The Assembly Language Level

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University

Wolfgang.Schreiner@risc.uni-linz.ac.at
http://www.risc.uni-linz.ac.at/people/schreine

Assembly Language Level

Implemented by translation rather than by interpretation.

- Source: assembly language.
  - The source program is a sequence of text statements.
- Target: machine language.
  - Each statement of the source program is translated to exactly one machine instruction.
  - Object file is generated.
- Translator: assembler
  - Translates symbolic instruction names to numerical instruction codes.
  - Translates register names to register numbers.
  - Translates symbolic variable names to numerical memory locations.

Programmer still has access to all features of the target machine.

Assembly Language Use

Why would one want to program in assembly language?

- Performance.
  - Assembly language programmer can write faster code than high-level language programmer.
    - Perhaps 3 times faster.
  - Example: time-critical numerical operations, graphics operations, embedded applications, ...
- Hardware Access.
  - Assembly language programmer has complete access to hardware.
  - Device controllers, OS interrupt handlers, ...

An assembler is typically also the back-end of a compiler.

Assembly Language Statement

Compute $N = I + J$.

Label	Opcode	Operands	Comments
FORMULA:	MOV	EAX,I	; register EAX = I
	ADD	EAX,J	; register EAX = I+J
	MOV	N,EAX	; N=I+J
I	DW	3	; reserve 4 bytes initialized to 3
J	DW	4	; reserve 4 bytes initialized to 4
N	DW	0	; reserve 4 bytes initialized to 0

Machine instructions plus declaratives for memory reservation.

Pseudoinstructions

Assembler directives that control the operation of the assembler.

- Example: Microsoft MASM directives.
  - `SEGMENT`, `ENDS`: start/end a segment (text, data, etc).
  - `ALIGN`: control alignment of next instruction or data.
  - `EQU`: give a symbolic name to an expression.
  - `DW`: allocate storage for one or more 32 bit words.
  - `IF`, `ELSE`, `ENDIF`: conditional assembly.

```
WORDSIZE EQU 16
IF WORDSIZE GT 16
  WSIZE: DW 32
ELSE
  WSIZE: DW 16
ENDIF
```

Macros

- A macro is a named piece of text.
  - Macro definition gives name to piece of text.
  - A macro call inserts the name of the text.
  - A macro may have parameters for customization.

```
SWAP MACRO P1,P2     ; P1 and P2 are interchanged
     MOV EAX,P1      ; EAX := P1
     MOV EBX,P2      ; EBX := P2
     MOV P2,EAX      ; P2 := EAX
     MOV P1,EBX      ; P1 := EBX
     ENDM

     SWAP P,Q
     SWAP R,S
```

Macros allow to avoid the overhead of procedure calls.

The Assembly Process

Many assemblers process the program in two passes.

- Two-pass assemblers.
  - Forward references have to be resolved.

```
        JMP LABEL
        ...
LABEL   ...
```

- Pass One: collect symbol information.
  - Definition of symbols (labels) are read and stored in table.
- Pass Two: resolve symbols.
  - Each statment is read, assembled, and output.

First pass may create intermediate form of program in memory.

Pass One

Build symbol table that contains values of all symbols.

Label	Opcode	Operands	Comments	Length	Position
MARIA:	MOV	EAX,I	EAX=I	5	100
	MOV	EBX,J	EBX=J	6	105
ROBERTA:	MOV	ECX,K	ECX=K	6	111
	IMUL	EAX,EAX	EAX=I*I	2	117
	IMUL	EBX,EBX	EBX=J*J	3	119
	IMUL	ECX,ECX	ECX=K*K	3	122
MARILYN:	ADD	EAX,EBX	EAX=I*I+J*J	2	125
	ADD	EAX,ECX	EAX=I*I+J*J+K*K	2	127

Symbol	Value	Other Information
MARIA	100	...
ROBERTA	111	...
MARILYN	125	...

Pass Two

Generates the object file and prints an assembly listing.

