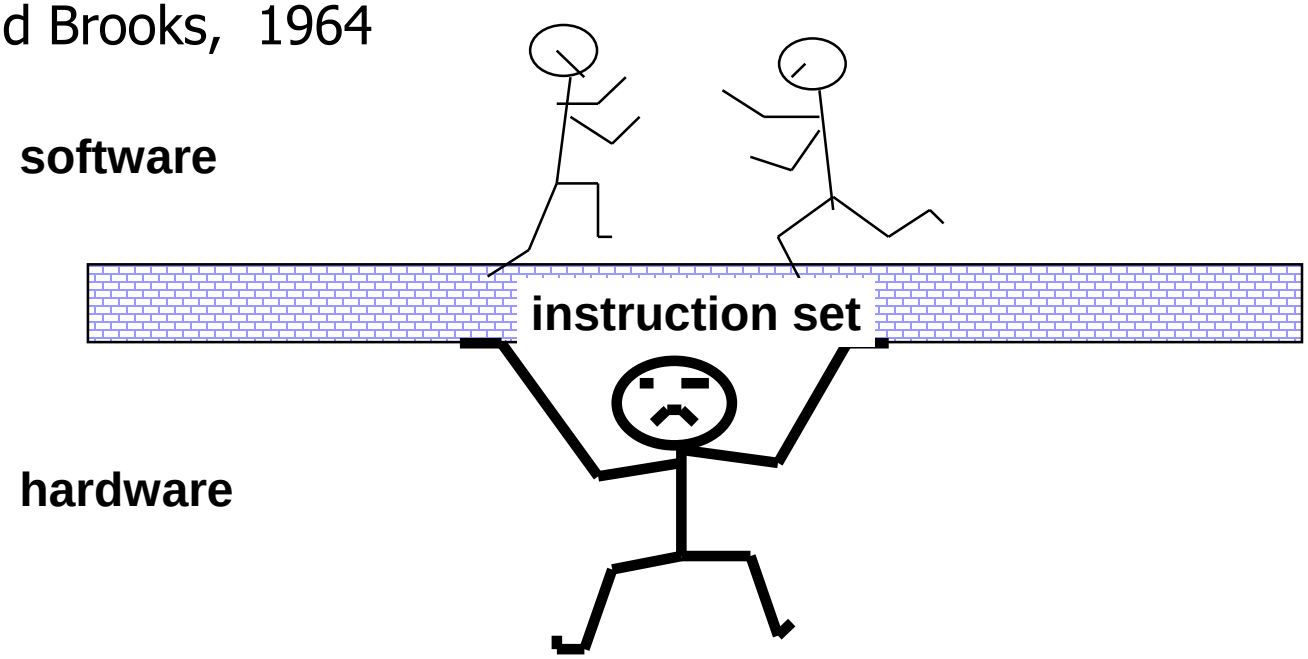
Instruction Set Architecture

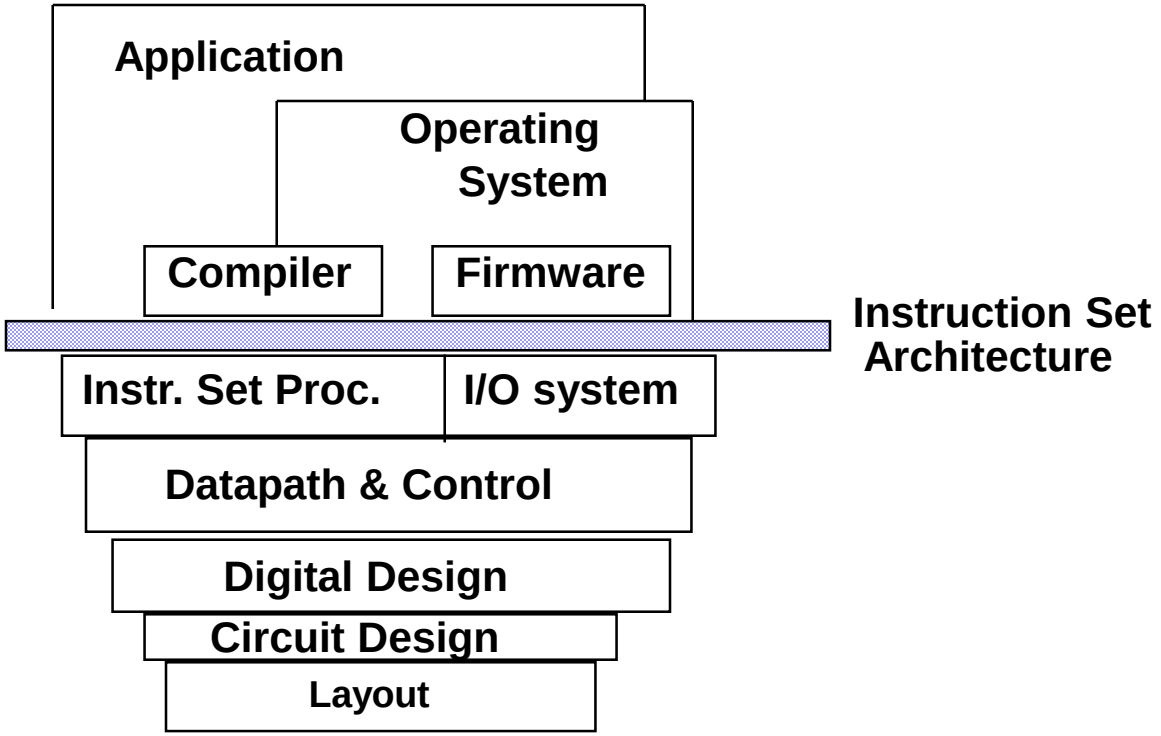
M S Bhat
Dept. of E&C,
NITK Suratkal

The Instruction Set: a Critical Interface

- ISA refers to the attributes of a [computing] system as seen by the programmer, i.e. the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls of the logic design, and the physical implementation.
  - Amdahl and Brooks, 1964



The Instruction Set: a Critical Interface



- Coordination of many *levels of abstraction*
- Under a rapidly changing set of forces
- Design, Measurement, and Evaluation

Instruction Set Architecture

- What is an ISA?
  - Organization of Programmable Storage
  - Data Types & Data Structures:
    - Encodings & Representations
  - Instruction Formats and Instruction (or OpCode) Set
