Lecture 03: Low-Level Software

August 26, 2024

Overview

- High-level software
- Low-level aspects
- High-to-low mapping
- Assembly languages
- Compilers and execution

High-Level Perspectives

Humans and computers “think” about software differently. High-level languages allow humans to reason about software more clearly.

- Program structure
  - Files, namespaces, modules, classes, etc.
- Data management
  - Data structures, memory allocation, scopes, etc.
- Basic control flow
  - blocks, loops, conditionals, etc.

It’s best to understand the software engineering process and tools.

High-Level Perspectives

Program Structure

Relative advantages and disadvantages

High-Level Perspectives

Program Structure

Relative advantages and disadvantages

- ⊕ Make large complex software manageable
- ⊕ Provides alternative to “spaghetti code”
- ⊕ Breaks program into “units”
  - Mental tracking, division of effort
- ⊖ Tends to be lost in compilation and linking
  - units combined
  - functions inlined
  - redundancies removed
- ⊖ Must be reconstructed, at least in part, by a reverser

High-Level Perspectives
Program Structure

“Encapsulation” an abstraction of a concept and its related functions into a usable unit of software, which hides the implementation details.

- Happens at various levels – “nesting of boxes”
- Each thoroughly tested
- Each having a well-defined interface

A reverser must have some notion of the program structure in order to navigate to its interesting parts.

High-Level Perspectives

Modules

Largest program unit other than the program itself

Static libraries: `.obj` `.o` `.a`

- ▶ a collection of source files built together
- ▶ typically represent a feature or area of functionality
- ▶ often third-party developed
- ⊖ difficult to isolate while reversing

High-Level Perspectives

Modules

Dynamic libraries: `.dll` `.so`

- like static, but not embedded

High-Level Perspectives

Modules

Dynamic libraries: `.dll` `.so`

- ▶ like static, but not embedded
- ⊕ allows upgrading components independently
- ⊖ requires consistent interface
- ⊖ “DLL hell”
- ⊕ easy to isolate while reversing (separate file)
- ⊕ exposes interface between components

High-Level Perspectives

Common Code Constructs

Procedures

- most fundamental unit of software
- a unit of code with a well-defined purpose
- can be invoked by other areas
- can take input and output: “function”
- most common form of encapsulation

High-Level Perspectives

Common Code Constructs

Objects

- object-oriented design – supplant procedure as unit of reason
- probably most popular design methodology
- contain both data and code
- code typically manipulates data within the object
- data typically hidden from external components
- “clients” interact with objects via an interface

inheritance allows a specific implementation to borrow generic components and interfaces

- facilitates polymorphism

High-Level Perspectives
Common Code Constructs

Aspects[1]

- permits organization of code into concerns
- some concerns “cross-cut” the class structure (e.g., logging)
- usually involves joining “advice” to other code
  “When you would ordinarily execute `f()`, also execute this”

Implementations

- Weave code at the source or bytecode level
- Or change interpreter to implement AOP

AspectJ extension to Java

Partial classes in C# (not quite)

[1]Wikipedia: Aspect Oriented Programming

High-Level Perspectives
Data Management

All programs (and their component operations):

- take input,
- use temporary storage, and
- produce output.

Like code, the high-level constructs are translated to lower-level, causing some information loss.

High-Level Perspectives

Data Management

Data Structures

High-Level Perspectives
Data Management

Data Structures

Variable a named storage location of a type

- declared with some scope, defining its access and storage
- name usually replaced by address

High-Level Perspectives

Data Management

Data Structures

Variable a named storage location of a type
- declared with some scope, defining its access and storage
- name usually replaced by address

User-defined Structure a group of related fields each of its own type
- usually stored and handled together
- reversing structures is critical to understanding a program
  record incremental details as you go

High-Level Perspectives

Data Structures

Collection	a group of fields all of the same type

- come in different flavors and implementations:
  - expected size
  - preserves order / sorted
  - requires uniqueness
  - frequency/location of searches, insertions, deletions

