332
Advanced Computer Architecture Chapter 1

Introduction and review of Pipelines, Performance, Caches, and Virtual Memory

January 2009
Paul H J Kelly

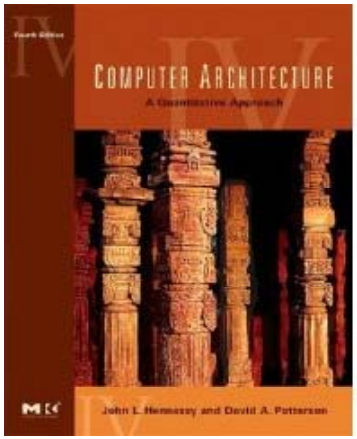
These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach (4th ed), and on the lecture slides of David Patterson's Berkeley course (CS252)*

Course materials online at
http://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture.html

Pre-requisites

- This a third-level computer architecture course
- The usual path would be to take this course after following a course based on a textbook like "Computer Organization and Design" (Patterson and Hennessy, Morgan Kaufmann)
- This course is based on the more advanced book by the same authors (see next slide)
- You *can* take this course provided you're prepared to catch up if necessary
  - Read chapters 1 to 8 of "Computer Organization and Design" (COD) if this material is new to you
  - If you have studied computer architecture before, make sure COD Chapters 2, 6, 7 are familiar
  - See also "Appendix A Pipelining: Basic and Intermediate Concepts" of course textbook
- FAST review today of Pipelining, Performance, Caches, and Virtual Memory

This is a textbook-based course



- Computer Architecture: A Quantitative Approach (4th Edition)

  John L. Hennessy, David A. Patterson

  - ~580 pages. Morgan Kaufmann (2007); ISBN: 978-0-12-370490-0 with substantial additional material on CD
  - Price: £ 37.99 (Amazon.co.uk, Nov 2006
  - Publisher's companion web site:
    - http://textbooks.elsevier.com/0123704901/
  - Textbook includes some vital introductory material as appendices:
    - Appendix A: tutorial on pipelining (read it NOW)
    - Appendix C: tutorial on caching (read it NOW)
  - Further appendices (some in book, some in CD) cover more advanced material (some very relevant to parts of the course), eg
    - Networks
    - Parallel applications
    - Implementing Coherence Protocols
    - Embedded systems
    - VLIW
    - Computer arithmetic (esp floating point)
    - Historical perspectives

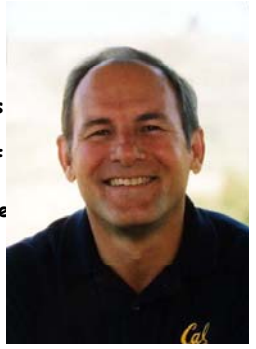
Who are these guys anyway and why should I read their book?

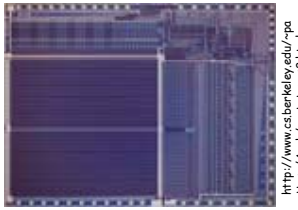
- John Hennessy:
  - Founder, MIPS Computer Systems
  - President, Stanford University

  (previous president: Condoleezza Rice)

RAID-I (1989) consisted of a Sun 4/280 workstation with 128 MB of DRAM, four dual-string SCSI controllers, 28 5.25-inch SCSI disks and specialized disk striping software.

- David Patterson
  - Leader, Berkeley RISC project (led to Sun's SPARC)
  - RAID (redundant arrays of inexpensive disks)
  - Professor, University of California, Berkeley
  - Current president of the ACM
  - Served on Information Technology Advisory Committee to the US President





http://www.cs.berkeley.edu/~pattrsn/Arch/prototypes2.html

RISC-I (1982) Contains 44,420 transistors, fabbed in 5 micron NMOS, with a die area of 77 mm², ran at 1 MHz. This chip is probably the first VLSI RISC.

Administration details

- **Course web site:**
  - http://www.doc.ic.ac.uk/~phjk/AdvancedCompArchitecture.html
  - **Course textbook: H&P 4th ed**
    - **Read Appendix *A right away***

- ***Background for 2008 context…***
  - ***See Workshop on Trends in Computing Performance***
    http://www7.nationalacademies.org/CSTB/project_computing-performance_workshop.html

Course organisation

- Lecturer:
  - **Paul Kelly** – Leader, Software Performance Optimisation research group
- Tutorial helper:
  - **Anton Lokhmotov** – postdoctoral researcher: PhD from Cambridge on optimisation and algorithms for SIMD. Industry experience with Broadcom (VLIW hardware), Clearspeed (massively-multicore SIMD hardware), Codeplay (compilers for games), ACE (compilers)
- 3 hours per week
- Nominally two hours of lectures, one hour of classroom tutorials
- We will use the time more flexibly
- Assessment:
  - Exam
    - For CS M.Eng. Class, exam will take place in last week of term
    - For everyone else, exam will take place early in the summer term
    - The goal of the course is to teach you how to think about computer architecture
    - The exam usually includes some architectural ideas not presented in the lectures
  - Coursework
    - You will be assigned a substantial, laboratory-based exercise
    - You will learn about performance tuning for computationally-intensive kernels
    - You will learn about using simulators, and experimentally evaluating hypotheses to understand system performance
    - You are encouraged to bring laptops to class to get started and get help during tutorials
- Please do not use computers for anything else during classes

Course overview (plan)

- **Ch1**
  - Review of pipelined, in-order processor architecture and simple cache structures
- **Ch2**
  - Caches in more depth
  - Software techniques to improve cache performance
  - Virtual memory
  - Benchmarking
  - Fab
- **Ch3**
  - Instruction-level parallelism
  - Dynamic scheduling, out-of-order
  - Register renaming
  - Speculative execution
  - Branch prediction
  - Limits to ILP
- **Ch4**
  - Compiler techniques – loop nest transformations
  - Loop parallelisation, interchange, tiling/blocking, skewing
- **Ch5**
  - Multithreading, hyperthreading, SMT
  - Static instruction scheduling
  - Software pipelining
  - EPIC/IA-64; instruction-set support for speculation and register renaming
- **Ch6**
  - GPUs, GPGPU, and manycore
- **Ch7**
  - Shared-memory multiprocessors
  - Cache coherency
  - Large-scale cache-coherency; ccNUMA. COMA

- **Lab-based coursework exercise:**
  - Simulation study
  - "challenge"
  - Using performance analysis tools
- **Exam:**
  - Partially based on recent processor architecture article, which we will study in advance (see past papers)

A "Typical" RISC

- 32-bit fixed format instruction (3 formats, see next slide)
- 32 32-bit general-purpose registers
  - (R0 contains zero, double-precision/long operands occupy a pair)
- Memory access only via load/store instructions
  - No instruction both accesses memory and does arithmetic
  - All arithmetic is done on registers
- 3-address, reg-reg arithmetic instruction
  - Subw r1,r2,r3 means r1 := r2-r3
  - registers identifiers always occupy same bits of instruction encoding
- Single addressing mode for load/store: base + displacement
  - ie register contents are added to constant from instruction word, and used as address, eg "lw R2,100(r1)" means "r2 := Mem[100+r1]"
  - no indirection
- Simple branch conditions
- Delayed branch

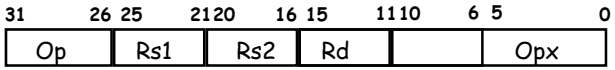
see: SPARC, MIPS, ARM, HP PA-Risc, DEC Alpha, IBM PowerPC, CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Not: Intel IA-32, IA-64 (?), Motorola 68000, DEC VAX, PDP-11, IBM 360/370

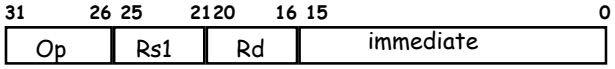
Eg: VAX matchc, IA32 scas instructions!

