Comparative Architectures

CST Part II, 8 lectures

Lent Term 1998

Ian Pratt

Ian.Pratt@cl.cam.ac.uk

OHP slides and additional material

With complementary notes for supervisors

The extra notes contained in this booklet were originally written for the lecturer's use, but are included to provide assistance to supervisors.

Additional material in the form of data sheets, journal extracts, and research papers appear throughout this booklet to illustrate the points made in the text.

Course Outline

1. Comparing Implementations
   - Developments in technology
   - Cost, power, performance, compatibility
   - Benchmarking
2. Comparing Architectures
   - Instruction Set Architecture (ISA)
3. Effects of Architecture on Implementation
   - Pipelined
   - Super-scalar (multi-issue)
4. Memory System Architecture
   - Memory Hierarchy

Design tradeoffs

Course Outline

1. *Instruction Set Architecture is the user-visible processor features.*
2. *Instruction Set Architecture, not Industry Standard Architecture (PC I/O bus).*
3. *Course looks at processors and memory systems, not I/O.*
4. *Examines tradeoffs available to processor architects.*
5. *Every feature has costs. Some make chip more efficient, other less so.*

Reading material

- OHP slides, articles
- Recommended Book:
  John Hennessy & David Patterson,
  Computer Architecture: a Quantitative Approach
  (2nd ed.) 1996 Morgan Kaufmann
  (Chapters 1-5 in particular)
- The Web
  `http://infopad.eecs.berkeley.edu/CIC/`
  `http://open.specbench.org`
  `http://www.cs.colorado.edu/~klauser/ilp/ilp.html`
- comp.arch News Group

Further Reading and Reference

- 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

1. *Course uses a range of architectures including x86,ARM,Alpha,MIPS,PowerPC,PA-RISC and SPARC to illustrate points made*

Course Pre-requisites

- Processor Architecture (Ib)
  - Some ARM Assembler
  - Classic RISC pipeline model
  - Load/branch delay slots
- Computer Structures (Ib)
  - Cache hierarchies
  - Memory Systems
- Compilers (Ib/II)
  - Code generation
  - Linkage conventions

The Microprocessor Revolution

- Mainframe CPUs
  - 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
    - workstations / PCs
    - SMP / Parallel supercomputers
  - Embedded controllers
  - DSPs / Media Processors

The Microprocessor Revolution

1. *Many/some applications are unable to benefit from parallel processing, so fast scalar CPUs are still required*
2. *More I/O pins and specialist packaging techniques allow greater bandwidth for mainframe CPUs*
3. *Microprocessors have benefited over mainframe CPUs from greater financial investment, and from physical limits imposed by the speed of light benefiting smaller devices.*
4. *CMOS typically offers reduced time to market (more mature tools)*
5. *High performance embedded e.g. AMD 29K, ARM, Intel i960*
6. *Multiple embedded controllers in everything these days. 20+ in the average family car. A year 2000 time bomb...*
7. *This course focuses on General Purpose CPUs and memory systems*

Developments in CMOS

- Fabrication line size reduction
  - 1.0μ, 0.8, 0.5, 0.35, 0.25, 0.18 ...
  - 10-20% reduction p.a.
  - switching delay reduces with line size
    - $\rightarrow$ increases in clock speed
      - Pentium 66Mhz @ 0.8μ, 150Mhz @ 0.6μ, 233MHz @ 0.35μ
  - density increases at square of 1/line size
- Die size increases at 10-25% p.a.
- $\rightarrow$ Transistor count increase at 60-80% p.a.
  - enables architectural jumps
  - 8,16,32,64 bit ALUs
  - large caches
    - PA-8500: 1.5MB on-chip
  - new functional units (e.g. multiplier)
  - duplicated functional units (multi-issue)

Developments in Technology

1. *'drawn line size' is the size of smallest drawn feature*
2. *drawn as opposed to effective (due to depletion) 25% difference*
3. *for comparison, wavelength of light is 0.6μ*
4. *people have repeatedly predicted an end to line size reduction for years, but the technology just keeps improving... Quantum effects bellow 0.1μ may put an end to CMOS's progress. Will another technology be ready to take over over?*
5. *first 0.35 Pentium parts were 133MHz, process tweaking, 'stepping', and speed-grade binning has enabled 233. 266MHz Pentium about to be released...*
6. *'stepping' is the name for creating new processor variants with minor changes e.g. to fix bugs or optimize a critical path.*
7. *continual performance leapfrogging of each of the different manufacturer's fab technology*
8. *But, wire propagation delay INCREASES with reduced line size. This is becoming increasingly a problem. IBM have developed a Cu (as opposed to Al) interconnect. 3D construction would be a better long-term solution.*
9. *PA-8500 will have greater than 120 million transistors–by far the vast majority of them will be in the huge 1.5MB L1 caches.*

Developments in DRAM Technology

- DRAM density
  - increases at 60% p.a.
  - equivalent to 0.5-1 address bits p.a.
  - cost dropping at same rate
  - progress usually occurs in steps of x4
- Consequences for processor architectures:

→ May not be able to address whole of memory from a single pointer

  - segmentation

→ May run out of physical address bits

  - banked memory

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

Developments in DRAM Technology

1. *Consume DRAM at same rate as development. Mainly due to software bloat.*
2. *Always pay $200 for the 'standard' amount of ram in a PC. Now get 64MB*
3. *High-end desktops are 256MB. Days of 32bit CPUS numbered? (PPro supports 36bit physical addresses, but only a 32bit flat addressing model)*
4. *segmentation: can't address all of memory in one go. Horrid.*
5. *banked memory: Switch between several banks of memory via some out-of-band signal. E.g. PC VGA cards/80286 Expanded Memory. Very, very horrid.*
6. *Fundamental density/latency tradeoff*
7. *New high clock rate synchronous interfaces (e.g. SDRAM,RAMBUS) enable much higher bandwidth to data in the same DRAM ROW.*
8. *SRAM is lower latency, but much more expensive (6T vs 1). Hence the need for caching.*

μprocessor Development Cycle

- Fabrication technology has huge influence on power and performance

$\rightarrow$ must use the latest fabrication process

- Keep development cycle short (3-4 years)
  - Non CMOS technology is often dangerous
- Advance teams to research:
  - process characteristics
  - packaging
  - number of transistors
  - required performance
  - investigate key problems
- Keep up or die!
- Alpha architects planned for 1000x performance improvement over 25 years

μprocessor Development Cycle

1. *The performance advantage of using ECL or GaAs is often squandered through extended time-to-market.*
2. *CPUs must be fabricated on a modern process to be competitive.*
3. *At the start of development, CPUs are being designed for a target manufacturing process that doesn't yet exist.*
4. *Costs for each generation of chip are rising almost exponentially*
5. *Ideally, architectures should be designed such that they contain no features that will hamper the ability of future implementations to make use of techniques enabled by developments in fabrication technology. This requires incredible foresight.*
6. *The Alpha architects, Sites and Witek, planned a 25 year life for Alpha, over which they expect to see a >1000x increase in performance. (10x on multiple issue, 10-100x on clock, 1-10x on thread level parallelism)*
7. *Many other architectures were not planned with so much forethought. They were optimal for the implementation at the time.*

Power Consumption

- Important for laptops, PDAs, mobile phones, set-top boxes, etc.
- 72W for Digital Alpha 21264 @ 500MHz
- 35W for Intel Pentium Pro @ 200MHz
- 25W for Digital Alpha 21164 @ 400MHz
- 17W for Intel Pentium @ 200MHz
- 420mW for Digital StrongArm @ 233MHz, 2.0V
- 130mW for Digital StrongArm @ 100MHz, 1.65V
- Smaller line size results in lower power
- Reduce clock speed often reduced to scale power

Performance per Watt

Power Consumption

1. *increasingly important in some markets.*
2. *batteries wouldn't last too long in a palm top with a 21264!*
3. *Fan noise from set-top boxes would be a problem*
4. *Can dissipate up to about 100W in a desktop machine using heat pipes etc,*
5. *Do need significant compute power in PDAs: handwriting recognition*
6. *Both SA and 21164 fabricated in Digital's 0.35μm process. One designed for all out performance, the other for low power.*
7. *Sleep modes and clock frequency scaling to reduce power. (Some fully static components enable clock to be stopped)*

Price

- E.g.:
  - \$1 for an 8bit micro controller
  - \$15 for a StrongArm (50mm^2)
  - \$500 for a Pentium Pro (197mm^2, 4.3M + 150mm^2, 25M)
  - \$1000 for Alpha 21164A (209mm^2, 9.3M[1.8M])
  - \$2500+ for an Alpha 21264 (300mm^2, 15.2M[6M])
- Some influence by die size, packaging
- Large influence by manufacturing volume
- Costs reduce over product life (e.g. 40% p.a.)
  - Yield improves
  - Speed grade binning

1. *logic vs. cache density. Figure in square brackets is number of core transistors.*
2. *Number of pins, packaging costs, memory bandwidth. Current BGA (solderBall Grid Array) packages allow 600+ pins*
3. *Need to recover R&D costs, and Fab Plant construction.*

4. *Design of a new CPU costs >100M. >150 people working for 3 years*
5. *Component prices aren't static, they reduce over product life, as much as factor of 10*
6. *Speed grade binning and marketing influence. Over clocking*

Compatibility

- Backwards Binary Compatibility
  - 8086, 80286, 80386, 80486, Pentium, Pentium Pro, Pentium II
  - may still need to re-optimize
- Pin Compatibility
  - x86: Intel, IBM, AMD, NexGen, Cyrix
- Often awkward to change architecture
  - Users owns old s/w
  - Binary translators e.g. FX!32, WABI
  - Need co-operation from s/w vendors
  - Most computer sales are upgrades
- Platform independence initiatives
  - Source, p-Code, JAVA bytecode

Compatibility is very important

Compatibility

1. *pin compatible: use same motherboard. (Though Intel have patented slot-1)*
2. *Need to persuade s/w vendors to optimise their products for CPU*
3. *WABI is an x86 emulator available for a number of workstation platforms. FX!32 is Digital's x86 to Alpha binary translator.*
4. *binary2binary translators: hard to get software vendor to support translated product.*
5. *binary2binary optimisers: re-write old binary to produce an optimised one. Used by Digital to optimise Alpha NT versions of MS products that were compiled with a lousy compiler.*
6. *Changing arch e.g. Sun/HP/Apple from 68k, DEC from VAX and MIPS. Often loose customers.*
7. *Will we see Java byte code versions of MSWord? How will they perform?*

Performance Measurement

- Try before you buy! (but, often not possible)
- 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
  - TPC, SPEC

Performance is application dependent

Performance Measurement

1. *Performance is often not particularly critical*
2. *Often not possible to try application: live system or no optimized binary available*
3. *desktop workloads are especially difficult: highly varied, idle much of the time, response time is important*
4. *fallacies of using MHz: 450MHz Alpha 21164 = 180MHz HP PA-8000*
5. *fallacies of using MIPS: Rarely able to occupy all functional units to achieve peak MIPS performance. PA-8000 better at doing this than Alpha 21164*
6. *livermore loops: good if that's your application, won't tell you much about anything else. (Livermore loops are an FP intensive algorithm that forms part of Gaussian elimination)*
7. *synthetic benchmarks: derived from instruction traces of real apps. They are usually too small, don't reward real compiler optimizations, and allow unrealistic ones.*
8. *eg. dhystone: 25%discarded by optimizing compiler. Function inlining had to be forbidden to enable 'fair' comparison.*
9. *whetstone: trick optimization sqrt(exp(x)) = exp(x/2)*
10. *Application performance is not dependent on just the CPU performance, but on the compiler and memory system too.*
11. *Application performance may also depend on IO e.g. graphics, disk, network performance*
12. *TPC for transaction processing systems*

Standard Performance Evaluation Corporation

- SPEC is most widely used benchmark
  - processor manufactures
  - workstation vendors
- CPU INT and FP 89, 92, 95, soon 98
- Suite updated to reflect current workloads
- CINT95: 8 integer C programs
- CFP95: 10 floating point in Fortran
- measures:
  - processor
  - memory system
  - compiler
  - NOT OS,libc,disk,graphics,network

1. *SPEC so important that it used to provide input into CPU design*
2. *execution time in non SPEC supplied code is minimal (i.e. not OS or libc)*
3. *disk,video,network performance can be very important—the CPU is often not the bottleneck.*
4. *Video performance has an important effect on how fast a system 'feels'.*

Choosing programs for SPEC95

- More programs than SPEC92
- Bigger programs than SPEC92
  - Don't fit in on-chip caches
- Reflect some real workloads
- Run for several minutes
- Not susceptible to trick transformations
  - Vendors invest huge s/w effort
- Fit in 64MB
- Moving target, hence SPEC98
- SPEC92 and 95 results not translatable

1. *one of the SPEC92 programs was susceptible to a trick transformation*
2. *huge amount of effort goes into SPEC benchmarks by vendors*
3. *compiler optimization - some may even be useful for real progs!*
4. *I&D trace data, processors designed to execute SPEC*
5. *multiple submissions for same machine as compilers improve*
6. *no simple relationship between SPEC 92 and 95 numbers - 95 more demanding on memory systems.*

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	A 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

CINT95 suite (C)

1. *Considerable effort goes into selecting programs for SPEC. They are all (modified) real applications; Applications that someone, somewhere cares about the performance of.*
2. *126.gcc cross compiles number of programs for SUN, not native*
3. *Jpeg and Gcc have large data sets, which do exercise filesystem I/O (eg. 13% worse over NFS, 5% better with RAM disk)*
4. *Which of these apps represent my workload? What will SPEC say about the performance of emacs, Netscape, MS Word, Quake?*

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

SPEC reporting

1. *Take median of string of results*
2. *full disclosure: OS and compiler versions, system configuration*
3. *compiler and system must be for sale to be official*
4. *peak results: great long lists of cryptic compiler flags*
5. *base results: more likely what a real user would get*
6. *unsafe optimizations: but, must be correct for any input data set*
7. *→ Intel accidently cheated on one of the fp benchmarks*
8. *SPECrate: copies run*reference*seconds in day / elapsed time=rate in jobs/day*
9. *SPECrate: important for batch processing systems*

Totalling Results

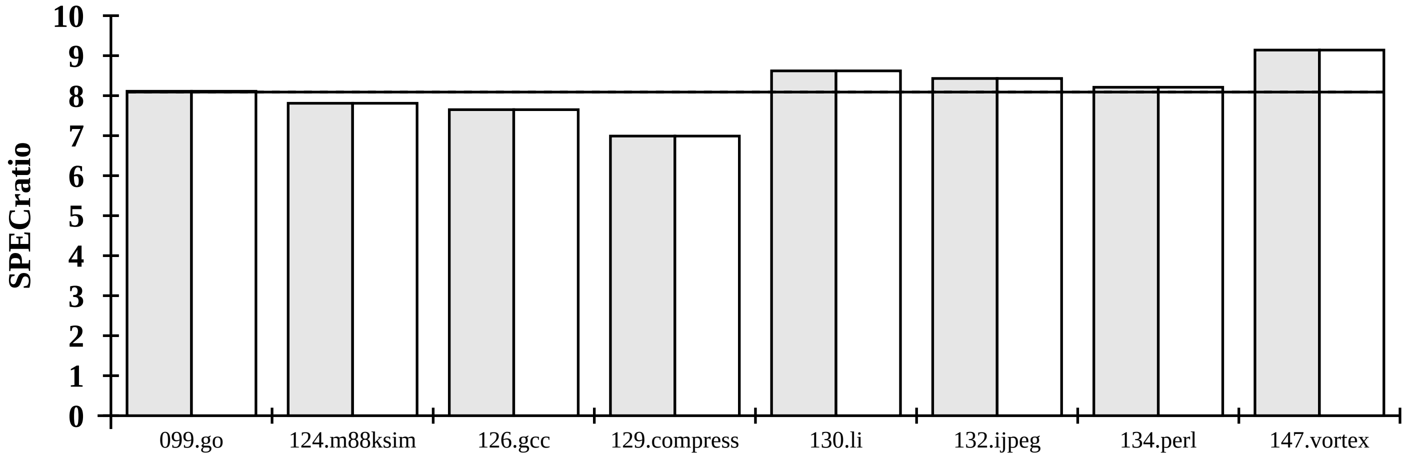
- How to present results?
  - Present individual results?
  - Arithmetic mean?
  - Weighted harmonic mean?
  - SPEC uses Geometric mean, normalised against a reference platform
    * allows normalization before or after mean
    * performance ratio can be predicted by dividing means
- SPEC95 uses Sun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

