# Comparative Architectures

CST Part II, 16 lectures

Lent Term 2005

**Ian Pratt**

Ian.Pratt@cl.cam.ac.uk

# Course Outline

1. Comparing Implementations
   - Developments fabrication technology
   - Cost, power, performance, compatibility
   - Benchmarking

2. Instruction Set Architecture (ISA)
   - Classic CISC and RISC traits
   - ISA evolution

3. Microarchitecture
   - Pipelining
   - Super-scalar
     - static & out-of-order
   - Multi-threading
   - Effects of ISA on $\mu$architecture and vice versa

4. Memory System Architecture
   - Memory Hierarchy

5. Multi-processor systems
   - Cache coherent and message passing

## Understanding design tradeoffs

# Reading material

- OHP slides, articles
- Recommended Book:
  John Hennessy & David Patterson,
  Computer Architecture: a Quantitative Approach
  (3rd ed.) 2002 Morgan Kaufmann
- MIT Open Courseware:
  6.823 Computer System Architecture,
  by Krste Asanovic
- The Web
  `http://bwrc.eecs.berkeley.edu/CIC/`
  `http://www.chip-architect.com/`
  `http://www.geek.com/procspec/procspec.htm`
  `http://www.realworldtech.com/`
  `http://www.anandtech.com/`
  `http://www.arstechnica.com/`
  `http://open.specbench.org/`
- comp.arch News Group

# Further Reading and Reference


- M Johnson
  Superscalar microprocessor design
  1991 Prentice-Hall

- P Markstein
  IA-64 and Elementary Functions
  2000 Prentice-Hall

- A Tannenbaum,
  Structured Computer Organization (2nd ed.)
  1990 Prentice-Hall

- A Someren & C Atack,
  The ARM RISC Chip,
  1994 Addison-Wesley

- R Sites,
  Alpha Architecture Reference Manual,
  1992 Digital Press

- G Kane & J Heinrich,
  MIPS RISC Architecture
  1992 Prentice-Hall

- H Messmer,
  The Indispensable Pentium Book,
  1995 Addison-Wesley

- Gerry Kane and HP,
  The PA-RISC 2.0 Architecture book,
  Prentice Hall

# Course Pre-requisites

- Computer Design (Ib)
  - Some ARM/x86 Assembler
  - Classic RISC pipeline model
  - Load/branch delay slots
  - Cache hierarchies
  - Memory Systems
- Compilers (Ib/II)
  - Code generation
  - Linkage conventions
- Structured Hardware Design
  - Critical paths
  - Memories
- (Concurrent Systems)

# The Microprocessor Revolution

- Mainframe / Scalar Supercomputer
  - CPU consists of multiple components
  - performance improving at 20-35% p.a.
  - often ECL or other exotic technology
  - huge I/O and memory bandwidth
- Microprocessors
  - usually a single CMOS part
  - performance improving at 35-50% p.a.
  - enabled through improvements in fabrication technology
  - huge investment
  - physical advantages of smaller size
  - General Purpose Processors
    - desktop / server
    - SMP / Parallel supercomputers
  - Embedded controllers / SoCs
  - DSPs / Graphics Processors

# Developments in CMOS

- Fabrication line size reduction
  - $0.8\mu$, 0.5, 0.35, 0.25, 0.18, 0.15, *0.13*, *0.09*
  - 10-20% reduction p.a.
  - switching delay reduces with line size
    - $\rightarrow$ increases in clock speed
      - ∗ Pentium 66Mhz @ $0.8\mu$, 150Mhz @ $0.6\mu$, 233MHz @ $0.35\mu$
  - density increases at square of 1/line size
- Die size increases at 10-29% p.a.

$\Rightarrow$ Transistor count increase at 55% p.a.

  - enables architectural jumps
  - 8, 16, 32, 64, *128* bit ALUs
  - large caches
    - ∗ PA-8500: 1.5MB on-chip
  - new functional units (e.g. multiplier)
  - duplicated functional units (multi-issue)
  - whole System On a Chip (SoC)

# Developments in DRAM Technology

- DRAM density
  - increases at 40-60% p.a.
  - equivalent to 0.5-1 address bits p.a.
  - cost dropping at same rate
    - 16M, 64M, *256M*, *1G*

- Consequences for processor architectures:

$\rightarrow$ May not be able to address whole of memory from a single pointer

  - segmentation

$\rightarrow$ May run out of physical address bits

  - banked (windowed) memory

- DRAM performance
  - just 35% latency improvement in 10 years!
  - new bus interfaces make more *sequential b/w* available
    - SDRAM, RAMBUS, DDR, DDR2

# $\mu$processor Development Cycle

- Fabrication technology has huge influence on power and performance

$\rightarrow$ must use the latest fabrication process

- Full custom design *vs.* semi custom
- Keep development cycle short (3-4 years)
  - Non CMOS technology leads to complications
- Advance teams to research:
  - process characteristics
  - key circuit elements
  - packaging
  - floor plan
  - required performance
  - microarchitecture
  - investigate key problems
- Hope ISA features don't prove to be a handicap
- Keep up or die!
- Alpha architects planned for 1000x performance improvement over 25 years

# Power Consumption

- Important for laptops, PDAs, mobile phones, set-top boxes, etc.
- 155W for Digital Alpha 21364 @ 1150MHz
- 130W for Itanium-2 @ 1500MHz
- 90W for AMD Opteron 148 @ 2GHz
- 81W for Pentium-IV @ 3GHz
- 12W for Intel Mobile Pentium M @ 1100Hz
- 420mW for Digital StrongArm @ 233MHz, 2.0V
- 130mW for Digital StrongArm @ 100MHz, 1.65V
- Smaller line size results in lower power
  - lower core voltage, reduced capacitance
  - greater integration avoids inter-chip signalling
- Reduce clock speed to scale power
  - $P = CV^2f$
  - may allow lower voltage
    - potential for cubic scaling
    - better than periodic HALTing

# Performance per Watt

# Cost and Price

- E.g.:
  - \$0.50: 8bit micro controller
  - \$3: XScale (ARM)
    (400MHz, 0.18$\mu$m, 20mm$^2$, 2.1M[1M])
  - \$500: Pentium IV Celeron
    (1.2GHz, 0.13$\mu$m, 131mm$^2$, 28M[4M])
  - \$150: Pentium IV
    (3.2GHz, 0.09$\mu$m, 180mm$^2$, 42M[7M])
  - \$2200: Itanium2
    (1Ghz, 0.18$\mu$m, 421mm$^2$, 221M[15M])
- Costs influenced by die size, packaging, testing
- Large influence by manufacturing volume
- Costs reduce over product life (e.g. 40% p.a.)
  - Yield improves
  - Speed grade binning
  - Fab ‘shrinks’ and ‘steppings’

# Compatibility

- 'Pin' Compatibility (second sourcing)
- Backwards Binary Compatibility
  - 8086, 80286, 80386, 80486, Pentium, Pentium Pro, Pentium II/III/IV, *Itanium*
  - NexGen, Cyrix, AMD, Transmeta
  - typically need to re-optimize
- Typically hard to change architecture
  - Users have huge investment in s/w
  - Binary translators e.g. FX!32, WABI
    - typically interface to native OS
  - Need co-operation from s/w vendors
    - multi-platform support costs $'s
  - Most computer sales are upgrades
- Platform independence initiatives
  - Source, p-Code, JAVA bytecode, .NET

## Compatibility is very important

# Performance Measurement

- Try before you buy! (often not possible)
- System may not even exist yet
  - use cycle-level simulation
- Real workloads often hard to characterize and measure improvements
  - especially interactive
- Marketing hype
  - MHz, MIPS, MFLOPS
- Algorithm kernels
  - Livermore Loops, Linpack
- Synthetic benchmarks
  - Dhrystones, Whetstones, iCOMP
- Benchmark suites
  - SPEC-INT, SPEC-FP, SPEC-HPC, NAS
- Application Benchmarks
  - TPC-C/H/R, SPECNFS, SPECWeb, Quake

**Performance is application dependent**

# Standard Performance Evaluation Corporation

- SPEC is most widely used benchmark
  - processor manufactures
  - workstation vendors
- CPU INT / FP 89, 92, 95, 2000, (2004)
- Suite updated to reflect current workloads
- CINT95/2K: 8/12 integer C programs
- CFP95/2K: 10/14 floating point in C&Fortran
- measures:
  - processor
  - memory system
  - compiler
  - NOT OS, libc, disk, graphics, network

# Choosing programs for SPEC2000

- More programs than SPEC95
- Bigger programs than SPEC95
  - Don't fit in on-chip caches
- Reflect some real workloads
- Run for several minutes
  - Amortize startup overhead & timing inaccuracies
- Not susceptible to trick transformations
  - Vendors invest huge s/w effort
- Fit in 256MB (95 was 64MB)
- Moving target...
- SPEC92, 95, 2K results not translatable

## CINT95 suite (C)

| | |
|---|---|
| 099.go | An AI go-playing program |
| 124.m88ksim | A chip simulator for the Motorola 88100 |
| 126.gcc | Based on the GNU C compiler version 2.5.3 |
| 129.compress | An in-memory version of the utility |
| 130.li | Xlisp interpreter |
| 132.ijpeg | De/compression on in-memory images |
| 134.perl | An interpreter for the Perl language |
| 147.vortex | An object oriented database |

## CFP95 suite (Fortran)

| | |
|---|---|
| 101.tomcatv | Vectorized mesh generation |
| 102.swim | Shallow water equations |
| 103.su2cor | Monte-Carlo method |
| 104.hydro2d | Navier Stokes equations |
| 107.mgrid | 3d potential field |
| 110.applu | Partial differential equations |
| 125.turb3d | Turbulence modelling |
| 141.apsi | Weather prediction |
| 145.fpppp | Quantum chemistry |
| 146.wave5 | Maxwell's equations |

# SPEC reporting

- Time each program to run
- Reproduceability is paramount
  - Take mean of $\geq$ 3 runs
  - Full disclosure
- Baseline measurements
  - SPECint_base95
  - Same compiler optimizations for whole suite
- Peak measurements
  - SPECint95
  - Each benchmark individually tweaked
  - Unsafe optimizations can be enabled!
- Rate measurements for multiprocessors
  - SPECint_rate95, SPECfp_rate95
  - time for N copies to complete x N

# Totalling Results

- How to present results?
  - Present individual results?
  - Arithmetic mean?
  - Weighted harmonic mean?
  - SPEC uses Geometric mean, normalised against a reference platform
    * allows normalization before or after mean
    * performance ratio can be predicted by dividing means
- SPEC95 uses Sun SS10/40 as reference platform

# SPEC CINT95 Results

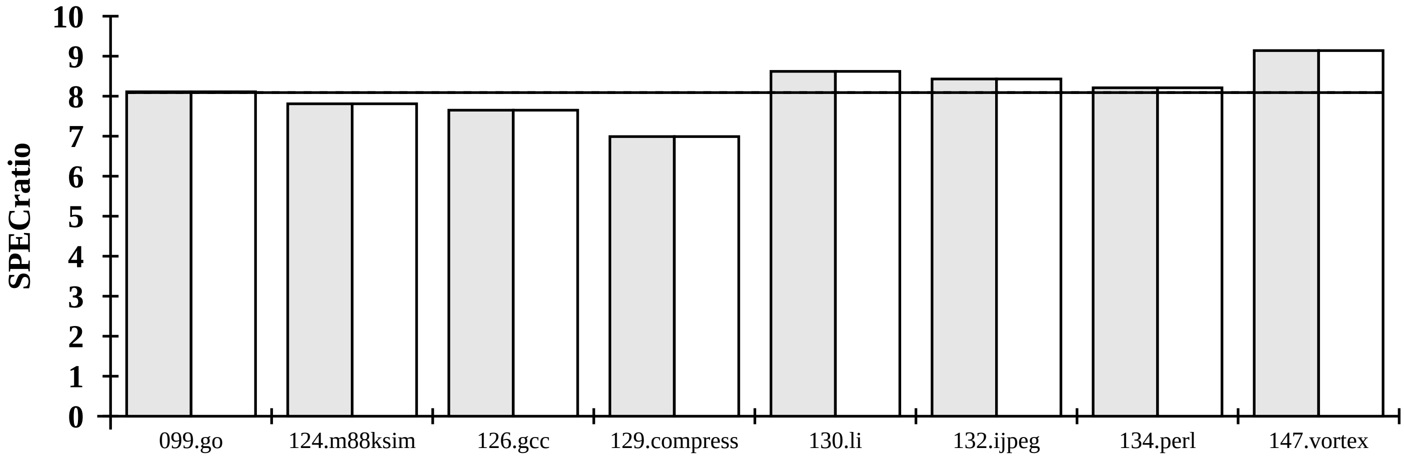


## Intel Corporation
## Alder System (200MHz, 256KB L2)

SPECint95 = 8.09

SPECint_base95 = 8.09

| SPEC license # | 14 | Tested By: | Intel | Test Date: | Oct-95 | Hardware Avail: | May-96 | Software Avail: | Feb-96 |
|---|---|---|---|---|---|---|---|---|---|




Hardware/Software Configuration for:
Alder System (200MHz, 256KB L2)

**Hardware**

| | |
|---|---|
| Model Name: | Alder |
| CPU: | 200MHz Pentium Pro Processor |
| FPU: | Integrated |
| Number of CPU(s): | 1 |
| Primary Cache: | 8KBI+8KBD |
| Secondary Cache: | 256KB(I+D) |
| Other Cache: | None |
| Memory: | 128MB (60ns fast page) |
| Disk Subsystem: | 2GB ST32550W |
| Other Hardware: | AHA-2940W Controller |

**Software**

| | |
|---|---|
| Operating System: | UnixWare 2.0, SDK |
| Compiler: | Intel C Reference Compiler 2.2 Beta |
| File System: | ufs, vxfs (/tmp as 8MB /tmpfs) |
| System State: | Single user (root + killall) |

| Benchmark # and Name | Reference Time | Base Run Time | Base SPEC Ratio | Run Time | SPEC Ratio |
|---|---|---|---|---|---|
| 099.go | 4600 | 567 | 8.11 | 567 | 8.11 |
| 124.m88ksim | 1900 | 243 | 7.81 | 243 | 7.81 |
| 126.gcc | 1700 | 222 | 7.65 | 222 | 7.65 |
| 129.compress | 1800 | 258 | 6.99 | 258 | 6.99 |
| 130.li | 1900 | 220 | 8.62 | 220 | 8.62 |
| 132.ijpeg | 2400 | 285 | 8.43 | 285 | 8.43 |
| 134.perl | 1900 | 232 | 8.21 | 232 | 8.21 |
| 147.vortex | 2700 | 295 | 9.14 | 295 | 9.14 |
| SPECint_base95 (G. Mean) | | | 8.09 | | |
| | | SPECint95 (G. Mean) | | | 8.09 |

## Notes/Tuning Information

```
Base and non-base flags are the same and use Feedback Directed Optimization
Pass1: -tp p6 -ipo -xi -prof_gen -ircdb_dir /tmp/IRCDB
Pass2: -tp p6 -ipo -xi -prof_use -ircdb_dir /tmp/IRCDB
-ircdb_dir is a location flag and not an optimization flag
Portability: 124: -DSYSV -DLEHOST 130, 134, 147: -lm  132: -DSYSV  126: -lm -lc -L/usr/ucblib -lucb -lmalloc
Memory subsystem is four-way interleaved.
```

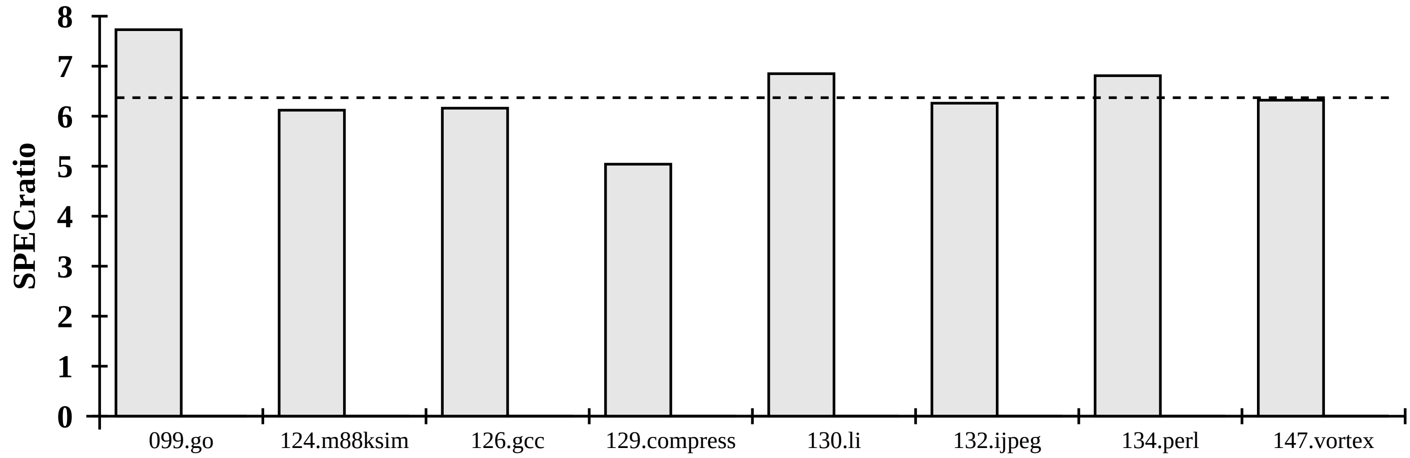
For More Information Contact:
SPEC
10754 Ambassador Drive, Suite 201
Manassas, VA 22110
(703) 331-0180
info@specbench.org
http://www.specbench.org

# SPEC CINT95 Results



| | |
|---|---|
| Intel 440LX motherboard<br>Pentium Pro 200 | SPECint95 = --<br>SPECint_base95 = 6.37 |

| SPEC license # 1178 | Tested By: Ian Pratt, CUCL | Test Date: Date | Hardware Avail: Date | Software Avail: Date |
|---|---|---|---|---|




Hardware/Software Configuration for:
Pentium Pro 200

**Hardware**

Model Name: Intel 440LX
CPU: Pentium Pro 200
FPU:
Number of CPU(s): 1
Primary Cache: 8KB+8KB
Secondary Cache: 256KB
Other Cache:
Memory: 128MB
Disk Subsystem: 4GB
Other Hardware:

**Software**

Operating System: Linux 20.0.30
Compiler: gcc 2.7.2p
File System: ext2
System State: multiuser

| Benchmark # and Name | Reference Time | Base Run Time | Base SPEC Ratio | Run Time | SPEC Ratio |
|---|---|---|---|---|---|
| 099.go | 4600 | 595 | 7.73 | -- | -- |
| 124.m88ksim | 1900 | 310 | 6.12 | -- | -- |
| 126.gcc | 1700 | 276 | 6.16 | -- | -- |
| 129.compress | 1800 | 357 | 5.04 | -- | -- |
| 130.li | 1900 | 277 | 6.85 | -- | -- |
| 132.ijpeg | 2400 | 384 | 6.26 | -- | -- |
| 134.perl | 1900 | 279 | 6.81 | -- | -- |
| 147.vortex | 2700 | 427 | 6.32 | -- | -- |
| SPECint_base95 (G. Mean) | | | 6.37 | | |
| | | SPECint95 (G. Mean) | | | -- |

## Notes/Tuning Information

```
Portability flags were:
Baseline flags were: -O2 -fomit-frame-pointer
Nonbase flags were:
```

For More Information contact
SPEC c/o NCGA
2722 Merrilee Drive, Suite 200
Fairfax, VA 22031
(703) 698-9604 ext 318
spec-ncga@cup.portal.com

