EI338: Computer Systems and Engineering

(Computer Architecture & Operating Systems)

Chentao Wu 吴晨涛
Associate Professor
Dept. of Computer Science and Engineering
Shanghai Jiao Tong University

SEIEE Building 3-513
wuct@cs.sjtu.edu.cn

Download Lectures

- ftp://public.sjtu.edu.cn
- User: wuct
- Password: wuct123456

Appendix A

Instruction Set Principles

Outline

- Instruction Set Architecture
- 5 stage pipelining
- Structural and Data Hazards
- Forwarding
- Branch Schemes
- Exceptions and Interrupts
- Conclusion

Instruction Set Architecture

- **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.**

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

Evolution of Instruction Sets

上海交通大學

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 432 1977-80)

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

RISC
(Mips,Sparc,HP-PA,IBM RS6000,PowerPC . . .1987)

LIW/"EPIC"? (IA-64. . .1999)

Evolution of Instruction Sets

- Major advances in computer architecture are typically associated with landmark instruction set designs
  - Ex: Stack vs GPR (System 360)
- Design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems
- And they in turn influence these

Instructions Can Be Divided into 3 Classes (I)

- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - *Load*—source is memory and destination is register
  - *Store*—source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Change the form of one or more operands to produce a result stored in another location
  - *Add*, *Sub*, *Shift*, etc.
- Branch instructions (control flow instructions)
  - Alter the normal flow of control from executing the next instruction in sequence
  - *Br Loc*, *Brz Loc2*,—unconditional or conditional branches

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]

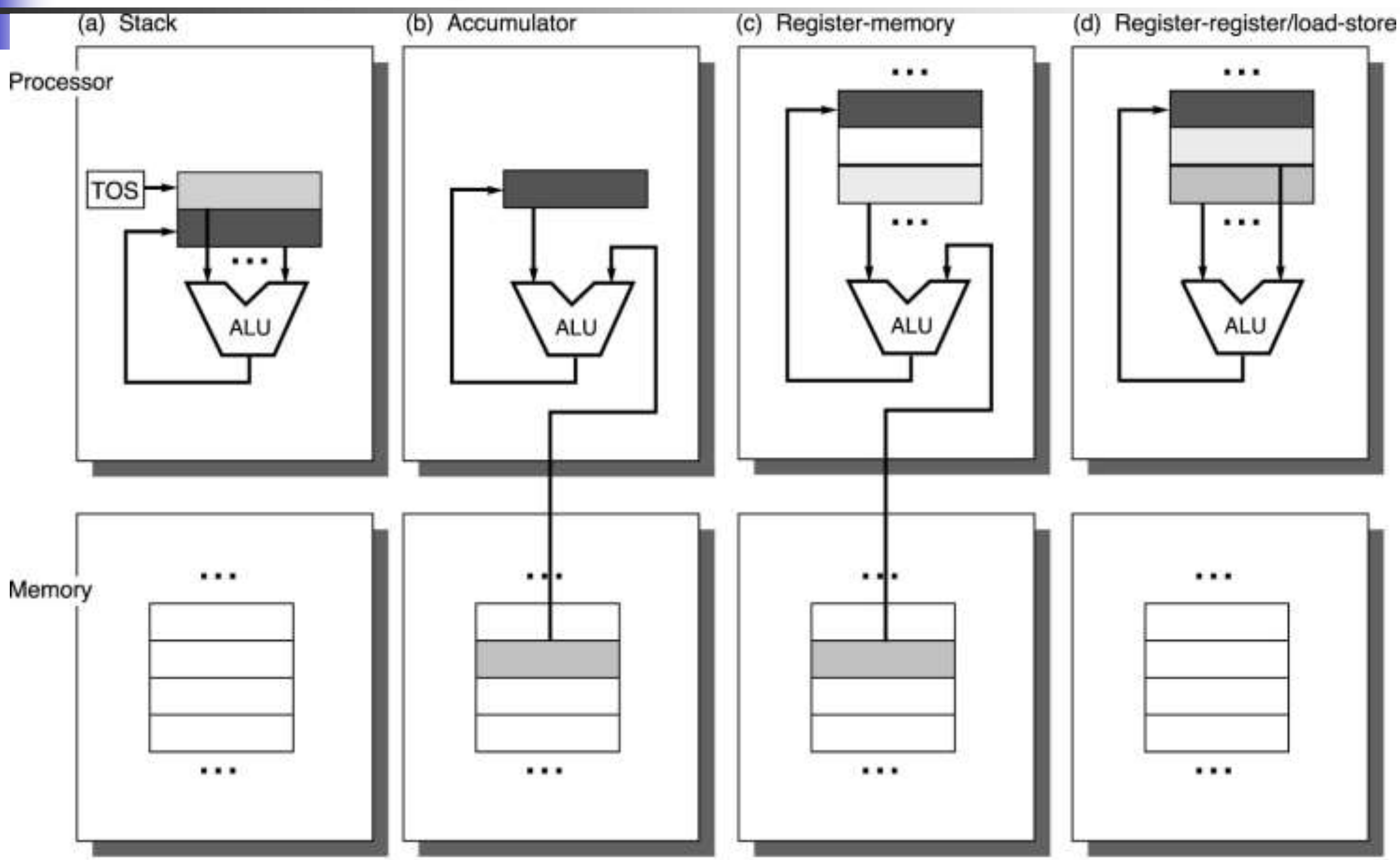
Register-Memory (1970s to present):

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



Stack Architectures

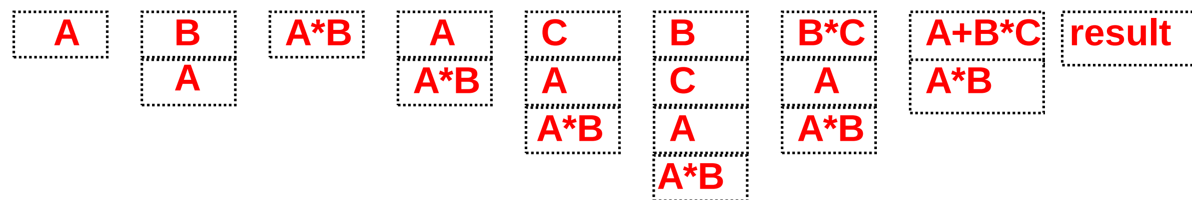
- **Instruction set:**

  add, sub, mult, 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 operand addressing→ top of stack)
  - Low hardware requirements
  - Easy to write a simpler compiler for stack architectures
- Cons
  - Stack becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Data is not always at the top of stack when need, so additional instructions like TOP and SWAP are needed
  - Difficult to write an optimizing compiler for stack architectures

