CSE 240A:
Principles in Computer Architecture

ISA

Jishen Zhao (https://cseweb.ucsd.edu/~jzhao/)

[Adapted in part from Dean Tullsen, Mary Jane Irwin, Joe Devietti, Onur Mutlu, and others]

Review: Performance

- Performance metrics: Latency & throughput
- Comparing performance: Speedup
- Averaging performance:
  - Arithmetic mean
  - Harmonic mean
  - Geometric mean
- Measuring CPU performance:
  - CPI (IPC)
  - MIPS
  - FLOPS (later)
- Performance laws
  - Amdahl's law
  - Little's law
- Benchmarks
  - Micro-benchmarks / benchmarks

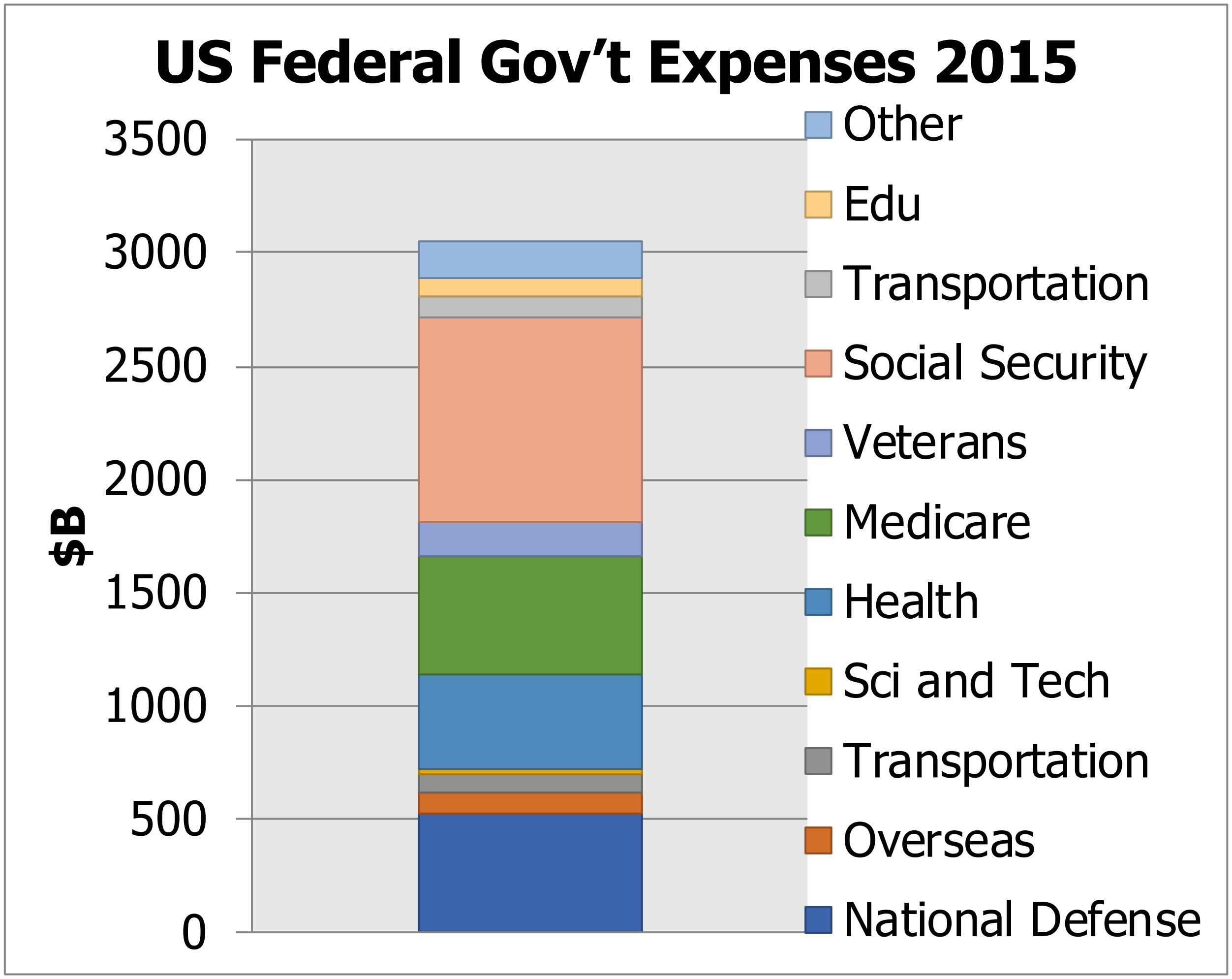
Review: Amdahl’s Law

- Amdahl’s Law:

How much does
an optimization improve performance?

