CMSC 611
Spring 2000

Prof. Ethan L. Miller (elm@csee.umbc.edu)

Welcome to CMSC 611!

- Instructor: Professor Ethan Miller (`elm@csee.umbc.edu`)
- TA: Joy Su (`xsu1@umbc.edu`)
- Text: *Computer Architecture: A Quantitative Approach* (Hennessy and Patterson)
- Supplementary reading material: available online
- Course web pages
  - `http://www.csee.umbc.edu/courses/graduate/611/Spring00/`
  - Students are required to read Web pages regularly
  - Assignments & announcements will be on the Web
  - Class slides will be on the Web (available only within the `umbc.edu` domain)

Why take this course?

- Introduction to many of the methods used by designers to implement
  - General purpose processors
  - Memory systems
  - I/O systems
  - Multiprocessors
- Introduction to methods of analyzing CPU performance

More reasons to take this course

- Exposure to many characteristics of today's architectures
  - CISC: Pentium/Merced, 680x0
  - RISC: PowerPC, DEC Alpha, Sun SPARC, HP PA-RISC, MIPS, …
- Learn how to do graduate research in computer systems
- It's required….

Course mechanics

- Semester long project (40% of grade)
  - Work in groups of 3-4 students
  - Grade based on technical content & presentation
- Homework every 1-2 weeks (total 20% of grade)
- Exams
  - Midterm (15% of final grade)
  - Final exam (20% of final grade)
- Class participation: 5% of grade
- Project & both exams ***required*** to pass the class

Projects

- Done in groups of 3-4 people
- Many topics available
  - Pick one that the whole group is interested in
    - Work-related?
    - Issue of current relevance
  - Do something different from what everyone else is doing
- Meet project milestones throughout the semester
- Make a presentation on the project to the class
- Hand in a paper on the project

Course summary

- Measuring performance & cost
- Instruction sets
- Improving CPU performance
  - Dynamic instruction scheduling & instruction-level ||ism
  - Branch prediction
- Vector processors
- Memory hierarchies
- Storage systems
- Multiprocessors

What is computer architecture?

- Answer: the design of computer systems
- What must a computer system provide?
  - Application support
  - Software compatibility
  - OS requirements
  - Standards
- Computer system designer must take into account
  - Cost & performance
  - Design complexity
  - reliability and fault tolerance

Trends in hardware

- IC technology
  - Transistor density improves about 50% per year
  - 60% - 80% increase per year of transistors per chip
- Memory (RAM)
  - Density increases at around 60%/year
  - Cycle time decreases around 30% in ten years.
  - Bandwidth proportional to cycle time and data width

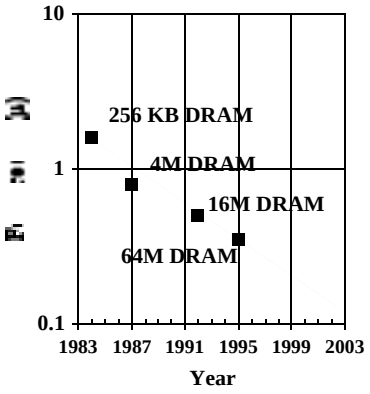
Moore's Law

- Gordon Moore (Intel Chairman, 1965) said "# of transistors in an IC will double every 18 months"
- Accurate for last 20 years; will it hold for the next 15 years?
  - Physics : lithographic limits
  - Economics: Design and test complexity, fabrication costs

	1995 (350 nm)	1998 (250 nm)	2001 (180 nm)	2004 (130 nm)	2007 (100 nm)	2010 (80 nm)
DRAM (bits)	64M	256M	1G	4G	16G	64G
MPU transistors / cm^2	4M	7M	13M	25M	50M	90M
DRAM chip size (mm^2)	190	280	420	640	960	1400
MPU chip size (mm^2)	250	300	360	430	520	620

Moore's Law (2)

- Minimum transistor feature size must decrease by a factor of 0.7 every three years
- Where is the lower limit for feature size?



Moore's Second Law

- Moore's Second Law:
  The cost of building a semiconductor fab is doubling every three to four years.
- In 1995, approximately 50 fabs in operation worldwide
  - Another 50 in some state of completion
  - Current cost > $1 billion
- Physical Limits
  - "In 2010, we will run into the physical limitation of having a fraction of an electron show up at a gate to switch the state of the transistor" (Joel Birnbaum, HP senior president of R&D)

More trends in hardware

- Disk technology
  - Storage density increases around 50% per year
  - Rotation speed increases slowly
  - Bandwidth proportional to square root of density; increases around 25% per year
  - Access time (seek) drops around 30% in ten years
- Systems must take trends into account
  - Moving targets!
  - Optimal system today may not be optimal tomorrow

Trends in cost

- Learning curve : products drop in cost over time
  - Minor improvements in design & manufacturing
  - Amortization of startup costs (R&D, etc.)
- Volume decreases per-unit cost
  - Fixed costs amortized over more units
  - May have more competitors to help keep prices down
  - Commodities: standardized components available from many vendors
    - Little or no profit margin for commodities
    - Often *much* less expensive than low-volume components
- Relative prices of components can change...

Where does the money go?

- Component costs: raw material costs.
- Direct costs: costs incurred to actually make a single item (usually 20% to 40% of component costs)
- Gross margin: overhead not associated with a single item — R&D, plant equipment, profit, taxes, etc. (typically 20% to 55% of average selling price)
- Average selling price: direct cost + gross margin.
- List price: ASP + reseller's margin

Design tradeoff examples

- Designing for low cost
  - Make choices that minimize cost no matter what
  - Design for ease of manufacture as well as low component cost
- Designing for high performance
  - Spend more money on R&D
  - Worry less about how much it costs to manufacture
- Designing for good cost/performance
  - Use more expensive parts if they have "bang for the buck"
  - Control costs, but not if they lower performance
- Cray vs. desktop workstation vs. Palm III

Trends in cost

- Learning curve : products drop in cost over time
  - Minor improvements in design & manufacturing
  - Amortization of startup costs (R&D, etc.)
- Volume decreases per-unit cost
  - Fixed costs amortized over more units
  - May have more competitors to help keep prices down
  - Commodities: standardized components available from many vendors
    - Little or no profit margin for commodities
    - Often *much* less expensive than low-volume components
- Relative prices of components can change...

Where does the money go?

- Component costs: raw material costs.
- Direct costs: costs incurred to actually make a single item (usually 20% to 40% of component costs)
- Gross margin: overhead not associated with a single item — R&D, plant equipment, profit, taxes, etc. (typically 20% to 55% of average selling price)
- Average selling price: direct cost + gross margin.
- List price: ASP + reseller's margin

Design tradeoff examples

- Designing for low cost
  - Make choices that minimize cost no matter what
  - Design for ease of manufacture as well as low component cost
- Designing for high performance
  - Spend more money on R&D
  - Worry less about how much it costs to manufacture
- Designing for good cost/performance
  - Use more expensive parts if they have "bang for the buck"
  - Control costs, but not if they lower performance
- Cray vs. desktop workstation vs. Palm III

Chip costs

- Chip cost = $\frac{(\text{die cost} + \text{testing cost} + \text{packaging cost})}{\text{final test yield}}$
- Die cost = $\frac{\text{wafer cost}}{\text{dies/wafer} \times \text{die yield}}$
- Bigger dies => higher cost (dies/wafer decreases)
- How does die size affect yield?

Die yields

$$die\ \ yield = wafer\ \ yield \times \left(1 + \frac{defects\ \ per\ \ unit\ \ area \times die\ \ size^{-\alpha}}{\alpha}\right)$$

- "Wafer yield" accounts for wafers that are completely bad.
- Model assumes randomly distributed defects
  - 1995: 0.6 - 1.2 per cm^2
  - Learning curve reduces this value over time
- Alpha corresponds to the complexity of the manufacturing process
  - Roughly equal to the number of masking levels
  - Approximately 3 today

IC cost : the bottom line

- # of good dies/wafer = dies/wafer * die yield.
  - Larger chips => fewer dies/wafer
  - More complex fab process => lower yield
  - Not a linear relationship
- Die cost is proportional to $area^4$ for $\alpha = 3.0$
  - Large area => very expensive dies!
  - Reduce feature size rather than increase die size
- Designer controls only the die size
  - Decides which features/functions to include
  - Doesn't control feature size!
- Moral: smaller dies save money!

Computer performance

- Goal: quantitative comparison of two or more systems
  - Figure out which is "faster"
  - Definition of "faster" can vary…
    - User: response time & execution time
    - Sysadmin: throughput & bandwidth
- Measure relative performance using ratios
  - $ExecTime_A/ExecTime_B = 2$ => B is 2x the speed of A
  - Performance = 1/ExecTime: this corrects ratios so that $Performance_A/Performance_B = 2$ => A is 2x the speed of B
- Execution time of real programs is the only consistent & reliable measure of performance!

Performance metrics

- Measure time in several ways
  - Wall clock time: total time to complete a task (including all operations & wait time)
  - CPU time: includes only the time spent actually executing, and excludes wait time & I/O time
- User time vs. system time
  - User time = time spent running user code
  - System time = time spent by process in the OS
  - System performance => elapsed time on an unloaded system
  - CPU performance => elapsed *user* CPU time
- Focus on CPU performance (exclude OS effects…)

Measuring performance: benchmarks

- Goal of performance evaluation: determine the speed of a computer on your specific workload
  - Best accomplished by simply running your workload, but…
  - Difficult to do without a lot of work on your part!
- Alternative: standard workloads called *benchmarks*
  - Real programs: spice, gcc, matmult, etc.
    + Might match programs you might run
    - May not include the specific programs you run
  - Kernels: key (usually small) pieces of real programs
    + Isolate individual performance elements
    - Exclude potentially time-consuming setup & other code

More on benchmarks

- Other kinds of benchmarks
  - Toy benchmarks: small programs that produce known results
    + Easily ported to different architectures
    - Very small, and may not exercise full system
  - Synthetic benchmarks: artificial programs that try to "exercise" the system in the same way as an "average" real program
    + Easily ported to different architectures
    - Average behavior may not be the same as the behavior of individual programs
- Benchmark suites include several different benchmarks
  - Different kinds of programs test different things
  - Workload can be formed from a combination of benchmarks

Running benchmarks

- Benchmarks are like other kinds of experiments
  - Results must be ***reproducible***
  - Conditions need to be controlled as much as possible
- Guidelines for running benchmarks
  - Record as much information as possible
    - Hardware used: CPU, memory size, cache, disk, etc.
    - OS & compiler used (including compiler options, etc.)
    - Program version (preferably entire code!)
    - Program options
    - Program input & output
- Another person should be able to reproduce your results!

Comparing performance

- Goal: compare performance of several computer systems
  - Single number that shows performance?
  - Number should be larger for faster machines (use performance rather than execution time)
- Problem: this isn't as straightforward as it might seem
  - Benchmark INT takes 5 seconds on A, 10 seconds on B
  - Benchmark FP takes 18 seconds on A, 10 seconds on B
  - Which machine is faster, A or B?

Averaging benchmark results

Averaging benchmark results

$$\text{Arithmetic mean} = \frac{1}{n}\sum_{i=1}^{n} Execution\ \ time_i$$

$$\text{Harmonic mean} = \frac{n}{\sum_{i=1}^{n}\frac{1}{rate_i}}$$

$$\text{Weighted arithmetic mean} = \frac{1}{n}\sum_{i=1}^{n} weight_i \times Execution\ \ time_i$$

Combining benchmark results

- Use arithmetic mean for execution times
  - Use weighting to emphasize one benchmark
  - Capture relative frequency with weight
- Use harmonic mean for rates
  - Weighting can apply to harmonic means

$$\text{Arithmetic mean} = \frac{1}{n}\sum_{i=1}^{n} Execution\ \ time_i$$

$$\text{Harmonic mean} = \frac{n}{\sum_{i=1}^{n}\frac{1}{rate_i}}$$

$$\text{Weighted arith. mean} = \frac{1}{n}\sum_{i=1}^{n} weight_i \times Execution\ \ time_i$$

Another approach: normalize

- Normalize performance or execution time to that of a reference machine
  - Often use a VAX 11/780, set to 1 MIPS
  - Combine results with geometric mean:

$$Geom\ \ mean = \sqrt[n]{\prod_{i=1}^{n} Performance_i}$$

- Doesn't predict execution time
  - Allows easy comparison of two machines
  - Comparison doesn't depend on choice of "base" machine
- Ideal solution: measure a real workload on each machine!

Quantitative design principles

- Make the common case fast!
  - Optimize the most frequently used operations
  - Calculate overall performance to see if an optimization is worthwhile
- Example: arithmetic with overflow
  - Overflow is uncommon, so use exceptions to handle it
  - Don't make programs check every arithmetic operation!
- Use Amdahl's Law to quantify potential improvements in performance from improving particular operations

Amdahl's Law

- Performance improvement for a particular optimization is limited by the fraction of time the optimization is in use
- Amdahl's Law defines this speedup as:
  - Speedup = Execution time$_{orig}$ /Execution time$_{enhanced}$
  - Speedup = Performance$_{enhanced}$/Performance$_{orig}$
- Factors affecting execution time:
  - Fraction$_{enhanced}$: fraction of execution time that can be sped up by the enhancement (0≤ fraction ≤1)
  - Speedup$_{enhanced}$: performance improvement on the enhanced portion of the code (speedup > 1)
    - Speedup = Execution time$_{orig}$ /Execution time$_{enhanced}$

Defining Amdahl's Law

$$Exec\ \ time_{new} = \underbrace{Exec\ \ time_{old} \times \left(1 - Fraction_{enhanced}\right)}_{\text{Time spent in original program}} + \underbrace{\frac{Fraction_{enhanced}}{Speedup_{enhanced}}}_{\text{Time spent in enhanced program}}$$

$$Speedup_{overall} = \frac{Exec\ \ time_{old}}{Exec\ \ time_{new}} = \frac{1}{\left(1 - Fraction_{enhanced}\right) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

- This formula gives the overall speedup for a single enhancement

Amdahl's Law: Example

- A CPU without floating point hardware spends 40% of its time doing FP calculations
- Adding FP hardware will speed up FP calculations by 8x
- How much faster will the computer run with FP hardware?
- Answer:
  - Fraction$_{enhanced}$ = 0.4
  - Speedup$_{enhanced}$ = 8
  - Speedup$_{overall}$ =
    1 / ((1 - Fraction$_{enhanced}$) + (Fraction$_{enhanced}$ / Speedup$_{enhanced}$))
    = 1 / ((1 - 0.4) + (0.4 / 8)) = 1/0.65 = 1.54
  - The CPU with FP hardware will run the program 1.54x faster

Implications of Amdahl's Law

- Law of diminishing returns
  - Speedup is limited by the fraction of execution time that can be sped up
  - If we only enhance 75% of the code, maximum speedup is 4!
- Parallel processing is hard!
  - In most parallel programs, there is some serial portion of the code that can't be parallelized
  - Suppose $\text{fraction}_{\text{serial}} = 0.02$
  - Maximum speedup is then 50x!
- In general, maximum speedup = $1/(1\text{-Fraction}_{\text{enhanced}})$

CPU performance equations

- It can be difficult to directly measure performance improvements using new techniques
  - Must build (or simulate) the system first!
  - What about different programs?
- Another method: break execution time into three components
  - Instruction count (IC): instructions executed for a task
  - Cycles per instruction (CPI): average number of cycles each instruction requires
  - Clock cycle time (CCT): frequency of the CPU's clock
  - Execution time = IC * CPI * CCT
  - This method can easily predict performance gains from reducing any factor in the equation

CPI as a performance metric

- Cycles per instruction (CPI) is an important metric
  - Cycle time improvements usually come from advances in VLSI techniques
  - Instruction count reductions require changing the instruction set
  - CPI can be reduced for a given instruction set
- Calculate average CPI by:
  - CPU clock cycles in a program / instructions executed
  - Cycles in a program = execution time * clock frequency
- Lower CPIs are better
  - CPIs below 1 are possible (execute > 1 instruction per cycle)
  - Reducing average CPI improves performance for a given instruction set and cycle time

Improving CPU performance

- It can be difficult to change one factor in isolation
  - Cycle time: hardware (VLSI) & organization
  - CPI: organization & instruction set architecture (ISA)
  - Instruction count: ISA & compiler technology
- Often, different instructions behave differently:

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

  - CPI_i is the average CPI for instruction *i*
  - CPI_i isn't a constant for all instructions *i*!
  - IC_i is the number of times instruction *i* appears in the program
  - Measuring individual instruction CPIs and counts isn't too hard, and allows performance prediction for new programs

Fallacies & pitfalls

- MIPS (millions of instrs per second) isn't a good metric!
  - MIPS is dependent on the instruction set: difficult to compare across ISAs
  - MIPS varies between programs on the same computer
  - MIPS can be lower on computers with special purpose hardware and good compilers
    - Example: floating point hardware vs. emulation
    - Example: vector instructions
  - MIPS considers only CCT and CPI, but not instruction count
- More important: how much work gets done
- MIPS = "Meaningless Indication of Processor Speed"

More fallacies & pitfalls

- Beware synthetic benchmarks (ie, Whetstone & Dhrystone)
  - Often contain code sequences not in real code
  - Compilers can be tuned specifically to optimize for them
  - Behave unlike real programs in many ways
  - Fit into cache, unlike real programs
- Peak performance ≠ observed performance
  - Peak = "guaranteed not to exceed"
  - Most programs run nowhere near peak performance
  - Pipeline hazards, superscalar issue, branches, and cache issues all slow down the CPU and prevent peak performance

Instruction set architectures

- ISA metrics
- Basic ISA taxonomy
- Differentiating ISAs
  - Memory access
    - Big vs. little endian
    - Alignment
    - Addressing modes
  - Kinds of instructions
- Making life easier for compiler writers
- Example ISA: DLX

ISA metrics

- Instruction density: how much space per task?
- Instruction count: how many instructions per task?
- Instruction complexity
  - How much decoding is necessary?
  - How many “simple” operations per instruction?
- Instruction length
  - Constant vs. variable-length
  - Average instruction length

⇒ Keep these metrics in mind when discussing individual ISAs

ISA taxonomy

- Accumulator
  - One operand is in the accumulator, the other in memory
  - Instructions move data between accumulator and memory
- Stack-based
  - All operands on the top of the stack
  - Instructions move data between top of stack and rest of memory
- These are less common ISAs for CPUs
  - Java Virtual Machine is stack-based
- More common today: General Purpose Register (GPR) ISAs

GPR ISAs

- Memory-memory (example: VAX)
  - May have as many as 3 operands in memory
  - Usually have relatively few registers (used to save memory references)
- Register-memory (example: 680x0)
  - One operand in a register, the other may be in memory
  - Usually has only two operands (register is both source & dest)
  - Generalization of accumulator ISA
- Load-store (example: PowerPC, MIPS, DLX)
  - Data moved explicitly between registers and memory
  - ALU operates on registers (usually 2 source & 1 destination)

More on GPR ISAs

- GPR ISAs are the most popular design today
  - Registers are much faster than memory (2 ns vs. 70ns)
  - Compilers can optimize register use for:
    - Holding intermediate values in calculations & addresses
    - Caching variable values
    - Passing parameters
- GPRs can be classified by
  - The number of ALU operands (2 or 3)
  - The number of operands in memory (0-3)

Accessing memory

- Endianness
  - Big-endian: most significant byte stored in first byte of word
  - Little-endian: least significant byte stored in first byte of word
  - No intrinsic advantage to either approach
  - Networks usually use big-endian for transmitting words
- Alignment: must *n*-byte objects be aligned to addresses evenly divisible by *n*?
  - Advantage: more flexible for programs
  - Disadvantage: more complicated hardware
  - May be relaxed slightly, ie, 8-byte objects 4-byte aligned

Addressing modes

- Instructions specify operands using addressing modes
- Register: value is in a register (ADD **R4**, **R5**, **R6**)
- Immediate: value contained in the instruction (ADD R4, **#4**)
- Memory
  - Indirect / Displacement: address in a register (LW R1, **8(R4)**)
  - Indexed: add two registers to get address (LW R1, **8(R1+R4)**)
  - Modifiers include
    - Auto-increment/decrement: used for stacks (LW R1, **(R2)+**)
    - Scaling: address scaled by size of data
    - Multiple levels of indirection
  - Absolute/Direct: address contained directly in instruction

More addressing modes

- PC-relative addressing
  - Use current value of PC as the base rather than a register
  - Often used for branches and static variables
- Other addressing modes
  - Register update on PowerPC: put the effective address into the base register (implements auto-increment/decrement flexibly)
  - Implicit: top of stack or accumulator used “by default”
  - Memory deferred: multiple levels of indirection on a memory access

Implications of addressing modes

- More addressing modes can
  - Lower instruction count & increase code density
  - Increase implementation complexity
  - Increase CPI (maybe)
  - Reduce execution time (maybe)
- Fewer addressing modes can
  - Increase instruction count
  - Allow smaller, simpler (and easier to decode) ISA
  - Decrease CPI (maybe)
  - Reduce execution time (maybe)

Minimum set of addressing modes

- Register
  - Must have a way of accessing registers!
  - Not necessary for stack-based architectures
- Indirect
  - Need a way of accessing memory
  - May require that all addresses first be loaded into a register...
  - Displacement isn't much harder (often used)
    - 75% of displacements are <12 bits
    - 99% of displacements are <16 bits

Minimum set of addressing modes

- Immediate
  - Must be a way of getting constant values into the CPU
  - Could use constants in memory, but how would they get there?
  - How big must immediate values be?
    - Most values are less than 8 bits (50%+)
    - Addresses require large immediate values: combine two 16-bit values to make a 32-bit address
    - Usually use 8-16 bit immediates
- Other addressing modes could be useful, but are they worth the complexity?

Instruction set operations

- Required operations
  - Arithmetic/logical: ALU operations
  - Loads/stores: moving data between registers & memory
  - Control: branches, jumps, subroutines, traps
- Optional (but very useful!)
  - Operating system support: traps (OS calls), VM support (TLB) & cache support
  - Floating point
- Other operations (may be useful for some situations)
  - Graphics / vector operations (becoming more common)
  - Binary-coded decimal & string (becoming less common)

⇒ **Make the common case (ALU, load/store, control) fast!**

Control flow instructions

- Types of control flow instructions
  - Conditional branches (>80% of all control flow instructions)
  - Jumps (also unconditional branches)
  - Procedure calls & returns
- Addressing for control flow
  - PC-relative: allows position-independent (relocatable) code
    - Commonly used for branches & jumps
  - Indirect: address in register; used for
    - Procedure return in some architectures
    - Jump tables (switch/case statements) & virtual functions
    - Dynamic linking
  - Absolute: address in instruction; used for ROM calls

Conditional branches

- Displacement length
  - Most conditional branches go fewer than 10 instructions away
  - Fields of 8 bits (signed) should be enough for most situations!
- Branch condition
  - Compare register to 0/1: test condition and put result in register
  - Compare register to other values (register, immediate)
  - Condition codes
    - Comparison sets flags in the CPU
    - Can be slower and involves more state to keep track of
- Branch destinations
  - Non-loop branches usually forward (75% of branches, hard to predict)
  - Looping branches tend to be backward (usually taken)

Subroutines

- Transfer control and save some state
  - At a minimum, save return address
  - Save register state: in CALL instruction or done by compiler?
    - Caller save: calling routine saves registers it'll use after routine
    - Callee save: called routine saves registers it'll modify
- Return from subroutine
  - Restore original state
  - Jump to instruction after subroutine call
- Complex subroutine call / return can be very slow
  - Provide simple instructions & let compiler combine instructions to save the necessary state
  - Leaf subroutines may not even need to access memory to save state!

Fixed & variable length instructions

- Variable length instructions (common in CISC)
  - Compose instructions of "pieces": operation type, operand specifiers
  - Pack more instructions into a fixed space (better code density)
  - Decoding is more difficult: must decode instruction to figure out where the next one starts
  - Instruction fetch must be able to handle unaligned accesses
- Fixed length instructions (common in RISC)
  - Operation & addressing modes fixed into opcode
  - Supports relatively few addressing modes: common in load/store
  - Decoding is much less complex
  - Prefetch many instructions ahead without decoding intervening instrs
  - Code is less dense: requires more memory for given functionality

Goal of ISA design: compilers

- 99%+ of computer code produced by compilers
  - Make an instruction set easy for a compiler to use
  - Make it possible (but not necessarily easy) for a human to read
- Compiler passes
  - Parsing / language front end
  - High-level optimization: inlining, loop unrolling
  - Global optimization: register allocation, subexpression elimination
  - Code generation: output the actual assembly or machine language
    - Instruction selection
    - Instruction reordering
    - Delay slot filling

Designing an ISA for compiler use

- Provide a regular instruction set
  - Three components of ISA (operations, data types, addressing modes) should be orthogonal: all operations work on all data types & addressing modes
  - General purpose registers (vs. special purpose)
- Provide primitives, not full solutions
  - Complex instructions may not match language (C vs. Pascal strings)
  - Allow the compiler to build up its own code sequences
- Simplify tradeoffs
  - Don't make the compiler writer choose from 20 options!
- Allow constants to be specified at compile time

⇒ **KISS: KEEP IT SIMPLE, STUPID!**

DLX: example load/store ISA

- Skim the material on the DLX ISA
- Highlights include
  - Simple load/store architecture
    - Relatively large register set (32 GPRs)
    - Only load/store can access memory; all other instructions operate on registers
  - Addressing modes
    - Displacement (16-bit signed offset)
    - Immediate (16-bit signed or unsigned values)
    - Register
  - Fixed length instructions (32 bits)

Here be dragons...

- Don't design an ISA oriented towards a specific HLL
  - Attempts to reduce the semantic gap may result in a semantic clash!
  - Instruction mismatch is likely, in which special instructions do more work than is required for the frequent case.
- There is no such thing as a typical program.
  - Programs can vary significantly in how they use an instruction set
  - Many times it is meaningless to average frequency criteria over several programs (i.e. the mix of data transfer sizes)
- Avoid the temptation to put in lots of cool instructions
  - What's cool today may be useless tomorrow
  - You'll have to support for a ***very*** long time

Building the perfect ISA

- There's no such thing as a flawless ISA
  - Every ISA involves tradeoffs!
    - Doing one thing well means doing something else less well
    - The perfect ISA for one program isn't perfect for another program
  - Predicting technology 10+ years into the future is very difficult!
- Flawed ISAs can be successful (example: Intel x86 ISA)
  - Register architecture is messy (no GPRs)
  - Segmentation is somewhat messier than pure paging
  - Stack-based FPU isn't as efficient as register-based FPUs
- ISAs eventually die off, but it takes a while
  - Intel x86 ISA still going strong
  - Motorola 680x0 ISA runs PalmPilots, printers, and more!

Branch behavior

- Observations using SPEC subset on DLX:
  - Dynamic frequencies (used here) count number of executions
  - Static frequencies count number of occurrences in program
  - Conditional branches are much more common:
- Conditional outnumber unconditional about 3-4 to 1.
  - 14% to 16% is normal for integer benchmarks
  - FP benchmarks are much more varied at 3%-12%
- Forward branches more common: outnumber backwards branches by 3 to 1
- Frequency of taken branches
  - 67% of conditional branches are taken on average.
  - 60% of the forward and 85% of the backward branches.

Reducing branch penalties

- Move the address calculation and decision of whether to take the branch back into ID
  - If the comparison is to zero, we know the address and the decision at the end of ID.
  - If comparing one register to another, we wait until after EX to decide if the branch is taken
- Method 1: freeze pipeline until decision known
  - Always flush the pipeline of instructions up until the branch destination and condition are known
  - Branch penalty is fixed and cannot be reduced by software & compiler

Static branch prediction

- Treat every branch as not taken (predict-not-taken)
  - Continue to fetch instructions.
  - Flush the pipeline if the branch is taken.
  - Note that successor instructions may NOT change the state of the machine

  ⇒ This results in a 1 cycle stall for DLX since the decision (for zero compares) is known after ID
- Treat every branch as taken (predict-taken)
  - Wait until the target address is computed and then fetch instructions using the new PC value.
  - Flush the pipeline if the branch is NOT taken.
  - For DLX, this doesn't do much good since BOTH the branch target address and the decision (for zero compares) is known after ID

Delayed branches

- An execution cycle with a branch delay of length n is:

```
Branch instruction
Sequential successor1   }
…                       }  Branch delay slots
Sequential successorn   }
Branch target
```

- Instructions(s) in the branch delay slot(s) after the branch are always executed
  - The compiler should try to put a "useful" instruction here.
  - If none are available, then a "no-op" is inserted in the delay slot.
- Improves performance by letting the CPU do useful work while waiting for branch target and condition resolution

Delayed branch scheduling

From before	From target	From fall-through

`ADD R1,R2,R3`
`BEQZ R2,label`
`  delay slot`
	`loop:`
	`SUB R4,R5,R6`
	`ADD R1,R2,R3`
	`BEQZ R1,loop`
	`  delay slot`
		`ADD R1,R2,R3`
		`BEQZ R1,label`
		`  delay slot`
		`label:`
		`SUB R4,R5,R6`

⇩	⇩	⇩

`BEQZ R2, label`
`  ADD R1,R2,R3`
	`SUB R4,R5,R6`
	`loop:`
	`ADD R1,R2,R3`
	`BEQZ R2,loop`
	`  SUB R4,R5,R6`
		`ADD R1,R2,R3`
		`BEQZ R1,label`
		`  SUB R4,R5,R6`
		`label:`

Delayed branch limitations

- Restrictions on the instructions that are scheduled into the delay slots (i.e. data dependencies.)
  - Compiler's ability to predict accurately whether or not a branch is taken determines how much useful work is actually done.
- Many machines have introduced a cancelling or nullifying branch instruction.
  - Includes the direction that the branch is predicted to go.
  - If branch is predicted incorrectly, CPU turns the instruction in the branch delay slot into a no-op.
- Compilers can usually fill a single branch delay slot & improve performance
- More delay slots => more difficult to fill with useful work

Delayed branch performance

Pipeline performance: $pipeline\ \ speedup = \frac{pipeline\ \ depth}{1 + pipeline\ \ stalls\ \ from\ \ branches}$

This assumes no delays from other hazards...

Calculate pipeline stalls due to branches by:

$$stall\ \ cycles\ \ from\ \ branches = branch\ \ frequency \times branch\ \ penalty$$

Total pipeline speedup is thus:

$$pipeline\ \ speedup = \frac{pipeline\ \ depth}{1 + branch\ \ frequency \times branch\ \ penalty}$$

Branch performance example

- Assume: branches have a single delay slot
  - Filled with a useful instruction 65% of the time
- Assume: branch condition not known for two cycles beyond the delay slot
  - If predicted properly, there is no penalty
  - If mispredicted, the two intervening instructions must be cancelled
- Forward branches (75%) are always predicted not taken
- Backward branches (25%) are always predicted taken
- Branches are 20% of all instructions.
  - 50% of forward branches taken
  - 85% of backward branches are taken
- What is the new CPI (assuming the original CPI is 1)?

Branch performance example: solution

- For 35% of the branch instruction, the delay slot isn't filled
  - This adds 0.35 cycles of branch stalls
- 50% of forward branches suffer a 2 cycle penalty.
  - 75% of branches are forward => 0.50 * 0.75 * 2 = 0.75 cycles
- 15% of backward branches suffer a 2 cycle penalty
  - Penalty is 0.15 * 0.25 * 2 = 0.075 cycles
- Total branch penalty is 0.35 + 0.75 + 0.075 = 1.175 cycles.
- Since branches make up 20% of all instructions, the penalty to the CPI is 1.175 * 0.2 = 0.235 cycles.
- The new CPI is thus 1 + 0.235 = 1.235

More compiler optimizations for branches

- Having accurate information about branch behavior at compile time is also helpful for scheduling data hazards
- Suppose we knew that the branch was almost always taken and value in R7 was not needed in the fall through part
  - Compiler could move ADD R7, R8, R9 after the load instruction
- Suppose we knew that the branch was rarely taken and value in R4 was not needed on the taken path
  - Compiler could move OR R4, R5, R6 after the load instruction.
- These optimizations are in addition to any branch delay scheduling.

```
    LW R1,0(R2)
    SUB R1,R1,R3
    BEQZ R1,L
    OR R4,R5,R6
    ADD R10,R4,R3
L:  ADD R7,R8,R9
```


Compilers & static branch prediction

- Predict all branches taken
  - Surprisingly effective since 85% of backward branches and 60% of forward branches are taken
  - Still leaves more than a third of the branches improperly predicted
  - For some programs, this method is excellent (< 10% mispredictions), but for others, it does badly (> 50%)
- Predict forward not taken and backward taken
  - This scheme is similar to predicting all branches as taken except that it uses information about the types of branches.
  - Forward branches are usually part of if-else constructs, and may be less likely to be taken
  - Backward branches are often part of loops and more likely to be taken
  - Won't perform much better than simply predicting not-taken

Using profiling to predict branches

- Use profile information from previous runs
  - The compiler can instrument the code using the profile information from previous runs of the program.
  - It can build a higher performance program by predicting that branches taken in the practice run(s) will be taken in the final version.
- Not perfect since many branches are both taken and not taken in the course of execution.
  - Provides better prediction than other static methods.
  - Misprediction rates for this method range from 5% to 20%, even if different input data is used for the program

Wrapping up static branch techniques

- Studies have shown that profile-based prediction is almost always better than predict-taken or other non-profile-based methods
  - Since profile-based prediction is so good, why not use it?
  - Limits to static prediction
    - Branches behave differently at different times
    - Behavior can be very input-dependent
- Dynamic branch prediction provides a better solution
  - Predict branch behavior while program is running
  - Visit this topic in a week or two

Pipeline difficulties

- Why is pipelining difficult?
  - Now that we've seen how pipelining can be done and how to detect and resolve hazards, the question arises: what's so hard about this?
  - Exceptions
  - Instruction set complications
- Exceptions
  - An instruction in the pipeline can raise an exception that may force other instructions in the pipeline to be aborted
  - These other instructions may have altered the state of the machine.
  - Exceptions introduce the possibility that an exception in a later instruction (i.e. in ID or EX) will prevent a previous instruction (i.e. in MEM or WB) from completing

Exception causes

- I/O device requests
- User OS service requests
- Breakpoints
- Integer arithmetic overflow/underflow
- FP arithmetic anomaly
- Page fault
- Misaligned memory accesses
- Memory protection violations
- Hardware malfunctions
- Undefined instructions

Classifying exceptions

- Synchronous vs. asynchronous
  - Synchronous: exception comes as a result of execution at the same place for every run of a program with the same data and memory allocation
  - Asynchronous: generated external to the CPU
  - Asynchronous events can usually be handled after the completion of the current instruction, making them easier to handle.
- User requested vs. coerced
  - Did the user request an exception, i.e. through a trap?
  - Or did it happen as a result of something beyond the user program's control, i.e. a hardware event?
  - Coerced exceptions are harder to implement since they are not predictable

Classifying exceptions

- User maskable vs. non-maskable
  - Can the user prevent the hardware from responding?
  - Note that for maskable interrupts, the user can choose to respond to them, and therefore they are similar to non-maskable interrupts
  - Maskable interrupts must still be handled properly!
- Within vs. between instructions
  - Does the exception prevent instruction completion, by occurring in the middle of execution?
  - Or is it recognized between instructions?
  - Exceptions occurring within instructions are usually synchronous, since the instruction triggers the exception
  - **Within** is more difficult to implement than **between** since the former must be restarted

Classifying exceptions

- Resume vs. terminate
  - Terminate: the exception stops the program from running
  - Resumable: the program must be restartable after the interrupt
  - Restarting is harder (obviously), and is the more common case
  - The most difficult case is handling interrupts within an instruction, where the instruction must be resumed
    - Save the state of the executing program
    - Fix the cause of the exception
    - Restore the state of the original program, and restart it as if nothing had happened
    - Exceptions of this type occur for virtual memory management systems

$\Rightarrow$ Restartable instructions

Saving pipeline state

- For exceptions that occur within instructions (i.e. in EX or MEM) and must be restarted (page fault), the pipeline state must be saved
  - Insert a trap instruction into the pipeline on the next IF.
  - Turn off all writes for the faulting instruction and the instructions following it in the pipeline
  - Allow previous instructions to complete
  - Save the PC of the faulting instruction so it can be restarted (usually done by OS)
- This method requires as many PCs as there are delay slots
  - Instructions currently in the pipeline may not be sequentially related!
  - Save at least one PC value: the location of the faulting instruction

Precise vs. imprecise exceptions

- Precise exceptions mean:
  - All instructions before the faulting instruction complete
  - Instructions following the faulting instruction, including the faulting instruction, do not change the state of the machine.
- Restarting is easy with precise exceptions!
  - Simply re-execute the original faulting instruction
  - If it's not a resumable instruction, i.e. an integer overflow, start with the next instruction
- Precise exceptions can be difficult because of instruction completions and exceptions that occur out of order
  - Solution: imprecise exceptions.
  - Often used for floating point pipelines more so than integer pipelines
- Integer -> precise, FP -> imprecise (usually)

When do exceptions occur?

