Computer System Architecture

Instruction Set Principles and Examples

Chalermek Intanagonwiwat

Slides courtesy of Graham Kirby, Mike Schulte, and Peiyi Tang

Hot Topics in Computer Architecture

- 1950s and 1960s:
  - Computer Arithmetic
- 1970 and 1980s:
  - Instruction Set Design
  - ISA Appropriate for Compilers
- 1990s:
  - Design of CPU
  - Design of memory system
  - Instruction Set Extensions

Hot Topics in Computer Architecture (cont.)

- 2000s:
  - Computer Arithmetic
  - Design of I/O system
  - Parallelism

Instruction Set Architecture (ISA)

- "Instruction set architecture is the structure of a computer that a machine language programmer must understand to write a correct (timing independent) program for that machine."
  - Source: IBM in 1964 when introducing the IBM 360 architecture, which eliminated 7 different IBM instruction sets.

ISA (cont.)

- The instruction set architecture is also the machine description that a hardware designer must understand to design a correct implementation of the computer.

ISA (cont.)

- The instruction set architecture serves as the critical interface (abstraction) between software and hardware.
- It provides the mechanism by which the software tells the hardware what should be done.

ISA (cont.)

ISA includes

-- Organization of Programmable Storage

-- Data Types & Data Structures: Encodings & Representations

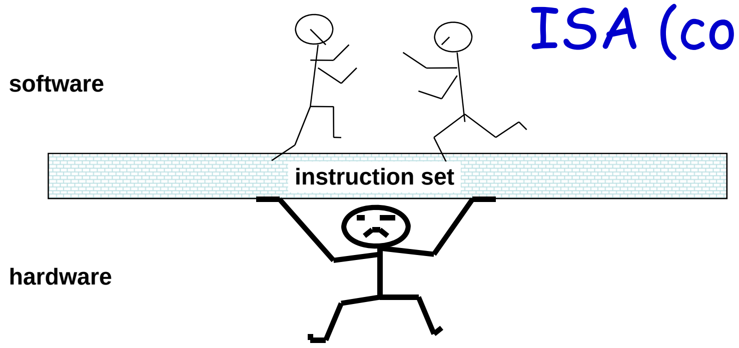
-- Instruction Formats

-- Instruction (or Operation Code) Set

-- Modes of Addressing and Accessing Data Items and Instructions

-- Exceptional Conditions

ISA (cont.)



- Properties of a good abstraction
  - Lasts through many generations (portability)
  - Used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels

ISA Metrics

- Orthogonality
  - No special registers, few special cases, all operand modes available with any data type or instruction type
- Completeness
  - Support for a wide range of operations and target applications

ISA Metrics (cont.)

- Regularity
  - No overloading for the meanings of instruction fields
- Streamlined
  - Resource needs easily determined
- Ease of compilation (or assembly language programming)
- Ease of implementation

Instruction Set Design Issues

- Instruction set design issues include:
  - Where are operands stored?
    - registers, memory, stack, accumulator
  - How many explicit operands are there?
    - 0, 1, 2, or 3
  - How is the operand location specified?
    - register, immediate, indirect, . . .

Instruction Set Design Issues (cont.)

- What type & size of operands are supported?
  - byte, int, float, double, string, vector. . .
- What operations are supported?
  - add, sub, mul, move, compare . . .

Evolution of Instruction Sets

Single Accumulator (EDSAC 1950)

Accumulator + Index Registers (Manchester Mark I, IBM 700 series 1953)

Separation of Programming Model from Implementation

High-level Language Based (B5000 1963)

Concept of a Family (IBM 360 1964)

General Purpose Register Machines

Complex Instruction Sets (Vax, Intel 8086 1977-80)

Load/Store Architecture (CDC 6600, Cray 1 1963-76)

RISC (Mips,Sparc,88000,IBM RS6000, . . .1987+)

Classifying ISAs

Accumulator (before 1960):

1 address	add A	acc ← acc + mem[A]

Stack (1960s to 1970s):

0 address	add	tos ← tos + next

Memory-Memory (1970s to 1980s):

2 address	add A, B	mem[A] ← mem[A] + mem[B]
3 address	add A, B, C	mem[A] ← mem[B] + mem[C]

Classifying ISAs (cont.)

