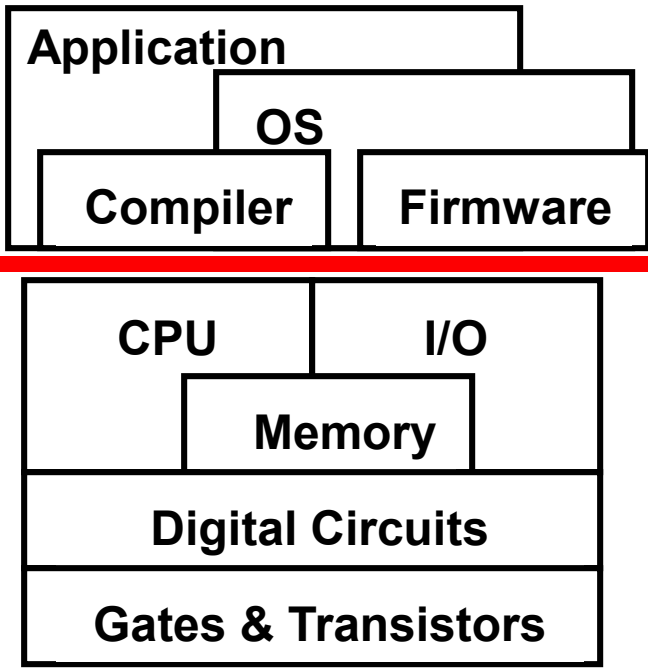


# Schedule

---

- Introduction and Transistors
- Parallel computing (Isaac D. Scherson, University of California, Irvine, in October)
- ISAs
- Performance
- Pipelining Basic
- Branch Prediction
- Caches
- Virtual Memory
- Out-of-Order Execution
- Multicore multi-thread
- Vectors/GPUs for data parallelism.

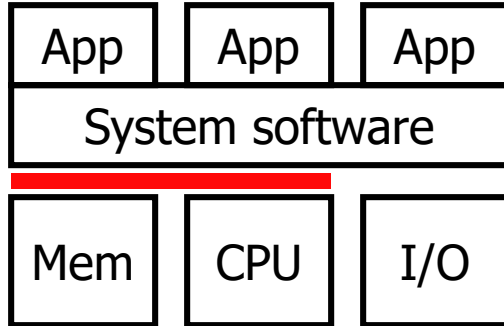
# Instruction Set Architecture (ISA)



- What is an ISA?
  - A functional contract
- All ISAs similar in high-level ways
  - But many design choices in details
  - Two “philosophies”: CISC/RISC
    - Difference is blurring
- A Good ISA...
  - Enables high-performance
  - At least doesn't get in the way
- Compatibility is a powerful force
  - Tricks: binary translation,  $\mu$ ISAs

# Execution Model

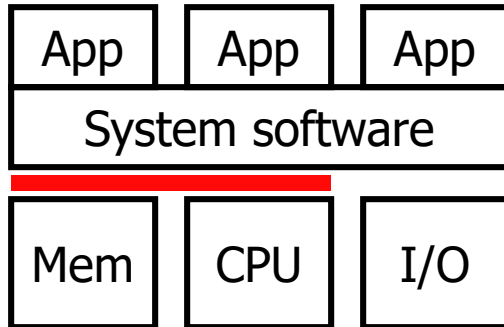
# Program Compilation



```
int array[100], sum;
void array_sum() {
    for (int i=0; i<100;i++) {
        sum += array[i];
    }
}
```

- **Program** written in a “high-level” programming language
  - C, C++, Java, C#
  - Hierarchical, structured control: loops, functions, conditionals
  - Hierarchical, structured data: scalars, arrays, pointers, structures
- **Compiler**: translates program to **assembly**
  - Parsing and straight-forward translation
  - Compiler also optimizes
  - Compiler is itself a program...who compiled the compiler?

# Assembly & Machine Language

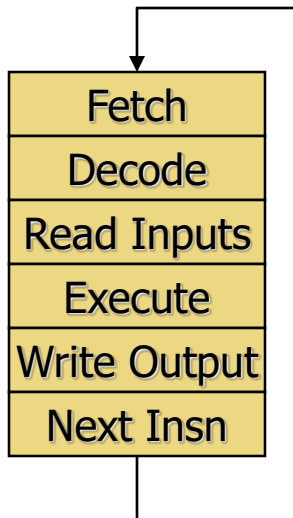
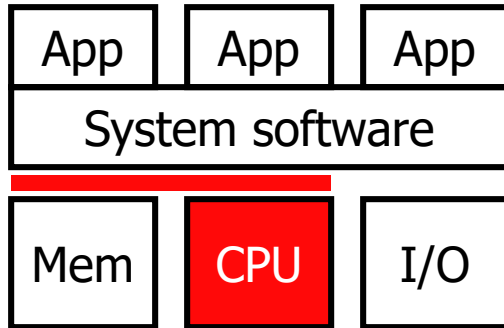


- **Assembly language**
  - Human-readable representation
- **Machine language**
  - Machine-readable representation
  - 1s and 0s (often displayed in “hex”)
- **Assembler**
  - Translates assembly to machine

<u>Machine code</u>	<u>Assembly code</u>
x9A00	CONST R5, #0
x9200	CONST R1, array
xD320	HICONST R1, array
x9464	CONST R2, sum
xD520	HICONST R2, sum
x6640	LDR R3, R1, #0
x6880	LDR R4, R2, #0
x18C4	ADD R4, R3, R4
x7880	STR R4, R2, #0
x1261	ADD R1, R1, #1
x1BA1	ADD R5, R5, #1
x2B64	CMPI R5, #100
x03F8	BRn array_sum_loop

