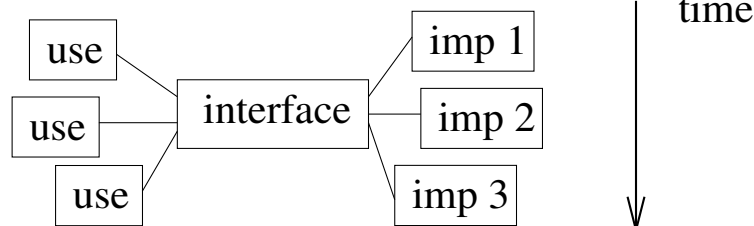


## Instruction Set Design: Principles

## What is Instruction Set Architecture

- **Instruction Set Architecture**: portion of the machine that is visible to the programmer or the compiler writer
- A good interface:
  - Lasts through many implementations (portability, compatibility)
  - Is used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels



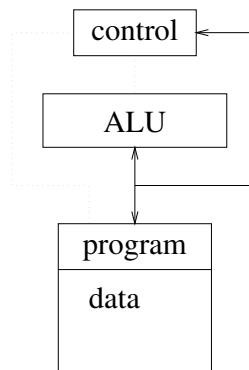
## Evolution of Instruction Sets

- Major advances in computer architecture are typically associated with landmark instruction set designs
- Instruction set design decisions must take into account:
  - technology
  - machine organization
  - programming languages
  - compiler technology
  - operating systems
- And they, in turn, influence these

## Von Neumann Machines

Von Neumann “invented” stored program computers in 1945.

- Instead of program code being hardwired, the program code is placed in memory along with the data



## Anatomy of A Stored Program Computer

- **ALU (arithmetic and logic unit)**
  - capable of performing one of few arithmetic and logical operations like add, subtract, negate, logical and, logical or, etc...
  - operation performed is selected by controller
  - operations are performed on operands of a limited fixed size (8, 16 or 32 bits)
  - a small number of registers to hold operands
- **Memory**
  - a set of cells each capable of holding a small fixed size binary number (8, 16 or 32 bits for example)
  - cells are indexed with consecutive integers (addresses)
  - cell address and transfer direction (read or write) are set by controller
  - data is transferred to/from ALU or to controller

## Anatomy of a Stored Program Computer

- **Controller**
  - contains a counter called the program counter
  - program counter contains the address of the instruction being executed
  - program counter automatically incremented after each instruction
  - program counter can be changed by certain instructions
  - machine control depends on the code at the cell whose address is in the program counter
- **Controller algorithm**
  - fetch memory cell (instruction) using PC as address
  - increment program counter to get ready for next instruction
  - decode instruction: set control lines of ALU and memory according to the code of the instruction
  - execute
  - repeat

## Memory-to-Memory Machine

- Every instruction contains a full memory address for each operand
- Assumptions:
  - two operands per operation
  - second operand is also the destination
  - memory address = 16 bits (2 bytes)
  - operand size = 32 bits (4 bytes)
  - instruction code = 8 bits (1 byte)
  - we want to evaluate:  $A \leftarrow (B + C + D + E) / X$

## Memory-to-Memory Machine – cont'd.

Assembly language code (hypothetical):

```

move    B, A           ; A <- B
add     C, A           ; A <- A + C (B+C)
add     D, A           ; A <- A + D (B+C+D)
add     E, A           ; A <- A + E (B+C+D+E)
div     X, A           ; A <- A / X
  
```

- An add/div instruction results in the transfer of 17 bytes between memory and CPU:
  - 5 bytes for instruction
  - 4 bytes each to fetch 1st and 2nd operands
  - 4 bytes to store result
- Move requires 13; Total memory traffic:  $4 \times 17 + 13 = 81$  bytes

## Why CPU Storage?

Consider the idea of providing a small amount of storage in the CPU

- Goal: To reduce memory traffic by keeping repeatedly used operands in the CPU, and thus:
  - Avoid re-referencing memory
  - Avoid having to specify full memory address of the operand

This is a perfect example of “[optimizing the frequent case](#)”