Example: MIPS (Note register location)

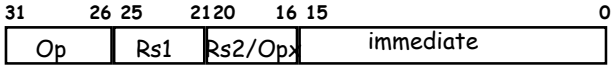
Register-Register



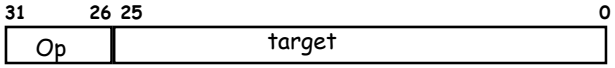
Register-Immediate



Branch



Jump / Call



Q: What is the largest signed immediate operand for "subw r1,r2,X"?
Q: What range of addresses can a conditional branch jump to?

So where do I find a MIPS processor?

- MIPS licensees shipped more than 350 million units during fiscal year 2007 (http://www.mips.com/company/about-us/milestones/)

A machine to execute these instructions

- To execute this instruction set we need a machine that fetches them and does what each instruction says
- A "universal" computing device – a simple digital circuit that, with the right code, can compute *anything*
- Something like:

```
Instr = Mem[PC]; PC+=4;

rs1 = Reg[Instr.rs1];
rs2 = Reg[Instr.rs2];
imm = SignExtend(Instr.imm);

Operand1 = if(Instr.op==BRANCH) then PC else rs1;
Operand2 = if(immediateOperand(Instr.op)) then imm else rs2;
res = ALU(Instr.op, Operand1, Operand2);

switch(Instr.op) {
case BRANCH:
  if (rs1==0) then PC=PC+imm; continue;
case STORE:
  Mem[res] = rs1; continue;
case LOAD:
  lmd = Mem[res];
}
Reg[Instr.rd] = if (Instr.op==LOAD) then lmd else res;
```


5 Steps of MIPS Datapath

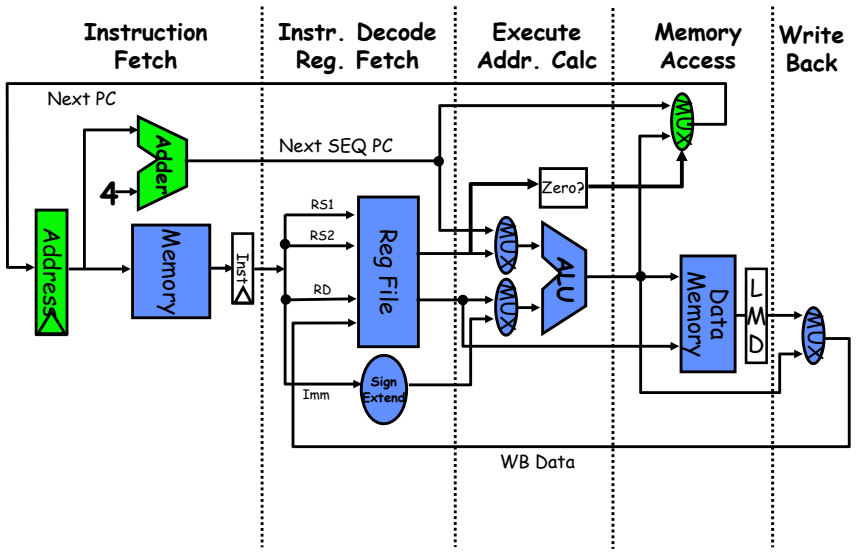
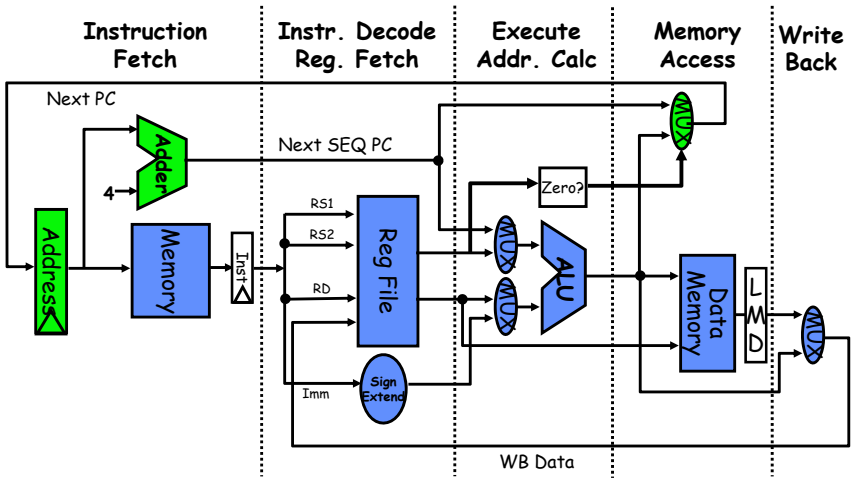


Figure 3.1, Page 130, CA:AQA 2e

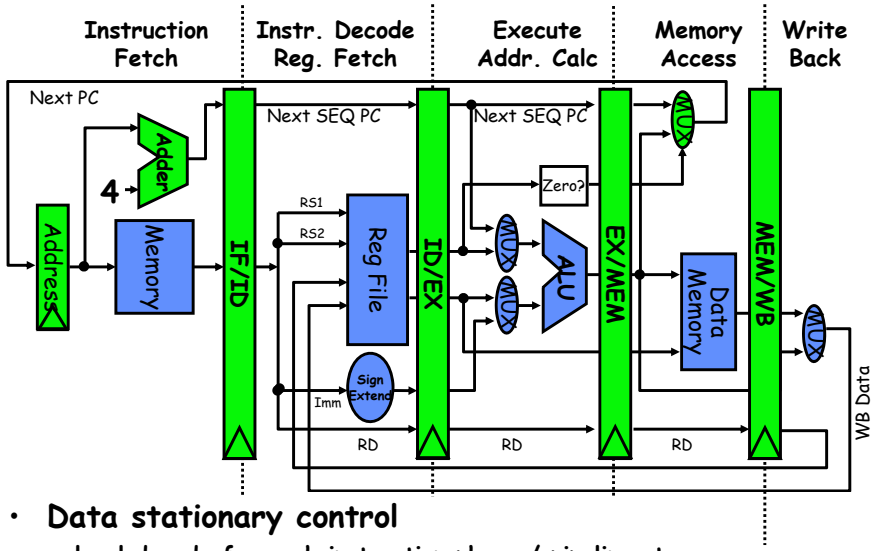
Pipelining the MIPS datapath



We will see more complex pipeline structures later.
For example, the Pentium 4 "Netburst" architecture has 31 stages.

