CS47 - Lecture 05

Kaushik Patra
(kaushik.patra@sjsu.edu)

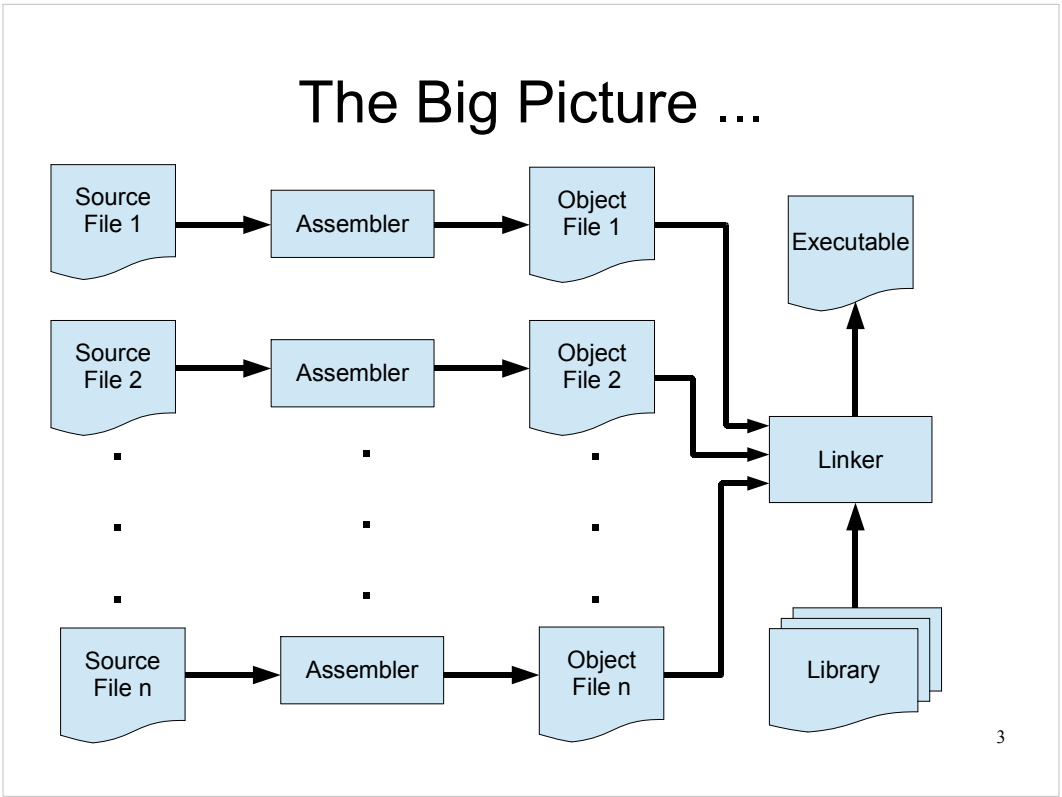
1

- Topics
  - Executable Generation
  - Assembler
  - Linker & Loader

[Chapter A.2-A.4 of Computer Organization & Design by Patterson]

Executable Generation …

2



- Application program is usually written in modular way – i.e. an application is partitioned into multiple interacting modules. Each module is usually responsible for one specific task. Modules are implemented in separate source files.

- Assuming development environment is in assembly language, each module is written in separate assembly source file. Each source file is independently assembled and corresponding machine codes are generated in an object file. However, since one source file can refer to labels in other source file, at this level all the labels may not be resolved – therefore the generated machine code is not complete.

- Linker is a tool which stitches all the partially assembled objects into a completely assembled executable. This tool has access to all the cross referenced labels, therefore all the references should be resolved at this stage of executable generation.

- System usually provides bunch of common routines (e.g. print or scan) , that an application uses, into already assembled object code collection. These collection is known as library. Linker can find out specific routine used from the library and integrate it into the final executable.

Intermediate file …

- Assembler produces single object file from single assembly source code.
- Object file contains
  - Machine instructions
  - Book keeping information to stitch several modules.
- Source code can contain reference to a label in a separate independent module.
  - These cross references are unresolved at object file.

4

- Assembler always deals with single assembly source file and produces a single object file containing machine code and information about instructions using unknown label. Linker uses this information to patch the machine code using label defined in another source file.
- Since the assembler does not have any idea of any label outside the scope of the current program, the machine code it generates is not self sufficient and may not be executed on a processor. This intermediate file is called object file and usually stored in a file with '.o' extension (e.g. my_example.o).

