Computer Systems and Networks

ECPE 170 – Jeff Shafer – University of the Pacific

MIPS Assembly

(Memory Fundamentals)

Stub Program

```
    # Declare main as a global function
    .globl main

    # All program code is placed after the
    # .text assembler directive
    .text

# The label 'main' represents the starting point
main:
    # MAIN CODE GOES HERE

    # Exit the program by means of a syscall.
    # There are many syscalls - pick the desired one
    # by placing its code in $v0. The code for exit is "10"

    li $v0, 10 # Sets $v0 to "10" to select exit syscall
    syscall    # Exit

    # All memory structures are placed after the
    # .data assembler directive
    .data

    # The .word assembler directive reserves space
    # in memory for a single 4-byte word (or multiple 4-byte words)
    # and assigns that memory location an initial value
    # (or a comma separated list of initial values)
    #For example:
    #value:  .word 12
```

MIPS Memory Access

Memory

- Challenge: **Limited supply of registers**
  - Physical limitation: We can't put more on the processor chip, and maintain their current speed
  - *Many elements compete for space in the CPU…*
- Solution: **Store data in memory**
- MIPS provides instructions that transfer data between memory and registers

MIPS Memory Declarations

- All of the memory values must be declared in the `.data` section of the code
  - You ask the assembler to reserve a region of memory in the *data* section and refer to that region with a *label*

- Examples
  - Declare a 32-bit word with initial value of 12:
    ```
    Z:      .word 12
    ```
  - Declare a 256 byte region of memory (could be 64 integers, 256 chars, etc…)
    ```
    array:        .space 256
    ```
  - Declare a null-terminated string with initial value
    ```
    msg:   .asciiz "Hello world!"
    ```

Memory Fundamentals

MIPS **cannot** directly manipulate data in memory!

Data must be moved to a register first! (And results must be saved to a register when finished)

This is a common design in *RISC-style* machines: a *load-store* architecture

Memory Fundamentals

Yes, it's a **pain** to keep moving data between registers and memory.

But consider it your *motivation* to reduce the number of memory accesses. That will **improve program performance**!

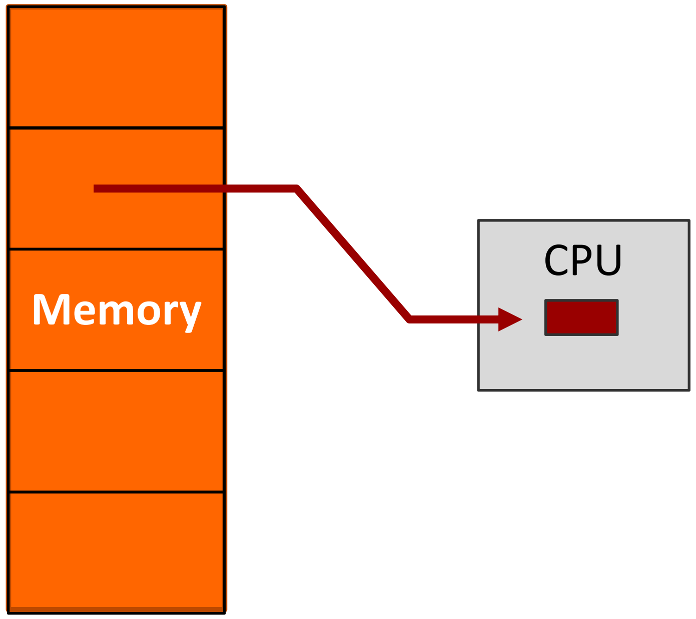
Memory Fundamentals

- Four questions to ask when accessing memory:
  1. What **direction** do I want to copy data?
     (i.e. to memory, or from memory?)
  2. What is the specific **memory address**?
  3. What is the specific **register name**? (or number)
  4. How **much data** do I want to move?

