MIPS32 Assembly Language Programming

Outline

- Introduction to Assembly Language
- Arithmetic Instructions
- Control Flow Instructions
- Load/Store Instructions
- Floating Points Instructions

What is Assembly Language?

- Low-level programming language for a computer
- One-to-one correspondence with the machine instructions
- Assembly language is specific to a given processor
- Assembler: converts assembly program into machine code
- Assembly language uses:
  - Mnemonics: to represent the names of low-level machine instructions
  - Labels: to represent the names of variables or memory addresses
  - Directives: to define data and constants
  - Macros: to facilitate the inline expansion of text into other code

Assembly Language Statements

- Three types of statements in assembly language
  - Typically, one statement should appear on a line

1. Executable Instructions
   - Generate machine code for the processor to execute at runtime
   - Instructions tell the processor what to do
2. Pseudo-Instructions and Macros
   - Translated by the assembler into real instructions
   - Simplify the programmer task
3. Assembler Directives
   - Provide information to the assembler while translating a program
   - Used to define segments, allocate memory variables, etc.
   - Non-executable: directives are not part of the instruction set

Assembly Language Instructions

- Assembly language instructions have the format:

```
[label:]   mnemonic   [operands]    [#comment]
```

- Label: (optional)
  - Marks the address of a memory location, must have a colon
  - Typically appear in data and text segments
- Mnemonic
  - Identifies the operation (e.g. **`add`**, **`sub`**, etc.)
- Operands
  - Specify the data required by the operation
  - Operands can be registers, memory variables, or constants
  - Most instructions have three operands

```
L1:   addiu $t0, $t0, 1       #increment $t0
```

Comments

- Single-line comment
  - Begins with a hash symbol # and terminates at end of line
- Comments are very important!
  - Explain the program's purpose
  - When it was written, revised, and by whom
  - Explain data used in the program, input, and output
  - Explain instruction sequences and algorithms used
  - Comments are also required at the beginning of every procedure
    - Indicate input parameters and results of a procedure
    - Describe what the procedure does

Program Template

```
# Title:                                  Filename:
# Author:                                 Date:
# Description:
# Input:
# Output:
################# Data segment #####################
.data
 . . .
################# Code segment #####################
.text
.globl main
main:                                     # main program entry
 . . .
li $v0, 10                                # Exit program
syscall
```

.DATA, .TEXT, & .GLOBL Directives

- **.DATA** directive
  - Defines the data segment of a program containing data
  - The program's variables should be defined under this directive
  - Assembler will allocate and initialize the storage of variables
- **.TEXT** directive
  - Defines the code segment of a program containing instructions
- **.GLOBL** directive
  - Declares a symbol as global
  - Global symbols can be referenced from other files
  - We use this directive to declare *main* function of a program

Data Definition Statement

- The assembler uses directives to define data
- It allocates storage in the static data segment for a variable
- May optionally assign a name (label) to the data
- Syntax:

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

`var1: .WORD 10`

- All initializers become binary data in memory

Data Directives

- **.BYTE** Directive
  - Stores the list of values as 8-bit bytes
- **.HALF** Directive
  - Stores the list as 16-bit values aligned on half-word boundary
- **.WORD** Directive
  - Stores the list as 32-bit values aligned on a word boundary
- **.FLOAT** Directive
  - Stores the listed values as single-precision floating point
- **.DOUBLE** Directive
  - Stores the listed values as double-precision floating point

String Directives

- **.ASCII** Directive
  - Allocates a sequence of bytes for an ASCII string
- **.ASCIIZ** Directive
  - Same as **.ASCII** directive, but adds a NULL char at end of string
  - Strings are null-terminated, as in the C programming language
- **.SPACE** Directive
  - Allocates space of *n* uninitialized bytes in the data segment

Examples of Data Definitions

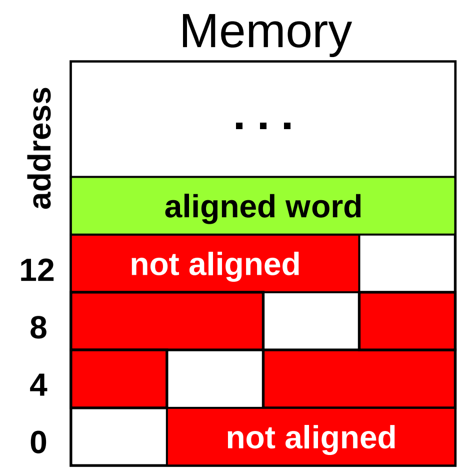
```
.DATA
var1:  .BYTE    'A', 'E', 127, -1, '\n'
var2:  .HALF    -10, 0xffff
var3:  .WORD    0x12345678:100
var4:  .FLOAT   12.3, -0.1
var5:  .DOUBLE  1.5e-10
str1:  .ASCII   "A String\n"
str2:  .ASCIIZ  "NULL Terminated String"
array: .SPACE   100
```

Array of 100 words Initialized with the same value

100 bytes (not initialized)

Memory Alignment

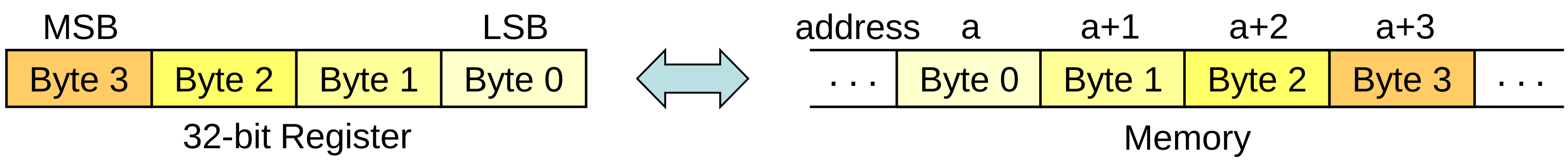
- Memory is viewed as an **addressable array of bytes**
- **Byte Addressing**: address points to a byte in memory
- However, words occupy 4 consecutive bytes in memory
  - MIPS instructions and integers occupy 4 bytes
- **Memory Alignment**:
  - Address must be multiple of size
  - Word address should be a multiple of **4**
  - Double-word address should be a multiple of **8**