High-Level Perspectives

Collections

Arrays

- most basic - items placed sequentially
- referenced by index
- can be multidimensional

High-Level Perspectives

Collections

Arrays

- most basic - items placed sequentially
- referenced by index
- can be multidimensional

⊕ fastest access – direct support on x86

⊕ most intuitive

⊖ difficult to add and remove at the middle
especially inefficient for large lists

High-Level Perspectives

Collections

Linked Lists

- items placed individually anywhere
- each item contains a reference or “link” to the next
- may also have a link to the previous

High-Level Perspectives

Collections

Linked Lists

- items placed individually anywhere
- each item contains a reference or “link” to the next
- may also have a link to the previous

⊕ fast insertion and removal

⊖ access by index is slow

⊖ uses more memory for links

High-Level Perspectives

Collections

Trees

- many kinds (binary, AVL, RB, etc.)
- items placed indvidually anywhere
- items linked in hierarchy rather than in sequence

High-Level Perspectives

Collections

Trees

- many kinds (binary, AVL, RB, etc.)
- items placed indvidually anywhere
- items linked in hierarchy rather than in sequence

$\supset$ like linked list

⊖ uses more memory (at least two links)

⊕ fast searching (implicitly sorted)

⊖ does not preserve order

High-Level Perspectives
Control Flow

High-level	Conditional blocks and loops
Low-level	Jumps / conditional jumps

- Some processors support conditional operations (ARM)
- Some processors support repeated operations (x86)

Conditional blocks

- in C: `if`, `else if`, and `else`
- Conditional controls whether / which block executes

High-Level Perspectives

Control Flow

Switch blocks

- in C: `switch`, `case`, `default`, and `break`
- Takes an input and chooses block(s) to execute
- Sometimes a lookup table, sometimes converted to if/else

Loops

- in C: `for`, `do`, `while`, and `break`
- Repeated block
- Interrupted when a condition is met or unmet
- Usually includes a counter

High-Level Perspectives
Languages

- Abstract the machine details
- Assembly simply isn’t suitable for complex software
- Balance simplicity and flexibility
  - do much with little direction
  - preclude few operations

- Reverser is exposed to more details than developer
- Must seek a high-level understanding
  - Know the language(s) used
  - Be familiar with its abstractions

High-Level Perspectives
Languages

C

- Probably lowest of the high-level languages
- Arrays have no bounds checking
- Has data and control structures
- Is compiled
- Can be cross-platform at the source level
- Made for UNIX, but also used by Windows
- Can be very efficient – little overhead
- Also very easy to reverse

High-Level Perspectives

Languages

C++

- Extension to C for OOP
- Mostly a superset of C
- Introduces classes – structs with code components and inheritance
- Supports polymorphism via virtual methods
- Similar to reversing C – emphasis on determining:
    - class hierachy
    - virtual method calls
    - constructor calls, etc.

High-Level Perspectives

Languages

Java

- Compiled to bytecode rather than native machine code
  - Bytecodes like assembly, but interpreted
- Cross-platform at the bytecode level – platform must have a JVM
- Reversing process is very different
- Easy to decompile with very high accuracy
  - Binaries tend to be obfuscated as a countermeasure

High-Level Perspectives
Languages

C#

- Competitor to Java from Microsoft
- Developed to address shortcomings of C++
- Bytecodes are called “Microsoft Intermediate Language” or MSIL
- Cross-platform at the MSIL level – platform must have a .NET VM (Common Language Runtime)
  - The Microsoft runtime API was limited to Windows
- Reversing is similar to Java
  - Binaries often obfuscated

Low-Level Perspectives

Data Management

```
int Multiply(int x, int y) {
  int z;
  z = x * y;
  return z;
}
```

- save machine state
- allocate memory
- load parameters
- perform operations (storing intermediate and final values)
- restore machine state
- communicate z to the caller
- return control