## Example: Accumulator Machine

Consider a machine with 1 cell of CPU storage: the accumulator

- Assumptions:
  - two operands per operation
  - 1st operand is in the accumulator
  - 2nd operand is in memory
  - accumulator is also the destination of the operation (except for store)
  - memory address = 16 bits (2 bytes)
  - operand size = 32 bits (4 bytes)
  - instruction code = 8 bits (1 byte)
  - we want to evaluate:  $A \leftarrow (B + C + D + E)/X$

## Accumulator Machine – cont'd.

Assembly language code (hypothetical)

```
load    B                ; acc <- B
add     C                ; acc <- acc + C  (B+C)
add     D                ; acc <- acc + D  (B+C+D)
add     E                ; acc <- acc + E  (B+C+D+E)
div     X                ; acc <- acc / X
store   A                ; A <- acc
```

- Each of the above instructions results in the transfer of 7 bytes between memory and CPU:
  - 3 bytes for instruction; 4 bytes to fetch (or store) 2nd operand
  - Total memory traffic:  $6 \times 7 = 42$  bytes
- **Moral:** local CPU storage reduces memory traffic, and also effects the instruction set design

## Classification of Instruction Set Architectures

- Classification of stored programs computers based on CPU storage organization
- Characterize instructions (and machines) by the number of explicit memory addresses of operands in its instructions (excluding data move instructions)
  - Stack machines: 0 address machines
  - Accumulator machines: 1 address machines
  - General purpose register machines: 1, 2, or 3 address (or more) machines

## Stack Machines

- Most instructions manipulate the top few data items (mostly top 2) of a pushdown stack
- Additional instructions are provided to move data between memory and top of stack
- Top few items of the stack are kept in the CPU
- Instructions manipulate the top of the stack implicitly
- Ideal for evaluating expressions (stack holds intermediate results)
- Were thought to be a good match for high level languages
- Awkward:
  - become very slow if stack grows beyond CPU local storage
  - no simple way to get data from “middle of stack”

## Stack Machines

- Binary arithmetic and logic operations:
  - operands: top 2 items on stack
  - operands are removed from stack
  - result is placed on top of stack
- Unary arithmetic and logic operations:
  - operand: top item on the stack
  - operand is replaced by result of operation
- Data move operations:
  - push: place memory data on top of stack
  - pop: move top of stack to memory

## Stack Machines – Sample Program

Evaluate expression  $y \leftarrow ax^2 + bx + c$

```
push    a                ; tos: a
push    x                ; tos: a x
dup                       ; tos: a x x
mult                      ; tos: a x^2
mult                      ; tos: ax^2
push    b                ; tos: ax^2 b
push    x                ; tos: ax^2 b x
mult                      ; tos: ax^2 bx
push    c                ; tos: ax^2 bx c
add                       ; tos: ax^2 bx+c
add                       ; tos: ax^2+bx+c
pop      y               ; y <- ax^2+bx+c
```

- *Stack Machines – Examples:* B5500, HP 3000/70

## Accumulator Machines

- CPU storage consists of a single accumulator
- Accumulator is an (implicit) operand for most instructions
- Memory is source of 2nd operand for 2 operand instructions
- Cheap (in terms of CPU hardware) and simple
- Memory traffic can be reduced significantly more if more local storage is available



## Accumulator Machines – Sample Program

Evaluate the expression  $y \leftarrow ax^2 + bx + c$

```
load    a        ; acc <- a
mult    x        ; acc <- ax
mult    x        ; acc <- ax^2
store   y        ; y <- ax^2
load    b        ; acc <- b
mult    x        ; acc <- bx
add     c        ; acc <- bx + c
add     y        ; acc <- ax^2 + bx + c
store   y        ; y <- ax^2 + bx + c
```

- *Accumulator Machines – Examples:* PDP-8, MOTOROLA 6809

## General Purpose Register Machines

- With stack machines, only the top two elements of the stack are directly available to instructions. In general purpose register machines, the CPU storage is organized as a set of **registers** which are equally available to the instructions.
- Frequently used operands are placed in registers (under program control)
  - Reduces instruction size
  - Reduces memory traffic

