Lecture 03 Instruction Set Principles

CSCE 513 Computer Architecture

Department of Computer Science and Engineering
Yonghong Yan
yanyh@cse.sc.edu
http://cse.sc.edu/~yanyh

Contents

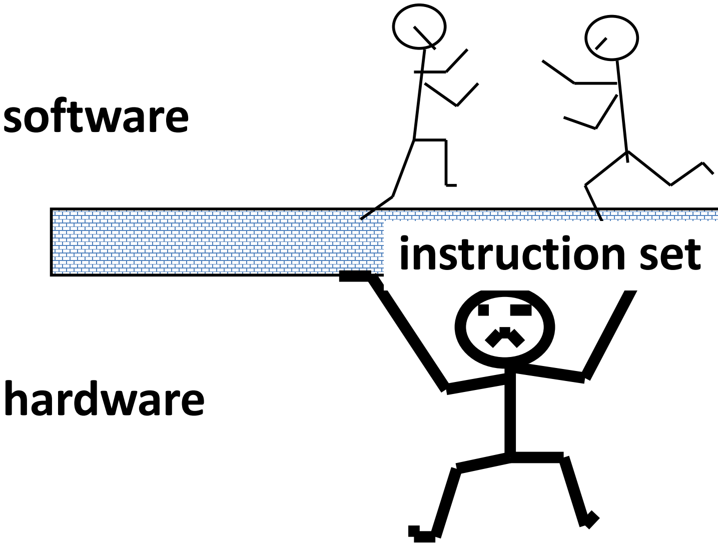
1. Introduction
2. Classifying Instruction Set Architectures
3. Memory Addressing
4. Type and Size of Operands
5. Operations in the Instruction Set
6. Instructions for Control Flow
7. Encoding an Instruction Set
8. Crosscutting Issues: The Role of Compilers
9. RISC-V ISA

- **Supplement (not covered)**
  - RISC vs CISC
  - Comparison of ISA
    - Appendix K

1 Introduction

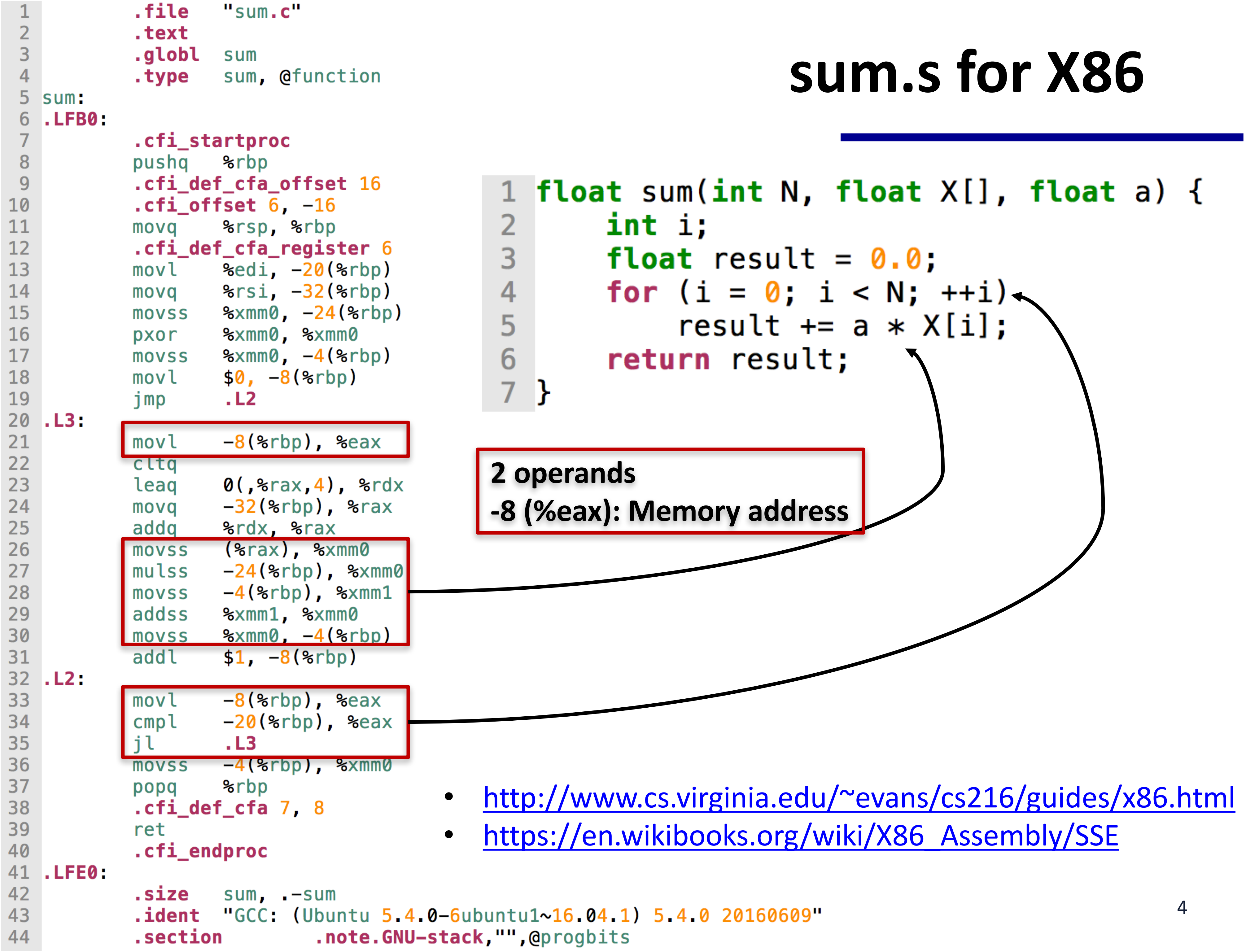
Instruction Set Architecture – the portion of the machine visible to the assembly level programmer or to the compiler writer

- To use the hardware of a computer, we must *speak* its language
- The words of a computer language are called *instructions*, and its vocabulary is called an *instruction set*



Instr. #	Operation+Operands
i	movl -4(%ebp), %eax
(i+1)	addl %eax, (%edx)
(i+2)	cmpl 8(%ebp), %eax
(i+3)	jl L5
:	
L5:	

sum.s for X86



```
        .file   "sum.c"
        .text
        .globl  sum
        .type   sum, @function
sum:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    %edi, -20(%rbp)
        movq    %rsi, -32(%rbp)
        movss   %xmm0, -24(%rbp)
        pxor    %xmm0, %xmm0
        movss   %xmm0, -4(%rbp)
        movl    $0, -8(%rbp)
        jmp     .L2
.L3:
        movl    -8(%rbp), %eax
        cltq
        leaq    0(,%rax,4), %rdx
        movq    -32(%rbp), %rax
        addq    %rdx, %rax
        movss   (%rax), %xmm0
        mulss   -24(%rbp), %xmm0
        movss   -4(%rbp), %xmm1
        addss   %xmm1, %xmm0
        movss   %xmm0, -4(%rbp)
        addl    $1, -8(%rbp)
.L2:
        movl    -8(%rbp), %eax
        cmpl    -20(%rbp), %eax
        jl      .L3
        movss   -4(%rbp), %xmm0
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   sum, .-sum
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.1) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
```

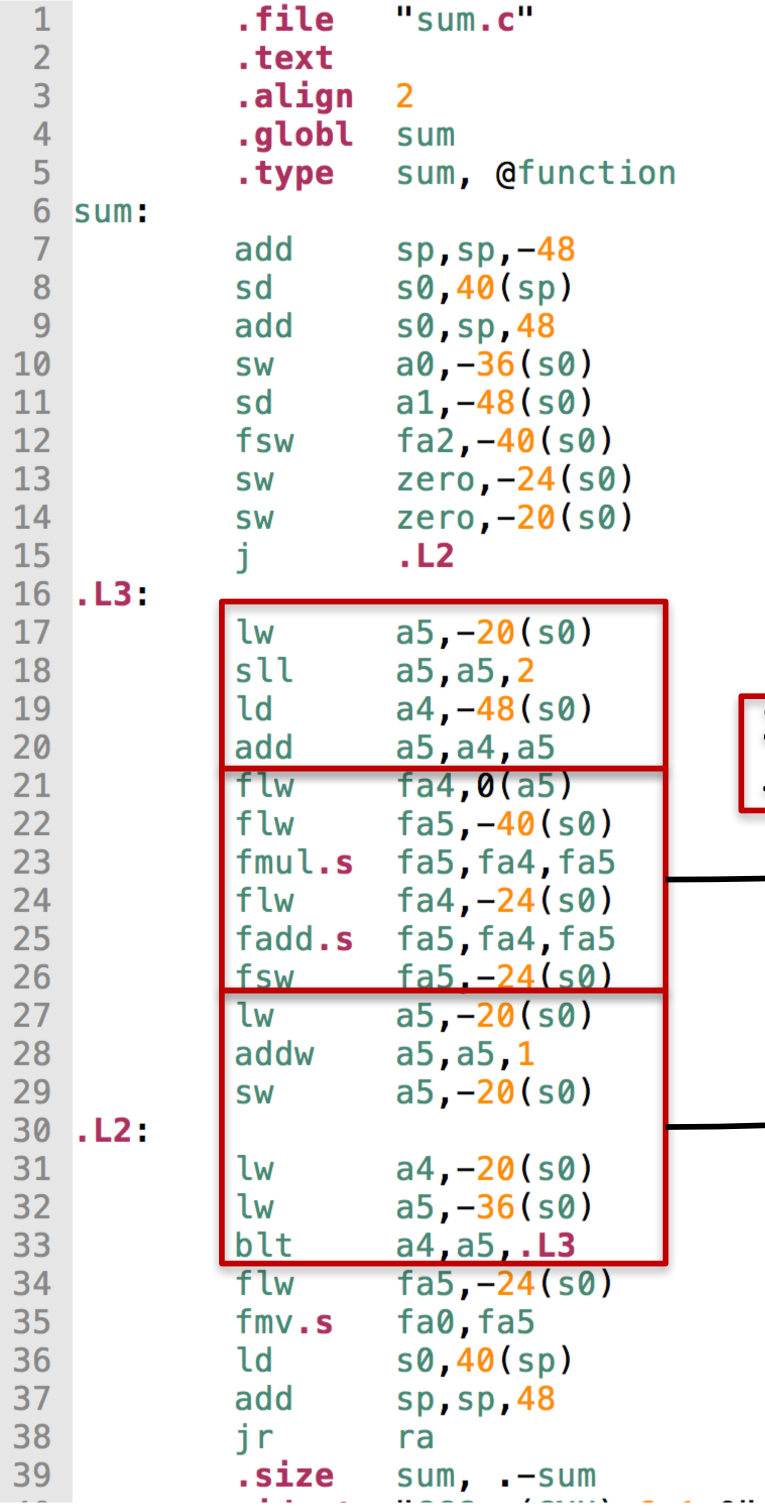
```
float sum(int N, float X[], float a) {
    int i;
    float result = 0.0;
    for (i = 0; i < N; ++i)
        result += a * X[i];
    return result;
}
```

2 operands

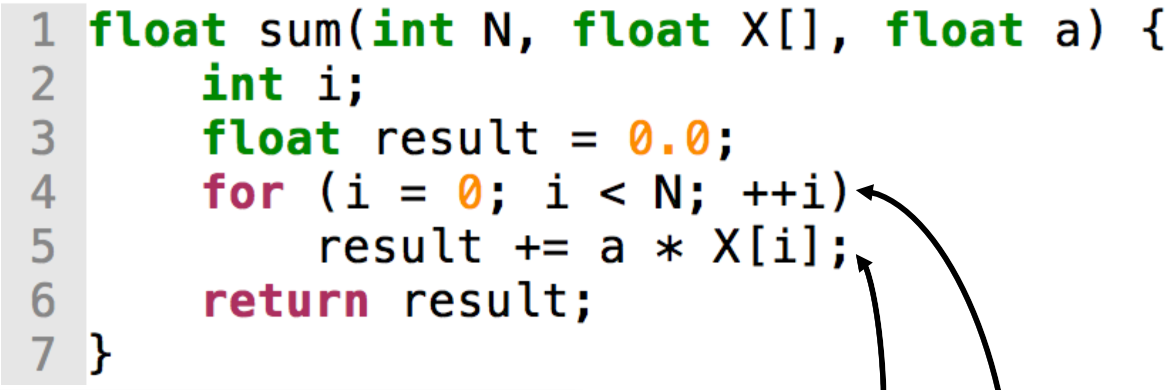
-8 (%eax): Memory address

- http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- https://en.wikibooks.org/wiki/X86_Assembly/SSE

sum.s for RISC-V



```
        .file   "sum.c"
        .text
        .align  2
        .globl  sum
        .type   sum, @function
sum:
        add     sp,sp,-48
        sd      s0,40(sp)
        add     s0,sp,48
        sw      a0,-36(s0)
        sd      a1,-48(s0)
        fsw     fa2,-40(s0)
        sw      zero,-24(s0)
        sw      zero,-20(s0)
        j       .L2
.L3:
        lw      a5,-20(s0)
        sll     a5,a5,2
        ld      a4,-48(s0)
        add     a5,a4,a5
        flw     fa4,0(a5)
        flw     fa5,-40(s0)
        fmul.s  fa5,fa4,fa5
        flw     fa4,-24(s0)
        fadd.s  fa5,fa4,fa5
        fsw     fa5,-24(s0)
        lw      a5,-20(s0)
        addw    a5,a5,1
        sw      a5,-20(s0)
.L2:
        lw      a4,-20(s0)
        lw      a5,-36(s0)
        blt     a4,a5,.L3
        flw     fa5,-24(s0)
        fmv.s   fa0,fa5
        ld      s0,40(sp)
        add     sp,sp,48
        jr      ra
        .size   sum, .-sum
```



```c
float sum(int N, float X[], float a) {
      int i;
      float result = 0.0;
      for (i = 0; i < N; ++i)
            result += a * X[i];
      return result;
}
```

2 or 3 operands

-20 (s0): Memory address

https://riscv.org/

ISA In Real

- A pdf document that defines the model/architecture/interface of the machine
  - X86 and Intel SDM: https://software.intel.com/en-us/articles/intel-sdm
    - Several thousands pages
  - RISC-V ISA Spec: https://riscv.org/specifications/
    - Latest version 2.2, 145 pages

- A specification that provides the ISA details

- Review Chapter 2 of the COD book

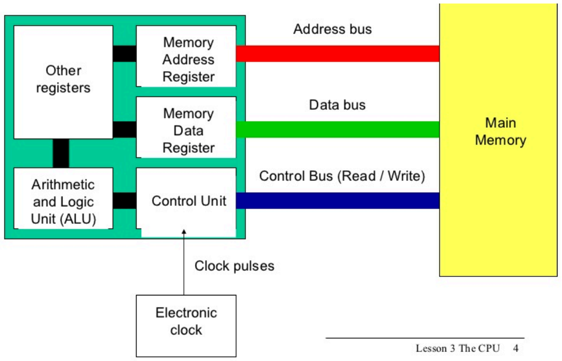
2 Classifying Instruction Set Architectures

Operand storage in CPU	Where are they other than memory
# explicit operands named per instruction	How many? Min, Max, Average
Addressing mode	How the effective address for an operand calculated? Can all use any mode?
Operations	What are the options for the opcode?
Type & size of operands	How is typing done? How is the size specified?

