CMSC 411
Computer Systems Architecture
Lecture 13
Virtual Memory, cont. & Instruction Level Parallelism

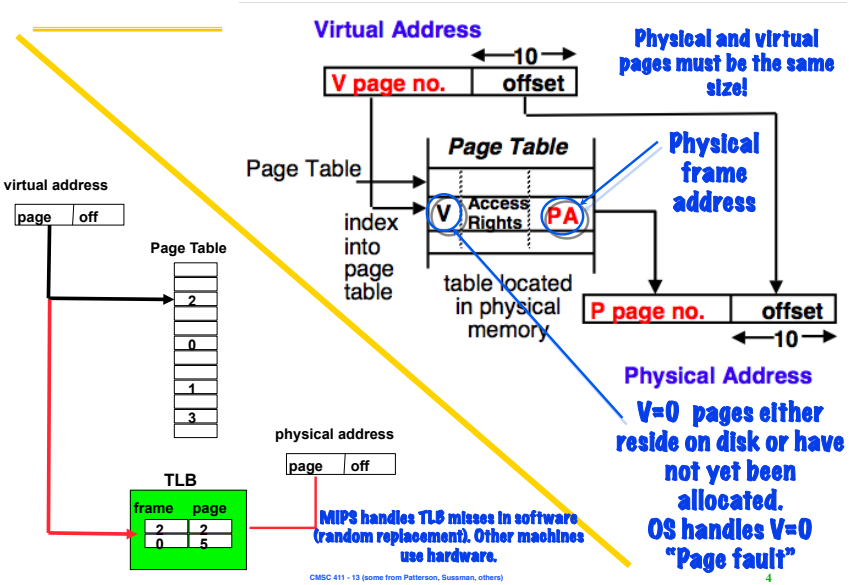
Alan Sussman
als@cs.umd.edu

Administrivia

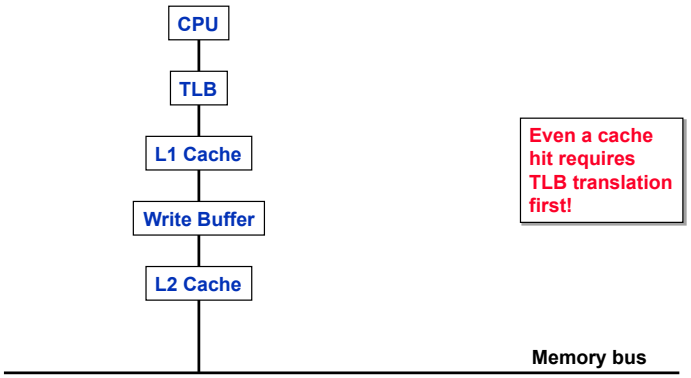
- HW #3, on memory hierarchy,posted and due next Tuesday, April 1
- Start reading Chapter 3 of H&P
- First exam returned today
  - answers posted and grades visible in grades server
  - Mean: 70 StDev: 14
- Project #1 due tomorrow

Paging

The TLB caches page table entries

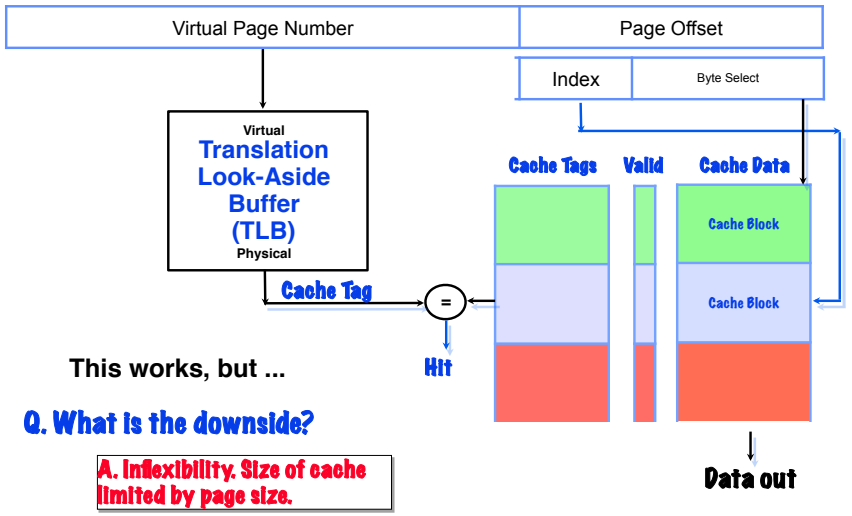


Common Organization



Even a cache hit requires TLB translation first!

