Lecture 2: EITF20 Computer Architecture

ISA - Instruction Set Architecture

Anders Ardö

EIT – Electrical and Information Technology, Lund University

October 30, 2013

Outline

1. Reiteration
2. Instruction set principles
3. Compiler role
4. Example - MIPS

Previous lecture

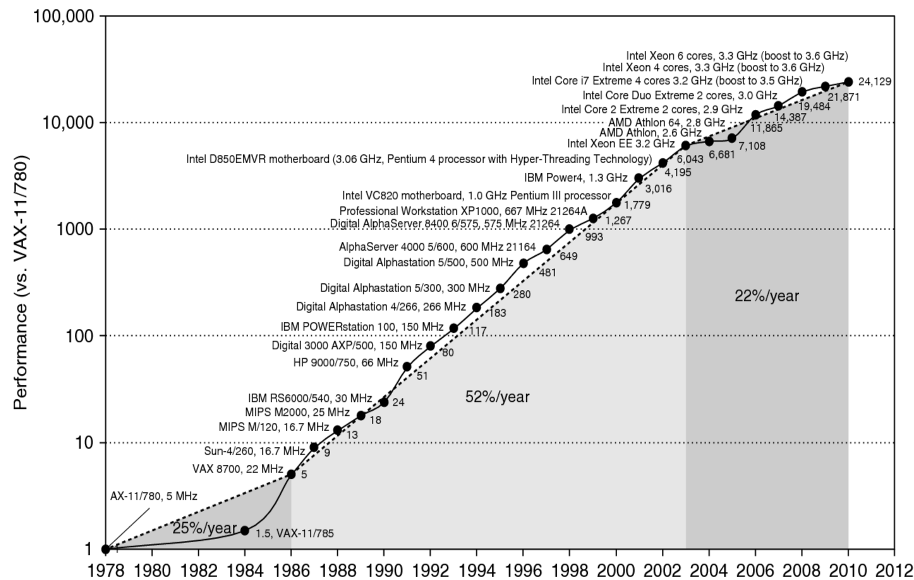
- Computer Architecture
- Performance
- Quantitative principles

What computer architecture?

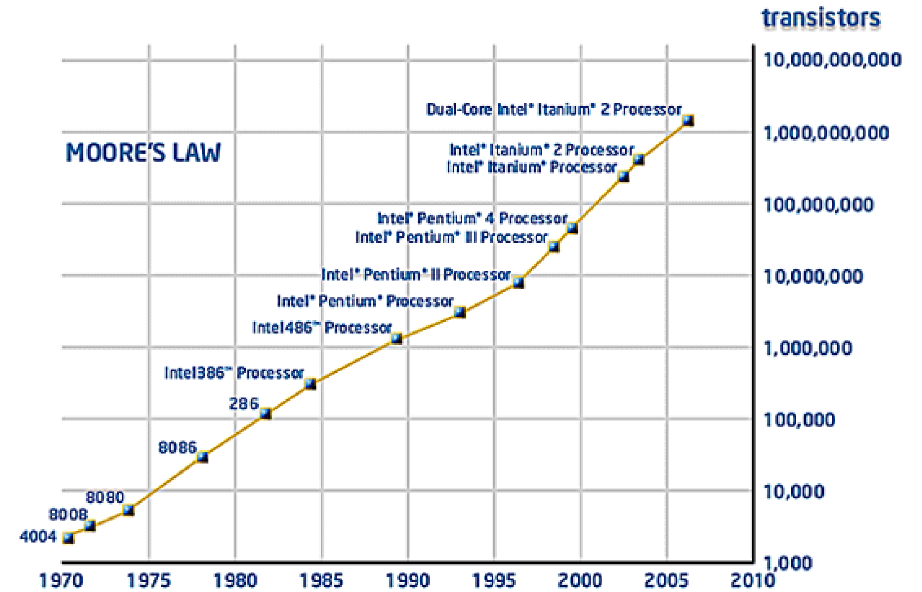
Designing
- ISA
- Organization
- Hardware

to meet
- functional requirements (applications, standards, ...)
- price
- power
- performance
- availability
- dependability