In a  
**instruction set architecture, or ISA**

# Instruction Execution Model



**Instruction → Insn**

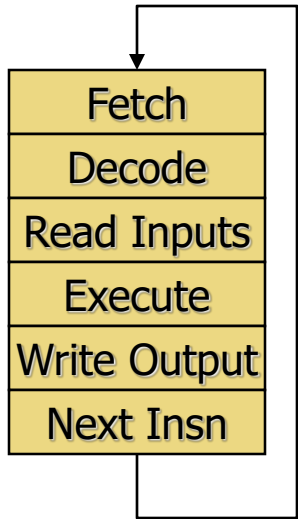
- A computer is just a finite state machine
  - **Registers** (few of them, but fast)
  - **Memory** (lots of memory, but slower)
  - **Program counter** (next insn to execute)
    - Called *instruction pointer* (IP) in x86
- A computer executes **instructions**
  - **Fetches** next instruction from memory
  - **Decodes** it (figure out what it does)
  - **Reads** its **inputs** (registers & memory)
  - **Executes** it (adds, multiply, etc.)
  - **Write** its **outputs** (registers & memory)
  - **Next insn** (adjust the program counter)
- **Program is just “data in memory”**
  - Makes computers programmable (“universal”)

# What Is An ISA?

---

- **ISA (instruction set architecture)**
  - A well-defined hardware/software **interface**
  - The “**contract**” between software and hardware
    - **Functional definition** of storage locations & operations
      - Storage locations: registers, memory
      - Operations: add, multiply, branch, load, store, etc
    - **Precise description** of how to invoke & access them
- Not in the contract: non-functional aspects
  - How operations are implemented
  - Which operations are fast and which are slow
  - Which operations take more power and which take less
- Instructions (Insns)
  - Bit-patterns hardware interprets as commands

# The Sequential Model



- **Basic structure of all modern ISAs**
  - Often called Von Neumann, but in ENIAC before
- **Program order**: total order on dynamic insns
  - Order and **named storage** define computation
- Convenient feature: **program counter (PC)**
  - Insn itself stored in memory at location pointed to by PC
  - Next PC is next insn unless insn says otherwise
- Processor logically executes loop at left
- **Atomic**: insn finishes before next insn starts
  - Implementations can break this constraint physically
  - But must maintain illusion to preserve correctness



# What Makes a Good ISA?

---

- **Programmability**
  - Easy to express programs efficiently?
- **Performance/Implementability**
  - Easy to design high-performance implementations?
  - Easy to design low-power implementations?
  - Easy to design low-cost implementations?
- **Compatibility**
  - Easy to maintain as languages, programs, and technology evolve?
  - x86 (IA32) generations: 8086, 286, 386, 486, Pentium, PentiumII, PentiumIII, Pentium4, Core2, Core i7, ...

# Programmability

---

- Easy to express programs efficiently?
  - For whom?
- Before 1980s: **human**
  - Compilers were terrible, most code was hand-assembled
  - Want high-level coarse-grain instructions
    - As similar to high-level language as possible
- After 1980s: **compiler**
  - Optimizing compilers generate much better code than you or I
  - Want low-level fine-grain instructions
    - Compiler can't tell if two high-level idioms match exactly or not
- This shift changed what is considered a “good” ISA...

# Performance, Performance, Performance

- How long does it take for a program to execute?
  - Three factors
- 1. How many insn must execute to complete program?
  - **Instructions per program** during execution
  - “Dynamic insn count” (not number of “static” insns in program)
- 2. How quickly does the processor “cycle”?
  - **Clock frequency** (cycles per second)      1 gigahertz (Ghz)
  - or expressed as reciprocal, **Clock period**      nanosecond (ns)
  - Worst-case delay through circuit for a particular design
- 3. How many *cycles* does each instruction take to execute?
  - **Cycles per Instruction** (CPI) or reciprocal, **Insn per Cycle** (IPC)

$$\text{Execution time} = (\text{instructions/program}) * (\text{seconds/cycle}) * (\text{cycles/instruction})$$

# Maximizing Performance

$$\text{Execution time} = (\text{instructions/program}) * (\text{seconds/cycle}) * (\text{cycles/instruction})$$

$$(1 \text{ billion instructions}) * (1\text{ns per cycle}) * (1 \text{ cycle per insn}) \\ = 1 \text{ second}$$

- Instructions per program:
  - Determined by program, compiler, instruction set architecture (ISA)
- Cycles per instruction: "CPI"
  - Typical range today: 2 to 0.5
  - Determined by program, compiler, ISA, micro-architecture
- Seconds per cycle: "clock period"
  - Typical range today: 2ns to 0.25ns
  - Reciprocal is frequency: 0.5 Ghz to 4 Ghz (1 Hz = 1 cycle per sec)
  - Determined by micro-architecture, technology parameters
- For minimum execution time, minimize each term
  - Difficult: *often pull against one another*

# Example: Instruction Granularity

$$\text{Execution time} = (\text{instructions/program}) * (\text{seconds/cycle}) * (\text{cycles/instruction})$$

- **CISC** (Complex Instruction Set Computing) **ISAs**
  - Big heavyweight instructions (lots of work per instruction)
  - + Low “insns/program”
  - Higher “cycles/insn” and “seconds/cycle”
    - We have the technology to get around this problem
- **RISC** (Reduced Instruction Set Computer) **ISAs**
  - Minimalist approach to an ISA: simple insns only
  - + Low “cycles/insn” and “seconds/cycle”
  - Higher “insn/program”, but hopefully not as much
    - Rely on compiler optimizations

# Compiler Optimizations

---

- Primarily goal: reduce instruction 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 (later)
  - Reduce cache misses (later)
  - Reduce dependences between nearby insns (later)
    - 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 years (“Proebsting’s Law”): 4% per year
  - Funny but ... shouldn’t leave 4X performance on the table