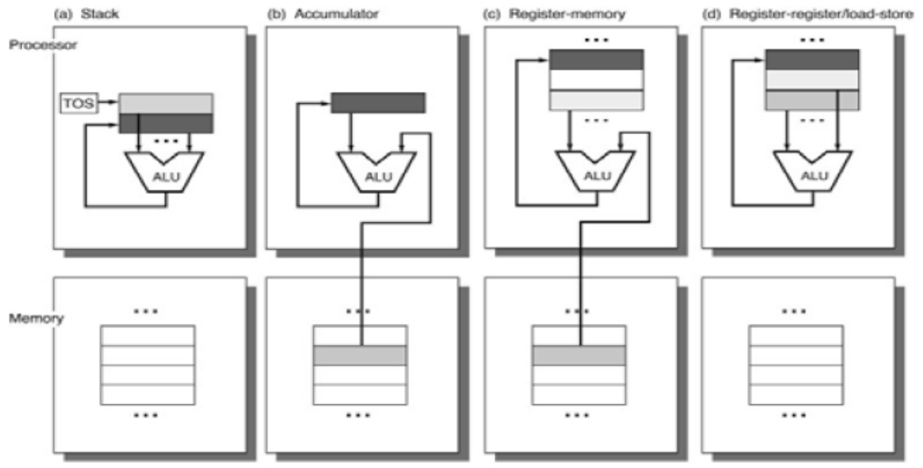
Register-Memory (1970s to present):

2 address	add R1, A	R1 ← R1 + mem[A]
	load R1, A	R1 ← mem[A]

Register-Register (Load/Store) (1960s to present):

3 address	add R1, R2, R3	R1 ← R2 + R3
	load R1, R2	R1 ← mem[R2]
	store R1, R2	mem[R1] ← R2

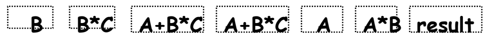
Classifying ISAs (cont.)



Accumulator Architectures

- Instruction set: add A, sub A, mult A, div A, load A, store A
- Example: A*B - (A+C*B)

```
load B
mul C
add A
store D
load A
mul B
sub D
```



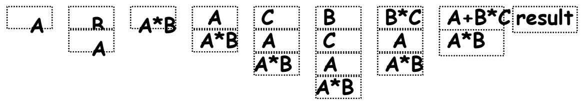
Accumulators: Pros and Cons

- Pros
  - Very low hardware requirements
  - Easy to design and understand
- Cons
  - Accumulator becomes the bottleneck
  - Little ability for parallelism or pipelining
  - High memory traffic

Stack Architectures

- Instruction set: add, sub, mul, div, . . .
  push A, pop A
- Example: A*B - (A+C*B)

```
push A
push B
mul
push A
push C
push B
mul
add
sub
```



Stacks: Pros and Cons

- Pros
  - Good code density (implicit top of stack)
  - Low hardware requirements
  - Easy to write a simpler compiler for stack architectures

Stacks: Pros and Cons (cont.)

- Cons
  - Stack becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Data is not always at the top of stack when need
  - Difficult to write an optimizing compiler for stack architectures

Memory-Memory Architectures

- Instruction set:

  (3 operands) add A, B, C   sub A, B, C
  mul A, B, C

  (2 operands) add A, B   sub A, B   mul A, B

- Example: A*B - (A+C*B)
  - 3 operands

3 operands	2 operands
mul D, A, B	mov D, A
mul E, C, B	mul D, B
add E, A, E	mov E, C
sub E, D, E	mul E, B
	add E, A
	sub D, E

Memory-Memory: Pros and Cons

- Pros
  - Requires fewer instructions
  - Easy to write compilers for
- Cons
  - Very high memory traffic
  - Variable number of clocks per instruction
  - With two operands, more data movements are required

Register-Memory Architectures

- Instruction set:

```
add R1,  A          sub R1, A          mul R1, B
load R1, A          store R1, A
```