Low-Level Perspectives

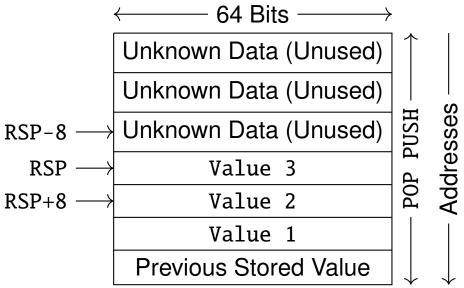
Data Management

Registers

- A CPU's closest working space
- To avoid constant RAM access
- Little or no penalty for access
- Usually fairly few
- Managing registers and RAM is not automatic
- Reverse must determine prupose of register in context

The Stack

- Secondary storage area for short-term information
- Special area in RAM
- Typically, stacks grow backward (into lower addresses)



```
PUSH Value 1
PUSH Value 2
PUSH Value 3
```

Low-Level Perspectives

Data Management

Stack Usage

Low-Level Perspectives

Data Management

Stack Usage

Temporarily-saved register values	if a function needs a (callee-save) register, it must save and restore its original value
Local variables	if a variable doesn't fit into registers or has operations requiring RAM storage
Function parameters	values passed as parameters are usually placed in registers or on the stack
Return addresses	the address following a call must be saved to resume later

Low-Level Perspectives

Data Management

Heaps

- Managed region of dynamically allocatable memory
- Program requests block of given size, heap returns pointer
- Typically implemented by library or OS
- Could be custom implementation
  - Reversers should locate heaps and management routines
  - Returned block can be traced
  - Size of block is known from parameter

Low-Level Perspectives

Data Management

Executable Data Sections

- Usually named `.data` or `.rodata`
- Used to store globals and pre-initialized data
- e.g., C string literals
- e.g., hard-coded addresses emitted by compiler

Low-Level Perspectives

Control Flow

- Much more friendly in high-level
- Conditional statements implemented using primitives
- Implementations depend on hardware
- Our discussions will use x86

Compilation

- Most software is compiled
- Compilers tend to generate non-intuitive code
- Primary job is code translation
  - Usually text source to machine binary
  - Often apply optimizations
  - Output not meant for human consumption

Compilation

Architecture

Front end

input: source code

output: intermediate representation

Intermediate Representations

- used to communicate program from stage to stage
- sensitive to the purpose of the stage
- may be closer to high or low level

Compilation
Architecture

Front end

input:	source code
lexical analysis	groups characters into tokens
syntax analysis	checks for correct grammatical structure
semantic analysis	derives meaning from structure
output:	intermediate representation

Intermediate Representations

- used to communicate program from stage to stage
- sensitive to the purpose of the stage
- may be closer to high or low level

Compilation
Architecture

Optimizer

- A huge reason to understand compilation
- Goal is to create most performant, smallest, etc. code

Compilation

Architecture

Optimizer

- A huge reason to understand compilation
- Goal is to create most performant, smallest, etc. code

Question: What other goals?

Compilation
Architecture

Optimizer

- A huge reason to understand compilation
- Goal is to create most performant, smallest, etc. code

Question: What other goals?

Generic	operate on IR
Target-specific	operate on machine code

Compilation
Optimizations

Code Structure

loop unrolling	produces larger code but reduces branching
switches	implemented as table or tree
post-test loops	save a conditional branch

Redundancy Elimination

- eliminating human errors
- eliminating repeated computations (or pulling outside of a loop)
- eliminating overwritten and/or unused values
- caching pointers (e.g., in array indexing)

Compilation

Architecture

Back end (“code generator”)

- converts IR to target-specific code
- May perform platform-specific optimizations
  (where most annoying transformations take place)

Instruction selection translate IR to machine

Register allocation decide which vars go in hardware registers

Instruction scheduling order to maximize parallelism

Compilation

Architecture

Back end ("code generator")

- converts IR to target-specific code
- May perform platform-specific optimizations
  (where most annoying transformations take place)

Instruction selection	translate IR to machine
Register allocation	decide which vars go in hardware registers
Instruction scheduling	order to maximize parallelism

Question: Why separate front and back ends?