Linking objects ...

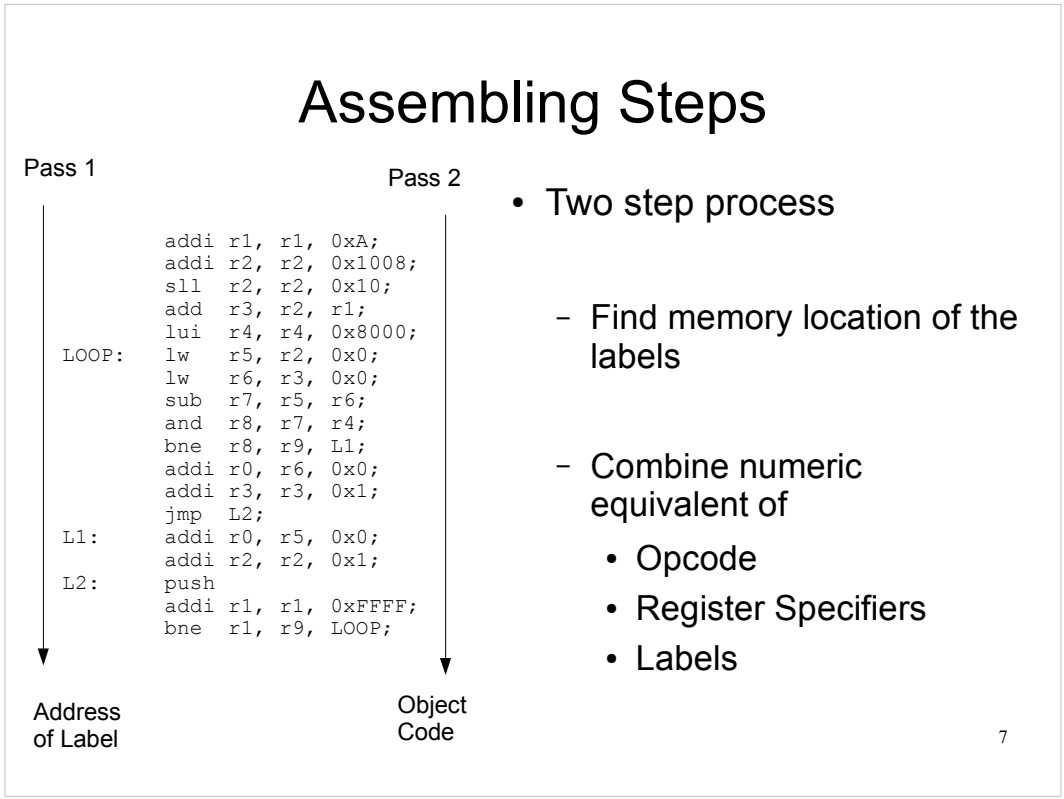
- Object code with unresolved reference can not be executed directly.
- Linker takes multiple modules, resolve the cross references and write machine codes with all the references resolved.
  - The outcome is an executable which can be executed on a machine.
  - Linker also takes pre-assembled object codes providing common functionality as procedure call (subroutines).

5

- Linker combines several object files into one single executable. It resolves references to labels that assembler could not resolve. It also has ability to search through system's library to search for any label that is defined within the scope of the library. If a program uses labels, defined in the library, linker has ability to extract out that part of code from the library and include it in the final executable.