# Compatibility

---

- In many domains, ISA must remain compatible
  - IBM's 360/370 (the *first* "ISA family")
  - Another example: Intel's x86 and Microsoft Windows
    - x86 one of the worst designed ISAs EVER, but it survives
- **Backward compatibility**
  - New processors supporting old programs
    - **Hard to drop features**
    - Use software/OS to emulate dropped features (slow)
- **Forward (upward) compatibility**
  - Old processors supporting new programs
    - Include a "CPU ID" so the software can test for features
    - Add ISA hints by overloading no-ops (example: x86's PAUSE)
    - New firmware/software on old processors to emulate new insn

# Translation and Virtual ISAs

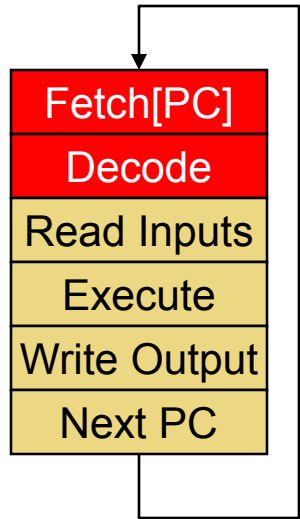
---

- New compatibility interface: ISA + translation software
  - **Binary-translation**: transform static image, run native
  - **Emulation**: unmodified image, interpret each dynamic insn
    - Typically optimized with just-in-time (JIT) compilation
  - Examples: FX!32 (x86 on Alpha), Rosetta (PowerPC on x86)
  - Performance overheads reasonable (many advances over the years)
- **Virtual ISAs**: designed for translation, not direct execution
  - Target for high-level compiler (one per language)
  - Source for low-level translator (one per ISA)
  - Goals: Portability (abstract hardware nastiness), flexibility over time
  - Examples: Java Bytecodes, C# CLR (Common Language Runtime), NVIDIA's "PTX"



# ISA Details

# Length and Format



- **Length**

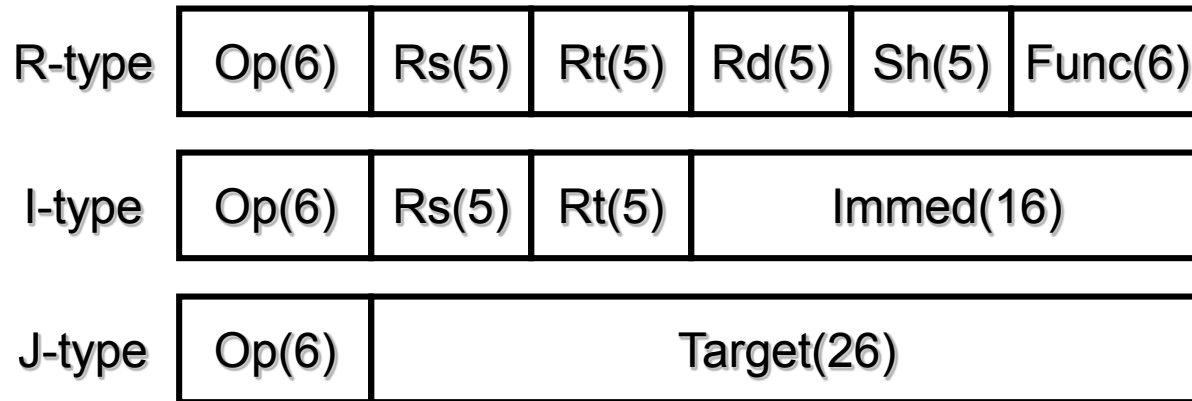
- Fixed length
  - Most common is 32 bits
    - + Simple implementation (next PC often just PC+4)
    - Code density: 32 bits to increment a register by 1
- Variable length
  - + Code density
    - x86 averages 3 bytes (ranges from 1 to 16)
    - Complex fetch (where does next instruction begin?)
- Compromise: two lengths
  - E.g., ARM's Thumb (16 bits)

- **Encoding**

- A few simple encodings simplify decoder
  - x86 decoder one nasty piece of logic

# Example Instruction Encodings

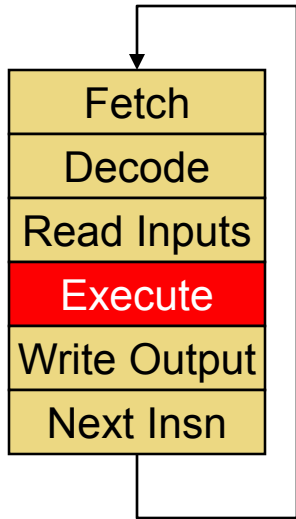
- MIPS
  - Fixed length
  - 32-bits, 3 formats, simple encoding



- x86
  - Variable length encoding (1 to 15 bytes)

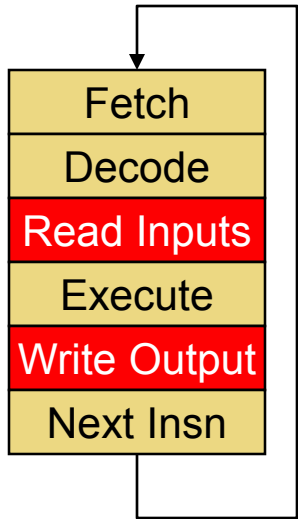


# Operations and Datatypes



- Datatypes
  - Software: attribute of data
  - Hardware: attribute of operation, data is just 0/1's
- All processors support
  - Integer arithmetic/logic (8/16/32/64-bit)
  - IEEE754 floating-point arithmetic (32/64-bit)
- More recently, most processors support
  - "Packed-integer" insns, e.g., MMX
  - "Packed-floating point" insns, e.g., SSE/SSE2/AVX
  - For "data parallelism", more about this later
- Other, infrequently supported, data types
  - Decimal, fixed-point arithmetic, strings

# Where Does Data Live?



- **Registers**

- "short term memory"
- Faster than memory, quite handy
- Named directly in instructions

