CMSC 611: Advanced Computer Architecture

Instruction Set Architecture (2)

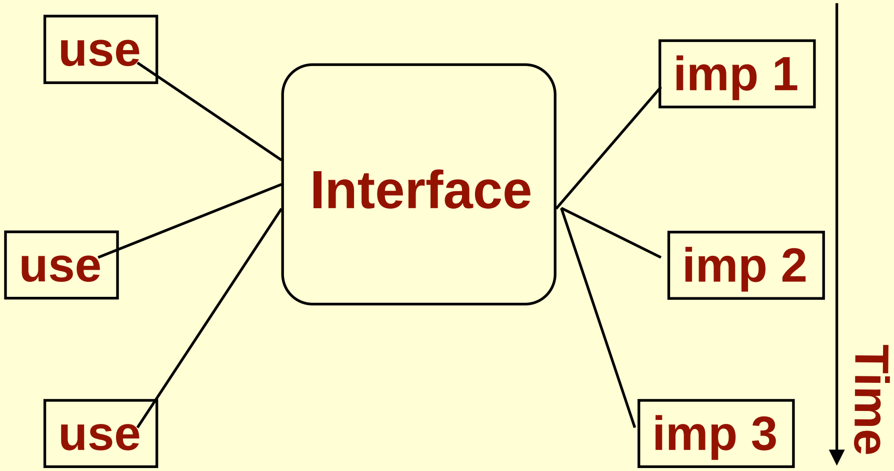
Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides
Some material adapted from Hennessy & Patterson / © 2003 Elsevier Science

Instruction Set Architecture

- To command a computer's hardware, you must speak its language
  - Instructions: the “words” of a machine's language
  - Instruction set: its “vocabulary
- Goals:
  - Introduce design alternatives
  - Present a taxonomy of ISA alternatives
    - + some qualitative assessment of pros and cons
  - Present and analyze some instruction set measurements
  - Address the issue of languages and compilers and their bearing on instruction set architecture
  - Show some example ISA’s

Interface Design

- A good interface:
  - Lasts through many implementations (portability, compatibility)
  - Is used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels
- Design decisions must take into account:
  - Technology
  - Machine organization
  - Programming languages
  - Compiler technology
  - Operating systems



Slide: Dave Patterson

Memory ISAs

- Terms
  - Result = Operand <operation> Operand
- Stack
  - Operate on top stack elements, push result back on stack
- Memory-Memory
  - Operands (and possibly also result) in memory

Register ISAs

- Accumulator Architecture
  - Common in early stored-program computers when hardware was expensive
  - Machine has only one register (accumulator) involved in all math & logic operations
  - Accumulator = Accumulator op Memory
- Extended Accumulator Architecture (8086)
  - Dedicated registers for specific operations, e.g stack and array index registers, added
- General-Purpose Register Architecture (MIPS)
  - Register flexibility
  - Can further divide these into:
    - Register-memory: allows for one operand to be in memory
    - Register-register (load-store): all operands in registers

Other types of Architecture

- High-Level-Language Architecture
  - In the 1960s, systems software was rarely written in high-level languages
    - virtually every commercial operating system before Unix was written in assembly
  - Some people blamed the code density on the instruction set rather than the programming language
  - A machine design philosophy advocated making the hardware more like high-level languages

Well Known ISA

- Stack
- Memory-Memory
- Accumulator Architecture
- Extended Accumulator Architecture
- General-Purpose Register Architecture

Machine	# general-purpose registers	Architecture style	Year
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	32	Register-memory	1985
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992

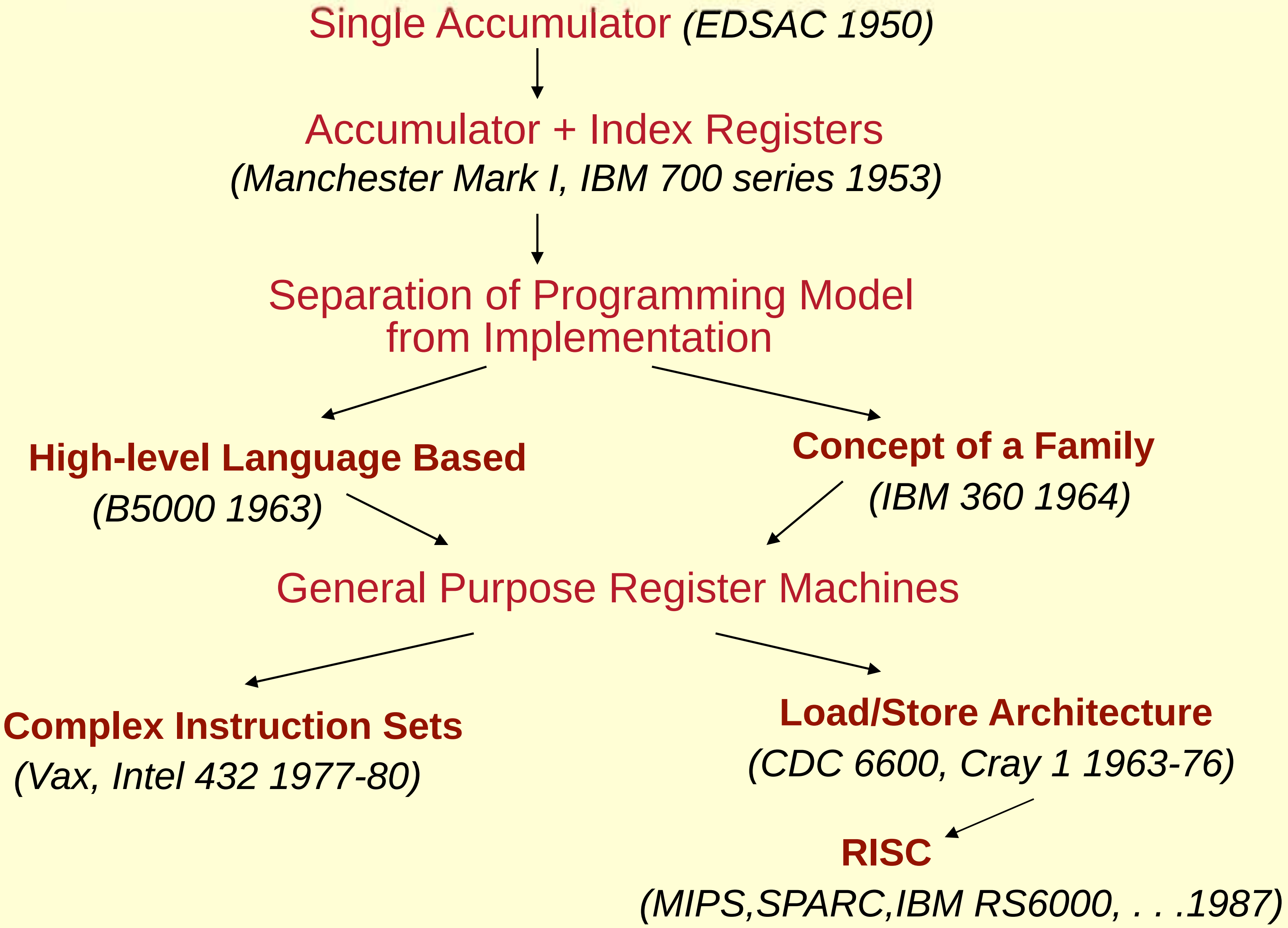
ISA Complexity

- Reduced Instruction Set Architecture
  - With the recent development in compiler technology and expanded memory sizes less programmers are using assembly level coding
  - Drives ISA to favor benefit for compilers over ease of manual programming
- RISC architecture favors simplified hardware design over rich instruction set
  - Rely on compilers to perform complex operations
- Virtually all new architecture since 1982 follows the RISC philosophy:
  - fixed instruction lengths, load-store operations, and limited addressing mode