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

Amdahl's Law for the US Budget

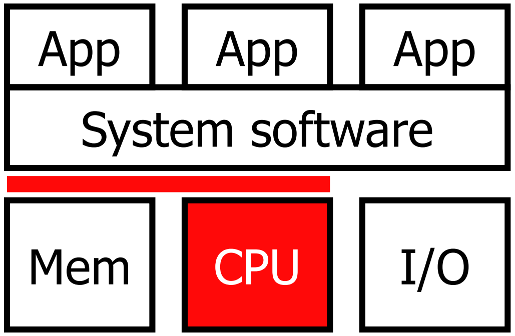


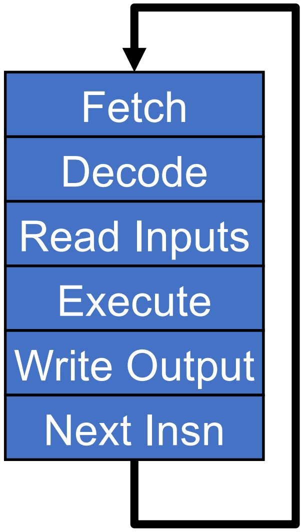
https://en.wikipedia.org/wiki/2015_United_States_federal_budget

Scrapping Dept of Transportation ($81B) cuts budget by 2.6%

Review: ISA

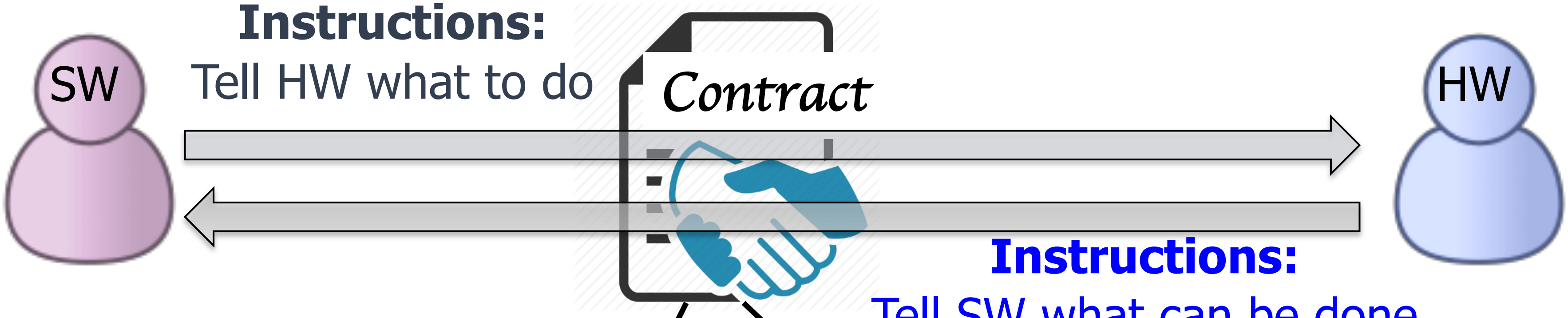
- What is ISA?
- Execution model:
  - Compilation
  - Assembly & machine language
- Instruction execution model
  - Registers, memory, PC
  - Instruction execution
- ISA design goals
  - Programmability
  - Performance/implementability
  - Compatibility





Review: What is ISA?

Instruction Set
Architecture (ISA)

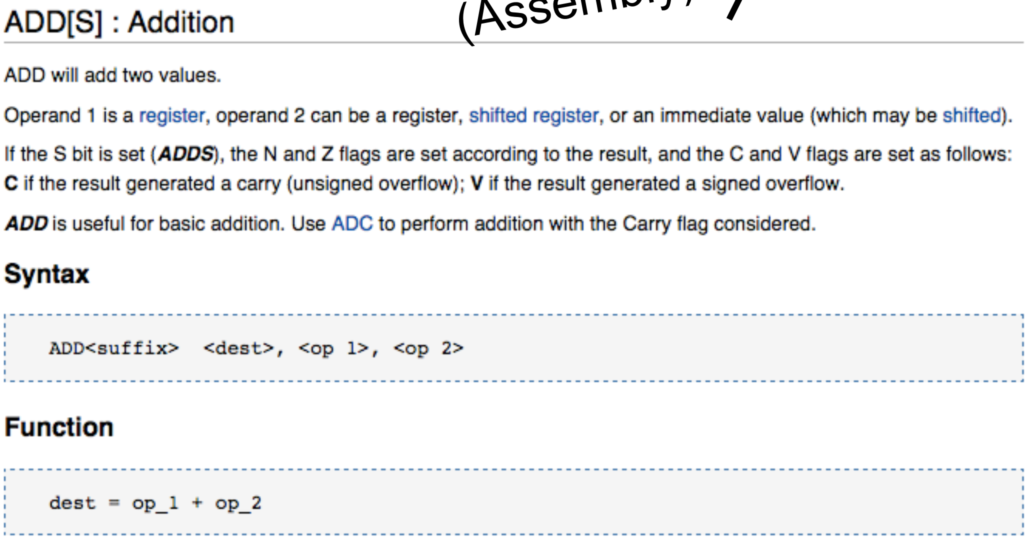


Instructions:
Tell HW what to do

Instructions:
Tell SW what can be done

Instruction definition
(Assembly)

Instruction bit pattern
(Machine code)



ADD[S] : Addition

ADD will add two values.