- IF
  - Page fault for instruction
  - Unaligned memory access
  - Memory protection fault
- ID
  - Undefined / illegal opcode
- EX
  - Arithmetic exception
- MEM
  - Page fault for data
  - Unaligned memory access
  - Memory protection fault
- WB: no faults

Exception ordering

- Two consecutive instructions cause exceptions in the same cycle
  - Which should be handled?
  - Handle the one belonging to the earlier instruction
- Cancel the later instruction (the ADD)
- Handle the page fault in the earlier (LW) instruction

```
LW  R4, 8(R5)      ; causes page fault for data
ADD R9, R10, R11   ; causes overflow
```

Cycle ->	1	2	3	4	5	6
LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

Exception ordering

- Page fault in ADD occurs **first**
- Must finish LW before handling the page fault in ADD if we want precise exceptions!
- Solution: keep an **exception vector**
  - Posted exceptions added to the vector, disabling writes for that instruction
  - Vector checked at end of each instruction...

```
LW  R4, 8(R5)      ; causes page fault for data
ADD R9, R10, R11   ; causes page fault for instr
```

Cycle ->	1	2	3	4	5	6
LW	IF	ID	EX	MEM	WB	
ADD		IF	ID	EX	MEM	WB

Exception vectors

- When the instruction is about to exit the pipeline (MEM/WB), any pending exceptions for the instruction are examined
  - If an instruction generates multiple exceptions, the exception occurring in the earliest stage takes precedence.
  - For the DLX, the faulting instruction has not updated any state (since all updates occur in WB)
- Many CPUs support both precise & imprecise exceptions
  - Precise exceptions are slower (often)
  - Imprecise exceptions are faster but don't note where the exception occurred (OK if the process will be killed anyway…)

Instruction set complications

- An instruction is **committed** when it is guaranteed to complete
  - In DLX, all instructions are committed at the end of MEM
  - Since no updates occur before instructions commit, precise interrupts are straightforward.
- In most RISC systems, each instruction writes only one result
  - Instruction can be cancelled any time before the instruction is committed, with no harm to the system state
  - Not true for many CISC machines, i.e. VAX: system state may be modified well before the instruction or its predecessors are committed
    - Example: instruction using autoincrement mode is aborted because of an exception, altering the machine state
    - It's difficult to restart the instruction or keep precise exception

Exceptions in CISC architectures

- The situation is worse for instructions that access and write memory in multiple places
  - These instructions can generate multiple faults.
  - Therefore, it becomes difficult to know where to resume
  - For string instructions, the CPU must also know how far into the operation it was when the exception occurred
- Usually solved by using general purpose registers as scratch space (that are saved and restored)
- General solution used by more complex instruction set machines is to pipeline the microcode.

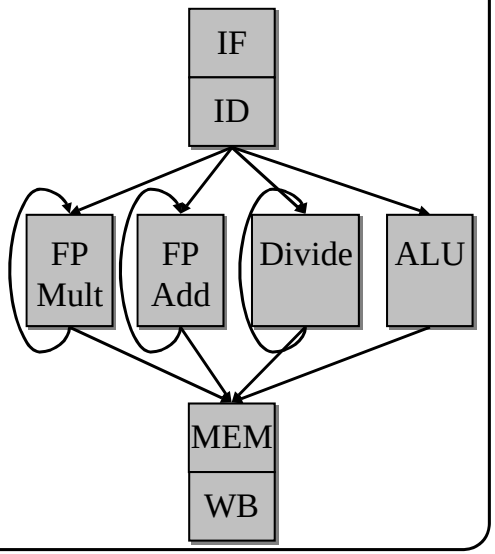
⇒ **RISC has often been compared to having the microcode as the actual assembly language**

Multi-cycle pipeline operations

- Many operations can't finish in one or two cycles
  - FP operations
  - Vector operations
- It might be possible to allow it, but would require
  - Slow clock
  - Lots of logic (more than we want to dedicate…)
- Use FP operations as an example (vectors later…)
- Implement several FP units (adder, multiplier, divider, etc.)
  - Allow a longer latency (several clocks)
  - May pipeline them to avoid structural hazards
  - One possible way to implement an FP pipeline is to allow the EX stage to repeat as many times as necessary

Non-pipeline FP in DLX

- FP multiplier, adder
- FP/integer divide unit
- FP units take multiple cycles
  - Non-pipelined
  - Structural hazards may occur if successive instructions use the same functional unit
- Structural hazards can cause stalls!



Pipelining the FP functional units

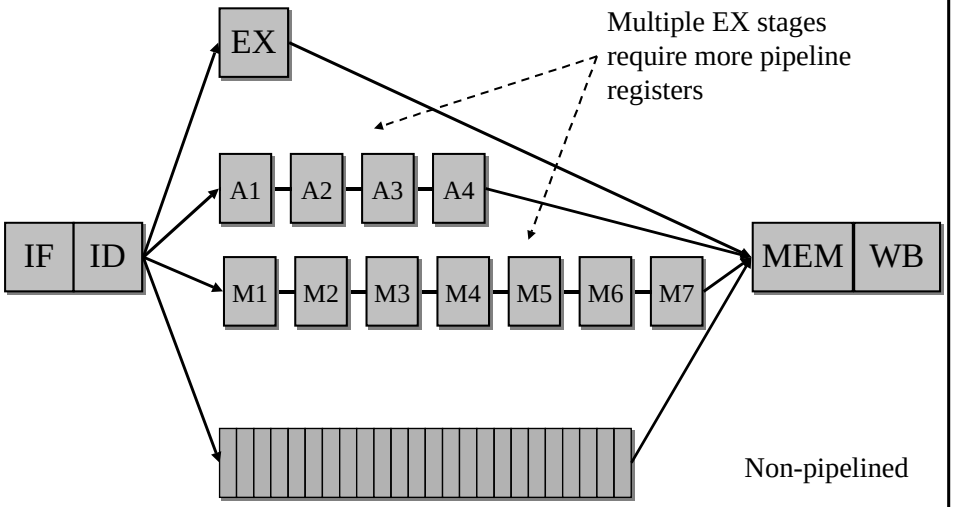
- Define the latency and the initiation interval for FP units
  - Latency => number of cycles needed beyond the first
    - Integer ALU has its result ready at the end of its first cycle
    - Integer ALU requires 0 *additional* EX stages
  - Initiation interval => number of cycles that must elapse between instruction issue to the same unit
- Pipelined units have
  - Various latencies
  - Initiation interval of 1
    - May issue a new operation every cycle
    - May have multiple outstanding operations!

Sample pipeline unit parameters

Functional unit	Latency	Initiation interval	# of pipe stages
Integer ALU	0	1	1
Data memory	1	1	1
FP add	3	1	4
FP/int multiply	6	1	7
FP/int divide	24	25	1

- Integer ALU: 0 cycle latency - result can be used on next clock cycle
- Data memory: 1 cycle latency, pipelined - result can be used after one intervening cycle
- FP add: 3 cycle latency, pipelined
- FP/integer multiplier: 6 cycle latency, pipelined
- FP/integer divide and FP square root: 24 cycle latency, non-pipelined

Pipeline including FP units



Pipeline characteristics

- Structural hazards can occur for the divide unit
- Several instructions can reach WB in a single cycle because of variable length instructions
  - More than 1 register write could occur in one cycle

  ⇒ Structural hazard if the CPU can only write one register per cycle
- Instructions no longer necessarily complete in the order in which they were issued (out-of-order completion)
  - **WAW** hazards are now possible and must be detected.
  - More problems with exceptions: the exception handling mechanism relied on the fact that instructions completed in issue order
- Since instructions have longer latencies, stalls for **RAW** operations will be more frequent

RAW hazards in the FP pipeline

- MULTD stalls due to load latency.
- ADDD stalls until multiply produces F0 value, which is forwarded.
- SD stalls in MEM waiting on result from ADDD

```
                1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
LD    F4,0(R2)  IF ID EX MM WB
MULTD F0,F4,F6     IF ID -  M1 M2 M3 M4 M5 M6 M7 MM WB
ADDD  F2,F0,F8        IF -  ID -  -  -  -  -  -  A1 A2 A3 A4 MM WB
SD    0(R2),F2           -  IF -  -  -  -  -  -  ID EX -  -  -  MM
```


Contention for the register write port

- Assume the FP register file has only one write port
- In this example, three instructions write to it in one cycle: hazard!
- Possible solutions
  - Increase the number of write ports: may not be worth it if the situation doesn't happen very often
  - Serialize the writes (stall instructions conflicting for resource)

```
                1  2  3  4  5  6  7  8  9  10 11
MULTD F0,F4,F6  IF ID M1 M2 M3 M4 M5 M6 M7 MM WB
ADD   R1,R2,#1     IF ID EX MM WB
SUB   R4,R4,#4        IF ID EX MM WB
ADDD  F2,F0,F8           IF ID A1 A2 A3 A4 MM WB
OR    R8,R8,#8              IF ID EX MM WB
OR    R9,R9,#1                 IF ID EX MM WB
LD    F8,0(R7)                    IF ID EX MM WB
```


Serializing writes to the FP registers

- Solving the write port structural hazard through serialization can be done in two ways
- Stall the instruction when it tries to enter the MEM or WB stage.
  - \+ Easy to detect the conflict at this point
  - – Complicates pipeline control since stalls can now occur in two places
- Keep track of when each instruction will use the WB stage and stall instructions in ID if their "slot" is already in use
  - Can be done using a shift register that tracks when already-issued instructions will use the register file
  - \+ Instructions are stalled only in ID
  - – Requires additional hardware (shift register and write conflict logic)

WAW hazards

- Consider situation below: WAW hazard since LD writes F0 one cycle earlier then MULTD
  - ONLY a hazard when MULTD is overwritten without any instruction ever using it -- it appears to be a useless instruction
  - If there was a use between MULTD and LD, then a RAW hazard would stall the pipeline and the WAW would not occur
- However, we must still detect them since they do occur in reasonable code (as we will see).

```
                1  2  3  4  5  6  7  8  9  10 11
MULTD F0,F4,F6  IF ID M1 M2 M3 M4 M5 M6 M7 MM WB
ADD   R1,R2,#1     IF ID EX MM WB
SUB   R4,R4,#4        IF ID EX MM WB
OR    R8,R8,#8           IF ID EX MM WB
LD    F0,0(R7)              IF ID EX MM WB
```

Dealing with WAW hazards

- The WAW hazard can be handled in one of two ways
  - Stall an instruction that would "pass" another until after the earlier instruction reaches the MEM phase
  - Cancel the WB phase of the earlier instruction
- Either can be done in ID, i.e. when LD is about to issue
- Pure WAW hazards are not common => use either method
  - Pick the one that simplest to implement.

⇒ Simplest solution for the DLX pipeline is to hold the instruction in ID if it writes the same register as an instruction already issued

Hazard checking during ID

- Possible sources of hazards
  - Hazards among FP instructions (already handled)
  - Hazards between an FP instruction and an integer instruction
- Separate FP & integer register files => only FP loads and stores and FP register moves to integer registers involve hazards
- Assume all hazards are detected during ID
- Three checks are required in ID (before instruction issue)
  - Check for structural hazards
    - Divide unit
    - Register write port

Hazard checking during ID

- Check for RAW hazards: CPU stalls the instruction at ID stage until
  - Its source registers are no longer listed as destinations in any of the execution pipeline registers (registers between stages of M and A) OR
  - Its source registers are no longer listed as the destination of a load in the EX/MEM register
- Check for WAW hazards
  - Check instructions in A1, ..., A4, Divide, or M1, ...,M7 for the same destination register (check pipeline registers)
  - Stall instruction in ID if necessary
- Instructions after the current one might have been able to execute, but they'll all have to wait (until the next chapter...)

FP pipelining & exception handling

- Exceptions are difficult because instructions may now finish out of order

```
DIVF F0, F2, F4
ADDF F10, F10, F8
SUBF F12, F12, F14
```

- ADDF and SUBF are expected to complete before DIVF
- Suppose SUBF causes an arithmetic exception at a point where ADDF has completed but DIVF has not
  - Imprecise exception!
  - Fix here is to let pipeline drain
- Suppose DIVF has an exception after ADDF completes
  - ADDF destroys one of its operands => can not restore the state to what it was before the DIVF instruction, even with software!

Handling exceptions

- Ignore the problem (imprecise exceptions)!
  - This may be fast and easy, but it's difficult to debug programs without precise exceptions
  - Many modern CPUs,, provide a precise mode that allows only a single outstanding FP instruction at any time (DEC Alpha 21064, IBM Power-1, MIPS R8000)
  - Precise mode is much slower than the imprecise mode!
- Buffer the results and delay commitment: CPU doesn't actually make any state (register or memory) changes until the instruction is guaranteed to finish
  - Becomes difficult when the difference in running time among operations is large.
  - Lots of intermediate results have to be buffered (and forwarded…)

Handling exceptions: save values

- History file: saves the original values of the registers that have been changed recently
  - If an exception occurs, the original values can be retrieved
  - File must have enough entries for one register modification per cycle for the longest possible instruction
  - Similar to the solution used for the VAX for autoincrement and autodecrement addressing
- Future file
  - This method stores the newer values for registers
  - When all earlier instructions have completed, the main register file is updated from the future file
  - On an exception, the main register file has the precise values for the interrupted state

Handling exceptions: save pipeline state

- Keep enough information for the trap handler to create a precise sequence for the exception
  - Instructions in the pipeline and the corresponding PCs must be saved.
  - After the exception, the software finishes any instructions in the pipeline that precede the latest instruction completed
- State must be saved by hardware with software assist
- Technique used in the SPARC

```
Instruction0     ; causes exception
Instruction1     ; not completed
Instruction2     ; not completed
…
Instructionn     ; not completed
Instructionn+1   ; not yet started
```

Pipeline state saved

Handling exceptions: delay issue

- Allow instruction to issue only if it is known that all previous instructions will complete without causing an exception
  - Floating point function units must determine if an exception is possible early in the EX stage
    - Necessary to keep the pipeline flowing smoothly (avoid stalls)
    - Pipeline may need to be stalled in order to maintain precise interrupts
  - Solution may cause unnecessary stalls by delaying a sequence of instructions...
- R4000 and Pentium use this solution

ISA and pipelining

- Avoid variable instruction lengths and running times whenever possible
  - Variable length instructions complicate hazard detection and precise exception handling
    - Sometimes it is worth it because of performance advantages such as caching, but this can cause instruction timings to vary
    - Added complexity may be handled by freezing the pipeline
- Avoid sophisticated addressing modes
  - Addressing modes that update registers (post-autoincrement)
    - Complicate exceptions and hazard detection
    - Make it harder to restart instructions
  - Allowing addressing modes with multiple memory accesses also complicates pipelining

ISA and pipelining

- Don't allow self-modifying code
  - Instruction being modified may already be in the pipeline => address being written must constantly be checked
  - Conflict => pipeline must be flushed or the instruction updated!
  - Even if it's not in the pipeline, it could be in the instruction cache..
- Avoid implicitly setting condition codes in instructions
  - Harder to avoid control hazards => impossible to determine if condition codes are set on purpose or as a side effect
  - Implementations that set the CC almost unconditionally make instruction reordering difficult => hard to find instructions that can be scheduled between the condition evaluation and the branch.

Sample pipeline: MIPS R4000

- IF: first half of instruction fetch
  - PC selection occurs
  - Cache access is initiated
- IS: second half of instruction fetch.
  - Allows cache access to take two cycles
- RF: decode and register fetch
  - Hazard checking
  - I-cache hit detection
- EX: execution
  - Address calculation
  - ALU Ops
  - Branch target calculation
  - Condition evaluation.
- DF/DS/TC: data memory
  - Data fetched from cache in the first two cycles
  - The third cycle involves determine if it was a cache hit
- WB: write back
  - Write result for loads and R-R operations

Stalls & delays in the MIPS R4000

- Load delay: two cycles
  - Delay might seem to be three cycles, since the tag isn't checked until the end of the TC cycle
  - However, if TC indicates a miss, the data must be fetched from main memory and the pipeline is backed up to get the real value
- Branch delay: three cycles (including one branch delay slot)
  - Branch is resolved during EX, giving a 3 cycle delay
  - First cycle may be
    - Regular branch delay slot (instruction always executed)
    - Branch-likely slot (instruction cancelled if branch not taken)
  - MIPS uses a predict-not-taken method presumably because it requires the least hardware

Effects of longer pipeline in MIPS R4000

- Disadvantages of a longer pipeline
  - Longer (and possibly more frequent) stalls
  - Additional forwarding hardware
  - More complex hazard detection to find dependencies in the additional stages
- Benefits of longer pipeline
  - Each stage may be shorter
    - Clock cycle can be shorter

    ⇒More instructions can be issued in a fixed time
- Do added stalls might eat up this benefit?

⇒ Hopefully, at least some speedup will be left

Performance issues in MIPS R4000

- Ideal CPI for the pipelined CPU is 1
  - Biggest contributor to stalls is branch stalls
  - Load stalls contribute very little (compiler can usually reorganize code to avoid stalling on loads)
  - Load latency is two cycles => job is harder than it might be on processors with a single-cycle latency

Pipelining improves performance!

- Pipelining is one of the main tools designers use to improve performance!
  - Allows the CPU to issue one instruction per cycle even when finishing an instruction takes many cycles
  - Allows faster and faster cycle times by spreading work over many cycles
- Pipelining has been the major factor allowing consumer-level microprocessors to run at 500 MHz or higher
- Next few weeks: more ways to squeeze performance out of the CPU, such as
  - Dynamic optimizations
  - Multiple instructions per cycle

Instruction level parallelism

- Potential overlap among instructions is called ILP
  - ⇒ Implies a lack of dependence between instructions.
  - All of the techniques in this chapter exploit parallelism among *instruction sequences*
- ILP techniques include
  - Static techniques (done by compiler)
  - Basic dynamic scheduling (done by hardware)
  - Additional "hidden" registers (done by hardware)
  - Branch prediction
  - Superscalar execution

How much can ILP help?

- Goal: execute as many instructions as possible in as little time as possible
  - Keep functional units busy with useful work
    - Execute instructions out of order
    - Use other techniques to do as much as possible at one time
  - Allow more functional units to be useful
- Limits to ILP
  - Processor: ILP limited by number and type of functional units
  - Program: interdependence of instructions can limit the instructions that can be done in parallel

ILP and CPI

- How is CPI calculated?
  CPI = ideal CPI + structural stalls + RAW stalls + WAW stalls + WAR stalls + control stalls
- Previous chapter: reduce RAW & control stalls
- This chapter: reduce all components of the CPI equation
  - Additional reductions in RAW & control stalls
  - Reduce ideal CPI
  - Use more hardware to reduce or eliminate structural stalls

Techniques for reducing CPI via ILP

- Loop unrolling (reduces control stalls)
- Basic pipeline scheduling (reduces RAW stalls)
- Scoreboarding (reduces RAW stalls)
- Register renaming (reduces both WAR and WAW stalls)
- Dynamic branch prediction (reduces control stalls)
- Issuing multiple instructions per cycle (reduces ideal CPI)
- Compiler dependence analysis (reduces ideal CPI and data stalls)
- Software pipelining and trace scheduling (reduces ideal CPI and data stalls)
- Speculation (execute "possible" instructions, reducing data & control stalls)
- Dynamic memory disambiguation (reduce RAW memory stalls)

Where does ILP come from?

- ILP comes from basic blocks
  - Block of code with no branches into the code except at the start and no branches out of the code except at the end
- Code inside the average basic block is quite small
  - Average dynamic branch frequency in integer programs ≈ 15%
  - About 6 to 7 instructions are executed between a pair of branches
    - Average = usual case?
    - Instructions in basic block depend on one another because they tend to operate on the same data in sequence

⇒ Exploit ILP across multiple basic blocks!

Loop-level parallelism

- Exploit parallelism among iterations of a loop
  - Iterations of a loop are often independent
  - Each iteration can overlap with any other iteration even though individual iterations have few (if any) overlappable instructions
- Techniques exist for exploiting the ILP in loops
  - Done statically by the compiler (loop unrolling)
  - Done dynamically by the CPU
  - Vector processors can run very quickly on simple loop operations

```
for (j = 0; j < 2000; j++) {
  dp[j] = x[j] * y[j] + z[j];
}
```


Pipeline scheduling

- Compiler tries to separate a dependent instruction from the source instruction so there's no data stalls
  - Compiler must have intimate knowledge of the internal hardware workings
  - Code optimized for one version of a processor may not be optimized on a future version of the processor...
- Assume the following latencies:

Instruction producing result	Instruction using result	Latency (clock cycles)
FP ALU operation	Another FP ALU op	3
FP ALU operation	Store double	2
Load double	FP ALU op	1
Load double	Store double	0 (using forwarding)

Pipeline scheduling example: before

- Compile the following code:

```
for (j = 0; j<2000; j++)
  x[j] = x[j] + c;
```

  - Assume R1 holds the address of x[1999]
  - Assume F2 has the scalar value c
- Unscheduled code has many stalls!
  - 5 cycles of useful work
  - 5 cycles of stalls!
  - Total = 10 cycles per iteration

```
Loop:
LD   F0,0(R1)   ; F0=array elem
ADDD F4,F0,F2   ; add scalar
SD   0(R1),F4   ; store result
SUBI R1,R1,#8   ; pointer--
BNEZ R1,Loop    ; repeat loop
```

```
Loop:
LD   F0,0(R1)   ; F0=array elem
STALL
ADDD F4,F0,F2   ; add scalar
STALL
STALL
SD   0(R1),F4   ; store result
SUBI R1,R1,#8   ; pointer--
STALL
BNEZ R1,Loop    ; repeat loop
STALL
```

Pipeline scheduling: after

- Reschedule code to reduce stalls
  - Move SUBI earlier
  - Move SD later (and fix address)
- Still one stall
  - SD stalls 1 cycle waiting for ADDD result
- Reduced total time from 10 -> 6
  - Still only 3 cycles of work!
  - 2 instructions & 1 stall overhead
- Goal: get more "useful" operations per loop overhead

⇒ Replicate the loop body multiple times and adjust loop control

```
Loop:
LD   F0,0(R1)   ; F0=array elem
ADDD F4,F0,F2   ; add scalar
SD   0(R1),F4   ; store result
SUBI R1,R1,#8   ; pointer--
BNEZ R1,Loop    ; repeat loop
```

```
Loop:
LD   F0,0(R1)
SUBI R1,R1,#8
ADDD F4,F0,F2
BNEZ R1,Loop
[STALL]SD   8(R1),F4
```


Loop unrolling

- Loop unrolling => create multiple copies of the loop body
  - Improves scheduling by
    - Eliminating branches (control hazards cost time!)
    - Allowing instructions from multiple iterations to be interleaved, exposing more parallelism
  - Allows CPU to amortize loop overhead across several loop iterations
    - Comparison at end of loop
    - Pointer / index increments
- Loop unrolling increases register usage
  - Better utilization of a scarce resource
  - More chance for cycles between uses of a register to be filled

Loop unrolling: example

- Unroll & schedule code from previous example
  - Unroll 4 times
  - Use displacement addressing mode to increment index once per "macro" loop
  - Assume R1 *MOD* 32 == 0
- Code has no stalls!
  - 14 clock cycles for 4 elements => 3.5 clock cycles / element
  - Speedup of 6/3.5 = 1.7x
  - Using different registers => avoid *false dependencies*
  - Reordering code eliminates stalls!

```
loop:
 LD   F0,0(R1)
 LD   F6,-8(R1)
 LD   F10,-16(R1)
 LD   F14,-24(R1)
 ADDD F4,F0,F2
 ADDD F8,F6,F2
 ADDD F12,F10,F2
 ADDD F16,F14,F2
 SD   0(R1),F4
 SD   -8(R1),F8
 SUBI R1,R1,#32
 SD   16(R1),F12
 BNEZ R1,loop
 SD   8(R1),F16
```


Loop unrolling: details

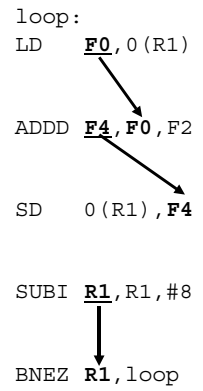
- What if loop index in example isn't a multiple of 32?
  - Real code: don't always know upper bound on loop
  - Real code: don't know how many times the loop will be executed!
- Solution: assume loop unrolled *k* times and iterated *n* times
  - First code body contains original code, and executes *n* MOD *k* times
  - Second code contains unrolled body and executes $\lfloor n/k \rfloor$ times
  - Saves time if the number of iterations was large
  - Another solution: jump into the middle of the code (if possible)
- Loop unrolling is easy to recognize
  - Not trivial for a compiler to perform these optimizations!
  - Compilers can, however, do scheduling ***very*** well

Dependencies: the basics

- Dependency => instruction B uses a result of instruction A
  - Dependencies are a property of programs, not of CPUs and pipelines.
  - Dependence between two instructions will always exist unless the program is changed
- Presence of a dependence indicates the *potential* for a hazard
  - Actual hazard and the length of any stall is a property of the pipeline
  - Goal is to eliminate stalls, not dependencies!
- Three types of dependencies
  - Data
  - Name
  - Control

Data dependence

- Instruction *j* is dependent on *i* if *i* produces a result used by *j*
- Dependence is transitive
  - *j* is dependent upon *i* and *k* is dependent upon *j*, => *k* is dependent on *I*
- Dependence chains can be arbitrarily long!
- A compiler scheduling instructions cannot move *j* before *i* if *j* depends upon *i*



Overcoming data dependencies

- Data dependencies
  - Indicate the possibility of a hazard
  - Determine the order in which results must be generate
  - Place an upper bound on the amount of ILP available
- Data dependencies can be overcome in two ways
  - Keeping the dependence but avoiding a hazard
  - Eliminating the dependence by transforming the code
- Scheduling is the primary way to **avoid hazards** without altering dependencies
  - See previous example with LD, ADDD and SD
  - Code scheduled to avoid the hazard, but the dependence remained in the code

Eliminating data dependencies

- It's possible to eliminate data dependencies
  - Eliminate instructions!
  - Loop unrolling: eliminate branches and index updates
- Compiler removes dependence by eliminating instructions
  - BNEZ instructions dropped
  - Eliminate SUBI instructions & fold computation into offset

```
Loop:
LD   F0,0(R1)   ; F0=array elem
ADDD F4,F0,F2   ; add scalar
SD   0(R1),F4   ; store result
SUBI R1,R1,#8   ; pointer--
BNEZ R1,Loop    ; repeat loop
LD   F6,0(R1)   ; F0=array elem
ADDD F8,F6,F2   ; add scalar
SD   0(R1),F8   ; store result
SUBI R1,R1,#8   ; pointer--
BNEZ R1,Loop    ; repeat loop
LD   F10,0(R1) ; F0=array elem
ADDD F12,F10,F2; add scalar
SD   0(R1),F12 ; store result
SUBI R1,R1,#8   ; pointer--
BNEZ R1,Loop    ; repeat loop
```

More on data dependencies

- Eliminating data dependencies requires a fair amount of analysis => done by the compiler
- Avoiding hazards through scheduling can be done in hardware or software or both
- What about data dependence through a memory location?
  - Registers are easy to figure out at compile time
  - Memory dependences may not be known until runtime => much more difficult to deal with!
  - Example: 100(R4) and 20(R6) may refer to the same memory location
    - Not known until runtime, though!
  - Explore hardware and software techniques that detect data dependencies that involve memory locations (later…)

Name dependencies

- Two instructions use the same register or memory location, but there's an intervening write to it
  - These are ***NOT*** data dependences because no information is passed between the two instructions
  - The instructions could be executed out of order or in parallel if the CPU renamed the register or memory location involved
- Register renaming can either be done by
  - Compiler, as in earlier loop unrolling
  - CPU (dynamic register renaming)
- In example
  - Second LD replaces value in F0
  - Second ADDD replaces value in F4

```
Loop:
LD   F0,0(R1)
ADDD F4,F0,F2
SD   0(R1),F4
SUBI R1,R1,#8
LD   F0,0(R1)
ADDD F4,F0,F2
SD   0(R1),F4
SUBI R1,R1,#8
```

Unrolled loop before register renaming

Control dependencies

- A control dependency determines the ordering of the instructions with respect to branch instructions
  - If an instruction depends on the outcome of an earlier branch then it is only executed on one of the two forks
  - This instruction is dependent on the preceding branch
- Example:

```
if (cond1)
  S1;
else
  S2;
```

- Obviously, we cannot:
  - Move S1 or S2 before the *if* statement
  - Move other instructions before the *if* stmt into "then" or "else"

Control dependencies

- In loop unrolling before removing branches:
  - Control hazards after each branch!
- Eliminated branches because we knew the outcome of each branch
  - Iterations divisible by 4
  - All "internal" branches taken (to top of loop)
  - Eliminated control dependencies!

```
Loop:
LD   F0,0(R1)
ADDD F4,F0,F2
SD   0(R1),F4
SUBI R1,R1,#8
BNEZ R1,Loop
LD   F6,0(R1)
ADDD F8,F6,F2
SD   0(R1),F8
SUBI R1,R1,#8
BNEZ R1,Loop
...
```

Preserving program correctness

- Preserving control dependence is NOT a critical property
  - Program can be rewritten to violate control dependence!
  - Program correctness is the critical property that must be preserved
- Violating control dependence may be OK if program correctness is preserved!
- Two properties critical to program correctness are
  - Preserving exception behavior: any changes in the ordering of instructions must NOT change how exceptions are raised
    - An instruction that should not have been executed can't cause an exception
    - Memory operations and floating point often cause problems like this
  - Preserving data flow

Preserving data flow

- Branches make the flow of information between instructions dynamic
  - Different values for particular registers depend on whether or not branches are taken
  - This information flow must be preserved!
- Data flow can be preserved by
  - CPU cancels instructions that were wrongly executed
  - Compiler cancels things out (add to cancel out a subtract that shouldn't have been executed)

Dealing with control dependencies

- Sometimes, violating control dependencies can't affect execution behavior or data flow

```
ADD   R1,R2,R3
BEQZ  R12,skipnext
SUB   R4,R5,R6
ADD   R5,R4,R9
skipnext:
OR    R7,R8,R9
```

- Could move SUB before BEQZ if we knew
  - The SUB instruction could not generate an exception
  - If R4 were not 'live', i.e., used after the skipnext label
- This type of scheduling is called speculation: the compiler is betting that the branch will not be taken
  - Hardware can do this too...

Control dependencies: summary

- Control dependence is preserved by implementing control hazard detection
- Control hazard detection causes control stalls
- Control stalls can be avoided by:
  - Scheduling instructions in delay slots
  - Loop unrolling
  - Conditional execution
  - Speculation by both compiler and CPU
- We will cover the latter two shortly along with other dynamic methods for taking advantage of ILP

Dynamic scheduling

- Last time: data hazards that prevent instruction issue were hidden by:
  - Forwarding
  - Static scheduling by the compiler
- Dynamic scheduling is also possible:
  - CPU rearranges the instructions (while preserving dependences) to reduce stalls
- Dynamic scheduling has several advantages over static
  - Handles dependencies that are ***UNKNOWN*** at compile time such as
    - Memory references
    - Branches
  - Allows code compiled with one pipeline in mind to run efficiently on a different pipeline

Out-of-order execution: basics

- Until now, all techniques require in-order instruction issue
  - A stalled instruction holds up those behind it
- What if following instructions could "pass" the stalled one?

```
DIVD F0,F2,F4     ; long latency
ADDD F10,F0,F8    ; stalled waiting for F0
SUBD F12,F8,F14   ; could proceed with this one!
```

- Out-of-order execution: allow instructions to issue in any order as long as dependencies aren't violated
  - Execute SUBD before ADDD in above example, reducing stalls
  - Handle out-of-order completion
    - May cause problems handling exceptions
    - May not gain if there are long dependence chains

Implementing out-of-order execution

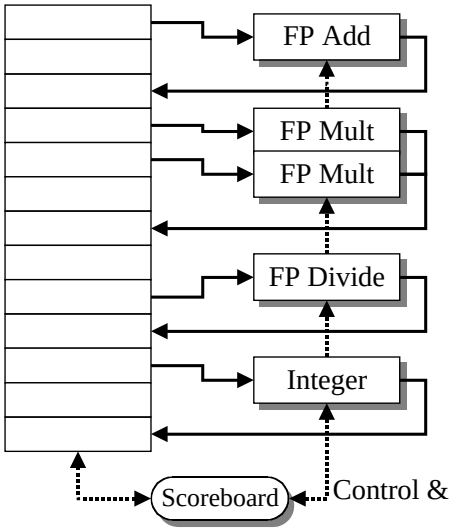
- Split the ID stage into two halves
  - Issue: decode instructions and check for structural hazards
  - Read operands
    - Wait until there are no data hazards, then
    - Read the operands
  - Continue to use forwarding to remove data hazards
- Designs of this type may use an instruction queue to hold instructions that have been fetched but are waiting to be executed
  - An instruction is considered to be in **execution** at any time that it's in an EX stage
  - Multiple instructions can be in execution at any given time

Scoreboarding

- This technique issues instructions in order (in-order issue)
  - Instructions can pass other waiting instructions in the "read operands" phase
  - WAR hazards are now possible (didn't exist in previous pipelines)
- Scoreboarding first used in the CDC 6600 (the designers named it)
- Goal: maintain an execution rate of one instruction per cycle
  - Execute instructions as soon as possible
  - Use either multiple functional units or pipelined functional units (they're equivalent for the purposes of pipeline control)
  - We'll assume multiple functional units

```
DIVD F0,F2,F4   ; divide takes a long time
ADDD F10,F0,F8  ; stalled waiting for F0 from divide
SUBD F8,F8,F14  ; stalled waiting for ADDD to read F8 (WAR)
```

DLX implementation using a scoreboard



- Focus on analysis of scoreboarding in the FP units
- Integer units rarely encounter hazards!
  - Only stalls when waiting for a value that has just been loaded
  - Don't deal with integer hazards for now...