Compact Code

- Scarce memory or limited transmit time (JVM)
- Variable-length instructions (Intel 80x86)
  - Match instruction length to operand specification
  - Minimize code size
- Stack machines abandon registers altogether
  - Stack machines simplify compilers
  - Lend themselves to a compact instruction encoding
  - BUT limit compiler optimization

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 432 1977-80)

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

RISC
(MIPS,SPARC,IBM RS6000, . . .1987)

Slide: Dave Patterson

Register-Memory Arch

# memory addresses	Max. number of operands	Examples
0	3	SPARC, MIPS, PowerPC, ALPHA
1	2	Intel 60X86, Motorola 68000
2	2	VAX (also has 3 operands format)
3	3	VAX (also has 2 operands format)

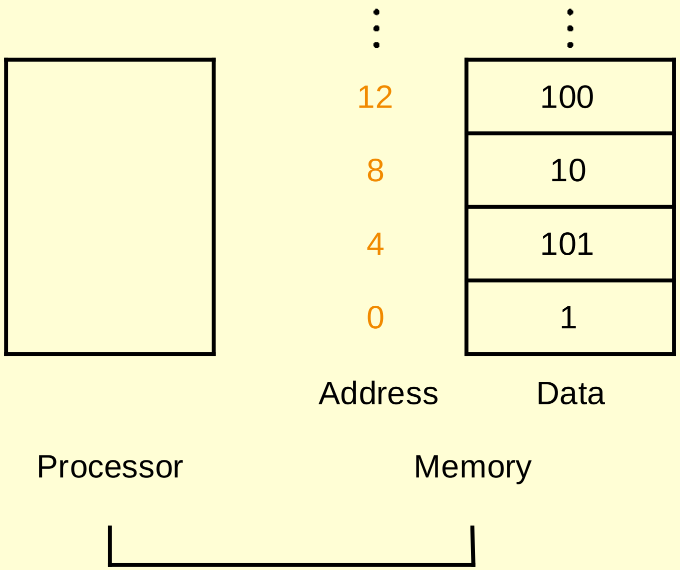
Effect of the number of memory operands:

Type	Advantages	Disadvantages
Reg-Reg (0,3)	- Fixed length instruction encoding - Simple code generation model - Similar execution time (pipeline)	- Higher instruction count - Some instructions are short leading to wasteful bit encoding
Reg-Mem (1,2)	- Direct access without loading - Easy instruction encoding	- Can restrict # register available for use - Clocks per instr. varies by operand type - Source operands are destroyed
Mem-Mem (3,3)	- No temporary register usage - Compact code	- Less potential for compiler optimization - Can create memory access bottleneck

Memory Alignment

- The address of a word matches the byte address of one of its 4 bytes
- The addresses of sequential words differ by 4 (word size in byte)
- Words' addresses are multiple of 4 (alignment restriction)
  - Misalignment (if allowed) complicates memory access and causes programs to run slower

Object addressed	Aligned at byte offsets	Misaligned at byte offsets
Byte	1,2,3,4,5,6,7	Never
Half word	0,2,4,6	1,3,5,7
Word	0,4	1,2,3,5,6,7
Double word	0	1,2,3,4,5,6,7



Byte Order

- Given N bytes, which is the most significant, which is the least significant?
  - “Little Endian”
    - Leftmost / least significant byte = word address
    - Intel (among others)
  - “Big Endian”
    - Leftmost / most significant byte = word address
    - Motorola, TCP/IP (among others)
- Byte ordering can be as problem when exchanging data among different machines
- Can also affect array index calculation or any other operation that treat the same data a both byte and word.

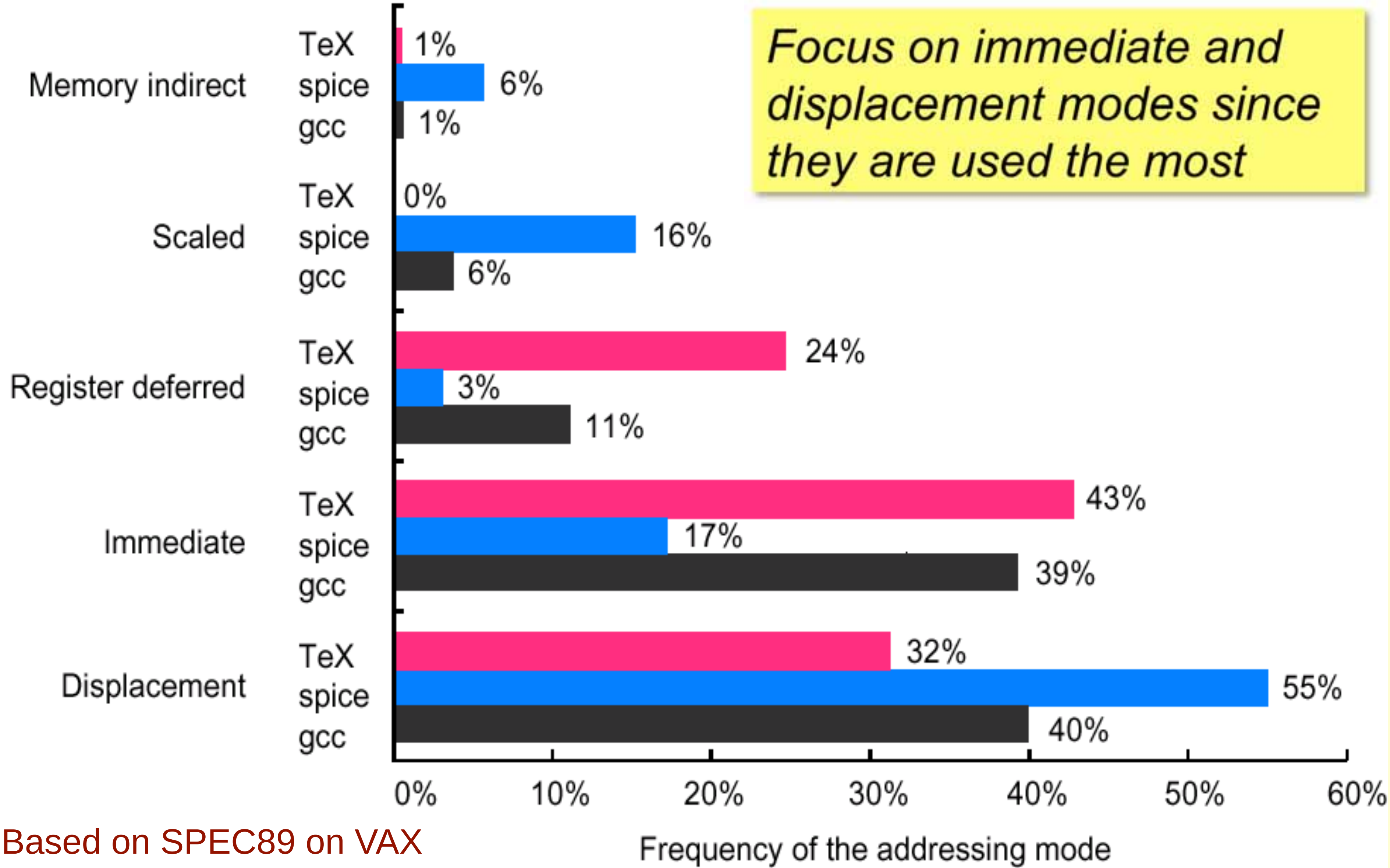
Addressing Modes

- How to specify the location of an operand (effective address)
- Addressing modes have the ability to:
  - Significantly reduce instruction counts
  - Increase the average CPI
  - Increase the complexity of building a machine
- VAX machine is used for benchmark data since it supports wide range of memory addressing modes
- Can classify based on:
  - source of the data (register, immediate or memory)
  - the address calculation (direct, indirect, indexed)