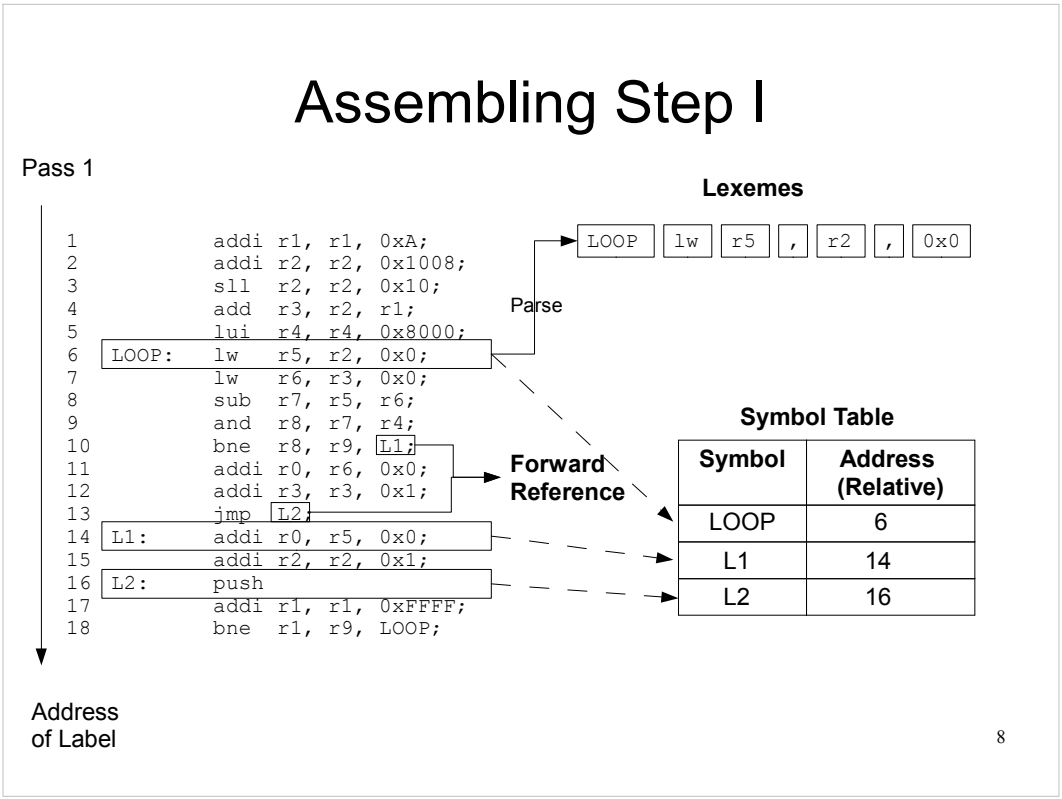
Assembler …

6



- Assembling a source code is usually two step process. Since assembler supports naming (or labeling) of address (either to identify data location or code location) and refer them in the code, it has to know location / address corresponding to the name precisely, before it can use that information to generate machine code. For example in the above code there is a label 'LOOP' which corresponds to a code location which has instruction 'lw r5, r2, 0x0'. The last line of the code uses that name 'LOOP' to jump back to that instruction. To generate code for the last line, assembler needs to know address value represented by the name 'LOOP'.

- The very first pass assembler collects all the label information and in second pass it generates the actual machine code.



- For given processor, the start address for code and data is fixed (later we'll study this in details in memory layout). Usually code and data segment starts at different address.

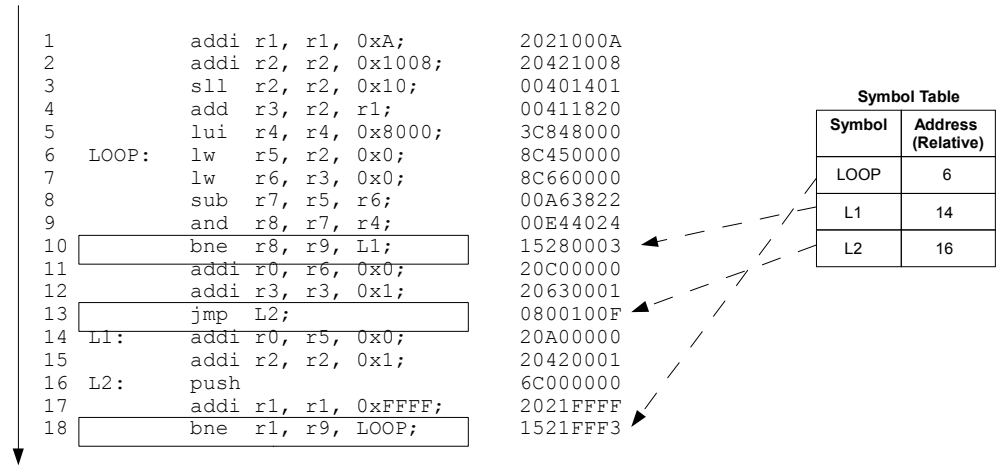
- During first pass, assembler reads the source code line be line and parse them. Parsing step involves breaking up a code line into individual words and symbol. Each such part is called a lexeme. If a code line, for example, starts with a label (identified if the first word has ':' as separator from rest of the sentence) the label is name is used a key of a table (called symbol table) entry with the corresponding relative address as value for that key. For example for the label 'LOOP' address value '6' is inserted into the symbol table.

