The Software Stack:

From Assembly Language to Machine Code

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.
Students enrolled in Comp 506 at Rice University have explicit permission to make copies of these materials for their personal use.
Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.
The Hardware Execution Model

Computers execute individual operations, encoded into binary form

- Basic cycle of execution is: *(fetch, decode, execute)*
  - **Fetch** brings an operation from memory into the processor’s decode unit
  - **Decode** breaks the operation into its fields, based on the opcode, & sets up the operation’s execution — writes values to appropriate control registers
  - **Execute** clocks the processor’s functional unit to carry out the computation

- Each *operation* has a fixed format
  - A *processor* will support several formats
  - *Opcode* determines format

<table>
<thead>
<tr>
<th>opcode</th>
<th>reg₁</th>
<th>reg₂</th>
<th>reg₂ or constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10</td>
<td>15</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>31</td>
</tr>
</tbody>
</table>

People are not particularly good at reading & writing binary operations

- Productivity & error rate are better with higher level of abstraction
- People need a tool to translate some symbolic representation to binary
  ⇒ We invented *assemblers* to perform that translation

Bits in the instruction format are a serious architectural limitation
Symbolic Assembly Code

The ILOC that we have seen in examples is a symbolic assembly code for a simplified fictional RISC processor

- Symbolic names for operations
  - load, store, add, mult, jump, ...
- Labels for addresses
  - @a, @b, @c, ...
- Register names specified with integers
  - r₁, r₂, r₃, ...

Of course, the processor would not recognize all these symbolic names. Real processors run from code expressed in binary form.

Symbolic assembly code must be translated before it can execute.
Symbolic Assembly Code

The ILOC that we have seen in examples is a symbolic assembly code for a simplified fictional RISC processor

• Symbolic names for operations
  ♦ load, store, add, mult, jump, ...

• Labels for addresses
  ♦ @a, @b, @c, ...

• Register names specified with integers
  ♦ r₁, r₂, r₃, ...

ILOC has symbolic labels and virtual registers

Machine code needs addresses and physical registers

• Compiler maps virtual registers to physical registers
  ♦ Job of the compiler’s register allocator

• Software stack (assembler, linker, loader) converts labels to virtual addresses

ILOC Code:
load @b ⇒ r₁
load @c ⇒ r₂
mult r₁,r₂ ⇒ r₃
load @d ⇒ r₄
add r₃,r₄ ⇒ r₅
store r₅ ⇒ @a
load @f ⇒ r₆
add r₅,r₆ ⇒ r₇
store r₇ ⇒ @e
Symbolic Assembly Code

In assembly code, the fields in an operation are alphanumeric symbols

- Opcodes are represented with mnemonics — character strings
  - “add” instead of “0110”
- Registers and constants are written using base 10 or base 16 numbers
  - “r15” instead of “01111”

To prepare an assembly program for execution, it must be translated

- Mnemonics map directly to opcodes – bit patterns
- Symbolic labels map directly to addresses – bit patterns
- Base 10 numbers convert to base 2 numbers – bit patterns
- For the most part, these translations are simple and direct
Building an Assembler

What does an object file contain?

• **A list of operations**
  ♦ First operation (almost always) has an exported label
  ♦ Complete operations are in final binary form
  ♦ Operations with a symbolic reference have a hole & an index into a symbol table

• **A symbol table**
  ♦ Each symbol defined in the module has a symbol name, a type, and an offset from the start of the module
  ♦ Each imported symbol has a symbol name, a storage class, and a list of operations in the module that reference this symbol

Storage class might be “static” or “dynamic”, “local” or “global”
Building an Assembler

Assemblers convert assembly code to object code

• Typical assemblers operate in two passes
• Pass 1 builds up a symbol table that maps names to addresses
  ♦ Use a hash table to record the map
  ♦ Linear pass over the input to find symbols & record <name, address> pairs
• Pass 2 rewrites the operations into binary form
  ♦ All symbols should be defined after pass 1
  ♦ Linear pass over the input to rewrite operations into binary form
    → Symbolic references marked with their name and table entry

At the end, the assembler writes out an object file
Building an Assembler

The algorithm for Pass 1 is simple

• Initialize a counter to zero
• Initialize an empty map (a hash table ¹)
• At each operation, from the first to the last
  ♦ If the operation is labeled, enter the label in the map with the current counter
  ♦ Enter any undefined labels used as operands in the map, with an invalid counter
  ♦ Compute the length of the current operation
  ♦ Increment the counter by the operation’s length
• Iterate over the map
  ♦ If any symbol has an invalid counter, mark it as an external symbol or signal error
    → Choice of action depends on the rules of the particular assembly language

Pass 1 should operate in $O(|\text{operations}|)$ time  
   (if hash lookup is $O(1)$)
   → Number of symbols is $O(|\text{operations}|)$

¹ See discussion of hash tables in Chapter 6 and Appendix B in EaC2e
Building an Assembler

