ISA

Giorgio Richelli

Instruction Set Architecture

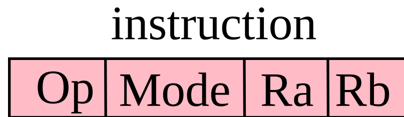
- Contract between programmer and the hardware
  - Defines visible state of the system
  - Defines how state changes in response to instructions

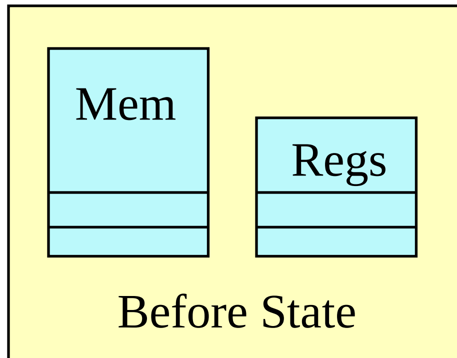
- Programmer: ISA is model of how a program will execute
- Hardware Designer: ISA is formal definition of the correct way to execute a program

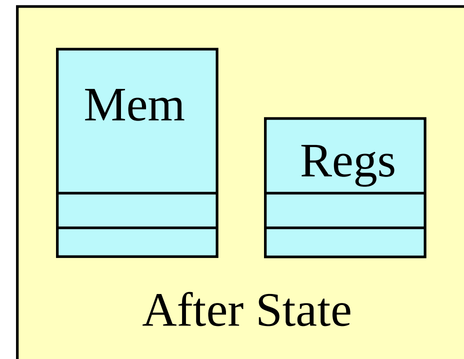
- ISA specification
  - The binary encodings of the instruction set

ISA Basics

Instruction formats
Instruction types
Addressing modes







Data types
Operations
Interrupts/Events

Machine state
Memory organization
Register organization

Architecture vs. Implementation

- Architecture: defines what a computer system does in response to a program and a set of data
  - Programmer visible elements of computer system

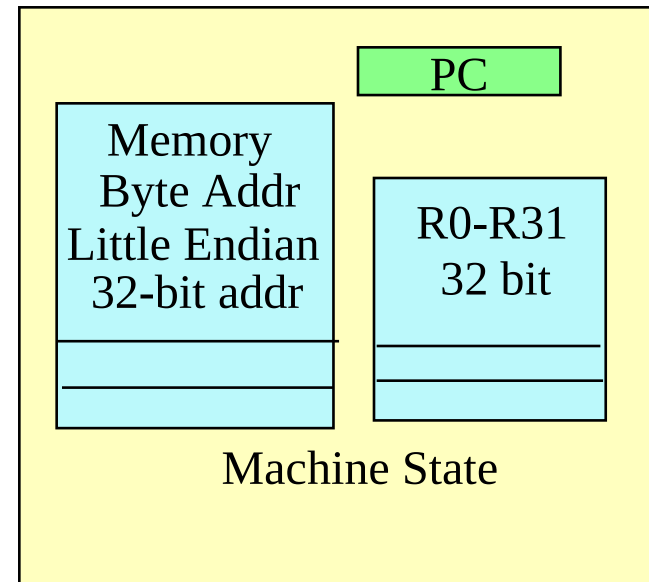
- Implementation: defines how a computer does it
  - Sequence of steps to complete operations
  - Time to execute each operation
  - Hidden “bookkeeping” functions

Examples

- Architecture or Implementation?
  - Logical number of GP registers
  - Width of memory bus
  - Size of the instruction cache
  - Binary representation of the instruction `sub r4,r2,#27`
  - Number of cycles to execute FP instruction
  - Which condition code bits are set on a move instruction
  - Type of FP format

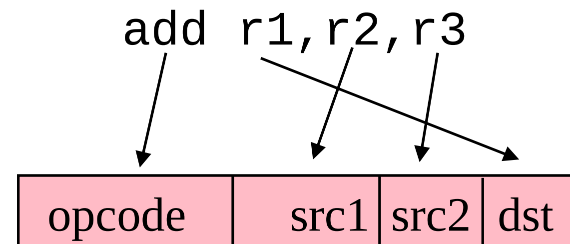
Machine State

- Registers
  - Size/Type
    - Program Counter (= IP)
    - accumulators
    - index registers
    - general registers
    - control registers
- Memory
  - Visible hierarchy (if any)
  - Addressibility
    - byte, word, bit
    - byte order (endian-ness)
    - maximum size
  - protection/relocation



Components of Instructions

- Operations (opcodes)
- Number of operands
- Operand specifiers



- Instruction encodings
- Instruction classes
  - ALU ops (add, sub, shift)
  - Branch (beq, bne, etc.)
  - Memory (ld/st)