Figure 3.1, Page 130, CA:AQA 2e

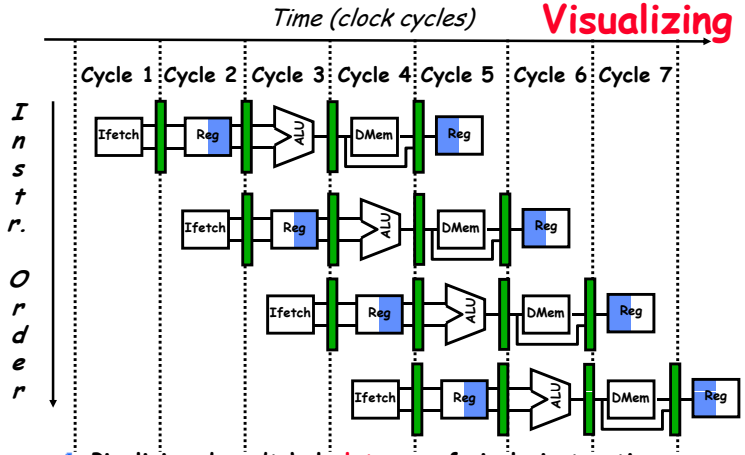
5-stage MIPS pipeline with pipeline buffers



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

Figure 3.4, Page 134 , CA:AQA 2e

Visualizing Pipelining



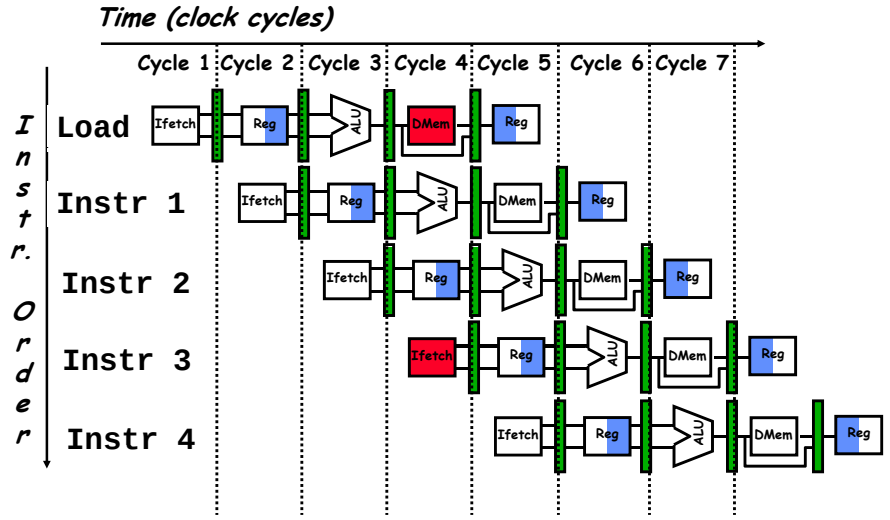
- Pipelining doesn't help latency of single instruction
  - it helps throughput of entire workload
- Pipeline rate limited by slowest pipeline stage
- Potential speedup = Number pipe stages
- Unbalanced lengths of pipe stages reduces speedup
- Time to "fill" pipeline and time to "drain" it reduces speedup
- Speedup comes from parallelism
  - For free - no new hardware

Figure 3.3, Page 133 , CA:AQA 2e

It's Not That Easy for Computers

- Limits to pipelining: Hazards prevent next instruction from executing during its designated clock cycle
  - 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).

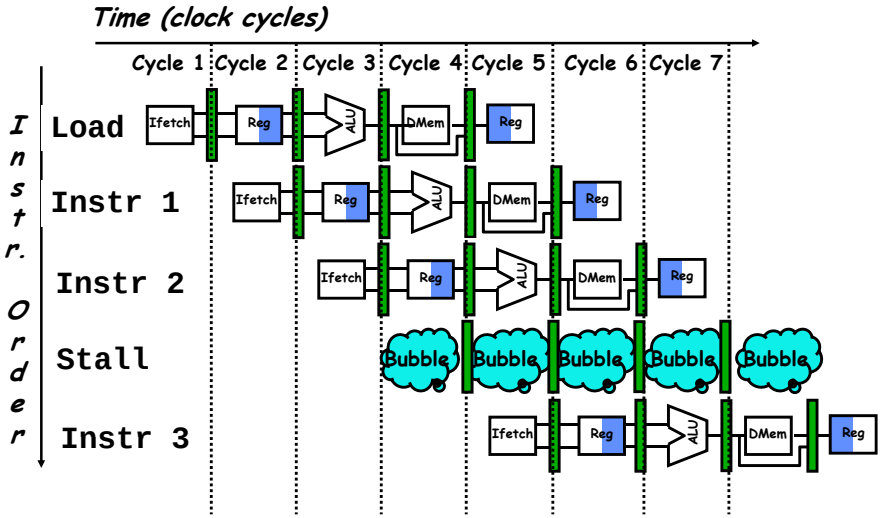
One Memory Port/Structural Hazards



- Eg if there is only one memory for both instructions and data
- Two different stages may need access at same time
- Example: IBM/Sony/Toshiba Cell processor

Figure 3.6, Page 142 , CA:AQA 2e

One Memory Port/Structural Hazards



- Instr 3 cannot be loaded in cycle 4
- ID stage has nothing to do in cycle 5
- EX stage has nothing to do in cycle 6, etc. "Bubble" propagates

Figure 3.7, Page 143 , CA:AQA 2e

Data Hazard on R1

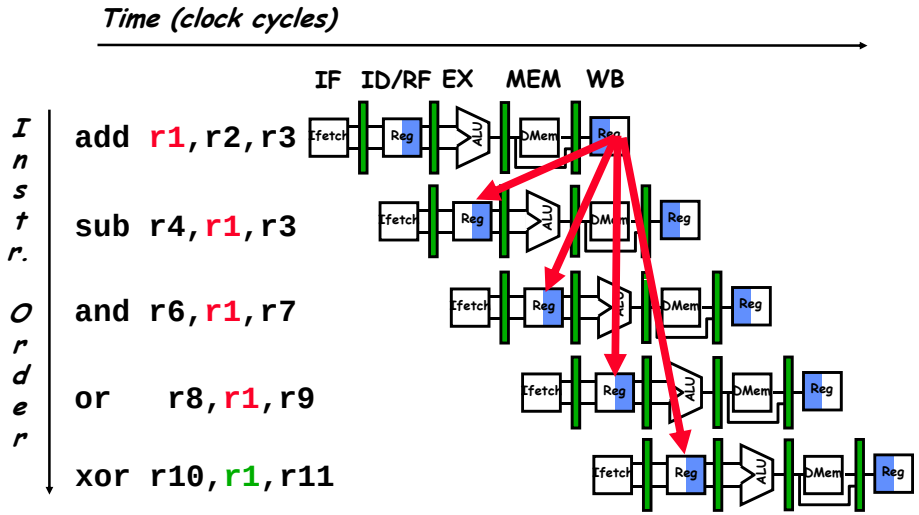


Figure 3.9, page 147 , CA:AQA 2e

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 "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