These choices critically affect number of instructions, CPI, and CPU cycle time

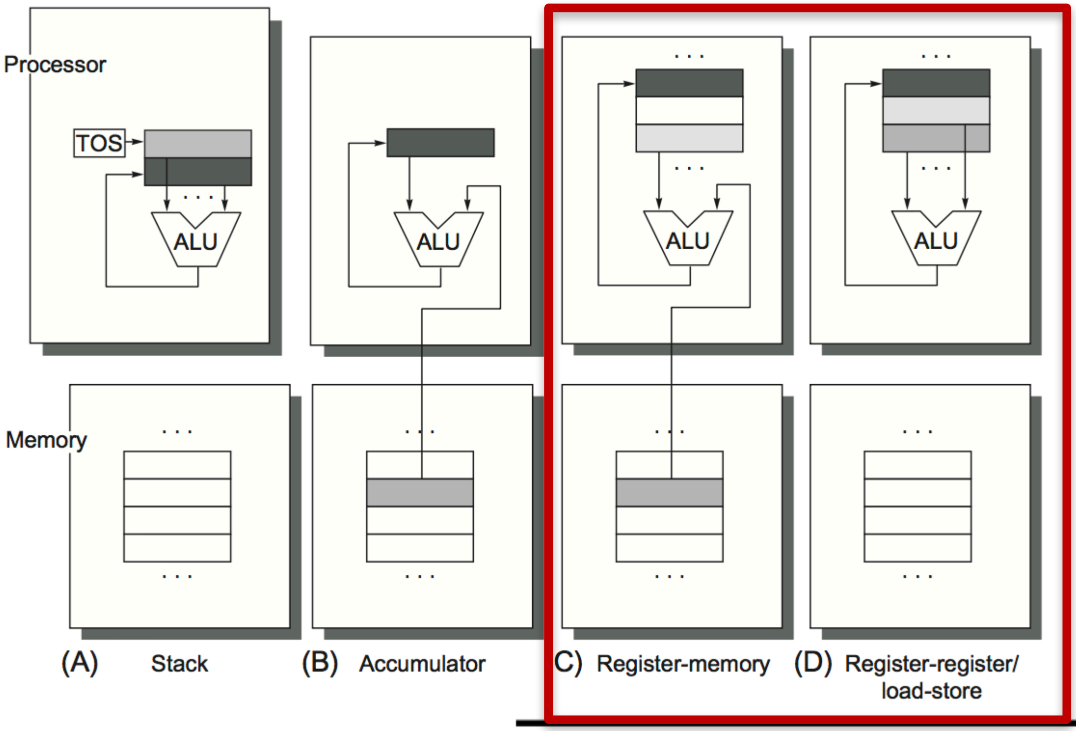
ISA Classification

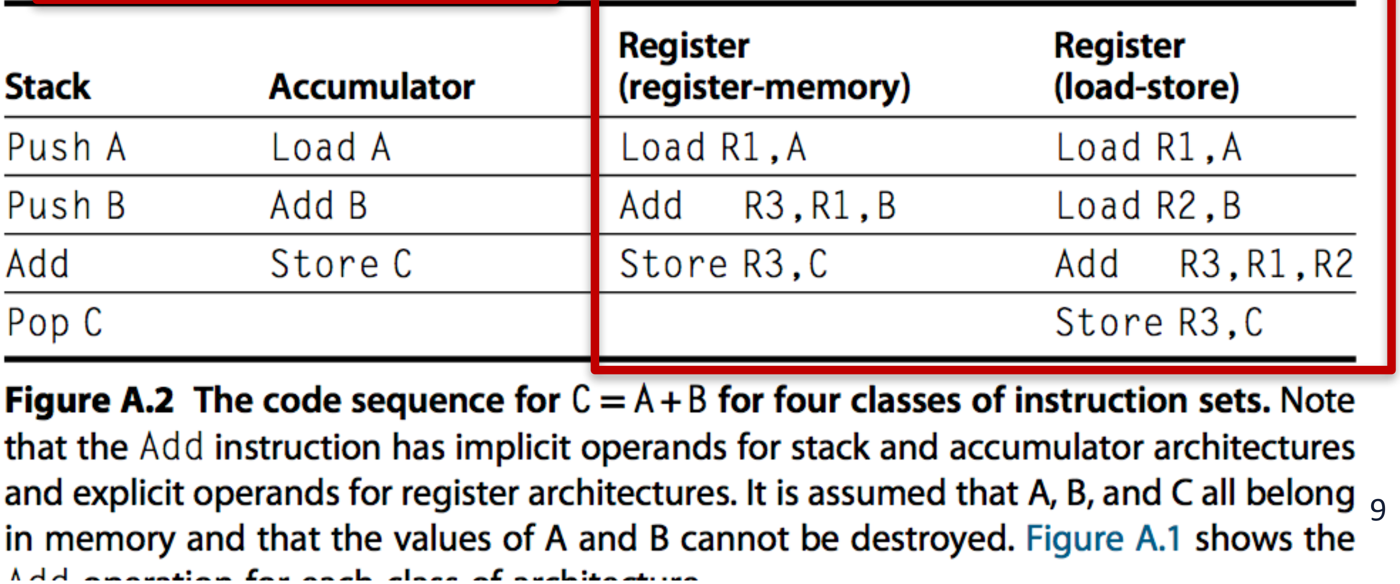
- Most basic differentiation: internal storage in a processor
  - Operands may be named **explicitly** or **implicitly**



- Major choices:
  1. In an **accumulator architecture** one operand is *implicitly* the accumulator => similar to calculator
  2. The operands in a **stack architecture** are *implicitly* on the top of the stack
  3. The **general-purpose register architectures** have only *explicit* operands – **either registers or memory location**

Four ISA Classes





Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Figure A.2 The code sequence for C = A + B for four classes of instruction sets. Note that the Add instruction has implicit operands for stack and accumulator architectures and explicit operands for register architectures. It is assumed that A, B, and C all belong in memory and that the values of A and B cannot be destroyed. Figure A.1 shows the Add operation for each class of architecture.

Register Machines

- How many registers are sufficient?
- General-purpose registers vs. special-purpose registers
  - *compiler flexibility and hand-optimization*
- Two major concerns for arithmetic and logical instructions (ALU)

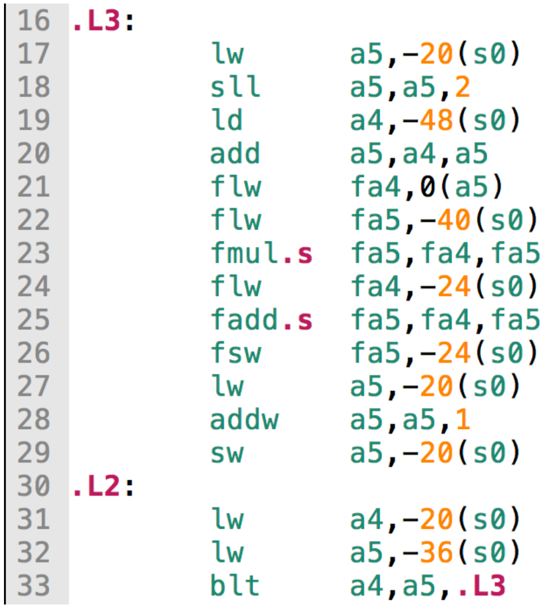
  1. Two or three operands
     - **X + Y ⇒ X**
     - **X + Y ⇒ Z**
  2. How many of the operands may be memory addresses (0 – 3)

Number of memory addresses	Maximum number of operands allowed	Type of Architecture	Examples
0	3	Load-Store	Alpha, ARM, MIPS, PowerPC, SPARC, SuperH, TM32
1	2	Register-Memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory – memory	VAX (also has 3 operand formats)
3	3	Memory - memory	VAX (also has 2 operand formats)

Hence, register architecture classification (# mem, # operands)

(0, 3): Register-Register (RISC)

- ALU is Register to Register – also known as
  - pure ***Reduced Instruction Set Computer (RISC)***



```
.L3:
        lw      a5,-20(s0)
        sll     a5,a5,2
        ld      a4,-48(s0)
        add     a5,a4,a5
        flw     fa4,0(a5)
        flw     fa5,-40(s0)
        fmul.s  fa5,fa4,fa5
        flw     fa4,-24(s0)
        fadd.s  fa5,fa4,fa5
        fsw     fa5,-24(s0)
        lw      a5,-20(s0)
        addw    a5,a5,1
        sw      a5,-20(s0)
.L2:
        lw      a4,-20(s0)
        lw      a5,-36(s0)
        blt     a4,a5,.L3
```

- Advantages
  - Simple fixed length instruction encoding
  - Decode is simple since instruction types are small
  - Simple code generation model
  - Instruction CPI tends to be very uniform
    - Except for memory instructions of course
      - but there are only 2 of them - load and store

- Disadvantages
  - Instruction count tends to be higher
  - Some instructions are short - wasting instruction word bits

(1, 2): Register-Memory (CISC, X86)

- Evolved RISC and also old CISC
  - new RISC machines capable of doing speculative loads
  - predicated and/or deferred loads are also possible

- Advantages
  - data access to ALU immediate without loading first
  - instruction format is relatively simple to encode
  - code density is improved over Register (0, 3) model

- Disadvantages
  - operands are not equivalent - source operand may be destroyed
  - need for memory address field may limit # of registers
  - CPI will vary
    - if memory target is in L0 cache then not so bad
    - if not - life gets miserable

(2, 2) or (3, 3): Memory-Memory

Not used today

- True and most complex CISC model
  - currently extinct and likely to remain so
  - more complex memory actions are likely to appear but not
  - directly linked to the ALU

- Advantages
  - most compact code
  - doesn’t waste registers for temporary values
    - good idea for use once data - e.g. streaming media

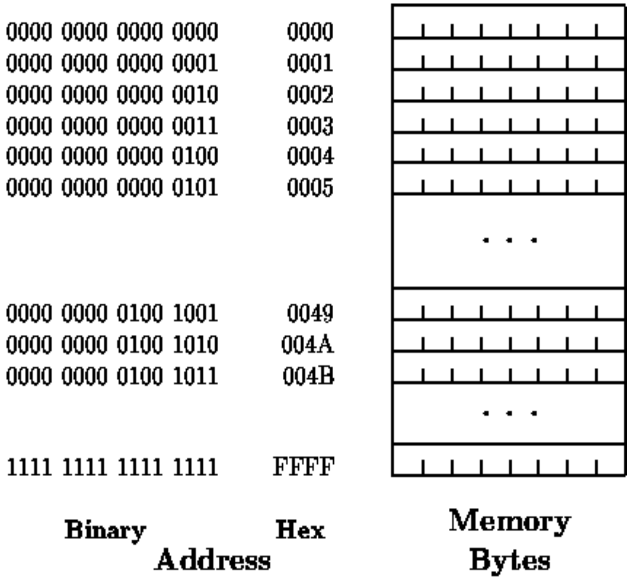
- Disadvantages
  - large variation in instruction size - may need a shoe-horn
  - large variation in CPI - i.e. work per instruction
  - exacerbates the infamous memory bottleneck
    - register file reduces memory accesses if reused

Summary: Tradeoffs for the ISA Classes

Type	Advantages	Disadvantages
Register-register (0,3)	**Simple, fixed length instruction encoding. Simple code generation model. Instructions take similar numbers of clocks to execute.**	**Higher instruction count than architectures with memory references in the 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 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. Does not 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)

3 Memory Addressing

- Objects have ***byte addresses***
  - the number of bytes counted from the beginning of memory
- *Object Length*:
  - bytes (8 bits), half words (16 bits),
  - words (32 bits), and double words (64 bits).
  - The type is implied in opcode, e.g.,
    - LDB – load byte
    - LDW – load word, etc



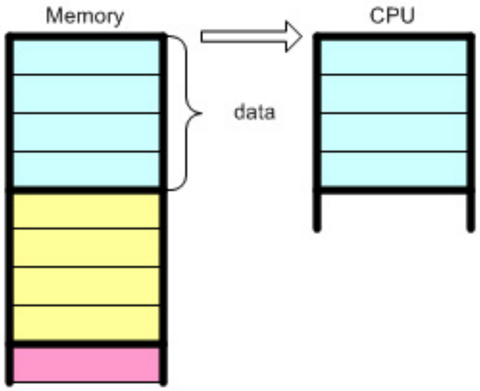
- *Byte Ordering*
  - Little Endian: puts the byte whose address is xx00 at the least significant position in the word. (7,6,5,4,3,2,1,0)
  - Big Endian: puts the byte whose address is xx00 at the most significant position in the word. (0,1,2,3,4,5,6,7)
    - Problem occurs when exchanging data among machines with different orderings

Interpreting Memory Addresses

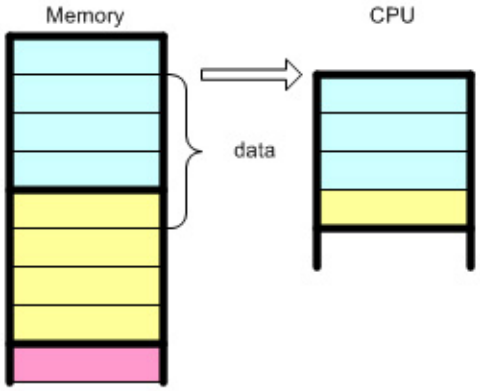
- Alignment Issues
  - Accesses to objects larger than a byte must be aligned.
    - An access to an object of size s bytes at byte address A is aligned if A mod s = 0.
  - Misalignment causes hardware complications
    - since memory is typically aligned on a word or a double-word boundary
    - Misalignment typically results in an alignment fault that must be handled by the OS

- Hence
  - byte address is anything - never misaligned
  - half word - even addresses - low order address bit = 0 (XXXXXXX0) else trap
  - word - low order 2 address bits = 0 (XXXXXX00) else trap
  - double word - low order 3 address bits = 0 (XXXXX000) else trap