Pipeline changes for scoreboarding

- Every instruction goes through the scoreboard
  - Scoreboard determines when an instruction can read its operands and write its results

  ⇒All hazard detection and resolution is centralized
- ID stage replaced with two stages:
  - Issue (IS)
    - An instruction is issued if:
      - The functional unit is available and
      - No other active instruction has the same destination register
    - This avoids WAW hazards and structural hazards
    - During a stall, this causes the buffer between IF and IS to fill
      - A one-entry buffer fills quickly!
  - Read operands (RD)

More pipeline changes for scoreboarding

- Read operands (RD)
  - Read operation is delayed until operands are available

    ⇒No previously issued but uncompleted instruction has the operand as its destination
  - RAW hazards resolved dynamically
- Execution (EX) stage changed
  - Notify the scoreboard when EX is completed
    - Allow a new instruction to use the functional unit
  - EX may take multiple cycles if necessary

Writeback (WB) with scoreboarding

- The scoreboard checks for WAR hazards and stalls the completing instruction if necessary
  - In the earlier example, SUBD would be stalled in WB until ADDD reads its operands
- Writeback is stalled if
  - A preceding instructions has not read its operands and
  - One of the operands is the same register as the destination of the completing instruction
- The DLX pipeline is now six cycles long

  `IF IS RD EX MEM WB`
  - Forwarding is not used here: not a large penalty since write-back occurs as soon as the result is available
  - Instructions that do NOT need the MEM stage don't execute it

Scoreboard components

- Instruction status
  - Keeps track of the current stage of each instruction
  - There is one entry for each instruction that has passed the IF stage but has not yet completed
- Functional unit status
  - Holds the status of each functional unit
    - "Busy" indicates whether or not the unit is busy
    - "Op" indicates the operation being performed (some functional units can do more than one operation)
  - F_i, F_j and F_k indicate the instruction's source and destination registers
  - Q_j and Q_k indicate the functional units producing the instruction's source registers
  - R_j and R_k indicate whether the values are ready (avoid WAR hazards)

Scoreboarding: sample code

- Register result status
  - Holds the ID of the functional unit that will eventually write a register
  - If the register is not the destination of an issued instruction, the field will indicate no functional unit
- For this example, use the code on the right
  - Examine snapshots of the three components of the scoreboard during execution
  - See how hazards are handled

```
LD     F6,40(R2)
LD     F2,52(R3)
MULTD  F0,F2,F4
SUBD   F8,F6,F2
DIVD   F10,F0,F6
ADDD   F6,F8,F2
```

```
F0: RAW hazard
F2: RAW hazard
F6: RAW hazard
F8: RAW hazard
```


Scoreboard: snapshot #1

Instruction status

Instruction		Issue	Read operands	Exec complete	Write result
LD	F6,40(R2)	X	X	X	X
LD	F2,52(R3)	X	X	X	
MULTD	F0,F2,F4	X			
SUBD	F8,F6,F2	X			
DIVD	F10,F0,F6	X			
ADDD	F6,F8,F2				

Mult & Sub waiting for WB

First load complete

Function unit status

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes	Load	F2	R3	-	-	-	No	-
Mult1	Yes	Mult	F0	F2	F4	Int	-	No	Yes
Mult2	No		-	-	-	-	-	-	-
Add	Yes	Sub	F8	F6	F2	-	Int	Yes	No
Divide	Yes	Div	F10	F0	F6	Mult1	-	No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	F12	…	F30
FuncUnit	Mult1	Int			Sub	Div			

Scoreboard: snapshot #2

Instruction status

Instruction		Issue	Read operands	Exec complete	Write result
LD	F6,40(R2)	X	X	X	X
LD	F2,52(R3)	X	X	X	X
MULTD	F0,F2,F4	X	X	X	
SUBD	F8,F6,F2	X	X	X	X
DIVD	F10,F0,F6	X			
ADDD	F6,F8,F2	X	X	X	

Why can't ADDD finish?

Function unit status

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No	-	-	-	-	-	-	-	-
Mult1	Yes	Mult	F0	F2	F4	-	-	No	No
Mult2	No	-	-	-	-	-	-	-	-
Add	Yes	Add	F6	F8	F2	-	-	No	No
Divide	Yes	Div	F10	F0	F6	Mult1	-	No	Yes

Register result status

	F0	F2	F4	F6	F8	F10	F12	…	F30
FuncUnit	Mult1			Add		Div			

Distinguishes RAW from WAR

Scoreboard: snapshot #3

Instruction status

Instruction		Issue	Read operands	Exec complete	Write result
LD	F6,40(R2)	X	X	X	X
LD	F2,52(R3)	X	X	X	X
MULTD	F0,F2,F4	X	X	X	**X**
SUBD	F8,F6,F2	X	X	X	X
DIVD	F10,F0,F6	X	**X**	**X**	
ADDD	F6,F8,F2	X	X	X	**X**

Function unit status

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	No	-	-	-	-	-	-	-	-
Mult1	Yes	Mult	-	-	-	-	-	-	-
Mult2	No	-	-	-	-	-	-	-	-
Add	Yes	Add	-	-	-	-	-	-	-
Divide	Yes	Div	F10	F0	F6	-	-	No	No

Register result status

	F0	F2	F4	F6	F8	F10	F12 … F30
FuncUnit						Div	

Handling hazards with a scoreboard

- RAW hazards
  - Detect RAW hazards by checking to see if a source register is listed in the Register Result Status table
  - ⇒ If it is, we have a RAW hazard
  - If the pending instruction is receiving a value from the current instruction, then set one of the pending instruction's R_j/R_k fields to **No**
- WAR hazards
  - Before writing the value, check to make sure that no pending instruction is using a previous value for the register to be modified
  - If some pending instruction has already "received" the value it needs but hasn't yet read it, then R_j/R_k is set to **Yes** and any instruction writing the register must stall (WAR)
- This is how we distinguish between a RAW and WAR

Scoreboard limitations

- ILP: if there aren't any independent instructions to execute, scoreboarding and other dynamic techniques don't help much
- Size of the "issued" queue (the **window**)
  - Determines how far ahead the CPU can look for instructions
  - For now, assume that a window cannot span a branch
    - Window includes instructions only within basic blocks
    - The window can be extended beyond the branch: details later
- Number, types, and speed of the functional units
- Presence of antidependences and output dependences
  - WAR and WAW hazards limit scoreboard more than RAW hazards
  - RAW hazards are problems for any technique
  - WAR and WAW hazards can be solved using other mechanisms

Tomasulo's approach

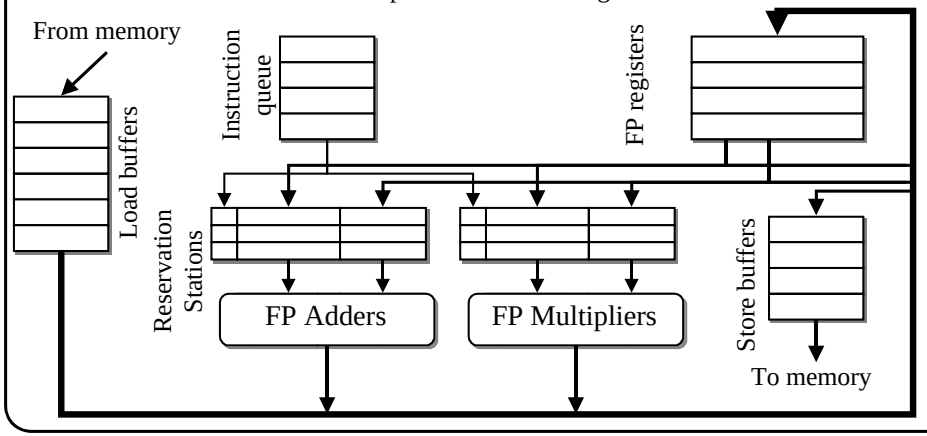
- Tomasulo's approach is a technique to allow execution to proceed in the presence of hazards
  - First introduced in the IBM 360/91
  - Applied only to floating-point operations (including FP memory ops)
- Uses renaming to avoid WAW and WAR hazards
  - Compiler can rename registers (statically) to avoid **WAW** and **WAR** hazards
  - Tomasulo's scheme performs this function dynamically
    - Buffers operands of instructions waiting to issue, fetching them as soon as they are available, avoiding the register file
    - The register specifiers of instructions are renamed to reservation station numbers as they are issued, *eliminating* **WAW** and **WAR** hazards

Scoreboarding vs. Tomasulo's approach

- Register renaming
  - Register renaming is used to eliminate **WAR** and **WAW** hazards
  - Scoreboarding must wait for **WAR** and **WAW** hazards to clear
- Distributed control
  - Hazard detection and execution control are distributed to each functional unit
  - Scoreboarding has a centralized control unit
- Common Data Bus
  - Used to forward results directly to the functional units without going through the register file
  - Scoreboarding connects each functional unit to the register file

Tomasulo's approach: design

- Reservation stations are the heart of Tomasulo's approach
  - Located at each functional unit (may be more reservations than func units)
  - Hold values for each computation before it begins



Tomasulo's approach: issue stage

- Take an instruction from the instruction queue
  - If there's a station available for it, send the instruction to the station
  - Otherwise, stall for a structural hazard
- This step checks to see if the source operands will be produced by a current instruction
  - If so, renaming is done by checking to see if the desired register is being written by an instruction already at a reservation station
    - If the value is not being generated by a functional unit, it is fetched from the register file
    - If the value is being generated, the name of the reservation station generating the result is used instead
  - If the operation is a load or a store, it can issue if there is an available load or store buffer

Tomasulo's approach: execute & WB

- Execute
  - If at least one operand is missing, monitor the CDB until it is generated
  - When a needed operand is put out onto the CDB, it is placed into the appropriate reservation station
  - When both operands are ready, the operation is executed

  ⇒ RAW hazards are handled here
- Write result
  - When the result is ready, write it on the CDB and into the register file and any waiting reservation station

    ⇒Only one value can be written on the CDB in any single cycle!
  - Indicate that the reservation station is no longer busy

Tomasulo's approach: design details

- Control structures
  - Operation (Op): the operation to be performed.
  - Operand sources (Q_j, Q_k): the reservation stations that will produce the values for the two operands
    - A 0 in either slot means the source operand is already in V_j or V_k, or that the slot is not needed
  - Operand values (V_j, V_k): the values for the two operands.
    - They are valid if and only if the corresponding Q is 0
  - Busy: indicates the reservation station and the accompanying functional unit are busy
- Register file & store buffer
  - Field Qi for each element: indicates which reservation station is producing the result that will go into this element (0 if blank)

Tomasulo's approach: control example

Instruction status

Instruction		Issue	Read operands	Exec complete	Write result
LD	F6,40(R2)	X	X	X	X
LD	F2,52(R3)	X	X	X	
MULTD	F0,F2,F4	X			
SUBD	F8,F6,F2	X			
DIVD	F10,F0,F6	X			
ADDD	F6,F8,F2	X			

MULTD & SUBD waiting for WB

DIVD waiting on MULTD

Function unit status

Name	Busy	Op	V_j	V_k	Q_j	Q_k
Add1	Yes	Sub	M[40+Regs[R2]]	-	-	Load2
Add2	Yes	Add	-	-	Add1	Load2
Add3	No	-	-	-	-	-
Mult1	Yes	Mult	-	Regs[F4]	Load2	-
Mult2	Yes	Div	-	M[40+Regs[R2]]	Mult1	-

Register result status

	F0	F2	F4	F6	F8	F10	F12 … F30
FuncUnit	Mult1	Load2		Add2	Add1	Mult2	

Tomasulo's approach: advantages

- Hazard detection logic is distributed
  - If multiple instructions are waiting on the second of two operands, the instructions can be released simultaneously broadcasting on the CDB
- WAW and WAR hazards are eliminated because
  - Register renaming is performed using the reservation stations.
  - Operands are stored into the reservation tables as soon as they are available
- The WAR hazard was eliminated because the reservation station held the value of F6 for the DIVD instruction
  - Even if **LD F6, 40(R2)** hadn't completed before the **DIVD** had issued
    - The **WAR** hazard & potential WAW hazard are eliminated
    - Q_k would point to the Load1 reservation table for the value of F6

Tomasulo's approach: loop unrolling

- Loop unrolling is performed dynamically !
- With only 4 FP registers, WAW and WAR hazards would severely limit loop unrolling, even by the compiler
  - Virtual registers provided by the reservation stations make it possible to execute multiple iterations of some loops simultaneously
- Memory disambiguation
  - Since the store functional unit keeps a memory address as well as a value, it's possible to do disambiguation
  - When a memory operation is issued, check to see if that location is already involved in an operation

⇒ LOADs and STOREs from different iterations of the loop can be executed *non-sequentially*

Dynamic branch prediction

- Until now, we have focused on overcoming data hazards.
- However, control hazards contribute greatly to reduced CPI, especially as pipelines become longer
- More evident for machines that issue multiple instructions per cycle (CPI < 1)
  - Branches arrive *n* times more frequently in a *n*-issue machine.
  - Latency of resolving a branch does not decrease
    - ⇒CPI is more significantly affected than it is for a single-issue machine

Effect of branch prediction on performance

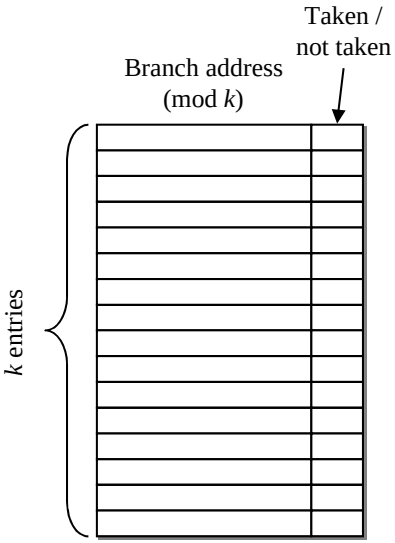
- Static vs. dynamic prediction
  - Static prediction: all decisions are made at compile time
    - ⇒This does not allow the prediction scheme to adapt to program behavior that changes over time
- Effects of prediction on performance:
  - Accuracy
    - Accuracy of a branch prediction scheme impacts CPU performance
    - A scheme that is not accurate may make CPU performance worse than it would be without prediction
  - Latency: two orthogonal aspects to performance
    - Branch may be **taken** or **not taken**
    - Branch may be **correctly predicted** or **incorrectly predicted**
    - ⇒Up to four different latencies for a single branch instruction

Predicting a branch's result

- The simplest thing to do with a branch is to predict whether or not it is taken
  - Helps in pipelines where the branch delay is longer than the time it takes to compute the possible target PCs
    - Most pipelines can calculate branch destination quickly!
  - By saving the decision time, the CPU can branch sooner
- This scheme does NOT help with the DLX
  - Branch decision and target PC are computed in ID, assuming there is no hazard on the register tested
  - Only helps when branch decision is calculated *after* branch target

Branch prediction buffer

- Keep a buffer (cache) indexed by the lower portion of the address of the branch instruction
  - Include bit(s) to indicate whether or not the branch was recently taken or not
  - If the prediction is incorrect, the prediction bit is inverted and stored back
- Branch direction could be incorrect because
  - Branch mispredicted
  - Instruction mismatch
  - ⇒ Either way, the worst outcome is paying the full branch latency

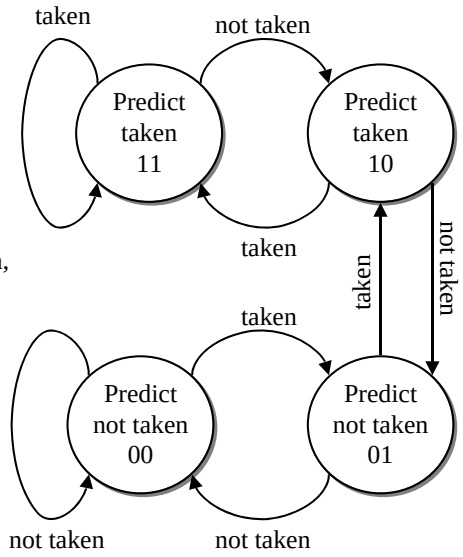


Improving prediction accuracy

- Sample code

```
for (j = 0; j < 10; j++) {
  for (k = 0; k < 10; k++) {
    // stuff
  } // Predict this branch
}
```

- Problem
  - If branch is *almost always* taken, this scheme will likely predict incorrectly twice
  - Mispredicts when j==0,k==10
  - Mispredicts when j==1,k==0
- Solution
  - ⇒ Use 2-bit predictor!



Multi-bit predictors

- 2-bit predictor scheme
  - Allows the accuracy of the predictor to approach the taken branch frequency (i.e. 90% for highly regular branches)
  - Implements "forgiveness" for a single misprediction
- *n*-bit predictors
  - Keep an *n*-bit saturating counter for each branch
    - Increment it on branch taken
    - Decrement it on branch not taken
  - If the counter is greater than or equal to half its maximum value, predict the branch as taken
  - This can be done for any *n*
- *n*=2 performs almost as well as larger values for *n*
  - ⇒ Use *n*=2 because it requires less hardware!

Location of the branch prediction buffer

- "Special cache"
  - Accessed during IF (with the PC)
  - Prediction bits used during ID if the instruction is decoded as a branch
- Instruction cache
  - Requires more space (the instruction cache is usually much larger than the "special cache")
  - Reduces the likelihood that "conflicts" occurs between different branches
- Accuracy of branch prediction
  - Misprediction rates range from **1%** to **18%** (using a 4K entry branch prediction buffer)
  - Static rates are around **30%** for many programs

Improving accuracy: correlated predictions

- The accuracy of our predictor is critical to exploiting more ILP
- How can we improve accuracy?
  - Increasing the size of the cache does not help (much)
  - Increasing the number of bits beyond 2 does not help (much)
- Consider the behavior of "surrounding" branches?
  - Works particularly well if there are common "paths" through code that require several branches, as in the following code:

```
if (aa == 2)  // B1
  aa = 0;
if (bb == 2)  // B2
  bb = 0;
if (aa != bb) … // B3
```

  - B3 is correlated with B1 and B2

    ⇒If both *if* statements are **TRUE**, then (aa != bb) is **FALSE**

Correlated branch predictors

```
if (d == 0)
  d = 1;
if (d == 1)
  ...
```

Assume **d** is held in R3

```
  BNEZ R3,label   ; Branch b1
  ADDI R3,R0,#1
label:
  SUBI R1,R3,#1
  BNEZ R1,label2  ; Branch b2
  …
label2:
```

Initial value of d	d==0	b1	Value of d before b2	d==1	b2
0	Yes	**Not taken**	1	Yes	**Not taken**
1	No	Taken	1	Yes	Not taken
2	No	Taken	2	No	Taken

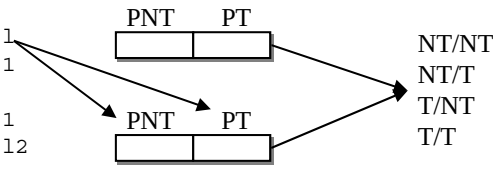
- If b1 is **not taken**, then b2 will also be **not taken**

⇒ A *correlating predictor* can take advantage of this

Correlated branch prediction

- Use two-level predictors
  - Keep track of the behavior of *previous* branches
  - Use history to predict the behavior of the *current* branch
- Implement this by assigning two bits to each branch instruction
  - One bit predicts the direction of the current branch if the previous branch was not taken (PNT)
  - One bit predicts the direction of the current branch if the previous branch was taken (PT)

```
  ...
  BNEZ R3,label
  ADDI R3,R0,#1
label:
  SUBI R1,R3,#1
  BNEZ R1,label2
  …
label2:
```



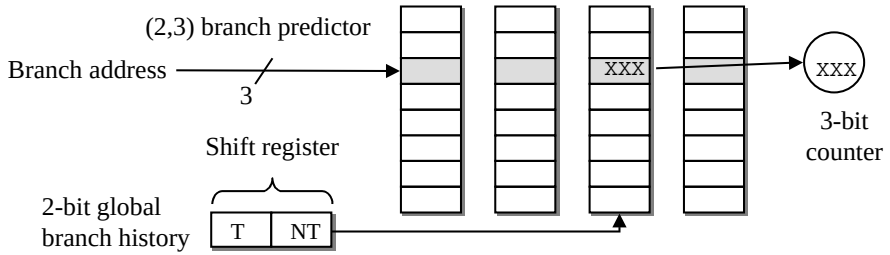
How do two-level predictors help?

- Assume the value of **d** alternates between 2 and 0 in a loop
  - The correct prediction of **b2** shows the advantage of correlating predictors
  - The correct prediction of **b1** is due to the choice of *d*, since there is no obvious correlation

```
if (d == 0)  // Branch b1
  d = 1;
if (d == 1)  // Branch b2
```

	d==?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
Simple branch prediction	2	NT	T →	T	NT ✗	T →	T
	0	T	NT →	NT	T ✗	NT →	NT
	2	NT	T →	T	NT ✗	T →	T
	0	T	NT →	NT	T ✗	NT →	NT
Correlated branch prediction	2	**NT**/NT	T	**T**/NT	NT/**NT**	T	NT/**T**
	0	T/**NT**	**NT**	T/**NT**	**NT**/T	**NT**	**NT**/T
	2	**T**/NT	**T**	**T**/NT	NT/**T**	**T**	NT/**T**
	0	T/**NT**	**NT**	T/**NT**	**NT**/T	**NT**	**NT**/T

(m,n) branch predictors



- (*m*,*n*) predictors use the behavior of the last *m* branches to choose from one of 2^m branch predictors, each of which is an *n*-bit predictor
  - ⇒ Results in better prediction rates than conventional n-bit prediction because it allows several "contexts"
- Global Branch History can be implemented using a shift register that shifts in the branch behavior (**not taken** or **taken**) when the branch is executed
- Since the branch prediction buffer is **NOT** a cache, there's no *guarantee* that the predictions correspond to the "correct" branch instruction

Branch target buffers

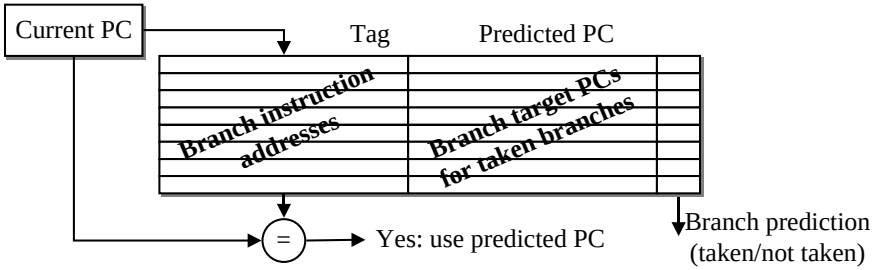
- Branch predictors help predict whether a branch is taken
- CPU needs to know which address to fetch from ASAP in order to reduce stalls even further, ideally to 0
  - Must do this even before the CPU knows the instruction is a branch

⇒ Use ***branch target buffer (BTB)*** (also called ***branch target cache***)

- A branch target buffer is very similar to a cache
  - Indexed exactly like a cache!
    - BTB must include a tag to catch collisions in the table
  - "Value" in the cache is the address of the next instruction, not the contents of the memory location

Branch target buffer operation

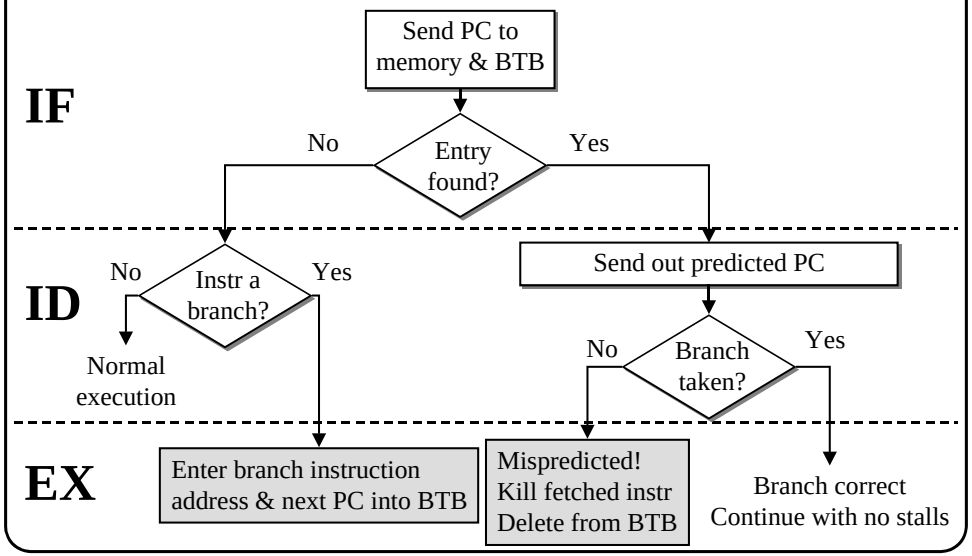
- If a hit occurs in the BTB, the CPU fetches the next instruction from the address stored in the BTB, and not PC + 4
  - This occurs by the end of IF!
  - CPU must compare the entire address (unlike prediction buffers)
- On an incorrect match(current instruction is NOT a branch instruction)
  - Slow things down because the predicted PC is always non-sequential by definition (and therefore, incorrect)



Adding prediction to BTBs

- Add 2 bits of prediction (the purpose of the last field in the previous figure)
  - By definition, the branch is predicted taken!
    - It has an entry in the BTB
    - Even happens if the predictor indicates that it should NOT be taken
  - In this case, it is better to have separate buffers for prediction and predicted PCs (which can be different sizes)
  - A "not taken" in the prediction buffer will override an entry in the BTB
- More complex prediction mechanisms such as (*m*,*n*) predictors can also be used in this way

Interaction of prediction & BTBs



Branch folding

- Instead of storing just the branch address, the BTB can store the actual instruction as well
  - Return the new instruction from the cache rather than just the new address
  - The branch "disappears" since it is replaced with the instruction given by its target address
  - ⇒ The branch instruction does NOT require any execution cycles!
- If it's a conditional branch, we will still have to make sure the condition is satisfied
- Branch folding works well for
  - Unconditional branches
  - Conditional branches where the condition is easy to test (including condition codes)

Limits to branch prediction

- Misprediction rate: limits branch prediction benefits
  - If it's too high, there's too little benefit to justify the added hardware
- Misprediction penalties: also important!
  - If these are no worse than the standard penalties for missed static prediction, dynamic prediction is a win
  - What if *dynamic* misprediction penalties are worse than *static* misprediction penalties?
    - ⇒ *Static* prediction might actually outperform *dynamic* prediction even though it has a worse misprediction rate

Multiple issue: superscalar & VLIW

- Prior techniques reduce ideal CPI to as close to 1 as possible
- To reduce CPI below 1, the CPU must be capable of issuing more than one instruction per cycle
  - Superscalar: CPU tries to issue more than one instruction per cycle to keep all of the functional units busy
    - May be limits (i.e., no more than one memory instruction per cycle, no more than one branch per cycle)
    - Use **static** & **dynamic** scheduling to issue as many as possible
  - VLIW: fixed number of instructions per clock cycle
    - Similar to cramming (for example) four "simple" instructions into a single 128-bit instruction (one per functional unit)
    - **Statically** scheduled by the compiler

Superscalar DLX hardware

- Ensure that there are no data and structural hazards between instructions issued together
  - The easiest way to accomplish this is to allow dual issue of one integer instruction (ALU, load/store) and one floating point instruction
- Hardware requirements
  - Instruction alignment
    - Require that instruction pairs be 64-bit aligned, and that the integer instruction be first
    - Relaxing this requirement would increase the complexity of detecting hazards and thus the cost of the hardware
  - Arithmetic units & pipelines
    - CPU must have sufficient FP hardware to support one issue/clock
    - ⇒ Requires pipelined FP units or multiple FP units (or both)

Superscalar DLX hardware

- Interactions between integer and FP
  - FP and integer are largely independent
  - Integer instructions such as FP loads and stores as well as movement between integer and FP registers can cause problems
    - Creates contention for the FP register ports
    - Creates RAW hazards between integer FP loads/stores and FP ALU instructions
  - Handle FP register contention by adding an extra port to the FP register file for memory operations
  - ⇒ Detect the case in which an FP ALU instruction is issued in the same cycle as the load that fetches a source operand for it (RAW hazard)

Superscalar DLX data & control hazards

- Simple DLX pipeline: loads had a latency of one clock cycle
- Superscalar pipeline: the result of a load cannot be used on the same clock or the next clock cycle
- Hazards impose a penalty measured in cycles, not instructions
  - The next **3** instructions cannot use the result without a stall!
  - The same is true for branch delays
- More ambitious compiler or hardware scheduling techniques and more complex instruction decoding for branches are needed
- If the CPU is not able to get a useful instruction in both of the two slots, the CPI increases and approaches 1

Static scheduling on a superscalar DLX

```
Loop: LD   F0,0(R1)
      ADDD F4,F0,F2
      SD   0(R1),F4
      SUBI R1,R1,#8
      BNEZ R1,Loop
```

1 cycle latency

	Integer instruction	FP instruction	Clock cycle
Loop:	LD F0,0(R1)	-	1
	LD F6,-8(R1)	-	2
	LD F10,-16(R1)	ADDD F4,F0,F2	3
	LD F14,-24(R1)	ADDD F8,F6,F2	4
	LD F18,-32(R1)	ADDD F12,F10,F2	5
	SD 0(R1),F4	ADDD F16,F14,F2	6
	SD -8(R1),F8	ADDD F20,F18,F2	7
	SD -16(R1),F12	-	8
	SUBI R1,R1,#40	-	9
	SD 16(R1),F16	-	10
	BNEZ R1,Loop	-	11
	SD 8(R1),F20	-	12

2 cycle latency

- Unrolled loop 5 times; average 2.4 cycles per iteration

Dynamic scheduling on a superscalar DLX

- Dynamic scheduling can improve on these results to an even greater extent
  - The CPU can dual issue instructions with dependencies and serialize them later using hazard detection logic
  - Additional hardware can reduce delays through the elimination of WAR and WAW hazards and memory disambiguation
    - Similar to Tomasulo's approach
- Dynamic scheduling
  - Allows the CPU to keep the functional units busy as often as possible
  - Permits the CPU to run well on code that was not scheduled for superscalar execution

Dynamic superscalar CPUs today

- Modern CPUs may have
  - 2+ integer ALUs
  - Load/store (memory) unit
  - Branch unit
- CPU attempts to keep each functional unit busy
  - Extensive dynamic scheduling to work around many RAW hazards
    - Integer instructions can now have RAW hazards!
    - Lots of dynamic reordering to keep the units busy
  - FP/integer conflicts often less of an issue: not much FP computation
- Branch delays are a huge problem
  - 2 cycle delay is up to 11 lost instructions for 4-way issue (3 in the same cycle, 4 each in following cycles)

VLIW processors

- Superscalar machines use hardware to reorder instructions and keep functional units busy
- In VLIW (Very Long Instruction Word) machines, all of this burden falls upon the compiler
  - Each VLIW "instruction" is composed of multiple independent instructions, each of which execute on different function units
    - Functional units might include integer ALUs, FP ALUs, memory units, and a branch unit
    - The instruction must allocate 16 or more bits to each unit to describe the operation that the unit will run on each cycle
  - To keep the functional units busy, parallelism is uncovered by the compiler by unrolling loops and scheduling code across basic blocks
- A VLIW CPU can also help by providing forwarding

Sample VLIW processor

- VLIW machine that can issue **two** memory references, **two** FP operations, and **one** integer/branch operation per clock cycle
- Loop unrolled 7 times
  - Ignoring branch delay, loop achieves 2.5 operations per clock
  - Total time is 9 cycles for 7 iterations

Memory 1	Memory 2	FP 1	FP 2	Integer/branch
LD F0,0(R1)	LD F6,-8(R1)			
LD F10,-16(R1)	LD F14,-24(R1)			
LD F18,-32(R1)	LD F22,-40(R1)	ADDD F4,F0,F2	ADDD F8,F6,F2	
LD F26,-48(R1)		ADDD F12,F10,F2	ADDD F16,F14,F2	
		ADDD F20,F18,F2	ADDD F24,F22,F2	
SD 0(R1),F4	SD -8(R1),F8	ADDD F28,F26,F2		
SD -16(R1),F12	SD -24(R1),F16			
SD -32(R1),F20	SD -40(R1),F24			SUBI R1,R1,#56
SD 8(R1),F28				BNEZ R1,Loop

Limits in multiple-issue processors

- Why stop at 5 instructions/clock? Why not 50?
- Limits on available ILP in programs
  - There are usually not enough operations to fill all of the available slots
  - It might seem that 5 independent instructions are sufficient in the example; however, the memory, branch and FP units will likely be pipelined and have a multicycle latency
    - Assume a latency of 6 clocks for the FP units, and that two FP pipelined units are available
    - This requires that there are 12 FP instructions that are independent of the most recently issued FP instruction!
  - If a branch requires just a one cycle latency, it results in a 5 instruction latency in the example CPU machine

Limits in multiple-issue processors

- Hardware complexity
  - Additional functional units: duplicate integer and FP units for multiple-issue
    - Their cost scales linearly
  - Added bandwidth to registers
    - More register file ports are required to sustain the multiple issue
    - A single integer pipeline requires 3 ports to a register file
    - Adding another pipeline requires 3 more ports
  - Added memory ports: necessary for multiple memory units
    - Much more expensive than register ports
  - Scheduling hardware
    - Relatively simple for VLIW
    - Can be very complex for superscalar architectures

Limits in multiple-issue processors

- Superscalar CPUs have complex instruction issue logic
- VLIW CPUs have other problems
  - Technical problems
    - Increase in code size from open slots (wasted bits for unused functional units) increases memory bandwidth requirements
    - A stall (i.e., cache miss) in any functional unit causes the entire processor to stall because of the lock step operation of VLIW
  - Logistical problems
    - Binary compatibility is a problem because adding functional units or changing latencies requires major code changes
- Complexity and access time penalties of a multiported memory hierarchy are probably the most serious hardware limitations of superscalar and VLIW implementations

Even more parallelism

- Previously discussed methods that the compiler can use to discover ILP
  - Works as long as branch behavior is relatively predictable
- Better: increase levels of ILP in programs
  - Conditional execution: instructions that are “executed” only when a certain condition holds
  - Speculative execution
    - Execute instructions that might be needed later
    - Example: execute both forks of a branch

Conditional instructions

- A conditional instruction refers to a condition which is evaluated as part of the instruction execution
  - Don’t use a branch to skip a single instruction
  - Instr always executes but only writes the result if the condition is met
- Eliminating the branch gives two benefits
  - The branch is not executed, reducing the instruction count by 1
  - The branch delay is avoided
- Conditional execution changes a control dependence into a data dependence
  - In an integer pipeline, **data** dependencies rarely cause stalls while **control** hazards do cause stalls

```
if (A==0)                BNEZ  R1,L
  S = T;          ⟶      MOV   R2,R3     ⟶   CMOVZ R2,R3,R1
                       L:
```


Benefits of conditional instructions

- Conditional instructions help a lot with superscalar machines because such machines suffer even more from branch stalls
  - Conditional instructions can be scheduled as normal instructions
  - Branches often cannot be scheduled this way because they may cause a change in the instruction stream

  ⇒ More slots in a superscalar machine can be filled
- Conditional instructions are of even greater benefit on a VLIW machine for similar reasons

Conditional instructions & exceptions

- Conditional instructions must not introduce an exception if its condition isn't satisfied
  - The instruction must have NO effect if the condition is not satisfied
  - In example below, if R10 contains *zero*, it's likely that the LW instruction will cause a protection violation if allowed to execute
- Solution:
  - In DLX, memory accesses are not started until MEM
  - It's easy to evaluate the condition (i.e. during EX) and prevent the memory access from happening in this case

```
  BEQZ R10,L
  LW   R8,20(R10)  ──────►  LWC R8,20(R10),R10
L:
```


Limits to conditional instructions

- Executing conditional instructions takes time
  - A conditional instruction always requires time, even if the instruction is annulled
  - Moving an instruction across a branch is essentially speculating on the outcome of the branch
  - May slow down a program if an instruction is executed but turned into a no-op, since another instruction may have executed during that slot
  - Conditional instructions are always a win when the cycle that they occupy would have been idle anyway
- Sequence length can affect performance
  - Trading a branch and move for a conditional move is usually a win
  - Longer sequences may not be

Limits to conditional instructions

- The condition must be evaluated early
  - The condition must be known before the processor's state is changed, and the earlier the better
- Conditional instructions are difficult for multiple conditions
  - These instructions work well for avoiding single branches
  - The task is more difficult for two or more branch options: it requires additional instructions to logically combine the multiple conditions
- Conditional instructions may impose a speed penalty
  - The cycle time for the entire CPU might be increased
  - A conditional instruction might take more clock cycles to execute than a non-conditional instruction

Compiler-directed speculative execution

- Conditional instructions eliminate control dependencies for small if-then blocks
- Moving larger blocks of code across (before) branches can yield larger performance gains
- Doing so creates problems in two areas
  - Registers that should not be modified (because of the branch) are modified anyway
  - Exceptions that should not occur may actually happen (as with conditional instructions)
- Resumable exceptions such as page faults aren't a big problem
  - May cause performance to suffer somewhat
  - Programs don't terminate incorrectly

Implementing speculation: ignore exceptions

- **Three** schemes for supporting speculation without introducing erroneous exception behavior have been investigated
- First scheme: ignore exceptions
  - The simplest method for speculation is for the CPU and OS to ignore non-resumable exceptions for speculative instructions
  - Rather than terminate the program, they return an undefined value for the instruction causing the exception
    - If the exception generating instruction was not speculative, the program is in error but it is allowed to continue!
      - ⇒ However, it'll probably generate incorrect results
    - If the exception generating instruction was speculative, the speculative result won't be used and the program will run properly
  - Either way, a correct program is not terminated improperly

Ignore exceptions: example

```
If (A==0)
  A = B;
else
  A += 4;
```

```
      LW   R1,0(R3) ; load A
      BNEZ R1,L1    ; test A
      LW   R1,0(R2) ; if clause
      J    L2       ; skip else
L1:   ADDI R1,R1,#4 ; else clause
L2:   SW   0(R3),R1 ; store A
```

Two possibilities assuming **`then`** almost always executed

```
      LW   R1,0(R3)
      LW   R14,0(R2)
      BEQZ R1,L3
      ADDI R14,R1,#4
L3:   SW   0(R3),R1
```

Speculative load B

Non-speculative store

```
      LW   R1,0(R3)
      LW   R14,0(R2)
      ADDI R16,R1,#4
      MVC  R14,R16,R1
      SW   0(R3),R1
```

- Why use R14?
- Where is the value of A at the end of this sequence?