Operand 1 is a register, operand 2 can be a register, shifted register, or an immediate value (which may be shifted).

If the S bit is set (***ADDS***), the N and Z flags are set according to the result, and the C and V flags are set as follows: **C** if the result generated a carry (unsigned overflow); **V** if the result generated a signed overflow.

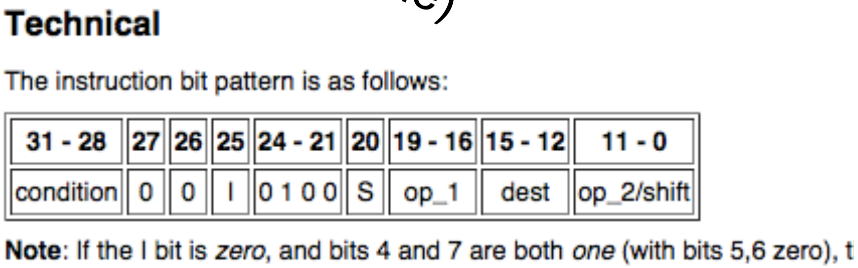
ADD is useful for basic addition. Use ADC to perform addition with the Carry flag considered.

Syntax

```
ADD<suffix>  <dest>, <op 1>, <op 2>
```

Function

```
dest = op_1 + op_2
```



Technical

The instruction bit pattern is as follows:

31 - 28	27	26	25	24 - 21	20	19 - 16	15 - 12	11 - 0
condition	0	0	I	0 1 0 0	S	op_1	dest	op_2/shift

Note: If the I bit is *zero*, and bits 4 and 7 are both *one* (with bits 5,6 zero), t

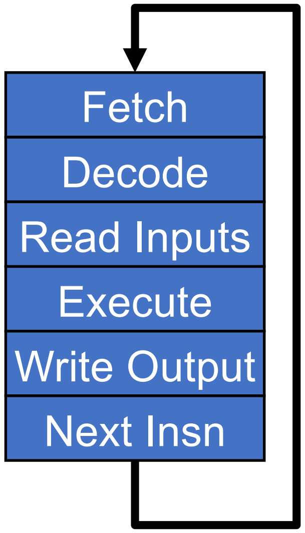
Review: what is/isn't defined in ISA?

- What's defined in ISA?
  - **Functional definition** of storage locations & operations
    - Operations: add, multiply, branch, load, store, etc
    - Data storage locations: registers, memory
  - **Precise description** of how to invoke operations & access data

- What's not in ISA? non-functional aspects
  - How operations are implemented
  - Which operations are fast and which are slow and when
  - Which operations take more power and which take less

Review: Instruction Execution Model

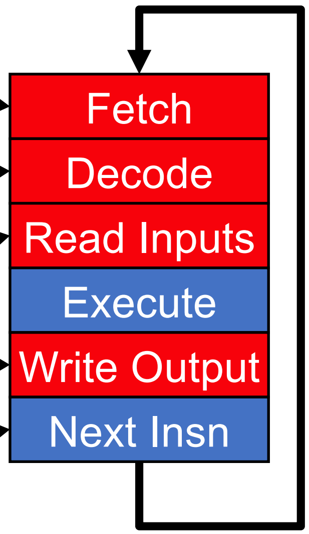
Instruction execution model
≠
Program execution model


Fetch
Decode
Read Inputs
Execute
Write Output
Next Insn

- 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)
- 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” in x86
- **Program is just “data in memory”**
  - Makes computers programmable (“universal”)

Today: Aspects of ISA

- Instruction length
  - Next instruction: PC+length
- Instruction format
  - Instruction encoding
- Where does data live?
  - Addressing modes
- Control transfers
  - How to find the next instruction
  - Branch
  - Jump
- How to design high-performance ISA (if have time)
  - #1 make common case faster
  - #2 make fast case common



More on RISC vs. CISC

CISC and RISC adoptions

- The CISCiest: VAX (**V**irtual **A**ddress e**X**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
- 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 vs. CISC Design Tenets

