Computer Organization &
Assembly Language Programming

CSE 2312
Lecture 21 Linking and Loading

Junzhou Huang, Ph.D.
Department of Computer Science and Engineering
Linking

• Why Linking
  – Most programs consist of more than one procedure.
  – Compilers and assemblers generally translate one procedure at a time and put the translated output on disk.
  – Before the program can be run, all the translated procedures must be found and linked together properly.
  – If virtual memory is not available, the linked program must be explicitly loaded into main memory as well.
  – Programs that perform these functions are called by various names, including linker, linking loader, and linkage editor.

• Two steps for the complete translation of a source program
  – Compilation or assembly of the source procedures.
  – Linking of the object modules.
  – The first step is performed by the compiler or assembler
  – The second one is performed by the linker.
Linker Example

Generation of an executable binary program from a collection of independently translated source procedures requires using a linker.
Translation vs. Linking

• **Levels: changed or not?**
  – The translation from source procedure to object module represents a change of level because the source language and target language have different instructions and notation.
  – The linking process does not represent a change of level, since both the linker’s input and the linker’s output are programs for the same virtual machine.
  – The linker’s function is to collect procedures translated separately and link them together to be run as a unit called an **executable binary program**.

• **Separate-object-module technique**
  – Compilers and assemblers translate each source procedure as a separate entity.
  – If a compiler or assembler were to read a series of source procedures and directly produce a ready-to-run machine language program, changing one statement in one source procedure would require that all the source procedures be retranslated.
  – If the separate-object-module technique is used, it is only necessary to retranslate the modified procedure and not the unchanged ones, although it is necessary to relink all the object modules again.
  – Linking is usually much faster than translating. So, two-step of translating and linking can save a great deal of time during the development of a program.
Tasks Performed by the Linker

• Different Modules
  – At the start of pass one of the assembly process, the instruction location counter is set to 0.
  – This step is equivalent to assuming that the object module will be located at (virtual) address 0 during execution.
  – The following Figure shows four object modules for a generic machine. In this example, each module begins with a BRANCH instruction to a MOVE instruction within the module.

Each module has its own address space, starting at 0.
Linker

• What the linkers do?
  – In order to run the program, the linker brings the object modules into main memory to form the image of the executable binary program.
  – Make an exact image of the executable program’s virtual address space inside the linker and position all the object modules at their correct locations.
  – If no enough (virtual) memory to form the image, a disk file can be used.
  – Typically, a small section of memory starting at address zero is used for other purposes, such as interrupt vectors, so programs often start above 0.
  – Of course, we also can (arbitrarily) started programs at address 100.
Linking Modules

• **Left**
  – The 4 object modules after being positioned in the binary image but before being relocated and linked.

• **Right**
  – The same object modules after linking and after relocation has been performed.
  – Together they form an executable binary program, ready to run
Relocation and External Reference

• Relocation Problem
  – Even the program is loaded into the image of the executable binary file, it is not yet ready for execution. Why?
  – Consider what would happen if execution began with the instruction at the beginning of module A? The program would not branch to the MOVE instruction as it should, because that instruction is now at 300.
  – This problem, called the relocation problem, occurs because each object module represents a separate address space.
  – So, before it is ready for execution, all the object modules must be merged together into a single address space.

• External Reference
  – The procedure call instructions in the above Figure (a) will not work either.
  – At address 400, the programmer had intended to call object module B, but because each procedure is translated by itself, the assembler has no way of knowing what address to insert into the CALL B instruction.
  – The address of object module B is not known until linking time. This problem is called the external reference problem.
  – Both of these problems can be solved in a simple way by the linker.
Steps for Linking

• Linking
  – The linker merges the separate address spaces of the object modules into a single linear address space in the following steps

• Steps
  – It constructs a table of all the object modules and their lengths.
  – Based on this table, it assigns a starting address to each object module.
  – It finds all the instructions that reference memory and adds to each a relocation constant equal to the starting address of its module.
  – It finds all the instructions that reference other procedures and inserts the address of these procedures in place.