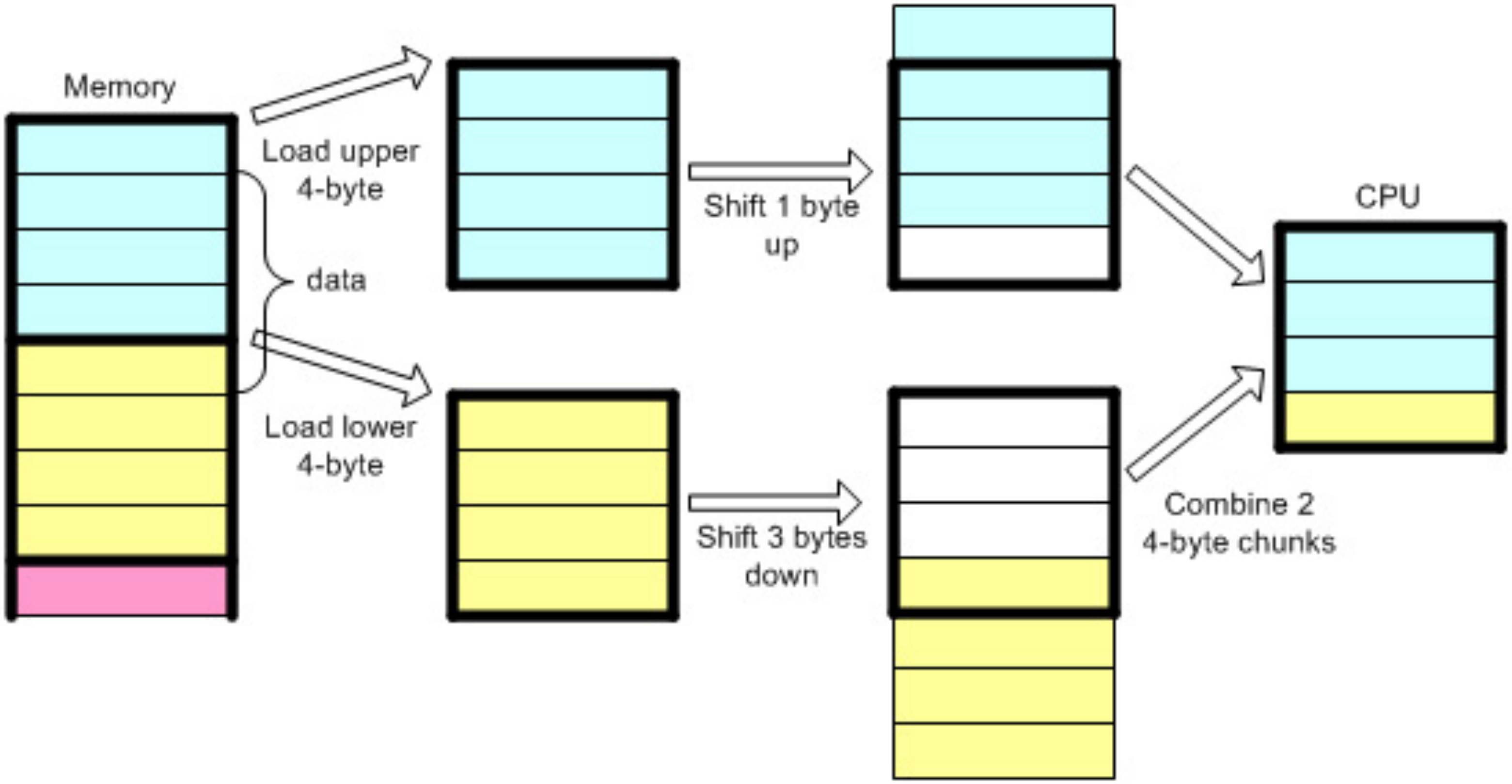
Memory Alignment



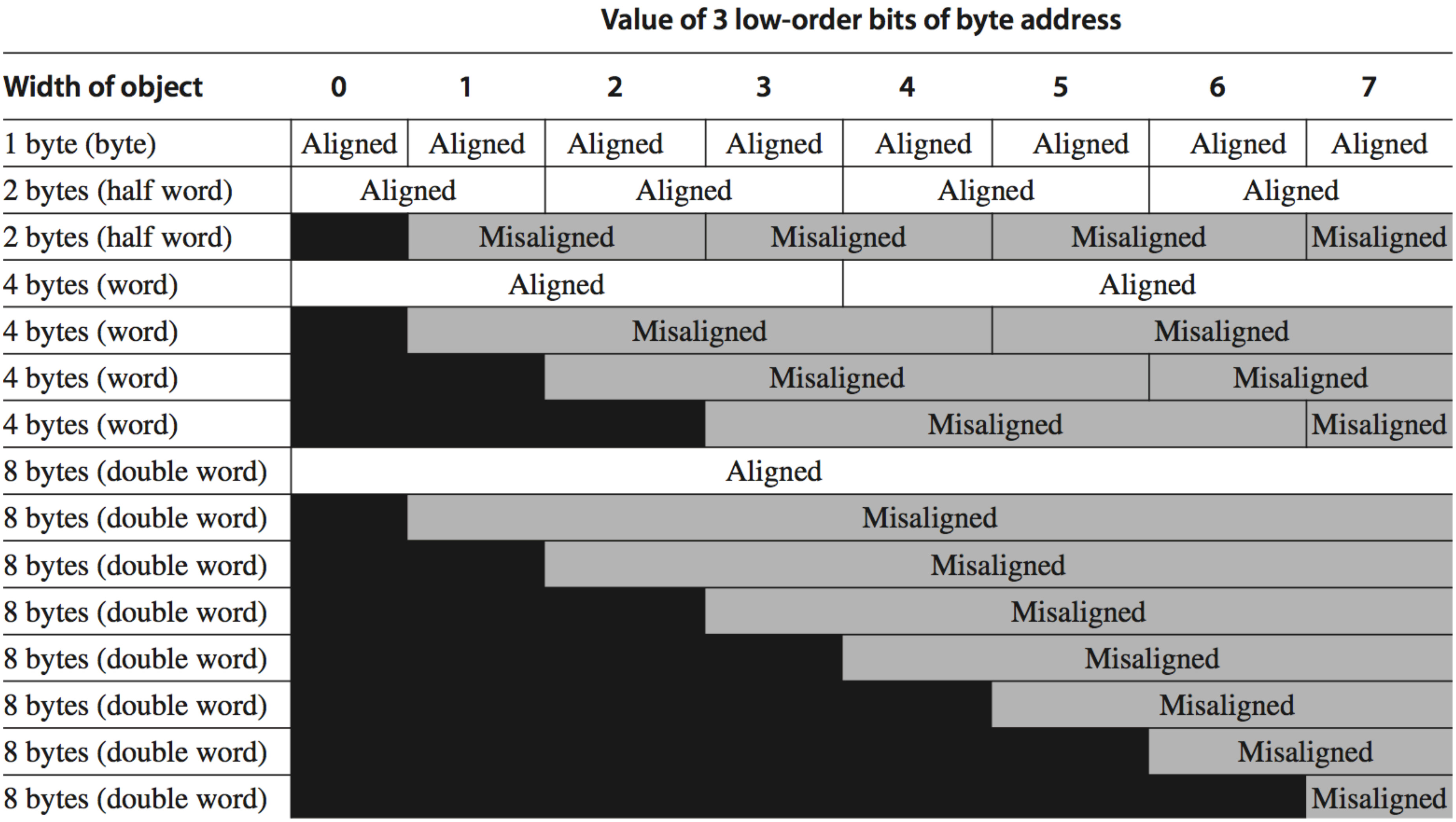
4-byte memory access for aligned data



4-byte memory access for misaligned data



Aligned/Misaligned Addresses



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

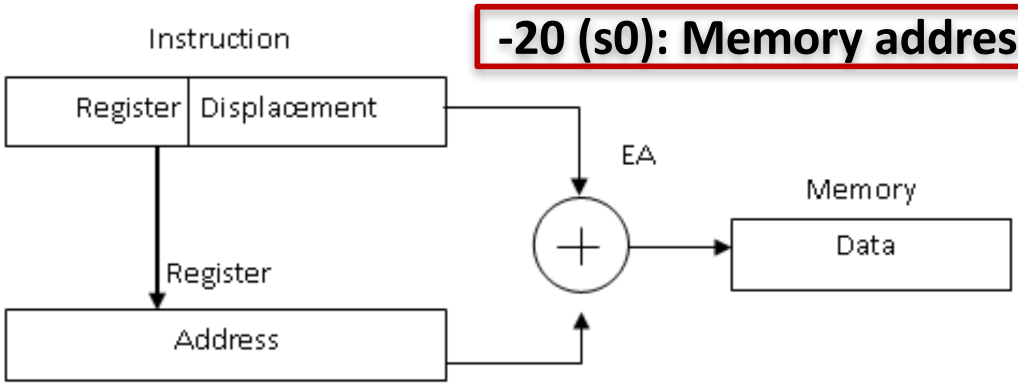
Figure A.5 Aligned and misaligned addresses of byte, half-word, word, and double-word objects for byte-addressed computers. For each misaligned example some objects require two memory accesses to complete. Every aligned object can always complete in one memory access, as long as the memory is as wide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

Addressing Modes

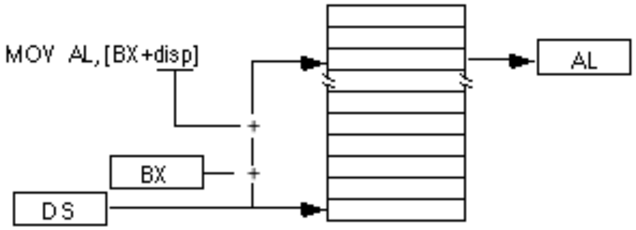
- How architecture specify the **effective address of** an object?
  - ***Effective address***: the actual memory address specified by the addressing mode.
    - E.g. Mem[R[R1]] *refers to the contents of the memory location whose location is given by the contents of register 1 (R1).*

-20 (s0): Memory address

- **Addressing Modes:**
  - Register.
  - Immediate
  - Displacement
  - Register indirect,……..

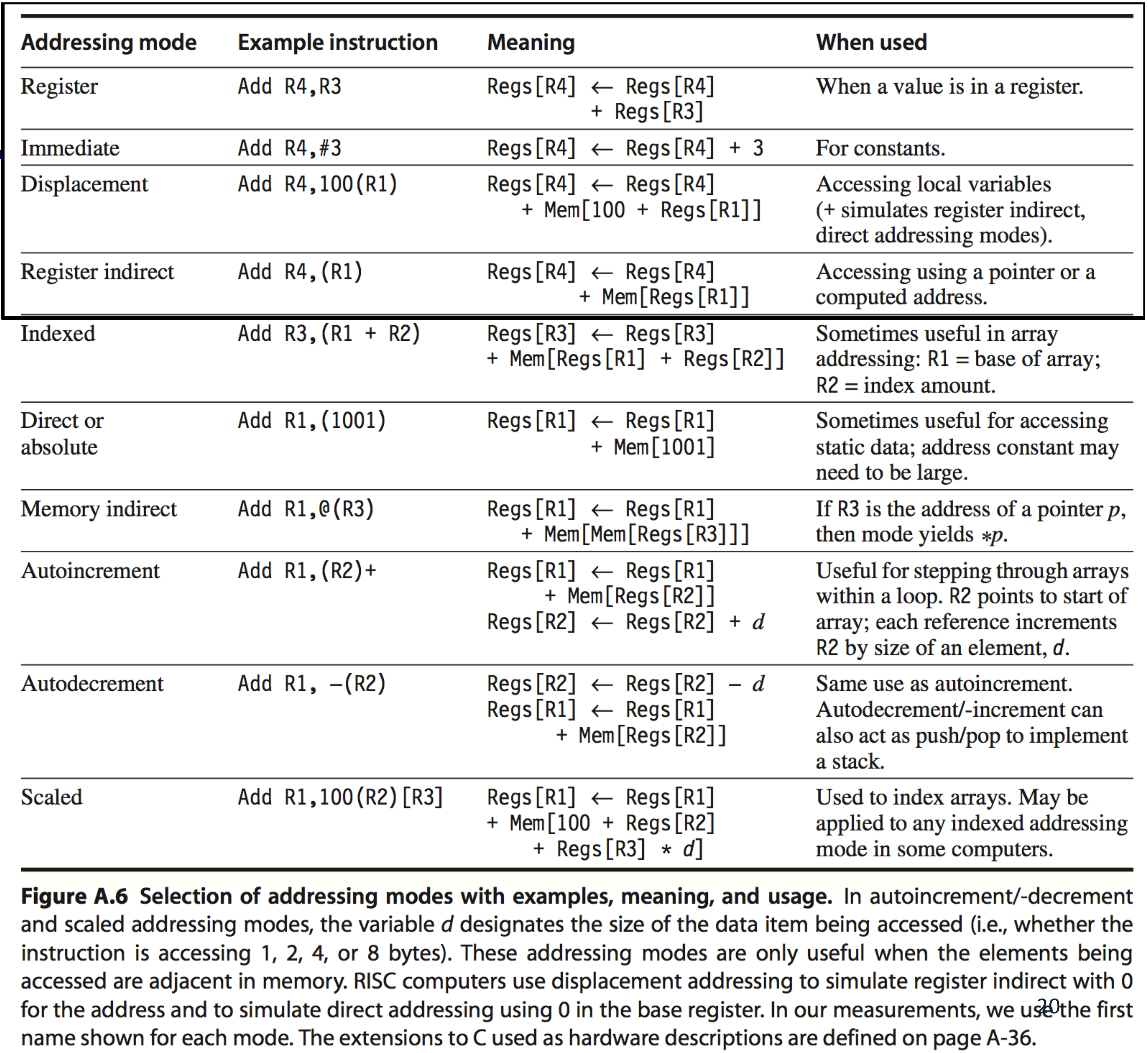


(a) Register Relative Addressing Mode



(b) Register Relative Addressing Mode

Address 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 indirect, direct addressing modes).
Register indirect	`Add R4,(R1)`	`Regs[R4] ← Regs[R4] + Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
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*, then mode yields *p.
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.

Figure A.6 Selection of addressing modes with examples, meaning, and usage. In autoincrement/-decrement and scaled addressing modes, the variable *d* designates the size of the data item being accessed (i.e., whether the instruction is accessing 1, 2, 4, or 8 bytes). These addressing modes are only useful when the elements being accessed are adjacent in memory. RISC computers use displacement addressing to simulate register indirect with 0 for the address and to simulate direct addressing using 0 in the base register. In our measurements, we use the first name shown for each mode. The extensions to C used as hardware descriptions are defined on page A-36.