- Branch replaced by conditional move
- Under what circumstances is this more expensive?

Speculative execution: poison bits

- Each register has a "poison bit" attached to it
  - If a speculative instruction causes an exception, the exception is handled by setting the poison bit of its destination register
  - If another speculative instruction uses a poisoned register as a source operand, its destination register poison bit is also set
  - If a non-speculative instruction uses a poisoned register, an exception is generated
    - It may, however, write to a poisoned register
    - If this occurs, the poison bit is cleared
- This method generates exceptions for incorrect programs (at about the right place)

  ⇒ The OS must be able to save, restore, and reset the poison bits, which requires special instructions

Speculative execution: boosting

- Previous schemes introduced register copies
- This approach (called boosting) provides ***renaming*** and ***buffering*** in hardware, similar to Tomasulo's approach
  - A boosted instruction is executed speculatively based on a branch
  - Its results are forwarded to and used by other boosted instructions
  - When the branch is reached, the results are committed to the register file if the prediction is correct
- Therefore, instructions that are control dependent on a branch can be executed **before the branch**

Speculative execution with renaming

```
If (A==0)
  A = B;
else
  A += 4;
```

```
        LW   R1,0(R3) ; load A
        BNEZ R1,L1    ; test A
        LW   R1,0(R2) ; if clause
        J    L2       ; skip else
L1:     ADDI R1,R1,#4 ; else clause
L2:     SW   0(R3),R1 ; store A
```

+ indicates the instruction is boosted across the next branch and predicts the branch taken

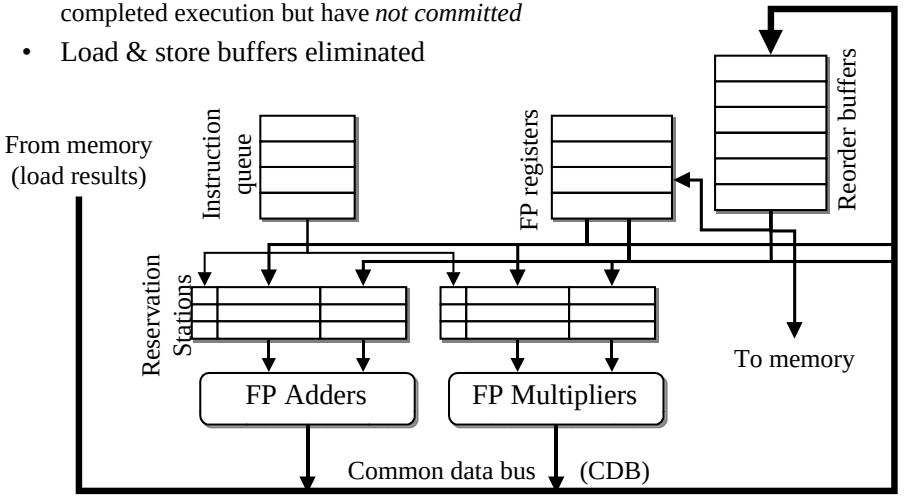
```
        LW   R1,0(R3)
        LW+  R1,0(R2) ; boosted load of B
        BEQZ R1, L3
        ADDI R1,R1,#4
L3:     SW   0(R3),R1 ; non-speculative store
```


Hardware-based speculation

- Combine speculative execution and dynamic scheduling based on Tomasulo's approach
  - Focus on floating-point operations
  - Similar structures can handle integer operations
- Change Tomasulo's approach to support speculation
  - Separate the process of completing execution and the bypassing of results among instructions from instruction commit (register file or memory update)
  - This allows other (speculative) instructions to execute, but no results are committed until we know the instruction is no longer speculative
- Allow instructions to execute out of order but force them to commit in order
  - Helps handle exceptions properly

Hardware-based speculation: design

- A set of hardware buffers (***reorder buffers***) hold the results of instructions that have completed execution but have *not committed*
- Load & store buffers eliminated



Hardware-based speculation: stages

- The reorder buffer provides additional virtual registers and is a source of operands for instructions
- An additional step is added to Tomasulo's algorithm
  - Issue
    - Get a floating-point instruction
    - Issue it if there is a reservation station open and an empty slot in the reorder buffer
    - Send the number of the reorder buffer assigned for the result to the reservation station so it can be used to tag the result
  - Execute
  - Monitor the CDB while waiting for source registers to be ready
  - When both operands are available, perform the operation

Hardware-based speculation: stages

- More steps in to Tomasulo's algorithm
- Write result
  - Write the result on the CDB with the reorder buffer tag
  - Result is stored into the reorder buffer as well as into any reservation stations waiting for the result
  - Reorder buffer can also serve as a source register for operands similar to the register file
- Commit
  - When the instruction reaches the head of the reorder buffer and its result is present in the buffer, update the register or write memory
  - When an incorrectly predicted branch arrives, flush the reorder buffer and restart execution at the correct successor of the branch
  - If the branch was correctly predicted, do nothing

Hardware-based speculation: advantages

- This scheme has several advantages over dynamic scheduling alone
- Instructions can "finish" out of order as long as they are not committed
  - ⇒ The CPU can keep *precise interrupts* even while executing out of order since changes are committed in order.
- The CPU to *speculatively* execute instructions past a branch before the branch is executed
  - ⇒ Instructions are canceled if the branch is mispredicted
- Handle exceptions just before the instruction is ready to commit
  - All previous instructions and no later instructions have committed
  - The CPU can do a precise exception even with out-of-order execution

Speculation & multiple-issue CPUs

- The techniques that work in single-issue CPUs work in multiple-issue CPUs as well
  - Speculate on both integer and floating point instructions
  - More complex design
    - More hazards to check for
    - CDB (maybe more than one!) gets crowded...
- Speculation may be more useful in such processors
  - Longer branch delays and operation latencies
  - More empty execution slots that speculation can fill (potentially usefully)

Lower CPI isn't always faster

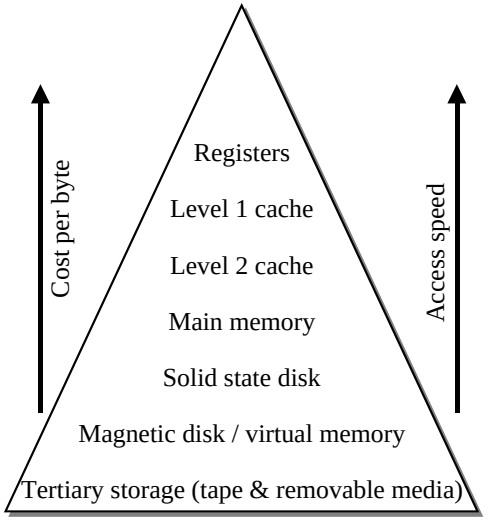
- If the lower CPI comes at the expense of a longer clock cycle, it may slow the processor down
  - Almost invariably true since lowering CPI using hardware means implementing more sophisticated techniques which increase clock cycle time
- This inclination arises because
  - Simulation tools to evaluate the impact of enhancements that affect CPI are more readily available than tools to evaluate the impact on clock cycle time
  - Accurate analysis on the impact of clock rate is not usually possible until the design is well underway

Improve the whole CPU, not just part

- As with uniprocessors, improving one aspect of a CPU does not help unless it was the bottleneck from the beginning
  - Improving FP latency for a multiple-issue CPU does not help much unless something is done about branching
  - Making branches faster doesn't help if the CPU stalls a lot waiting for integer hazards
- Speculative execution is great but is of limited benefit unless there are additional registers to use
  - Under compiler control (larger register set)
  - "Virtual" registers used by dynamic scheduler

Memory hierarchy: the storage pyramid

- Principle of locality: programs don't access code and data uniformly
- Faster hardware has less capacity and costs more per byte
- Result: memory hierarchy
  - Keep frequently used code & data in fast memory
  - Keep everything else in slower memory
- Two views of hierarchy
  - "Infinite supply of memory, some parts slower than others"
  - "Lots of objects"



Characterizing the memory hierarchy

- Questions about any 2 levels of the memory hierarchy:
  - Where can a block be placed in the upper level?
    - ⇒Block placement
  - How is a block found if it is in the upper level?
    - ⇒Block identification
  - Which block should be replaced on a miss?
    - ⇒Block replacement
  - What happens on a write?
    - ⇒Write strategy
- Focus on the interface between
  - CPU's memory cache and main memory
  - Dynamic RAM and disk (virtual memory)

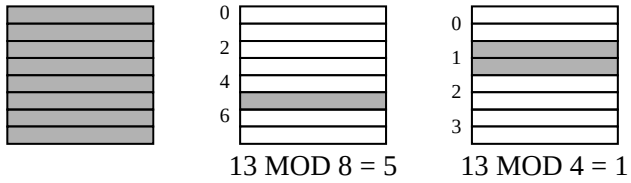
Memory system performance

- Evaluate the effectiveness of the memory hierarchy with:
  Memory stall cycles = IC * memory refs per instruction * miss rate * miss penalty
- Use a related formula to evaluate the performance of various memory system configurations
- Several factors in this equation:
  - IC * memory refs per instruction
    - Frequency with which the CPU uses memory
    - A memory system that needs to satisfy just 1-2 refs per cycle is easier to build than one that satisfies 4-5 refs per cycle
  - Miss rate: fraction of references not satisfied in the upper level
  - Miss penalty: length of time it takes to get a value from the lower level
  - Low miss rate doesn't help if miss penalty is too high (and vice versa)

What is a cache?

- What does "cache" refer to?
  - No modifiers => usually means the fast memory closest to the CPU
  - "Cache" has been used for everything from files to WWW pages
- Block placement: three options
  - Fully associative (block can go anywhere)
  - Direct mapped (block can go in one place)
  - Set associative (block mapped to set, but can go anywhere in set)
  - Direct mapped = 1-way set associative

Where does block 13 go in cache?



Block identification

- Block offset: the first few bits of the address give the offset of the byte within a block
- Block address (index): used to pick a set from the cache
- Tag
  - Only the tag is stored in the cache (the rest of the address is implied)
  - All tags within a set are searched in parallel
- Valid bit: indicates that the block in this location contains valid data
  - Otherwise, a random sequence of bits could be mistaken for a valid entry that matched the tag

Block replacement

- Which block is replaced?
  - For direct mapped, each block can only go in one location!
  - Question relevant for fully associative and set associative caches
- Block to replace chosen by
  - Random: choose a block from the set at random
  - LRU: least-recently used
    - Replace the block that has been unused for the longest time
    - This requires extra bits in the cache to keep track of accesses
  - ⇒ LRU isn't much better than random replacement for small sets
- ⇒ Use LRU only for 2-way set associative
  - Other caches use random
  - Even 2-way set associative may use random to save bits

Write strategy

- What happens on a write?
- Memory access distribution
  - All instruction access are reads
  - Most data accesses are reads (DLX, 9% stores and 26% loads)
- Make the common case fast => optimize caches for reads
  - The common case is also the easy case to handle since tag checking and reading can occur in parallel
  - Extra bytes read can be safely ignored
- Amdahl's law reminds us that we can't *ignore* writes!
  - Problem: Tag checking and writing ***can't*** occur in parallel
  - ⇒ Writing is usually slower than reading
  - Extra bytes can't be safely written

Write policy

- Determines when the write is communicated to the lower level
- Write-through: block is written to both the cache and main memory at the same time
  - Read misses don't result in writes
  - Memory hierarchy is consistent
  - Simple to implement
- Write back (also known as copy back): block modified in cache only at time of write
  - Main memory modified when the block must be replaced in the cache
  - Requires the use of a dirty bit to keep track of block modification status
  - Writes occur at speed of cache
  - Multiple writes occur to the same block can be "collapsed"

Write misses

- Two options when a write is made to a block not in cache
  - Write allocate: the block is loaded into the cache on a miss before anything else occurs
  - Write around (no write allocate): the block is only written to main memory; it isn't stored in the cache
- Generally,
  - Write-back caches use write-allocate
    - ⇒ Hopefully, subsequent writes to that block will be captured by the cache
  - Write-through caches use write-around
    - ⇒ Subsequent writes to that block will still go to memory even if the block is fetched into cache

Write buffers & write merging

- Many CPUs use a write buffer to avoid stalling on writes
  - Write buffer => a small cache that can hold a few values waiting to go to main memory
  - This buffer helps when writes are clustered
  - It doesn't entirely eliminate stalls since it is possible for the buffer to fill if the burst is larger than the buffer
- Write merging
  - Blocks are often larger than a machine word
  - Write buffers can merge memory writes to save write buffer slots and memory traffic
  - ⇒ Writes to the same location can be collapsed
  - ⇒ Writes to sequential locations can be merged into a single buffer slot

Split vs. unified caches

- Should there be a single or two caches in the system?
- Unified cache: all memory requests go through a single cache
  - \+ Requires less hardware
  - – Has lower bandwidth
  - – More opportunity for collisions
- Split I & D caches: instructions & data are stored in separate caches
  - – Uses additional hardware
    - Some simplifications (I-cache is read-only)
  - \+ Higher bandwidth (2 is greater than 1)
  - \+ No collisions between data & instructions

Cache performance

- Average memory access time (AMAT) is a useful measure to evaluate the performance of a memory-hierarchy configuration
  AMAT = hit time + miss rate * miss penalty
- AMAT shows how much penalty the memory system imposes on each access (on average)
  - ⇒ It can easily be converted into clock cycles for a particular CPU
- Leaving the penalty in nanoseconds allows two systems with different clock cycles times to be compared using a given memory system

Performance for split I & D caches

- Instruction and data accesses may have different penalties
  - They may have to be computed separately
  - This requires knowledge of the fraction of references that are instructions and the fraction that are data
  - For example, the text says that (usually) 75% of memory references are instructions, and 25% are data references
- The write penalty can also be computed separately from the read penalty
  - Miss rates may be different for each situation
  - Miss penalties may be different for each situation (i.e., writeback vs. write through)

$$CPU\ \ time = IC \times \left(CPI_{execution} + \frac{Memory\ \ accesses}{instruction} \times Miss\ \ rate \times Miss\ \ penalty \right) \times Clock\ \ cycle\ \ time$$

Cache performance example

- Problem: compare the performance of a 64KB unified cache with a split cache with 32KB data and 16KB instruction
  - Miss penalty for either cache is 100 ns, and the CPU clock runs at 500 MHz
  - Don't forget that the unified cache requires an extra cycle for load and store hits because of the structural conflict
  - Calculate the effect on CPI rather than the average memory access time
- Assume miss rates are as follows (Fig. 5.7 in text):
  - 64K Unified cache: 1.35%
  - 16K instruction cache: 0.64%
  - 32K data cache: 4.82%
- Assume a data access occurs in 1/3 of all instructions

Cache performance example

- Compute the CPI penalty separately for instructions and data
- First, figure out the miss penalty in terms of clock cycles: 100 ns/2 ns = 50 cycles
- Unified cache
  - Instruction access penalty is (0 + 1.35% * 50) = 0.675 cycles
  - Data access penalty is (1 + 1.35% * 50) = 1.675 cycles
  - Overall penalty is 0.675 + (1/3) * 1.675 = 1.23 cycles per instruction
- Split cache
  - Instruction access penalty is (0 + 0.64% * 50) = 0.32 cycles
  - Data access penalty is (0 + 4.82% * 50) = 2.41 cycles
  - Overall penalty is 0.32 + (1/3) * 2.41 = 1.12
- Split cache performs better => no stall on data accesses

Effects of cache on CPU performance

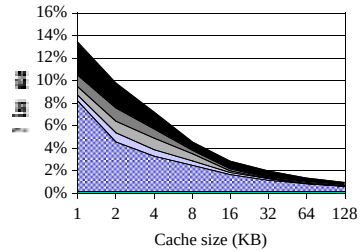
- Low CPI machines suffer more relative to some fixed CPI memory penalty
  - A machine with a CPI of 5 suffers little from a 1 CPI penalty.
  - A processor with a CPI of 0.5 has its execution time tripled!
- Cache miss penalties are measured in cycles, not nanoseconds
  - ⇒ A faster machine will stall more cycles on the same memory system
- Amdahl's Law raises its ugly head again
  - Fast machines with low CPI are affected significantly from memory access penalties
  - Fast machines spend most of their time accessing memory!

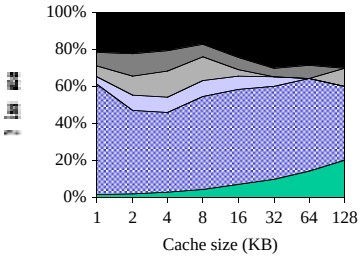
Improving cache performance

- The increasing speed gap between CPU and main memory has made the performance of the memory system increasingly important
- There are many distinct methods system architect use to reduce average memory access time
- These methods can be classified by whether they
  - Reduce the miss rate
  - Reduce the miss penalty
  - Reduce the time to hit in a cache
- Other methods may also increase capacity for a given cost...

Components of cache miss rate

- Three "C"s of cache misses
- Compulsory misses
  - First access to a block ***can't*** be in the cache
  - Occur regardless of cache size
- Capacity misses
  - Occur because cache isn't large enough to hold all blocks
  - Compulsory miss rate - the miss rate of a fully associative cache
- Conflict (collision) misses
  - The block can't be kept because the set is full
  - Difference between fully- and set- associative cache





Reducing cache miss rate

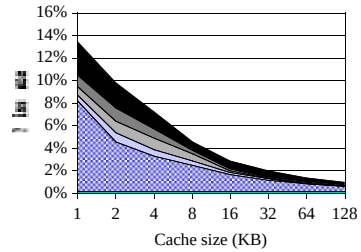
- To reduce cache miss rate, we must eliminate some of the misses due to the three C's
  - ***Capacity*** misses can't be reduced much except by making the cache larger
  - ***Conflict*** misses and ***compulsory*** misses can be reduced in several ways
- Larger cache blocks
  - Decrease the compulsory miss rate by taking advantage of spatial locality
  - May increase the miss penalty by requiring more data to be fetched per miss
  - Likely to increase conflict misses since fewer blocks can be stored in the cache
  - May even increase capacity misses in small caches

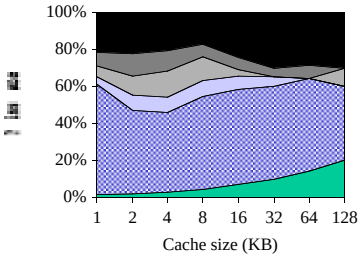
Improving cache performance

- The increasing speed gap between CPU and main memory has made the performance of the memory system increasingly important
- There are many distinct methods system architect use to reduce average memory access time
- These methods can be classified by whether they
  - Reduce the miss rate
  - Reduce the miss penalty
  - Reduce the time to hit in a cache
- Other methods may also increase capacity for a given cost...

Components of cache miss rate

- Three "C"s of cache misses
- Compulsory misses
  - First access to a block ***can't*** be in the cache
  - Occur regardless of cache size
- Capacity misses
  - Occur because cache isn't large enough to hold all blocks
  - Compulsory miss rate - the miss rate of a fully associative cache
- Conflict (collision) misses
  - The block can't be kept because the set is full
  - Difference between fully- and set- associative cache



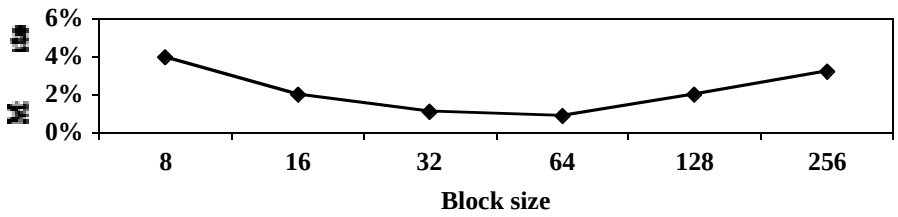


Reducing cache miss rate

- To reduce cache miss rate, we must eliminate some of the misses due to the three C's
  - ***Capacity*** misses can't be reduced much except by making the cache larger
  - ***Conflict*** misses and ***compulsory*** misses can be reduced in several ways
- Larger cache blocks
  - Decrease the compulsory miss rate by taking advantage of spatial locality
  - May increase the miss penalty by requiring more data to be fetched per miss
  - Likely to increase conflict misses since fewer blocks can be stored in the cache
  - May even increase capacity misses in small caches

Larger cache blocks

- The miss rate curve is U-shaped because
  - Small blocks have a higher miss rate
  - Large blocks have a higher miss penalty (even if miss rate is not too high)
- High latency, high bandwidth memory systems encourage large block sizes
  - The cache gets more bytes per miss for a small increase in miss penalty
  - 32-byte blocks are typical for 1-KB, 4-KB and 16-KB caches
  - 64-byte blocks are typical for larger caches
- Instruction caches tend to have larger blocks than data caches

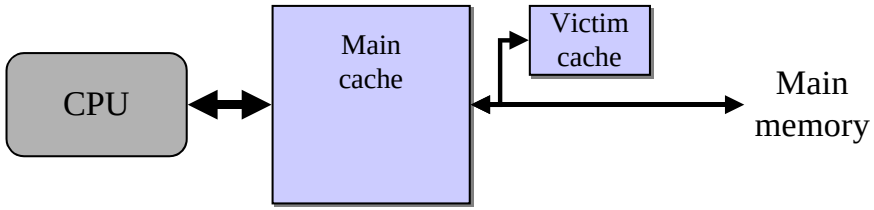


Higher associativity

- *Conflict* misses can be a problem for caches with low associativity (especially direct-mapped)
- Miss rates generally follow the 2:1 cache rule of thumb
  - A direct-mapped cache of size N has the same miss rate as a 2-way set-associative cache of size N/2
- However, higher associativity means
  - More hardware
  - Often, longer cycle times (increased hit time)
  - Possibly, more capacity misses
- 8-way set-associative caches are the maximum used today, and most systems use 4-way or less
  - ⇒ Higher hit rate is offset by the slower clock cycle time

Victim caches

- Victim cache => small cache (often fully associative) that holds a few of the most recently replaced blocks or victims from the main cache
  - Reduces conflict misses and (secondarily) capacity misses
  - Particularly effective for small, direct-mapped data caches
    - A 4 entry victim cache handled from 20% to 95% of the conflict misses from a 4KB direct-mapped data cache
- Check victim cache before main memory on a miss
  - Swap victim block & cache block if hit in victim cache



Pseudo-associative caches

- These caches use a technique similar to double hashing
  - On a miss, the cache searches a different set for the desired block
  - The second (pseudo) set to probe is usually found by inverting one or more bits in the original set index
- Two separate searches are conducted on a miss
  - The first search proceeds as it would for a direct-mapped cache: since there's no associative h/w, hit time is fast if block found on first probe
  - The second probe takes some time (usually an extra cycle or two), but it's a lot faster than going to main memory
    - The secondary block can be swapped with the primary block on a "slow hit"
- This method reduces the effect of conflict misses
- Also improves miss rates without affecting the clock rate

Hardware prefetch

- Prefetching is the act of getting data from memory before it is actually needed by the CPU
  - Typically, the cache requests the next consecutive block to be fetched with a requested block, hopefully avoiding a subsequent miss
  - Compulsory misses reduced by retrieving the data before it is requested
  - Other misses may *increase* => useful blocks replaced in the cache
- Many caches hold prefetched blocks in a special buffer until they are actually needed
  - This buffer is faster than main memory but only has a limited capacity
- Prefetching also uses main memory bandwidth
  - Prefetching works well if the data is actually used
  - However, it can adversely affect performance if the data is rarely used and the accesses interfere with 'demand misses'

Compiler-controlled prefetch

- Some CPUs include prefetching instructions
  - Instructions request that data be moved into either a register or cache
  - These special instructions can either be faulting or non-faulting
    - Non-faulting instructions do nothing (no-op) if the memory access would cause an exception
- Prefetching shouldn't interfere with normal CPU operations
  - The cache must be *nonblocking* (also called *lockup-free*)
  - This allows the CPU to overlap execution with the prefetching of data
- Improves prefetch "hit" rates over hardware prefetch
  - However ,it does so at the expense of executing more instructions
  - Thus, the compiler tends to concentrate on prefetching data that are likely to be cache misses anyway
  - Loops are key targets since they operate over large data spaces and their data accesses can be inferred from the loop index in advance

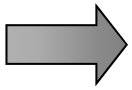
Compiler optimizations

- This method requires ***no*** hardware modifications
  - However, it's often the most efficient way to reduce cache misses
  - The improvement results from better code and data organizations
- Examples
  - Code can be rearranged to avoid conflicts in a direct-mapped cache
  - Accesses to arrays can be reordered to operate on blocks of data rather than processing rows of the array
  - Arrays can be resized to avoid cache conflicts for related elements

Compiler optimization: merging arrays

- Combine two separate arrays (that might conflict for a single block in the cache) into a single interleaved array
- Bring together corresponding elements in both arrays, which are likely to be referenced together
- Reorganizing and fetching them at the same time can reduce misses
- This technique reduces misses by improving spatial locality

```
int value[SIZE];
int key[SIZE];
```

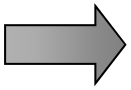


```
struct merged {
  int value;
  int key;
};
struct merge m[SIZE];
```


Compiler optimization: loop interchange

- Switch the order in which loops execute
  - Misses can be reduced due to improvements in spatial locality
- Example
  - These loops cause a miss on each memory access because of the long stride given by index *j* in the inner loop
  - Switching the order of the loops changes the stride to 1 => the elements are accessed in sequential order.

```
for (i=0; i<100; i++) {
  for (j=0; j<100; j++) {
    a[j][i] = a[j][i]*2;
  }
}
```

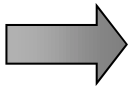


```
for (j=0; j<100; j++) {
  for (i=0; i<100; i++) {
    a[j][i] = a[j][i]*2;
  }
}
```

Compiler optimization: loop fusion

- Many programs have separate loops that operate on the same data
- Combining these loops allows a program to take advantage of **temporal locality** by grouping operations on the same (cached) data together
  - Caching may work even better because of sequential access between elements
  - Caching can hold results from previous iterations of the loop...

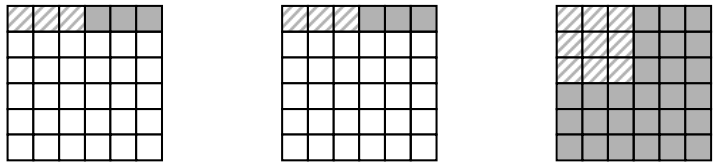
```
for (j=0; j<100; j++) {
  x[j] = x[j] + y[j];
}

for (j=0; j<100; j++) {
  y[j] = y[j] + x[j-1];
}
```



```
for (j=0; j<100; j++) {
  x[j] = x[j] + y[j];
  y[j] = y[j] + x[j-1];
}
```

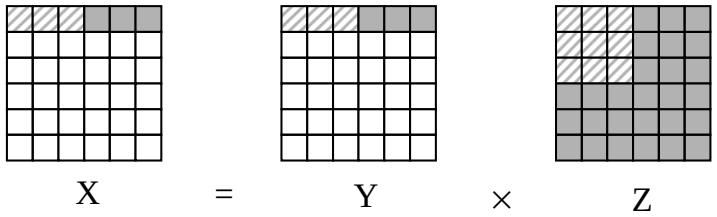

Compiler optimization: blocking

- Previous compiler optimizations work well on array accesses that occur along one dimension only
  - Loops that access both rows and columns can use other techniques
  - Unoptimized matrix multiplication => cache must hold the shaded areas
- Another technique: blocking
  - Capacity misses can occur for large matrices since it may not be possible to store all the elements of Z in the cache
  - Blocking operates on submatrices: reduces total memory words accessed by a factor of B (the blocking factor)



Compiler optimization: blocking

- Matrix multiplication is performed by multiplying the submatrices first
  - Matrix Y benefits from spatial locality
  - Matrix Z benefits from temporal locality
- This method is also used to reduce the number of blocks that must be transferred between disk and main memory
  - ⇒ The technique is effective for several levels of the hierarchy
- Given the increasing speed gap in processor speed and memory access times, these last two techniques will only increase in importance over time



Giving read misses priority

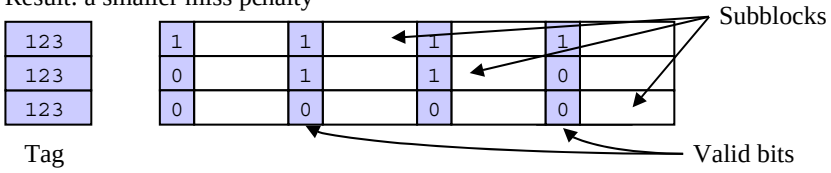
- If a system has a write buffer, delay writes to come after reads
- Problem: reads may request a value about to be written
- Solution 1: stall reads until the write buffer is empty
  - The write buffer in write-through is likely to have blocks queued up
  - Read miss penalty increases considerably
- Solution 2: check the write buffer for conflicts
  - In cases like this, the write buffer acts as a victim cache

```
SW 0(R3),R4
LW R11,4096(R3)
LW R12,0(R3)
```

If this is a direct-mapped 4KB cache, will R12 get the value from R4?

Using subblocks to reduce fetch time

- Tags can hurt performance by occupying too much space or by slowing down caches
  - Using large blocks reduces the amount of storage for tags (and makes them shorter), optimizing space on the chip
  - This may even reduce miss rate by reducing compulsory misses
  - However, the miss penalty for large blocks is high, since the entire block must be moved between the cache and memory
- Solution: divide each block into subblocks, each of which has a valid bit
  - Tag is valid for the entire block, but only a subblock needs to be read on a miss
  - A block is no longer the minimum unit transferred between cache and memory
  - Result: a smaller miss penalty



Early restart & critical word first

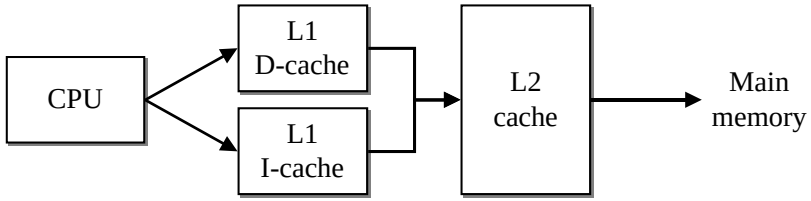
- Goal: optimize the order in which the words of a block are fetched and when the desired word is delivered to the CPU
- This strategy requires no extra hardware!
- Early restart
  - The CPU gets its data (and resumes execution) as soon as the desired word arrives in the cache
  - CPU doesn't wait for the rest of the block!
- Critical word first
  - Don't start the fetch of a block with the first word
  - Instead, fetch the requested word first and then fetch the rest afterwards
- Early restart & critical word first reduce the miss penalty
  - ⇒ CPU can continue execution while most of the block is still being fetched

Non-blocking cache

- A nonblocking cache can allow the CPU to continue executing instructions after a data cache miss
  - Works well in conjunction with out-of-order execution
  - The cache continues to supply hits while processing read misses (hit under miss)
  - The instruction needing the missed data waits for the data to arrive
- Complex caches can even have multiple outstanding misses (miss under miss)
  - This greatly increases cache complexity
  - May be of relatively little benefit relative to the design complexity

Second level caches

- This method focuses on the interface between the cache and main memory
- Add a second-level cache between main memory and a small, fast first-level cache
  - This helps satisfy the desire to make the cache fast and large
  - The second-level cache allows
    - A small first-level cache that fits on the chip with the CPU
    - A first-level cache fast enough to handle hits in 1-2 CPU cycles
  - Hits for many memory accesses that would go to main memory are handled in the L2 cache, lessening the effective miss penalty



Performance of multi-level caches

- Calculating performance of a two-level cache is done similarly to that of a one-level cache
  - Miss penalty for level 1 is calculated using the hit time, miss rate, and miss penalty for the level 2 cache
- For two level caches, there are two miss rates
  - Global miss rate: the number of misses in the cache divided by the total memory accesses generated by the CPU (Miss $rate_{L1}$*Miss $rate_{L2}$)
  - Local miss rate: the number of misses in the cache divided by the total memory accesses to *this* cache (Miss $rate_{L2}$ for the 2nd-level cache)
    - The local miss rate for L2 is high because it's only getting the misses from the L1 cache (instead of all memory accesses)
- Global miss rate is often a more useful measure => fraction of the memory accesses that must go all the way to memory

$$Avg\ \ memory\ \ access\ \ time = Hit\ \ time_{L1} \times Miss\ \ penalty_{L1}$$

$$Miss\ \ penalty_{L1} = Hit\ \ time_{L2} + Miss\ \ rate_{L2} \times Miss\ \ penalty_{L2}$$

Desirable characteristics for an L2 cache

- Larger than the L1 cache
  - A miss in L1 is unlikely to be a hit in L2 unless L2 is much larger
  - The local hit rate for L2 depends on the size ratio between L1 and L2!
- Higher associativity
  - The main reason for low associativity was fast, small caches
  - The L2 cache need be neither, and will benefit from the higher hit rate that more blocks per set provides
- Larger block size
  - This reduces compulsory misses that are fetched from main memory
  - Since the L2 cache is large, the effect of increasing conflict misses (as is true for a smaller cache) is minimal

Multilevel inclusion

- If all of the data in the L1 cache is also in the L2 cache, the L2 cache has the multilevel ***inclusion*** property
  - Most caches enforce this property since it is easier to deal with cache consistency
  - Consistency between I/O and caches (and between caches in a multiprocessor) can be determined by checking second-level cache

Multilevel cache design

- Design of L1 and L2 caches: although they can be designed separately, it is helpful to know if there's an L2 cache
  - Write-through in L1 is much more effective if there is an L2 writeback cache to buffer repeated writes
  - A direct-mapped L1 cache works well if the L2 cache satisfies most of the conflict misses
- Multilevel cache design summary
  - In general, cache design trades fast hits for few misses
  - For an L1 cache, fast hits are more important
  - For L2, there are many fewer hits, so fewer misses becomes more important
- Thus, larger caches with higher associativity and larger blocks are beneficial for L2 caches

Reducing hit time: small & simple caches

- Cache access time limits the clock cycle rate on many systems
  - ⇒ Cache design affects more than average memory access time - it affects everything
- Small & simple caches reduce hit time
  - Less hardware to implement a cache => shorter critical path through the hardware
  - Direct-mapped is faster than set-associative for both reads and writes: tag
    - There is only one block for each index
    - If tag check fails, the block is wrong and a (long) cache miss must be processed
  - Fitting the cache on the chip with the CPU is also very important for fast access times
- Fast clock cycle time encourages small direct-mapped caches

Avoid address translation during indexing

- The CPU uses virtual addresses that must be mapped to a physical address
  - The cache may either use virtual or physical addresses
- Cache indexed by virtual addresses => virtual cache
- Cache indexed by physical address => physical cache
- A virtual cache reduces hit time
  - Translation from a virtual address to a physical address is not necessary on hits
  - Address translation can be done in parallel with cache access => penalties for misses are reduced as well
- So why are they used so infrequently?

