

## 1 – Instruction Set Architecture Taxonomy

Instruction set design principles include:

1. Keep the CPI low.
2. Keep the Instructions per program low.
3. Use as little memory as possible to store the program.
4. Fetching values from memory may be **MUCH** slower than doing a computation.
5. Concentrate on making a simple, yet flexible set of features fast and emulate less frequently used ones.

Let's investigate some trends from these view points.

## 2 – Instruction Set Architecture Taxonomy

For “sequential” architectures the major architecture categories based on addressing are:

1. Stack — Reads the memory locations at the top of the stack and pushes results on the top of the stack. (AT&T Hobbit was probably the last one)
2. Accumulator — Implicitly uses a dedicated register called the *accumulator* to store results. (Used in older architectures, now obsolete).
3. Register-Memory — Operations may have an operand which resides in memory. Typically the result is written in a register. (Intel 80x86 Series)
4. (Register) Load-Store — Information is transferred into registers via *load* operations from memory and written to memory via *store* operations. (Almost all RISC chips).

### 3 – Instruction Set Architecture Taxonomy

Consider for example the code for  $C = A + B$  in each of the four instruction set styles:

Stack	Accumulator	Register Memory	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, R2, R1
Pop C			Store C, R3

## 4 – Why Register Machines Won

*General Purpose Register* (GPR) machines are overwhelmingly more popular, and most post 1980 architectures are load-store machines.

Why?

1. Registers are fast
2. Less memory traffic
3. Higher code density
4. Easier to get compiler level optimizations

More registers are (usually) better for ease of programming and efficiency reasons (counter example Intel 8086).

Registers are expensive, so there is a limit on how many a processor can have. Special purpose registers can make a machine hard to understand.

## 5 – GPR Machine Types

GPR machines can be categorized according to:

1. Register Memory Machines
2. Load Store Architectures
3. Do ALU operators take two or three operands

Examples include:

# Addresses	Max # Operands	Examples
0	3	Power PC, MIPS, ALPHA
1	2	Intel 80x86, Motorola 68K
2	2	VAX
3	3	VAX

## 6 – Cost/Benefit of GPR Machine Types

Let  $(a, b)$  represent operations with  $a$  memory operands and  $b$  total operands per operation.

1. Register-Register  $(0, 3)$  — has simple fixed length encoding possible, simplifying code generation. Instructions have a low CPI, but more instructions may be needed to express a program. Some instructions may not utilize all their encoding space.
2. Register-Memory  $(1, 2)$  — Data can be accessed without a load operation. Instruction format can be easily encoded with good density. Encoding of memory and register for each instruction limits number of registers. CPI varies by operand location.
3. Memory-Memory  $(3, 3)$  — Has large variation in instruction size and work done by an instruction. Memory access can be a bottleneck.

## 7 – Byte Ordering

Byte ordering is done in two ways:

1. *Little Endian* — The least significant byte is in the low order memory.
2. *Big Endian* — The high order byte is in the low order memory.

So if a 32 bit word were stored at location  $m$  in memory held the value  $0x12345678L$  then memory would look like:

Order	$m$	$m + 1$	$m + 2$	$m + 3$
Little Endian	$0x78$	$0x56$	$0x34$	$0x12$
Big Endian	$0x12$	$0x34$	$0x56$	$0x78$

## 8 – Memory Alignment

For memory to be accessed efficiently, the data must be *aligned* in the memory.

Hennessy and Patterson use the following notation, where  $m$  is a memory location:

Object	Size (in Bytes)	Where Aligned
Byte	1	Everywhere
Half Word	2	$m \bmod 2 = 0$
Word	4	$m \bmod 4 = 0$
Double Word	8	$m \bmod 8 = 0$

Having data aligned permits faster memory access, and typically the restriction is enforced in the name of efficiency (to keep access simple).

Compilers typically handle alignment. Most architectures use word size ALU operations, however Intel 80x86 maintains byte and halfword ALU operations.