Addressing Mode Impacts

- Instruction counts
- Architecture Complexity
- CPI

Summary of Use of Memory Addressing Modes

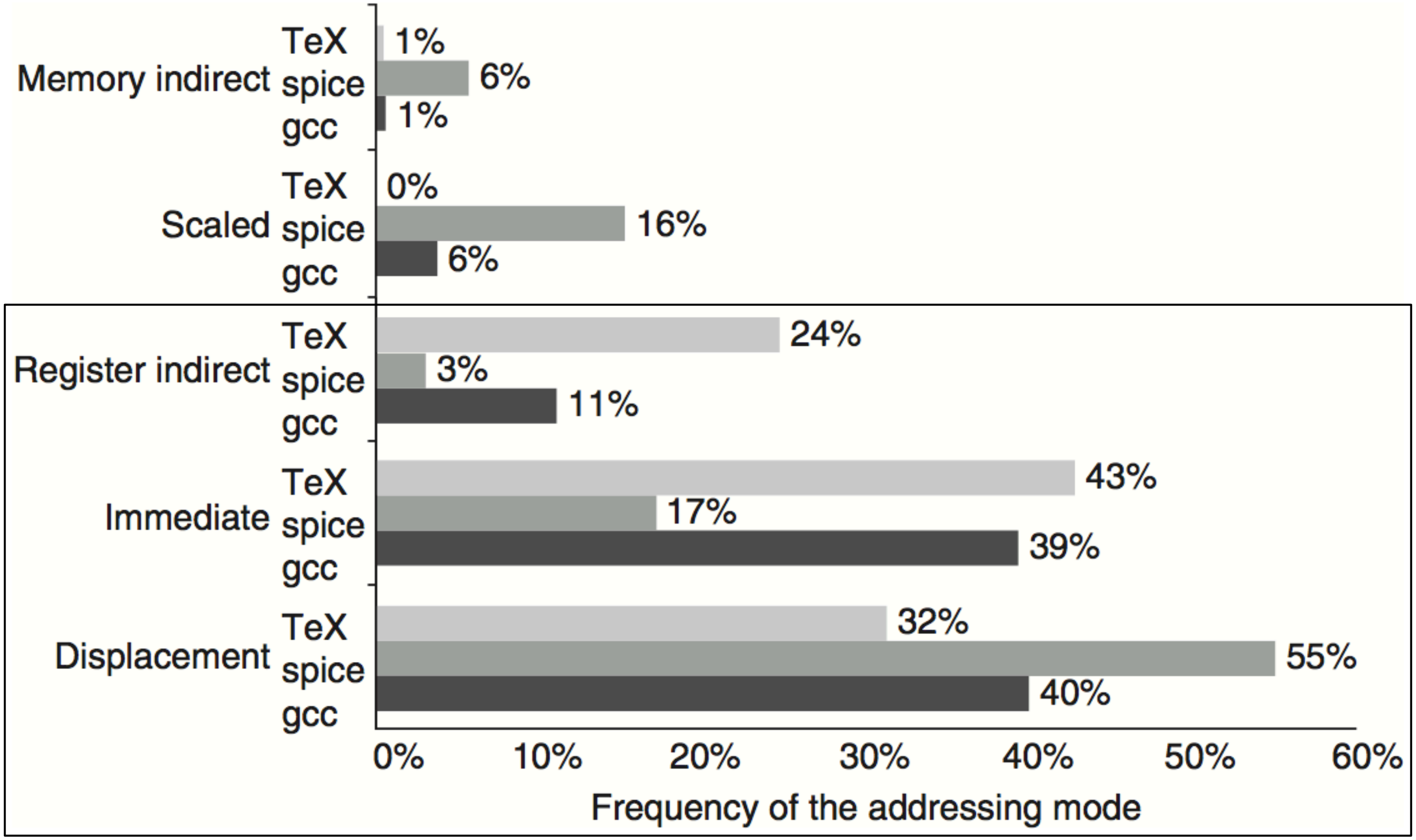


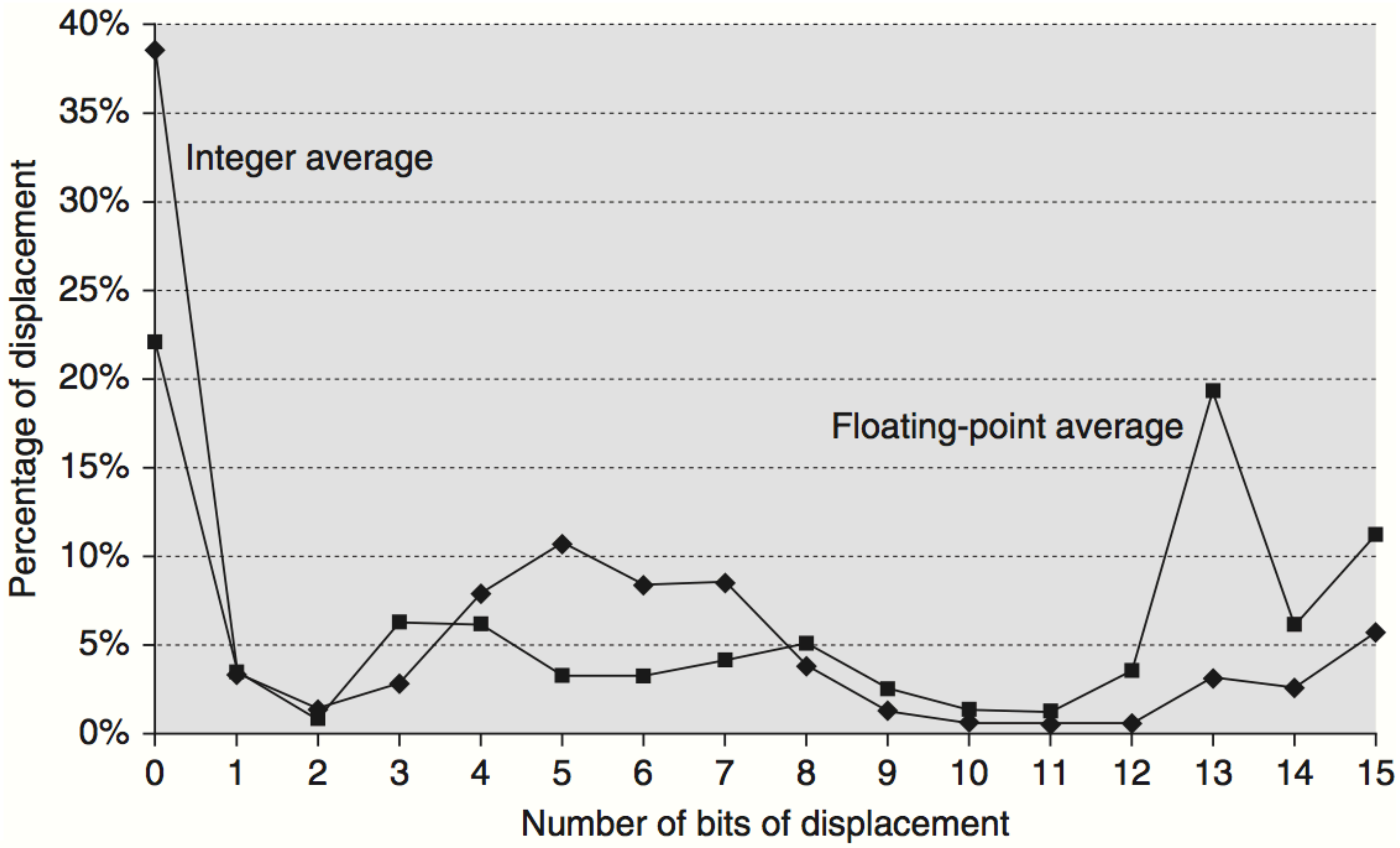
Figure A.7 Summary of use of memory addressing modes (including immediates). These major addressing modes account for all but a few percent (0% to 3%) of the memory accesses. Register modes, which are not counted, account for one-half of the operand references, while memory addressing modes (including immediate) account for the other half. Of course, the compiler affects what addressing modes are used; see Section A.8. The memory indirect mode on the VAX can use displacement, autoincrement, or autodecrement to form the initial memory address; in these programs, almost

Displacement Values are Widely Distributed

```
Add R4,100(R1)          Regs[R4] ← Regs[R4]
                           + Mem[100 + Regs[R1]]
```

Impact instruction length

Accessing local variables (+ simulates register indirect, direct addressing modes).



Displacement Addressing Mode

- **Benchmarks show**
  - **12 bits of displacement would capture about 75% of the full 32-bit displacements**
  - **16 bits should capture about 99%**

- **Remember:**
  - **optimize for the common case. Hence, the choice is at least 12-16 bits**

- For addresses that do fit in displacement size:

```
Add     R4, 10000 (R0)
```

- For addresses that don’t fit in displacement size, the compiler must do the following:

```
Load     R1, 1000000
Add     R1, R0
Add     R4, 0 (R1)
```

Immediate Addressing Mode

- Used where we want to get to a numerical value in an instruction
- **Around 25% of the operations have an immediate operand**

At high level:

```
a  =  b + 3;


if ( a > 17 )


goto    Addr
```

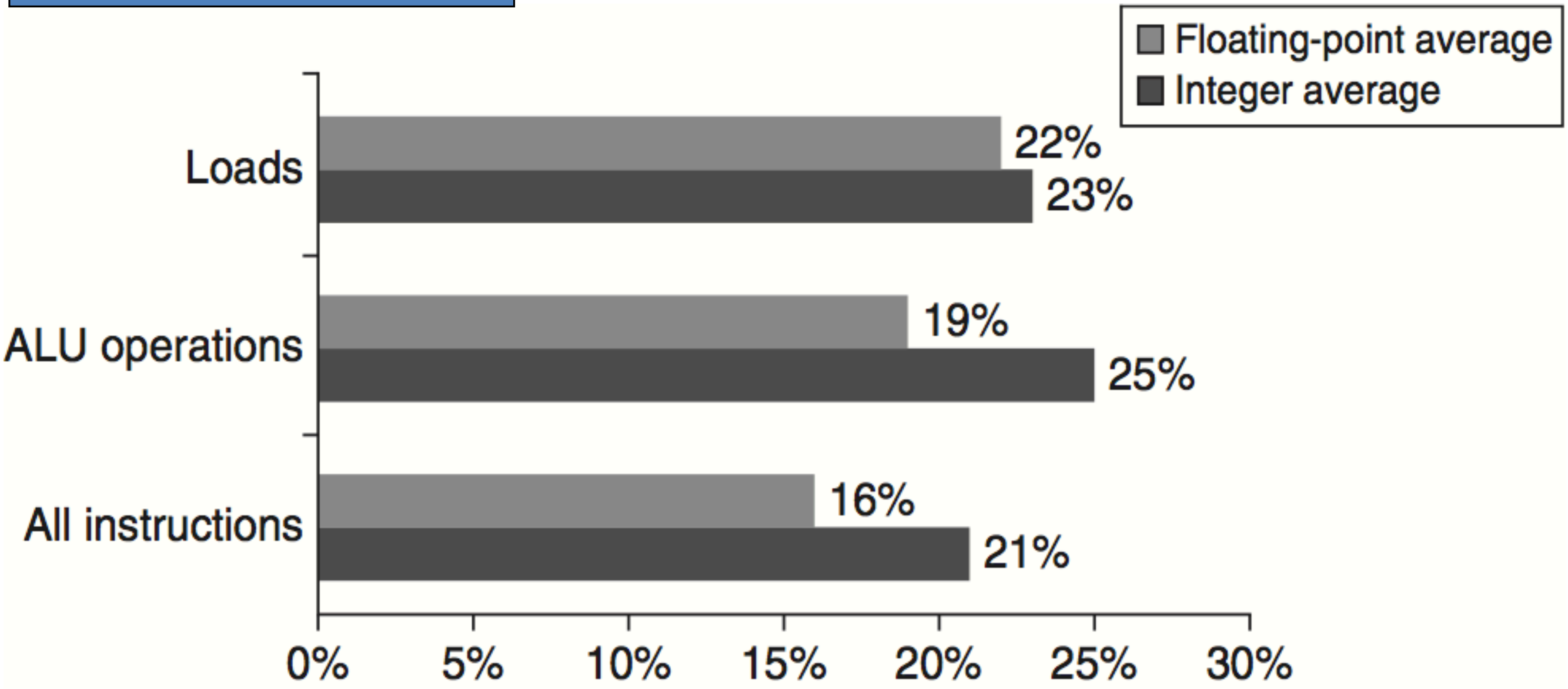
At Assembler level:

```
Load    R2, #3
Add      R0,  R1,  R2

Load          R2, #17
CMPBGT   R1, R2

Load          R1,  Address
Jump          (R1)
```

About 25% of data transfer and ALU operations have an immediate operand

Add R4,#3 Regs[R4] ← Regs[R4] + 3 For constants.

Impact instruction length

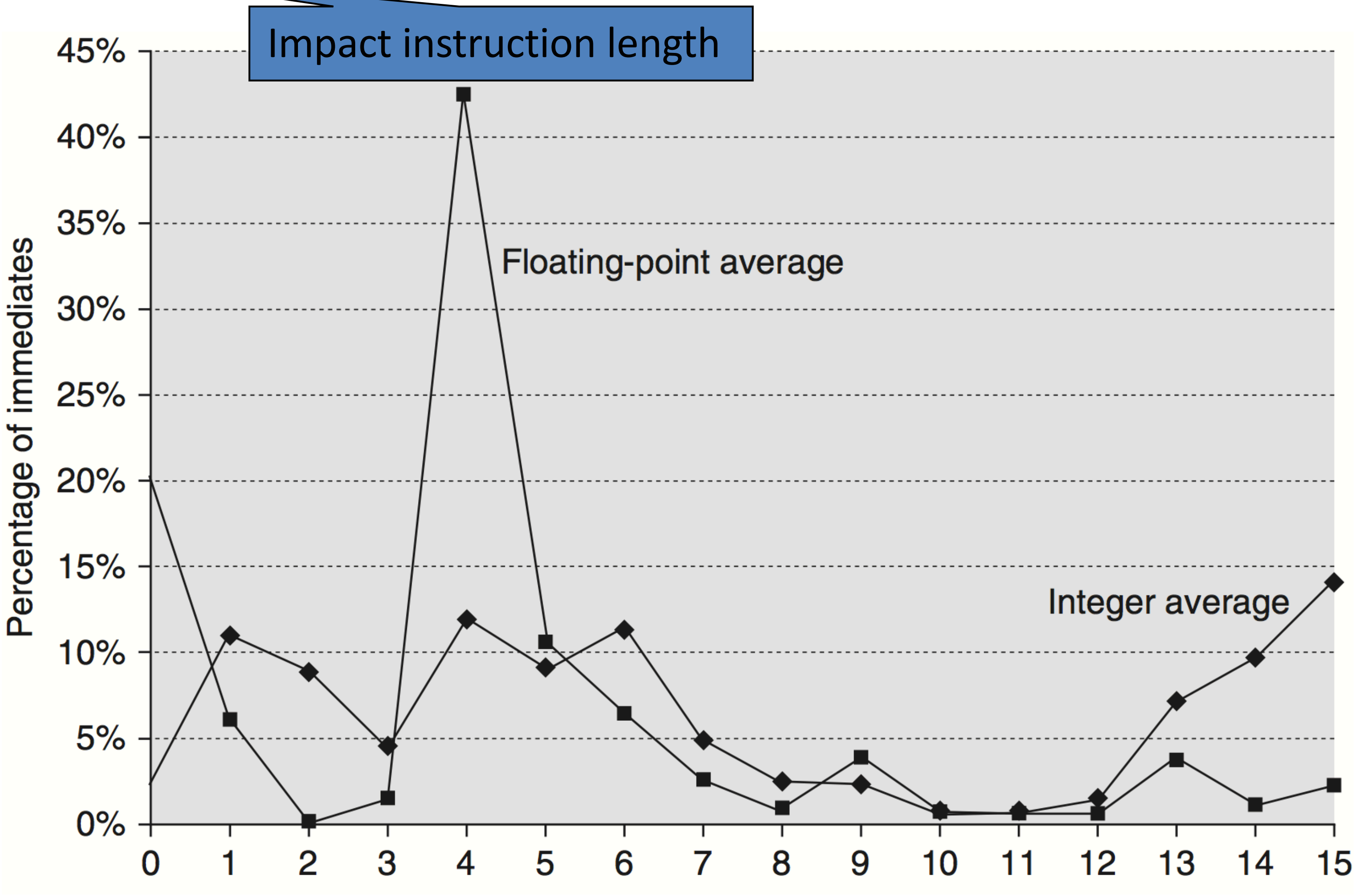


ıre A.9 About one-quarter of data transfers and ALU operations have an imme-e operand. The bottom bars show that integer programs use immediates in about -fifth of the instructions, while floating-point programs use immediates in about

Number of Bits for Immediate

- **16 bits would capture about 80% and 8 bits about 50%.**

Add R4,#3 Regs[R4] ← Regs[R4] + 3 For constant



Impact instruction length

Summary: Memory Addressing

- A new architecture expected to support at least: **displacement, immediate, and register indirect**
  - represent 75% to 99% of the addressing modes
- The size of the address for displacement mode to be at least **12-16** bits
  - capture 75% to 99% of the displacements
- The size of the immediate field to be at least **8-16** bits
  - capture 50% to 80% of the immediates

Processors rely on compilers to generate codes using those addressing mode

4 Type And Size of Operands

How is the type of an operand designated?

- The type of the operand is usually encoded in the <u>*opcode*</u>
  - e.g., LDB – load byte; LDW – load word
- Common operand types: (imply their sizes)

  Character (8 bits or 1 byte)

  Half word (16 bits or 2 bytes)

  Word (32 bits or 4 bytes)

  Double word (64 bits or 8 bytes)

  Single precision floating point (4 bytes or 1 word)

  Double precision floating point (8 bytes or 2 words)

  ✓ *Characters are almost always in ASCII*

  ✓ *16-bit Unicode (used in Java) is gaining popularity*

  ✓ *Integers are two's complement binary*

  ✓ *Floating points follow the IEEE standard 754*
- Some architectures support *packed decimal*: 4 bits are used to encode the values 0-9; 2 decimal digits are packed into each byte