Issues with virtual caches

- Process switches require cache purging
  - Different processes share the same virtual addresses even though they map to different physical addresses
  - When a process is swapped out, the cache must be purged of all entries to make sure that the new process gets the correct data
- One solution: PID tags
  - Increase the width of the cache address tags to include a process ID (instead of purging the cache)
  - The current process PID is specified by a register
  - If the PID does not match, it is not a hit even if the address matches

Virtual caches & aliasing

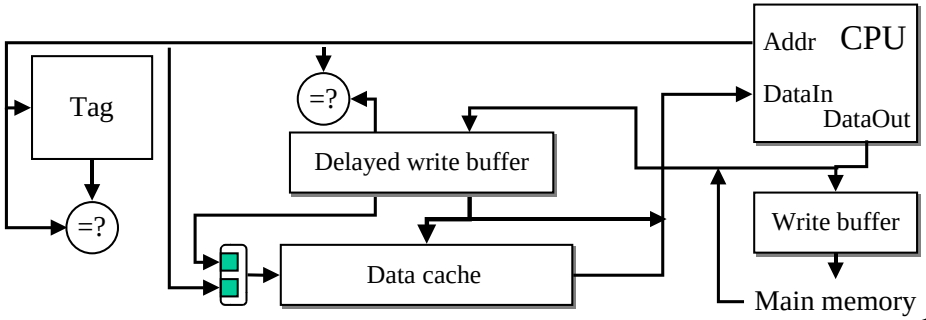
- Problem: two different virtual addresses may have the same physical address (even for a single process)
  - This may result in two copies of the same block in the cache!
  - The aliasing problem must be handled correctly
- Anti-aliasing hardware: guarantees every cache block a unique physical address
  - Every virtual address maps to the same location in the cache
  - This solution can be slow and difficult to implement in hardware
- Page coloring: software technique that forces aliases to share some address bits
  - The virtual address and physical address match over these k bits
  - A direct-mapped cache that is 2^k bytes or smaller can never have duplicate physical addresses for blocks

Reducing access time in virtual caches

- Use the page offset to index the cache: get the best of both virtual & physical caches
  - Overlap the virtual address translation process with the time required to read the tags
  - Page offset is unaffected by address translation
  - However, this restriction forces the cache size to be smaller than the page size because the index comes from the "physical" portion of the virtual address (the page offset)
- Basic operation
  - Send the page offset to the cache
  - At the same time, translate the virtual -> physical page number
  - Check the tag from cache against the physical address obtained by virtual -> physical translation
- High associativity allows for larger cache sizes

Reducing hit time with pipelined writes

- Write hits take longer than read hits because tag checking is required before the data is written
- One solution is to pipeline the writes (as in the Alpha AXP 21064)
  - The second stage of the write (cache is updated with new data) occurs during the first stage of the next write
  - Allows tag checking and data writing to occur simultaneously



Cache optimization techniques: summary

	Miss rate	Miss penalty	Hit time	Hardware complexity
Larger block sizes	+	—		0
Higher associativity	+		—	1
Victim caches	+			2
Pseudo-associativity	+			2
Hardware prefetching	+			2
Compiler-controlled prefetch	+			3
Compiler optimizations	+			0
Giving read misses priority		+		1
Subblock placement		+		1
Early restart / critical word 1st		+		2
Nonblocking cache		+		3
2nd level caches		+		2
Small & simple caches	—		+	0
Avoiding address translations			+	2
Pipelining writes			+	1

Main memory

- Main memory is usually made from DRAM while caches use SRAM
  - SRAM is faster (by almost an order of magnitude)
  - However, it's also more expensive per bit
    - DRAM uses 1 transistor & 1 capacitor per bit
    - SRAM uses 6 transistors => 4x to 8x the space
- There are methods for optimizing DRAM performance
- Performance measures for DRAM include:
  - Latency: important for caches (reduces miss penalty)
  - Bandwidth
    - Important for I/O
    - Also important for cache with second-level and larger block sizes

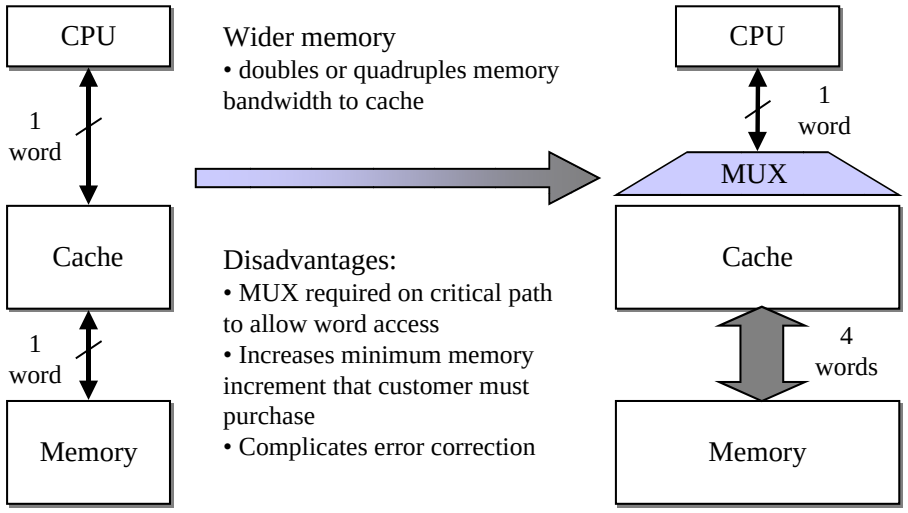
Main memory performance issues

- Latency measures include
  - Access time: time between when a read is requested and when the desired word arrives
  - Cycle time: the minimum time between the starts of two accesses to memory
    - This is at least as long as access time, and is usually longer
- DRAM refresh
  - DRAMs must occasionally refresh their data
  - This is done by reading all of the cells in a row and writing them back
  - Refresh must be done every few milliseconds
    - This operation consumes less than 5% of total time
    - The low time requirement occurs because the time necessary to refresh is proportional to the **square root** of the size of the DRAM

Main memory performance

- Amdahl suggested that memory capacity should grow linearly with CPU speed
  - Memory capacity grows **four-fold** every **three** years to supply this demand
  - The CPU-DRAM performance gap is a problem, however, since DRAM performance improvement is only about 7% per year
  - Cache innovations have addressed this problem to some degree
- There are innovations in main memory organizations that are more cost-effective

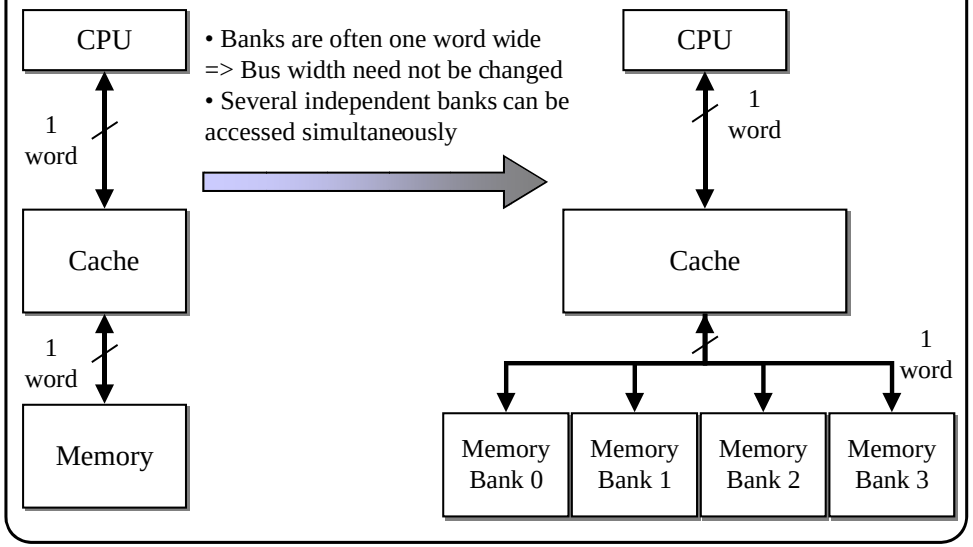
Wider main memory



Wider main memory

- DRAM chips are typically 1-8 bits wide
  - Any number of them can be accessed in parallel without extra delay
- By increasing the width of memory, the CPU can get more bits in a single cycle
  - This increases bandwidth between cache and memory
- Example: consider a cache with 4 word blocks
  - Main memory might require
    - 4 cycles to send the address
    - 40 cycles to access memory
    - 4 cycles to transfer over the bus
  - If the memory is only **one word** wide, a miss would require 4 x (4 + 40 + 4) = 192 cycles!
  - If the memory is enlarged to **4 words wide**, miss time is only 48 cycles

Interleaved main memory



Interleaved main memory

- Example: fetch a block by
  - Sending 1 address
  - Waiting for a single memory cycle
  - Transferring 4 words for a total time of 4 + 40 + (4 x 4) = 60 cycles
- A little slower than wider memory (due to bus limitations)
  - Reads must transfer more words
  - Writes can be overlapped if they are addressed to different banks
- Read access optimization may be possible
  - Example: cache block size is four words => parallel access is possible
- Write-back caches make writes sequential as well as reads
- How many banks are sufficient?
  - Possible rule: '# of banks >= # of clocks to access a word in a bank'
  - This allows up to 1 word per clock cycle in best case

Independent memory banks

- Interleaved memory concept can be extended to remove all restrictions on memory access
  - Interleaved memory => only a single memory controller in the system
    ⇒ Allows the interleaving of sequential access patterns
  - Address line sharing among the banks is possible in this scheme
- Instead, use multiple independent controllers
  - Example: one for I/O devices, one for cache reads and one for cache writes
  - Banks are still accessed in parallel, but now there may be multiple independent requests serviced simultaneously
- This can be particularly useful for
  - Nonblocking caches (that allow multiple outstanding read misses)
  - Multiprocessors

Avoiding memory bank conflicts

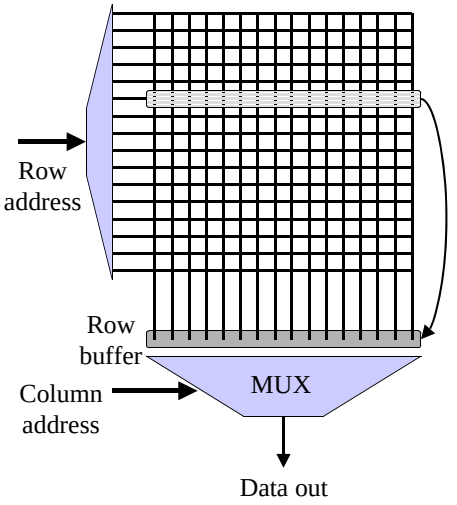
- As with caches, **programs** can be modified to improve memory performance
  - The most important principle is to keep all the banks running
  - Programs that access all banks evenly will perform best
- Problem: data memory references are **not** random and may end up going to the same bank
  - Using a **prime number** of memory banks makes this work well
  - However, using a prime number makes the division operation expensive
    Bank number = Address MOD Number of banks
    Address within bank = floor (Address / Number of banks)

Avoiding memory bank conflicts

- There are schemes distribute memory accesses to banks using
  - A prime number of banks
  - Fast modulo arithmetic
- For example, the following can be used:
  Bank number = Address MOD number of banks
  Address in bank = Address MOD number of words in bank
  - This avoids the use of an expensive 'non power of 2' division operation shown previously
  - There is a proof that guarantees that the above mapping provides a unique mapping between an address and a memory location
  - For numbers of the form 2^N-1, there is fast hardware to implement the MOD operation

Improving DRAM performance

- Previous methods work with any memory technology
- There are also techniques that take advantage of the nature of DRAMs
- DRAMs buffer a row of bits inside the DRAM for column access
  - The size of the buffer is usually the square root of the DRAM size, e.g. 16Kbits for 64MBits
- DRAMs are designed to allow multiple accesses to this buffer, **eliminating** the row access time



DRAM-specific techniques

- Nibble mode
  - The DRAM can supply three extra bits from locations sequential to the one just accessed, once after each RAS (Row Access Strobe)
- Page mode
  - The DRAM can act as an SRAM once a row has been selected
  - For example, random bits from the row can be selected by changing just the column address
  - This can occur until the next RAS or refresh
- Static column mode (Extended Data Out [EDO] RAM)
  - Very similar to page mode
  - No need to toggle (clock) the column access strobe line every time the column address changes
- These optimizations can improve bandwidth by a factor of 4!

DRAM-specific techniques

- Synchronous DRAM (SDRAM)
  - The clock is supplied to the RAM chip, and all signals are synchronized to it
  - This allows the RAM to run at higher speeds
  - Similarly, sequential data can be retrieved faster, at the rate of one bit per clock cycle (similar to page mode)
- VRAM
  - Video RAM is used to drive displays
  - Read or written using a normal interface
  - Read via special interface that outputs rows one bit at a time (good for video displays)
- Modern DRAM chips often output multiple bits at a time (4-8 bits per address)

Virtual memory

- Virtual memory is just another level in the memory hierarchy
  - It allows main memory to cache pages (blocks) normally stored on disk
  - As with caches, the operations performed by virtual memory are transparent to properly-running user programs
- Virtual memory's similarity to caching
  - Block ≡ page
    - Blocks in caches are equivalent to pages in virtual memory
    - Pages are anywhere from 1 KB to 64 KB (though today's page sizes are usually 4+ KB)
  - Miss ≡ page fault
    - A miss in a cache is analogous to a page fault
    - The only difference is the penalty...
      - Millions of clock cycles for VM
      - Tens of clock cycles for caches

Virtual memory

- Miss rate
  - The miss rate for VM is very low -- less than 0.001%
  - Fewer than one in a million accesses causes a VM miss (often lower)
- Size
  - Memory caches are 16 KB - 1 MB or more
  - VM "cache" is 16 MB to 1024 MB or more — a factor of 1000 larger
- Differences include:
  - Replacement mechanism.
    - In caches, it is primarily controlled by the hardware
    - In VM, replacement is primarily controlled by the OS
  - The number of bits in the address determines the size of VM where cache size is independent of the address size
- Two kinds of VM: paging systems and segmentation systems

Basic virtual memory caching questions

- Where can a block be placed?
  - Since miss penalties are very high, OS designers always choose lower miss rates over simple placement algorithms
  - VM is almost always fully-associative (blocks can be placed anywhere in main memory)
- Which block is replaced?
  - Most operating systems use LRU or an approximation to it
  - The page table often includes a reference bit to help do LRU replacement

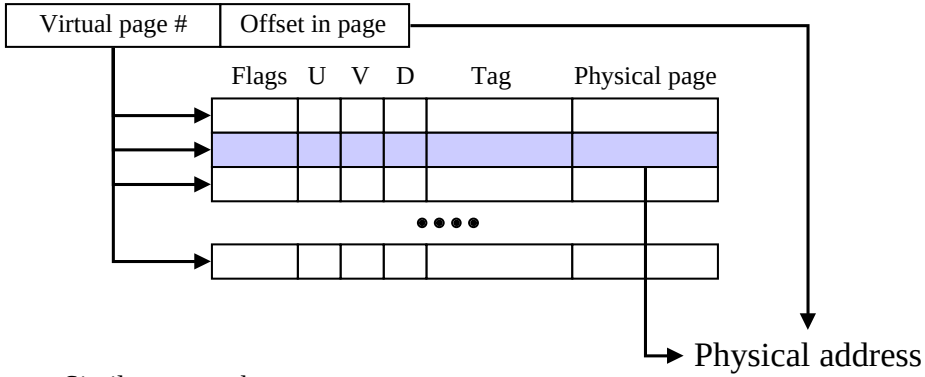
Basic virtual memory caching questions

- How is a block found?
  - Paging systems use a page table to translate virtual page numbers into physical page numbers
  - The physical address is constructed by concatenating the physical page number (found in the table) to the offset
  - Segmented systems use a similar structure except that the segment's physical address is ADDED to the offset
  - The page table needs enough entries to map the entire virtual address space since it is accessed using virtual page numbers
    - This results lots of space dedicated just to the page table
    - One optimization is to use hashing to restrict the number of page table entries to the number of physical pages (inverted page table)
  - Translation lookaside buffers (TLBs) are used to cache these translations, and reduce address translation time

Basic virtual memory caching questions

- What happens on a write?
  - VM is always writeback (capture as many writes as possible before writing the page to disk)
  - Write-through doesn't make sense => very large access penalty
  - The page table uses a dirty bit to keep track which pages have been modified and must be written to disk before they are replaced
  - Don't write pages back to disk unless they've been modified
- Page tables imply that a memory reference requires at least two memory accesses
  - One (or more) for the page table
  - One to get the data
- A TLB, which caches previous translations, can be effective in reducing memory references to the page table (uses locality)

Translation lookaside buffer (TLB)



- Similar to a cache
  - Tag holds the virtual address
  - Data portion holds the physical page frame number, protection field, valid bit, use bit and a dirty bit

Translation lookaside buffer (TLB)

- As with normal caches, the TLB may be fully-associative, direct-mapped, or set-associative
- Replacement may be done in hardware or may be assisted by software
  - For example, a miss in the TLB causes an exception which is handled by the OS, which places the appropriate page information into the TLB
  - Hardware handling is (usually) faster, but software is more flexible
- Small, fast TLBs are crucial because they are on the **critical path** to accessing data from the cache
  - ⇒ This is particularly true if the cache is physically addressed

Selecting page size

- Large page sizes are generally better because
  - They reduce the size of the page table
  - They are more efficient to transfer between memory and disk
  - They allow a TLB to cache translations for more of memory
- The biggest drawback to large pages is that they may waste memory: *internal fragmentation*
  - Assuming a process has three primary segments (text, heap and stack)
  - The average wasted storage per process will be 1.5 times the page size
  - When page size is 4 KB or 8 KB, this is negligible for machines with megabytes of memory
  - For larger pages, e.g., 64+ KB, lots of storage may be wasted
- Variable size pages can be used to get advantages of large pages without internal fragmentation

Memory protection

- VM is often used to protect a program from other programs
  - ⇒ Protection mechanisms must have hardware support
- Base & bounds
  - Each reference must fall between two addresses, given by the base & bound registers
  - This method also allows some relocation
  - User processes cannot be allowed to change these registers, but the OS must be able to do so on a process switch
- Therefore, the hardware must provide:
  - At least two modes of operations, user and kernel mode and a mechanism to switch between them
  - A protection mechanism for other portions of the CPU state to prevent user processes from being malicious

Using virtual memory for protection

- To ensure protection, CPU provides:
  - User/supervisor mode bit(s): separation of user & OS functions
  - Interrupt enable/disable bit(s): atomic operations
- Virtual memory offers a more fine-grained alternative
  - Each process has its own page table, which it cannot modify itself
  - Permission flags are provided with each segment or page
    - Read/write
    - Execute
  - *Concentric rings of security* and *capability lists* are more fine-grained alternatives, allowing more than two levels of protection

⇒ The OS course discusses VM in more detail

Effect of CPU design on memory hierarchy

- Superscalar & vector execution
  - A superscalar or vector machine may fetch several words per cycle
  - Clearly, the memory system must deliver the bandwidth to handle this; otherwise the benefit is lost
  - The brunt of the load falls upon the L1 cache
    - Bandwidth can be increased by widening the path to the cache or by providing extra ports to the cache
    - However, cache access is often the bottleneck in modern CPUs
- Speculative execution
  - Speculative execution and conditional instructions may generate invalid addresses that would not occur otherwise
  - The memory system must recognize and suppress these exceptions
  - Similarly, it must not stall the cache on a miss caused by a speculative instruction

I/O and cache consistency

- I/O devices move data from peripherals to memory
- This has two pitfalls:
  - Data written into memory is not automatically updated in the cache
  - Data in a writeback cache is not written to memory immediately so memory has stale data
- One solution is to flush blocks from the cache that are used in the I/O operation
  - Before the I/O for a write (so the write operation uses up-to-date information)
  - After the I/O for the read (before the I/O should work as well. The CPU should not access the data as it is being read into memory)
- An alternate method is simply to mark the blocks from I/O buffers as *uncacheable*

I/O and cache consistency: other solutions

- Watch the I/O buses for addresses in the tag
  - This eliminates the consistency problem
  - The drawback is that the checking slows down the cache
- Do I/O directly into the cache
  - This method guarantees consistency but it slows down the cache since both the CPU and I/O access it
  - Moreover, it displaces data in the cache with new data that is unlikely to be accessed soon by the CPU,

Stuff to beware of...

- Don't predict cache performance of code A from code B
  - Programs vary widely in how they use cache
  - A scientific program may have a small tight code loop but access large quantities of data
  - On the other hand, a word processing program might operate on relatively little data but use lots of code
- Simulate plenty of memory references
  - A CPU executes 500 million or more instructions per second
  - Simulating cache behavior using traces of only a few million traces can be misleading
  - Program locality behavior isn't constant over the entire program run
- Don't ignore the OS
  - Context switches can have a devastating effect on performance
  - The OS can miss or interfere with application programs, causing misses

Why is I/O so important?

Remember Amdahl's Law:

Speeding up part of the problem while largely ignoring the rest leads to diminishing returns.

This means that focusing on the CPU will result in larger fractions of time spent on I/O.

Response time versus **throughput**

It can be argued that I/O speed does not matter in a multiprogrammed environment.

If a process waits for a peripheral, run another task.

This is an argument that performance is measured as throughput and response time does not matter.

This is hardly the case today with desktop computers running interactive software.

Response time is directly related to productivity.

Storage devices

Storage devices are **mechanical**, so they are subject to real physical limitations.

For example, it is not possible to spin a disk at 100,000 RPM.

A disk made of any material that we know of today would fly apart.

These mechanical delays are far more difficult to eliminate than electronic delays.

Reducing the size of mechanical devices is more difficult than reducing the size of electronic circuits.

We will focus on storage devices of highest capacity:

- Magnetic disks.
- Magnetic tapes.
- CD-ROMS.
- Automated tape libraries.

Magnetic disks

Magnetic disks have dominated *secondary storage* since 1965.

Magnetic disks are used for two main purposes.

- They provide the lower level of the virtual memory system.
- They provide long-term, nonvolatile storage for files.

Disks are usually the highest level of the storage pyramid that is non-volatile.

Important terms associated with disks:

- *Platter*

Disks are built from several metal platters (1 to 20) covered with metal oxides or other magnetic recording media.

Platters are 1 - 8 inches wide.

- *Surface*

Each platter has two surfaces, both of which may be used for recording.

Magnetic disks

Important terms associated with disks:

- *Track*

A surface is divided into tracks, which are concentric circles containing data.

There are 500 - 2500+ tracks per surface.

The set of tracks at corresponding locations on all of the surfaces is called a cylinder.

- *Sector*

A track is divided into sectors each of which holds a fixed amount of data, usually 512 - 4096 bytes.

Older disks have a constant number of sectors per track.

Newer disks record more sectors on the outer tracks using a **constant bit density**.

Magnetic disks

Important terms associated with disks:

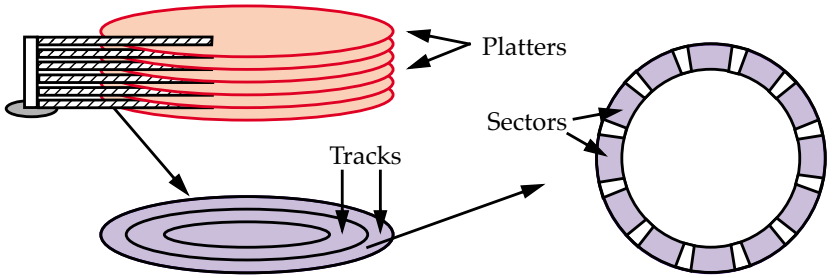
- *Disk arm/head*

Each surface is read and written by its own head.

All of the heads in a drive are connected together and move as a single unit.

Most disks have just one such unit.

However, more expensive drives may have two or more assemblies to increase the I/O rate.



Magnetic disks

Performance

There are three main components of each disk access:

- Seek time.
- Rotational delay.
- Transfer time.

Seek time

The seek time is the time necessary to move the arm from its previous location to the correct track for the current I/O request.

The industry standard is to use **average seek time**.

This is computed by averaging the time necessary for all possible seeks.

This tends to **overestimate** actual average seek time because of locality of disk references.

Advertised average seek times of 8 ms to 12 ms may be smaller by 25% to 33%.

Magnetic disks

Rotational delay

The time necessary for the requested sector to rotate under the head.

Most disks rotate at 3600 - 7200 RPM (note that disks rotated at 3600 RPM in 1975!).

On average, the disk must rotate *halfway* around to get to a desired sector.

On a disk rotating at 6000 RPM, this delay is 0.5 / (6000 RPM / 60,000 ms/min) = 5 ms.

Therefore, the two mechanical components, moving the disk arm and waiting for the data to rotate under the head, add to the latency.

Some disks can **read data out of order** into a buffer, reducing rotational delay for a large transfer.

A full track can be transferred in one rotation regardless of where the I/O actually starts.

Magnetic disks

Transfer time

The transfer time is the time it takes to read or write a sector.

For the disk, this is approximately equal to the time it takes for the sector to *pass fully* under the read/write head.

It is a function of the block size, rotation speed, recording density and the speed of the electronics.

Typical rates in 1995 were 2 to 8 MB/sec.

A disk rotating at 6000 RPM with 64 sectors per track and 512 bytes per sector can read a **sector** in 10 ms/64 = 0.156 ms.

Since 0.5 KB are transferred in this time, bandwidth is 3.2 KB/ms, or 3.2 MB/s.

Note that the transfer rate is higher for sectors on outside tracks (on disks that have variable number of sectors per track.)

Magnetic disks

Other components to latency

Note, too, that other parts of the system can add delays.

Controllers and I/O buses can add their own delays, possibly decreasing bandwidth and increasing latency.

For example:

Suppose we have a new disk and want to see how quickly we can read a single sector.

Each sector is 1 KB long, tracks have 32 sectors each, average seek time is 8 ms, and the disk rotates at 7200 RPM.

The controller adds 2 ms overhead to each request.

$$\text{Total time} = 8ms + \frac{0.5}{120RPS} + \frac{1}{120RPS \times 32} + 2ms$$

$$\text{Total time} = 8ms + 4.17ms + 0.26ms + 2ms = 14.43ms$$

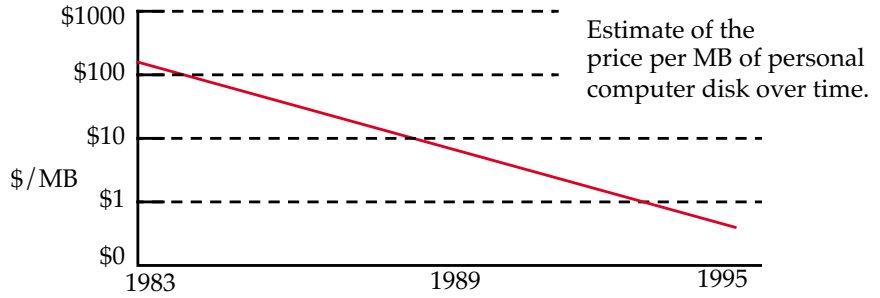
The Future of Disks

- *Density*

Disk capacity is measured in **areal density** (the number of bits per square inch).

This is the product of tracks per inch on a surface and bits per inch on a track.

Density has recently improved at a rate of 60% per year, matching density increases of DRAM.



The Future of Disks

- *Transfer rate*

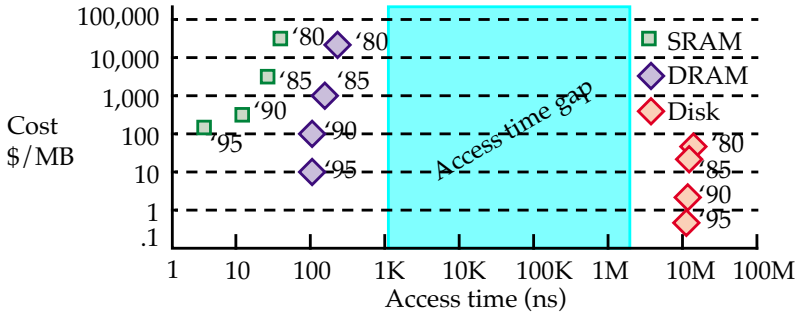
Since transfer rate is proportional to bits per track, improvements in **density** usually lead to higher bandwidth.

This increase is usually the **square root** of overall density increase.

Seek time is reduced by smaller disks and lighter heads.

However, seek time is still limited by mechanical considerations.

The Access Time Gap



The Future of Disks

In 1995, the price/megabyte of Disk is about 100 times cheaper than the price/megabyte of DRAM.

However, DRAM is about 100,000 time faster.

Many have tried to fill the **Access Time Gap** with a new technology.

So far, nobody has succeeded.

Technologies such as bubble memory and flash RAM have been proposed, but usually end up being "taken over" by either memory or disk.

Other Storage Devices

A challenger to magnetic disks.

- *Optical disks (CDs)*

Optical disks contain data stored optically rather than magnetically.

Optical storage is removable and inexpensive to manufacture.
For example, it costs $1 to $2 per 500MB CD-ROM.

CDs are a popular medium for the distribution of software.

Optical disks are usually write-once.
They are also called **WORM** (Write Once, Read Many) storage.

Agreement on standards for CDs have slowed their time-to-market, allowing magnetic disks to stay ahead.

However, writable optical disks may have the potential to compete with new tape technologies for archival storage.

Other Storage Devices

- *Magnetic tapes*

Magnetic tapes use the same basic kind of recording technology as magnetic disks.

The major difference:

- For tapes, the read/write heads actually touch magnetic tapes.
- For disks, they *float* above the disk surfaces (the Bernoulli effect).

Tapes are cheaper than disks for two reasons:

- They pack more medium into a fixed size by recording on long strips.
- They are removable (allowing one reader to access multiple tapes).

This means that a 9GB cartridge can cost less than $20.
$2/GB is considerably less than the $30/GB for magnetic disk.

Other Storage Devices

- *Magnetic tapes*
  - **Longitudinal scan (linear scan)**

This is similar to audio cassette tapes.
Data is written in a track that runs the length of the tape.

As with audio tape, a single tape may have more than one track, and the tracks may run in opposite directions.

  - **Helical scan**

These tapes put their heads on a rapidly spinning cylinder, which spin at on a diagonal to the tape.

By increasing the speed of the heads relative to the tape, they allow much denser storage (20 to 50 times).

The problem is that they cannot search as quickly, and they wear out more quickly (100's of passes versus >1000's of passes for longitudinal.)

Other Storage Devices

- *Robots (tape or optical disk)*

Robots can be used automatically move media (tapes or optical disks) between drives and shelves.

This allows **terabytes** of data to be accessible within ten's of seconds.

Since reader cost is fixed, tape robots cost less per unit of storage as more media share the same number of readers
The robot cost is also amortized.

Of course, more tapes per drive means more contention for drives and lower bandwidth per tape.

A large tape robot can cost as little as $40 per GB for a full system.

Other Storage Devices

- *Holographic storage (future?)*

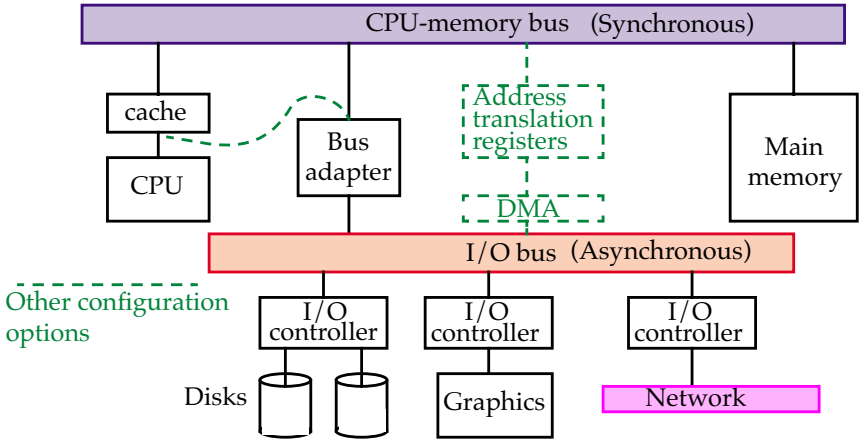
  There is lots of research on using cubes of *light-sensitive* material to store information holographically.

  A device would read or write an entire "slice" at once using lasers.
  This would provide very high bandwidth and high storage density.

  There are currently several problems researchers are working to solve.
  In particular, making this medium read/write is difficult.

Buses

The bus serves as a shared communication link between the subsystems of a computer, e.g. the CPU, memory and the I/O devices.



Buses

Bus advantages:

- They are *low cost.*

  A single set of wires is used to connect multiple components.
- They are *versatile.*

  New devices can be easily added or removed.

Disadvantage:

- They create a *communication bottleneck*, possibly limiting I/O throughput.

Two types:

- *CPU-memory buses*

  Short, generally high speed and matched to maximize memory-CPU bandwidth.
- *I/O buses*

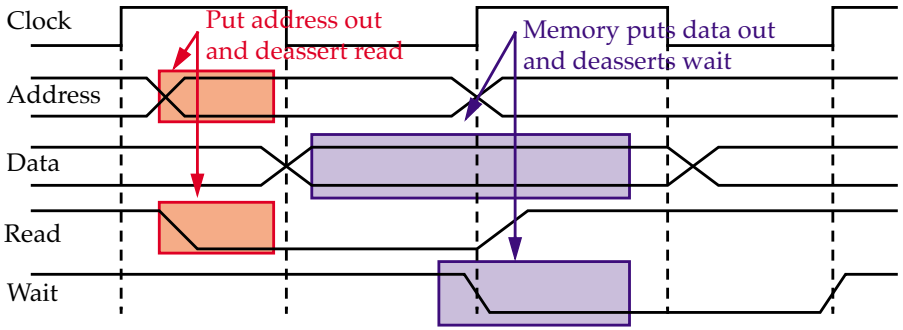
  Lengthy with many types of devices connected to them, each of which may have different data bandwidths.
  They usually follow a standard.

Buses

Basic bus transactions:

- If the transaction is a write (to memory), the data may be sent immediately after (or perhaps in parallel with) the address.
- For reads (from memory), there is usually a delay between the time the address is sent and the time the data is ready.
  This delay occurs because of the memory latency.