## 9 – Addressing Modes

Most architectures support a subset of these addressing modes:

1. Register
2. Immediate
3. Displacement
4. Register deferred (indirect)
5. Indexed
6. Direct
7. Memory indirect
8. Autoincrement/Autodecrement
9. Scaled or indexed

## **10 – Cost/Benefit of Addressing Modes**

Supporting many addressing modes has a high cost in terms of chip design by adding to the complexity of the memory access system.

RISC designs tend to minimize the number/type of addressing modes to the most frequently used ones.

**Recall that one consequence of Amdahl's law is that the greatest speedup comes from optimizing the most common cases.**

## 11 – Displacement Mode Addressing

Displacement mode addressing uses a wide range of displacement values, meaning that large displacements need an encoding mechanism.

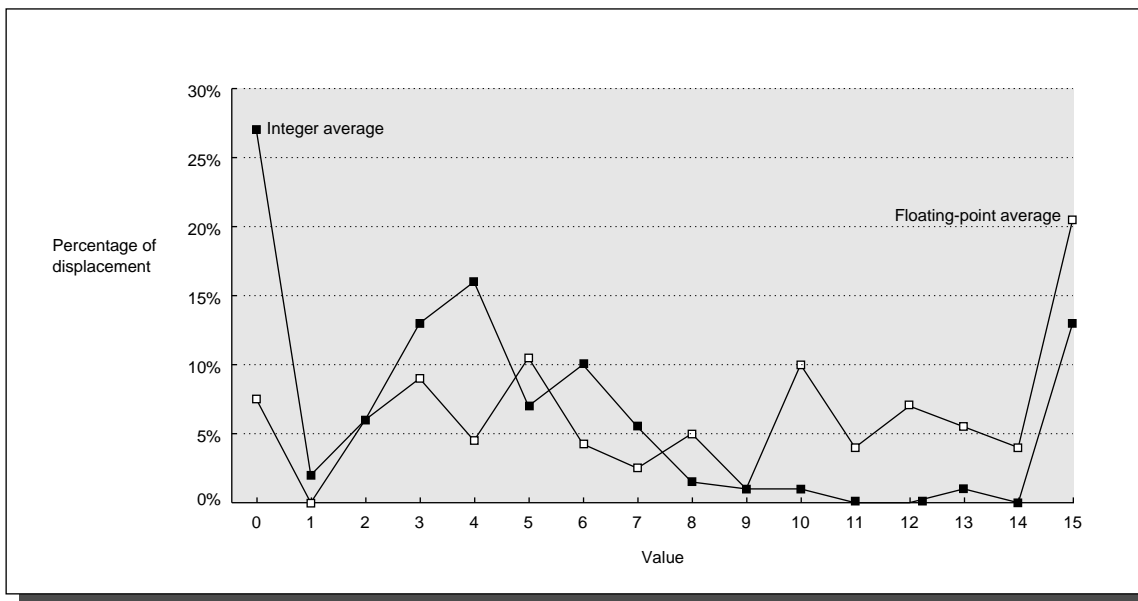
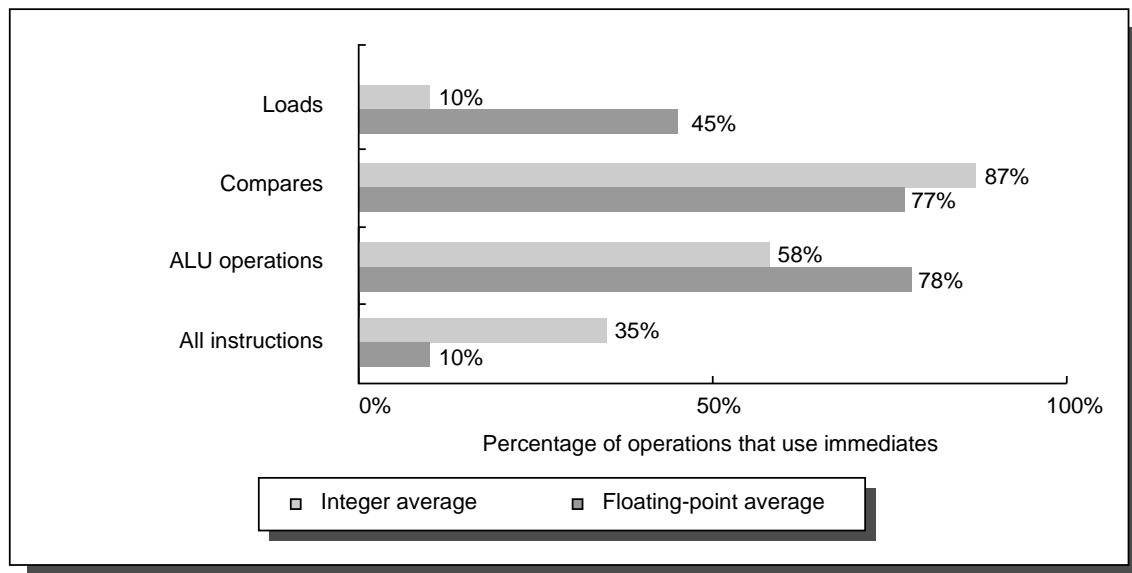