Distribution of Data Accesses by Size

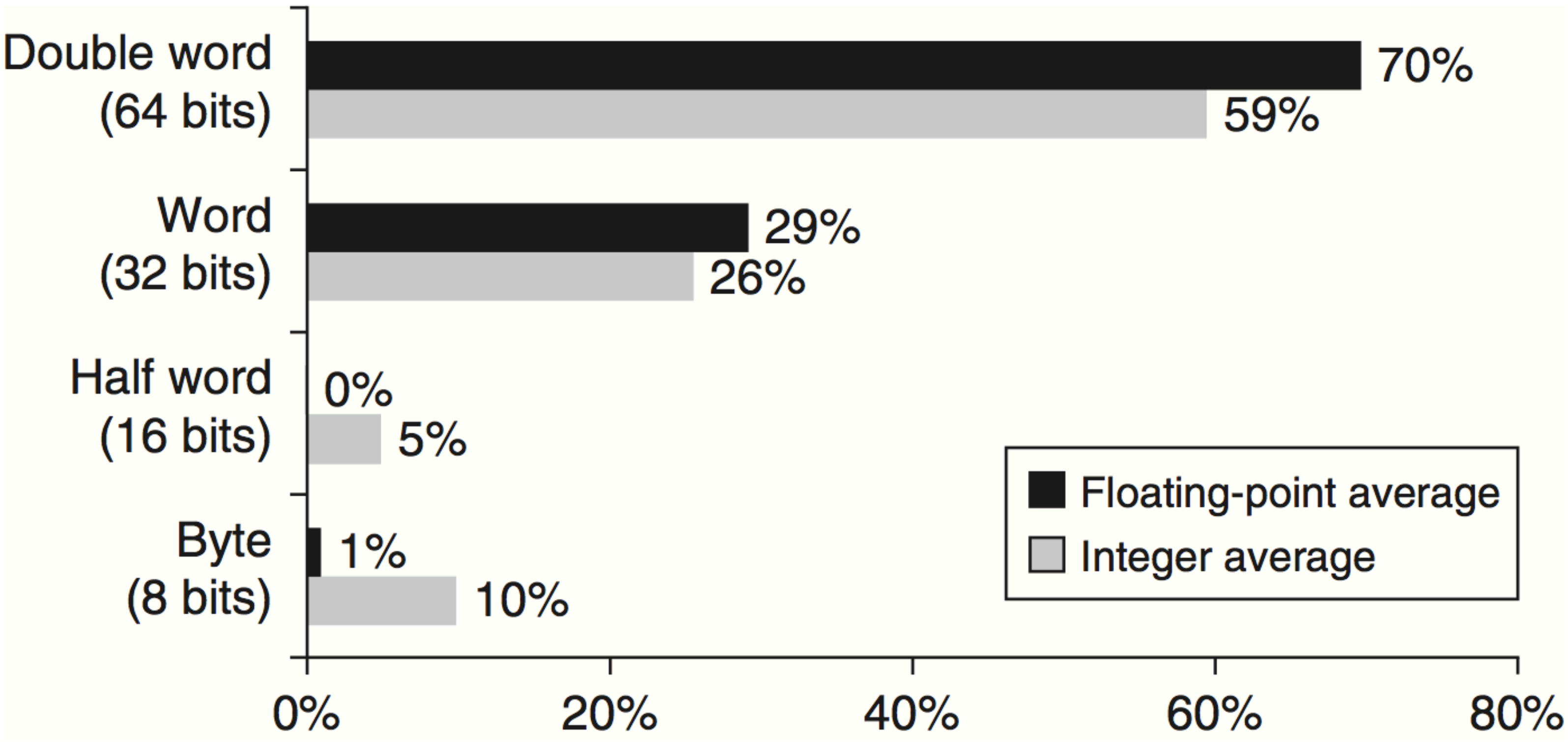
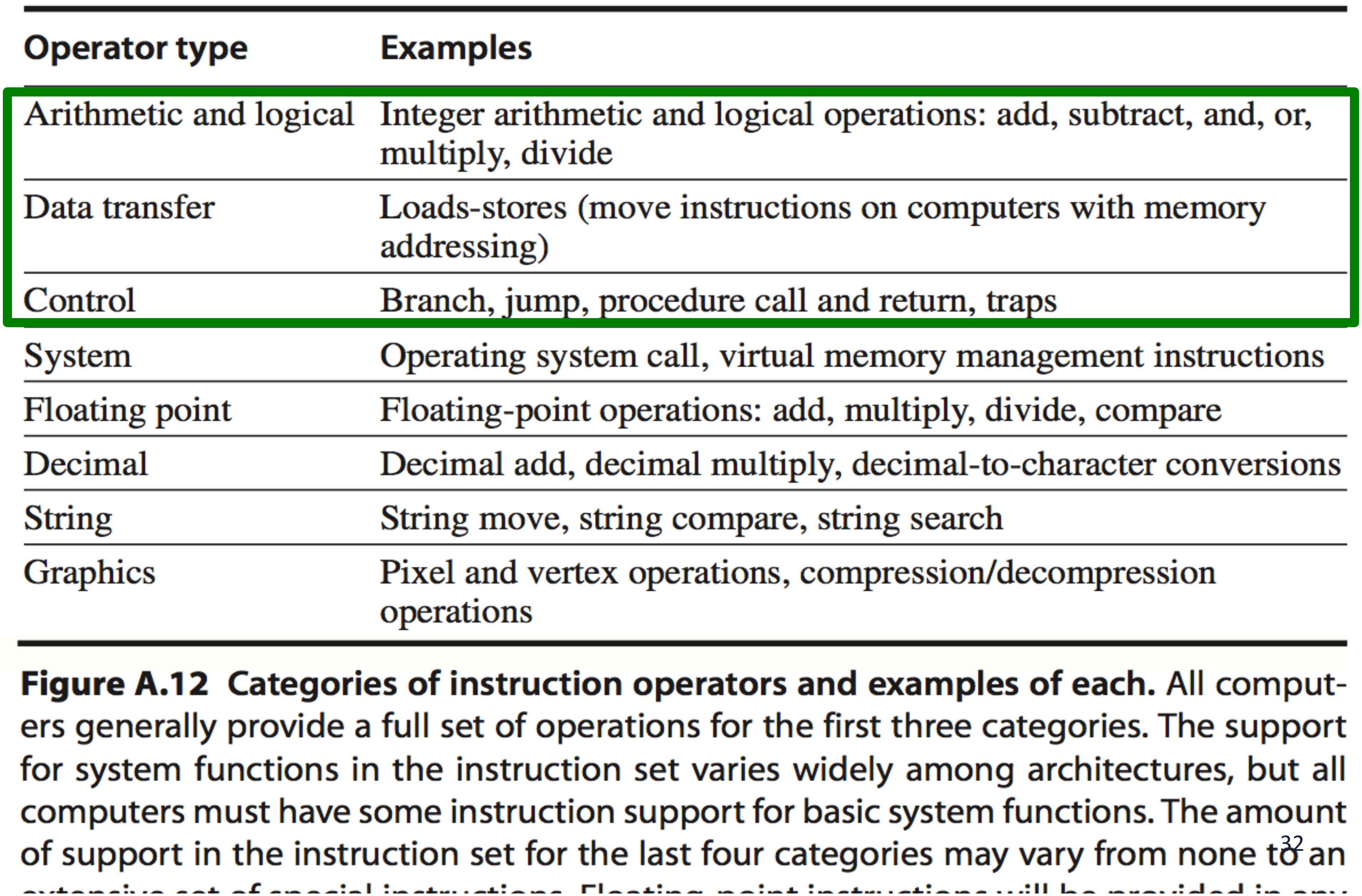


Figure A.11 Distribution of data accesses by size for the benchmark programs. The double-word data type is used for double-precision floating point in floating-point programs and for addresses, since the computer uses 64-bit addresses. On a 32-bit address computer the 64-bit addresses would be replaced by 32-bit addresses, and so almost all double-word accesses in integer programs would become single-word accesses.

Summary: Type and Size of operands

- 32-architecture supports 8-, 16-, and 32-bit integers, 32-bit and 64-bit IEEE 754 floating-point data.
- A new 64-bit address architecture supports 64-bit integers
- Media processor and DSPs need wider accumulating registers for accuracy.

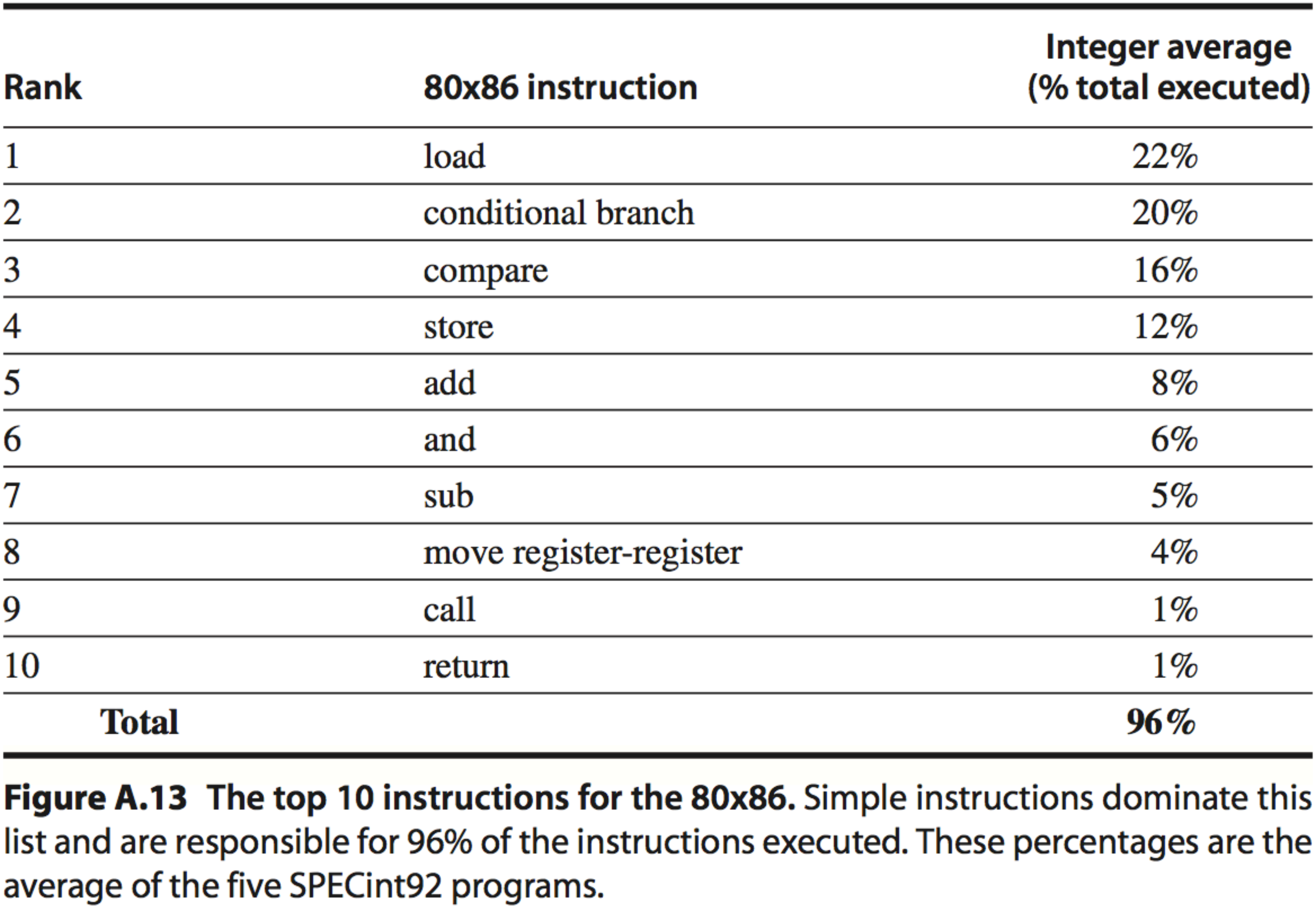
5 Operations in the Instruction Set



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

Figure A.12 Categories of instruction operators and examples of each. All computers generally provide a full set of operations for the first three categories. The support for system functions in the instruction set varies widely among architectures, but all computers must have some instruction support for basic system functions. The amount of support in the instruction set for the last four categories may vary from none to an extensive set of special instructions. Floating-point instructions will be provided in any

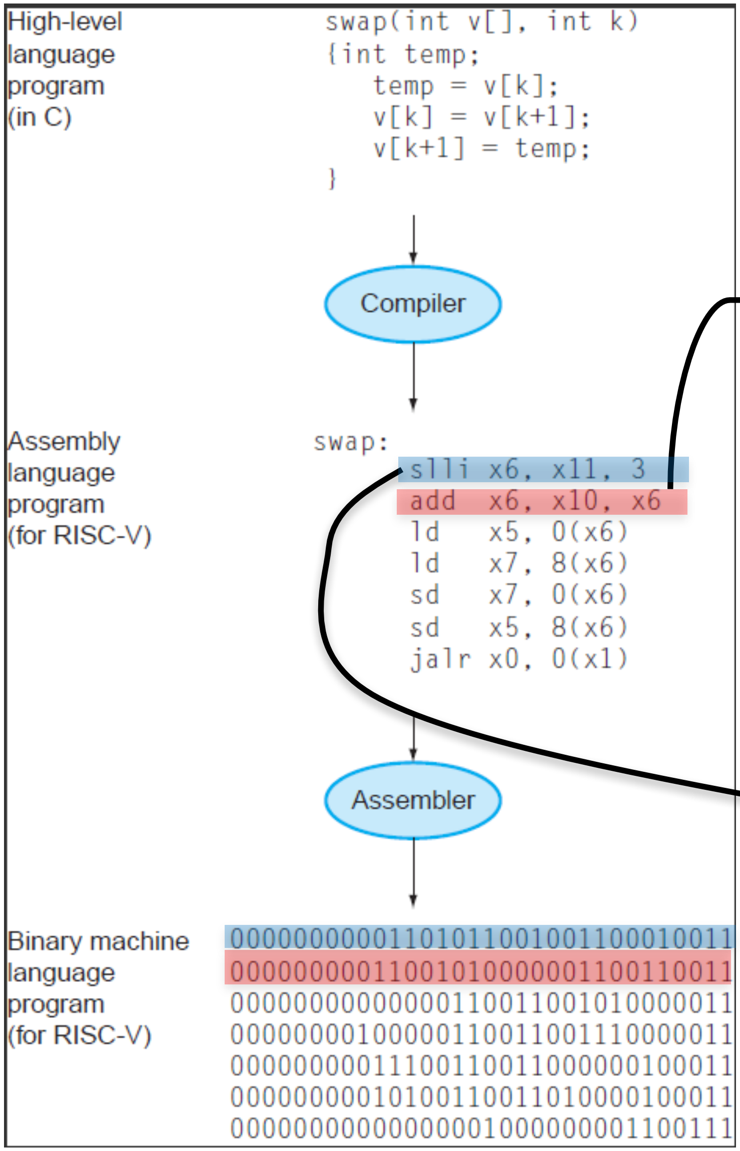
Top 10 Instructions for 80x86



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%**

Figure A.13 The top 10 instructions for the 80x86. Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.

Instruction Encoding



High-level language program (in C)
Compiler
Assembly language program (for RISC-V)
Assembler
Binary machine language program (for RISC-V)

```
swap(int v[], int k)
{int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