  - Modes of Addressing and Accessing Data Items and Instructions
  - Exceptional Conditions
- How is it represented?
  - By a binary format, typically bits, bytes, words.
  - Word size is typically 16, 32, 64 bits
  - Length format options
    - Fixed – each instruction encoded in same size field
    - Variable – HW, W, Multiple word instructions possible

Key ISA decisions

Instruction length

- Are all instructions the same length?

How many registers?

Where do operands reside?

- e.g., Can you add contents of memory to a register?

Instruction format

- Which bits designate what??

Operands

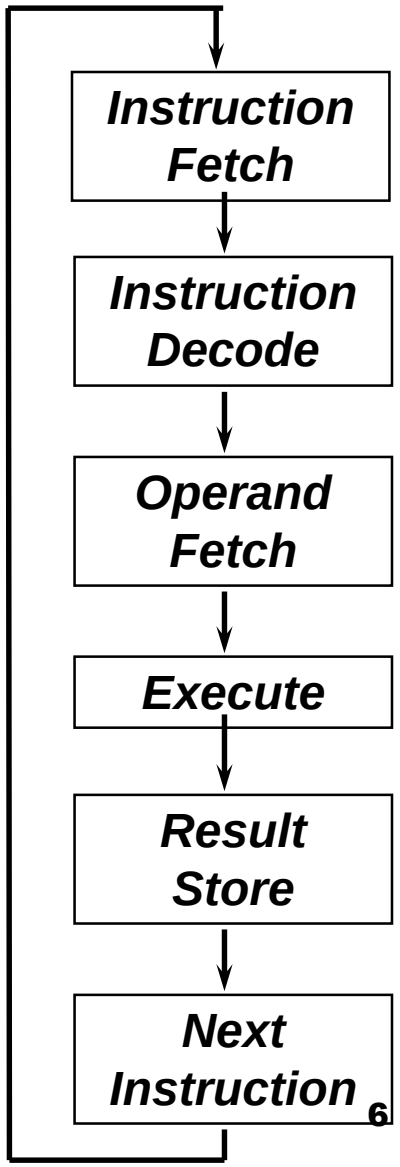
- How many? how big?
- How are memory addresses computed?

Operations

- What operations are provided??

ISA: What must be specified?