FIGURE 2.7 Displacement values are widely distributed.

## 12 – Immediate or Literal Addressing Modes

Immediate addressing modes take a value from the instruction stream as an operand. Loads, compares (branching), ALU operations all use immediate addressing for their operands.



**FIGURE 2.8** We see that for integer ALU operations about one-half to three-quarters of the operations have an immediate operand, while for integer compares 75% to 85% of the occurrences use an immediate operand.

## 13 – Range of Immediate Values

Immediate addressing patterns seem to strongly favor small values (between 75% and 80% fit in 16 bits).

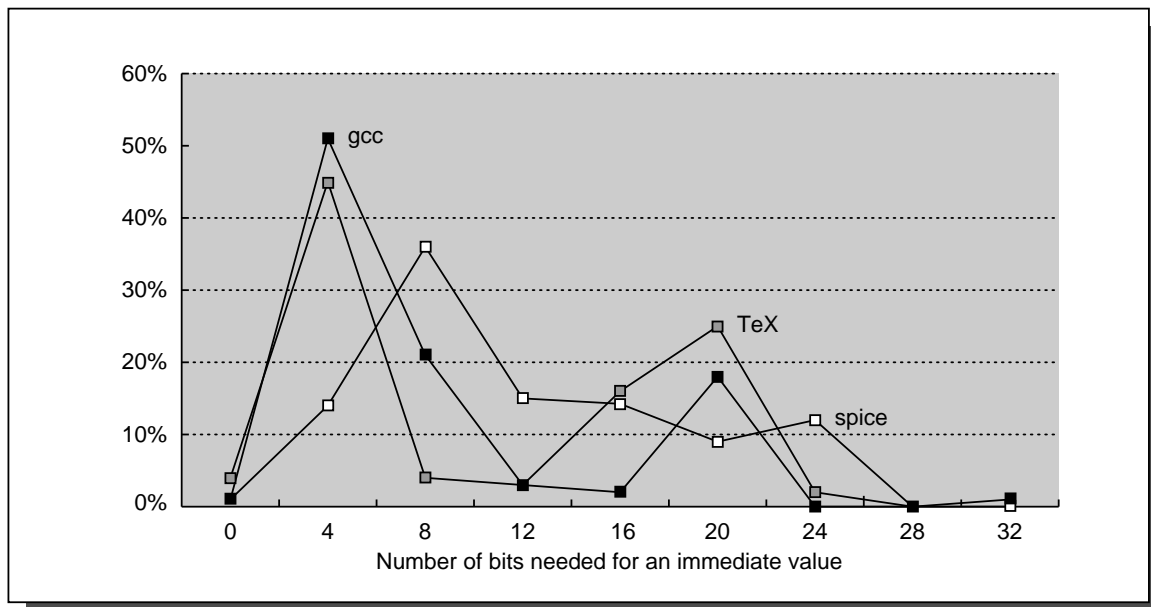


FIGURE 2.9 The distribution of immediate values is shown.