- **Memory**

- "longer term memory"
- Accessed via "addressing modes"
  - Address to read or write calculated by instruction

- "Immediates"

- Values spelled out as bits in instructions
- Input only

# 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)
- Another: they are **directly addressed** (no address calc)
  - More registers, means more bits per register in instruction
  - Thus, fewer registers per instruction or larger instructions
- **Not everything can be put in registers**
  - Although compilers are getting better at putting more things in
- More registers means **more saving/restoring**
  - Across function calls, traps, and context switches
- Trend toward more registers:
  - 8 (x86) → 16 (x86-64), 16 (ARM v7) → 32 (ARM v8)

# Memory Addressing

- **Addressing mode:** way of specifying address
  - Used in memory-memory or load/store instructions in register ISA
- Examples
  - **Displacement:**  $R1 = \text{mem}[R2 + \text{immed}]$
  - **Index-base:**  $R1 = \text{mem}[R2 + R3]$
  - **Memory-indirect:**  $R1 = \text{mem}[\text{mem}[R2]]$
  - **Auto-increment:**  $R1 = \text{mem}[R2], R2 = R2 + 1$
  - **Auto-indexing:**  $R1 = \text{mem}[R2 + \text{immed}], R2 = R2 + \text{immed}$
  - **Scaled:**  $R1 = \text{mem}[R2 + R3 * \text{immed1} + \text{immed2}]$
  - **PC-relative:**  $R1 = \text{mem}[\text{PC} + \text{imm}]$
- What high-level program idioms are these used for?
- What implementation impact? What impact on insn count?

# Addressing Modes Examples

- MIPS I-type 

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

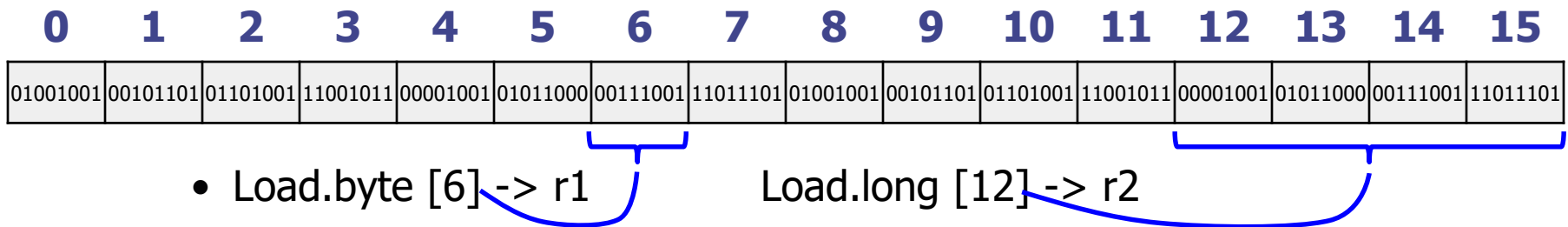
  - **Displacement**:  $R1 + \text{offset}$  (16-bit)
  - Why? Experiments on VAX (ISA with every mode) found:
    - 80% use small displacement (or displacement of zero)
    - Only 1% accesses use displacement of more than 16bits
- Other ISAs (SPARC, x86) have reg+reg mode, too
  - Impacts both implementation and insn count (How?)
- x86 (MOV instructions)
  - **Absolute**: zero + offset (8/16/32-bit)
  - **Register indirect**:  $R1$
  - **Displacement**:  $R1 + \text{offset}$  (8/16/32-bit)
  - **Indexed**:  $R1 + R2$
  - **Scaled**:  $R1 + (R2 * \text{Scale}) + \text{offset}$  (8/16/32-bit)    Scale = 1, 2, 4, 8



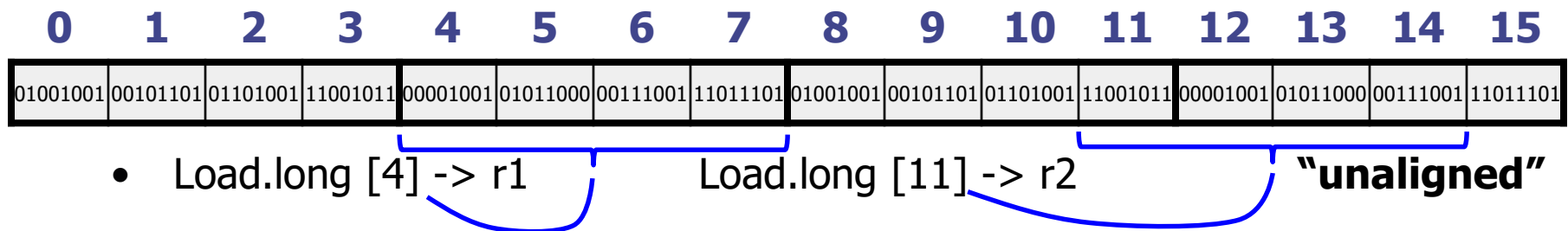
# Access Granularity & Alignment

- **Byte addressability**

- An address **points to** a byte (8 bits) of data
- The ISA's minimum granularity to read or write memory
- ISAs also support wider load/stores
  - "Half" (2 bytes), "Longs" (4 bytes), "Quads" (8 bytes)



However, physical memory systems operate on **even larger chunks**



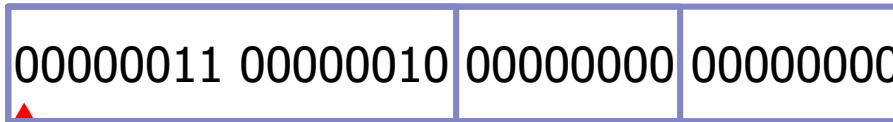
- **Access alignment:** if  $\text{address} \% \text{size} \neq 0$ , then it is "unaligned"
  - A single unaligned access may require multiple physical memory accesses