Operand Number

- No Operands HALT NOP
- 1 operand NOT R4 R4 ⇐ R4
  JMP _L1
- 2 operands ADD R1, R2 R1 ⇐ R1 + R2
  LDI R3, #1234
- 3 operands ADD R1, R2, R3 R1 ⇐ R2 + R3
- > 3 operands MADD R4,R1,R2,R3 R4 ⇐ R1+ (R2*R3)

Effect of Operand Number

```
E = (C+D)*(C-D)
```

Assign

C $\Rightarrow$ r1
D $\Rightarrow$ r2
E $\Rightarrow$ r3

3 operand machine

```
add  r3,r1,r2
sub  r4,r1,r2
mult r3,r4,r3
```

2 operand machine

```
mov  r3,r1
add  r3,r2
sub  r2,r1
mult r3,r2
```

Architectures

- Memory-memory
  - CISC idea
  - Usually allows any operand to be in register as well
- Register-memory
  - Example: x86
  - Can do one operand in register, one in memory, or 2 in regs
- Register-register
  - Only design used in modern machines
  - Lots of registers → fast flexible operand access
  - Simplicity of hardware
  - Compiler has full flexibility in register usage

Memory Organization

- Four components specified by ISA:
  - Smallest addressable unit of memory (byte? halfword? word?) (*addressibility*)
  - Maximum addressable units of memory (doubleword?)
  - Alignment
  - Endianness

Memory Addressing

- Originally just word addressing
- 8-bit bytes and byte addressing introduced on IBM 360 series
- Brief experiments with bit addressing

- Unaligned accesses ?

- Some machines byte-address but only load/store a word at a time
- Modern RISC designs allow short load/store, but not short arithmetic

Addressing Modes

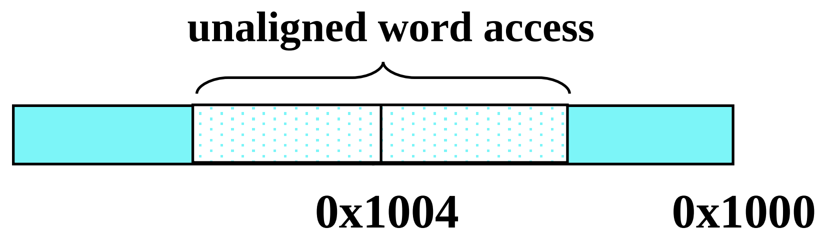
- How can an instruction reference memory?
- Early days: absolute address in instruction
  - Led to instruction modification
  - Improvement: “Indirection” picked up absolute location, used it as final address
- Minimum necessary today: follow pointer in register
  - Clumsy if only option
- Fanciest conceivable: *(R1+S*R2+constant), with either or both of R1 and R2 autoincremented or autodecremented as side effect, either before or after instruction
  - No machine went quite this far, but VAX came close

Addressing Modes (cont'd)

- What's actually useful?
  - Need to follow pointers: can restrict to registers
    - `ADD R1,(R2)`
  - Frequent stack access → register + constant useful
  - Immediates needed for built-in constants
  - Access to globals → absolute memory addresses
  - PC-relative modes
    - Used to be needed for data; not in modern systems
    - Still needed for calls and branches
  - Absolute addresses no longer needed for branches
    - Can always emulate with PC-relative, since PC known
    - Still available on some architectures

Alignment

- Some architectures restrict addresses that can be used for particular size data transfers
- E.g:
  - Bytes accessed at any address
  - Halfwords only at even addresses
  - Words accessed only at multiples of 4



Endian-ness

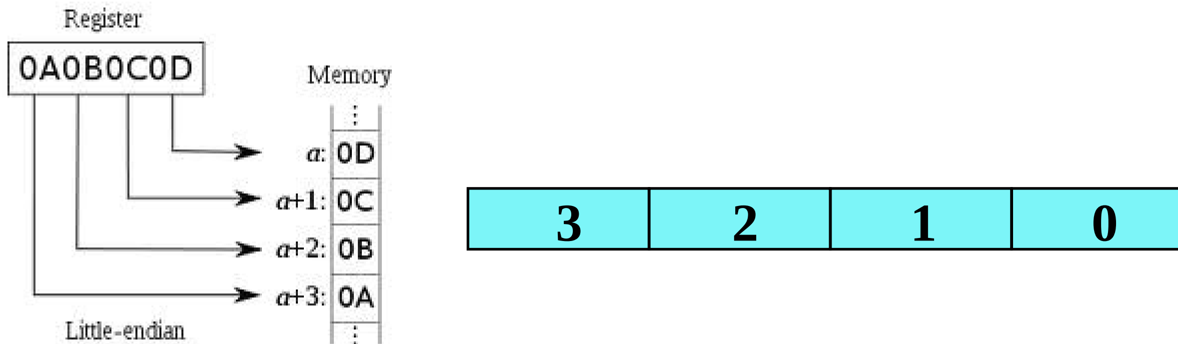
- Reference to *Gulliver's Travels:*
  - whereas royal edict in Lilliput requires cracking open one's soft-boiled egg at the small end, inhabitants of the rival kingdom of Blefuscu crack theirs at the big end (giving them the moniker Big-endians)