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

B	B*C	A+B*C	A+B*C	A	A*B	result

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

Memory-Memory Architectures

- **Instruction set:**

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

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

    mul D, A, B

    mul E, C, B

    add E, A, E

    sub E, D, E

Memory-Memory: Pros and Cons

- Pros
  - Requires fewer instructions (especially if 3 operands)
  - Easy to write compilers for (especially if 3 operands)
- Cons
  - Very high memory traffic (especially if 3 operands)
  - Variable number of clocks per instruction (especially if 2 operands)
  - 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 */

  store R1, D

  load R2, C

  mul R2, B   /* C*B */

  add R2, A   /* A + CB */

  sub R2, 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 orthogonality)
  - Variable number of clocks per instruction
  - May limit number of registers

Load-Store Architectures

- Instruction set:

  add R1, R2, R3 sub R1, R2, R3 mul R1, R2, R3

  load R1, R4 store R1, R4

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

```
load R1, &A
load R2, &B
load R3, &C
load R4, R1
load R5, R2
load R6, R3
mul R7, R6, R5          /*   C*B             */
add R8, R7, R4          /*   A + C*B         */
mul R9, R4, R5          /*   A*B             */
sub R10, R9, R8         /*   A*B - (A+C*B)   */
```

Load-Store: Pros and Cons

- Pros
  - Simple, fixed length instruction encoding
  - Instructions take similar number of cycles
  - Relatively easy to pipeline

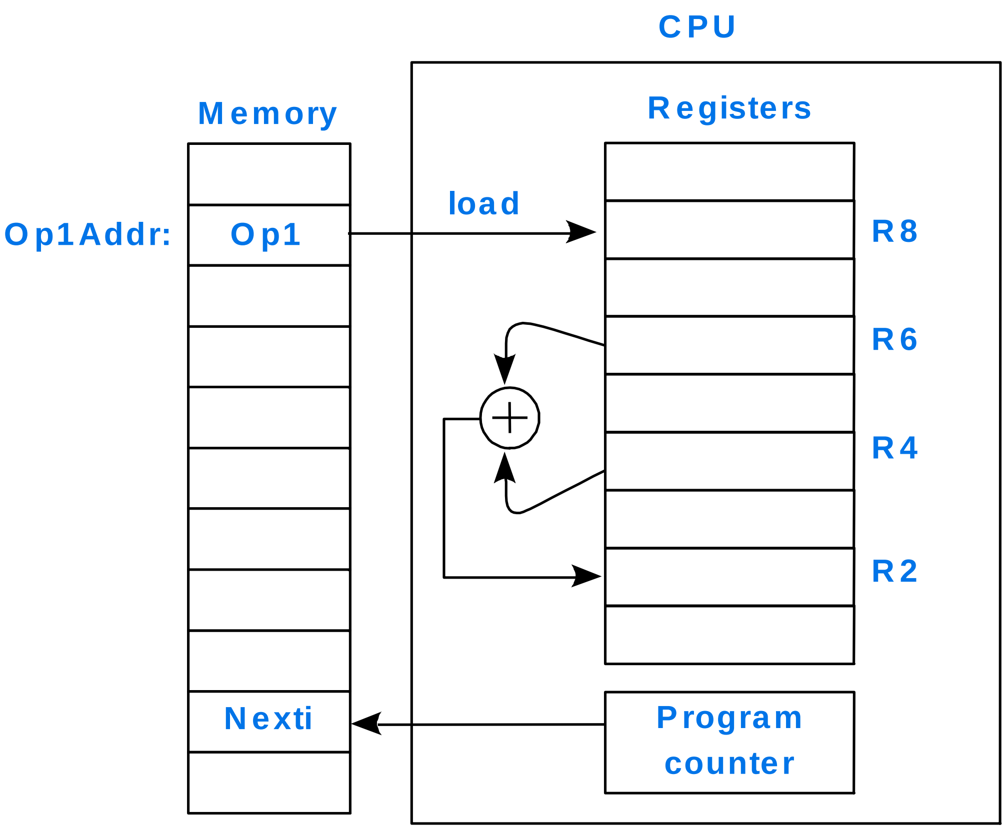
- Cons
  - Higher instruction count
  - Not all instructions need three operands
  - <u>Dependent on good compiler</u>

Registers: Advantages and Disadvantages

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

- 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

General Register Machine and Instruction Formats



Instruction formats

load R8, Op1 (R8 ฌ Op1)

load	R8	Op1Addr

add R2, R4, R6 (R2 ฌ R4 + R6)

add	R2	R4	R6

General Register Machine and Instruction Formats

- It is the most common choice in today's general-purpose computers
- *Which* register is specified by small "address" (3 to 6 bits for 8 to 64 registers)
- Load and store have one long & one short address: One and half addresses
- Arithmetic instruction has 3 "half" addresses

Real Machines Are Not So Simple

- Most real machines have a mixture of 3, 2, 1, 0, and 1- address instructions
- A distinction can be made on whether arithmetic instructions use data from memory
- If ALU instructions only use registers for operands and result, machine type is load-store
  - Only load and store instructions reference memory
- Other machines have a mix of register-memory and memory-memory instructions

Alignment Issues

- If the architecture does not restrict memory accesses to be aligned then
  - Software is simple
  - Hardware must detect misalignment and make 2 memory accesses
  - Expensive detection logic is required
  - All references can be made slower
- Sometimes unrestricted alignment is required for backwards compatibility
- 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