-- Prepared By: --

# CINT2000 Result

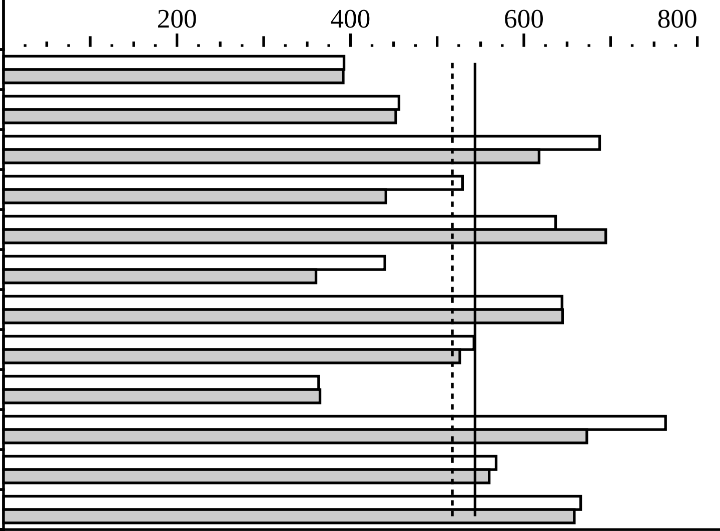


## Compaq Computer Corporation
## AlphaServer ES40 Model 6/833

SPECint2000 = 544
SPECint_base2000 = 518

SPEC license #: 2 | Tested by: Compaq NH | Test date: Oct-2000 | Hardware Avail: Jan-2001 | Software Avail: Nov-2000

| Benchmark | Reference Time | Base Runtime | Base Ratio | Runtime | Ratio |
|---|---|---|---|---|---|
| 164.gzip | 1400 | 358 | 392 | 357 | 393 |
| 175.vpr | 1400 | 309 | 452 | 307 | 456 |
| 176.gcc | 1100 | 178 | 617 | 160 | 687 |
| 181.mcf | 1800 | 408 | 441 | 340 | 529 |
| 186.crafty | 1000 | 144 | 694 | 157 | 637 |
| 197.parser | 1800 | 500 | 360 | 409 | 440 |
| 252.eon | 1300 | 202 | 645 | 202 | 644 |
| 253.perlbmk | 1800 | 342 | 526 | 332 | 543 |
| 254.gap | 1100 | 301 | 365 | 303 | 363 |
| 255.vortex | 1900 | 282 | 673 | 249 | 763 |
| 256.bzip2 | 1500 | 268 | 560 | 264 | 568 |
| 300.twolf | 3000 | 456 | 658 | 451 | 666 |




### Hardware

| | |
|---|---|
| CPU: | Alpha 21264B |
| CPU MHz: | 833 |
| FPU: | Integrated |
| CPU(s) enabled: | 1 |
| CPU(s) orderable: | 1 to 4 |
| Parallel: | No |
| Primary Cache: | 64KB(I)+64KB(D) on chip |
| Secondary Cache: | 8MB off chip |
| L3 Cache: | None |
| Other Cache: | None |
| Memory: | 16GB |
| Disk Subsystem: | 1x8GB BD0096349A |
| Other Hardware: | Ethernet |

### Software

| | |
|---|---|
| Operating System: | Tru64 UNIX V5.1 + Patch Kit 1 libc |
| Compiler: | Compaq C V6.3-129-44A8I<br>Compaq C++ V6.2-033-4298H |
| File System: | AdvFS |
| System State: | Multi-user |

### Notes/Tuning Information

```
Baseline C  : cc  -arch ev6 -fast GEMFB ONESTEP
         C++: cxx -arch ev6 -O2 ONESTEP

GEMFB: fdo_pre0     = mkdir /tmp/pb; rm -f /tmp/pb/${baseexe}*
       PASS1_CFLAGS = -prof_gen_noopt    -prof_dir /tmp/pb
       PASS2_CFLAGS = -prof_use_feedback -prof_dir /tmp/pb
       (base uses directory /tmp/pb; peak uses /tmp/pp)

SPIKEFB: fdo_post2  = spike -feedback ${baseexe} -o tmp ${baseexe};
                      mv tmp ${baseexe}
Peak: cc [except eon: cxx] -arch ev6 ONESTEP plus:
     164.gzip: -g3 -fast -O4 +GEMFB
      175.vpr: -g3 -fast -O4 +GEMFB
      176.gcc: -g3 -fast -O4 -xtaso_short +GEMFB
      181.mcf: -g3 -fast -xtaso_short +GEMFB
   186.crafty: -g3 -fast -O4 -inline speed
   197.parser: -g3 -fast -O4 -xtaso_short +GEMFB
      252.eon: -O2
  253.perlbmk: -g3 -fast +GEMFB +SPIKEFB
      254.gap: -g3 -fast -O4 +GEMFB
```

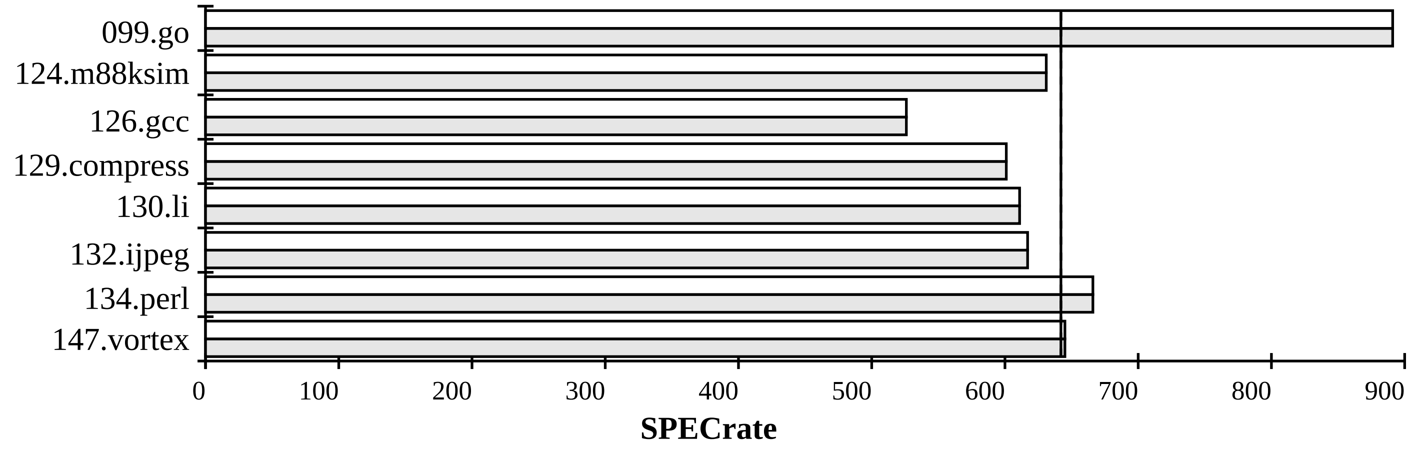
# SPEC CINT95rate Results



# Digital Equipment Corp.
# AlphaServer 8400 5/300

| | |
|---|---|
| SPECint_rate95 = | 642 |
| SPECint_rate_base95 = | 642 |

| SPEC license # | Tested By: | Test Date: | Hardware Avail: | Software Avail: |
|---|---|---|---|---|
| 2 | Digital PKO | Oct-95 | Apr-95 | Aug-95 |




Hardware/Software Configuration for:
AlphaServer 8400 5/300

**Hardware**

| | |
|---|---|
| Model Name: | AlphaServer 8400 5/300 |
| CPU: | 300 MHz 21164 |
| FPU: | Integrated |
| Number of CPU(s): | 10 |
| Primary Cache: | 8KBI+8KBD on chip |
| Secondary Cache: | 4MB |
| Other Cache: | none |
| Memory: | 1GB |
| Disk Subsystem: | 1 x 2GB<br>1 x 2GB |
| Other Hardware: | Ethernet |

**Software**

| | |
|---|---|
| Operating System: | Digital UNIX V3.2C (Rev 148) |
| Compiler: | DEC C V5.0-106 |
| File System: | UFS |
| System State: | Multi User |

| Benchmark # and Name | Base Copies | Base Run Time | Base SPEC Ratio | Copies | Run Time | SPEC Ratio |
|---|---|---|---|---|---|---|
| 099.go | 10 | 464 | 891 | 10 | 464 | 891 |
| 124.m88ksim | 10 | 271 | 631 | 10 | 271 | 631 |
| 126.gcc | 10 | 291 | 526 | 10 | 291 | 526 |
| 129.compress | 10 | 270 | 601 | 10 | 270 | 601 |
| 130.li | 10 | 280 | 611 | 10 | 280 | 611 |
| 132.ijpeg | 10 | 350 | 617 | 10 | 350 | 617 |
| 134.perl | 10 | 257 | 666 | 10 | 257 | 666 |
| 147.vortex | 10 | 377 | 645 | 10 | 377 | 645 |
| SPECint_rate_base95 (G. Mean) | | | 642 | | | |
| | | SPECint_rate95 (G. Mean) | | | | 642 |

## Notes/Tuning Information

```
Baseline Optimizations: -O5 -ifo -non_shared -om
Portibility Flags: 124.m88ksim: -DLEHOST  134.perl: -DI_TIME
147.vortex: -D__RISC_64__

Compiler invokation: cc -migrate -std1 (DEC C with -std1 for strict ANSI)
```

# Top SPEC2000 Results for each ISA

| machine | processor | cpu MHz | cache sizes | int | fp |
|---|---|---|---|---|---|
| Intel D875 | Pentium IV-X | 3200 | 12*/8+256+2M | 1620 | 1516 |
| AMD/ASUS | Opteron148 | 2200 | 64/64+1M | 1477 | 1514 |
| Intel D875 | Pentium IV | 3200 | 12*/8+512 | 1330 | 1287 |
| HP rx2600 | Itanium2 | 1500 | 16/16+256+6M | 1322 | 2119 |
| IBM p690 | Power4+ | 1700 | 64/32+1.5M+(128M) | 1113 | 1699 |
| HP Alpha ES45 | 21264C | 1250 | 64/64+(16M) | 928 | 1482 |
| Fujitsu | SPARC64-V | 1350 | 128+128/2M | 905 | 1340 |
| Apple | PPC970 (G5) | 2000 | 64/32+512 | 800 | 840 |
| HP | Pentium-M | 1000 | 32/32+1024 | 687 | 552 |
| HP c3750 | PA-8700 | 875 | 768/1.5M | 678 | 674 |
| SGI Orgin 3200 | R14000 | 600 | 32/32+(8M) | 500 | 529 |
| HP rx4610 | Itanium | 800 | 16/16+96+(4M) | 379 | 701 |

# Selected SPEC95 Results

| machine | processor | cpu MHz | cache sizes | int_base | fp_base |
|---|---|---|---|---|---|
| Sun SS10/40 | SuprSP | 40 | 20/16 | 1.00 | 1.00 |
| Intel 440BX | Pentium II | 300 | 16/16+(512) | 12.2 | 8.4 |
| Intel 440EX | Celeron A | 300 | 16/16+128 | 11.3 | 8.3 |
| Intel 440EX | Celeron | 300 | 16/16 | 8.3 | 5.8 |
| Compaq PC164LX | 21164 | 533 | 8/8+96+(4M) | 16.8 | 20.7 |
| Compaq PC164SX | 21164PC | 533 | 16/16+(1M) | 12.2 | 14.1 |
| Intel 440BX | Pentium II | 450 | 16/16+(512) | 17.2 | 11.8 |
| Intel 440BX | Pentium II | 400 | 16/16+(512) | 15.8 | 11.4 |
| Intel 440BX | Pentium II | 350 | 16/16+(512) | 13.9 | 10.2 |
| Intel 440BX | Pentium II | 330 | 16/16+(512) | 13.0 | 8.8 |
| Intel 440BX | Pentium II | 300 | 16/16+(512) | 11.9 | 8.1 |
| Intel 440BX | Pentium II | 266 | 16/16+(512) | 10.7 | 7.5 |
| Intel 440BX | Pentium II | 233 | 16/16+(512) | 9.4 | 6.7 |
| DEC 4100/5/400 | A21164 | 400/75 | 8/8+96+4M | 10.1 | 16.0 |
| DEC 4100/5/400 | 2xA21164 | 400/75 | 8/8+96+4M | 10.1 | 20.7 |
| DEC 4100/5/400 | 4xA21164 | 400/75 | 8/8+96+4M | 10.1 | 26.6 |
| Intel XXpress | Pentium | 200 | 8/8+1M | 5.47 | 2.92 |
| Intel Alder | PentPro | 200 | 8/8+256 | 8.09 | 5.99 |

# Comparing Implementations Summary

- Fabrication technology has a huge influence
- Exponential improvement in technology
- Processor for a product chosen on:
  - Instruction Set Compatibility
  - Power Consumption
  - Price
  - Performance
- Performance is application dependent
  - Avoid MIPS, MHz
  - Benchmark suites

# Instruction Set Architecture

- Processor s/w interface
- Externally visible features
  - Word size
  - Operation sets
  - Register set
  - Operand types
  - Addressing modes
  - Instruction encoding
- Introduction of new ISAs now rare
- ISAs need to last several generations of implementation
- How do you compare ISAs ?
  - yields 'best' implementation
    * performance, price, power
    * are other factors equal?
  - 'aesthetic qualities'
    * 'nicest' for systems programmers

# Instruction Set Architecture

- New implementations normally backwards compatible
  - Should execute old code correctly
  - Possibly some exceptions e.g.
    * Undocumented/unsupported features
    * Self modifying code on 68K
  - May add new features e.g. FP, divide, sqrt, SIMD, FP-SIMD
  - May change execution timings
  - $\rightarrow$ CPU specific optimization
  - Can rarely remove features
    * Unless never used
    * software emulation fast enough
  - $\rightarrow$ Layers of architectural baggage
    * (8086 16bit mode on Pentium IV)
- Architecture affects ease of utilizing new techniques e.g.
  - Pipelining
  - Super-scalar (multi-issue)
- But x86 fights real hard!
  - more T's tolerable unless on critical path

# Reduced Instruction Set Computers

- RISC loosely classifies a number of Architectures first appearing in the 80's
- Not really about reducing number of instructions
- Result of quantitative analysis of the usage of existing architectures
  - Many CISC features designed to eliminate the 'semantic gap' were not used
- RISC designed to easily exploit:
  - Pipelining
    - Easier if most instructions take same amount of time
  - Virtual Memory (paging)
    - Avoid tricky exceptional cases
  - Caches
    - Use rest of Si area
- Widespread agreement amongst architects

# Amdahl's Law

- Every 'enhancement' has a cost:
  - Would Si be better used elsewhere?
    - e.g. cache
  - Will it slow down other instructions?
    - e.g. extra gate delays on critical path
    - $\rightarrow$ longer cycle time
- Even if it doesn't slow anything else down, what overall speedup will it give?
- size and delay

$$speedup = \frac{execution\ \ time\ \ for\ \ entire\ \ task\ \ without\ \ using\ \ enhancement}{execution\ \ time\ \ for\ \ entire\ \ task\ \ using\ \ enhancment\ \ \boldsymbol{when\ \ possible}}$$

# Amdahl's Law :2

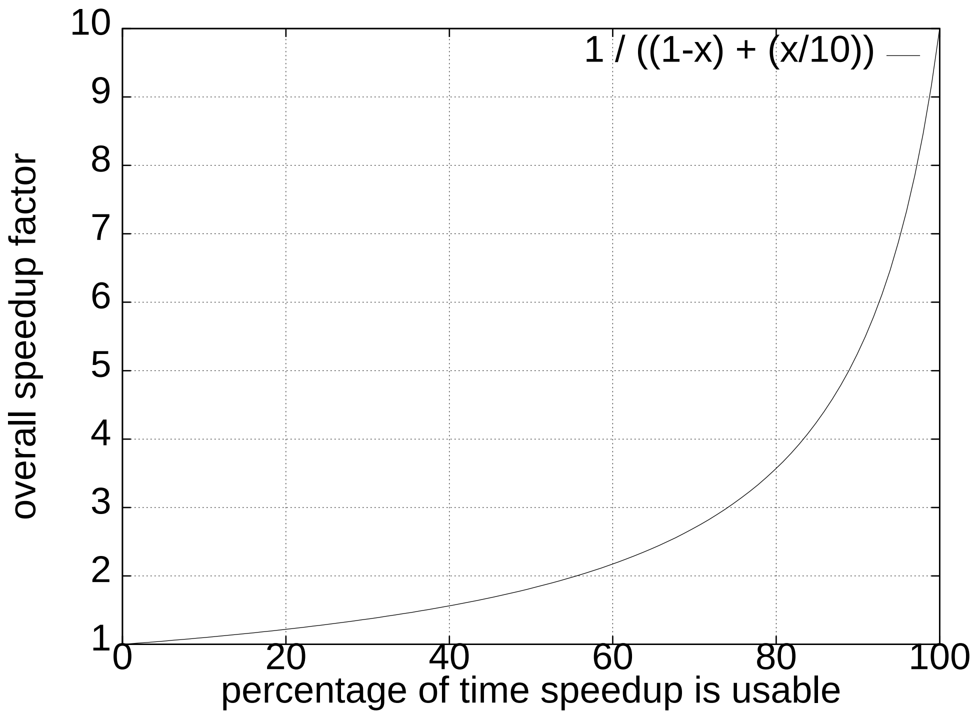
- How frequently can we use enhancement?
  - examine instruction traces e.g. SPEC
  - will code require different optimization?
  - $Fraction_{enhanced}$
- When we can use it, what speedup will it give?
  - $Speedup_{enhanced}$
  - e.g. cycles before/cycles after

$$Speedup_{overall} = \frac{1}{(1 - Fraction_{enhanced}) + \frac{Fraction_{enhanced}}{Speedup_{enhanced}}}$$

**$\rightarrow$ Spend resources where time is spent**

**Optimize for the common case**

# Amdahl's Law for Speedup=10



# Amdahl's Law Example

- FPSQRT is responsible for 20% of execution time in a (fictitious) critical benchmark
- FP operations account for 50% of execution time in total
- Proposal A:
  - New FPSQRT hardware with 10x performance

$$speedup_A = \frac{1}{(1-0.2)+\frac{0.2}{10}} = \frac{1}{0.82} = 1.22$$

- Proposal B:
  - Use Si area to double speed all FP operations

$$speedup_B = \frac{1}{(1-0.5)+\frac{0.5}{2}} = \frac{1}{0.75} = 1.33$$

- $\rightarrow$ Proposal B is better
- (Probably much better for other users)

# Word Size

- Native size of an integer register
  - 32bits on ARM, MIPS II, x86 32bit mode
  - 64bits on Alpha, MIPS III, SPARC v8, PA-RISC v2
- NOT size of FP or SIMD registers
  - 64 / 128 bit on Pentium III
- NOT internal data-path width
  - 64bit internal paths in Pentium III
- NOT external data-bus width
  - 8bit Motorola 68008
  - 128bit Alpha 21164
- NOT size of an instruction
  - Alpha, MIPS, etc instructions 32bit
- But, 'word' also used as a type size
  - 4 bytes on ARM, MIPS
  - 2 bytes on Alpha, x86
    - longword = 4 bytes, quadword = 8

# 64bit *vs* 32bit words

- Alpha, MIPS III, SPARC v8, PA-RISC v2

✔ Access to a large region of address space from a single pointer

  – large data-structures

  – memory mapped files

  – persistent objects

✔ Overflow rarely a concern

  – require fewer instructions

✘ Can double a program's data size

  – need bigger caches, more memory b/w

✘ May slow the CPU's max clock speed

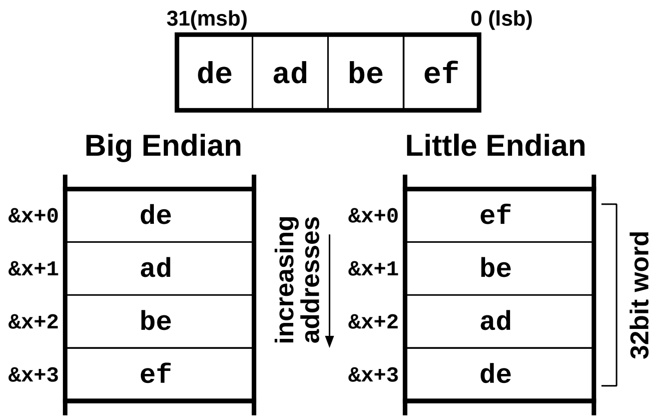
- Some programs gain considerably from 64bit, others get no benefit.