Memory – Fundamental Operations

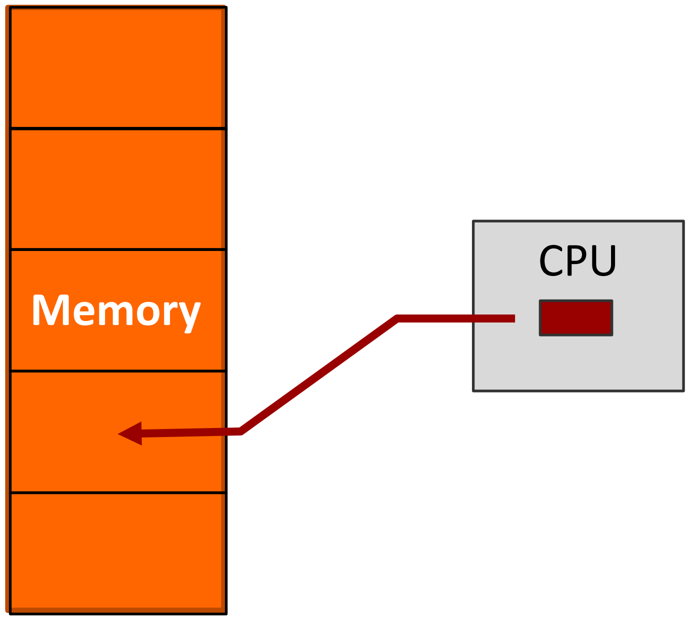
Load

- Copy data from memory to register



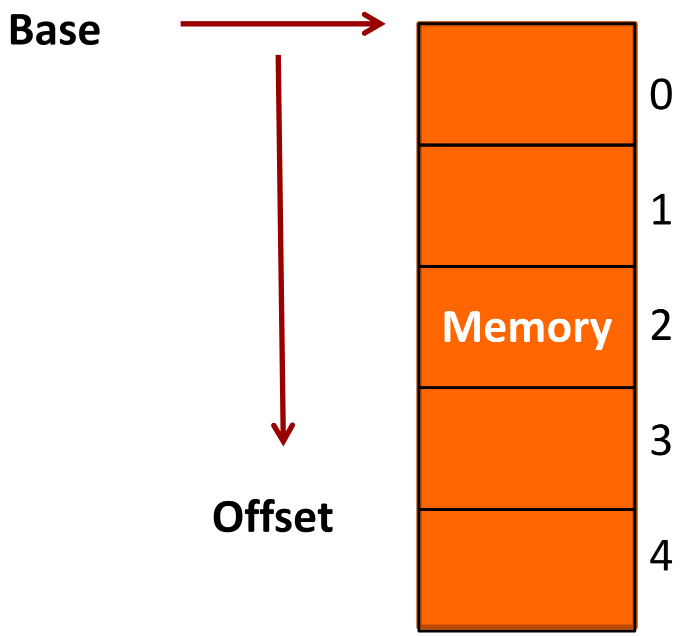
Store

- Copy data from register to memory



Memory – Determining Address

- There are many ways to calculate the desired memory address
  - These are called *addressing modes*
  - We'll just learn one mode now: **base + offset**
- The base address could be HUGE! (32 bits)
  - We'll place it in a **register**
- The offset is typically small
  - We'll directly include it in the instruction as an "immediate"



MIPS notation: offset(base)

Memory – Register Name

- What is the name of the register to use as either the data destination (for a *load*) or a data source (for a *store*)?
- Use the same register names previously learned

Memory - Data Transfer Size

- How much data do I want to load or store?
  - A full word? **(32 bits)**
  - A "half word"? **(16 bits)**
  - A byte? **(8 bits)**
- **We'll have a different instruction for each quantity of data**
- No option to load an entire array!
  - Will need a loop that loads 1 element at a time…

Memory – Data Transfer Instructions

- **Load** (copy from memory to register)

Word:
```
lw <reg>, <offset>(<base addr reg>)
```

Byte:
```
lb <reg>, <offset>(<base addr reg>)
```

- **Store** (copy from register to memory)

Word:
```
sw <reg>, <offset>(<base addr reg>)
```

Byte:
```
sb <reg>, <offset>(<base addr reg>)
```

$$\texttt{sb}\ \underset{\uparrow\ \text{Register}}{\texttt{<reg>}},\ \underbrace{\texttt{<offset>(<base addr reg>)}}_{\text{Memory Location}}$$

Example

- **What will this instruction do?**

```
lw $s1, 20($s2)
```

- Load word copies from memory to register:
  - Base address: stored in register $s2
  - Offset: 20 bytes
  - Destination register: $s1
  - Amount of data transferred: 1 word (32 bits)

Problem 1: Simple Program

- Declare memory variables *A*, *B*, and *C*, initialized to 20, 45, and 0, respectively. In *main*, set *C* to sum of *A* and *B*.

```
      .globl main
      .text
main: #Main goes here

       li $v0, 10 #v0 argument set to 10
                  # for system call "exit"
       syscall
      .data       #Data goes in this section
```

P1

Aside – Compiler

- **When programming in C / C++, are your variables (int, float, char, …) stored in memory or in registers?**
- **Answer: It depends**
- **Compiler will choose** where to place variables
  - Registers: Loop counters, frequently accessed scalar values, variables local to a procedure
  - Memory: Arrays, infrequently accessed data values

MIPS Array Access

Arrays Revisited

- Name of the array is the address of first element

```
int array[20];
printf("Address of first element:%u",array);
```