- **RISC: Single-cycle execution**
  - CISC: many multicycle operations
- **RISC: Hardwired (simple) control**
  - CISC: **microcode** for multi-cycle operations
- **RISC: Load/store architecture**
  - CISC: register-memory and memory-memory
- **RISC: Few memory addressing modes**
  - CISC: many modes
- **RISC: Fixed-length instruction format**
  - CISC: many formats and lengths
- **RISC: Reliance on compiler optimizations**
  - CISC: hand assemble to get good performance
- **RISC: Many registers** (compilers can use them effectively)
  - CISC: few registers

Intel's x86 Trick: RISC Inside

- 1993: Intel wanted "out-of-order execution" in Pentium Pro
  - Hard to do with a coarse grain 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
```

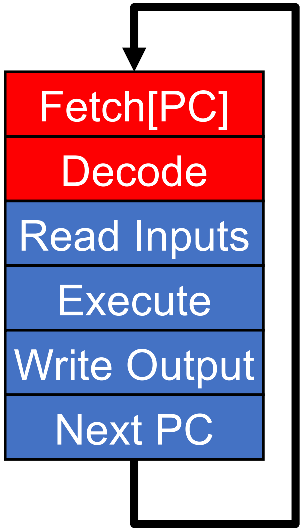
  + Processor maintains **x86 ISA externally for compatibility**
  + But executes **RISC μISA internally for implementability**
  - Given translator, x86 almost as easy to implement as RISC
    - Intel implemented "out-of-order" before any RISC company
    - "out-of-order" also helps x86 more (because ISA limits compiler)
  - Also used by other x86 implementations (AMD)
- Different **μops** for different designs
  - **Not part of the ISA specification**, not publically disclosed

Potential Micro-op Scheme

- Most instructions are a **single** micro-op
  - Add, xor, compare, branch, etc.
  - Loads example: mov -4(%rax), %ebx
  - Stores example: mov %ebx, -4(%rax)
- Each memory access adds a micro-op
  - “addl -4(%rax), %ebx” is two micro-ops (load, add)
  - “addl %ebx, -4(%rax)” is three micro-ops (load, add, store)