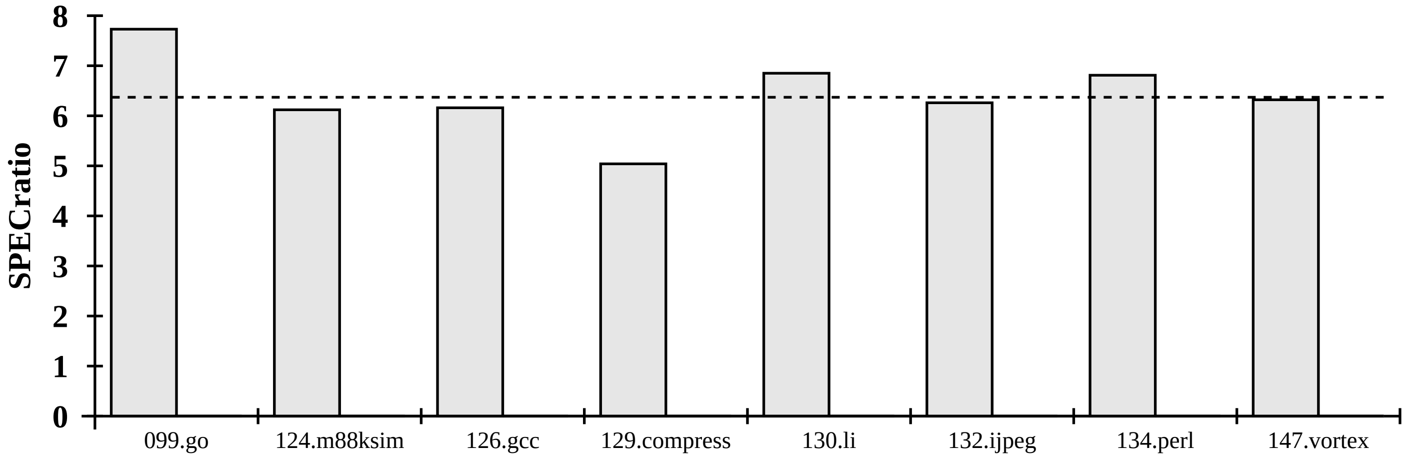
For More Information Contact:
SPEC
10754 Ambassador Drive, Suite 201
Manassas, VA 22110
(703) 331-0180
info@specbench.org
http://www.specbench.org

1. *full disclosure of machine / compiler configuration*
2. *compiler options used*
3. *Feedback Directed Optimization. Optimization pass uses the SPEC95 Training input set - not allowed to use the Reference input set for training.*
4. *Base SPEC ratio = Ref time / Base run time*
5. *SPECint_base = (r1*r2*r3...r8) ** 1/8*

SPEC CINT95 Results

Name of company making system Name of machine	SPECint95 = -- SPECint_base95 = 6.37

SPEC license # 1178	Tested By: CUCL	Test Date: Date on which test was run	Date when hardware can be publicly obtained	Date when software can be publicly obtained



Hardware/Software Configuration for:
Name of machine

Hardware

Model Name:	Model name of machine
CPU:	Pentium Pro 200
FPU:	Name of FP coprocessor or integrated (if part of CPU)
Number of CPU(s):	Number of CPUS in system
Primary Cache:	Size of primary cache
Secondary Cache:	Size of secondary cache
Other Cache:	Size of other caches
Memory:	Amount of memory in system
Disk Subsystem:	Size of hard disk(s)
Other Hardware:	Other special harware?

Software

Operating System:	Operating system name and version
Compiler:	gcc 2.7.2p
File System:	Type of file system
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:
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

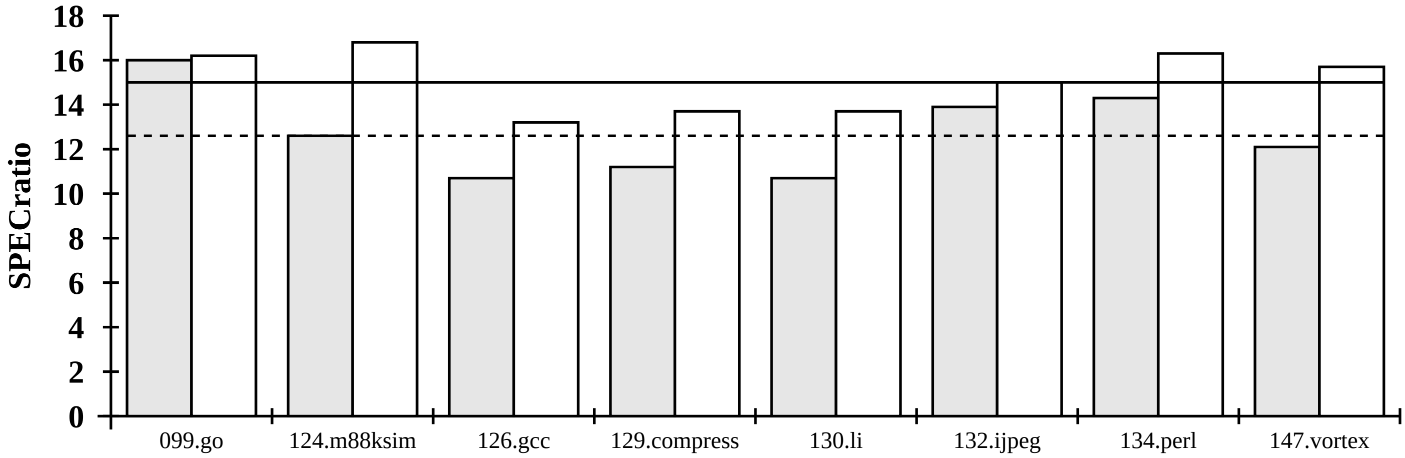
1. *Compiler does make a difference: gcc 2.7.2p 20% slower than the Intel reference compiler shipped with x86 Solaris*

SPEC CINT95 Results

Digital Equipment Corp.
AlphaStation 500/500

SPECint95	=	15.0
SPECint_base95	=	12.6

SPEC license #	2	Tested By:	Digital NH	Test Date:	Aug-96	Hardware Avail:	Oct-96	Software Avail:	Aug-96



Hardware/Software Configuration for:
AlphaStation 500/500

Hardware

Model Name:	AlphaStation 500/500
CPU:	500 MHz 21164
FPU:	Integrated
Number of CPU(s):	1
Primary Cache:	8KB(I)+8KB(D) on chip
Secondary Cache:	96KB
Other Cache:	8MB
Memory:	128MB
Disk Subsystem:	2 x 2GB
Other Hardware:	Ethernet

Software

Operating System:	Digital UNIX T4.0A-2
Compiler:	DEC C X5.3-148-334R /cc.alt/protect_system_headers.sh
File System:	UFS
System State:	Multi User; vm-page-free-min 5760

Benchmark # and Name	Reference Time	Base Run Time	Base SPEC Ratio	Run Time	SPEC Ratio
099.go	4600	287	16.0	283	16.2
124.m88ksim	1900	151	12.6	113	16.8
126.gcc	1700	159	10.7	129	13.2
129.compress	1800	161	11.2	132	13.7
130.li	1900	177	10.7	139	13.7
132.ijpeg	2400	173	13.9	160	15.0
134.perl	1900	133	14.3	117	16.3
147.vortex	2700	223	12.1	172	15.7
SPECint_base95 (G. Mean)			12.6		
		SPECint95 (G. Mean)			15.0

Notes/Tuning Information

```
Compiler: cc.alt -migrate -std1   Base optimizations: -O5 -arch ev56 -non_shared -om
Portability flags: m88ksim: -DLEHOST   perl: -DI_TIME   vortex: -D__RISC_64__

Peak flags: all use -ifo -non_shared    go: -O4 -inline speed -om    m88ksim: -O5 -speculate all -inline speed
with feedback    gcc: -O5 -inline speed -arch ev56 -xtaso_short with feedback    compress: -O5 -tune ev5 -32data
-assume whole-program -arch ev56 -inline speed -om -lsys5    li: -O5 -inline speed -xtaso_short -speculate all
-lsys5 with feedback    ijpeg: -O5 -fast -unsigned -inline speed -arch ev56 -speculate all -lsys5 with feedback
perl: -O5 -inline speed -arch ev56 -lsys5 with feedback pass-2 only    vortex: -O5 -fast -inline speed with feedback

feedback: PASS1=-gen_feedback -g1
fdo_pre1=mv %binary% %exename%.orig;pixie -pids %exename%.orig -o %exename%
fdo_run1=%command%
fdo_post1=prof -pixie -feedback ../../src/%exename%.fb %exename%.orig %exename%.orig.Addrs %exename%.orig.Counts.*
PASS2=-feedback $(EXENAME).fb -r -ifo
fdo_pre2=mv %exename% %exename%.rr;ld -o %exename%fb %exename%.rr -lexc;pixie -pids %exename%fb -o %exename%
fdo_run2=%command%
fdo_post2=ld -om %exename%.rr -o %exename%.rrom -lexc;prof -pixie -merge %exename%fb.Counts %exename%fb %exename%fb.Addrs
%exename%fb.Counts.*;/usr/lib/cmplrs/cc/om -om_ireorg_feedback %exename%fb -o %binary% %exename%.rrom
gcc and li add -taso to feedback commands om and ld
```


For More Information Contact:

SPEC
10754 Ambassador Drive, Suite 201
Manassas, VA 22110

(703) 331-0180
info@specbench.org
http://www.specbench.org

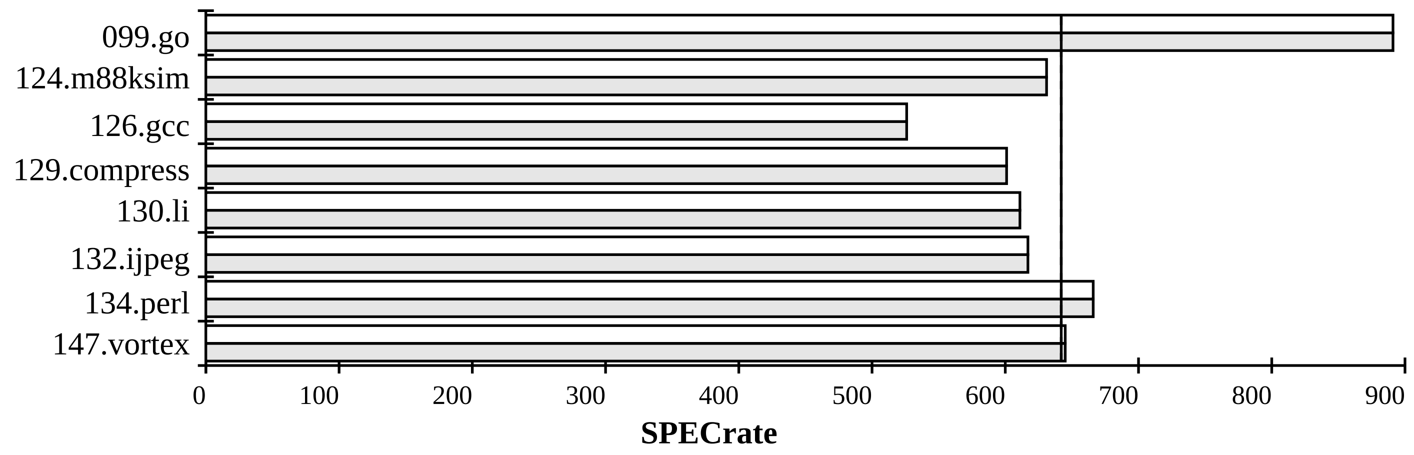
1. *Peak results can be significantly higher than base. Note the huge list of dodgy compiler options.*

SPEC CINT95rate Results

Digital Equipment Corp.
AlphaServer 8400 5/300

SPECint_rate95	=	642
SPECint_rate_base95 =		642

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

1. *Running 10 copies on a 10 CPU machine*
2. *Base SPEC ratio = copies run*reference*seconds in day / elapsed time (=rate in jobs/day)*

machine	processor	mem/cpu	cache sizes	int_base	fp_base	int	fp
Sun SS10/40	SuprSP	40/40	20/16	1.00	1.00	–	–
DEC 500/266	A21164	43/266	2M+96+8/8	6.32	10.0	7.93	11.1
DEC 500/333	A21164	46/333	2M+96+8/8	7.94	10.5	9.82	12.5
DEC 500/400	A21164	53/400	2M+96+8/8	9.77	12.8	12.3	14.1
DEC 500/500	A21164	48/500	8M+96+8/8	12.6	18.3	15.0	20.4
DEC 4100/5/400	A21164	75/400	4M+96+8/8	10.1	16.0	12.1	17.2
DEC 4100/5/400	2xA21164	75/400	4M+96+8/8	10.1	20.7	12.1	24.2
DEC 4100/5/400	4xA21164	75/400	4M+96+8/8	10.1	26.6	12.1	33.4
HP 9000/K460EG	PA8000	60/180	1M/1M	10.8	18.3	11.8	20.2
Intel XXpress	Pentium	66/200	1M+8/8	5.47	2.92	5.47	3.68
Intel Alder	PentPro	66/200	256+8/8	8.09	5.99	8.09	5.99
Digital EB264	A21264	100/500	8M+64/64	30	50	–	–

SPEC 95 Results

1. *memory/cpu speed in MHz*
2. *cache sizes L3+L2+L1-I/L1-D*
3. *although a uniprocessor benchmark, some auto-parallelizing compilers manager to reaps benefits from multiple CPUs*
4. *futility of MHz, even MIPS or FLOPS : compare PA8000 @180 and 21164 @400, or Pentium @200 and PPro @200*
5. *"Speed demons vs brainiacs": PA8000 gets better utilization from its 2 integer pipelines, and executes more expressive instructions than Alpha.*
6. *For a given processor core, SPEC does scale reasonably with MHz, providing Memory b/w is scaled comparably.*

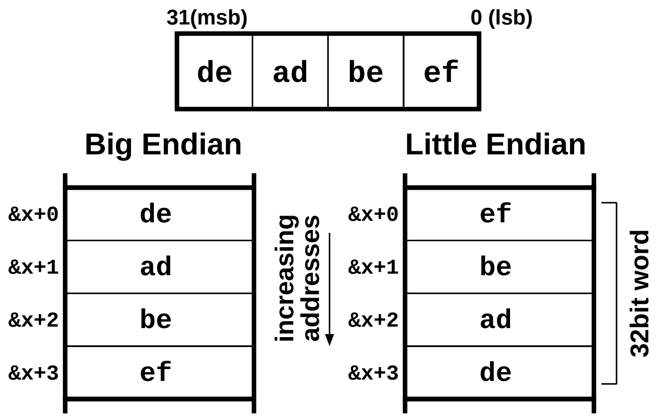
Comparing Products Summary

- 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

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");
```



Byte Sex

1. *There is no right or wrong...*
2. *bit and byte endian. IBM/PowerPC use weird big bit-endian in manuals*
3. *Only programs that access the same value as both bytes and words care, or pass data to other processors.*
4. *MIPS: little-E on Digital Ultrix, big-E on SGI IRIX*
5. *processor XORs address bits A0 and A1 based on endian-mode. Motherboard data bus must also be wired accordingly. (endian mode cannot be changed on-the-fly)*
6. *Machines that only access data as words do not have an endian per se. e.g. Cray*
7. *Some old machines had 9bit bytes. Should really call a byte an octet...*

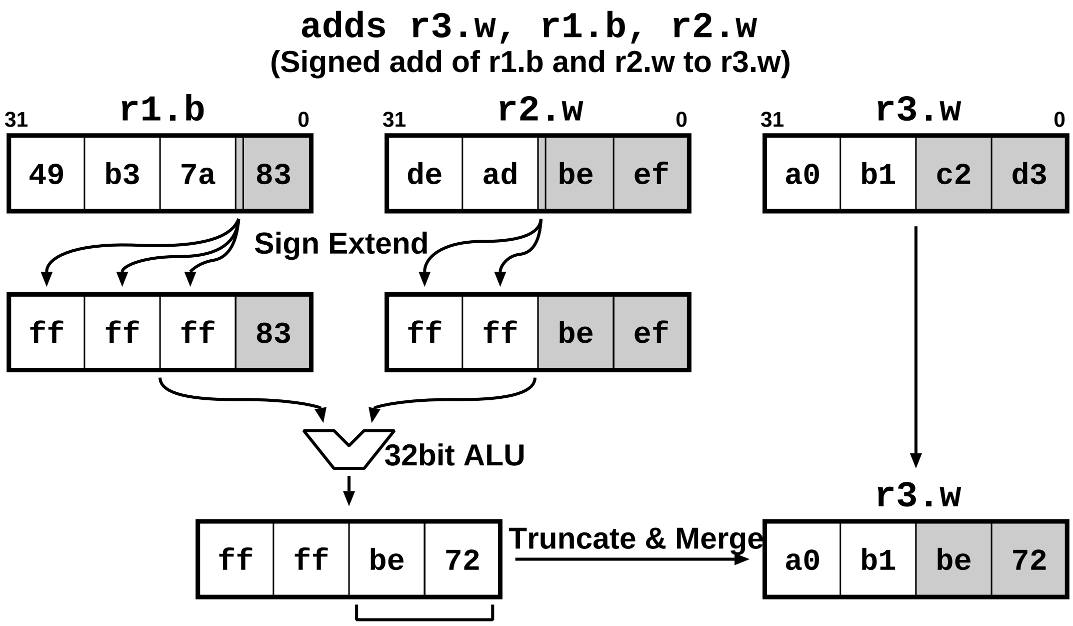
Data Processing Instructions

- 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

1. *add,sub: same ALU operation for signed and unsigned values*
2. *compare is a subtract in which throws result away*
3. *add overflow c: ignore, Fortran: need to know about it : exception vs test*
4. *divide: example of instruction that got added in Alpha*
5. *barrel shifters vs ALU (1 bit at a time shifts on old CISC e.g. 6502)*
6. *no rotate in C, but compilers can still synthesize it*
7. *rotates not in some RISC, 2 shifts and OR*
8. *arithmetic shift right replicates msb.*

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

Operand Size

1. *extension and merging can be a right pain on the critical path.*
2. *When doing truncation (part of merge), need to check for overflow*
3. *VAX and x86 allow operands of different sizes to be used in the same instruction.*
4. *Try to make Word size operations the common case*
5. *Compiler pads chars and shorts into words. (Can't be done for arrays).*
6. *Excluding 32bit ops from 64bit machines would result in too much performance loss on current code. But, Alpha always produces a 64bit result, avoiding merging.*

CISC instructions RISC dropped

- Emulated in RISC:

```
move   r1 ← r2   e.g. or   r1 ← r2, r2
zero   r1        e.g. xor  r1 ← r1, r1
neg    r1        e.g. sub  r1 ← #0, r1
nop              e.g. or   r1 ← r1, r1
sextb  r1 ← r2   e.g. lsl  r1 ← r2, #24;
                      asr  r1 ← 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 are awkward:
  - memcpy instruction
  - strcmp instruction

CISC instructions RISC dropped

1. *NB. Note use of $\leftarrow$ to indicate source and destination. Some assemblers use src,dest other use dest,src. The $\leftarrow$ convention is used in these notes to avoid this ambiguity.*
2. *nop, mov, zero would have had short instruction encodings on CISC*
3. *No execution time or opcode size advantage to be gained from having dedicated nop,mov,zero instructions on RISC. (would even result in more complicated decode logic).*
4. *Assembler may provide macros for mov, neg, clr etc.*
5. *Many 70's architectures attempted to close the 'semantic gap' between HLLs and machine code*
6. *Complex instructions that perform very specific functions*
7. *Although they are attractive to humans, Compilers often find them inconvenient to use.*
8. *Loop and procedure call primitives often weren't quite right*
9. *Procedure call primitives often were unable to make use of the compiler's static analysis to determine which registers to save and restore. (leaf procedures, call graph etc)*
10. *On more recent implementations, it is often slower to use the complex instruction than to use several standard ones. E.g. PPro, Pentium II*
11. *memcpy: has intermediate state–has to be resumed after an interrupt or exception rather than restarted. VAX/x86 put intermediate state in GPRs.*
12. *However, fast memcpy would be very useful (though normally limited by memory bandwidth anyway)*

New Instructions

- integer divide
- popcount, priority_encode
- Multimedia Instructions
  - Intel MMX, SPARC VIS, Alpha, PA-RISC MAX
  - MPEG, JPEG, 3D Shading
  - parallel processing of packed sub-words
  - E.g. 8x8,4x16 bit packed values
  - arithmetic with clipping
  - min,max
  - transposition
  - RMS error estimation (MPEG encode)
- New binaries may need to be backwards compatible
  - Software emulation
  - Feature registers

1. *divide - enough Si to be worth it*
2. *popcount, priority - used by OS, requested on Alpha by Cray*

3. *multimedia instructions: Probably won't be generated by compilers. (Months spent hand coding MPEG routines)*
4. *x86 MMX uses FP regs so as not to create more user state (thus no changes to the OS [MS Windows] are required)*
5. *software trap mechanism is slow–better to detect the new instructions absence (e.g. via the feature register), and branch to some code providing similar functionality.*
6. *MAX- see PA-RISC 2.0 Extensions article*
7. *Is adding all these instructions still adhering to the RISC philosophy? Given the increasing importance of multimedia, the answer is probably yes. It will be interesting to see if compilers ever manage to use these instructions...*

Accumulator Architectures

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

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

- Compact instruction encoding
- Few registers, typically ≤ 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

1. *historical perspective*
2. *3 mem locs labelled foo, bar and res. (Offsets calculated by assembler)*
3. *registers much faster for storing intermediates. registers not being used for their special functions can be used as temporaries.*
4. *never enough of the right type of register...*

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

1. *Like accumulator architecture, tricky to make go fast (especially super-scalar implementations)*
2. *Stacks are easier to cache then general memory accesses, but spotting dependencies between instructions is still harder than with registers.*
3. *One advantage of stack architectures is that they do not impose a limit on the number of architecturally visible 'registers', thus allowing future scalability.*

General Purpose Register Sets

- Post 1980 architectures, both RISC and CISC
- 16,32+ 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

1. *x86 32bit mode (IA-32) is basically an 8 register GPR set*
2. *AMD 29K embedded controller family (used in many laser printers) has 128 registers, as does Intel's IA-64 Merced*
3. *32 may not be enough (look at register renaming later)*
4. *FP regs are used by different functional units from INT regs. Interpreting integers as FP values and vice versa is relatively rare. Makes sense to keep them separate, enabling a dedicated register file to be located next to the appropriate functional units.*

5. *New 'media processors' (DSPs) tend to have common int and FP regs. They believe that conversion between int and fp will be a common operation.*
6. *Dedicated address registers may be mildly advantageous to implementation, as address registers could be located closer to the load/store unit.*
7. *But, Separate address and data registers can be a pain, as for some loops you wish you had more address registers, for other loops you wish you had more data. End up copying values back and forth between data and address.*

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 is two operand
  - more compact encoding (16bits)