- Instruction Format or Encoding
  - How is it decoded?
- Location of operands and result
  - Where other than memory?
  - How many explicit operands?
  - How are memory operands located?
  - Which can or cannot be in memory?
- Data type and Size
- Operations
  - What are supported
- Successor instruction
  - jumps, conditions, branches
  - *fetch-decode-execute is implicit!*



Organization

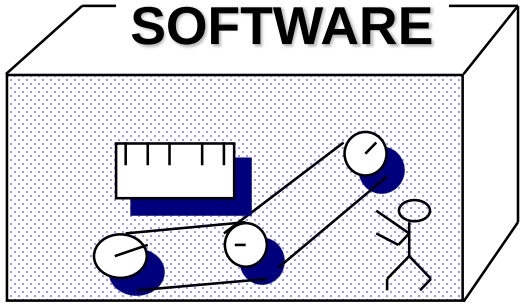
- Capabilities & Performance Characteristics of Principal Functional Units (e.g., Registers, ALU, Shifters, Logic Units, ...)

- Ways in which these components are interconnected
  - Information flows between components
  - Logic and means by which such information flow is controlled
  - Register Transfer Level (RTL) Description

Logic Designer's View

ISA Level

FUs & Interconnect



Different types of ISAs

- Determined by means for storing data in CPU:
- The major choices are:
  - An accumulator, a stack or a set of registers
- Stack architecture:
  - Operands are implicitly on top of the stack
- Accumulator architecture:
  - One operand is in an accumulator (register) and the others are elsewhere (Essentially this is a 1 register machine) (Found in earlier machines…)
- General purpose registers:
  - Operands are in registers or specific memory locations
- Load/Store
  - Operands are in explicit GPR

Basic ISA classes

- Most real machines are hybrids of these
- **Accumulator (1 register):**

1 address	add A	**acc <- acc + mem[A]**
1+x address	addx A	**acc <- acc + mem[A + x]**

- **Stack**:

0 address	add	**tos <- tos + nos (next on stack)**

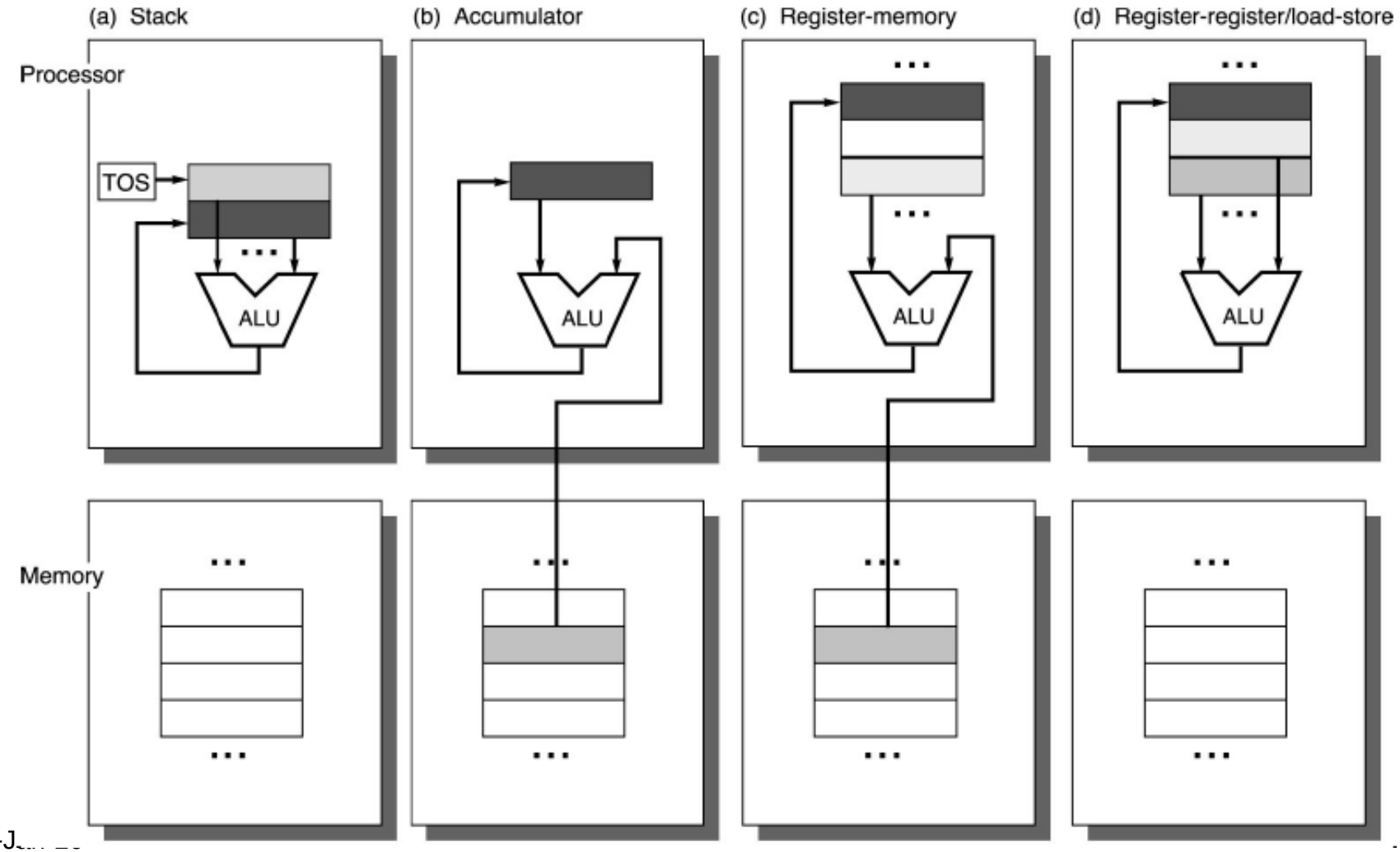
- **General Purpose Register** (can be memory/memory):