```
swap:
    slli x6, x11, 3
    add  x6, x10, x6
    ld   x5, 0(x6)
    ld   x7, 8(x6)
    sd   x7, 0(x6)
    sd   x5, 8(x6)
    jalr x0, 0(x1)
```

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000110011001010000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```

- RISC-V R-format instruction

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

add x6, x10, x6

0	6	10	0	6	51

0000000	00110	01010	000	00110	0110011

0000 0000 0110 0101 0000 0011 0011 0011_{two} = 00650333_{16}

- RISC-V I-format instruction

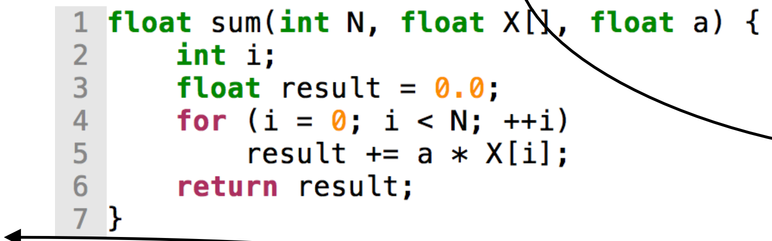
immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

Immediate arithmetic and load instructions

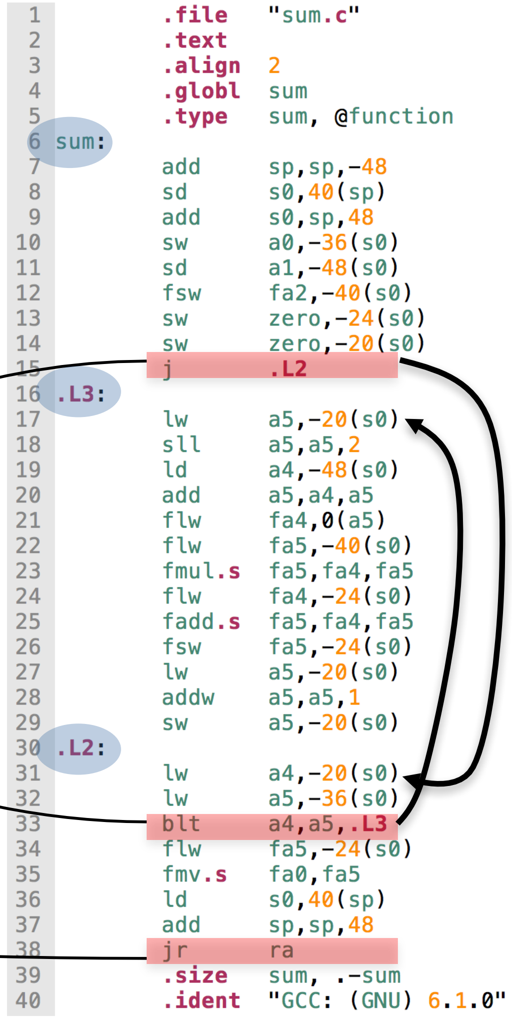
- rs1: source or base address register number
- immediate: constant operand, or offset added to base address
  - 2s-complement, sign extended

6 Instructions for Control Flow

- Control instructions change the flow of control:
  - instead of executing the next instruction, the program branches to the address specified in the branching instructions
- They break the pipeline
  - Difficult to optimize out
  - AND they are frequent
- Four types of control instructions
  - **Conditional branches**
    - **if…else, for/while, switch/case, …**
  - **Jumps – unconditional transfer**
    - **goto**
  - **Procedure calls**
    - **foo()**
  - **Procedure returns**
    - **return**

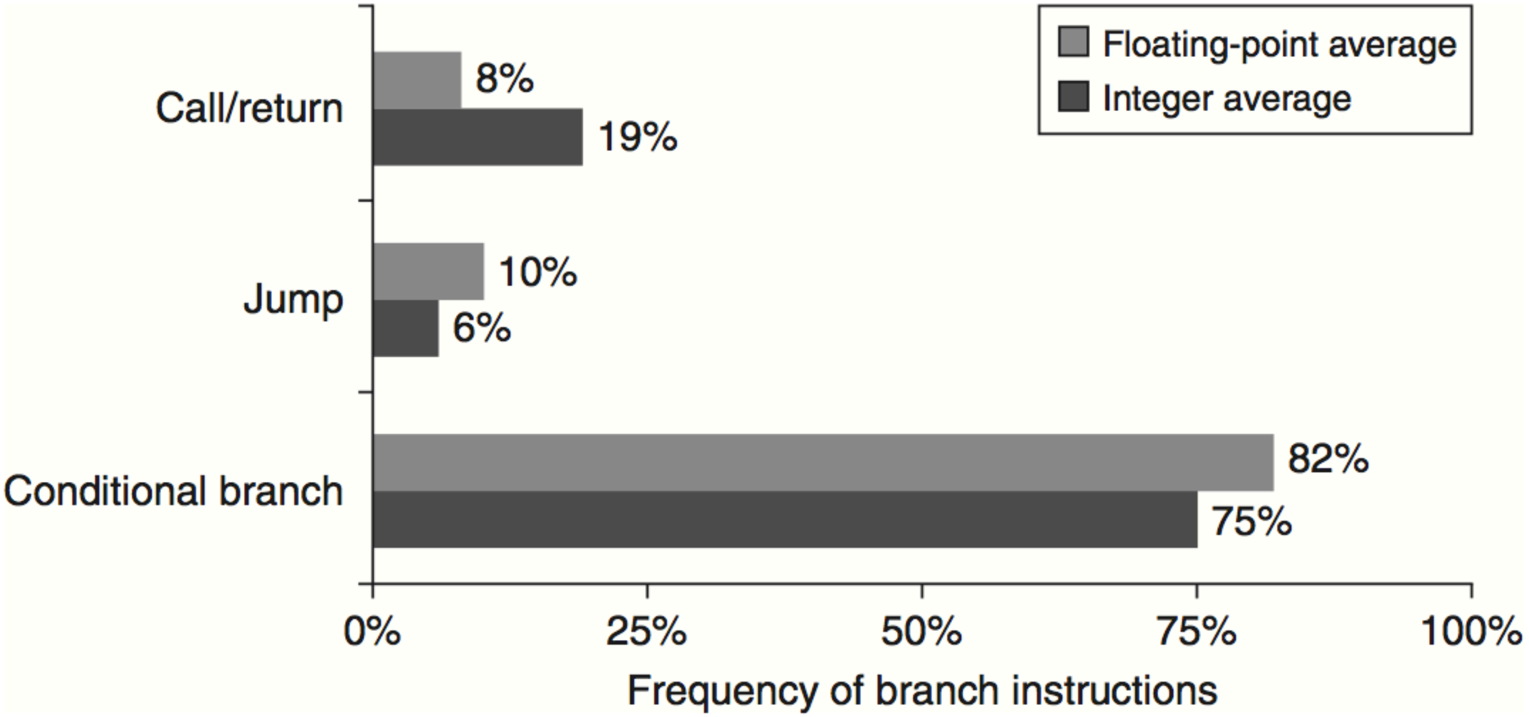


```c
float sum(int N, float X[], float a) {
    int i;
    float result = 0.0;
    for (i = 0; i < N; ++i)
        result += a * X[i];
    return result;
}
```