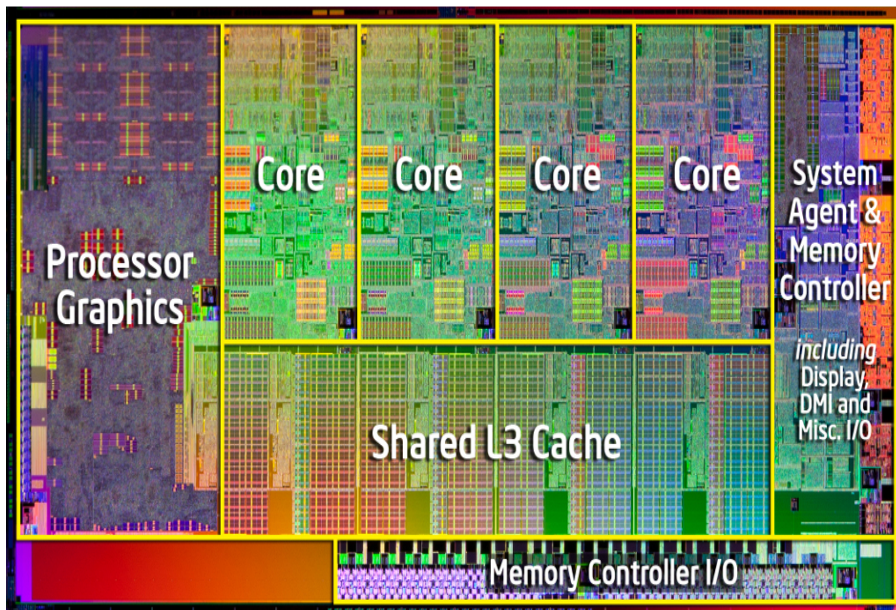
Performance of processors



Transistors in a CPU



Transistors in a CPU



Metrics of performance

- Time to complete a task (T_{exe})
  - Execution time, response time, latency
- Task per day, hour, second, ... (Performance)
  - Throughput, bandwidth

Application	$\Longleftarrow$	Answers/month
Programming	$\Longleftarrow$	Response time (seconds)
language	$\Longleftarrow$	Operations/second
Compiler		
Instruction set	$\Longleftarrow$	MIPS/MFLOPS
Data-path control	$\Longleftarrow$	Megabytes/second
Functional units		
Transistors, wires, pins	$\Longleftarrow$	Cycles per second (clock rate)

MIPS = millions of instructions per second
MFLOPS = millions of FP operations per seconds

Quantitative principles

- Take advantage of parallelism
- Principle of locality
- Focus on the common case
- **Amdahl's law**
  Enhancement E accelerates a fraction F of a program by a factor S
  $T_{exe}(with\ E) = T_{exe}(without\ E) * [(1 - F) + F/S]$
  $Speedup(E) = \frac{T_{exe}(without\ E)}{T_{exe}(with\ E)} = \frac{1}{(1-F)+F/S}$
- Processor performance equation

$$\text{Execution time} =$$
$$seconds/program =$$
$$\underbrace{instr./program}_{IC} * \underbrace{cycles/instr.}_{\overline{CPI}} * \underbrace{seconds/cycle}_{T_c}$$

Questions!

QUESTIONS?

COMMENTS?

Lecture 2 agenda

Chapter 1.3 and Appendix B in "Computer Architecture"
Peter Varhol, "GPU vs. CPU Computing"

1. Reiteration
2. Instruction set principles
3. Compiler role
4. Example - MIPS

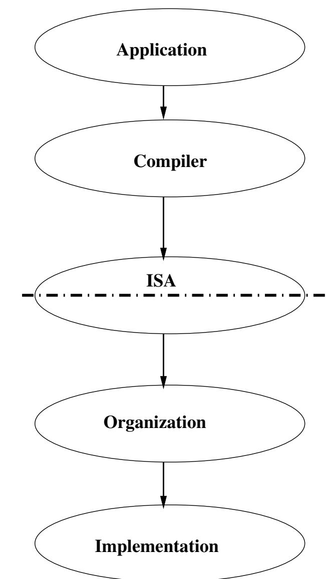
Outline

1. Reiteration
2. Instruction set principles
3. Compiler role
4. Example - MIPS

Instruction set principles

- Classification of instruction sets
- What's needed in an instruction set?
  - Addressing
  - Operands
  - Operations
  - Control Flow
- Encoding
- The impact of the compiler
- The MIPS instruction set architecture

Instruction Set Architecture - ISA



Instruction Set Architecture (ISA)

ISA (Instruction Set Architecture) is the
interface between software and hardware

A good interface:

- Lasts through many implementations (portability, compatibility)
- Can be used in many different ways (generality)
- Provides sufficient functionality to higher levels
- Permits an efficient implementation at lower levels

RISC - CISC

$$CPUtime = T_c * \overline{CPI} * IC$$

RISC *CISC*

RISC (Reduced Instruction Set Computing)

- simple instructions
- MIPS, ARM, ...
- easier to design, build
- less power
- larger code size
- easier for compiler

CISC (Complex Instruction Set Computing)