- Little-Endian invented by Digital Equipment on the PDP-11

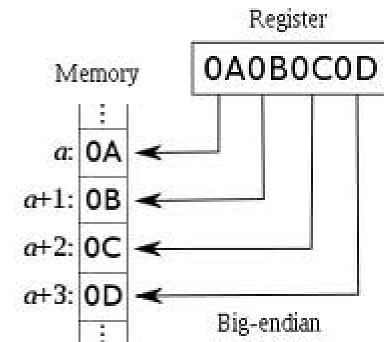
- Some machines can switch endianness with a control bit
  - awkward and seldom useful

Endianness

- How are bytes ordered within a word?
  - Little Endian (Intel/DEC)



  - Big Endian (IBM/Motorola)

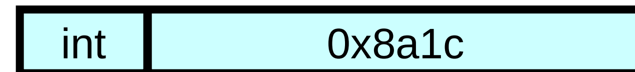


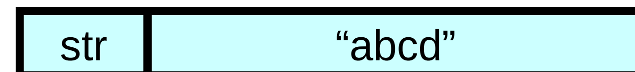
  - Internet Standard is Big Endian

Data Types

- How the contents of memory and registers are interpreted
- Can be identified by
  - tag
  - use
- Driven by application
  - Signal processing
    - 16-bit fixed point (fraction)
  - Text processing
    - 8-bit characters
  - Scientific computing
    - 64-bit floating point

- Most *general purpose* computers support several types
  - 8, 16, 32, 64-bit
  - signed and unsigned
  - fixed and floating





Examples of tags (ie. Symbolics machine)

Example: 32-bit Floating Point

- Type specifies mapping from bits to real numbers (plus symbols)
  - format
    - S, 8-bit exp, 23-bit mantissa
  - interpretation
    - mapping from bits to abstract set

$$v = (-1)^{S} \times 2^{(E-127)} \times 1.M$$

  - operations
    - add, mult, sub, sqrt, div

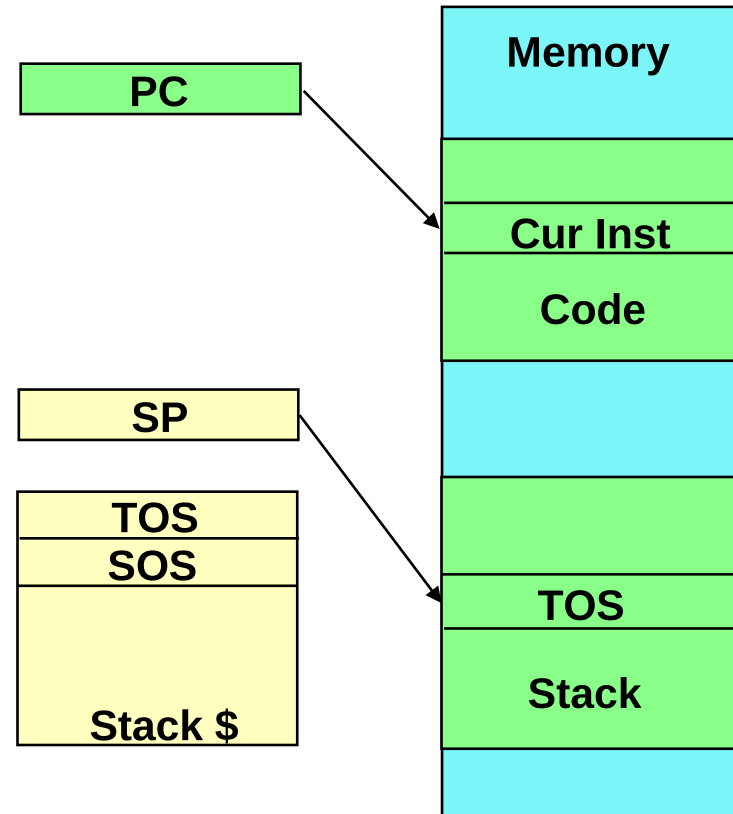
Types of ISAs

- Stack
  - Implicit operands (top of stack)
  - Heavy memory traffic
  - Limited ability to access operands at will
  - Obsolete
- Accumulator
  - Implicit register operand (“accumulator”)
  - One memory operand
  - Insufficient temporaries
  - Obsolete
- General-purpose register
  - Multiple registers
  - Several variations