1. *Only one port required on D-cache to avoid stalls due to structural hazards*

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
```

- 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

Register-Memory

1. *Variable length encoding: Intel 32bit mode 1-17bytes, 16bit mode 1-9, VAX 1-19*
2. *orthogonal: any combination is legal*
3. *each instruction may be slower, harder to pipeline*
4. *Although most post 1980 architectures use GPR sets, there are still some special registers...*

Special Registers : 1

- Zero register
  - Read as Zero, Writes discarded
  - e.g. Alpha, Mips, Sparc
  - Data move: `add r2 ← r1, r31`
  - nop: `add r31 ← r31, r31`
  - prefetch: `ldl r31 ← (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

1. *zero register: r0 on MIPS, r31 on Alpha*
2. *prefetch: Allows a value to be pulled into cache without risk of generating an exception*
3. *ARM's visible PC exposes the pipeline depth. This was a pain on the StrongArm which has a 5 stage pipe instead of 3.*

Special Registers : 2

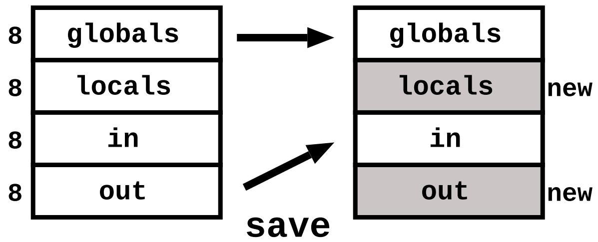
- Condition code (Flag) registers
  - Carry, Zero, Negative, Overflow
  - Used by branches, conditional moves
  - Critical for pipelining and super-scalar
  - CISC: one CC reg updated by all instructions
  - ARM, SPARC: one CC reg, optionally updated
  - PowerPC: multiple CC regs (instruction chooses)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

Special Registers : 2

1. *Revision: meanings of the various CC flags.*
2. *ARM/SPARC: Bit in opcode to determine whether CC updated.*
3. *Single CC register poses problems for super-scalar implementation due to false dependencies.*
4. *Alpha and MIPS have no CC registers. GPRs used in their place, thus using the normal dependency analysis logic. Have to test for carry/overflow etc using ALU instructions.*
5. *More on the implications of CC regs for control flow later...*
6. *register conventions: args, result, temps, caller saves, callee saves*
7. *push to stack: eg if recursive*
8. *The link register typically has to be moved to a callee saves register before making another procedure call. Alpha avoids this by allowing the current PC to be stored in a nominated register.*

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



Register Conventions

1. *languages can have different conventions for the same CPU*
2. *Caller saves its own args and temps*
3. *goal: save as few as possible in total. Don't save all before procedure call as callee may not use many. Don't save all after as caller may not have used many.*
4. *Compiler should allocate variables to 'Saved' registers, then put any that are left over in 'Temps'.*
5. *Call graph optimization can be used to tune which registers are actually saved.*
6. *Sparc: don't have to move incoming args to safety before making another call–it's done in hardware.*
7. *Sparc: running out of windows is bad – normally need to save the full 16 registers in the window regardless of which are actually used.*
8. *Sparc: Total context is very large—painful to context switch.*

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

RISC Addressing Modes

1. *Different assemblers have different syntax - Even for same arch!*
2. *register indirect obviously a special case of register indirect with displacement - just a more compact encoding on CISC.*
3. *Pretty much all processors have the RISC addr modes*
4. *Immediate values: on most CISCs they must be prefixed with # or $*
5. *reg indirect with displacement used for accessing local vars off stack pointer, or global variables from the GP register*
6. *displacement size is often limited on RISC architectures.*

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

Less RISCy addr modes

1. *can improve instruction density*
2. *reg+reg requires an extra read port on register file*
3. *ARM: shift unit can be used to scale one operand for any instruction.*
4. *syntax for displacement and update highly variable. For some architectures, displacement is fixed to the sizeof the value being loaded.*
5. *Full descending stack use: pre dec, post inc*
6. *Empty descending stack use: post dec, pre inc*
7. *Displacement with update requires an extra write port on register file.*

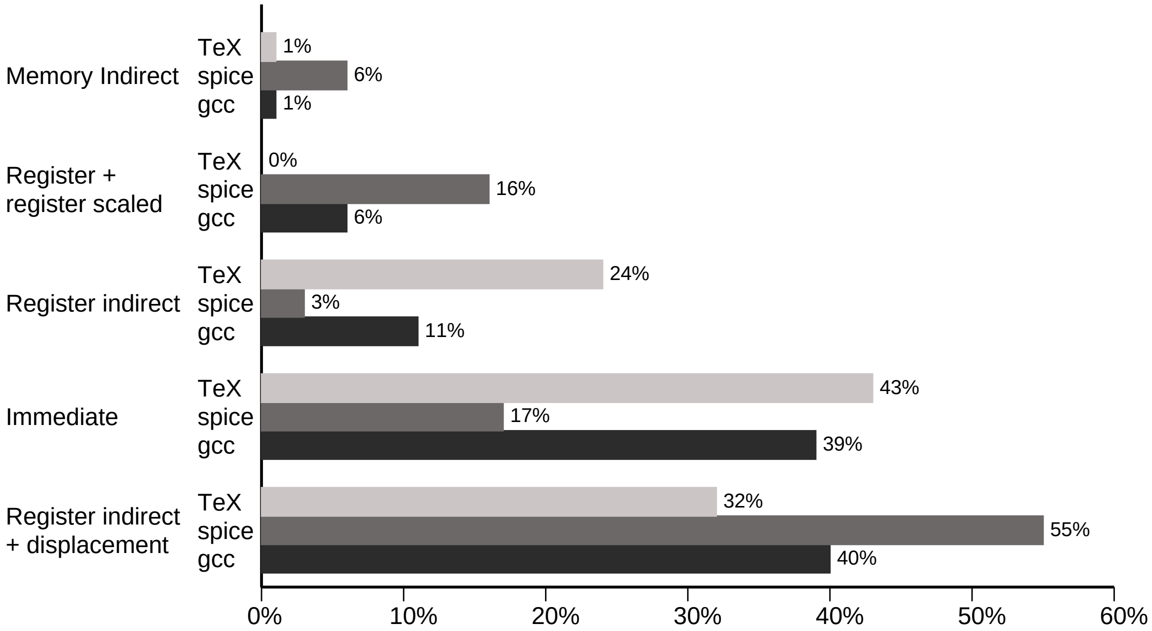
CISC Addressing Modes

- 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

CISC Addressing Modes

1. *Absolute mode: eg. 6502 Zero page*
2. *absolute mode synthesizable on RISC with zero register*
3. *absolute mode is not compatible with object oriented programming*
4. *Memory indirect: two memory references required, thus two possibilities for exceptions to occur.*
5. *Even more complicated modes exist e.g. 68020's Mem Indirect Pre Indexed mode: two displacement, a scale, two registers and two memory accesses. Abandoned on 68040?*
6. *PC indirect useful for Position Independent code, e.g. for single address space operating systems, shared libraries etc.*
7. *branch instructions are typically PC relative – PC indirect mode enables relative access to data. Particularly useful for read-only data, which can be placed in the text segment (usually just after the function in which it is used).*
8. *Assembler can usually fills in the offset between the instruction doing the access and the data being accessed.*
9. *ARM supports PC indirect mode, as PC is just another GPR*
10. *Other RISC processors generally use other conventions to access literal constants in a position-independent manner. e.g. a register containing the 'procedure variable'*
11. *PC indirect mode can be synthesized on other RISC processors by doing a subroutine call to the following instruction, and using the PC value stored in the link register.*

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

1. Quantitative approach

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
  - MIPS 16, Alpha 8, ARM 8 with 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

1. *Immediates that can not fit in instruction must be synthesized in a register, or loaded from the literal pool code segment.*
2. *max size of displacement can limit the size of the global data area under some linkage conventions.*
3. *displacement is usually signed because negative displacements are common.*

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

VAX: fully variable. Each operand 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

1. *RISC: All instruction accesses must be to aligned addresses*
2. *Many encodings reserved so as to enable future extensions e.g. the whole swathe of instructions added to MIPS, SPARC and PA-RISC for 64 bit processing.*

3. *Read As Zero, IGNore, Must Be Zero, Should Be Zero : used to specify instruction encoding in a well defined way.*
4. *RISC: very regular format, simple to decode. (The example is Alpha. One more format exists, branch)*
5. *Alpha: 8bit immediate, 16 displ*

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	
alph	253952	57344	222240	533536	

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

1. *data is the initialised data segment, bss is allocated at load time and zeroed.*
2. *Shared libraries can make binaries much smaller—just look at an X application.*
3. *Alpha's heap is also likely to be much bigger.*

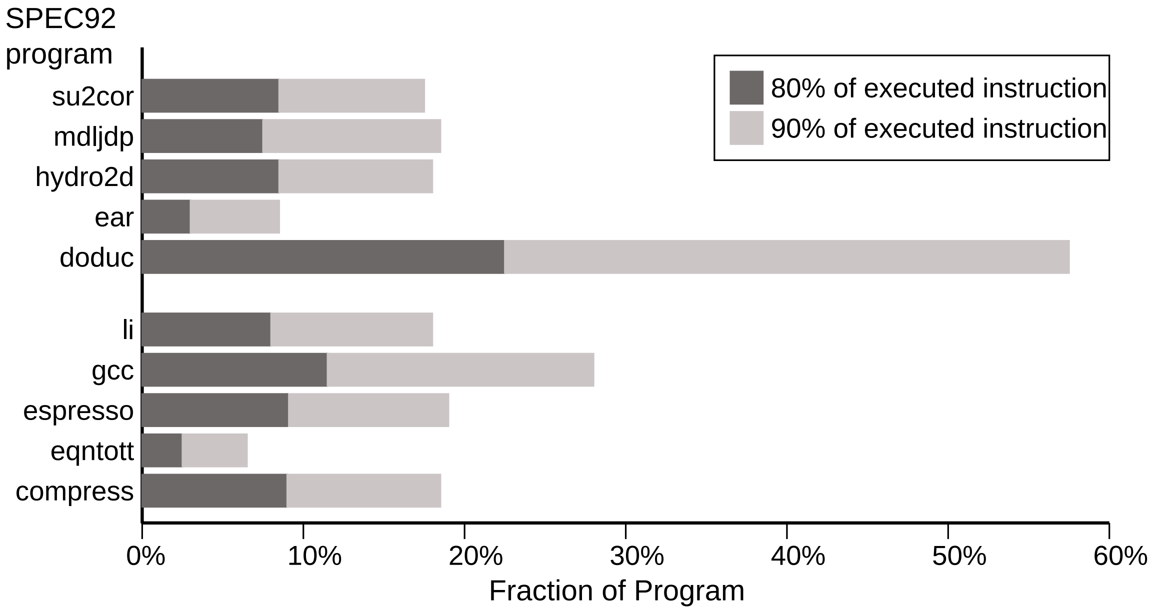
Code Density

- Important if:
  - Memory is expensive
    - ARM Thumb
    - Memory is expensive (in relative terms) in some embedded applications
    - eg. mobile phones
  - Executable loaded over slow network
    - Though Java not particularly dense!
- Speed vs. size optimization tradeoffs
  - loop unrolling
  - function inlining

Code Density

1. *In an embedded application, the memory is often on the same ASIC as the processor. It must fit.*
2. *ARM Thumb decodes 16 bit instructions into normal ARM 32 bit ones which it stores in the on-chip I-cache.*
3. *Java byte code class section compacters/obfiscators (Kaffe)*

Instruction caches

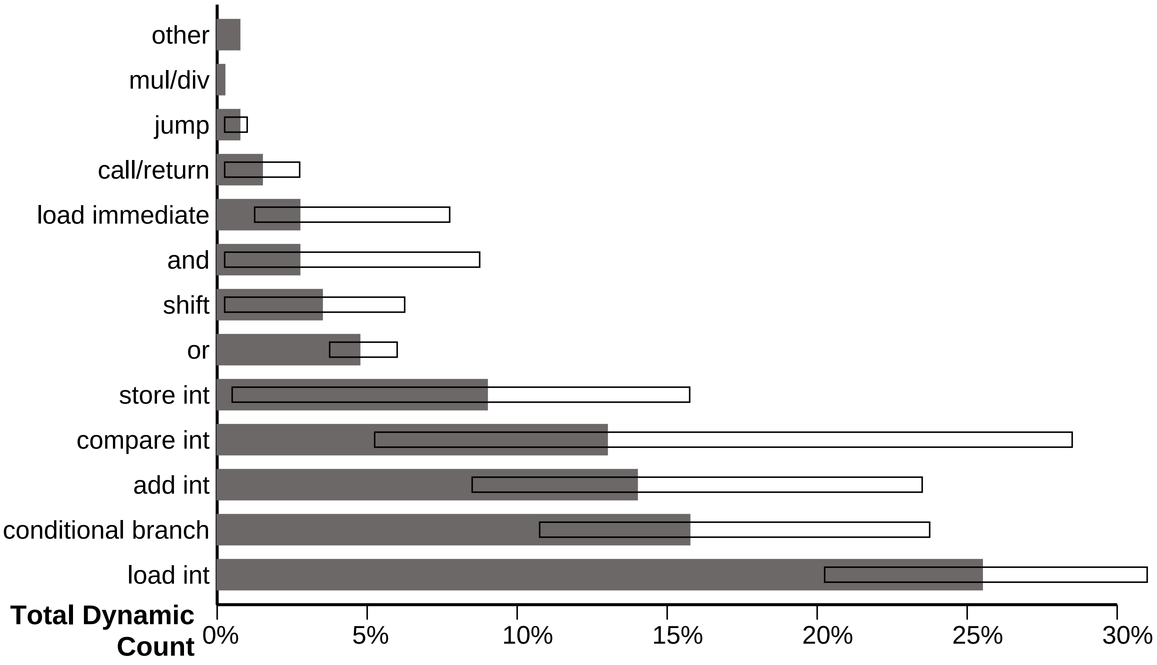


Fraction of program responsible for 80% and 90% of instruction executions

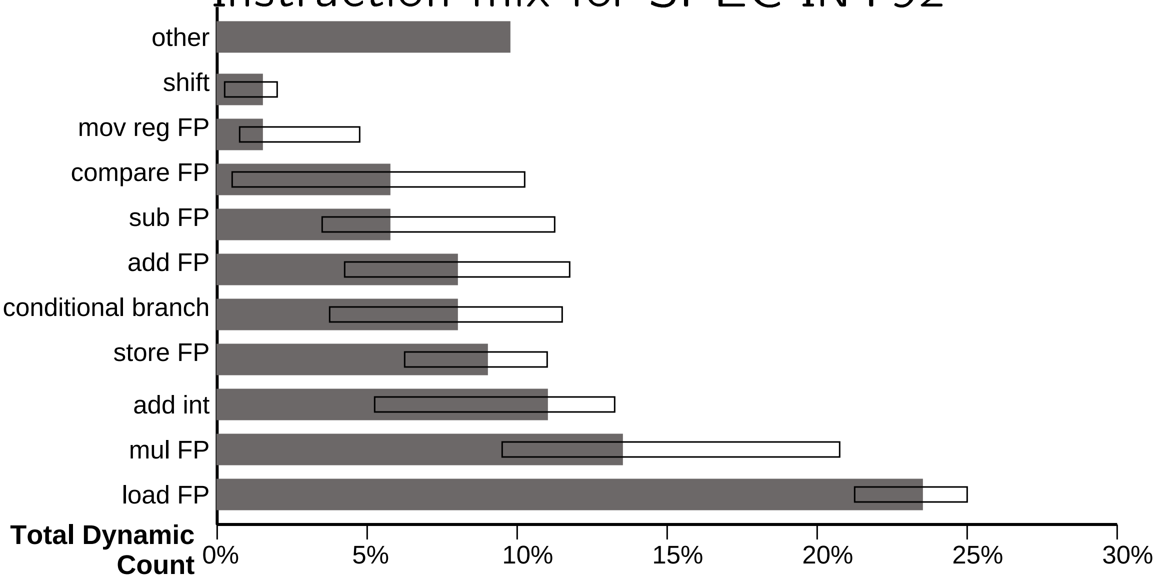
- Caches generally solve I-stream b/w requirements
  - 4bytes x 500MHz x 2-4 instructions = 4-8GB/s !
  - Loops are common! (90% in 10%)
  - Internal I-caches often get 95%+ hit-rates
  - Out-lining and In-lining
  - Code density not usually a performance issue
- D-Cache generally more of a problem

1. *99% hit rate for SPEC92 with 8KB I-cache.*
2. *Some figures by Sites showing that I-caches get very poor hit rates with some commercial applications e.g. MS SQL server. These figures could be greatly improved through use of better compilers/better structured applications.*
3. *D-cache hit rates are usually much lower than I-cache hit rates, and are thus more likely to limit performance. A return to CISC-style instruction encodings is unlikely for general purpose CPUs.*

Instruction Mix



Instruction mix for SPEC INT92



Instruction mix for SPEC FP92

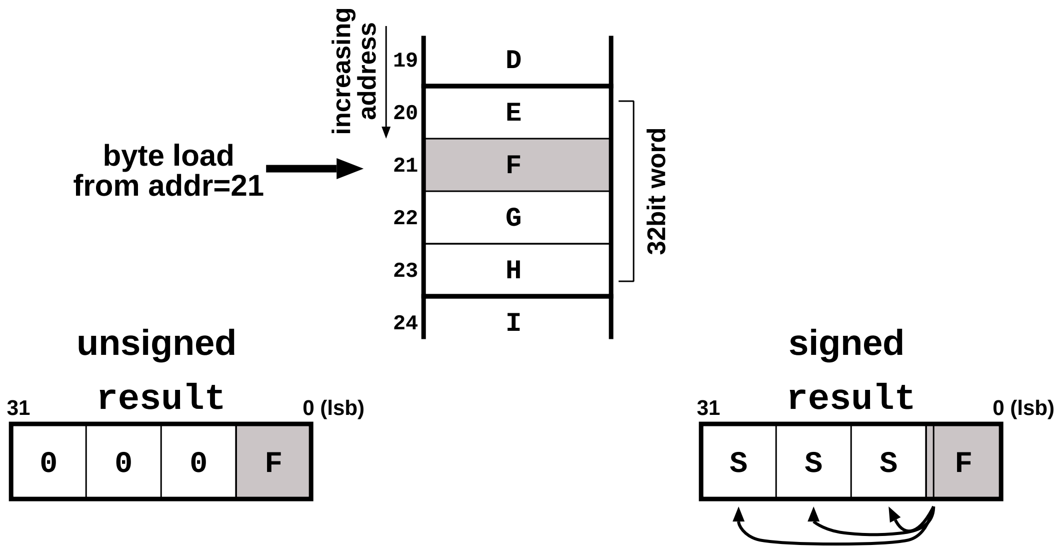
There are no 'typical' programs

Instruction Mix

1. *SPEC92 instruction mix for a MIPS processor (unknown compiler)*
2. *Dynamic vs Static mix - e.g. because compiler often moves multiplies outside of loops, the dynamic mix % will be lower than the static mix.*
3. *There is no such thing as a typical program. There exists a large degree of variance*
4. *SPEC 95's JPEG uses mul a lot - expect to see multiply getting beefed up on new processors.*
5. *loads and stores are frequent. On a CISC architecture, many would be incorporated into arithmetic instructions.*

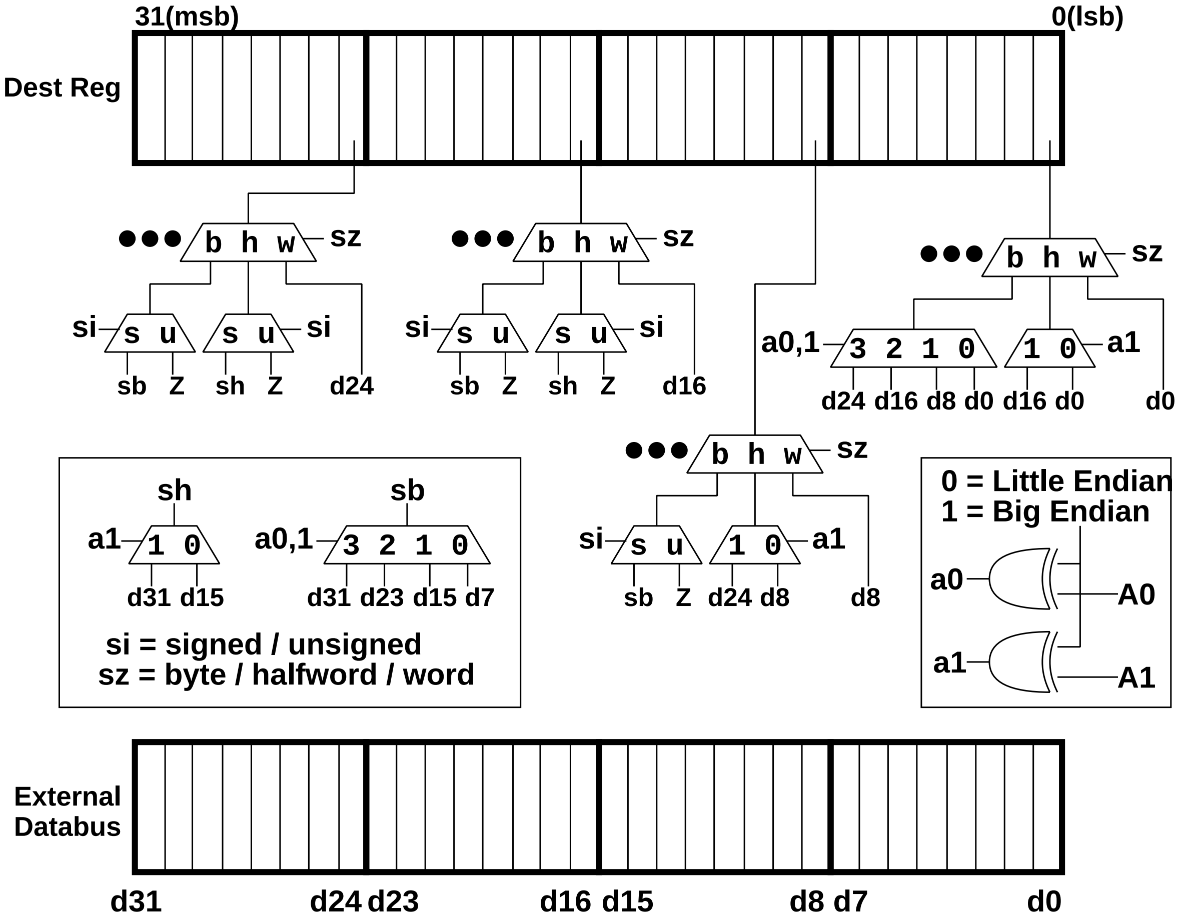
Aligned Loads and Stores

- 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:



1. *All byte accesses are (by definition) aligned.*
2. *On a CISC, loads generally merge into destination*

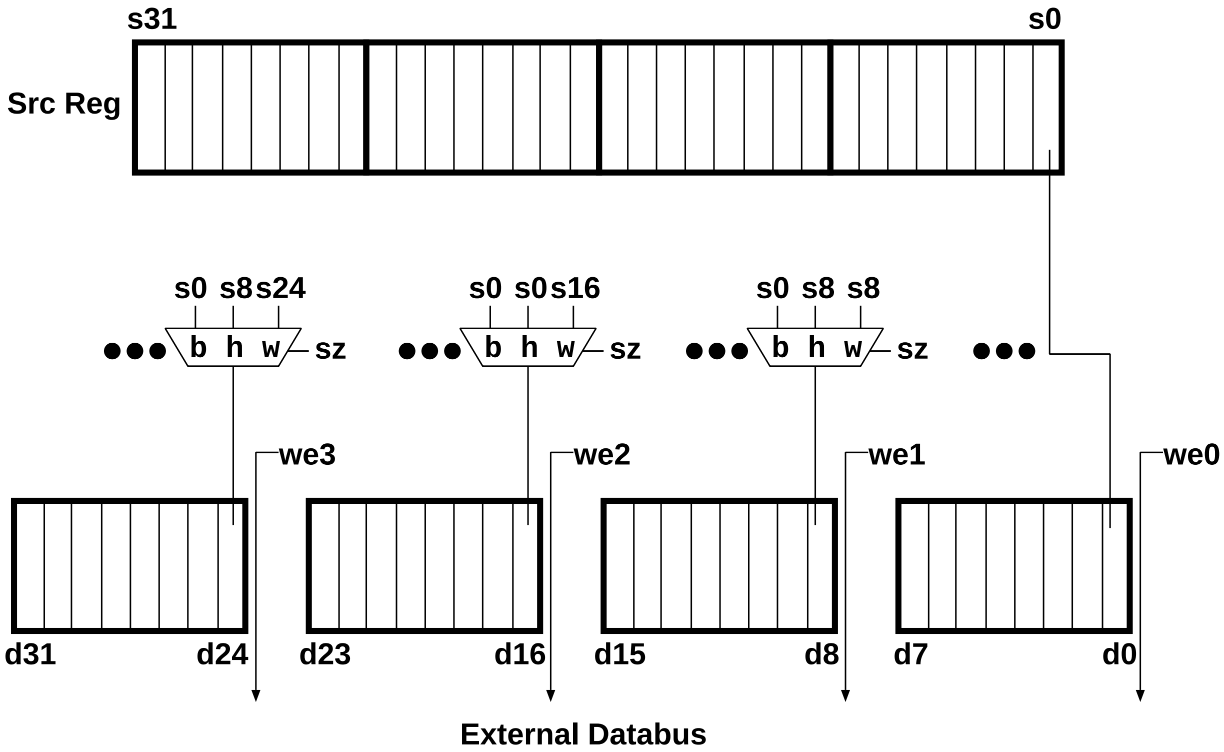
Aligned Sub-word Load Logic



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

1. *These days, data value will probably actually be read from a cache line (16 or 32 bytes long), so require another set of multiplexers to select the word within the line.*
2. *Extra multiplexers will be required to perform sign extension for signed sub-word loads*
3. *Extra multiplexers will be required to perform half-word loads*
4. *In big endian mode, byte loads will capture the most significant byte. NB: I/O devices and other peripherals must be wired up appropriately for this to work correctly as a big endian machine.*

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	1
1	1	b	1	0	0	0

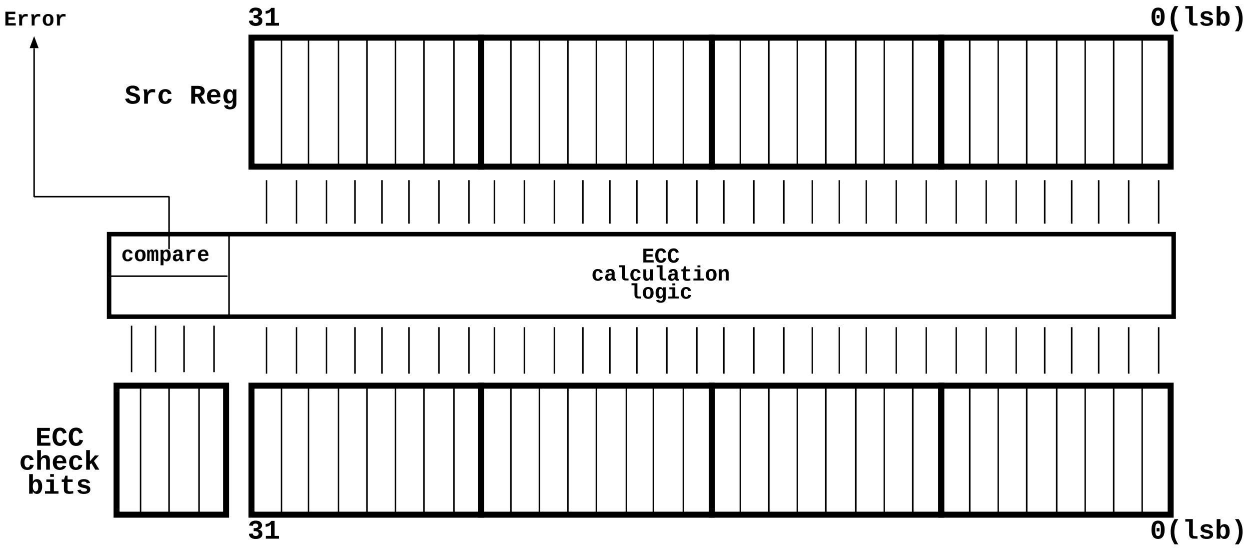
- Replicate bytes/halfwords across bus
- Write enable lines tell memory system which byte lanes to latch

1. *Byte must be steered into the appropriate byte line (or duplicated across the whole bus*

2. *Big/Little Endian select can be done for stores in a similar manner to loads.*
3. *Modified bytes are merged into cache line/memory*
4. *Not all machines have individual byte write enables. Need to read-insert-write.*

Sub-Word Load/Stores

- Word addressed machines
  - Addr bit A0 addresses words
- Alpha:
  - 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



1. *word addressed machines (e.g. super computers) can't directly access values smaller than a word.*

2. *Alpha now has sub-word load/store extensions, but the primary reason for adding these is alleged to be for assisting x86 s/w emulation.*
3. *Whole word has to be written to (easily) calculate ECC bits. Generally requires a read/merge/write-back.*

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

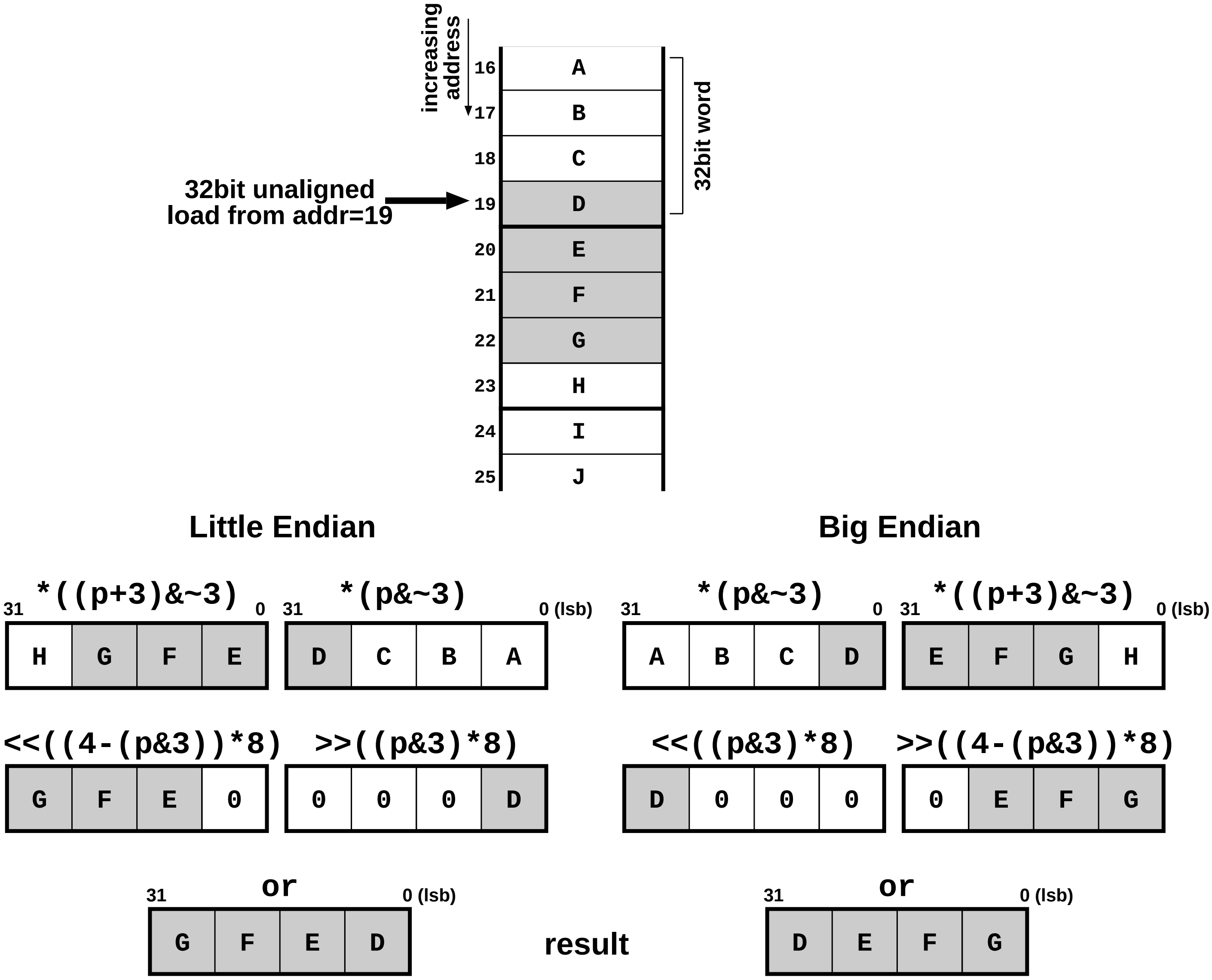
```
long temp;
temp = *(p&(~7));
temp = temp << ( (7-(p&7)) * 8);
reg  = temp >> 56;
```

Emulating Byte Loads

1. *(p & ∼3) can be done with a* `bic` *instruction*
2. *Above is pseudo code. C syntax would requires casts of p to (unsigned long) and then (int *)*
3. *Show how to do it using assembler. Note use of lsr and asr.*

Unaligned Accesses

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



1. *4 byte quantities have four possible alignments : only one is aligned.*
2. *show how this can be achieved in hardware using a 2*32 bit register and 32 four input multiplexers.*

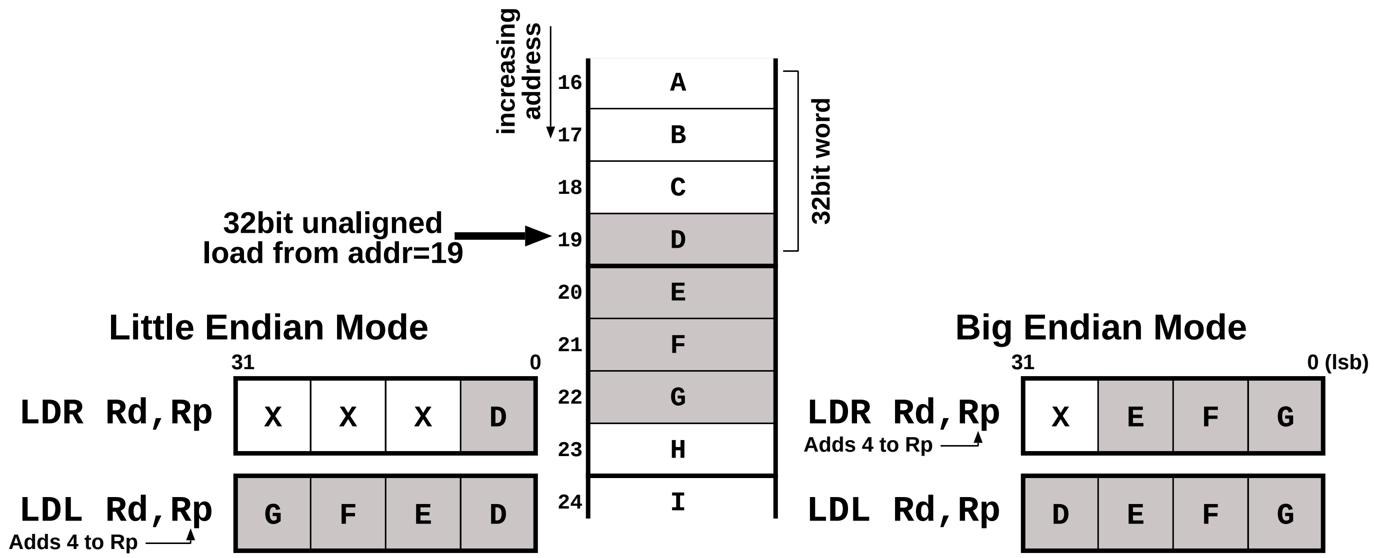
Unaligned Accesses

- 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

1. *PPC unusual.*
2. *Many bugs in older CISC processors to do with dealing with exceptions when crossing page boundaries*
3. *Sub word unaligned accesses may not require two memory accesses*
4. *Some operating systems automatically 'fix-up' unaligned accesses by executing a s/w handler. This is very inefficient, especially as the trap is usually logged to syslog.*

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

1. *LWL/LRL instructions are still quite complicated. Slow the pipeline?*
2. *Normal LW/SW instructions trap if address not aligned.*
3. *See man page in supplied material.*

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 Loads

1. *Simple shift-and-mask instructions.*
2. *Q stands for quad word (8 bytes), L for long word (4bytes)*
3. *Same instructions used for sub-word accesses e.g. unsigned byte load with LDQ_U Rd, Rp; EXTBL Rd, Rd, Rp*
4. *for signed byte: LDQ_U Rd, Rp; add Rp, Rp,1; EXTQL Rd, Rd, Rp; asr #56*
5. *See Alpha man pages (attached)*

Alpha unaligned stores

- No byte hardware, so load quad words, merge, and store back
- `INSQL Rd` $\leftarrow$ `Rs, Rp`
  Shift Rs left by Rp&7 bytes
- `INSQH Rd` $\leftarrow$ `Rs, Rp`
  Shift Rs right by 8-Rp&7 bytes
- `MSKQL Rd` $\leftarrow$ `Rs, Rp`
  Zero top 8-Rp&7 bytes
- `MSKQH Rd` $\leftarrow$ `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
```