## 14 – Kinds of Operations

Operations fall into the following categories:

Type	Examples
Arithmetic and Logical	+,-,*,/ and, or, xor
Data Transfer	Load and Store
Control	branch, jump, call, return, trap
System	O/S calls, privileged ops, VM
Floating Point	fp. +,-,*,/
Decimal	BCD add and multiply
String	String move, compare, search
Graphics	Pixel Ops, compress

## 15 – Types and Size of Operands

Floating point predominantly uses 64 bit double words for operands, while integer operations rely primarily on 32 bit operands.

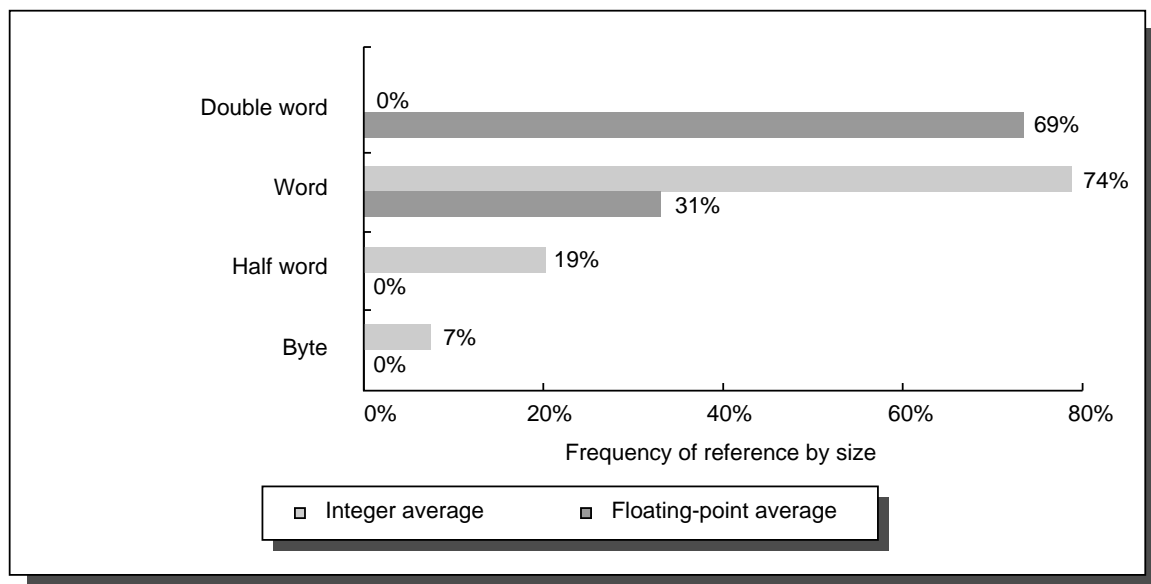


FIGURE 2.16 Distribution of data accesses by size for the benchmark programs.