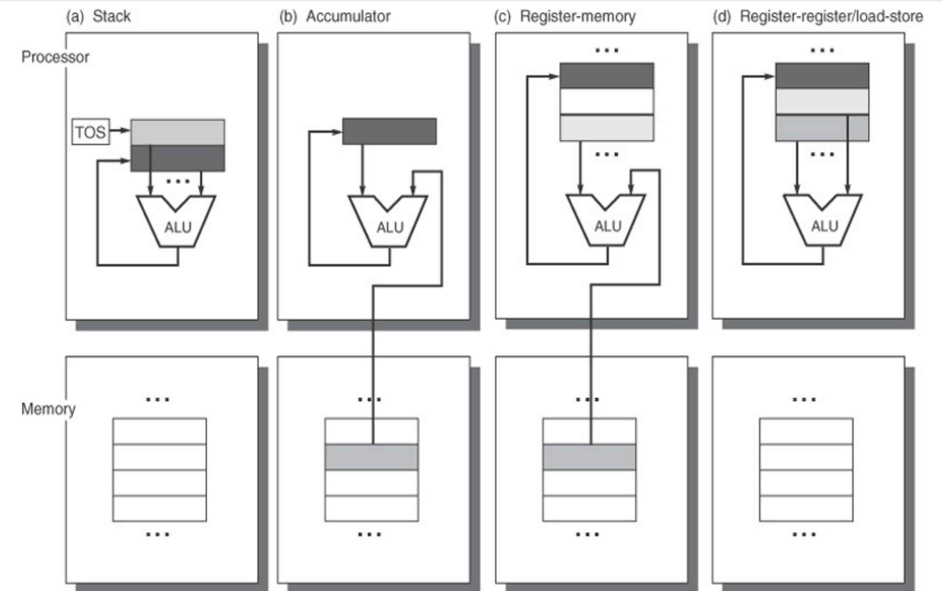
- complex instructions
- VAX, Intel 80x86 (now RISC-like internally), ...

Instruction Set Architecture

An ISA can be classified according to:

- Register model
- The number of operands for instructions
- Addressing modes
- The operations provided in the instruction set
- Type and size of operands
- Control flow instructions
- Encoding

Instruction set architecture classes



The register model

How are operands accessed by the CPU?

Temporary storage	Examples	Explicit operands	Result destination	Accessing operands
stack	B5500	0	stack	Push, Pop
Accumulator	PDP-8	1	Accumulator	Load/Store
Register (GPR)	IBM 360 VAX MIPS 80x86	2 or 3	Register or Memory	Load/Store or Reg/Mem

Only GPR (General Purpose Register) architectures are used today.

The register model

Example: C = A + B

stack	accumulator	register memory	register register
PUSH A	LOAD A	LOAD R1,A	LOAD R1,A
PUSH B	ADD B	ADD R3,R1,B	LOAD R2,B
ADD	STORE C	STORE R3,C	ADD R3,R1,R2
POP C			STORE R3,C

Only GPR (General Purpose Register) architectures are used today.

Interpretation of memory addresses

A 32 bit integer variable (0x01234567) stored at address 0x100

- **Big Endian**:
  The *most* significant byte is stored at address: 0x100

address		0x100	0x101	0x102	0x103	
value	...	01	23	45	67	...

- **Little Endian**:
  The *least* significant byte is stored at address: 0x100

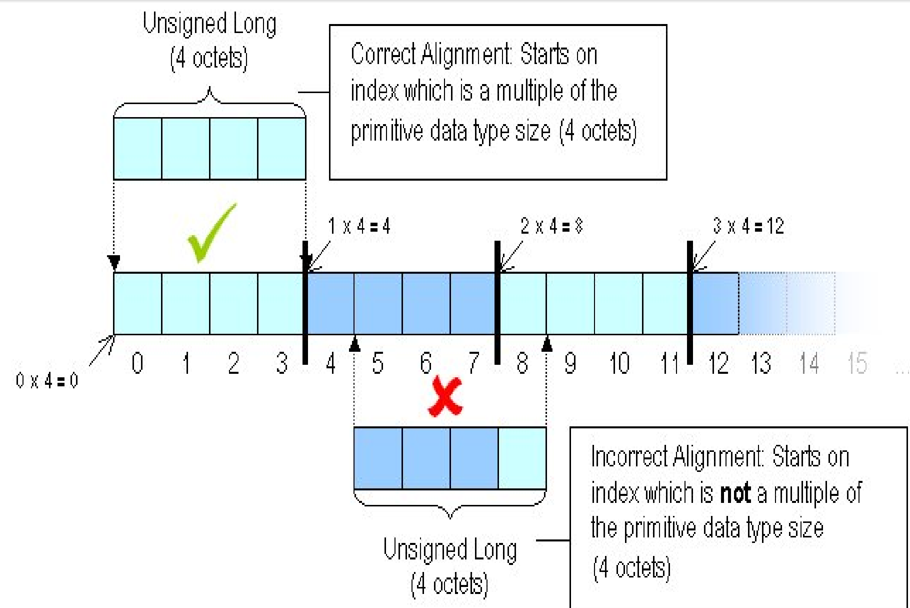
address		0x100	0x101	0x102	0x103	
value	...	67	45	23	01	...

