CSE 530
Fundamentals of Computer Architecture

Spring 2021

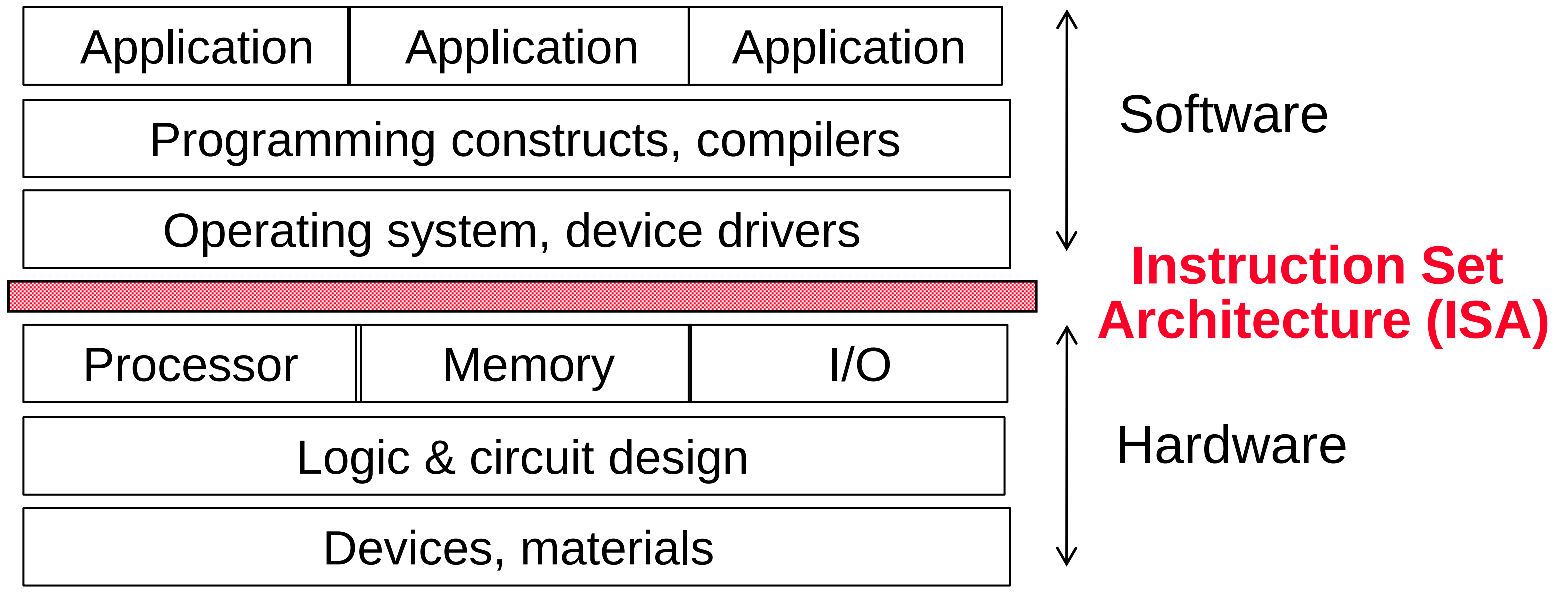
Instruction Set Architectures

John (Jack) Sampson

Course material on CANVAS: canvas.psu.edu

[Adapted in part from Mary Jane Irwin, V. Narayanan, A. Kolli @PSU, and includes materials originally developed by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch of Carnegie Mellon University, Purdue University, University of Michigan, University of Pennsylvania, and University of Wisconsin. Material flow organized in part around *Computer Architecture: A Quantitative Approach* by Hennessey and Patterson]

Review: Instruction Set Architecture (ISA)



ISA = contract btn hw and sw. Part of processor visible to programmers/compilers

- The ISA definition serves as the abstraction boundary between the software and hardware
  - Facilitates the parallel development of software layers and hardware layers
  - Enables implementations of varying cost and performance to run on identical software
  - Lasts through many generations (portable)

What makes a good ISA?

- **Programmability**
  - Easy to express programs efficiently? Easy to compile to?
    - However, beware. E.g., Intel 432 en.wikipedia.org/wiki/Intel_iAPX_432

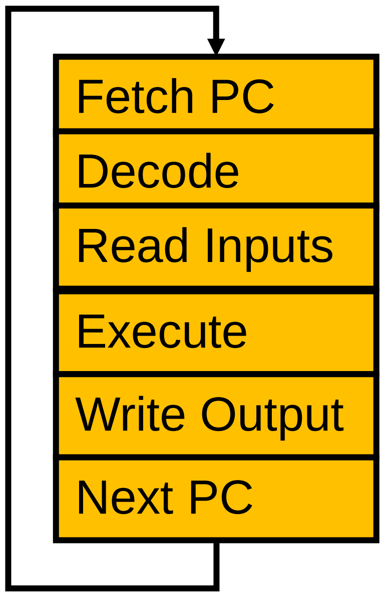
Easy to write programs but did not last!

- **Implementability**
  - Easy to design high-performance implementations?
  - More recently
    - Easy to design low-power implementations?
    - Easy to design high-reliability implementations?
    - Easy to design low-cost implementations?
- **Compatibility**

an economic requirement. if we had to redesign software with each hardware version (because of no compatibility with new hardware, then it would be bad)

  - Easy to maintain programmability (implementability) as languages and programs (technology) evolves?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, Pentium II, Pentium III, Pentium 4, Core, Core 2, Nehalem, …