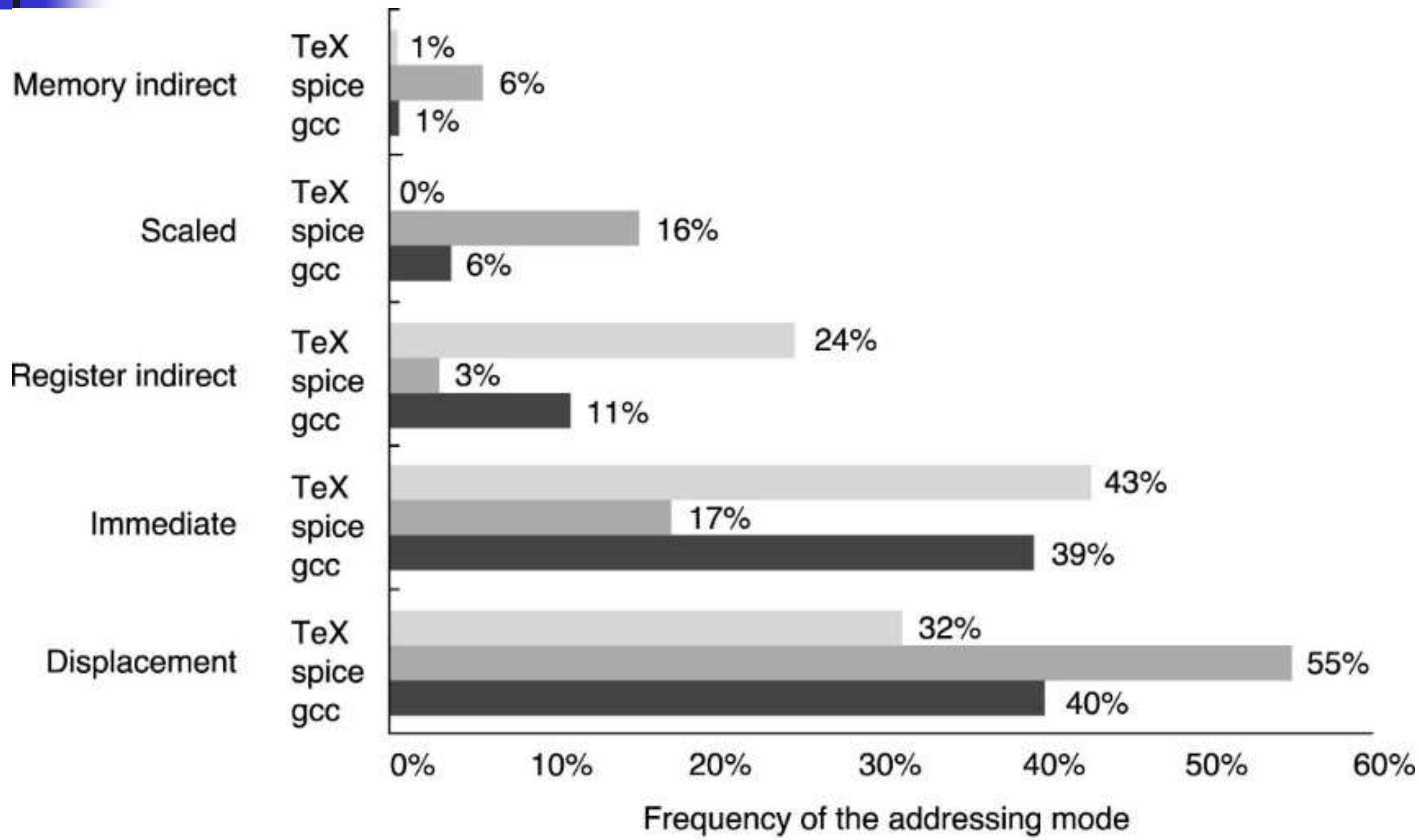
Types of Addressing Modes (VAX)

1.	Register direct	Ri
2.	Immediate (literal)	#n
3.	Displacement	M[Ri + #n]
4.	Register indirect	M[Ri]
5.	Indexed	M[Ri + Rj]
6.	Direct (absolute)	M[#n]
7.	Memory Indirect	M[M[Ri]]
8.	Autoincrement	M[Ri++]
9.	Autodecrement	M[Ri - -]
10.	Scaled	M[Ri + Rj*d + #n]