## 16 – Instruction Set Encoding Strategies

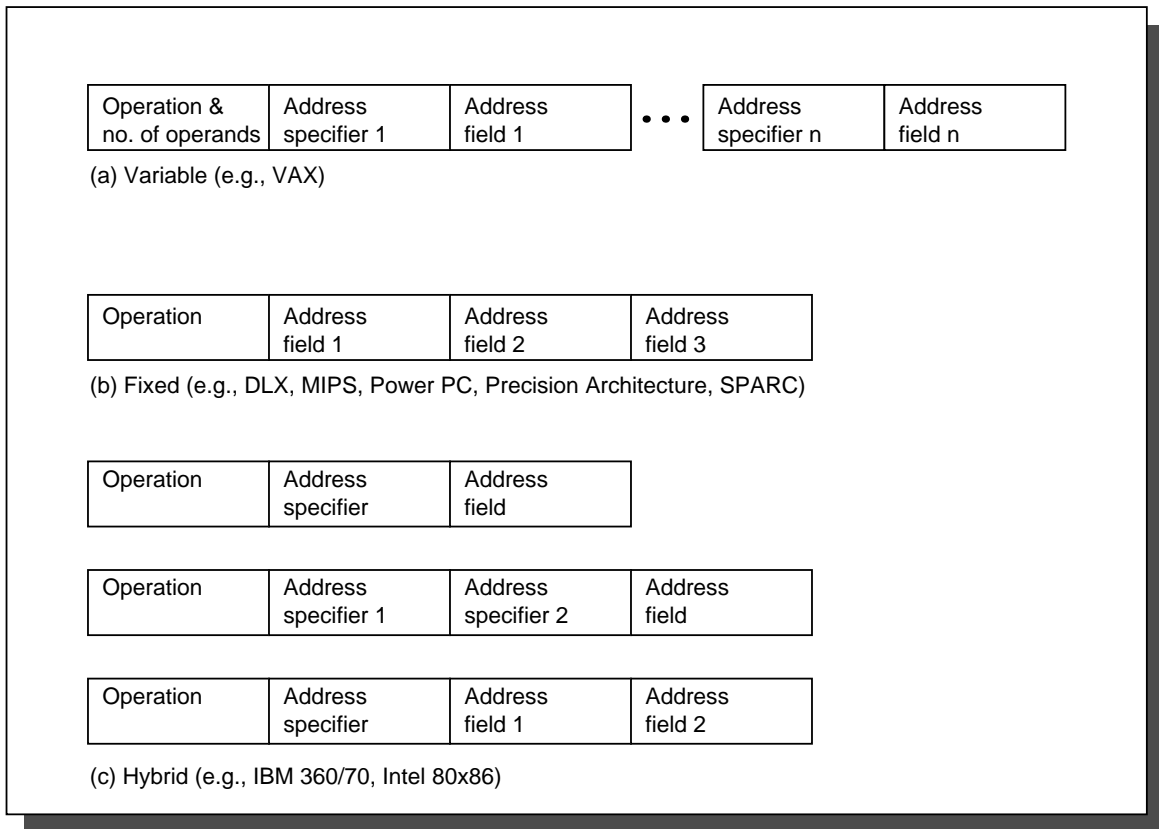
Traditionally the instruction set has the instruction partitioned into operation and operand address specifiers. Some flavors of partitioning include:

- Fixed Length — All instructions have the same layout. (May reduce CPI but also may need more instructions to express a program).
- Variable Length — The operator includes a specifier for how many operands follow in the header. (May require more CPI but may need fewer instructions per program).
- Hybrid — Permits a few instruction encodings, which are determined by the operation specified.

There is a tradeoff in memory utilization and efficiency in decoding the instructions.

