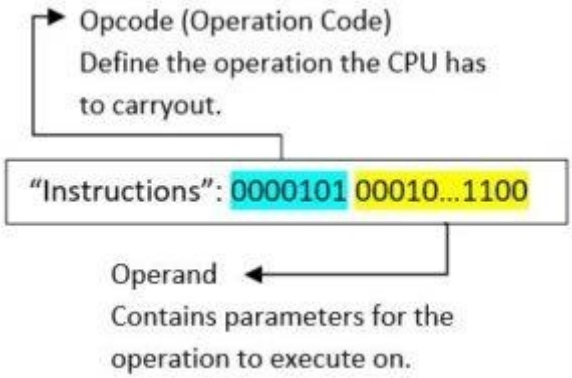
Instruction Set Architecture

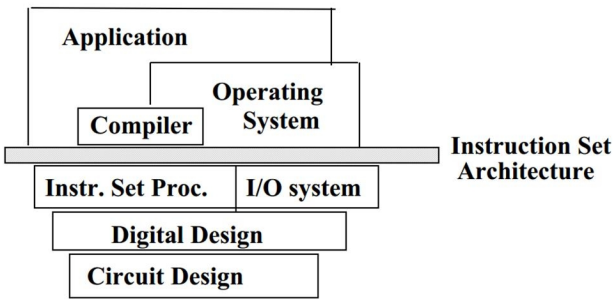
The instruction set architecture (ISA), also called instruction set is part of a computer that is applicable to programming, which is basically machine language. The instruction set provides commands to the processor, to tell it what it needs to do. The instruction set consists of addressing modes, instructions, data types, registers, memory architecture, interrupt and exception handling, and external I/O.

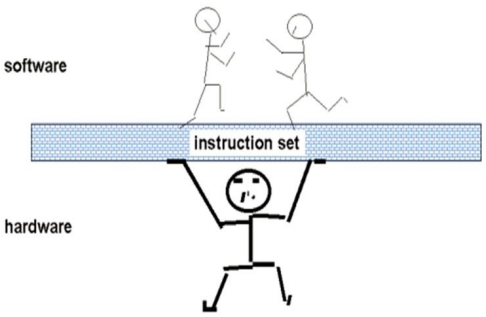
The instruction set architecture is also the machine description that a hardware designer must understand to design a correct implementation of the computer.

Instruction sets have their rules like a language has grammar. An instruction includes an opcode that specifies the operation to perform, such as add/multiply/move contents of memory to register and operand specifiers, which may specify registers, memory locations, or literal data.



- The ISA serves as the boundary between software and hardware.
- Provides a mechanism by which the software tells the hardware what should be done.





- What Makes a Good ISA?

Programmability
• Easy to express programs efficiently?

Implementability
• Easy to design high-performance implementations?

Compatibility
• Easy to maintain programmability as languages and programs evolve?
• Easy to maintain implementability as technology evolves?
• Easy to express programs efficiently?

Interface Design
Interface: A device or program enabling a user to communicate with a computer.

What makes a good user interface?

A good interface makes it easy for users to tell the computer what they want to do, for the computer to request information from the users, and for the computer to present understandable information. Clear communication between the user and the computer is the working premise of good UI design.

Good interfaces are:

Clear

A clear interface helps prevent user errors, makes important information obvious, and contributes to ease of learning and use.

Consistent

A consistent interface allows users to apply previously learned knowledge to new tasks. Effective applications are both consistent within themselves and consistent with one another.

Simple

The best interface designs are simple. Simple designs are easy to learn and to use and give the interface a consistent look. A good design requires a good balance between maximizing functionality and maintaining simplicity through progressive declaration of information.

User-Controlled

The user, not the computer, initiates and controls all actions.

Direct

Users must see the visible cause-and-effect relationship between the actions they take and the objects on the screen. This allows users to feel that they are in charge of the computer's activities.

Forgiving

Users make mistakes. User actions should be reversible. A good interface facilitates exploration and trial and error learning.

Provide feedback

Keep the user informed and provide immediate feedback. Also, ensure that feedback is appropriate to the task.