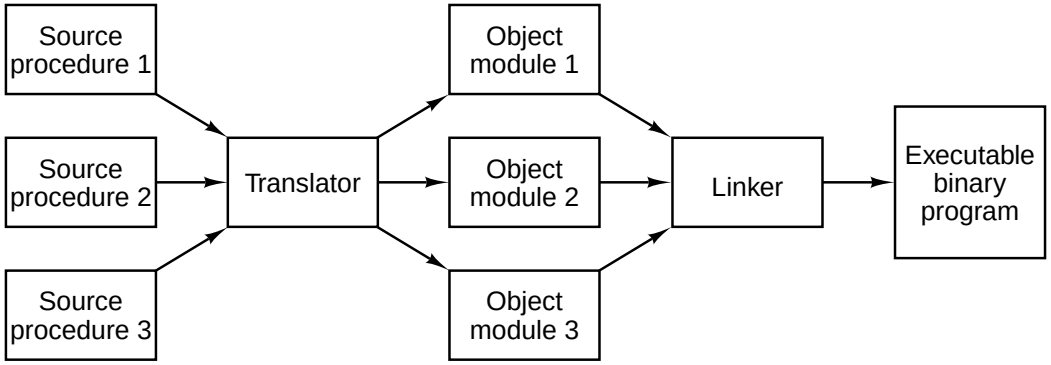
- Errors reported by assembler:
  - Symbol has been used but not defined.
  - Symbol has been defined more than once.
  - Name in opcode field is not a legal opcode.
  - Opcode is supplied with too few or too many operands.
  - Illegal register use (e.g. a branch to a register).
- Also produces certain information needed for the linker.
  - Links up procedures from different object files into a single executable file.

Only low level errors are detected.

Linking and Loading

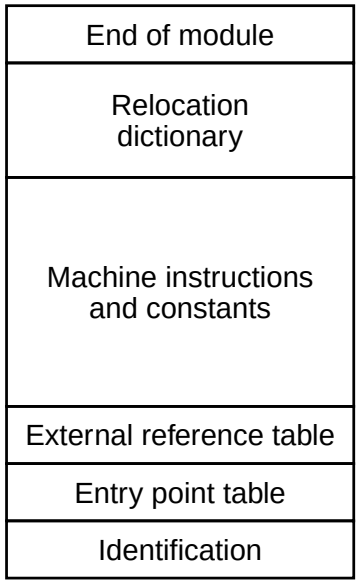
Most programs consist of multiple object files.

- High-level language program consists of multiple modules.
  - Compiler generates assembly language file from each module.
  - Assembler generates object module from each assembly language file.
  - Object modules must be linked together.
- Linker (linking loader).
  - Generates executable binary program from object files.