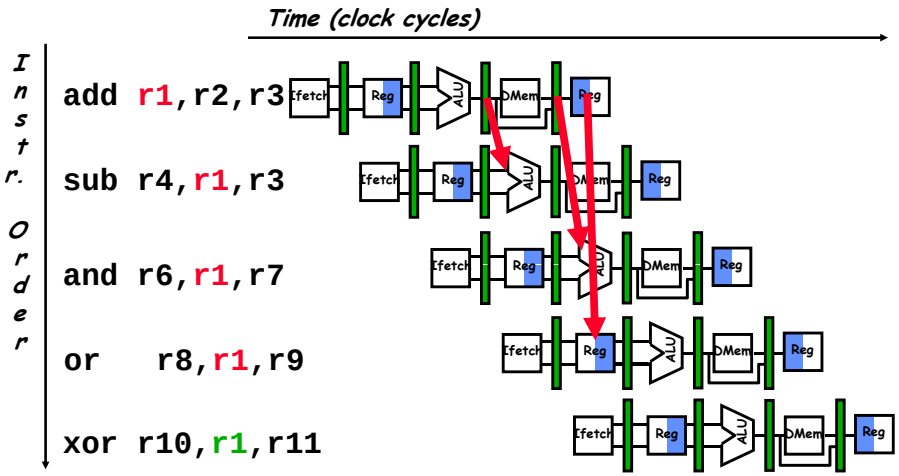
- Write After Write (WAW)
  Instr$_J$ writes operand *before* Instr$_I$ writes it.

```
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

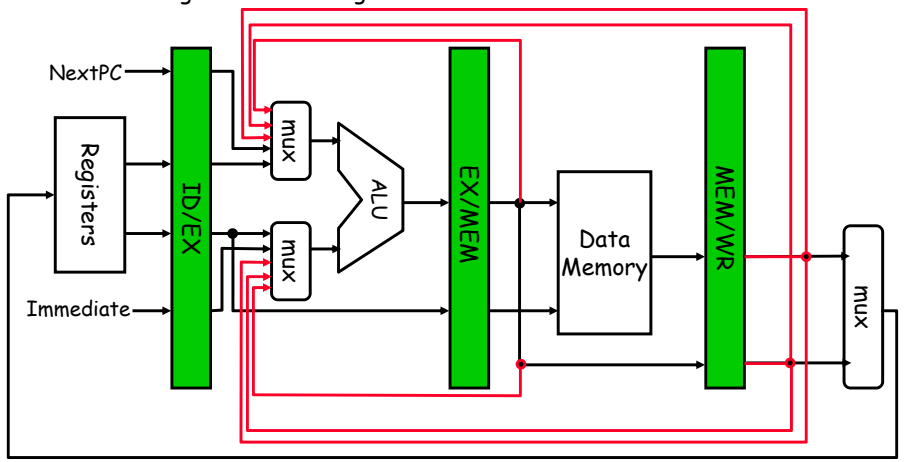
Figure 3.10, Page 149 , CA:AQA 2e



HW Change for Forwarding

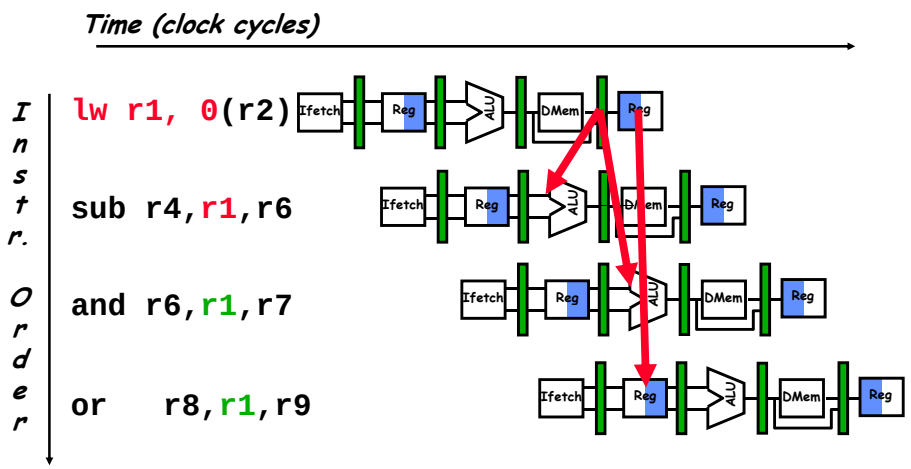
Figure 3.20, Page 161, CA:AQA 2e

- Add forwarding ("bypass") paths
- Add multiplexors to select where ALU operand should come from
- Determine mux control in ID stage
- If source register is the target of an instrn that will not WB in time



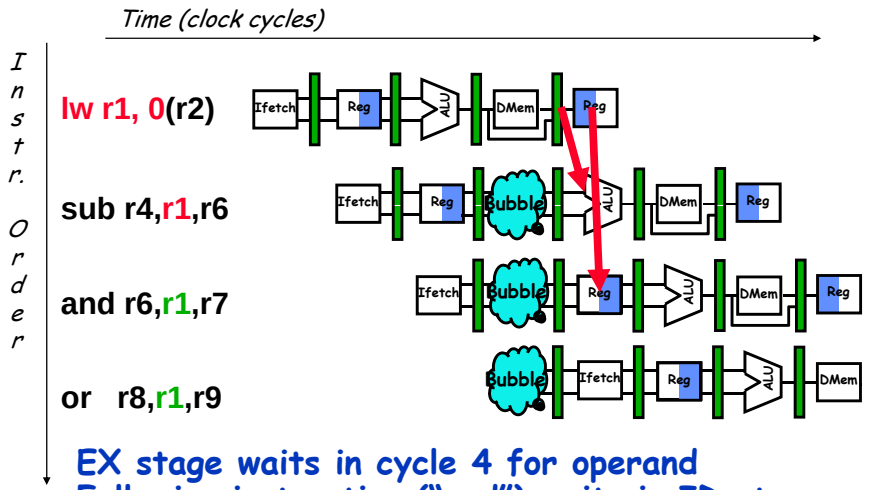
Data Hazard Even with Forwarding

Figure 3.12, Page 153 , CA:AQA 2e



Data Hazard Even with Forwarding

Figure 3.13, Page 154 , CA:AQA 2e



EX stage waits in cycle 4 for operand
Following instruction ("and") waits in ID stage
Missed instruction issue opportunity…

Software Scheduling to Avoid Load Hazards

Try producing fast code for

a = b + c;

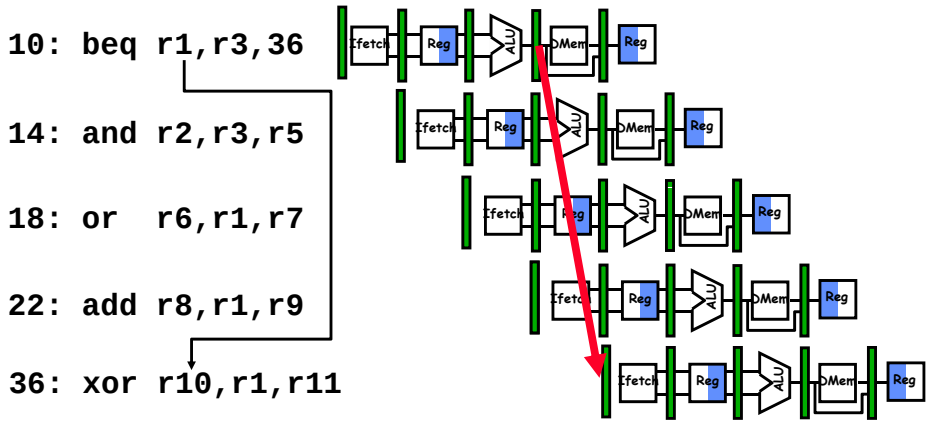
d = e – f;

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