Aesthetic

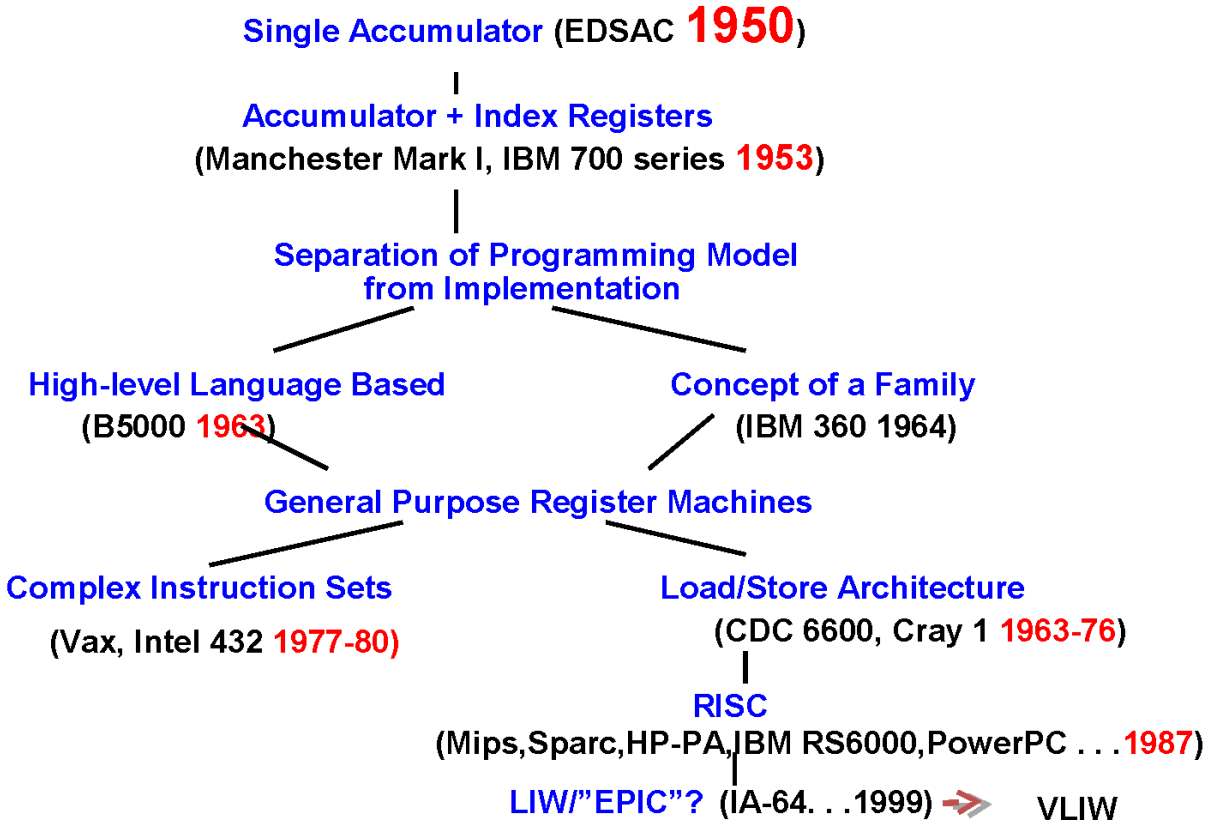
Every visual element that appears on the screen potentially competes for the user's attention. Provide an environment that is pleasant to work in and contributes to the user's understanding of the information presented.

Instruction Set Design Issues

Instruction set design issues include:

- Where are operands stored? : registers, memory, stack, accumulator
- How many explicit operands are there? : 0, 1, 2, or 3
- How is the operand location specified? : register, immediate, indirect, . . .
- What type & size of operands are supported? : byte, int, float, double, string, . .
- What operations are supported? : add, sub, mul, move, compare . . .

Evolution of Instruction Sets



Classifying ISAs

Accumulator(before 1960s e.g. 68HC11)

- 1-address
  - add A
- acc ← acc + mem[A]

Stack(1960s to 1970s):

- 0 (zero) addresses
- TOS ← TOS + next [∵ TOS = Top Of Stack]

Memory-Memory (1970s to 1980s):

- 2-address
  - add A,B mem[A] ← mem[A] + mem[B]
- 3-address
  - add A,B,C mem[A] ← mem[B] + mem[C]