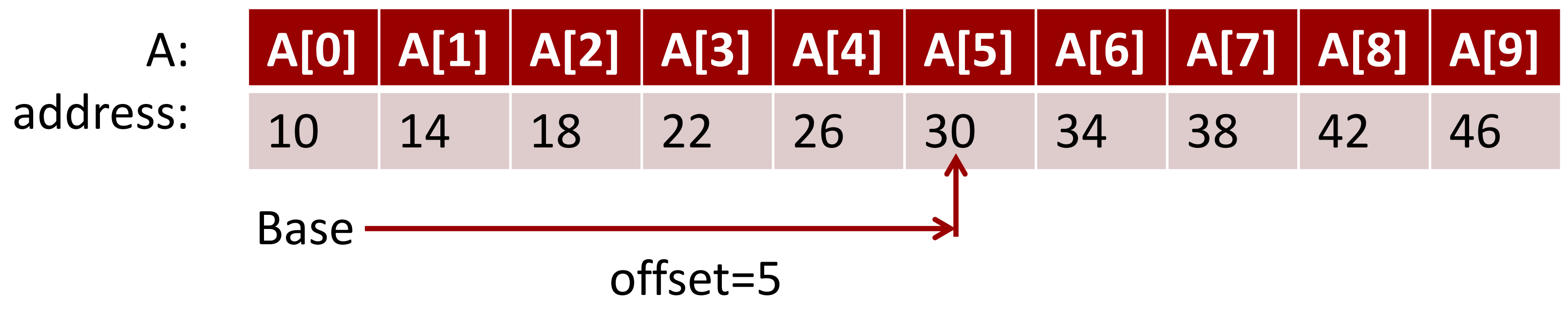
- Values are spaced by the size of the data
  - Integers – Spaced by 4 bytes
  - Doubles – Spaced by 8 bytes

```
int array[20];
printf("Address of the first element:%u",
        &array[0]); // Say it prints 65530
printf("Address of the second element:%u",
        &array[1]); // Will print 65534
```

Arrays Revisited

Base offset addressing / Indexed Addressing:

`A[5], array[i], ...`



Pointer arithmetic:

```
int array[10];
printf("array[5]:%u", *(array+5));
  //Adds 20 bytes to base address to access array[5]
```

Remember, in C, pointer arithmetic is done with respect to data size!

Problem 2: Arrays Revisited

- Write a C for-loop to print the values of a 1-D `array` of size `N` using:
  1. Indexed addressing
  2. Pointer arithmetic

P2

Task : Write Code

- **Write MIPS assembly for:**

g = h + array[16]

(Array of words. Can leave g and h in registers)

Map:

$s1 = g
$s2 = h
$s3 = base address of array

Code:

```
# Assume $s3 is already set
lw $t0, 16($s3)
add $s1, $s2, $t0
```

Memory Address

- Slight flaw in previous solution
  - The programmer intended to load the 16th array element
  - Each element is 4 bytes (1 word)
  - The offset is in bytes
  - 16 * 4 = 64

Correct Code:

```
# Assume $s3 is already set
lw $t0, 64($s3)
add $s1, $s2, $t0
```

C vs. MIPS

C Programming

- C has the format: `base[offset]`
- The C compiler multiplies the `offset` with the size of the data to compute the correct offset in bytes

MIPS Programming

- MIPS has the format: `offset(<base-addr-reg>)`
- In MIPS, <u>**YOU**</u> multiply the offset with size of the data to compute the correct `offset` in bytes

Problem 3: Base Offset Addressing

- **Write MIPS assembly for:**

array[12] = h + array[8]

(Array of words. Assume h is in register)

Map:
$s2 = h
$s3 = base address of array
$t1 = temp

Code:

```
# Assume $s3 is already set
lw $t0, 32($s3)
add $t1, $s2, $t0
sw $t1, 48($s3)
```

P3

Problem 4: Pointer Arithmetic

- **Write MIPS assembly for:**

g = h + array[i]

(Array of words. Assume g, h, and i are in registers)

Map:

$s1 = g
$s2 = h
$s3 = base address of array
$s4 = i

Code:

```
# "Multiply" i by 4
add $t1, $s4, $s4   # x2
add $t1, $t1, $t1   # x2 again
# Get addr of array[i]
add $t1, $t1, $s3
# Load array[i]
lw $t0, 0($t1)
# Compute add
add $s1, $s2, $t0
```

P4

Addresses

- Tip: To get the address of a label in the `.data` section, use the "load address" instructions (`la`)

```
la <reg>, label
```

Example:

```
# Load the starting address of
# the label 'array' into $s0
la $s0, array
```

Problem 5: Full Program

- Write a **complete MIPS program** which implements the C code below. Test your program in QtSPIM.

```
int array[7]; // Store in memory
int main()
{
   int i=0; // Store in register
   array[0]=5;
   array[1]=4;
   for(i=2; i<7; i++)
      array[i] = array[i-2] + array[i-1];
}
```

P5