Stack Machines

- Register state is PC and SP
- All instructions performed on TOS (top of stack) and SOS
  - pushes/pops of stack implied

    op TOS SOS
    op TOS M
    op TOS *M
    op TOS *(M+SP)
- Many instructions are *zero* address
- Stack cache for performance
  - similar to register file
  - hardware managed
- Why do we care? **JVM**



Evolution of ISA

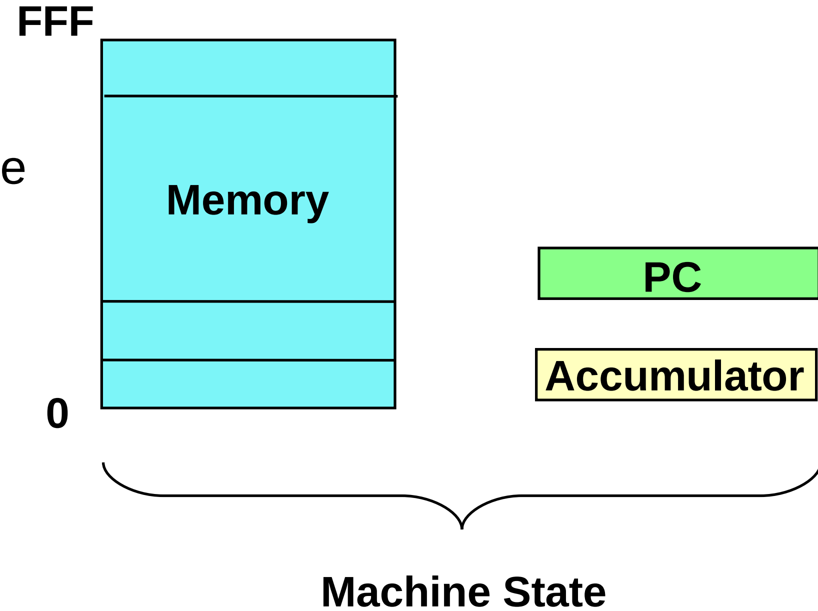
In the beginning…the accumulator

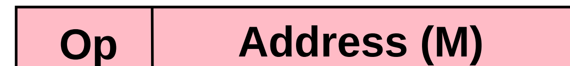
- Two instruction types: op and store

  A ←A op M

  A ←A op *M

  *M ←A
- One address architecture
  - each instruction encodes one memory address
- Two addressing modes
  - *immediate*: M
  - *indirect addressing*: *M





Instruction Format

(Op encodes addressing mode)

Why Accumulator Architectures?

- Registers expensive in early technologies (vacuum tubes)
- Simple instruction decode
  - Logic also expensive
  - Critical programs were small (efficient encoding)
- Less logic $\Rightarrow$ faster cycle time
- Model similar to earlier “tabulating” machines
  - Think of an “adding machine”

The Index Register

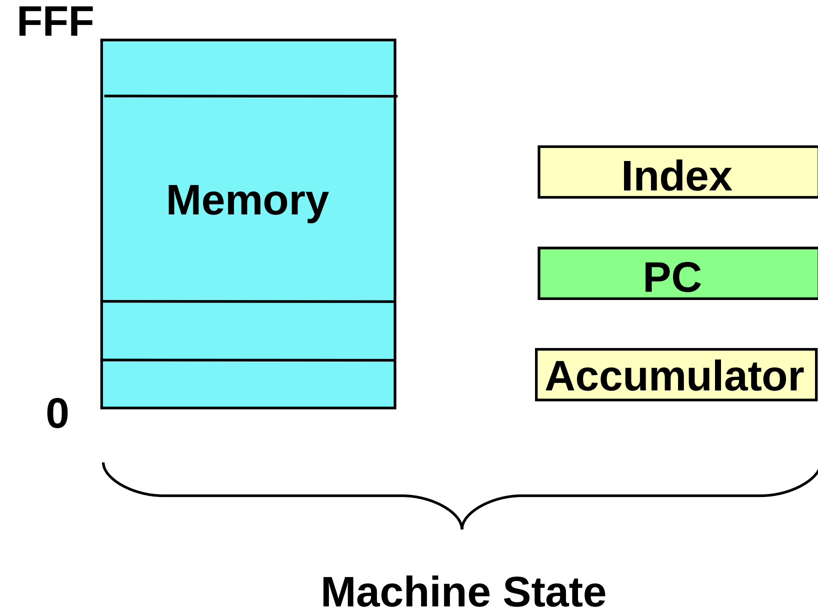
- Add an indexed addressing mode

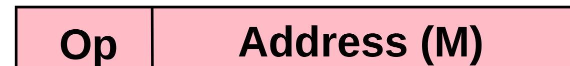
  A ← A op (M+I)

  A ← A op *(M+I)

  ***(M+I) ← A**

  - good for array access: x[j]
    - address of x[0] in instruction
    - j in index register
  - one register for each key function
    - IP → instructions
    - I → data addresses
    - A → data values
  - new instructions to use I
    - INC I, CMP I, etc.





