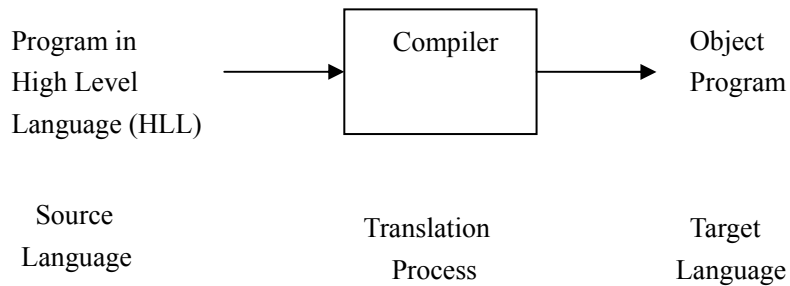


Lecture 14

Assemblers & the Assembly Process

Assembler

- We know what a compiler does.



- An assembler is also a language translator.
- When the source language is essentially a symbolic representation of machine language, it is called an assembly language.
- A translator for assembly language (into machine language) is called an assembler.
- It is easier to write a program in assembly language than writing it in binary or octal or hex.

We use symbolic names for opcodes, addresses, *etc.*

- The assembler changes these symbolic names to their numerical values.
- At the assembly language level we have access to most features of the machine. Example: Testing an overflow bit. We cannot do this in Java.
- Another important property is that assembly language programs can run on only one family of machines.
- If a compiler is available, HLL program can run on different machines.

Why do we program in assembly?

- We know it is difficult.
- It takes much longer to write.
- It is difficult to debug and maintain.

There are two reasons we still use assembly language:

(i) Performance (ii) Access to the machine

- Assembly language code is much smaller in size, and it is much faster than compiler-generated code.

Example: Testing odd or even number.

Assembly language will allow us to test a bit & decide.

In Java or C,

$$\text{if } (i == (i / 2) * 2) \rightarrow \text{even}$$

Here division and multiplication will require much more time.

- Small size and speed make assembly language ideal for embedded applications, where we may not have a lot of memory.

Example: spacecraft, cell phones.

- Since it enables access to the machine, low level OS functions such as interrupts and the trap handler are written in assembly language.

Also device controllers / drivers.

- Frequently executed parts of a large program are sometimes written in assembly language.

It is linked to other parts later which are written in HLL.

But we lose portability.

- Understanding the assembly process is a must for a compiler writer.

Some compilers generate assembly code. The assembler then converts this to machine language.

- Finally students in CSE 220 need to know assembly language. It is a good vehicle for explaining and understanding CPU architecture.

- Our objective here is not to become expert assembly language programmers.

We want to learn all the basic concepts and ideas.

In this process, we want to learn computer organization.

Assembly Process

- We know that an assembler converts symbolic names for opcodes, registers, and locations to their numerical values.
- This is done in two steps.
 - The first step is to map labels to memory locations.
 - The second step is to translate each assembly statement into its equivalent machine instruction.
- The assembler produces an object file (stored on disk) which contain machine instructions, data and some book-keeping information.
- The assembler needs to handle what is known as a ‘forward reference’.

- Unlike HLL assembly labels may be used, even before they are defined.

Example: `la str` address of `str` is not known
 `str: .asciiz "Hello"`

If we have the data segment before the text segment then we may not have this problem, but now about a forward branch?

- But now about a forward branch?

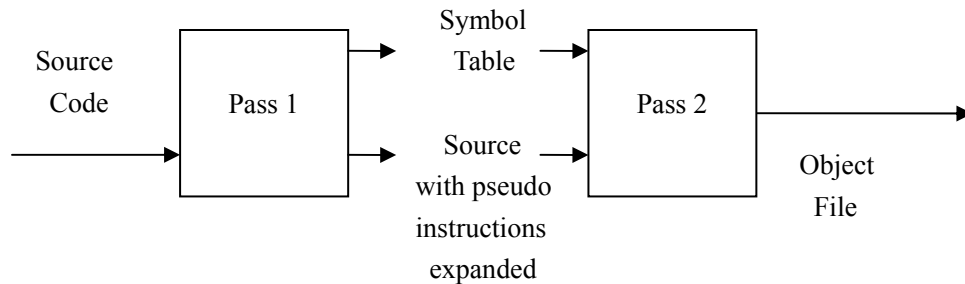
Example: `bge $t1, $t2, great`
 `great: add...`

When we are processing `bge`, address of label `great` is not known.