- Some OS's and compilers provide support for 32bit binaries

# Byte Sex

- Little Endian camp
  - Intel, Digital
- Big Endian camp
  - Motorola, HP, IBM
  - Sun: ‘Network Endian’, JAVA
- Bi-Endian Processors
  - Fixed by motherboard design
  - MIPS, ARM
- Endian swapping instructions

```
int   x= 0xdeadbeef;
char *p= (char*)&x;
if(*p == 0xde) printf("Big Endian");
if(*p == 0xef) printf("Little Ebdian");
```

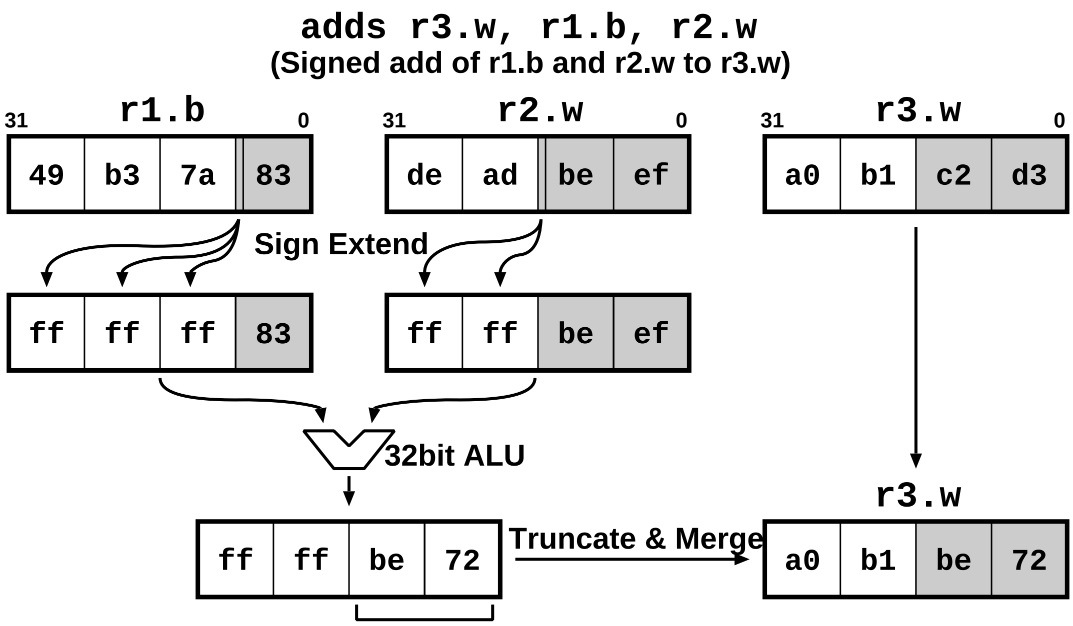


# Data Processing Instructions

- 2's Complement Arithmetic
  - add, subtract, multiply, compare, multiply
  - some: divide, modulus
- Logical
  - and, or, not, xor, bic, . . .
- Shift
  - shift left, logical shift right, arithmetic shift right
  - some: rotate left, rotate right

# Operand Size

- CISC
  - 8,16,32 bit operations
  - zero/sign extend sources
    - need unsigned/signed instrs
  - merge result into destination
  - some even allow mixed size operands




- RISC
  - Word size operations only
  - (except 64bit CPUs often support 32bit ops)
  - Pad `char` and `short` to word

# (Zero/Sign Extension)

- Unsigned values: zero extend
  - e.g. 8bit values to 32bit values
    ```
    unsigned char a; int b;
     and  b ← a, #0xff
    ```
- Signed values: sign extend
  - e.g. 8bit values to 32bit values
  - Replicate sign bit
    ```
    char a; int b;
     lsl  b ← a, #24
     asr  b ← b, #24
    ```
- C: 32bit to 8bit
  - Just truncate
    ```
     and  b ← a, #0xff
    ```

# CISC instructions RISC dropped

- Emulated in RISC:

| | | | | |
|---|---|---|---|---|
| `move` | `r1` $\leftarrow$ `r2` | e.g. | `or` | `r1` $\leftarrow$ `r2, r2` |
| `zero` | `r1` | e.g. | `xor` | `r1` $\leftarrow$ `r1, r1` |
| `neg` | `r1` | e.g. | `sub` | `r1` $\leftarrow$ `#0, r1` |
| `nop` | | e.g. | `or` | `r1` $\leftarrow$ `r1, r1` |
| `sextb` | `r1` $\leftarrow$ `r2` | e.g. | `lsl` | `r1` $\leftarrow$ `r2, #24;` |
| | | | `asr` | `r1` $\leftarrow$ `r1, #24` |

- Used too infrequently:
  - POLY, polynomial evaluation (VAX)
  - BCD, bit-field operations (68k)
  - Loop and Procedure call primitives
    - ∗ Not quite right for every HLL
    - ∗ Unable to take advantage of compiler's analysis

- Exceptions & interrupts are awkward:
  - memcpy/strcmp instructions

# New Instructions

- integer divide, sqrt
- popcount, priority_encode
- Integer SIMD (multimedia)
  - Intel MMX, SPARC VIS, Alpha, PA-RISC MAX
  - MPEG, JPEG, polygon rendering
  - parallel processing of packed sub-words
  - E.g. 8x8, 4x16 bit packed values in 64b word
  - arithmetic ops with 'saturation'
    - s8 case: 125+4 = 127
  - min/max, logical, shift, permute
  - RMS error estimation (MPEG encode)
  - Will compilers ever use these instrs?
- FP SIMD (3D geometry processing)
  - E.g. 4x32 bit single precision
  - streaming vector processing
  - Intel SSE, AMD 3D-Now, PPC AltiVec
- prefetch / cache hints (e.g. non-temporal)
- Maintaining backwards compatiblity
  - Use alternate routines
  - Query CPU feature set

# Registers and Memory

- Register set types
  - Accumulator architectures
  - Stack
  - GPR
- Number of operands
  - 2
  - 3
- Memory accesses
  - any operand
  - one operand
  - load-store only

# Accumulator Architectures

- Register implicitly specified
- E.g. 6502, 8086 (older machines)

```
LoadA   foo
AddA    bar
StoreA  res
```

- Compact instruction encoding
- Few registers, typically $\leq 4$ capable of being operands in arithmetic operations
- Forced to use memory to store intermediate values
- Registers have special functions
  - e.g. loop iterators, stack pointers
- Compiler writers don't like non-orthogonality

# Stack Architectures

- Operates on top two stack items
- E.g. Transputer, (Java)

```
Push  foo
Push  bar
Add
Pop   res
```

- Stack used to store intermediate values
- Compact instruction encoding
- Smaller executable binaries, good if:
  - memory is expensive
  - downloaded over slow network
- Fitted well with early compiler designs

# General Purpose Register Sets

- Post 1980 architectures, both RISC and CISC
- 16,32,*128* registers for intermediate values
- Separate INT and FP register sets
  - Int ops on FP values meaningless
  - RISC: Locate FP regs in FP unit
- Separate Address/Data registers
  - address regs used as bases for mem refs
  - e.g. Motorola 68k
  - not favoured by compiler writers ($8 + 8 \neq 16$)
  - RISC: Combined GPR sets

# Load-Store Architecture

- Only load/store instructions ref memory
- The RISC approach

→ Makes pipelining more straightforward

```
Load   r1  ← foo
Load   r2  ← bar
Add    r3  ← r1, r2
Store  res ← r3
```

- Fixed instruction length (32bits)
- 3 register operands
- Exception: ARM-Thumb, MIPS-16 is two operand
  - more compact encoding (16bits)

# Register-Memory

- ALU instructions can access 1 or more memory locations
- E.g. Intel x86 32bit modes
  - 2 operands
  - can't both be memory

```
Load   r1←foo
Add    r1←bar
Store  res←r1
```

- E.g. DEC VAX
  - 2 and 3 operand formats
  - fully orthogonal

```
Add  res←bar,foo
```

- Fewer instructions
  - Fewer load/stores
  - Each instruction may take longer
  - $\rightarrow$ Increased cycle time
- Variable length encoding
  - May be more compact
  - May be slower to decode

# Special Registers : 1

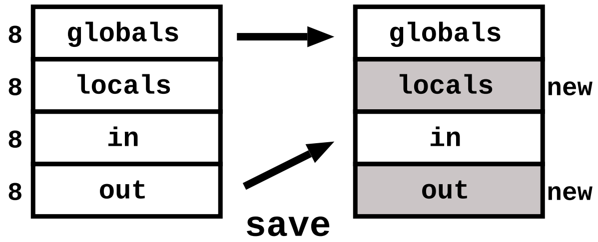
- Zero register
  - Read as Zero, Writes discarded
  - e.g. Alpha, Mips, Sparc, IA-64
  - Data move: `add r2` $\leftarrow$ `r1, r31`
  - nop: `add r31` $\leftarrow$ `r31, r31`
  - prefetch: `ldl r31` $\leftarrow$ `(r1)`
  - Zero is a frequently used constant
- Program Counter
  - NOT usually a GPR
  - Usually accessed by special instructions e.g. branch, branch and link, jump
  - But, PC is GPR r15 on ARM

# Special Registers : 2

- Condition code (Flag) registers
  - Carry, Zero, Negative, Overflow
  - Used by branches, conditional moves
  - Critical for pipelining and super-scalar
  - CISC: one CC reg updated by all instructions
  - ARM, SPARC: one CC reg, optionally updated
  - PowerPC: multiple CC regs (instr chooses)
  - IA64: 64 one bit predicate regs
  - Alpha, MIPS: no special CC regs
- Link registers
  - Subroutine call return address
  - CISC: pushed to stack
  - RISC: saved to register
    - register conventions
    - only push to stack if necessary
  - Jump target/link regs (PowerPC, IA-64)
  - fixed GPR (r14, ARM) (r31,MIPS)
  - GPR nominated by individual branch (Alpha)

# Register Conventions

- Linkage (Procedure Call) Conventions
  - Globals: `sp`, `gp` etc.
  - Args: First (4-6) args (rest on stack)
  - Return value: (1-2)
  - Temps: (8-12)
  - Saved: (8-9) Callee saves
- Goal: spill as few registers as possible in total
- Register Windows (SPARC)
  - `save` and `restore`
  - 2-32 sets of windows in ring
  - 16 unique registers per window
  - spill/fill windows to special stack




- IA-64: Allows variable size frames
  - 32 globals
  - 0-8 args/return, 0-96 locals/out args
  - h/w register stack engine operates in background

# Classic RISC Addressing Modes

- Register
  - `Mov r0 ← r1`
  - `Regs[r0] = Regs[r1]`
  - Used when value held in register
- Immediate
  - `Mov r0 ← 42`
  - `Regs[r0] = 42`
  - Constant value limitations
- Register Indirect
  - `Ldl r0 ← [r1]`
  - `Regs[r0] = Mem[ Regs[r1] ]`
  - Accessing variable via a pointer held in reg
- Register Indirect with Displacement
  - `Ldl r0 ← [r1, #128]`
  - `Ldl r0 ← 128(r1)`
  - `Regs[r0] = Mem[ 128 + Regs[r1] ]`
  - Accessing local variables

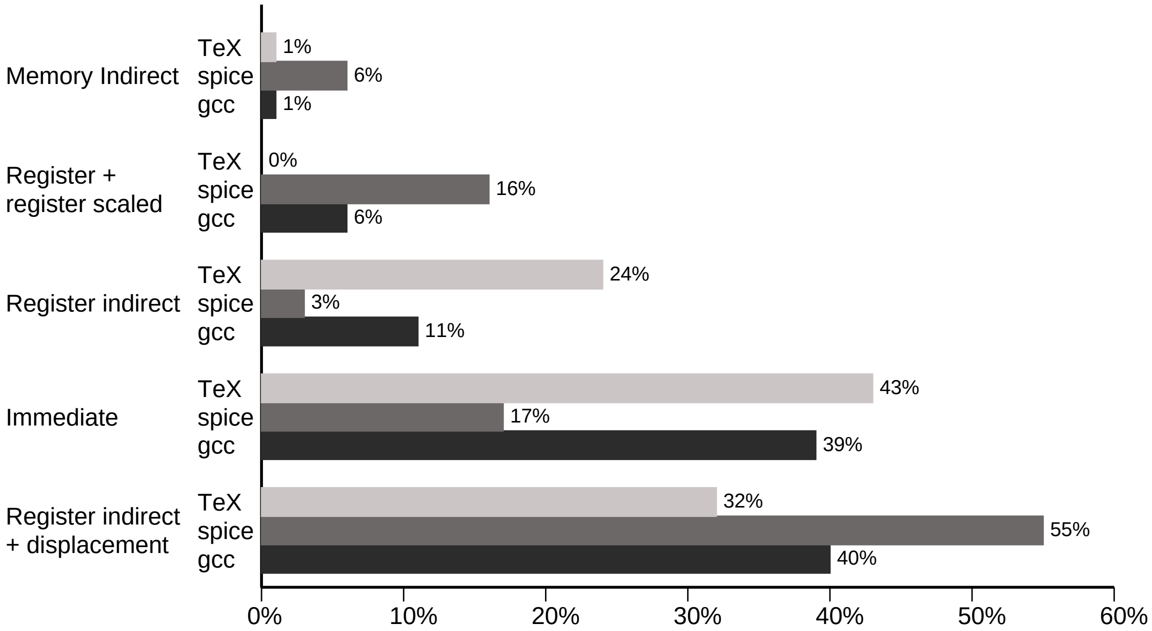
# Less RISCy addr modes

- ARM and PowerPC
- Register plus Register (Indexed)
  - `Ldl r0` $\leftarrow$ `[r1,r2]`
  - `Regs[r0] = Mem[ Regs[r1] + Regs[r2] ]`
  - Random access to arrays
  - e.g. r1=base, r2=index
- Register plus Scaled Register
  - `Ldl r0` $\leftarrow$ `[r1, r2, asl #4]`
  - `Regs[r0] = Mem[ Regs[r1] + (Regs[r2]`$\ll$`4)]`
  - Array indexing
  - sizeof(element) is power of 2, r2 is loop index
- Register Indirect with Displacement and Update
  - Pre inc/dec `Ldl r0` $\leftarrow$ `[r1!, #4]`
  - Post inc/dec `Ldl r0` $\leftarrow$ `[r1], #4`
  - C `*(++p)` and `*(p++)`
  - Creating stack (local) variables
  - Displacement with post update is IA-64's only addressing mode

# CISC Addressing Modes

- Direct (Absolute)
  - `Mov r0` $\leftarrow$ `(1000)`
  - `Regs[r0] = Mem[1000]`
  - Offset often large
  - x86 Implicit base address
  - Most CISCs
- Memory Indirect
  - `Mov r0` $\leftarrow$ `@[r1]`
  - `Regs[r0] = Mem[ Mem[ Regs[r1] ] ]`
  - Two memory references,
  - C `**ptr`, linked lists
- PC Indirect with Displacement
  - `Mov r0` $\leftarrow$ `[PC, #128]`
  - `Regs[r0] = Mem[ PC + 128 ]`
  - Accessing constants

# Why did RISC choose these addressing modes?




Frequency of addressing modes (VAX)

- RISC
  - immediate
  - register indirect with displacement

- ARM, PowerPC reduce instruction counts by adding:
  - register + register scaled
  - index update

# Immediates and Displacements

- CISC: As instructions are variable length, immediates and displacements can be any size (8,16,32 bits)
- RISC: How many spare bits in instruction format?
- Immediates
  - used by data-processing instructions
  - usually zero extended (unsigned)
    - `add` $\rightarrow$ `sub`
    - `and` $\rightarrow$ `bic`
  - For traces on previous slide:
    50-70% fit in 8bits, 75-80% in 16bits
  - IA-64 22/14, MIPS 16, Alpha 8,
    ARM 8 w/ shift
- Displacement values in load and stores
  - Determine how big a data segment you can address without reloading base register
  - usually sign extended
  - MIPS 16, Alpha 16, ARM 12, IA-64 9

# Instruction Encoding

RISC: small number of fixed encodings of same length

| | | | | | | |
|---|---|---|---|---|---|---|
| Operation | Ra | Rb | Signed Displacement | | | load/ store |
| Operation | Ra | Rb | Zero SBZ | Function | Rdest | operate |
| Operation | Ra | Immediate Value | | Function | Rdest | operate immediate |
| Operation | Ra | Signed Displacement | | | | branch |

RISC instruction words are 32 bit

IA-64 packs three 41 bit instructions into a 128 bit ‘bundle’

VAX: fully variable. Operands specified independently

| | | | | | |
|---|---|---|---|---|---|
| Operation and # of operands | Address specifier 1 | Address field 1 | •••• | Address specifier N | Address field N |

x86: knows what to expect after first couple of bytes

| | | | | |
|---|---|---|---|---|
| Operation | Address specifier | Address field | | |
| Operation | Address specifier | Address field1 | Address field2 | |
| Operation | Address specifier | Extended specifier | Address field1 | Address field2 |

# Code Density Straw Poll

- CISC: Motorola 68k, Intel x86
- RISC: Alpha, Mips. PA-RISC
- Very rough-figures for 68k and Mips include statically linked libc

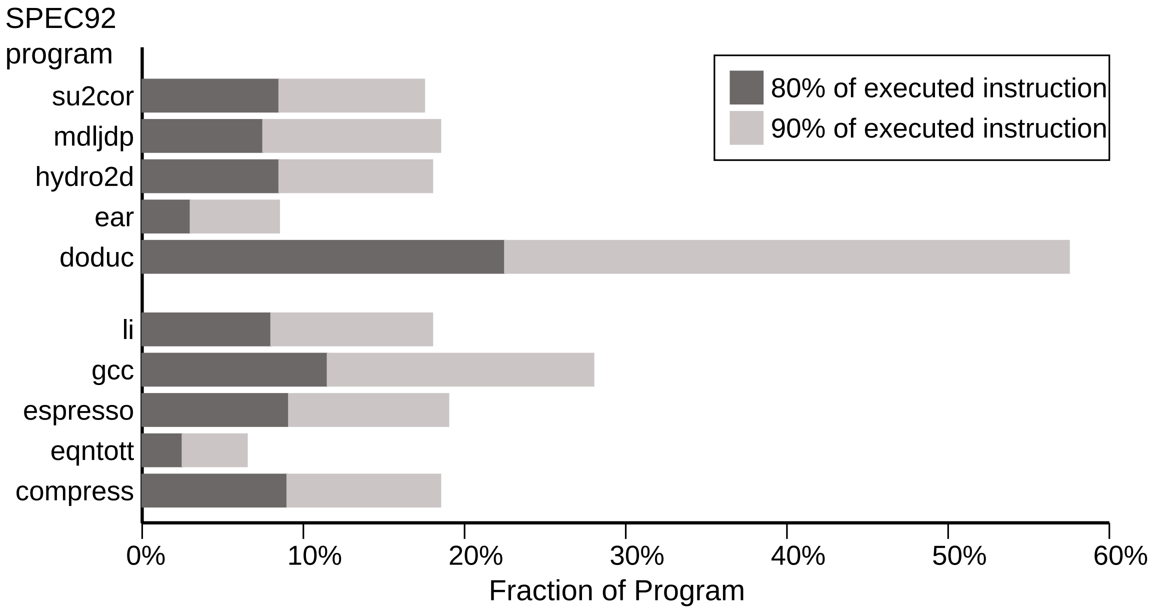
| arch | text | data | bss | total | filename |
|---|---|---|---|---|---|
| x86 | 29016 | 14861 | 468 | 44345 | gcc |
| 68k | 36152 | 4256 | 360 | 40768 | |
| alpha | 46224 | 24160 | 472 | 70856 | |
| mips | 57344 | 20480 | 880 | 78704 | |
| hp700 | 66061 | 15708 | 852 | 82621 | |
| x86 | 995984 | 156554 | 73024 | 1225562 | gcc-cc1 |
| alpha | 1447552 | 272024 | 90432 | 1810008 | |
| hp700 | 1393378 | 21188 | 72868 | 1487434 | |
| 68k | 932208 | 16992 | 57328 | 1006528 | |
| mips | 2207744 | 221184 | 76768 | 2505696 | |
| 68k | 149800 | 8248 | 229504 | 387552 | pgp |
| x86 | 163840 | 8192 | 227472 | 399504 | |
| hp700 | 188013 | 15320 | 228676 | 432009 | |
| mips | 188416 | 40960 | 230144 | 459520 | |
| alpha | 253952 | 57344 | 222240 | 533536 | |

