1 Program Structure and Execution

1.1 Information Storage

Computers store information as a series of bits. These bits can be interpreted by users in either source binary, comfortable decimal, or compatible hexadecimal.

Computers also have a default word size, i.e. the largest continuous block of memory the computer can access. Currently, most computers are 32 bit, however more are becoming 64.

Going with word size, each data type also has a typical size in memory:

C Declaration	32 Bit	64 Bit
char	1	1
short int	2	2
int	4	4
long int	4	8
long long int	8	8
char*	4	8
float	4	4
double	8	8

Table 1: Size of C Data Types

Besides the bits themselves, the order also matters, which brings up the distinction between little-endian and big-endian[1]. Big Endian has the highest place values put in the lowest memory location, while Little Endian has the lowest place values put in the lowest memory location.

1.2 Integer Arithmetic

Depending on the type of numbers involved in addition, we can get strange or unexpected behavior.

If we're dealing with large numbers, ones that are near to the word size in length, we have to be concerned about overflow. Overflow occurs when the full integer result cannot fit within the word size limits of the given data type.

Unsigned integer addition results in a something that resembles modular addition. If we have signed integers, we need to now concern ourselves with the negative numbers as well. Negative overflow often results in a positive number due to the definition of two's complement.[2]

[1] Name arises from Gulliver's Travels

[2] Computers express negative numbers by essentially inverting the bits. Normal binary adds each place, this version subtracts each place from the highest set bit. For a 4 bit word size, the number 1011 would actually be -5 instead of 11.

1.3 Floating Point

The first method of expressing floating point numbers for a computer was through fractional binary numbers. These numbers had the form:

$$\overbrace{b_m}^{2^m}\overbrace{b_{m-1}}^{2^{m-1}}\cdots\overbrace{b_1}^{2}\overbrace{b_0}^{1}.\underbrace{b_{-1}}_{1/2}\underbrace{b_{-2}}_{1/4}\cdots\underbrace{b_{-n-1}}_{1/2^{n-1}}\underbrace{b_{-n}}_{1/2^n}$$

The inherent issue with this method, is that it's not very good at dealing with larger numbers. This is where IEEE floating point standard comes in, which has the form:

$$V = (-1)^s \times M \times 2^E$$

Where:

- s determines sign
- M is a fractional binary number ranging from 1 and $2 - \epsilon$ or between 0 and $1 - \epsilon$
- E weighs the number by a power of 2.
- Final format looks as such:

S	E	M

With these floating point numbers, we have three cases to deal with.

1. **Normalized Values** are the most common. This occurs when E is neither all ones nor all zeros.
2. **Denormalized Values** occur when the exponent field is all zeros. These values express zero, as well as numbers that are very close to zero in absolute value.
3. **Special Values** occur when the exponent field is all ones. This either indicates $\pm\infty$ or NaN when the fractional value is non-zero.

2 Machine Level Representation of Programs

Most computers primarily use assembly for a more human-readable machine code. This code is much more explicit than the C that it was derived from.

Several fields that are visible in assembly that we lacked access to before:[3]

- The program counter, referred to as PC, and called %eip, refers to the address in memory of the next instruction to be executed.
- The integer register file contains eight named locations storing 32 bit values.
- Condition code registers
- Floating point registers

[3] For sake of simplicity, all memory addresses are for 32 bit platforms only.

There are a number of different commands that assembly uses in order to provide all the functionality available in C code. These are summarized in the table below.

We have to delve slightly deeper in the jump command however. The jump command depends on operational flags, and allows for sophisticated code.

2.1 Procedures

A procedure call involves taking data from one part of the program to another. This is managed by the program stack.

The program stack has two registers assigned to it, the stack pointer %esp and the frame pointer %ebp.

By convention, certain registers are kept private. Usually the registers %eax, %edx, and %ecx are classified as caller-save registers, that is the registers that are written to by the function caller. On the other hand, the registers %ebx, %esi, and %edi are classified as callee-save registers. That is the registers needed by the called function.

2.2 Arrays

Arrays are a common tool, and are fairly simplistic in nature. These work in principle by allocating the required memory for the specified data type, and referencing to it in order.

Structures are treated (to an extent) the exact same as arrays. When a structure is created, the required memory for the data types fills contiguous space in memory.