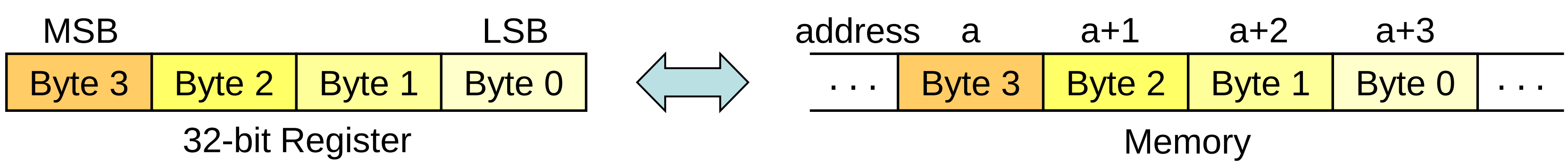
- **`.ALIGN n`** directive
  - Aligns the next data definition on a 2^n byte boundary
  - Forces the address of next data definition to be multiple of 2^n

Byte Ordering (Endianness)

- Processors can order bytes within a word in two ways
- Little Endian Byte Ordering
  - Memory address = Address of **least significant byte**
  - Example: Intel IA-32



- Big Endian Byte Ordering
  - Memory address = Address of **most significant byte**
  - Example: SPARC architecture



- MIPS can operate with both byte orderings

Symbol Table

- Assembler builds a **symbol table** for labels
  - Assembler computes the address of each label in data segment
- Example

```
.DATA
var1:   .BYTE   1, 2,'Z'
str1:   .ASCIIZ "My String\n"
var2:   .WORD   0x12345678
.ALIGN  3
var3:   .HALF   1000
```

Symbol Table

Label	Address
var1	0x10010000
str1	0x10010003
var2	0x10010010
var3	0x10010018

var1 str1

0x10010000	1	2	'Z'	'M'	'y'	' '	'S'	't'	'r'	'i'	'n'	'g'	'\n'	0	0	0	Unused
0x10010010	0x12345678				0	0	0	0	1000								

var2 (aligned) Unused var3 (address is multiple of 8)

System Calls

- Programs do input/output through system calls
- The MIPS architecture provides a **`syscall`** instruction
  - To obtain services from the operating system
  - The operating system handles all system calls requested by program
- Since MARS is a simulator, it simulates the **`syscall`** services
- To use the **`syscall`** services:
  - Load the service number in register **`$v0`**
  - Load argument values, if any, in registers **`$a0`**, **`$a1`**, etc.
  - Issue the **`syscall`** instruction
  - Retrieve return values, if any, from result registers

Syscall Services

Service	$v0	Arguments / Result
Print Integer	1	$a0 = integer value to print
Print Float	2	$f12 = float value to print
Print Double	3	$f12 = double value to print
Print String	4	$a0 = address of null-terminated string
Read Integer	5	Return integer value in $v0
Read Float	6	Return float value in $f0
Read Double	7	Return double value in $f0
Read String	8	$a0 = address of input buffer $a1 = maximum number of characters to read
Allocate Heap memory	9	$a0 = number of bytes to allocate Return address of allocated memory in $v0
Exit Program	10	

Syscall Services – Cont'd

Print Char	11	\$a0 = character to print
Read Char	12	Return character read in \$v0
Open File	13	\$a0 = address of null-terminated filename string \$a1 = flags (0 = read-only, 1 = write-only) \$a2 = mode (ignored) Return file descriptor in \$v0 (negative if error)
Read from File	14	\$a0 = File descriptor \$a1 = address of input buffer \$a2 = maximum number of characters to read Return number of characters read in \$v0
Write to File	15	\$a0 = File descriptor \$a1 = address of buffer \$a2 = number of characters to write Return number of characters written in \$v0
Close File	16	\$a0 = File descriptor

Reading and Printing an Integer

```
################## Code segment ######################
.text
.globl main
main:                           # main program entry
  li    $v0, 5                  # Read integer
  syscall                       # $v0 = value read

  move  $a0, $v0                # $a0 = value to print
  li    $v0, 1                  # Print integer
  syscall

  li    $v0, 10                 # Exit program
  syscall
```

Reading and Printing a String

```
################### Data segment #######################
.data
  str: .space  10          # array of 10 bytes
################### Code segment #######################
.text
.globl main
main:                      # main program entry
  la   $a0, str            # $a0 = address of str
  li   $a1, 10             # $a1 = max string length
  li   $v0, 8              # read string
  syscall
  li   $v0, 4              # Print string str
  syscall
  li   $v0, 10             # Exit program
  syscall
```

Sum of Three Integers

```
# Sum of three integers
# Objective: Computes the sum of three integers.
# Input: Requests three numbers, Output: sum
################### Data segment ###################
.data
prompt:  .asciiz     "Please enter three numbers: \n"
sum_msg: .asciiz     "The sum is: "
################### Code segment ###################
.text
.globl main
main:
   la    $a0,prompt          # display prompt string
   li    $v0,4
   syscall
   li    $v0,5               # read 1st integer into $t0
   syscall
   move  $t0,$v0
```