1. *Also Long word, word and byte versions*
2. *Unaligned stores are painful without logic to perform sub-word stores. Probably why they've since been added to Alpha...*

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...

1. *memmove: may need to reverse copy direction e.g. dest = src+1*

copy_aligned

- E.g. for 32bit machine

```
void copy_aligned( int32 *d, const int32 *s, int n)
{
  sub n, n, #4
  ble n, return
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
  bge 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

copy_aligned

1. *memcpy function has src as a constant.*
2. *Utilizing delay slot*
3. *Unroll loop to reduce per iteration loop overhead*
4. *Touch the next cache line well in advance of use, thus avoiding a data dependent stall.*
5. *Re-order instructions such that they get issued to functional units without stalls due to structural hazards.*
6. *Highest throughput copy on some x86's can be achieved by using the 64bit data-paths provided by the FP regs. The dedicated MOVS instruction is actually slower! (has to deal with all the special cases...)*

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
  - n mod 32 != 0
- To further optimize:
  - Consider memory system effects
  - E.g. Touch cache lines in advance

1. *Use of stylised C to direct compiler — can help considerably, as current product compilers are not as good as one might wish/expect.*
2. *Will be looking at how to schedule memory accesses later...*

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++;
```

```
}
```

1. *load and store each word once. Align to dest*
2. *use of copy_aligned if (s&3 == d&3)*
3. *big endian version: simply reverse shifts in inner loop*
4. *Consider hardware implementation of copy_aligned*

ISA Summary

- RISC
  - Quantitative Analysis
  - Amdahl's Law
  - Load-Store GPRs
  - ALU operates on words
  - Relatively simple instructions
  - Simple addressing modes
  - Limited unaligned access support
- Architecture extensions
- Copying memory efficiently

1. *So far, haven't looked at branches. Different kinds of branch instruction have a big impact on control-flow, and are examined in the next section.*

CPU Performance Equation

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

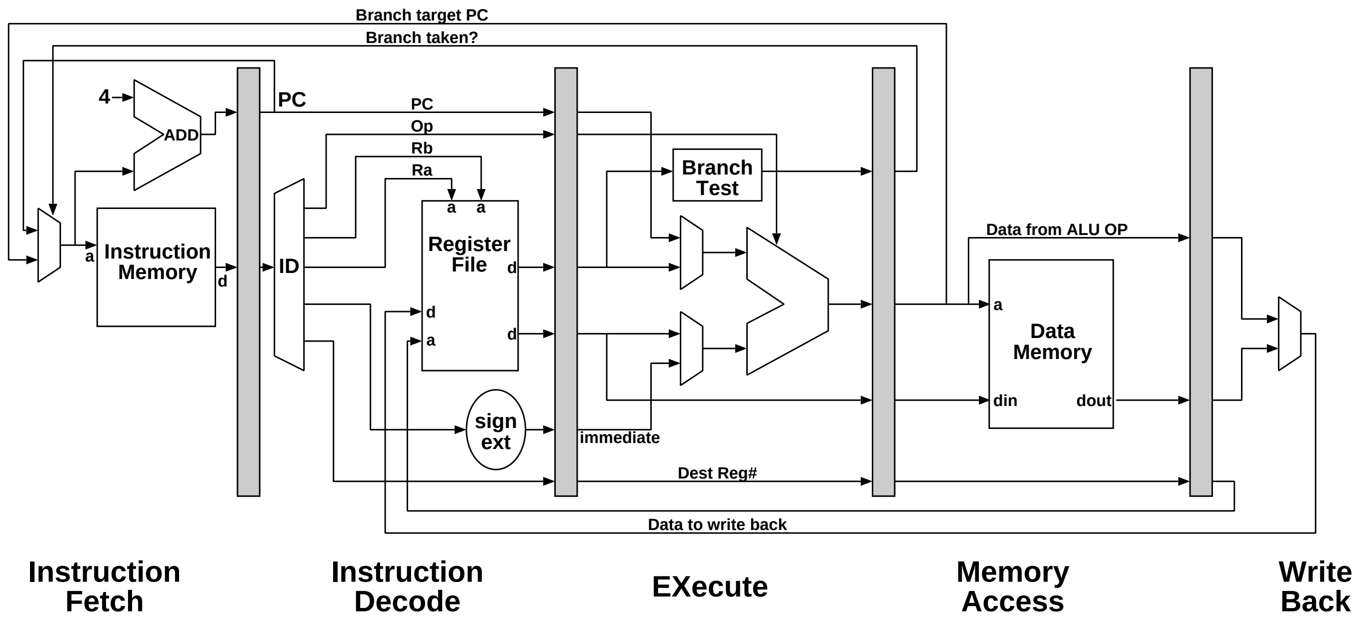
C =Cycles per instruction
T =Time per cycle
I =Instructions per task

- 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 the available Instruction Level Parallelism to exploit

CPU Performance Equation

1. *RISC: reduce C and T. PowerPC, ARM & PA-RISC have lower I than most*
2. *super-pipelining: instruction completion latency longer, thus more forwarding paths required*
3. *super-scalar: Exploits Instruction Level Parallelism*
4. *super-scalar: Issuing 1fp and 1int ALU instruction is easy, as there are no inter-dependencies. Issuing multiple int or fp is harder.*
5. *Most modern high-performance processors are super-scalar. The challenge is to build machines capable of searching far ahead into the instruction stream to resolve all the instruction interdependencies and seek out the available parallelism.*

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 classic RISC pipe

1. *Basically DLX from H&P's book, with a few mods where it suits my purposes*
2. *Very simple instruction decode. Ra and Rb get accessed regardless of whether they are needed or not.*
3. *Register file has 2 read and 1 write port*
4. *Write Back always occurs at same point for simplicity*

The cost of pipelining

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

⇒ Effectiveness determined by CPI achieved

Pipelining is efficient

1. *Modern machines need many ports on register file e.g 5R/3W*

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 (μops)
  - AMD x86 K6 (r-ops)

Easiest if all instructions do a similar amount of 'work'

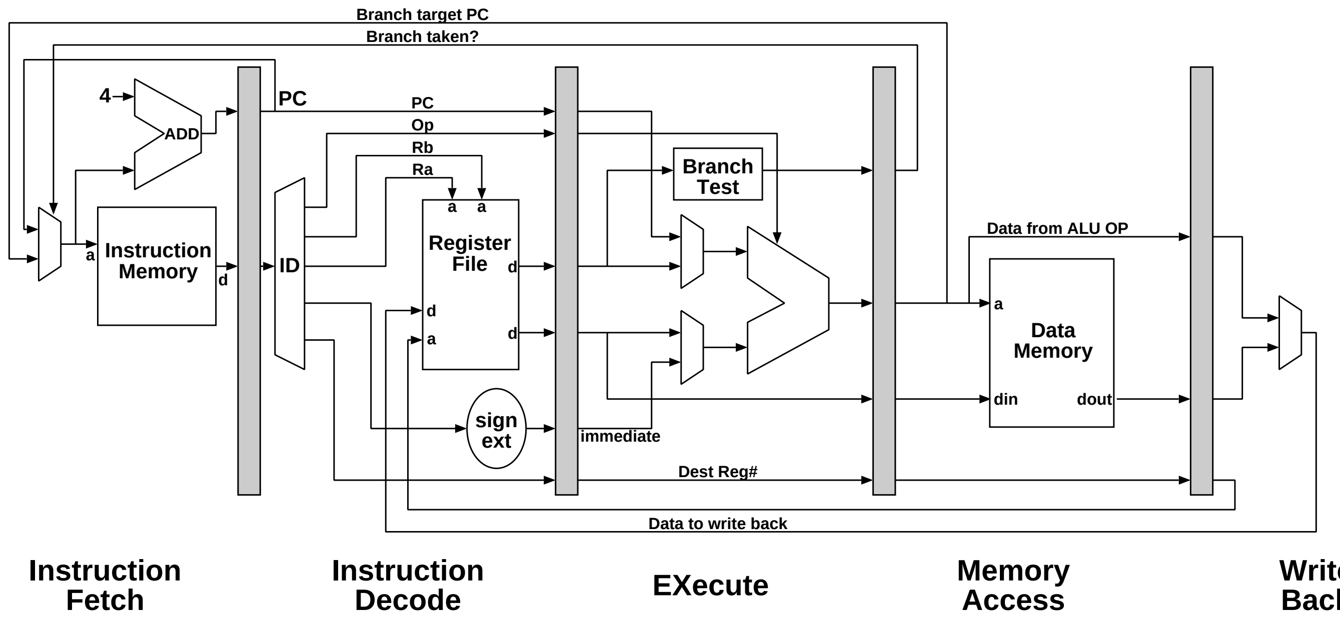
Non Load-Store Architectures

1. *A single-pass pipeline capable of executing a 3 memory operand instruction (eg VAX) would be long and inefficient*
2. *Each x86 instruction can only reference one memory location. On the Pentium, the address is calculated using a dedicated adder in one of the two instruction decode stages. The second instruction decode stage and the write back stage can access the data cache as well as the register file.*
3. *All modern x86 implementations use simpler instructions internally. A decode unit translates the architectural instructions into RISC-like internal instructions, that execute using a RISC-like execution engine. For some particularly complicated x86 instructions, microcode is used to synthesize the necessary internal instruction sequence (eg FP transcendental functions etc).*
4. *Different implementations of the same CISC architecture can use different internal instruction sets. (PPro uses 'u-ops', AMD K6 'r-ops')*
5. *The extra decode stage adds latency to instruction execution, but modern dynamic Out-Of-Order execution engines are relatively effective in hiding this.*
6. *See the Pentium Pro white paper (additional material)*