- CISC text generally more compact, but not by a huge amount
- Alpha's 64bit data/bss is larger

# Code Density

- Important if:
  - Memory is expensive
    - can be in embedded applications
    - eg. mobile phones
    
    ⇒ ARM Thumb, MIPS-16
  - Executable loaded over slow network
    - Though Java not particularly dense!
- Speed vs. size optimization tradeoffs
  - loop unrolling
  - function inlining
  - brunch/jump target alignment

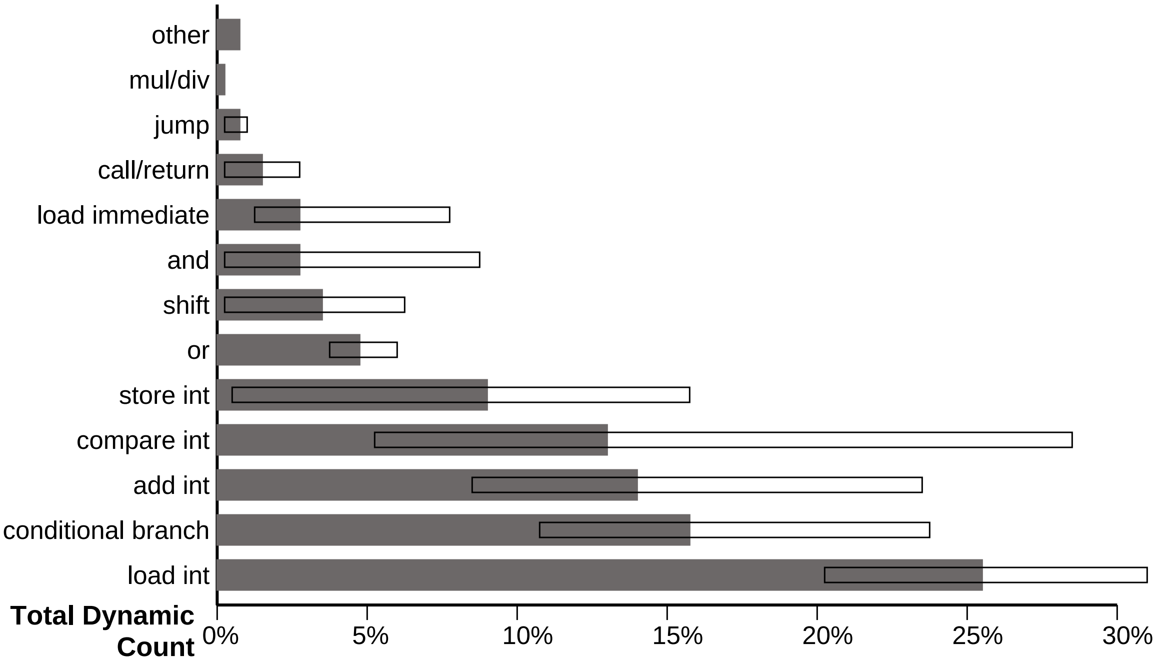
# Instruction caches



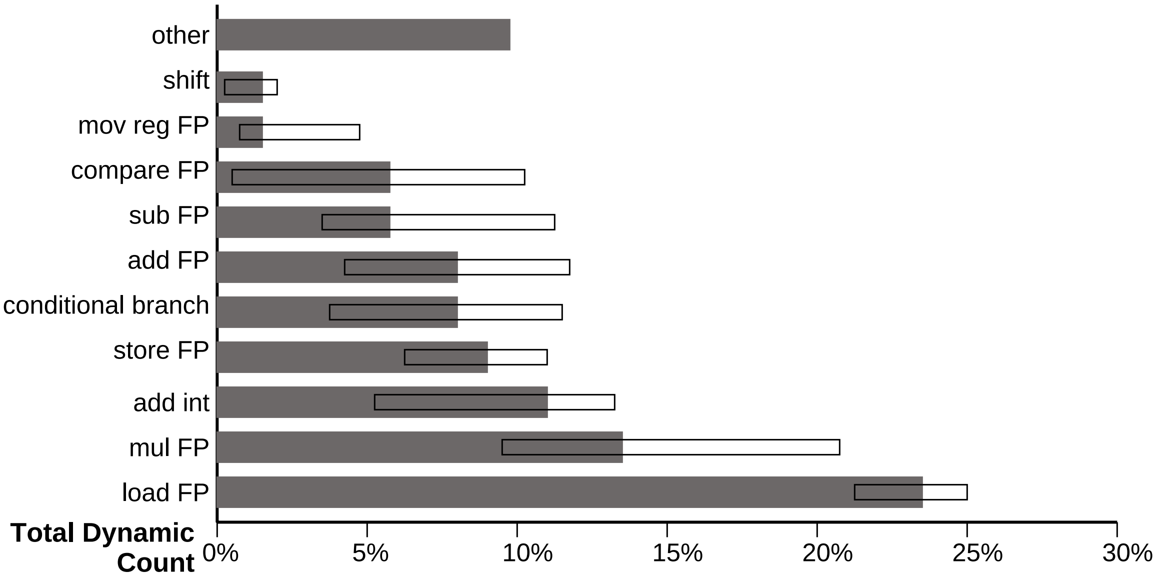

Fraction of program responsible for 80% and 90% of instruction executions

- Caches generally solve I-stream b/w requirements
  - 4bytes x 1GHz x 2-4 instrs = 8-16GB/s !
  - Loops are common! (90% in 10%)
  - Internal I-caches often get 95%+ hit-rates
  - Code density not usually a performance issue
    - ∗ assuming decent compilers and app design
    - ∗ code out-lining (trace straightening) vs. function in-lining and loop unrolling
- D-Cache generally much more of a problem

# Instruction Mix




Instruction mix for SPEC INT92

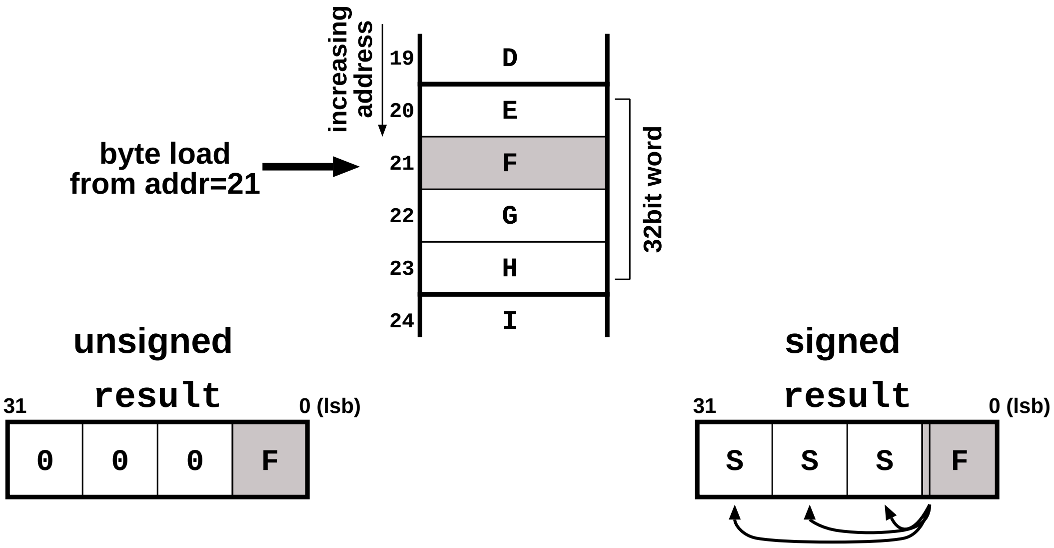



Instruction mix for SPEC FP92

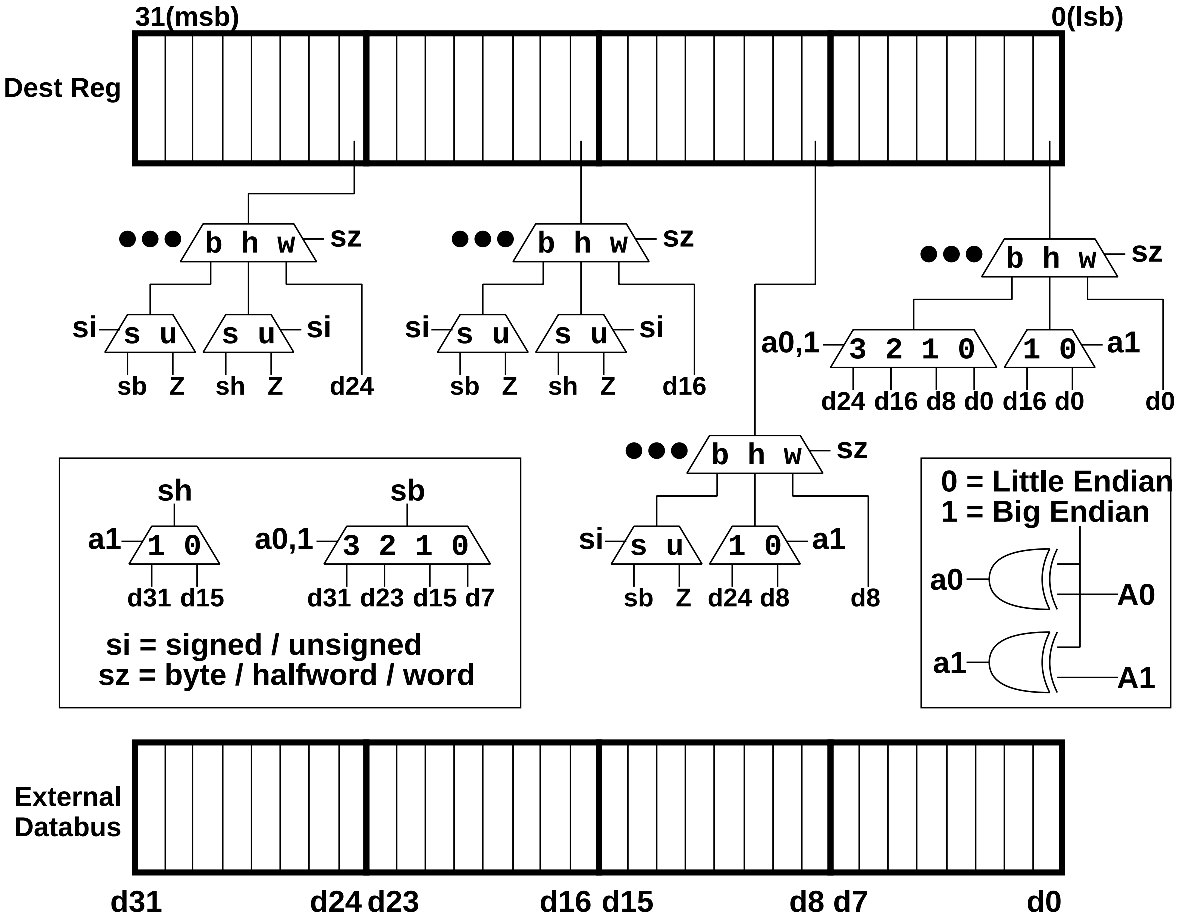
**There are no 'typical' programs**

# Aligned Loads and Stores

- Address mod sizeof(type) = 0
- Most ISA support 8,16,32,(64) bit loads and stores in hardware
- Signed and unsigned stores same
- Sub-word loads can be Signed and Unsigned
  - CISC: loads merge into dest reg
  - RISC: loads extend into dest reg E.g:

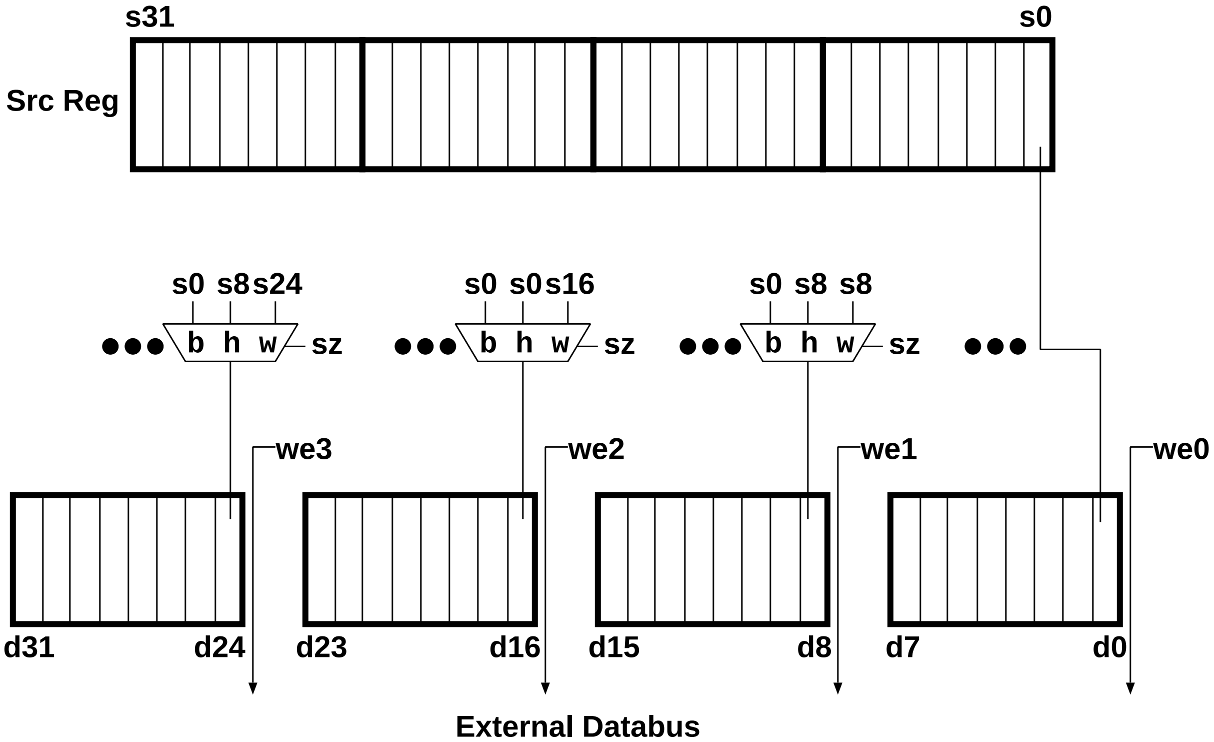


# Aligned Sub-word Load Logic




- byte-lane steering
- sign/zero extension
- Big/Little endian modes

# Aligned Sub-word Store Logic

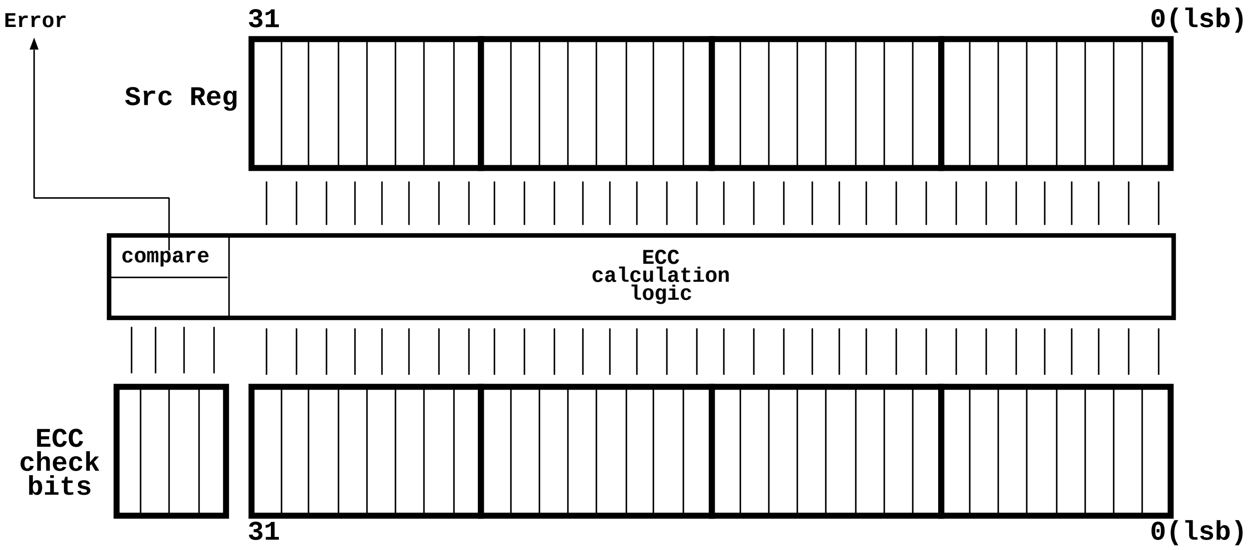



| a1 | a0 | sz | we3 | we2 | we1 | we0 |
|---|---|---|---|---|---|---|
| 0 | 0 | w | 1 | 1 | 1 | 1 |
| 0 | 0 | h | 0 | 0 | 1 | 1 |
| 1 | 0 | h | 1 | 1 | 0 | 0 |
| 0 | 0 | b | 0 | 0 | 0 | 1 |
| 0 | 1 | b | 0 | 0 | 1 | 0 |
| 1 | 0 | b | 0 | 1 | 0 | 0 |
| 1 | 1 | b | 1 | 0 | 0 | 0 |

- Replicate bytes/halfwords across bus
- Write enable lines tell memory system which byte lanes to latch

# Sub-Word Load/Stores

- Word addressed machines
  - Addr bit A0 addresses words
- Alpha (v1):
  - Byte addressed, but 32/64 load/stores only
  - Often critical path
  - Sub-word stores hard with ECC memory
  - So, emulate in s/w using special instructions for efficiency



# Emulating Byte Loads

1. Align pointer
2. Do word load
3. Shift into low byte
4. Mask
5. (sign extend)

- e.g. 32bit, Little Endian, unsigned

```
unsigned int temp;
temp = *(p&(~3));
temp = temp >> ((p&3) *8);
reg  = temp & 255;
```