Can TLB and caching be overlapped?



This works, but ...

Q. What is the downside?

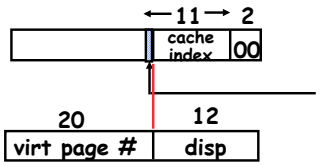
A. Inflexibility. Size of cache limited by page size.

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

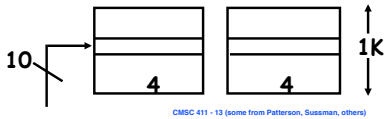
This usually limits things to small caches, large page sizes, or high n-way set associative caches if you want a large cache

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



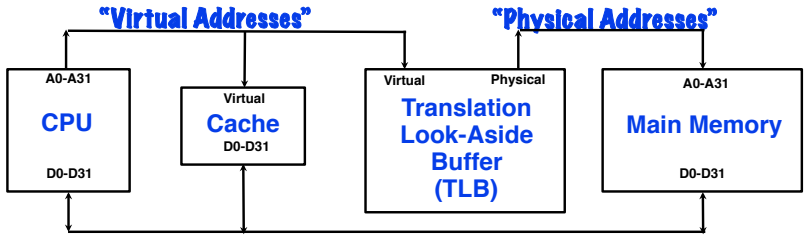
This bit is changed by VA translation, but is needed for cache lookup

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

Use virtual addresses for cache?



Only use TLB on a cache miss !

Downside: a subtle, fatal problem. What is it?

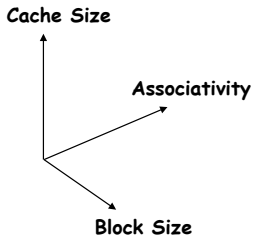
A. Synonym problem. If two address spaces share a physical frame, data may be in cache twice. Maintaining consistency is a nightmare.

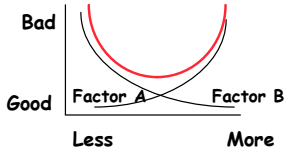
Paging vs. segmentation – Fig. B.22

	Page	Segment
Words per address	One	Two (segment/offset)
Programmer visible?	Invisible to app programmer	May be visible to app programmer
Replacing a block	Trivial (all blocks same size)	Hard (must find contiguous, variable-sized chunk)
Memory use inefficiency	Internal fragmentation (within page)	External fragmentation (in unused memory)
Efficient disk traffic	Yes (can adjust page size)	Not always (small segment problem)

Summary #1/3: The Cache Design Space

- Several interacting dimensions
  - cache size
  - block size
  - associativity
  - replacement policy
  - write-through vs write-back
  - write allocation
- The optimal choice is a compromise
  - depends on access characteristics
    - » workload
    - » use (I-cache, D-cache, TLB)
  - depends on technology / cost
- Simplicity often wins





Summary #2/3: Caches

- The Principle of Locality:
  - Program accesses a relatively small portion of the address space in any short interval 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. Nightmare Scenario: ping pong effect!
- Write Policy: Write Through vs. Write Back
- Today CPU time is a function of (ops, cache misses) vs. just f(ops): affects Compilers, Data structures, and Algorithms

Summary #3/3: TLB, Virtual Memory

- Page tables map virtual address to physical address
- TLBs are important for fast translation
- TLB misses are significant in processor performance
  - funny times, as most systems can't access all of 2nd level cache without TLB misses!
- 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 replaced on miss?
  4) How are writes handled?
- Today VM allows many processes to share single memory without having to swap all processes to disk; today VM protection is maybe even more important than memory hierarchy benefits, but computers still insecure