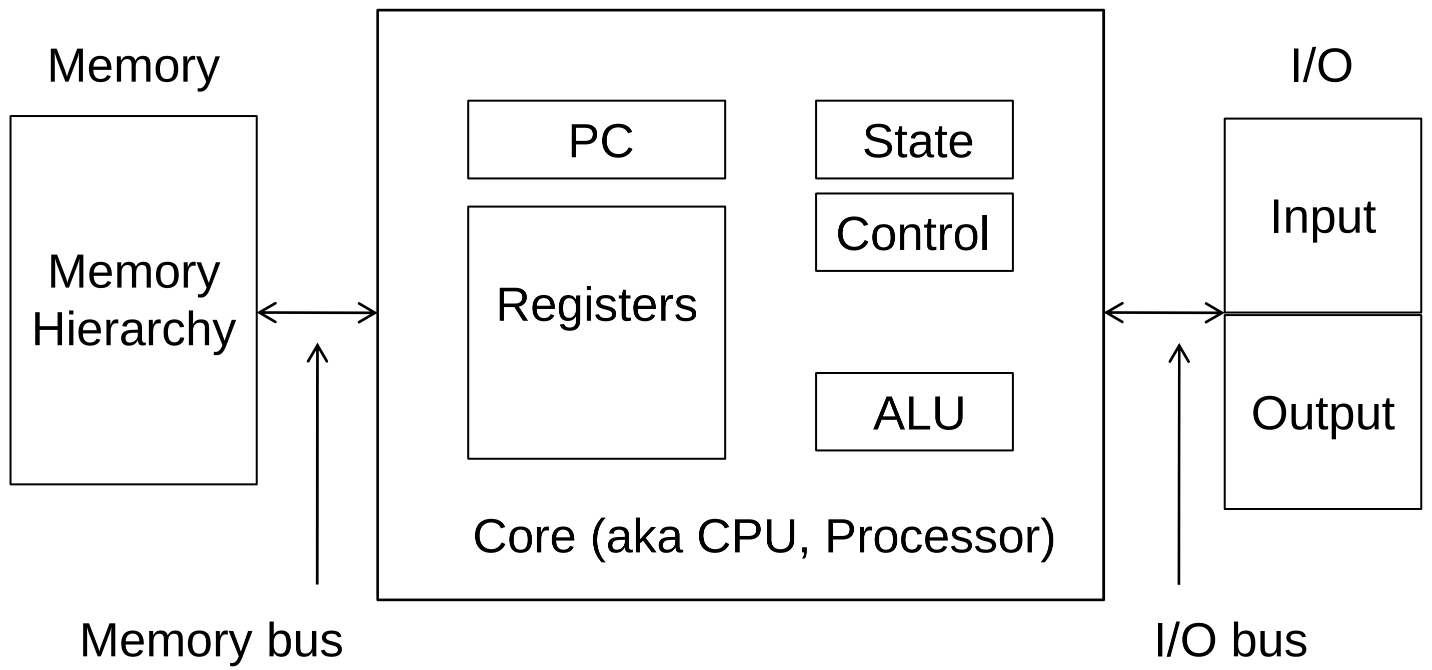
The sequential model



- Implicit model of nearly all modern ISAs
  - Often called Von Neuman machine model
- Basic feature: the **program counter (PC)**
  - Defines **total order** on dynamic instructions
    - Next PC is PC++ unless instr says otherwise
  - Order and **named storage** define computation
    - Value flows from instr X to Y via storage A iff…
    - X names A as output, Y names A as input…
    - And Y after X in total order
- Processor logically executes loop at left
  - Instruction execution assumed atomic
  - Instruction X finishes before instr X+1 starts
- More parallel alternatives have been proposed

The von Neumann machine model

- The vast majority of microprocessor architectures still follow the Von Neumann stored program architecture model (1945) http://en.wikipedia.org/wiki/First_Draft_of_a_Report_on_the_EDVAC



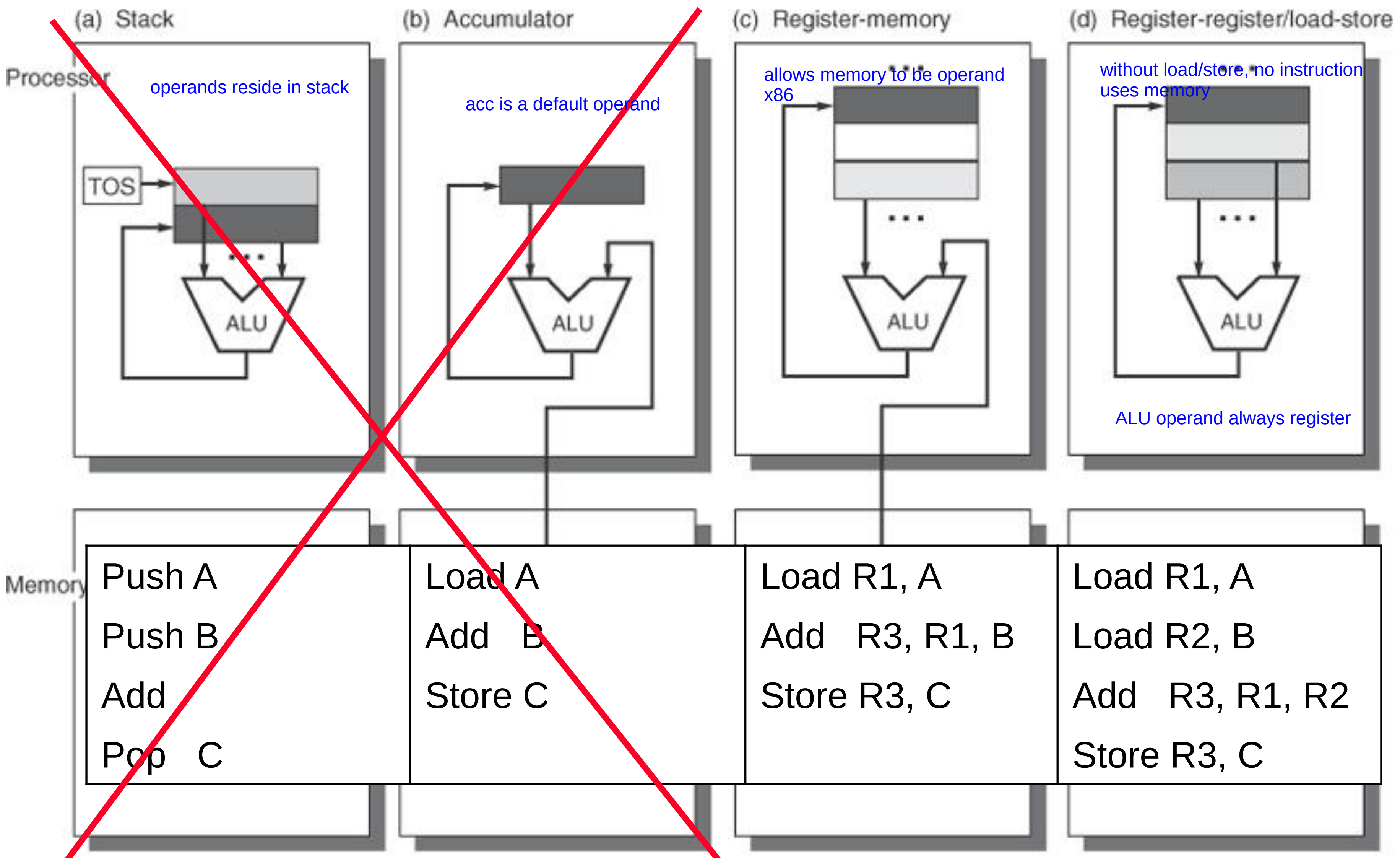
difference between execution models: when the next instruction consumes its operands and executes
Von: when PC points to it
Data Flow: when all data are ready
Event Driven: when some event fires

Seven dimensions of an ISA

1. Class of ISA – general-purpose register arch's
   - register-memory (80x86) or load-store (MIPS)
2. Instr encoding – fixed length and variable length
3. Types and sizes of operands – 8-bit (ASCII, pixel), 16-bit (Unicode), 32-bit, 64-bit, 80-bit (xdouble precision)
4. Addressing modes – register, immediate (for constants), displacement, absolute, register indirect, …
5. Memory addressing – byte addressing (aligned or not)
6. Operations – data transfer, arithmetic (integer and floating point), logical, control
7. Control flow – conditional branches, jumps, procedure calls & returns; PC relative, return addr on stack

1. Class of ISA

based on where operand is other than memory



(a) Stack
(b) Accumulator
(c) Register-memory
(d) Register-register/load-store
Processor
Memory
TOS
ALU
operands reside in stack
acc is a default operand
allows memory to be operand
x86
without load/store, no instruction uses memory
ALU operand always register

~~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

How many explicit operands / ALU instr?