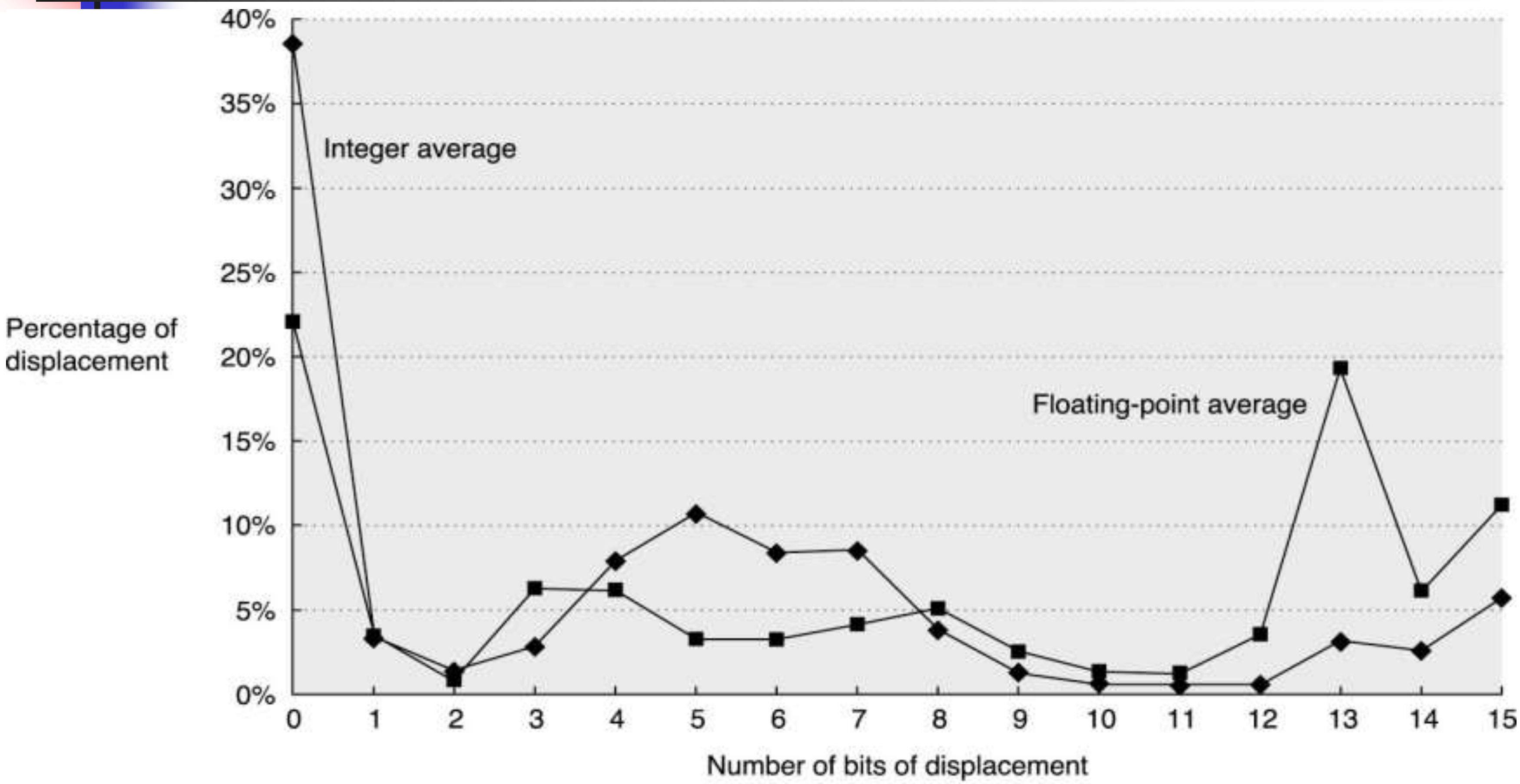
reg. file

memory

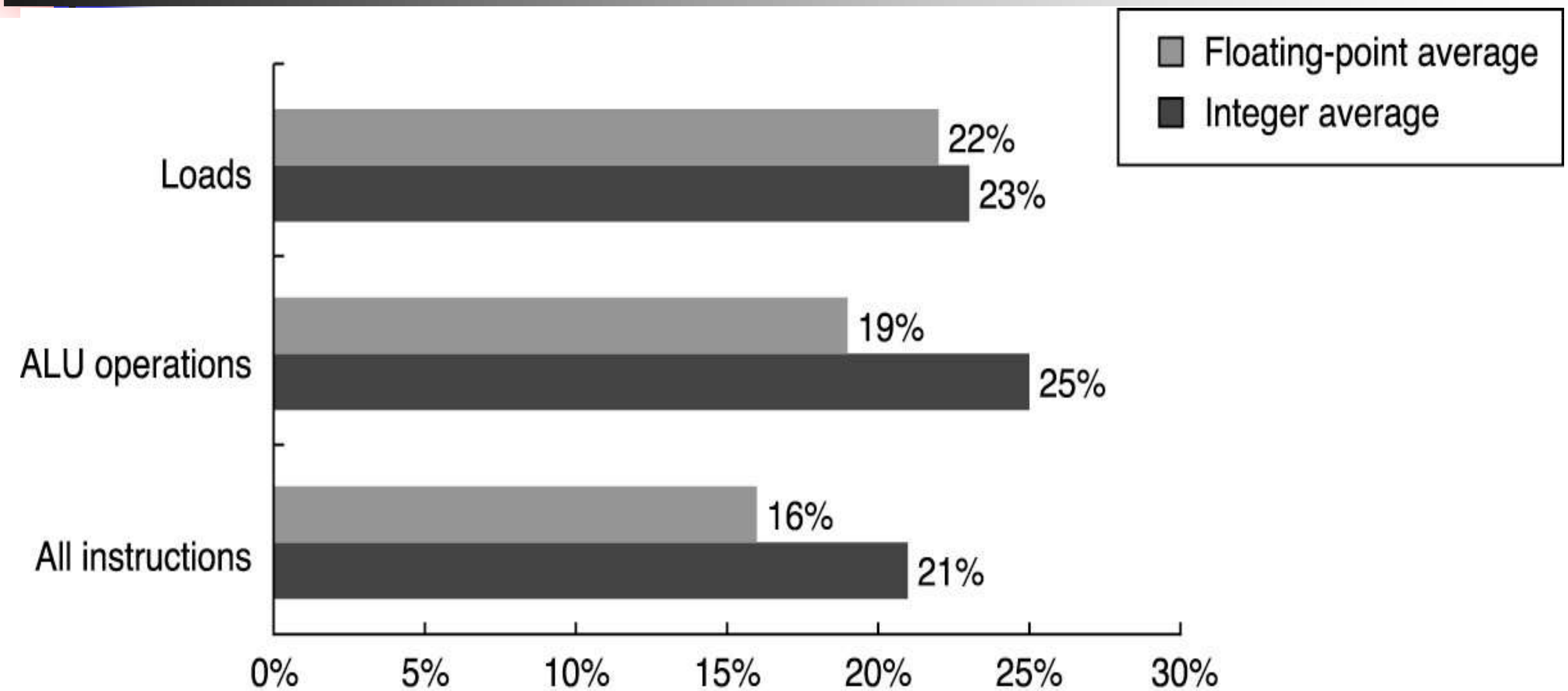
Summary of Use of Addressing Modes



Distribution of Displacement Values



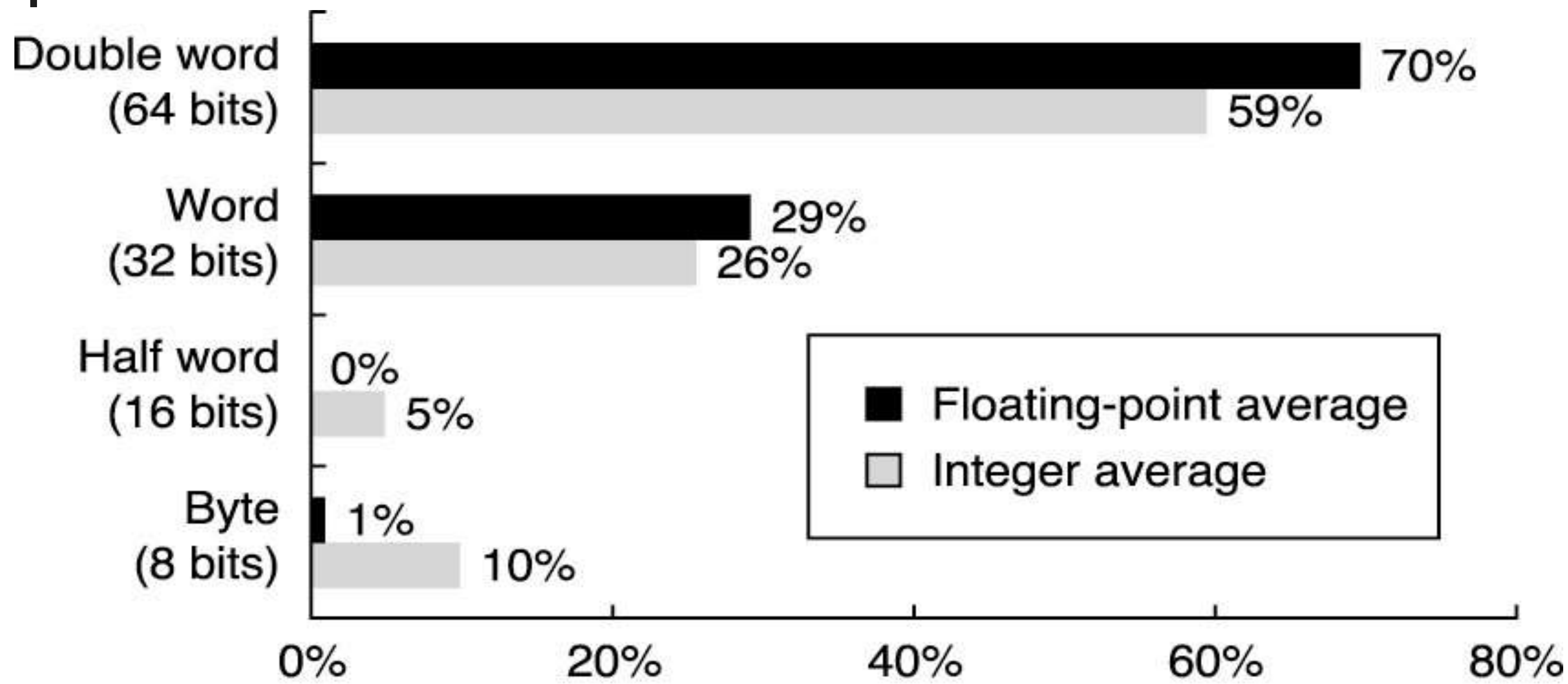
Frequency of Immediate Operands



