MIPS Assembly Fundamentals

Some material taken from Assembly Language for x86 Processors by Kip Irvine © Pearson Education, 2010

Slides revised 2/6/2014 by Patrick Kelley

Overview

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling and Running Programs
- Defining Data
- System I/O Services

Basic Elements of Assembly Language

- Integer constants
- Character and string constants
- Reserved words and identifiers
- Directives and instructions
- Labels
- Mnemonics and Operands
- Comments
- Examples

Integer Constants

- Decimal or hexadecimal digits
- By default decimal
- Decimal optional leading – sign
- Hexadecimal begins with 0x

Examples: 30, 0x6A, -42

Illegal: +256, -0xCB

Character and String Constants

- Enclose strings in double quotes
  - Each character occupies a single byte
  - In C style, strings end with zero
  - Special characters follow C convention: \n \t \"
  - "ABC"
  - "This is a two line string. \n Here is the second line."
  - "Say \"Goodnight\", Gracie."

Reserved Words and Identifiers

- Reserved words cannot be used as identifiers
  - Instruction mnemonics, directives
  - See Quick Reference in Appendix A
- Identifiers
  - No specified length (but be reasonable)
  - Case sensitive
  - Consist of letters, numbers, _, or .
  - first character cannot be a number

Directives

- Commands that are recognized and acted upon by the assembler
  - Not part of the processor instruction set
  - Used to declare code, data areas, define data, etc.
- Different assemblers have different directives
  - MASM, for example, has directives to declare platform type and delineate procedures.

Instructions

- Assembled into machine code by assembler
- Executed at runtime by the CPU
- An instruction may contain:
  - Label (optional)
  - Mnemonic (required)
  - Operands (depends on the instruction)
  - Comment (optional)

Labels

- Act as place markers
  - marks the address in code and data
- Follow identifer rules
- Begin on first space of a line
- End with a colon
  - LoopHere:
  - my_data:
  - Illegal – my data:
  - Illegal – 45bytes:

Mnemonics and Operands

- Instruction Mnemonics
  - memory aid
  - examples: move, add, sub, mult, nop, ror
- Operands
  - constant
  - register
  - memory (data label)

Constants are often called immediate values

Comments

- Comments are good!
  - explain the program's purpose
  - when it was written, and by whom
  - revision information
  - tricky coding techniques
  - application-specific explanations
- MIPS Comments
  - begin with #
  - only language element besides a label to begin a line
  - everything after the # to end of line is ignored

Instruction Format Examples

- No operands
  - syscall # perform a system service
- One operand
  - j next_input # jump to label 'next_input'
- Two operands
  - move $s1, $v0 # contents of $v0 into $s1
  - la $a0, bgErr # store address of label
- Three operands
  - addiu $t1, $t1, 3 # add 3 to contents of $t1 and # store back into $t1

What's Next

- Basic Elements of Assembly Language
- **Example: Adding and Subtracting Integers**
- Assembling and Running Programs
- Defining Data
- System I/O Services

Example: Adding and Subtracting Integers

```
# TITLE Add and Subtract              (AddSub.s)
# This program adds and subtracts 32-bit integers.

     .data
# variables
Num1:    .word    0x1000
Num2:    .word    0x5000
Num3:    .word    0x3000
Sum:     .word    0

     .text
     .globl    main

main:    # start of the main procedure
     lw    $t0, Num1        # Put Num1 into $t0
     lw    $t1, Num2        # Put Num2 into $t1
     lw    $t2, Num3        # Put Num3 into $t2
     add   $t4, $t0, $t1    # Add first two numbers, put in $t4
     sub   $t4, $t4, $t2    # Subtract third number from result
     sw    $t4, Sum         # Put result in sum

     jr    $ra              # return to caller (exit program)
```

Suggested Coding Standards (1 of 2)

- Some approaches to capitalization
  - capitalize nothing
  - capitalize everything
  - camel case
  - be consistent
- Other suggestions
  - descriptive identifier names
  - blank lines between procedures
  - blank lines between code groups

Suggested Coding Standards (2 of 2)