- Example: A*B - (A+C*B)

```
load R1, A
mul R1, B          /*     A*B            */
load R2, C
mul R2, B          /*     C*B            */
add R2, A          /*     A + CB               */
store R2, D
sub R1, D          /*     AB - (A + C*B)   */
```

Memory-Register: Pros and Cons

- Pros
  - Some data can be accessed without loading first
  - Instruction format easy to encode
  - Good code density
- Cons
  - Operands are not equivalent (poor orthogonal)
  - Variable number of clocks per instruction
  - May limit number of registers

Register-Register/Load-Store Architectures

- Instruction set:

```
add R1,  R2, R3     sub R1, R2, R3     mul R1, R2, R3
load R1, A          store R1, A        move R1, R2
```

- Example: A*B - (A+C*B)

```
load R1, A
load R2, B
load R3, C
mul R7, R3, R2      /*    C*B              */
add R8, R7, R1      /*    A + C*B              */
mul R9, R1, R2      /*    A*B              */
sub R10, R9, R8     /*    A*B - (A+C*B)    */
```

Register-Register/Load-Store: Pros and Cons

- Pros
  - Simple, fixed length instruction encodings
  - Instructions take similar number of cycles
  - Relatively easy to pipeline
- Cons
  - Higher instruction count
  - Dependent on good compiler

Registers: Advantages and Disadvantages

- Advantages
  - Faster than cache or main memory (no addressing mode)
  - Deterministic (no misses)
  - Can replicate (multiple read ports)
  - Short identifier (typically 3 to 8 bits)
  - Reduce memory traffic

Registers: Advantages and Disadvantages (cont.)

- Disadvantages
  - Need to save and restore on procedure calls and context switch
  - Can't take the address of a register (for pointers)
  - Fixed size (can't store strings or structures efficiently)
  - Compiler must manage
  - Limited number

Current Trends: Computation Model

- Practically every modern design uses a load-store architecture
- For a new ISA design:
  - would expect to see load-store with plenty of general purpose registers

Byte Ordering

- Little Endian (e.g., in DEC, Intel)
  - » low order byte stored at lowest address
  - » byte0 byte1 byte2 byte3
- Big Endian (e.g., in IBM, Motorolla, Sun, HP)
  - » high order byte stored at lowest address
  - » byte3 byte2 byte1 byte0
- Programmers/protocols should be careful when transferring binary data between Big Endian and Little Endian machines

Operand Alignment

- An access to an operand of size s bytes at byte address A is said to be *aligned* if

$$A \bmod s = 0$$

40	41	42	43	44
D0	D1	D2	D3	
	D0	D1	D2	D3

Unrestricted Alignment

- If the architecture does not restrict memory accesses to be aligned then
  - Software is simple
  - Hardware must detect misalignment and make two memory accesses
  - Expensive logic to perform detection
  - Can slow down all references
  - Sometimes required for backwards compatibility

Restricted Alignment

- If the architecture restricts memory accesses to be aligned then
  - Software must guarantee alignment
  - Hardware detects misalignment access and traps
  - No extra time is spent when data is aligned
- Since we want to make the common case fast, having restricted alignment is often a better choice, unless compatibility is an issue.

Types of Addressing Modes (VAX)

Addressing Mode	Example	Action
1. Register direct	Add R4, R3	R4 <- R4 + R3
2.Immediate	Add R4, #3	R4 <- R4 + 3
3.Displacement	Add R4, 100(R1)	R4 <- R4 + M[100 + R1]
4.Register indirect	Add R4, (R1)	R4 <- R4 + M[R1]
5.Indexed	Add R4, (R1 + R2)	R4 <- R4 + M[R1 + R2]
6.Direct	Add R4, (1000)	R4 <- R4 + M[1000]
7.Memory Indirect	Add R4, @(R3)	R4 <- R4 + M[M[R3]]

Types of Addressing Modes (cont.)