Important for exchange of data, (and strings)

Interpretation of memory addresses

- An architecture may require that data is *aligned*:
  - byte always aligned
  - half word (16 bits) aligned at byte offsets 0,2,4,6,...
  - word (32 bits) aligned at byte offsets 0,4,8,12,...
  - double word (64 bits) aligned at byte offsets 0,8,16,24,...

Memory alignment



Memory alignment

	Value of 3 low-order bits of byte address							
Width of object	0	1	2	3	4	5	6	7
1 byte (byte)	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned	Aligned
2 bytes (half word)	Aligned		Aligned		Aligned		Aligned	
2 bytes (half word)		Misaligned		Misaligned		Misaligned		Misaligned
4 bytes (word)	Aligned				Aligned			
4 bytes (word)		Misaligned				Misaligned		
4 bytes (word)			Misaligned				Misaligned	
4 bytes (word)				Misaligned				Misaligned
8 bytes (double word)	Aligned							
8 bytes (double word)		Misaligned						
8 bytes (double word)			Misaligned					
8 bytes (double word)				Misaligned				
8 bytes (double word)					Misaligned			
8 bytes (double word)						Misaligned		
8 bytes (double word)							Misaligned	
8 bytes (double word)								Misaligned

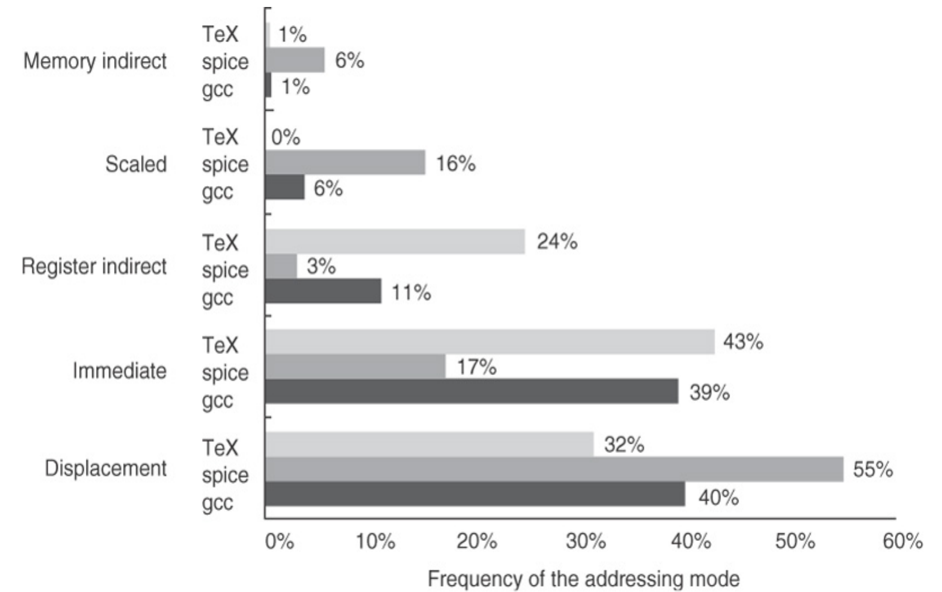
Figure B.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide as the object. The figure shows the memory organized as 8 bytes wide. The byte offsets that label the columns specify the low-order 3 bits of the address.

Addressing modes

Are all these addressing modes needed?