- e.g. 32bit, Big Endian, unsigned

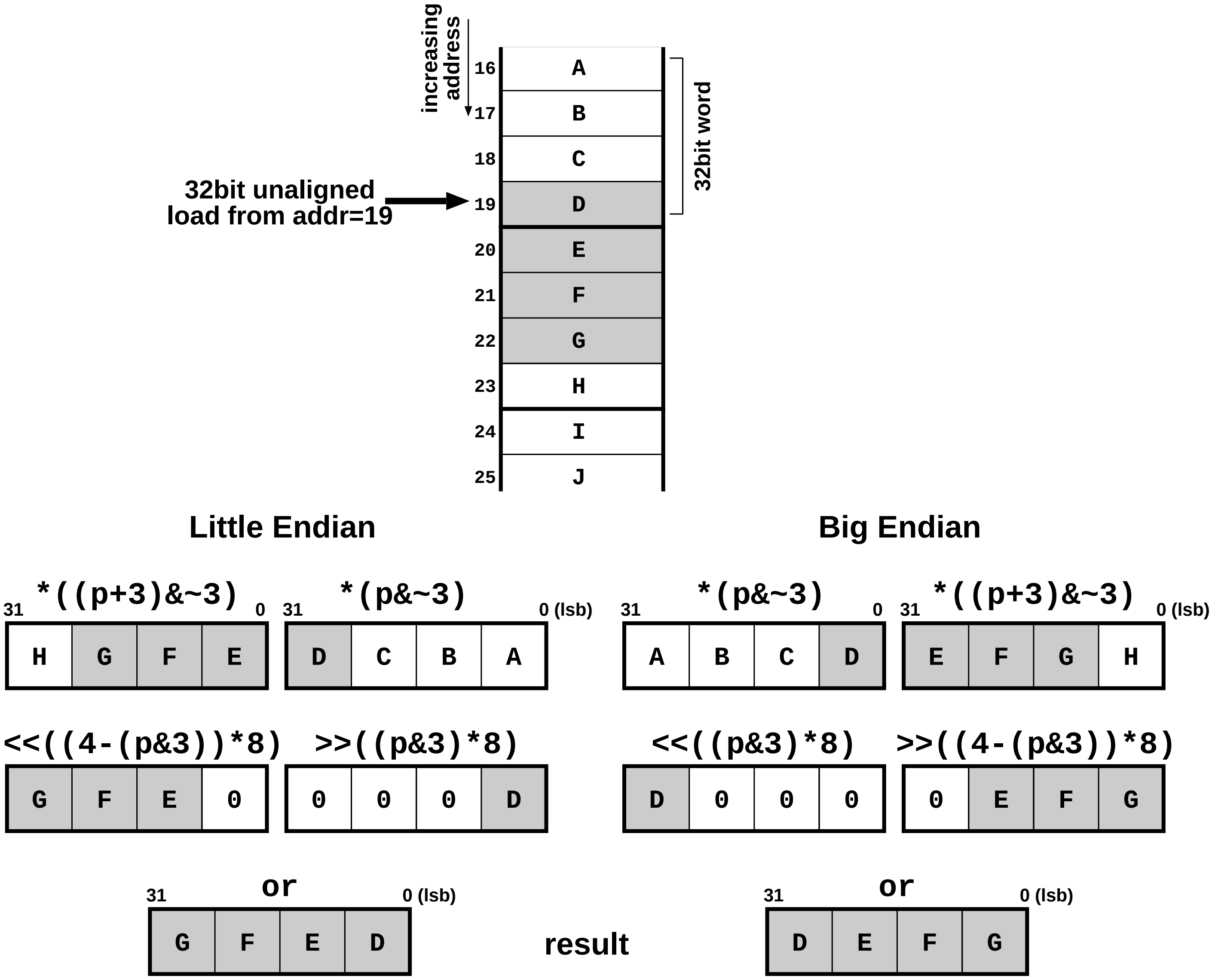
```
unsigned int temp;
temp = *(p&(~3));
temp = temp >> ( (3-(p&3)) * 8);
reg  = temp & 255;
```

- e.g. 64bit, Little Endian, signed

```
long temp;
temp = *(p&(~7));
temp = temp << ( (7-(p&7)) * 8);
reg  = temp >> 56;
```

# Unaligned Accesses

- Address mod sizeof(value) $\neq$ 0
- E.g. :

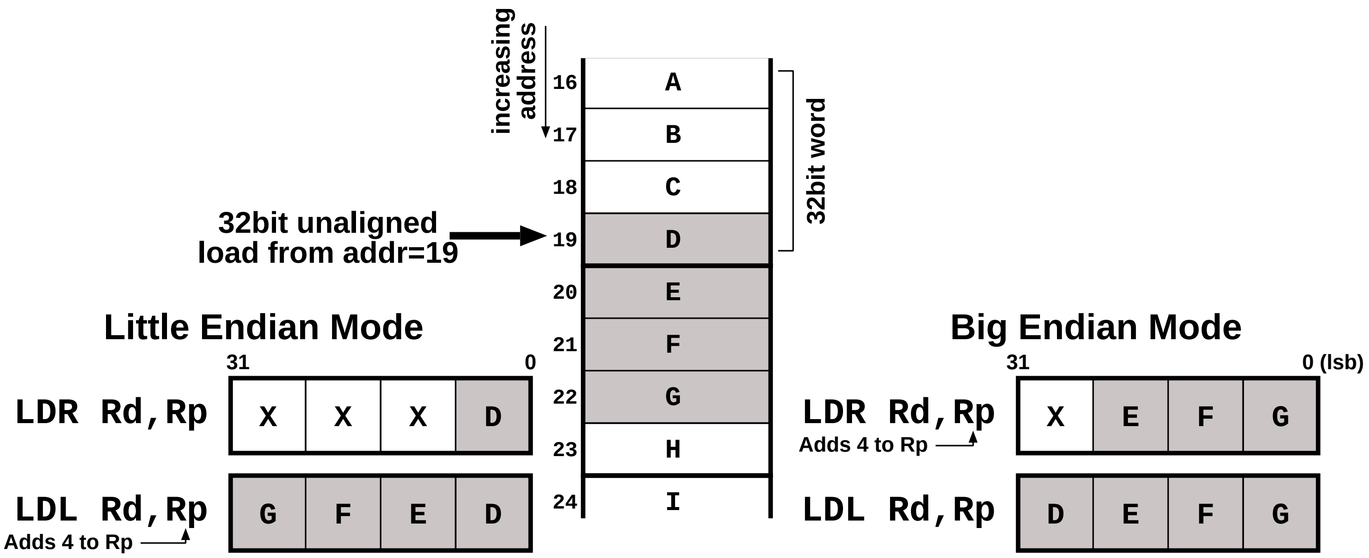


# Unaligned Accesses

- CISC and Power PC support unaligned accesses in hardware
  - Two memory accesses
    - $\rightarrow$ Less efficient
  - May cross page boundaries
- Most RISCs synthesize in software
  - Provide special instructions
- Compilers try to keep data aligned
  - `struct` element padding
- Casting `char *` to `int *` dangerous

# MIPS Unaligned Support

- `LWR` Load Word Right
- `LWL` Load Word Left
  - Only one memory access per instruction
  - Does shifting and merging as well as load
  - → Unaligned load in 2 instrs




- `STR` Store Word Right
- `STL` Store Word Left
- Uses byte store hardware to merge into memory/cache

# Alpha Unaligned Loads

- `LDQ` trap if not 8byte aligned
- `LDQ_U` ignore a0-a2
- `EXTQL Rd ← Rs, Rp`
  Shift Rs right by Rp&7 bytes and extracts quad word into Rd.
- `EXTQH Rd ← Rs, Rp`
  Shift Rs left by 8-Rp&7 bytes and extracts quad word into Rd.
- Alpha requires 5 instrs for arbitrary unaligned load

```
LDQ_U   Rd ← Rp
LDQ_U   Re ← Rp + #7
EXTQL   Rd ← Rd, Rp
EXTQH   Re ← Re, Rp
OR      Rd ← Rd, Re
```

- `EXTBL Rd ← Rs, Rp`
  Shift Rs right by Rp&7 bytes and extracts low byte into Rd.
- also `EXTLL, EXTLH, EXTWL, EXTWH`
- If alignment of pointer is known, may use optimized sequence
  E.g. load 4bytes from address 0x123

```
LDQ Rd ← -3(Rp)
EXTLL Rd ← Rd, #3
```

# Alpha unaligned stores

- No byte hardware, so load quad words, merge, and store back
- `INSQL Rd ← Rs, Rp`
  Shift Rs left by Rp&7 bytes
- `INSQH Rd ← Rs, Rp`
  Shift Rs right by 8-Rp&7 bytes
- `MSKQL Rd ← Rs, Rp`
  Zero top 8-Rp&7 bytes
- `MSKQH Rd ← Rs, Rp`
  Zero bottom Rp&7 bytes
- E.g.: Store quad word Rv to unaligned address Rp

```
LDQ_U   R1 ← Rp          Load both quad words
LDQ_U   R2 ← Rp + #7
INSQH   R4 ← Rv, Rp      Slice & Dice Rv
INSQL   R3 ← Rv, Rp
MSKQH   R2 ← R2, Rp      Zero bytes to be replaced
MSKQL   R1 ← R1, Rp
OR      R2 ← R2, R4      Merge
OR      R1 ← R1, R3
STQ_U   R2 → Rp + #7     Store back
STQ_U   R1 → Rp          Order important:aligned case
```

# Copying Memory

- Often important:
  - OS: user args, IPC, TCP/IP
  - user: realloc, pass-by-value
- `memmove`
  - Must deal correctly with overlapping areas
- `memcpy`
  - Undefined if areas overlap
  - Enables fixed direction
- `copy_aligned`
  - Source and Dest `long` aligned
  - Fastest
- Small copies ($<$ 100 bytes)
  - Avoid large start-up costs
- Medium sized copies (100–100KB bytes)
  - Use highest throughput method
- Large copies
  - Probably memory b/w limited anyway...

# copy_aligned

- E.g. for 32bit machine

```
void copy_aligned( int32 *d, const int32 *s, int n)
{
  sub n, n, #4
  blt n, return    ; if n<0 exit
loop:
  ldw tmp, (s)
  add d, d, #4
  sub n, n, #4     ; set branch value early
  add s, s, #4
  stw tmp, -4(d)   ; maximise load-to-use
  bgt n, loop      ; if n>0 branch (no delay slot)
}
```

- Use widest datapath
  - (64bit FP regs on PPro)
- Maximize cycles before tmp is used
- Update n well in advance of branch
- To further optimize:
  - Unroll loop to reduce loop overhead
  - Instruction scheduling of unrolled loop
  - (software pipelining)

# copy_aligned (2)

```
void copy_8_aligned( int32 d[], const int32 s[], int n)
{
    int32 t0,t1,t2,t3,t4,t5,t6,t7;
top:
    t0 = s[0];   t1 = s[1];
    t2 = s[2];   t3 = s[3];
    t4 = s[4];   t5 = s[5];
    t6 = s[6];   t7 = s[7];
    n  = n - 32; s  = s + 32;
    d[0] = t0;   d[1] = t1;
    d[2] = t2;   d[3] = t3;
    d[4] = t4;   d[5] = t5;
    d[6] = t6;   d[7] = t7;
    d  = d + 32; if (n) goto top;
}
```

- Need to deal with boundary conditions
  - e.g. if $n$ mod 32 != 0
- Get cache line fetch started early
  - Issue a load for the next cache line
    - OK if non-blocking cache
    - beware exceptions (array bounds)

  ⇒ prefetch or speculative load & check

  ⇒ non-temporal cache hints
- IA-64: 'Rotating register files' to assist software pipelining without the need to unroll loops

# Unaligned copy

- E.g. 32bit, Little Endian

```
void memcpy( char *d, const char *s, int n)
{
  uint32 l,h,k,*s1,*d1;

  /* Align dest to word boundary */
  while ( ((ulong)d&3) && n>0 ) {*d++ = *s++; n--;}

  /* Do main work copying to aligned dest */
  if( ((ulong)s & 3) == 0 ) {       /* src aligned ? */
     k = n & ~3;                    /* round n down  */
     copy_aligned(d, s, k);
     d+=k; s+=k; n&=3;              /* ready for end */
  }
  else
  {
    s1 = (uint32 *)((ulong)s & ~3); /* round s down  */
    d1 = (uint32 *) d;              /* d is aligned  */
    h  = *s1++;                     /* init h        */
    k  = (ulong)s &3;               /* src alignment */
    for(; n>=4; n-=4) {             /* stop if n<4   */
      l = *s1++;
      *d1++ = ( h >> (k*8)     ) |
              ( l << ((4-k)*8) ) ;
      h = l;
    }
    d = (char *) d1;                /* ready for end */
    s = ((char *)s1) - 4 + k;
  }

  /* Finish off if last 0-3 bytes if necessary */
  for( ; n>0; n-- ) *d++ = *s++;
}
```

# Memory Translation and Protection

- Protection essential, even for embedded systems
  - isolation, debugging
- Translation very useful
  - demand paging, CoW, avoids relocation
- Segmentation *vs.* Paging
  - x86 still provides segmentation support
  - descriptor tables: membase, limit
  - segment selectors : cs, ds, ss, fs, gs
- Page protection preferred in contemporary OSes
- Translation Lookaside Buffer (TLB)
  - translate Virtual Frame Number to PFN
  - check user/supervisor access
  - check page present (valid)
  - check page writeable (DTLB)
- Separate D-TLB and I-TLB
  - often a fully associative CAM
  - separate I-TLB and D-TLB
  - typically 32-128 entries
  - sometimes an L2 Joint-TLB e.g. 512 entry
- Hardware managed *vs.* software managed TLB

# Hardware page table walking

- Hierarchical lookup table
- E.g. x86/x86_64 4KB pages evolved over time:
  - 2-level : 4GB virt, 4GB phys (4B PTEs)
  - 3-level : [512GB] virt, 64GB phys (8B PTEs)
  - 4-level : 256TB virt, 1TB phys (8B PTEs)
    (48 bit VAs are sign extended to 64bit)
- 'set PT base' instruction
  - implicit TLB flush (on x86)
- Flush virtual address
- Global pages not flushed
  - special bit in PTE
  - should be same in every page table!
  - typically used for kernel's address space
  - special TLB flush all
- Superpages are PTE 'leaves' placed in higher levels of the page table structure
  - e.g. 4MB pages on x86 2-level

# Software managed TLB

- OS can use whatever page table format it likes
  - e.g. multilevel, hashed, guarded, etc.
  - (generally more compact than hierarchical)
  - use privileged 'untranslated' addressing mode
- Install TLB Entry instruction
  - specify tag and PTE
  - replacement policy usually determined by h/w
    - e.g. not most recently used
- (may allow TLB contents to be read out for performance profiling)
- Flush all, flush ASN, flush specified VA
- Flexible superpage mappings often allowed of e.g. 8, 64, 512 pages.
- Notion of current Address Space Number (ASN)
- TLB entries tagged with ASN
- Try to assign each process a different ASN
  - no need to flush TLB on process switch
  - (only need to flush when recycling ASNs)
- IA-64 : s/w TLB with hardware PT walking assist
- PPC: h/w fill from larger s/w managed hash table

# ISA Summary

- RISC
  - Product of quantitative analysis
  - Amdahl's Law
  - Load-Store GPRs
  - ALU operates on words
  - Relatively simple instructions
  - Simple addressing modes
  - Limited unaligned access support
  - (s/w managed TLB)
- Architecture extensions
  - Backwards compatibility
- Copying memory efficiently

# Does Architecture matter?

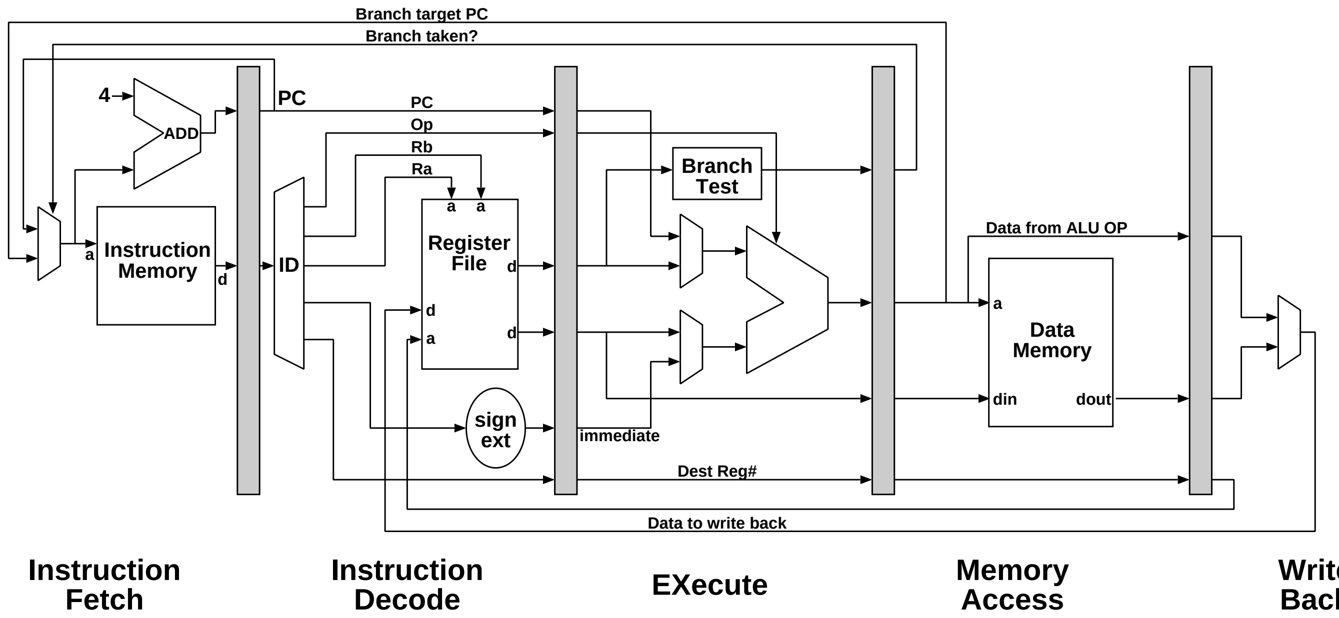
# CPU Performance Equation

$$Time\ for\ task = C * T * I$$

| | |
|---|---|
| C | =Average # Cycles per instruction |
| T | =Time per cycle |
| I | =Instructions per task |

- Pipelining
  - e.g. 3-5 pipeline steps (ARM, SA, R3000)
  - Attempt to get C down to 1
  - Problem: stalls due to control/data hazards
- Super-Pipelining
  - e.g. 8+ pipeline steps (R4000)
  - Attempt to decrease T
  - Problem: stalls harder to avoid
- Super-Scalar
  - Issue multiple instructions per clock
  - Attempt to get C below 1
  - Problem: finding parallelism to exploit
    - typically Instruction Level Parallelism (ILP)

# The classic RISC pipe




| | |
|---|---|
| IF | Send out PC to I-cache. Read instruction into IR. Increment PC. |
| ID | Decode instruction and read registers in parallel (possible because of fixed instruction format). Sign extend any immediate value. |
| EX | Calculate Effective Address for Load/Stores. Perform ALU op for data processing instructions. Calculate branch address. Evaluate condition to decide whether branch taken. |
| MA | Access memory if load/store. |
| WB | Write back load data or ALU result to register file. |

# The cost of pipelining

- Pipeline latches add to cycle time
- Cycle time determined by slowest stage
  - Try to balance each stage
- Some resources need to be duplicated to avoid some Structural Hazards
  - (PC incrementer)
  - Multiple register ports (2R/1W)
  - Separate I&D caches