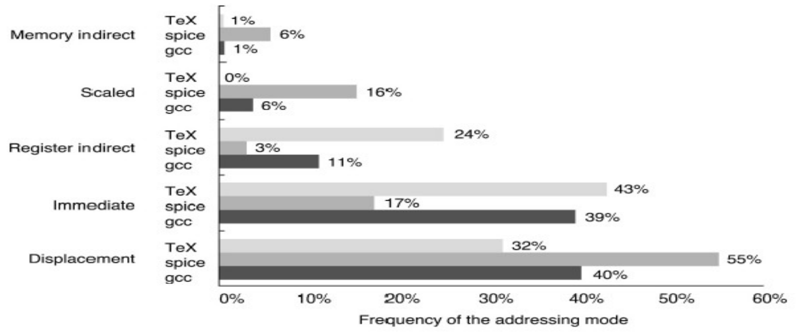
8.Autoincrement	Add R4, (R2)+	R4 <- R4 + M[R2]
		R2 <- R2 + d

9.Autodecrement	Add R4, (R2)-	R4 <- R4 + M[R2]
		R2 <- R2 - d

10.Scaled	Add R4, 100(R2)[R3]	R4 <- R4 + M[100 + R2 + R3*d]

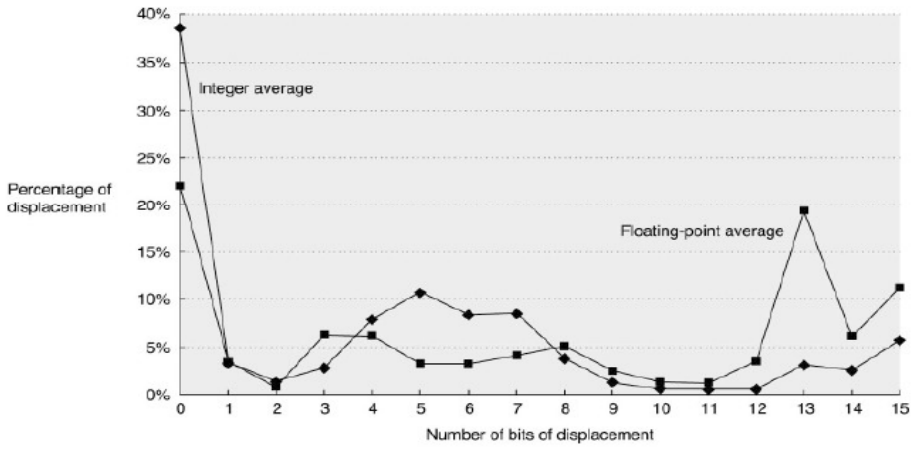
- Studies by [Clark and Emer] indicate that modes 1-4 account for 93% of all operands on the VAX.

Use of Addressing Modes (VAX)

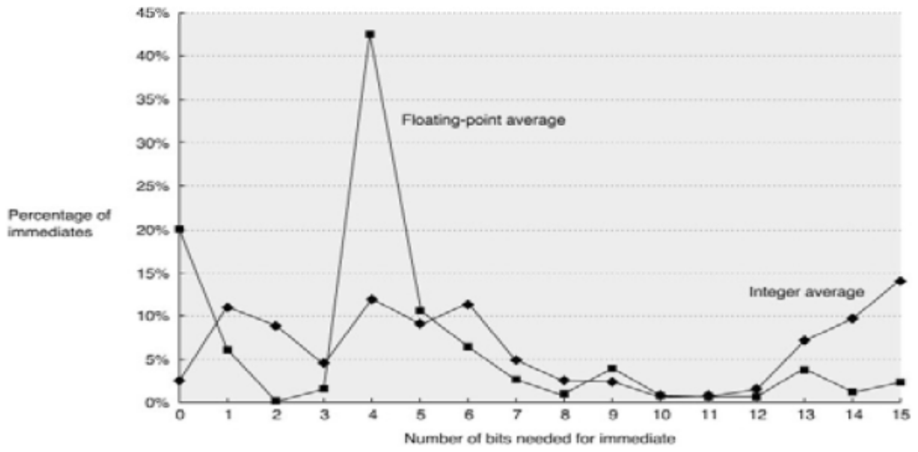


- Displacement dominates the memory addressing mode (32% - 55%).
- Immediate tails displacement (17% - 43%).