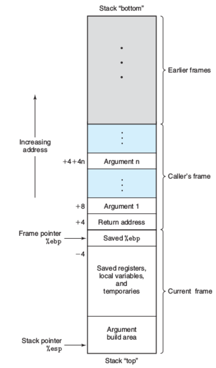


Figure 1: IA32 Stack Structure

3 Processor Architecture

3.1 Y86 Instruction Set

In this instruction set some commands are split up into several.

3.2 Sequential Implementations vs. Pipelining

In a sequential implementation, all cycles occur one after another. No new operation can start until the old one has finished.

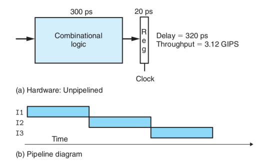


Figure 2: Sequential Implementation

Conversely in a pipelined implementation, we split up the cycles into stage and begin operations before old ones have finished.

4 Optimization

Optimization is an art.

We can use the metric *cylces per element* (CPE) to express how effective a program is.

First step in code optimization is to reduce the number of bottlenecks. This is referred to as code motion.

After that's complete, we remove unnecessary memory referencing.

Follower by loop unrolling, the strategy that involves taking loops with a concrete number of steps and turning them into a set of similar commands.

Parallelization can be utilized in certain cases.

Command	Generic Syntax	Example	Description
$M[Imm + R[E_b] + R[E_i] \cdot s]$	$Imm(E_b, E_i, s)$	`0xFC(%eax,%edx,4)`	Reference memory location based o
`mov`	`mov src,dst`	`mov (%esp),%edx`	Copy the data from the source locat
`push`	`push item`	`push $0xFF`	Push an item onto the stack.
`pop`	`pop item`	`pop $0xFF`	Pop an item from the stack.
`leal`	`leal src,dst`	`leal 6(%eax),%edx`	Load Effective Address takes whate into `dst`.
`inc`	`inc dst`		$dst = dst + 1$
`dec`	`dec dst`		$dst = dst - 1$
`neg`	`neg dst`		$-dst$
`not`	`not dst`		$\sim dst$
`add`	`add src,dst`		$dst = dst + src$
`sub`	`sub src,dst`		$dst = dst - src$
`imul`	`imul src,dst`		$dst = dst * src$
`xor`	`xor src,dst`		$dst = dst \wedge src$
`or`	`or src,dst`		$dst = dst \| src$
`and`	`and src,dst`		$dst = dst \& src$
`sal`	`sal k,dst`		$dst = dst << k$
`shl`	`shl k,dst`		$dst = dst << k$
`sar`	`sar k,dst`		$dst = dst >>_A k$
`shr`	`shr k,dst`		$dst = dst >>_L k$
`cmp`	`cmp src2,src1`	`cmp %eax,%edx`	Sets flags on `src1 - src2`.
`test`	`test src2,src1`	`test %eax,%edx`	Sets flags on `src1 & src2`.
`jmp`	`jmp dst`		Direct Jump
`je/jne`	`jmp dst`		Jump equal/not equal
`js/jns`	`jmp dst`		Jump negative/not negative
`jg/jge/jl/jle`	`jmp dst`		Jump greater/less
`ja/jae/jb/jbe`	`jmp dst`		Jump above/below
`call`	`call label`	`call 8049908`	Call procedure for execution
`leave`	`leave`		Prepare stack for return. This sets th of the frame and restores the saved
`ret`	`ret`		Return from call

Table 2: Assembly Reference

halt	stop the program
nop	no operation
rrmovl	register → register
irmovl	immediate → register
rmmovl	register → memory
mrmovl	memory → register
OP1	integer operation
jxx	jumps
call	call function
ret	Return
pushl	push onto stack
popl	pop from stack

Table 3: Y86 Instruction Set

5 Memory Hierarchy

5.1 RAM

RAM comes in two forms, static and dynamic. Static RAM is much more expensive, but way faster.

5.1.1 SRAM

Each bit is stored in a bistable memory cell. It can only be in one position or another, never both, or neither. It will also keep its state indefinitely as long as it's kept powered.

5.1.2 DRAM

Each bit is stored as a charge on a capacitor. They lose power relatively quickly, and are much cheaper.