Sum of Three Integers - (cont'd)

```
    li      $v0,5               # read 2nd integer into $t1
    syscall
    move    $t1,$v0
    li      $v0,5               # read 3rd integer into $t2
    syscall
    move    $t2,$v0
    addu    $t0,$t0,$t1         # accumulate the sum
    addu    $t0,$t0,$t2
    la      $a0,sum_msg         # write sum message
    li      $v0,4
    syscall
    move    $a0,$t0             # output sum
    li      $v0,1
    syscall
    li      $v0,10              # exit
    syscall
```

Instruction Categories

- Integer Arithmetic
- Arithmetic, logic, and shift instructions
- Data Transfer
  - Load and store instructions that access memory
  - Data movement and conversions
- Jump and Branch
  - Flow-control instructions that alter the sequential sequence
- Floating Point Arithmetic
  - Instructions that operate on floating-point registers
- Miscellaneous
  - Instructions that transfer control to/from exception handlers
  - Memory management instructions

R-Type Instruction Format

Op^6	Rs^5	Rt^5	Rd^5	sa^5	funct^6

- **Op**: operation code (opcode)
  - Specifies the operation of the instruction
  - Also specifies the format of the instruction
- **funct**: function code – extends the opcode
  - Up to $2^6 = 64$ functions can be defined for the same opcode
  - MIPS uses opcode 0 to define many R-type instructions
- Three Register Operands (common to many instructions)
  - **Rs, Rt**: first and second source operands
  - **Rd**: destination operand
  - **sa**: the shift amount used by shift instructions

R-Type Integer Add and Subtract

Instruction	Meaning	Op	Rs	Rt	Rd	sa	func
add $t1, $t2, $t3	$t1 = $t2 + $t3	0	$t2	$t3	$t1	0	0x20
addu $t1, $t2, $t3	$t1 = $t2 + $t3	0	$t2	$t3	$t1	0	0x21
sub $t1, $t2, $t3	$t1 = $t2 – $t3	0	$t2	$t3	$t1	0	0x22
subu $t1, $t2, $t3	$t1 = $t2 – $t3	0	$t2	$t3	$t1	0	0x23

- **add, sub**: **arithmetic overflow causes an exception**
  - In case of overflow, result is not written to destination register
- **addu, subu**: **arithmetic overflow is ignored**
- **addu, subu**: compute the same result as **add, sub**
- Many programming languages ignore overflow
  - The **+** operator is translated into **addu**
  - The **–** operator is translated into **subu**

Using Add / Subtract Instructions

- Consider the translation of: `f = (g+h)-(i+j)`
- Programmer / Compiler allocates registers to variables
- Given that: `$t0=f`, `$t1=g`, `$t2=h`, `$t3=i`, and `$t4=j`
- Called temporary registers: `$t0=$8, $t1=$9, …`
- Translation of: `f = (g+h)-(i+j)`

```
addu $t5, $t1, $t2  # $t5 = g + h
addu $t6, $t3, $t4  # $t6 = i + j
subu $t0, $t5, $t6  # f = (g+h)-(i+j)
```

- Assembler translates `addu $t5,$t1,$t2` into binary code

Op	$t1	$t2	$t5	sa	addu
000000	01001	01010	01101	00000	100001

Logic Bitwise Operations

- Logic bitwise operations: **and, or, xor, nor**

x	*y*	*x* and *y*
0	0	0
0	1	0
1	0	0
1	1	1

x	*y*	*x* or *y*
0	0	0
0	1	1
1	0	1
1	1	1

x	*y*	*x* xor *y*
0	0	0
0	1	1
1	0	1
1	1	0

x	*y*	*x* nor *y*
0	0	1
0	1	0
1	0	0
1	1	0

- AND instruction is used to clear bits: ***x* and 0 → 0**
- OR instruction is used to set bits: ***x* or 1 → 1**
- XOR instruction is used to toggle bits: ***x* xor 1 → not *x***
- NOT instruction is not needed, why?

  not $t1, $t2 is equivalent to: **nor $t1, $t2, $t2**

Logic Bitwise Instructions

Instruction	Meaning	Op	Rs	Rt	Rd	sa	func
`and $t1, $t2, $t3`	`$t1 = $t2 & $t3`	0	$t2	$t3	$t1	0	0x24
`or  $t1, $t2, $t3`	`$t1 = $t2 | $t3`	0	$t2	$t3	$t1	0	0x25
`xor $t1, $t2, $t3`	`$t1 = $t2 ^ $t3`	0	$t2	$t3	$t1	0	0x26
`nor $t1, $t2, $t3`	`$t1 = ~($t2|$t3)`	0	$t2	$t3	$t1	0	0x27