- Function call (CALL) – 4 uops
  - Get program counter, store program counter to stack, adjust stack pointer, unconditional jump to function start
- Return from function (RET) – 3 uops
  - Adjust stack pointer, load return address from stack, jump register
- Again, just a basic idea, micro-ops are specific to each chip

Aspects of ISAs

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 15)

    – Complex fetch (where does next instruction begin?)
  - Compromise: two lengths
    - E.g., MIPS16 or ARM’s Thumb (16 bits)
- **Encoding**
  - A few simple encodings simplify decoder
  - Machine code (1s and 0s) <-> assembly

Example Instruction Encodings

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

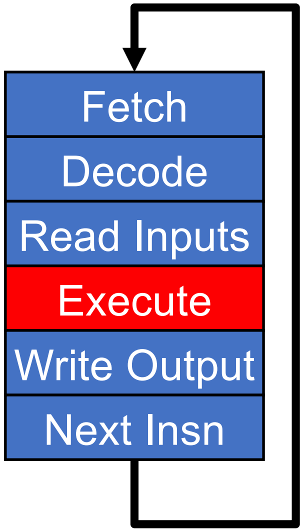
`add R1,R2,R3`

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

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

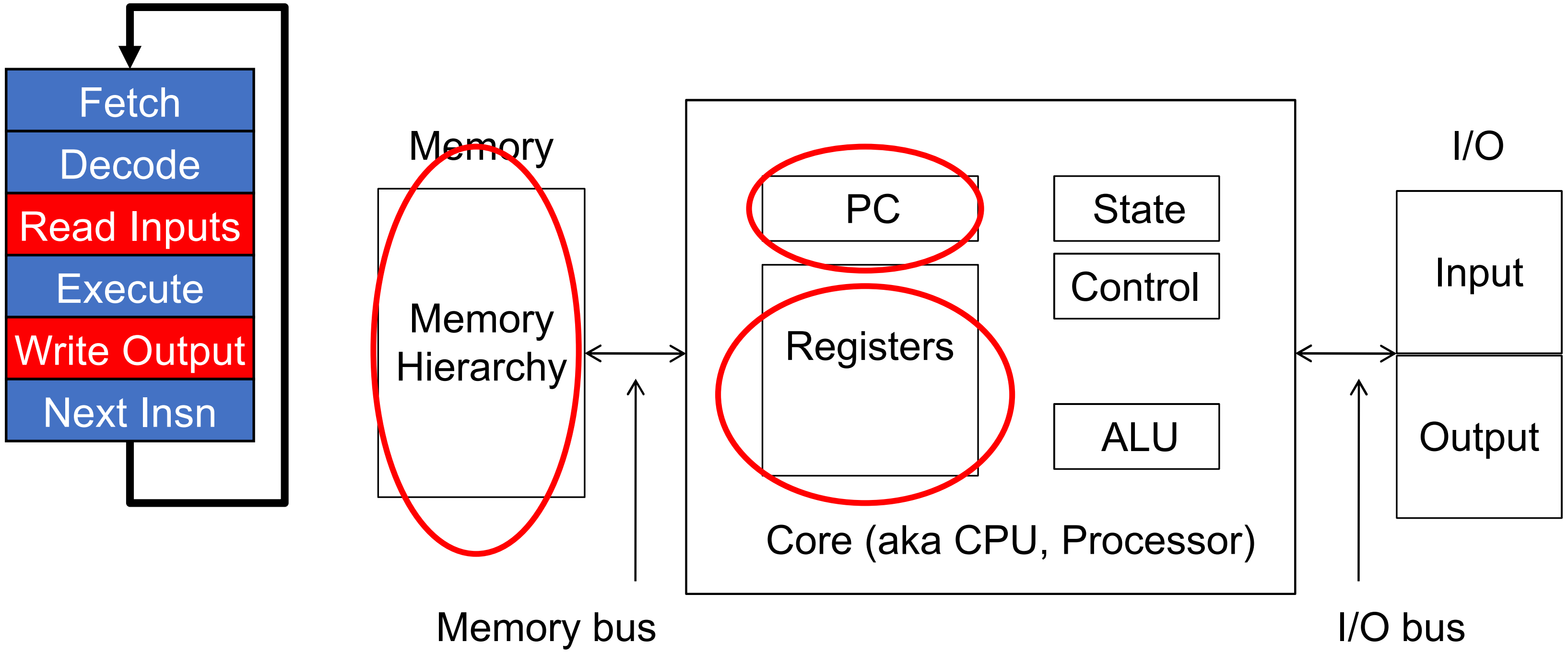
Prefix*(1-4)	Op	OpExt*	ModRM*	SIB*	Disp*(1-4)	Imm*(1-4)

Operations and Datatypes

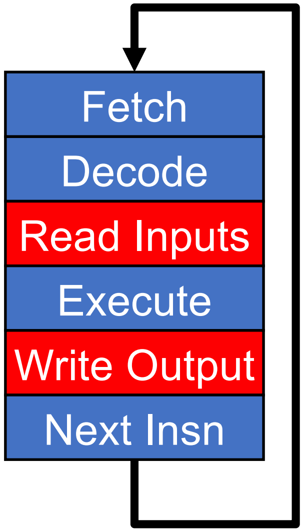


- Datatypes
  - Software: attribute of data
  - Hardware: attribute of operation, data is just 0/1's
- All processors support
  - Integer arithmetic/logic (8/16/32/64-bit)
  - IEEE754 floating-point arithmetic (32/64-bit)
- More recently, most processors support
  - "Packed-integer" insns, e.g., MMX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

Where Does Data Live?



Where Does Data Live?



- **Registers (e.g., R0, R1, F0)**
  - "short term memory"
  - Faster than memory, quite handy
  - Named directly in instructions