Example of Addressing Modes

Mode	Example	Meaning	When used
Register	ADD R4, R3	*Regs[R4] = Regs[R4] + Regs[R3]*	When a value is in a register
Immediate	ADD R4, #3	*Regs[R4] = Regs[R4] + 3*	For constants
Register indirect	ADD R4, (R1)	*Regs[R4] = Regs[R4] + Mem[Regs[R1]]*	Accessing using a pointer or a computed address
Direct or absolute	ADD R4, (1001)	*Regs[R4] = Regs[R4] + Mem[1001]*	Sometimes useful for accessing static data; address constant may need to be large
Displacement	ADD R4, 100 (R1)	*Regs[R4] = Regs[R4] + Mem[100 + Regs[R1]]*	Accessing local variables
Indexed	ADD R4, (R1 + R2)	*Regs[R4] = Regs[R4] + Mem[Regs[R1] + Regs[R2]]*	Sometimes useful in array addressing: R1 = base of the array: R2 = index amount
Autoincrement	ADD R4, (R2) +	*Regs[R4] = Regs[R4] + Mem[Regs[R2]]* *Regs[R2] = Regs[R2] + d*	Useful for stepping through arrays within a loop. R2 points to start of the array; each reference increments R2 by d.
Auto decrement	ADD R4, -(R2)	*Regs[R2] = Regs[R2] – d* *Regs[R4] = Regs[R4] + Mem[Regs[R2]]*	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack
Scaled	ADD R4, 100 (R2) [R3]	*Regs[R4] = Regs[R4] + Mem[100 + Regs[R2] + Regs[R3] * d]*	Used to index arrays.