# 17 – Instruction Set Encoding Strategies

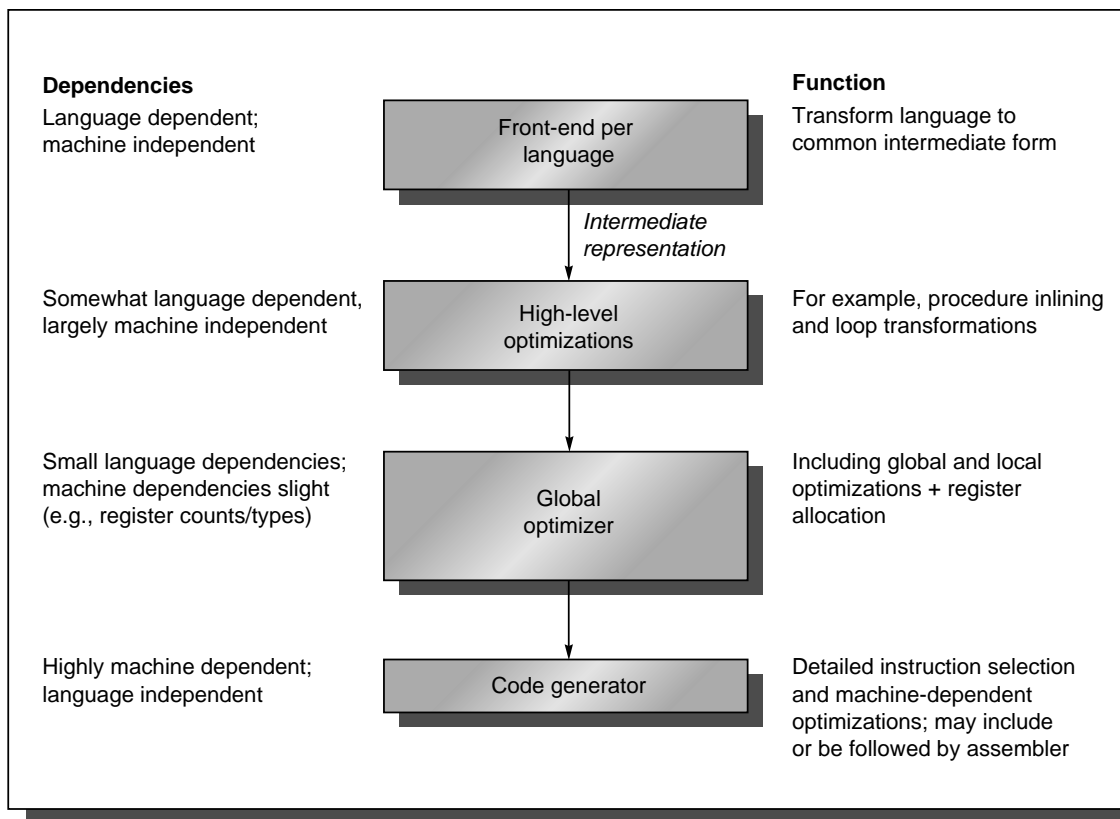
For example, consider:



**FIGURE 2.17 Three basic variations in instruction encoding.**

## 18 – Compiler/Architecture Interaction

Compilers and architectures interact rather closely.

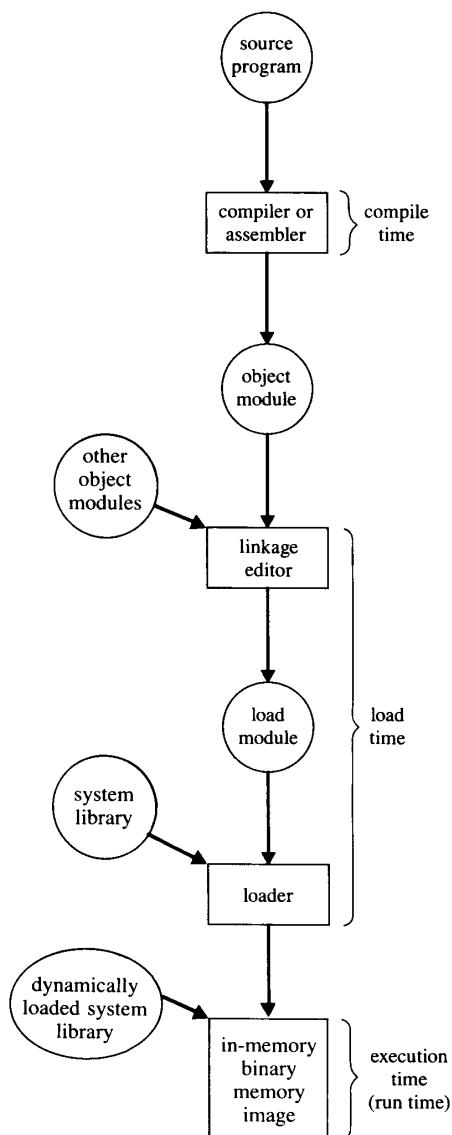


**FIGURE 2.18** Current compilers typically consist of two to four passes, with more highly optimizing compilers having more passes.