2 address	add A B	**EA[A] <- EA[A] + EA[B]**
3 address	add A B C	**EA[A] <- EA[B] + EA[C]**

- **Load/Store**:

3 address	add Ra Rb Rc	**Ra <- Rb + Rc**
	load Ra Rb	**Ra <- mem[Rb]**
	store Ra Rb	**mem[Rb] <- Ra**

Basic ISA classes (cont'd)



Comparing Number of Instructions

- Code sequence for (C = A + B) for four classes of instruction sets: (A, B, C are implicit memory locations for stack and accumulator architectures.)

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

- x86 FP uses stack (complicates pipelining)
- All RISC ISAs are load/store
- IBM360, Intel x86, M68K are register-memory

Stacks: Pros and Cons

- Pros
  - Good code density (implicit top of stack)
  - Low hardware requirements
  - Easy to write a simpler compiler for stack architectures
- Cons
  - Stack becomes the bottleneck
  - Little ability for parallelism or pipelining
  - Data is not always at the top of stack when needed, so additional instructions like TOP and SWAP are needed
  - Difficult to write an optimizing compiler for stack architectures

Accumulators: Pros and Cons

- Pros
  - Very low hardware requirements
  - Easy to design and understand
- Cons
  - Accumulator becomes the bottleneck
  - Little ability for parallelism or pipelining
  - High memory traffic

Memory-Memory: Pros and Cons

- Pros
  - Requires fewer instructions (especially if 3 operands)
  - Easy to write compilers for
- Cons
  - Very high memory traffic (especially if 3 operands)
  - Variable number of CPI

Register - Memory : Pros and Cons

- Pros
  - Instruction format easy to encode
  - Good code density
- Cons
  - Operands are not equivalent (poor orthogonal)
  - Variable number of clocks per instruction
  - May limit number of registers

In computer engineering, an **orthogonal instruction set** is an instruction set architecture where **all instruction types can use all addressing modes**. It is "orthogonal" in the sense that the instruction type and the addressing mode vary independently. An **orthogonal instruction set does not impose a limitation that requires a certain instruction to use a specific register**

Load-Store: Pros and Cons

- Pros
  - Simple, fixed length instruction encodings
  - Instructions take similar number of cycles
  - Relatively easy to pipeline and make superscalar

- Cons
  - Higher instruction count
  - Not all instructions need three operands
  - Dependent on good compiler

Evolution of Instruction Sets

- There are *no standards* to follow in designing an instruction set
- The trend up to early 80's was based on the *CISC* (Complex Instruction Set Computers) instruction set design philosophy:
  - include many instructions in the set,
  - have complex instructions that carry out the job of several simpler instructions (e.g., loop instruction),
  - have many instruction formats and addressing modes,
  - have many different kinds of registers,

Evolution of Instruction Sets (cont'd)

- A CISC architecture causes:
  - Size of the machine language programs to shrink
    - simpler (smaller) with less number of instructions to execute.
  - Complexity of the machine architecture
    - more complex (due to complex instructions), requiring more time for execution of each instruction.
  - Compiler optimizations
    - more complex (due to many choices in deciding which instructions and/or addressing modes to use).