- **Operand model**: how many explicit operands / ALU instr?
  - **3**: general-purpose

    `add R1,R2,R3` means [R1] = [R2] + [R3] **(MIPS uses this)**
  - **2**: multiple explicit accumulators (output doubles as input)

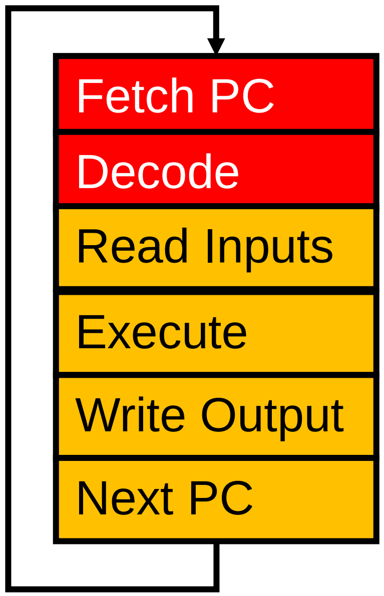
    `add R1,R2` means [R1] = [R1] + [R2] **(x86 uses this)**
  - **1**: one implicit accumulator

    `add R1` means ACC = ACC + [R1]
  - **0**: hardware stack

    `add` means STK[TOS++] = STK[--TOS] + STK[--TOS]
  - **4+**: useful only in special situations
- Examples above show register operands …
  - But operands can be memory addresses, or mixed register/memory addresses
  - ISAs with register-only ALU instr's are **"load-store"**

just like last page

2. Instruction length and encoding



- **Length**
  - Fixed length
    - Most common is 32 bits
    - + Simple implementation (next PC usually just PC+4)
    - – Code density: takes 32 bits to increment a register by 1
  - Variable length
    - + Code density
      - x86 can do increment in one 8-bit instruction
    - – Complex fetch (where does the next instruction begin?)
  - Compromise: support two lengths
    - E.g., MIPS16 or ARM's Thumb
- **Encoding**
  - A few simple encodings simplify the decoder
    - x86 decoder one nasty piece of logic

x86 - variable length instruction, also, which bit means what -- no uniformity there
combining these all, x86 decoder is very very complicated

Examples of instruction encodings

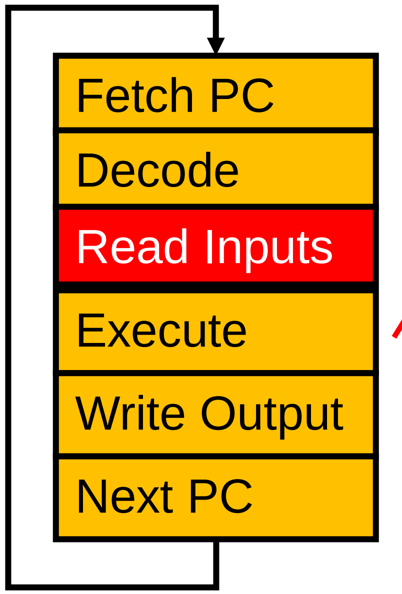
- MIPS
  - Fixed length
  - 32-bits, 3 formats, simple encoding
  - (MIPS16 has 16-bit versions of common instr for code density)

R-type	Op(6)	Rs(5)	Rt(5)	Rd(5)	Sh(5)	Func(6)
I-type	Op(6)	Rs(5)	Rt(5)	Immed/Displace(16)		
J-type	Op(6)	Target(26)				

- x86
  - Variable length encoding (1 to 17 bytes) (* is optional)

Prefix*(1-5)	Op	OpX*	RM*	SIB*	Disp*(1-4)	Imm*(1-4)

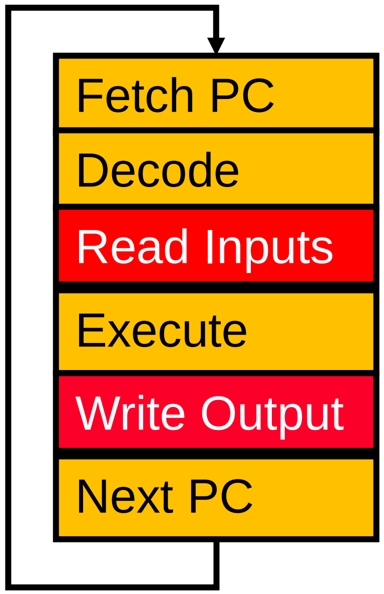
3. Types and sizes of operands



- Datatypes
  - Software: attribute of data
  - Hardware: attribute of operation, data is just 0/1's
- All processors support
  - Characters: ASCII (8-bit), Unicode (16-bit)
  - 2C integer arithmetic/logic (8/16/32/64-bit)
  - IEEE754 floating-point arithmetic (32/64 bit)
    - Intel has 80-bit floating-point
- More recently, most processors support
  - "Packed-integer" instr's, e.g., MMX
  - "Packed-fp" instr's, e.g., SSE/SSE2
- Processors no longer *support*
  - Decimal, other fixed-point arithmetic
    - Binary-coded decimal (BCD)

BCD's are no longer supported but still part of the ISA
You can stop compiler from emitting BCD code, but you cannot change the contract for that machine!

Where does the data live?



- **Memory**
  - Fundamental storage space
- **Registers**
  - Faster than memory, quite handy
  - Most processors have these too
- **Immediates** (part of the instruction)
  - Values spelled out as bits in the instruction
  - Input only

How much memory ?   Address size ?