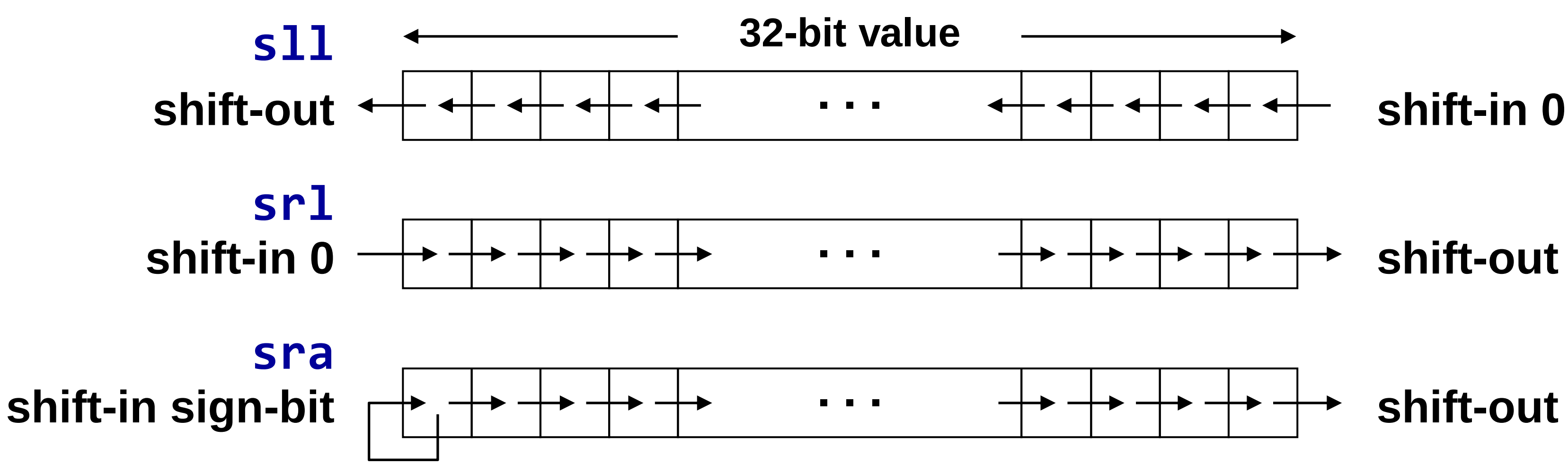
- Examples:

Given: `$t1 = 0xabcd1234` and `$t2 = 0xffff0000`

```
and $t0, $t1, $t2        # $t0 = 0xabcd0000
or  $t0, $t1, $t2        # $t0 = 0xffff1234
xor $t0, $t1, $t2        # $t0 = 0x54321234
nor $t0, $t1, $t2        # $t0 = 0x0000edcb
```

Shift Operations

- Shifting is to move the 32 bits of a number left or right
- `sll` means **shift left logical** (insert zero from the right)
- `srl` means **shift right logical** (insert zero from the left)
- `sra` means **shift right arithmetic** (insert sign-bit)
- The **5-bit shift amount** field is used by these instructions



Shift Instructions

Instruction	Meaning	Op	Rs	Rt	Rd	sa	func
`sll  $t1,$t2,10`	`$t1 = $t2 <<  10`	0	0	`$t2`	`$t1`	10	0
`srl  $t1,$t2,10`	`$t1 = $t2 >>> 10`	0	0	`$t2`	`$t1`	10	2
`sra  $t1,$t2,10`	`$t1 = $t2 >>  10`	0	0	`$t2`	`$t1`	10	3
`sllv $t1,$t2,$t3`	`$t1 = $t2 << $t3`	0	`$t3`	`$t2`	`$t1`	0	4
`srlv $t1,$t2,$t3`	`$t1 = $t2 >>>$t3`	0	`$t3`	`$t2`	`$t1`	0	6
`srav $t1,$t2,$t3`	`$t1 = $t2 >> $t3`	0	`$t3`	`$t2`	`$t1`	0	7

- `sll, srl, sra:` **shift by a constant amount**
  - The shift amount (**sa**) field specifies a number between 0 and 31
- `sllv, srlv, srav:` **shift by a variable amount**
  - A source register specifies the variable shift amount between 0 and 31
  - Only the lower 5 bits of the source register is used as the shift amount

Shift Instruction Examples

❖ Given that: **`$t2 = 0xabcd1234 and $t3 = 16`**

```
sll  $t1, $t2, 8            $t1 = 0xcd123400
srl  $t1, $t2, 4            $t1 = 0x0abcd123
sra  $t1, $t2, 4            $t1 = 0xfabcd123
srlv $t1, $t2, $t3          $t1 = 0x0000abcd
```

Op	Rs = $t3	Rt = $t2	Rd = $t1	sa	srlv
000000	01011	01010	01001	00000	000110

Binary Multiplication

- Shift Left Instruction (**sll**) can perform multiplication
  - When the multiplier is a power of 2
- You can factor any binary number into powers of 2
- Example: multiply **$t0** by **36**

$t0*36 = $t0*(4 + 32) = $t0*4 + $t0*32

```
sll  $t1, $t0, 2        # $t1 = $t0 * 4
sll  $t2, $t0, 5        # $t2 = $t0 * 32
addu $t3, $t1, $t2      # $t3 = $t0 * 36
```

Your Turn . . .

Multiply **`$t0`** by **`26`**, using shift and add instructions

Hint: **26 = 2 + 8 + 16**

```
sll  $t1, $t0, 1        # $t1 = $t0 * 2
sll  $t2, $t0, 3        # $t2 = $t0 * 8
sll  $t3, $t0, 4        # $t3 = $t0 * 16
addu $t4, $t1, $t2      # $t4 = $t0 * 10
addu $t5, $t4, $t3      # $t5 = $t0 * 26
```

Multiply **`$t0`** by **31**, Hint: **31 = 32 – 1**

```
sll  $t1, $t0, 5        # $t1 = $t0 * 32
subu $t2, $t1, $t0      # $t2 = $t0 * 31
```

I-Type Instruction Format

- Constants are used quite frequently in programs
  - The R-type shift instructions have a 5-bit shift amount constant
  - What about other instructions that need a constant?
- I-Type: Instructions with Immediate Operands

Op^6	Rs^5	Rt^5	immediate^{16}

- 16-bit immediate constant is stored inside the instruction
  - Rs is the source register number
  - Rt is now the destination register number (for R-type it was Rd)
- Examples of I-Type ALU Instructions:
  - Add immediate: `addi $t1, $t2, 5   # $t1 = $t2 + 5`
  - OR immediate: `ori  $t1, $t2, 5   # $t1 = $t2 | 5`

