Chapter 7 - Assembly Language

• Macros
• Assembly
• Linking and Loading

• Final Study Guide: #1, 5, 9, 13, 18, 23
Assembly Language

• What is an assembly language?
  – Symbols
    • OpCodes
    • Operands
    • Statement labels
  – Meta-level language facilities
    • Macros
    • Conditional assembly
  – Modularization facilities

<table>
<thead>
<tr>
<th>Label</th>
<th>Opcode</th>
<th>Operands</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>EAX,1</td>
<td></td>
<td>; register EAX = 1</td>
</tr>
<tr>
<td>ADD</td>
<td>EAX,J</td>
<td></td>
<td>; register EAX = I + J</td>
</tr>
<tr>
<td>MOV</td>
<td>N,EAX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The assembly language level is the last level we will look at, and the highest level you will probably never touch!

A radical shift: the assembly language level is the first level implemented via translation. We’ve seen direct execution and interpretation before, but never translation. Most higher levels are implemented via translation.

Essentially, this chapter is a very brief introduction to many issues in compilation of higher-level languages.

Assembly language: each statement corresponds to one instruction at the OS level.

Plus: access to ALL machine instructions and capabilities

Minus: NOT PORTABLE.

What is an assembly language?

1. Symbolic Op codes
2. Symbolic operands
3. Labels
4. Macros
5. Modularization facilities

Symbols make it easier to read and write code. Also make it easier to change, since you don’t have to recalculate a bunch of numbers every time to add or
PsuedoInstructions

- BASE EQU 100
- Count DW 0
- Fibonacci Proc ...
- Swap Macro
- ...
- PUBLIC/EXTERN
  - xyz PUBLIC xyz
  - DW 3
  - Extern xyz
  - addc xyz, 3

<table>
<thead>
<tr>
<th>Pseudoinstr</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEGMENT</td>
<td>Start a new segment (text, data, etc.) with certain attributes</td>
</tr>
<tr>
<td>ENDS</td>
<td>End the current segment</td>
</tr>
<tr>
<td>ALIGN</td>
<td>Control the alignment of the next instruction or data</td>
</tr>
<tr>
<td>EQU</td>
<td>Define a new symbol equal to a given expression</td>
</tr>
<tr>
<td>DB</td>
<td>Allocate storage for one or more (initialized) bytes</td>
</tr>
<tr>
<td>DD</td>
<td>Allocate storage for one or more (initialized) 16-bit halfwords</td>
</tr>
<tr>
<td>DW</td>
<td>Allocate storage for one or more (initialized) 32-bit words</td>
</tr>
<tr>
<td>DQ</td>
<td>Allocate storage for one or more (initialized) 64-bit double words</td>
</tr>
<tr>
<td>PROC</td>
<td>Start a procedure</td>
</tr>
<tr>
<td>ENDP</td>
<td>End a procedure</td>
</tr>
<tr>
<td>MACRO</td>
<td>Start a macro definition</td>
</tr>
<tr>
<td>ENDM</td>
<td>End a macro definition</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>Export a name defined in this module</td>
</tr>
<tr>
<td>EXTERN</td>
<td>Import a name from another module</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>Fetch and include another file</td>
</tr>
<tr>
<td>IF</td>
<td>Start conditional assembly if the IF condition above was true</td>
</tr>
<tr>
<td>ENDIF</td>
<td>End conditional assembly</td>
</tr>
<tr>
<td>COMMENT</td>
<td>Define a new start-of-comment character</td>
</tr>
<tr>
<td>PAGE</td>
<td>Generate a page break in the listing</td>
</tr>
<tr>
<td>END</td>
<td>Terminate the assembly program</td>
</tr>
</tbody>
</table>

Assembly introduces a few additional opcodes beyond the OS level. Some of these make writing assembly easier, others affect the assembly process.

Many of these have correlates in higher level languages (EQU, DB, PROC, PUBLIC)

Many others don’t (MACRO, IF,….)

PUBLIC allows you to declare that a symbol defined in one file should be made available to other files

(why not just make ALL symbols available? Namespace issues… explain this)

EXTERN allows you to say that a symbol you are referencing in one file is defined somewhere else.

(then how can you run the program? Can’t, until you gather all the files together.)
Conditional Assembly

- WDSZ EQU 16
- IF WDSZ GT 16
- WSIZE: DW 32
- ELSE
- WSIZE: DW 16
- ENDIF

- WDSZ EQU 16
- IF WDSZ GT 16
- Const1: DD 0
- ELSE
- Const1: DW 0
- ENDIF

Conditional assembly statements are INTERPRETED AT ASSEMBLY TIME!