- What does “64-bit” in a 64-bit ISA mean?
  - **Support memory size of 2^{64}**
  - Alternative (wrong) definition: width of calculation operations
- **“Virtual” address size**

  ISA's do not talk about physical memory, only virtual memory

  ISA just talks about where operands can come from, and where they can go to
  - Determines size of addressable (usable) memory
    - Current 32-bit or 64-bit address spaces

      2^32 different named locations supported
  - Most critical, inescapable ISA design decision
    - Too small? Will limit the lifetime of ISA
    - May require nasty hacks to overcome (E.g., x86 segments)
  - x86 evolution:
    - 4-bit (4004), 8-bit (8008), 16-bit (8086), 24-bit (80286),
    - 32-bit + protected memory (80386)
    - 64-bit (AMD’s Opteron & Intel’s EM64T Pentium4)

- All ISAs have pretty much moved to 64-bit (but some systems still use 32-bit version – why?)

How many registers ?

- Registers faster than memory, have as many as possible?
  - **No**
- One reason registers are faster: there are **fewer of them**
  - Small is fast (hardware truism)
  - More registers means **more saving/restoring**
- Another: they are **directly addressed** (no address calc)
  - More of them, means larger specifiers
  - Fewer registers per instruction or indirect addressing
- **Not everything can be put in registers**
  - Structures, arrays, anything pointed-to
  - Although compilers are getting better at putting more things in
- Trend: more registers: 8 (x86)→32 (MIPS) →128 (IA64)
  - 64-bit x86 has 16 64-bit integer and 16 128-bit FP registers

How are memory locations specified?

- Registers are specified **directly**
  - Register names are short, can be encoded in a few bits in the instruction (32 registers → 5 bits, 64 registers → 6 bits, etc.)
  - Some instructions implicitly read/write certain registers (e.g., subroutine calls/returns)

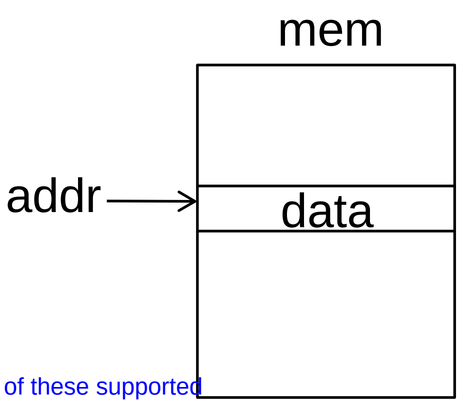
- How are (other than register) addresses specified?
  - Addresses are long (e.g., 64-bit)
  - **Addressing mode**: how are instr bits converted to addresses?
  - Think about: what high-level idiom addressing mode captures
  - What is the implementation impact (can it be made to run fast, how does it affect the instruction length) ?
  - What is the instruction count impact ?

these are considered when we are fixing how many bits the ISA should support

4. Addressing modes

- **Addressing mode:** way of specifying memory address in memory-memory or load/store instr's
- Examples

vast majority of instructions are just the first two



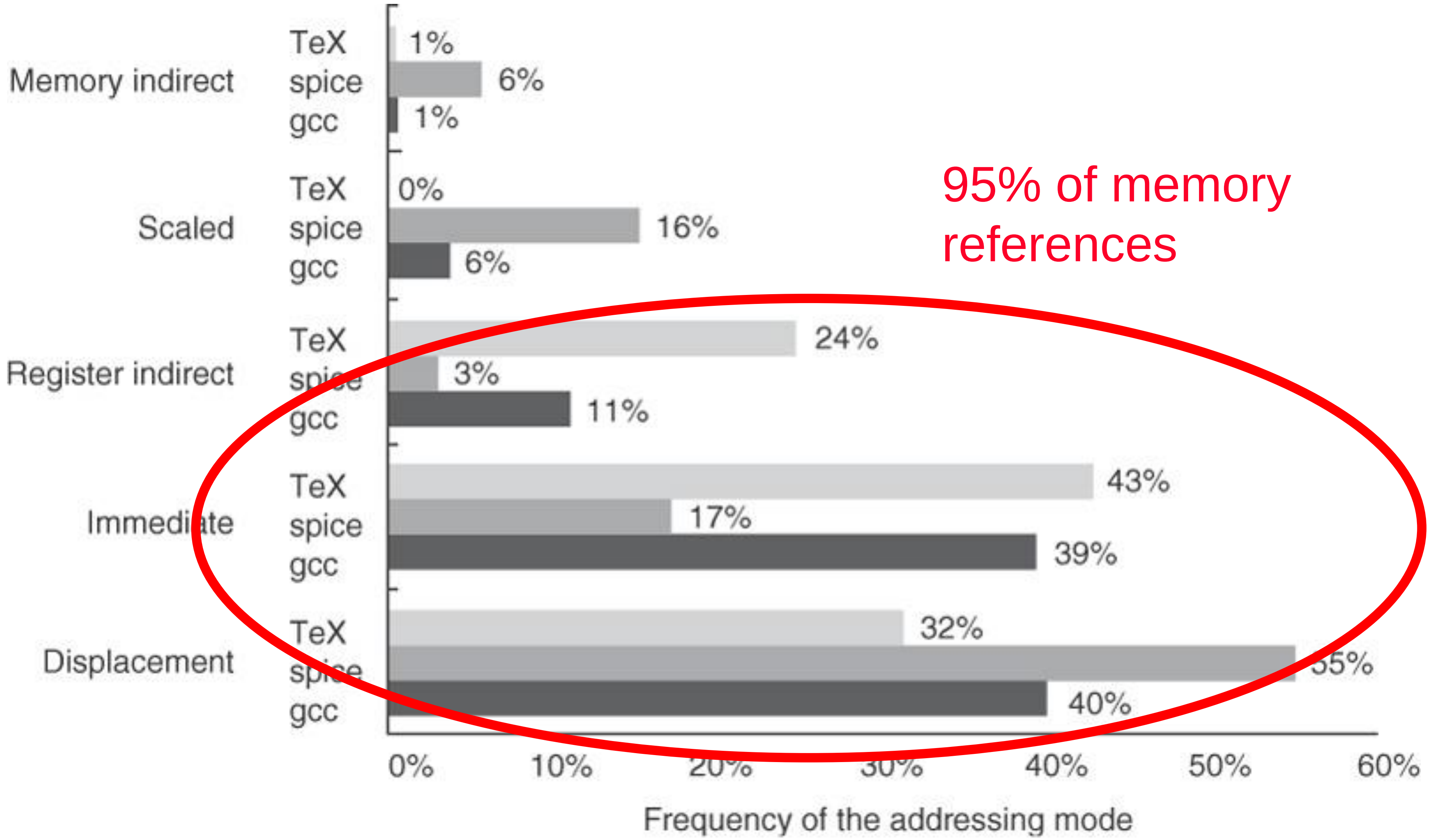
  - **Register-indirect:** R1=mem[R2]
  - **Displacement:** R1=mem[R2+displace]
  - **Index-base:** R1=mem[R2+R3]
  - **Memory-indirect:** R1=mem[mem[R2]]
  - **Auto-increment:** R1=mem[R2], R2=R2+1
  - **Auto-decrement:** R2=R2-1, R1=mem[R2]
  - **Auto-indexing:** R1=mem[R2+displace], R2=R2+displace
  - **Scaled:** R1=mem[R2+R3*displace1+displace2]
  - **PC-relative (for branches):** PC=mem[PC+displace]

the more of these supported
the more complex the ISA will be (and the hardware implementation)

- What high-level program idioms are these used for?
- What implementation impact? What instr count impact?