ALU Result Forwarding

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

  a: add r1 ← r8, r9
  b: add r2 ← r1, r2
  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

ALU Result Forwarding

1. *Draw on the 4 forwarding paths*
2. *Too many MUX inputs will slow cycle*
3. *Could move WB stage forward for ALU ops, but create structural hazard*
4. *R4000 8 forwarding paths just for ALU ops*
5. *Adding too many pipeline stages can be counter productive.*

Load Data Hazards

- Impossible without a stall:

```
lw r1 ← r9(4)
add r2 ← r1, r6
```

- 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
    - Used by issue logic
  - e.g. R4000 and most others

- More stalls for deeper pipes
  - 2 stalls and 2 more forwarding paths for R4000

Load Data Hazards

1. *Draw on forwarding path from WB/dout to ALU a and b*
2. *Re-ordering to avoid stalls is generally more tractable on FP programs*
3. *Maximise number of cycles between load issue and use – modern CPUs won't stall until they actually need the data. If access misses L1 cache, many spare cycles will be needed. Dynamic super-scalar processors (described later) like the PPro are able to search ahead past the instruction that uses the loaded value in order to find other non-dependent instructions which can be executed.*
4. *Register scoreboard tracks the status of all the registers – which are ready to read, which can be forwarded, which are not ready to read yet (issue logic must stall an instruction requiring a register that is not ready)*
5. *Scoreboard is used to determine when instructions can be issued down the pipe, and where they will collect the operands from.*
6. *Even to get 2 stalls for R4K requires speculative use of data returned from D-cache before tag check to see if valid/correct*

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
  - Read after Write hazards more likely
    - compiler instruction scheduling
  - Instruction complete out of order
    - Write after Write hazards possible
    - Dealing with interrupts/exceptions
- Often hard to insert stalls after ID stage

Longer Latency Instructions

1. *Multiply latency of 6-16 cycles was common 5 years ago. Now, more silicon has been thrown at the problem...*
2. *No need to fully pipeline a unit unless it is going to be needed by a number of instructions that are in close proximity in the code. Divides occur in 'bunches' relatively infrequently, so there is no need to pipeline the unit.*
3. *21164 multiply unit has 8cycle latency, and can accept a new operation every 4 cycles.*
4. *21264 multiply unit has 4cycle latency and is fully pipelined*
5. *Each functional unit either needs its own write port on the register file (inefficient use of resources), or the issue logic should insure that the WB stage is not in use by the main pipe. This can be achieved either by stalling the pipe at the MEM stage (hard), or by dropping a NOP into the pipe after the ID stage. The NOP will pass through the pipe and will arrive at the WB stage just when the other functional unit wants to use it.*
6. *Read after Write hazards: The compiler should re-order code in order to insert cycles between when an instruction is issued to a high-latency unit, and when the result is first used.(Just as for loads). The register scoreboard will ensure the dependent instruction is held in the ID stage until the result can be made available (send NOPs down the pipe in the meantime).*
7. *Write after Write and Write after Read hazards: Issue logic has to be careful to avoid re-ordering writes to the same register, and to ensure that writes cannot overtake reads. This can be a particular problem if the pipeline contains multiple register write-back points (eg different latency functional units), or if reads can occur late in the pipe (eg.*

the source operand for a store). Some Write after Write hazards can be solved by aborting the first instruction.

8. *This is not “full-blown” out-of-order execution, as all instructions are still issued in order. (dynamic execution is described later)*
9. *exceptions occurring within the pipe can be tricky.... (next slide)*

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
    - branch delay slots
  - Roll back state/In-order commit (PPro)
- Imprecise arithmetic exceptions
  - Exception raised many cycles later
  - Alpha: Trap Barriers
  - PPC: Serialise mode

Exceptions and Pipelining

1. *Illegal instruction: ID stage, so still in-order*
2. *Instruction fetch TLB miss: in order, so no problem*
3. *Data TLB miss/unaligned access: easy on the 'classic RISC pipe', as no other instructions have been committed to register Write Back yet.*
4. *Divide by 0 can usually be detected quite early*
5. *Arithmetic overflow (int and fp pipes): Can be tricky, particularly if they are generated by instructions executing in functional units with long latencies—many subsequent instructions may already have been issued. INT Mul and Divide, FP instructions*
6. *Processors supporting exact exceptions need to appear as though they are executing instructions serially, one after the other. When a fault occurs, the exception handler is given the PC of the instruction that caused the fault. As far as the user is concerned, no instructions after the faulting instruction were executed. On modern processors, this is not what happens in reality — other instructions will have been executed, and there effects must be undone. The PPro achieves this by committing updates to the user-visible architectural state (the register contents) in-order. Care must also be taken to ensure that exceptions are raised in-order too. Exceptions should not be raised from instructions that should not have been executed.*
7. *Watch out for exceptions occurring in Branch delay slots! Need to re-execute the branch.*
8. *Newer RISC architectures have an imprecise exception model for dealing with arithmetic exceptions. When a fault is detected, rather than appearing to stop on the faulting instruction, such architectures simply stop issuing new instructions. Instructions already in the pipe run to*

completion as normal. The exception is then raised, giving the PC on which execution was stopped as the fault address. The number of instructions between the faulting instruction and the point where execution stopped is potentially undefined. Some of these instructions may have been dependent on the result of the faulting instruction. They may be executed with an undefined value as one of their inputs. This can cause havoc eg. causing an invalid memory location to be accessed, or causing a branch to go to the wrong place.

9. *Some programs rely on being able to trap and fix-up arithmetic exceptions. This can still be achieved on Alpha and PPC despite their imprecise exception models.*
10. *Alpha has the TRAPB instruction, which prevents all following instructions from being issued until all previous instructions have advanced beyond the point where they could possibly generate an arithmetic trap. TRAPB instructions should be sprinkled around the code to ensure that exceptions are generated early enough such that the fault can be fixed-up. i.e. no destruction of important register or memory contents has occurred in following instructions, the faulting PC can still be worked out (the program may have branched or taken another exception in the meantime).*
11. *In contrast, PPC can be put into a 'serialize' mode, where it generates exact exceptions by being far less aggressive about overlapping instruction execution.*
12. *Operating out of imprecise exception mode can easily loose a factor of 10 in performance.*
13. *On a machine that ISSUES instructions out of order, all exceptions are potentially tricky. (though in actual fact get dealt with by the same h/w mechanism that allows speculative execution across multiple branches)*

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

1. *interrupts and SWI traps are no generally no problem on RISC – treat like a JUMP instruction being added to the very front of the pipe. Interrupt latency will be determined by how long it takes everything already in the pipe to complete.*
2. *Some CISC instructions would take too long to complete - need to interrupt. But, some instructions can not simply be re-issued from the beginning, as they have had an information destructive effect (an overlapping copy would have this effect). This can be solved if the instruction is defined to update the contents of it's source operands to reflect its progress. It can then simply be re-issued, and will continue where it left off.*
3. *Reset: similar to an interrupt.*

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
- 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
    - CISC: stack
    - most RISC: special register
    - ALPHA: nominated register

1. *(x86 calls all control flow changing instructions jumps)*
2. *Jump subroutine more common than branch subroutine*
3. *Some architectures have conditional jumps as well.*

Conditional Branches

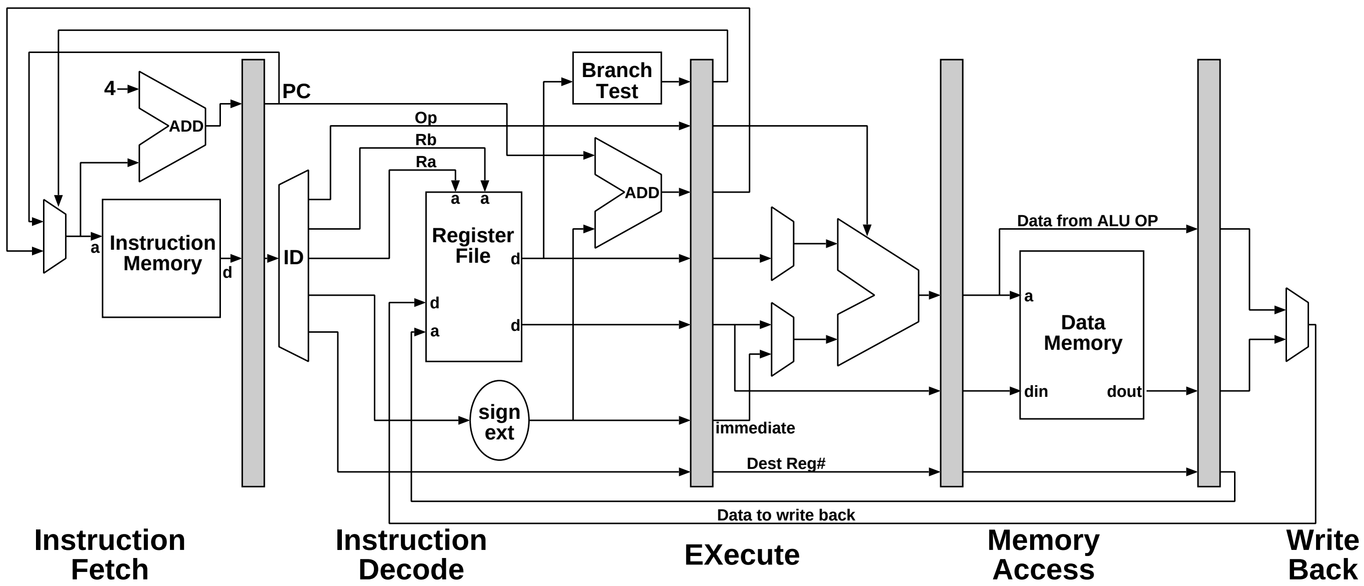
- Conditional branch types
  - most: Test condition code flags
    - Z, C, N, V
    - `Bcc label`
  - Alpha/MIPS: Test bits in named reg
    - msb (sign), lsb, 'zero'
    - `Bcc Rx, label`
  - some: Compare two registers and branch
    - `Bcc Rx, Ry, label`
    - (PA-RISC, MIPS, some CISC)
- Conditional 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)

1. *Alpha/MIPS branch instructions can test for the following conditions (and combinations thereof): low bit (set/clear);*

high bit (positive/negative); value=0 (special zero bit stored with each register)

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 `Bcc Ra, Rb, label`



1. *register bits: b0, top bit (N), Zero (Implement this by adding an extra bit to each register that indicates whether the current value is zero. This bit should be generated by the ALU and treated like any other register bit, i.e. all forwarding paths should be 33 bits wide)*
2. *On classic, need to abort the following two instructions is branch taken. (Effectively predicts all branches as not taken. However, statistics show the majority of branches are taken)*
3. *compare and branch: PA RISC, MIPS (equal)*

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), or GPR

- Branch delay slots avoided the taken branch stall on early MIPS

  – Always execute following instruction

  – Can't be another branch

  – Compiler fills slot ∼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, and fewer cycles between branch instructions

Control hazards can cripple multi-issue CPUs

1. *set flags early: compiler should be instruction scheduling.*
2. *better to have* > 1 *CC reg (PowerPC), or use GPRs instead.*
3. *Branches are common. multi-issue machines will come across them more quickly, and the penalties are typically worse due to the longer front-end to the execution pipe. Control flow hazards could ruin the CPU's CPI. Need to use prediction/speculation.*

Static Branch Prediction

- Speculation should not change semantics!
- Simple prediction
  - e.g. predict backward taken, forward not
- Branch instructions with hints
  - Branch likely/unlikely
  - Use Feedback Directed Optimization
  - Fetch based on hint
- Branch instructions 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

1. *must be careful about changing semantics: e.g. register writes, stores, exceptions, wasted mem b/w*
2. *If backward branches are predicted taken, forward not, the compiler should endeavour to structure code to assist this.*
3. *likely/unlikely: If loop, likely. Or use FDO for IF/ELSE clauses*
4. *FDO: Many branches turn out to be highly biased one way or tother. Enables miss prediction rates of less than 20%*

5. *hints with annulment: Hint hopefully is usually correct, so do useful work instead of stall. Compiler should still prefer to use non nullifying*

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
- Branch history method
  - Single bit predictor
    * Bit predicts branch as taken/not taken
    * Update bit with actual behaviour
    * Gets first and last loop iterations 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

1. *Some branch prediction caches are actually implemented within the I-cache, where a number of prediction bits are stored with each cache line.*
2. *Aliasing of entries in the prediction cache will not effect correctness, but will hurt performance.*
3. *2bit predictors: Don't work with branches that are alternately taken/not taken (e.g. odd/even iterations). Some CPUs e.g. the PPro/PII use 3 bit predictor schemes that can cope with this.*

Dynamic Prediction :2

- Global correlating predictors
  - Behaviour of some branches can be predicted by observing behaviour of the last few branch instructions executed
  - Shift register to hold processor branch history
  - E.g. Two bit correlating predictor:
    * 2 bit shift register
    * 4 prediction bits in each cache entry, one for each possible history

- Combination of history and correlating predictor works best
  - $\geq$95% for 4K entry buffer on SPEC92
  - E.g. Alpha 21264

1. *Multiple if statements often have the same variables in their clauses*
2. *Branch may or may not be part of its own global history*
3. *21264: has a couple of bits to select between the two schemes dynamically on a per branch basis*

Avoiding branches

- Count Register
  - Decrement and branch
  - PowerPC, x86
  - Perfect prediction
  - Only available for inner most loops
- Predicated Execution (e.g. ARM)
  - Inline simple if/else statements
  - Execution of all instructions is conditional
  - Current ARMs only skip one instruction per cycle
  - Instruction 'boosting'
- Conditional Moves
  - Alpha, new MIPS, new on PPro
  - Move if flags/nominated reg set
  - Provides a 'commit' operation—beware side effects on 'wrong' path
  - PA-RISC supports arbitrary nullification

Avoiding branches

1. *Count Register: bit of a throw back to Extended Accumulator*
2. *Count register is decremented by DECB instruction. CPU knows exactly when loop will terminate, and can thus predict the exit correctly. There is only one count register, so it can only be used for inner-most loops (where it really matters).*
3. *predicated execution consumes 4 or the ARM's opcode bits*
4. *predicated execution allows compiler to speculate e.g move instructions in predicted path ahead of the branch. (Sometimes referred to as instruction boosting)*
5. *CMOV: provides similar sort of functionality to predicated execution, but does not cost opcode bits. Both paths of a simple if/else statement can be evaluated, and then one committed. CMOV is not as flexible as predicated execution. Care has to be taken not to cause side effects on the 'wrong' path e.g. stores etc.*
6. *CMOV instructions can avoid some control flow problems, turning them into data flow issues further down the execution pipe.*

Optimizing Jumps

- Alpha: Jumps have static target address hint
  - A_{16-2} of target instruction virtual address
  - Enough for speculative I-cache fetch
- Subroutine Call Returns
  - Return address stack
  - Alpha: Push/pop hints for jumps
  - PowerPC: Dedicated link register
  - 8 entry stack gives $\geq$ 95% for SPEC92
- Next fetch predictors
  - Very simple, early prediction to avoid fetch bubbles
  - E.g. PPro, A21264
  - I-cache lines have pointer to predicted next fetch
- Branch target buffers
  - Like branch prediction caches, but provide a target address in IF stage

1. *Alpha jumps: Linker fills in hint same time it fills in word containing real procedure address*
2. *Alpha return addr stack: only enough bits to address I-cache*
3. *Branch prediction doesn't provide a target PC early enough to avoid a wasted cycle. Next fetch predictors are a simple mechanism to avoid bubbles in the I-fetch stream. The next-fetch predictor is updated based on the 'correct' behaviour. As such it will learn the targets of computed jumps e.g. to DLLs (shared libraries) or function pointers.*
4. *Branch target buffers: for branches that are predicted taken, store a next-fetch target*

Super-Scalar CPUs

- # execution units (INT/FP)
  - Pentium (2/1), Pentium Pro/PII (2+/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, Pentium Pro 3μops
  - 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)
  - Dynamic out-of-order execution (21264, PPro)

Super-Scalar CPUs

1. *Function units are specialised according to the expected instruction mix.*
2. *PPRO: 5 more specialised units (2 general int ALU, 2 FP, 1 load/store unit [capable of 1 load and 1 store per cycle])*
3. *The Pentium has two pipelines, U&V. The V pipeline can only execute a subset of (common) operations.*
4. *Achieving the max issue rate requires the right instruction mix, avoiding structural, control and data-flow hazards. In practice, it is rarely sustained. Dynamic execution is better at achieving this than static in-order.*
5. *The current wave of high-performance microprocessors all use dynamic execution.*