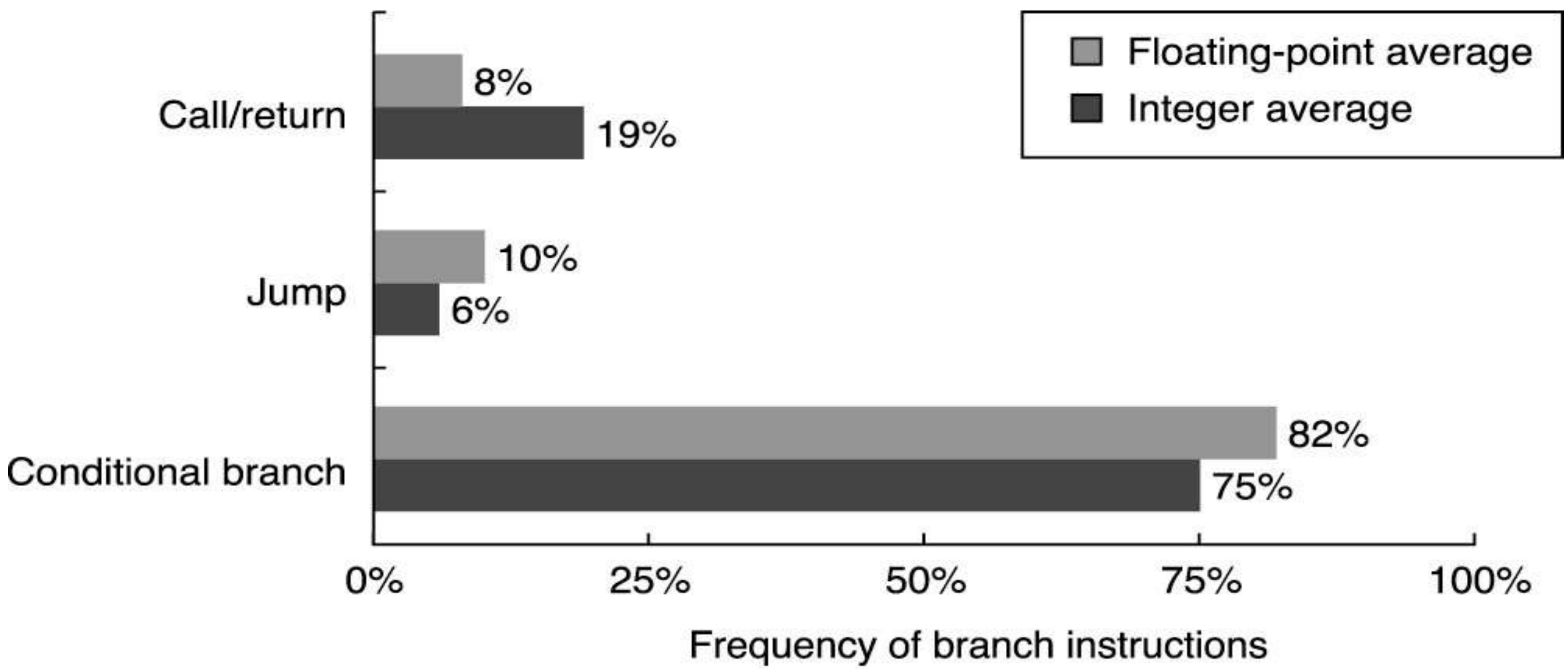
Types of Operations

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

Distribution of Data Accesses by Size



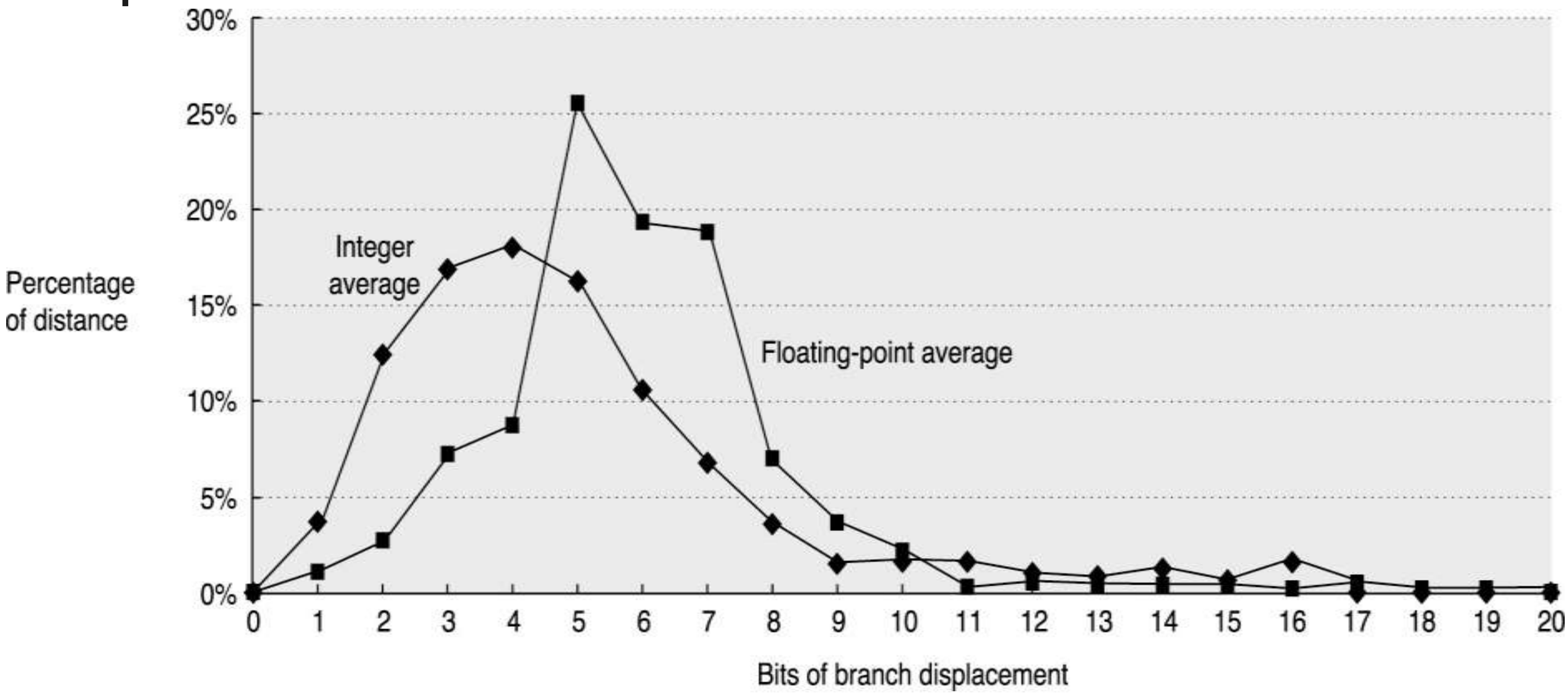
Relative Frequency of Control Instructions