VAX addressing mode (to memory) frequencies

- Register mode accounted for 50% of operand references



MIPS addressing modes

- MIPS implements only displacement
  - Why? Experiment on VAX (the ISA with every mode) found distribution (not counting immediate)
    - Disp: 61%, reg-ind: 19%, scaled: 11%, mem-ind: 5%, other: 4%
  - 80% use small displacement or reg indirect (displacement 0)
- I-type instructions: 16-bit displacement

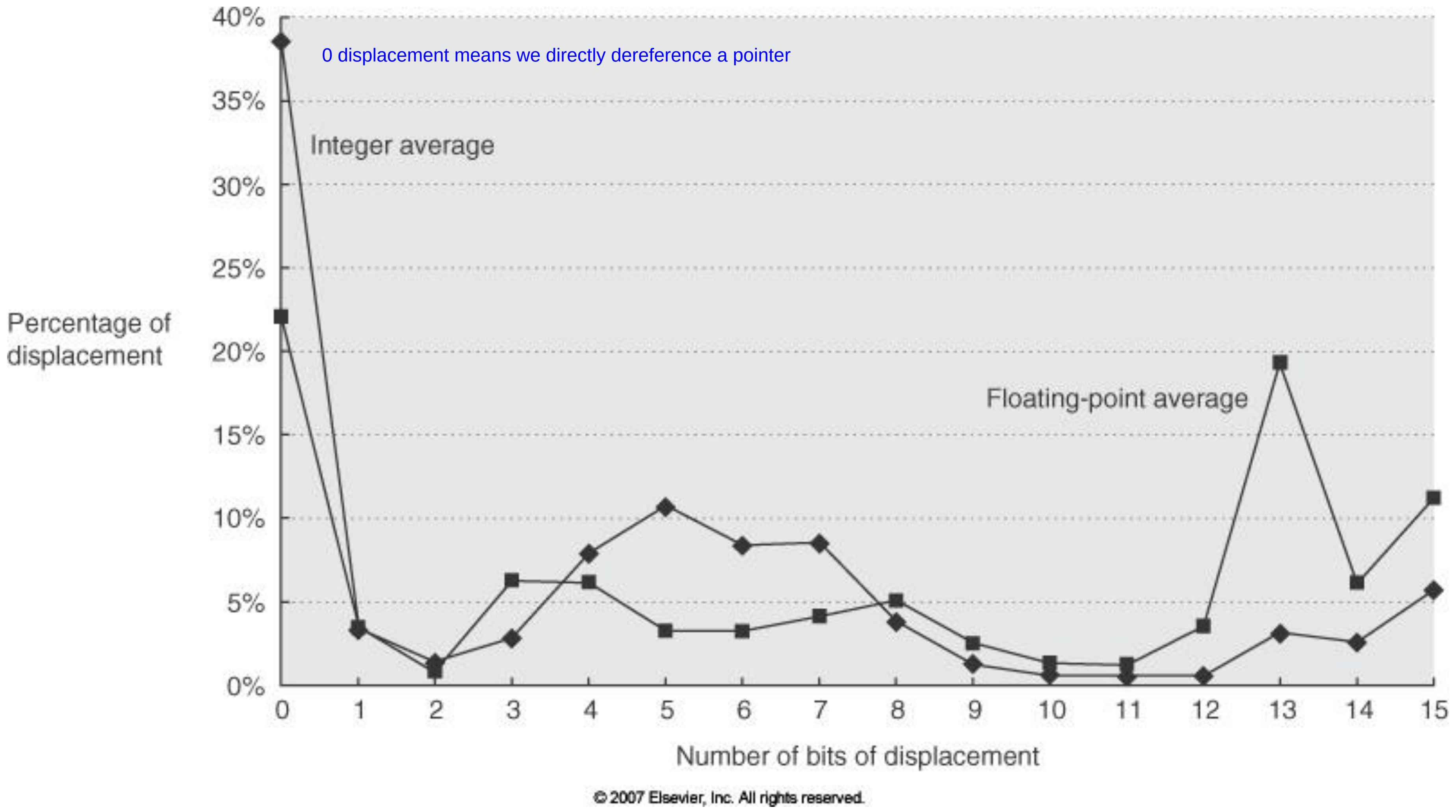
I-type	Op(6)	Rs(5)	Rt(5)	Displace(16)

  - Is 16-bits enough? this is a design choice
  - Yes? VAX experiment showed 1% accesses use displacement >16 very small, therefore, to make the common case fast, implement 16 bits
- SPARC adds Reg+Reg mode
  - Why? What impact on both implementation and instr count?

Alpha displacement values (SPEC2000)

- Alpha has a 16-bit displacement field

  another RISC ISA



x86 addressing modes examples

supports a wider range of addressing modes

Prefix*(1-5)	Op	OpX*	RM*	SIB*	Disp*(1-4)	Imm*(1-4)

- **Absolute**: zero + displace(8/16/32-bit)
- **Register indirect**: R1
- **Indexed**: R1+R2
- **Displacement**: R1+ displace(8/16/32-bit)
- **Based Index:** R1+R2 + displace(8/16-bit)
- **Scaled**: R1 + (R2*Scale)
  Scale = 1, 2, 4, 8
- **Scaled w/Displacement:**

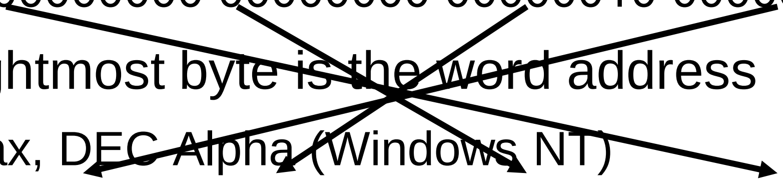
  R1 + (R2*Scale) + displace(8/16/32-bit)
  Scale = 1, 2, 4, 8

5. Memory addressing: Little vs big endian

- **Endian-ness**: ordering of bytes in a word
  - Big Endian: leftmost byte is the word address (sensible order)