• Object Module Table
  – Constructed in step 1
  – Shown in the right Figure
  – Gives the name, length, and starting address of each module.

<table>
<thead>
<tr>
<th>Module</th>
<th>Length</th>
<th>Starting address</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>400</td>
<td>100</td>
</tr>
<tr>
<td>B</td>
<td>600</td>
<td>500</td>
</tr>
<tr>
<td>C</td>
<td>500</td>
<td>1100</td>
</tr>
<tr>
<td>D</td>
<td>300</td>
<td>1600</td>
</tr>
</tbody>
</table>
Structure of an Object Module

- Object modules contain 6 parts
  - Identification
  - Entry Point Table
  - External Reference Table
  - Machine Instructions and Constants
  - Relocation Dictionary
  - End of Module
6 Parts

- **Machine Instructions and Constants**
  - The fourth part of the object module is the assembled code and constants.
  - This part is the only one that will be loaded into memory to be executed.
  - The other five parts will be used by the linker to help it do its work and then discarded before execution begins.

- **Relocation Dictionary**
  - The fifth part of the object module is the relocation dictionary.
  - Instructions that contain memory addresses must have a relocation constant added.
  - Since the linker has no way of telling by inspection which of the data words in part four contain machine instructions and which contain constants, information about which addresses are to be relocated is provided in this table.
  - The information may take the form of a bit table, with 1 bit per potentially relocatable address, or an explicit list of addresses to be relocated.

- **End of Module**
  - The sixth part is an end-of-module mark, perhaps a checksum to catch errors made while reading the module, and the address at which to begin execution.
6 Parts

• Identification
  – This first part contains the name of the module, certain information needed by
    the linker, such as the lengths of the various parts of the module, and sometimes
    the assembly date.

• Entry Point Table
  – The second part of the object module is a list of the symbols defined in the
    module that other modules may reference, together with their values.
  – For example, if the module consists of a procedure named bigbug, the entry
    point table will contain the character string “bigbug” followed by the address to
    which it corresponds.
  – The assembly language programmer indicates which symbols are to be declared as
    entry points by using a pseudoinstruction such as PUBLIC

• External Reference Table
  – The third part of the object module consists of a list of the symbols that are
    used in the module but which are defined in other modules, along with a list of
    which machine instructions use which symbols.
  – The linker needs the latter list in order to be able to insert the correct addresses
    into the instructions that use external symbols.
  – A procedure can call other independently translated procedures by declaring the
    names of the called procedures to be external.
Links require 2 passes

• **Pass one**
  – The linker reads all the object modules
  – Builds up a table of module names and lengths,
  – Build up a global symbol table consisting of all entry points and external references.

• **Pass Two**
  – The object modules are read, relocated, and linked one module at a time.
Binding Time and Dynamic Relocation

• **Problem**
  – In a multiprogramming system, a program can be read into main memory, run for a little while, written to disk, and then read back into main memory to be run again. With many programs, it is difficult to ensure that a program is read back into the same locations every time.

• **Binding Time**
  – The problem of moving programs that have already been linked and relocated is intimately related to the time at which the final binding of symbolic names onto absolute physical memory addresses is completed.
  – When a program is written, it contains symbolic names for memory addresses, for example, BR L. The time at which the actual main memory address corresponding to L is determined is called the **binding time**.

• **Six possibilities for the binding time exist:**
  – When the program is written.
  – When the program is translated.
  – When the program is linked but before it is loaded.
  – When the program is loaded.
  – When a base register used for addressing is loaded.
  – When the instruction containing the address is executed.
Example

• Example
  – Figure shows what would happen if the already relocated program were reloaded at address 400 instead of address 100 where the linker put it originally.
  – All the memory addresses are incorrect; moreover, the relocation information has long since been discarded.
  – Even if the relocation information were still available, the cost of having to relocate all the addresses every time the program was swapped would be too high.
  – The linking method described above binds symbolic names to absolute addresses during linking, which is why moving programs after linking fails.
Dynamic Linking
Dynamic Linking

• **Why Dynamic Linking**
  – The introduced linking strategy above has the property that all procedures that a program might call are linked before the program can begin execution.
  – Completing all linking before beginning execution does not take advantage of the full capabilities of the virtual memory.
  – Many programs have procedures that are called only under special cases. For example, compilers have procedures for compiling rarely used statements, plus procedures for handling error conditions that seldom occur.
  – A more flexible way to link each procedure at the time it is first called. This process is known as dynamic linking.