$\Rightarrow$ Effectiveness determined by CPI achieved

# Pipelining is efficient

# Non Load-Store Architectures

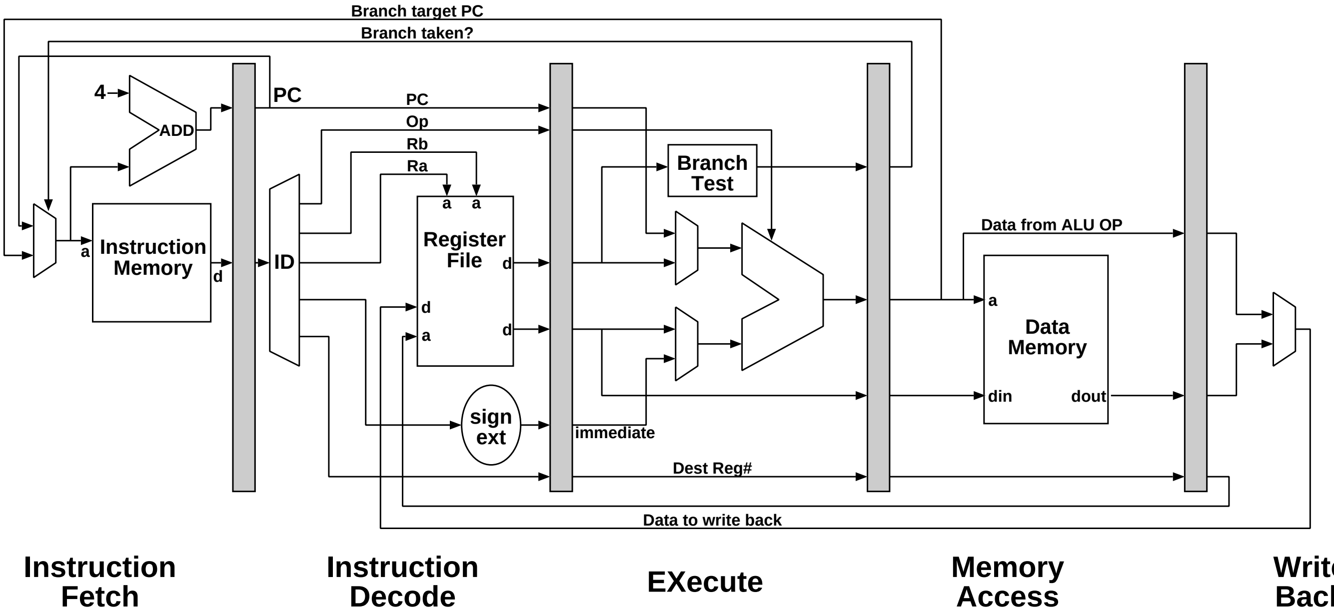
- Long pipe with multiple add and memory access stages
  - Lots of logic
  - Many stages unused by most instructions
- Or, multiple passes per instruction
  - Tricky control logic
- Or, convert architectural instructions into multiple RISC-like internal operations
  - Good for multi-issue
  - More ID stages
  - Pentium Pro/II/III ($\mu$ops)
  - AMD x86 K7 (r-ops)

**Pipelining easiest if all instructions do a similar amount of 'work'**

# ALU Result Forwarding

- E.g. 4 forwarding paths to avoid stalls:

```
a:  add r1 ← r8, r9
b:  add r2 ← r1, r7
c:  add r3 ← r1, r2
d:  add r4 ← r1, r2
```




- Read after Write
- Doubled # mux inputs
- Deeper pipes → more forwarding
  - R4000, 8 deep pipe
    forward to next 4 instructions

# Load Data Hazards

- Impossible without a stall:

  ```
  lw r1 ← r9(4)
  add r2 ← r1, r6
  ```

- Read after Write (RaW) hazard
- New forwarding path saves a cycle
- Re-order code to avoid stall cycle
  - Possible for 60-90% of loads
- Software Interlocked
  - Compiler must insert `nop`
  - e.g. R2000/R3000
- Hardware Interlocked
  - Save `nop`: better for I-stream density
  - Register scoreboard
    - track location of reg values e.g.:
    - File, EX, MA, MUL1, MUL2, MemVoid
    - hold back issue until RaW hazard resolved
    - control operand routeing
  - Required for all sophisticated pipelines
- More stalls for deeper pipes
  - 2 stalls and 2 more forwarding paths for R4000

# Longer Latency Instructions

- Mul/Div, Floating Point
- Different functional units operating in parallel with main pipeline
- Extra problems:
  - Structural hazards
    - Unit may not be fully pipelined, eg:
      - 21264 FDiv: 16cy latency, not pipelined
      - 21164 FMul: 8cy latency, issue every 4cy
      - → 21264 FMul: 4cy latency, fully pipelined
    - Multiple Write Back stages
      - more register write ports?
      - or, schedule pipeline bubble
  - Read after Write hazards more likely
    - compiler instruction scheduling
  - Instruction complete out of order
    - Write after Write hazards possible
    - Dealing with interrupts/exceptions
- Use scoreboard to determine when safe to issue
- Often hard to insert stalls after ID stage
  - synthesize NOPs in ID to create bubble
  - ‘replay trap’ : junk & refetch

# Exceptions and Pipelining

| | | |
|---|---|---|
| User SWI/trap | ID | Precise (easy) |
| Illegal Instruction | ID | Precise (easy) |
| MMU TLB miss | IF/MA | Precise required |
| Unaligned Access | MA | Precise required |
| Arithmetic | EX 1..N | Imprecise possible |

- Exceptions detected past the point of in-order execution can be tricky
  - FP overflow
  - Int overflow from Mul/Div
- Exact arithmetic exceptions
  - Appears to stop on faulting instruction
  - Need exact PC
    - care with branch delay slots
  - Roll back state/In-order commit (PPro)
- Imprecise arithmetic exceptions
  - Exception raised many cycles later
  - Alpha: Trap Barriers
  - PPC: Serialise mode
  - IA-64: Poison (NaT) bits on registers
    - instructions propagate poison
    - explicit collection with 'branch if poison'

# Interrupts

- Interrupts are asynchronous
- Need bounded latency
  - Real-time applications
  - Shadow registers avoid spilling state
    - Alpha, ARM
- Some CISC instructions may need to be interruptible
  - Resume *vs.* Restart
    - eg. overlapping memcpy
  - Update operands to reflect progress
    - VAX MOVC

# Control Flow Instructions

- Absolute *jumps*
  - To an absolute address
    (usually calculated by linker)
  - Immediate / Register modes
  - usage: function pointers, procedure call/return into other compilation modules, shared libraries, switch/case statements

- PC Relative *branches*
  - Signed immediate offset
  - Limited range on RISC
    * Typically same compilation module
      (calculated by compiler)
  - Conditional

- Branch/Jump to Subroutine
  - Save PC of following instruction into:
    * CISC: stack
    * most RISC: special register
    * ALPHA: nominated register
    * IA-64: nominated Branch Reg

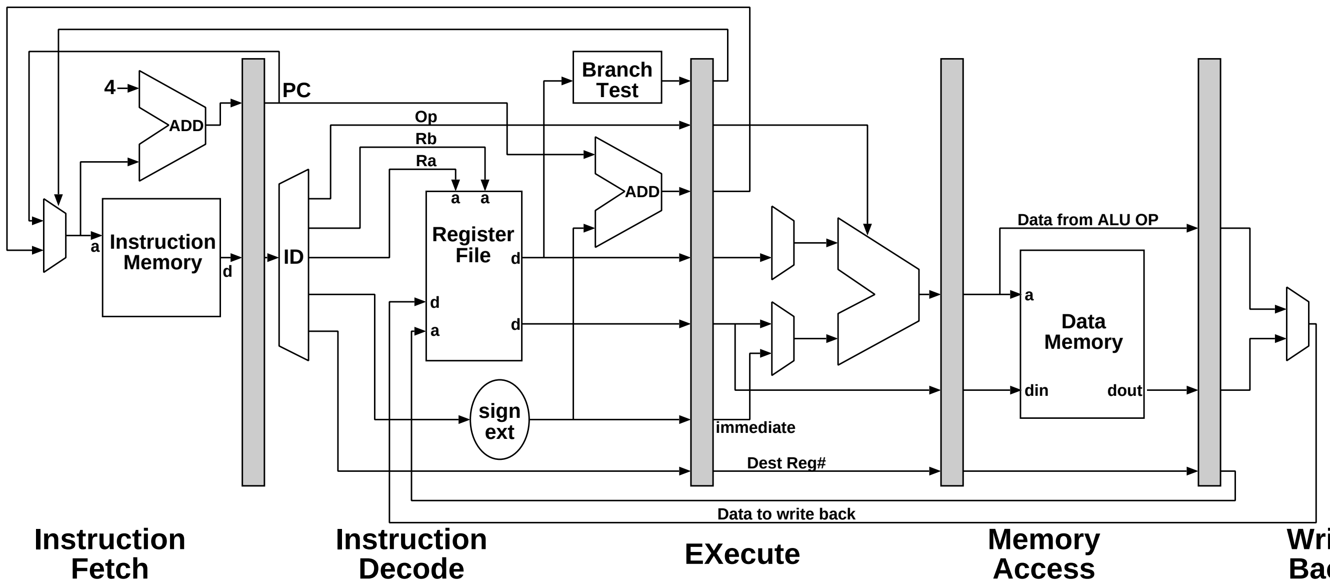
# Conditional Branches

- Conditional branch types
  - most: Test condition code flags
    - Z, C, N, V
    - `Bxx label`
  - Alpha/MIPS: Test bits in named reg
    - msb (sign), lsb, 'zero'
    - `Bxx Ra, label`
  - some: Compare two registers and branch
    - `Bxx Ra, Rb, label`
    - (PA-RISC, MIPS, some CISC)
  - IA-64: Test one of 64 single bit predicate regs

- Conditional branch stats (for MIPS and SPEC92)
  - 15% of executed instructions
    - 73% forward (if/else)
    - 27% backward (loops)
  - 67% of branches are taken
    - 60% forward taken
    - 85% backward taken (loops)

# Control Hazards

- ‘classic’ evaluates conditional branches in EX
  - Identify branch in ID, and stall until outcome known
  - Or better, assume not taken and abort if wrong
  - → 2 stall taken-branch penalty
- If evaluating branch is simple, replicate h/w to allow early decision
  - Branch on condition code
  - Alpha/MIPS: Test bits in named reg
    - ∗ Special ‘zero’ bit stored with each reg
  - Hard if `Bxx Ra, Rb, label`



# Control Hazards (2)

- Evaluate branches in ID (when possible)

⇒ Only 1 cycle stall if test value ready
(Set flags/reg well before branch)

  - Bad if every instruction sets flags (CISC)
  - Helps if setting CC optional (SPARC/ARM)
  - Good if multiple CC regs (PPC/IA-64), or none (Alpha/MIPS)

- Branch delay slots avoided the taken branch stall on early MIPS
  - Always execute following instruction
  - Can't be another branch
  - Compiler fills slot $\sim$60% of the time
  - Branches with optional slots: avoid `nop`

- Modern CPUs typically have more stages before EX, due to complicated issue-control logic, thus implying a greater taken-branch cost

- Stalls hurt more on a multi-issue machine. Also, fewer cycles between branch instructions

**Control hazards can cripple multi-issue CPUs**

# Static Branch Prediction

- Speculation should not change semantics!
- Simple prediction
  - e.g. predict backward as taken, forward not
- Branch instructions with hints
  - Branch likely/unlikely
    - strong/weak hint varients
  - Use Feedback Directed Optimization (FDO)
  - Fetch I-stream based on hint
- Delayed branch instrs with hints and annulment
  - If hint is correct execute following instruction else don't
  - e.g. new MIPS, PA-RISC
  - Compiler able to fill delay slot more easily

# Dynamic Branch Prediction

- Static hints help, but need to do better

- Branch prediction caches
  - Indexed by significant low order bits of branch instruction address
  - Cache entries do not need tags (they're only hints)
  - E.g. 512-8K entries

- Bi-modal prediction method

  $\Rightarrow$ many branches are strongly biased
  - Single bit predictor
    * Bit predicts branch as taken/not taken
    * Update bit with actual behaviour
    * Gets first and last iterations of loops wrong
  - Two bit predictors
    * Counter saturates at 0 and 3
    * Predict taken if 2 or 3
    * Add 1 if taken, subtract 1 if not
    * Little improvement above two bits
    * $\geq$90% for 4K entry buffer on SPEC92

# Local History predictors

- Able to spot repetitive patterns
- Copes well with minor deviations from pattern
- E.g. 4 bit local history branch predictor
  - 4 bit shift reg stores branch's prior behaviour
  - 16 x 2 bit bi-modal predictors per entry
  - use shift reg to select predictor to use
  - perfectly predicts all patterns $<$ length 6, as well as some longer ones (up to length 16)
  - used on Pentium Pro / Pentium II
    - 512 entries x (16 x 2 + 4) = 18K bits SRAM
  - trained after two sequence reps
  - other seqs up to 6% worse than random
- An alternative approach is to use two arrays. One holds branch history, the other is a shared array of counters indexed by branch history
  - branches with same pattern share entries in 2nd array (more efficient)
  - 21264 LH predictor: 1024 entries x 10 bits of history per branch, and shared array of 1024 counters indexed by history

# Global Correlating predictors

- Behaviour of some branches is best predicted by observing behaviour of other branches

- (Spatial locality)

⇒ Keep a short history of the direction that the last few branch instructions executed have taken

- E.g. Two bit correlating predictor:
  - 2 bit shift register to hold *processor* branch history
  - 4 bi-modal counters in each cache entry, one for each possible global history

- Rather than using branch address, some GC predictors use the processor history as the *index* into a single bi-modal counter array. Also possible to use a hash of (branch address, global history)
  - Alpha 21264 GC predictor uses a 12 bit history and 4096 x 2 bit counters

- Combination of Local History and Global Correlating predictor works well
  - $\geq$95% for 30K bit table on SPEC92
  - E.g. Alpha 21264

# Reducing Taken-Branch Penalty

- Branch predictors usually accessed in ID stage, hence at least one bubble required for taken-branches
- Need other mechanisms to try and maintain a full stream of useful instructions:
- Branch target buffers
  - In parallel with IF, look up PC in BTB
  - if PC is present in BTB, start fetching from the address indicated in the entry
  - Some BTBs actually cache instructions from the target address
- Next-fetch predictors
  - Very simple, early, prediction to avoid fetch bubbles, used on PPro, A21264
  - I-cache lines have pointer to the next line to fetch
  - Update I-cache ptr. based on actual outcome
- Trace caches (Pentium IV)
  - Replace traditional I-cache
  - Cache commonly executed instr sequences, crossing basic block boundaires
  - (c.f. "trace straightening" s/w optimization)
  - Table to map instr address to position in cache
  - Instrs typically stored in decoded form

# Avoiding branches

- Loop Count Register (PowerPC, x86, IA-64)
  - Decrement and branch instruction
  - Only available for innermost loops

- Predicated Execution (ARM, IA-64)
  - Execution of all instructions is conditional
    * ARM: on flags registers
    * IA-64: nominated predicate bit (of 64)
  - IA-64: cmp instructions nominate two predicate bits, one is set and cleared depending on outcome
  - E.g. if( [r1] && [r2] && [r3] ) {...} else {...}

```
ld r4 <- [r1]
p6,p0 <= cmp( true )
p1,p2 <= cmp( r4==true )
<p1> ld r5 <- [r2]
<p1> p3,p4 <= cmp( r5==true )
<p3> ld r6 <- [r3]
<p3> p5,p6 <= cmp( r6==true )
<p6> br else
...
```

  ✔ Transform control dependency into data dep

  ✔ Instruction ‘boosting’
    * e.g. hoist a store ahead of a branch

  ✔ Inline simple if/else statements

  ✘ Costs opcode bits

  ✘ Issue slots wasted executing nullified instrs

# Avoiding branches 2

- Conditional Moves (Alpha, new on MIPS and x86)
  - Move if flags/nominated reg set
  - Just provides a 'commit' operation
    - beware side effects on 'wrong' path
  - PA-RISC supports arbitrary nullification

- Parallel Compares (IA-64)
  - Eliminate branches in complex predicates
  - Evaluate in parallel
    - (despite predicate dependancy)
  - if ( (rA<0) && (rB==-15) && (rC>0) ) {...}

```
cmp.eq p1,p0 = r0, r0 ;; // p1 =1
cmp.ge.and p1,p0 = rA,r0
cmp.ne.and p1,p0 = rB,-15
cmp.le.and p1,p2 = rB,10
(p1) br.cond if-block
```

## Avoid hard to predict branches

# Optimizing Jumps

- Alpha: Jumps have static target address hint
  - $A_{16-2}$ of target instruction virtual address
  - Enough for speculative I-cache fetch
  - Filled in by linker or dynamic optimizer
- Subroutine Call Returns
  - Return address stack
  - Alpha: Push/pop hints for jumps
  - 8 entry stack gives $\geq$ 95% for SPEC92
- Jump target registers (PowerPC/IA64)
  - Make likely jump destinations explicit
  - Buffer instructions from each target
- Next-fetch predictors / BTBs / trace caches help for jumps too
  - Learn last target address of jumps
  - Good for shared library linkage

# Super-Scalar CPUs

- # execution units (INT/FP)
  - Pentium (2/1), P6 (2+/2), P7 (4/2)
  - 21164 (2/2), 21264 (4/2)
- Units often specialised e.g 21264:
  - Int ALU + multiply
  - Int ALU + shifter
  - 2x Int ALU + load/store
  - FP add + divide + sqrt
  - FP multiply
- Max issue rate
  - Pentium (2), P6&P7 (3$\mu$ops)
  - 21164 (4), 21264 (4)
  - Ideal instruction sequence
    - Right combination of INT and FP
  - Lower than number of exec units
- Two basic types
  - Static in-order issue (21164, Pentium, Merced)
  - Dynamic out-of-order execution (21264, PPro)

# Static Scheduled

- All instructions begin execution in-order

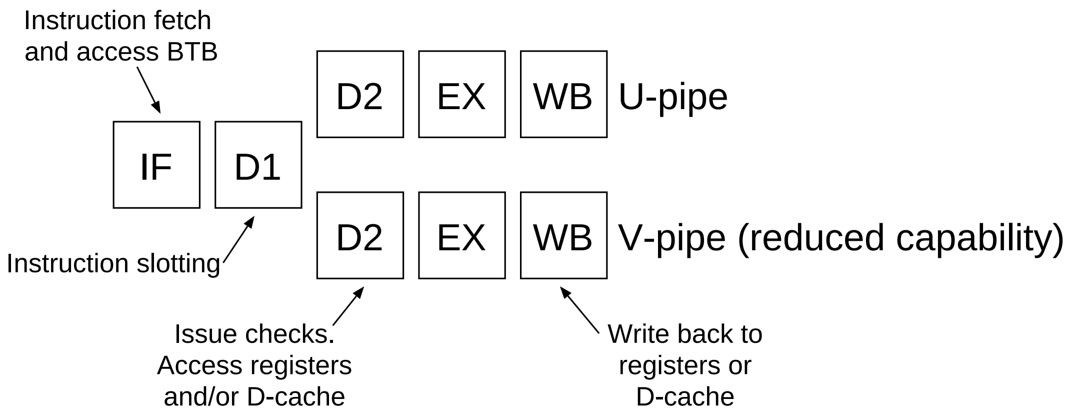
1. Fetch and decode a block of instructions

2. 'Slot' instructions to pipes based on function unit capability and current availability

