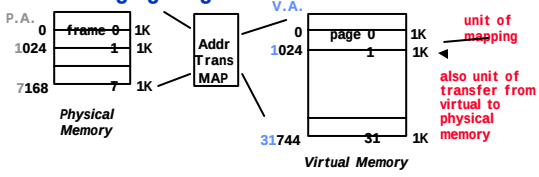
= ∅ if data at virtual address a is not present in M

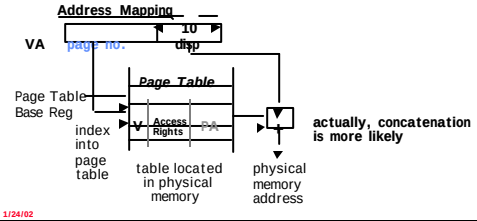


Implications of Virtual Memory for Pipeline design

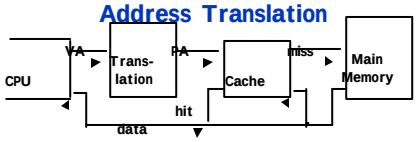
- Fault?
- Address translation?

Paging Organization





Address Translation



- Page table is a large data structure in memory
- Two memory accesses for every load, store, or instruction fetch!!!
- Virtually addressed cache?
  - synonym problem
- Cache the address translations?

TLBs

A way to speed up translation is to use a special cache of recently used page table entries -- this has many names, but the most frequently used is Translation Lookaside Buffer or TLB

Virtual Address	Physical Address	Dirty	Ref	Valid	Access

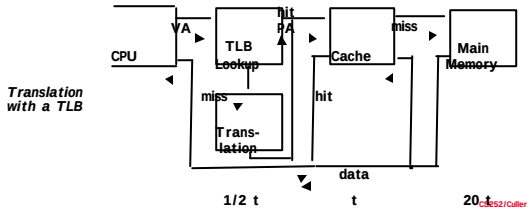
Really just a cache on the page table mappings

TLB access time comparable to cache access time (much less than main memory access time)

Translation Look-Aside Buffers

Just like any other cache, the TLB can be organized as fully associative, set associative, or direct mapped

TLBs are usually small, typically not more than 128 - 256 entries even on high end machines. This permits fully associative lookup on these machines. Most mid-range machines use small n-way set associative organizations.



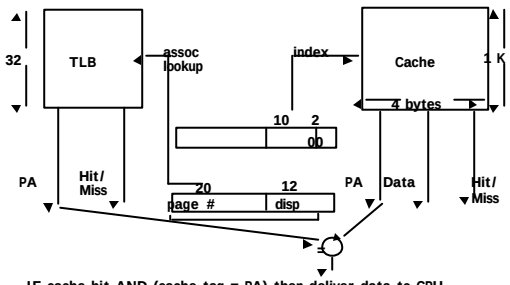
Reducing Translation Time

Machines with TLBs go one step further to reduce # cycles/cache access

They overlap the cache access with the TLB access:

high order bits of the VA are used to look in the TLB while low order bits are used as index into cache

Overlapped Cache & TLB Access



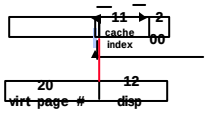
IF cache hit AND (cache tag = PA) then deliver data to CPU
ELSE IF [cache miss OR (cache tag = PA)] and TLB hit THEN access memory with the PA from the TLB
ELSE do standard VA translation

Problems With Overlapped TLB Access

Overlapped access only works as long as the address bits used to index into the cache *do not change* as the result of VA translation

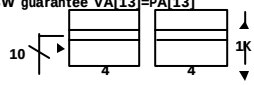
This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache

Example: suppose everything the same except that the cache is increased to 8 K bytes instead of 4 K:



This bit is changed by VA translation, but is needed for cache lookup

Solutions:
go to 8K byte page sizes;
go to 2 way set associative cache; or
SW guarantee VA[13]=PA[13]



2 way set assoc cache

Another Word on Performance

SPEC: System Performance Evaluation Cooperative

- First Round 1989
  - 10 programs yielding a single number ("SPECmarks")
- Second Round 1992
  - SPECInt92 (6 integer programs) and SPECfp92 (14 floating point programs)
    - » Compiler Flags unlimited. March 93 of DEC 4000 Model 610:
      `spice: unix.c:/def=(sysv,has_bcopy,"bcopy(a,b,c)= memcpy(b,a,c)"`
      `wave5: /ali=(all,dcom=nat)/ag=a/ur=4/ur=200`
      `nasa7: /norecu/ag=a/ur=4/ur2=200/lc=blas`
- Third Round 1995
  - new set of programs: SPECint95 (8 integer programs) and SPECfp95 (10 floating point)
  - "benchmarks useful for 3 years"
  - Single flag setting for all programs: SPECint_base95, SPECfp_base95