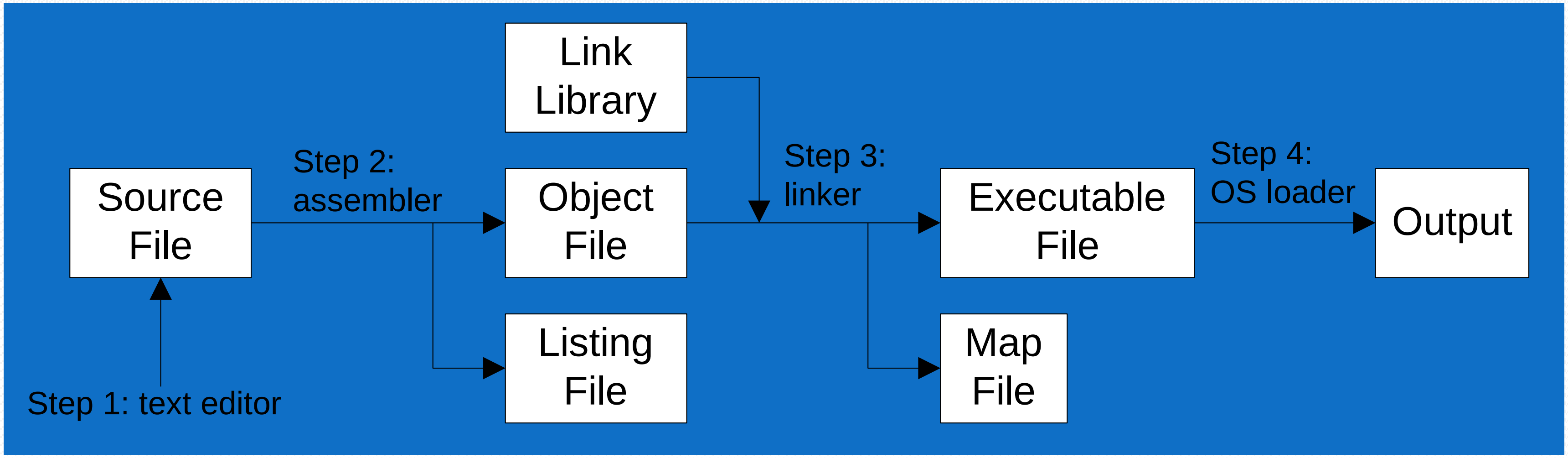
- Indentation and spacing
  - code and data labels – no indentation
  - executable instructions – indent 3-5 spaces
  - comments: right side of page, aligned vertically
  - 1-3 spaces between instruction and its operands
    - ex: add $t1, $t3, $s2
    - vary spacing so operands align vertically
    - setting tabs for alignment is a good idea

What's Next

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- **Assembling and Running Programs**
- Defining Data
- System I/O Services

Assemble-Link Execute Cycle

- The following diagram describes the steps from creating a source program through executing the compiled program.
- If the source code is modified, Steps 2 through 4 must be repeated.



Irvine, Kip R. Assembly Language for x86 Processors 6/e, 2010.

Running in QTSpim

- Linking is built in; you insert your code into the system code
- Assembling happens automatically when the source is loaded
- QTSpim shows listing interleaved with actual code
- Simulator can be operated as a debugger
  - Breakpoints can be set
  - Program can single-step
  - Data can be inspected directly
- Running program uses a ‘console’ for I/O
  - Separate window (be careful not to hide it)
  - Waits on input even if no breakpoint is set

What's Next

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling and Running Programs
- **Defining Data**
- System I/O Services

Defining Data

- Intrinsic Data Types
- Data Definition Statement
- Defining byte data
- Defining half word data
- Defining word data
- Defining string data
- Defining real number data
- Little Endian Order

Intrinsic Data Types

- byte
  - 8-bit integer
- half word
  - 16-bit integer
- word
  - 32-bit integer
- float
  - 32-bit single precision real number
- double
  - 32-bit double precision real number

Data Definition Statement

- A data definition statement sets aside storage in memory for a variable.
- May optionally assign a name (label) to the data
- Syntax:

  [*name*] *directive initializer* [,*initializer*] . . .

```
value1: .byte 10
```

- All initializers become binary data in memory

Defining byte data

Each of the following defines a single byte of storage:

```
Value1: .byte 0x3a          # hex constant
Value2: .byte 0             # smallest unsigned byte (0x00)
Value3: .byte 255           # largest unsigned byte (0xff)
Value4: .byte -128          # smallest signed byte (0x80)
Value5: .byte 127           # largest signed byte (0x7f)
```

Defining Byte Arrays

Examples that use multiple initializers:

```
List1: .byte 10,20,30,40
List2: .byte 10,20,30,40
       .byte 50,60,70,80
       .byte 81,82,83,84
```

Defining half word data

Each of the following defines two bytes of storage:

```
Value1: .half 0x3a9d        # hex constant
Value2: .half 0             # smallest unsigned 0x00)
Value3: .half 65535         # largest unsigned (0xffff)
Value4: .half -32768        # smallest signed (0x8000)
Value5: .half 32767         # largest signed (0x7fff)
Value6: .half 32,32,30,29   # array of half words
```

Defining word data

Each of the following defines a four bytes of storage:

```
Value1: .word 0x3a9d        # hex constant
Value2: .word 0             # smallest unsigned 0x00000000)
Value3: .word 4294967295    # largest unsigned (0xffffffff)
Value4: .word -2147483648   # smallest signed (0x80000000)
Value5: .word 2147483647    # largest signed (0x7fffffff)
Value6: .word 32,32,30,29   # array of words
Value7: .word 45:20         # stores 45 into 20 successive
                            # words of memory
```

