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

Label	Opcode	Operands	Comments
FORMULA:	MOV	EAX,I	; register EAX = I
	ADD	EAX,J	; register EAX = I + J
	MOV	N,EAX	; N = I + J

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,

PsuedoInstructions

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

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

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 correlaries 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 avaialble 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, untill 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) pgmrs. 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

```
MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX

MOV     EAX,P
MOV     EBX,Q
MOV     Q,EAX
MOV     P,EBX
```

```
MOV   EAX,P                CHANGE   MACRO P1, P2
MOV   EBX,Q                         MOV EAX,P1
MOV   Q,EAX                         MOV EBX,P2
MOV   P,EBX                         MOV P2,EAX
                                    MOV P1,EBX
MOV   EAX,R                         ENDM
MOV   EBX,S
MOV   S,EAX                         CHANGE P, Q
MOV   R,EBX
                                    CHANGE R, S
```

```
SWAP    MACRO
        MOV EAX,P
        MOV EBX,Q
        MOV Q,EAX
        MOV P,EBX
        ENDM

        SWAP

        SWAP
```

Item	Macro call	Procedure call
When is the call made?	During assembly	During execution
Is the body inserted into the object program every place the call is made?	Yes	No
Is a procedure call instruction inserted into the object program and later executed?	No	Yes
Must a return instruction be used after the call is done?	No	Yes
How many copies of the body appear in the object program?	One per macro call	1

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…)

The Assembly Process

```
BUFSZ  EQU   100
L1     MOV   EAX, I         5
L2     MOV   EBX, K         6
       MOV   ECX, BUFSZ     6
       IMUL  EAX, EAX       2
       JMP   L2             5
I      DW    1001           4
J      DD    1              8
K      DD    3              8
```

Symbol	Value	Other
BUFSZ	100	

Opcode	First operand	Second operand	Hexadecimal opcode	Instruction length	Instruction class
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

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.

```
public static void pass_one( ) {
  // This procedure is an outline of pass one of a simple assembler.
  boolean more_input = true;      // flag that stops pass one
  String line, symbol, literal, opcode;     // fields of the instruction
  int location_counter, length, value, type;  // misc. variables
  final int END_STATEMENT = -2;  // signals end of input

  location_counter = 0;             // assemble first instruction at 0
  initialize_tables( );             // general initialization

  while (more_input) {              // more_input set to false by END
    line = read_next_line( );       // get a line of input
    length = 0;                     // # bytes in the instruction
    type = 0;                       // which type (format) is the instruction

    if (line_is_not_comment(line)) {
      symbol = check_for_symbol(line);    // is this line labeled?
      if (symbol != null)           // if it is, record symbol and value
        enter_new_symbol(symbol, location_counter);
      literal = check_for_literal(line);    // does line contain a literal?
      if (literal != null)          // if it does, enter it in table
        enter_new_literal(literal);

      // Now determine the opcode type.  -1 means illegal opcode.
      opcode = extract_opcode(line);  // locate opcode mnemonic
      type = search_opcode_table(opcode);   // find format, e.g. OP REG1,REG2
      if (type < 0)                 // if not an opcode, is it a pseudoinstruction?
        type = search_pseudo_table(opcode);
      switch(type) {                // determine the length of this instruction
        case 1: length = get_length_of_type1(line);  break;
        case 2: length = get_length_of_type2(line);  break;
        // other cases here
      }
    }

    write_temp_file(type, opcode, length, line);// useful info for pass two
    location_counter = location_counter + length;// update loc_ctr
    if (type == END_STATEMENT) {    // are we done with input?
      more_input = false;           // if so, perform housekeeping tasks
      rewind_temp_for_pass_two( );    // like rewinding the temp file
      sort_literal_table( );        // and sorting the literal table
      remove_redundant_literals( );// and removing duplicates from it
    }
  }
```

Assembly Pass 1

Pass one reads the code,

process label, if present, t

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?

Assembly Pass 2

```
public static void pass_two( ) {
  // This procedure is an outline of pass two of a simple assembler.
  boolean more_input = true; // flag that stops pass one
  String line, opcode;          // fields of the instruction
  int location_counter, length, type;   // misc. variables
  final int END_STATEMENT = -2;   // signals end of input
  final int MAX_CODE = 16;// max bytes of code per instruction
  byte code[ ] = new byte[MAX_CODE]; // holds generated code per

  location_counter = 0;         // assemble first instruction at 0

  while (more_input) {          // more_input set to false by END
    type = read_type( );        // get type field of next line
    opcode = read_opcode( );  // get opcode field of next line
    length = read_length( ); // get length field of next line
    line = read_line( );        // get the actual line of input

    if (type != 0) {            // type 0 is for comment lines
      switch(type) {            // generate the output code
        case 1: eval_type1(opcode, length, line, code);  break;
        case 2: eval_type2(opcode, length, line, code);  break;
        // other cases here
      }
    }

    write_output(code);         // write the binary code
    write_listing(code, line);// print one line on the listing
    location_counter = location_counter + length;// update loc_ctr
    if (type == END_STATEMENT) {// are we done with input?
      more_input = false;    // if so, perform housekeeping tasks
      finish_up( );             // odds and ends
    }
  }
}
```