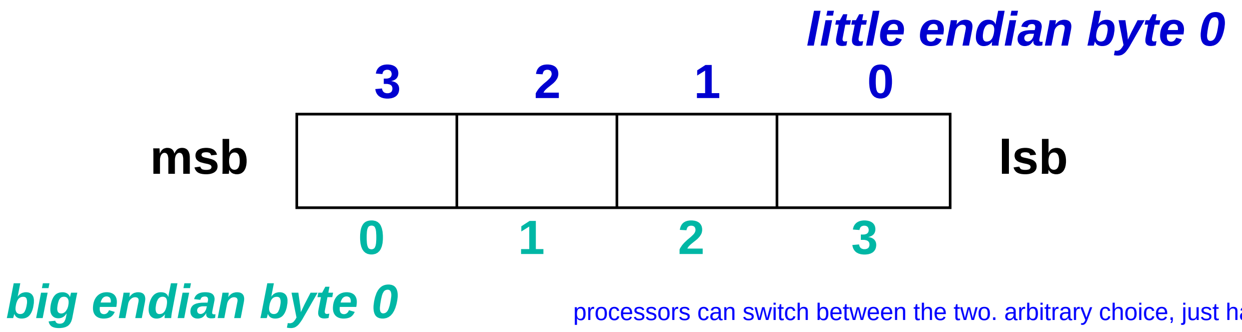
    word size = 4, but we can address each byte separately. so, must specify how bytes are laid out in words

    IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA, PowerPC
    - A 4-byte integer: “00000000 00000000 00000010 00000011” is 515



  - Little Endian: rightmost byte is the word address

    Intel 80x86, DEC Vax, DEC Alpha (Windows NT)
    - A 4-byte integer: “00000011 00000010 00000000 00000000 ” is 515
    - Why little endian?

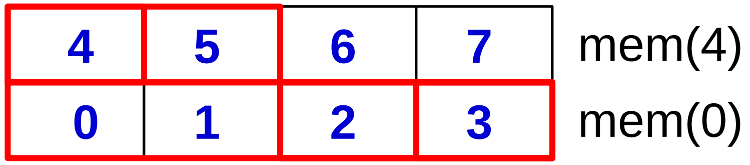


processors can switch between the two. arbitrary choice, just have to be consistent

- Terms coined by Danny Cohen, *Computer*, 1981
  en.wikipedia.org/wiki/Endianness

Aligned vs nonaligned addresses

- **Alignment restriction** - the memory address of a word must be on natural word boundaries (a multiple of 4 in MIPS-32)
  - If the address is not aligned, then two word have to be read from memory, the appropriate parts extracted from each word, and then those parts assembled
- **Access alignment**:
  - Aligned: `load-word @XXXX00`, `load-half @XXXXX0`
  - Unaligned: `load-word @XXXX10`, `load-half @XXXXX1`



ISA does not talk about cache, but forcing alignment is about cache and increasing locality

  - Question: what to do with unaligned (word) accesses (uncommon case)?
    - Support in hardware? Makes all accesses slow
    - Trap to software routine? Possibility
    - Do two loads and compose (`load`, `shift`, `load`, `shift`, `and`)
    - **MIPS? ISA support**: unaligned access using **two** instructions

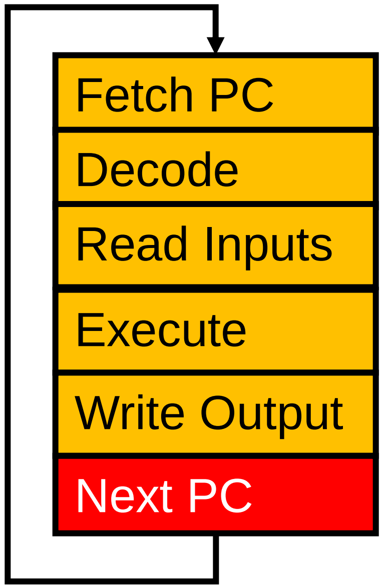
      `lwl @XXXX10; lwr @XXXX10`

6. Operations in the instruction set

- Rule of thumb: the most widely executed instr's should be the simplest (i.e., fastest to execute) operations in the ISA (make the common case fast)

Rank	x86 Instr	SPECint92 Avg
1	load	22%
2	cond branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	5%
7	sub	4%
8	move reg-reg	4%
9	call	1%
10	return	1%

7. Control flow



- Default next-PC is PC + sizeof(current instr)

  fixed size encoding: very easy. otherwise, more complicated

- Branches and jumps can change that
  - Otherwise dynamic program == static program

    branches are not too common to make this fast

- **Computing targets**: where to jump to
  - For all branches and jumps
  - Absolute / PC-relative / indirect

- **Testing conditions**: whether to jump at all
  - For (conditional) branches only
  - Compare-branch / condition-codes / condition registers

Control flow I: Computing targets

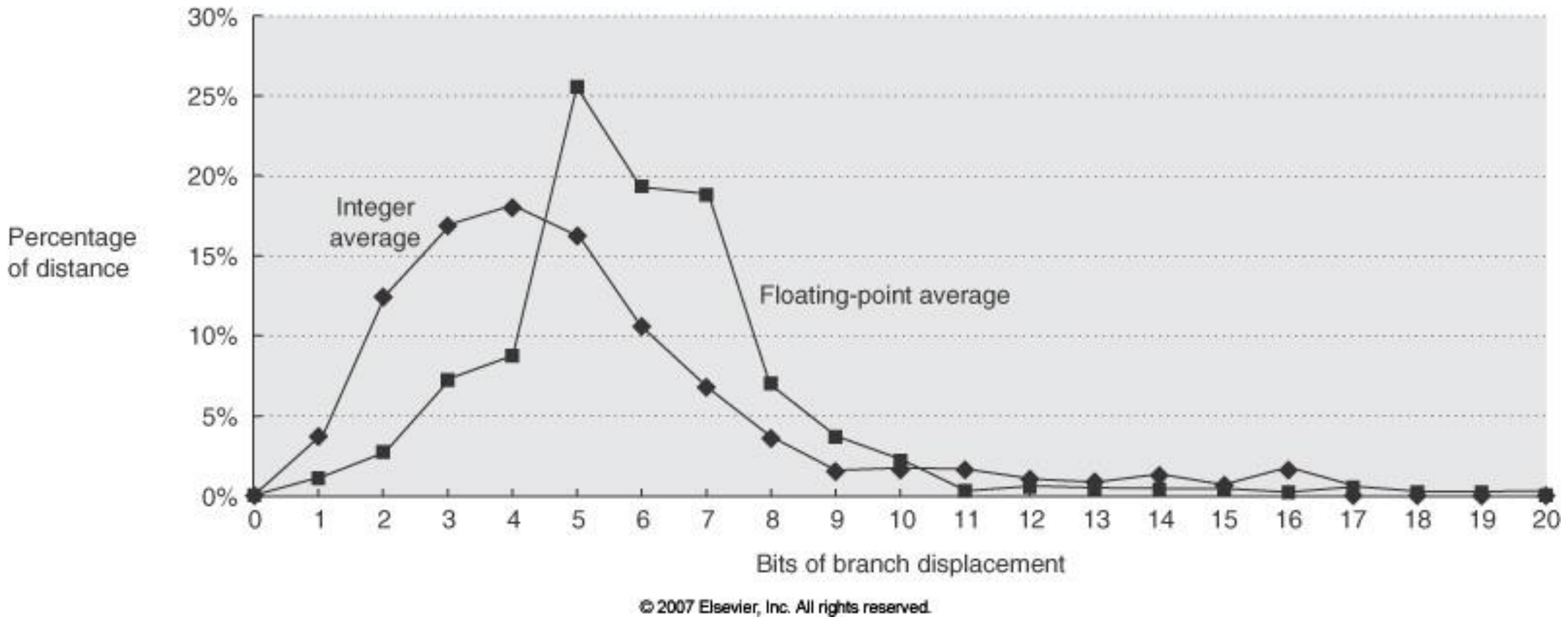
- The issues
  - How far (statically) do you need to jump?
    - Not far within procedure, further from one procedure to another
  - Do you need to jump to a different place each time?
- **PC-relative**