STALL		LW	Re,e
ADD	Ra,Rb,Rc	ADD	Ra,Rb,Rb
SW	a,Ra		
LW	Re,e		
LW	Rf,f	LW	Rf,f
STALL		SW	a,Ra
SUB	Rd,Re,Rf	SUB	Rd,Re,Rf
SW	d,Rd	SW	d,Rd
10 cycles (2 stalls)		8 cycles (0 stalls)	

Show the stalls explicitly

Control Hazard on Branches Three Stage Stall



Pipelined MIPS Datapath with early branch determination

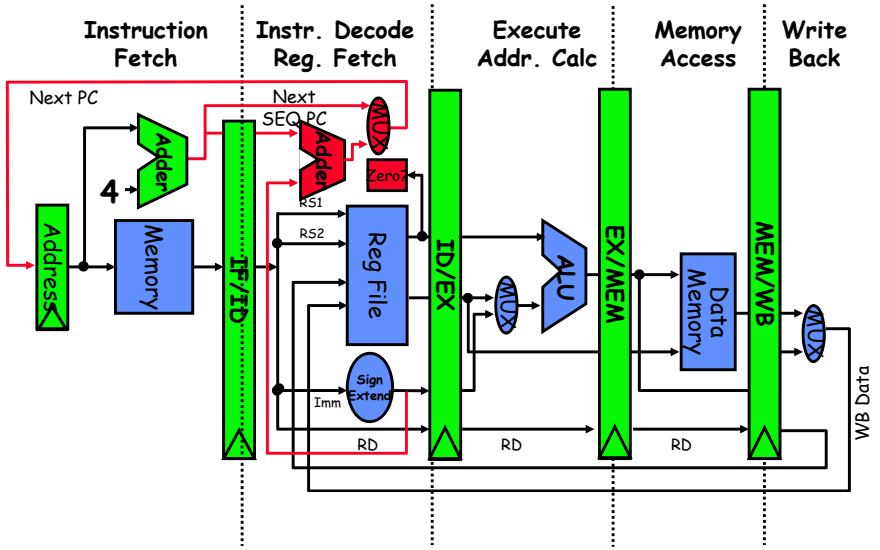


Figure 3.22, page 163, CA:AQA 2/e

Four Branch Hazard Alternatives

#1: Stall until branch direction is clear
(wasteful – the next instruction is being fetched during ID)

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

#3: Predict Branch Taken
- 53% MIPS branches are taken on average
- But in MIPS instruction set we haven't calculated branch target address yet (because branches are relative to the PC)
  - MIPS still incurs 1 cycle branch penalty
  - With some other machines, branch target is known before branch condition

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; eg in

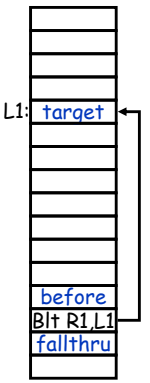
```
  LW R3, #100
  LW R4, #200
  BEQZ R1, L1
  SW R3, X
  SW R4, X
L1:
  LW R5,X
```