• **Dynamic Linking in MULTICS**
  – Dynamic linking was pioneered by MULTICS. In the MULTICS form of dynamic linking, each program is associated with a linkage segment, which contains one block of information for each procedure that might be called.
  – When dynamic linking is being used, procedure calls in the source language are translated into instructions that indirectly address the first word of the corresponding linkage block.
  – The dynamic linker is invoked only the first time a procedure is called and not thereafter.
Dynamic Linking in MULTICS

Before EARTH is called.

After EARTH has been called and linked.
Dynamic Linking in Windows

• Dynamic Link Library (DLL)
  – All Windows OS support and rely on dynamic linking heavily. Dynamic linking uses a special file format called a DLL (Dynamic Link Library). DLLs can contain procedures, data, or both.
  – They are commonly used to allow two or more processes to share library procedures or data. The most common form of a DLL is a library consisting of a collection of procedures that can be loaded into memory and accessed by multiple processes at the same time.
  – A DLL is constructed by the linker from a collection of input files. DLLs are commonly constructed from collections of library procedures that are likely to be needed by multiple processes.

• Advantages of DLLs
  – First advantage of using DLLs is saving space in memory and on disk.
    For static cases, some common library would appear in many executable binaries in memory, which wastes space. With DLLs, each library only appears once in memory.
  – This approach makes it easy to update library procedures, even after the programs using them have been compiled and linked.
    Using DLLs can fix bugs in the libraries by just distributing new DLL files over the Internet, without requiring any changes to the main program binaries (good for commercial software packages)
Dynamic Linking in Windows

Use of a DLL file by two processes.
Implicit and Explicit Linking

• **Implicit Linking**
  – The user’s program is statically linked with a special file called an import library that is generated by a utility program that extracts certain information from the DLL.
  – When a program using implicit linking is loaded into memory for execution, Windows examines it to see which DLLs it uses and checks to see if all of them are already in memory. Those that are not in memory are loaded immediately.
  – They also have to be mapped into the program’s virtual address space. At this point, the user program is ready to run and can call the procedures in the DLLs as though they had been statically bound with it.

• **Explicit Linking**
  – This approach does not require import libraries and does not cause the DLLs to be loaded at the same time the user program is.
  – Instead, the user program makes an explicit call at run time to bind to a DLL, then makes additional calls to get the addresses of procedures it needs.
  – Once these have been found, it can call the procedures.
  – When it is all done, it makes a final call to unbind from the DLL.
  – When the last process unbinds from a DLL, the DLL can be removed from memory.
DLL vs. Executable Binary

• **DLL vs. Executable Binary**
  – Building a DLL file is very much like building an executable binary program, except that a special flag is given to the linker to tell it to make a DLL.
  – The main difference between them is that a DLL cannot be started and run on its own (because it has no main program).
  – It also has different information in its header.
  – In addition, the DLL as a whole has several extra procedures not related to the procedures in the library.
  – These procedures can allocate and deallocate memory or manage other resources needed by the DLL.