SPEC: System Performance Evaluation Cooperative

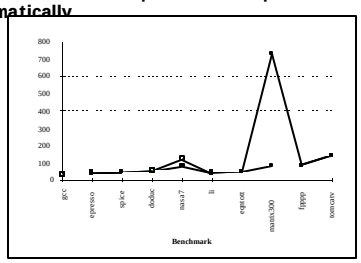
- Fourth Round 2000: SPEC CPU2000
  - 12 Integer
  - 14 Floating Point
  - 2 choices on compilation; "aggressive" (SPECint2000,SPECfp2000), "conservative" (SPECint_base2000,SPECfp_base); flags same for all programs, no more than 4 flags, same compiler for conservative, can change for aggressive
  - multiple data sets so that can train compiler if trying to collect data for input to compiler to improve optimization

How to Summarize Performance

- Arithmetic mean (weighted arithmetic mean) tracks execution time:
  $\Sigma(T_i)/n$ or $\Sigma(W_i*T_i)$
- Harmonic mean (weighted harmonic mean) of rates (e.g., MFLOPS) tracks execution time:
  $n/\Sigma(1/R_i)$ or $n/\Sigma(W_i/R_i)$
- Normalized execution time is handy for scaling performance (e.g., X times faster than SPARCstation 10)
- But do not take the arithmetic mean of normalized execution time, use the geometric mean:
  $(\Pi\ T_i / N_i)^{1/n}$

SPEC First Round

- One program: 99% of time in single line of code
- New front-end compiler could improve dramatically



Performance Evaluation

- "For better or worse, benchmarks shape a field"
- Good products created when have:
  - Good benchmarks
  - Good ways to summarize performance
- Given sales is a function in part of performance relative to competition, investment in improving product as reported by performance summary
- If benchmarks/summary inadequate, then choose between improving product for real programs vs. improving product to get more sales;
  Sales almost always wins!
- Execution time is the measure of computer performance!

Summary #1/4: Pipelining & Performance

- Just overlap tasks; easy if tasks are independent
- Speed Up £ Pipeline Depth; if ideal CPI is 1, then:

$$\text{Speedup} = \frac{\text{Pipeline depth}}{1 + \text{Pipeline stall CPI}} \times \frac{\text{Cycle Time}_{\text{unpipelined}}}{\text{Cycle Time}_{\text{pipelined}}}$$

- Hazards limit performance on computers:
  - Structural: need more HW resources
  - Data (RAW,WAR,WAW): need forwarding, compiler scheduling
  - Control: delayed branch, prediction
- Time is measure of performance: latency or throughput
- CPI Law:

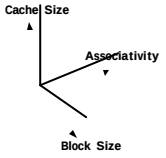
$$\text{CPU time} = \frac{\text{Seconds}}{\text{Program}} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Cycle}}$$

Summary #2/4: Caches

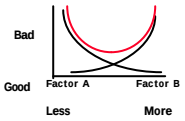
- The Principle of Locality:
  - Program access a relatively small portion of the address space at any instant of time.
    - » Temporal Locality: Locality in Time
    - » Spatial Locality: Locality in Space
- Three Major Categories of Cache Misses:
  - Compulsory Misses: sad facts of life. Example: cold start misses.
  - Capacity Misses: increase cache size
  - Conflict Misses: increase cache size and/or associativity.
- Write Policy:
  - Write Through: needs a write buffer.
  - Write Back: control can be complex
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): What does this mean to Compilers, Data structures, Algorithms?

Summary #3/4: The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back



- The optimal choice is a compromise
  - depends on access characteristics
    - » workload
    - » use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins



Review #4/4: TLB, Virtual Memory

- Caches, TLBs, Virtual Memory all understood by examining how they deal with 4 questions: 1) Where can block be placed? 2) How is block found? 3) What block is repalced on miss? 4) How are writes handled?
- Page tables map virtual address to physical address
- TLBs make virtual memory practical
  - Locality in data => locality in addresses of data, temporal and spatial
- TLB misses are significant in processor performance
  - funny times, as most systems can't access all of 2nd level cache without TLB misses!
- Today VM allows many processes to share single memory without having to swap all processes to disk; today VM protection is more important than memory hierarchy