How do we generate machine code for `bge`?

- Forward reference problem is handled by making two passes over assembly code.
 - It uses an Instruction Location Counter (ILC). The ILC is loaded with the start address of assembly program.
 - The assembler then processes assembly language statements (instructions), one by one.
 - If a machine has variable length instructions, then for every opcode the assembler knows the precise size of that instruction.
 - It then updates ILC: $ILC \leftarrow ILC + \text{size of instruction}$.
 - In the case of MIPS, all instructions are 1 word (4 bytes) long.

- When the assembler encounters a label, it checks the symbol table, which is a reference table for all labels.
- If a label is not there, then assembler adds it to the table with the value of ILC as its address.
- This action is taken only if a line begins with a label (not when it sees a label as a target in branch or jump instruction).



- Remember, pseudo-instructions must be mapped to machine instructions.
- Pass 2 is similar to Pass 1, except that all labels and variables now get their addresses assigned as this information is now available in the symbol table.
- Using the symbol table, the assembler replaces labels in branch & jump instructions by proper offsets.
- It also assigns numerical values to opcodes, registers, and immediate operands.
- By processing the source code line by line, it generates the binary file (object file) and writes it to a disk.
- If an address reference is not resolved at the end of this, then it is reported.

Symbol Table

Symbol	Address	Other Info
main	400020	
done		
str	10010000	

- Other info: Program label or data, length of data field, local/global, relocation information used by linker.
 - Pass 1: All labels get their addresses.
 - Pass 2: Calculate offsets.
 - $\text{Offset} = (\text{Address of the target} - \text{Address of current instruction} - 4)/4$
 - Relative to (ILC + 4) (all conditional branches)

- Example:

```
        bne    $t0, $t3, con
        add    $t2, $t2, 1
con:    add    $t1, $t1, 1
```

 - How do we calculate address offset for bne instruction?
 - Suppose that the address of the bne instruction is 400040 in hex.
 - Address of con is 400048 in hex from the symbol table.
 - Difference is 8 in hex. Now subtract 4, we get 4 (in hex).
 - In decimal this is also 4. Now divide by 4 to get (decimal) answer to word offset.
 - It is 1.

- For backward branches, offsets are negative. They can be calculated in the first pass. Why?

- For direct jumps, we more or less have the actual address stored in the instruction (unless we exceed the 256MB boundary).

- From the symbol table, get the address and insert it in the instruction, while generating machine code.

- We have pseudo-instructions `la <var>`, `lw <var>` which refer to a variable in an instruction.
 - We know variables are stored from 0x1001 0000 (data).
 - Pseudo- instructions are expanded using their table.
 - Once all data items get their addresses, we just need the offsets from 0x1001 0000.
 - In the second pass, these offsets are inserted in their proper places, which were left blank in the first pass.
 - Why do we use a table to expand pseudo-instructions?
 - We know exactly how many pseudo-instructions are there.
 - Each one is expanded exactly the same way every the time (except for the individual offsets).
 - Table look-up is faster.

- How to handle 32-bit wide numbers or addresses?
 - Small constants can be easily loaded using `li` (load immediate) or added using `addi`
 - `li` is a pseudo-instruction. It is assembled as `ori`.

```

li $t0,10      ← here 10 is a small constant
  ↓
ori $8, $0, 10 ← bit pattern for value 10 in immediate field

```

- Large constants (32 bits) are split into two, upper half and lower half.
- How do we load the following?

```

0000 0000 0011 1101   0000 1001 0000 0000
      upper           lower

```

- The upper half has value 61 in decimal or 3D in hex.
The lower half has value 2304 in decimal.
- We want this large value in `$s0`
- We use `lui` instruction (load upper immediate).