• **Procedures in DLL**
  – It is important to realize that a procedure in a DLL does not have any identity of its own (as a thread or process does).
  – It runs in the caller’s thread and uses the caller’s stack for its local variables.
  – It can have process-specific static data (as well as shared data) and otherwise behaves the same as a statically-linked procedure.
  – The only essential difference is how the binding to it is performed.
Data Transfer Instructions

- **Data Transfer Instructions**
  - MOV or MOVB
  - XCHG or XCHGB
  - PUSH or POP
  - PUSHF or POPF
  - XLAT

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description</th>
<th>Operands</th>
<th>Status flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV(B)</td>
<td>Move word, byte</td>
<td>$r \leftarrow e, e \leftarrow r, e \leftarrow #$</td>
<td>O: - S: - Z: - C: -</td>
</tr>
<tr>
<td>XCHG(B)</td>
<td>Exchange word</td>
<td>$r \leftarrow e$</td>
<td>O: - S: - Z: - C: -</td>
</tr>
<tr>
<td>LEA</td>
<td>Load effective address</td>
<td>$r \leftarrow #e$</td>
<td>O: - S: - Z: - C: -</td>
</tr>
<tr>
<td>PUSH</td>
<td>Push onto stack</td>
<td>$e, #$</td>
<td>O: - S: - Z: - C: -</td>
</tr>
<tr>
<td>POP</td>
<td>Pop from stack</td>
<td>$e$</td>
<td>O: - S: - Z: - C: -</td>
</tr>
<tr>
<td>PUSHF</td>
<td>Push flags</td>
<td>-</td>
<td>O: - S: - Z: - C: -</td>
</tr>
<tr>
<td>POPF</td>
<td>Pop flags</td>
<td>-</td>
<td>O: - S: - Z: - C: -</td>
</tr>
<tr>
<td>XLAT</td>
<td>Translate AL</td>
<td>-</td>
<td>O: - S: - Z: - C: -</td>
</tr>
</tbody>
</table>
Move Instructions

• **Copy and Move Instructions**
  – Move from source to destination. Syntax: **MOV destination, source**
  – If the source is a register, the destination can be an effective address. In this table a register operand is indicated by an $r$ and an effective address by an $e$, so this operand combination is denoted by $e \leftarrow r$.
  – Since, in the instruction syntax, the destination is the first operand and the source is the second operand, the arrow $\leftarrow$ is used to indicate the operands. Thus, $e \leftarrow r$ means that a register is copied to an effective address.
  – For the MOV instruction, the source can also be an effective address and the destination a register, which will be denoted by $r \leftarrow e$.
  – Another choice is immediate data as source, and effective address as destination, which yields $e \leftarrow \#$. Immediate data in the table is indicated by the sharp sign (#).
  – The Move instructions do not move data. They make copies, meaning that the source is not modified as would happen with a true move.

• **MOV or MOVB**
  – word move MOV and the byte move MOVB
  – No more than one memory operand permitted
  – CS, EIP, and IP cannot be the destination
  – No immediate to segment moves
Example

• **MOV DS, 45**
  – Explain why it is invalid?
  – Immediate move to DS not permitted
• **MOV 25, VAR**
  – Explain why it is invalid?
  – Immediate value cannot be destination
• **MOV EIP, VAR**
  – Explain why it is invalid?
  – EIP can not be destination
• **MOV AX, BH**
  – Explain why it is invalid?
  – The sizes are missed to match
XCHG Instruction

- **XCHG**
  - Exchanges the contents of a register with the contents of an effective address.
  - For the exchange, the table uses the symbol $\leftrightarrow$. In this case, there exists a byte version as well as a word version.
  - Thus, the instruction is denoted by XCHG and the Operand field contains $r\leftrightarrow e$.
  - At least one operand must be a register.
  - No immediate operands are permitted.

- Example:
  - `XCHG AX, BX` ! Exchange 16-bit registers
  - `XCHGB AH, AL` ! Exchange 8-bit registers
  - `XCHG VAR1, BX` ! Exchange memory and register
  - `XCHG VAR1, VAR2` ! Error: two memory operands
Example

• Problem
  – Suppose AX=1, BX=2, CX=3; Write a program that rearranges their values such that AX=3; BX=1; CX=2.

• Solution
  – Step1: Exchange the values in AX and CX.
    \[\text{XCHG AX, CX}\]
  – Step2: Exchange the values in BX and CX.
    \[\text{XCHG BX, CX}\]
PUSH and POP

- **PUSH**
  - PUSH pushes its operand onto the stack.
  - The explicit operand can either be a constant (# in the Operands column) or an effective address (e in the Operands column).
  - There is also an implicit operand, SP, which is not mentioned in the instruction syntax.
  - What the instruction does is decrement SP by 2, then store the operand at the location now pointed to by SP.