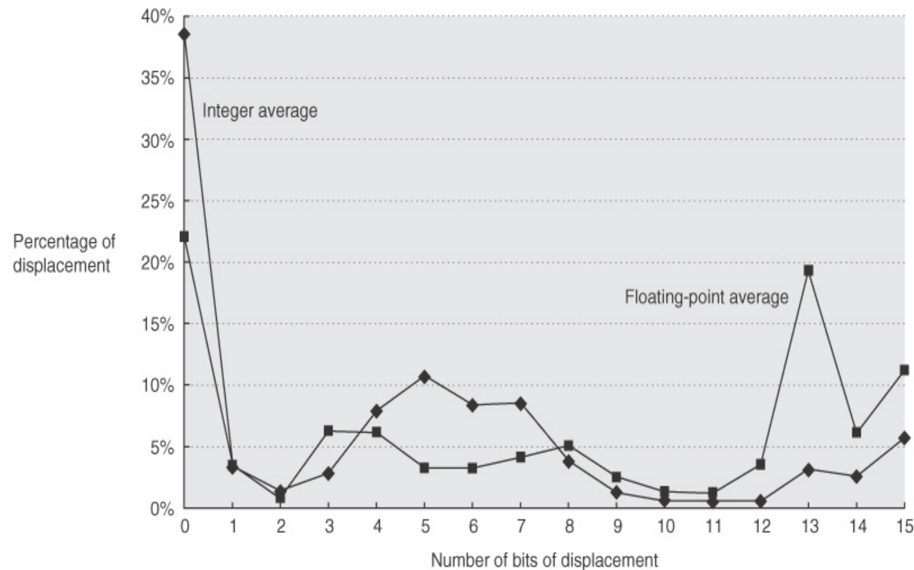
- Register
- Immediate
- Displacement
- Register indirect
- Indexed
- Direct or absolute
- Memory indirect
- Auto-increment
- Auto-decrement
- Scaled

Actual use of addressing modes



Size of displacement

- How many bits are needed for address displacements?



What does it mean?

00100000010010000110100100100001

	Byte 0	Byte 1	Byte 2	Byte 3
binary	00100000	01001000	01101001	00100001
hex	20	48	69	21
ASCII	SP	H	i	!
	' Hi!'			
INT	541616417			
FP	$1.6975443 * 10^{-19}$			

What does it mean?

00100000010010000110100100100001

	Byte 0	Byte 1	Byte 2	Byte 3	
binary	00100000	01001000	01101001	00100001	BNEZ, AND
hex	20	48	69	21	JR
ASCII	SP	H	i	!	
	' Hi!'				
INT	541616417				ADD
FP	$1.6975443 * 10^{-19}$				ADD.S

What does it mean?

00100000010010000110100100100001

001000	00010	01000	0110100100100001
instruction	source	target	immediate
ADDI	2	8	26913

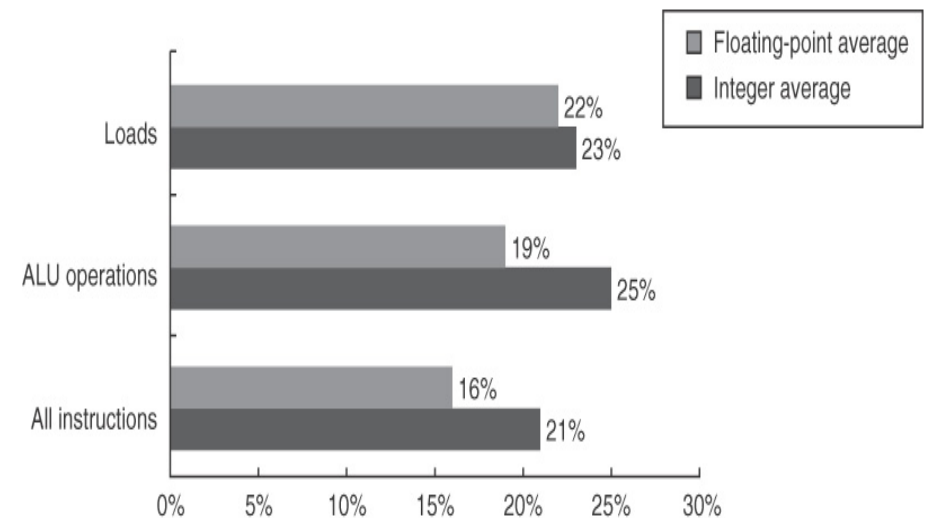
ADDI R8,R2,26913

Type and size of operands

- integer
- floating point (single precision)
- character
- packed decimal
- ... etc ...