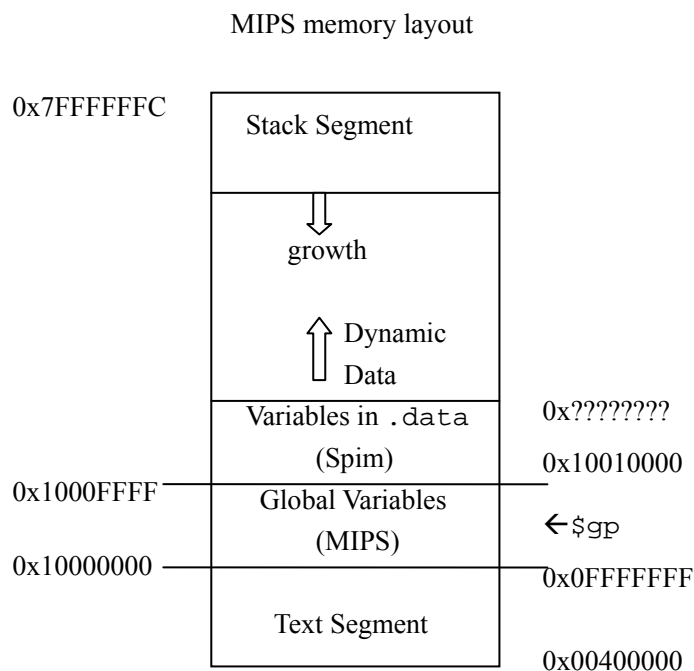
```

lui $s0,61
addi $s0,$s0,2304

```

- After the second instruction, `$s0` will have the correct value.
- When we load to upper half of a register, lower half is all zeros.

- Assembler or compiler must break constants into two.
- Assembler may use `$at` for this or similar situations.
- `la` (load address) is also assembled using `lui`.
 - `la $t0, str` → here `str` is a variable at address `0x10010000`
 - `0x1001` is 4097 in decimal. [upper].
 - `lui $1, 4097` uses `$1 [$at]`
 - `ori $8, $1, 0`
 - Now `$8` (*i.e.* `$t0`) will have address of variable `str`.
 - `str` is declared in data part.
 - Why is its address `0x10010000`?
 - Data segment starts at `0x10000000` in MIPS.
 - `0x10000000` to `0x1000FFFF` are reserved for global variables (variables defined in `.data`).
 - In the Spim simulator, however, we start at **`0x10010000`** (not `0x10000000`).



- See Textbook, Appendix B, page B-21 and read how global variables are accessed using `$gp`, which points to the middle of the global area.

- How to assemble the indexed addressing mode of a MIPS assembly language instruction?

Example: `lw $t0, table($t1)`

- Suppose `table` is first variable / (label) declared in `.data`.
- It begins at `0x10010000`. Why?
- Assembler uses register `$at` to form an address.
- It generates


```
lui $at, 4097          #load upper immediate 1001 in hex
addu $at, $at, $t1    #at ← at + t1
lw $t0, ($at)
```

If `table` has address `0x10010072`, how would the assembler proceed?

Additional facilities

- There are several data layout directives provided by assembler.
 - These just help the programmer to specify data in a more concise and natural way (and not in binary).
 - `.byte`, `.word`, `.space`, `.ascii`
 - When these are read, the assembler takes appropriate action.
- **Macros:** A 'macro' is a pattern-matching and replacement facility.
 - It helps us avoid writing the same sequence of code at multiple places in our program.
 - macro definition gives a name to a piece of code.
 - Whenever we need to use the same code, we invoke the macro.
 - During assembly, assembler replaces macro call by the actual code.
 - We can have arguments in a macro.
 - It is somewhat similar to a function, but not exactly.

- Structure of a macro:

```
macro <name> {arguments}
```

```
    ( Body of  
      macro )
```

```
end_macro    ← end of definition.
```

- A macro can usually occur anywhere in the code.
- But it must be declared or defined before its first use.
- Every instance of a macro call is replaced by its body.
- If a macro has formal arguments then they are replaced by the actual parameters during expansion.
- Macro expansion is a textual substitution, somewhat similar to that done by an editor.

- Example: `.text`

```

...
Prints integer in P1    .macro print (P1)
                        move $a0, P1
                        li $v0,1
                        syscall
                        .end_macro
```

Use as follows:

```

..... code that calculates
        an integer in $s1
call macro print ($s1)  → after expansion and parameter substitution {
                        move $a0, $s1
                        li $v0,1
                        syscall
```

- Here P1 has to be a register.
- Programmer has to know this. We can also use this macro with \$v0 as a parameter.
- Macro does not use call-return mechanism.
 - In the case of functions there is no code expansion and we do need call-return mechanism.
- Assembler maintains a table of macro-definitions.

- Whenever it sees a macro, it is installed in this table.
 - When it sees a macro-call, it expands it using the macro definition.
 - That is why the macro has to be declared first.
 - This all is done in Pass 1.
- There are no macros in SPIM.