```
        .file   "sum.c"
        .text
        .align  2
        .globl  sum
        .type   sum, @function
sum:
        add     sp,sp,-48
        sd      s0,40(sp)
        add     s0,sp,48
        sw      a0,-36(s0)
        sd      a1,-48(s0)
        fsw     fa2,-40(s0)
        sw      zero,-24(s0)
        sw      zero,-20(s0)
        j       .L2
.L3:
        lw      a5,-20(s0)
        sll     a5,a5,2
        ld      a4,-48(s0)
        add     a5,a4,a5
        flw     fa4,0(a5)
        flw     fa5,-40(s0)
        fmul.s  fa5,fa4,fa5
        flw     fa4,-24(s0)
        fadd.s  fa5,fa4,fa5
        fsw     fa5,-24(s0)
        lw      a5,-20(s0)
        addw    a5,a5,1
        sw      a5,-20(s0)
.L2:
        lw      a4,-20(s0)
        lw      a5,-36(s0)
        blt     a4,a5,.L3
        flw     fa5,-24(s0)
        fmv.s   fa0,fa5
        ld      s0,40(sp)
        add     sp,sp,48
        jr      ra
        .size   sum, .-sum
        .ident  "GCC: (GNU) 6.1.0"
```

Breakdown of Control Flow Instructions



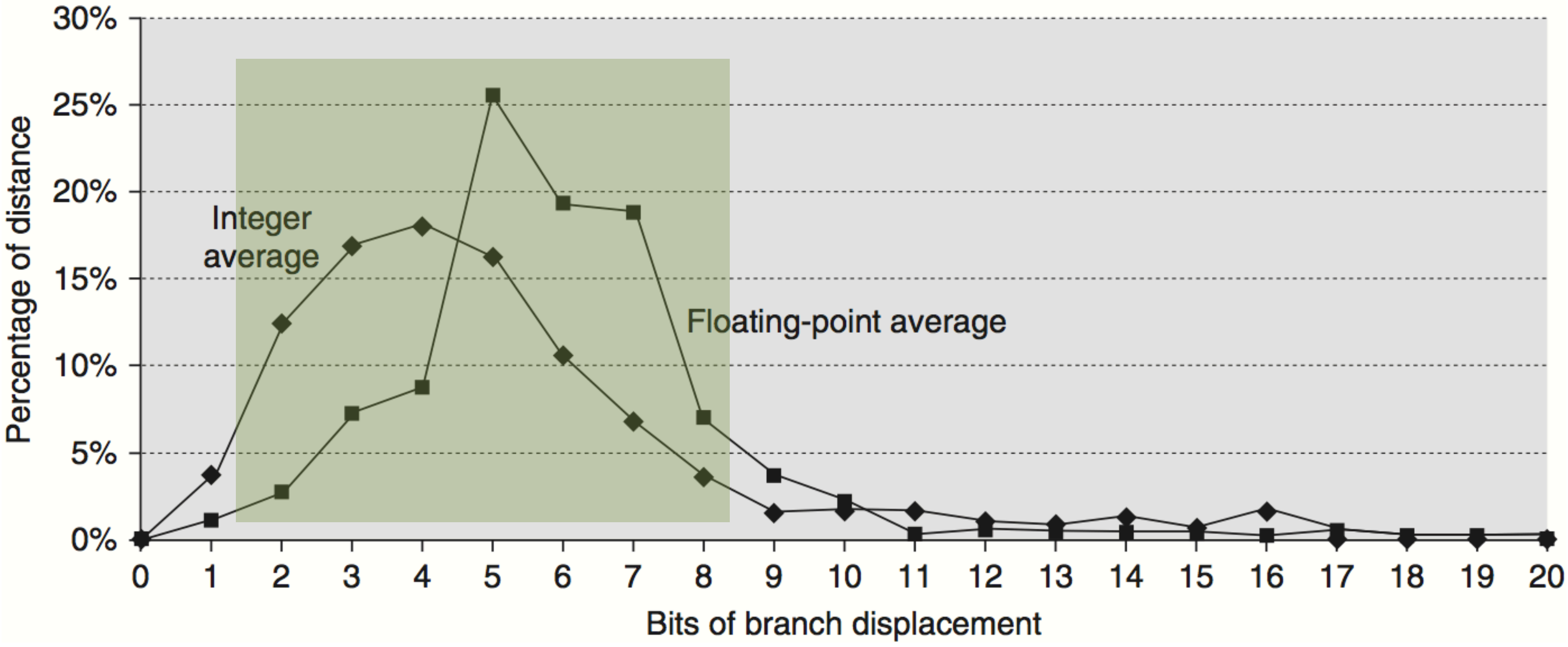
- **Issues:**
  - **Where is the target address? How to specify it? (label)**
  - **Caller: Where is return address kept? How are the arguments passed?**
  - **Callee: Where is return address? How are the results passed?**

Addressing Modes for Control Flow Instructions

- PC-relative (Program Counter)
  - Supply a displacement added to the PC
    - Known at compile time for jumps, branches, and calls (specified within the instruction)
  - The target is often near the current instruction
    - Requiring fewer bits
    - Independently of where it is loaded (position independence)

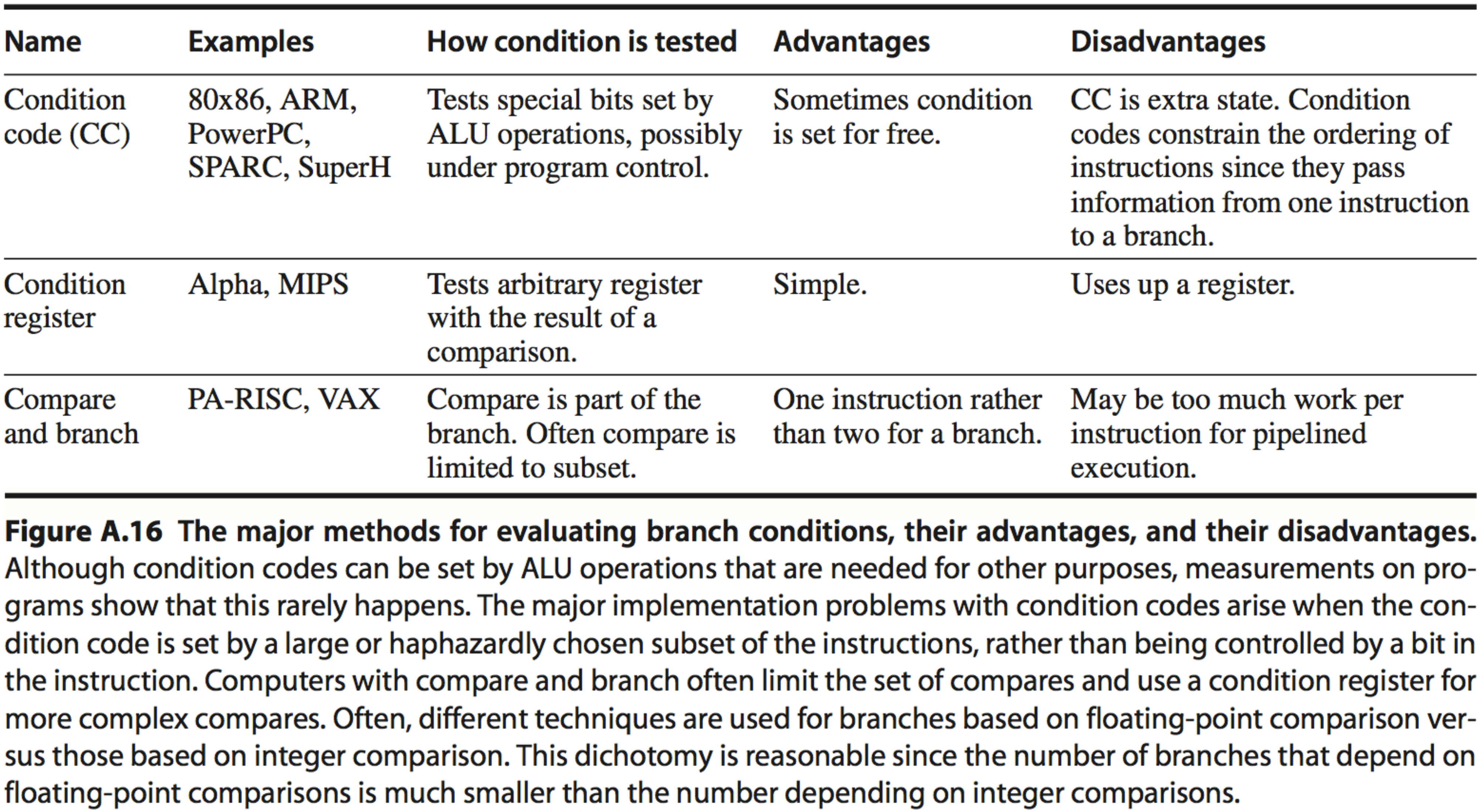
- Register indirect addressing – dynamic addressing
  - The target address may not be known at compile time
  - Naming a register that contains the target address
    - Case or switch statements
    - Virtual functions or methods in C++ or Java
    - High-order functions or function pointers in C or C++
    - Dynamically shared libraries

Branch Distances



igure A.15 Branch distances in terms of number of instructions between the target and the branch instruction. he most frequent branches in the integer programs are to targets that can be encoded in 4 to 8 bits. This result tells ıs that short displacement fields often suffice for branches and that the designer can gain some encoding density by ıaving a shorter instruction with a smaller branch displacement. These measurements were taken on a load-store :omputer (Alpha architecture) with all instructions aligned on word boundaries. An architecture that requires fewer nstructions for the same program, such as a VAX, would have shorter branch distances. However, the number of bits ıeeded for the displacement may increase if the computer has variable-length instructions to be aligned on any byte ›oundary. The programs and computer used to collect these statistics are the same as those in Figure A.8.

Conditional Branch Options



Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

Figure A.16 The major methods for evaluating branch conditions, their advantages, and their disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Computers with compare and branch often limit the set of compares and use a condition register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This dichotomy is reasonable since the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

Comparison Type vs. Frequency

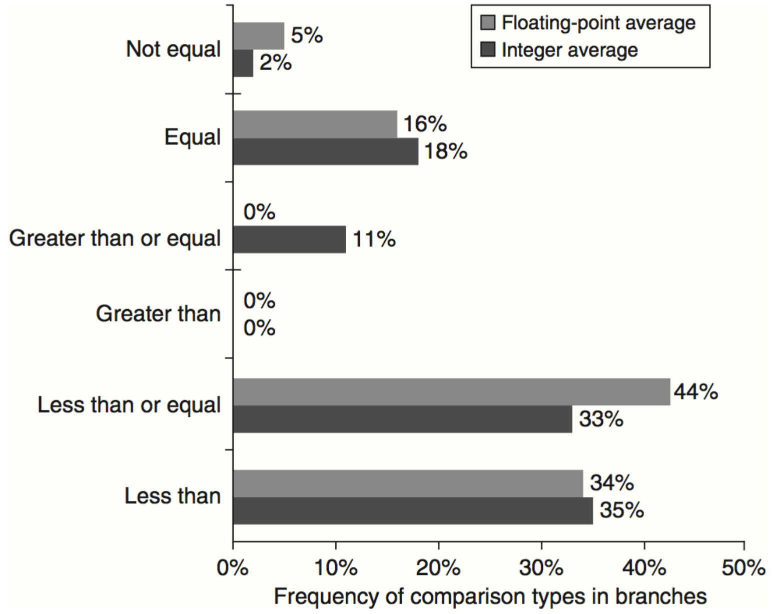


Figure A.17 Frequency of different types of compares in conditional branches.

- Most loops go from 0 to n.
- Most backward branches are loops – taken about 90%

Program	% backward branches	**% all control instructions that modify PC**
gcc	26%	**63%**
spice	31%	**63%**
TeX	17%	**70%**
Average	25%	**65%**

Procedure Invocation Options

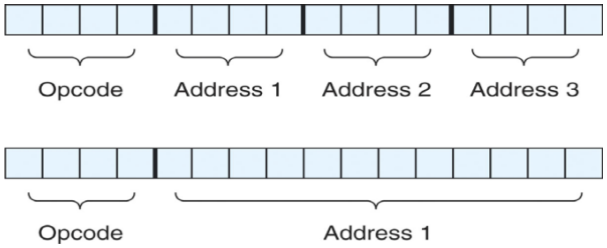
- Procedure calls and returns
  - control transfer
  - state saving; the return address must be saved

  Newer architectures require the compiler to generate stores and loads for each register saved and restored

- Two basic conventions in use to save registers
  - ***caller saving***: the calling procedure must save the registers that it wants preserved for access after the call
    - the called procedure need not worry about registers
  - ***callee saving***: the called procedure must save the registers it wants to use
    - leaving the caller unrestrained

  most real systems today use a combination of both
    - Application binary interface (ABI) that set down the basic rules as to which register be caller saved and which should be callee saved

7. Encoding an Instruction Set

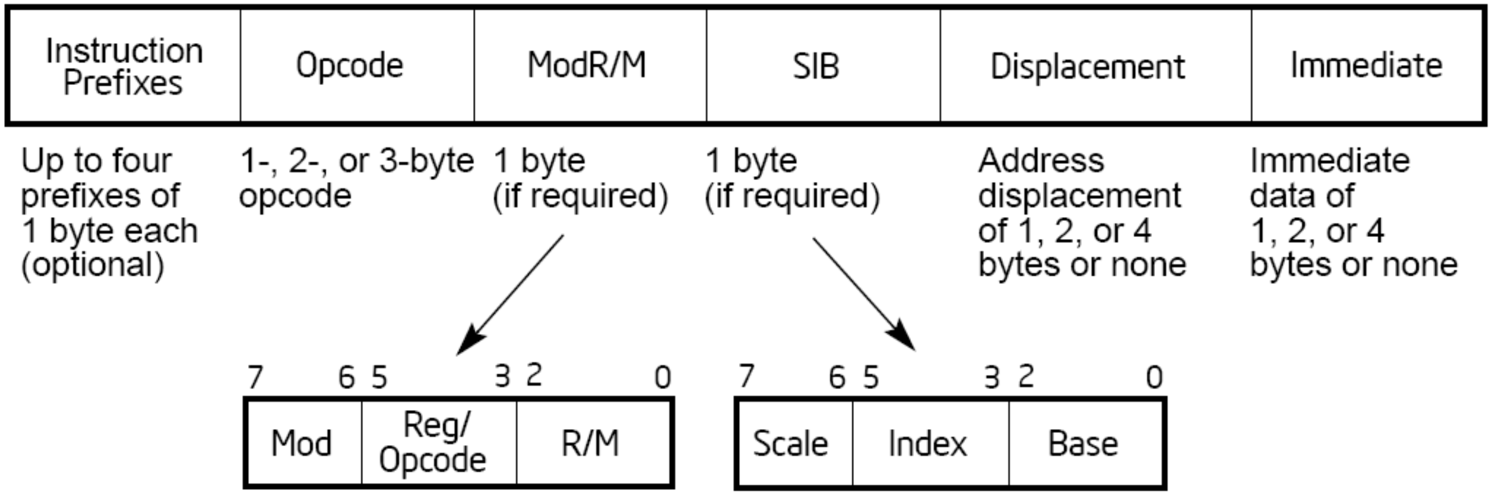


- Opcode: specifying the operation
- # of operand
  - addressing mode
  - address specifier: tells what addressing mode is used
  - Load-store computer
    - Only one memory operand
    - Only one or two addressing modes

- The architecture must balancing several competing forces when encoding the instruction set:
  - # of registers && Addressing modes
  - Size of registers && Addressing mode fields
  - Average instruction size && Average program size.
  - Easy to handle in pipeline implementation.

Example: x86 and Alpha

- x86:



Instruction Prefixes | Opcode | ModR/M | SIB | Displacement | Immediate

Instruction Prefixes: Up to four prefixes of 1 byte each (optional)

Opcode: 1-, 2-, or 3-byte opcode

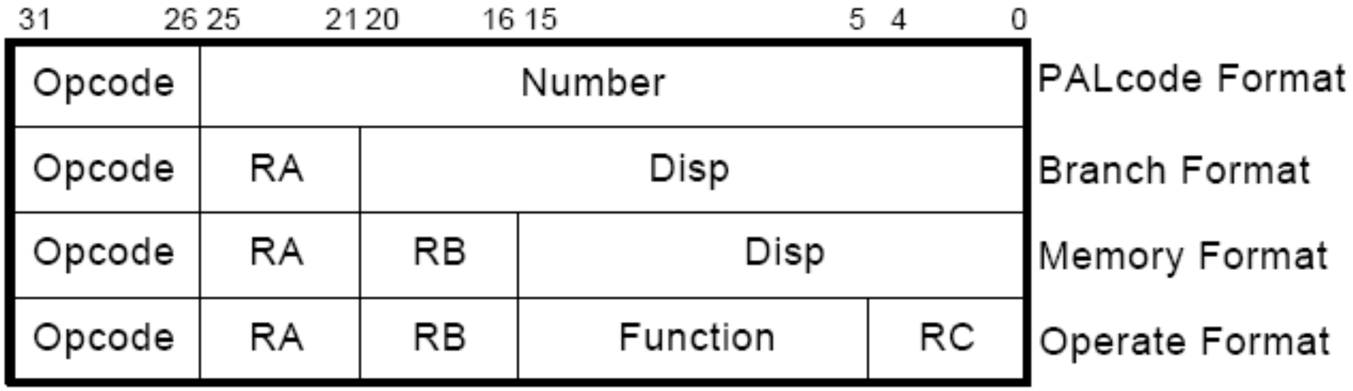
ModR/M: 1 byte (if required)

SIB: 1 byte (if required)

Displacement: Address displacement of 1, 2, or 4 bytes or none

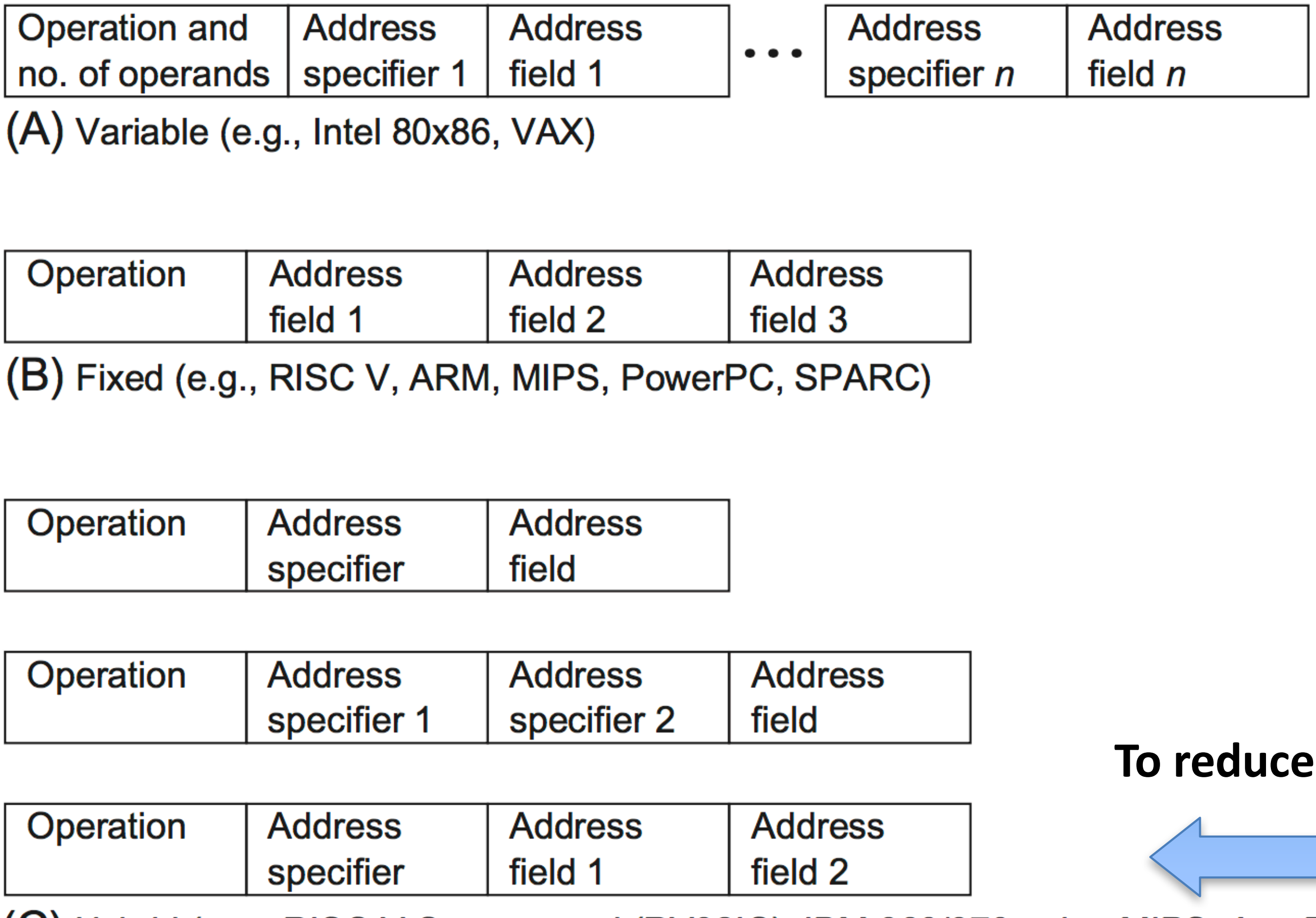
Immediate: Immediate data of 1, 2, or 4 bytes or none

- Alpha:



31–26	25–21	20–16	15–5	4–0	
Opcode	Number				PALcode Format
Opcode	RA	Disp			Branch Format
Opcode	RA	RB	Disp		Memory Format
Opcode	RA	RB	Function	RC	Operate Format

Three Basic Variations for Instruction Encoding



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

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

Operation	Address field 1	Address field 2	Address field 3

(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)

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., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

The length of 80x86 (CISC) instructions varies between 1 and 17 bytes.

The length of most RISC ISA instructions are 4 bytes.

X86 program are generally smaller than RISC ISA.

To reduce RISC code size

Figure A.18 Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, because

Instruction Length Tradeoffs

- Fixed length: Length of all instructions the same

  + Easier to decode single instruction in hardware

  + Easier to decode multiple instructions concurrently

  -- Wasted bits in instructions (Why is this bad?)

  -- Harder-to-extend ISA (how to add new instructions?)

- Variable length: Length of instructions different (determined by opcode and sub-opcode)

  + Compact encoding (Why is this good?)

    Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. How?

  -- More logic to decode a single instruction

  -- Harder to decode multiple instructions concurrently

- Tradeoffs
  – Code size (memory space, bandwidth, latency) vs. hardware complexity
  – ISA extensibility and expressiveness
  – Performance? Smaller code vs. imperfect decode

Uniform vs Non-uniform Decode

- Uniform decode: Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location
  - immediate values, …
  - Many "RISC" ISAs: Alpha, MIPS, SPARC

  + Easier decode, simpler hardware

  + Enables parallelism: generate target address before knowing the instruction is a branch

  -- Restricts instruction format (fewer instructions?) or wastes space

- Non-uniform decode
  - E.g., opcode can be the 1st-7th byte in x86

  + More compact and powerful instruction format

  -- More complex decode logic

Reduced Code Size in RISCs

- Hybrid encoding – support 16-bit and 32-bit instructions in RISC, eg. ARM Thumb, MIPS 16 and RISC-V
  - Narrow instructions support fewer operations, smaller address and immediate fields, fewer registers, and two-address format rather than the classic three-address format
  - Claim a code size reduction of up to 40%

- Compression in IBM's CodePack
  - Adds hardware to decompress instructions as they are fetched from memory on an instruction cache miss
  - The instruction cache contains full 32-bit instructions, but compressed code is kept in main memory, ROMs, and the disk
  - Claim code reduction 35% - 40%
  - PowerPC create a Hash table in memory that map between compressed and uncompressed address. Code size 35%~40%

- Hitachi's SuperH: fixed 16-bit format
  - 16 rather than 32 registers
  - fewer instructions

Summary of Instruction Encoding

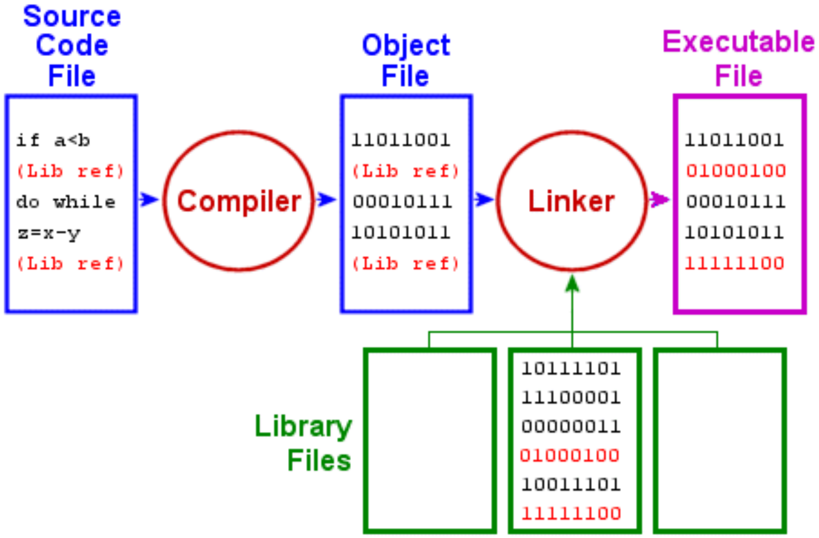
- Three choices
  - Variable, fixed and hybrid
  - Note the differences of hybrid and variable
- Choices of instruction encoding is a tradeoff between
  - For performance: fixed encoding
  - For code size: variable encoding

- How hybrid encoding is used in RISC to reduce code size
  - 16bit and 32bit
- In general, we see:
  - RISC: fixed or hybrid
  - CISC: variable

8 The Role of Compilers

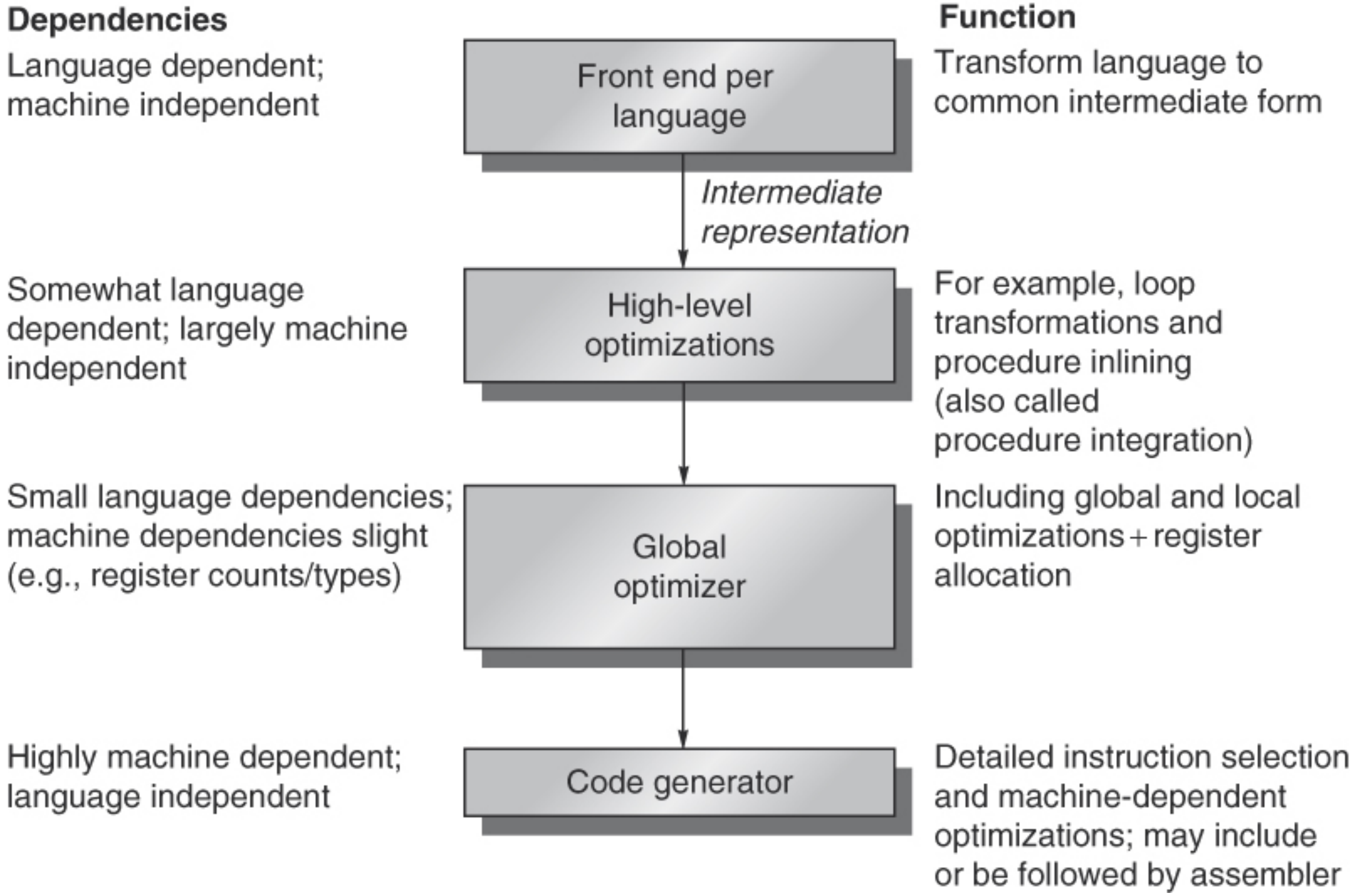
- Almost all programming is done in high-level languages.
  - An ISA is essentially a complier target.

- *See backup slides for the compilation stage by most compiler, e.g. gcc*



- Compiler goals:
  - All correct programs execute correctly
  - Most compiled programs execute fast (optimizations)
  - Fast compilation
  - Debugging support

Typical Modern Compiler Structure



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

Figure A.19 Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower-quality code is acceptable. A *pass* is simply one phase in which the compiler reads and transforms the entire program. (The term *phase* is often used inter-changeably with *pass*.) Because the optimizing passes are separated, multiple languages can use the same optimizing and code generation passes. Only a new front end is required for a new language.

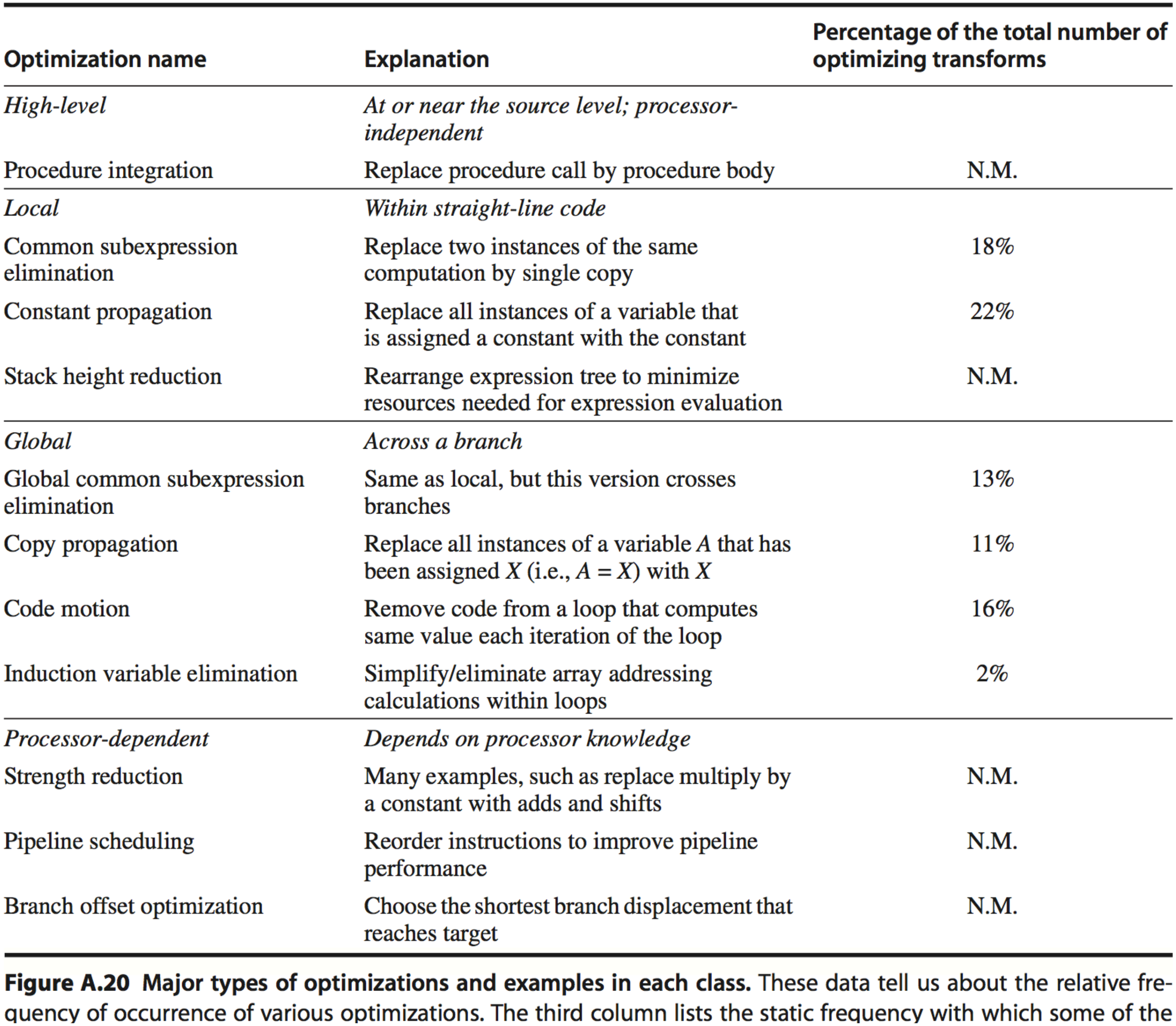
Optimization Types

- High level – done at or near source code level
  - If procedure is called only once, put it in-line and save CALL
  - more general case: if call-count < some threshold, put them in-line
- Local – done within straight-line code
  - common sub-expressions produce same value – either allocate a register or replace with single copy
  - constant propagation – replace constant valued variable with the constant
  - stack height reduction – re-arrange expression tree to minimize temporary storage needs
- Global – across a branch
  - copy propagation – replace all instances of a variable A that has been assigned X (i.e., A=X) with X.
  - code motion – remove code from a loop that computes same value each iteration of the loop and put it before the loop
  - simplify or eliminate array addressing calculations in loops

Optimization Types

- Machine-dependent optimizations – based on machine knowledge
  - strength reduction – replace multiply by a constant with shifts and adds
    - would make sense if there was no hardware support for MUL
    - a trickier version: 17 🎞 = arithmetic left shift 4 and add
- Pipelining scheduling – reorder instructions to improve pipeline performance
  - dependency analysis
  - branch offset optimization - reorder code to minimize branch offsets

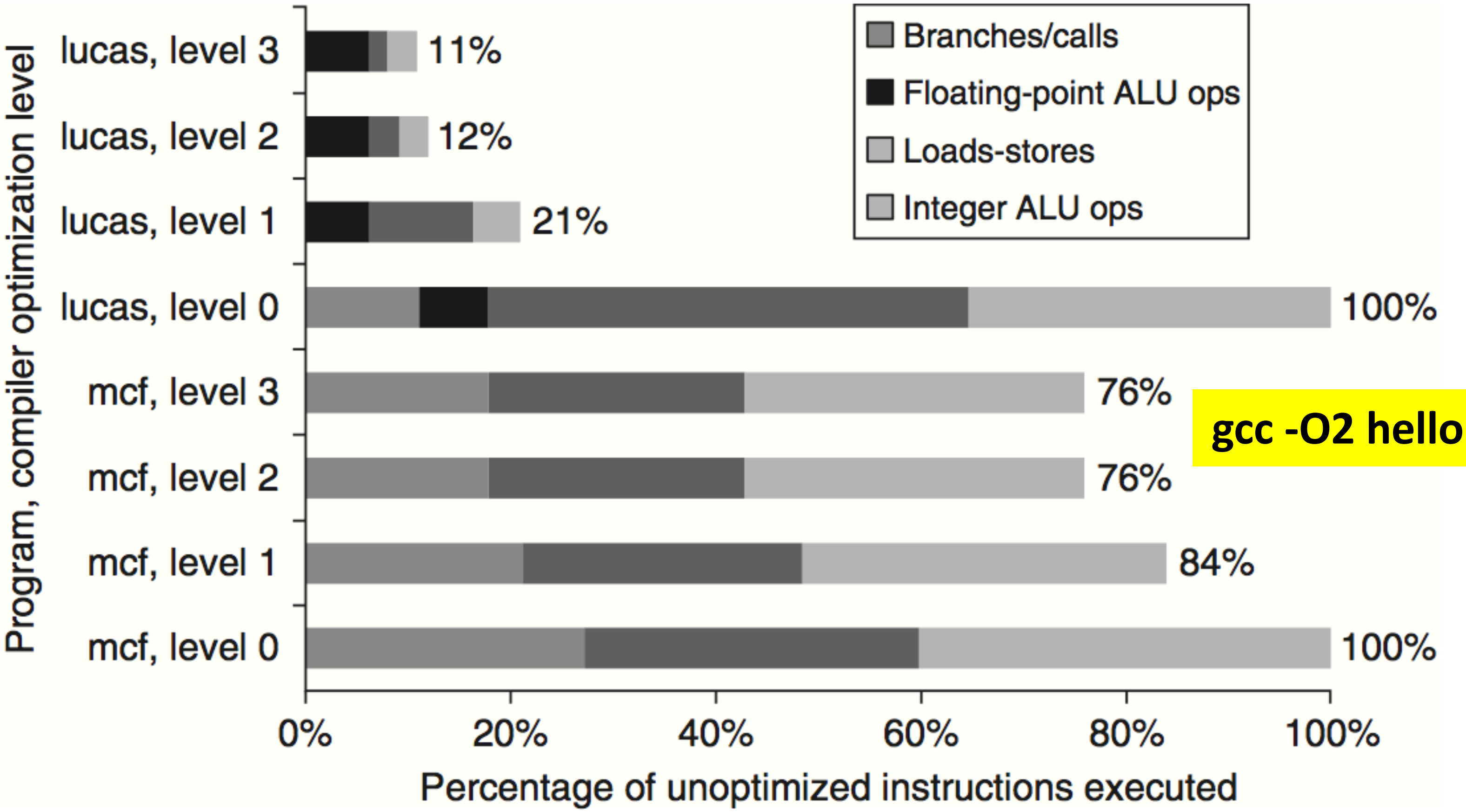
Major Types of Optimizations



Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	*At or near the source level; processor-independent*	
Procedure integration	Replace procedure call by procedure body	N.M.
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M.
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable *A* that has been assigned *X* (i.e., $A = X$) with *X*	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable elimination	Simplify/eliminate array addressing calculations within loops	2%
Processor-dependent	*Depends on processor knowledge*	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

Figure A.20 Major types of optimizations and examples in each class. These data tell us about the relative frequency of occurrence of various optimizations. The third column lists the static frequency with which some of the

Complier Optimizations – Change in IC