I-Type ALU Instructions

Instruction	Meaning	Op	Rs	Rt	Immediate
addi $t1, $t2, 25	$t1 = $t2 + 25	0x8	$t2	$t1	25
addiu $t1, $t2, 25	$t1 = $t2 + 25	0x9	$t2	$t1	25
andi $t1, $t2, 25	$t1 = $t2 & 25	0xc	$t2	$t1	25
ori $t1, $t2, 25	$t1 = $t2 | 25	0xd	$t2	$t1	25
xori $t1, $t2, 25	$t1 = $t2 ^ 25	0xe	$t2	$t1	25
lui $t1, 25	$t1 = 25 << 16	0xf	0	$t1	25

- **addi**: overflow causes an arithmetic exception
  - In case of overflow, result is not written to destination register
- **addiu**: same operation as **addi** but overflow is ignored
- Immediate constant for **addi** and **addiu** is signed
  - No need for **subi** or **subiu** instructions
- Immediate constant for **andi**, **ori**, **xori** is unsigned

Examples of I-Type ALU Instructions

❖ Given that registers **`$t0, $t1, $t2`** are used for **`A, B, C`**

Expression	Equivalent MIPS Instruction
`A = B + 5;`	`addiu $t0, $t1, 5`
`C = B – 1;`	`addiu $t2, $t1, -1`
`A = B & 0xf;`	`andi  $t0, $t1, 0xf`
`C = B | 0xf;`	`ori   $t2, $t1, 0xf`
`C = 5;`	`addiu $t2, $zero, 5`
`A = B;`	`addiu $t0, $t1, 0`

Op = addiu	Rs = $t1	Rt = $t2	-1 = 0b1111111111111111

No need for **`subiu`**, because **`addiu`** has signed immediate

Register **`$zero`** has always the value **`0`**

32-bit Constants

- I-Type instructions can have only 16-bit constants

Op[6]	Rs[5]	Rt[5]	immediate[16]

- What if we want to load a 32-bit constant into a register?
- **Can't have a 32-bit constant in I-Type instructions ☹**
  - The sizes of all instructions are fixed to 32 bits
- **Solution: use two instructions instead of one ☺**
- Suppose we want: **`$t1 = 0xAC5165D9`** (32-bit constant)

`lui`: load upper immediate

	Upper 16 bits	Lower 16 bits
`lui $t1, 0xAC51` — `$t1`	0xAC51	0x0000
`ori $t1, $t1, 0x65D9` — `$t1`	0xAC51	0x65D9

Pseudo-Instructions

- Introduced by the assembler as if they were real instructions
- Facilitate assembly language programming

Pseudo-Instruction	Equivalent MIPS Instruction
`move $t1, $t2`	`addu $t1, $t2, $zero`
`not $t1, $t2`	`nor $t1, $t2, $zero`
`neg $t1, $t2`	`sub $t1, $zero, $t2`
`li $t1, -5`	`addiu $t1, $zero, -5`
`li $t1, 0xabcd1234`	`lui $t1, 0xabcd` `ori $t1, $t1, 0x1234`

The MARS tool has a long list of pseudo-instructions

Control Flow

- High-level programming languages provide constructs:
  - To make decisions in a program: IF-ELSE
  - To repeat the execution of a sequence of instructions: LOOP
- The ability to make decisions and repeat a sequence of instructions distinguishes a computer from a calculator
- All computer architectures provide control flow instructions
- Essential for making decisions and repetitions
- These are the **conditional branch** and **jump** instructions

MIPS Conditional Branch Instructions

- MIPS **compare and branch** instructions:

  `beq Rs, Rt, label` if (`Rs == Rt`) branch to `label`

  `bne Rs, Rt, label` if (`Rs != Rt`) branch to `label`

- MIPS **compare to zero & branch** instructions:

  Compare to zero is used frequently and implemented efficiently

  `bltz Rs, label` if (`Rs < 0`) branch to `label`

  `bgtz Rs, label` if (`Rs > 0`) branch to `label`

  `blez Rs, label` if (`Rs <= 0`) branch to `label`

  `bgez Rs, label` if (`Rs >= 0`) branch to `label`

- `beqz` and `bnez` are defined as pseudo-instructions.

Branch Instruction Format

- Branch Instructions are of the I-type Format:

Op^6	Rs^5	Rt^5	16-bit offset

Instruction	I-Type Format			
`beq  Rs, Rt, label`	Op = 4	Rs	Rt	16-bit Offset
`bne  Rs, Rt, label`	Op = 5	Rs	Rt	16-bit Offset
`blez Rs, label`	Op = 6	Rs	0	16-bit Offset
`bgtz Rs, label`	Op = 7	Rs	0	16-bit Offset
`bltz Rs, label`	Op = 1	Rs	0	16-bit Offset
`bgez Rs, label`	Op = 1	Rs	1	16-bit Offset

- The branch instructions modify the **PC register** only
- **PC-Relative addressing**:

  If (branch is taken) **PC = PC + 4 + 4×offset** else **PC = PC+4**

Unconditional Jump Instruction

❖ The unconditional Jump instruction has the following syntax:

```
j    label      # jump to label
   . . .
label:
```

❖ The jump instruction is **always taken**

❖ The Jump instruction is of the J-type format:

$Op^6 = 2$	26-bit address

❖ The jump instruction modifies the program counter PC:

PC^4	26-bit address	00

multiple of 4

❖ The upper 4 bits of the PC are unchanged

Translating an IF Statement

- Consider the following IF statement:

  `if (a == b) c = d + e; else c = d – e;`

  Given that **a, b, c, d, e** are in **`$t0`** … **`$t4`** respectively

- How to translate the above IF statement?

```
        bne   $t0, $t1, else
        addu  $t2, $t3, $t4
        j     next
else:   subu  $t2, $t3, $t4
next:   . . .
```

Logical AND Expression

- Programming languages use **short-circuit evaluation**
- If first condition is **false**, second condition is **skipped**

```
if (($t1 > 0) && ($t2 < 0)) {$t3++;}
```

```
# One Possible Translation ...
    bgtz   $t1, L1          # first condition
    j      next             # skip if false
L1: bltz   $t2, L2          # second condition
    j      next             # skip if false
L2: addiu  $t3, $t3, 1      # both are true
next:
```

Better Translation of Logical AND

```
if (($t1 > 0) && ($t2 < 0)) {$t3++;}
```

Allow the program to **fall through** to second condition

`!($t1 > 0)` is equivalent to **`($t1 <= 0)`**

`!($t2 < 0)` is equivalent to **`($t2 >= 0)`**

Number of instructions is reduced from **5** to **3**

```
# Better Translation ...
    blez   $t1, next          # 1st condition false?
    bgez   $t2, next          # 2nd condition false?
    addiu  $t3, $t3, 1        # both are true
next:
```

Logical OR Expression

- **Short-circuit evaluation** for logical OR
- If first condition is **true**, second condition is **skipped**

```
if (($t1 > 0) || ($t2 < 0)) {$t3++;}
```

- Use **fall-through** to keep the code as short as possible

```
      bgtz  $t1, L1        # 1st condition true?
      bgez  $t2, next      # 2nd condition false?
L1:   addiu $t3, $t3, 1    # increment $t3
next:
```

Compare Instructions

- MIPS also provides **set less than** instructions

```
slt   Rd, Rs, Rt
```
if (Rs < Rt) Rd = 1 else Rd = 0

```
sltu  Rd, Rs, Rt
```
unsigned <

```
slti  Rt, Rs, imm
```
if (Rs < imm) Rt = 1 else Rt = 0

```
sltiu Rt, Rs, imm
```
unsigned <

- **Signed / Unsigned** comparisons compute different results

Given that: `$t0 = 1` and `$t1 = -1 = 0xffffffff`

`slt   $t2, $t0, $t1` computes `$t2 = 0`

`sltu  $t2, $t0, $t1` computes `$t2 = 1`

Compare Instruction Formats

Instruction	Meaning	Format					
`slt Rd, Rs, Rt`	`Rd=(Rs` $<_s$ `Rt)?1:0`	Op=0	Rs	Rt	Rd	0	0x2a
`sltu Rd, Rs, Rt`	`Rd=(Rs` $<_u$ `Rt)?1:0`	Op=0	Rs	Rt	Rd	0	0x2b
`slti Rt, Rs, im`	`Rt=(Rs` $<_s$ `im)?1:0`	0xa	Rs	Rt	16-bit immediate		
`sltiu Rt, Rs, im`	`Rt=(Rs` $<_u$ `im)?1:0`	0xb	Rs	Rt	16-bit immediate		

❖ The other comparisons are defined as pseudo-instructions:

`seq, sne, sgt, sgtu, sle, sleu, sge, sgeu`

Pseudo-Instruction	Equivalent MIPS Instructions
`sgt $t2, $t0, $t1`	`slt $t2, $t1, $t0`
`seq $t2, $t0, $t1`	`subu $t2, $t0, $t1` `sltiu $t2, $t2, 1`

Pseudo-Branch Instructions

- MIPS hardware does NOT provide the following instructions:

`blt, bltu`	branch if less than	(signed / unsigned)
`ble, bleu`	branch if less or equal	(signed / unsigned)
`bgt, bgtu`	branch if greater than	(signed / unsigned)
`bge, bgeu`	branch if greater or equal	(signed / unsigned)

- MIPS assembler defines them as pseudo-instructions:

Pseudo-Instruction	Equivalent MIPS Instructions
`blt $t0, $t1, label`	`slt $at, $t0, $t1` `bne $at, $zero, label`
`ble $t0, $t1, label`	`slt $at, $t1, $t0` `beq $at, $zero, label`

`$at` (**`$1`**) is the **assembler temporary register**

Using Pseudo-Branch Instructions

- Translate the IF statement to assembly language
- **$t1** and **$t2** values are **unsigned**

```
if($t1 <= $t2) {
  $t3 = $t4;
}
```

```
   bgtu  $t1, $t2, L1
   move  $t3, $t4
L1:
```

- **$t3**, **$t4**, and **$t5** values are **signed**

```
if (($t3 <= $t4) &&
    ($t4 >= $t5)) {
  $t3 = $t4 + $t5;
}
```

```
   bgt   $t3, $t4, L1
   blt   $t4, $t5, L1
   addu  $t3, $t4, $t5
L1:
```

Conditional Move Instructions

Instruction	Meaning	R-Type Format					
`movz Rd, Rs, Rt`	`if (Rt==0) Rd=Rs`	Op=0	Rs	Rt	Rd	0	0xa
`movn Rd, Rs, Rt`	`if (Rt!=0) Rd=Rs`	Op=0	Rs	Rt	Rd	0	0xb

```
if ($t0 == 0) {$t1=$t2+$t3;} else {$t1=$t2-$t3;}
```

```
      bne   $t0, $0, L1
      addu  $t1, $t2, $t3
      j     L2
L1:   subu  $t1, $t2, $t3
L2:   . . .
```

```
addu  $t1, $t2, $t3
subu  $t4, $t2, $t3
movn  $t1, $t4, $t0
. . .
```