Addressing Mode Use

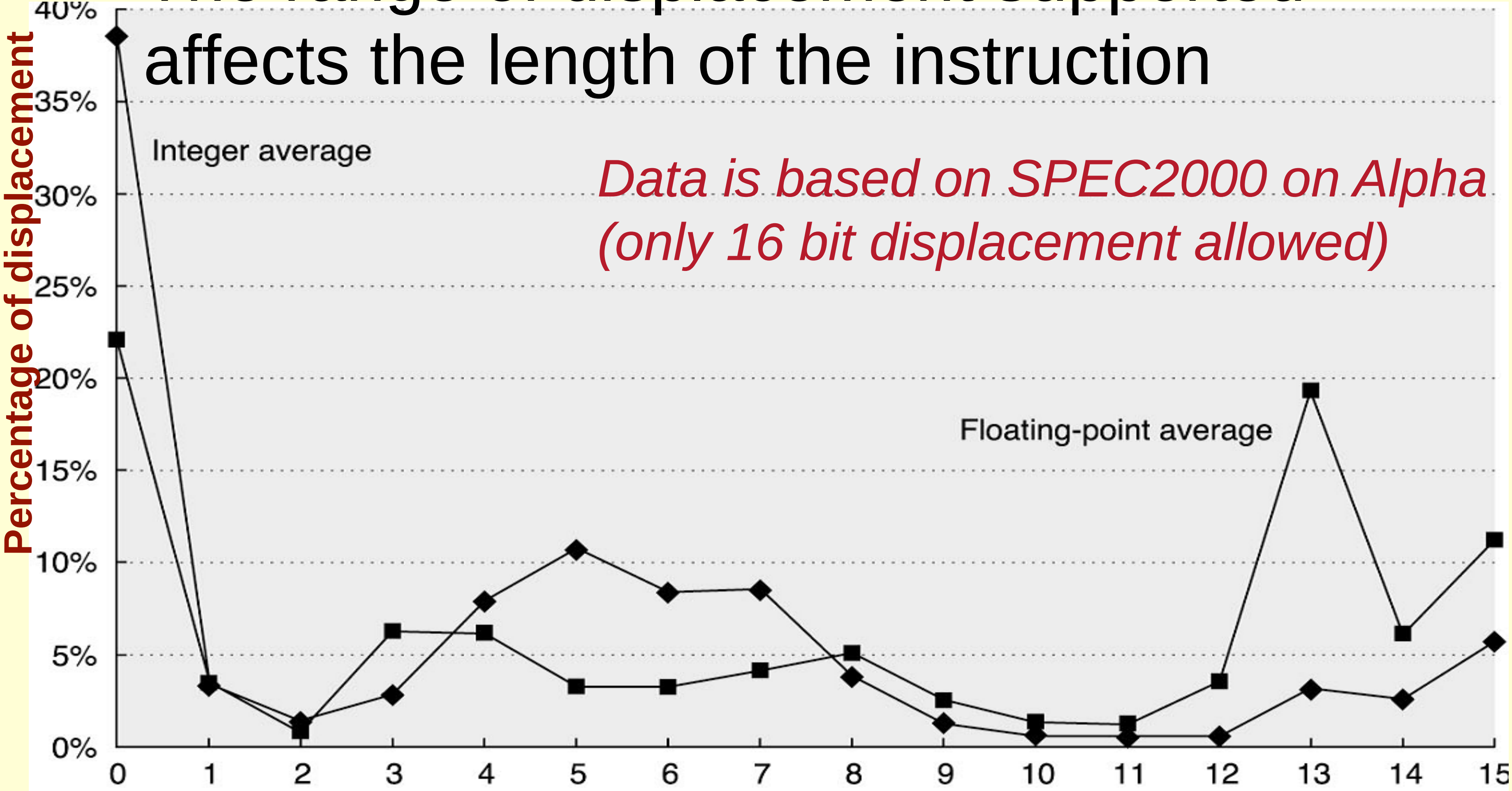


Focus on immediate and displacement modes since they are used the most

Based on SPEC89 on VAX

Displacement Addressing Modes

- The range of displacement supported affects the length of the instruction



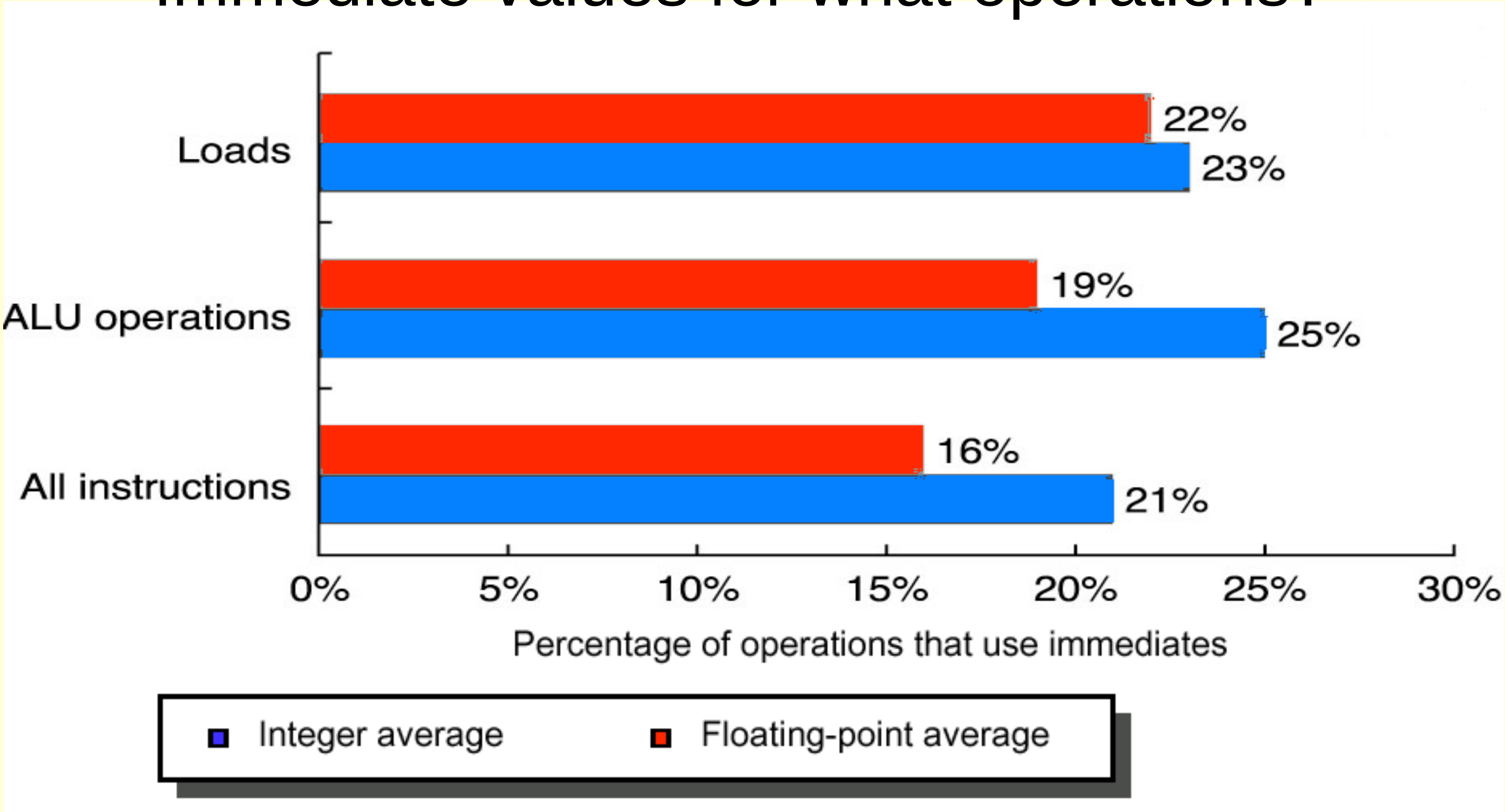
Data is based on SPEC2000 on Alpha (only 16 bit displacement allowed)

Percentage of displacement

Number of bits needed for a displacement value in SPEC2000 benchmark

Immediate Addressing Modes

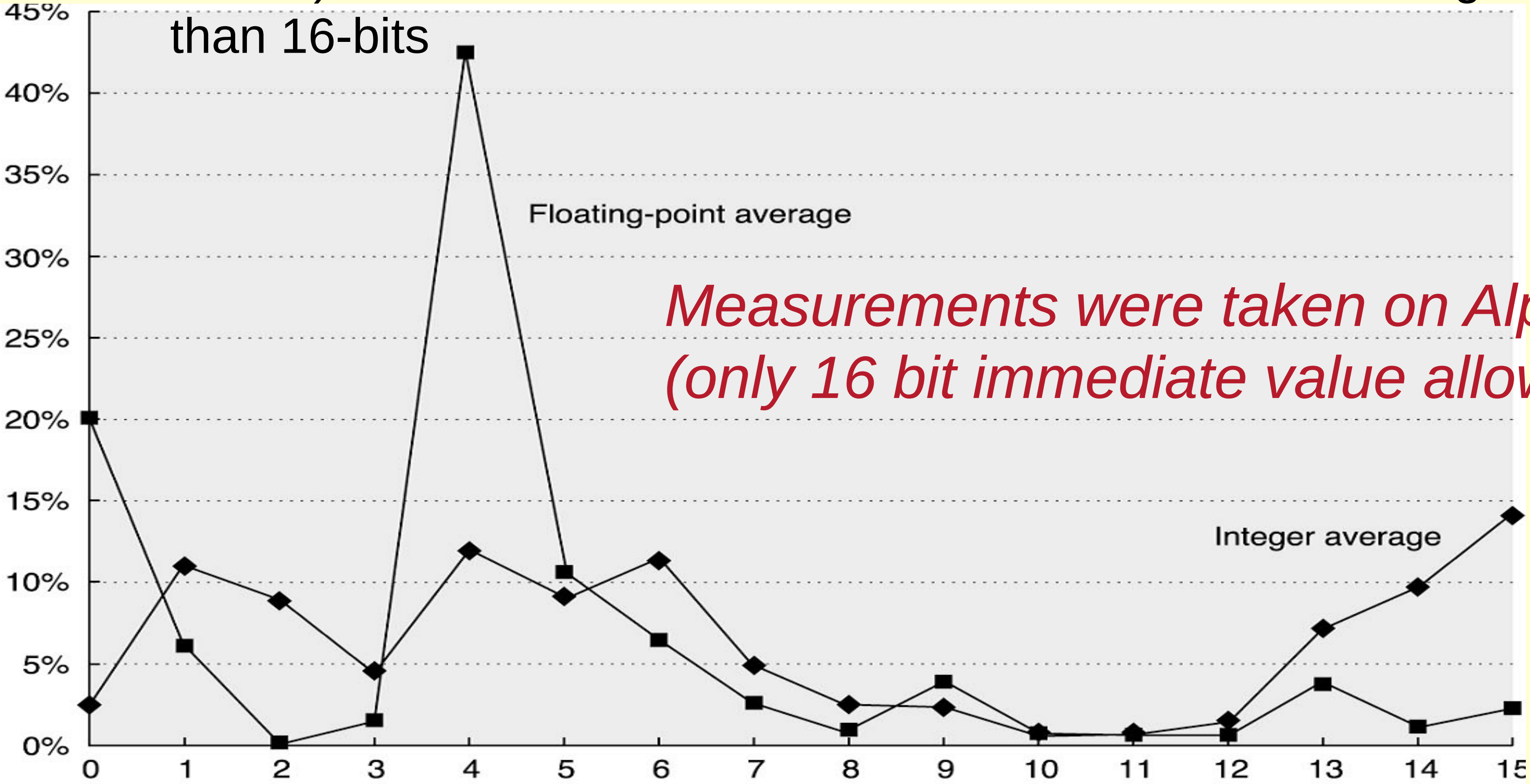
- Immediate values for what operations?