Many optimizations store frequently accessed values in registers, and benefit from having a large number of general purpose registers.

## 19 – Binding Time

Often an early binding time provides greater run time efficiency but reduces flexibility of the software.



## 20 – Memory Management in a C program

Traditional Unix/C memory images of programs partition memory into *segments*.

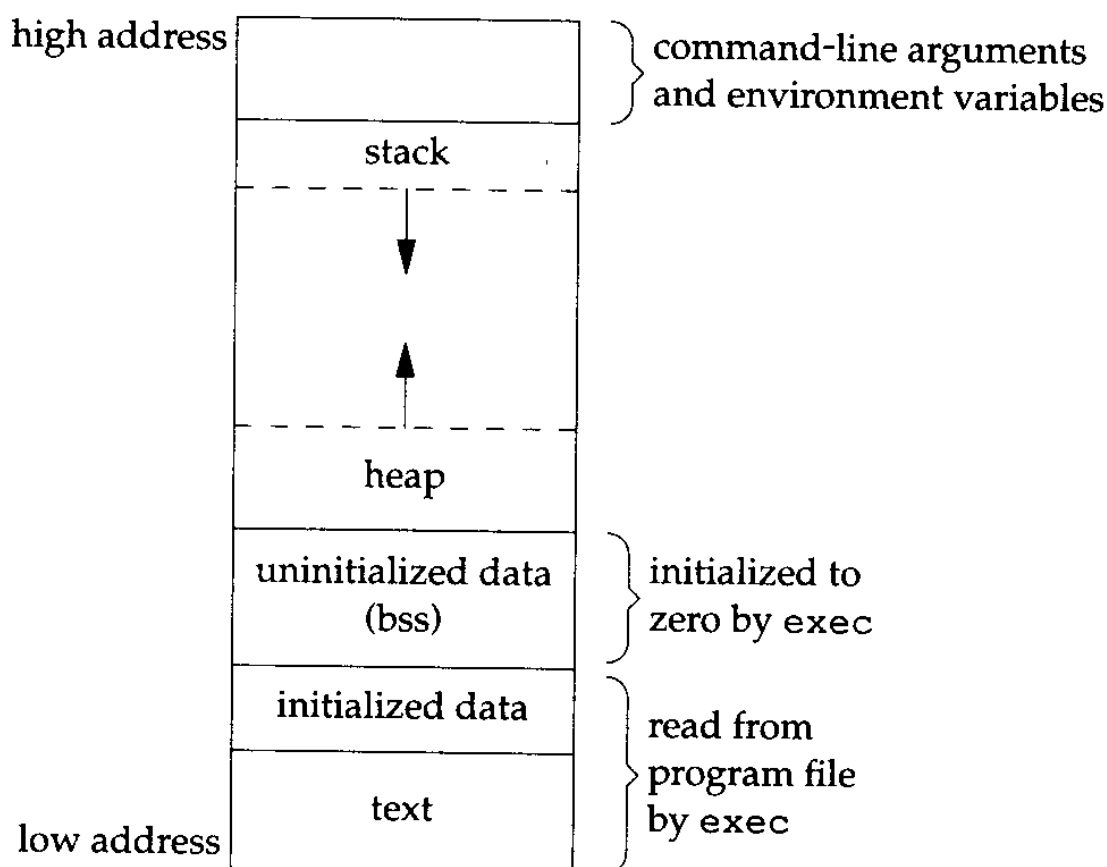


Figure 1: Memory Layout of A C Program

## 21 – How to Defeat Optimization

An optimizer should never break a working program. Optimizers have limits, pointers especially defeat optimization:

1. Using pointers to functions prevents inlining function calls.
2. Aliasing prevents register allocation for variables, e.g.

```
int a, b
    *ptr = & a,;

a = 5;
b = a + *ptr;
*ptr = 10;
```

3. Pointer arithmetic defeats alias detection schemes.
4. Multitasking limits register allocation of shared data.
5. Global variables limits register allocation.