Pass 2 iterates over the input again and rewrites it
• Pass 1 resolved all the symbols
• Pass 2 constructs the object code

For each operation
• Mnemonic becomes a bit pattern for the opcode
• Opcode determines instruction format
• Operands become bit fields
  ♦ The exception is a reference to an externally defined symbol
  ♦ Replace externally defined symbol with a reference into table of such symbols
  ♦ Format for that reference and table is a system-wide convention
• Write out the finished binary operation

At the end, write out the symbol table information

Again, it is all $O(|\text{operations}|)$ \hspace{1cm} (if hash lookup is $O(1)$)
An assembler can do most of its translation in the first pass

- Can translate most operations directly in pass 1
  - E.g., `add r1, r2 => r3` contains all the information that it needs
  - Exception arises from a reference to a symbol that is not yet defined

- When the assembler finds an (as yet) undefined label, it can:
  - Enter that symbol into the symbol table and mark it as undefined
  - Add the current operation to a list of operations containing symbolic references
  - It can process any operation with an (already) defined label

- At the end of pass 1, the assembler can:
  - Traverse the list of unfinished instructions
  - Translate them in place

- After that “half pass”, it can output the object code
Assembler Pseudo-Operations

Almost all assemblers provide pseudo-operations to manipulate layout

- Pseudo-operations define storage, define labels, initialize space, and mark symbols as external
- To create space for a global array \texttt{A}, define storage for it
  - Label that pseudo-operation with a known label, say \_@A
- The pseudo-operations serves two functions
  - It advances the assembler’s internal counter for addresses (by its argument)
  - It provides a place to “hang” the label, at the start of that block of space

If we define \_@A as an external symbol, other code will be able to access it

- The linker will tie the symbols and addresses together
- The linker will match them by name, so only one object file can define \_@A

The process of creating unique labels from names is called “mangling”

- Procedure \texttt{fee()} might create something like \_.fee_
- Static \texttt{fee()} in file \texttt{foe()} might create something like \_.foe.fee_

\texttt{@A: dss 1024}
\texttt{@B: bss 128}
Lab 2

Generating ILOC from your parser for demo

• In Lab 2, you will extend your lab 1 code to generate ILOC
• You will have an ILOC simulator to test and debug the output
• You have the opportunity to apply all of the knowledge gained in the first half of COMP 506

The Big Question

• Do you want the lab now, before break, or Monday after break?
So, What Happens With All Those Object Modules?

The compiler produces a .asm file for each “compilation unit.”
The assembler produces a .o file for each .asm file.

What happens next?

• A collection of .o files can be combined to form an executable file (a.out)
• The program that performs this task is called a linker
• The linker takes a collection of object files and libraries of object files
• It selects out the pieces that it needs to build a complete executable
• It lays out the executable, computes addresses for all external symbols, and
  rewrites the executable, replacing the external symbols with virtual
  addresses
Building a Linker

A linker composes one or more object modules into an executable program

• Finds all of the entry points and labels in the object code
  ◦ Both exported and imported names
• Finds the main entry point
• Determines the spatial layout of the executable
• Resolves all imported names in an appropriate way
  ◦ Names that are statically linked
    → Imported names must match an exported name
    → Imported names must be rewritten with the exported name’s address
  ◦ Names that are dynamically linked
    → Imported names must be rewritten with a stub to load & link the name at runtime
Building a Linker

How does a linker work?

- **Pass 1:** Build a map of all symbols (exported & imported) by the object modules and libraries, and find the the **main** entry point

- **Pass 2:** Add object modules until all static symbols are resolved
  
  Starting with the module containing **main**:
  
  - Add the module to the end of the code
  - Assign addresses to its internal labels

- **Pass 3:** Rewrite the code with resolved symbols
  
  - Static symbols are rewritten with addresses
  - Dynamic symbols are rewritten with a jump to a stub that finds, loads, and links the needed object code

Granularity of inclusion is an object file

Library can be one big file or a collection of smaller modules
How does a linker work?

- **Pass 1:** Build a map of all symbols (exported & imported) by the object modules and libraries, and find the main entry point

- **Pass 2:** Add object modules until all static symbols are resolved
  - Starting with the module containing **main**:
    - Add the module to the end of the code
    - Assign addresses to its internal labels

- **Pass 3:** Rewrite the code with resolved symbols
  - Static symbols are rewritten with addresses
  - Dynamic symbols are rewritten with a jump to a stub that finds, loads, and links the needed object code

Granularity of inclusion is an object file
Library can be one big file or a collection of smaller modules
What Happens at Runtime?

How does an a.out execute?

• Some running process\(^1\) \(\text{(the “parent”)}\) starts the execution
  - In a shell, the user types the name of an executable at an input prompt
• The parent process creates (or “spawns”) a new process
• The new process executes a.out and returns
  - In a shell, the parent process waits for the child to complete
  - In a parallel computation, the parent may spawn many children and wait for them to complete, or it may simply continue without waiting

Obviously, this high-level view obscures some of the detail ...