Displacement Value Distribution



Immediate Value Distribution



Current Trends: Memory Addressing

- For a new ISA design would expect to see:
  - support for displacement, immediate and register indirect addressing modes
    - »75% to 99% of SPEC measurements
  - size of address for displacement to be at least 12-16 bits
  - size of immediate field to be at least 8-16 bits
- As use of compilers becomes more dominant, emphasis is on simpler addressing modes

Types of Operations

• Arithmetic and Logic:	AND, ADD
• Data Transfer:	MOVE, LOAD, STORE
• Control	BRANCH, JUMP, CALL
• System	OS CALL
• Floating Point	ADDF, MULF, DIVF
• Decimal	ADDD, CONVERT
• String	MOVE, COMPARE
• Graphics	(DE)COMPRESS

Instruction Frequency (Intel 80x86 Integer)

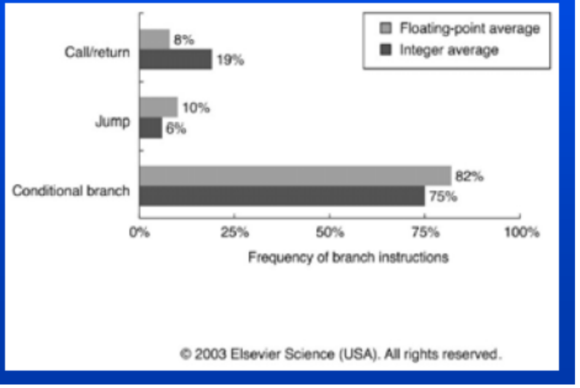
load	22%
conditional branch	20%
compare	16%
store	12%
add	8%
and	6%
sub	5%
register move	4%
call	1%
return	1%
total	96%

Current Trends: Operations and Operands

- For a new ISA design would expect to see:
  - support for 8, 16, 32, (64) bit integers
  - support for 32 and 64 bit IEEE 754 floating point
  - emphasis on efficient implementation of the simple common operations

Control Flow

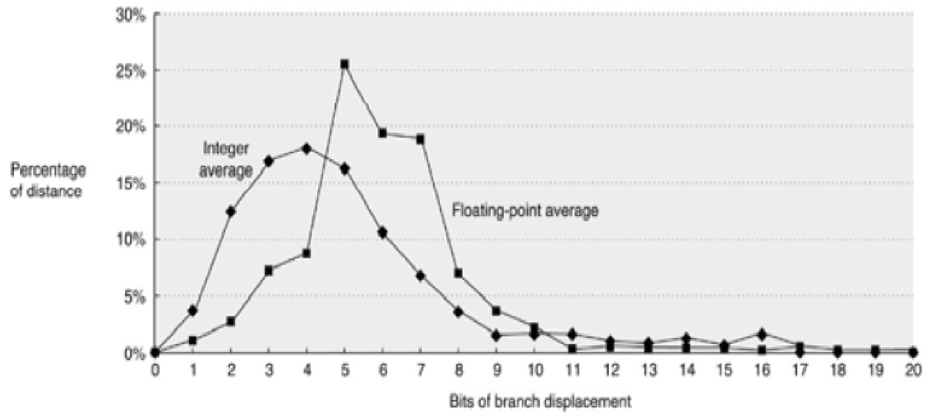
- Types
  - conditional branch
  - jump
  - procedure call
  - procedure return