Statistics are based on SPEC2000 benchmark on Alpha

Distribution of Immediate Values

- Range affects instruction length
  - Similar measurements on the VAX (with 32-bit immediate values) showed that 20-25% of immediate values were longer than 16-bits



Measurements were taken on Alpha (only 16 bit immediate value allowed)

Percentage of Immediate Values

Number of bits needed for a immediate values in SPEC2000 benchmark

Addressing Mode for Signal Processing

- DSP offers special addressing modes to better serve popular algorithms
- Special features requires either hand coding or a compiler that uses such features

Addressing Mode for Signal Processing

- Modulo addressing:
  - Since DSP deals with continuous data streams, circular buffers common
  - Circular or modulo addressing: automatic increment and decrement / reset pointer at end of buffer
- Reverse addressing:
  - Address is the reverse order of the current address
  - Expedites access / otherwise require a number of logical instructions or extra memory accesses

Fast Fourier Transform

$0\ (000_2) \rightarrow 0\ (000_2)$

$1\ (001_2) \rightarrow 4\ (100_2)$

$2\ (010_2) \rightarrow 2\ (010_2)$

$3\ (011_2) \rightarrow 6\ (110_2)$

$4\ (100_2) \rightarrow 1\ (001_2)$

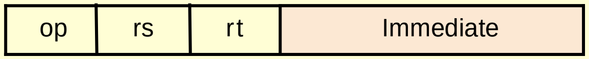
$5\ (101_2) \rightarrow 5\ (101_2)$

$6\ (110_2) \rightarrow 3\ (011_2)$

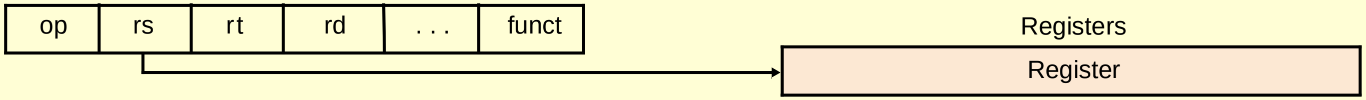
$7\ (111_2) \rightarrow 7\ (111_2)$

Summary of MIPS Addressing Modes

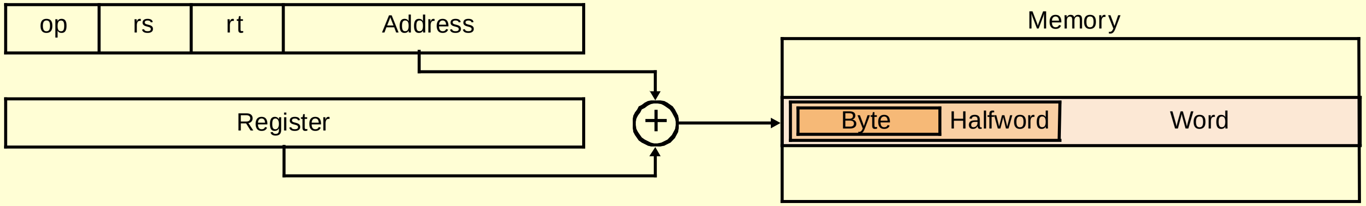
1. Immediate addressing



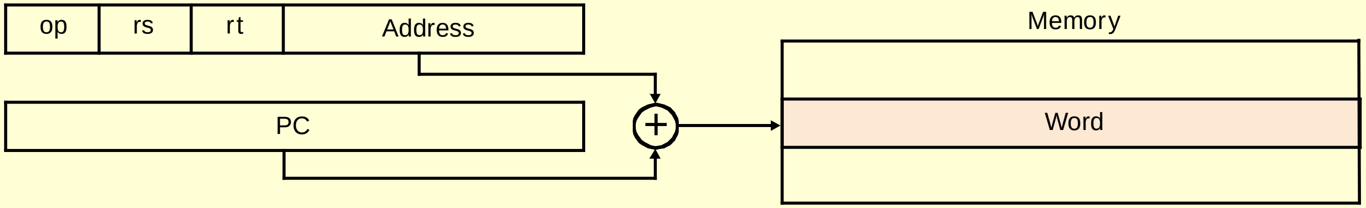
2. Register addressing



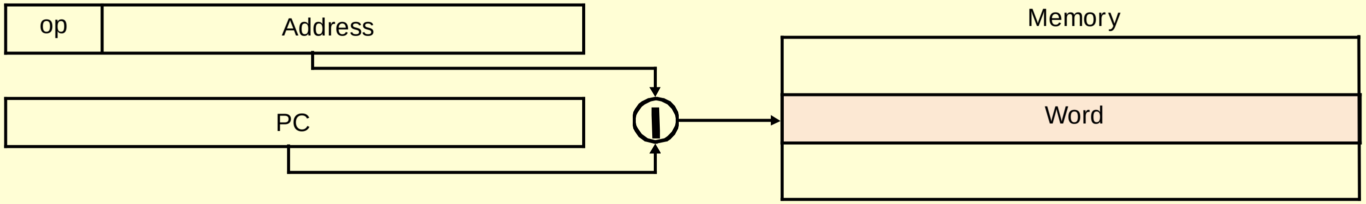
3. Base addressing



4. PC-relative addressing



5. Pseudodirect addressing



Operations of the Computer Hardware

“There must certainly be instructions for performing the fundamental arithmetic operations.”

Burkes, Goldstine and Von Neumann, 1947

MIPS assembler allows only one instruction/line and ignore comments following # until end of line

Example:

Translation of a segment of a C program to MIPS assembly instructions:

C: f = (g + h) - (i + j)

(pseudo)MIPS:

```
add     t0, g, h      # temp. variable t0 contains "g + h"
add     t1, i, j      # temp. variable t1 contains "i + j"
sub     f, t0, t1     # f = t0 - t1 = (g + h) - (i + j)
```

Operations in the Instruction Set

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, and, subtract , or
Data Transfer	Loads-stores (move instructions on machines with memory addressing)
Control	Branch, jump, procedure call and return, trap
System	Operating system call, Virtual memory management instructions
Floating point	Floating point instructions: add, multiply
Decimal	Decimal add, decimal multiply, decimal to character conversion
String	String move, string compare, string search
Graphics	Pixel operations, compression/decompression operations

- Arithmetic, logical, data transfer and control are almost standard categories for all machines
- System instructions are required for multi-programming environment although support for system functions varies
- Others can be primitives (e.g. decimal and string on IBM 360 and VAX), provided by a co-processor, or synthesized by compiler.

Operations for Media & Signal Process.