# Handling Unaligned Accesses

- **Access alignment:** if  $\text{address} \% \text{size} \neq 0$ , then it is “unaligned”
  - A single unaligned access may require multiple physical memory accesses
- How to handle such unaligned accesses?
  1. Disallow (unaligned operations are considered illegal)
    - MIPS, ARMv5 and earlier took this route
  2. Support in hardware? (allow such operations)
    - x86, ARMv6+ allow regular loads/stores to be unaligned
      - Unaligned access still slower, adds significant hardware complexity
  3. Trap to software routine?
    - Simpler hardware, but high penalty when unaligned
  4. In software (compiler can use regular instructions when possibly unaligned)
    - Load, shift, load, shift, and (slow, needs help from compiler)

# Another Addressing Issue: Endian-ness

- **Endian-ness**: arrangement of bytes in a multi-byte number
  - Big-endian: sensible order (e.g., MIPS, PowerPC, ARM)
    - 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?



starting address

integer casts are free on  
little-endian architectures

# Operand Model: Register or Memory?

- “Load/store” architectures
  - Memory access instructions (loads and stores) are distinct
  - Separate addition, subtraction, divide, etc. operations
  - Examples: MIPS, ARM, SPARC, PowerPC
- Alternative: mixed operand model (x86, VAX)
  - Operand can be from register **or** memory
  - x86 example: `addl 100, 4(%eax)`
    - 1. Loads from memory location `[4 + %eax]`
    - 2. Adds “100” to that value
    - 3. Stores to memory location `[4 + %eax]`
    - Would require three instructions in MIPS, for example.

# x86 Operand Model: Accumulators


- x86 uses explicit accumulators
  - Both register and memory
  - Distinguished by addressing mode

```
.LFE2
.comm array,400,32
.comm sum,4,4


.globl array_sum
array_sum:
    movl $0, -4(%rbp)
```

```
.L1:
    movl -4(%rbp), %eax
    movl array(,%eax,4), %edx
    movl sum(%rip), %eax
    addl %edx, %eax
    movl %eax, sum(%rip)
    addl $1, -4(%rbp)
    cmpl $99, -4(%rbp)
    jle .L1
```

Register accumulator:  $\%eax = \%eax + \%edx$



Memory accumulator:  
 $\text{Memory}[\%rbp-4] = \text{Memory}[\%rbp-4] + 1$

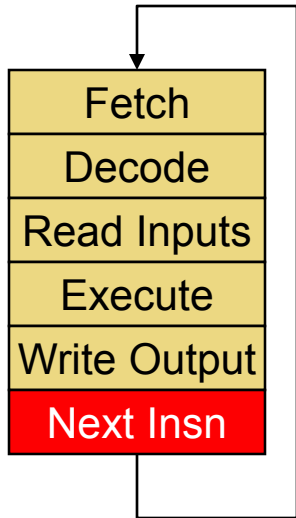


# How Much Memory? Address Size

---

- What does “64-bit” in a 64-bit ISA mean?
  - **Each program can address (i.e., use)  $2^{64}$  bytes**
  - 64 bits is the size of a **virtual address (VA)**
  - Alternative (wrong) definition: width of arithmetic operations
- 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 Pentium4)
- All modern ISAs are at 64 bits

# Control Transfers



- Default next-PC is  $PC + \text{sizeof}(\text{current insn})$ 
  - Branches and jumps can change that
- **Computing targets:** where to jump to
  - For all branches and jumps
  - **PC-relative:** for branches and jumps with function
  - **Absolute:** for function calls
  - **Register indirect:** for returns, switches & dynamic calls
- **Testing conditions:** whether to jump or not
  - **Implicit condition codes or "flags" (ARM, x86)**  
`cmp R1,10 // sets "negative" flag`  
`branch-neg target`
  - **Use registers & separate branch insns (MIPS)**  
`set-less-than R2,R1,10`  
`branch-not-equal-zero R2,target`

# ISAs Also Include Support For...

---

- Function calling conventions
  - Which registers are saved across calls, how parameters are passed
- Operating systems & memory protection
  - Privileged mode
  - System call (TRAP)
  - Exceptions & interrupts
  - Interacting with I/O devices
- Multiprocessor support
  - “Atomic” operations for synchronization
- Data-level parallelism
  - Pack many values into a wide register
    - Intel’s SSE2: four 32-bit float-point values into 128-bit register
  - Define parallel operations (four “adds” in one cycle)



# The RISC vs. CISC Debate

# RISC and CISC

---

- **RISC**: reduced-instruction set computer
  - Coined by Patterson in early 80's
  - RISC-I (Patterson), MIPS (Hennessy), IBM 801 (Cocke)
  - Examples: PowerPC, ARM, SPARC, Alpha, PA-RISC
- **CISC**: complex-instruction set computer
  - Term didn't exist before "RISC"
  - Examples: x86, VAX, Motorola 68000, etc.
- Philosophical war started in mid 1980's
  - RISC "won" the technology battles
  - CISC "won" the high-end commercial space (1990s to today)
    - Compatibility, process technology edge
  - RISC "winning" the mobile computing space

# CISCs and RISCs

---

- The CISCiest: VAX (**V**irtual **A**ddress **eX**tension to PDP-11)
  - Variable length instructions: 1-321 bytes!!!
  - 14 registers + PC + stack-pointer + condition codes
  - Data sizes: 8-, 16-, 32-, 64-, 128-bit, decimal, string
  - Memory-memory instructions for all data sizes
  - Special insns: `crc`, `insque`, `polyf`, and a cast of hundreds
- x86: “Difficult to explain and impossible to love”
  - variable length insns: 1-15 bytes

# CISCs and RISCs

---

- The RISCs: MIPS, PA-RISC, SPARC, PowerPC, Alpha, ARM
  - 32-bit instructions
  - 32 integer registers, 32 floating point registers
  - Load/store architectures with few addressing modes
  - Why so many basically similar ISAs? Everyone wanted their own

