Graduate Computer Architecture

Chapter 2 –
Instruction Set Principles

Outline

- Classifying instruction set architectures
- Memory addressing
- Addressing modes
- Type and size of operands
- Instructions for control
- The role of compiler

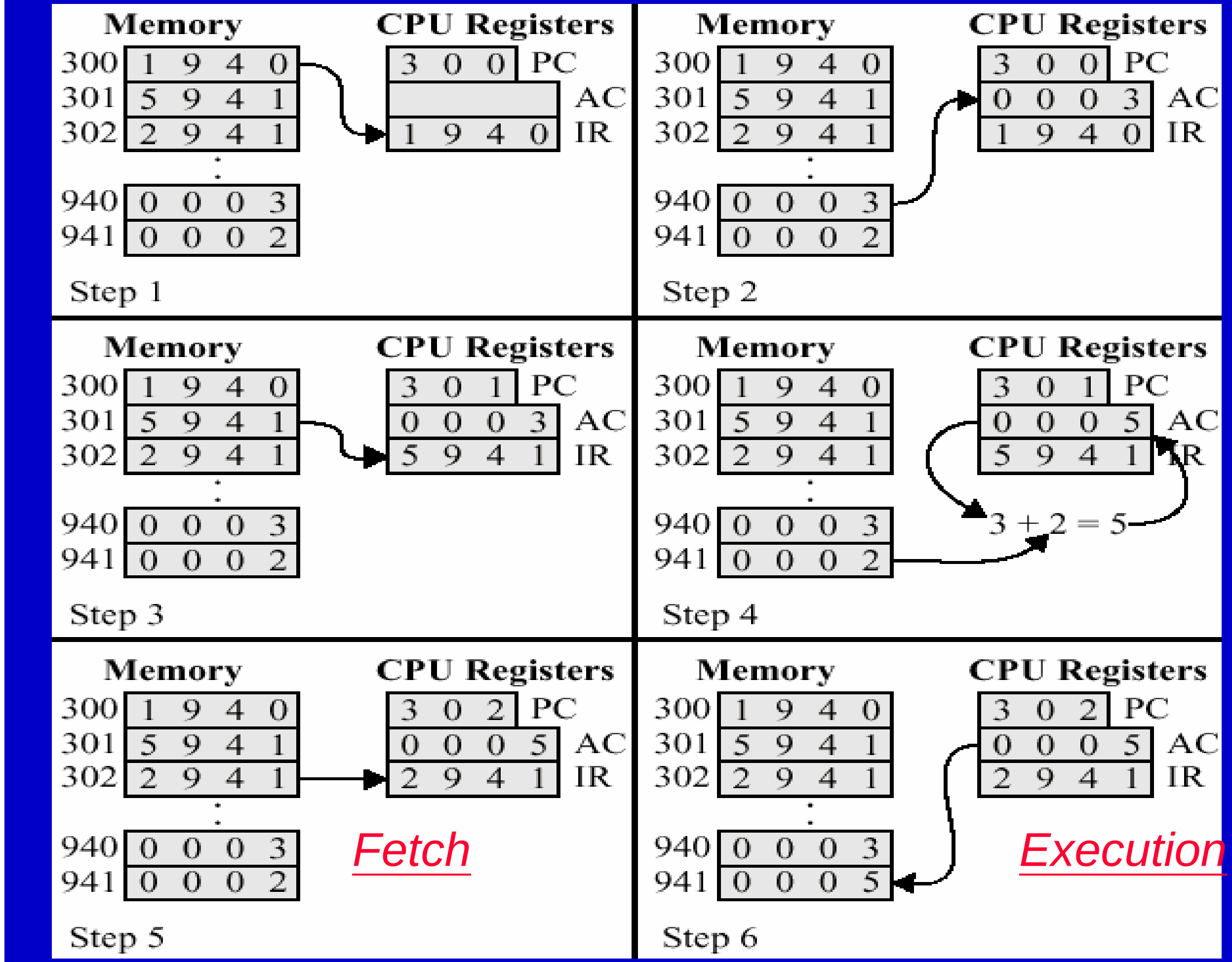
Brief Introduction to ISA

- Instruction Set Architecture: a set of instructions
  - Each instruction is directly executed by the CPU's hardware
- How is it represented?
  - By a binary format since the hardware understands only bits
    - Concatenate together binary encoding for instructions, registers, constants, memories

Brief Introduction to ISA (cont.)

- Options - fixed or variable length formats
  - Fixed - each instruction encoded in same size field (typically 1 word)
  - Variable – half-word, whole-word, multiple word instructions are possible
- Typical physical blobs are bits, bytes, words, n-words
- Word size is typically 16, 32, 64 bits today

An Example of Program Execution



Step 1

Memory	
300	1 9 4 0
301	5 9 4 1
302	2 9 4 1
:	
940	0 0 0 3
941	0 0 0 2

CPU Registers	
3 0 0	PC
	AC
1 9 4 0	IR

Step 2

Memory	
300	1 9 4 0
301	5 9 4 1
302	2 9 4 1
:	
940	0 0 0 3
941	0 0 0 2

CPU Registers	
3 0 0	PC
0 0 0 3	AC
1 9 4 0	IR

Step 3

Memory	
300	1 9 4 0
301	5 9 4 1
302	2 9 4 1
:	
940	0 0 0 3
941	0 0 0 2

CPU Registers	
3 0 1	PC
0 0 0 3	AC
5 9 4 1	IR

Step 4

Memory	
300	1 9 4 0
301	5 9 4 1
302	2 9 4 1
:	
940	0 0 0 3
941	0 0 0 2

CPU Registers	
3 0 1	PC
0 0 0 5	AC
5 9 4 1	IR

3 + 2 = 5

Step 5

Memory	
300	1 9 4 0
301	5 9 4 1
302	2 9 4 1
:	
940	0 0 0 3
941	0 0 0 2

CPU Registers	
3 0 2	PC
0 0 0 5	AC
2 9 4 1	IR

Fetch

Step 6

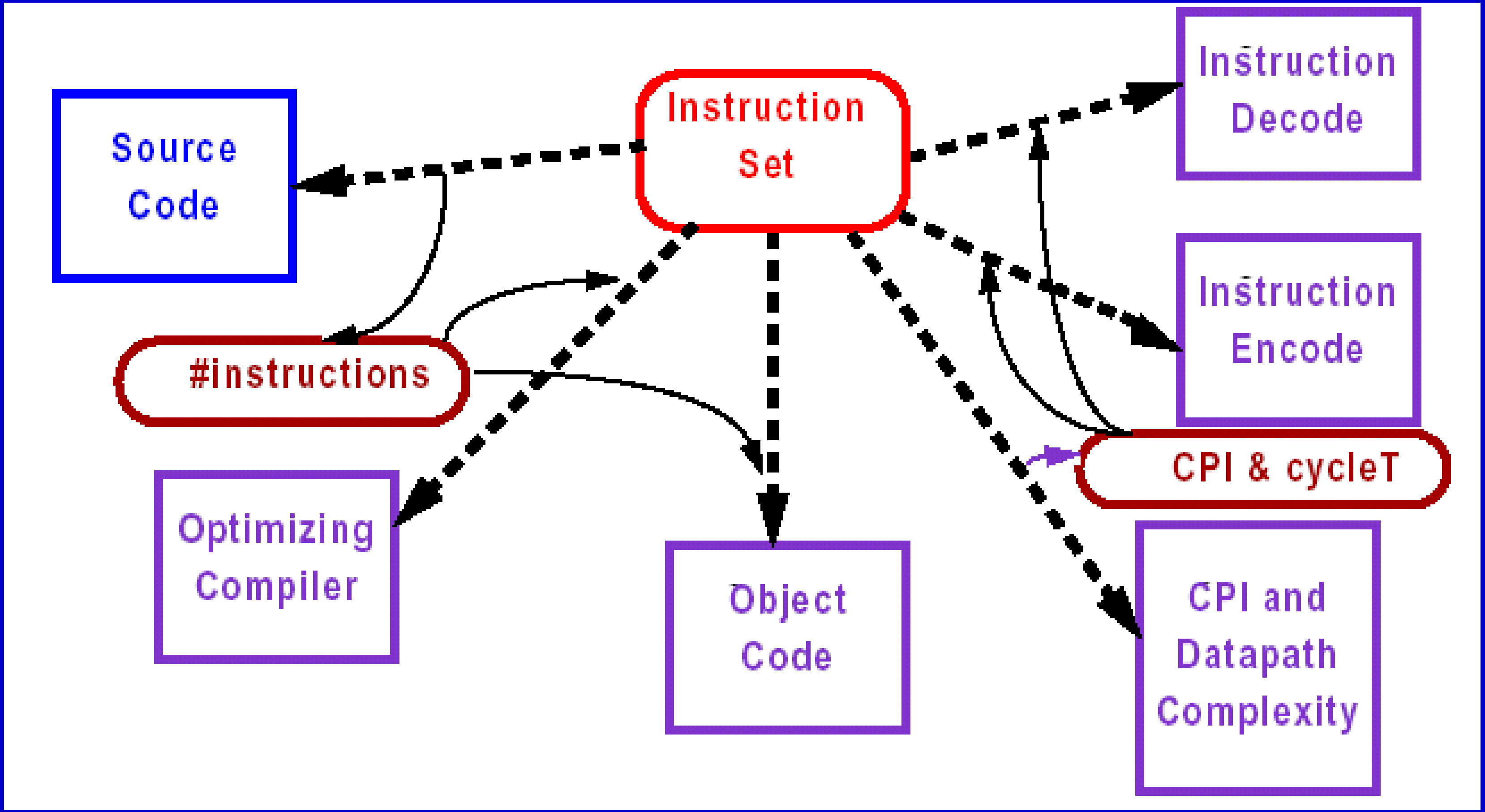
Memory	
300	1 9 4 0
301	5 9 4 1
302	2 9 4 1
:	
940	0 0 0 3
941	0 0 0 5

CPU Registers	
3 0 2	PC
0 0 0 5	AC
2 9 4 1	IR

Execution

- Command
  - Load AC from Memory
  - Add to AC from memory
  - Store AC to memory
- Add the contents of memory 940 to the content of memory 941 and stores the result at 941

Instruction Set Design



$$CPU\_Time = IC * CPI * Cycle\_time$$

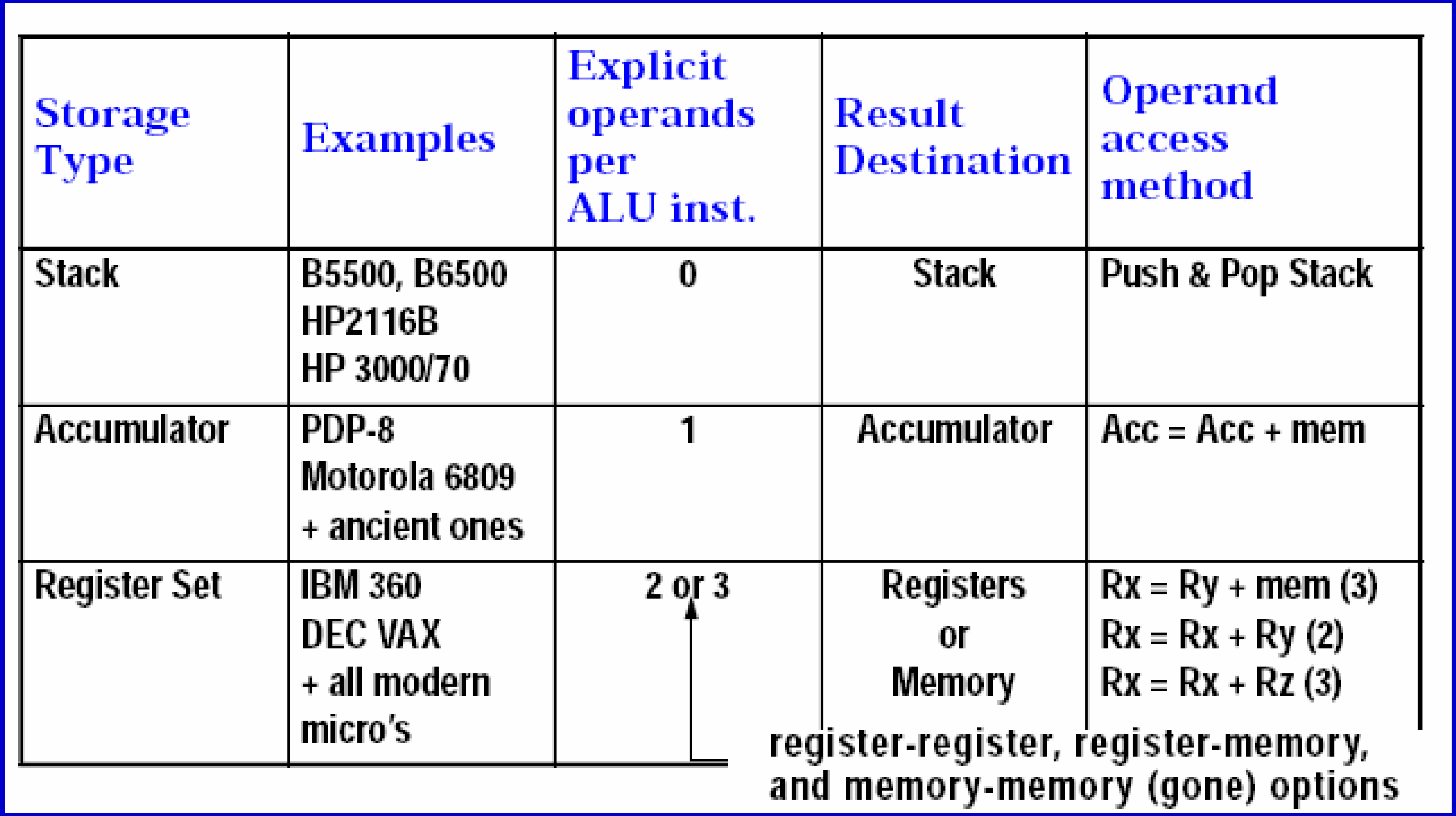
The instruction set influences everything

Classifying Instruction Set Architectures

Operand Storage in CPU	**Where are they other than memory**
Number of explicit operands named per instruction	**How many? Min, Max - maybe even average**
Addressing Modes	**How is the effective address for an operand calculated? Can all operands use any mode?**
Operations	**What are the options for the opcode?**
Type and size of operands	**How is typing done? How is the size specified**

These choices critically affect - #instructions, CPI, and cycle time

Basic CPU Storage Options

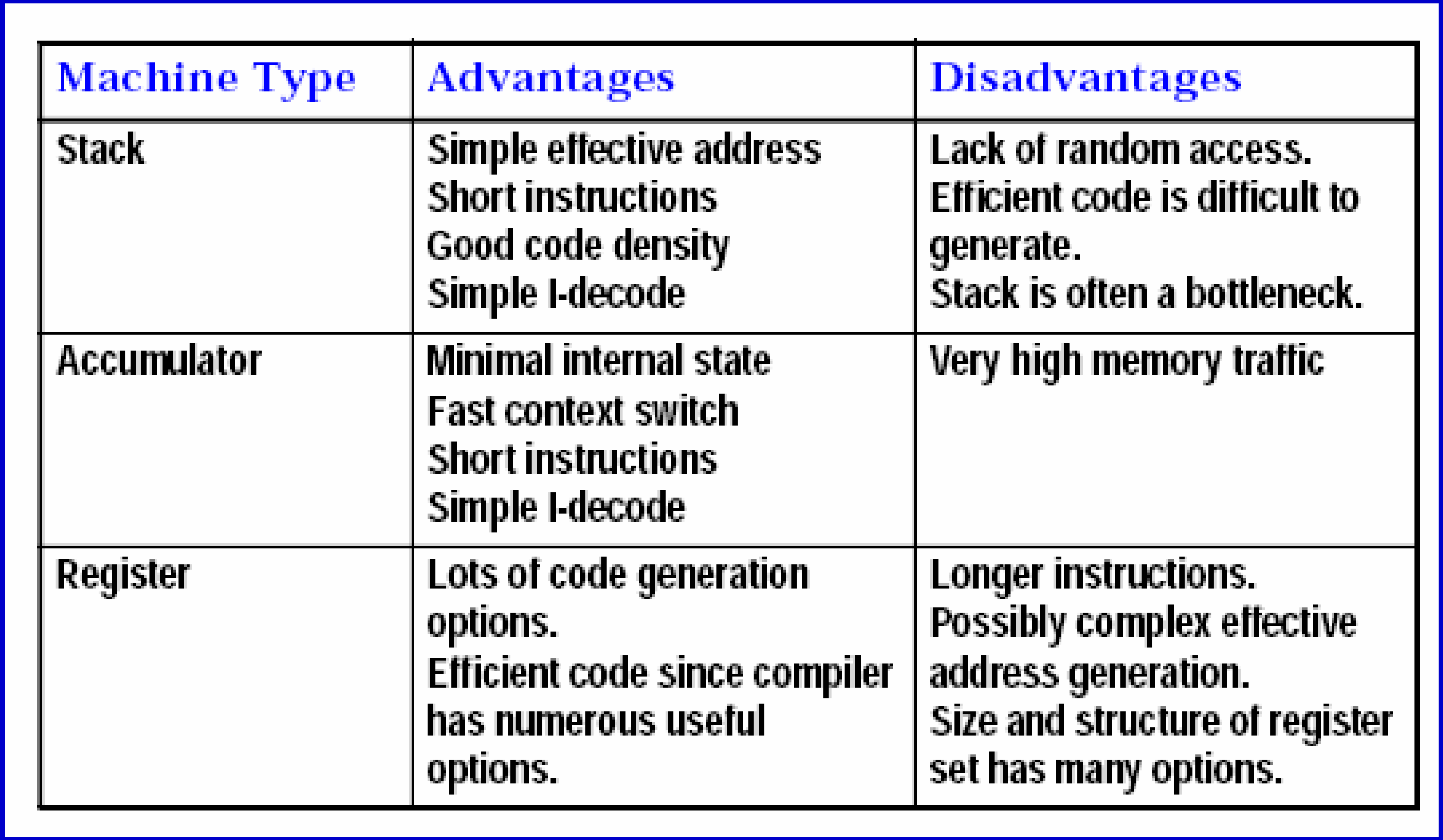


Storage Type	Examples	Explicit operands per ALU inst.	Result Destination	Operand access method
Stack	B5500, B6500 HP2116B HP 3000/70	0	Stack	Push & Pop Stack
Accumulator	PDP-8 Motorola 6809 + ancient ones	1	Accumulator	Acc = Acc + mem

Register Set	IBM 360
	DEC VAX
	+ all modern micro's
		2 or 3	Registers
			or
			Memory
				Rx = Ry + mem (3)
				Rx = Rx + Ry (2)
				Rx = Rx + Rz (3)

register-register, register-memory, and memory-memory (gone) options

Comparison



Machine Type	Advantages	Disadvantages
Stack	Simple effective address Short instructions Good code density Simple I-decode	Lack of random access. Efficient code is difficult to generate. Stack is often a bottleneck.
Accumulator	Minimal internal state Fast context switch Short instructions Simple I-decode	Very high memory traffic
Register	Lots of code generation options. Efficient code since compiler has numerous useful options.	Longer instructions. Possibly complex effective address generation. Size and structure of register set has many options.