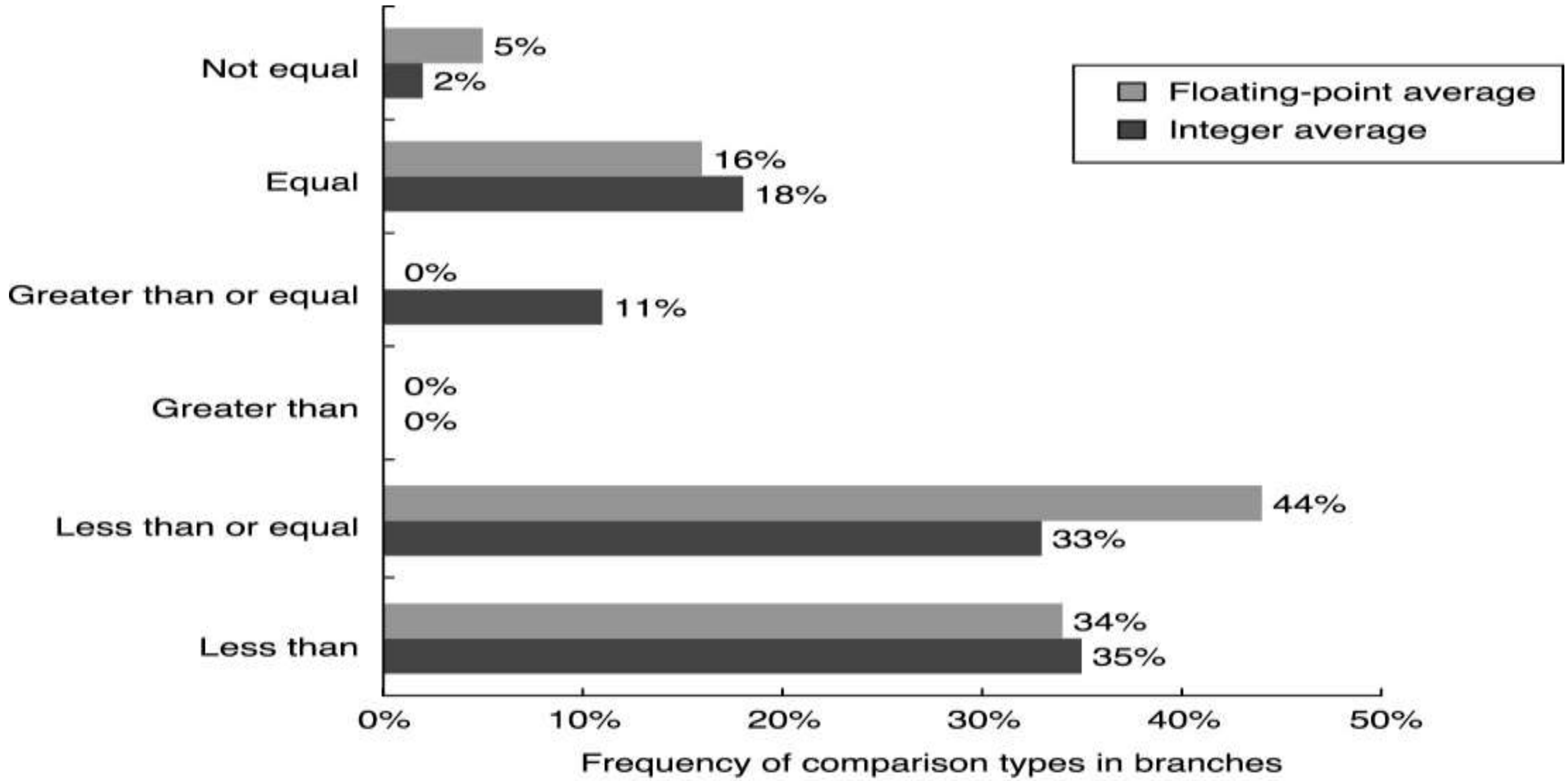
Control instructions (contd.)

- Addressing modes
  - PC-relative addressing (independent of program load & displacements are close by)
    - Requires displacement (how many bits?)
    - Determined via empirical study. [8-16 works!]
  - For procedure returns/indirect jumps/kernel traps, target may not be known at compile time.
    - Jump based on contents of register
    - Useful for switch/(virtual) functions/function ptrs/dynamically linked libraries etc.

Branch Distances (in terms of number of instructions)



Frequency of Different Types of Compares in Conditional Branches



Encoding an Instruction set

- a desire to have as many registers and addressing mode as possible
- the impact of size of register and addressing mode fields on the average instruction size and hence on the average program size
- a desire to have instruction encode into lengths that will be easy to handle in the implementation

Three choice for encoding the instruction set

Operation and no. of operands	Address specifier 1	Address field 1	• • •	Address specifier	Address field

(a) Variable (e.g., VAX, Intel 80x86)

Operation	Address field 1	Address field 2	Address field 3

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operation	Address specifier	Address field

Operation	Address specifier 1	Address specifier 2	Address field

Operation	Address specifier	Address field 1	Address field 2

(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

Compilers and ISA

- Compiler Goals
  - All correct programs compile correctly
  - Most compiled programs execute quickly
  - Most programs compile quickly
  - Achieve small code size
  - Provide debugging support
- Multiple Source Compilers
  - Same compiler can compiler different languages
- Multiple Target Compilers
  - Same compiler can generate code for different machines

Compilers Phases

Dependencies		Function
Language dependent; machine independent	Front end per language	Transform language to common intermediate form
	Intermediate representation	
Somewhat language dependent; largely machine independent	High-level optimizations	For example, loop transformations and procedure inlining (also called procedure integration)
Small language dependencies; machine dependencies slight (e.g., register counts/types)	Global optimizer	Including global and local optimizations + register allocation
Highly machine dependent; language independent	Code generator	Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler

Compiler Based Register Optimization

- Assume small number of registers (16-32)
- Optimizing use is up to compiler
- HLL programs have no explicit references to registers
  - usually – is this always true?
- Assign symbolic or virtual register to each candidate variable
- Map (unlimited) symbolic registers to real registers
- Symbolic registers that do not overlap can share real registers
- If you run out of real registers some variables use memory

Allocation of Variables

- Stack
  - used to allocate local variables
  - grown and shrunk on procedure calls and returns
  - register allocation works best for stack-allocated objects
- Global data area
  - used to allocate global variables and constants
  - many of these objects are arrays or large data structures
  - impossible to allocate to registers if they are *aliased*
- Heap
  - used to allocate dynamic objects
  - heap objects are accessed with pointers
  - never allocated to registers

Designing ISA to Improve Compilation

- Provide enough general purpose registers to ease register allocation (more than 16).
- Provide regular instruction sets by keeping the operations, data types, and addressing modes orthogonal.
- Provide primitive constructs rather than trying to map to a high-level language.
- Simplify trade-off among alternatives.
- Allow compilers to help make the common case fast.

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
- Regularity
  - No overloading for the meanings of instruction fields
- Streamlined Design
  - Resource needs easily determined. Simplify tradeoffs.
- Ease of compilation (programming?), Ease of implementation, Scalability

Quick Review of Design Space of ISA

Five Primary Dimensions

- Number of explicit operands (0, 1, 2, 3)
- Operand Storage — Where besides memory?
- Effective Address — How is memory location specified?
- Type & Size of Operands — byte, int, float, vector, . . . How is it specified?
- Operations — add, sub, mul, . . . How is it specifed?