Defining Strings

- A string is implemented as an array of characters
  - It often will be null-terminated
  - Easiest to initialize with a string directive and constant
- Examples:

```
str1 .byte 0,0x73,0x65,0x79  # 'yes',0 as 0,s,e,y
str2 .asciiz "yes"           # same as above
str3 .byte 0x6f,0x6e         # 'no' as o,n
str4 .ascii "no"             # same as above
lstr .ascii "This is going to be a long string.\n  It"
     .ascii "spans more than one line in both\ncode and"
     .ascii "output.  Only the last line\nin code"
     .asciiz "has a null terminator."
```

More on the 'backward' storage in a couple slides

Defining real data

MIPS uses the IEEE single-precision and double-precision formats we've already studied. You initialize floating point data like this:

```
Real1: .float 32.57           # stored in 32 bits
Real2: .float 12              # same as 12.0
Real3: .float 0.41
Real4: .float -17.2
Real5: .float 17.2,5,10.4,13.3    # array of floats
Real6: .double 32.5           # stored in 64 bits
Real7: .double 13.6,22.2,-17,2.5  # array of doubles
```

Little Endian Order

- All data types larger than a byte store their individual bytes in reverse order. The least significant byte occurs at the first (lowest) memory address.

- Example:

```
val1 .word 0x12345678
```

0000:	78
0001:	56
0002:	34
0003:	12

- Strings are also stored in 4-byte ‘chunks’ that are reversed. QTSpim translates them but be careful when accessing them directly.

What's Next

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling and Running Programs
- Defining Data
- **System I/O Services**

Using SPIM I/O Services

- How to do I/O varies in real world systems
  - Accessing a special register or ‘port’
  - Accessing reserved memory locations
  - Using built-in (either in the system or assembler) services
- SPIM uses the latter method
- Available System Services are listed in Appendix A right after the instruction set
- These services are accessed using the *syscall* instruction

Example: Some system I/O (1 of 5)

```
# TITLE Iodemo                                  (IOdemo.s)
# This program demonstrates some system I/O.

     .data
# variables
FloatPrompt:     .asciiz "Enter a float: "
DblPrompt:       .asciiz "Enter a double: "
IntPrompt:       .asciiz "Enter a integer: "
StrPrompt:       .asciiz "Enter a string: "
OutStr:          .asciiz "\nYour input was "
NewLine:         .asciiz "\n"
FloatIn:         .float  0.0
DoubleIn:        .double 0.0
IntIn:           .word   0
StrIn:           .space  256
```

Example: Some system I/O (2 of 5)

```
    .text
    .globl  main

main:   # start of the main procedure

# Get and print a string
    la      $a0, StrPrompt          # point to StrPrompt
    li      $v0, 4                  # print_string
    syscall
    la      $a0, StrIn              # point to input buffer
    li      $a1, 255                # set length of buffer
    li      $v0, 8                  # read_string
    syscall
    la      $a0, OutStr             # point to OutStr
    li      $v0, 4                  # print_string
    syscall
    la      $a0, StrIn              # point to input buffer
    li      $v0, 4                  # print_string
    syscall
    la      $a0, NewLine            # point to NewLine
    li      $v0, 4                  # print_string
    syscall
```

Example: Some system I/O (3 of 5)

```
# Get and print a float
la          $a0, FloatPrompt          # point to FloatPrompt
li          $v0, 4                    # print_string
syscall
li          $v0, 6                    # read_float
syscall
la          $a0, OutStr               # point to OutStr
li          $v0, 4                    # print_string
syscall
mov.s       $f12, $f0                 # move float input to output
li          $v0, 2                    # print_float
syscall
la          $a0, NewLine              # point to NewLine
li          $v0, 4                    # print_string
syscall
```

Example: Some system I/O (4 of 5)

```
# Get and print a double
la       $a0, DblPrompt        # point to DblPrompt
li       $v0, 4                # print_string
syscall
li       $v0, 7                # read_double
syscall
la       $a0, OutStr           # point to OutStr
li       $v0, 4                # print_string
syscall
mov.d    $f12, $f0             # move float input to output
li       $v0, 3                # print_double
syscall
la       $a0, NewLine          # point to NewLine
li       $v0, 4                # print_string
syscall
```

Example: Some system I/O (5 of 5)

```
# Get and print an integer
la        $a0, IntPrompt          # point to IntPrompt
li        $v0, 4                  # print_string
syscall
li        $v0, 5                  # read_integer
syscall
move      $t0, $v0                # move input before it gets changed
la        $a0, OutStr             # point to OutStr
li        $v0, 4                  # print_string
syscall
move      $a0, $t0                # move the integer we saved into $a0
li        $v0, 1                  # print_integer
syscall
la        $a0, NewLine            # point to NewLine
li        $v0, 4                  # print_string
syscall

li        $v0, 10                 # Exit the prograrm
syscall
```