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]

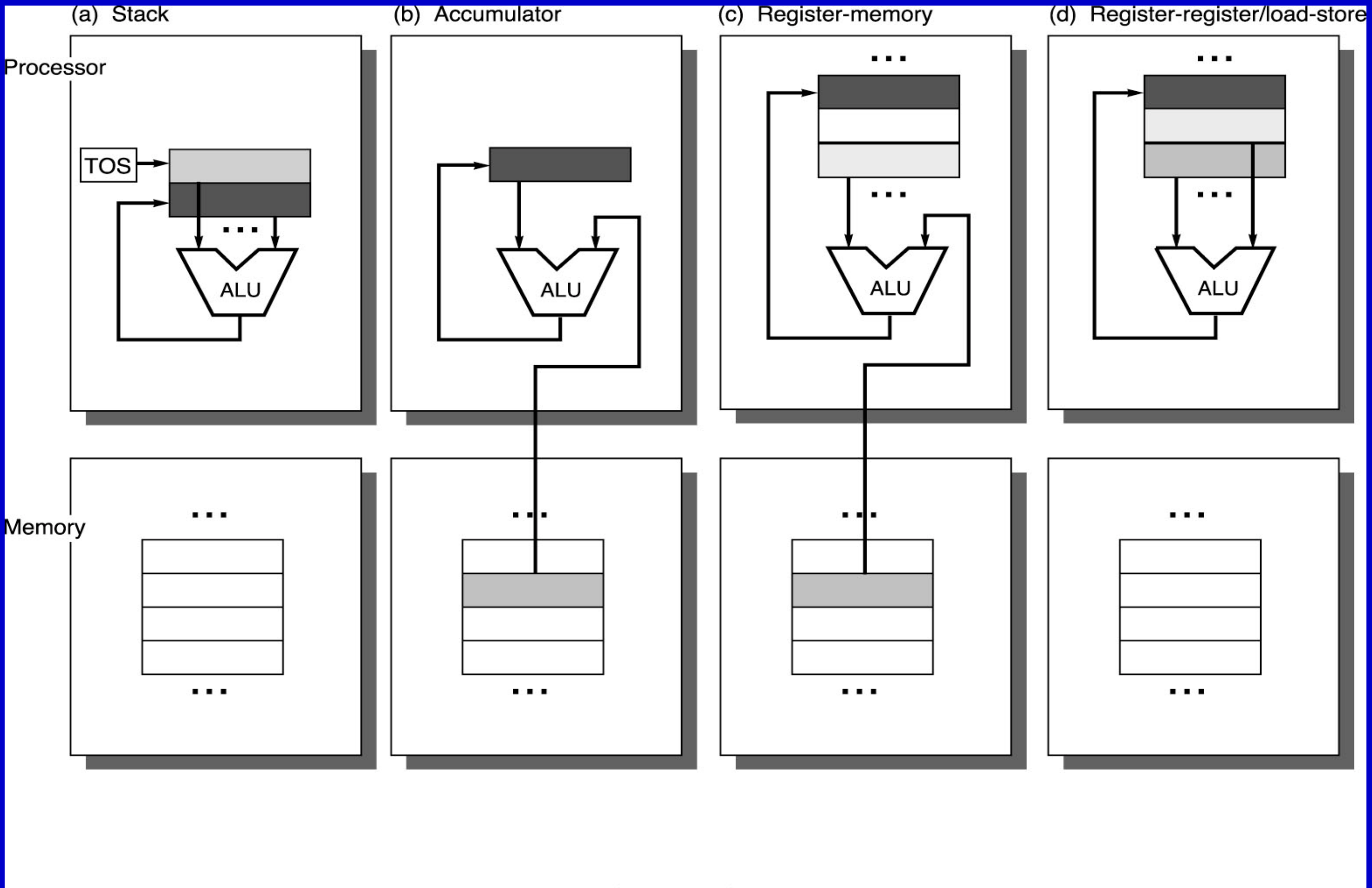
Register-Memory (1970s to present):

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

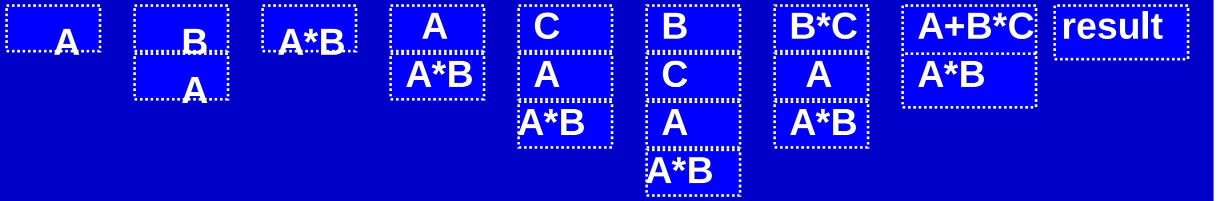
  push C

  push B

  mul

  add

  sub



A

B
A

A*B

A
A*B

C
A
A*B

B
C
A
A*B

B*C
A
A*B

A+B*C
A*B

result

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

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

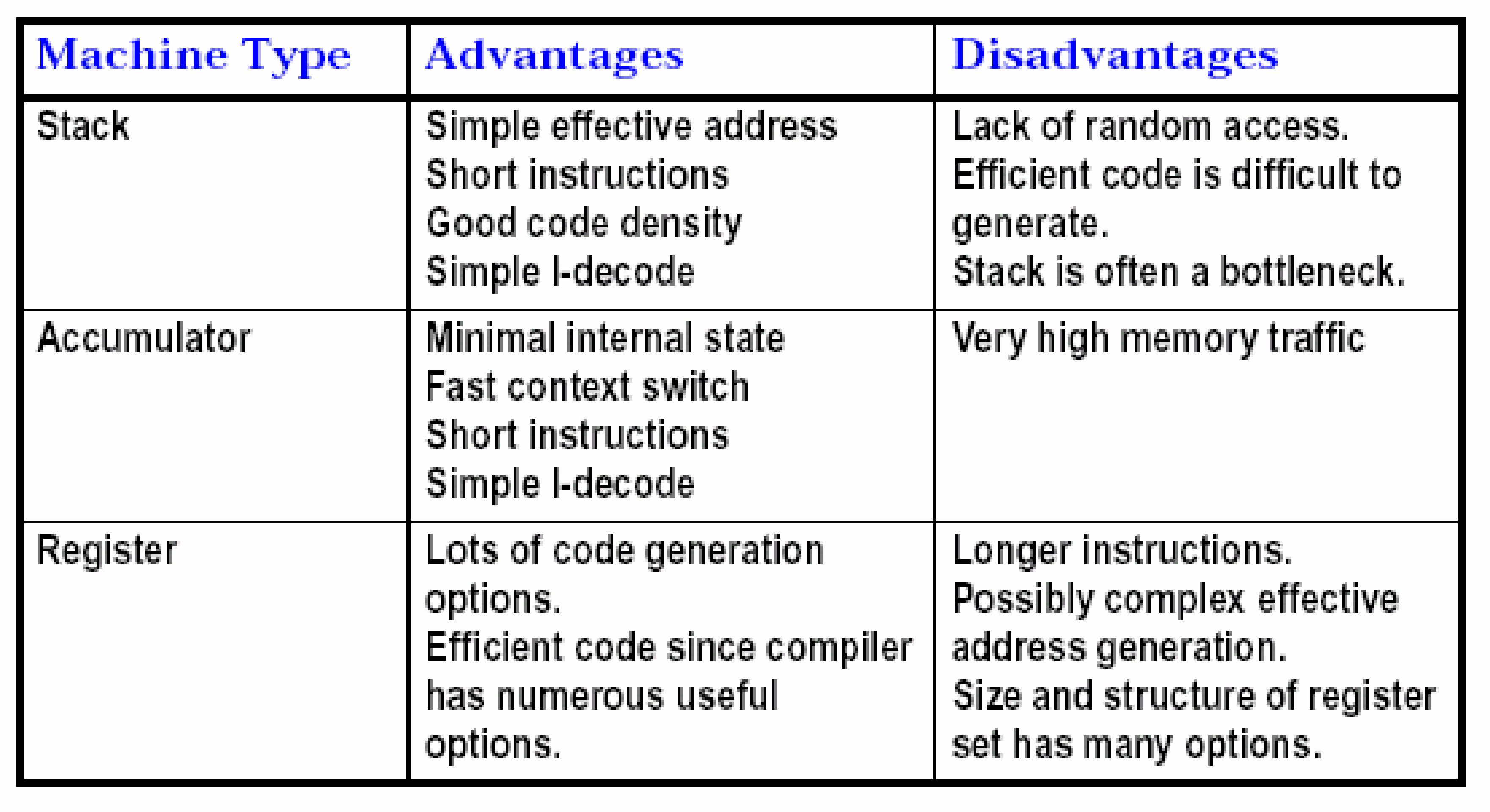
- Pros
  - Simple, fixed length instruction encoding
  - Instructions take similar number of cycles
  - Relatively easy to pipeline
- Cons
  - Higher instruction count
  - Not all instructions need three operands
  - Dependent on good compiler

Registers: Advantages and Disadvantages

- Advantages
  - Faster than cache (no addressing mode or tags)
  - 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

Pro’s and Con’s of Stack, Accumulator, Register Machine



Machine Type	Advantages	Disadvantages
Stack	Simple effective address Short instructions Good code density Simple I-decode	Lack of random access. Efficient code is difficult to generate. Stack is often a bottleneck.
Accumulator	Minimal internal state Fast context switch Short instructions Simple I-decode	Very high memory traffic
Register	Lots of code generation options. Efficient code since compiler has numerous useful options.	Longer instructions. Possibly complex effective address generation. Size and structure of register set has many options.

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

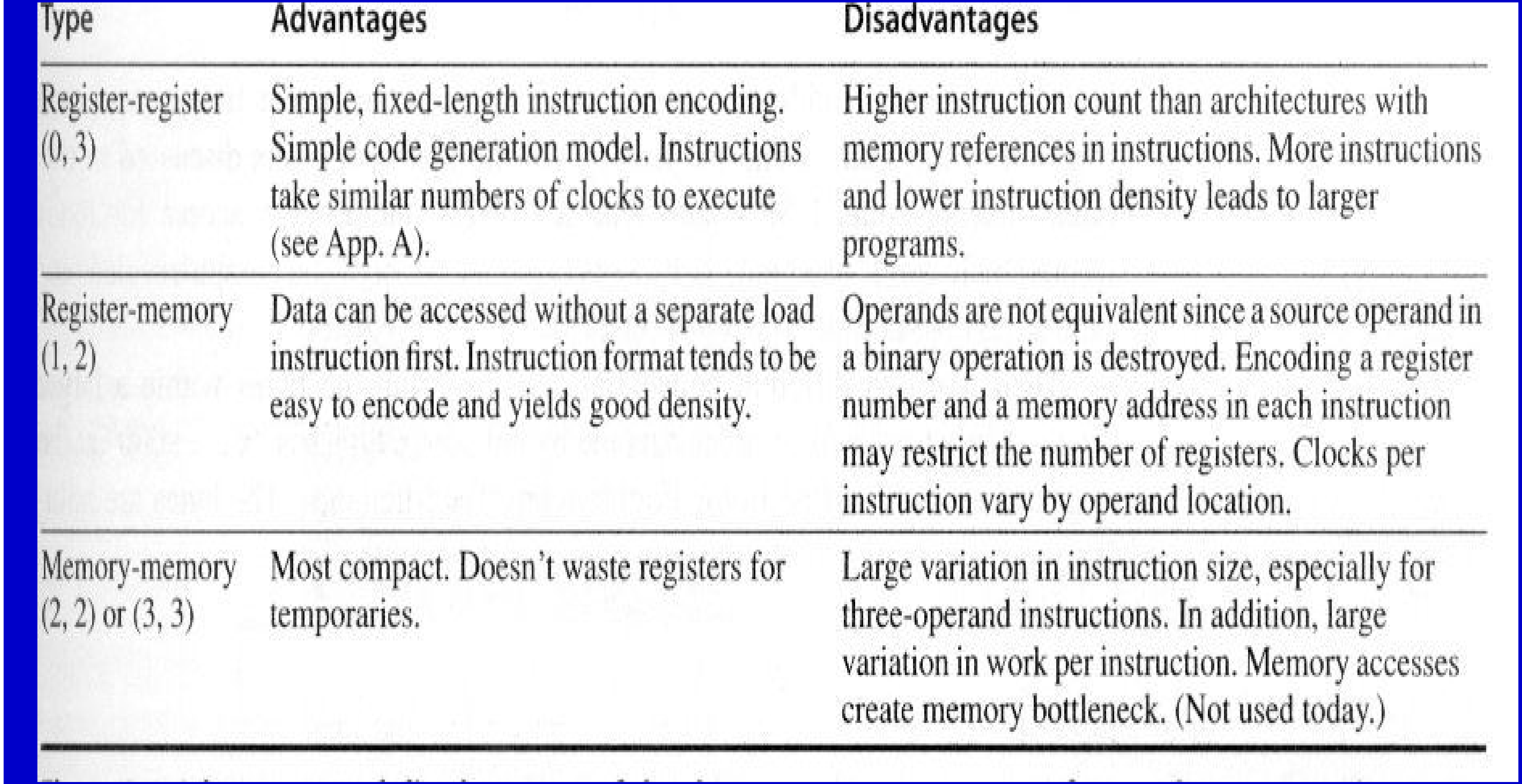
Real Machines Are Not So Simple

- Most real machines have a mixture of 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

Combinations of Number of Memory Addresses and Operands Allowed

Number of memory address	Maximum number of operands allowed	Types of architecture	Examples
0	3	Register-register	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, Trimedia TM5200
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX
3	3	Memory-memory	VAX

Compare Three Common General -Purpose Register Computers



Type	Advantages	Disadvantages
Register-register (0, 3)	Simple, fixed-length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute (see App. A).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1, 2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2, 2) or (3, 3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

where (*m*,*n*) means *m* memory operands and *n* total operands

Outline

- Classifying instruction set architectures
- Memory addressing
- Addressing modes
- Type and size of operands
- Instructions for control
- The role of compiler

Memory Addressing

- How memory addresses are interpreted
  - Endian order
  - Alignment
- How architectures specify the address of an object they will access
  - Addressing modes

Memory Addressing (cont.)