- The main artifact (or output) of the first pass of an assembler is the symbol table. Additionally it can check for any syntax error in this phase.

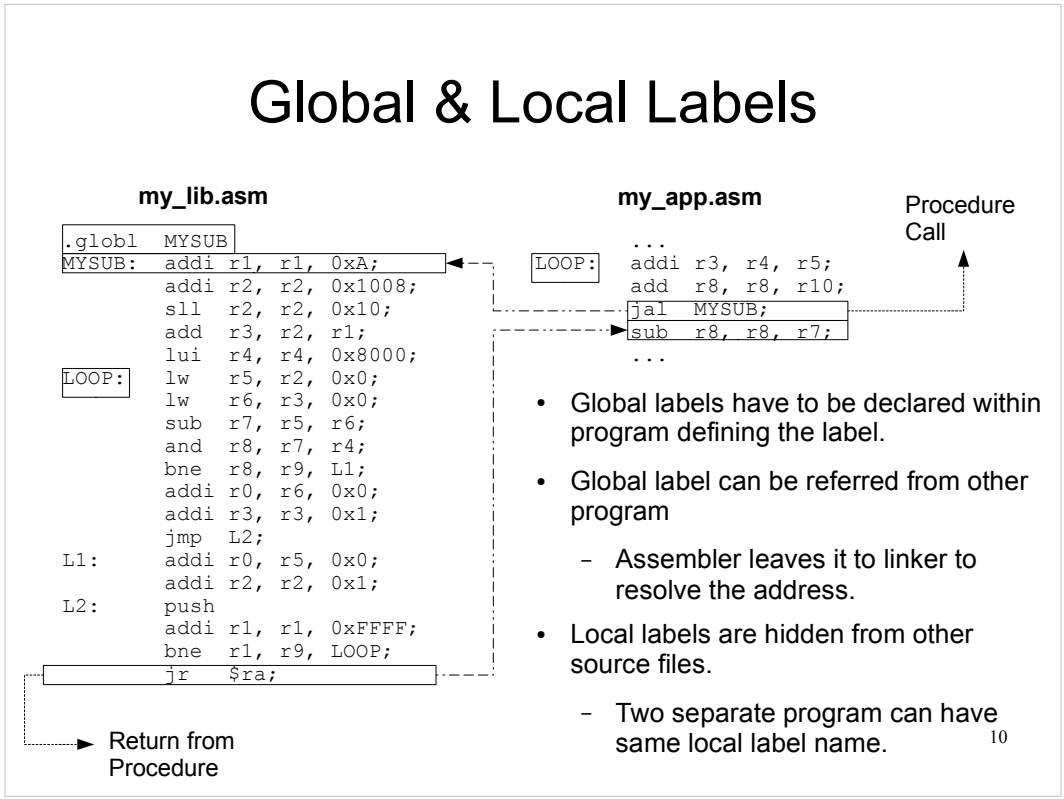
- Assembler needs to have 2 pass because of forward reference to a label. It is allowed in assembly language to use a label name before it is defined. Therefore, single pass is not sufficient to resolve all the references to labels.

Assembling Step II

Pass 2



- The 2nd pass is straight forward for the assembler. It scans through the code from the beginning and put together the machine code (the process is discussed in earlier lecture) and write in the object file. When it encounters reference to a label, it search for the corresponding address value from the symbol table and substitute the label reference with the address value before it combines parts into a machine code.



- Two type of labels are there in assembly language – local and global label. The global labels sometime also refer as 'external label'.

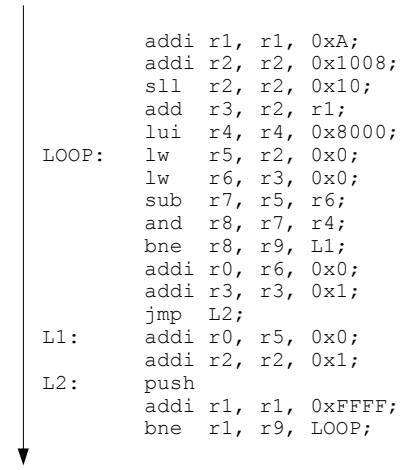
- Assembler supports feature of referring to a label defined in a separate source code. This type of labels are called external label or global label. Though assembler can not resolve the address for a referred global label defined outside the source file, it does not complain about it. If it find any label that is not in symbol table, it does not produce error assuming that these missing labels are defined outside the source code in a separate source file. It relies that linker will resolve the label during executable generation.