```
ADD R1,R2,R3
```

- **Memory (e.g., (R3), #20(R5))**
  - "longer term memory"
  - Accessed via "addressing modes"
    - Address to read or write calculated by instruction

```
ADD R1,R2,(R3)
```

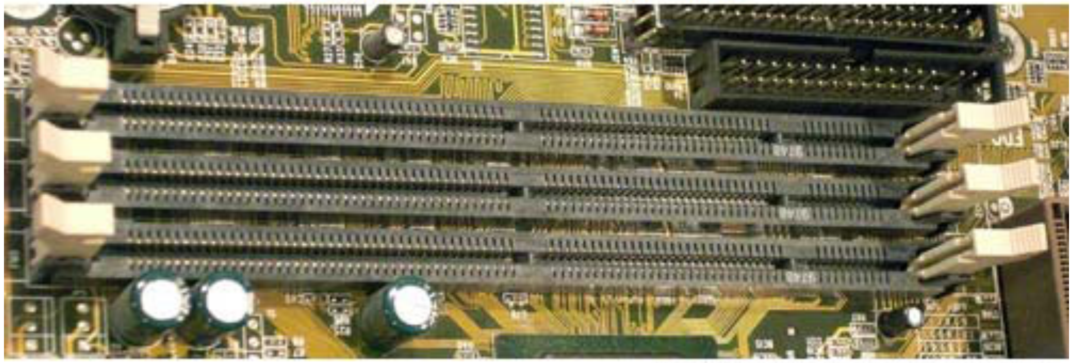
- "Immediates" (e.g., #36, #7)
  - Values spelled out as bits in instructions
  - Input only

How Many Registers?

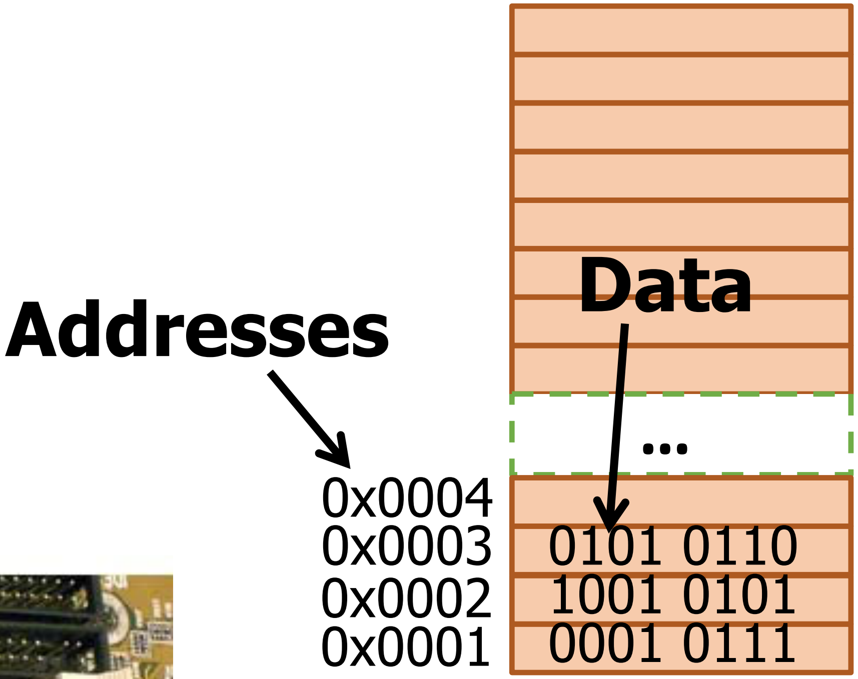
- Registers faster than memory, have as many as possible?
  - **No**
- One reason registers are faster: there are **fewer of them**
  - Small is fast (hardware truism)
- Another: they are **directly addressed** (no address calc)
  - More registers, means more bits per register in instruction
  - Thus, fewer registers per instruction or larger instructions
- **Not everything can be put in registers**
  - Structures, arrays, anything pointed-to
  - 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

Computer Memory Slots



http://www.computerhope.com



Addresses

Data

...

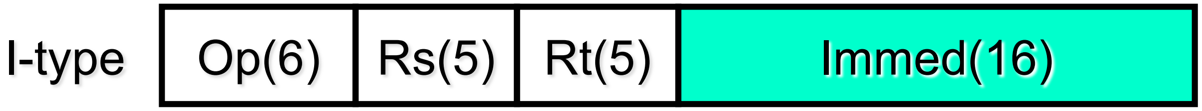
0x0004
0x0003 0101 0110
0x0002 1001 0101
0x0001 0001 0111

Memory Addressing

- **Addressing mode:** way of specifying address
- Examples
  - **Displacement:** address = [R2+immed], e.g., #20(R2)
  - **Index-base:** address = [R2+R3]
  - **Memory-indirect:** address =[mem[R2]]
  - **Auto-increment:** address=[R2], R2= R2+1
  - **Auto-indexing:** address =[R2+immed], R2=R2+immed
  - **Scaled:** address =[R2+R3*immed1+immed2]
  - **PC-relative:** address =[PC+imm]

Addressing Modes Examples

- MIPS



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

  - **Displacement**: R1+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+offset (8/16/32-bit)
  - **Indexed**: R1+R2
  - **Scaled:** R1 + (R2*Scale) + offset(8/16/32-bit)   Scale = 1, 2, 4, 8

Example: x86 Addressing Modes

```
    .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
```

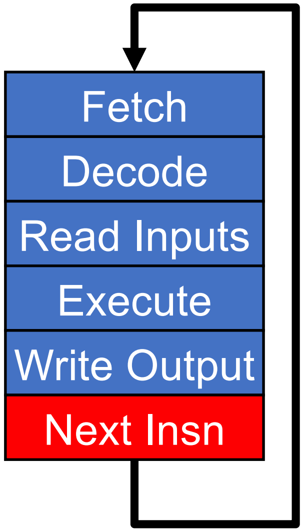
Displacement

Scaled: address = array + (%eax * 4)
Used for sequential array access

PC-relative: offset of sum wrt %rip

Note: "mov" can be load, store, or reg-to-reg move

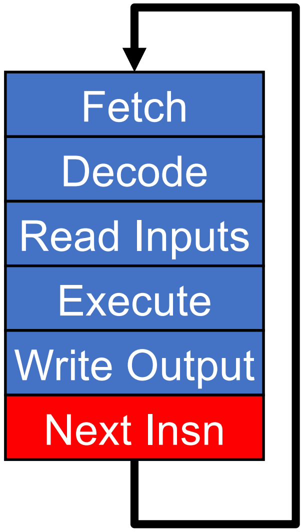
Control Transfers



- Default next-PC = PC + sizeof(current insn)
  - Branches and jumps can change that
- **Computing targets**: where to jump to
  - For all branches and jumps
  - PC-relative: e.g., bne R3, R6, L3 for branches and jumps with function
  - Absolute: e.g., J L3 for function calls
  - Register indirect: e.g., JR R5 for returns, switches & dynamic calls

