MIPS Load and Store Instructions

Cptr280

Dr Curtis Nelson

MIPS Program Template

# Comment giving name of program and description of function
# Template.s
# Bare-bones outline of MIPS assembly language program

.data
# Variable declarations here
# Starts at address 0x10010000

.text
.globl main
# starts at address 0x00400000
main:  # indicates start of code (first instruction to execute)
# remainder of program code here
# ...
# ...
MIPS Assembler Directives

- Top-level Directives:
  - .text
    - Indicates that following items are stored in the user text segment, typically instructions.
  - .data
    - Indicates that following data items are stored in the data segment.
  - .globl sym
    - Declare that symbol sym is global and can be referenced from other files.

Pseudo-instructions

- The MIPS assembler supports several pseudo-instructions:
  - Not directly supported in hardware;
  - Implemented using one or more supported instructions;
  - Simplify assembly language programming and translation.
- For example, the pseudo-instruction
  la $t0, label
  is implemented as
  lui $8, 4097 [label] (we will discuss this instruction shortly)
- The pseudo-instruction
  move $t0, $t1
  is implemented as
  add $t0, $zero, $t1 (because register 0 is always = 0)
- The pseudo-instruction
  li $t4, 20
  is implemented as
  ori $t4, $0, 20
- See the link "QTSpim Quick Reference" on the course web page for a list of the MIPS instructions and pseudo-instructions (macros) supported by the QTSpim assembler.
Load and Store

• The operands for all arithmetic and logic operations are contained in registers. To operate on data in main memory, the data is first copied into registers. A load operation copies data from main memory into a register. A store operation copies data from a register into main memory.
• When a word (4-bytes) is loaded or stored, the memory address must be a multiple of four. This is called an alignment restriction. Addresses that are a multiple of four are called word aligned. This restriction makes the hardware simpler and faster.
• The lw instruction loads a word into a register from memory. The sw instruction stores a word from a register into memory. Each instruction specifies a register and a memory address (details in a few slides).

Quiz

• Which of the following addresses are word aligned?
  – 0x000AE430
  – 0x00014432
  – 0x000B0737
  – 0x0E0D8844
Big Endian and Little Endian

• A load word or store word instruction uses only one memory address. The lowest address of the four bytes is used for the address of a block of four contiguous bytes.

• How is a 32-bit pattern held in the four bytes of memory? There are 32 bits in the four bytes and 32 bits in the pattern, but a choice has to be made about which byte of memory gets what part of the pattern. There are two ways that computers commonly do this:
  – Big Endian Byte Order: The most significant byte (the "big end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.
  – Little Endian Byte Order: The least significant byte (the "little end") of the data is placed at the byte with the lowest address. The rest of the data is placed in order in the next three bytes in memory.

Big Endian and Little Endian

• In these definitions, the data, a 32-bit pattern, is regarded as a 32-bit unsigned integer. The "most significant" byte is the one for the largest powers of two: \(2^{31}, \ldots, 2^{24}\). The "least significant" byte is the one for the smallest powers of two: \(2^{7}, \ldots, 2^{0}\).

• For example, say that the 32-bit pattern 0x12345678 is at address 0x00400000. The most significant byte is 0x12; the least significant is 0x78. Here are the two byte orders:

• Within a byte, the order of the bits is the same for all computers (no matter how the bytes themselves are arranged).
Byte Order of MIPS and QTSpim

- Within a byte, for all processors, bit 7 is the most significant bit. So the big end byte looks the same for both byte orderings. Usually in printed material this bit is shown at the left, as in 00010010. **Note:** except when discussing byte ordering, the "big end" byte is called the "high-order byte" or the "most significant byte".
- The MIPS processor chip can be set up in hardware to use either byte ordering. A computer system designer makes whatever choice best fits the rest of the components in the computer system. The QTSpim simulator uses the byte ordering of the computer it is running on.
  - Intel 80x86: little-endian.

Portability Problems

- When a word is loaded from memory, the electronics puts the bytes into the register in the correct order. Operations (such as addition) inside the processor use the same order. When the register is stored to memory the bytes are written in the same order. As long as the electronics is consistent, either byte order works. Usually you don't need to think about which order is used.
- However, when data from one computer is used on another you do need to be concerned. Say that you have a file of integer data that was written by an old mainframe computer. To read it correctly, you need to know (among other things):
  - The number of bits used to represent each integer;
  - The representational scheme used to represent integers (two's complement or other);
  - Which byte ordering (little or big endian) was used.
MIPS Addresses

- The MIPS instruction that loads a word into a register is the lw instruction. The store word instruction is sw. Each must specify a register and a memory address. A MIPS instruction is 32 bits (always). A MIPS memory address is 32 bits (always). How can a load or store instruction specify an address that is the same size as itself?
- An instruction that refers to memory uses a base register and an offset. The base register is a general purpose register that contains a 32-bit address. The offset is a 16-bit signed integer contained in the instruction. The sum of the address in the base register with the (sign-extended) offset forms the memory address.
- Here is the load word instruction in assembly language:
  - lw d,offset(b)  # $d ← Word from memory address (b + offset)
  # b is a register, offset is 16-bit two's complement.
- At execution time two things happen: (1) an address is calculated using the base register b and the offset offset, and (2) data is fetched from memory at that address.