- The local labels are hidden across the source files. This means that any local label is not required to be unique across all the source file. Same local label name (for example 'LOOP' is used in both the source file in the above example) can be used in multiple source file as needed. However, global label name must be unique across multiple source file, otherwise linker will have name conflict error.

- Global label must be declared as global at the source code where it is defined.

One Pass Assembler

Pass 1



```
        addi r1, r1, 0xA;
        addi r2, r2, 0x1008;
        sll  r2, r2, 0x10;
        add  r3, r2, r1;
        lui  r4, r4, 0x8000;
LOOP:   lw   r5, r2, 0x0;
        lw   r6, r3, 0x0;
        sub  r7, r5, r6;
        and  r8, r7, r4;
        bne  r8, r9, L1;
        addi r0, r6, 0x0;
        addi r3, r3, 0x1;
        jmp  L2;
L1:     addi r0, r5, 0x0;
        addi r2, r2, 0x1;
L2:     push
        addi r1, r1, 0xFFFF;
        bne  r1, r9, LOOP;
```

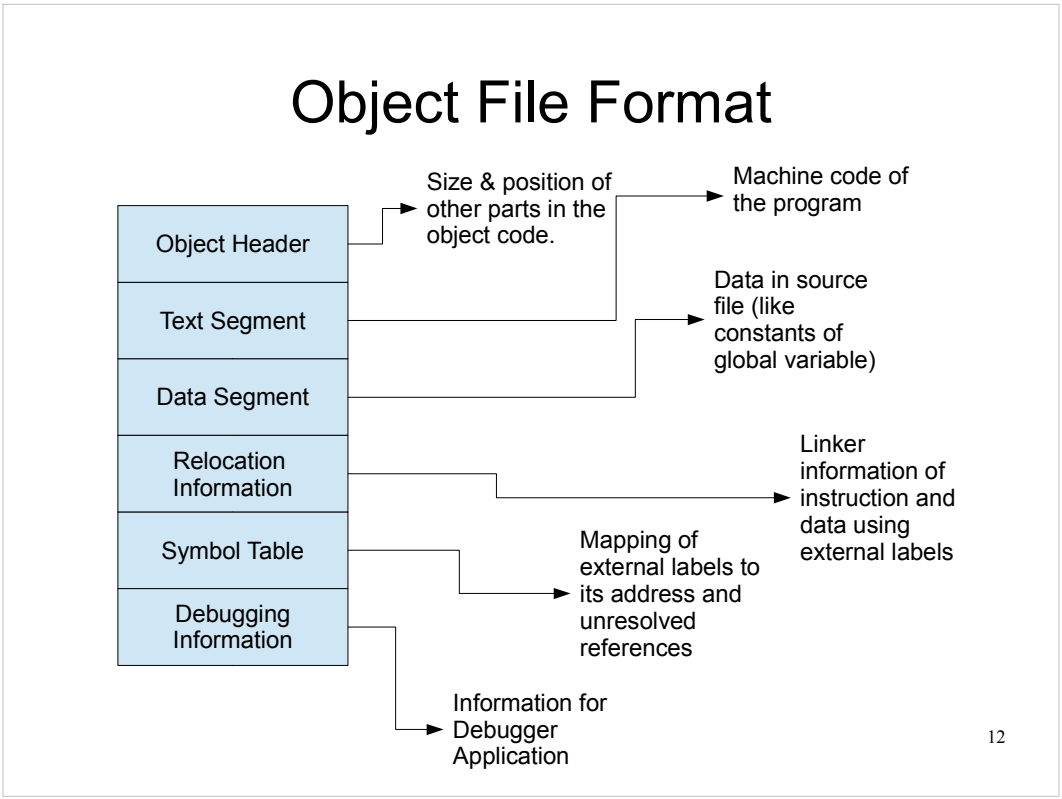
- Used to make fast object code generation.
  - Leave the forward reference to be patched later.
- Need to hold entire object code into memory.
- Impose challenge for machine with variable length format.

Partial Code Generation,,
Patch back forward reference.

11

- Once pass assembler can be implemented by doing a lazy code generation for the places where label reference is used. In this process as the source code is parsed line by line, and symbol table is inserted with (label, address) pair if a label is defined. However, at the same time the assembler tries to generate machine code as well. If there is no reference to a label, the code conversion is complete. Now, if a line refers to a label, the symbol table is consulted and any backward reference is resolved right at the time of code generation. However, in the case of forward reference the label will not be found in the symbol table. In that case partial code is generated replacing the label with some dummy value and the code is marked to be patched later. When a new label is encountered, incomplete forward references are reviewed and patched in-place if required.