Structure of an Object Module

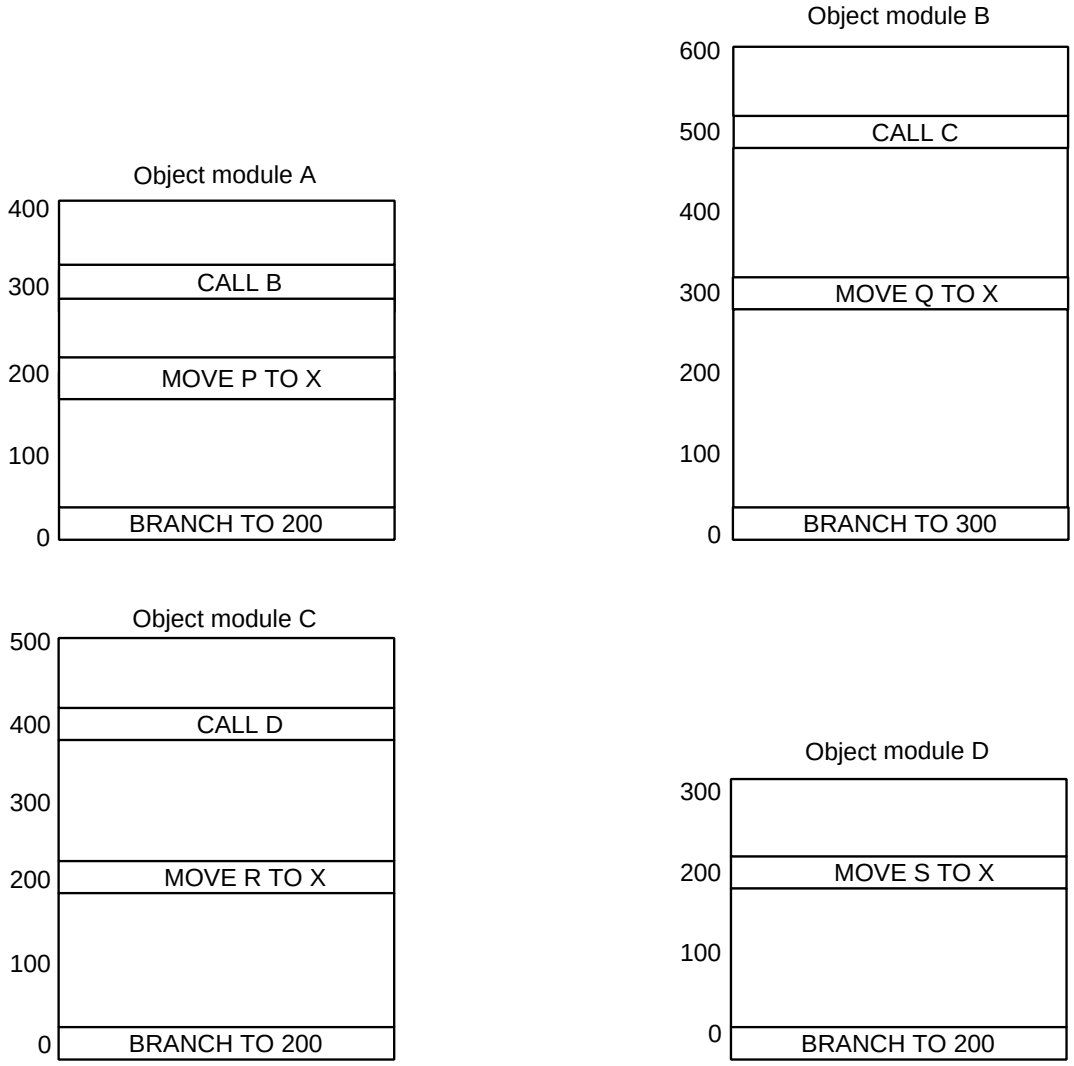
- Entry point table
  - List of symbols defined in module.
  - Can be referenced by other modules.
  - Name plus address.
- External reference table
  - List of symbols referenced by the module.
  - Are defined in other modules.
  - Name plus machine instructions that use this symbol.
- Relocation dictionary
  - List of addresses in program that need to be relocated.



Most linkers use two passes of table building and module relocation.