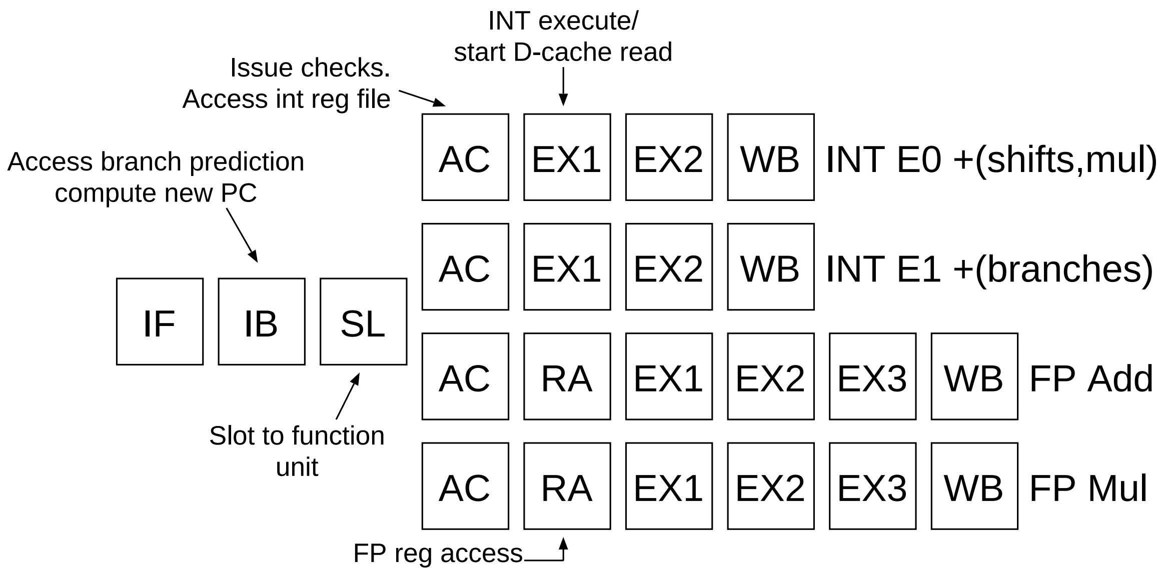
3. Issue checks:
   - Data Hazards
     - Are the instructions independent?
     - Check register scoreboard
       * Are the source operands ready?
       * Will write order be preserved?
     - Non-blocking missed loads
       * Do not stall until value is used
     - *Maintain in-order dispatch*
   - Control hazards
     - Is one of the instructions a predicted-taken branch?
       * Discard instructions past branch
     - Be prepared to squash speculated instrs

4. move onto next block when all issued

# Static Scheduled Examples




Pentium pipeline




Alpha 21164 pipeline

# Static Scheduled Super-Scalar

- Relies greatly on compiler
  - Instruction scheduling
    - slotting
    - data-dependence
  - Issue loads early (or prefetch)
  - Reduce # branches and jumps
    - unroll loops
    - function inlining
    - use of CMOV/predication
  - Align branches and targets
    - avoid wasted issue slots

- Optimization can be quite implementation dependent

- Static analysis is imperfect
  - basic blocks can be reached from multiple sources
  - compiler doesn't know which loads will miss
  - Feedback Directed Optimization can help

$\Rightarrow$ On most code, actual issue rate will be $<<$ max

# Helping the compiler

- Wish to issue loads as early as possible, but
  - mustn't overtake a store/load to same address
  - Stack / Global variables solvable
    - `[r12,4] != [r12,16]`
  - Heap refs harder to disambiguate
    - `[r2,8] != [r4,32] ???`
    - `C/C++` particularly bad in this respect

⇒ Data speculation (IA-64, Transmeta)

  - allows loads to be moved ahead of possibly conflicting load/stores
  - `ld.a r3 = [r5]` enters address into Address Aliasing Table
  - any other memory reference to same address removes entry
  - `ld.c r3 = [r5]` checks entry is still present else reissues load

- Predication enables load issue to be hoisted ahead of branch, but not above compare

⇒ Control speculation (IA-64)

  - `ld.s r3 = [r5]` execute load before it is known if it should actually be executed
  - `chk.s r3, fixup` check poison bit and branch if load generated an exception

# Dynamic Scheduling

- Don't stop at the first stalled instruction, continue past it to find other non-dependent work for execution units

- Search window into I-stream
  - Data-flow analysis to schedule execution
  - Out-of-order execution
  - In-order retirement to architectural state
  - P6 core $\leq$ 30 $\mu$ops, P7 $\leq$ 126

- Use *speculation* to allow search window to extend across multiple basic blocks
  - (Loops automatically unrolled)
  - Need excellent branch prediction
  - Track instructions so they can be aborted if prediction was wrong
  - Try to make branch result available ASAP (to limit waste caused by mis-prediction)

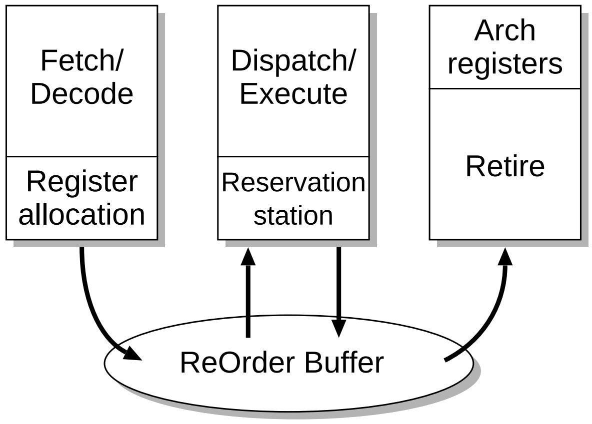
# Register Renaming

- Register reuse causes false dependencies
- (Often referred to as name-dependencies)
  - WaR, WaW: no data transfer
- Undo compiler's register colouring
- Necessary to unroll loops

⇒ Register renaming
  - Large pool of internal physical registers
  - P6 40, 21264 80+72, P7 128
  - New internal register allocated for the result operand of every instruction
  - Re-mapper keeps track of which internal registers instructions should use for their source operands
    * needs to be able to rollback upon exception or mispredict

- Architectural register state updated when instructions retire

# Out-of-Order Execution




1. Fetch and decode instructions

2. Re-map source operands to appropriate internal registers. Allocate a destination register from register free list. Place instruction in a free Re-Order Buffer (ROB) slot.

3. Reservation station scans ROB to find instructions for which all source operands are available, and a suitable execution unit is free (Favour older instructions if multiple ready)

4. Executed instructions and results are returned to the ROB (internal registers which are no longer needed are placed on free list)

5. Retire unit removes completed instructions from ROB in-order, and updates architecturally visible state. Detect exceptions & mis-predicted branches; Roll-back ROB contents and mapping register state, start fetch from new PC

# Loads and Stores

- Dyn Exec helps hide latency of L2/L3 cache
  - Find other work to do in the meantime
  - Allow loads to issue early

- Stores cannot be undone
  $\Rightarrow$ Update memory in Retirement stage
  - Hold in Store Queue until retirement

- Loads that overtake stores must be checked to see if they refer to the same location (alias)
  - Address of store may not yet be known
  $\Rightarrow$ Speculate load and check later:
  - Load Queue stores addresses of issued loads until they retire
  - when a store 'executes' (target address is known) it checks the LQ to see whether a newer load has issued to the same address
  - if so, execution is rolled back to the store instruction (replay load)
    * 21264 has 32 entry LQ and a 1024 entry prediction cache to predict which loads to 'hold back' and thus avoid replay trap

- Loads overtaking loads treated similarly to maintain ordering rules with other CPUs/DMA

# Out-of-order Execution

- Less dependency on compiler than static-sched
- Better at avoiding wasted issue slots
- But, O-o-O execution uses a lot of transistors
  - ReOrder Buffer and Reservation Stations are large structures incorporating lots of Content Addressable Memory
  - Tend to be at least $O(N^2)$ in complexity
  - Tend to be on critical path
    - diminishing returns...
  - 20%+ of chip area on 21264
- Factors effecting usable ILP
  - Window size
  - Number of renamed registers
  - Memory reference alias analysis
  - Branch and jump prediction accuracy
  - Data cache performance
  - (Value speculation performance)
- Simulation suggests the ‘perfect’ processor: 18-150 instructions per cycle for SPEC92
- 10 way for int progs feasible, more for FP
- Some code just exhibits very poor ILP...

# VLIW Architectures

- Very Long Instruction Word (VLIW)
- Each instruction word (or 'packet') contains fields to specify an operation for each function unit
- Compiler instruction scheduling:
  - allocates sub-instructions to function units
  - avoids any resource restrictions
  - ensures producer-consumer latencies satisfied (delay slots)

✔ CPU doesn't need to worry about issue-checks

$\Rightarrow$ High clock speed

✘ Relies heavily on compiler / assembler programmer

- loop unrolling
- trace scheduling

✘ Stall in any function unit causes whole processor to stall

- D-cache misses a big problem

✘ Often sparse I-stream (lots of nops)

✘ Exposes processor internals

- Typically no binary compatibility

# Intel EPIC (VLIW-like)

- Intel: Explicitly Parallel Instruction Computer
  - Merced (Itanium) , McKinley
- Three 41 bit instrs packed into 128 bit 'bundle' with 5 template bits
- Template determines function unit dispatch
  - restricted set of possibilities simplifies instruction dispersement hardware
    - e.g. `[Mem,Int,Branch]`, no `[Int,Int,Int]`
- Stop bits: barriers between independent instructions groups
  - groups can cross multiple bundles
- Compiler collects instrs into independent groups
- Hardware interlock of longer-latency instructions as well as load-use latencies

✔ Reduces issue-check complexity for CPU

✔ Retains binary compatibility

- Need good compilers
  - hope extensive use of load speculation instructions enables hoisting of loads to avoid stalling whole CPU
- Optimization for new implementations important?

# Transmeta 'Code Morphing'

- VLIW core hidden behind x86 emulation
- Uses combination of interpretation, translation and on-line feedback-directed optimization
- Only 'code morphing' s/w written for VLIW
  - Apps, OS and even BIOS are x86
- Keeps an in-memory translation cache
- Translate and optimise along frequently executed paths (trace scheduling)
  - speculative load instrs increase trace length
- Hardware features to assist translation:
  - Shadow registers with commit instruction
    - assist rollback upon x86 exceptions/mispredicts
  - hold-back stores until commit
- Performance counters assist re-optimization

✔ Binary compatibility, High clock speed, Low power

✔ Potential for more complex scheduling than h/w

✘ Overhead of performing translation

✘ Less dynamic than h/w scheduling

# Beyond ILP

- Diminishing returns for further effort extracting ILP from a single thread?
- *System-level* parallelism
  - some workloads naturally parallel
    * multi-user machine
    * application plus XServer
    * application plus asynchronous I/O
- *Process/Thread-level* parallelism
  - Some applications already multithreaded
    * database, HTTP server, NFS server
    * fork, pthreads
  - may have smaller cache footprint
  - may be same Virtual address space
- *Loop-level* parallelism
  - generated by auto-parallelizing compilers
  - co-operative threads
  - need fast synchronization, communication, fork

# Exploiting Parallelism

- Multiple CPUs on a chip
  - Exploit thread/process level parallelism
  - Use traditional SMP mechanisms
  - ✘ Need correspondingly bigger caches and external memory bandwidth
  - IBM Power4 2-way SMP on a chip
- **Multi-threading**
  - Use one CPU to execute multiple threads
  - Replicate PCs, architectural register file
  - Different virtual address spaces?
- Static multi-threading
  - Round-robin issue from a large # threads
  - ✔ No instruction dependencies
  - ✔ Hides memory latency
    - ∗ No expensive caches
  - ✔ Fast synchronization / fork possible
  - ✘ Requires many register files
  - ✘ Progress of an individual thread is slow
    - ∗ Poor SPEC marks (great SPEC Rate)
  - Tera/Cray MTA, 128 threads
- Course-grained multi-threading
  - Switch between threads on a major stall
  - e.g. cache miss on Stanford SPARCLE

# Simultaneous Multi-Threading (SMT)

- Work on a small number of threads at once, aiming to keep all function units busy
- Duplicate architectural state
- Duplicate instruction fetch units
- Need to control allocation of resources
  - priority . fair share
  - (prioritising can be counter productive)

✔ Progress of individual threads is pretty good

✔ Cooperating threads may have smaller cache footprint than independent ones

✔ Potential for register-register synchronization and communication

✔ Potential for lightweight thread create

- Pentium IV Xeon uses 2-way "hyperthreading"
  - 2 virtual CPUs per chip
  - looks like SMP - separate VM contexts
  - Staticly partitions resources if both active
  - SMT `halt` and `pause` instructions
  - OS scheduler should understand SMT

# Other techniques

- Data-flow processors
  - Fine-grained control-flow, course-grained data-flow (opposite of standard super-scalar)
  - Begin execution of a block of sequential instructions when all inputs become available
  - ✘ Inputs are memory locations. The matching store required to figure out when all inputs are ready is large and potentially slow. (matching is easier with a small number of registers *a la* out-of-order execution)

# Caching

- Caches exploit the temporal and spatial locality of access exhibited by most programs

- Cache equation:

  $$Access\ Time_{Avg} = (1 - P) * Cost_{Hit} + P * Cost_{Miss}$$

  Where P consists of:

  - Compulsory misses
  - Capacity misses (size)
  - Conflict misses (associativity)
  - Coherence misses (multi-proc/DMA)

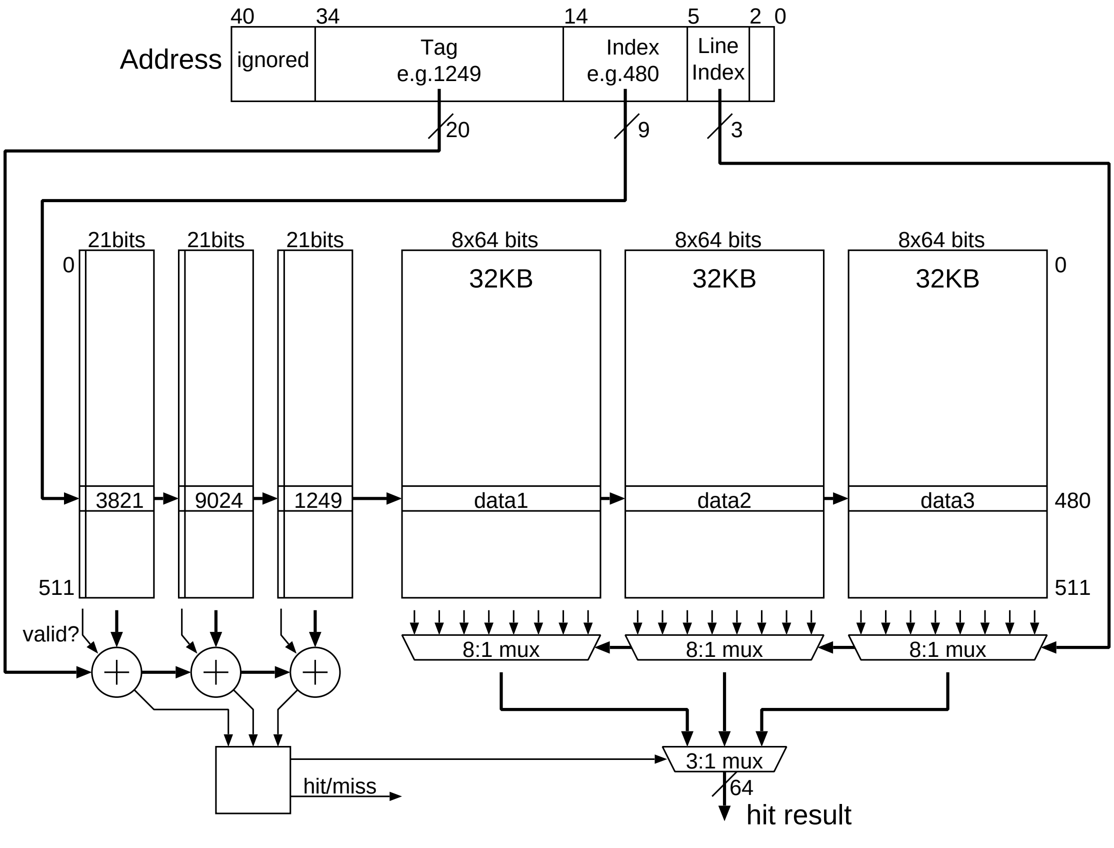
✘ Caches can increase $Cost_{Miss}$

- Build using fast (small and expensive) SRAM

- Tag RAM and Data RAM

# Associativity

- Direct Mapped (1-way, no choice)
  - potentially fastest: tag check can be done in parallel with speculatively using data
- $n$-way Set Associative (choice of $n$ e.g. 2/4/8)
- Fully associative (full choice)
  - many-way comparison is slow




A 96KB 3-way set associative cache with 64 byte lines (supporting $2^{35}$ bytes of cacheable memory)

# Replacement Policy

- Associative caches need a replacement policy
- FIFO
  - ✘ Worse than random
- Least Recently Used (LRU)
  - ✘ Expensive to implement
  - ✘ Bad degenerate behaviour
    - ∗ sequential access to large arrays
- Random
  - – Use an LFSR counter
  - ✔ No history bits required
  - ✔ Almost as good as LRU
  - ✔ Degenerate behaviour unlikely
- Not Last Used (NLU)
  - – Select randomly, but NLU
  - ✔ $log_2 n$ bits per set
  - ✔ Better than random

# Caching Writes

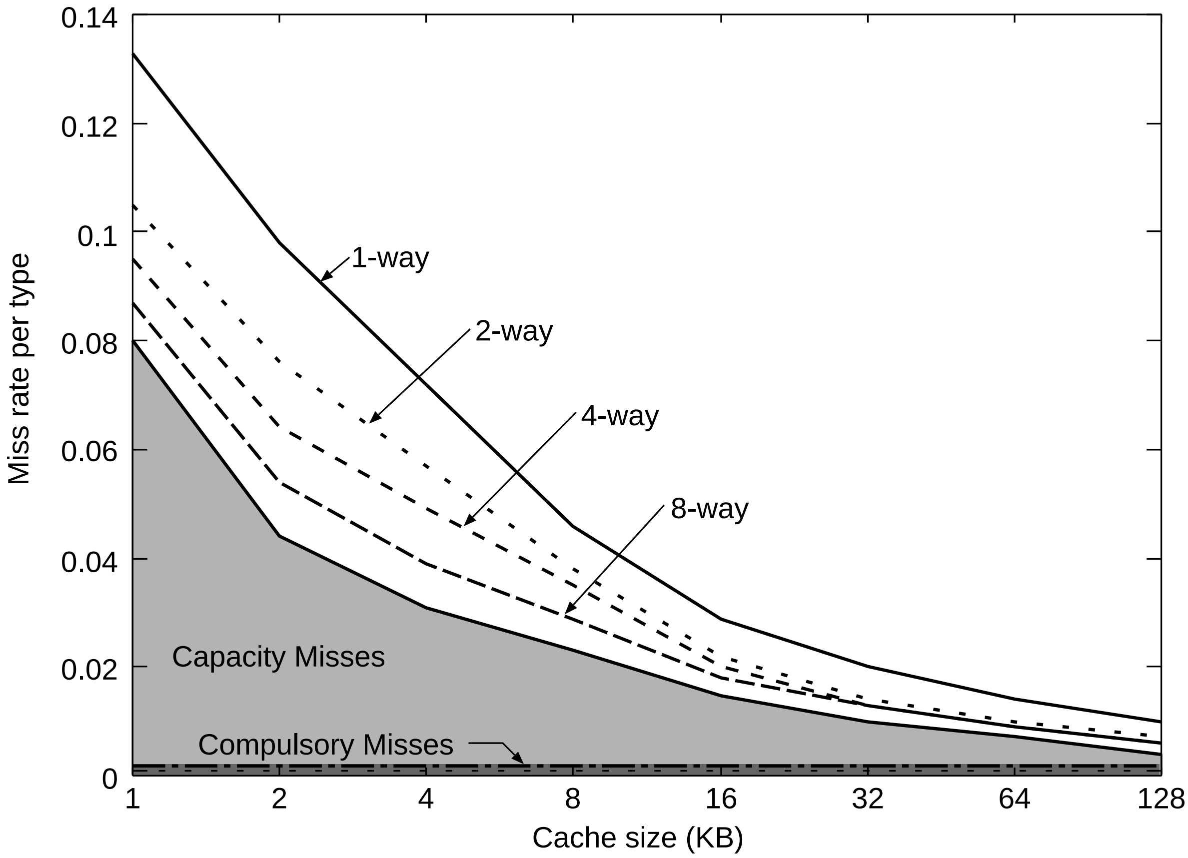
- Write-Back *vs.* Write Through
- Read Allocate *vs.* Read/Write Allocate
- Allocate only on reads and Write-Through
  - Writes update cache only if line already present
  - All writes are passed on to next level
  - Normally combined with a *Write Buffer*
- Read/Write Allocate and Write-Back
  - On write misses: allocate and fetch line, then modify
  - Cache holds the only up-to-date copy
  - Dirty bit to mark line as modified
  - ✔ Helps to reduce write bandwidth to next level
  - Line chosen for eviction may be dirty
    - ∗ *Victim writes* to next level
    - ∗ e.g. write victim, read new line, modify

