Machine-Level Programming I

Intel Processor History & Compilation

- **Intel x86:** Dominant CISC architecture, backward-compatible.
- **Milestones: 8086** (16-bit), **386** (32-bit IA32), **Pentium 4E** (64-bit x86-64), **Core 2** (multi-core).
- **Compilation:** C Code $\xrightarrow{\text{Compiler (gcc -S)}}$ Assembly $\xrightarrow{\text{Assembler (as)}}$ Object Code $\xrightarrow{\text{Linker (ld)}}$ Executable.

Assembly Basics

Table 1: Key x86-64 Registers

Register	Role	Saved By
%rax	Return Value	Caller
%rdi, %rsi, %rdx, %rcx	Arguments 1-4	Caller
%r8, %r9	Arguments 5-6	Caller
%r10, %r11	Temporaries	Caller
%rbx, %r12-%r15	Temporaries	Callee
%rsp, %rbp	Stack/Base Pointer	Callee

Addressing Modes

General Form: `D(Rb, Ri, S)` $\rightarrow$ `Mem[Rb + S*Ri + D]`

- **D:** Displacement (constant)
- **Rb:** Base register
- **Ri:** Index register
- **S:** Scale factor (1, 2, 4, or 8)

Examples:

```
// Immediate: Constant data
movq $0x4, %rax
// Register: Data in a register
movq %rax, %rdx
// Memory: Data at an address
movq (%rax), %rdx // Normal
movq 8(%rbp), %rdx // Displacement
```

Arithmetic & Logical Operations

Table 2: Common Instructions (q=8, l=4, w=2, b=1)

Instruction	Description
`leaq Src, Dst`	Load Effective Address (address calc)
`addq Src, Dst`	Add: Dst = Dst + Src
`subq Src, Dst`	Subtract: Dst = Dst - Src
`imulq Src, Dst`	Multiply: Dst = Dst * Src
`salq/shlq k, Dst`	Left Shift: Dst = Dst ¡¡ k
`sarq/shrq k, Dst`	Right Shift (Arith/Logical)
`incq Dst`	Increment: Dst++
`decq Dst`	Decrement: Dst–
`negq Dst`	Negate: Dst = -Dst
`notq Dst`	Bitwise NOT: Dst = Dst

Control, Procedures, Data

Control Flow & Comparisons

- **Condition Codes:** Single-bit flags (CF, ZF, SF, OF) set by arithmetic operations.
- **Compare and Test:**
  - `cmpq Src2, Src1`: Sets flags based on 'Src1 - Src2'.
  - `testq Src2, Src1`: Sets flags based on 'Src1 Src2'.

Table 3: Conditional Set and Jump Instructions

SetX/jX	Condition	Based On
e	Equal / Zero	ZF
ne	Not Equal / Not Zero	∼ZF
s / ns	Negative / Non-negative	SF / ∼SF
g / ge	Greater / ...or Equal (signed)	$\sim (SF \wedge OF) \& \sim ZF$
l / le	Less / ...or Equal (signed)	$SF \wedge OF$
a / b	Above / Below (unsigned)	$\sim CF \& \sim ZF$ / CF

Loop Implementation

Loops are converted to 'do-while' forms with conditional jumps. 'while' loops often use a "jump-to-middle" pattern.

Listing 1: While-loop translation

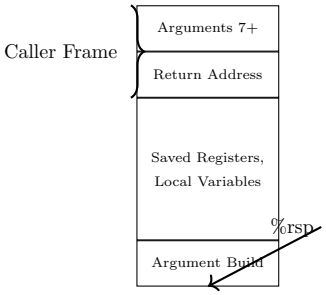
```
goto test;
loop:
    Body;
test:
    if (Test) goto loop;
```

Switch Statements & Jump Tables

For dense cases, 'switch' statements are implemented with a **jump table** (an array of code addresses) for efficient, direct branching to the correct code block via an indirect jump ('jmp *...').

Procedures (Functions)

- **Stack Frame:** Region of the stack for a call. Holds return info, local variables, and arguments.
- **Control Transfer:** 'call' pushes return address and jumps; 'ret' pops address and jumps back.



Data Structures

Arrays

A nested array 'int A[R][C]' is a single contiguous memory block. Access is 'Address(A) + (i*C + j) * sizeof(int)'. A multi-level array 'int* A[R]' is an array of pointers, requiring two memory accesses to get to an element.

Structures & Alignment

Structs group data but may include padding for alignment.

- **Alignment Rule**: Data of size K bytes must have an address that is a multiple of K.
- **Padding**: Compiler inserts unused bytes to align fields and ensure total struct size is a multiple of the largest field's alignment. To save space, declare larger fields first.