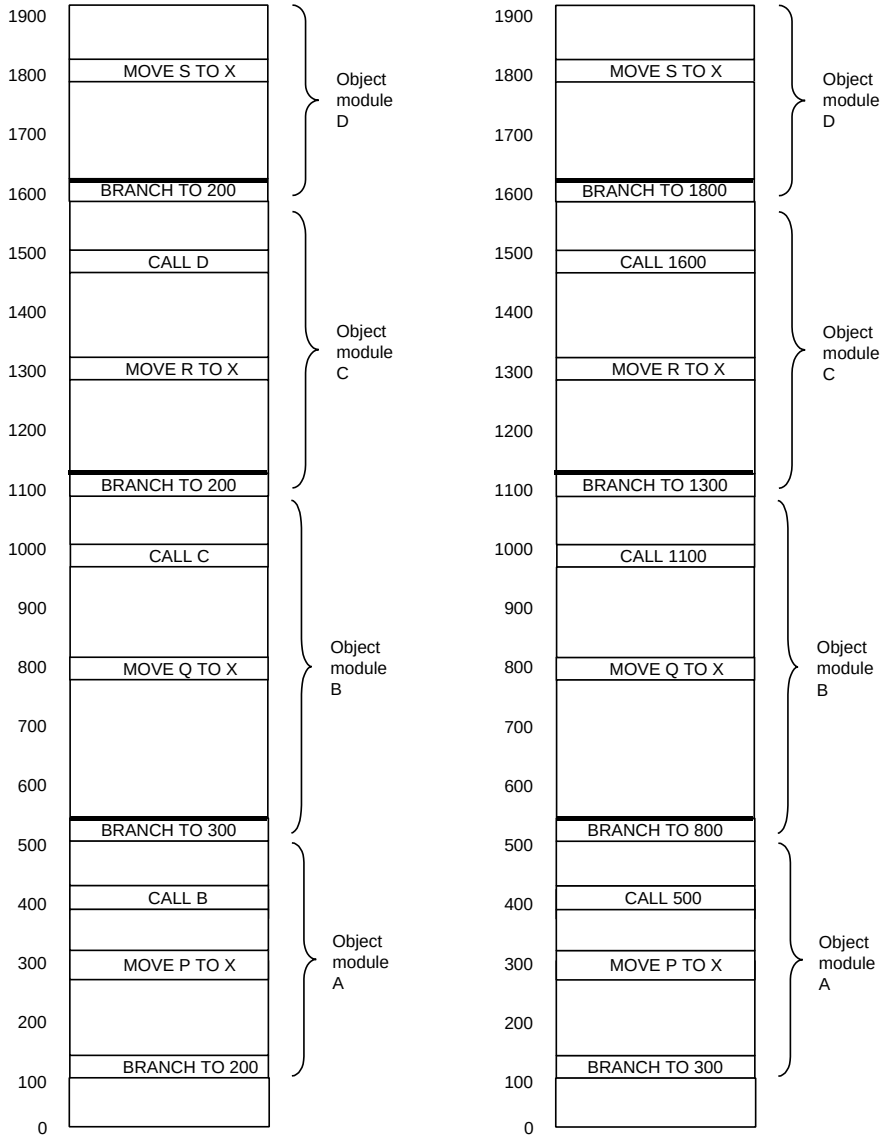
Address Spaces

- Each object module has its own address space.
  - Starts at address 0.

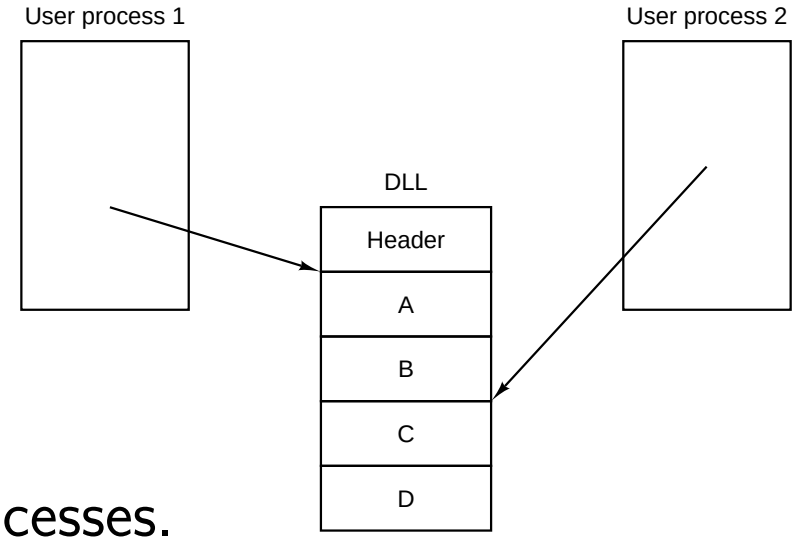


Linking Process

1. Construct table of modules.
   - Length of each module.
2. Assign start address to modules.
   - Modules are placed in sequence.
3. Find all memory instructions.
   - Add module address to each address.
4. Find procedure call instructions.
   - Insert procedure address.



Dynamic Linking



Linking may occur during execution.

- Supported by modern OS.
  - MS windows: DLL (Dynamic Link Library).
  - Unix: shared library.
- Module may be used by multiple processes.
  - Implicit linking: program is statically linked with an import library which refers to the DLL; when OS loads program, it checks for missing DLLs and loads them.
  - Explicit linking: user program makes explicit calls at runtime to bind to a DLL and to get the addresses of the procedures it needs.

Dynamic linking reduces the sizes of program files.