CS31001 COMPUTER ORGANIZATION AND ARCHITECTURE

Debdeep Mukhopadhyay,
CSE, IIT Kharagpur

Assembler

Steps in Transforming an Assembly Language

- Assembly languages have been developed that allow the use of symbolic names for instructions and memory locations.
- Assemblers convert instruction sequences in assembly languages to machine language.
- Assemblers accept numbers in a variety of simple and natural representations, and automatically convert them to required machine formats.
- Further, assemblers allow pseudo-instructions and macros.

Linkers and Loaders

- Multiple program modules are assembled independently.
- They, along with the library routines are linked subsequently by the linker.
- The linked routine forms a complete executable program which is then loaded to the memory.

Simulators

- Instead of loading the machine language instructions, they are often interpreted by a simulator.
- It examines each instructions, and carries out its functions by updating variables and data structures, that correspond to registers, and other machine parts.
- These simulators are needed in the design phase.

The assembly process

- Two passes exist.
- First pass: main function is to construct a symbol table.
- A symbol is a string of characters that is used as an instruction label.
- As instructions are read, the assembler maintains an instruction location counter that determines the relative position of the instruction.
- It is assumed that the program is loaded in a memory with address 0.
- There is also an additional relocation information produced by the assembler, which is used by the loader according to the eventual location in memory.

Example

- check: beq $t0,$t1,loop
- The assembler detects the operation symbol, “beq”
- The register symbols, $t0 and $s0 are also read.
- The labels “check” and “loop” are also read.
- However if “loop” refers to an backward memory location, it already exists in the symbol table.
- Else, we put it in the symbol table with its location blank.
- We solve these missing locations in a second pass.

Assembler Directives

- Provides the assembler with information on how to translate the program but does not themselves lead to the generation of machine instructions.
  - may specify the layout of data in the program’s data segment
  - define variables with symbolic names and desired initial values
- Assembler reads this directives and takes them while executing the rest of the programs.

Some examples

```
,macro                    #start macro
.end_macro                #end macro
.text                     #start program's text segment
…
.data                     #start program's data segment
tiny: .byte 156, 0x7a     #name and initialize data byte(s)
max: .word 1000000        #name and initialize data words(s)
small: float 2E-3         #name short float
big: .double 2E-3         #name long float
     .align 2             #align next item on word boundary
array: .space 600         #reserve space for 150 words
str1: .ascii  "a*b"       #name and initialize ASCII string
str2: .asciiz "xyz"       #null-terminated ASCII string
     .global main         #consider main as a global name
```

The .byte directive

- tiny .byte 156 0x7a
  - a byte holding 156 followed by a second byte containing 0x7a
  - tiny($s0), refer to the first byte if $s0 is 0, second byte if $s0 is 1 and so on.
- The directives .word, .float and .double are similar, except that they define words, short float, and long float respectively
- .float f1,..., fn Store the *n* floating-point single precision numbers (32 bits) in successive memory locations.
- double is for double precision (64 bits)

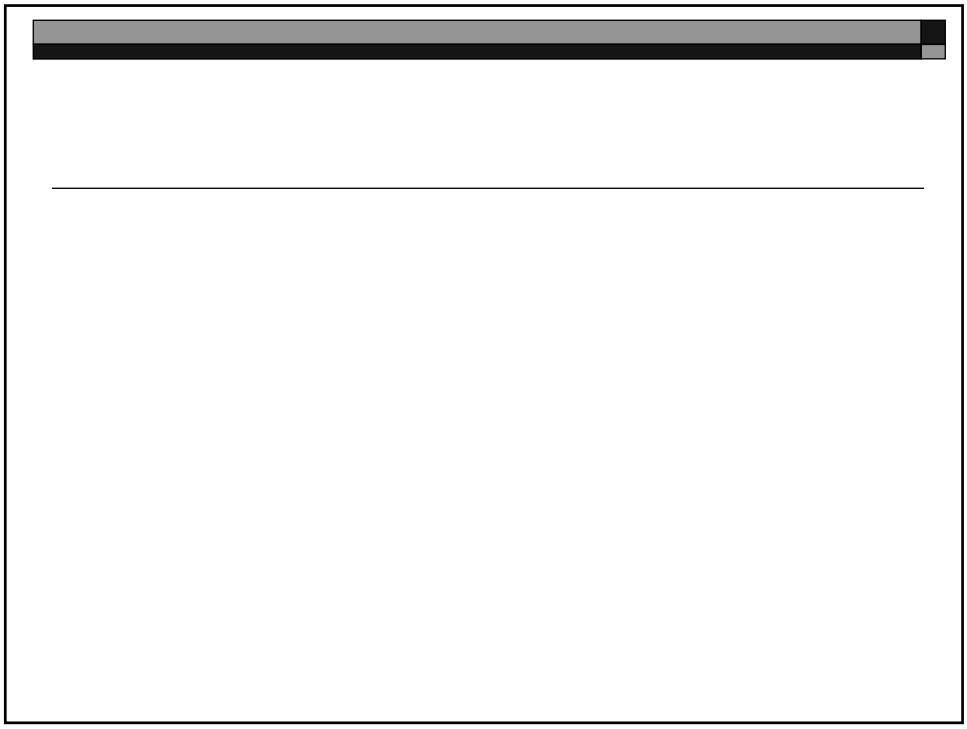
Pseudo-instructions

- Pseudo-instructions allow us to formulate computations and decisions in alternative forms not directly supported by hardware.
- The assembler takes care of translating these to basic hardware supported instructions.
- Example: MIPS lacks a logical NOT instruction.
  - same effect can be achieved by **nor $s0,$s0,$zero**

The abs instruction

- Some pseudo-instructions need more than one instruction.
- abs $t0,$s0 #put |$s0| into $t0
- The assembler translates this into:

  add $t0,$s0,$zero
  slt $at,$t0,$zero
  beq $at,$zero,+4
  sub $t0,$zero,$s0



Some conversions

- neg $t0,$s0: sub $t0,$zero,$s0
- rem $t0,$s0,$s1: div $s0,$s1
  mfhi $t0
- li $t0,imm:
  - addi $t0,$zero,imm #if imm fits in 16 bits
  - lui $t0,upperhalf #if imm needs 32 bits
    ori $t0,lowerhalf
- blt $s0,$s1,label
  - slt $at,$s0,$s1
  - bne $at,$zero,label

Macro-instructions

- A mechanism to give a name to an often used sequence of instructions (helps like a short form).
- .macro(arg list)

  …

  .end_macro

Two important points

- How is a macro different from a pseudo-instruction?
  - Pseudo-instructions are a part of the assembler design. They are fixed for a user. On the other hand, macros are user defined.
  - Further, a pseudo-instruction looks exactly like an instruction. We can say by looking that move/not/li/la are pseudo-instructions. But a macro is more like a high level language code.

The other question.

- How is a macro different from a procedure?
  - A procedure execution takes place by at least two jump instructions (jal and jr).
  - A macro is just a short hand for several lines of assembly.
  - The macro is replaced by the assembler with the equivalent lines of code for each time the macro is called.
  - After that there is no trace of macro in the final assembly.

Example

- Determine the largest of three values in registers and put the result in a fourth register. Write a macro for this.

```
.macro max3reg(m,a1,a2,a3)
    move m,a1
    bge m,a2,+4
    move m,a2
    bge m,a3,+4
    move m,a3
 .end_macro
```

When the macro is used like

max3reg($t0,$s0,$s4,$s3), the assembler simply replaces the arguments m,a1,a2, and a3 in the text of the macro with these registers.