\(^1\)The first running process is the boot process, created by a hardware signal and the code in the boot sequence.
Creating a New Process

To create a new process, the parent calls fork()

- fork() clones the parent process
  - New child has its own copies of all data
    → Includes file descriptors, program counter, ...
    → New child has a distinct process id (PID)
  - Fork() returns child’s PID to the parent and 0 to the child
- The parent waits on the child
- The child executes the new command

If the child only executes the new command ...

- It does not need the parent’s full address space
- vfork() copies just enough of the address space to allow an exec() call
  - It clones file descriptors, process status information, and the code
  - vfork() is safe if the process immediately calls exec()

```c
pid_t vfork(void);
pid_t process;
...
process = vfork();
if (process < 0)
  printf(stderr,"vfork() error.
");
else if (process == 0) {
  ... execute new command ...
}
else {
  wait()
}
```
Replacing the Address Space (Unix/Linux model)

To replace the child process’s address space, it calls exec()\(^1\)

• `execl()` takes a path to the executable file and a list of arguments
• `exec()` constructs the address space of the executable file
  ♦ If the file begins with `#! interpreter`, it runs the interpreter on the rest of the file
  ♦ If the file is an executable, it reads the file’s header information and:
    → Loads the program text, starting at the specified address
    → Loads any initialized data areas, starting at their specified addresses
    → Zeros any pages that are so specified
    → Branches to the **start address of main()**, as specified in the file’s header

• The original address space overwritten, except for parts of its environment
  ♦ File descriptors remain open (unless modified by `fcntl()`)
  ♦ The contents of the global `environ` remain intact
  ♦ Arguments passed through `exec()` are passed to `main` as `argc`, `argv`, and `envp`

```c
int main( int argc, char **argv, char **envp) {
  ...
}
```

\(^1\) `execve()` is the general form; `execl()` is a front end to `execve()`
A Couple More Points

The linker describes the address space

- The executable file is fully resolved, except for dynamically linked labels
  - `exec()`, in effect, inflates the address space based on header info in the `a.out`
  - It creates the code region of the address space, along with static data areas and global data areas

- **main()** must create the stack and the heap
  - `main()` starts the language’s **runtime system**
  - Allocates and initializes space for the stack and the heap
  - Takes care of other critical details
  - The compiler inserts a call to the runtime initialization code at the start of `main()`
    → Special case for the main entry point
    → In **C**, the compiler recognizes `main()` by name.
    → In other languages, the compiler may recognize it by declaration

- On exit, **main()** must shut down the environment
  - Compiler inserts a call to the runtime finalization code at the end of `main()`
What Happened to _@A?

We went off on this tale to learn how a compile-time label becomes a physical address. What did happen to _@A?

Two cases:

• _@A was a label in the code
  ♦ The assembler converted _@A to an offset from the start of the object module

• _@A was a label on a pseudo-operation
  ♦ The assembler converted _@A to an address of a data area (initialized or not)

Then, ...

• The linker computed _@A’s virtual address as it laid out memory
• The linker replaced occurrences of _@A with its actual virtual address
• exec() built the address space and initialized the memory at _@A
• The relevant instructions executed with the _@A’s virtual address
  ♦ Load, store, branch, jump, call, etc. see a virtual address rather than _@A
What Happens to the Virtual Address?

The hardware uses physical address *(for the most part)*

- At runtime, the operating system and the hardware must translate the virtual address of @_A_ to a physical address
- The process requires cooperation between the processor and the **OS**
- The process requires both hardware and software support

Remember last lecture?
Cache Memory

Modern hardware features multiple levels of cache & of TLB

- **L1** is typically core-private & tagged with virtual addresses
- **L2** (and beyond) is typically shared between cores & tagged with physical addresses
- Translation from virtual address to physical address is assisted by the **TLB** & requires cooperation between **OS** and the hardware
- Lookup in **L1** and **TLB** proceed in parallel
  - **TLB** can be as fast as **L1** because it is just a cache with very short lines
What Happens to the Virtual Address?

The hardware uses physical address *(for the most part)*

• At runtime, the operating system and the hardware must translate the virtual address of _@A to a physical address
• The process requires cooperation between the processor and the OS
• The process requires both hardware and software support

Remember last lecture?

• The L1 caches (code & data) have virtual tags
  ♦ If the code is in the L1 cache, the virtual address works
• Everything above L1 uses physical addresses
  ♦ Requires a fast mechanism to translate a virtual address to a physical address
  ♦ Hence, the creation of the TLB to speed that translation
    → TLB caches page frame address (physical address modulo page size) by virtual address
    → If the virtual address is not in the TLB, it requires an expensive lookup in the page tables and, perhaps a page fault (which changes the TLB and the virtual to physical mapping)
What Happened to __@A?

The combination of the software stack and the memory management hardware ensure that __@A in the code is a usable address at runtime

⇒ *It is complex, but it works ... billions of times a second*