- All instruction sets discussed in this book are byte addressed
- The instruction sets provide access for bytes (8 bits), half words (16 bits), words (32 bits), and even double words (64 bits)
- Two conventions for ordering the bytes within a larger object
  - Little Endian
  - Big Endian

Little Endian

- The low-order byte of an object is stored in memory at the lowest address, and the high-order byte at the highest address. (The little end comes first.)
- For example, a 4-byte object
  - (Byte3 Byte2 Byte1 Byte0)
  - Base Address+0 Byte0
  - Base Address+1 Byte1
  - Base Address+2 Byte2
  - Base Address+3 Byte3
- Intel processors (those used in PC's) use "Little Endian" byte order.

Dr. William T. Verts An Essay on Endian Order, http://www.cs.umass.edu/~verts/cs32/endian.html, April 19, 1996

Big Endian

- The high-order byte of an object is stored in memory at the lowest address, and the low-order byte at the highest address. (The big end comes first.)
- For example, a 4-byte object
  - (Byte3 Byte2 Byte1 Byte0)
  - Base Address+0 Byte3
  - Base Address+1 Byte2
  - Base Address+2 Byte1
  - Base Address+3 Byte0

Dr. William T. Verts An Essay on Endian Order, http://www.cs.umass.edu/~verts/cs32/endian.html, April 19, 1996

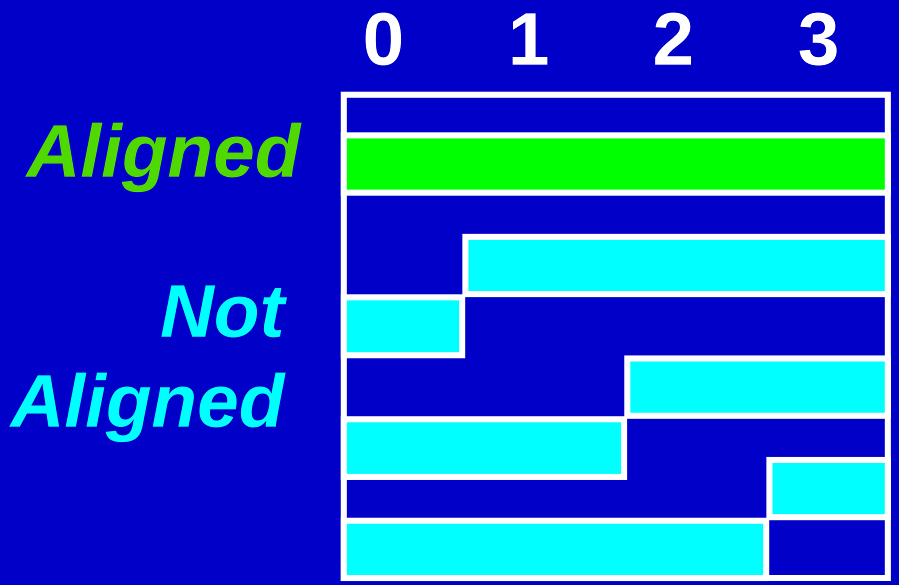
Endian Order is Also Important to File Data

- Adobe Photoshop -- Big Endian
- BMP (Windows and OS/2 Bitmaps) -- Little Endian
- DXF (AutoCad) -- Variable
- GIF -- Little Endian
- JPEG -- Big Endian
- PostScript -- Not Applicable (text!)
- Microsoft RIFF (.WAV & .AVI) -- Both, Endian identifier encoded into file
- Microsoft RTF (Rich Text Format) -- Little Endian
- TIFF -- Both, Endian identifier encoded into file

Dr. William T. Verts An Essay on Endian Order, http://www.cs.umass.edu/~verts/cs32/endian.html, April 19, 1996

A Note about Memory: Alignment

- MIPS requires that all words start at addresses that are multiples of 4 bytes



- Called Alignment: objects must fall on address that is multiple of their size

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

Memory Addressing

- Alignment restrictions
  - Accesses to objects larger than a byte must be aligned
  - An access to an object of size *s* bytes at byte address *A* is aligned if $A \bmod s = 0$
  - A misaligned access takes multiple aligned memory references

Outline

- Classifying instruction set architectures
- Memory addressing
- Addressing modes
- Type and size of operands
- Instructions for control
- The role of compiler

Memory Addressing

- All architectures must address memory

A number of questions naturally arise

- What is accessed - byte, word, multiple words?
  - today's machine are byte addressable
    - this is a legacy and probably makes little sense otherwise
  - main memory is really organized in n byte lines
    - e.g. the cache model
- Hence there is a natural alignment problem
  - accessing a word or double-word which crosses 2 lines
    - requires 2 references
  - automagic alignment is possible but hides the number of references
    - also therefore hides an important case of CPI bloat
    - hence a bad idea - guess which company does this?

Addressing Modes

- An important aspect of ISA design
  - has major impact on both the HW complexity and the IC
  - HW complexity affects the CPI and the cycle time
- Basically a set of mappings
  - from address specified to address used
  - address used = *effective address*
  - effective address may go to memory or to a register array
    - which is typically dependent on it’s location in the instruction field
    - in some modes multiple fields are combined to form a memory address
    - register addresses are usually more simple - e.g. they need to be fast
  - effective address generation is an important focus
    - since it is the common case - e.g. every instruction needs it
    - it must also be fast

Example for Addressing Modes

Addressing mode	Example instruction	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
Displacement	Add R4,100(R1)	Regs[R4] <- Regs[R4] + Mem[100 + Regs[R1]]	Accessing local variables (+ simulates register redirect, direct addressing modes)
Register indirect	Add R4, (R1)	Regs[R4] <- Regs[R4] + Mem[Regs[R1]]	Accessing using a pointer or a computed address

Example for Addressing Modes (cont.)

Addressing mode	Example instruction	Meaning	When used
Indexed	Add R3,(R1+R2)	Regs[R3] <- Regs[R3] + Mem[Regs[R1] + Regs[R2]]	Sometimes useful in array addressing: R1 = base of array; R2= index amount
Direct or absolute	Add R1,(1001)	Regs[R1] <- Regs[R1] + Mem[1001]	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect	Add R1,@(R3)	Regs[R1] <- Regs[R1] + Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer p, the mode yields *p

Example for Addressing Modes (cont.)

Addressing mode	Example instruction	Meaning	When used

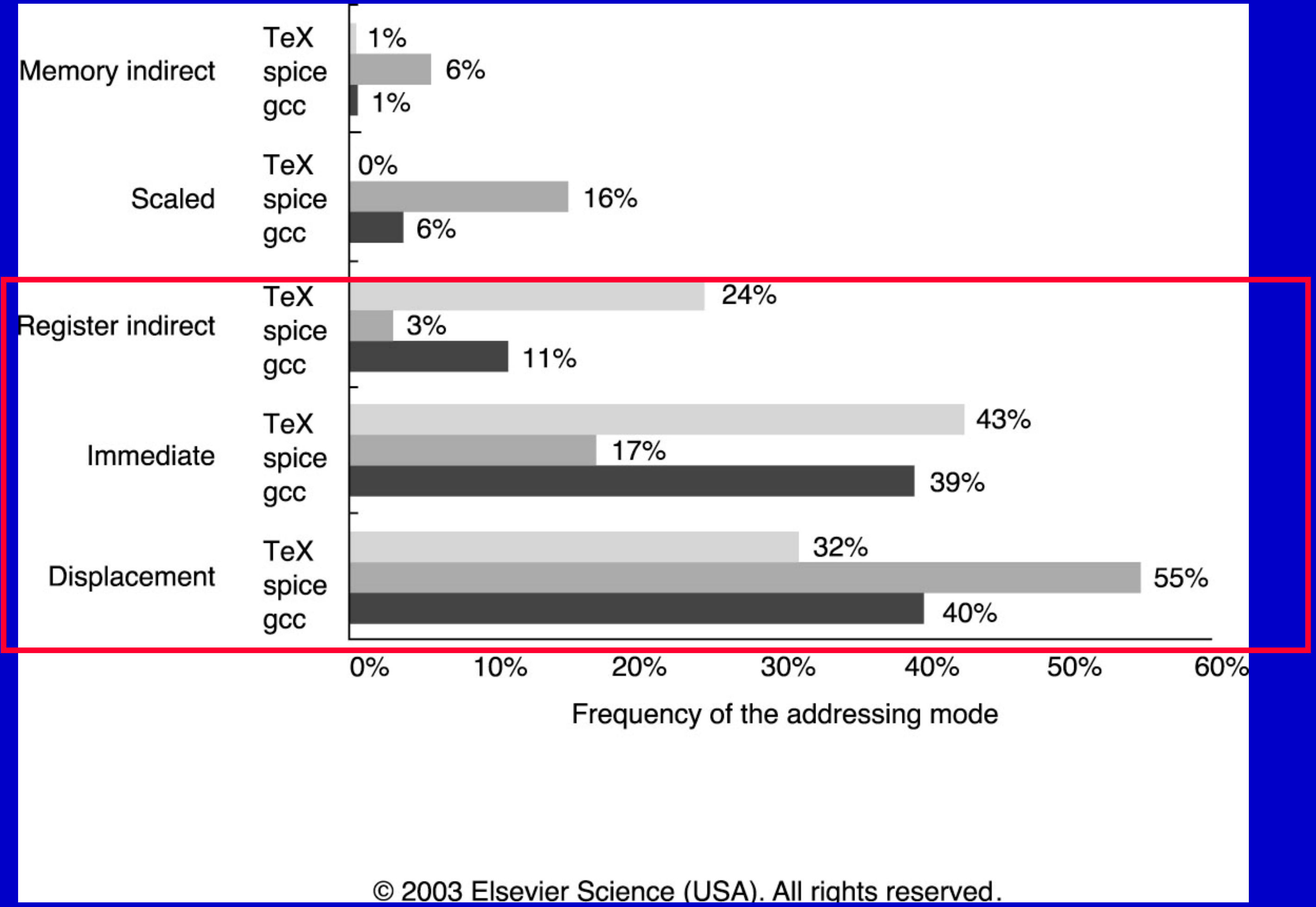
Autoincrement	Add R1,(R2)+	Regs[R1] <- Regs[R1] + Mem[Regs[R2]]
		Regs[R2] <- Regs[R2] + d
			Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d

Autodecrement	Add R1,-(R2)	Regs[R2] <- Regs[R2] – d
		Regs[R1] <- Regs[R1] + Mem[Regs[R2]]
			Same use as autoincrement. Autodecrement/-increment can also act as push/pop to implement a stack

Scaled	Add R1,100(R2)[R3]	Regs[R1] <- Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]	Used to index arrays. May be applied to any indexed addressing mode in some computers

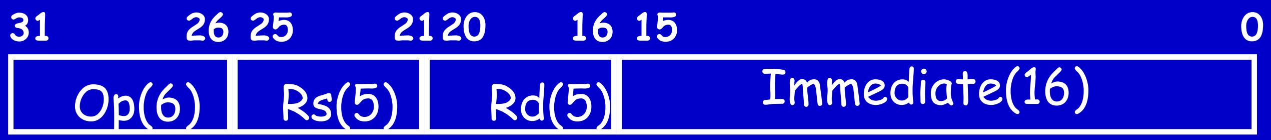
Summary of Use of Memory Addressing Mode

displacement, immediate, and register indirect addressing modes represent 75% to 99% of the addressing mode usage



Example: MIPS Addressing Modes

- MIPS implements only displacement
  - Why? Experiment on VAX (ISA with every mode) found distribution
  - Disp: 61%, reg-ind: 19%, scaled: 11%, mem-ind: 5%, other: 4%
  - 80% use small displacement or register indirect (displacement 0)
- I-type instructions: 16-bit displacement
  - Is 16-bits enough?
  - Yes? VAX experiment showed 1% accesses use displacement >16



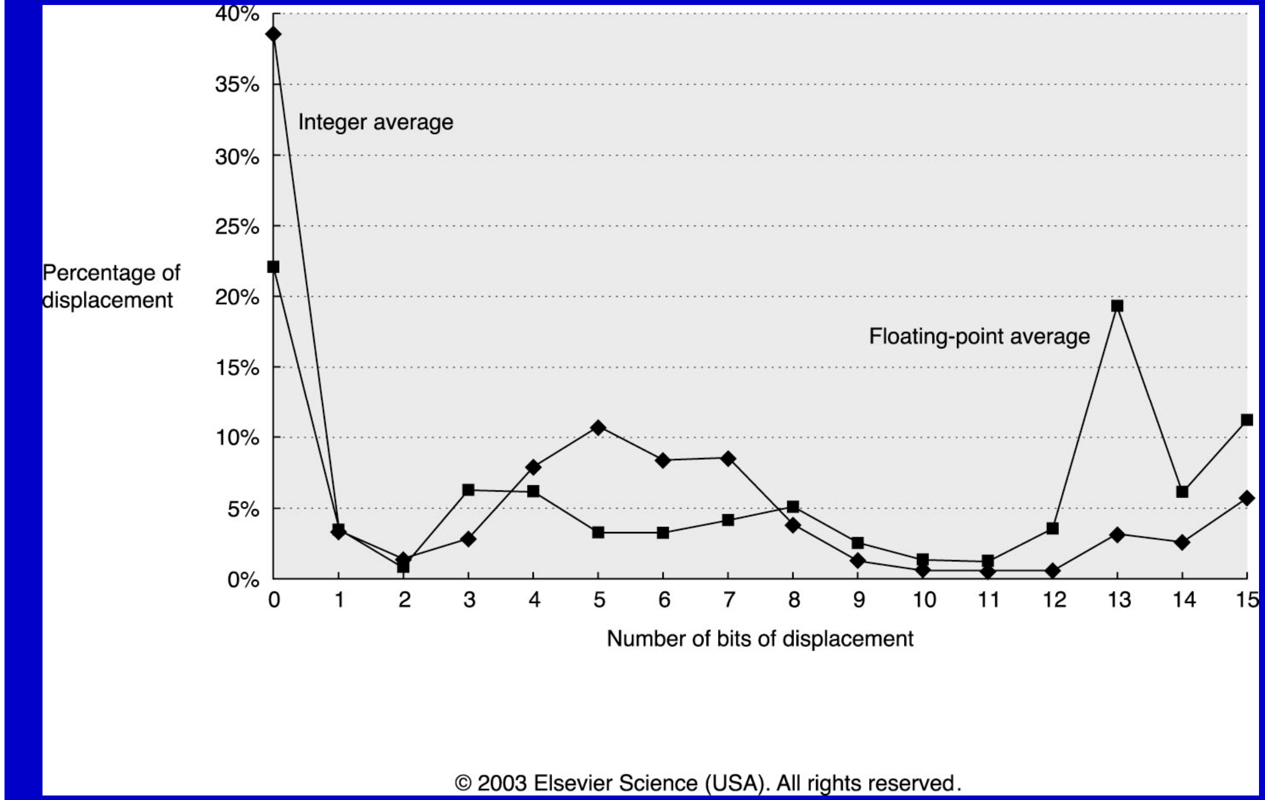
31 26	25 21	20 16	15 0
Op(6)	Rs(5)	Rd(5)	Immediate(16)

Determining Field Size

- Analyze your programs
  - using dynamic traces
  - proper application mix
  - optimizing compiler
- Choose
  - displacement field size
  - immediate or literal field size
  - address modes
  - register file size and structure
- Consider cost of these choices
  - datapath ➔CPI and cycle time
  - code density and encoding

Displacement Addressing Mode

- What’s an appropriate range of the displacements?