Static Scheduled

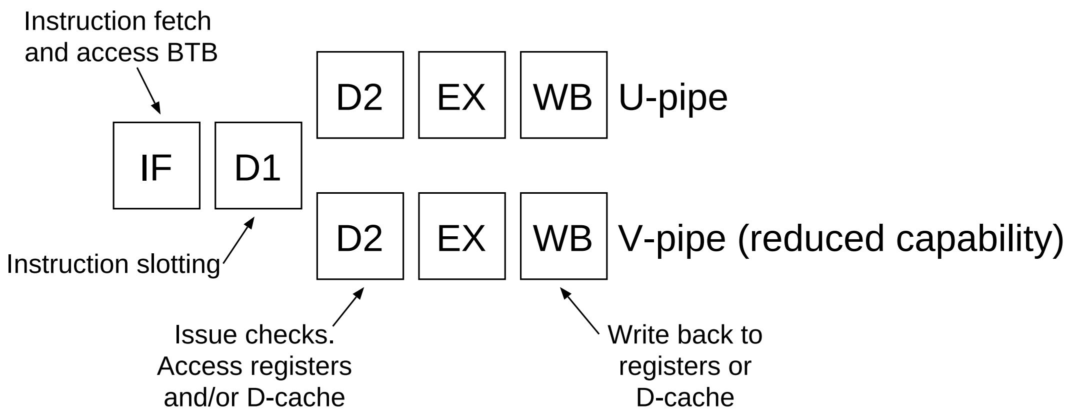
- All instructions begin execution in-order

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

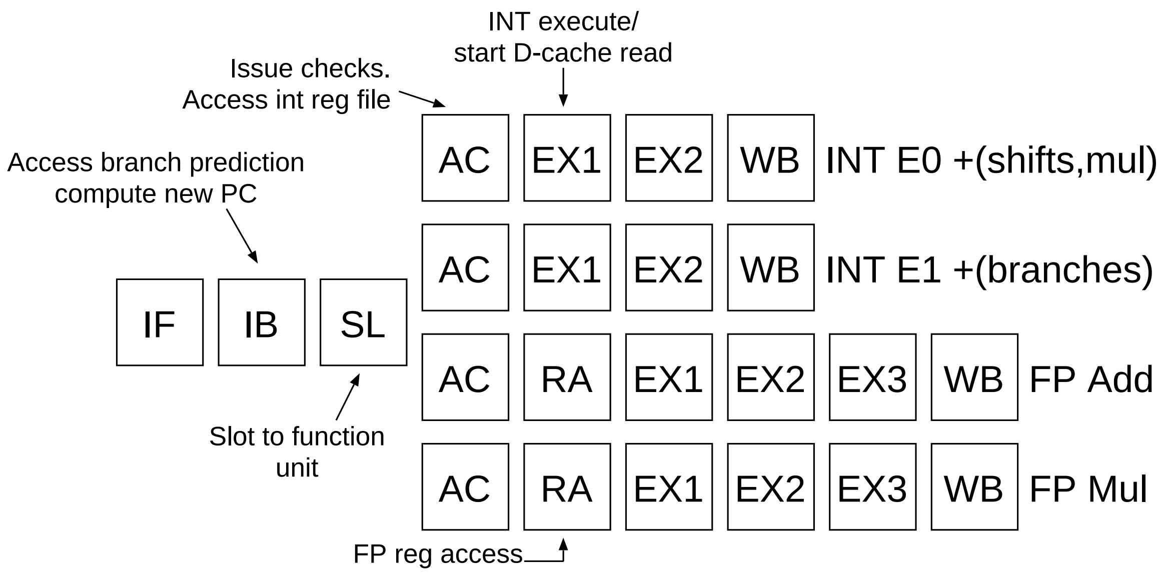
Static Scheduled Super-Scalar

1. *Fetch and decode a block of instructions e.g. the 21164 reads a block of 4 16 byte-aligned instructions.*
2. *Some function units will not be fully pipelined, so will not currently be available for use. Some units are not capable of executing all instructions e.g. the Pentium's V pipe, or the 21164's E1 pipe.*
3. *issuing an Int and FP operation simultaneously is easy, since they generally use different register sets. Doing multiple int is more challenging.*
4. *independent instructions have no producer-consumer (RaW) or producer-producer (WaW) conflicts. If such conflicts exist, issue of the later instruction must be delayed. [WaR conflicts are typically not a problem, except if a stores read their source late in the pipe.*
5. *Non-blocking loads allow instruction issue to continue until the first instruction that references the register (as a source or dest)*
6. *typically, only one control-flow instruction is dispatched in a single cycle. Branch prediction is used to drive the instruction fetch, and to decide whether to discard any subsequent instructions in the current block.*

Static Scheduled Examples



Pentium pipeline



Alpha 21164 pipeline

1. *See additional material*

Static Scheduled Super-Scalar

- Relies greatly on compiler
  - Instruction scheduling
    * slotting
    * data-dependence
  - Reduce # branches and jumps
    * unroll loops
    * function inlining
    * use of CMOV/predicates
  - 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 now which loads will miss
  - Feedback Directed optimization can help

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

1. *Branches should ideally be at the end of a fetch block. If fetch blocks need to be aligned (e.g. 16 byte aligned on 21164), branch targets should ideally be the first instruction of the*

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
  - PPro up to 30 instructions, 21264 up to 80

- Need to speculate 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

Dynamic Scheduling

1. *Whereas a static scheduled processor stops issuing instructions as soon as one instruction stalls, a processor with dynamic execution will continue past the stalled instruction and find other non-dependent work to do. This is why they achieve a lower CPI than a statically scheduled CPU with a similar number of function units.*
2. *On wide-issue machines, it is very difficult for the compiler to find instructions to avoid interlocks with load delays or producer-consumer latencies.*
3. *Cost of a miss-predicted branch depends on how many instructions past the branch have been issued.*
4. *Letting the hardware schedule instructions rather than compiler*
5. *May eventually come to a halt due to exhausting ILP in window*
6. *Instruction timing charts just not possible...*
7. *Speculate: 20% branch (CMOV), multiple branch, need damn good predictor or wasted work - worse, cache misses*

Register Renaming

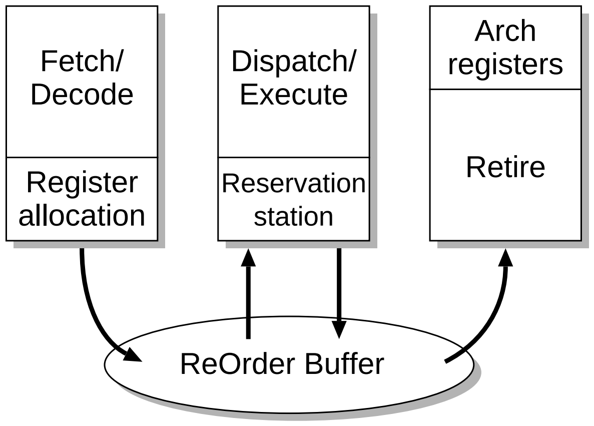
- Register reuse causes false dependencies
- (Often referred to as name-dependencies)
  - WaR, WaW: no data transfer
- Undo compiler's register colouring
- Necessary to unroll loops

⇒ Register renaming
  - Large pool of internal physical registers
  - PPro 40, 21264 80+72
  - New internal register allocated on every write
  - Re-mapper keeps track of which internal register instructions should use for source operands

- Architectural registers updated when instructions retire

1. *Compilers use live variable analysis to enable them to re-use registers for different variables. A dynamically scheduled processor needs to use register renaming to undo the register colouring, separating the variables into different internal registers so that the false dependency is eliminated.*
2. *Without register renaming loops could not be unrolled due to name-dependencies*

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 (If multiple instructions are ready use FIFO order)

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

1. *Stop issuing if there are no ROB slots, or no free registers*
2. *Try to execute older instructions first*
3. *There are lots of variations on the above algorithm dependent on what data structures are actually implemented. E.g. the 21264 still has a 'traditional' file of internal registers. In contrast, the PPro forwards results from instructions direct into the reservation stations and re-order buffer. (See H&P and addition material in notes.)*

Loads and Stores

- Dynamic execution helps hide latency of L2/L3 cache
  - Find other work to do in the meantime
  - Allow loads to overtake loads
- Stores cannot be undone
  - Execute in-order in Retirement stage
- Loads that overtake stores must be checked to see if they refer to same location
  - Address of store may not yet be known
  - May need to speculate load and check later
- Future processors will need to be more aggressive about load/store re-ordering, and thus require better alias analysis
  - Stack / Global variables solvable
    * `[r12,4]`, `[r12,16]`
  - Heap refs harder to disambiguate
    * `[r2,8]`, `[r4,32]`

1. *A miss to main memory takes so long that its highly unlikely that the processor will be able to hide all the latency.*

2. *Allowing loads to overtake other loads and stores is important for performance.*
3. *Current processors don't re-ordering stores, or let stores pass loads. In order to extract more ILP future processors will have to consider these issues.*

Dynamic Execution

- Less dependent on compiler
- Better at avoiding wasted issue slots than static scheduled
- But, dynamic execution uses a lot of transistors
  - ReOrder Buffer and Reservation Stations are large structures incorporating lots of Content Addressable Memory
  - Tend to be $O(N^2)$ in complexity
  - Tend to be on critical path...
  - 25%+ of chip area
- Factors effecting usable ILP
  - Window size
  - Number of renamed registers
  - Memory reference alias analysis
  - Branch and jump prediction accuracy
  - Data cache performance
- The 'perfect' processor: 18-150 instructions per cycle for SPEC92

- 10 way for int progs feasible, more for FP

1. *ILP - Instruction Level Parallelism*
2. *The perfect processor assumes infinite window size, infinite number of renamed registers, perfect mem ref analysis, perfect prediction and a 100% D-cache hit rate.*
3. *A real 10 way INT processor would need something like: 100 instr window, 100 registers, stack and global mem ref analysis.*

Other Techniques :1

- Very Long Instruction Word (VLIW)
  - Each instruction word contains fields to specify an operation for each function unit
  - ✔ CPU doesn't need to worry about issue-checks
    - ∗ High clock speed
  - ✘ Relies heavily on compiler
  - ✘ Sparse I-stream (lots of nops)
  - ✘ Exposes processor internals
    - ∗ No binary compatibility

- Intel 'Explicitly Parallel Instruction Computer' (EPIC)
  - IA-64, Merced
  - Opcode format contains bits to indicate groups of independent instructions
  - Compiler collects instructions into independent groups
  - ✔ Reduces issue-check frequency for CPU
  - ✔ Retains binary compatibility

1. *CPU can issue all of the sub-instructions without issue checks—the compiler has guaranteed them to be independent.*
2. *VLIW now somewhat out of fashion, except with DSPs and alike. This is perhaps due to the importance of binary compatibility, and realization that static scheduling by compilers is hard.*

Other Techniques :2

- Diminishing returns for further effort extracting ILP from a single thread?
- Multi-threading
  - Interleave execution of instructions from (a large number of different program threads
  - ✔ No instruction dependencies
  - ✔ Hides memory latency
    - ∗ No expensive caches
  - ✘ Requires multiple register files
  - ✘ Progress of an individual thread is slow
    - ∗ Poor SPEC marks
- Multiple CPUs on a chip
  - Exploit thread and process level parallelism
  - ✘ Need correspondingly bigger caches and external memory bandwidth

1. *Multi-threading relies on having lots of thread-level parallelism in the application. Auto-parallelising compilers are supposed to seek this out and create light-weight threads. The TERA machine supports many threads and executes one instruction from each in turn. In contrast, the SPARCLE processor (not a true M-T) concentrates on a single thread until an L1 cache miss, which causes it to change thread.*

Other Techniques :3

- Simultaneous Multi-threading
  - Work on a small number of threads at once, aiming to keep all function units busy
  - ✔ Progress of individual threads is pretty good
  - ✔ Cooperating threads may have smaller cache footprint than independent ones
- 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* dynamic execution)

1. *Simultaneous Multi-Threading is rumoured to be used in the next generation Alpha*

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 $Cost_{Miss}$

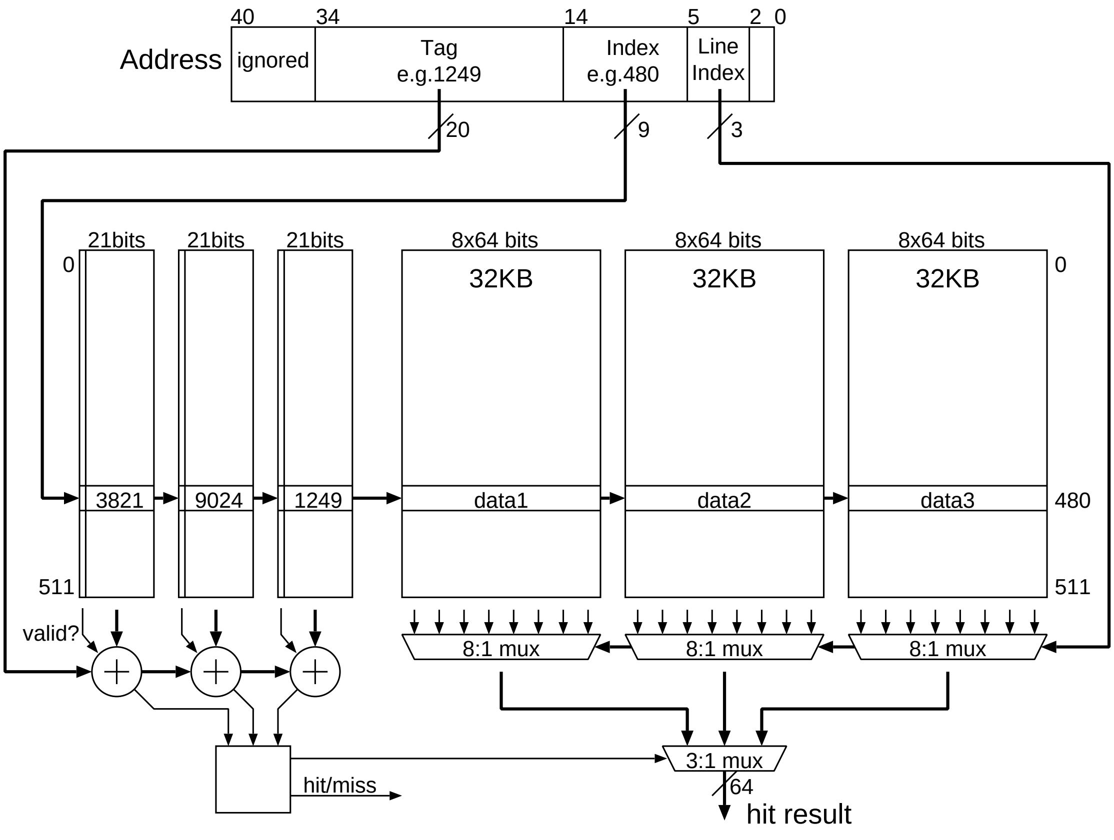
- Build using fast (small and expensive) SRAM
- Tag RAM and Data RAM

1. *Spatial locality is exploited through fetching a block of data rather than the word requested.*
2. *Compulsory misses occur when a block is requested for the first time. The number of compulsory misses is independent of cache size and associativity. However, increasing the line size may reduce the number of compulsory misses.*
3. *Capacity misses: was in cache once, but was replaced.*

4. *Conflict misses are highest for a direct mapped (1-way) cache, and zero for a fully associative cache.*
5. *Most modern system architectures demand hardware cache coherence for both multiprocessor and I/O transactions.*
6. *Tag ram and data ram are typically implemented in physically separate blocks. (e.g. see the 21164 floor plan in the additional material)*

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)

Associativity

1. *Beware terminology: An n-way cache has size/(n*line size) sets, not n.*
2. *Associative caches require extra gated delays due to the multiplexors to select between the different banks. Data cannot be speculatively forwarded to the CPU in parallel with the tag lookup, since the tag comparisons are necessary to select the appropriate block.*
3. *When associative caches are implemented external to the CPU, implementing the multiplexors can be troublesome due to the large number of input pins. Instead, a tristate bus is often used, but this can impair the cache access frequency.*
4. *Some systems do not have wide enough tag fields to enable the entire physical address space to be cached. Many Pentium motherboards would only cache the first 64MB of physical memory; references beyond this forced to miss.*

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

1. *LFSR: Linear Feedback Shift Register (pseudo random number generator)*
2. *Random replacement requires no per line or per set, and performs rather well. NLU is potentially better, and still relatively simple—just a single pointer to the last line used in each set.*

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

Caching Writes

1. *Allocation policy and write through/back are orthogonal parameters, but in general, only Read-only Allocate combined with Write-Through and Read-Write Allocate combined with Write-Back are useful.*
2. *Write back caches reduce the number of stores seen by the next level (the next level may only ever see victim writes). This is essential in multiprocessor systems to reduce traffic on the interprocessor bus, but requires a coherence mechanism.*
3. *Write through caches avoid some of the coherence problems of write back caches, and are generally simpler. Write-through is generally the desired behaviour when communicating with memory mapped peripherals e.g. frame buffers. For this reason, many architectures allow control of cache-ability on a per page basis e.g. the x86 allows pages to be marked as un-cacheable, write-back cacheable and write-through cacheable.*
4. *Write through can actually be a win over write back for certain operations. E.g. Consider scanning over a large array initialising it to zero. The initial line fetch will be a waste of bandwidth, since all the words will be updated. Some architectures provide a write-line invalidate instruction that allocates a line in the cache without the initial fetch; the program guarantees to update all words in the line. Alternatively, some architectures provide store instructions that force write-through behaviour.*