  a nice property to have
  PC +/- distance
  - Position-independent within procedure
  - Used for branches and jumps within a procedure
- **Absolute**

  not always PC-relative possible, so absolute is necessary
  - Position independent outside procedure (linker-loader tracks)
  - Used for procedure calls
- **Indirect** (target found in register, e.g., **`jr`** in MIPS)
  - Needed for jumping to dynamic targets
  - Used for **returns**, dynamic procedure calls, `switch` statements

Branch distances (Alpha)

- Distribution of displacements for PC-relative branches
  - About 75% of branches are in the forward direction



Control flow II: Testing conditions

various choices across various processors (and corresponding ISAs)

- **Compare and branch (VAX)**

```
branch-less-than   R1,10,target
```

  - + Simple, one instr rather than two for the branch
  - – Need two ALUs: one for condition, one for target address
  - – Extra latency

- **Implicit condition codes (CC) (x86)**

```
subtract     R2,R1,10   // sets “negative” CC
branch-neg   target
```

uses FLAGs

  - + Condition Codes set “for free”
  - – CC is extra (machine) state; implicit dependence is tricky

- **Conditions in regs, separate branch (MIPS)**

```
set-less-than             R2,R1,10
branch-not-equal-zero R2,target
```

  - – Additional instr’s
  - + one ALU per instr *and* explicit dependence

MIPS and x86 control flows

- MIPS

I-type

Op(6)	Rs(5)	Rt(5)	Immed(16)

  - 16-bit offset PC-relative conditional branches
  - **Uses registers for condition**
    - Compare two regs: **`beq`**, **`bne`**
    - Compare reg to 0 (Reg0): **`bgtz`**, **`bgez`**, **`bltz`**, **`blez`**

these cover most of the branches, easy to implement

  - Why?
    - More than 80% of branches are (in)equalities or comparisons to 0
    - Don't need adder for these cases (fast, simple)
    - OK to take two instr's to do remaining branches
      - It's the uncommon case
      - Explicit "set condition into registers" instructions: **`slt`**, **`sltu`**, **`slti`**, **`sltiu`**, …
- x86
  - 8-bit offset PC-relative branches
  - **Uses Condition Codes**
    - Explicit compare instructions (and others) to set Condition Codes

ISA's also include support for…

- Operating systems & memory protection
  - Privileged mode
  - System call (trap)
  - Exceptions & interrupts
  - Interacting with I/O devices

- Multiprocessor/multithreaded support
  - "Atomic" operations for synchronization

- Data-level parallelism
  - Pack many values into a wide register
    - Intel's SSE, SSE2, … AVX: multiple 32-bit float-point values into 128-512 bit registers
  - Define parallel operations (e.g. four "adds" in one cycle for SSE2)

RISC vs CISC

- Recall the performance equation:

  (instructions/program) * (cycles/instruction) * (seconds/cycle)

- **CISC** (Complex Instruction Set Computing)
  - Reduce “instr’s/program” with “complex” instructions
    - But tends to increase “cycles/instr” or clock period (“seconds/cycle”)
  - Easy for assembly-level programmers, good code density
- **RISC** (Reduced Instruction Set Computing)
  - Improve “cycles/instruction” with many single-cycle instructions
  - Increases “instr’s/program”, but hopefully not much
    - Help from smart compiler
  - Perhaps improve clock period (“seconds/cycle”)
    - via aggressive implementation allowed by simpler instructions

The RISC tenets

- **Single-cycle execution**
  - CISC: many multicycle operations
- **Hardwired control** in CISC, a lot of 1-cycle instructions, just not part of ISA, directly part of microcoded ROM
  - CISC: microcoded multi-cycle operations
- **Load/store architecture**
  - CISC: register-memory and memory-memory
- **Few memory addressing modes**
  - CISC: many modes
- **Fixed-length instruction format**
  - CISC: many formats and lengths
- **Reliance on compiler optimizations**
  - CISC: hand assemble to get good performance
- **Many registers** (compilers are better at using them)
  - CISC: few registers

RISC “won” the technology battles

RISC won the embedded computing war

CISC won the high-end commercial war (1990’s to today)

Compatibility is a stronger force than anyone (but Intel) thought

Intel's compatibility trick: RISC inside

- 1993: Intel wanted out-of-order (OoO) execution in Pentium Pro
  - Hard to do with a coarse grain ISA like x86
- Solution? Translate x86 to RISC **μops** in hardware

```
push  $eax
```

two instruction caches: one for x86 instructions, one for translated RISC-like uops

becomes (we think, uops are proprietary)

```
store $eax,[$esp-4]
addi  $esp,$esp,-4
```

maintains the same contract but implements differently

  + Processor maintains **x86 ISA externally for compatibility**
  + But executes **RISC μISA internally for implementability**

because front end of an x86 machine: very complicated

  - Given a translator, x86 almost as easy to implement as RISC
    - Intel implemented out-of-order before any RISC company
    - Also, OoO also benefits x86 more (because ISA limits compiler)
  - Idea co-opted by other x86 companies: AMD and Transmeta