Evolution of Instruction Sets (cont'd)

- Since the early 80's the trend for new ISA:
  - RISC (Reduced Instruction Set Computers) instruction set philosophy
    - Simplicity favors regularity
      - all instructions are of the same size
      - all instructions of the same type & follow the same format

Evolution of Instruction Sets (cont'd)

- RISC ISA design Philosophies:
  - Smaller is faster
    - small number of instructions
    - relatively small number of register types
  - Good design demands compromise
    - only a few instruction formats to handle special needs
  - Make the common case fast
    - most often executed instructions or heavily used features need to be optimized
  - Pipelining, single-cycle execution, compiler technology, etc.

Addressing Modes

- Addressing modes refer to how to specify the location of an operand (effective address)
- Addressing modes have the ability to:
  - Significantly reduce instruction counts
  - Increase the average CPI
  - Increase the complexity of building a machine
- Famous addressing modes can be classified based on:
  - the source of the data - register, immediate or memory
  - the address calculation - direct and indirect
- An indexed addressing mode is usually provided to allow efficient implementation of loops and array access

Examples of addressing modes

Address. mode	Example	Meaning	When used
Register	ADD R4, R3	*Regs[R4] = Regs[R4] + Regs[R3]*	When a value is in a register
Immediate	ADD R4, #3	*Regs[R4] = Regs[R4] + 3*	For constants
Displacement	ADD R4, 100 (R1)	*Regs[R4] = Regs[R4] + Mem[100 + Regs[R1]]*	Accessing local variables
Register indirect	ADD R4, (R1)	*Regs[R4] = Regs[R4] + Mem[Regs[R1]]*	Accessing using a pointer or a computed address
Indexed	ADD R4, (R1 + R2)	*Regs[R4] = Regs[R4] + Mem[Regs[R1] + Regs[R2]]*	Sometimes useful in array addressing: R1 = base of the array: R2 = index amount
Direct or absolute	ADD R4, (1001)	*Regs[R4] = Regs[R4] + Mem[1001]*	Sometimes useful for accessing static data; address constant may need to be large
Memory indirect or memory deferred	ADD R4, @(R3)	*Regs[R4] = Regs[R4] + Mem[Mem[Regs[R3]]]*	If R3 is the address of the pointer p, then mode yields *p
Autoincrement	ADD R4, (R2) +	*Regs[R4] = Regs[R4] + Mem[Regs[R2]]* *Regs[R2] = Regs[R2] + d*	Useful for stepping through arrays within a loop. R2 points to start of the array; each reference increments R2 by d.
Auto decrement	ADD R4, -(R2)	*Regs[R2] = Regs[R2] – d* *Regs[R4] = Regs[R4] + Mem[Regs[R2]]*	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack
Scaled	ADD R4, 100 (R2) [R3]	*Regs[R4] = Regs[R4] + Mem[100 + Regs[R2] + Regs[R3] * d]*	Used to index arrays.

General Purpose Registers Dominate

- Advantages of registers
  - Faster than cache or main memory (no addressing mode or tags)
  - Deterministic (no misses)
  - Can replicate (multiple read ports)
  - Short identifier (typically 3 to 8 bits)
  - Reduce memory traffic
  - Registers are easier for a compiler to use
    - e.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack (for stack architecture – use reverse polish notation)
  - Registers can hold variables
    - Memory traffic is reduced, so program speeds up
    - Code density improves (since reg. named with fewer bits than mem. Location)

General Purpose Registers (cont'd)

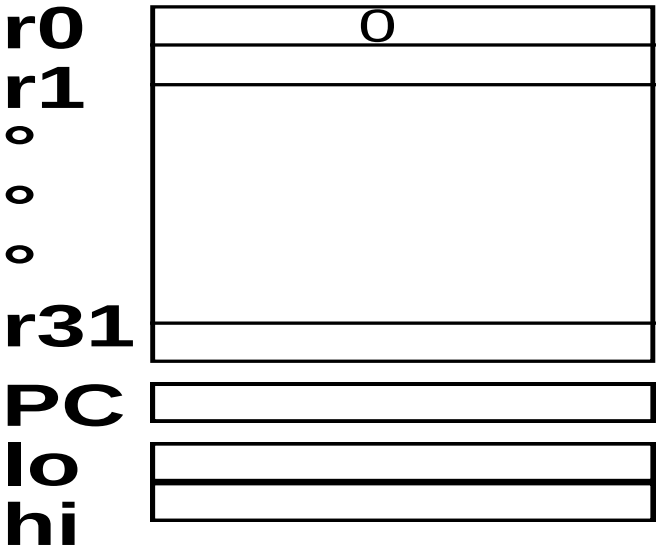
- Disadvantages
  - Need to save and restore on procedure calls and context switch
  - Can't take the address of a register (for pointers)
  - Fixed size (can't store strings or structures efficiently)
  - Compiler must manage
  - Limited number