## GPR Machines – Sample Program

Evaluate  $y \leftarrow ax^2 + bx + c$  on a hypothetical machine with 16 registers, R0 to R15, and 2 operand register-register ALU operations

```
load    x, R1    ; R1 <- x
load    a, R2    ; R2 <- a
load    b, R3    ; R3 <- b
load    c, R4    ; R4 <- c
mult    R1, R2    ; R2 <- ax
mult    R1, R2    ; R2 <- ax^2
mult    R1, R3    ; R3 <- bx
add     R2, R3    ; R3 <- ax^2 + bx
add     R3, R4    ; R4 <- ax^2 + bx + c
store   R4, y     ; y <- ax^2 + bx + c
```

- *GPR machines – Examples:* VAX, IBM 360, IBM 370, PC-RT, RISC 6000, SPARC, MIPS, ....

## Classifying GPR Machines

GPR machines are sub-classified based on whether or not memory operands can be used by typical ALU instructions

- **Register-memory machines:** machines where some ALU instructions can specify at least one memory operand and one register operand
- **Load-store machines:** register-register machines with the restriction that the only instructions that can access memory are the “load” and the “store” instructions
  - load: transfers data from memory to a register
  - store: transfers data from a register to memory

## Memory Addressing

- Motivation: Early machines provided only absolute addressing. A memory cell is to provide its address in the instructions.
- Absolute addressing is less than ideal for stepping through arrays; looping with different indices etc.
  - to use same section of code repeatedly, only way is to modify the instructions of the loop before the loop is repeated
- Problems of Self Modifying Code
  - Error prone
  - Difficult to debug; to analyze

## Index Registers

Index registers were invented (Manchester University machine, early 50's) to simplify addressing and avoid self modifying code:

- One or more index registers are added to the CPU
- A few instructions are provided to manipulate the index register
  - clear index register
  - add a small constant to an index register
- Two “addressing modes” are now available
  - absolute (same as what we had before)
  - indexed: instruction specifies which index register is to be added to the address to form a final address (effective address)

## Addressing Modes

Index registers were generalized and the different ways of specifying memory addresses became known as addressing modes

- An addressing mode specifies how the effective address is to be computed from the “ingredients” spelled out in the instruction

Notation: ‘M[x]’ refers to the data at memory location x

## Addressing Modes Encountered in Practice

mode	sample syntax	effective address
----	-----	-----
immediate	add #3,R4	none: operand in instruction
register	add R1,R2	none: operand in register
reg. deferred (or indirect)	clear (R4)	R4
displacement (based)	clear 100(R2)	100 + R2
indexed	clear (R4,R7)	R4+R7
direct	clear (943)	943

## Addressing Modes Encountered in Practice

mode ----	sample syntax -----	effective address -----
Memory indirect (mem deferred)	clear at(R3)	M[R3]
Auto increment	clear (R2)+	R2, R2 incremented
Auto decrement	clear -(R2)	R2, R2 pre-decremented
scaled	clear 10(R1) [R2]	10+R1+R2

- Addressing Modes can significantly reduce instruction count
- Addressing Modes can add to complexity of the machine

## Memory Organization

The instruction set architecture specifies the logical organization of memory in terms of four items:

- Smallest addressable unit of memory (byte? halfword? word?)
- Addressable units of memory (double-word?...)
- Alignment
- Endianness

## Alignment

For efficiency of implementation, certain architectures restrict the addresses that can be used based on the size of the data transfer

e.g.: Strict alignment. A size  $n$  ( $n$  is a power of 2) data item can be accessed only at addresses that are multiples of  $n$

- Bytes can be accessed at any address
- Halfwords can be accessed only at even addresses
- Words can be accessed only at addresses that are multiples of 4

## Endianness

How are bytes ordered within a word?