Aside: More about x86 micro-ops

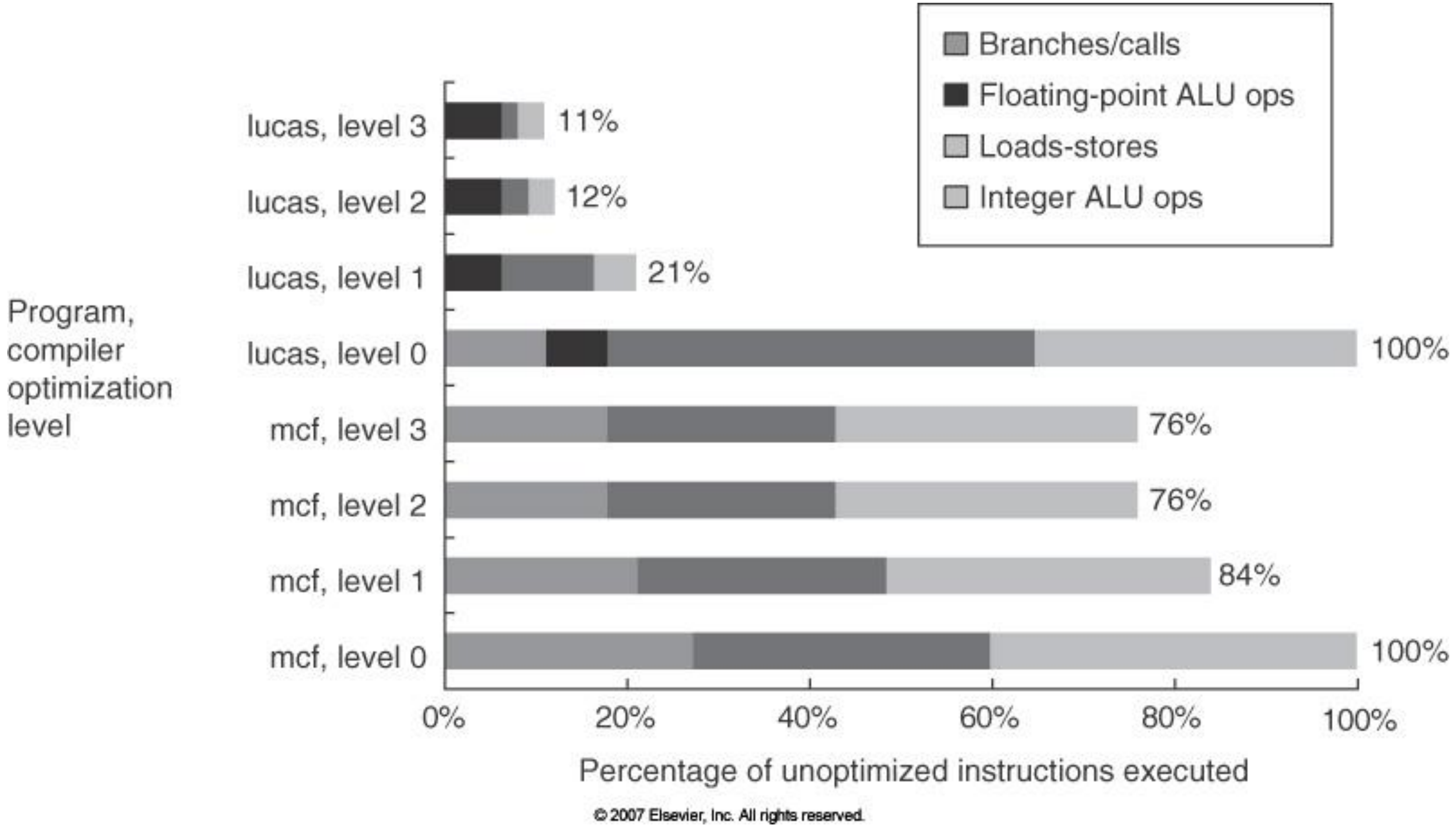
- Even better? Two forms of hardware translation
  - Hard-coded logic: fast, but complex
  - Table: slow, but “off to the side”, doesn’t complicate rest of machine
- x86: average 1.6 μops / x86 instr
  - Logic for common instr’s that translate into 1–4 μops
  - Table for rare instr’s that translate into 5+ μops
- x86-64: average 1.1 μops / x86 instr
  - More registers (can pass parameters too), fewer **`pushes/pops`**
  - Core 2: logic for 1–2 μops, Table for 3+ μops?
- More recent: “macro-op fusion” and “micro-op fusion”
  - Intel’s recent processors fuse certain instruction pairs

ISAs and compiler optimizations

- Compiler optimizations primarily reduce dynamic instr count
  - Eliminate redundant computation, keep more things in registers
    - + Registers are faster, fewer loads/stores
    - – An ISA can make this difficult by having too few registers
- But also…
  - Reduce branches and jumps
  - Reduce cache misses
  - Reduce dependences between nearby instr's (for parallelism)
    - – An ISA can make this difficult by having implicit dependences
- How effective are these?
  - + Can give 4X performance over unoptimized code
  - – Collective wisdom of 40 yrs ("Proebsting's Law"): 4% per yr
  - Funny but … shouldn't leave 4X performance on the table

if ISA allows, and compiler knows, can optimize and produce very good machine code

Example: Compiler optimization impact on IC



ISA implementability

- Every ISA can be implemented
  - But … not every ISA can be implemented efficiently

- Classic high-performance implementation techniques
  - Pipelining, parallel execution, out-of-order execution (more later)

- Certain ISA features make these difficult
  - Variable instruction lengths/formats: complicate decoding
  - Implicit state & dependence: complicates dynamic scheduling
  - Variable latencies: complicates scheduling
  - Difficult to interrupt instructions: complicates many things
    - Example: memory copy instruction

ISA compatibility

- In many domains, ISA *must* remain compatible
  - IBM’s 360/370 (the *first* “ISA family”), Intel’s x86 and Microsoft Windows
- **Backward compatibility** – New processors supporting old programs
  - Can’t drop features (cumbersome)
  - Or, update software/OS to emulate dropped features (slow)
- **Forward (upward) compatibility** – Old processors supporting new programs

  really relying on software so that it can generate old machine codes based on this ID
  - Include a “CPU ID” so the software can test for features
  - Add ISA hints by overloading no-ops (example: x86’s PAUSE)

    not discussed in class
  - New firmware/software on old processors to emulate new instruction not discussed in class

Discussion Prompts for F3, F4

- “A Characterization of the Processor Performance in the VAX 11/78” by Emer and Clark
  - Q1: A read stall (cache miss) takes 6 cycles on the 11/780. Compare this to the case for machines today. What impact does this have on machine designs of yesterday vs today?
  - Q2: “Table 8 shows where 11/780 performance may be improved. For example, saving the non-overlapped Decode cycle …” What two other instruction-cycles would you focus on if you were the 11/780 designer to improve machine performance?

- “Evolution of RISC” by John Cocke
  - Q1: What were the advantages (if any) and disadvantages (if any) of each of the changes from the improved 801 to the RS/6000? Did each of these changes make the newer processor more or less RISC-like than its predecessor?
  - Q2: Software technology had a large impact on the 801 project. Describe both the role of software simulation of the 801 and the 801’s compiler in the 801 project.