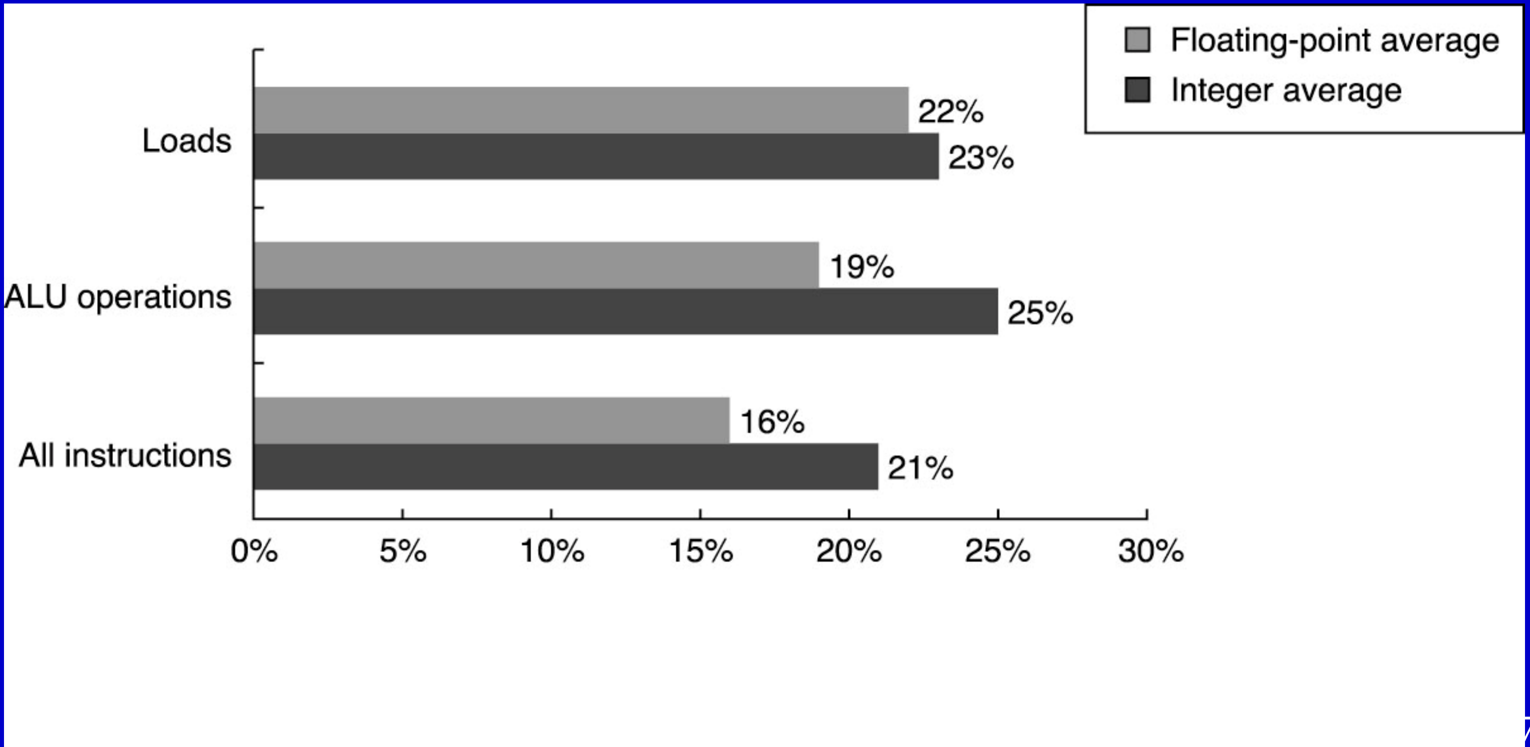


The size of address should be at least 12-16 bits, which capture 75% to 99% of the displacements

For Alpha architecture

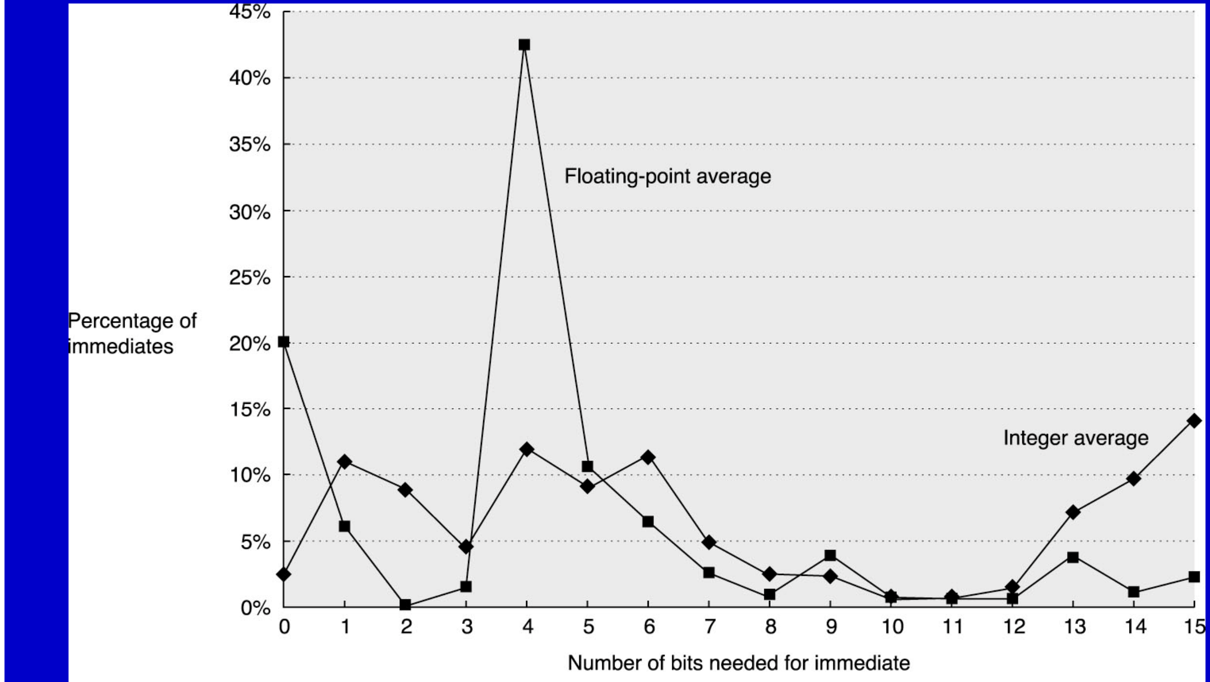
Immediate or Literal Addressing Mode

- Does the mode need to be supported for all operations or for only a subset?



Immediate Addressing Mode (cont.)

- What's a suitable range of values for immediates?

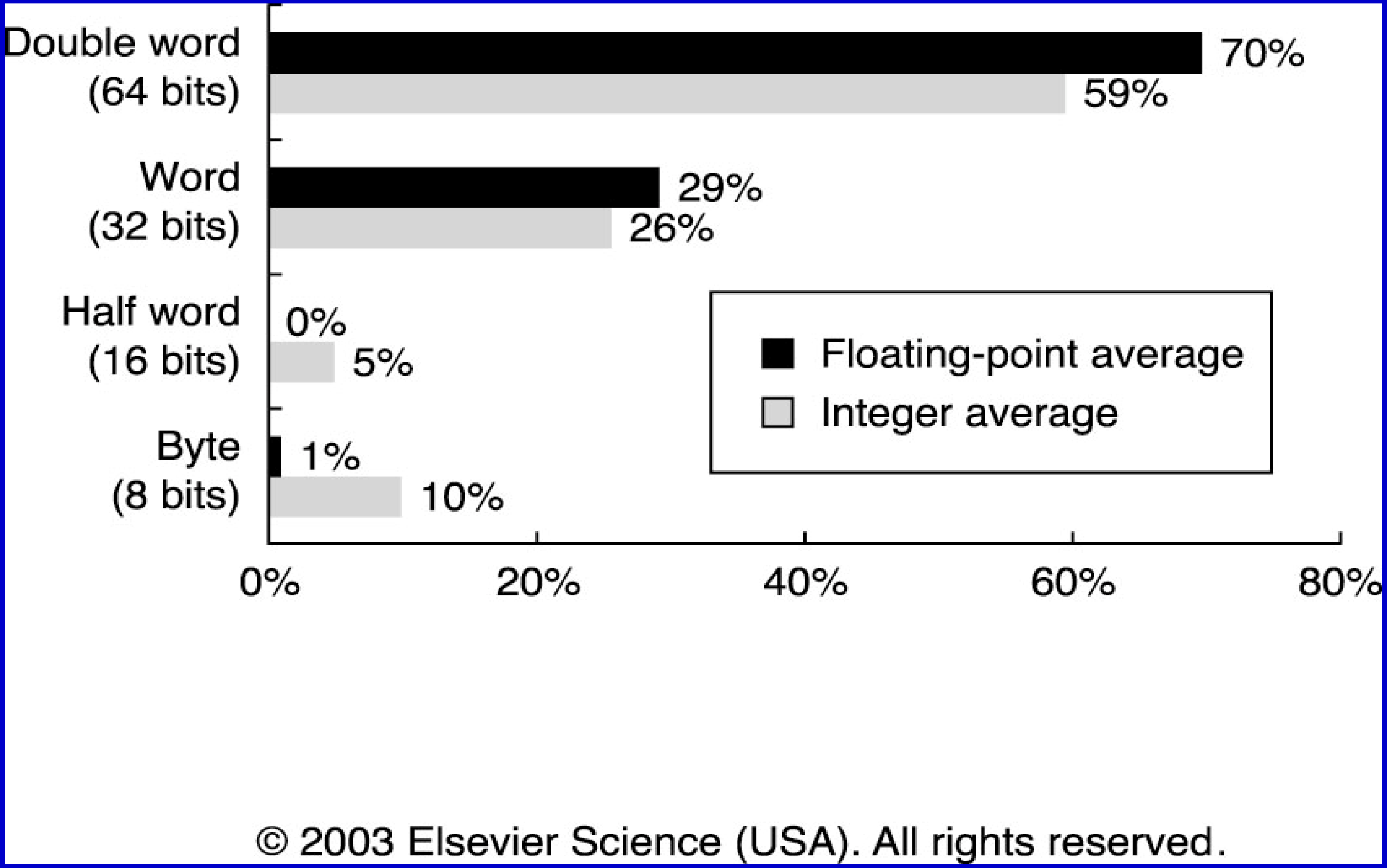


The size of the immediate field should be at least 8-16 bits, which capture 50% to 80% of the immediates

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



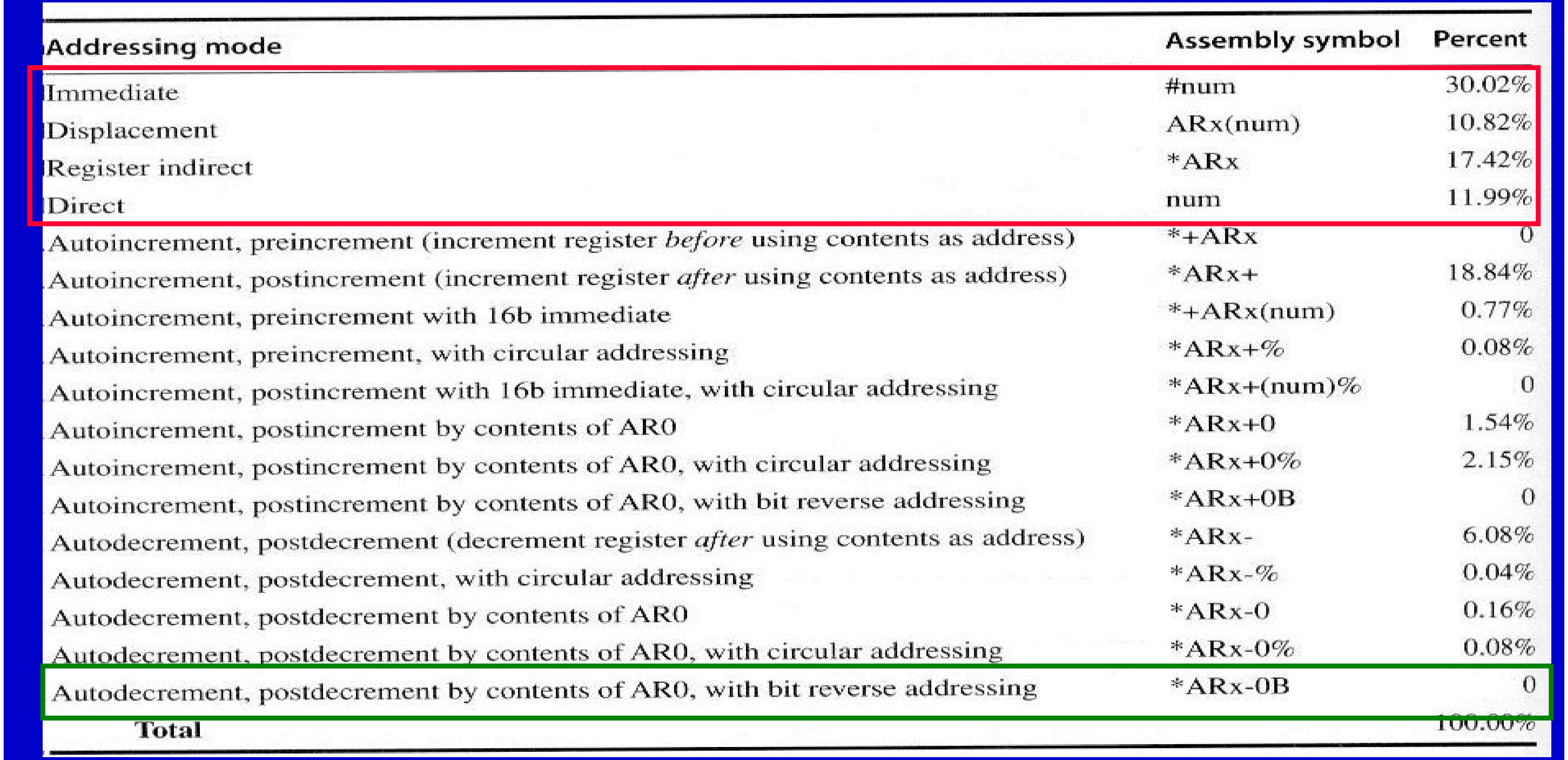
Addressing Modes for the DSP World

- Data is essentially an infinite stream
  - hence model memory as a circular buffer
    - register holds a pointer
    - 2 other registers hold start and end mark
    - autoincrement and autodecrement detect end and then reset
  - hence modulo or circular addressing mode
- FFT is an important application
  - FFT shuffle or butterfly
    - reverses the bit order of the effective address
  - bit-reverse mode
    - hw reverses the low order bits of an address
    - number of bits reversed is a parameter depending on which step of`the FFT algorithm you're in at the time
- Importance - 54 DSP algo's on a TI C54x DSP
  - immediate, displacement, register indirect, direct = 70%
  - auto inc/dec = 20% and the rest counts for < 10%

Addressing Modes for Signal Processing

- DSPs deal with infinite, continuous streams of data, they routinely rely on circular buffers
  - Modulo or circular addressing mode
- For Fast Fourier Transform (FFT)
  - Bit reverse addressing
  - $011_2 \rightarrow 110_2$

Frequency of Addressing Modes for TI TMS320C54x DSP



Addressing mode	Assembly symbol	Percent
Immediate	#num	30.02%
Displacement	ARx(num)	10.82%
Register indirect	*ARx	17.42%
Direct	num	11.99%
Autoincrement, preincrement (increment register *before* using contents as address)	*+ARx	0
Autoincrement, postincrement (increment register *after* using contents as address)	*ARx+	18.84%
Autoincrement, preincrement with 16b immediate	*+ARx(num)	0.77%
Autoincrement, preincrement, with circular addressing	*ARx+%	0.08%
Autoincrement, postincrement with 16b immediate, with circular addressing	*ARx+(num)%	0
Autoincrement, postincrement by contents of AR0	*ARx+0	1.54%
Autoincrement, postincrement by contents of AR0, with circular addressing	*ARx+0%	2.15%
Autoincrement, postincrement by contents of AR0, with bit reverse addressing	*ARx+0B	0
Autodecrement, postdecrement (decrement register *after* using contents as address)	*ARx-	6.08%
Autodecrement, postdecrement, with circular addressing	*ARx-%	0.04%
Autodecrement, postdecrement by contents of AR0	*ARx-0	0.16%
Autodecrement, postdecrement by contents of AR0, with circular addressing	*ARx-0%	0.08%
Autodecrement, postdecrement by contents of AR0, with bit reverse addressing	*ARx-0B	0
Total		100.00%

Outline

- Classifying instruction set architectures
- Memory addressing
- Addressing modes
- Type and size of operands
- Instructions for control
- The role of compiler

Type and Size of Operands

- Character: 8 bits
- Half word: 16 bits
- Words: 32 bits
- Double-precision floating point: 2 words (64 bits)

Type and Size of Operands

- How is the type of an operand designated?
  - Encoding in the opcode
    - For an instruction, the operation is typically specified in one field, called the opcode
  - By tag (not used currently)
- Common operand types
  - Character
    - 8-bit ASCII
    - 16-bit Unicode (not yet used)
  - Integer
    - One-word 2’s complement

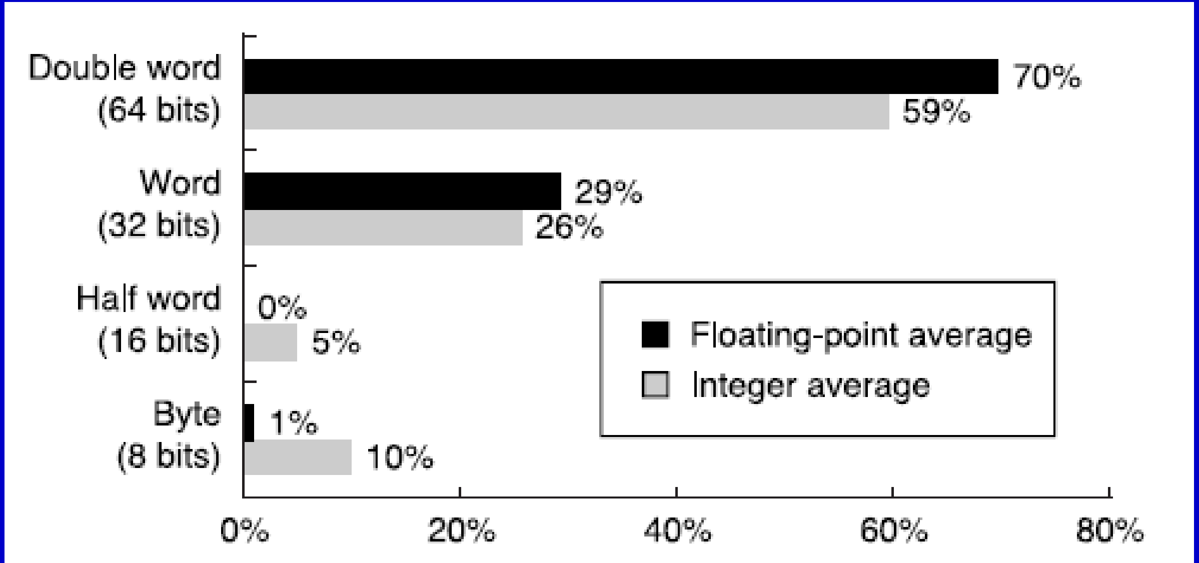
Common Operand Types (cont.)