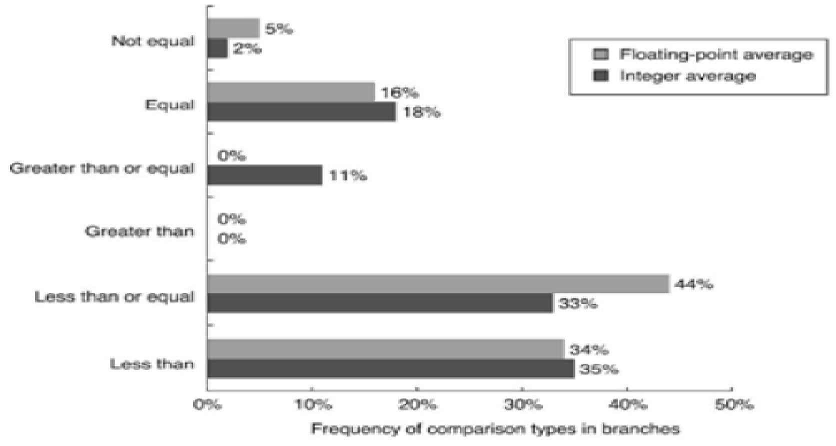
Addressing Modes

- Need to specify destination address
  - usually explicitly
    - »displacement relative to PC (why?)
  - sometimes not known statically
    - »specify register containing address

Jump Distances (Alpha)



Comparisons in Conditional Branches (Alpha)



Procedure Call/Return

- Need to save return address somewhere (minimum)
  - special register or GPR
- CISC architectures tended to provide more complex support
  - e.g. saving whole batch of registers
  - now just get the compiler to generate code for this

Procedure Call/Return (cont.)

- Caller save
  - calling procedure saves registers it will need afterwards
- Callee save
  - called procedure saves registers it needs to use

CALLS Instruction (VAX)

- Callee save

  1. Align stack if needed
  2. Push argument count on stack
  3. Save registers indicated by procedure call mask on stack
  4. Push return address on stack, push top & base pointers

CALLS Instruction (cont.)

5. Clear condition codes
6. Push status word
7. Update stack pointers
8. Branch to first instruction

Current Trends: Control Flow

- For a new ISA design would expect to see:
  - conditional branches able to jump hundreds of instructions forwards or backwards
    - »PC-relative branch displacement of at least 8 bits
  - jumps using PC-relative and register indirect addressing

Encoding Instruction Set

- Encoding of instructions affects:
  - size of compiled program
  - decoding work required by processor
- Depends on:
  - range of addressing modes
  - degree of independence between opcodes and modes

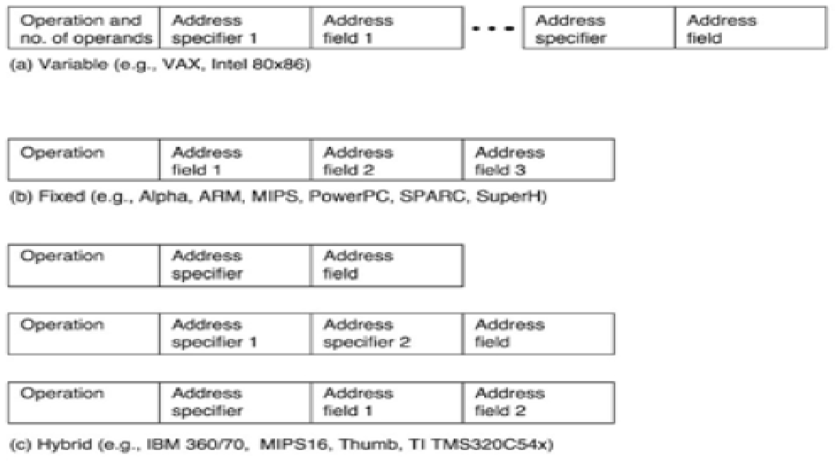
Three Competing Forces

- Desire to have as many registers and addressing modes as possible
- Desire to reduce the size of instructions and programs
- Desire to ease the decoding and implementation of instructions