That is, you can write code that runs inside the assembler.

This can be VERY confusing, but is very powerful, and is heavily used by experienced assembly (and C) pgmr's. Note that since this code runs at ASSEMBLY time, it can only refer to variables whose values are known at assembly time (typically, those defined by EQU or statement labels or the like).

So, in the above two examples, space for only ONE copy of the parameter WSIZE or Const1 will be allocated.
Macros

C++ calls them “inline” methods.

Macros do a code substitution at assembly time!

A macro can include formal parameters and conditional assembly statements, in which case it is best thought of as a program for generating code!

Most modern higher level languages don’t have anything like this - Until … the web level!

This style is very common in web programming (e.g., javascript or VBS embedded in HTML is pretty much the same idea…)

<table>
<thead>
<tr>
<th>Item</th>
<th>Macro call</th>
<th>Procedure call</th>
</tr>
</thead>
<tbody>
<tr>
<td>When is the call made?</td>
<td>During assembly</td>
<td>During execution</td>
</tr>
<tr>
<td>Is the body inserted into the object program every place the call is made?</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Is a procedure call instruction inserted into the object program and later executed?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Must a return instruction be used after the call is done?</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>How many copies of the body appear in the object program?</td>
<td>One per macro call</td>
<td>1</td>
</tr>
</tbody>
</table>
Why do we care how an assembler works?

For the same reason we care how digital logic implements the microarchitecture level

- because it is good for you.

(because many of the same problems occur in programs you will have to write
And analogous solutions will be useful).

Symbol table is usually maintained as a hash table.

Consider the small piece of code above, and the corresponding IJVM binary.
What do we need to do to assemble it?
Well, we could just try translating one instruction at a time.
Note we have to remember the constant assigned to BufSize.
But, how do we compile the mov? We don’t know where I is yet…

So, we need TWO passes. Pass 1 figures out where everything is and remembers
In the Symbol Table

Pass 2 outputs code.
Pass one reads the code, process label, if present, look up opcode, checks type process line according to opcode type (real or psuedo) then outputs information it has gathered

Note it sorts literal table at end - why?
Now we have enough information to actually output code. Let’s walk through the process to see what we need to do to generate code. Pass 2 can skip EQU, it has already processed it.

MOV - according to table, when first operand is EAX, then a special opcode is available that saves one byte.

We know where I is when processing L1, so can build instruction and write out binary.

Similarly, we know where L2 is (location 5) so can build code for JMP instruction.

So are we done? Not quite. What if we had tried to call an OS procedure? Or reference a label in ANOTHER file.

How would we know where it was?

We wouldn’t.

Notice we started object code at 0 - assembler does that for each file!

We’re not done yet!

But is L1 really at location 0 in memory? We’ll talk about that later.
Obj module format

1. Identification - module a
2. Entry point table: labels that this module has declared PUBLIC, and the addresses (0 offset)
3. External ref table: labels that this module has declared EXTERN, and locations in the instructions and constants that need this address
   Remember, an EXTERN is a label you reference, but don’t define, like an operating system procedure entry label or the name of a procedure in another module.
5. Relocation dictionary - a list of all the memory references in the machine code and constants. We’ll see in a minute why we need that.
6. End - misc stuff.
Linking

- Construct table of obj modules and lengths
- Assign start addr to each module
- Add relocation constant to each mem ref
- Set addr in procedure calls

Linking is the process of combing assembler output from several modules, deciding where each should go, resolving references to labels defined elsewhere, etc.

Suppose we have four modules: a, b, c, d all of which need to be combined to build a program ABCD

Step one: decide where each module will go in binary image, and put it there.

Step two: change all the addresses in the image so they are correct (this is called: “relocation”).

Note we don’t want to change the BUFSZ constant, so we need to know the difference between an address and an immediate.

So, may not put them both in the same table.

Steps shown in slide.
A DLL is a *dynamic link library*. That is, a collection of modules (library) that is linked when the program is started rather than at application build time.

Major reasons for this:

1. to allow multiple applications to share a single copy of the code for the library,
2. to reduce the size of the distributed application (if you already have the library)
3. To allow updates to the library without re-distributing the app.

Both Windows and Unix support dynamic linking. In Unix it is called shared libraries.