- Single-precision floating point
  - One-word IEEE 754
- Double-precision floating point
  - 2-word IEEE 754
- Packed decimal (binary-coded decimal)
  - 4 bits encode the values 0-9
  - 2 decimal digits are packed into one byte

Two More Addressing Issues

- **Access alignment**: address % size == 0?
  - Aligned: **load-word @XXXX00, load-half @XXXXX0**
  - Unaligned: **load-word @XXXX10, load-half @XXXXX1**
  - Question: what to do with unaligned accesses (uncommon case)?
    - Support in hardware? Makes all accesses slow
    - Trap to software routine? Possibility
    - Use regular instructions
      - Load, shift, load, shift, and
    - **MIPS? ISA support**: unaligned access using two instructions
      - **lwl @XXXX10; lwr @XXXX10**
- **Endian-ness**: arrangement of bytes in a word
  - Big-endian: sensible order (e.g., MIPS, PowerPC)
    - A 4-byte integer: “00000000 00000000 00000010 00000011” is 515
  - Little-endian: reverse order (e.g., x86)
    - A 4-byte integer: “00000011 00000010 00000000 00000000 ” is 515
  - Why little endian? To be different? To be annoying? Nobody knows

SPEC2000 Operand Sizes



Media and Signal Processing

- New data types
  - e.g. vertex
    - 32-bit floating-point values for x,y,z and w
  - pixel
    - 4 8-bit channels: RGB and A (transparency)
- New numeric type for DSP land
  - fixed point for numbers between +1 and -1
    - 0100 0000 0000 0000 ➔ 2^{-1}
- New operations
  - inner product is a common case
    - hence MAC performance is important
    - effectively an ax + previous b style of computation
    - sometimes called a *fused operation*

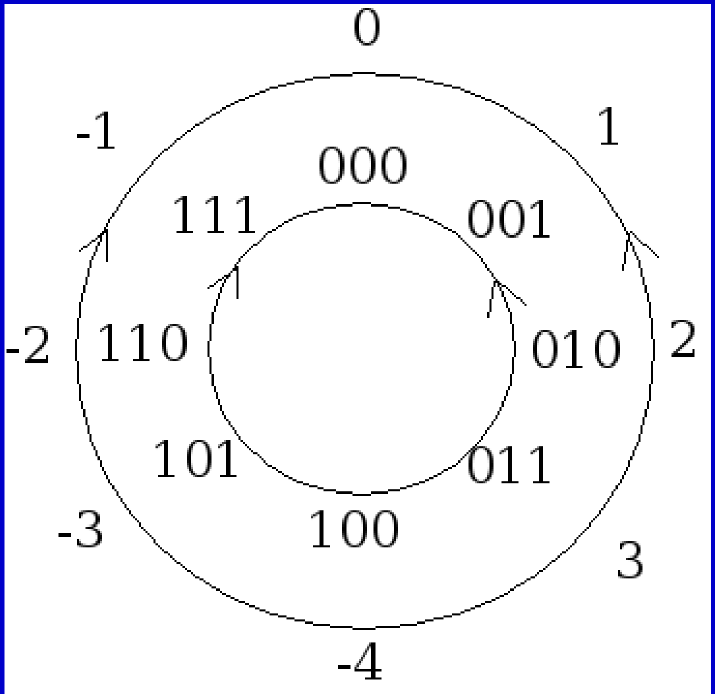
Operands for Media and Signal Processing

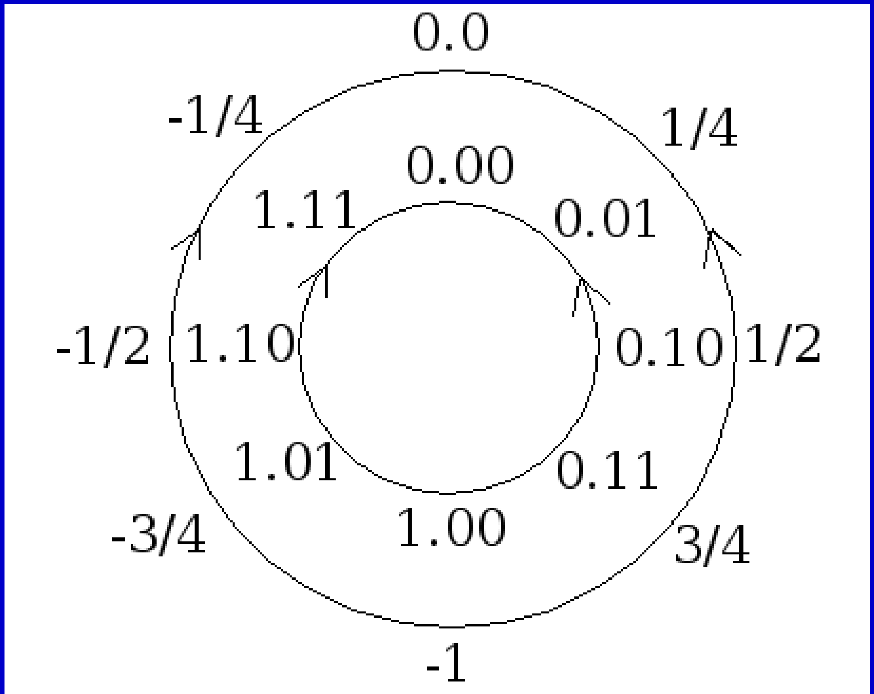
- Vertex
  - (x, y, z) + w to help with color or hidden surfaces
  - 32-bit floating-point values
- Pixel
  - (R, G, B, A)
  - Each channel is 8-bit

Special DSP Operands

- Fixed-point numbers
  - A binary point just to the right of the sign bit
  - Represent fractions between –1 and +1
  - Need some registers that are wider to guard against round-off error
    - Round-off error
      - a computation by rounding results at one or more intermediate steps, resulting in a result different from that which would be obtained using exact numbers

Fixed-point Numbers (cont.)





2‘ complement number

Fixed-point numbers

Douglas L. Jones, http://cnx.org/content/m11930/latest/

Example

- Give three 16-bit patterns:

0100 0000 0000 0000

0000 1000 0000 0000

0100 1000 0000 1000

What values do they represent if they are two’s complement integers? Fixed-point numbers?

- Answer

Two’s complement: 2^{14}, 2^{11}, $2^{14} + 2^{11} + 2^{3}$

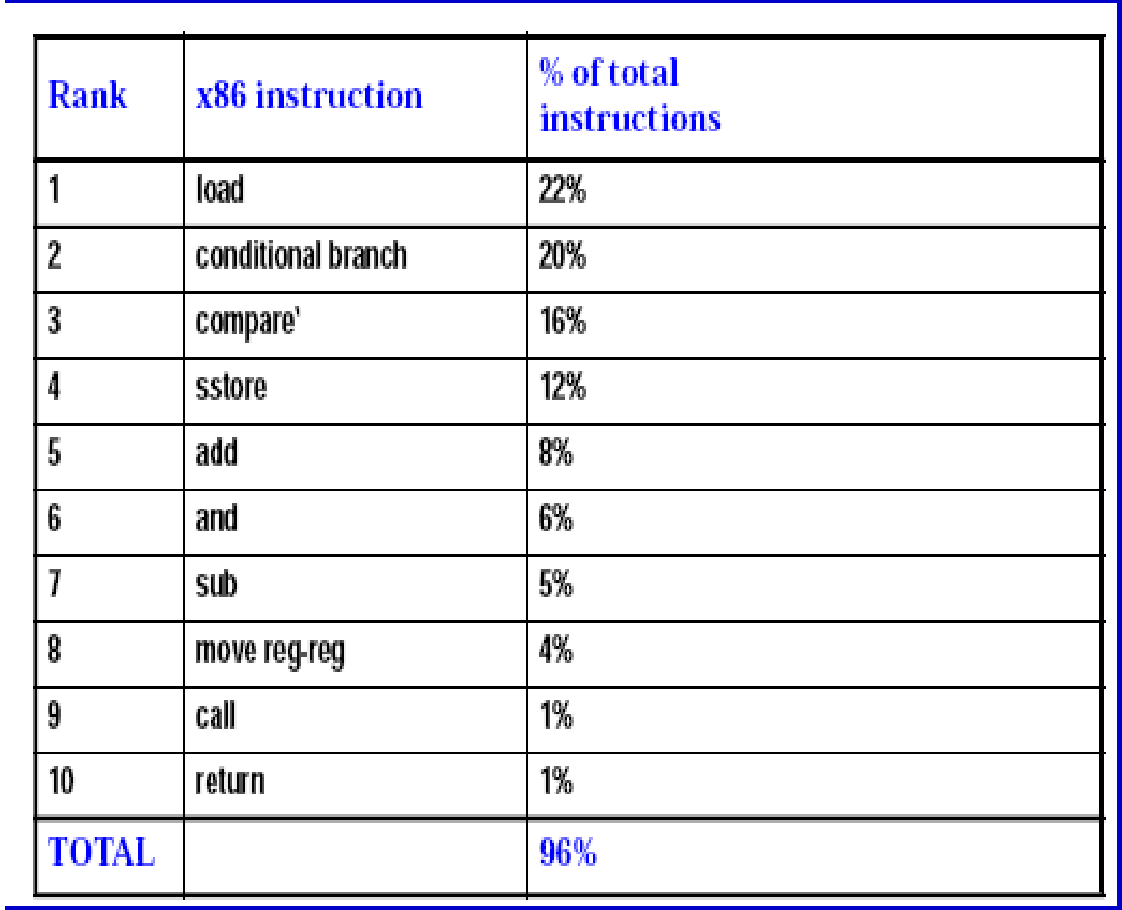
Fixed-point numbers: 2^{-1}, 2^{-4}, $2^{-1} + 2^{-4} + 2^{-12}$

Bit-Pattern,	Unsigned,	2’s Comp,	1’s Comp,
$b_3b_2b_1b_0$			
1111	15	-1	0
1110	14	-2	-1
1101	13	-3	-2
1100	12	-4	-3
1011	11	-5	-4
1010	10	-6	-5
1001	9	-7	-6
1000	8	-8	-7
0111	7	7	7
0110	6	6	6
0101	5	5	5
0100	4	4	4
0011	3	3	3
0010	2	2	2
0001	1	1	1
0000	0	0	0

Outline

- Classifying instruction set architectures
- Memory addressing
- Addressing modes
- Type and size of operands
- Instructions for control
- The role of compiler

SPECint92 codes



Rank	x86 instruction	% of total instructions
1	load	22%
2	conditional branch	20%
3	compare'	16%
4	sstore	12%
5	add	8%
6	and	6%
7	sub	5%
8	move reg-reg	4%
9	call	1%
10	return	1%
TOTAL		96%

- The most widely executed instructions are the simple operations of an instruction set
- The top-10 instructions for 80x86 account for 96% of instructions executed
- Make them fast, as they are the common case

What Operations are Needed

- Arithmetic and Logical
  - Add, subtract, multiple, divide, and, or
- Data Transfer
  - Loads-stores
- Control
  - Branch, jump, procedure call and return, trap
- System
  - Operating system call, virtual memory management instructions

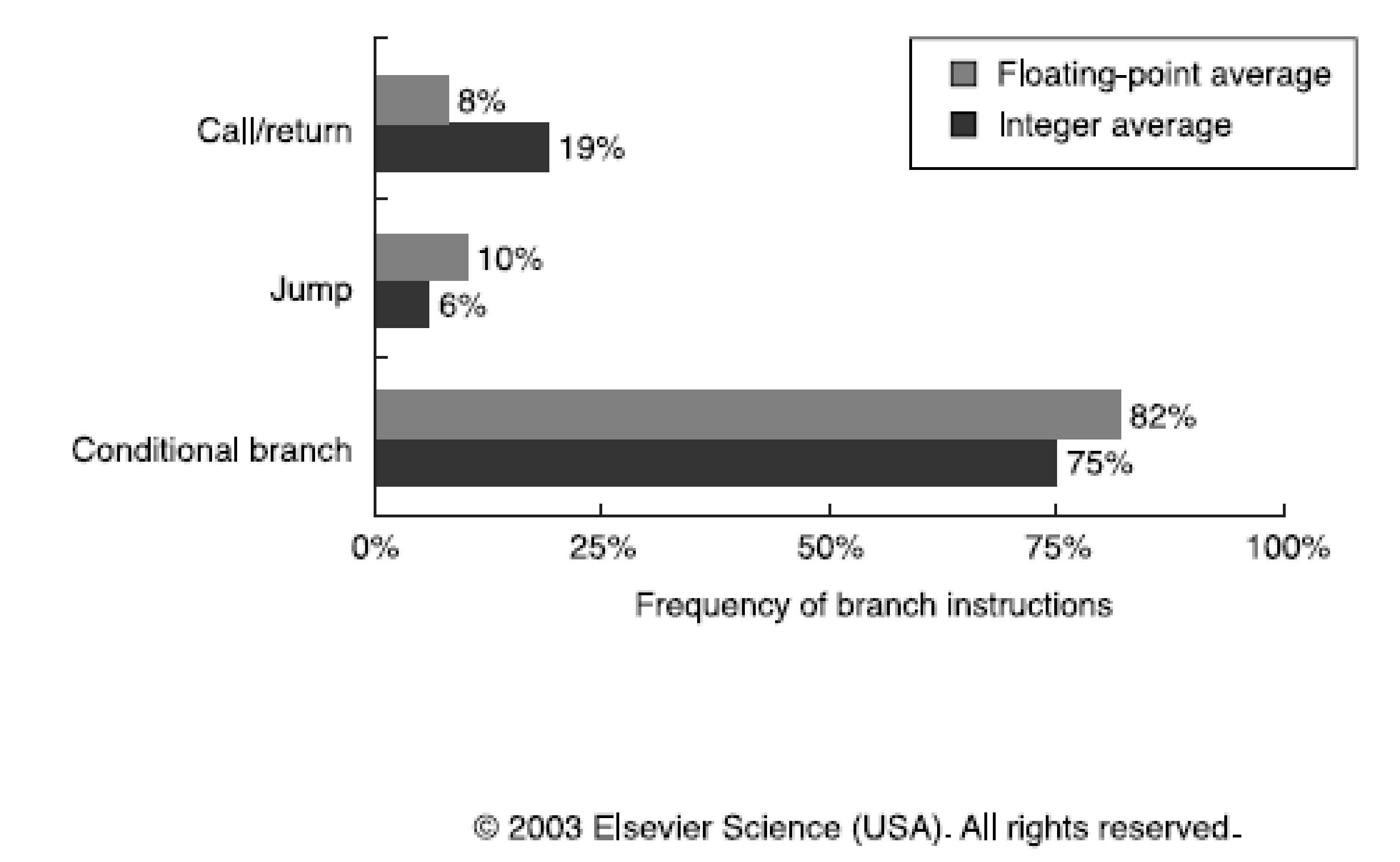
All computers provide the above operations

What Operations are Needed (cont.)

- Floating Point
  - Add, multiple, divide, compare
- Decimal
  - Add, multiply, decimal-to-character conversions
- String
  - move, compare, search
- Graphics
  - pixel and vertex operations, compression/decompression operations

The above operations are optional

Relative Frequency of Control Instructions



Operations for Media and Signal Processing

- Partitioned add
  - 16-bit data with a 64-bit ALU would perform four 16-bit adds in a single clock cycle
  - Single-Instruction Multiple-Data (SIMD) or vector
- Paired single operation
  - Pack two 32-bit floating-point operands into a single 64-bit register

Operations for Media and Signal Processing (cont.)