- Partitioned Add:
  - Partition a single register into multiple data elements (e.g. 4 16-bit words in 1 64-bit register)
  - Perform the same operation independently on each
  - Increases ALU throughput for multimedia applications
- Paired single operations
  - Perform multiple independent narrow operations on one wide ALU (e.g. 2 32-bit float ops)
  - Handy in dealing with vertices and coordinates
- Multiply and accumulate
  - Very handy for calculating dot products of vectors (signal processing) and matrix multiplication

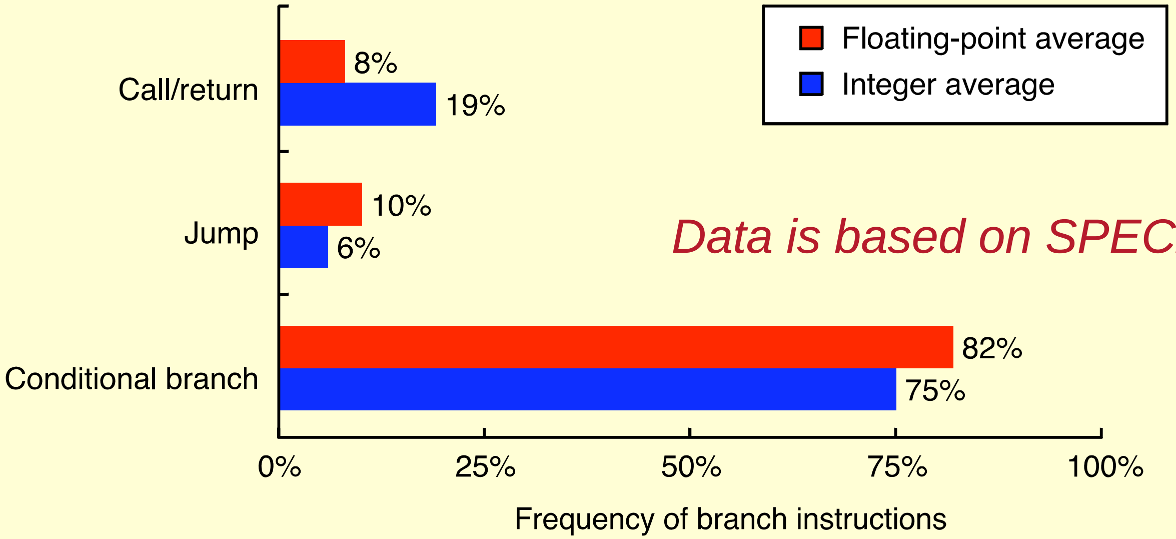
Frequency of Operations Usage

- The most widely executed instructions are the simple operations of an instruction set
- Average usage in SPECint92 on Intel 80x86:

Rank	80x86 Instruction	Integer Average (% total executed)
1	Load	22%
2	Conditional branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move register-register	4%
9	Call	1%
10	Return	1%
Total		96%

Make the common case fast by focusing on these operations

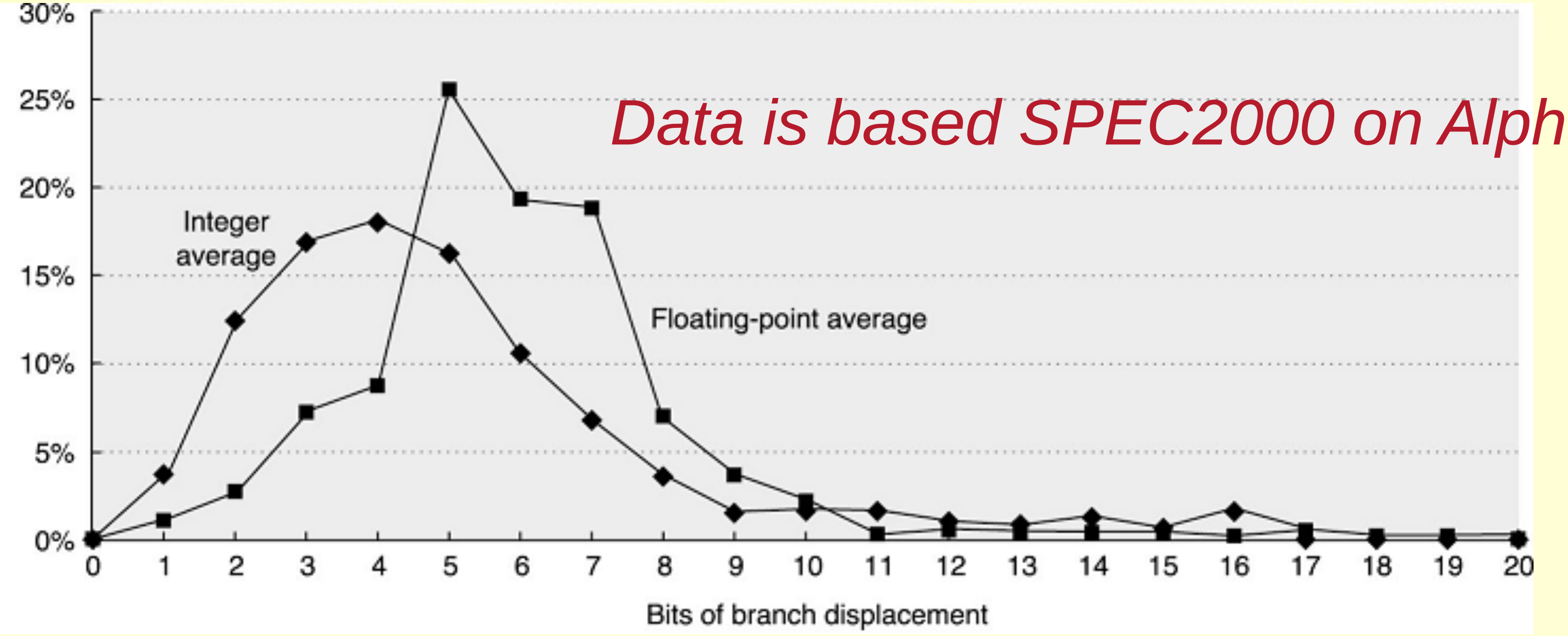
Control Flow Instructions



Data is based on SPEC2000 on Alpha

- Jump: unconditional change in the control flow
- Branch: conditional change in the control flow
- Procedure calls and returns

Destination Address Definition



Data is based SPEC2000 on Alpha

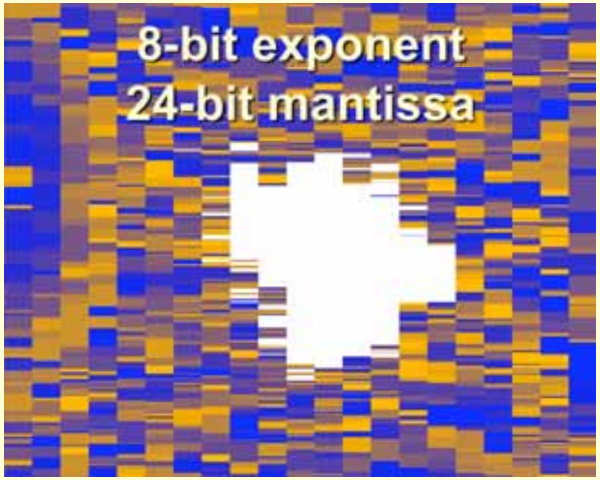
- PC-relative addressing
  - Good for short position-independent forward & backward jumps
- Register indirect addressing
  - Good for dynamic libraries, virtual functions & packed case statements