Execution

Execution environment The machine that executes a program, its API, libraries, etc.

Execution

Software Execution Environments (VMs)

- execute an IR program, e.g., Java bytecodes
- each VM is implemented for a specific machine
- the IR is common for all implementations of the VM
- e.g., Java (JVM), and .NET (CLR)

Execution
Software Execution Environments (VMs)

- execute an IR program, e.g., Java bytecodes
- each VM is implemented for a specific machine
- the IR is common for all implementations of the VM
- e.g., Java (JVM), and .NET (CLR)

⊕ platform isolation

⊕ enhanced functionality

- garbage collection
- runtime type safety
- checked memory access

⊖ decreased performance

⊖ limited access to low-level APIs

⊖ limited access to OS features

Execution

Software Execution Environments (VMs)

Interpreters

- original approach to VMs
- everything is decoded and executed by a progam
- “registers” mapped to memory in interpreter
- performance is very slow
  - every instruction separately decoded and executed
  - requires dozens of machine instructions per bytecode instruction

Execution

Software Execution Environments (VMs)

Just-in-Time (JIT) Compilers

- a compiler back end at run time
- blocks of bytecode translated to machine code
- instructions fed to native CPU

Execution

Software Execution Environments (VMs)

Just-in-Time (JIT) Compilers

- a compiler back end at run time
- blocks of bytecode translated to machine code
- instructions fed to native CPU

Question: Why not translate whole program at load?

Execution

Software Execution Environments (VMs)

Just-in-Time (JIT) Compilers

- a compiler back end at run time
- blocks of bytecode translated to machine code
- instructions fed to native CPU

Question: Why not translate whole program at load?

- Run-time link/loading of modules
- Translation of polymorphic calls

Execution

Software Execution Environments (VMs)

Reversing Strategies

Execution

Software Execution Environments (VMs)

Reversing Strategies

- take advantage of metadata (class names, method parameters, variable types, etc.
- use a decompiler
  - ⊕ very easy and effective
  - ⊖ motivate equally effective obfuscation techniques
- bytecodes still reveal higher-level info

Execution

Hardware Execution Environments

processor	digital circuits controlled by machine code
simple loop	decode instruction, activate circuits
parellelism	physical limitations, desire for speed
backward compatibility	old programs on new hardware
data dependencies	limit parallelism
multiple scheduling	execute when input data finishes

Execution

Intel Haswell[2]

- latest (circa 2013) Intel microarchitecture used in Core i7
- Micro-operation cache of 1.5K operations
- 14- to 19-stage instruction pipeline
- 4 ALUs per core, 3 AGUs, 2 branch predictors
- HyperThreading
- integrated GPU
- typically 4+ cores
- dynamically partitioned decode cache in HyperThreading
- support for NUMA (non-uniform memory access)

[2] http://en.wikipedia.org/wiki/Haswell_%28microarchitecture%29

Execution

μ-ops

- x86 instructions translated to μ-instructions executed internally
- core fairly primitive, most x86 logic in decoder
- core significantly improved based on now-defuct Itanium
- microcodes loaded from ROM (cached for performance)
- microcodes can be patched