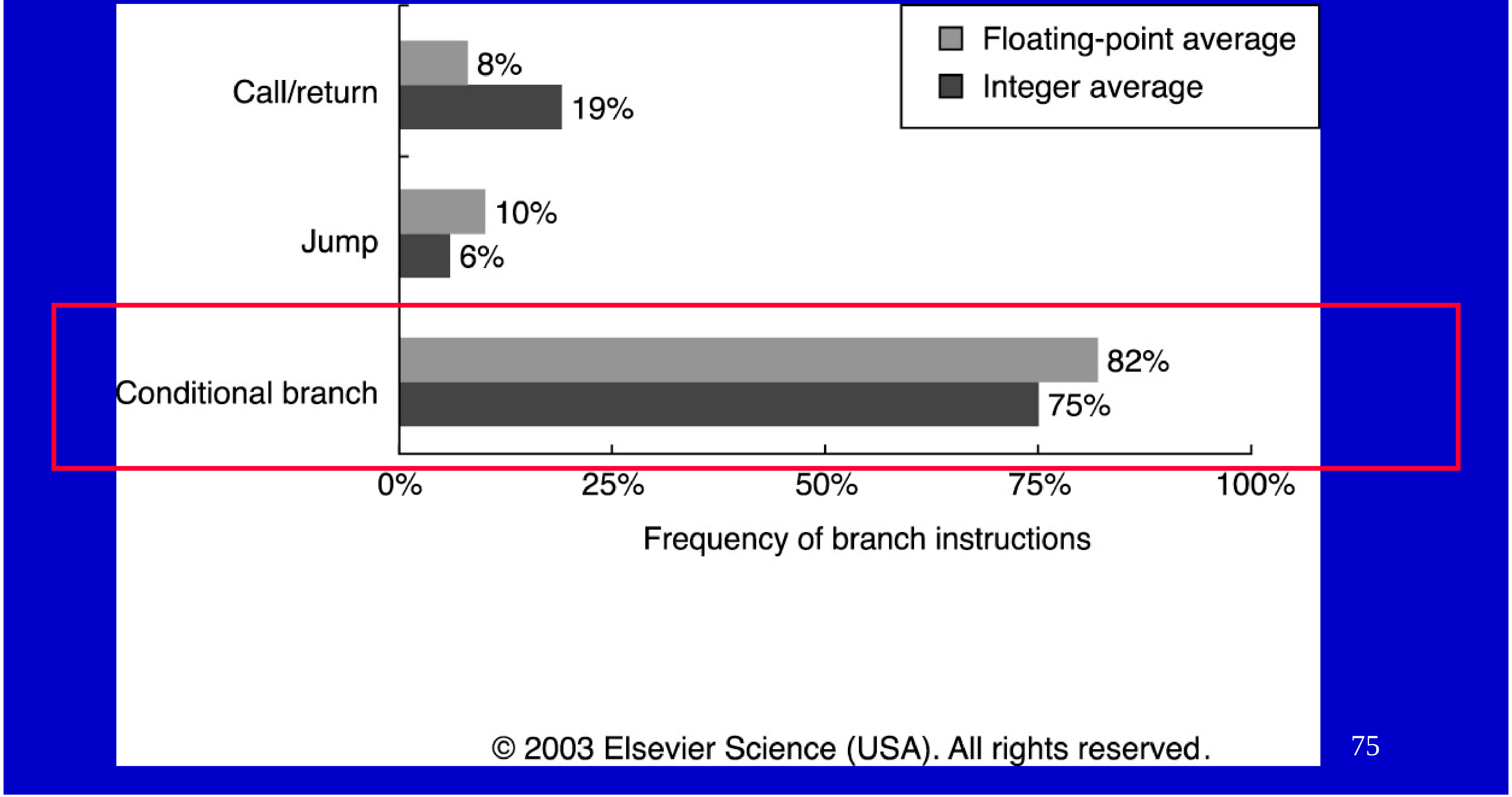
- Saturating arithmetic
  - If the result is too large to be represented, it is set to the largest representable number, depending on the sign of the result
- Several modes to round the wider accumulators into the narrower data words
- Multiply-accumulate instructions
  - a <- a + b*c

Instructions for Control Flow

Control Instruction Types

- Jumps - unconditional transfer
- Conditional Branches
  - how is condition code set?
  - how is target specified? How far away is it?
- Calls
  - how is target specified? How far away is it?
  - where is return address kept?
  - how are the arguments passed? Callee vs. Caller save!
- Returns
  - where is the return address? How far away is it?
  - how are the results passed?

Distribution of Control Flows



Addressing Modes for Control Flow Instructions

- How to get the destination address of a control flow instruction?
  - PC-relative
    - Supply a displacement that is added to the program counter (PC)
    - Position independence
      - Permit the code to run independently of where it is loaded
  - A register contains the target address
  - The jump may permit any addressing mode to be used to supply the target address

Usage of Register Indirect Jumps

- Case & Switch
- Virtual functions or methods
  - C++ or Java
- High-order functions or function pointers
  - C, C++, and Lisp
- DLL's
  - shared libraries that get dynamically loaded

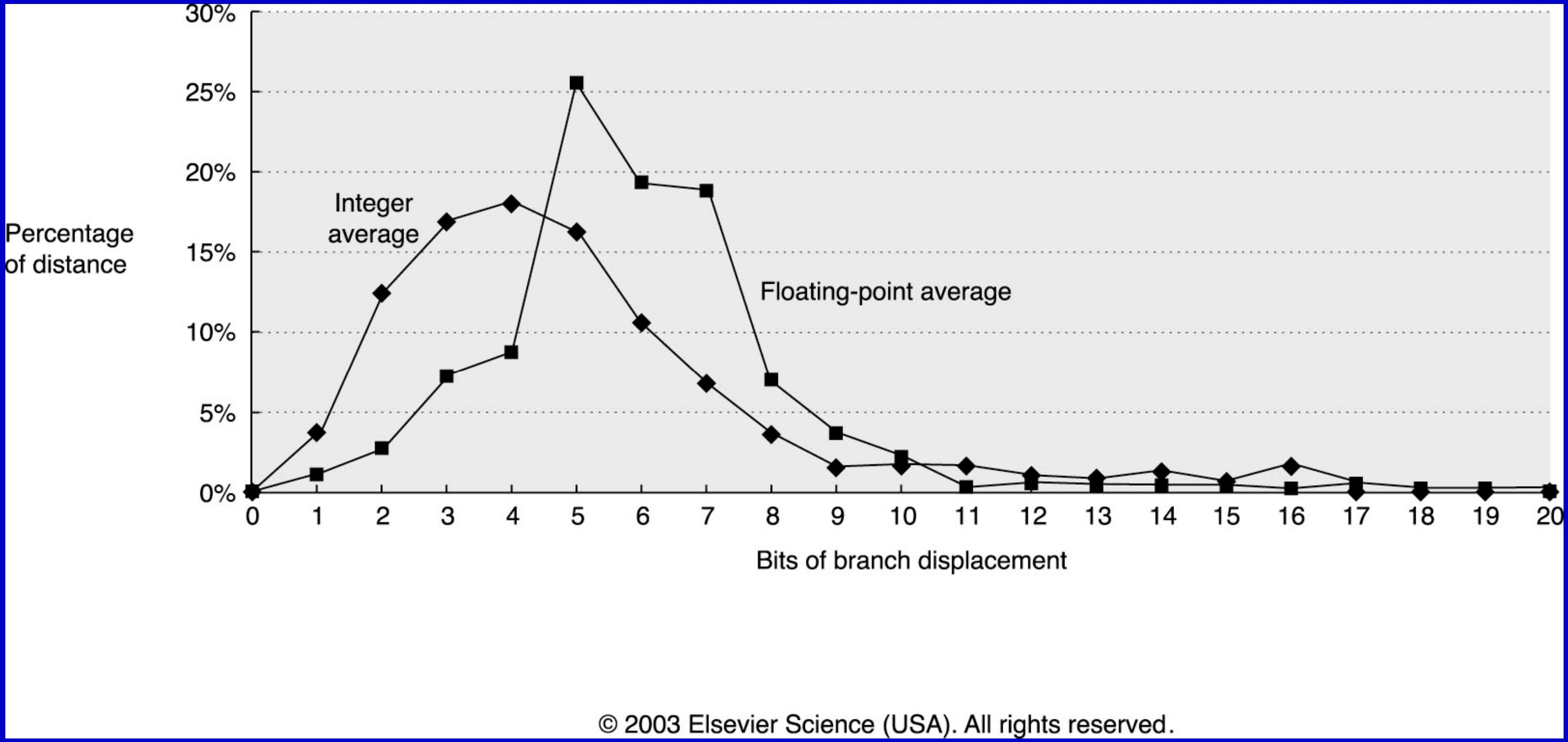
Addressing Modes for Control Instructions

- Known at compile time for unconditional and conditional branches - hence specified in the instruction
  - as a register containing the target address
  - as a PC-relative offset
- Consider word length addresses, registers, and instructions
  - full address desired? Then pick the register option.
  - BUT - setup and effective address will take longer.
  - if you can deal with smaller offset then PC relative works
  - PC relative is also *position independent* - so simple linker duty
    - consider the ease in particular for DLL’s
- How do you find out what works?
  - start by measuring your programs of course.

Control instructions

- 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.

How Far are Branch Targets from Branches?

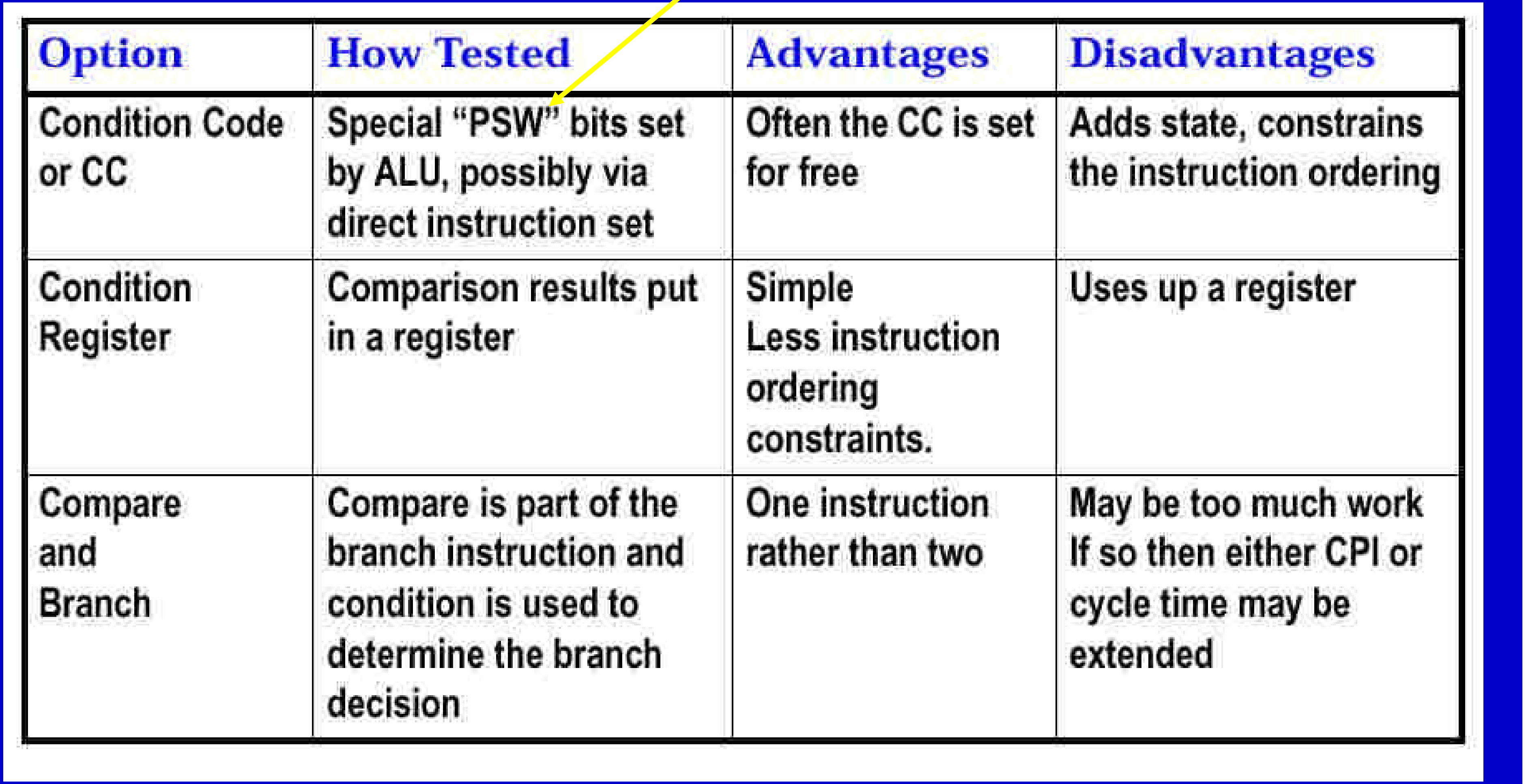


© 2003 Elsevier Science (USA). All rights reserved.

- The most frequent in the integer? programs are to targets that can be encoded in 4-8 bits
- About 75% of the branches are in the forward direction

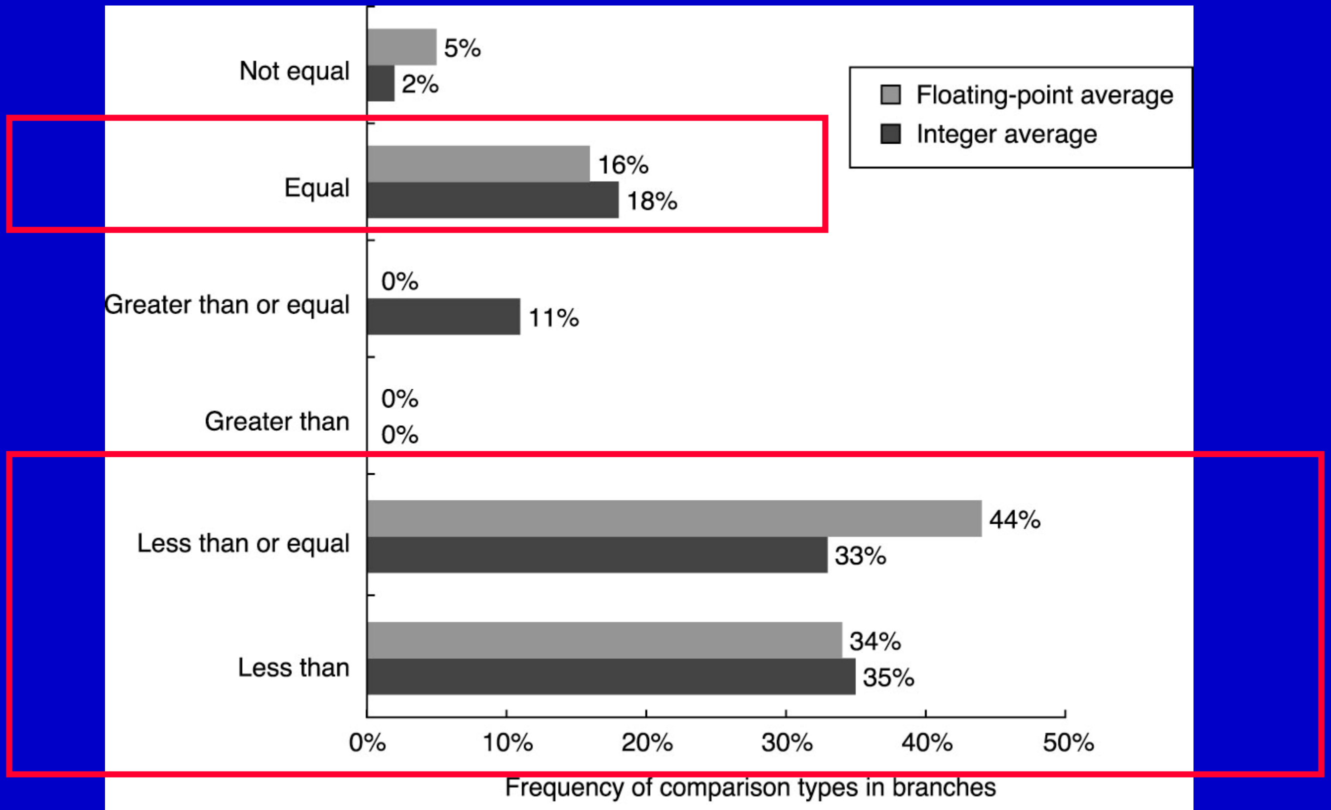
How to Specify the Branch Condition?