# The RISC Design Tenets

---

- **Single-cycle execution**
  - CISC: many multicycle operations
- **Hardwired (simple) control**
  - CISC: **microcode** for 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 can use them effectively)
  - CISC: few registers

# The Debate

---

- RISC argument
  - CISC is fundamentally handicapped
  - For a given technology, RISC implementation will be better (faster)
    - Current technology enables single-chip RISC
    - When it enables single-chip CISC, RISC will be pipelined
    - When it enables pipelined CISC, RISC will have caches
    - When it enables CISC with caches, RISC will have next thing...
- CISC rebuttal
  - CISC flaws not fundamental, can be fixed with **more transistors**
  - Moore's Law will narrow the RISC/CISC gap (true)
    - Good pipeline: RISC = 100K transistors, CISC = 300K
    - By 1995: 2M+ transistors had evened playing field
  - Software costs dominate, **compatibility** is paramount

# Intel's x86 Trick: RISC Inside

- 1993: Intel wanted “out-of-order execution” in Pentium Pro
  - Hard to do with a CISC ISA like x86
- Solution? Translate x86 to RISC micro-ops (**μops**) in hardware
  - `push $eax`  
becomes (we think, uops are proprietary)  
`store $eax, -4($esp)`  
`addi $esp, $esp, -4`
  - + Processor maintains **x86 ISA externally for compatibility**
  - + But executes **RISC μISA internally for implementability**
  - Given translator, x86 almost as easy to implement as RISC
    - Intel implemented “out-of-order” before any RISC company
    - “out-of-order” also helps x86 more (because ISA limits compiler)
    - Also used by other x86 implementations (AMD)
  - Different **μops** for different designs
    - **Not part of the ISA specification**, not publically disclosed

# Winner for Desktops/Servers: CISC

---

- x86 was first mainstream 16-bit microprocessor by ~2 years
  - IBM put it into its PCs...
  - The rest is historical inertia, Moore's law, and "financial feedback"
    - x86 is most difficult ISA to implement and do it fast but...
    - Because Intel sells the most **non-embedded** processors...
    - It hires more and better engineers...
    - Which help it maintain competitive performance ...
    - **And given competitive performance, compatibility wins...**
    - So Intel sells the most **non-embedded** processors...
  - AMD has also added pressure, e.g., beat Intel to 64-bit x86
- Moore's Law has helped Intel in a big way
  - Most engineering problems can be solved with more transistors



# Winner for Mobile: RISC

---

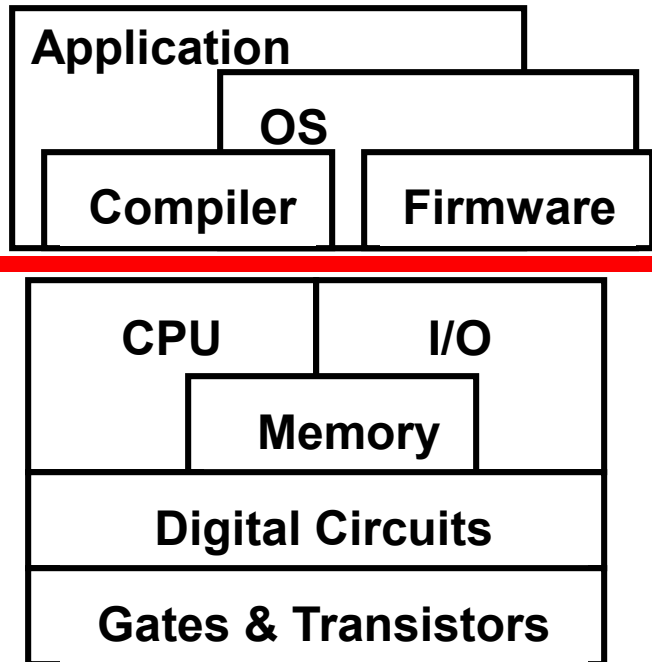
- ARM (Acorn RISC Machine → Advanced RISC Machine)
  - First ARM chip in mid-1980s (from Acorn Computer Ltd).
  - 6 billion units sold in 2010
  - Low-power and **embedded/mobile** devices (e.g., phones)
    - Significance of embedded? ISA compatibility less powerful force
- 64-bit RISC ISA
  - 32 registers, PC is one of them
  - Rich addressing modes, e.g., auto increment
  - Condition codes, each instruction can be conditional
- ARM does not sell chips; it licenses its ISA & core designs
- ARM chips from many vendors
  - Apple, Freescale (néé Motorola), Philips, Qualcomm, STMicroelectronics, Samsung, Sharp, Texas Instruments, etc.

# Redux: Are ISAs Important?

---

- Does “quality” of ISA actually matter?
  - **Mostly not**
    - Mostly comes as a design complexity issue
    - Insn/program: everything is compiled, compilers are good
    - Cycles/insn and seconds/cycle:  $\mu$ ISA, many other tricks
- Does “nastiness” of ISA matter?
  - **Mostly no**, only compiler writers and hardware designers see it
- Comparison is confounded by, e.g., transistor technology, microarchitecture design
- Even compatibility is not what it used to be
  - cloud services, virtual ISAs, interpreted languages

# Instruction Set Architecture (ISA)



- What is an ISA?
  - A functional contract
- All ISAs similar in high-level ways
  - But many design choices in details
  - Two “philosophies”: CISC/RISC
    - Difference is blurring
- Good ISA...
  - Enables high-performance
  - At least doesn't get in the way
- Compatibility is a powerful force
  - Tricks: binary translation,  $\mu$ ISAs