Execution

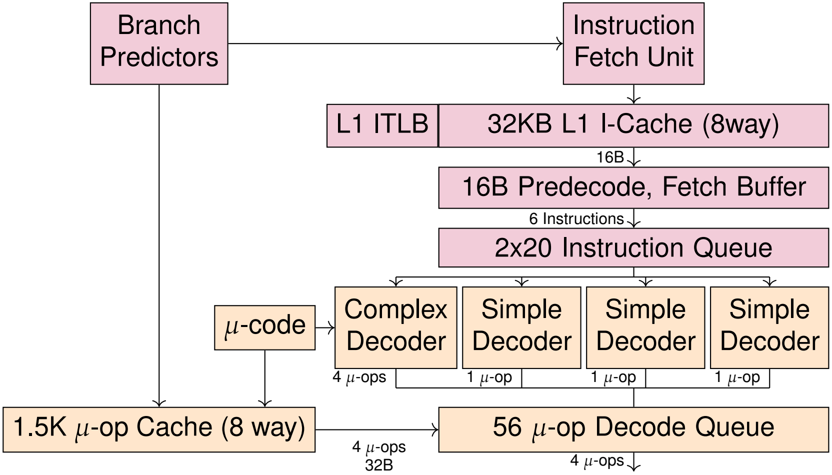
Pipelines[3]

Instructions broken into stages within a pipeline

- **Front end** decodes each instruction in μ-ops
- **Out-of-order core** executes instructions when resources are available
  - can dispatch up to 8 μ-ops per cycle
  - has 8 execution ports
  - can use pipeline forwarding
  - can re-order instructions preserving data dependencies
- **Retirement section** ensure proper ordering
  - 4 μ-ops per cycle per core

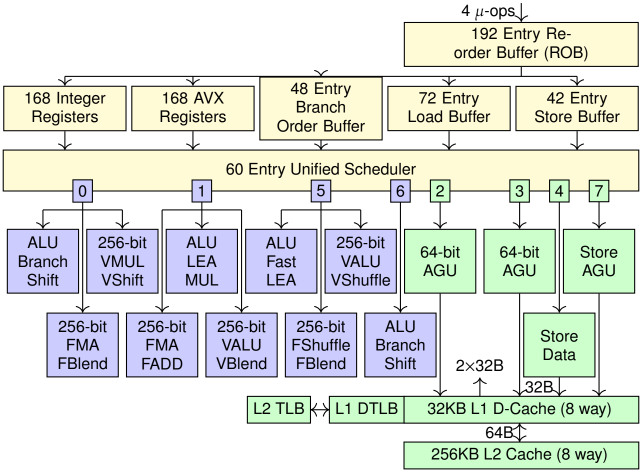
[3] http://www.realworldtech.com/haswell-cpu/2/

Execution



Branch Predictors
Instruction Fetch Unit
L1 ITLB
32KB L1 I-Cache (8way)
16B
16B Predecode, Fetch Buffer
6 Instructions
2x20 Instruction Queue
μ-code
Complex Decoder
Simple Decoder
Simple Decoder
Simple Decoder
4 μ-ops
1 μ-op
1 μ-op
1 μ-op
1.5K μ-op Cache (8 way)
4 μ-ops
32B
56 μ-op Decode Queue
4 μ-ops

Execution


4 μ-ops
192 Entry Re-order Buffer (ROB)
168 Integer Registers
168 AVX Registers
48 Entry Branch Order Buffer
72 Entry Load Buffer
42 Entry Store Buffer
60 Entry Unified Scheduler
0
1
5
6
2
3
4
7
ALU Branch Shift
256-bit VMUL VShift
ALU LEA MUL
ALU Fast LEA
256-bit VALU VShuffle
64-bit AGU
64-bit AGU
Store AGU
256-bit FMA FBlend
256-bit FMA FADD
256-bit VALU VBlend
256-bit FShuffle FBlend
ALU Branch Shift
Store Data
2×32B
32B
L2 TLB
L1 DTLB
32KB L1 D-Cache (8 way)
64B
256KB L2 Cache (8 way)

Execution

Branch Prediction

Knowing which instructions to load after conditional branches

Two approaches

- Stall pipeline until result of branch can be known
- Attempt to predict and take appropriate path
  - an incorrect prediction can be very wasteful

heuristic	e.g., forward vs. backward (loops)
tracing	record previous results

Features affect how optimizers may behave