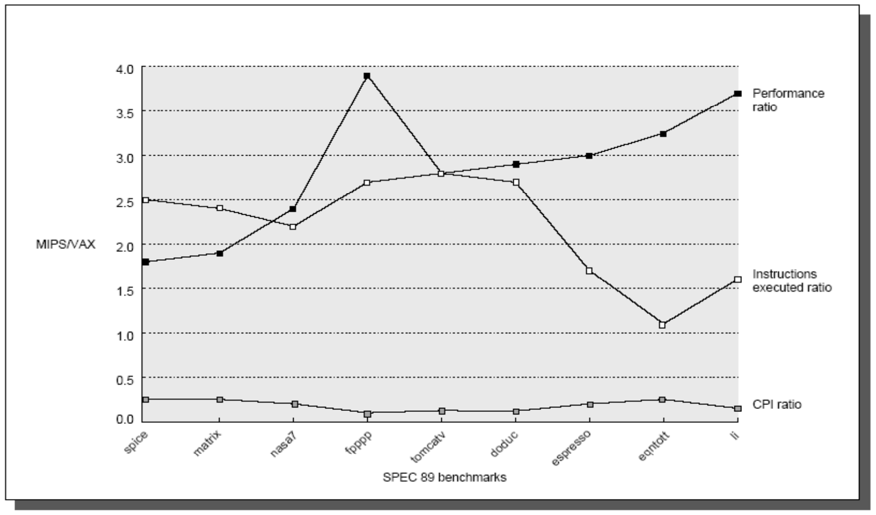
Three Basic Variations

- Variable instruction length (VAX)
  - independence between addressing mode and opcode (operation)
  - use address specifier: (mode, reg)
- Fixed instruction length (MIPS)
  - small number of addressing modes
  - use opcode to imply the addressing mode
- Hybrid approach

Three Basic Variations (cont.)



Performance Ratio between MIPS and VAX



Why does MIPS beat VAX?

- IC_{MIPS} is about $2*IC_{VAX}$
- CPI_{MIPS} is about $CPI_{VAX}/6$
- MIPS is about 3 times as fast as VAX given the same clock cycle times

Current Trends: Encoding Instruction Set

- For a new ISA design would expect to see:
  - fixed length instruction encoding, or hybrid
  - choice will be dictated by previous design decisions

ISA vs. Computer Architecture

- Old definition of computer architecture = instruction set design
  - Other aspects of computer design called implementation
  - Insinuates implementation is uninteresting or less challenging
- Our view is computer architecture >> ISA

ISA vs. Computer Architecture (Cont.)

- Architect's job much more than instruction set design;
- Technical hurdles today *more* challenging than those in instruction set design

Comp. Arch. is an Integrated Approach

- What really matters is the functioning of the complete system
  - hardware, runtime system, compiler, operating system, and application
  - In networking, this is called the "End to End argument"

Comp. Arch. is an Integrated Approach (Cont.)

- Computer architecture is not just about transistors, individual instructions, or particular implementations
  - E.g., Original RISC projects replaced complex instructions with a compiler + simple instructions

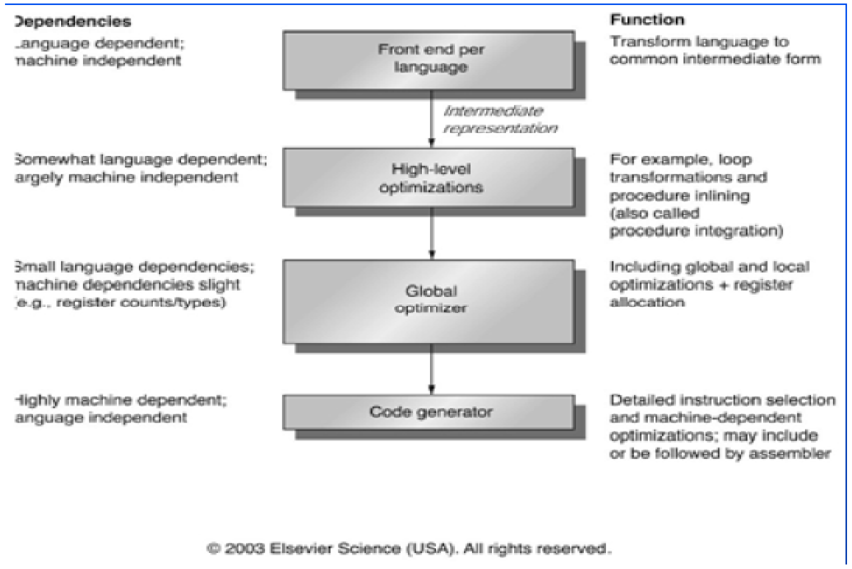
The Role of Compiler

- Why is compiler important to computer designers?
- Instruction set architecture is the target of compiler.
  - Instruction set architecture should allow compilers to generate efficient code.