# Performance Metrics

# Performance: Latency vs. Throughput

- **Latency (execution time)**: time to finish a fixed task
- **Throughput (bandwidth)**: number of tasks per unit time
  - Different: exploit parallelism for throughput, not latency (e.g., bread)
  - Often contradictory (latency **vs.** throughput)
    - Will see many examples of this
  - Choose definition of performance that matches your goals
    - Scientific program? latency. web server? throughput.
- Example: move people 10 miles
  - Car: capacity = 5, speed = 60 miles/hour
  - Bus: capacity = 60, speed = 20 miles/hour
  - Latency: **car = 10 min**, bus = 30 min
  - Throughput: car = 15 PPH (count return trip), **bus = 60 PPH**
- Fastest way to send 10TB of data? (1+ gbits/second)

# Frequency as a performance metric

---

- 1 Hertz = 1 cycle per second  
1 Ghz is 1 cycle per nanosecond, 1 Ghz = 1000 Mhz
- (Micro-)architects often ignore dynamic instruction count...
- ... but general public (mostly) also ignores CPI
  - and instead equate clock frequency with performance!
- Which processor would you buy?
  - Processor A: CPI = 2, clock = 5 GHz
  - Processor B: CPI = 1, clock = 3 GHz
  - Probably A, but B is faster (assuming same ISA/compiler)
- Classic example
  - Core i7 faster clock-per-clock than Core 2
  - Same ISA and compiler!
- **partial performance metrics are dangerous!**

# MIPS (performance metric, not the ISA)

---

- (Micro) architects often ignore dynamic instruction count
  - Typically work in one ISA/one compiler → treat it as fixed
- CPU performance equation becomes
  - Latency:  $\text{seconds} / \text{insn} = (\text{cycles} / \text{insn}) * (\text{seconds} / \text{cycle})$
  - Throughput: **insn / second** =  $(\text{insn} / \text{cycle}) * (\text{cycles} / \text{second})$
- **MIPS** (millions of instructions per second)
  - **Cycles / second**: clock frequency (in MHz)
  - Example:  $\text{CPI} = 2, \text{clock} = 500 \text{ MHz} \rightarrow 0.5 * 500 \text{ MHz} = 250 \text{ MIPS}$
- Pitfall: may vary inversely with actual performance
  - Compiler removes insns, program gets faster, MIPS goes down
  - Work per instruction varies (e.g., multiply vs. add, FP vs. integer)

# Cycles per Instruction (CPI)

- **CPI**: Cycle/instruction **on average**
  - **IPC** =  $1/\text{CPI}$ 
    - Used more frequently than CPI
    - Favored because “bigger is better”, but harder to compute with
  - Different instructions have different cycle costs
    - E.g., “add” typically takes 1 cycle, “divide” takes >10 cycles
  - Depends on relative instruction frequencies
- CPI example
  - A program executes equal: integer, floating point (FP), memory ops
  - Cycles per instruction type: integer = 1, memory = 2, FP = 3
  - What is the CPI?  $(33\% * 1) + (33\% * 2) + (33\% * 3) = 2$
  - **Caveat**: this sort of calculation ignores many effects
    - Back-of-the-envelope arguments only



# CPI Example

---

- Assume a processor with instruction frequencies and costs
  - Integer ALU: 50%, 1 cycle
  - Load: 20%, 5 cycle
  - Store: 10%, 1 cycle
  - Branch: 20%, 2 cycle
- Which change would improve performance more?
  - A. “Branch prediction” to reduce branch cost to 1 cycle?
  - B. Faster data memory to reduce load cost to 3 cycles?
- Compute CPI
  - Base =  $0.5 * 1 + 0.2 * 5 + 0.1 * 1 + 0.2 * 2 = 2$  CPI

# Measuring CPI

---

- How are CPI and execution-time actually measured?
  - Execution time? stopwatch timer (Unix “time” command)
  - $\text{CPI} = (\text{CPU time} * \text{clock frequency}) / \text{dynamic insn count}$
  - How is dynamic instruction count measured?
- More useful is CPI breakdown ( $\text{CPI}_{\text{CPU}}$ ,  $\text{CPI}_{\text{MEM}}$ , etc.)
  - So we know what performance problems are and what to fix
  - Hardware event counters
    - Available in most processors today
    - One way to measure dynamic instruction count
    - Calculate CPI using counter frequencies / known event costs
  - Cycle-level micro-architecture simulation
    - + Measure exactly what you want ... and impact of potential fixes!
    - Method of choice for many micro-architects

# Comparing Performance

# Speedup and % Increase and Decrease

- Program A runs for 200 cycles
- Program B runs for 350 cycles
- Percent increase and decrease are **not the same**.
  - % increase:  $((350 - 200)/200) * 100 = 75\%$
  - % decrease:  $((350 - 200)/350) * 100 = 42.3\%$
- Speedup:
  - $350/200 = 1.75$  – Program A is 1.75x faster than program B
  - As a percentage:  $(1.75 - 1) * 100 = 75\%$
- If program C is 1x faster than A, how many cycles does C run for? – 200 (the same as A)
  - What if C is 1.5x faster? 133 cycles (50% faster than A)

# Mean (Average) Performance Numbers

- **Arithmetic:**  $(1/N) * \sum_{P=1..N} P\_latency$ 
  - For units that are proportional to time (e.g., latency)
- **Harmonic:**  $N / \sum_{P=1..N} 1/P\_throughput$ 
  - For units that are inversely proportional to time (e.g., throughput)
- You can add latencies, but not throughputs
  - $Latency(P1+P2,A) = Latency(P1,A) + Latency(P2,A)$
  - $Throughput(P1+P2,A) \neq Throughput(P1,A) + Throughput(P2,A)$ 
    - 1 mile @ 30 miles/hour + 1 mile @ 90 miles/hour
    - Average is **not** 60 miles/hour
- **Geometric:**  $N\sqrt[N]{\prod_{P=1..N} P\_speedup}$ 
  - For unitless quantities (e.g., speedup ratios)

# Answer

---

- You drive two miles
  - 30 miles per hour for the first mile
  - 90 miles per hour for the second mile
- Question: what was your average speed?
  - Hint: the answer is not 60 miles per hour
  - 0.03333 hours per mile for 1 mile
  - 0.01111 hours per mile for 1 mile
  - 0.02222 hours per mile on average
  - = 45 miles per hour

# Measurement Challenges

# Experiment Design

---

- Two kinds of errors: **systematic** and **random**
- removing **systematic error**
  - aka “measurement bias” or “not measuring what you think you are”
  - Run on an unloaded system
  - Measure something that runs for *at least* several seconds
  - Understand the system being measured
    - simple empty-for-loop test => compiler optimizes it away
  - Vary experimental setup
  - Use appropriate statistics
- removing **random error**
  - Perform many runs: how many is enough?