Register – Register (0,3)

- Notation (m,n): *m* memory operands, *n* total operands in ALU instruction
- Advantages:
  - Simple fixed length instruction encoding
  - Decode is simple since instruction types are small
  - Simple code generation model
  - Instruction CPI tends to be very uniform
- Disadvantages
  - Instruction count tends to be higher
  - Some instructions are short – wasting instruction word bits

Example: MIPS Registers

- Programmable storage
  - 2^32 x bytes of memory
  - 31 x 32-bit GPRs (R0 = 0)
  - 32 x 32-bit FP regs
  - HI, LO, PC



Register-Memory (1,2)

- Evolved RISC and also old CISC
- Register-Memory ALU architecture
- Advantages:
  - Data access to ALU immediate without loading first
  - Instruction format is relatively simple
  - Code Density is improved over Register (0,3) model
- Disadvantages
  - Operands are not equivalent – source may be destroyed
  - Need for memory address field (in registers)
  - CPI will vary

Memory-Memory (3,3)

- True Memory-Memory architecture
- True and most complex CISC model
  - Currently almost extinct
- Advantages
  - Most compact
  - Doesn’t waste registers for temporary values
- Disadvantages
  - Large variation in instruction size
  - Large variation in CPI
  - Slow

Memory Addressing

- All architectures clearly need some way to address memory
  - What is accessed – byte, word, multiple words?
    - Since 1980's, almost all machines are byte addressable
    - But the main memory is organized in 32-64 bits lines to match cache model
  - Can a word be placed on any boundary?
    - Accessing a word or double-word which crosses 2 lines requires 2 references
    - Automatic alignment is possible but hides the number of references

Byte Ordering (“Endianness”)

- Layout of multi-byte operands in memory
- Little endian (x86)
  - Least significant byte at lowest address in memory
- Big endian (most other ISAs - IBM, Motorola, Sun, HP)
  - Most significant byte at lowest address in memory
  - Assume this ordering unless otherwise specified
- Some ISAs support both byte ordering (e.g., ARM, by default little endian)
- At word address 100 (assume a 4-byte word)

```
long a = 0x11223344;
```

little-endian (LSB at word address) layout

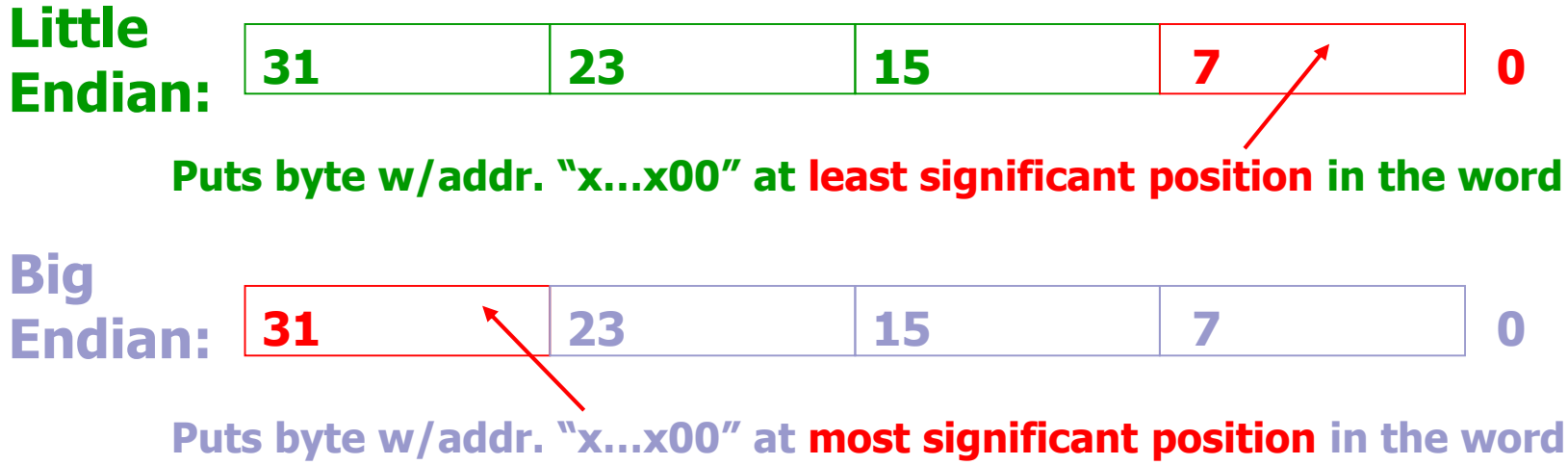
```
              103 102 101 100
              11  22  33  44    100
              +3  +2  +1  +0
```