```
L3:
   addu  R7, R4, R3
   lw  R7, (R7)
   addu  R8, R5, R3
    J L3
   bne R3, R6, L3
```

Control Transfers



- **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
    ```

```
L3:
   addu  R7, R4, R3
   lw  R7, (R7)
   addu  R8, R5, R3
    J L3
   bne R3, R6, L3
```

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)

ISA Code Examples

Code examples

```
int foo(int x, int y) {
  return (x+10) * y;
}
```

```
int max(int x, int y) {
  if (x >= y) return x;
  else return y;
}
```

check out http://gcc.godbolt.org to examine these snippets

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

x86 and ARM, -O0 and -O3
x86: destination reg on the right
arm: destination reg on the left

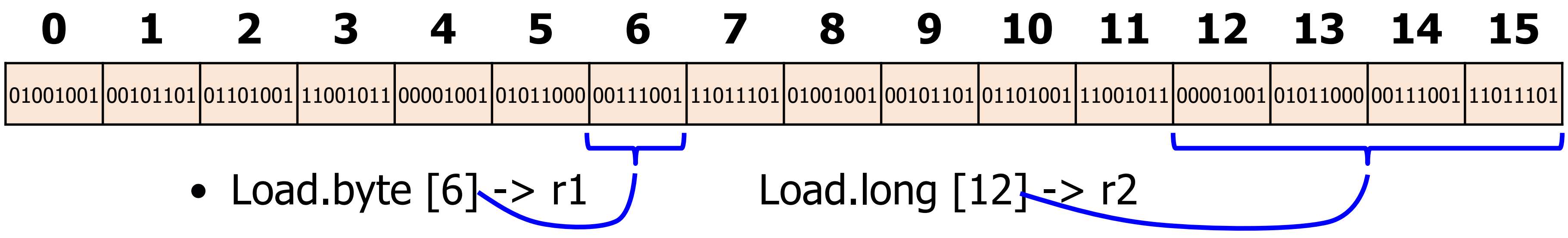
How to design high-performance ISA?

Performance Rule #1

make the common case fast

Access Granularity & Alignment

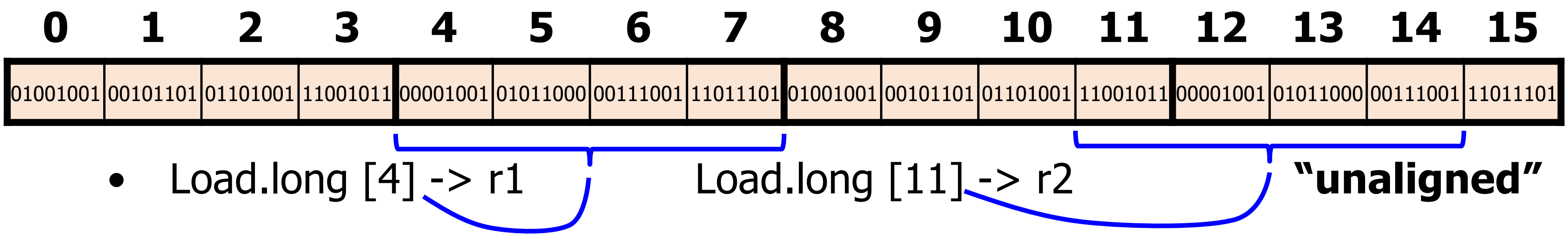
- **Byte addressability**
  - An address points to a byte (8 bits) of data
  - The ISA’s minimum granularity to read or write memory
  - ISAs also support wider load/stores
    - “Half” (2 bytes), “Longs” (4 bytes), “Quads” (8 bytes)



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
01001001	00101101	01101001	11001011	00001001	01011000	00111001	11011101	01001001	00101101	01101001	11001011	00001001	01011000	00111001	11011101

- Load.byte [6] -> r1 Load.long [12] -> r2

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



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
01001001	00101101	01101001	11001011	00001001	01011000	00111001	11011101	01001001	00101101	01101001	11001011	00001001	01011000	00111001	11011101

- Load.long [4] -> r1 Load.long [11] -> r2 **“unaligned”**

- **Access alignment**: if address % size is not 0, then it is “unaligned”
  - A single unaligned access may require multiple physical memory accesses

Handling Unaligned Accesses

- **Access alignment**: if address % size is not 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? (allow, but hardware traps to software)
     - 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)
  5. MIPS ISA support: unaligned access by compiler using two instructions
     - Faster than above, but still needs help from compiler