- Since some machine code generated needs to be patched / fixed later, machine code can not be dumped into the target object file as soon as the conversion is done. The entire machine code lists needs to be in the memory till end for any pending patch process. This increase the memory requirement for one pass assembler. However, since the target source code is scanned only once, it is faster in execution.



- The object header defines the size and position of other parts in the rest of the object file. This information is used by linker to generate final executable.

- The text segment contains the machine language code corresponding to the source assembly code. This code may not be executed on the target processor due to unresolved references.

- The data segment contains the binary representation of the data in the source file. This portion also may contain unresolved references.

- The relocation information identifies instruction and data words that depends on the absolute addresses. These references needs to be modified (relocated) if the portion of the program is moved in memory.

- The symbol table associates addresses with the external (global) labels that the defined in the source file and the lists of unresolved references.

- The debugging section contains information for debugger to map a specific machine code or data to specific line in the source file.

Additional Facilities

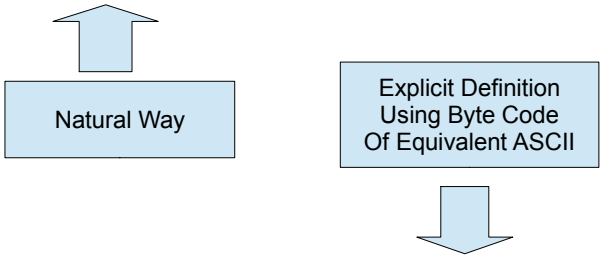
- Data layout directives
  - Provides easier way to describe data
- Macros
  - Provides way to shorten code writing for repetitive codes.
- Pseudo instructions
  - Combines two or more fundamental instructions into one single instruction.

13

- Usually assembler provides many additional facilities to ease the assembly code writing. Three most commonly used and important features are data layout directives, macros and pseudo instruction.
- Frequently an application program deals with constants and predefined global variables. These information are stored into specific location (to be discussed later) and can be defined as a part of the program. The most natural form of this directive from computer perspective is to define a byte (directive .byte) value at storage location. Assembler provides other types of storage directive as well, like for half word, word, double word etc.
- Macro is a mechanism to enable reuse of commonly occurring code sequence in a program.
- Pseudo instruction is almost like macro, but user do not explicitly define it. These instructions look alike to the native instruction that the processor supports – but assembler breaks the operation down to use native instruction when generating assembly code.

Data Layout for String

```
my_str:
        .asciiz "The sum from 0 .. 100 is %d\n"
```




```
my_str:
        .byte  84, 104, 101,  32, 115, 117, 109, 32
        .byte 102, 114, 111, 109,  32,  48,  32, 46
        .byte  46,  32,  49,  48,  32, 115
        .byte  32,  37, 100,   0
```

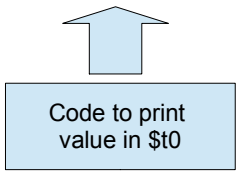
14

- The most useful data layout directive is the support for string definition. Assembler supports a directive '.asciiz' / '.ascii' to define string in human natural way (i.e. to define a string within quote using words). The actual storage of string in the memory is always in terms of ascii code (usually byte long). Assembler converts the defined string into its equivalent multi-byte storage.

- Programmer can always define the string in terms of bytes in the memory, but that is very tedious job. Ability to write a string in sentence using words in human language is far more effective than defining string using byte values.

Macros

```
.data
int_str: .asciiz "%d"

.text
        la    $a0, int_str
        mov   $a1, $t0
        jal   printf
```



- Pattern matching and replacement facility
- Provides mechanism to name frequently used code sequence.
- How to print numbers in registers $t0 to $t7?

15

- Macro facility is the most important that assembler provides to programmers. It is a string patter matching (to match a name) to replace the name with a known sequence of code. In that sense it provides a mechanism to give a name to a known sequence of code to accomplish certain end goal. Once the 'name' is defined it can be used directly in place of sequence of code – thus reduces the size of code and programmers effort to write code. This is also can be thought as code reuse for commonly used codes.