Typical bus read transaction (on a synchronous bus):



Bus Design Decisions

Cost versus performance trade-offs:

Option	Better performance	Lower cost
Bus width	Separate address & data lines (adv for writes)	Multiplex address & data lines
Number of bus lines	Wider is faster	Narrower is cheaper
Transfer size	Multiple words have less bus overhead	Single word transfer has simpler logic
Bus masters	Multiple, arbitrated	Single, no arbitration
Split transaction	Yes. Separate request and reply packets get higher bandwidth	No. Continuous connection is cheaper and has lower latency
Clocking	Synchronous	Asynchronous

The choice of using any of the options listed in the first three rows is straightforward.

All give higher performance at more cost.

Bus Options

Bus masters

A bus master is a device on the bus that has the power to initiate a transaction.

The CPU is always a bus master.

I/O devices can also be bus masters.

With multiple masters, an arbitration scheme is required to determine who gets the bus next.

Split transactions

For many buses, a transaction must be performed atomically.

This means the bus is busy from the time the data is request until the time the request is complete.

This may work well for memory, but it can be intolerable for I/O where data may not be ready for 10ms after the request.

Bus Options

Split transactions

In systems that have **multiple bus masters**, split transactions provide an improvement in bandwidth.

In a split transaction, the read request is sent and the bus is *released* while the data is prepared.

Once memory has retrieved the data, it arbitrates for the bus and 'tags' the data reply for the CPU.

This makes the bus available for other bus masters while memory reads the words from the requested address.

Split transactions can be even more beneficial for I/O buses.

For example, a SCSI bus can have **several** long disk transactions in progress at the same time.

However, a split transaction may have *higher latency* than a bus that is held during the complete transaction (due to arbitration.)

Bus Options

Synchronous vs. asynchronous

Synchronous buses require a fixed protocol for addresses and data relative to a global clock that all devices agree upon.

Signals are only considered valid at certain points on the clock waveform (usually the rising or falling edge).

If all devices are equally "fast," synchronous works well and it is inexpensive and fast.

Two disadvantages:

- Cheap devices, such as keyboards, must have logic that is able to run under the same fast clock rate used for disks.
- Clock-skew problems limit the length of synchronous buses.

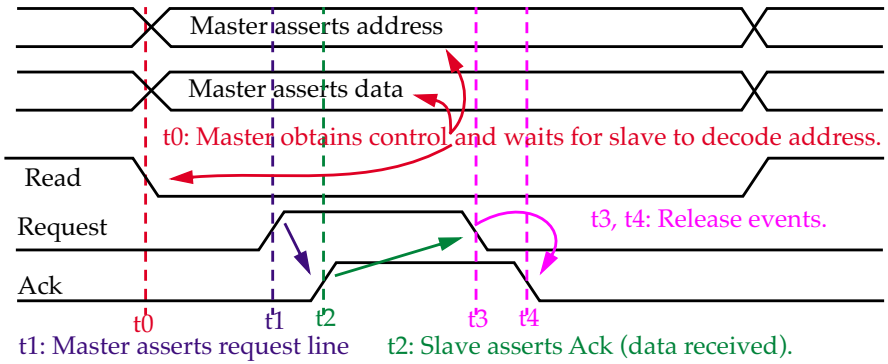
CPU-memory buses are typically synchronous.

Bus Options

Synchronous vs. asynchronous

The alternative is *asynchronous logic*, which uses hardware handshaking to exchange information on the bus (usually an **I/O bus**).

Typical write transaction on a asynchronous bus:



Asynchronous buses address previous disadvantages are usually slower because of the overhead associated with the handshaking protocol.

Performance Metrics

- *Throughput*

  For I/O devices, throughput is usually called *I/O bandwidth*.

  This measures the amount of data a particular I/O device can transfer in a given period of time.

- *Response time*

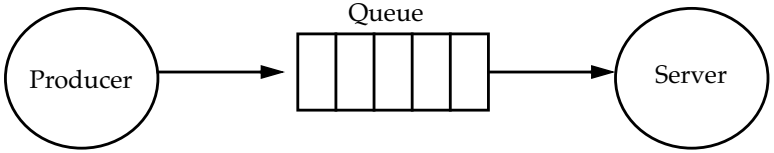
  This is called *I/O latency*.

  This measures the time taken from when a request is placed into a device queue until the time that service is completed.

  Note that this may include time spent waiting in a queue as well as the usual service time.

Performance Metrics

Trading off throughput & response time.



For minimum **response time**, queues should be kept empty so an incoming request can be serviced as soon as possible.

However, maximum **bandwidth** makes the opposite true.

Try and keep the queue full so the device always has something to do.

Throughput can be improved in many ways.

For example, a designer could add additional disks to a system allowing more requests to be serviced at a given time.

Requests would likely spend less time waiting in queues, which **may** improve response time as well.

Human-computer interaction

In general, improving performance does not always improve both response time and throughput.

Given that this is true, which is more important ?

In a multiprogramming environment, one might say *throughput*.

However, user productivity is **non-linearly** related to *response time* as shown below.

Interactions between humans and computers are divided into three “types” of time:

- *Entry time*

  This is the time to enter a command into the computer.

- *System response time*

  The delay between when the command is completed and the full answer is returned.

- *Think time*

  The time from the reception of the response until the user begins to enter the next command.

Human-computer interaction

The sum of these times is the **transaction time**.

One study shows that user productivity is inversely proportional to transaction time.

In particular, people get more productive as the computer reacts more quickly.

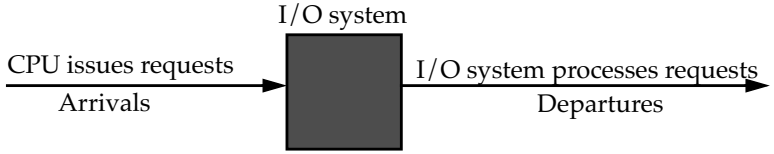
In other words, reducing response time actually decreases transaction time by more than just the reduction in response time.

In general, people need less time to think when given a faster response time.

Queuing theory

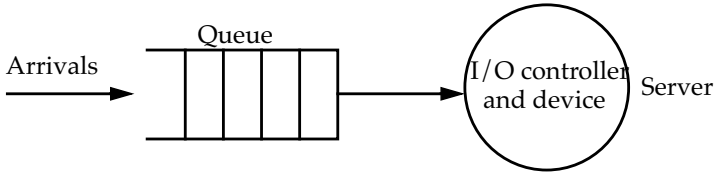
Given the important of response time (and throughput), we need a means of computing values for these metrics.

Our black box model:



Let's assume our system is in steady-state (input rate = output rate).

The contents of our black box.



I/O requests "depart" by being completed by the server.

Queuing theory

Elements of a queuing system:

- *Request & arrival rate*
  - This is a single "request for service".
  - The rate at which requests are generated is the *arrival rate*.
- *Server & service rate*
  - This is the part of the system that services requests.
  - The rate at which requests are serviced is called the service rate.
- *Queue*
  - This is where requests wait between the time they arrive and the time their processing starts in the server.

Queuing theory

Useful statistics

- $\text{Length}_{\text{queue}}$, $\text{Time}_{\text{queue}}$
  - These are the average length of the queue and the average time a request spends waiting in the **queue**.
- $\text{Length}_{\text{server}}$, $\text{Time}_{\text{server}}$
  - These are the average number of tasks being serviced and the average time each task spends in the **server**.
  - Note that a server may be able to serve *more than one* request at a time.
- $\text{Time}_{\text{system}}$, $\text{Length}_{\text{system}}$
  - This is the average time a request (also called a task) spends in the **system**.
  - It is the sum of the time spent in the queue and the time spent in the server.
  - The length is just the average number of tasks anywhere in the system.

Queuing theory

Useful statistics

- *Little's Law*
  - The **mean number of tasks in the system = arrival rate * mean response time**.
  - $\text{Length}_{\text{System}} = \text{Arrival Rate} \times \text{Time}_{\text{System}}$
  - This is true only for systems in equilibrium.
    - We must assume any system we study (for this class) is in such a state.
- *Server utilization*
  - This is just
  - $\text{Server utilization} = \frac{\text{Arrival Rate}}{\text{Server Rate}}$ where Rate = 1/Time
  - This must be between 0 and 1.
    - If it is larger than 1, the queue will grow infinitely long.
  - This is also called *traffic intensity*.

Queuing theory

Queue discipline

This is the order in which requests are delivered to the server.

Common orders are FIFO, LIFO, and random.

For FIFO, we can figure out how long a request waits in the queue by:

$$\text{Time}_{\text{System}} = \text{Length}_{\text{Queue}} \times \text{Time}_{\text{Server}} + \text{Mean time for server to finish current tasks when request arrives}$$

The last parameter is the hardest to figure out.

We can just use the formula:

$$\text{Average residual service time} = \frac{1}{2} \times \text{Weighted mean time} \times (1 + C)$$

C is the coefficient of variance, whose derivation is in the book.
(don't worry about how to derive it - this isn't a class on queuing theory.)

Queuing theory

Example: Given:

- Processor sends 10 disk I/O per second (which are exponentially distributed).
- Average disk service time is 20 ms.

On average, how utilized is the disk?

$$\text{Server utilization} = \frac{\text{Arrival Rate}}{\text{Server Rate}} = \frac{10}{\frac{1}{0.02}} = 0.2$$

What is the average time spent in the queue?

When the service distribution is exponential, we can use a simplified formula for the average time spent waiting in line:

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 20ms \times \frac{0.2}{(1 - 0.2)} = 5ms$$

What is the average response time for a disk request (including queuing time and disk service time)?

$$\text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 5 + 20ms = 25ms$$

Queuing theory

Basic assumptions made about problems:

- System is in equilibrium.
- Interarrival time (time between two successive requests arriving) is exponentially distributed.
- Infinite number of requests.
- Server does not need to delay between servicing requests.
- No limit to the length of the queue and queue is FIFO.
- All requests must be completed at some point.

This is called an M/G/1 queue

M = exponential arrival
G = general service distribution (i.e. not exponential)
1 = server can serve 1 request at a time

It turns out this is a good model for computer science because many *arrival* processes turn out to be **exponential**.

Service times, however, may follow any of a number of distributions.

Disk Performance Benchmarks

We use these formulas to *predict* the performance of storage subsystems.

We also need to measure the performance of real systems to:

- Collect the values of parameters needed for prediction.
- To determine if the queuing theory assumptions hold (e.g., to determine if the queueing distribution model used is valid).

Benchmarks:

- *Transaction processing*

  The purpose of these benchmarks is to determine how many small (and usually random) requests a system can satisfy in a given period of time.

  This means the benchmark stresses **I/O rate** (number of disk accesses per second) rather than **data rate** (bytes of data per second).

  Banks, airlines, and other large customer service organizations are most interested in these systems, as they allow simultaneous updates to little pieces of data from many terminals.

Disk Performance Benchmarks

- *TPC-A and TPC-B*

  These are benchmarks designed by the people who do transaction processing.

  They measure a system's ability to do random updates to small pieces of data on disk.

  As the number of transactions is increased, so must the *number of requesters* and the *size of the account file*.

  These restrictions are imposed to ensure that the benchmark really measures disk I/O.

  They prevent vendors from adding more main memory as a database cache, artificially inflating TPS rates.

- *SPEC system-level file server (SFS)*

  This benchmark was designed to evaluate systems running Sun Microsystems network file service, NFS.

Disk Performance Benchmarks

- *SPEC system-level file server (SFS)*

  It was synthesized based on measurements of NFS systems to provide a reasonable mix of reads, writes and file operations.

  Similar to TPC-B, SFS **scales** the size of the file system according to the reported *throughput*, i.e.,

  It requires that for every 100 NFS operations per second, the size of the disk must be increased by 1 GB.

  It also limits average *response time* to 50ms.

- *Self-scaling I/O*

  This method of I/O benchmarking uses a program that **automatically scales several** parameters that govern performance.

  - Number of unique bytes touched.

    This parameter governs the total size of the data set.

    By making the value large, the effects of a cache can be counteracted.

Disk Performance Benchmarks

- *Self-scaling I/O*
  - Percentage of reads.
  - Average I/O request size.

    This is scalable since some systems may work better with large requests, and some with small.

  - Percentage of sequential requests.

    The percentage of requests that sequentially follow (address-wise) the prior request.

    As with request size, some systems are better at sequential and some are better at random requests.

  - Number of processes.

    This is varied to control concurrent requests, e.g., the number of tasks simultaneously issuing I/O requests.

Disk Performance Benchmarks

- *Self-scaling I/O*

  The benchmark first chooses a nominal value for each of the five parameters (based on the system's performance).

  It then varies each parameter in turn while holding the others at their nominal value.

  Performance can thus be graphed using any of five axes to show the effects of changing parameters on a system's performance.

Reliability, Availability and RAID

Some definitions:

- *Reliability*

  Refers to the dependability of individual components of a system.

- *Availability*

  Is the system still available to the user after a failure of one or more of its components ?

Adding hardware can therefore improve *availability* but it can **NOT** improve *reliability.*

Disk arrays:

The basic idea behind **disk arrays** is that by adding disks and therefore more disk arms working in parallel, *bandwidth* is improved.

Individual process-request seek latencies can be overlapped in time.

This is cost effective since price/megabyte is independent of disk size.

However, *latency* for small requests is not improved because it still takes all of the usual latency to get to a randomly selected block.

Reliability, Availability and RAID

- *Striping*

  In disk arrays, the data from files can be **striped** across several disks.

  This *increases bandwidth* by allowing a file to be read from *more than one* disk at a time.

  The data is distributed round robin between the disks.

Problems with disk arrays:

- *Reliability*

  Disks have a *mean time to failure* of about 20 years.

  However, a collection of 8 disks will experience a failure (on average) every 2.5 years.

  N devices generally have 1/N the reliability of a single device.

Reliability, Availability and RAID

Problems with disk arrays:

- *Reliability*

  Increase the number to 1000 disks, and a failure occurs every 1/50th of a year or every week !

  When a disk fails, it takes its data with it.

- *Availability*

  The other problem with disk arrays is that the disk array becomes unusable after a single failure.

We need a scheme to prevent data loss when a disk fails, and to allow the system to recover from the failure (remain available.)

Reliability, Availability and RAID

The solution: **Redundant Arrays of Inexpensive Disks.**

Improves availability by adding *redundant* disks.

When a disk fails, the lost information can be reconstructed from redundant information.

This works since the *mean time to failure* (MTTF) of disks is long (years) and the *mean time to repair* (MTTR) is short (hours).

The idea behind RAID is to use some of the disks to store error correction information for the rest of the disks.

RAIDs do NOT have to detect errors.

The ECC kept on each sector by each disk allows the disk electronics to check and detect disk failures.

RAID Levels

There are 7 levels of RAID, each of which can be characterized by their availability and overhead.

Raid Level		Failures survived	Data Disks	Check Disks
0	Nonredundant	0	8	0
1	Mirrored	1	8	8
2	Memory-style ECC	1	8	4
3	Bit-interleaved parity	1	8	1
4	Block-interleaved parity	1	8	1
5	Block-interleaved distributed parity	1	8	1
6	P+Q redundancy	2	8	2

- *RAID 0*

This is no redundancy at all.

However, it is the fastest and cheapest RAID level.

This refers to the nonredundant disk array discussed previously.

RAID Levels

- *RAID 1*

Disks in this configuration are mirrored or copied to another disk.

With this arrangement, the data on a failed disk can be easily replaced by reading it from the other disk.

In addition, **reading** is actually *faster* than it is for RAID 0 because read requests to the same disk can be split between the two disks.

Writing is a little *slower*, though, because the file system must wait for the slower of the two requests.

Remember, both disks must be updated.

Also, seek time is different between the two disks since they are not synchronized.

The main problem with RAID 1 is that it imposes a 50% space penalty.

Therefore, it is the most expensive solution.

RAID Levels

- *RAID 3 (Bit interleaved parity)*

In this scheme, data is striped across disks in *very small units*.

These units are so small that all disks must work together on both reads and writes because a **single sector** actually spans **all** of the disks.

Redundancy is implemented by *calculating parity* and storing it on the check disk.

The overhead for **n** data disks is **1** disk, for a storage efficiency of **n/(n+1)**.

This level can survive a single disk failure and reconstruct data using the remaining disks.

This RAID level is somewhat limited since the entire disk system can only handle one request at a time.

Thus, the sustainable request rate is no higher than that of a single disk.

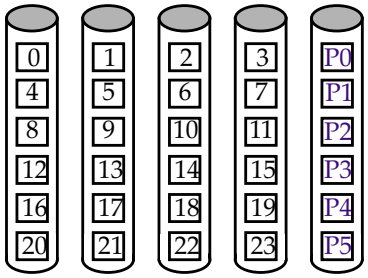
RAID Levels

- *RAID 4 (Block interleaved parity)*

This is similar to RAID 3, but individual disks each have a **block** (or more) of consecutive data within a stripe.

This means that *each disk* can handle an individual small read request if all disks are working.

However, writes are still a problem since the *parity disk* must participate in all write operations, which creates a bottleneck.



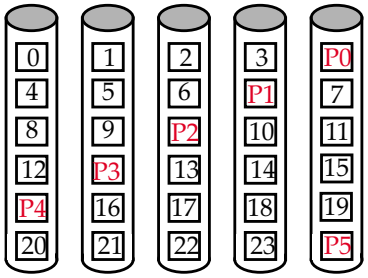
RAID 4

RAID Levels

- *RAID 5 (Rotated parity)*

The biggest problem with RAID 4 is that the parity disk is a bottleneck.

In RAID 5, the parity information may be stored on any disk.



RAID 5

Advantages:

Reads can be spread among all n+1 disks.

For writes, the parity disk bottleneck is eliminated.

RAID Levels

- *RAID 5 (Rotated parity)*

With respect to the parity calculation and update, RAID 5 does have a problem writing small pieces of data.

If a full stripe is written, the parity calculation is simple to implement.

XOR all the data together to get the parity.

If only a **single block** is written, though, the system must somehow figure out the new parity.

The most straightforward solution is to read the rest of the blocks in the stripe and XOR them together.

A better solution involves only four accesses:

- Read the old data.
- Compare old data to new data (to determine which bits change).
- Read and change the old parity.
- Write the new data and new parity.

RAID Levels

- *RAID 6 (P+Q parity)*

The objective of RAID 6 is to survive two disk failures.

This is done using schemes such that only **n** out of **n+2** disks are necessary to reconstruct the data.

This RAID level is not common today because disk failures are still relatively rare.

If a disk in a parity-based RAID fails, no data is lost if a new disk is installed and "updated" before another crash takes place.

Since stripes tend to be relatively small (usually less than 32 disks), the chance that another disk will fail is relatively low.

Most people don't worry about it.

RAID Issues:

- *Mapping data to disks.*

The first problem in a RAID system is how to map data to disks.

Usually, this is done using simple modulo arithmetic.

It is more complex for RAID 5, but there is still a formula that allows you to determine where a given block is located.

- *Reconstruction*

After a disk has failed, it is replaced.

But we are not done since the new disk must be reconstructed with the lost information in order to allow the array to withstand another crash.

Since the lost information is "embedded" in the remaining disks, reconstruction is possible and carried out immediately.

Designing an I/O system: Basics

Now that we've seen how to *estimate* performance on an I/O system.
And how to actually *measure* performance.
We are ready to talk about how to build one.

The objective is to find a design that is expandable and that meets goals for **cost** and **variety of devices** while *avoiding bottlenecks* to I/O performance.

In designing an I/O system, analyze *performance*, *cost* and *capacity* using *various I/O connection schemes* and *different numbers* of I/O devices of each type.

Here are the steps to follow in designing an I/O system:

- *List the types of I/O devices and buses, and their costs.*
- *List the physical requirements of each device.*
  These include volume, power, connectors, bus slots, etc.
  This won't be a problem for paper examples, but it certainly will be for real systems.

Designing an I/O system: Basics

- *Figure out the CPU resource demands for each I/O device.*
  Clock cycles to initiate, support the operation of, and complete requests.
  Clock stalls from I/O access to memory.
  Clock cycles to recover from an I/O activity, such as a cache flush.

- *List memory and I/O bus resource demands for each device.*
  Bandwidth of main memory and the I/O bus can often be a bottleneck, particularly when many devices are connected to a single bus.

- *Compute performance for various configurations of devices, buses, etc.*
  This can really only be done by building the system and measuring it.

  If this is not feasible (which is usually the case), the next best thing is a detailed simulation.

  Queuing models can be used to get a rough estimate of performance.

Designing an I/O system: Basics

Remember, performance can be measured as:

- Megabytes per second.
- I/Os per second.

This is dependent on the needs of the applications.

The goals for the design should also be clear:

- Is it a design to maximize performance at any cost ?
- Is it the cheapest system that will satisfy minimum requirements ?
- Is it the best price/performance ?

Look over the examples in the text !

Tertiary storage

Many modern computer systems now use **removable media** to store their data.

Advantages:

- *Inexpensive*
  Removable media cost a lot less because you only pay once for the machinery to read, write, and transport the medium.
  Since removable media use similar technology to non-removable media, the media costs are similar but the mechanism cost is much lower.

- *Low power*
  Disks use power to rotate.
  A major advantage of removable media is that they do not consume power.

- *Unerasability*
  Some removable media (WORM) can not be erased even if the system requests it.

Tertiary storage

Components:

- *Tape robots*

These systems typically hold thousands of tape cartridges and can load any cartridge in under 20 seconds.

IBM 3490 cartridges (the most common today) hold 9 GB of data per cartridge and transfer at 9 MB/sec.

- *Optical disk jukeboxes*

These are usually used for two purposes:

Small randomly-accessed data.

Data that should not ever be overwritten (even accidentally).

The cost per GB is higher than for tape, but seek time is much better.

Tertiary storage

How it works:

- *Moving data from disk to tertiary*

Data is moved from disk to tertiary storage when the disk gets full.

This is called *file migration* or simply *migration*.

Migrating data is done to free up disk space.

Files are picked for migration according to several factors:

- How big they are.
- When they were last used.
- Cost to retrieve the file.
- Other factors (possibly file type and/or user).

The device to which the file is migrated can also depend on these factors.

Small files might go to optical disk while large files are sent to a tape robot.

Migration can also occur from one tertiary storage device to another.

Tertiary storage

- *Moving data from tertiary to disk*

Data is moved from tertiary to disk when it is needed.

It may also be prefetched if the system believes that the file might be used soon.

File migration issues

Tertiary storage is very much an *ad hoc* art these days.

System designers build their systems based on what others have done because there is relatively little concrete research on what works and what does not.

Tertiary storage

File migration issues

- *When should a file be migrated ?*

How should the system choose the files to move from disk to tape ?

This is very important because a user notices even a single miss.

It may take close to a minute to retrieve a file from tape !

Migrating just one file that should not have been moved can adversely impact a user's session.

- *When should a file be deleted from disk ?*

Just because a file has been migrated to tape does not mean it should be deleted from disk.

When should its space be reclaimed ?

Tertiary storage

File migration issues

- *When should a file be moved from tape to disk ?*

  For demand fetches, this is obvious.

  However, there is also **prefetching** and **clustering** that might help improve performance.

- *What kinds of devices and layouts work best for various kinds of files ?*

  Again, clustering is important, as is transfer time versus time to first byte.

Fallacies and Pitfalls

- *Media cost is not the same as actual cost.*

  Just because a disk costs \$0.20/MB does **NOT** mean you can pay \$200,000 and get a terabyte of disk.

  There are lots of other costs associated with I/O systems.

  For example, disks need controllers, I/O buses, system buses, power supplies, and mounting hardware.

  This supporting structure becomes much more expensive as the number of disks supported grows.

  The same is true for removable media.

  It only costs \$1000 to buy a terabyte of magnetic tape.

  But that does not include tape readers, a robot, software, and all the other pieces necessary to build a full system.

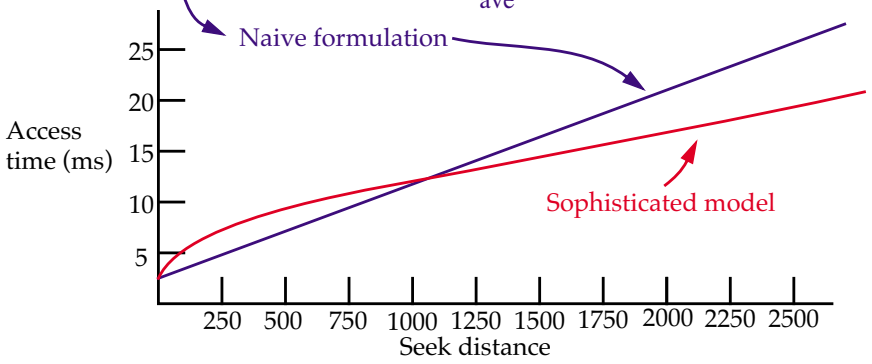
Fallacies and Pitfalls

- *Disk seek time is not linear.*

  A disk head must accelerate to maximum velocity, travel across tracks, decelerate, and settle.

  Most of the head's time is spent accelerating and decelerating, a non-linear activity.

$$\text{Time}_{seek} = \text{Time}_{min} + \frac{\text{Distance}}{\text{Distance}_{ave}} \times (\text{Time}_{ave} - \text{Time}_{min})$$



Fallacies and Pitfalls

- *Disk seek time is not linear.*

  For disks with more than 200 cylinders, Chen and Lee [1995] modeled the seek distance as:

$$\text{Seek time (distance)} = a \times \sqrt{\text{Distance} - 1} + b \times (\text{Distance} - 1) + c$$

  where a, b and c were computed as:

$$a = \frac{-10 \times \text{Time}_{min} + 15 \times \text{Time}_{ave} - 5 \times \text{Time}_{max}}{3 \times \sqrt{\text{Number of cylinders}}}$$

$$b = \frac{7 \times \text{Time}_{min} - 15 \times \text{Time}_{ave} + 8 \times \text{Time}_{max}}{3 \times \text{Number of cylinders}}$$

$$c = \text{Time}_{min}$$

  The curve represented by this model in shown on the previous slide in red.

Fallacies and Pitfalls

- *I/O will become more important with time.*

  As our society stores more and more information on computer media, the ability to get to that information will become ever more important.

  The NASA Mission to Project Earth will capture more than a terabyte of data per day from satellites.

  How can we find a needle in that haystack ?

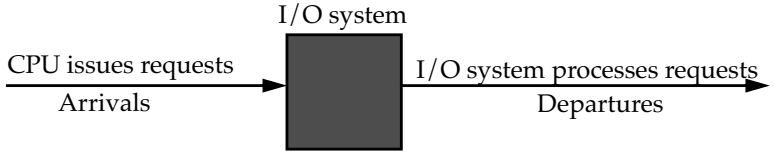
  Similarly, future libraries may dispense with physical books and instead keep information online.

  This makes it easier to distribute the information, but getting data to and from storage will be a bottleneck unless progress is made.

Queuing theory

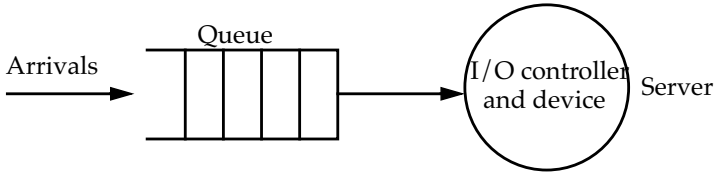
Given the important of response time (and throughput), we need a means of computing values for these metrics.

Our black box model:



Let's assume our system is in steady-state (input rate = output rate).

The contents of our black box.



I/O requests "depart" by being completed by the server.

Queuing theory

Elements of a queuing system:

- *Request & arrival rate*
  - This is a single "request for service".
  - The rate at which requests are generated is the *arrival rate*.
- *Server & service rate*
  - This is the part of the system that services requests.
  - The rate at which requests are serviced is called the service rate.
- *Queue*
  - This is where requests wait between the time they arrive and the time their processing starts in the server.

Queuing theory

Useful statistics

- $\text{Length}_{\text{queue}}$, $\text{Time}_{\text{queue}}$
  - These are the average length of the queue and the average time a request spends waiting in the **queue**.
- $\text{Length}_{\text{server}}$, $\text{Time}_{\text{server}}$
  - These are the average number of tasks being serviced and the average time each task spends in the **server**.
  - Note that a server may be able to serve *more than one* request at a time.
- $\text{Time}_{\text{system}}$, $\text{Length}_{\text{system}}$
  - This is the average time a request (also called a task) spends in the **system**.
  - It is the sum of the time spent in the queue and the time spent in the server.
  - The length is just the average number of tasks anywhere in the system.

Queuing theory

Useful statistics

- *Little's Law*
  - The **mean number of tasks in the system = arrival rate * mean response time**.
  - $\text{Length}_{\text{System}} = \text{Arrival Rate} \times \text{Time}_{\text{System}}$
  - This is true only for systems in equilibrium.
    - We must assume any system we study (for this class) is in such a state.
- *Server utilization*
  - This is just
  - $\text{Server utilization} = \frac{\text{Arrival Rate}}{\text{Server Rate}}$ where Rate = 1/Time
  - This must be between 0 and 1.
    - If it is larger than 1, the queue will grow infinitely long.
  - This is also called *traffic intensity*.

Queuing theory

Queue discipline

This is the order in which requests are delivered to the server.

Common orders are FIFO, LIFO, and random.

For FIFO, we can figure out how long a request waits in the queue by:

$$\text{Time}_{\text{System}} = \text{Length}_{\text{Queue}} \times \text{Time}_{\text{Server}} + \text{Mean time for server to finish current tasks when request arrives}$$

The last parameter is the hardest to figure out.

We can just use the formula:

$$\text{Average residual service time} = \frac{1}{2} \times \text{Weighted mean time} \times (1 + C)$$

C is the coefficient of variance, whose derivation is in the book.
(don't worry about how to derive it - this isn't a class on queuing theory.)

Queuing theory

Example: Given:

- Processor sends 10 disk I/O per second (which are exponentially distributed).
- Average disk service time is 20 ms.

On average, how utilized is the disk?

$$\text{Server utilization} = \frac{\text{Arrival Rate}}{\text{Server Rate}} = \frac{10}{\frac{1}{0.02}} = 0.2$$

What is the average time spent in the queue?

When the service distribution is exponential, we can use a simplified formula for the average time spent waiting in line:

$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})} = 20ms \times \frac{0.2}{(1 - 0.2)} = 5ms$$

What is the average response time for a disk request (including queuing time and disk service time)?

$$\text{Time}_{\text{queue}} + \text{Time}_{\text{server}} = 5 + 20ms = 25ms$$

Queuing theory

Basic assumptions made about problems:

- System is in equilibrium.
- Interarrival time (time between two successive requests arriving) is exponentially distributed.
- Infinite number of requests.
- Server does not need to delay between servicing requests.
- No limit to the length of the queue and queue is FIFO.
- All requests must be completed at some point.

This is called an M/G/1 queue

M = exponential arrival
G = general service distribution (i.e. not exponential)
1 = server can serve 1 request at a time

It turns out this is a good model for computer science because many *arrival* processes turn out to be **exponential**.

Service times, however, may follow any of a number of distributions.

Disk Performance Benchmarks

We use these formulas to *predict* the performance of storage subsystems.

We also need to measure the performance of real systems to:

- Collect the values of parameters needed for prediction.
- To determine if the queuing theory assumptions hold (e.g., to determine if the queueing distribution model used is valid).

Benchmarks:

- *Transaction processing*

  The purpose of these benchmarks is to determine how many small (and usually random) requests a system can satisfy in a given period of time.

  This means the benchmark stresses **I/O rate** (number of disk accesses per second) rather than **data rate** (bytes of data per second).

  Banks, airlines, and other large customer service organizations are most interested in these systems, as they allow simultaneous updates to little pieces of data from many terminals.

Disk Performance Benchmarks

- *TPC-A and TPC-B*

  These are benchmarks designed by the people who do transaction processing.

  They measure a system's ability to do random updates to small pieces of data on disk.

  As the number of transactions is increased, so must the *number of requesters* and the *size of the account file*.

  These restrictions are imposed to ensure that the benchmark really measures disk I/O.

  They prevent vendors from adding more main memory as a database cache, artificially inflating TPS rates.

- *SPEC system-level file server (SFS)*

  This benchmark was designed to evaluate systems running Sun Microsystems network file service, NFS.

Disk Performance Benchmarks

- *SPEC system-level file server (SFS)*

  It was synthesized based on measurements of NFS systems to provide a reasonable mix of reads, writes and file operations.

  Similar to TPC-B, SFS **scales** the size of the file system according to the reported *throughput*, i.e.,

  It requires that for every 100 NFS operations per second, the size of the disk must be increased by 1 GB.

  It also limits average *response time* to 50ms.

- *Self-scaling I/O*

  This method of I/O benchmarking uses a program that **automatically scales several** parameters that govern performance.

  - Number of unique bytes touched.

    This parameter governs the total size of the data set.

    By making the value large, the effects of a cache can be counteracted.

Disk Performance Benchmarks

- *Self-scaling I/O*
  - Percentage of reads.
  - Average I/O request size.

    This is scalable since some systems may work better with large requests, and some with small.

  - Percentage of sequential requests.

    The percentage of requests that sequentially follow (address-wise) the prior request.

    As with request size, some systems are better at sequential and some are better at random requests.

  - Number of processes.

    This is varied to control concurrent requests, e.g., the number of tasks simultaneously issuing I/O requests.

Disk Performance Benchmarks

- *Self-scaling I/O*

  The benchmark first chooses a nominal value for each of the five parameters (based on the system's performance).

  It then varies each parameter in turn while holding the others at their nominal value.

  Performance can thus be graphed using any of five axes to show the effects of changing parameters on a system's performance.

Reliability, Availability and RAID

Some definitions:

- *Reliability*

  Refers to the dependability of individual components of a system.

- *Availability*

  Is the system still available to the user after a failure of one or more of its components ?

Adding hardware can therefore improve *availability* but it can **NOT** improve *reliability.*

Disk arrays:

The basic idea behind **disk arrays** is that by adding disks and therefore more disk arms working in parallel, *bandwidth* is improved.

Individual process-request seek latencies can be overlapped in time.

This is cost effective since price/megabyte is independent of disk size.

However, *latency* for small requests is not improved because it still takes all of the usual latency to get to a randomly selected block.

Reliability, Availability and RAID

- *Striping*

  In disk arrays, the data from files can be **striped** across several disks.

  This *increases bandwidth* by allowing a file to be read from *more than one* disk at a time.

  The data is distributed round robin between the disks.

Problems with disk arrays:

- *Reliability*

  Disks have a *mean time to failure* of about 20 years.

  However, a collection of 8 disks will experience a failure (on average) every 2.5 years.

  N devices generally have 1/N the reliability of a single device.

Reliability, Availability and RAID

Problems with disk arrays:

- *Reliability*

  Increase the number to 1000 disks, and a failure occurs every 1/50th of a year or every week !

  When a disk fails, it takes its data with it.

- *Availability*

  The other problem with disk arrays is that the disk array becomes unusable after a single failure.

We need a scheme to prevent data loss when a disk fails, and to allow the system to recover from the failure (remain available.)

Reliability, Availability and RAID

The solution: **Redundant Arrays of Inexpensive Disks.**

Improves availability by adding *redundant* disks.

When a disk fails, the lost information can be reconstructed from redundant information.

This works since the *mean time to failure* (MTTF) of disks is long (years) and the *mean time to repair* (MTTR) is short (hours).

The idea behind RAID is to use some of the disks to store error correction information for the rest of the disks.

RAIDs do NOT have to detect errors.

The ECC kept on each sector by each disk allows the disk electronics to check and detect disk failures.

RAID Levels

There are 7 levels of RAID, each of which can be characterized by their availability and overhead.