```
lwl @XXXX10; lwr @XXXX10
```

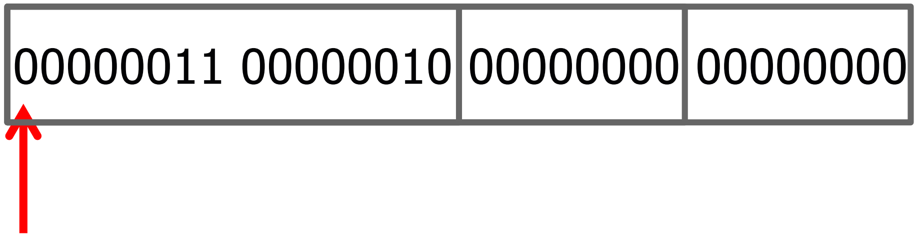
How big is this struct?

```
struct foo {
  char c;
  int i;
}
```

Hint: avoid unaligned accesses

Another Addressing Issue: Endian-ness

- **Endian-ness**: arrangement of bytes in a multi-byte number
  - Big-endian: sensible order (e.g., MIPS, PowerPC, ARM)
    - The most significant byte (the "big end") of the data is placed at the byte with the lowest address
    - A 4-byte integer: “00000000 00000000 00000010 00000011” is 515_{10}
  - Little-endian: reverse order (e.g., x86)
    - A 4-byte integer: “00000011 00000010 00000000 00000000” is 515_{10}
    - The least significant byte (the "little end") of the data is placed at the byte with the lowest address
  - Why little/big endian?



00000011 00000010	00000000	00000000

Little-endian: Integer casts are free

starting address

Big-endian: Sign checks cheaper ← The sign bit is the most significant bit, and thus will be in the last byte in a little-endian format

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 requires three instructions in MIPS, for example.

x86 Operand Model: Accumulators

```
    .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
```

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

Register accumulator: %eax = %eax + %edx

Memory accumulator:
Memory[%rbp-4] = 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 is the size of **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

Performance Rule #2

make the fast case common

Winner for Desktops/Servers: CISC

- x86 was first mainstream 16-bit microprocessor by ~2 years
  - IBM put it into its PCs…
  - 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

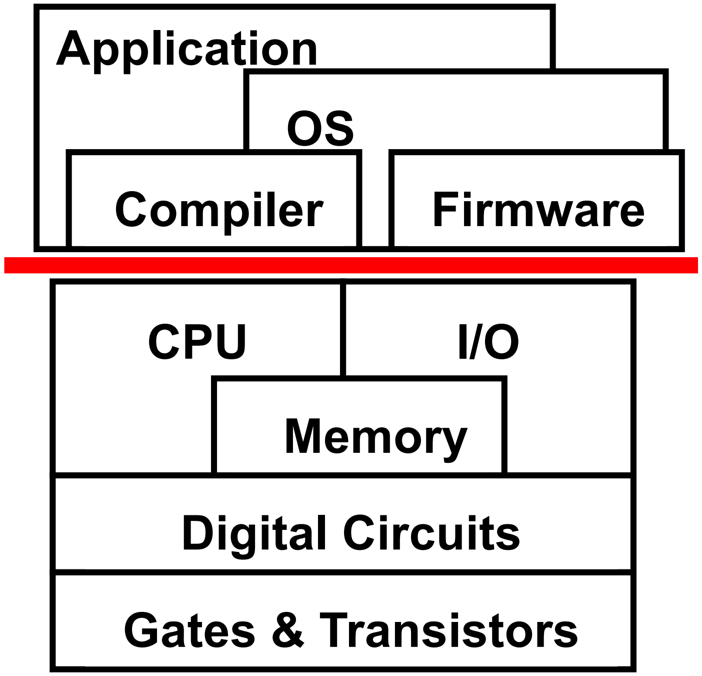
Winner for Embedded: RISC

- 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, Qualcomm, Freescale (neé Motorola), Texas Instruments, STMicroelectronics, Samsung, Sharp, Philips, etc.

Redux: Are ISAs Important?

- Does "quality" of ISA actually matter?
  - Not for performance (mostly)
    - Mostly comes as a design complexity issue
    - Insn/program: everything is compiled, compilers are good
    - Cycles/insn and seconds/cycle: μISA, many other tricks
  - What about power efficiency? Maybe
    - ARMs are most power efficient today…
      - …but Intel is moving x86 that way (e.g., Atom)
- Does "nastiness" of ISA matter?
  - Mostly no, only compiler writers and hardware designers see it
- Comparison is confounded by, e.g., transistor technology
- 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, μISAs