```
gcc -O2 hello.c -o hello
```

Figure A.21 Change in instruction count for the programs lucas and mcf from the SPEC2000 as compiler optimization levels vary. Level 0 is the same as unoptimized code. Level 1 includes local optimizations, code scheduling, and local register allocation. Level 2 includes global optimizations, loop transformations (software pipelining), and global register allocation. Level 3 adds procedure integration. These experiments were performed on Alpha compilers.

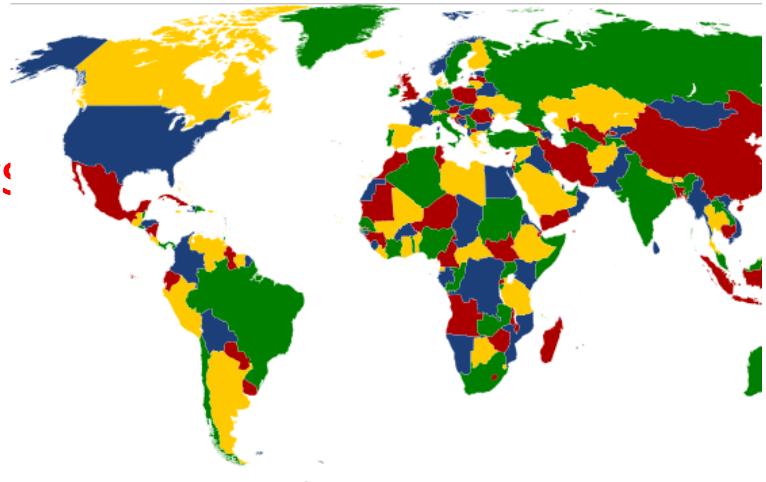
Compiler Based Register Optimization

- **Compiler assumes small number of registers (16-32)**
  - Optimizing use is up to compiler
  - HLL programs have no explicit references to registers

- Compiler Approach
  - 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
    - Spilling

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



https://en.wikipedia.org/wiki/Graph_coloring

- Registration allocation
  - 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

Iron-code Summary

- *Section A.2*—Use general-purpose registers with a load-store architecture.
- *Section A.3*—Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register indirect.
- *Section A.4*—Support these data sizes and types: 8-, 16-, 32-, and 64-bit integers and 64-bit IEEE 754 floating-point numbers.
  - Now we see 16-bit FP for deep learning in GPU
    - http://www.nextplatform.com/2016/09/13/nvidia-pushes-deep-learning-inference-new-pascal-gpus/
- *Section A.5*—Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register- register, and shift.
- *Section A.6*—Compare equal, compare not equal, compare less, branch (with a PC-relative address at least 8 bits long), jump, call, and return.
- *Section A.7*—Use fixed instruction encoding if interested in performance, and use variable instruction encoding if interested in code size.
- *Section A.8*—Provide at least 16 general-purpose registers, be sure all addressing modes apply to all data transfer instructions, and aim for a minimalist IS
  - Often use separate floating-point registers.
  - The justification is to increase the total number of registers without raising problems in the instruction format or in the speed of the general-purpose register file. This compromise, however, is not orthogonal.

Real World ISA

Arch	Type	# Oper	# Mem	Data Size	# Regs	Addr Size	Use
Alpha	Reg-Reg	3	0	64-bit	32	64-bit	Workstation
ARM	Reg-Reg	3	0	32/64-bit	16	32/64-bit	Cell Phones, Embedded
MIPS	Reg-Reg	3	0	32/64-bit	32	32/64-bit	Workstation, Embedded
SPARC	Reg-Reg	3	0	32/64-bit	24-32	32/64-bit	Workstation
TI C6000	Reg-Reg	3	0	32-bit	32	32-bit	DSP
IBM 360	Reg-Mem	2	1	32-bit	16	24/31/64	Mainframe
x86	Reg-Mem	2	1	8/16/32/64-bit	4/8/24	16/32/64	Personal Computers
VAX	Mem-Mem	3	3	32-bit	16	32-bit	Minicomputer

The details in design is to trade-off!