5.2 Disk Storage

Disks have several platters that spin at a fixed rate. The capacity of a disk is determined by

$$Recording\,Density \times Track\,Density = Areal\,Density$$

or

$$Capacity = \frac{bytes}{sector} \times \frac{averagesectors}{track} \times \frac{track}{surface} \times \frac{surfaces}{platter} \times \frac{platters}{disk}$$

There is an actuator arm responsible for reading and writing from and to the disk.

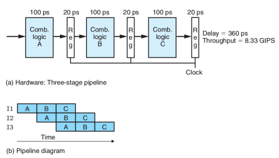


Figure 3: Sequential Implementation

5.3 Locality

There are two types of locality: Spatial and Temporal. Spatial locality refers to the fact that when memory is accessed, the memory around it is likely to be accessed in the near future. Temporal locality on the other hand refers to the notion that when memory is accessed, it's likely to be accessed again.

5.4 Memory Hierarchy

Each layer down in the memory hierarchy you go down, you slow the speed but decrease the cost. Therefore it's essential to introduce the concept of caching.

Caching involves storing smaller chunks of memory in the fastest spots so that you can access them only when you need them. A cache hit occurs when the memory needed is in the cache. The memory is then used directly from there. Conversely, a cache miss occurs when the memory is not cached.

Whenever there is a miss, the cache must use its replacement policy in order to determine which block to evict in order to make room for the new block.

5.4.1 Cache Capacity

The capacity of a cache can be expressed using the tuple (S, E, B, m), where each memory address has m bits, forming $M = 2^m$ unique addresses, $S = 2^s$ cache sets, E cache lines, and $B = 2^b$ data blocks. Caches also need $t = m - (b + s)$ tag bits that uniquely identify the cache line.

Therefore capacity can be expressed as $C = S \times E \times B$

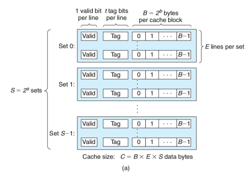


Figure 4: Sequential Implementation

5.4.2 Direct Mapped Caches

A cache with one line per set is a direct-mapped cache. When we have a cache miss, three steps occur.

Step One: Set Selection. The correct set is located using s.

Step Two: Line Matching. Determine if any line already contains the data. If one does, we have a cache hit.

Step Three: Line Replacement. If we have a cache miss, we need to replace a line with the new data.

5.4.3 Set Associative Caches

Conflict misses arise when we have direct mapped caches, where there is one line per set. We can relax this constraint a little, and establish each set to have at least 2 lines. A set with $1 < E < C/B$ is called an E way set associative cache. This is a much more sophisticated caching method, and we need to search each line separately instead of each set like before. The replacement policies differ from machine to machine, and it can be tricky determining what replacement method is utilized.

5.4.4 Fully Associative Caches

A fully associative cache has $E = C/B$. Set selection is trivial as there is only one set, however indexing is tricky given that there are many similar tags. These caches are not effective at a large scale.

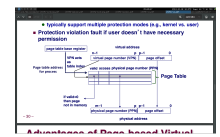
6 Linking

7 Virtual Memory

All Assembly code is executed using virtual memory. At no point is it aware of the actual hardware address.

The MMU is in charge of converting these physical addresses to virtual ones, and vice-versa. This can lead to problems however with fragmentation.

7.1 Page Tables



To help solve this, most machines use fixed length pages, usually 4 KB. The virtual page number is equal to the virtual address divided by the size of the page.

Page tables also map main memory to these fixed size pages as well. The page table records a virtual page → hardware page mapping.

A page table is an array of page tables entries (PTE) that maps virtual pages to physical ones.

The virtual address space, $\{0, 1, 2, \cdots, N-1\}$, with $N = 2^n$, is mapped to the physical address space, $\{0, 1, 2, 3, \cdots, M-1\}$.

Physical pages are $P = 2^p$ bytes in size and have been partitioned into 3 disjoint subsets. Unallocated have not been create by the VM system. Cached are currently in memory. Unchached are not.

There is a valid bit on each row that indicates whether or not the row is currently in memory.

Symbol	*Description*
$N = 2^n$	Number of addresses in virtual space
$M = 2^m$	Number of addressses in physical space
$P = 2^p$	Page size (bytes)
VPO	Virtual page offset (bytes)
VPN	Virtual page number
TLBI	TLB index
TLBT	TLB tag
PPO	Physical page offset
PPN	Physical page number
CO	Byte offset within cache block
CI	Cache index
CT	Cache tag