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 and replay

- *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

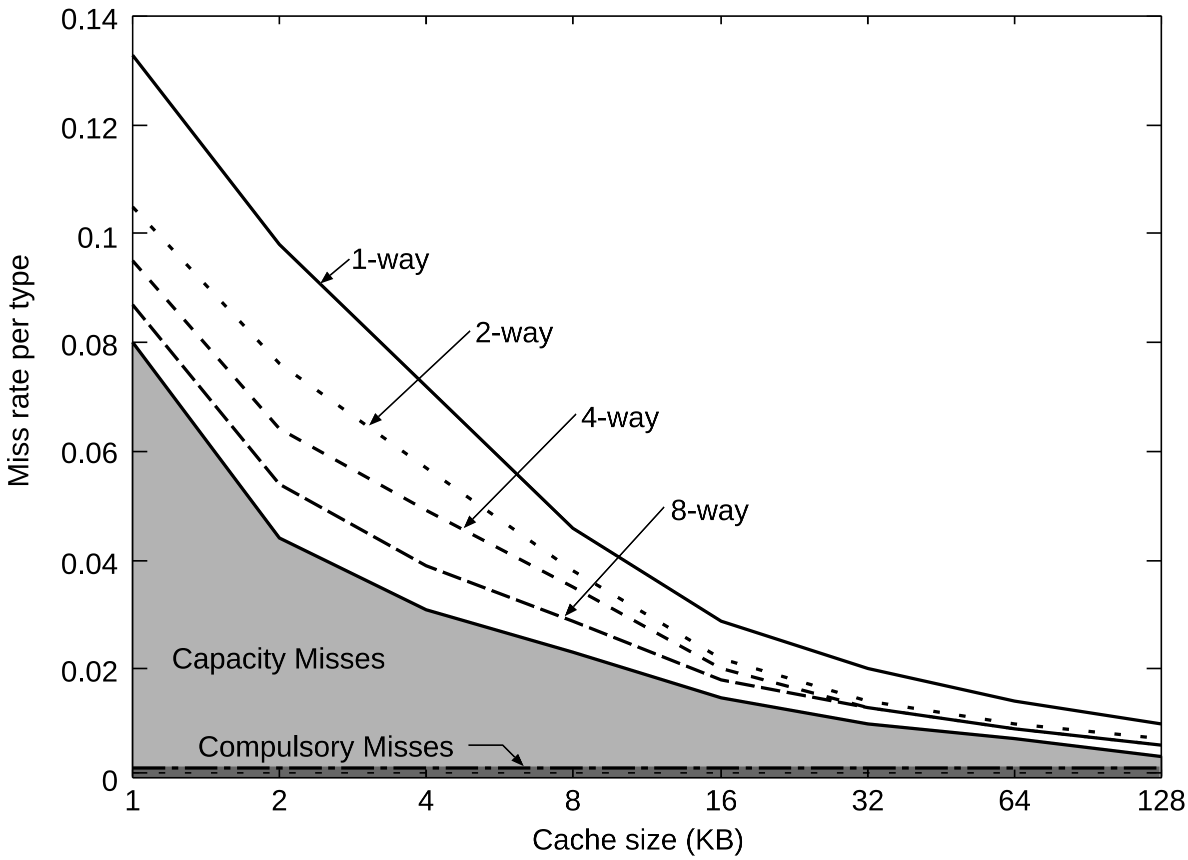
1. *Write buffers accept store data from the CPU core at full-rate, and attempt to hide the lower throughput of the underlying memory system. By allowing loads to pass stores they also result in reduced average load latency.*
2. *On a cache miss, the write buffer must be examined to see whether it contains a write to the requested load address. In theory, a write buffer could service the requested load address directly, but in practise, the write buffer is normally*

flushed to the next level of the memory hierarchy and the load replayed.

3. *If a write buffer becomes full, the next store will cause a stall until a slot becomes available. The 21164 write buffer has slots for six addresses and 32 data bytes per address. Other current CPUs are similar.*

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 $N/2$

1. *Compulsory miss rate is independent of size or associativity.*
2. *The Capacity miss rate was calculated by considering a fully associative cache of size n.*
3. *Little advantage is gained with associativity greater than 4-way for caches >16KB (most current generation CPUs have 16KB or more L1 caches)*

L1 Caches

- L1 I-cache
  - Single-ported
  - Wide access (e.g. block of 4 instrs)
- L1 D-cache
  - Generally 8-64KB 1/2/4-way on-chip
    - Exception: HP PA-RISC
  - 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)

L1 Caches

1. All current high-performance processors have separate L1 I&D caches—It's much better to have separate caches rather a single double size cache with an extra read port.
2. x86 I-cache snoops D-cache (as does the rest of the pipeline)
3. Dynamic execution can hide D-cache latency better, enabling larger more complex caches with longer latency but lower miss rate.
4. A few years ago L1 cache size may have been constrained by available die area, but now performance is generally the limiting factor.
5. The exception is the PA-RISC machines, which in the past have had large off-chip L1 caches. HP's assertion has been that their execution engines have been able to hide the load-use latency, and that their machines perform better on 'commercial applications' that have worse locality of reference than SPEC95.
6. Cache complexity (and thus miss rate) is often traded for hit access time. Direct mapped caches are often used to enable the tag check to be done in parallel with speculatively feeding the data to the CPU (hence reducing load-use penalties). Modern dynamic-issue CPUs tend to be less sensitive to such stalls, thus the complexity of L1 caches is increasing. (see the point below also—adding associativity can help to keep the cache virtually indexed without special OS support)
7. Address aliasing in a virtually indexed cache occurs when two VAs which map to the same physical address map to a different set in the cache. The OS can prevent this by ensuring that all VAs referring to the same physical address have sufficient low-order address bits in common. Due to the nature of paging, all VAs referring to the same physical

location must have at least log2(page size) bits in common. Aliasing is impossible provided the cache index requires fewer bits than this. Adding more associativity to a cache reduces the number of index address bits. E.g. x86's have a 4KB page size. The Pentium Pro has an 8KB 2-way L1 D-cache, the Pentium II a 16KB 4-way.

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 restart early
  - 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 :1

1. *High memory system bandwidth is typically very important for FP programs. In contrast, integer programs often benefit more from reduced latency. (Sequential vector (array) operations vs. pointer chasing (e.g. scanning linked lists))*
2. *Many systems use 'linear wrap', where the word requested (signalled by the low-order address bits) is returned first, and then cache fill proceeds sequentially, wrapping at the end of the block back to the beginning. Rather than linear wrap, Intel use a bizarre ordering that is supposed to optimize the likelihood that the subsequent word requested by the CPU is the second word in the fill order.*
3. *Most memory systems now allow transactions to be serviced out-of-order. The 21264 is particularly aggressive, allowing up to 16 transactions.*

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)

Enhancing Performance :2

1. *Victim caches are employed on some systems with large off-chip direct-mapped L1 caches. Adding a normal L2 cache would not help much (the L1 is already big), but the extra associativity of the victim cache can help. (E.g. The HP PA-7200)*
2. *The contents of victim buffers can be written out when the memory system is otherwise idle. If the victim buffers are already full when a dirty line is evicted, the victim must be written before the fill can occur. Victim buffers don't help if large numbers of victims are being generated, e.g. by a program performing sequential writes to a very large array (memset).*
3. *As with write buffers, missed loads must check the contents of the victim buffer before proceeding.*
4. *Victim buffers are particularly useful if the latency of the Ln+1 memory system is high. (Alternatively, some systems use an address routing trick to ensure that both the victim and the line that is to be fetched must come from the same DRAM page, reducing the total transaction time.)*
5. *Another way of enhancing performance is to use a L2 (or L3) cache to reduce average miss time...*

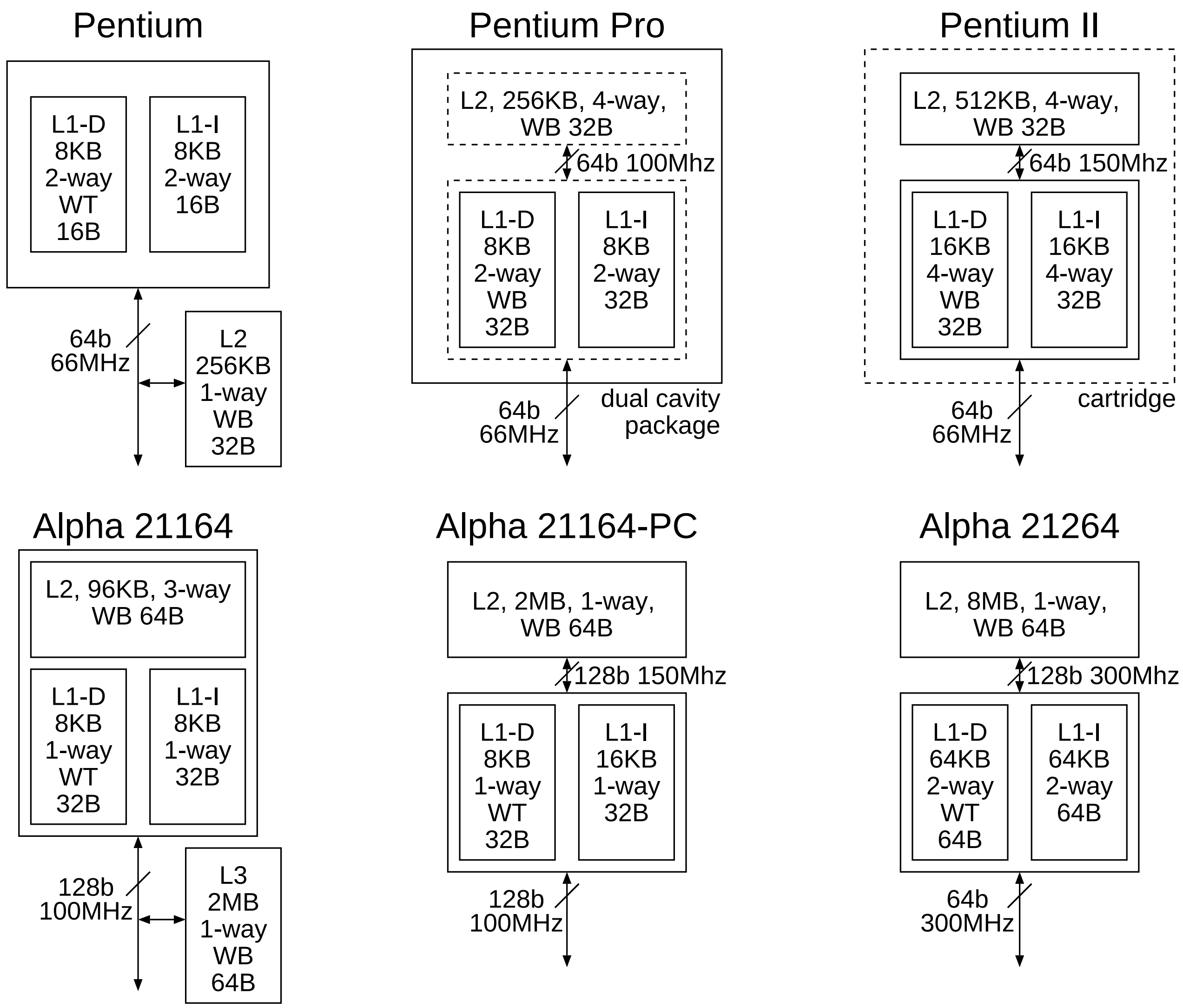
L2 caches

- L2 caches help hide poor DRAM latency
- 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

L2 caches

1. *Large write-back L2 caches are essential for good SPEC95 results*
2. *Unless an L3 cache can be made substantially (e.g. >16x) bigger than the L2 it is unlikely to be of much benefit (it may even increase the main memory access time).*
3. *The backside bus is typically not a bus, but a point-to-point link between CPU pins and cache RAM pins. The point-to-point links typically use high speed low voltage signalling and care is taken to control skew and propagation delays.*
4. *The dual cavity packaging approach used on the Pentium Pro is expensive to manufacture, hence the Pentium II.*
5. *Intel introduced cartridge to avoid the burden of cache layout falling to motherboard designers (and to maximise their own profits...)*
6. *Older caches used 'wave pipelining' to maximise throughput. This is effectively using propagation delay to provide asynchronous pipelining. Now, modern Synchronous SRAMs (SSRAMs) have inputs and outputs synchronous to a clock input, and are pipelined internally.*

Hierarchy Examples



1. *Conventional system buses are 64-128 bits wide and run at 66-100MHz. Bus signals have multiple electrical loads. New CPUs (e.g. the 21264 and AMD K7) will have point-to-point system 'buses' clocked at >150MHz. 'Switch chips' will be used to provide the connections to the I/O and memory*

sub-systems. (This is done for the same reasons as having a dedicated backside bus for the external cache).

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

Performance Examples

1. *This table refers only to data caches. All the data was collected using a test program written by the author. Care was taken to perform 'reasonable optimization' of the loops for each of the different systems. Despite this, the numbers reported are in some cases rather lower than might be expected from reading the system documentation. Achieving the documented figures often requires 'tricks' that are not generally applicable, hence these figures represent the sort of memory system performance that real programs might hope to achieve.*
2. *The load bandwidth test simply reads words sequentially from an array of size n (where n is the size of the cache to be tested). When testing the L2 bandwidth, the L1 cache does not interfere since almost all loads will miss due to capacity misses; any useful data the cache holds will be pushed out before it is needed on the next iteration. (likewise for testing L3 and main memory)*
3. *Store bandwidth is measured by writing sequentially to an array of size n.*
4. *The latency tester creates a linked list of cache line sized elements, of total size n. It then accurately measures the time to chases down the list multiple times, and computes the load-to-use latency.*
5. *Many Pentium systems have write back L2 caches; I'm not sure why the BIOS hadn't enabled write-back on this system.*
6. *Note how the 21164's L1 cache is write through, and thus how the L1 store bandwidth is the same as the L2. (Likewise for both caches on the Pentium system).*
7. *The Pentium Pro and PII have longer latency L1 D-caches (enabled by the dynamic execution engine. Despite the two cycle latency, both are fully pipelined)*

8. *The Pentium II's off-chip L2 cache has longer latency than the Pentium Pro's L2 in the 2nd die. However, The PII's cache is made from off-the-shelf SSRAM parts, and is thus cheaper. (The use of SSRAM enables the throughput to be the same despite the latency being greater).*
9. *Larger caches sometimes benefit from larger line sizes (compare 21164 L1 and L2, or Pentium L1 and L2).*

Main Memory

- Increasing sequential bandwidth
  - Wide memory bus
  - Interleave memory chips

  ⇒ 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

Main Memory

1. *Wide memory systems (e.g. 256 bits on Alpha EB164) typically require a large number of memory modules (e.g. SIMMs/DIMMs) to be installed. This results in a large, expensive memory configuration which is often hard to upgrade due to a limited number of motherboard SIMM slots.*
2. *Wide memory systems are expensive. The higher clock rate of SDRAM has helped to reduce the need for wide memory systems in many applications. Pentium II and Pentium Pro systems make do with 64 bit memory systems. Alpha SDRAM systems are 128-256 bits wide.*
3. *Intel are committed to using RAMBUS on future processors. This will result in main memory connecting directly to the processor rather than system support chips. Each CPU will probably support 4-16 separate RAMBUS buses (known as 'strings')operating concurrently.*
4. *With a simple memory system, DRAM latency often results in the system bus laying idle much of the time, waiting for the row access to complete. Increasing the cache line size increases the percentage of cycles where useful work is done.*
5. *With independent memory banks, each cache line is serviced from a single memory bank (no word-level interleaving). The memory system typically accepts multiple transactions from the CPU, and services them out-of-order depending on the availability of each bank. Each bank is kept 'open' after an access, enabling subsequent accesses to the same DRAM page to continue with low latency. This type of scheme is becoming increasingly popular due to the fact that individual SDRAM/RAMBUS parts are now able to deliver relatively high sequential bandwidth—it's just the latency that's the real problem. Having multiple banks helps to reduce the average latency. Individual SDRAM and RAMBUS parts now*

contain multiple banks internally as well; the memory controller must keep track of which rows are currently latched and thus available for rapid access.

6. *The STREAM benchmark is often used to measure memory system bandwidth. lmbench is often used to measure latency.*

Programming for caches

- Design data structures such that working set fits in cache
- Organise data for spatial locality
- Fuse together loops that access the same data
- Prefer sequential accesses to non-unit strides
  - innermost loop should access array sequentially if possible
- If row and column access to 2D arrays is necessary, use *blocking*
  - divide problem into sub-matrices that fit cache
- Avoid access patterns that are likely to cause conflict misses (aliasing)
- Large strides can thrash the TLB
- Touch cache lines ahead of use (prefetch)
  - Load instructions that suppress exceptions make boundary conditions easier

Optimizing for caches can yield substantial results

Programming for caches

1. *Caches can lead to large performance variances on different machines—possibly even on the same machine due to different physical page placement for different runs.*
2. *Organise data for spatial locality, e.g. collect elements of different arrays that are accessed together into a single* `struct` *array. This will also reduce the possibility of aliasing.*
3. *Fuse loops that access the same data in the same pattern into a single loop.*
4. *Large strides can cause cause thrashing of the TLB*
5. *Large strides that are powers of 2 are more likely to cause aliasing.*
6. *Two good articles on programming for caches are included in the additional material. (one for the 21164, one for x86). H&P is also very good.*