big-endian (MSB at word address) layout

```
              100 101 102 103
    100       11  22  33  44
              +0  +1  +2  +3
```

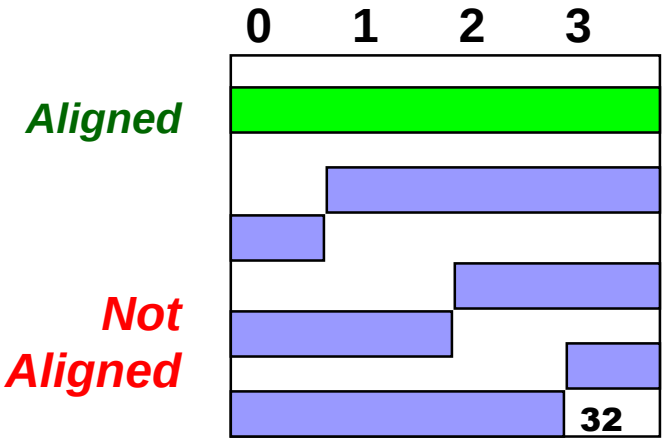
Byte ordering (cont'd)

- Usually instruction sets are byte addressed
- Provide access for bytes (8 bits), half-words (16 bits), words (32 bits), & double words (64 bits)
- Two different ordering types: big/little endian



Alignment issues

- Alignment: require that objects fall on address that is multiple of their size
- If the architecture restricts memory accesses to be aligned then
  - Software must guarantee alignment
  - Hardware detects misalignment access and traps
  - No extra time is spent when data is aligned
- Common convention is to expect aligned data objects
  - Compiler is responsible for keeping this straight
  - Implies that op-code is type specific
    - Load Byte, Load Word