- **POP**
  - POP removes an operand from the stack to an effective address.
  - What the instruction does is increment SP by 2, then store the operand at the location now pointed to by SP.

- **PUSHF and POPF**
  - They have implied operands, the push and pop the flags register, respectively.
  - This is also the case for XLAT which loads the byte register AL from the address computed from AL + BX. This instruction allows for rapid lookup in tables of size 256 bytes.
ADD and SUB Instructions

• ADD and SUB
  – ADD destination, source; Logic: destination ← destination + source
  – SUB destination, source; Logic: destination ← destination – source

• Operand Rules
  – Each of these has the same three operand combinations as MOV
  – Effective address to register, register to effective address, and constant to effective address. Thus, the Operands column of the table contains \texttt{r←e, e←r}, and \texttt{r←#}.
  – In them, the overflow flag, \texttt{O}, the sign flag, \texttt{S}, the zero flag, \texttt{Z}, and the carry flag, \texttt{C} are all set, based on the result of the instruction.
  – For example, that \texttt{O} is set if the result cannot be correctly expressed in the allowed number of bits, and cleared if it can be.
  – When the largest 16-bit number, 0x7fff (32,767 in decimal), is added to itself, the result cannot be expressed as a 16-bit signed number, so \texttt{O} is set to indicate the error.
  – For all additions and subtractions, byte versions also exist.
Example

• Problem

```assembly
.SECT .TEXT  ! --AX--
MOV  AX,1028 ! 1028
MOV  BX,4    !  4
ADD  AX,BX   ! 1032
ADDB AH,AL   ! 1036
```
MUL and DIV Instructions

• **Operand Rules**
  – Unsigned integer operands require the MUL and DIV instructions;
  – AH:AL combination is the implied destination in the byte version.
  – DX:AX combination is the implied destination in the word version.
  – Even if the result is only a word/byte, the DX/AH register is rewritten.

• **Multiplication**
  – It is always possible because the destination contains enough bits.
  – The overflow/carry bits are set if the product is over one word/byte.
  – The zero and and negative flags are undefined after a multiply.

• **Division**
  – Uses the register combinations DX:AX or AH:AL as the destination.
  – The quotient goes into AX or AL and the remainder into DX or AH.
  – All four flags, carry, overflow, zero and negative, are undefined after a divide operation.
  – If the divisor is 0, it executes a trap to stops the program unless a trap handler routine is present.
  – It is sensible to handle minus signs before and after the divide, because in the 8088 definition the sign of the remainder equals the sign of the dividend, whereas in mathematics, a remainder is always nonnegative.
MUL Instruction

- **MUL**
  - The MUL (unsigned multiply) instruction multiplies an 8-, 16-, or 32-bit operand by either AL, AX, or EAX.

- **Format**
  - MUL r/m8
  - MUL r/m16
  - MUL r/m32

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>AL</td>
<td>reg/mem8</td>
<td>AX</td>
</tr>
<tr>
<td>AX</td>
<td>reg/mem16</td>
<td>DX:AX</td>
</tr>
<tr>
<td>EAX</td>
<td>reg/mem32</td>
<td>EDX:EAX</td>
</tr>
</tbody>
</table>
MUL Example

```
.SECT .TEXT

    MOV AX, 257    ! AX = 257
    MOV BX, 4      ! BX = 4
    MOV CX, 2      ! CX = 2
    MUL BX         ! [DX:AX] = AX * BX

    MOV AX, CX     ! AX = CX
    MUL BX         ! [DX:AX] = AX * BX
```

<table>
<thead>
<tr>
<th></th>
<th>--AX--</th>
<th>--BX--</th>
<th>--CX--</th>
</tr>
</thead>
<tbody>
<tr>
<td>257</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>257</td>
<td>4</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>257</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1028</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