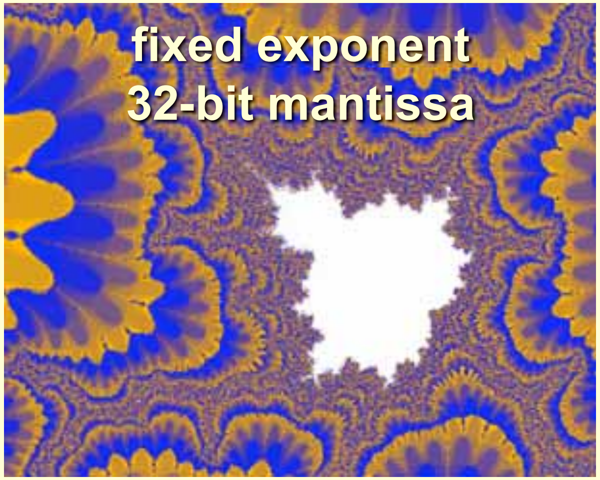
Type and Size of Operands

- Operand type encoded in instruction opcode
  - The type of an operand effectively gives its size
- Common types include character, half word and word size integer, single- and double-precision floating point
  - Characters are almost always in ASCII, though 16-bit Unicode (for international characters) is gaining popularity
  - Integers in 2’s complement
  - Floating point in IEEE 754

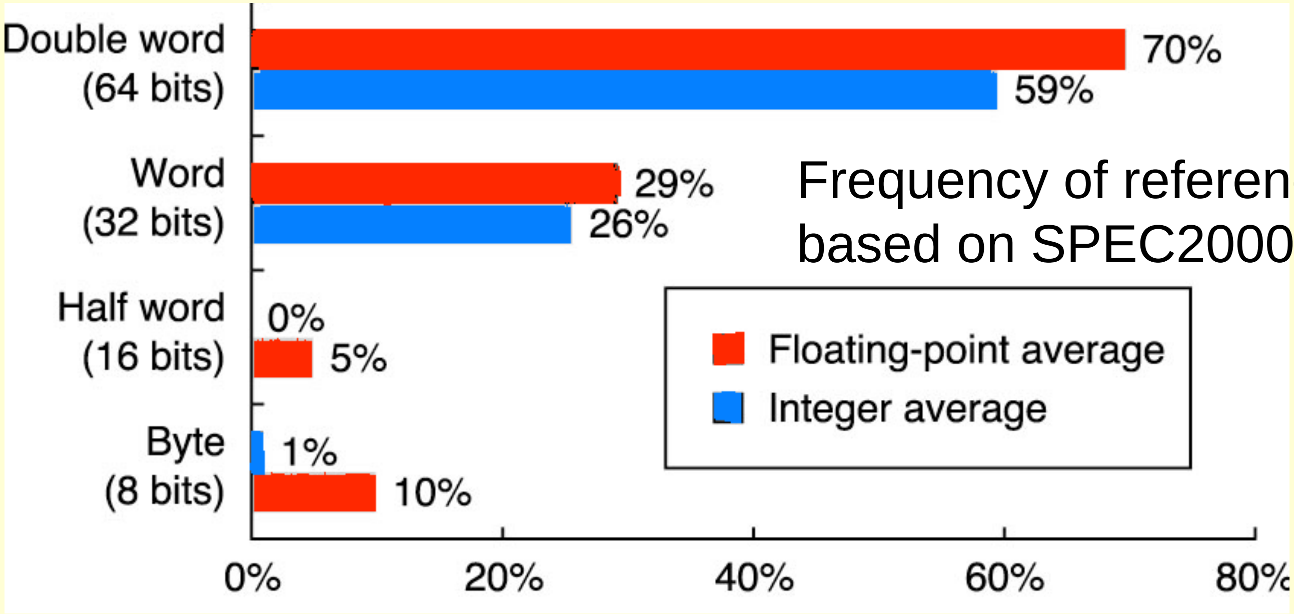
Unusual Types

- Business Applications
  - Binary Coded Decimal (BCD)
    - Exactly represents all decimal fractions (binary doesn’t!)
- DSP
  - Fixed point
    - Good for limited range numbers: more mantissa bits
  - Block floating point
    - Single shared exponent for multiple numbers
- Graphics
  - 4-element vector operations (RGBA or XYZW)
    - 8-bit, 16-bit or single-precision floating point





Size of Operands



Frequency of reference by size based on SPEC2000 on Alpha

- Double-word: double-precision floating point + addresses in 64-bit machines
- Words: most integer operations + addresses in 32-bit machines
- *For the mix in SPEC*, word and double-word data types dominates

Instruction Representation

- All data in computer systems is represented in binary
- Instructions are no exception
- The program that translates the human-readable code to numeric form is called an *Assembler*
- Hence *machine-language* or *assembly-language*

Example:

Assembly: `ADD $t0, $s1, $s2`

Note: by default MIPS $t0..$t7 map to reg. 8..15, $s0..$s7 map to reg. 16-23

M/C language (hex by field): `0x0 0x11 0x12 0x8 0x020`

M/C language (binary):