6. Separate Compilation (prevents inlining for example).

## 22 – Architectural Help for Compilers

*Systems designers should make the frequent cases fast and rare cases correct.*

1. Regularity — Instruction sets consist of:
  - (a) Operations
  - (b) Data Types
  - (c) Addressing modeswhich should be orthogonal (i.e. treated independently).
2. Provide primitives, not solutions — special features add hardware complexity, and may be infrequently used.
3. Simplify Tradeoffs — Let the user know the costs of instructions.
4. Provide instructions supporting compile time binding — Early binding time provides speed.

## 23 – Counterexamples to good architecture

1. Irregular Architectures — Intel 8086, as John Walker [1] wrote:

It's become clear that the plague called the 8086 ... is not going to go away.

... I have never encountered a machine so hard to understand, one where the most basic decisions in designing a program are made so unnecessarily difficult ...

2. Too many language features in the architecture — Symbolics made LISP machines. They were overtaken by LISP environments which ran faster on normal hardware.
3. Simplify Tradeoffs — Kendall Square Research provided a machine in which the programmer had very limited control over memory management. This prevented optimizations.

4. Provide instructions supporting compile time binding — The VAX had a `calls` instruction saving for a function call. However the mask was immediate, which prevented its use in separate compilation.

## 24 – DLX — A Simple Example

DLX is a simplified version of the MIPS RX000 instruction set (used in SGIs).

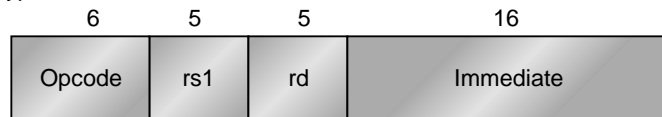
DLX features include:

1. General Purpose Registers Load/store architecture.
2. Support for common addressing modes: displacement (12-16 bit), immediate, register deferred.
3. Simple Instruction Set.
4. Support simple data sizes — 8,16,32 bit integer and 32, 64 bit IEEE floating point.
5. Fixed length instruction encoding.
6. Provides 32 GPRS (R0, R1, ..., R31) and 32 separate 32 bit FPRs (odd/even pairs can be used for 64 bit FP). R0 is always 0.
7. Suitable for pipelining and compiler optimization.

## 25 – DLX — Instruction Format

DLX's 2 addressing modes are encoded in the opcode, all instructions are 32 bit.

I - type instruction



Encodes: Loads and stores of bytes, words, half words  
All immediates ( $rd \leftarrow rs1 \text{ op immediate}$ )

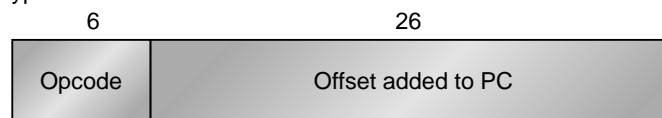
Conditional branch instructions (rs1 is register, rd unused)  
Jump register, jump and link register  
( $rd = 0$ , rs = destination, immediate = 0)

R - type instruction



Register-register ALU operations:  $rd \leftarrow rs1 \text{ func } rs2$   
Function encodes the data path operation: Add, Sub, ...  
Read/write special registers and moves

J - type instruction



Jump and jump and link  
Trap and return from exception

Figure 2.21— Hennessy/Patterson  
*Computer Architecture*  
Morgan Kaughmann Pub.  
100% — Illustrious, Inc. — 6/26/95-dc.jp

## References

- [1] J. Walker. *The Autodesk File: Bits of History, Words of Experience*. 1994. Available online on line at.