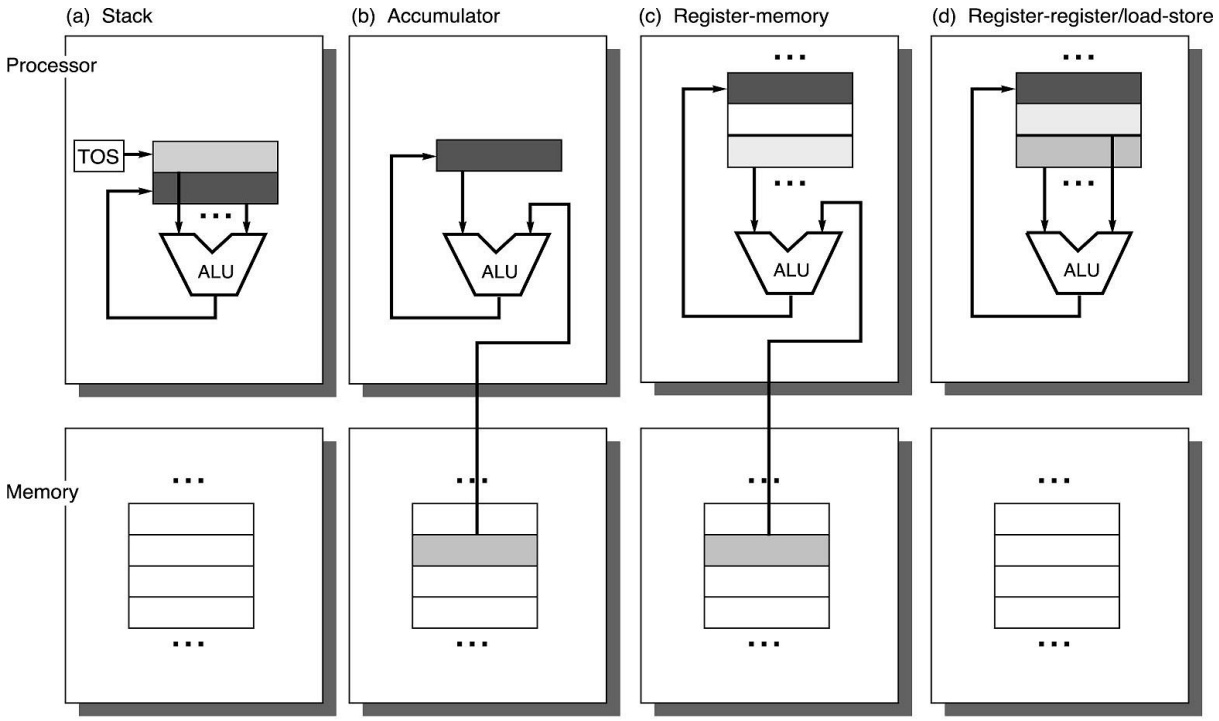
Register-Memory (1970s to present, e.g. 80x86):

- 2-address
- add R1, A R1 ← R1 + mem[A]
- load R1, A R1 ← mem[A]

Register-Register (Load/Store) (1960s to present, e.g. MIPS):

- 3-address
  - add R1, R2, R3 R1 ← R2 + R3
- load R1, R2 R1 ← mem[R2]
- store R1, R2 mem[R1] ← R2

Operand Locations in Four ISA Classes: (Basic Addressing Classes)



➢ Stack Architectures:

```
A=B+(C*D)
push B
push C
push D
mul
add
pop A
```

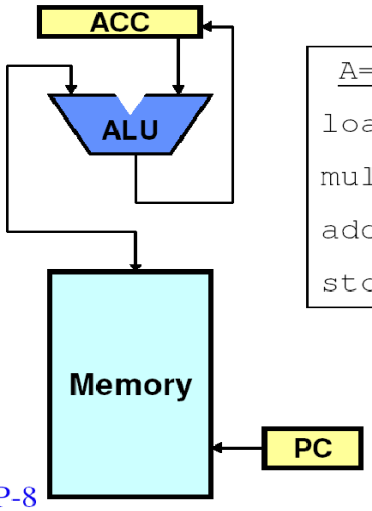
Stack: First In Last Out data structure
Instruction Operands:
- One for push/pop,
- None for ALU operations

Advantages:	Disadvantages:
- Short instructions	- Code is inefficient
- Compiler is easy to write	- Stack size and latency
- Low hardware requirements	- Little ability for parallelism or pipelining

Example: Java VM

Accumulator Architectures

- Single register (accumulator)
- Instructions
  - ALU (Acc ← Acc + *M)
  - Load to accumulator (Acc ← *M)
  - Store from accumulator (*M ← Acc)
- Instruction operands
  - One explicit (memory address)
  - One implicit (accumulator)
- Attributes:
  - Short instructions
  - Minimal internal state; simple design
  - Many loads and stores
- Examples:
  - Early machines: IBM 7090, DEC PDP-8
  - Today: DSP architectures




```
A=B+(C*D)
load C
mul D
add B
store A
```