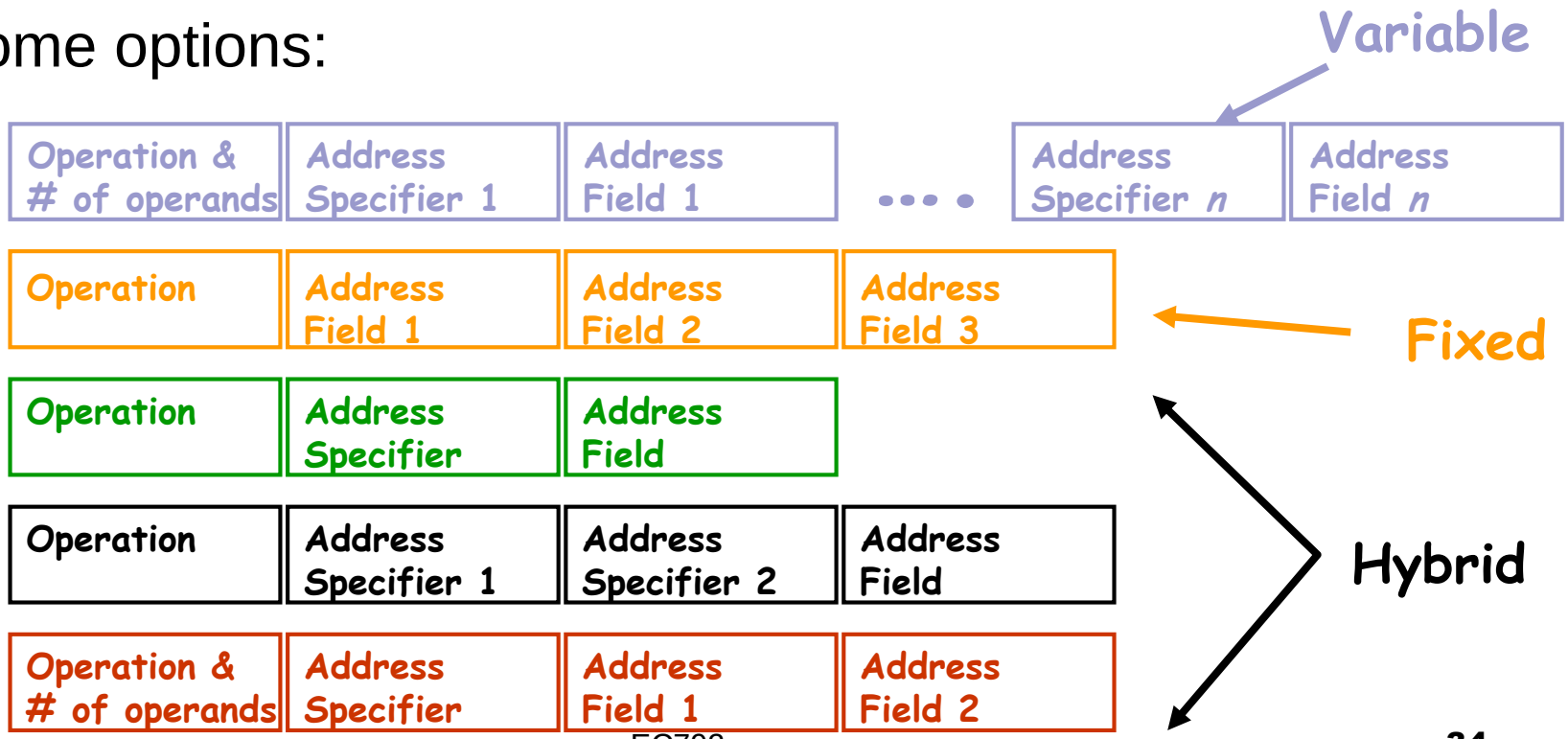


Instruction Encoding

- Instruction must specify
  - What is supposed to be done (opcode)
  - What are the operands
- Three popular formats
- Variable format (VAX, x86)
  - Opcode specifies how many operands are listed after opcode
  - Each operand has an address specifier and an address field
  - Address specifier describes addressing mode for that operand
- Fixed format (RISC)
  - All instructions of the same size
  - Opcode specifies addressing mode for load/store operations
  - All other operations use register operands
- Hybrid Format (IBM 360, some DSP processors)
  - Several (but few) fixed size instruction formats
  - Some formats have address specifier fields

How is the operation specified?

- Typically in a bit field called the opcode
- Also must encode *addressing modes* .
- Some options:



Instruction Encoding Tradeoffs

- Decoding vs. Programming
  - Fixed formats easy to decode
  - Variable formats easier to program (assembler)

- Number of registers
  - More registers help optimization (a lot)
  - Operand fields smaller with few registers
  - In general, we want many (e.g. 32) registers,

- If code size is most important,
  - use variable length instructions

- If performance is most important,
  - use fixed length instructions

The role of compilers

- Register allocation:
  - Decide which variables go to registers and which stay in memory
- Common sub-expression elimination
- Constant propagation
  - Replace a “constant” variable by a constant
- Code motion
  - Move iteration independent code outside loop body

Compilers Phases

- Compilers use phases to manage complexity
  - Front end
    - Convert language to intermediate form
  - High level optimizer
    - Procedure inlining and loop transformations
  - Global optimizer
    - Global and local optimization, plus register allocation
  - Code generator (and assembler)
    - Dependency elimination, instruction selection, pipeline scheduling

Compilers and ISA

- How do you help the compiler writer, in terms of designing the right ISA?
- Ease of compilation
  - Orthogonality: no special registers, few special cases, all operand modes available with any data type or instruction type
  - Completeness: support for a wide range of operations and target applications
  - Regularity: no overloading for the meanings of instruction fields
  - Streamlined: resource needs easily determined
- Register Assignment is critical
  - Easier if lots of registers

ISA Summary

- Evaluate ISA with respect to:
  - Number of Instructions executed
    - Nature of the high level language
    - Influence on the compiler and the optimization options
  - CPI and cycle time
    - Implied understanding of the control and data paths
    - Instruction decode complexity
    - Effective address generation complexity
    - Memory data access patterns
    - Control structure implementation options
  - Encoding and alignment issue