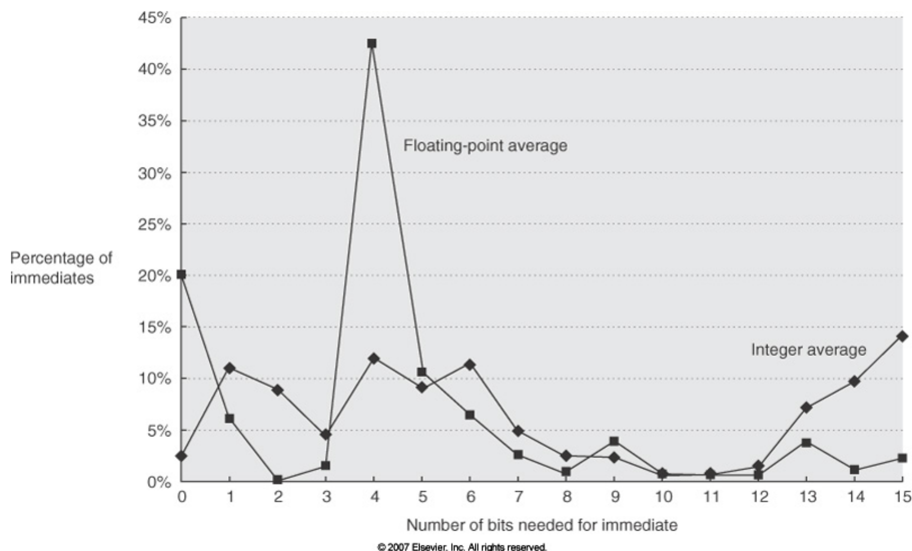
Immediate operands

- How important are immediates and how large?



Immediate operands

- How important are immediates and how large?

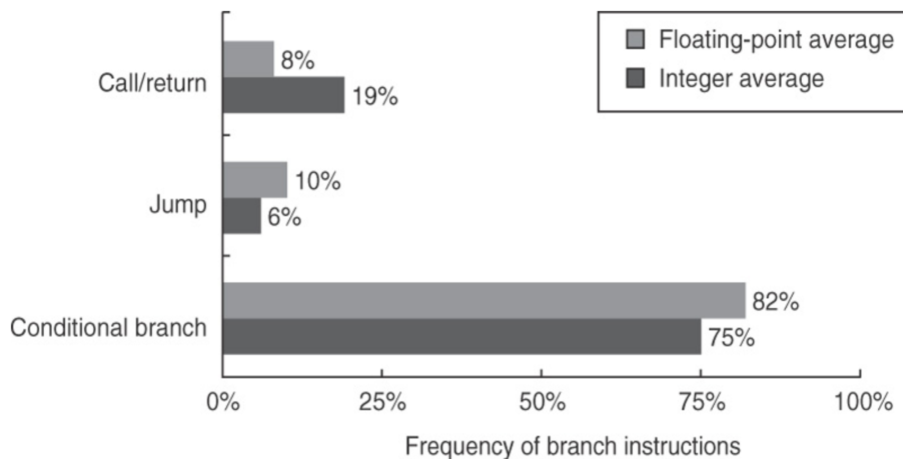


Types of operations

Operator type	Examples
Arithmetic Logical	addition, subtraction, multiply, divide and, or, xor
Data transfer	load, store, move
Control flow	branch, jump, procedure call, return, traps
System	OS call, virtual memory instructions
Floating point	add, subtract, multiply, divide, compare
Decimal	add, multiply, divide,
String	move, compare, search
Graphics	Pixel and vertex operations; compression/decompression

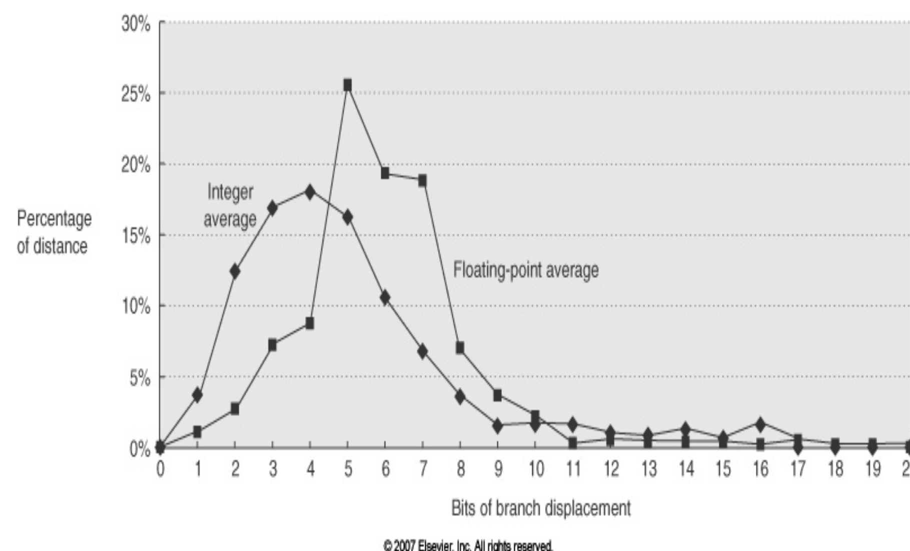
Types of control instructions

- Conditional branches
- Unconditional branches (jumps)
- Procedure call/returns



Branch displacements

- How large need the branch displacement be?