The Role of Compiler (cont.)

- Compiler optimization affects the performance of the code generated.
  - Instruction set architecture should allow to simplify compilers. efficient code.
  - Computer designers should know the limitation of compiler optimization.

Compiler Structure



ependencies
anguage dependent; nachine independent
Front end per language
Function
Transform language to common intermediate form
Intermediate representation
omewhat language dependent; argely machine independent
High-level optimizations
For example, loop transformations and procedure inlining (also called procedure integration)
mall language dependencies; nachine dependencies slight e.g., register counts/types)
Global optimizer
Including global and local optimizations + register allocation
ighly machine dependent; anguage independent
Code generator
Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler
© 2003 Elsevier Science (USA). All rights reserved.

Compiler Goals

- Correctness
- Speed of compiled code
- Speed of compiler
- Debugging support

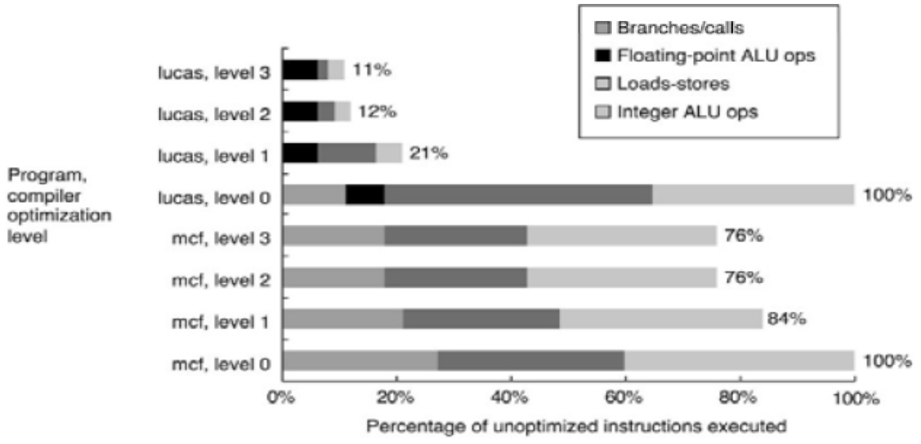
Optimizations

- High-level
  - Procedure integration: procedure in-lining
- Local: within basic block (linear code fragment)
  - Common expression elimination (18%)
  - Constant propagation (22%)
  - Stack height reduction

Optimizations (cont.)

- Global: across branches, loop optimization
  - Global common expression elimination (13%)
  - Copy propagation (11%)
  - Code motion (16%), e.g., move code out of the loop
  - Induction variable elimination (2%), e.g., simplify or remove location calculation of elements in array
- Machine-Dependent
  - Strength reduction, e.g., use shift instead of multiplication
  - Pipeline scheduling
  - Branch offset optimization

Effectiveness of Optimization



Limitation of Compiler Optimization

- Phase-ordering problem
  - too hard, and expensive, to undo transformation steps
  - high-level transformations carried out before form of resulting code is known
- Alias prevents allocating registers to variables

How the Architect Can Help the Compiler Writer

- Factors behind compiler complexity:
  - programs are locally simple but globally complex
  - structure of compilers means optimization decisions are made linearly

How the Architect Can Help the Compiler Writer (cont.)

- Graph coloring for register allocation is more efficient with more than 16 registers.
- Alias limits the use of large number of registers
- Make the frequent cases fast and the rare case correct...

How the Architect Can Help the Compiler Writer (cont.)

- Desirable ISA properties
  - Regularity
    - Orthogonality among addressing mode, data types, and operations
  - Simplicity
    - Provide primitives, not solutions
  - Simplify tradeoffs
    - That the compiler has to make
  - Provide instructions that bind the quantities known at compile-time