Instruction Format

Example of Indexed Addressing

```
sum = 0;
for(i=0; i<n; i++)
  sum = sum + y[i];
```

```
START:  CLR    i
        CLR    sum
LOOP:   LOAD   IX
        AND    #MASK
        OR     i
        STORE  IX
        LOAD   sum
IX:     ADD    y
        STORE  sum
        LOAD   i
        ADD    #1
        STORE  i
        CMP    n
        BNE    LOOP
```

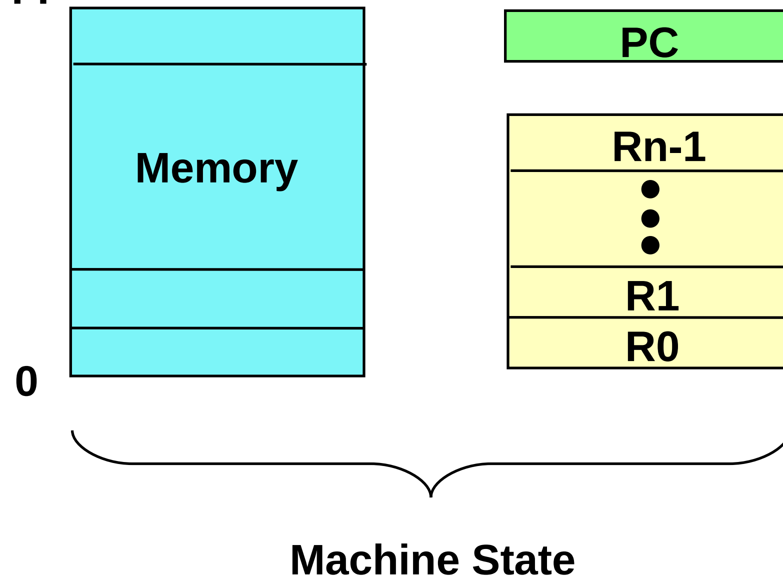
Without Index Register

```
START:  CLRA
        CLRX
LOOP:   ADDA   y(X)
        INCX
        CMPX   n
        BNE    LOOP
```

With Index Register

General Registers

- Merge accumulators (data) and index (address)
- Any register can hold variable or pointer
  - simpler
  - more orthogonal (opcode independent of register usage)
  - More fast local storage
  - but….addresses and data must be same size
- How many registers?
  - More - fewer loads and stores but more instruction bits



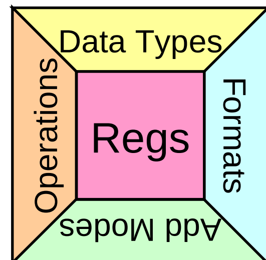
3-address Instruction Format

Five Ways to Do C = A + B

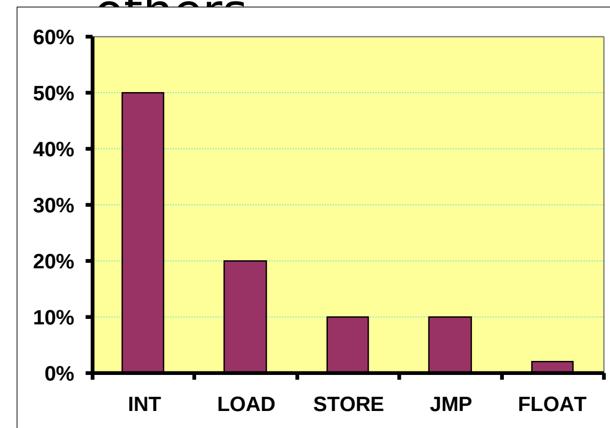
STACK	ACCUM	MEM-MEM	REG-MEM	REG-REG
PUSH A	LOAD A	ADD C,A,B	LOAD R1,A	LOAD R1,A
PUSH B	ADD B		ADD R1,B	LOAD R2,B
ADD	STORE C		STORE R1,C	ADD R3,R1,R2
POP C				STORE R3,C