Question

- Write the instruction that loads the word at address 0x00400060 into register $8. Assume that register $10 contains 0x00400000.

  lw $8, _____( )
Machine Instruction for Load Word

- Here is the machine code version of the instruction. It specifies the base register, the destination register, and the offset. It does not directly contain the memory address:

  100011 01010 01000 0000 0000 0110 0000
  lw $8,0x60($10)

  opcode  base dest offset  -- meaning of the fields

- Here is how this instruction is executed:
  1. The 32-bit address in $10 is: 0x00400000
  2. The offset is sign-extended to 32 bits: 0x00000060
  3. The memory address is the 32-bit sum of the above: 0x00400060
  4. Main memory is asked for data from that address.
  5. After a one machine cycle delay the data reaches $8. $8 ← 4 bytes from location 0x00400060

- There is a one machine cycle delay before the data from memory is available. Reaching outside of the processor chip into main memory takes time. But the processor does not wait and executes one more instruction while the load is going on. This is the load delay slot. The instruction immediately after a lw instruction should not use the register that is being loaded. Sometimes the instruction after the lw is a no-operation instruction.

Store Word Instruction

- The store word instruction, sw, copies data from a register to memory. The register is not changed. The memory address is specified using a base/register pair.

  sw $t,offset(b)  # Word at memory address (b+offset) ← $t
  # b is a register, offset is 16-bit two's complement

- As with the lw instruction, the memory address must be word aligned (a multiple of four).
- There are similar instructions to load and store bytes (8 bits) and half-words (16 bits).
Example: add2numbersProg2.asm

# Program adds 10 and 20

.text # text section
.globl main # call main by SPIM

main:
  la $t0, value # load address 'value' into $t0
  lw $t1, 0($t0) # load word 0 (value) into $t1
  lw $t2, 4($t0) # load word 4 (value) into $t2
  add $t3, $t1, $t2 # add two numbers into $t3
  sw $t3, 8($t0) # store word $t3 into 8 ($t0)

.data # data section
value: .word 10, 20, 0 # load data integers
# Default data start address is 0x10010000
# This is where (value) is stored.

Example: swap2memoryWords.asm

### Program to swap two memory words

.data # load data
   .word 7
   .word 3

.text
   .text
   .globl main

main:
  lui $s0, 0x1001 # load data area start address 0x10010000
  lw  $s1, 0($s0)
  lw  $s2, 4($s0)
  sw  $s2, 0($s0)
  sw  $s1, 4($s0)
Example: storeWords.asm

## Program shows memory storage and access (big vs. little endian)

`.data

here:  .word 0xabc89725, 100
       .byte 0, 1, 2, 3
       .asciiz "Sample text"

there: .space 6
       .byte 85
       .align 2
       .byte 32

.text
.globl main

main:
   la $t0, here
   lbu $t1, 0($t0)
   lbu $t2, 1($t0)
   lw $t3, 0($t0)
   sw $t3, 36($t0)
   sb $t3, 41($t0)

---

SPIM’s memory storage depends on the underlying machine: Intel 80x86 processors are little-endian.

Word placement in memory is exactly the same in big or little endian – a copy is placed.

Byte placement in memory depends on if it is big or little endian. In big-endian, bytes in a Word are counted from the byte 0 at the left (most significant) to byte 3 at the right (least significant); in little-endian it is the other way around.

Word access (lw, sw) is exactly the same in big or little endian – it is a copy from register to a memory word or vice versa.

Byte access depends on if it is big or little endian, because bytes are counted 0 to 3 from left to right in big-endian and counted 0 to 3 from right to left in little-endian.

---

More SPIM Example Programs

In the class examples directory you will find 18 simple well-documented MIPS assembly programs. Run the code in the order below of increasing complexity.

1. add2numbersProg1
2. add2numbersProg2
3. storeWords
4. swap2memoryWords
5. branchJump
6. systemCalls
7. overflow
8. averageOfBytes
9. printLoop
10. sumOfSquaresProg1
11. sumOfSquaresProg2
12. sumOfSquaresProg3
13. procCallsProg1
14. procCallsProg1Modified
15. procCallsProg2
16. addFirst100
17. factorialNonRecursive
18. factorialRecursive
Summary

- Memory usage (memory map);
- Pseudo instructions;
- Load and store instructions.