- Conditional move can eliminate branch & jump instructions

Arrays

- In a high-level programming language, an array is a homogeneous data structure with the following properties:
  - All array elements are of the same type and size
  - Once an array is allocated, its size cannot be modified
  - The base address is the address of the first array element
  - The array elements can be indexed
  - The address of any array element can be computed
- In assembly language, an array is just a block of memory
- In fact, all objects are simply blocks of memory
- The memory block can be allocated statically or dynamically

Static Array Allocation

- An array can be allocated statically in the data segment
- A data definition statement allocates static memory:

  `label: .type value0 [, value1 ...]`

  `label:` is the name of the array

  `.type` directive specifies the size of each array element

  `value0, value1 ...` specify a list of initial values

- Examples of static array definitions:

```
arr1: .half 20, -1   # array of 2 half words
arr2: .word 1:5      # array of 5 words (value=1)
arr3: .space 20      # array of 20 bytes
str1: .asciiz "Null-terminated string"
```

Watching Values in the Data Segment

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	0xffff0014	0x00000001	0x00000001	0x00000001	0x00000001	0x00000001	0x00000000	0x00000000
0x10010020	0x00000000	0x00000000	0x00000000	0x6c6c754e	0x7265742d	0x616e696d	0x20646574	0x69727473
0x10010040	0x0000676e	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000	0x00000000

0x10010000 (.data) — Hexadecimal Addresses, Hexadecimal Values, ASCII

Data Segment

Address	Value (+0)	Value (+4)	Value (+8)	Value (+c)	Value (+10)	Value (+14)	Value (+18)	Value (+1c)
0x10010000	. . \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 .	\0 \0 \0 \0	\0 \0 \0 \0
0x10010020	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	l l u N	r e t -	a n i m	d e t	i r t s
0x10010040	\0 \0 g n	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0	\0 \0 \0 \0

0x10010000 (.data) — Hexadecimal Addresses, Hexadecimal Values, ASCII

- The labels window is the **symbol table**
  - Shows labels and corresponding addresses
- The `la` pseudo-instruction loads the address of any label into a register

Labels

Label	Address ▲
comparisons.asm	
arr1	0x10010000
arr2	0x10010004
arr3	0x10010018
str1	0x1001002c