Program Status Word



Option	How Tested	Advantages	Disadvantages
Condition Code or CC	Special “PSW” bits set by ALU, possibly via direct instruction set	Often the CC is set for free	Adds state, constrains the instruction ordering
Condition Register	Comparison results put in a register	Simple Less instruction ordering constraints.	Uses up a register
Compare and Branch	Compare is part of the branch instruction and condition is used to determine the branch decision	One instruction rather than two	May be too much work If so then either CPI or cycle time may be extended

Frequency of Different Types of Compares in Branches



Procedure Invocation Options

- The return address must be saved somewhere, sometimes in a special link register or just a GPR
- Two basic schemes to save registers
  - Caller saving
    - The calling procedure must save the registers that it wants preserved for access after the call
  - Callee saving
    - The called procedure must save the registers it want to use

Encoding an Instruction Set

Encoding an Instruction Set

- How the instructions are encoded into a binary representation for execution?
  - Affects the size of code
  - Affects the CPU design
- The operation is typically specified in one field, called the opcode
- How to encode the addressing mode with the operations
  - Address specifier
  - Addressing modes encoded as part of the opcode

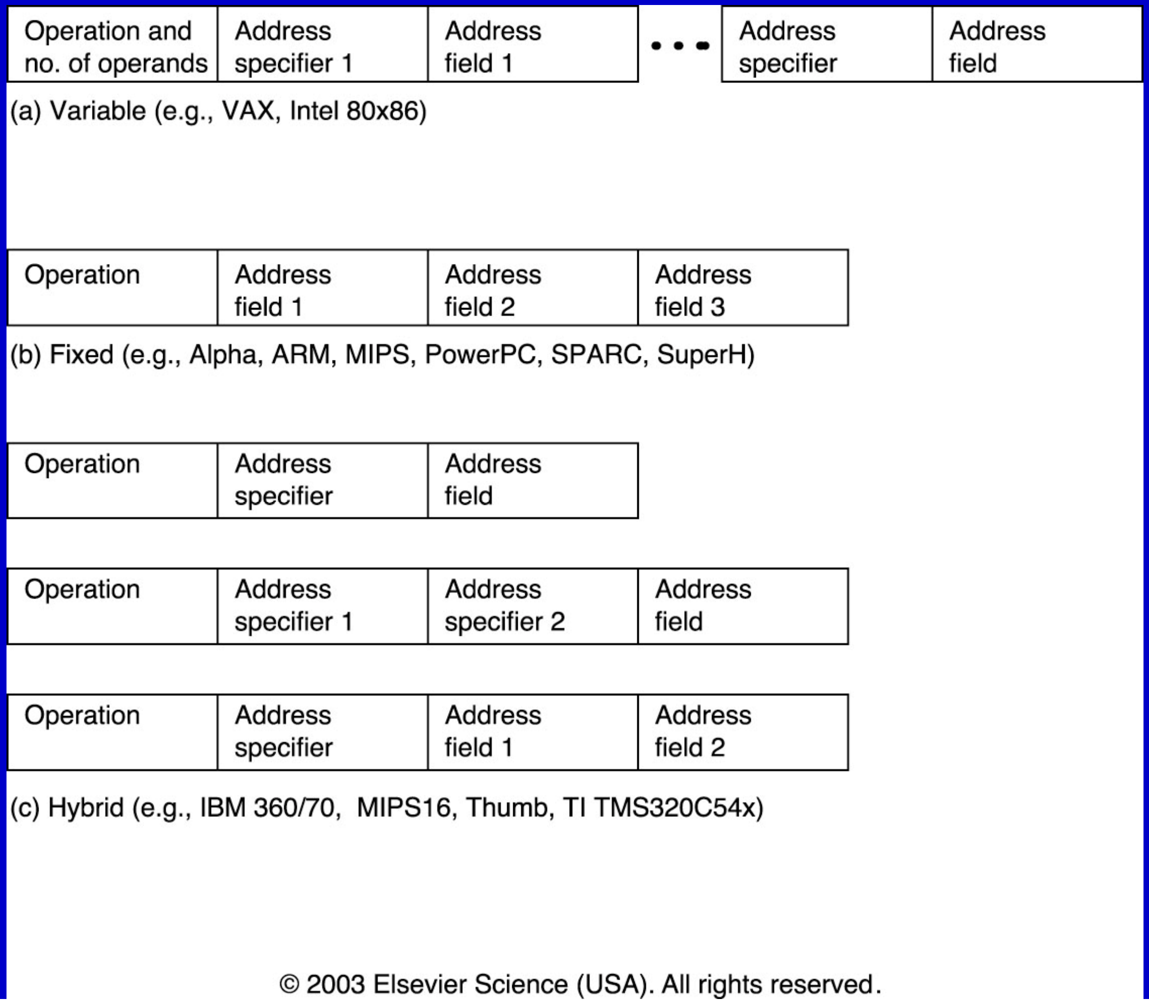
Issues on Encoding an Instruction Set

- Desire for lots of addressing modes and registers
- Desire for smaller instruction size and program size with more addressing modes and registers
- Desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation
  - Multiple bytes, rather than arbitrary bits
  - Fixed-length

3 Popular Encoding Choices

- Variable
  - Allow virtually all addressing modes to be with all operations
- Fixed
  - A single size for all instructions
  - Combine the operations and the addressing modes into the opcode
  - Few addressing modes and operations
- Hybrid
  - Size of programs vs. ease of decoding in the processor
  - Set of fixed formats

3 Popular Encoding Choices (Cont.)



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)

Reduced Code Size in RISCs

- More narrower instructions
- Compression

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

Outline

- Classifying instruction set architectures
- Memory addressing
- Addressing modes
- Type and size of operands
- Instructions for control
- The role of compiler

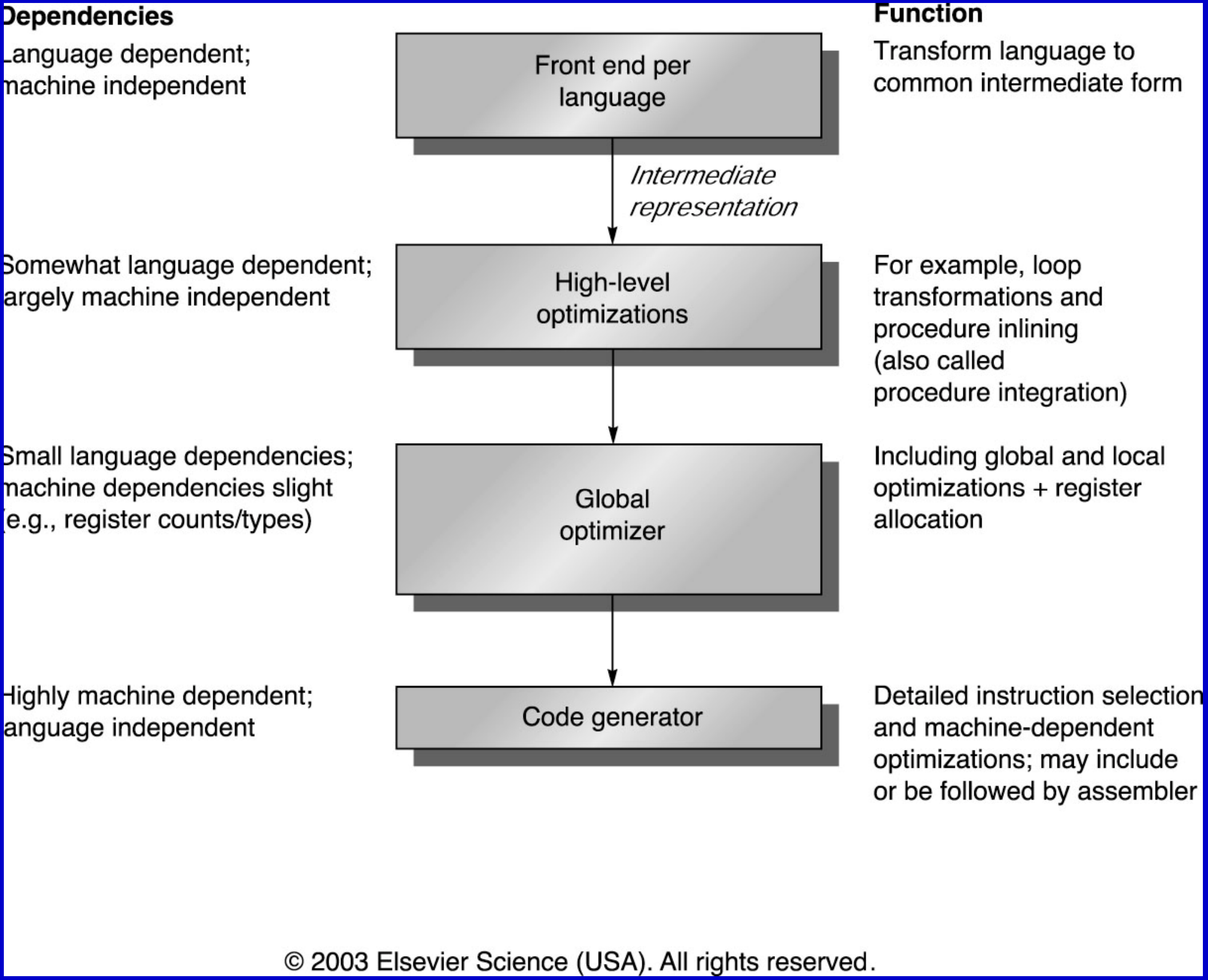
Compiler vs. ISA

- Almost all programming is done in high-level language (HLL) for desktop and server applications
- Most instructions executed are the output of a compiler
- So, separation from each other is impractical

Goals of a Compiler

- Correctness
- Speed of the compiled code
- Others
  - Fast compilation
  - Debugging support
  - Interoperability among languages

Structure of Compiler



Dependencies

Language dependent; machine independent

Somewhat language dependent; largely machine independent

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

Highly machine dependent; language independent

Function

Transform language to common intermediate form

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

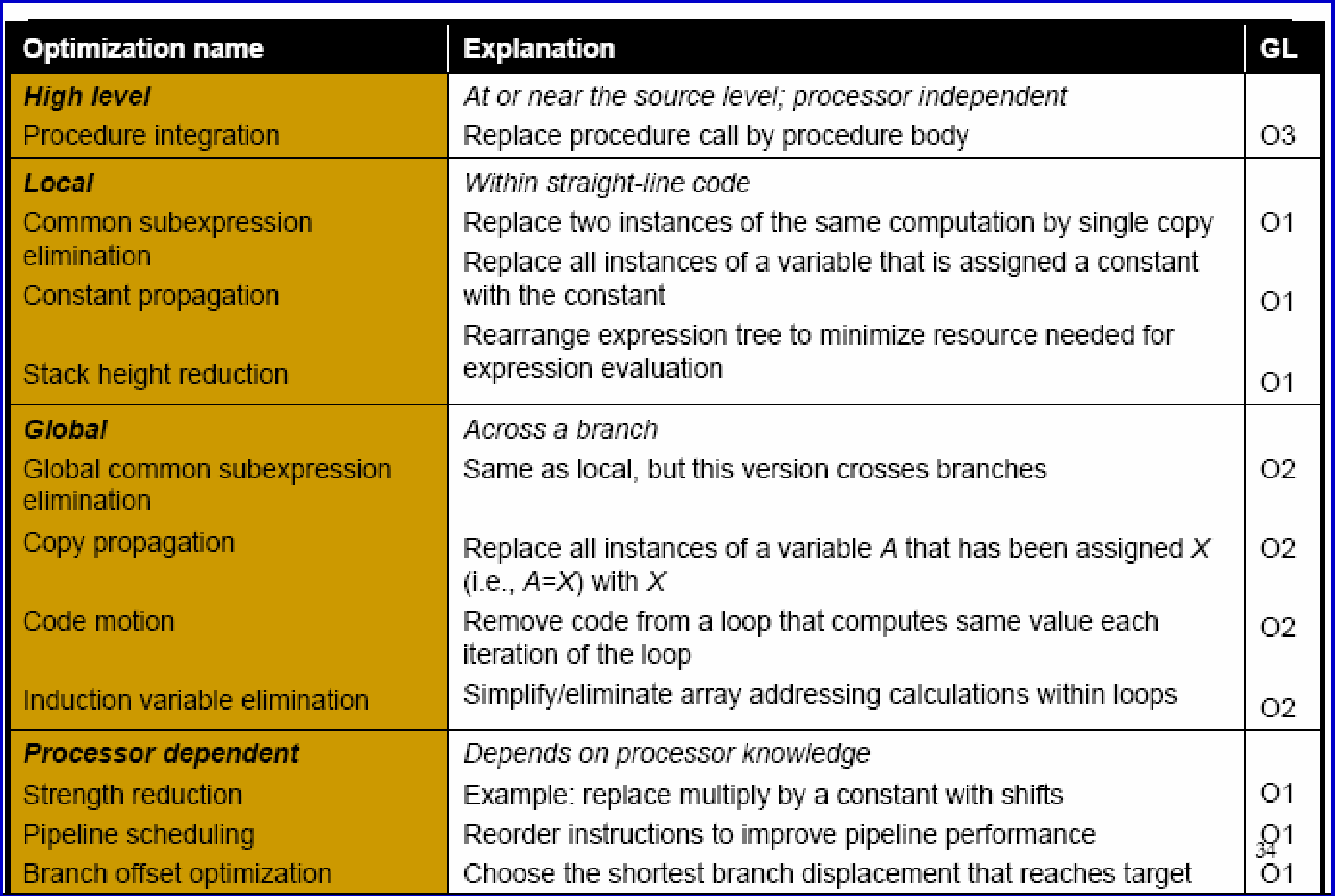
Including global and local optimizations + register allocation

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

Optimization types

- High level - done at source code level
  - procedure called only once - so put it in-line and save CALL
    - more general is to in-line if call-count < some threshold
- Local - done on basic sequential block
  - common subexpressions produce same value - either allocate a register or replace with single copy
  - constant propagation - replace constant valued variable with the constant - saves multiple variable accesses with same value
  - stack reduction - rearrange expression to minimize temporary storage needs
- Global - same as local but done across branches
  - primary goal = optimize loops
    - code motion - remove code from loops that compute same value on each pass and put it before the loop
    - simplify or eliminate array addressing calculations in loops
- Register allocation
  - Associate registers with operands
    - Graph coloring
- Processor-dependent
  - Depend on processor knowledge

Compiler Optimization Summary



Optimization name	Explanation	GL
High level	*At or near the source level; processor independent*	
Procedure integration	Replace procedure call by procedure body	O3
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	O1
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	O1
Stack height reduction	Rearrange expression tree to minimize resource needed for expression evaluation	O1
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	O2
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., *A*=*X*) with *X*	O2
Code motion	Remove code from a loop that computes same value each iteration of the loop	O2
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	O2
Processor dependent	*Depends on processor knowledge*	
Strength reduction	Example: replace multiply by a constant with shifts	O1
Pipeline scheduling	Reorder instructions to improve pipeline performance	O1
Branch offset optimization	Choose the shortest branch displacement that reaches target	O1

Machine Dependent Optimizations

- Strength reduction
  - replace multiply with shift and add sequence
    - would make sense if there was no hardware support for MUL
    - a trickier version: 17x = arithmetic left shift 4 and add
- Pipeline scheduling
  - reorder instructions to minimize pipeline stalls
  - dependency analysis
  - compiler can’t see run time dynamics - e.g. branch direction resolution
- Branch offset optimization
  - reorder code to minimize branch offsets

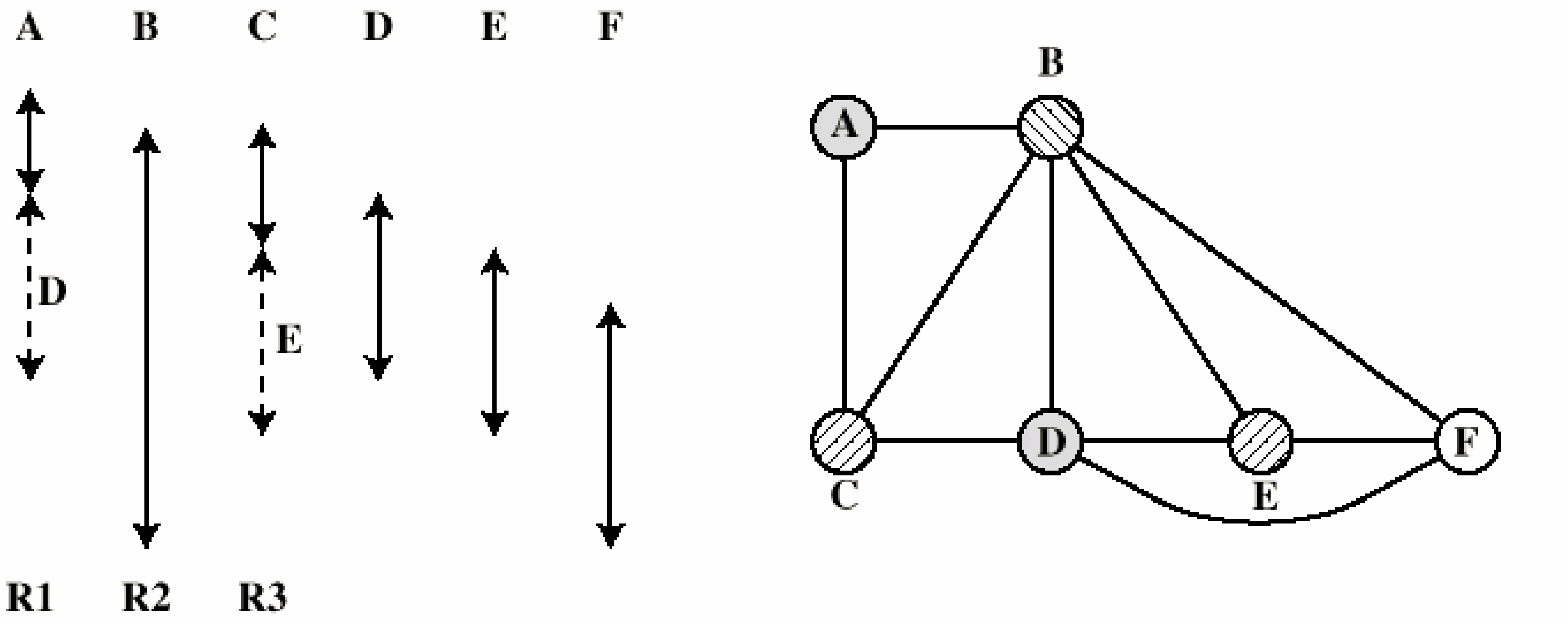
Compiler Based Register Optimization

- 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

Graph Coloring

- Given a graph of nodes and edges
- Assign a color to each node
- Adjacent nodes have different colors
- Use minimum number of colors
- Nodes are symbolic registers
- Two registers that are live in the same program fragment are joined by an edge
- Try to color the graph with *n* colors, where *n* is the number of real registers
- Nodes that can not be colored are placed in memory