INSTRUCTION-LEVEL PARALLELISM

Outline

- **ILP**
- **Compiler techniques to increase ILP**
- **Loop Unrolling**
- **Static Branch Prediction**
- **Dynamic Branch Prediction**
- **Overcoming Data Hazards with Dynamic Scheduling**
- **(Start) Tomasulo Algorithm**
- **Conclusion**

Recall from Pipelining

- Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls
  - Ideal pipeline CPI: measure of the maximum performance attainable by the implementation
  - Structural hazards: HW cannot support this combination of instructions
  - 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)

Instruction Level Parallelism

- Instruction-Level Parallelism (ILP): overlap the execution of instructions to improve performance
- 2 approaches to exploit ILP:
  1) Rely on hardware to help discover and exploit the parallelism dynamically (e.g., Pentium 4/core2/i3/i5/i7, AMD Opteron, IBM Power) , and
  2) Rely on software technology to find parallelism, statically at compile-time (e.g., Itanium 2/IA-64)

Instruction-Level Parallelism (ILP)

- Basic Block (BB) ILP is quite small
  - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
  - average dynamic branch frequency 15% to 25%
    => 4 to 7 instructions execute between a pair of branches
  - Plus instructions in BB likely to depend on each other
- Need ILP *across* multiple basic blocks
- Simplest: loop-level parallelism to exploit parallelism among iterations of a loop. E.g.,

```
for (i=1; i<=1000; i=i+1)
        x[i] = x[i] + y[i];
```


Loop-Level Parallelism

- Exploit loop-level parallelism by "unrolling loop" either by
  1. dynamic via branch prediction or
  2. static via loop unrolling by compiler

  (Another way is vectors, to be covered later)
- Determining dependences *critical*
- If 2 instructions are
  - parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
  - dependent, they are not parallel and must be executed in order, although they may often be partially overlapped

Data Dependence and Hazards

- $Instr_J$ is data dependent (aka true dependence) on $Instr_I$:
  1. $Instr_J$ tries to read operand before $Instr_I$ writes it

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

  2. or $Instr_J$ is data dependent on $Instr_K$ which is dependent on $Instr_I$
- If two instructions are data dependent, *they cannot execute simultaneously* or be completely overlapped
- Data dependence in instruction sequence
  ⇒ data dependence in source code

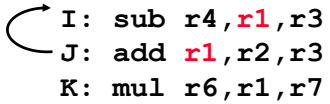
  ⇒ effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, that's a Read After Write (RAW) hazard

ILP and Data Dependencies,Hazards

- HW/SW must preserve *illusion* of program order: order instructions would execute in if executed sequentially as determined by original source program
  - dependences are a property of programs
- Presence of dependence indicates potential for a hazard, but
  - actual hazard and length of any stall is property of the pipeline
- Importance of the data dependencies
  1) indicates the possibility of a hazard
  2) determines order in which results must be calculated
  3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Name Dependence #1: Anti-dependence

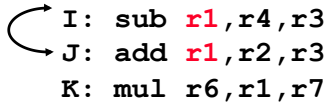
- Name dependence: when 2 instructions use same register or memory location, called a name, but no flow of data between the instructions associated with that name; 2 versions of name dependence
- $Instr_J$ writes operand *before* $Instr_I$ reads it



  Called an "anti-dependence" by compiler writers.
  This results from reuse of the name "r1"
- If anti-dependence caused a hazard in the pipeline, that's a Write After Read (WAR) hazard

Name Dependence #2: Output dependence

- $Instr_J$ writes operand *before* $Instr_I$ writes it.



- Called an "output dependence" by compiler writers
  This also results from the reuse of name "r1"
- If anti-dependence caused a hazard in the pipeline, that's a Write After Write (WAW) hazard
- Instructions involved in a name dependence can execute simultaneously if name used in instructions is changed so instructions do not conflict
  - Register renaming resolves name dependence for regs
  - Either by compiler or by HW