Instruction formats

- A variable instruction format yields compact code but the instruction decoding is more complex and thus slower
  Examples: VAX, Intel 80x86

Operation # operands	Address specifier 1	Address field 1	...	Address specifier x	Address field x

- A fixed instruction format is easy and fast to decode but gives large code size
  Examples: Alpha, ARM, MIPS, PowerPC, SPARC

Operation	Address field 1	Address field 2	Address field 3

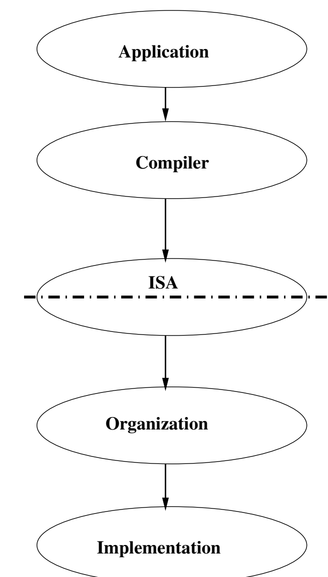
Outline

1. Reiteration
2. Instruction set principles
3. Compiler role
4. Example - MIPS

Compilers and instruction set

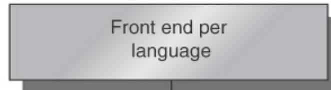
- Instruction set architecture is a compiler target
- by far most instructions executed are generated by a compiler (exception certain special purpose processors)
- interaction compiler - ISA critical for overall performance

Instruction Set Architecture - ISA



The structure of a compiler

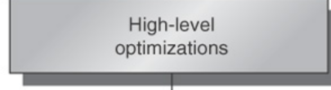
Dependencies
Language dependent; machine independent



Function
Transform language to common intermediate form

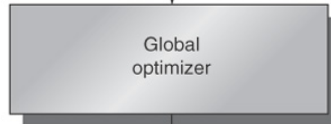
Intermediate representation

Somewhat language dependent; largely machine independent



For example, loop transformations and procedure inlining (also called procedure integration)

Small language dependencies; machine dependencies slight (e.g., register counts/types)



Including global and local optimizations + register allocation

Highly machine dependent; language independent

Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

© 2007 Elsevier, Inc. All rights reserved.

GCC Optimization Options

-O -O0 -O1 -O2 -O3 -Os

-falign-functions -falign-jumps -falign-labels -falign-loops -fassociative-math -fauto-inc-dec -fbranch-probabilities

-fbranch-target-load-optimize -fbranch-target-load-optimize2 -fbtr-bb-exclusive -fcaller-saves -fcheck-data-deps -fcprop-registers

-fcrossjumping -fcse-follow-jumps -fcse-skip-blocks -fcx-limited-range -fdata-sections -fdce -fdce -fdelayed-branch

-fdelete-null-pointer-checks -fdse -fdse -fearly-inlining -fexpensive-optimizations -ffast-math -ffinite-math-only -ffloat-store

-fforward-propagate -ffunction-sections -fgcse -fgcse-after-reload -fgcse-las -fgcse-lm -fgcse-sm -fif-conversion -fif-conversion2

-finline-functions -finline-functions-called-once -finline-limit -finline-small-functions -fipa-cp -fipa-matrix-reorg -fipa-pta

-fipa-pure-const -fipa-reference -fipa-struct-reorg -fipa-type-escape -fivopts -fkeep-inline-functions -fkeep-static-consts

-fmerge-all-constants -fmerge-constants -fmodulo-sched -fmodulo-sched-allow-regmoves -fmove-loop-invariants -fmudflap

-fmudflapir -fmudflapth -fno-branch-count-reg -fno-default-inline -fno-defer-pop -fno-function-cse -fno-guess-branch-probability

-fno-inline -fno-math-errno -fno-peephole -fno-peephole2 -fno-sched-interblock -fno-sched-spec -fno-signed-zeros

-fno-toplevel-reorder -fno-trapping-math -fno-zero-initialized-in-bss -fomit-frame-pointer -foptimize-register-move

-foptimize-sibling-calls -fpeel-loops -fpredictive-commoning -fprefetch-loop-arrays -fprofile-generate -fprofile-use -fprofile-values

-freciprocal-math -fregmove -frename-registers -freorder-blocks -freorder-blocks-and-partition -freorder-functions

-frerun-cse-after-loop -freschedule-modulo-scheduled-loops -frounding-math -frtl-abstract-sequences -fsched2-use-superblocks

-fsched2-use-traces -fsched-spec-load -fsched-spec-load-dangerous -fsched-stalled-insns-dep -fsched-stalled-insns