Macros

```
.data
int_str: .asciiz "%d"

.text
         la    $a0, int_str
         mov   $a1, $t0
         jal   printf

         mov   $a1, $t1
         jal   printf
         mov   $a1, $t2
         jal   printf
         mov   $a1, $t3
         jal   printf

            ...

         mov   $a1, $t7
         jal   printf
```

- One way is to repeat the code for printing $t0.

16

- Without the macro facility in assembler, programmer must repeat the code to print integer from the register value. This example shows how the code is replicated to implemented printing of register values from $t0 to $t7.

Macros

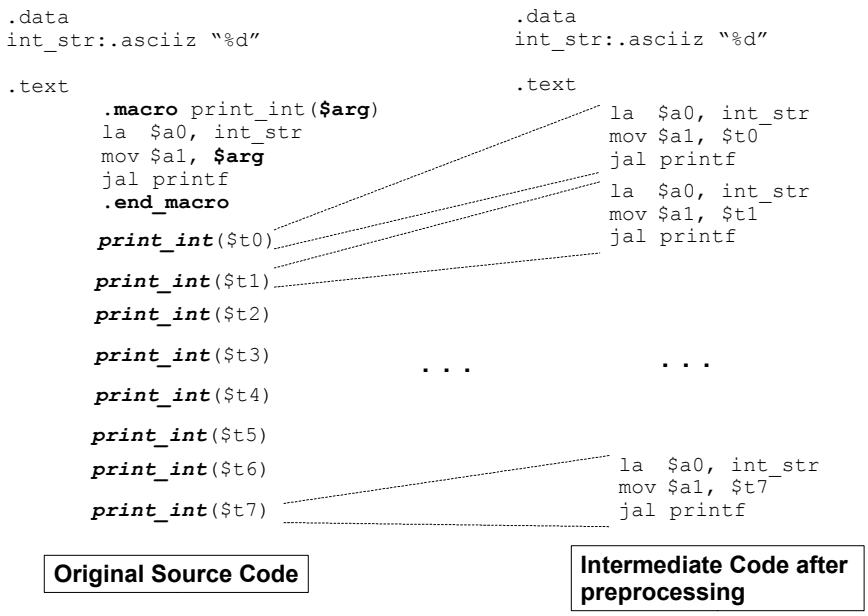
```
.data
int_str: .asciiz "%d"

.text
        .macro print_int($arg)
        la    $a0, int_str
        mov   $a1, $arg
        jal   printf
        .end_macro
        print_int($t0)
        print_int($t1)
        print_int($t2)
        print_int($t3)
        print_int($t4)
        print_int($t5)
        print_int($t6)
        print_int($t7)
```

- More elegant way is to
  - define a macro
  - use it.

17

- Macro facility in assembler provides programmer a way to save typing of repetitive code. In this example, the print_int macro is define with an argument $arg within scope of .macro … .end_macro pair. The the macro is called with argument value of $t0 to $t7. $arg in the macro definition in each calling will be replaced by the value passed, thus the same macro will be able to print the value of the register passed as macro argument.

Macros – Preprocessing Step



18

- All the assembler supporting macro does perform a preprocessing step on the source code, where it collects the macro definition and expand the macro call to the exact code corresponding to the macro definition. This expanded intermediate code is then goes into real assembling process. Therefore in a sense, the macro preprocessing is performing the code replication process that a programmer would otherwise need to do.