Principles of Instruction Set Design

- Keep it simple (KISS)
  - complexity
    - increases logic area
    - increases pipe stages
    - increases development time
  - evolution tends to make kludges
- Orthogonality (modularity)
  - simple rules, few exceptions
  - all ops on all registers

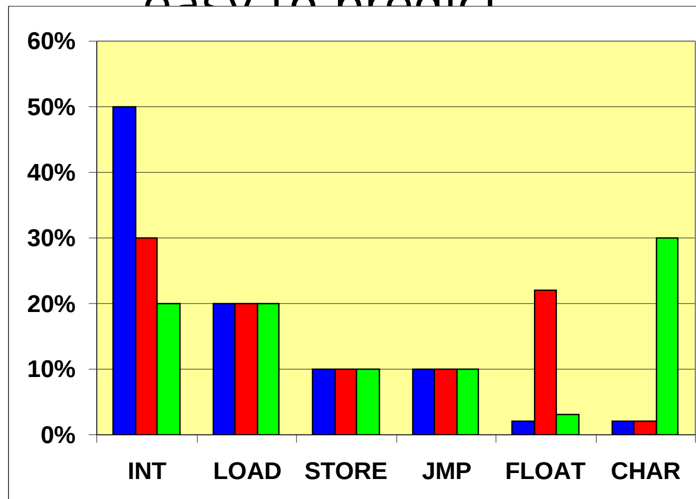


- Frequency
  - make the common case fast
    - some instructions (cases) are more important than others

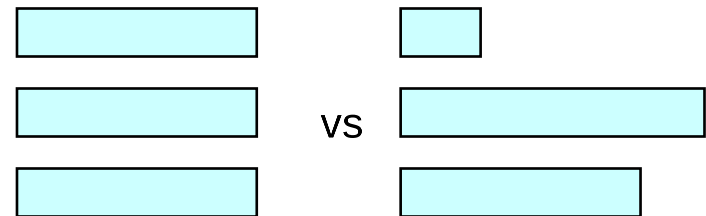


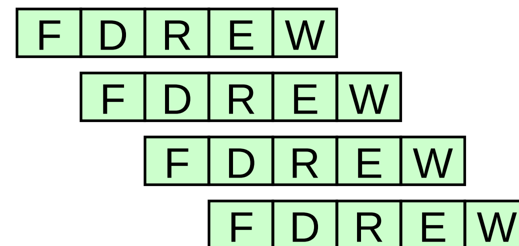
Principles of Instruction Set Design (part 2)

- Generality
  - not all problems need the same features/instructions
  - principle of *least surprise*
  - performance should be easy to predict



- Locality and concurrency
  - design ISA to permit efficient implementation
    - today
    - 10 years from now





Operand Types and Sizes

- Type usually implies size
- Integers can safely be widened to word size
  - Shrink again when stored
  - Takes advantage of two's-complement representation
- Single-precision FP gives different results than double-precisions
  - → Necessary to support both widths
  - Some FPUs can do two SP operations in parallel
- Older machines allowed "packed" decimal (2 digits per byte)
  - x86 supports with DAA (Decimal Add Adjust) instruction
  - Still useful in business world, though dying
- 32 bits / 64 bits standard today
  - 128?

Operations Provided

- Only one instruction truly needed: SJ
  - Subtract A from B, giving C; if result is < 0, jump to D
  - It's Turing-complete!
- Practical machines need a bit more at minimum:
  - Arithmetic and logical (add, multiply, divide?, and, or, …)
  - Data movement (load/store, move between registers)
  - Control (conditional/unconditional branch, procedure call and return, trap to OS)
  - System control (return from interrupt, manage VM, set unprivileged mode, access I/O devices)
- Other builtins can be useful:
  - Basic floating point
  - Decimal
  - String
  - Vector, graphics

Control Flow

- Addressing modes are important
  - PC-relative means code can run at any virtual address
  - Useful for dynamically linked (shared) libraries
- Pointer-following jump needed for returns
  - Also useful for switch statements, function pointers, virtual functions, and shared libraries
- How to specify condition for conditional branches?
  - Condition code as side effect of every instruction
  - Condition register explicitly set by comparison
  - Compare as part of branch
    - Adds delay slots in pipeline

Encodings

- Variable-length instructions
  - Highly efficient (few wasted bits)
  - Allows complex specifications (e.g., x86 addressing modes)
  - Usually means misaligned instruction fetch
  - Greatly complicates fetch/decode units
- Fixed-length instructions
  - May limit number of registers
  - Usually very few instruction formats
  - Wastes space but gains speed (e.g., only aligned fetches)
  - Limits width of immediate operands