# Write Buffers

- Small high-bandwidth resource for receiving store data
- Give reads priority over writes to reduce load latency
  - All loads that miss must check write buffer
  - If RaW hazard detected:
    - flush buffer to next level cache and replay
    - or, service load from buffer (PPro, 21264)
- *Merge* sequential writes into a single transaction
- *Collapse* writes to same location
- Drain write buffer when bus otherwise idle
- 21164: 6 addresses, 32 bytes per address
- ARM710: 4 addresses, 32 bytes total

# Cache Miss Rate E.g.

- SPEC 92 on MIPS
- 32 byte lines
- LRU replacement
- Write allocate/write back




A direct-mapped cache of size $N$ has about the same miss rate as a 2-way set-associative cache of size $N/2$

# L1 Caches

- L1 I-cache
  - Single-ported, read-only (but may snoop)
  - Wide access (e.g. block of 4 instrs)
  - (trace caches)
- L1 D-cache
  - Generally 8-64KB 1/2/4-way on-chip
    - Exception: HP PA-8200
  - Fully pipelined, Low latency (1-3cy), multi-ported
  - Size typ constrained by propagation delays
  - Trade miss rate for lower hit access time
    - May be direct-mapped
    - May be write-through
  - Often virtually indexed
    - Access cache in parallel with TLB lookup
    - Need to avoid virtual address aliasing
      - Enforce in OS
      - or, Ensure index size < page size (add associativity)

# Enhancing Performance :1

- Block size (Line size)
  - Most currently 32 or 64
  - ✔ Increasing block size reduces # compulsory misses
  - ✔ Typically increases bandwidth
  - ✘ Can increase load latency and # conflict misses

- Fetch critical-word-first and early-restart
  - Return requested word first, then wrap
  - Restart execution as soon as word ready
  - ✔ Reduces missed-load latency
  - Widely used. Intel order *vs.* linear wrap

- Nonblocking caches
  - Allow hit under miss (nonblocking loads)
  - Don't stall on first miss: allow multiple outstanding misses
    - * merge misses to same line
  - Allow memory system to satisfy misses out-of-order
  - ✔ Reduces effective miss penalty

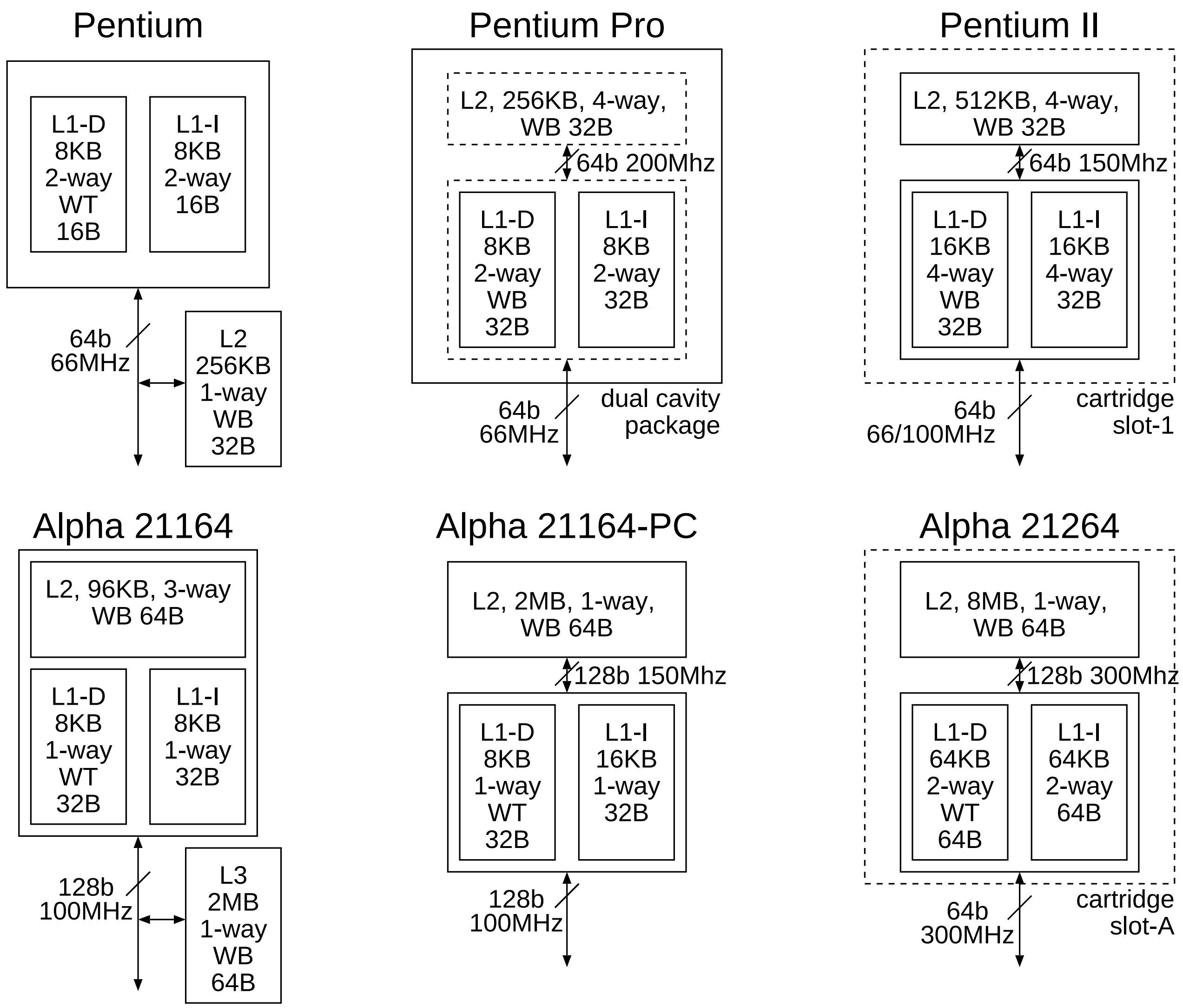
# Enhancing Performance :2

- Victim caches
  - Small highly associative cache to backup up a larger cache with limited associativity
  - ✔ Reduces the cost of conflict misses
- Victim buffers
  - A small number of cache line sized buffers used for temporarily holding dirty victims before they are written to $L_{n+1}$
  - Allows victim to be written *after* the requested line has been fetched
  - ✔ Reduces average latency of misses that evict dirty lines
- Sub-block presence bits
  - Allows size of tag ram to be reduced without increasing block size
  - Sub-block dirty bits can avoid cache line fills on write misses
    - (would break coherence on multiprocessors)

# L2 caches

- L2 caches help hide poor DRAM latency
  - large write-back cache
- L2 caches used to share the system bus pins (e.g. Pentium)
  - ✘ electrical loading limits performance
- now, a dedicated 'backside bus' is used
- L2 on same die (21164)
  - ✔ low latency and wider bus
  - ✔ associativity easier
  - ✘ limited die size, so may need an L3
    - (e.g. 21164 has 2-16MB L3)
- L2 in CPU package (Pentium Pro)
  - ✔ lower latency than external
- L2 in CPU 'cartridge' (Pentium II)
  - ✔ controlled layout
  - ✔ use standard SSRAM
- L2 on motherboard
  - ✘ requires careful motherboard design
- L1/L2 inclusive *vs.* exclusive

# Hierarchy Examples



# Performance Examples

| Ln | size | n-way | line | write | lat(cy) | lat(ns) | ld(MB/s) | st(MB/s) |
|---|---|---|---|---|---|---|---|---|
| L1 | 8KB | 1 | 32 | WT | 1 | 4 | 1205 | 945 |
| L2 | 96KB | 3 | 64 | WB | 7 | 26 | 654 | 945 |
| L3 | 2048KB | 1 | 64 | WB | 22 | 83 | 340 | 315 |
| MM | - | - | - | - | 96 | 361 | 140 | 113 |

266MHz 21164 EB164 (Alcor/CIA)

| Ln | size | n-way | line | write | lat(cy) | lat(ns) | ld(MB/s) | st(MB/s) |
|---|---|---|---|---|---|---|---|---|
| L1 | 8KB | 2 | 16 | WT | 1 | 5 | 634 | 82 |
| L2 | 256KB | 1 | 32 | WT | 11 | 55 | 193 | 82 |
| MM | - | - | - | - | 28 | 140 | 123 | 81 |

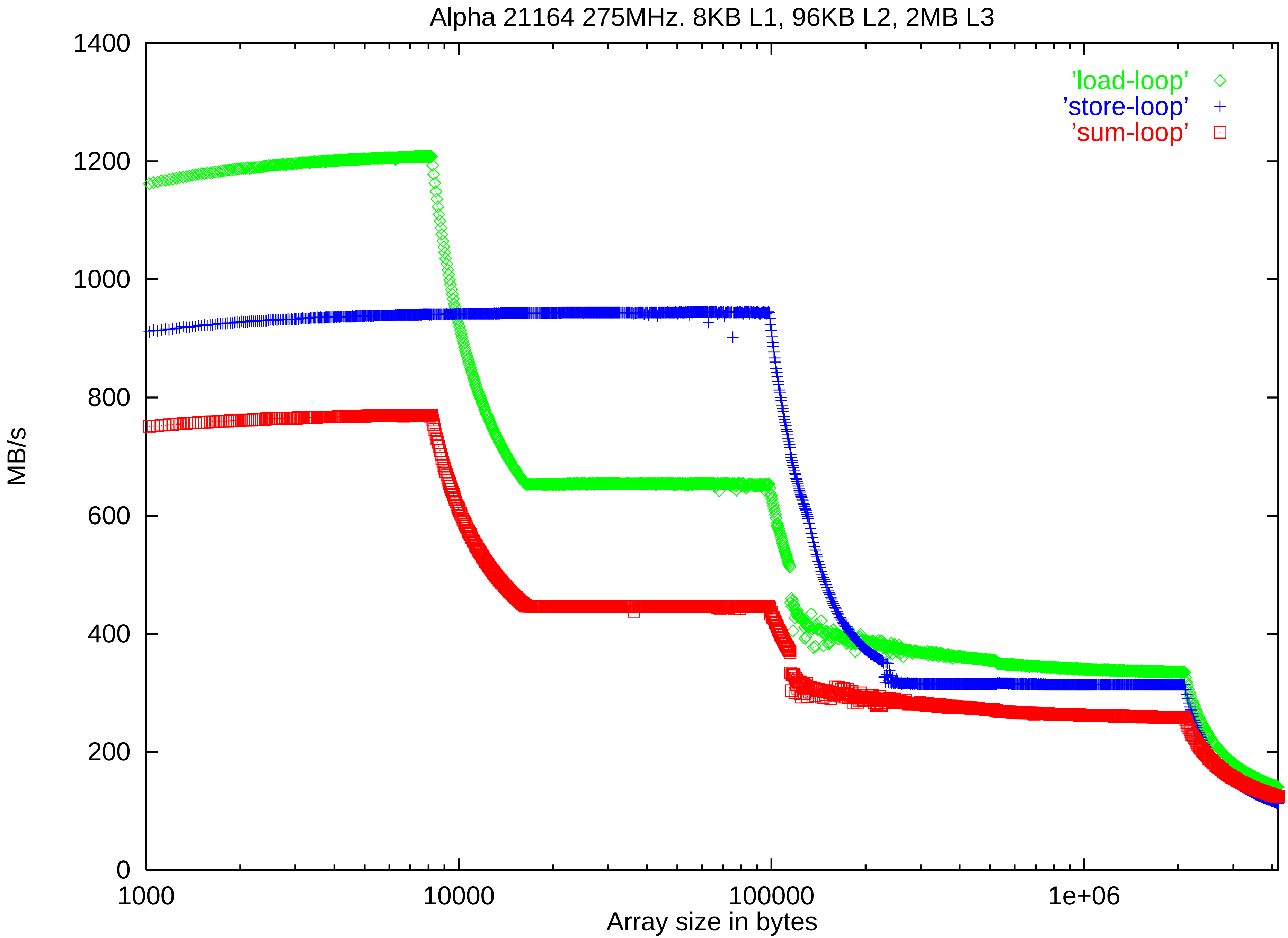
200MHz Pentium 430HX

| Ln | size | n-way | line | write | lat(cy) | lat(ns) | ld(MB/s) | st(MB/s) |
|---|---|---|---|---|---|---|---|---|
| L1 | 8KB | 2 | 32 | WB | 2 | 10 | 695 | 471 |
| L2 | 256KB | 4 | 32 | WB | 6 | 30 | 426 | 426 |
| MM | - | - | - | - | 44 | 220 | 179 | 87 |

200MHz PPro 440FX

| Ln | size | n-way | line | write | lat(cy) | lat(ns) | ld(MB/s) | st(MB/s) |
|---|---|---|---|---|---|---|---|---|
| L1 | 16KB | 4 | 32 | WB | 2 | 7 | 1048 | 735 |
| L2 | 512KB | 4 | 32 | WB | 15 | 50 | 545 | 282 |
| MM | - | - | - | - | 64 | 213 | 231 | 116 |

300MHz PII 440LX


Alpha 21164 275MHz. 8KB L1, 96KB L2, 2MB L3
'load-loop'
'store-loop'
'sum-loop'
MB/s
0
200
400
600
800
1000
1200
1400
1000
10000
100000
1e+06
Array size in bytes

# Main Memory

- Increasing sequential bandwidth
  - Wide memory bus
  - Interleave memory chips
  - $\Rightarrow$ DDR SDRAM or RAMBUS

- Access latency can impair bandwidth
  - Larger cache block sizes help

- Reducing average latency
  - Keep memory banks 'open'
    - Quick response if next access is to same DRAM Row
  - Multiple independent memory banks
    - Access to an open row more likely
    - SDRAM/RAMBUS chips contain multiple banks internally
  - System bus that supports multiple outstanding transaction requests
    - Service transactions out-of-order as banks become ready

# Programming for caches

- Design algorithms so working set fits in cache
  - Large lookup tables may be slower than performing the calculation
- Organise data for spatial locality
  - Merge arrays accessed with the same index
- Fuse together loops that access the same data
- Prefer sequential accesses to non-unit strides
  - innermost loop should access array sequentially
- If row and column access to 2D arrays is necessary, use *cache blocking*
  - divide problem into sub-matrices that fit cache
  - e.g. matrix multiply $C = C + A \times B$

```
for (kb=0;kb<N;kb+=b){
  for (jb=0;jb<N;jb+=b){
    for (ib=0;ib<N;ib+=b){
      for(k=kb;k<kb+b;++k){
        for(j=jb;j<jb+b;++j){
          for(i=ib;i<ib+b;++i){
            C[k][i] = C[k][i] + ( A[k][j] * B[j][i] );
} } } } } }
```

- Avoid access patterns that are likely to cause conflict misses (aliasing)
  - e.g. large powers of 2
- Large strides can thrash the TLB

# Special Instructions

- Prefetch
  - fetch data into L1, suppressing any exceptions
  - enables compiler to speculate more easily
    e.g. Alpha: `ld r0 ← [r1]`
- 'Two-part loads' (e.g. IA-64)
  - speculative load suppresses exceptions
  - 'check' instruction collects any exception
  - enables compiler to 'hoist' loads to as early as possible, across multiple basic blocks
  - `ld.s r4 ← [r5]`
    `chk.s r4`
- Load with bypass hint
  - indicates that the load should bypass the cache, and thus not displace data already there
  - e.g. random accesses to large arrays
- Load with spatial-locality-only hint
  - fetch line containing the specified word into a special buffer aside from the main cache
    - or, into set's line that will be evicted next
- Write invalidate
  - allocate a line in cache, & mark it as modified
  - avoids mem read if whole line is to be updated

# Multiprocessor Systems

Two main types:

1. Cache Coherenet

   ⇒ implicit shared memory communicationbetween processes/threads on single OS instance

   - Symmetric MultiProcessor (SMP) (Uniform Mem Access)
   - Non-Uniform Memory Access (ccNUMA)
   - 2-256+ processors

2. Message Passing

   ⇒ explicit communication between processes on multiple OS instances

   – May appear as 'single system image' cluster

   - Conventional networking (send/receive)
     – RPC
   - Remote DMA (read/write)
     – requires more trust & co-ordination
   - Gigabit Ethernet or specialist low-latency network
   - Avoid OS latency and overhead
     – zero-copy user-level accessible interface
     – (still need OS for blocking RX)
   - Highly scalable

⇒ Big supercomputers typically use a combination of both

# Cache Coherent Systems

1. Single shared memory (SMP)
    - shared bus (2-4 CPUs)
    - switched interconnect (2-8 CPUs)
2. Distriubted Memory (ccNUMA)
    - memory per group of CPUs (e.g. 4)
    - or, memory hangs off each CPU
    - addressable as single physical memory
        - e.g. top bits identify 'node'
        - accesses to remote memory slower
    - (physical memory may be sparse)
    - interconnect via: crossbar, mesh, hypercube

Desire large WB caches to reduce memory traffic

# Cache Coherence

- Coherence
  - Write serialization : all writes to the same location are seen in the same order
- Consistency
  - Behaviour of reads and writes wrt other memory locations

$\Rightarrow$ Implement *shared & exclusive* cache line ownership states (and *invalid*)

- CPU must get exclusive access to cache line before updating it
- Must inform all CPUs that have the line in 'shared' state
- On bus systems, snoop 'write upgrade request'
- On switched interconnect, broadcast, or use directory stored at 'home node'

# Programming considerations

- False sharing
  - variables with different ‘owners’ placed in same cache line
  - line ‘exclusive’ ownership thrashes between CPUs

  $\Rightarrow$ pack variables according to owner

  $\Rightarrow$ consider padding to cache line boundaries

- Page allocation - ccNUMA
  - physical address determines which node pages ’live’ on
  - ideally, same node they’ll be accessed from
  - different placement policies:
    - local
    - random
  - OS may employ page migration
    - copy page to different node, update all page tables

# Course Conclusions

- Modern CPUs are impressive feats of engineering
- Rely heavily on ‘common case’ assumptions
  - potential for large performance variance
  - need better compilers
  - on-line feedback-directed optimization
- Extracting loop and thread-level parallelism will be increasingly important
  - auto-parallelising compilers
  - functional / declarative languages
- ISA compatibility is currently very important
  - MSIL / Java byte code

# Questions

- How much ILP will be economically exploitable?
- Will we succeed in exploiting other forms of parallelism?
- Hardware vs. software scheduling?
- What will we do with 1 billion transistors?
- Will x86 ever die?