BUFSZ	**EQU**	**100**	
L1	**MOV**	**EAX, I**	**5**
L2	**MOV**	**EBX, K**	**6**
	MOV	**ECX, BUFSZ**	**6**
	IMUL	**EAX, EAX**	**2**
	JMP	**L2**	**5**
I	**DW**	**1001**	**4**
J	**DD**	**1**	**8**
K	**DD**	**3**	**8**

Symbol	Value	Other
BUFSZ	100	EQU
L1	0	INST
L2	5	INST
I	24	W
J	32	D
K	40	D

Opcode	First operand	Second operand	Hexadecimal opcode	Instruction length	Instruction class
AAA	—	—	37	1	6
ADD	EAX	immed32	05	5	4
ADD	reg	reg	01	2	19
AND	EAX	immed32	25	5	4
AND	reg	reg	21	2	19

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?

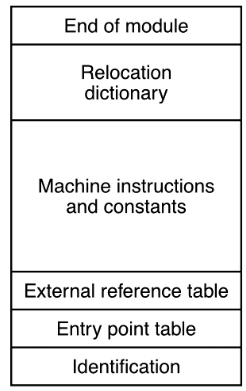
We wouldn't.

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

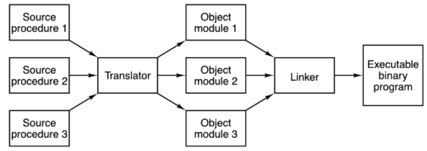


1. Identification - module a
2. Entry point table: labels that this module has declared PUBLIC, and the addresses (0 offset)
3. External ref table: labels that this module has declared EXTERN, and locations in the instructions and constants that need this address

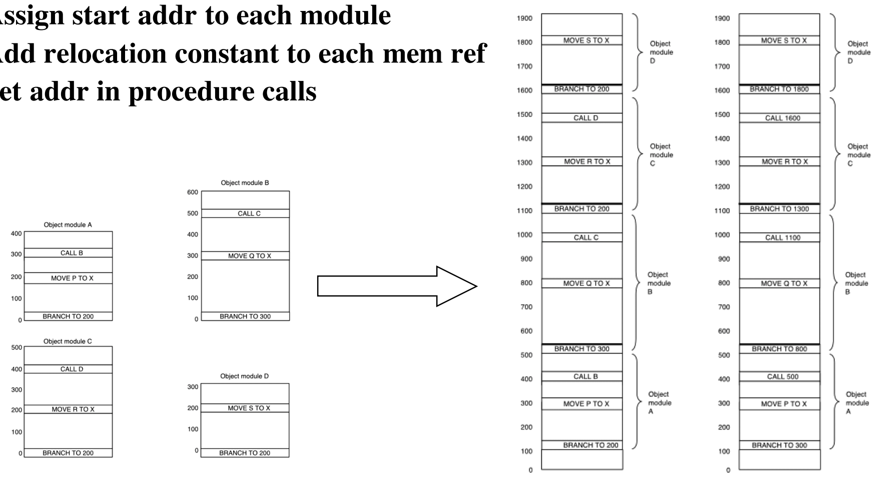
Remember, an EXTERN is a label you reference, but don't define, like an operating system procedure entry label or the name of a procedure in another module.

4. Machine code: binary produced by the assembler.
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.

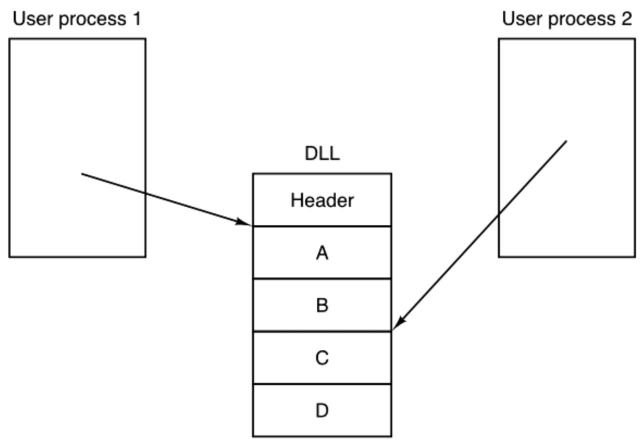
Step two: change all the addresses in the image so they are correct (this is called: “relocation”).

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.

Dynamic Linking (DLL)



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.