Raid Level		Failures survived	Data Disks	Check Disks
0	Nonredundant	0	8	0
1	Mirrored	1	8	8
2	Memory-style ECC	1	8	4
3	Bit-interleaved parity	1	8	1
4	Block-interleaved parity	1	8	1
5	Block-interleaved distributed parity	1	8	1
6	P+Q redundancy	2	8	2

- *RAID 0*

This is no redundancy at all.

However, it is the fastest and cheapest RAID level.

This refers to the nonredundant disk array discussed previously.

RAID Levels

- *RAID 1*

Disks in this configuration are mirrored or copied to another disk.

With this arrangement, the data on a failed disk can be easily replaced by reading it from the other disk.

In addition, **reading** is actually *faster* than it is for RAID 0 because read requests to the same disk can be split between the two disks.

Writing is a little *slower*, though, because the file system must wait for the slower of the two requests.

Remember, both disks must be updated.

Also, seek time is different between the two disks since they are not synchronized.

The main problem with RAID 1 is that it imposes a 50% space penalty.

Therefore, it is the most expensive solution.

RAID Levels

- *RAID 3 (Bit interleaved parity)*

In this scheme, data is striped across disks in *very small units*.

These units are so small that all disks must work together on both reads and writes because a **single sector** actually spans **all** of the disks.

Redundancy is implemented by *calculating parity* and storing it on the check disk.

The overhead for **n** data disks is **1** disk, for a storage efficiency of **n/(n+1)**.

This level can survive a single disk failure and reconstruct data using the remaining disks.

This RAID level is somewhat limited since the entire disk system can only handle one request at a time.

Thus, the sustainable request rate is no higher than that of a single disk.

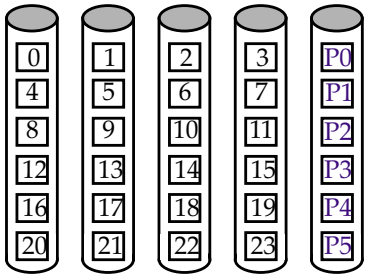
RAID Levels

- *RAID 4 (Block interleaved parity)*

This is similar to RAID 3, but individual disks each have a **block** (or more) of consecutive data within a stripe.

This means that *each disk* can handle an individual small read request if all disks are working.

However, writes are still a problem since the *parity disk* must participate in all write operations, which creates a bottleneck.



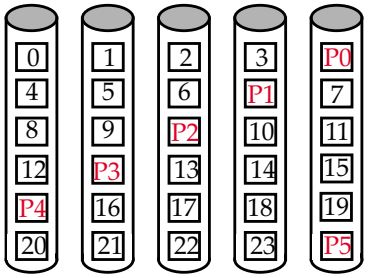
RAID 4

RAID Levels

- *RAID 5 (Rotated parity)*

The biggest problem with RAID 4 is that the parity disk is a bottleneck.

In RAID 5, the parity information may be stored on any disk.



RAID 5

Advantages:

Reads can be spread among all n+1 disks.

For writes, the parity disk bottleneck is eliminated.

RAID Levels

- *RAID 5 (Rotated parity)*

With respect to the parity calculation and update, RAID 5 does have a problem writing small pieces of data.

If a full stripe is written, the parity calculation is simple to implement.

XOR all the data together to get the parity.

If only a **single block** is written, though, the system must somehow figure out the new parity.

The most straightforward solution is to read the rest of the blocks in the stripe and XOR them together.

A better solution involves only four accesses:

- Read the old data.
- Compare old data to new data (to determine which bits change).
- Read and change the old parity.
- Write the new data and new parity.

RAID Levels

- *RAID 6 (P+Q parity)*

The objective of RAID 6 is to survive two disk failures.

This is done using schemes such that only **n** out of **n+2** disks are necessary to reconstruct the data.

This RAID level is not common today because disk failures are still relatively rare.

If a disk in a parity-based RAID fails, no data is lost if a new disk is installed and "updated" before another crash takes place.

Since stripes tend to be relatively small (usually less than 32 disks), the chance that another disk will fail is relatively low.

Most people don't worry about it.

RAID Issues:

- *Mapping data to disks.*

The first problem in a RAID system is how to map data to disks.

Usually, this is done using simple modulo arithmetic.

It is more complex for RAID 5, but there is still a formula that allows you to determine where a given block is located.

- *Reconstruction*

After a disk has failed, it is replaced.

But we are not done since the new disk must be reconstructed with the lost information in order to allow the array to withstand another crash.

Since the lost information is "embedded" in the remaining disks, reconstruction is possible and carried out immediately.

Designing an I/O system: Basics

Now that we've seen how to *estimate* performance on an I/O system.
And how to actually *measure* performance.
We are ready to talk about how to build one.

The objective is to find a design that is expandable and that meets goals for **cost** and **variety of devices** while *avoiding bottlenecks* to I/O performance.

In designing an I/O system, analyze *performance*, *cost* and *capacity* using *various I/O connection schemes* and *different numbers* of I/O devices of each type.

Here are the steps to follow in designing an I/O system:

- *List the types of I/O devices and buses, and their costs.*

- *List the physical requirements of each device.*
  These include volume, power, connectors, bus slots, etc.
  This won't be a problem for paper examples, but it certainly will be for real systems.

Designing an I/O system: Basics

- *Figure out the CPU resource demands for each I/O device.*
  Clock cycles to initiate, support the operation of, and complete requests.
  Clock stalls from I/O access to memory.
  Clock cycles to recover from an I/O activity, such as a cache flush.

- *List memory and I/O bus resource demands for each device.*
  Bandwidth of main memory and the I/O bus can often be a bottleneck, particularly when many devices are connected to a single bus.

- *Compute performance for various configurations of devices, buses, etc.*
  This can really only be done by building the system and measuring it.

  If this is not feasible (which is usually the case), the next best thing is a detailed simulation.

  Queuing models can be used to get a rough estimate of performance.

Designing an I/O system: Basics

Remember, performance can be measured as:

- Megabytes per second.
- I/Os per second.

This is dependent on the needs of the applications.

The goals for the design should also be clear:

- Is it a design to maximize performance at any cost ?
- Is it the cheapest system that will satisfy minimum requirements ?
- Is it the best price/performance ?

Look over the examples in the text !

Tertiary storage

Many modern computer systems now use **removable media** to store their data.

Advantages:

- *Inexpensive*
  Removable media cost a lot less because you only pay once for the machinery to read, write, and transport the medium.
  Since removable media use similar technology to non-removable media, the media costs are similar but the mechanism cost is much lower.

- *Low power*
  Disks use power to rotate.
  A major advantage of removable media is that they do not consume power.

- *Unerasability*
  Some removable media (WORM) can not be erased even if the system requests it.

Tertiary storage

Components:

- *Tape robots*

These systems typically hold thousands of tape cartridges and can load any cartridge in under 20 seconds.

IBM 3490 cartridges (the most common today) hold 9 GB of data per cartridge and transfer at 9 MB/sec.

- *Optical disk jukeboxes*

These are usually used for two purposes:

Small randomly-accessed data.

Data that should not ever be overwritten (even accidentally).

The cost per GB is higher than for tape, but seek time is much better.

Tertiary storage

How it works:

- *Moving data from disk to tertiary*

Data is moved from disk to tertiary storage when the disk gets full.

This is called *file migration* or simply *migration*.

Migrating data is done to free up disk space.

Files are picked for migration according to several factors:

- How big they are.
- When they were last used.
- Cost to retrieve the file.
- Other factors (possibly file type and/or user).

The device to which the file is migrated can also depend on these factors.

Small files might go to optical disk while large files are sent to a tape robot.

Migration can also occur from one tertiary storage device to another.

Tertiary storage

- *Moving data from tertiary to disk*

Data is moved from tertiary to disk when it is needed.

It may also be prefetched if the system believes that the file might be used soon.

File migration issues

Tertiary storage is very much an *ad hoc* art these days.

System designers build their systems based on what others have done because there is relatively little concrete research on what works and what does not.

Tertiary storage

File migration issues

- *When should a file be migrated ?*

How should the system choose the files to move from disk to tape ?

This is very important because a user notices even a single miss.

It may take close to a minute to retrieve a file from tape !

Migrating just one file that should not have been moved can adversely impact a user's session.

- *When should a file be deleted from disk ?*

Just because a file has been migrated to tape does not mean it should be deleted from disk.

When should its space be reclaimed ?

Tertiary storage

File migration issues

- *When should a file be moved from tape to disk ?*

  For demand fetches, this is obvious.

  However, there is also **prefetching** and **clustering** that might help improve performance.

- *What kinds of devices and layouts work best for various kinds of files ?*

  Again, clustering is important, as is transfer time versus time to first byte.

Fallacies and Pitfalls

- *Media cost is not the same as actual cost.*

  Just because a disk costs \$0.20/MB does **NOT** mean you can pay \$200,000 and get a terabyte of disk.

  There are lots of other costs associated with I/O systems.

  For example, disks need controllers, I/O buses, system buses, power supplies, and mounting hardware.

  This supporting structure becomes much more expensive as the number of disks supported grows.

  The same is true for removable media.

  It only costs \$1000 to buy a terabyte of magnetic tape.

  But that does not include tape readers, a robot, software, and all the other pieces necessary to build a full system.

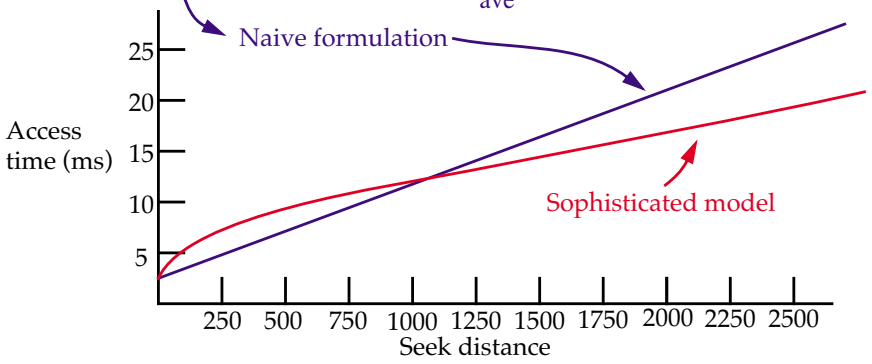
Fallacies and Pitfalls

- *Disk seek time is not linear.*

  A disk head must accelerate to maximum velocity, travel across tracks, decelerate, and settle.

  Most of the head's time is spent accelerating and decelerating, a non-linear activity.

$$\text{Time}_{seek} = \text{Time}_{min} + \frac{\text{Distance}}{\text{Distance}_{ave}} \times (\text{Time}_{ave} - \text{Time}_{min})$$



Fallacies and Pitfalls

- *Disk seek time is not linear.*

  For disks with more than 200 cylinders, Chen and Lee [1995] modeled the seek distance as:

$$\text{Seek time (distance)} = a \times \sqrt{\text{Distance} - 1} + b \times (\text{Distance} - 1) + c$$

  where a, b and c were computed as:

$$a = \frac{-10 \times \text{Time}_{min} + 15 \times \text{Time}_{ave} - 5 \times \text{Time}_{max}}{3 \times \sqrt{\text{Number of cylinders}}}$$

$$b = \frac{7 \times \text{Time}_{min} - 15 \times \text{Time}_{ave} + 8 \times \text{Time}_{max}}{3 \times \text{Number of cylinders}}$$

$$c = \text{Time}_{min}$$

  The curve represented by this model in shown on the previous slide in red.

Fallacies and Pitfalls

- *I/O will become more important with time.*

  As our society stores more and more information on computer media, the ability to get to that information will become ever more important.

  The NASA Mission to Project Earth will capture more than a terabyte of data per day from satellites.

  How can we find a needle in that haystack ?

  Similarly, future libraries may dispense with physical books and instead keep information online.

  This makes it easier to distribute the information, but getting data to and from storage will be a bottleneck unless progress is made.

Multiprocessors

- Are we reaching performance limits in uniprocessors?
- Performance enhancements are realized thru improvements in:
  - Architecture
  - Technology
- Panel session at VTS'99: "The end of Moore's Law era?"
  - Three say yes (within 10 years), two say no
  - Jury is still out on this one
  - However, it is generally believed that the physics of the process, e.g. the size of an atom, will impose a hard limit
- With reference to Moore's law:
  - "All exponentials in nature eventually saturate."
  - What is the scaling factor of the x-axis?
  - Where are we today on the curve?

Why parallel machines?

- What about improvements in architecture?
  - Uniprocessor improvements reaching a point of diminishing returns!
  - Parallel machines appear to be a natural candidate as a successor to the uniprocessor
- Multiprocessors: cost effective way to improve performance
  - It is unlikely that architectural innovations can be sustained indefinitely
    - ⇒ Analogous to the physical laws that limit technology except in reference to complexity
  - Instead, connect multiple uniprocessors together
  - There has been steady progress on the major obstacle to widespread use of parallel machines => ***software***
- Focus on the mainstream of multiprocessor design
  - Machines with small to medium numbers of processors (<100)
  - Viable architectures with more than 100 CPUs are difficult to predict

Classifying parallel architectures

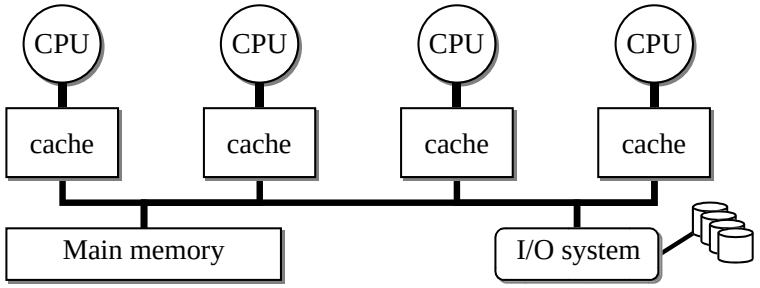
- Flynn's classification => based on parallelism in the instruction and data streams
- SISD (Single instruction & data stream): uniprocessor
- SIMD (Single instruction stream, multiple data streams)
  - Same instruction is executed by many CPUs on different data streams
  - Each processor has its own data memory
  - Only a single instruction memory and control processor which fetches and dispatches instructions
- MISD (Multiple instruction streams, single data stream)
  - No commercial versions built, but perhaps systolic processors?
- MIMD (Multiple instruction streams, multiple data streams)
  - Each CPU fetches its own instructions and operates on its own data
  - Often built using off-the-shelf microprocessors

Classifying parallel architectures

- SIMD model was popular through the 80s
  - Examples include the MasPar and Connection Machine (Thinking Machines)
  - However, less popular today => too expensive to develop
- MIMD model has clearly emerged as the architecture of choice in recent years
  - MIMD offers flexibility
  - Can operate as a single-user machine providing high performance for one application
  - Can operate as multiprogrammed machines running many tasks simultaneously
- MIMDs can build on the cost/performance advantages of off-the-shelf microprocessors & systems

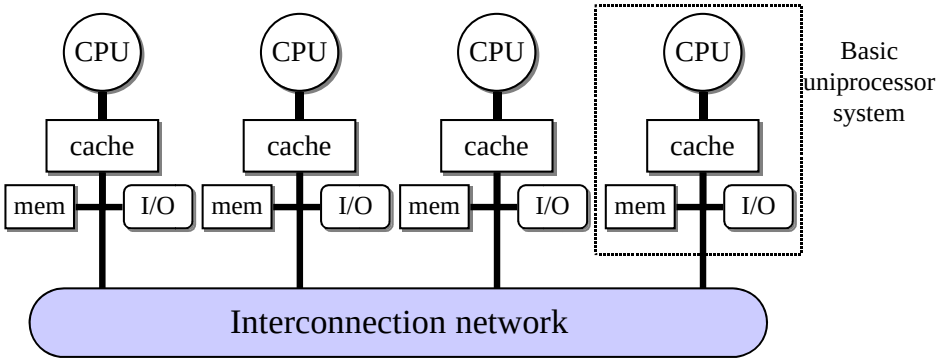
Centralized shared memory architecture

- Two basic types of MIMD architectures:
  - Centralized shared memory (*Uniform Memory Access*, or *UMA*)
  - Distributed shared memory (DSM)
- Centralized shared memory
  - At most a few dozen processors which share a bus and a single main memory
  - Large caches allow the bus and memory organization to satisfy the memory demands of a small number of processors



Distributed memory

- Supports larger processor counts by distributing the memory and allowing multiple memories to work in parallel
- Increases in processor bandwidth requirements => distributed memory beats out centralized shared memory for smaller groups of processors



Distributed vs. centralized shared memory

- Distributing the memory among the nodes has two major advantages
  - It's a cost-effective way to scale the memory bandwidth if most accesses are to local memory in the node
  - It reduces the latency for accesses to local memory, due to less contention
- Distributed memory has some disadvantages as well
  - Communicating data between processors becomes more complex
  - Interprocessor communication has higher latency
- Key characteristics that distinguish among distributed memory machines are
  - How communication is performed.
  - The architecture of the distributed memory

Distributed memory architecture models

- Physically separate memories can be addressed as one logically shared address space
  - The address space is shared—all processors see the same address space
  - These machines are referred to as *NUMA* (*Non-Uniform Memory Access*) in contrast to the centralized UMA machines
- Multicomputer architecture
  - Multiple private address spaces that are logically disjoint and cannot be addressed by a remote processor
  - An associated communication mechanism used for exchanging data
- For DSM, shared memory can be used to communicate data via load and store operations
- For a multicomputer, communication is done by either synchronous (RPC) or asynchronous message passing

Measuring communication bandwidth

- Three performance metrics are critical in any communication mechanism
  - Communication bandwidth
  - Communication latency
  - Communication latency hiding
- Communication bandwidth
  - Bisection bandwidth is the bandwidth across the "narrowest" part of the interconnection network
  - Bandwidth in and out of an individual processor is also important
  - Bandwidth is affected by the architecture within the node and by the communication mechanism
    - When communication occurs, resources are tied up or occupied
    - This can prevent other communication
  - *Occupancy* can limit communication bandwidth

Measuring communication latency

- Lower communication latency is better (of course)
- Communication latency =
  Sender overhead + Time of flight + Transport latency + Receiver overhead
  - Time of flight is preset
  - Transport latency is determined by interconnection network
  - Sender and receiver overhead are determined by communication mechanism
- Complex mechanisms (i.e. for naming and protection) increase latency, particularly those that require the OS
- Latency affects performance either by
  - Causing the processor to wait
  - Tying up processor resources

Hiding communication latency

- How well can the mechanism hide latency by overlapping communication with computation or with other communication?
  - For example, a system that only allows access to a word at a time may have low latency
    - However, it may be unable to hide the latency because each word transferred is treated as a cache miss
  - Another machine may have a higher latency but allow the processor to do other things while waiting for data
- Examples of latency hiding techniques for shared memory will be given later
- Latency hiding is more difficult to measure than the previous two and is application dependent

Performance metrics for communication

- These performance metrics are affected by
  - The size of the data items being communicated by the application
    ⇒ Size affects the latency and bandwidth in a direct way
  - The effectiveness of the different latency hiding techniques
  - The regularity in the communication patterns.
    ⇒ These two affect the cost of naming and protection (communication overhead)
- An ideal mechanism would perform well with
  - Large and small data requests.
  - Regular and irregular communication patterns

DSM vs. message passing

- Shared-memory advantages
  - Compatibility with well-understood mechanisms in centralized SM
  - Ease of programming, particularly for systems in which communication patterns are complex or vary dynamically during execution
  - Low overhead for communication: hardware used to enforce protection
  - The ability to use hardware-controlled caching => reduces the frequency of remote communication
- Message-passing advantages:
  - Simpler hardware (especially with respect to building coherent caches)
  - Explicit communication forces programmers and compiler writers to pay attention to what is costly and what is not: is this an advantage?
- Shared-memory communication is more popular today
- Centralized schemes still dominate
  - However, long-term trends favor distributed memory

Challenges of parallel processing

- Amdahl's law applies to parallel processing as well
  - Any program has a parallel portion and a serial portion.
  - The parallel portion is the only part that is sped up by having multiple processors
  - As with uniprocessors, speedup is limited by the fraction of the original program that can be parallelized
- For example, suppose we want to achieve a speedup of **80** with **100** processors
  - What is the fraction of the original computation that can be sequential?
  - With simplifying assumptions (see text):

$$80 = \frac{1}{\frac{Fraction_{parallel}}{100} + (1 - Fraction_{parallel})}$$

$$0.8 \times Fraction_{parallel} + 80 \times (1 - Fraction_{parallel}) = 1$$

$$Fraction_{parallel} = 0.9975$$

Challenges of parallel processing

- The second major challenge involves the large latency of remote memory access
  - This may cost anywhere from 50 clocks to 10,000 clocks!
- The latency is dependent on
  - The communication mechanism
  - The type of interconnection network
  - The scale of the machine
- Insufficient parallelism can be attacked in software with new algorithms that have better parallel performance
- Long communication latency can be attacked by
  - Architecture (caching)
  - Programmer (restructuring the data)
- Focus on techniques for reducing the impact of high latency

Caching in CSM architectures

- The use of large multilevel caches can substantially reduce memory bandwidth demands of a processor
  - This has made it possible for several CPUs to share the same memory through a shared bus
- Caching supports both private and shared data
  - For private data, once cached, its treatment is identical to that of a uniprocessor.
  - For shared data, the shared value may be replicated in many caches
- Replication has several advantages:
  - Reduced latency and memory bandwidth requirements
  - Reduced contention for data items that are read by multiple processors simultaneously
- However, it also introduces a problem: *cache coherence*

Cache coherence

- With multiple caches, one CPU can modify memory at locations that other CPUs have cached
- For example:
  - CPU A reads location *x*, getting the value *N*
  - Later, CPU B reads the same location, getting the value *N*
  - Next, CPU A writes location *x* with the value *N* - 1
  - At this point, any reads from CPU B will get the value *N*, while reads from CPU A will get the value *N* - 1
- This problem occurs both with write-through caches and (more seriously) with write-back caches
- Cache coherence (an informal definition): a memory system is coherent if any read of a data item returns the most recently written value of that data item

Cache coherence definitions

- Coherence defines what values can be returned by a read
- A memory system is coherent if:
  - Read after write works for a single processor
    - If CPU A writes *N* to location *X*, **all** future reads of location *X* will return *N* if no other processor writes location *X* after CPU A
  - Other processors' writes eventually propagate.
    - If CPU A writes value *N* to location *X*, CPU B will eventually be able to read value *N* from location *X*
    - Once it does so, it will continue to read value *N* until location *X* is written again
- This is our intuitive notion of a coherent view of memory

Cache coherence & consistency

- Coherence: writes to a single location are serialized
  - If CPUs A and B both write to location *X*, all processors see the same order of the writes
  - This does not mean that all reads must return the same value
    - If value *N1* is written "first" to location *X*, followed closely by reads of *X* and a write of *X* with value *N2*, some reads may return *N1* and some *N2*
  - However, a processor that reads *N2* will return *N2* for all future reads
- Consistency
  - This indicates when a modification to memory is seen by other processors (i.e. will be returned by a read)
  - Clearly, this ***can't*** be "instantaneous" since it may be that the new value has not even left the processor when a read occurs
  - Issue: when is a write visible to other processors?

Cache consistency

- Consistency issue: when must a written value be seen by a reader?
  - This is defined by a memory consistency model
  - For now, assume that a write is not complete until all processors have "seen" the effect of the write
  - Also, assume that a processor may not reorder memory accesses to move reads before an outstanding write
    - Reads can be reordered, but reads and writes can not be interchanged
- Coherent caches provide both
  - Replication of shared data items (reduces latency and contention)
    - Provide multiple copies of data so that several processors can access a single piece of memory without serialization
  - Migration of data items (reduces latency)
    - Data items are moved from one processor to another as needed

Cache coherence protocols

- Small-scale multiprocessor use hardware mechanisms to track the state of data blocks that are shared
- Two types of protocols
  - Directory based
    - The sharing status of a block of physical memory is kept in one location (the directory)
    - Interprocessor communication is used to maintain coherence
  - Snooping
    - The sharing status is distributed and kept with the block in each cache
    - The caches are usually on a shared memory bus
    - The cache controllers snoop the bus to watch for transactions that occur on data blocks that they hold

Bus snooping protocol: write invalidate

- Write invalidate is the most common protocol, both for snooping and for directory schemes
- The basic ideas behind this protocol:
  - Writes to a location invalidate other caches' copies of the block
  - Reads by other processors on invalidated data cause cache misses
  - If two processors write at the same time, one wins and obtains exclusive access
- Example assumes a write-back cache

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of mem location *X*
CPU A reads *X*	Cache miss	0	-	0
CPU B reads *X*	Cache miss	0	0	0
CPU A writes 1	Invalidate	1	-	0
CPU B reads *X*	Cache miss	1	1	**1**

Bus snooping protocol: write update

- An alternative is to update all cached copies of the modified data item
  - This is called *write update* or *write broadcast*
- To reduce bandwidth requirements, this protocol keeps track of whether or not a word in the cache is shared
  - If not, no broadcast is necessary
- Example again assumes a write-back cache

Processor activity	Bus activity	Contents of CPU A's cache	Contents of CPU B's cache	Contents of mem location *X*
CPU A reads *X*	Cache miss	0	-	0
CPU B reads *X*	Cache miss	0	0	0
CPU A writes 1	Broadcast	1	1	**1**
CPU B reads *X*	Cache hit	1	1	1

Comparing bus snooping protocols

- Write invalidate is much more popular than write update
- Write update requires more system-wide notifications
  - Multiple writes to the same word with no intervening reads require multiple broadcasts
  - With multiword cache blocks, each word written requires a broadcast
  - Delay between write by one processor and read by another is lower
- Write invalidate uses fewer system-wide notifications
  - The first word written invalidates the entire block
  - Write invalidate works on blocks, while write broadcast works on individual words or bytes
  - Reading an invalidated block causes a miss (somewhat slower)
- Since bus and memory bandwidth is more important in a bus-based multiprocessor, write invalidate performs better

Implementing the write-invalidate protocol

- Write invalidate is simple in bus-based schemes
  - Acquire the bus and broadcast the address to be invalidated
  - Since all processors snoop the bus, they can check the address against items in their cache
- Bus acquisition serializes writes to a memory location
  - Writes to a shared data item cannot complete until the bus is acquired
- How is a data item located when a cache miss occurs?
  - For write-through, it's in memory
  - For write-back, snooping can be used: if a processor finds that it has a dirty copy of the requested cache block, it provides the block instead of memory
- Write-back caches are greatly preferred in a multiprocessor environment since they reduce memory bandwidth

Writes in write-invalidate protocols

- Writes are an issue with cache coherence protocols in general
- The CPU needs to know if any other caches contain the block to be written by a processor.
  - If there are none, then the write need not be placed on the bus, reducing the time to complete the write and reduces memory bandwidth
- This can be tracked by adding an *extra state bit* (in addition to the valid and dirty bits) that indicates if the block is shared
  - If the bit is set (the block is shared), the cache generates an invalidation on the bus and marks the block as private
  - If another processor later requests the block, the miss is snooped and the "owner" sets the state bit to shared

Optimizations for tag checking

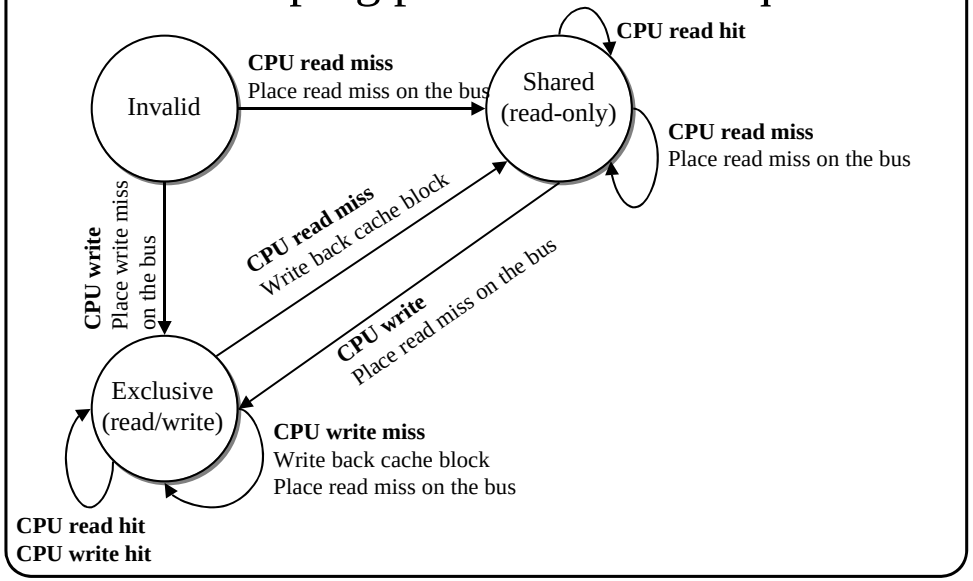
- Note that every bus transaction checks cache-address tags — this could potentially interfere with CPU cache access
- Reduce interference by
- Duplicate the tags: bus access proceeds in parallel with CPU
  - On misses, the processor arbitrates for and updates both sets of tags
  - Snoop also does this to perform an invalidate or to update the shared bit
  - However, a snoop may require fetching a block, thus stalling
- Employing a multilevel cache with inclusion
  - Snooping is directed to L2, where there are fewer processor accesses
  - If a snoop gets a hit, then it must arbitrate for L1 to update state and possibly retrieve data, usually stalling the processor
  - Since it is popular to use multi-level caches in multiprocessors (to reduce memory bandwidth), this solution is usually adopted

Sample bus snooping protocol

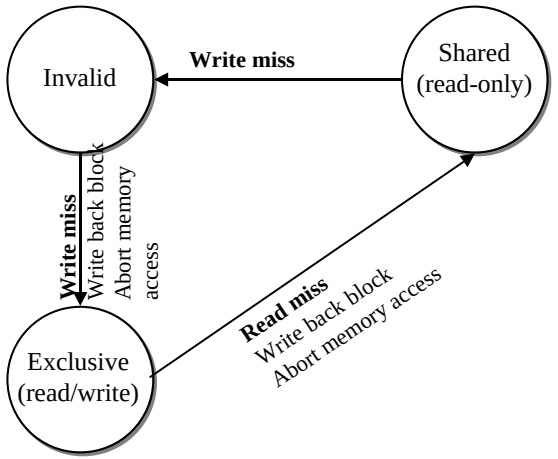
- Implemented by incorporating a finite state controller in each node
  - The controller responds to requests from the processor and bus
- To simplify the controller, write hits and write misses to shared blocks are treated as write misses
  - This causes processors with copies to invalidate them

Request	Source	Function
Read hit	Processor	Read data in cache
Write hit	Processor	Write data in cache
Read miss	Bus	Request data from cache or memory
Write miss	Bus	Request data from cache or memory, and perform any needed invalidates

Bus snooping protocol: CPU requests



Bus snooping protocol: bus requests



Snooping protocols: wrapping up

- Protocol assumes that operations are atomic
  - In reality, a write miss is not atomic — just too much work to do
  - Also, read misses on a split transaction bus are not atomic
  - Nonatomic actions introduce the possibility of deadlock...
- Real protocols distinguish between write hits and write misses
  - From the shared state, a write miss would require the action shown previously
  - However, a write hit does not require that the data be fetched since it is up-to-date — all that's needed is an invalidate operation
- Real protocols distinguish between shared and clean data in exactly one cache
  - A "clean and private" state eliminates the need to generate a bus transaction on a write to a "clean and private" block

Distributed shared memory protocols

- Distributed shared-memory machines need cache coherence for the same reasons as centralized shared-memory machines
  - Centralized protocols have drawbacks in these architectures due to the interconnection network and scalability requirements
- DSM might not use hardware mechanisms!
  - Instead, focus on scalability (Cray T3D)
  - In this scheme, only data that actually resides in the private memory may be cached (shared data is marked uncacheable)
  - Coherence is maintained by **software** => several disadvantages
  - Compiler mechanisms are very limited
    - They have to be very conservative, e.g. treat a block on another processor as dirty even though it may not be
    - This results in excessive coherence overhead (extra fetching)

Issues with DSM coherence protocols

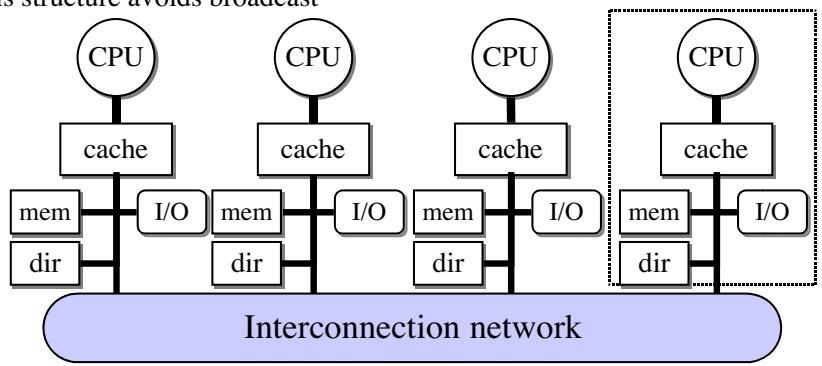
- More disadvantages of software implemented coherence
  - Multiple words in a block provide no advantage
    - Software coherence must be run each time a word is needed
    - The advantage of spatial locality (the "prefetch" of other words in the block) is lost in single word fetching
  - Latencies to remote memory are relatively high.
    - Remote references can take 50 - 1000 CPU cycles, making coherency "misses" a very costly proposition
- Snooping isn't feasible for DSM
  - Snooping coherence schemes aren't scalable => problem for DSMs
  - The distributed nature of the snooping protocol's data structure (which maintains the state of the cache blocks) does ***NOT*** scale well
  - Snooping requires broadcast (communication with all caches on every miss) => very expensive with an interconnection network

Directory-based coherence

- Alternative to snooping: directories, which hold:
  - The state of every block in memory (shared, uncached or exclusive)
    - Exclusive => the block has been written, is in one cache and memory is out-of-date
    - This information is also keep in the cache for efficiency reasons
  - Which caches have copies
    - May be implemented using a bit vector for each block with the processors identified by the bit's position
  - Whether or not the block is dirty
- The amount of information in the directory is proportional to number of processors * number of memory blocks
  - ⇒ This works O.K. for less than 100 processors -- other solutions are needed for >100 processors

Directory location

- Directory entries may be distributed along with the memory
  - High-order bits of an address can be used to find the location of a particular block of memory
  - Directory and data for a block at the same place
- This structure avoids broadcast



Directory-based cache coherence protocols

- Handle two basic primitives
  - Read miss
  - Write to a shared, clean cache block
  - Awrite miss is a combination of these two
- Simplifying assumptions still hold here
  - Writes to non-exclusive data generate write misses
  - Write misses are atomic (processors block until the access completes)
- This introduces two complications
  - There is no longer a bus => no single point of arbitration
  - Broadcast is to be avoided => the directory and cache must issue explicit response messages, e.g., *invalidate* and *write-back* request messages

Directory protocol: message types

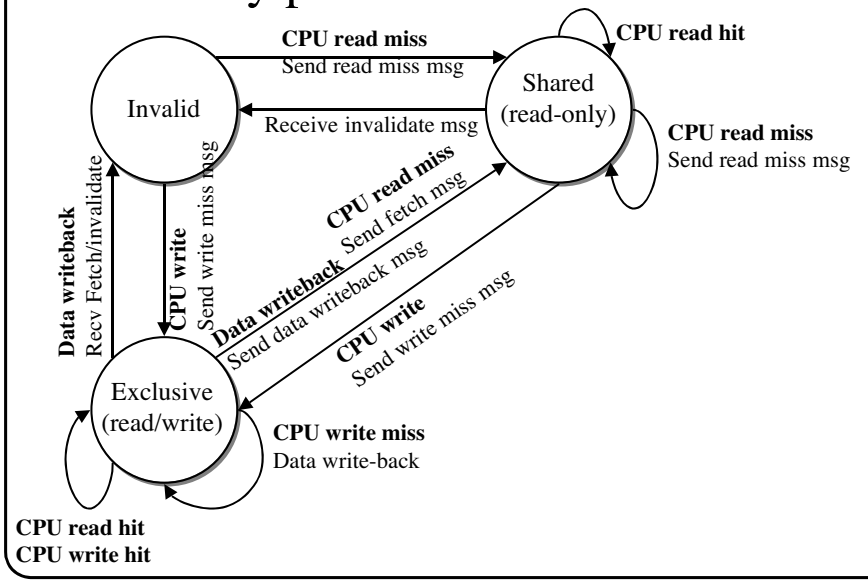
- States and transitions at each cache are identical to the snooping protocol
- Actions are somewhat different, however
- Message types for directory-based protocols
  - Local node: Where the request originates
  - Home node: Where memory and directory live
  - Remote node: Node that has a copy of the block (exclusive or shared)