- Machines where the least significant byte of a word is stored first (smaller address) in memory are called “little Endian”
- LSByte store last (larger address) in memory: “big Endian”
- Examples:
  - DEC and INTEL: little Endians
  - IBM and MOTOROLA: big Endians
  - MIPS-CO: configurable

Raging battles over which is better

## Operations and Operands

We now consider the types of operations and operands offered by typical machines.

category	Example
Arithmetic	add, subtract, negate, multiply, divide
Logical	and, or, xor, invert
Shift; Move	logical/arithmetic shifts; data move
Control	branch, jump, procedure call and return, traps
System	OS call, virtual memory management, cache management
Floating point	FP operations: add, sub, mult, div, inverse
Decimal	BCD add, sub, mult, div, BCD to char convert
String	string move, compare, search

The first four categories are almost universal

## Control Operations

- Program counter is used to sequentially step through the instructions
- Control operations alter the sequential flow by changing the program counter (to something besides the address of the next instruction)
- Four types of control operations
  - Conditional branches
  - Jumps (unconditional)
  - Procedure calls
  - Procedure returns

## Branch Instructions

- Information needed to perform the branch
  - Target address of the branch
  - Condition to be tested
  - Result of the test (whether or not branch is taken)
- Target Specification:
  - Use one of addressing modes or
  - Specify address as a program counter offset encoded in the instruction: this makes position independent (relocatable) code very simple
- Branch Condition Test
  - Condition code (register)
  - General purpose register with compare instructions
  - Compare and branch
- Some machines use more than 1 of the above

## Condition Code for Branch Instructions

- The CPU maintains a set of 1-bit flags (C,N,Z,etc.) that are set according to the results of ALU operations
- Branch instruction specifies one of the flags as condition for the branch, or a condition that is easily computed from the flags
- Advantage: free condition computation sometimes
- Disadvantages:
  - extra state in the CPU
  - constrains code reordering
  - constrains parallelism (if two instructions execute in parallel, which one sets the condition code?)



## General Purpose Register for Branch Condition

Branch instruction specifies a general purpose register as condition for the branch; the instruction set includes compare instructions that put the comparison results in a register

- **Advantage:**
  - simple
  - no constraint on instruction reordering
- **Disadvantage:** uses up a register between comparison and branch

Example: MIPS architecture

```
slt Ra,Rb,Rc    if Ra<Rb then Rc <- 1 else Rc <- 0
                Ra, Rb and Rc stand for any 3
                registers from the set R0 to R31
```

## Compare and Branch Technique

Some architectures offer instructions that compare 2 values and branch based on the result of the comparison

- Advantage: 1 instruction instead of two
- Disadvantage: may be too much work for 1 instruction
- Example (from VAX):

```
aoblss op1,op2,target ; add 1 to op2, if op2<op1
                      ; then branch to target
```

## Operand Types and Sizes

Two possibilities for designating the operand type:

- Encoding type information in the operand
  - Ex: Tags in LISP machines
- Encoding type information in the instruction
  - We need different operation codes for different data types: byte add, word add, floating point add ...

Typical Data Types: Integer, Floating point, and Character. Some architecture also provide support for

- Character strings (e.g. string move and string compare)
- Decimal arithmetic on BCD: (add, multiply, move, pack, unpack ...)

## Encoding an Instruction Set

- Each instruction has an opcode and a set of operands
- Variable-length instructions:
  - opcode + address specifier for each operand
  - Flexible, reduces program length, ... but individual instructions vary widely in size and in amount of work to be performed
- Fixed-length instructions:
  - All instructions of the same length, load-store machines with only one memory operand and only one or two addressing modes
  - Simple, efficient to implement ... but increases program length
- Hybrid instructions

## Putting it All Together

### RISC

- No firm definition
- Generally includes:
  - General purpose registers
  - Fixed 3-address instruction format
  - Simple addressing modes and instructions
  - [Strict load-store architecture](#)
- Examples: DEC Alpha, MIPS
- Advantages:
  - [Good compiler targets](#)
  - [Easier to implement/pipeline](#)

### CISC

- May include:
  - Variable length instructions
  - Memory-register instruction
  - Complex addressing modes and instructions
- Examples: x86, IBM 360, ...
- Advantages:
  - [Better code density](#)
  - [Legacy software](#)

## Top 10 80x86 Instructions

Rank	Instruction	% occurrence
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
	<a href="#">Total</a>	<a href="#">96%</a>

## ISA Metrics

- 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
- Ease of compilation (programming?)
- Ease of implementation
- Scalability