The Carry flag indicates whether or not the upper half of the product contains significant digits.
DIV Instruction

- **DIV**
  - The DIV (unsigned divide) instruction performs 8-bit, 16-bit, and 32-bit division on unsigned integers
  - A single operand is supplied (register or memory operand), which is assumed to be the divisor
  - Instruction formats:
    - `DIV reg/mem8`
    - `DIV reg/mem16`
    - `DIV reg/mem32`
DIV Example

.SECT .TEXT

<table>
<thead>
<tr>
<th>Instruction</th>
<th>-- AX --</th>
<th>-- BX --</th>
<th>-- CX --</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV AX, 1029</td>
<td>1029</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MOV BX, 4</td>
<td>1029</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>MOV CX, 2</td>
<td>1029</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>DIV BX</td>
<td>257</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>DIVB BL</td>
<td>320</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

DX:AX = [DX:AX]/BX, where DX is the remainder, AX is the quotient
AH:AL = [AH:AL]/BL, where AH is the remainder, AL is the quotient
Example Program 1

```assembly
.EXIT   = 1             ! 1
_PRINTF = 127           ! 2
.SECT .TEXT                     ! 3 ===== Begin of code segment =====
start:                          ! 4 ! Step 1)                       ! 5 initialize AX and BX using MOV
    MOV     AX, 514         ! 6 AX = 514
    MOV     BX, 2           ! 7 BX = 2
! 8
output1:                        ! 9 Line 4~12 is to printf("%d, %d\n", AX, BX);
    PUSH    BX              ! 10 push BX for printf, param 3
    PUSH    AX              ! 11 push AX for printf, param 2
    PUSH    pfmf2           ! 12 push format string for printf, param 1
    PUSH    _PRINTF         ! 13 push system call id of printf
    SYS                     ! 14 make system call
    ADD     SP, 4           ! 15 clean up 2 word of stack
    POP     AX              ! 16 store AX: *Because printf will return number of string in AX
    POP     BX              ! 17 store BX: This statement just for clean the stack
    PUSH    0               ! 18 Just return success(0)
    PUSH    _EXIT           ! 19 system call id exit
    SYS                     ! 20 make system call
! 21
.SECT .DATA                     ! 22 ===== Begin of data segment =====
pfmf2:                         ! 23 .ASCIZ "%d, %d\n"               ! 24 Define format string for print two 16bit integer
.SECT .BSS                      ! 25 ===== Begin of bss segment =====
```
Example Program 2

```
_EXIT = 1           ! 1
_PRINTF = 127      ! 2
.SECT .TEXT        ! 3 ===== Begin of code segment =====
start:
  MOV     AX, 513   ! 4 initialize AX = 514
  MOV     BX, 1     ! 5 initialize BX = 2
  ADDB    BH, BL    ! 6 8bit add: BH = BH + BL
  SUBB    AH, AL    ! 7 8bit add: AH = AH - AL
  PUSH    BX        ! 8 Line 18~24 is to printf("%d, %d\n", AX, BX);
  PUSH    AX        ! 9 push AX for printf
  PUSH    pfmf2     ! 10 push BX for printf, param 3
  PUSH    _PRINTF   ! 11 push system call id of printf
  SYS          ! 12 push system call id of printf
  ADDSP      ! 13 make system call
  POPAX      ! 14 clean up 2 word of stack
  POPBX      ! 15 If writing lab2 in one long program,
  POPBX      ! 16 should restore AX and BX for later use
exit:
  PUSH    0        ! 17 Line 18-20: exit(0)
  PUSH    _EXIT    ! 18 Just return success(0)
  SYS          ! 19 system call id exit
  ADDSP      ! 20 make system call
  ! 21
.SECT .DATA        ! 22 ===== Begin of data segment =====
pfmf2:              ! 23
  .ASCIZ "%d, %d\n" ! 24 Define format string for print two 16bit integer
.SECT .BSS         ! 26 ===== Begin of bss segment =====
```