```
If (R1==0)
  X=100
Else
  X=100
  X=200
R5 = X
```

- "SW R3, X" instruction is executed regardless
- "SW R4, X" instruction is executed only if R1 is non-zero

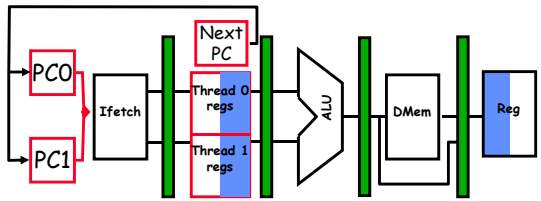
Delayed Branch

- **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
- **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)**
- **Canceling branches**
  - Branch delay slot instruction is executed but write-back is disabled if it is not supposed to be executed
  - Two variants: branch "likely taken", branch "likely not-taken"
  - allows more slots to be filled



Eliminating hazards with simultaneous multi-threading

- If we had no stalls we could finish one instruction every cycle
- If we had no hazards we could do without forwarding – and decode/control would be simpler too



Example: PowerPC processing element (PPE) in the Cell Broadband Engine (Sony PlayStation 3)

- IF maintains two Program Counters
- Even cycle – fetch from PC0
- Odd cycle – fetch from PC1
- Thread 0 reads and writes thread-0 registers
- No register-to-register hazards between adjacent pipeline stages

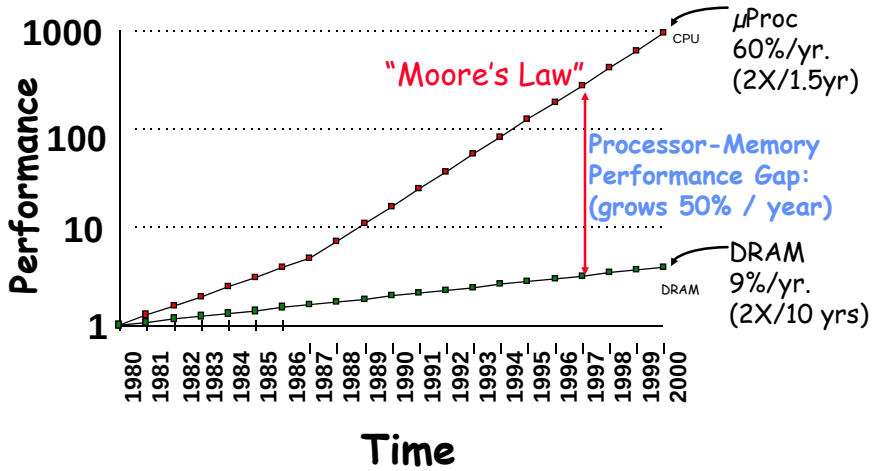
So – how fast can this design go?

- A simple 5-stage pipeline can run at >3GHz
- Limited by critical path through slowest pipeline stage logic
- Tradeoff: do more per cycle? Or increase clock rate?
  - Or do more per cycle, in parallel…
- At 3GHz, clock period is 330 picoseconds.
  - The time light takes to go about four inches
  - About 10 gate delays
    - for example, the Cell BE is designed for 11 FO4 ("fan-out=4") gates per cycle: www.fe.infn.it/~belletti/articles/ISSCC2005-cell.pdf
    - Pipeline latches etc account for 3-5 FO4 delays leaving only 5-8 for actual **work**

- How can we build a RAM that can implement our MEM stage in 5-8 FO4 delays?

Life used to be so easy

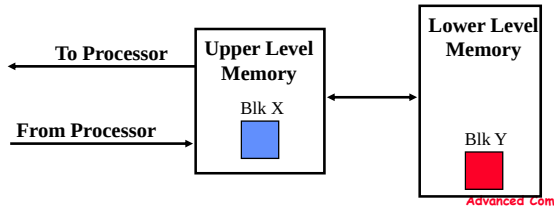
Processor-DRAM Memory Gap (latency)



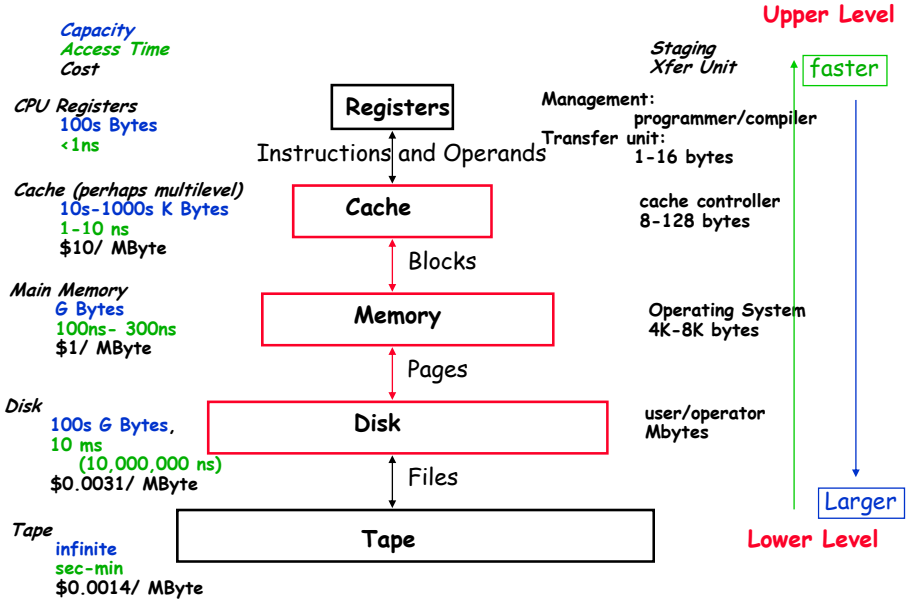
In 1980 a large RAM's access time was close to the CPU cycle time. 1980s machines had little or no need for cache. Life is no longer quite so simple.

Memory Hierarchy: Terminology

- Hit: data appears in some block X in the upper level
  - Hit Rate: the fraction of memory accesses 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 retrieved from a block Y in the lower level
  - 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
  - Typically *hundreds* of missed instruction issue opportunities



Levels of the Memory Hierarchy



The Principle of Locality

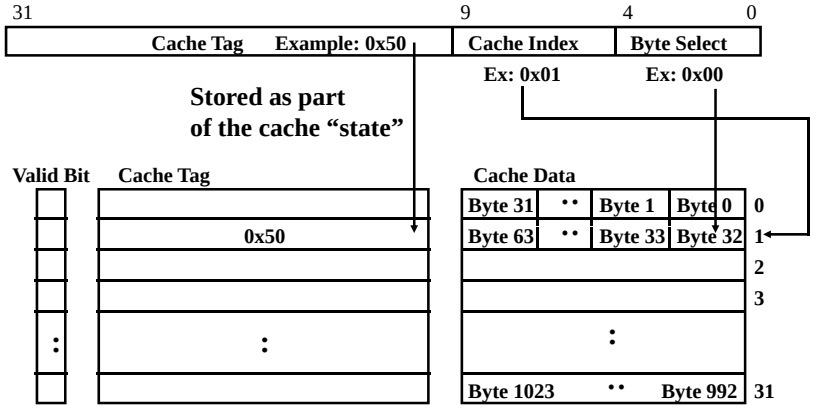
- The Principle of Locality:
  - Programs access a relatively small portion of the address space at any instant of time.
- Two Different Types of Locality:
  - Temporal Locality (Locality in Time): If an item is referenced, it will tend to be referenced again soon (e.g., loops, reuse)
  - Spatial Locality (Locality in Space): If an item is referenced, items whose addresses are close by tend to be referenced soon (e.g., straightline code, array access)
- In recent years, architectures have become increasingly reliant (totally reliant?) on locality for speed

Cache Measures