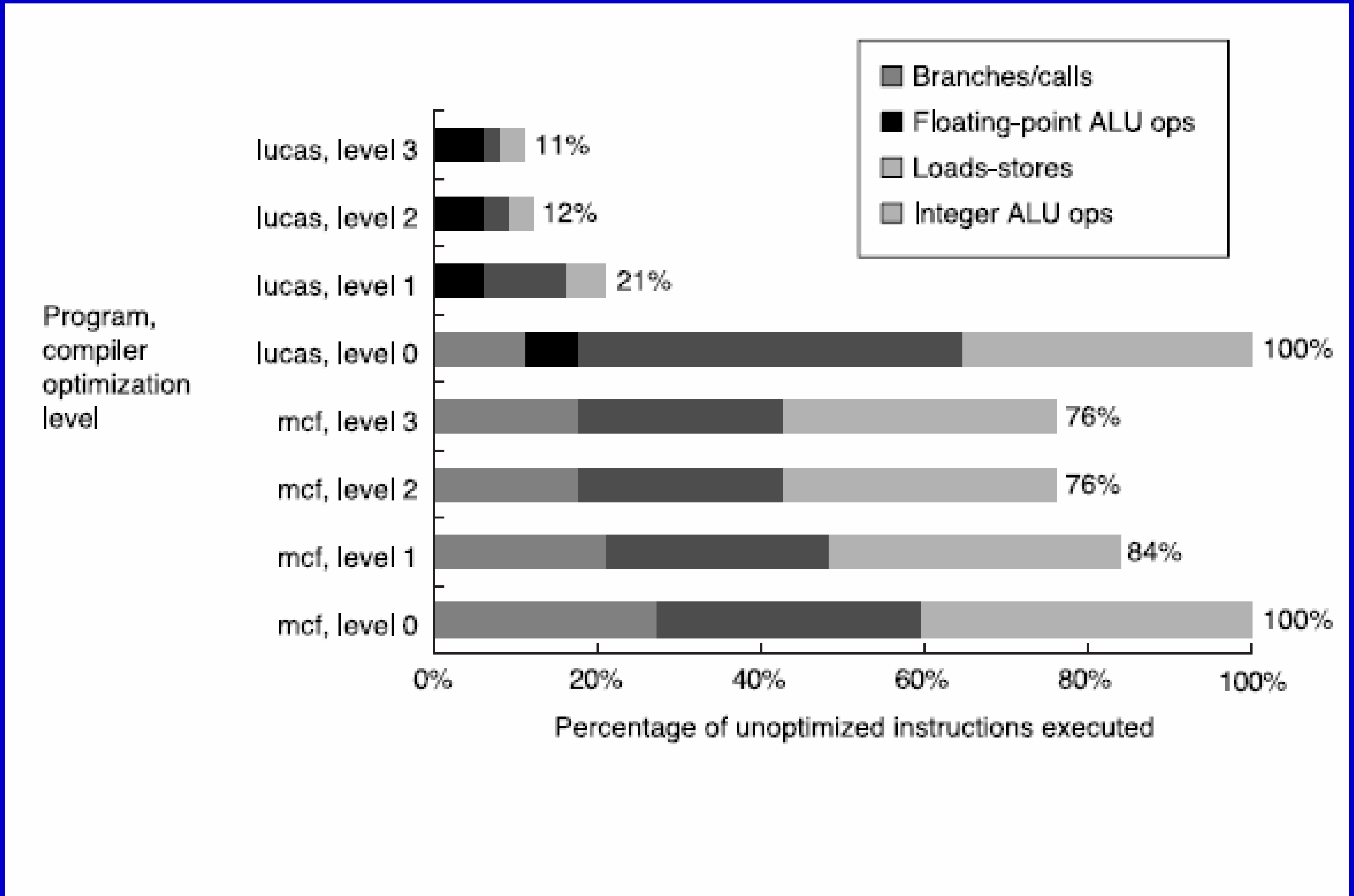
Graph Coloring Approach



(a) Time sequence of active use of registers

(b) Register interference graph

Compiler Optimization Levels



Level 0: unoptimized code
Level 1: local optimization, code scheduling, local register optimization
Level 2: global optimization, loop transformation, global register optimization
Level 3: procedure integration

Things you care about for the ISA design

- **Key: make the common case fast and the rare case correct**
- How are variables allocated?
- How are variables addressed?
- How many registers will be needed?
- How does optimization change the instruction mix?
- What control structures are used?
- How frequently are the control structures used?

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

Register Allocation Problem

- Register allocation is much effective for statck-allocated objects than for global variables.
- Register allocation is impossible for heap-allocated objects because they are accessed with pointers.
- Global variables and some stack variables are impossible to allocated because they are aliased.
  - Multiple ways to refer the address of a variable, making it illegal to put it into a register

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	

A "Typical" RISC

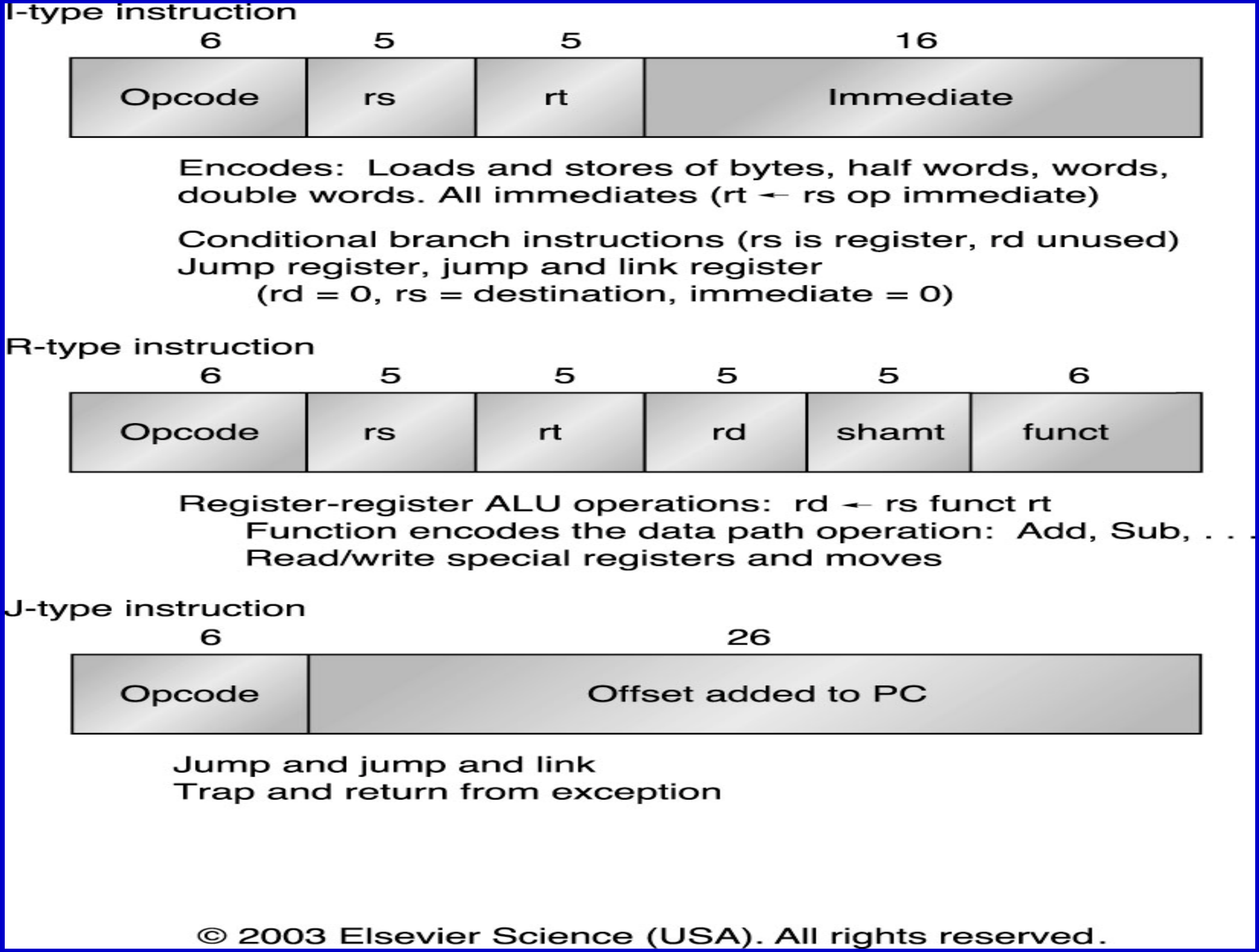
- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, Double Precision takes a register pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  base + displacement
  - no indirection
- Simple branch conditions

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

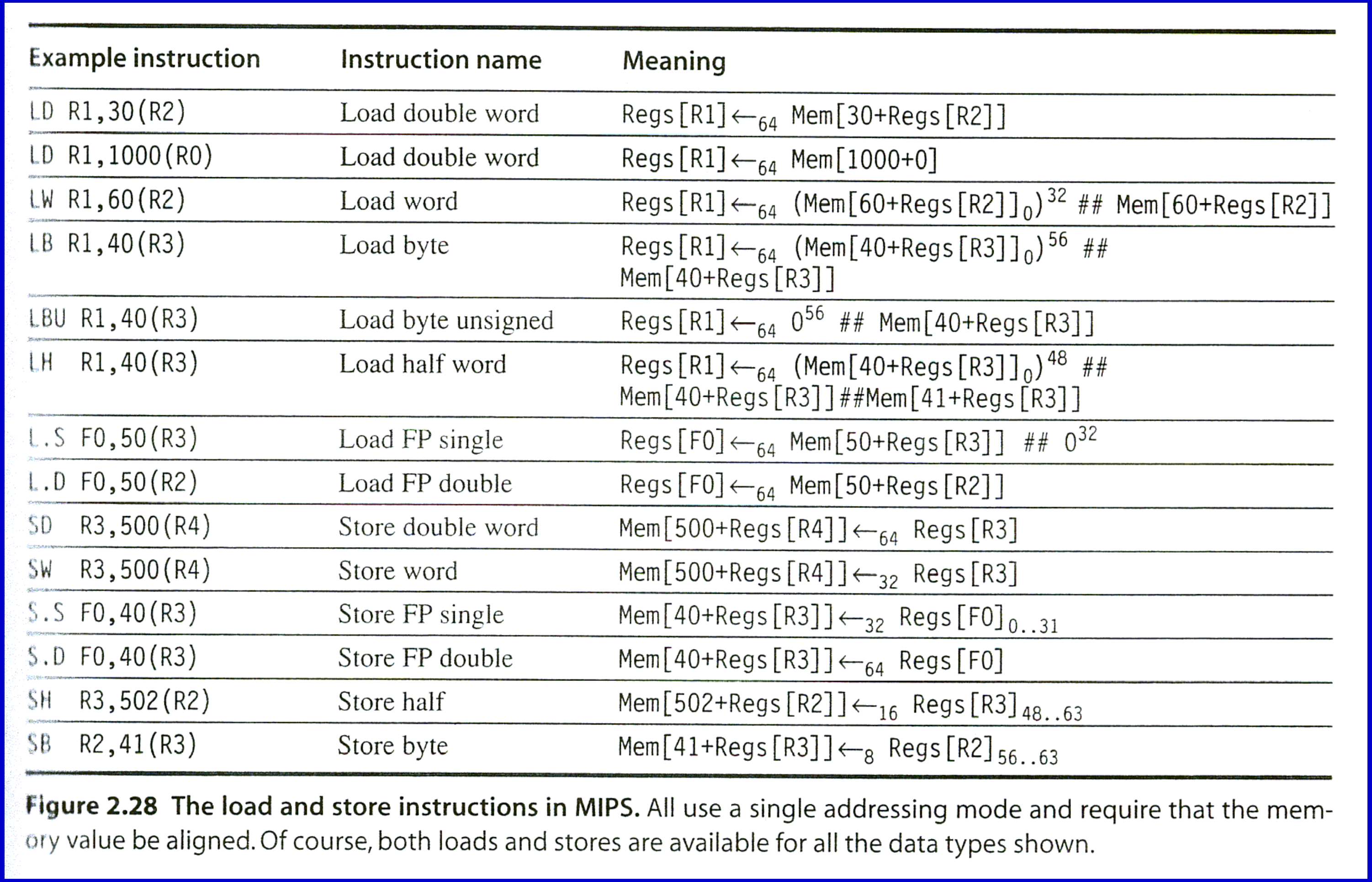
MIPS (addressing modes)

- 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)

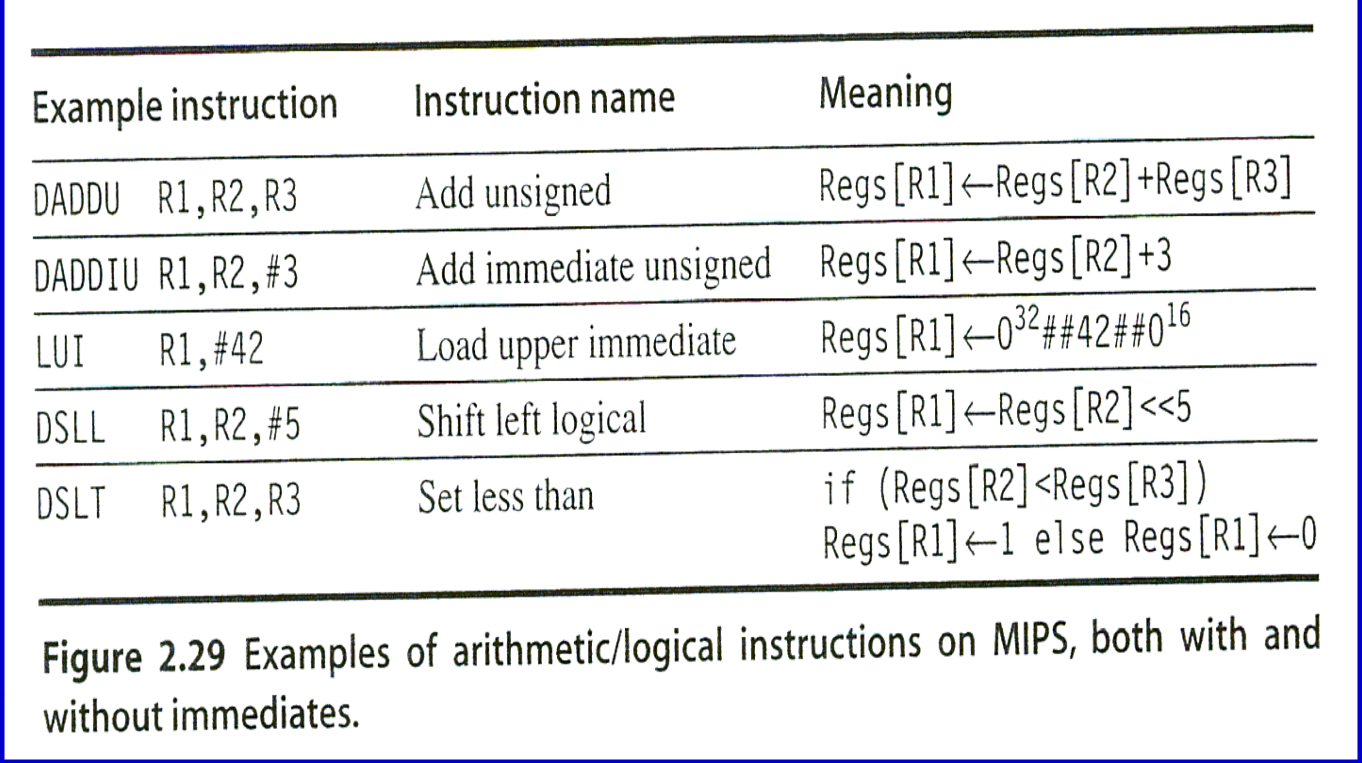
Load/Store Instructions



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.