Other Aspects

- Successor — How is it specified?
- Conditions — How are they determined?
- Encodings — Fixed or variable? Wide?
- Parallelism

ISA Metrics

Aesthetics:

- 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
- Regularity
  - No overloading for the meanings of instruction fields
- Streamlined
  - Resource needs easily determined

Ease of compilation (programming?)

Ease of implementation

Scalability

A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, Double Precision takes a register pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store: base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

see: SPARC, MIPS, MC88100, AMD2900, i960, i860
PARisc, DEC Alpha, Clipper,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

MIPS data types

- Bytes
  - characters
- Half-words
  - Short ints, OS related data-structures
- Words
  - Single FP, Integers
- Doublewords
  - Double FP, Long Integers (in some implementations)

Instruction Layout for MIPS

I-type instruction

6	5	5	16
Opcode	rs	rt	Immediate

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

6	5	5	5	5	6
Opcode	rs	rt	rd	shamt	funct

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

6	26
Opcode	Offset added to PC

Jump and jump and link
Trap and return from exception

MIPS (32 bit instructions)

1. Register-Register

31 26	25 21	20 16	15 11	10 6	5 0
Op	Rs1	Rs2	Rd		Opx

2a. Register-Immediate

31 26	25 21	20 16	15 0
Op	Rs1	Rd	Immediate

2b. Branch (displacement)

31 26	25 21	20 16	15 0
Op	Rs1	Rs2/Opx	Displacement

3. Jump / Call

31 26	25 0
Op	target

MIPS (addressing modes)

- Register direct
- Displacement
- Immediate
- Byte addressable & 64 bit address
- R0 ← always contains value 0
- Displacement = 0→ register indirect
- R0 + Displacement=0 → absolute addressing

Types of Operations

- Loads and Stores
- ALU operations
- Floating point operations
- Branches and Jumps (control-related)

Load/Store Instructions

Example instruction	Instruction name	Meaning
LD R1,30(R2)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[30+Regs[R2]]
LD R1,1000(R0)	Load double word	Regs[R1] $\leftarrow_{64}$ Mem[1000+0]
LW R1,60(R2)	Load word	Regs[R1] $\leftarrow_{64}$ (Mem[60+Regs[R2]]$_0$)^{32} ## Mem[60+Regs[R2]]
LB R1,40(R3)	Load byte	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{56} ## Mem[40+Regs[R3]]
LBU R1,40(R3)	Load byte unsigned	Regs[R1] $\leftarrow_{64}$ 0^{56} ## Mem[40+Regs[R3]]
LH R1,40(R3)	Load half word	Regs[R1] $\leftarrow_{64}$ (Mem[40+Regs[R3]]$_0$)^{48} ## Mem[40+Regs[R3]]##Mem[41+Regs[R3]]
L.S F0,50(R3)	Load FP single	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R3]] ## 0^{32}
L.D F0,50(R2)	Load FP double	Regs[F0] $\leftarrow_{64}$ Mem[50+Regs[R2]]
SD R3,500(R4)	Store double word	Mem[500+Regs[R4]] $\leftarrow_{64}$ Regs[R3]
SW R3,500(R4)	Store word	Mem[500+Regs[R4]] $\leftarrow_{32}$ Regs[R3]
S.S F0,40(R3)	Store FP single	Mem[40+Regs[R3]] $\leftarrow_{32}$ Regs[F0]$_{0..31}$
S.D F0,40(R3)	Store FP double	Mem[40+Regs[R3]] $\leftarrow_{64}$ Regs[F0]
SH R3,502(R2)	Store half	Mem[502+Regs[R2]] $\leftarrow_{16}$ Regs[R3]$_{48..63}$
SB R2,41(R3)	Store byte	Mem[41+Regs[R3]] $\leftarrow_{8}$ Regs[R2]$_{56..63}$

Figure 2.28 The load and store instructions in MIPS. All use a single addressing mode and require that the memory value be aligned. Of course, both loads and stores are available for all the data types shown.

Sample ALU Instructions

Example instruction	Instruction name	Meaning
DADDU R1,R2,R3	Add unsigned	Regs[R1]←Regs[R2]+Regs[R3]
DADDIU R1,R2,#3	Add immediate unsigned	Regs[R1]←Regs[R2]+3
LUI R1,#42	Load upper immediate	Regs[R1]←0^{32}##42##0^{16}
DSLL R1,R2,#5	Shift left logical	Regs[R1]←Regs[R2]<<5
DSLT R1,R2,R3	Set less than	if (Regs[R2]<Regs[R3]) Regs[R1]←1 else Regs[R1]←0

Figure 2.29 Examples of arithmetic/logical instructions on MIPS, both with and without immediates.

Control Flow Instructions

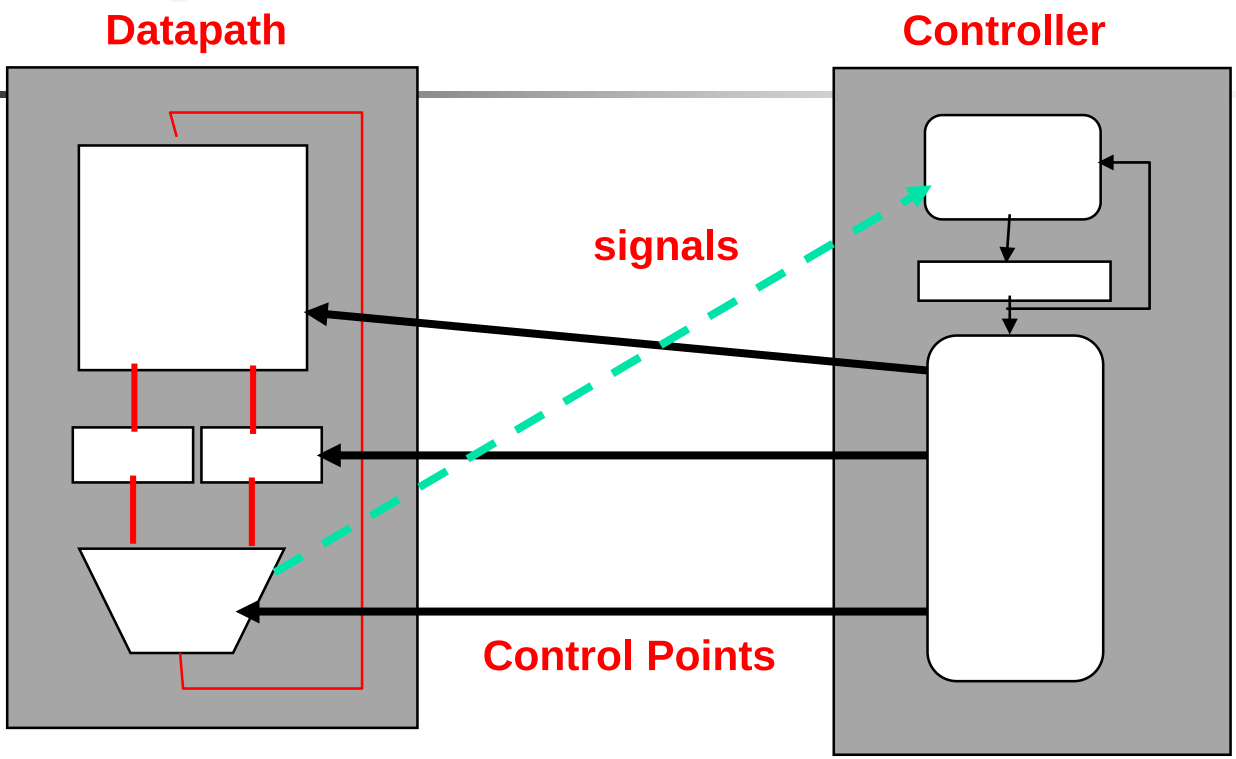
Example instruction	Instruction name	Meaning
J name	Jump	$PC_{36..63} \leftarrow$ name
JAL name	Jump and link	Regs[R31]$\leftarrow$PC+4; $PC_{36..63} \leftarrow$ name; $((PC+4)-2^{27}) \leq$ name $< ((PC+4)+2^{27})$
JALR R2	Jump and link register	Regs[R31]$\leftarrow$PC+4; PC$\leftarrow$Regs[R2]
JR R3	Jump register	PC$\leftarrow$Regs[R3]
BEQZ R4,name	Branch equal zero	if (Regs[R4]==0) PC$\leftarrow$name; $((PC+4)-2^{17}) \leq$ name $< ((PC+4)+2^{17})$
BNE R3,R4,name	Branch not equal zero	if (Regs[R3]!= Regs[R4]) PC$\leftarrow$name; $((PC+4)-2^{17}) \leq$ name $< ((PC+4)+2^{17})$
MOVZ R1,R2,R3	Conditional move if zero	if (Regs[R3]==0) Regs[R1]$\leftarrow$Regs[R2]