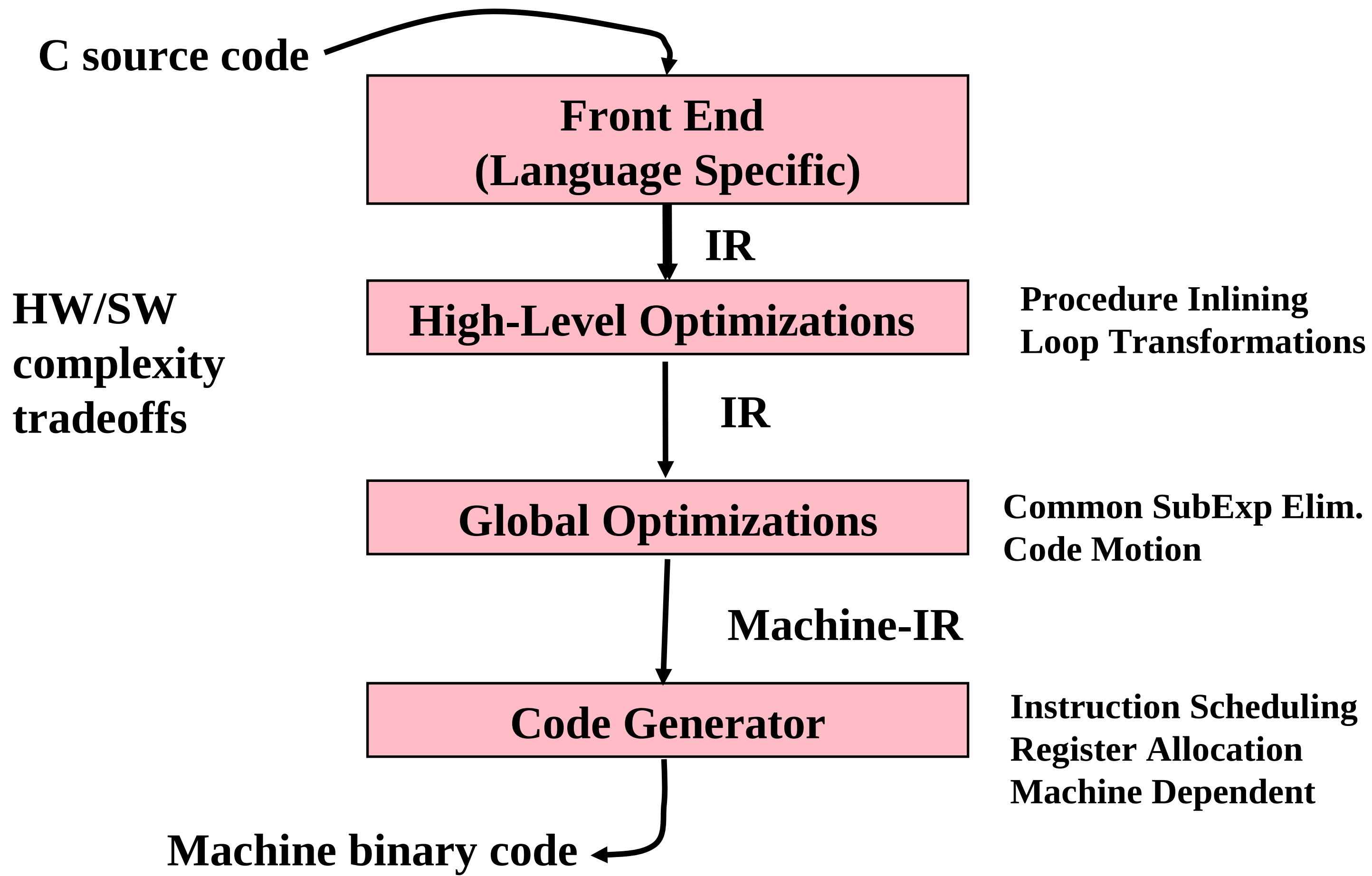
The Fight for Bits

- How wide should instruction be?
  - Wider → can encode more registers, more options
  - Wider → bigger programs, more memory bandwidth
  - Bigger programs → fewer cache hits
- Things you need to encode:
  - Operation code (16 to 1000 instructions)
  - Operands (at least one, normally two or three)
  - Immediate operands
  - Memory offsets
  - Branch targets
  - Branch conditions
  - Conditional operations (e.g., conditional load, add)

Interaction with Compilers

- Nearly all modern code generated by compilers
- Architect must make compiler's job easier:
  - Lots of registers
  - Orthogonal instruction set
  - Few side effects
  - Instructions and addressing modes matched to language constructs
    - But *NOT* attempt to implement them in detail!
    - Primitives are better than "solutions" even when solutions are correct
  - Good support for stack, globals, and pointers
  - Support for both compile-time and run-time binding
  - Don't ask compiler to predict dynamic information (e.g., branch targets)
  - Don't provide features language can't express
    - Example pro and con: vector architectures