- *Hit rate*: fraction found in that level
  - So high that usually talk about *Miss rate*
  - Miss rate fallacy: as MIPS to CPU performance, miss rate to average memory access time in memory
- Average memory-access time
  = Hit time + Miss rate x Miss penalty
  (ns or clocks)
- *Miss penalty*: time to replace a block from lower level, including time to replace in CPU
  - *access time*: time to lower level
    = f(latency to lower level)
  - *transfer time*: time to transfer block
    =f(BW between upper & lower levels)

1 KB Direct Mapped Cache, 32B blocks

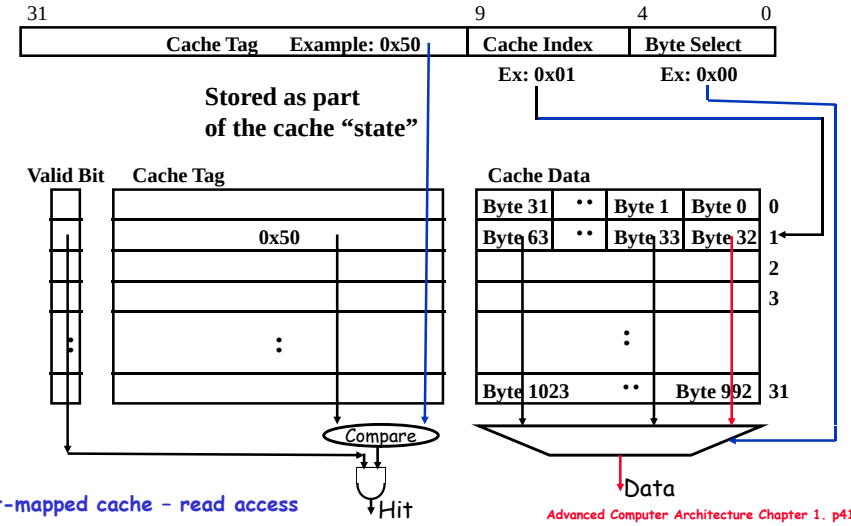
- For a 2^N byte cache:
  - The uppermost (32 - N) bits are always the Cache Tag
  - The lowest M bits are the Byte Select (Block Size = 2^M)



Direct-mapped cache - storage

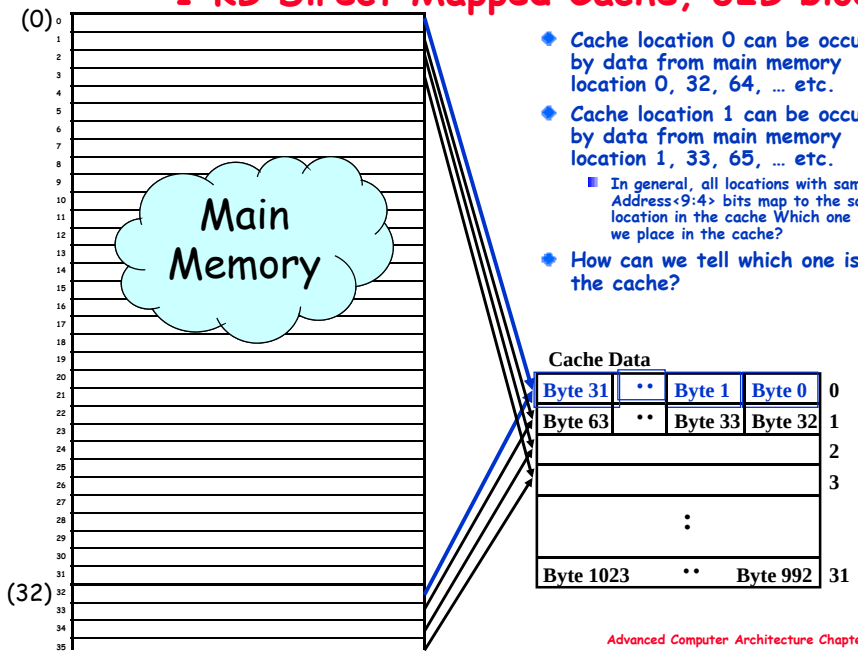
1 KB Direct Mapped Cache, 32B blocks

- 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)



Direct-mapped cache – read access

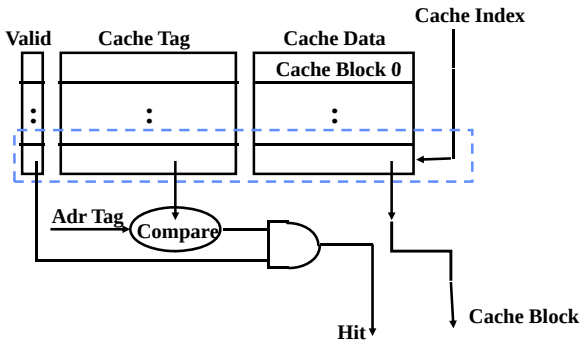
1 KB Direct Mapped Cache, 32B blocks



- Cache location 0 can be occupied by data from main memory location 0, 32, 64, … etc.
- Cache location 1 can be occupied by data from main memory location 1, 33, 65, … etc.
  - In general, all locations with same Address<9:4> bits map to the same location in the cache Which one should we place in the cache?
- How can we tell which one is in the cache?

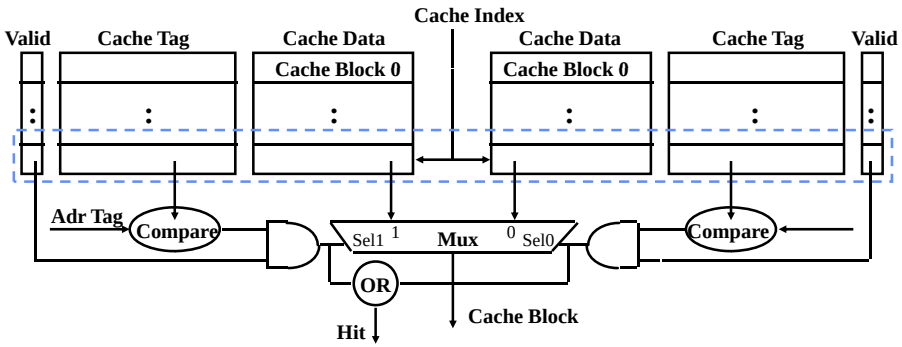
Direct-mapped Cache - structure

- Capacity: C bytes (eg 1KB)
- Blocksize: B bytes (eg 32)
- Byte select bits: 0..log(B)-1 (eg 0..4)
- Number of blocks: C/B (eg 32)
- Address size: A (eg 32 bits)
- Cache index size: I=log(C/B) (eg log(32)=5)
- Tag size: A-I-log(B) (eg 32-5-5=22)



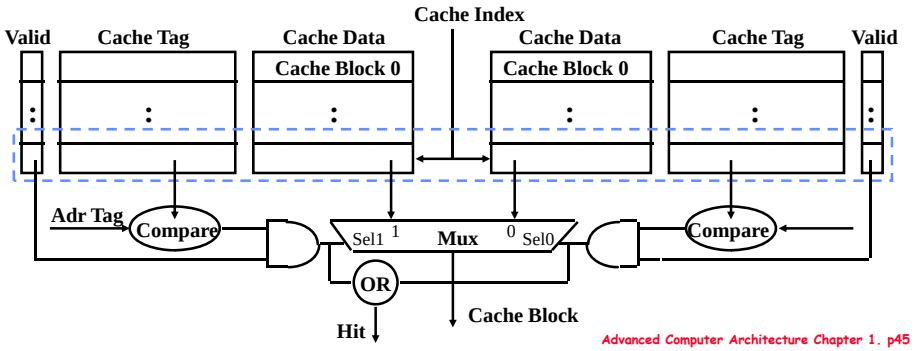
Two-way Set Associative Cache

- N-way set associative: N entries for each Cache Index
  - N direct mapped caches operated 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

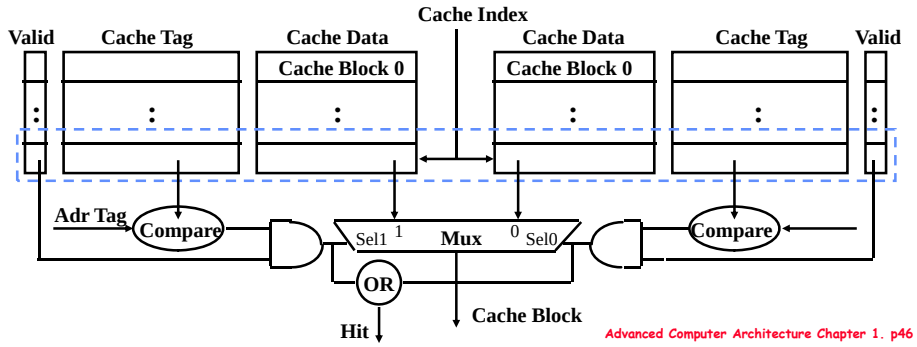
- N-way Set Associative Cache v. Direct Mapped Cache:
  - N comparators vs. 1
  - Extra MUX delay for the data
  - Data comes AFTER Hit/Miss
- In a direct mapped cache, Cache Block is available BEFORE Hit/Miss:
  - Possible to assume a hit and continue. Recover later if miss.



Basic cache terminology

Example: Intel Pentium 4 Level-1 cache (pre-Prescott)

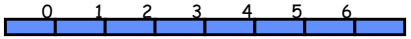
- **Capacity: 8K bytes** (total amount of data cache can store)
- **Block: 64 bytes** (so there are 8K/64=128 blocks in the cache)
- **Ways: 4** (addresses with same index bits can be placed in one of 4 ways)
- **Sets: 32** (=128/4, that is each RAM array holds 32 blocks)
- **Index: 5 bits** (since 2^5=32 and we need index to select one of the 32 ways)
- **Tag: 21 bits** (=32 minus 5 for index, minus 6 to address byte within block)
- **Access time: 2 cycles,** (.6ns at 3GHz; pipelined, dual-ported [load+store])



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)*

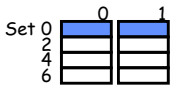
Q1: Where can a block be placed in the upper level?



In a fully-associative cache, block 12 can be placed in any location in the cache