Message type	Source	Destination	Contents	Function
Read miss	Local	Home	P, A	P has a read miss at addr A Request data & make P a read sharer
Write miss	Local	Home	P, A	P has a write miss at addr A Request data & make P exclusive owner
Invalidate	Home	Remote	A	Invalidate a shared copy of data at addr A

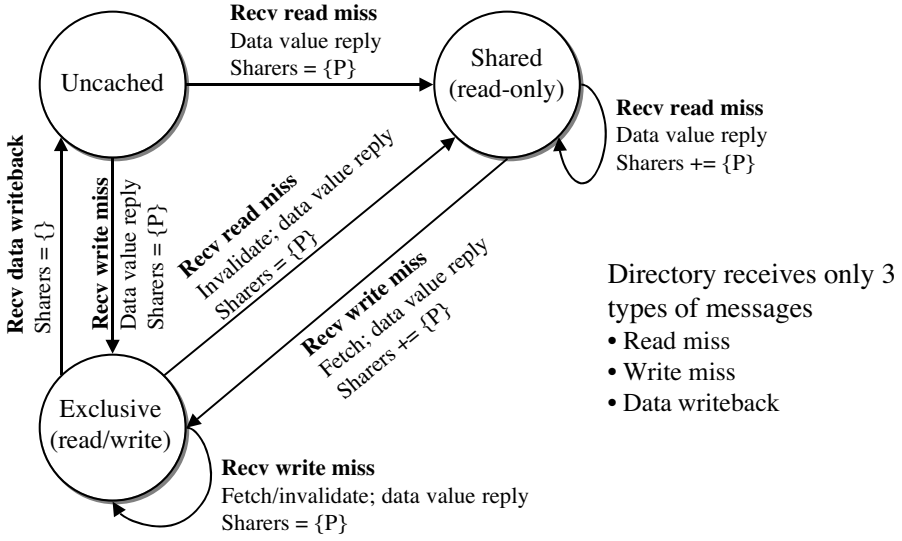
Directory protocol: more message types

Message type	Source	Destination	Contents	Function
Fetch	Home	Remote	A	Fetch block at addr A & send to home Change the state of A in remote to shared
Fetch/invalidate	Home	Remote	A	Fetch block at addr A & send to home Invalidate the block in the cache
Data value reply	Home	Local	Data	Return a data value from the home memory
Data writeback	Remote	Home	A, Data	Write back a data value for addr A

Directory protocol: cache state machine



Directory protocol: directory state machine



Directory receives only 3 types of messages
- Read miss
- Write miss
- Data writeback

Uncached & shared states

- Uncached state
  - When the block is uncached, the directory can only receive two kinds of messages: read miss and write miss
    - A read miss moves the block into the shared state
    - A write miss moves it into exclusive
  - In either case, the directory updates its list of sharing nodes to include only the node that requested the data
- Shared state
  - Again, only read or write misses are possible, since all caches have the same value as memory
  - Read miss => node requesting data is added to the list of sharing nodes
  - Write miss => block is moved to the exclusive state
    - Invalidate messages are sent to all current sharing nodes.
    - Sharing list is updated to only the requesting processor

Exclusive state

- Read miss
  - Owner is sent a fetch message telling it to write data back to memory
  - Requesting node is added to the sharing list
  - Block is marked as shared
- Write miss: block must be written back by the current owner
  - Directory sends out a fetch message
  - When the data is written, the directory forwards it to the new owner and replaces the old owner with the new owner in the sharing list
- Write-back: the data is updated in memory and the block goes into the uncached state and the sharing list is cleared
- Optimization: have the old owner send the data directly to the new owner on a write miss
  - May be done either instead of or in addition to writing the data to home

Directory protocol issues

- What happens when read-only data is replaced?
  - This scheme does not explicitly notify the directory when a clean block is replaced in the cache
  - This is OK => the cache will simply ignore invalidate messages for blocks that are not currently cached
  - Potential problem: the directory may send a few unnecessary messages
    - Probably not as bad as having the remote caches send a message each time they replace a block
- Synchronization
  - Deciding the order of accesses in a distributed memory system is harder
  - Without a shared bus, it's impossible to tell which **writes** come first
    - It's not feasible to stall all accesses until a write completes
  - Often, this can be handled by requiring all writes to be atomic
    - However, doing so slows down the system greatly

Synchronization in parallel processors

- Processors within a parallel system must have some method of synchronization
  - Software routines are usually built on top of hardware-supplied synchronization instructions
- For shared-memory machines, the key element is an uninterruptible instruction that can atomically retrieve and change a value
  - Atomic exchange: exchanges register and memory location atomically
  - Test-and-set
  - Implementation is challenging since it requires both a memory read and write to execute atomically
  - This complicates coherence and does not scale well

Load-linked & store conditional

- Another option: use a pair of instructions:

```
try: mov  R3,R4    ; move exchange value into R3
     ll   R2,0(R1) ; load the value at 0(R1) into R2
     sc   R3,0(R1) ; store the value R3 and set R3
     beqz R3,try   ; branch if value set is changed to 0
     mov  R4,R2    ; put load value into R4
```

- Store conditional (*sc*) fails if
  - Memory location specified by the *load linked* instruction is changed before the *store conditional* instruction (to the same address)
  - If it fails, the sequence is executed again
- *ll* is implemented by storing the address given in the instruction in a link register
  - If an interrupt occurs or if the cache block containing the address is invalidated, the *ll* register is set to 0 and *sc* fails

Implementing locks using coherence

- How can locks be cached in machines with cache coherence?
  - Spin-lock operation is performed on a local cached copy
    - This reduces memory bandwidth
  - There is often locality in lock access => caching reduces time to acquire the lock
- Assume that we have an exchange instruction
  - To implement a spin-lock, use the following where 0 indicates success:

```
         lwi  R2,#1     ; load immediate #1
lockit: exch R2,0(R1)   ; exchange R2 with 0(R1)
        bnez R2,lockit  ; if 1 returned, fail
```

- Problem:
  - Each *exch* operation requires a write
  - Most writes result in a write miss => writing requires exclusive access

Implementing locks using coherence

- In the loop, read instead and write only when the lock is free

```
lockit: lw   R2,0(R1)   ; read the lock
        bnez R2,lockit  ; keep reading if lock not free
        lwi  R2,#1      ; load the lock value
        exch R2,0(R1)   ; race to exchange & get 0
        bnez R2,lockit  ; if another processor beat us, start over
```

- A load linked/store conditional version need not cause any bus traffic during the testing operation:

```
lockit: ll   R2,0(R1)   ; read the lock
        bnez R2,lockit  ; keep reading if lock not free
        lwi  R2,#1      ; load the lock value
        sc   R2,0(R1)   ; Try to store & get 0
        beqz R2,lockit  ; if another processor beat us, start over
```

  - However, when the lock is released, a lot of traffic is generated
  - This makes it difficult to scale this implementation to many processors

Barrier synchronization

- Barrier synchronization: another common synchronization operation in programs with parallel loops
  - Allows multiple processes on multiple CPUs to wait until a certain number of processes have reached a "barrier"
  - When sufficient processes arrive, all waiting processes may continue

```
lock(counterlock);          // ensure count update is atomic
if (count == 0)             // First process to barrier -- reset 'release'
  release = 0;
count += 1;                 // Count arrivals
unlock(counterlock);        // Release lock
if (count == total) {       // All have arrived, so reset
  count = 0;                // counter and release processes
  release = 1;
} else {                    // Waiting for more processes
  spin (release == 1);      // Spin until release set to 1
}
```


Barrier synchronization

- Previous method can fail if one process 'races' ahead and resets release before the last process has been scheduled again and exits the spin
- Instead, use a sense-reversing barrier
  - A fix which uses private per-process variables.
  - local_sense is initialized to 1 for each process

```
Local_sense = !local_sense;  // toggle private variable
lock(counterlock);           // ensure count update is atomic
if (count == 0)              // First process to barrier -- reset 'release'
  release = 0;
count += 1;                  // Count arrivals
unlock(counterlock);         // Release lock
if (count == total) {        // All have arrived, so reset
  count = 0;                 // counter and release processes
  release = local_sense;
} else {                     // Waiting for more processes
  spin (release == local_sense);
}
```


Memory consistency models

- Deciding when a change to memory should be propagated to other CPUs is a difficult problem
  - Message-passing machines require explicit propagation, so the decision is left entirely to software
  - Shared memory in hardware has to support hardware mechanisms for making this decision

Future of parallel processors

- This class has only scratched the surface on multiprocessors
  - There is more than enough material to spend an entire semester discussing MP hardware design
  - Predicting the future of MPs is even harder than understanding how today's MPs work
- Large scale machines that scale up naturally
  - These would be built from commodity elements that can be added in small numbers to build a a big system
  - The interconnect is proprietary, but the processor is commodity
  - The SGI Origin 2000 series and Cray T3E work this way

Future of parallel processors

- Large-scale machines built of clusters of mid-range machines
  - Require faster uniprocessors but can be built from fewer machines
  - Examples include the SGI Challenge and Sun Enterprise
- Off-the-shelf nodes with custom interconnect
  - Uses standard processor boards
  - Uses a high-speed custom-built interconnect
  - Examples include IBM SP/2
- All off-the-shelf components
  - Everything (processor, network) is standard and relatively inexpensive
  - Workstation clusters (Beowulf) are a good example of this

Putting it all together: G4 vs. K7

- CPU designers have to solve “conflicting” problems
  - Run programs as fast as possible
  - Maintain compatibility with previous versions of hardware
  - Use as few transistors as possible
    - Lower cost (=> more markets for processor?)
    - Lower power consumption
- Designers of Motorola G4 and AMD K7 took different approaches to balancing these tradeoffs
  - G4 focused on simplicity, lower cost
  - K7 focused on performance
- Examining tradeoffs can show how computer architects make decisions about how to build a CPU...

G4 vs. K7: basic differences

- Motorola G4 is relatively small and simple
  - 10 million transistors
  - 4 stage pipeline
  - Few addressing modes
  - Fixed length instructions => hardware instruction decoding
- AMD K7 is beefy and more complex
  - 22.5 million transistors
  - 12+ stage pipeline
  - Many addressing modes
  - Variable length instructions => “MacroOps” to decode them

G4 vs. K7: basic similarities

- Both use out-of-order execution
  - Reorder instructions for maximum performance
  - K7 has to work harder because of deeper pipeline
- Both use branch prediction
  - Accuracy more important for K7 because of pipeline depth
  - K7 devotes more space & effort to it
- Both use large caches
  - Caches reduce memory access time
  - Caches allow high-bandwidth access to memory
- Both have superscalar issue
- Both have vector units (though they handle vectors differently)

G4 instruction handling

- Read fixed length instruction from memory
- Decode using hardware
- Issue instruction to functional unit
  - Integer
  - Vector
  - Floating point
  - Branch unit
- Retire instructions in order

K7 instruction handling

- X86 instructions packed into predecode cache
  - Breaks byte stream into individual instructions
  - Deals with variable length instructions
- Transform x86 instructions into *MacroOps*
  - Done in hardware for simple instructions, microcode for complex ones
  - A MacroOp is composed of a register-register operation and/or a memory access instruction (called *ops*)
  - Decoding process takes 3 pipeline stages!
- Ops fed into execution pipeline
- Instructions retired in order

G4 vs. K7: functional units

- G4 functional units include
  - 1 FP unit
  - 2 vector units
  - 2 integer units
  - 1 address calculator
- Loads & stores
  - 6 entry scheduler
  - Limited out-of-order loads
  - 1 load or store per cycle
- FP execution speed limited
- Good vector execution speed

- K7 functional units include
  - 3 FP/vector units
  - 3 integer units
  - 3 address calculators
- Loads & stores
  - 44 entry scheduler
  - Flexible out-of-order loads
  - 2 loads per cycle
- Good FP execution speed
- Vectors somewhat slow

G4 vs. K7: math units

- G4 integer units (2)
  - One FU can do any operation
  - One FU can do only simple stuff
- G4 FP unit
  - Does all FP operations (no specialization)
  - Only allows one FP operation at a time

- K7 integer units (3)
  - Can do any operation
  - Up to 3 simultaneous operations
- K7 FP units
  - Does all FP operations (no specialization)
  - Also perform vector operations...

G4 vs. K7: vector units

- G4 vector units
  - 2 separate vector units
  - FP & vectors done by independent units
  - Vector unit operates on distinct register set
- G4 vectors are 128 bits long
- G4 faster at vectors because
  - Separate FU for vectors
  - Longer 128 bit vectors

- K7 vector units
  - Same units as FP units
  - Can't do both FP & vectors at the same time
  - Must reuse FP registers for vectors
- K7 vectors are 64 bits long
- K7 slower at vectors because
  - Shared FU with FP operations
  - Smaller vectors

G4 vs. K7: branch prediction

- G4 has relatively simple branch prediction
  - Less space required
  - May predict wrong more often
  - Penalty isn't so bad => pipeline is only four stages deep!
- Branch prediction takes the approach of improving penalties at the cost of reducing accuracy
- K7 has relatively complex branch prediction
  - Uses lots of space & transistors
  - Reduced misprediction rate
  - Penalty is high with a 10+ stage pipeline!
- Branch prediction takes the approach of improving accuracy at the cost of increasing penalties

Comparing design K7 and G4 approaches

- K7 takes the "complexity wins" approach
  - Throws transistors at the instruction decoding problem
  - Throws transistors into integer & FP functional units
  - Uses a superpipelined architecture: pipeline has relatively many stages, each of which is short
    - Clock speed can be faster
    - Hazards (data, control) cost much more
- G4 takes the "simplicity wins" approach
  - Keeps decoding simple
  - Relatively few integer & FP units, but higher utilization
  - Short pipeline resistant to hazards, but lower clock speeds
  - Considerably lower cost => broader markets

G4 vs. K7: which is better?

- So what's the bottom line?
  - Neither G4 or K7 is clearly better!
  - Each has its advantages and disadvantages
- K7 may be better for
  - FP intensive code
  - Code with relatively few (or predictable) branches
  - Systems where power & cost are less important
- G4 may be better for
  - Vector intensive code
  - Code with lots of branches and data hazards
  - Systems where power & cost matter more

What is a vector instruction?

- What is a vector instruction?
  - Vector instructions handle many data values with a single instruction
    - Registers often contain a set of values
    - All values in set treated the "same" way
  - Vector instructions may operate on
    - Integers & floating point numbers
    - Bit vectors
- Where are they used?
  - Scientific computation (numerically intensive)
  - Graphics: bit-oriented vectors
    - MMX (x86)
    - AltiVec (PowerPC)

Problems with scalar processors

- How can scalar processors be sped up?
  - Use deeper pipelines
    - In longer pipelines, pipeline latencies become an issue
  - Reduce the instruction fetch/decode rate: for a given amount of data, fetch fewer instructions
    - Make instructions more complex?
    - Make instructions operate on more values?
- Speed up scalar processors with vectors
  - One instruction operates on many values
  - Rather than fetching 64 or more instructions to perform 64 FP adds, the CPU fetches only one
  - Good for small instruction caches!

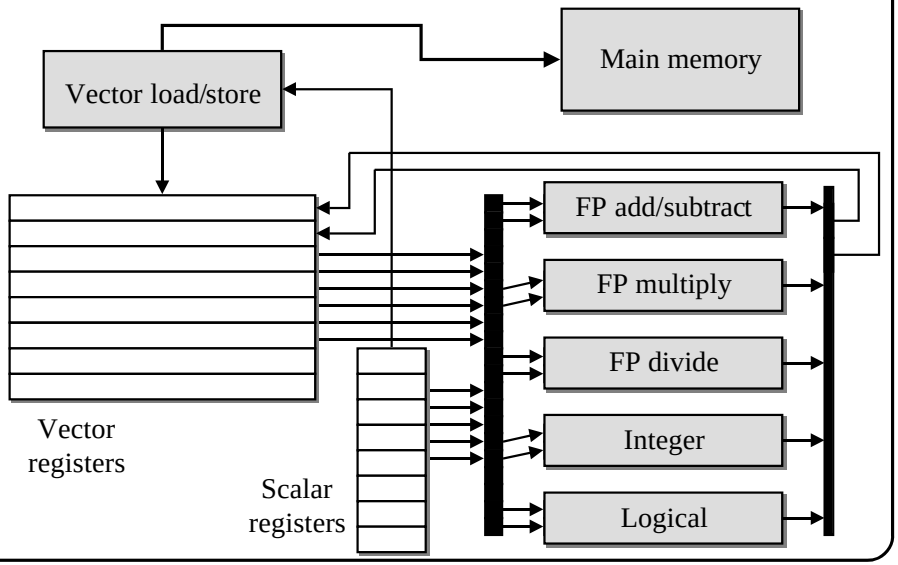
Using vectors to avoid scalar bottlenecks

- Independent computations
  - Computation on each vector element is independent of all other vector elements
  - Deep pipelines can be used without creating data hazards
  - ⇒ Compiler must generate vector instructions
- Lots of work per instruction
  - Each instruction can cause many operation
  - ⇒ Fewer instructions need to be fetched and decoded
- Reduce control hazards and loop overhead
  - A vector instruction can function as a loop in a scalar architecture
  - No branches!
    - Lower loop overhead
    - No control hazards

Using vectors to avoid scalar bottlenecks

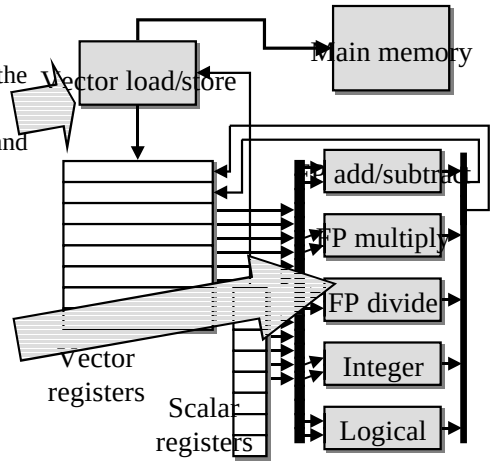
- Optimize memory access
  - The memory access pattern for an entire vector load/store is known at instruction issue
  - ⇒ CPU may be able to get all of the data by paying the latency for memory access only a few times
- Overlapping vector operations
  - Overlap vector operations if there are enough functional units
    - Keep CPU busy with useful work
    - Reduce execution time
  - Requires more hardware, but hardware provides performance improvements
- Memory-memory vs. register-memory vector architectures

Simple DLX vector architecture (DLXV)



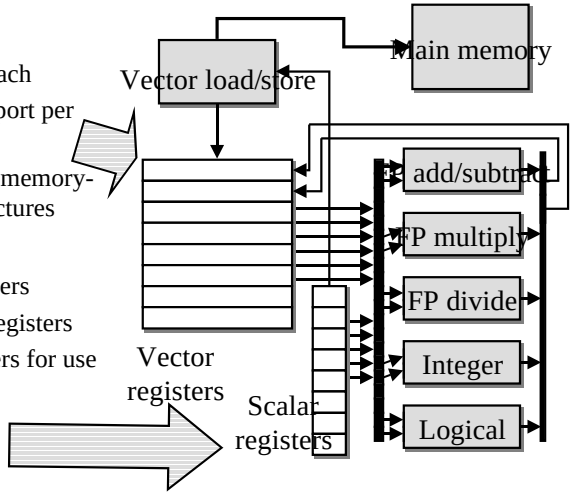
DLXV architecture

- Vector load-store unit
  - Interfaces the vector unit with the memory
  - Moves data between memory and CPU
  - Fully pipelined: one word per clock cycle after startup
- Vector functional units
  - Fully pipelined: start a new operation on every clock cycle
  - Control unit detects hazards



DLXV architecture

- Vector registers
  - 8 registers, 64 values each
  - 2 read ports & 1 write port per register (is it enough?)
  - May not be needed for memory-memory vector architectures
- Scalar registers
  - Regular DLX FP registers
  - Regular DLX integer registers
  - Special-purpose registers for use by the vector unit
    - Vector length
    - Vector-mask



Sample vectorizable code

- Implement $\vec{Y} = a\vec{X} + \vec{Y}$
  - X and Y are vectors
  - A is a scalar
  - SAXPY/DAXPY loop (S or D indicates single or double precision)
  - Very common operation in scientific codes
- Code for DLX at right
  - Interlocks between the MULTD and ADDD and the memory operations
  - Possible problems with branches
  - Total instruction count ≈600

```
for (i = 0; i < 64; i++) {
  Y[i] = a * X[i] + Y[i];
}
```

```
LD     F0,a
ADDI   R4,Rx,#512
Loop:
LD     F2,0(Rx)   ; load X[i]
MULTD  F2,F0,F2   ; a * X[i]
LD     F4,0(Ry)   ; load Y[i]
ADDD   F4,F2,F4   ; a*X[i]+Y[i]
SD     F4,0(Ry)   ; store Y[i]
ADDI   Rx,Rx,#8   ; X index++
ADDI   Ry,Ry,#8   ; Y index++
SUB    R20,R4,Rx  ; loop bound
BNEZ   R20,Loop   ; loop if not done
```

Vectorized code for DAXPY

- Original code had
  - Lots of dependencies
  - Lots of loop overhead
- Vectorized code has
  - No looping!
  - Only a few vector instructions
    - Reduced instruction bandwidth (6 instructions)
    - Fewer interlocks: encountered only at startup
  - Simpler decoding: fewer dependencies to work out

```
LD     F0,a
ADDI   R4,Rx,#512
Loop:
LD     F2,0(Rx)   ; load X[i]
MULTD  F2,F0,F2   ; a * X[i]
LD     F4,0(Ry)   ; load Y[i]
ADDD   F4,F2,F4   ; a*X[i]+Y[i]
SD     F4,0(Ry)   ; store Y[i]
ADDI   Rx,Rx,#8   ; X index++
ADDI   Ry,Ry,#8   ; Y index++
SUB    R20,R4,Rx  ; loop bound
BNEZ   R20,Loop   ; loop if not done
```

```
LD      F0,a
LV      V1,Rx     ; load X vector
MULTSV  V2,F0,V1  ; scalar-vector multiply
LV      V3,Ry     ; load Y vector
ADDV    V4,V2,V3  ; add
SV      Ry,V4     ; store result
```


Calculating vector execution time

- Terms (not "official"; made up by textbook authors)
  - Convoy: a group of vector instructions that could be issued in the "same" cycle because there are no dependencies between them
  - Chime: the time a maximum-length vector instruction takes to complete its execution
- Basic performance
  - Approximate execution time for a sequence of vector instructions is number of convoys * chime length
  - Only approximate because it ignores startup overheads

    ⇒Overheads are often short compared to instruction execution time

Vector startup latency

- Startup latency: delay before the first result comes out of the vector pipeline
  - Pipeline produces one result per cycle thereafter
  - Effect on performance reduced with longer vector lengths
  - Irrelevant if vectors infinitely long (but this isn't realistic)
- Startup latency == pipeline depth
- Sample startup latencies (for DLXV)
  - Load/store: 10 cycles
  - Multiply: 7 cycles
  - Add: 6 cycles
  - Memory often has a longer startup latency (why?)

Sample performance calculation

```
LD      F0,a
LV      V1,Rx     ; load X vector
MULTSV  V2,F0,V1  ; scalar-vector multiply
LV      V3,Ry     ; load Y vector
ADDV    V4,V2,V3  ; add
SV      Ry,V4     ; store result
```

Idealized

- Only MULTSV & second LV may be combined => 4 convoys
- Sequence requires 4 chimes
- Total time required => 64 * 4 = 256
- Requires 4 cycles/element

Realistic

- Compute actual start & finish times for each convoy

		Start	Finish
LD	F0,a	0	
LV	V1,Rx	0	9+n
MULTSV	V2,F0,V1	10+n	16+2n
LV	V3,Ry	10+n	20+2n
ADDV	V4,V2,V3	20+2n	26+3n
SV	Ry,V4	26+3n	36+4n

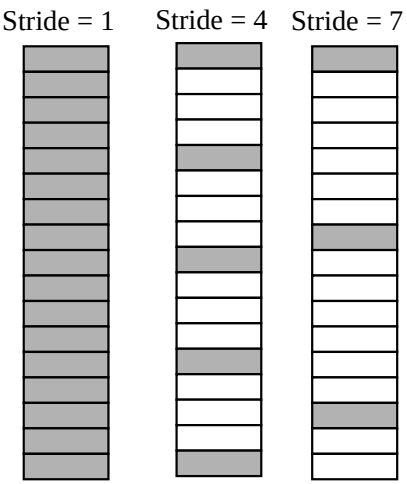
- Total time is 37+4n, or 293 cycles
- Requires 4.58 cycles/element

Vector load-store unit issues

- Load-store units may not be able to complete one result per cycle (unlike most pipelined functional units)
- Long startup latencies
  - Relatively slow memory delays the first word of data
  - Data caches don't usually help vector processors (why?)
- Avoid memory conflicts
  - Vector processors often access several banks of memory at once
  - CPU gets more than one word per memory cycle
- Non-sequential memory accesses
  - Programs often need to load a vector from non-sequential elements
  - Done using *strided* memory access

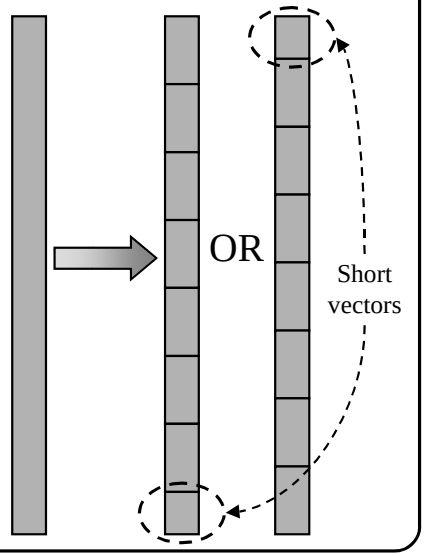
Vector stride

- Data in memory may not be sequential
  - Distance between adjacent elements in the vector is called the stride
- Strided access needed only for load and store
  - ⇒ Vectors held in a register are accessed normally
- Stride & vector address obtained from general purpose registers
- Stride and number of memory banks should be relatively prime
  - ⇒ Spreads accesses evenly for better performance



Handling odd-length vectors & strip mining

- Must handle vectors of less than the maximal length
  - Done by setting the vector length register (VLR)
- Strip mining example (500 elements)
  - Create 7 operations that run on full-length (64 element) vectors
  - Create 1 operation that runs on 52 elements
    - Shorter vector can be the first or last handled
    - Making it first simplifies the end-of-loop checks



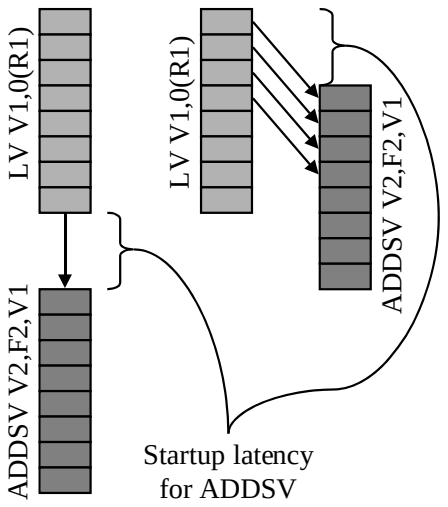
Performance for odd-size vectors

- Estimate loop performance using same method as before
  - Include time per element
  - Include vector instruction startup time
  - Include loop overhead
  - Doesn't include loop startup (paid once per execution, not once per loop)
- Compute T_{start} by adding up all of the vector startup latencies (excluding those that overlap in convoys)
- Compute T_{chime} by counting the convoys

$$T_n = \left\lceil \frac{n}{max\ vector\ length} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

Vector chaining

- Chaining ≈ forwarding for vector operations
  - Entire result from a vector operation may take 64+ cycles to produce
  - First element is ready after the startup latency
    - Could be fed into a second operation that uses the vector register as a source,
  - Time to do two chained operations is (length + startup_{op1} + startup_{op2})
- Chaining reduces overall time by overlapping vector instructions

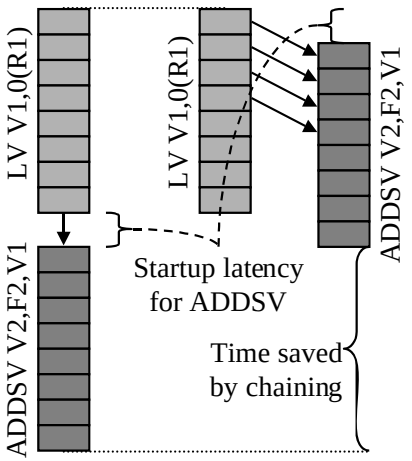


Vector chaining: example

- Example

```
LV      V1,0(Rx)
ADDSV V2,F0,V1
```

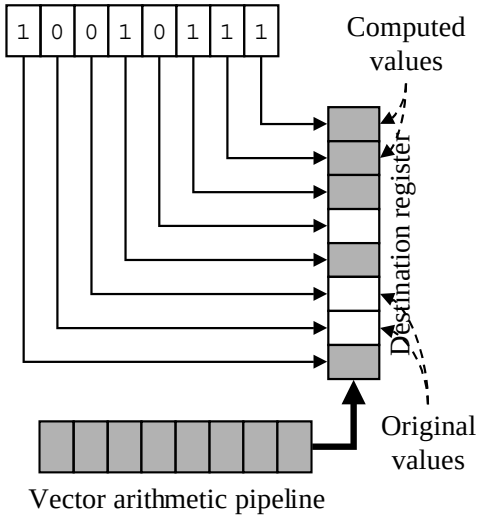
  - Without chaining, requires 10 + 64 + 6 + 64 = 146 cycles
  - With chaining, the ADDSV could start after the load produced its first element
    - Reduces total time to 10 + 6 + 64 = 82 cycles
    - Total time reduced to 56.2% of the original time
- Long chains (multiple instructions chained together) can drastically cut execution time



Vector mask control

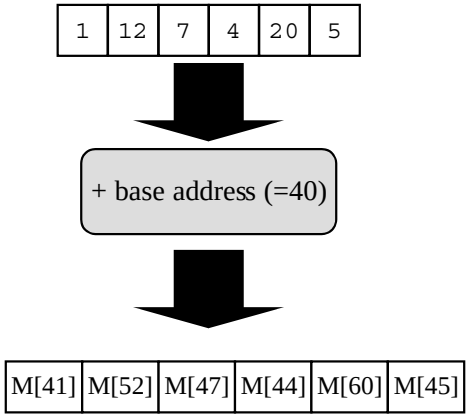
- Vector mask register: one bit for each element in a vector register
  - Bit set => perform operation
  - Bit clear => do no operation
  - Functional unit busy for a cycle regardless of bit set/clear
- Vector mask register set by
  - Copying a value from a scalar register
  - Doing a vector-based operation to set each bit individually
    - Example: SNESV

⇒ Allows the handling of conditions inside loops



Handling sparse matrices

- Sparse matrix: much of matrix is zero
  - Zero elements aren't stored
  - Non-zero elements represented as index-value pairs
  - Example: A[5] = 4 => 5,4
- Sparse matrices use *scatter-gather*
  - Memory indices stored in a vector
  - Load uses vector values as pointers or offsets
- Scatter-gather also useful when indices determined on-the-fly



Measures of vector processor performance

- R_∞: MFLOPS rate on an infinitely long vector
  - Ignores startup latency
  - Useful for determining the throughput on really long vectors
- $N_{1/2}$: vector length necessary to achieve half R_∞
  - Shows how quickly vector performance fails if non-maximal vectors are used
  - Affected greatly by startup latency
- N_v: vector length necessary so that using vector operations to do a computation will be faster than using scalar operations
  - Shows how long vectors have to be to be useful
  - Fast scalar operations and startup latency affect this number

Vector performance: example

- Assume
  - One memory pipeline, 500 MHz
  - $T_{base} = 0$ and $T_{loop} = 15$
- Compute
  - $T_{start} = 10$ (load) + 7 (multiply) + 10 + 6 (add) +12 (store) = 45
  - Need 3 chimes

Time to compute *n* elements

$$T_n = T_{base} + \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_n = 0 + \left\lceil \frac{n}{64} \right\rceil \times (15 + 45) + n \times 3$$

DAXPY with chaining

```
LV      V1,Rx     ; chained w/MULTSV
MULTSV  V2,F0,V1
LV      V3,Ry     ; chained w/ADDV
ADDV    V4,V2,V3
SV      Ry,V4     ; store the result
```

$$\frac{cycles}{element} = \lim_{n \to \infty} \left(3n + \frac{n \times (15 + 45)}{64} \right) = 3 + \frac{60}{64} = 3.94$$

$$R_\infty = 2 \times 500 / 3.94 = 253.8 \text{ MFLOPS}$$

For $N_{1/2}$: $\frac{253.8}{2} = \frac{2 \times 500}{\text{cycles/element}}, n \le 64$

$$7.88n = 0 + 1 \times (15 + 45) + n \times 3 \Rightarrow n = 12.3$$

Vector processor gotchas

- Remember the startup latency!
  - High startup latency will drastically lower performance because vectors can be short
- Improve scalar & vector performance together
  - Otherwise, the machine will perform poorly overall
  - Most *instructions* are scalar, even though most *operations* are vector
- Get a good memory system!
  - True for any processor running heavy scientific codes!
  - This is the main reason that desktop workstations can't match big iron for heavy-duty scientific use

    $\Rightarrow$ Can't move data in and out of the CPU fast enough

The future of computer architecture

- Computer architecture has come a long way in a short time
  - Dramatic improvements in design
  - Dramatic improvements in device performance
  - Where does it go from here?
- What will all those transistors be used for?
  - Prediction: chips will soon have a billion transistors on them
    - At least 100 million transistors, if not a billion…
  - What can be done with all that space?
    - Ever-larger L1/L2 caches?
    - Multiple threads on a chip?
    - Larger register files & vector units?
    - Multi-function integration on-chip (display, I/O, etc.)?
    - Programmable logic on-chip?

Bigger & better processors & storage

- Ever-faster CPUs
  - The Dept. of Energy wants 10 TFLOP machines within five years
    - Factor of 200 faster than current small-scale 50 GFLOP machines!
  - Programming these machines is already difficult!
  - How can we scale up systems to 10 TFLOPs?
- Large-scale storage
  - All of this computing power must be accompanied by ways to store all the data
  - Computing becomes more decentralized => storage may become *more* centralized
    - Storing data for the long-term is difficult: computing cycles are fleeting, but data is forever
    - Losing a few cycles isn't a big problem, but losing data can cause lots of trouble

Ubiquitous computing

- It's been said that a technology has truly arrived when we're not really conscious of using it
  - Telephone
  - Television
  - Automobile
- When will computing be so cheap that it's fully integrated into our lives such that we can't imagine living without it?
  - Computers in every device!
  - All of the computers exchanging information
- What form will it take?
  - Require as little explicit user direction as possible
  - Make it as invisible as possible