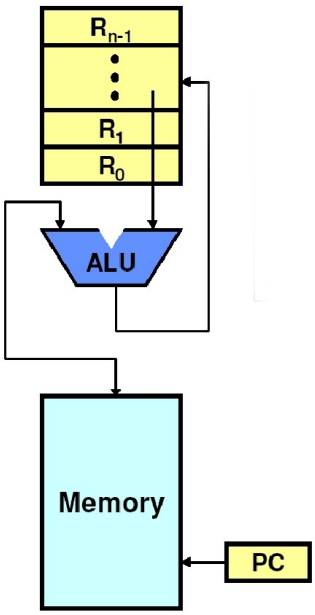
Memory-to-Memory Architectures

- All ALU operands from memory addresses
- Advantages
  - No register wastage
  - Lowest instruction count
- Disadvantages
  - Large variation in instruction length
  - Large variation in clocks per instructions
  - Huge memory traffic
- Examples
  - VAX

```
D=B+(C*D)
mul D <- C*D
add D <- D+B
```

- **Register to Memory Architecture**

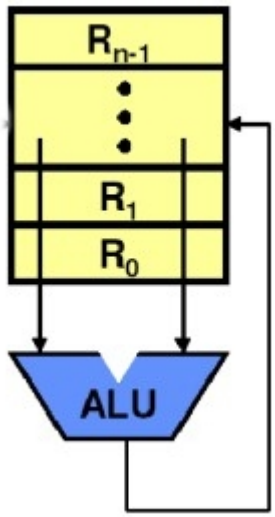
  One memory address in ALU ops
  Typically 2-operand ALU ops
  Advantage : small instruction count
  Disadvantage: - instruction length varies
  - harder to pipeline
  Example: IBM 360/370, VAX




```
A=B+(C*D)
load R1<-C
mul R1<-R1*D
add R1<-R1+B
store A<-R1
```

- **Register to Register Architecture**

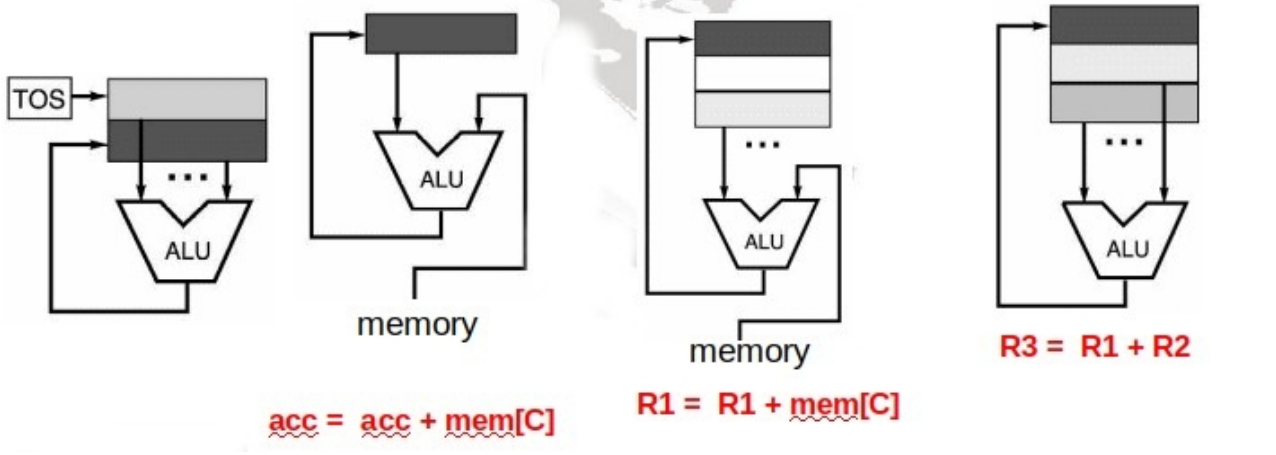
  - No memory addresses in ALU ops
  Advantage : - Simple fixed-length instructions
  - Relatively easy to pipeline
  Disadvantage: - Higher instruction count
  - Dependent on good compiler
  Ex: CRAY-1, most RISCs




```
A=B+(C*D)
load R1<-C
load R2<-D
load R3<-B
mul R4<-R1*R2
add R5<-R4+R3
store A<-R5
```

Code Sequence C = A + B for Four Instruction Sets

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



Why do almost all new architectures use registers?

- Registers are much faster than memory (even cache)
- Register values are available immediately
- Registers are convenient for variable storage
- Compiler assigns some variables just to registers
- More compact code since small fields specify registers(compared to memory addresses)
- Reduce memory traffic

Types of Operand Operations

Data Movement	Load (from memory) Store (to memory) memory-to-memory move register-to-register move input (from I/O device) output (to I/O device) push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP Add, Subtract, Multiply, Divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return

ISA Traditional Issues

- Strongly constrained by the number of bits available to instruction encoding
- Opcodes/operands
- Registers/memory
- Addressing modes
- 0, 1, 2, 3 address machines
- Instruction formats
- Decoding uniformity