In a direct-mapped cache, block 12 can only be placed in one cache location, determined by its low-order address bits –

(12 mod 8) = 4

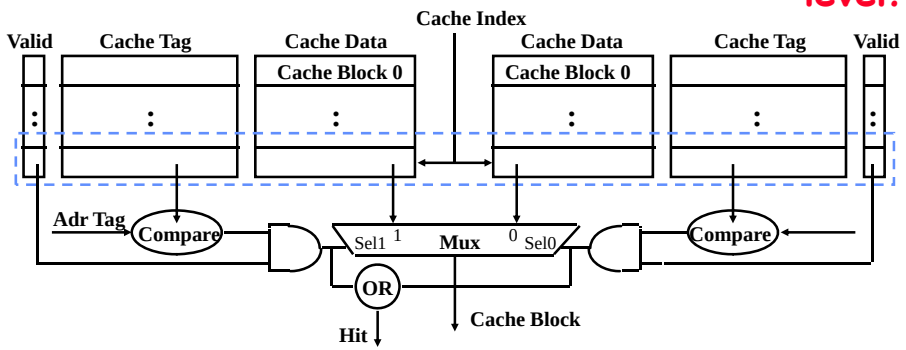


In a two-way set-associative cache, the set is determined by its low-order address bits –

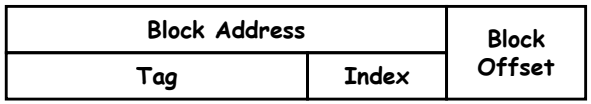
(12 mod 4) = 0

Block 12 can be placed in either of the two cache locations in set 0

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)

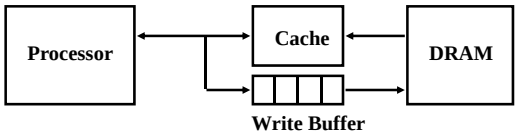
Assoc:	2-way		4-way		8-way	
Size	LRU	Ran	LRU	Ran	LRU	Ran
16 KB	5.2%	5.7%	4.7%	5.3%	4.4%	5.0%
64 KB	1.9%	2.0%	1.5%	1.7%	1.4%	1.5%
256 KB	1.15%	1.17%	1.13%	1.13%	1.12%	1.12%

Benchmark studies show that LRU beats random only with small caches

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

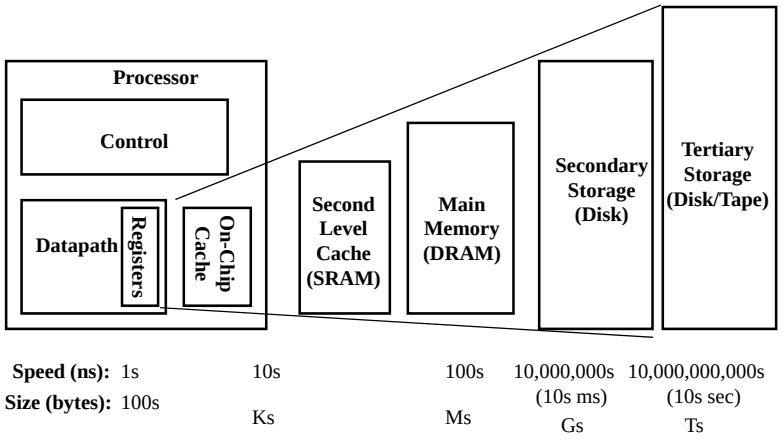
Write Buffer for Write Through



- 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 designer's nightmare:
  - 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.



Large-scale storage

- StorageTek STK 9310 ("Powderhorn")
  - 2,000, 3,000, 4,000, 5,000, or 6,000 cartridge slots per library storage module (LSM)
  - Up to 24 LSMs per library (144,000 cartridges)
  - 120 TB (1 LSM) to 28,800 TB capacity (24 LSM)
  - Each cartridge holds 300GB, readable up to 40 MB/sec
- Up to 28.8 petabytes
- Ave 4s to load tape

http://www.b2net.co.uk/storagetek/storagetek_powderhorn_9310_tape_library.htm
http://en.wikipedia.org/wiki/Tape_library
http://www.ibm.qassociates.co.uk/storage-tape-enterprise-tape-drive-J1A-specifications.htm

http://www.karo.com

Can we live without cache?

- Interesting exception: Cray/Tera MTA, first delivered June 1999:
  - www.cray.com/products/systems/mta/
- Each CPU switches every cycle between 128 threads
- Each thread can have up to 8 outstanding memory accesses
- 3D toroidal mesh interconnect
- Memory accessed hashed to spread load across banks
- MTA-1 fabricated using Gallium Arsenide, not silicon
- "nearly un-manufacturable" (wikipedia)
- Third-generation Cray XMT:
  - http://www.cray.com/Products/XMT.aspx

Where we are going…

- Ch1
  - Review of pipelined, in-order processor architecture and simple cache structures
- Ch2
  - Caches in more depth
  - Software techniques to improve cache performance
  - Virtual memory
  - Benchmarking
  - Fab
- Ch3
  - Instruction-level parallelism
  - Dynamic scheduling, out-of-order
  - Register renaming
  - Speculative execution
  - Branch prediction
  - Limits to ILP
- Ch4
  - Compiler techniques – loop nest transformations
  - Loop parallelisation, interchange, tiling/blocking, skewing
- Ch5
  - Multithreading, hyperthreading, SMT
  - Static instruction scheduling
  - Software pipelining
  - EPIC/IA-64; instruction-set support for speculation and register renaming
- Ch6
  - GPUs, GPGPU, and manycore
- Ch7
  - Shared-memory multiprocessors
  - Cache coherency
  - Large-scale cache-coherency; ccNUMA. COMA

- Lab-based coursework exercise:
  - Simulation study
  - "challenge"
  - Using performance analysis tools
- Exam:
  - Partially based on recent processor architecture article, which we will study in advance (see past papers)