Figure 2.30 Typical control flow instructions in MIPS. All control instructions, except jumps to an address in a register, are PC-relative. Note that the branch distances are longer than the address field would suggest; since MIPS instructions are all 32 bits long, the byte branch address is multiplied by 4 to get a longer distance.

Instruction type/opcode	Instruction meaning
Data transfers	*Move data between registers and memory, or between the integer and FP or special registers; only memory address mode is 16-bit displacement + contents of a GPR*
LB,LBU,SB	Load byte, load byte unsigned, store byte (to/from integer registers)
LH,LHU,SH	Load half word, load half word unsigned, store half word (to/from integer registers)
LW,LWU,SW	Load word, load word unsigned, store word (to/from integer registers)
LD,SD	Load double word, store double word (to/from integer registers)
L.S,L.D,S.S,S.D	Load SP float, load DP float, store SP float, store DP float
MFC0,MTC0	Copy from/to GPR to/from a special register
MOV.S,MOV.D	Copy one SP or DP FP register to another FP register
MFC1,MTC1	Copy 32 bits from/to FP registers to/from integer registers
Arithmetic/logical	*Operations on integer or logical data in GPRs; signed arithmetic trap on overflow*
DADD,DADDI,DADDU, DADDIU	Add, add immediate (all immediates are 16 bits); signed and unsigned
DSUB,DSUBU	Subtract; signed and unsigned
DMUL,DMULU,DDIV, DDIVU,MADD	Multiply and divide, signed and unsigned; multiply-add; all operations take and yield 64-bit values
AND,ANDI	And, and immediate
OR,ORI,XOR,XORI	Or, or immediate, exclusive or, exclusive or immediate
LUI	Load upper immediate; loads bits 32 to 47 of register with immediate, then sign-extends
DSLL,DSRL,DSRA,DSLLV, DSRLV,DSRAV	Shifts: both immediate (DS__) and variable form (DS__V); shifts are shift left logical, right logical, right arithmetic
SLT,SLTI,SLTU,SLTIU	Set less than, set less than immediate; signed and unsigned
Control	*Conditional branches and jumps; PC-relative or through register*
BEQZ,BNEZ	Branch GPR equal/not equal to zero; 16-bit offset from PC + 4
BEQ,BNE	Branch GPR equal/not equal; 16-bit offset from PC + 4
BC1T,BC1F	Test comparison bit in the FP status register and branch; 16-bit offset from PC + 4
MOVN,MOVZ	Copy GPR to another GPR if third GPR is negative, zero
J,JR	Jumps: 26-bit offset from PC + 4 (J) or target in register (JR)
JAL,JALR	Jump and link: save PC + 4 in R31, target is PC-relative (JAL) or a register (JALR)
TRAP	Transfer to operating system at a vectored address
ERET	Return to user code from an exception; restore user mode
Floating point	*FP operations on DP and SP formats*
ADD.D,ADD.S,ADD.PS	Add DP, SP numbers, and pairs of SP numbers
SUB.D,SUB.S,ADD.PS	Subtract DP, SP numbers, and pairs of SP numbers
MUL.D,MUL.S,MUL.PS	Multiply DP, SP floating point, and pairs of SP numbers
MADD.D,MADD.S,MADD.PS	Multiply-add DP, SP numbers and pairs of SP numbers
DIV.D,DIV.S,DIV.PS	Divide DP, SP floating point, and pairs of SP numbers
CVT._._	Convert instructions: CVT.x.y converts from type x to type y, where x and y are L (64-bit integer), W (32-bit integer), D (DP), or S (SP). Both operands are FPRs.
C.__.D,C.__.S	DP and SP compares: "__" = LT,GT,LE,GE,EQ,NE; sets bit in FP status register

Figure 2.31 Subset of the instructions in MIPS64. Figure 2.27 lists the formats of these instructions. SP = single precision; DP = double precision. This list can also be found on the page preceding the back inside cover.

Datapath vs Control



- **Datapath: Storage, Functional Units, Interconnections sufficient to perform the desired functions**
  - **Inputs are Control Points**
  - **Outputs are signals**
- **Controller: State machine to orchestrate operation on the data path**
  - **Based on desired function and signals**

Approaching an ISA

- Instruction Set Architecture
  - Defines set of operations, instruction format, hardware supported data types, named storage, addressing modes, sequencing
- Meaning of each instruction is described by RTL (register transfer language) on *architected registers* and memory
- Given technology constraints, assemble adequate datapath
  - Architected storage mapped to actual storage
  - Function Units (FUs) to do all the required operations
  - Possible additional storage (eg. Internal registers: MAR, MDR, IR, …{Memory Address Register, Memory Data Register, Instruction Register}
  - Interconnect to move information among registers and function units
- Map each instruction to a sequence of RTL operations
- Collate sequences into symbolic controller state transition diagram (STD)
- Lower symbolic STD to control points
- Implement controller

Homework

- **A.1, A.5, A.7**