Listing 2: Alignment Padding

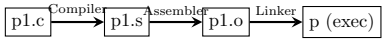
```
struct S { char c; int i; }; // sizeof is 8, not 5
// Layout: |c|pad|pad|pad|  i  |
```

Floating Point

Uses separate **XMM registers** (%xmm0-%xmm15) for arguments and return values. Instructions like 'addss' (single-precision) and 'addsd' (double-precision) perform FP arithmetic.

Advanced Systems Concepts

Linking & Compilation



- **Symbol Resolution:** Matching symbol references to definitions.
- **Relocation:** Updating addresses once final memory locations are known.

Memory and Buffers

- **Memory Layout:** Stack (grows down), Heap (grows up, 'malloc'), Data (globals), Text (code).
- **Buffer Overflow:** Writing beyond a buffer's boundaries, a major security vulnerability.
  - **Protections:** Stack Canaries, Address Space Layout Randomization (ASLR), and Non-Executable (NX) stacks help prevent attacks.
- **Byte Ordering (Endianness):**
  - **Little Endian (x86):** Least significant byte at lowest address.
  - **Big Endian:** Most significant byte at lowest address.

Compiler Optimization

- **Constant Folding:** Pre-calculating constant expressions.
- **Dead Code Elimination:** Removing unreachable code.
- **Common Subexpression Elimination:** Reusing results of identical calculations.
- **Code Motion:** Moving loop-invariant code out of loops.
- **Inlining:** Replacing a function call with its body.

ECF & Memory Hierarchy

Exceptional Control Flow (ECF)

ECF is a mechanism for reacting to system events by transferring control from normal program flow to the kernel.

Table 4: Types of Exceptions

Type	Cause	Return Behavior
Interrupt	External event (I/O)	Returns to next instruction
Trap	Intentional (syscall)	Returns to next instruction
Fault	Error (page fault)	Might re-execute or abort
Abort	Fatal error	Does not return

Processes & Signals

A **process** is an instance of a running program. A **signal** is a kernel message to a process about an event.

- `fork()`: Creates a near-identical child process. Returns child's PID to parent, 0 to child.
- `execve()`: Replaces the current process image with a new program.
- `waitpid()`: Parent reaps a terminated child process to prevent it from becoming a "zombie".
- **SIGCHLD:** Signal sent to parent when a child terminates.
- **SIGINT:** Sent by Ctrl-C.
- **SIGSEGV:** Segmentation fault (invalid memory access).

Memory Hierarchy

- **Locality:** The tendency for programs to access data and instructions near those they have recently accessed.
  - **Temporal:** Recently referenced items are likely to be referenced again soon.
  - **Spatial:** Items with nearby addresses tend to be referenced together in time.
- **Caching:** Using smaller, faster storage (e.g., SRAM) to stage data for larger, slower storage (e.g., DRAM). Locality makes caching effective.
- **Cache Misses:**
  - **Cold (Compulsory):** Cache is empty.
  - **Conflict:** Multiple data objects map to the same cache block, causing evictions.
  - **Capacity:** The working set of active blocks is larger than the cache.
- **Disk Access Time** = $T_{avg\_seek} + T_{avg\_rotation} + T_{avg\_transfer}$.

Virtual Memory (VM)

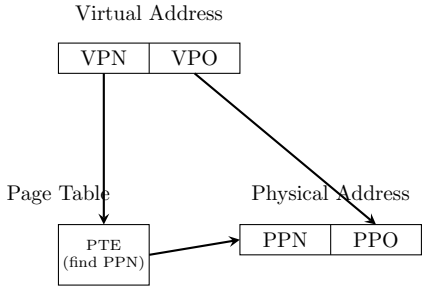
VM Concepts

- **Address Spaces:**
  - **Virtual (VA):** Uniform address space seen by a process.
  - **Physical (PA):** Actual addresses in hardware memory.
- **Pages & Page Tables:** VM is divided into fixed-size blocks (pages). A per-process **page table** maps virtual pages to physical pages.
- **Page Fault:** An exception when a referenced page is not in physical memory. The OS handles it by loading the page from disk (demand paging).
- **Benefits:** Efficient memory use (caching), simplified memory management, and memory protection via permission bits in Page Table Entries (PTEs).

Address Translation

The Memory Management Unit (MMU) translates VAs to PAs. A **Translation Lookaside Buffer (TLB)** is a small hardware cache of PTEs to speed this up.

- **VA parts:** Virtual Page Number (VPN) and Virtual Page Offset (VPO).
- **PA parts:** Physical Page Number (PPN) and Physical Page Offset (PPO).
- **Process:** VPN is used to find the PTE in the TLB or page table. The PPN from the PTE is combined with the original VPO (which is the same as the PPO) to form the physical address.



- **Memory Mapping ('mmap'):** Creates a new VM area and associates it with an object on disk (e.g., a file or anonymous memory).
- **Copy-on-Write (COW):** 'fork()' uses COW to defer physical memory copying until a write occurs, making process creation fast.