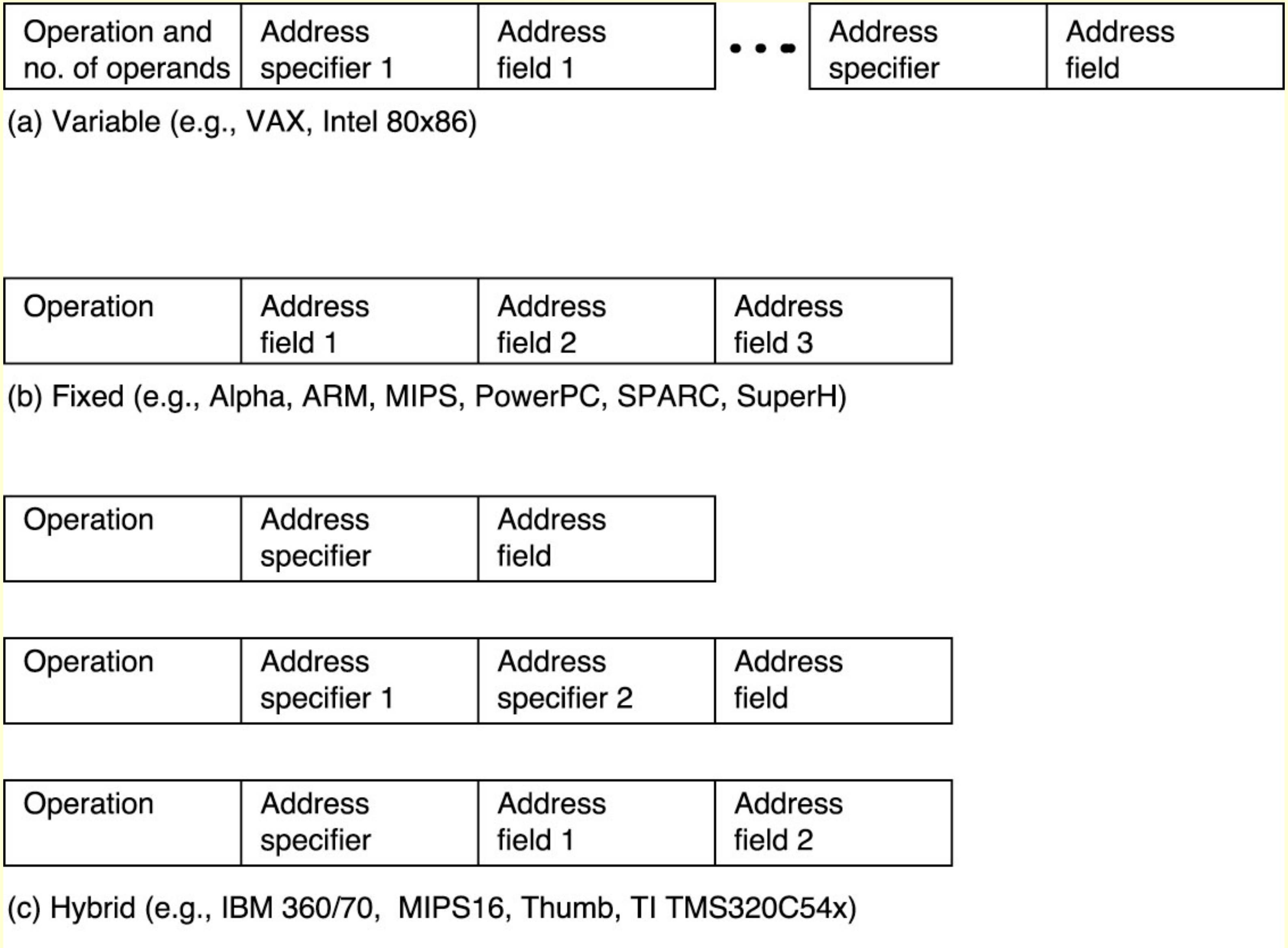
0000	0010	0011	0010	0100	0000	0010	0000

M/C language (hex): `0x02324020`

Encoding an Instruction Set

- Affects the size of the compiled program
- Also complexity of the CPU implementation
- Operation in one field called opcode
- Addressing mode in opcode or separate field
- Must balance:
  - Desire to support as many registers and addressing modes as possible
  - Effect of operand specification on the size of the instruction (and program)
  - Desire to simplify instruction fetching and decoding during execution
- Fixed size instruction encoding simplifies CPU design but limits addressing choices

Encoding Examples



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)

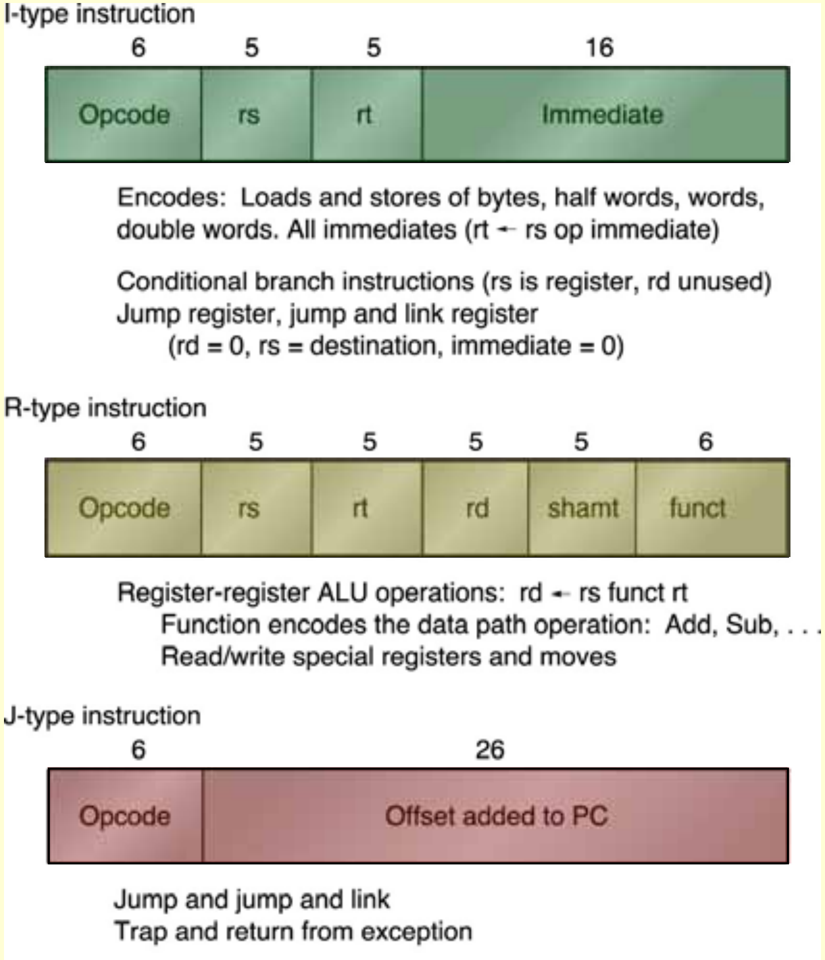
Operation	Address specifier	Address field

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)

MIPS Instruction Formats



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)

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

Jump and jump and link
Trap and return from exception

opcodes

	000	001	010	011	100	101	110	111
000	R-type		j	jal	beq	bne	blez	bgtz
001	addi	addiu	slti	sltiu	andi	ori	xori	
010								
011	llo	lhi	trap					
100	lb	lh		lw	lbu	lhu		
101	sb	sh		sw				
110								
111								

funct codes

	000	001	010	011	100	101	110	111
000	sll		srl	sra	sllv		srlv	srav
001	jr	jalr						
010	mfhi	mthi	mflo	mtlo				
011	mult	multu	div	divu				
100	add	addu	sub	subu	and	or	xor	nor
101			slt	sltu				
110								
111								

GPU Shading ISA

- Data
  - IEEE-like floating point
  - 4-element vectors
    - Most instructions perform operation on all four
- Addressing
  - No addresses
  - ATTRIB, PARAM, TEMP, OUTPUT
  - Limited arrays
  - Element selection (read & write)
    - C.xyw, C.rgba

GPU Shading ISA

- Instructions:

Instruction	Operation	Instruction	Operation
ABS r,s	r = abs(s)	MIN r,s1,s2	r = min(s1,s2)
ADD r,s1,s2	r = s1+s2	MOV r,s1	r = s1
CMP r,c,s1,s2	r = c<0 ? s1 : s2	MUL r,s1,s2	r = s1*s2
COS r,s	r = cos(s)	POW r,s1,s2	$r \approx s1^{s2}$
DP3 r,s1,s2	r = s1.xyz • s2.xyz	RCP r,s1	r = 1/s1
DP4 r,s1,s2	r = s1 • s2	RSQ r,s1	r = 1/sqrt(s1)
DPH r,s1,s2	r = s1.xyz1 • s2	SCS r,s1	r = (cos(s),sin(s),?,?)
DST r,s1,s2	r = (1,s1.y*s2.y,s1.z,s2.w)	SGE r,s1,s2	r = s1≥s2 ? 1 : 0
EX2 r,s	$r \approx 2^{s}$	SIN r,s	r = sin(s)
FLR r,s	r = floor(s)	SLT r,s1,s2	r = s1<s2 ? 1 : 0
FRC r,s	r = s - floor(s)	SUB r,s1,s2	r = s1-s2
KIL s	if (s<0) discard	SWZ r,s,cx,cy,cz,cw	r = swizzle(s)
LG2 r,s	$r \approx \log_2(s)$	TEX r,s,name,nD	r = texture(s)
LIT r,s	r = lighting computation	TXB r,s,name,nD	r = textureLOD(s)
LRP r,t,s1,s2	r = t*s1 + (1-t)*s2	TXP r,s,name,nD	r = texture(s/s.w)
MAD r,s1,s2,s3	r = s1*s2 + s3	XPD r,s1,s2	r = s1×s2
MAX r,s1,s2	r = max(s1,s2)		

GPU Shading ISA

- Notable:
  - Many special-purpose instructions
  - No binary encoding, interface is text form
    - No ISA limits on future expansion
    - No ISA limits on registers
    - No ISA limits on immediate values
  - Originally no branching! (exists now)