# Confidence Intervals

- Compute mean *and* confidence interval (CI)

$$\pm t \frac{s}{\sqrt{n}}$$

$t$  = critical value from t-distribution

$s$  = sample standard error

$n$  = # experiments in sample

- Meaning of the 95% confidence interval  $x \pm 1.3$ 
  - collected 1 **sample** with  $n$  experiments
  - given repeated sampling,  $x$  will be within 1.3 of the true mean 95% of the time
- If CIs overlap, differences not statistically significant

# CI example

---

- setup
  - 130 experiments, mean = 45.4s, stderr = 10.1s
- What's the 95% CI?
- $t = 1.962$  (depends on %CI and # experiments)
  - look it up in a stats textbook or online
- at 95% CI, performance is  $45.4 \pm 1.74$  seconds
- What if we want a smaller CI?

# Benchmarking

# Processor Performance and Workloads

---

- Q: what does performance of a chip mean?
- A: nothing, there must be some associated workload
  - **Workload**: set of tasks someone (you) cares about
- **Benchmarks**: standard workloads
  - Used to compare performance across machines
  - Either are, or highly representative of, actual programs people run
- **Micro-benchmarks**: non-standard non-workloads
  - Tiny programs used to isolate certain aspects of performance
  - Not representative of complex behaviors of real applications
  - Examples: binary tree search, towers-of-hanoi, 8-queens, etc.

# Example: GeekBench

---

- Set of cross-platform multicore benchmarks
  - Can run on iPhone, Android, laptop, desktop, etc
- Tests integer, floating point, memory bandwidth performance
- GeekBench stores all results online
  - Easy to check scores for many different systems, processors
- **Pitfall:** Workloads are simple, may not be a completely accurate representation of performance
  - We know they evaluate compared to a baseline benchmark

# Performance Laws

# Amdahl's Law

---

$$\frac{1}{(1 - P) + \frac{P}{S}}$$

How much will an optimization improve performance?

$P$  = proportion of running time affected by optimization

$S$  = speedup

What if I speedup 25% of a program's execution by 2x?

1.14x speedup

What if I speedup 25% of a program's execution by  $\infty$ ?

1.33x speedup

# Amdahl's Law for Parallelization

---

$$\frac{1}{(1 - P) + \frac{P}{N}}$$

How much will parallelization improve performance?

$P$  = proportion of parallel code  
 $N$  = threads

What is the max speedup for a program that's 10% serial?

What about 1% serial?



# Increasing proportion of parallel code

---

- Amdahl's Law requires *extremely* parallel code to take advantage of large multiprocessors
- two approaches:
  - **strong scaling**: shrink the serial component
    - + same problem runs faster
    - becomes harder and harder to do
  - **weak scaling**: increase the problem size
    - + natural in many problem domains: internet systems, scientific computing, video games
    - doesn't work in other domains

# Little's Law

---

$$L = \lambda W$$

$L$  = items in the system

$\lambda$  = average arrival rate

$W$  = average wait time

- Assumption:
  - system is in steady state, i.e., average arrival rate = average departure rate
- No assumptions about:
  - arrival/departure/wait time distribution or service order (FIFO, LIFO, etc.)
- Works on **any** queuing system
- Works on **systems of systems**

# Little's Law for Computing Systems

---

- Only need to measure two of  $L$ ,  $\lambda$  and  $W$ 
  - often difficult to measure  $L$  directly
- Describes how to meet performance requirements
  - e.g., to get high throughput ( $\lambda$ ), we need either:
    - low latency per request (small  $W$ )
    - service requests in parallel (large  $L$ )
- Addresses many computer performance questions
  - sizing queue of L1, L2, L3 misses
  - sizing queue of outstanding network requests for 1 machine
    - or the whole datacenter
  - calculating average latency for a design

# Performance Rules of Thumb

---

- Design for actual performance, **not peak performance**
  - Peak performance: “Performance you are guaranteed not to exceed”
  - Greater than “actual” or “average” or “sustained” performance
    - Why? Caches misses, branch mispredictions, limited ILP, etc.
  - For actual performance  $X$ , machine capability must be  $> X$
- Easier to “buy” bandwidth than latency
  - say we want to transport more cargo via train:
    - (1) build another track or (2) make a train that goes twice as fast?
  - Use bandwidth to reduce latency
- **Build a balanced system**
  - Don’t over-optimize 1% to the detriment of other 99%
  - System performance often determined by slowest component

# Summary

---

- Latency = seconds / program =
  - (instructions / program) \* (cycles / instruction) \* (seconds / cycle)
- **Instructions / program**: dynamic instruction count
  - Function of program, compiler, instruction set architecture (ISA)
- **Cycles / instruction**: CPI
  - Function of program, compiler, ISA, micro-architecture
- **Seconds / cycle**: clock period
  - Function of micro-architecture, technology parameters
- Optimize each component
  - this class focuses mostly on CPI (caches, parallelism)
  - ...but some on dynamic instruction count (compiler, ISA)
  - ...and some on clock frequency (pipelining, technology)