Pseudo Instruction

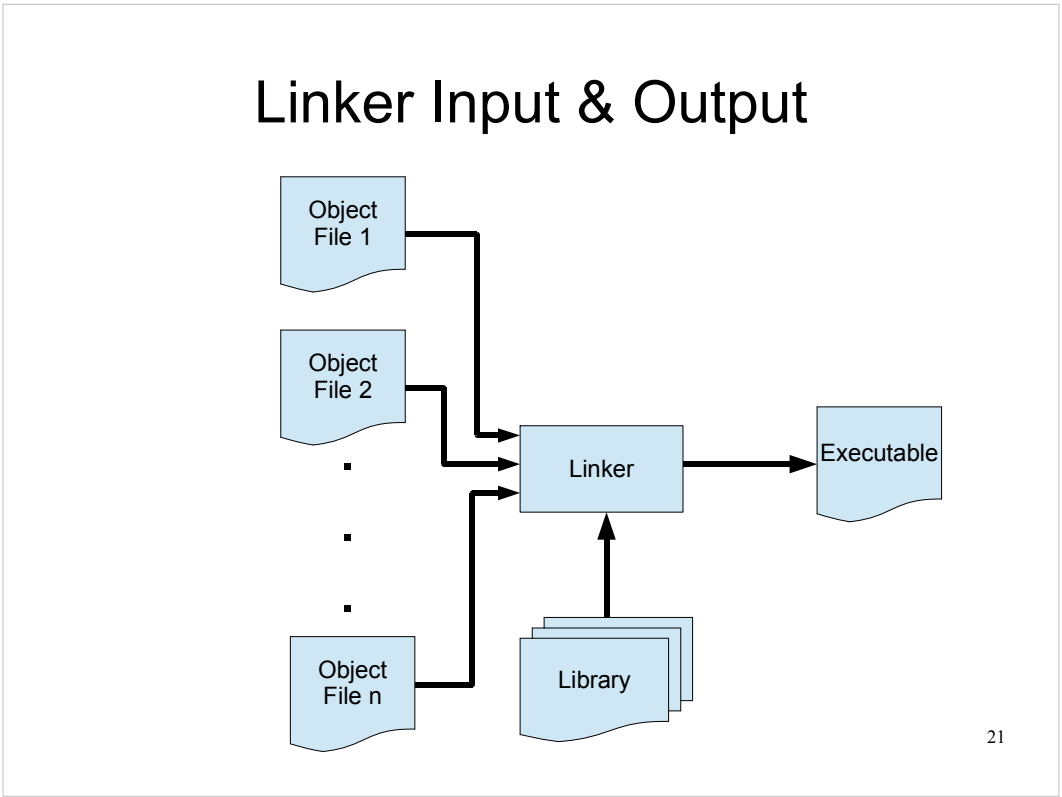
- Pseudo instructions are
  - Not implemented on processor
  - Can be written using couple of native instructions\
  - Assembler automatically replace those pseudo instructions into combination of native instructions.
- Example : blt $8, $9, L1; #if (R[$8] < R[$9]) goto L1
  - slt $1, $8, $9; # R[$1] = (R[$8] < R[$9])?1:0;
  - bne $1, $0, L1; # if (R[$1] != R[$0]) goto L1

19

- Pseudo instruction is the instruction that the processor does not implement (or understand meaning they do not have a corresponding machine opcode) but assembler supports to perform some common operation in program. For example blt (stands for 'branch less than') is very common operation in programming, but MIPS does not support it. However, this command can be expressed using two commands (slt – 'set less than' and bne – 'branch on not equal') that the processor supports. Assembler usually supports instruction like 'blt' and convert it to call to 'slt' and 'bne' combination.
- We can think pseudo instruction as macros – but the definition of these macros are internal to the assembler and can not be defined within a program.

Linker & Loader …

20



- A linker combines multiple object codes and part of library files (which are used within the application object codes) into an executable file. This executable file is a binary file and can be loaded into the memory to execute.

Linker Tasks

- Search the libraries to find out library routine used by the program.
- Determine memory locations of each module.
- Resolve the references among files.

22

- A linker's first task is to ensure that a program does not contain any undefined labels. It builds table of all the defined labels in different object codes and tries to resolve the references. If the label is still missing, it searches through the library files. If the label is found in the library file, the corresponding code is extracted from the library and combined with other object files to produce the final executable.
- Once all the labels are known, linker determines the memory location of each module. This determination helps linker to finalize the address of each label. Assembler creates the addresses relative within the source file. Linker places objects in a sequence, therefore the absolute address is of the target module is known to linker.
- Once absolute addresses are known to linker, it modifies the code with unresolved reference in each module and combine them into an executable file (binary format).

Loader

- Operating system's tool loader loads executable into memory and starts the program to execute on system.
  - Determine size of text and data segment
  - Creates new address space for the program
  - Copies instruction and data from executable file to memory.
  - Copies arguments to be passed to the program onto the stack
  - Initialize machine registers – stack pointer to point to free stack location.
  - Jumps to the startup routine that copies arguments from stack to registers and call the 'main' routine.

23

- This part is to be discussed more in Operating System course.
- OS has a tool names loader which loads the executable into memory and transfer the program execution by calling a known subroutine of the target application (it is usually 'main(...)').

CS47 - Lecture 05

Kaushik Patra
(kaushik.patra@sjsu.edu)

24

- Topics
  - Executable Generation
  - Assembler
  - Linker & Loader

[Chapter A.2-A.4 of Computer Organization & Design by Patterson]