Role of the Optimizing Compiler



Example: Loop Optimization

```
sum=0;
for(i=0;i<max;i++)
  sum+=x[i];
```

```
         LW   R1, X
         ADD  R2,R0,R0
         ADD  R3,R0,R0
LOOP:    SLT  R5,R2,#MAX
         BEQZ R5,CONT
         LW   R4,R1
     7   ADD  R3,R3,R4
         ADD  R1,R1,#4
         ADD  R2,R2,#1
         J LOOP
CONT:
```

Loop Reordering

```
         LW   R1, X
         ADD  R2,R0,R0
         ADD  R3,R0,R0
LOOP:    LW   R4,R1
         ADD  R3,R3,R4
         ADD  R1,R1,#4
     6   ADD  R2,R2,#1
         SLT  R5,R2,#MAX
         BNEZ R5,LOOP
CONT:
```

Induction Variable Analysis

```
         LW   R1, X
         ADD  R2,R0,#MAX
         SLLI R2,R2,#2
         ADD  R2,R1,R2
         ADD  R3,R0,R0
LOOP:    LW   R4,R1
         ADD  R3,R3,R4
     5   ADD  R1,R1,#4
         SLT  R5,R1,R2
         BNEZ R5,LOOP
CONT:
```

Cycles Per Instruction (CPI)

- Depends on the instruction

$$CPI_i = \text{Execution time of instruction } i * \text{Clock Rate}$$

- Average cycles per instruction

$$CPI = \sum_{i=1}^{n} CPI_i * F_i \quad \text{where } F_i = \frac{IC_i}{IC_{tot}}$$

- Example:

Op	Freq	Cycles	CPI(i)	%time
ALU	50%	1	0.5	33%
Load	20%	2	0.4	27%
Store	10%	2	0.2	13%
Branch	20%	2	0.4	27%
		CPI(total)	1.5	

Comparing and Summarizing Performance

- Fair way to summarize performance?
- Capture in a single number?

- Example: Which of the following machines is best?

	Computer A	Computer B	Computer C
Program 1	1 sec	10 sec	20 sec
Program 2	1000 sec	100 sec	20sec

RISC/CISC

- RISC "*concept*" was developed by John Cocke of IBM Research during 1974
  - A computer uses only 20% of the instructions, making the other 80% superfluous to requirement.
  - By reducing the number of transistors and instructions to only those most frequently used, the computer would get more done in a shorter amount of time.
- The RISC "*term*" (Reduced Instruction Set Computer) was introduced by David Patterson.

RISC/CISC

- CISC (Complex Instruction Set Computer) is a retroactive definition that was introduced to distinguish the design from RISC microprocessors.
- In contrast to RISC, CISC chips have a large amount of different and complex instruction.
- The argument for its continued use indicates that the chip designers should make life easier for the programmer by reducing the amount of instructions required to program the CPU

CISC Rationale

- Historic:
  - High cost of memory.
  - Need for compact code.
- Support for high-level languages
- Ease of adding new microinstructions

CISC Effects

- Compact code
- Ease of compiler design
- Software easier to debug
- Moved complexity from s/w to h/w

=> Lengthened design times

=> Increased design errors

RISC Evolution

- Increasingly cheap memory
- Improvement in compiler technology

Patterson: "Make the common case fast"

RISC Effect

- Move complexity from h/w to s/w
- Provided a single-chip solution
- Better use of chip area
- Better Speed
- Feasibility of pipelining
  - Single cycle execution stages
  - Uniform Instruction Format

Key arguments

- RISC argument
  - for a given technology, RISC implementation will be faster
  - current VLSI technology enables single-chip RISC
  - when technology enabled single-chip CISC, RISC were pipelined
  - when technology enabled pipelined CISC, RISC had caches
  - ...
- CISC argument
  - CISC flaws not fundamental (fixed with more transistors)
  - Moore's Law will narrow the RISC/CISC gap (true)
  - software costs will dominate (very true)

And now .. Hybrid Architectures (accelerators)

Role of Compiler:RISC vs. CISC

- CISC instruction:

  MUL <addr1>, <addr2>

- RISC instructions:

  LOAD A, <addr1>

  LOAD B, <addr2>

  MUL A, B

  STORE <addr1>

- RISC is dependent on optimizing compilers

Post-RISC Architecture

- Additional functional units for superscalar
- Additional “non-RISC” (but fast) instructions
- Increased pipeline depth
- Branch prediction
- Out of order execution