-fschedule-insns -fschedule-insns2 -fsection-anchors -fsee -fsignaling-nans -fsingle-precision-constant -fsplit-ivs-in-unroller

-ftree-copy-prop -ftree-copyrename -ftree-dce -ftree-dominator-opts -ftree-dse -ftree-fre -ftree-loop-im -ftree-loop-ivcanon

-ftree-loop-linear -ftree-loop-optimize -ftree-parallelize-loops -ftree-pre -ftree-reassoc -ftree-salias -ftree-sink -ftree-sra

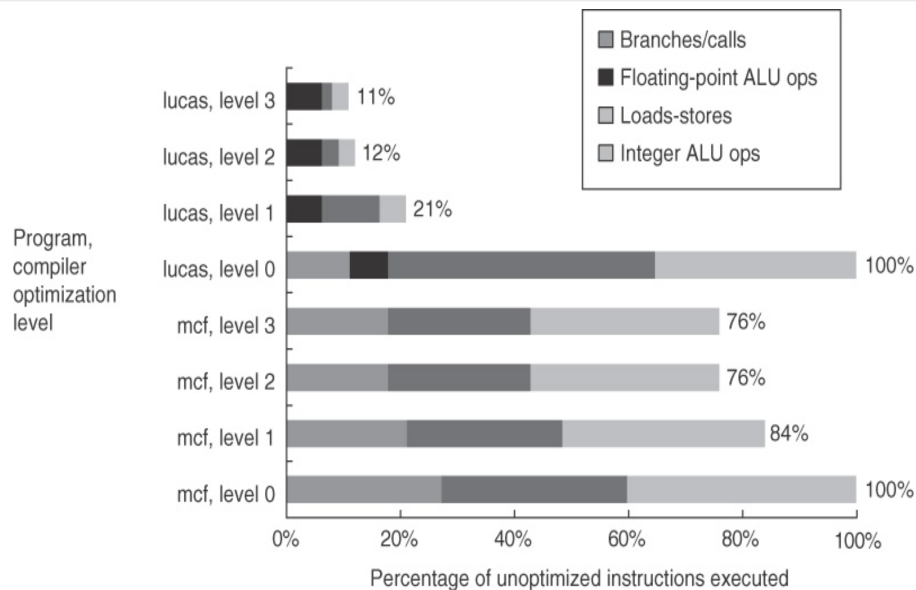
Examples of compiler optimizations

- Code improvements made by the compiler are called *optimizations* and can be classified:
  - *High-order transformations*: procedure inlining
  - *Optimizations*: dead code elimination
    constant propagation
    common sub-expression elimination
    loop-unrolling
  - *Register allocation*
  - *Machine-dependent optimizations*: takes advantage of specific architectural features
- **Almost all of these optimizations are easier to do if there are many general registers available!**

How can you aid the compiler?

- Rules of thumb when designing an instruction set:
  - Regularity (operations, data types and addressing modes should be orthogonal)
  - Provide primitives, not high-level constructs or solutions. Complex instructions are often too specialized.
  - Simplify trade-offs among alternatives
  - Provide instructions that bind quantities known at compile time as constants

The impact of compiler optimizations



Outline

1. Reiteration
2. Instruction set principles
3. Compiler role
4. Example - MIPS

The MIPS64 architecture

- An architecture representative of modern instruction set architectures:
  - 64 bit load/store GPR architecture
  - 32 general integer registers (R0 = 0) and 32 floating point registers
  - Supported data types: bytes, half word (16 bits), word (32 bits), double word (64 bits), single and double precision IEEE floating points
  - Memory byte addressable with 64 bit addresses
  - Addressing modes: immediate and displacement

Example MIPS instructions

LW	R1,60(R7)	Load word
SB	R2,41(R5)	Store byte
MUL	R2,R1,R3	Integer multiply
AND	R3,R2,R1	Logical AND
DADDI	R5,R6,#17	Add immediate
J	lable	Jump
BEQZ	R4,lable	Branch if R4 zero
JALR	R7	Procedure call

MIPS instruction format

I-type instruction

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt ← rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
(rd = 0, rs = destination, immediate = 0)

R-type instruction

Register-register ALU operations: rd ← rs funct rt
Function encodes the data path operation: Add, Sub, . . .
Read/write special registers and moves

J-type instruction

Jump and jump and link
Trap and return from exception

© 2007 Elsevier, Inc. All rights reserved.

Summary

- The instruction set architecture have importance for the performance
- The important aspects of an ISA are:
  - register model
  - addressing modes
  - types of operations
  - data types
  - encoding
- Benchmark measurements can reveal the most common case
- Interaction compiler - ISA important