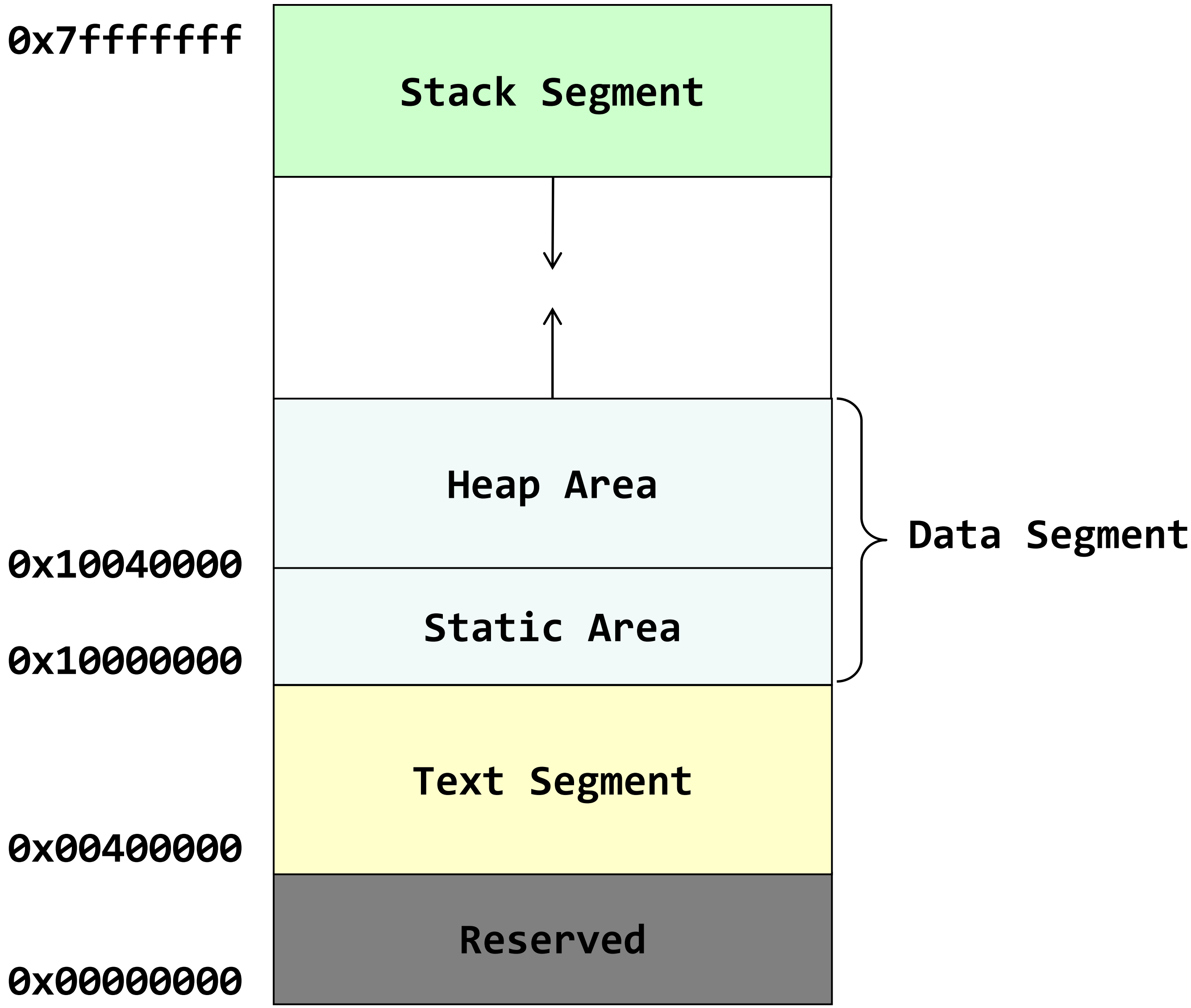
Data Text

Dynamic Memory Allocation

- One of the functions of the OS is to manage memory
- A program can allocate memory on the heap at runtime
- The heap is part of the data segment that can grow at runtime
- The program makes a system call (**$v0=9**) to allocate memory

```
.text
 . . .
 li $a0, 100        # $a0 = number of bytes to allocate
 li $v0, 9          # system call 9
 syscall            # allocate 100 bytes on the heap
 move $t0, $v0      # $t0 = address of allocated block
 . . .
```

Allocating Dynamic Memory on the Heap

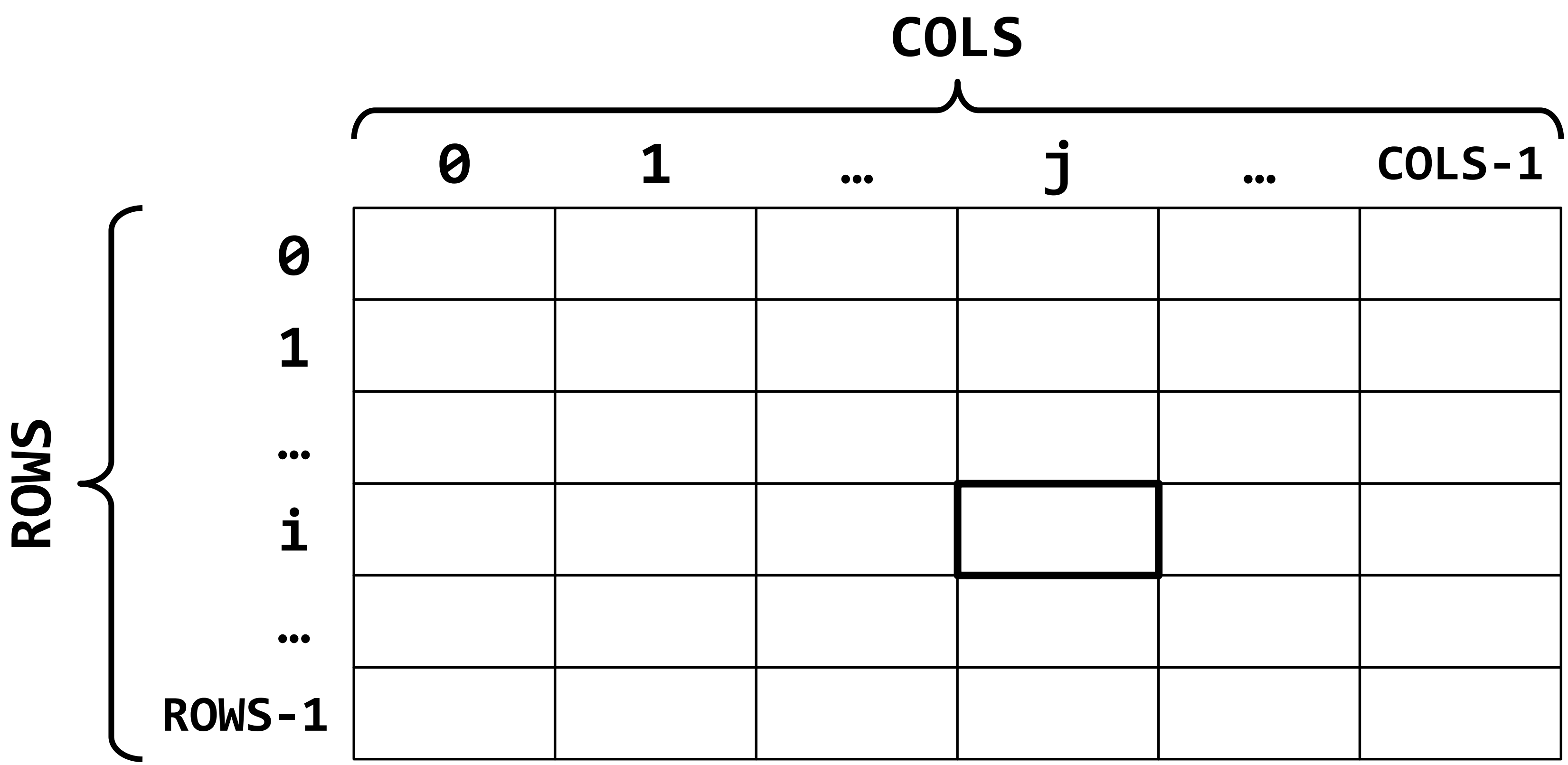


Computing the Addresses of Elements

- In a high-level programming language, an array is indexed
  - **`array[0]`** is the first element in the array
  - **`array[i]`** is the element at index **`i`**
  - **`&array[i]`** is the address of the element at index **`i`**
  - **`&array[i] = &array + i × element_size`**
- For a 2D array, the array is stored linearly in memory
  - **`matrix[Rows][Cols]`** has **`(Rows × Cols)`** elements
  - **`&matrix[i][j] = &matrix + (i×Cols + j) × element_size`**
- For example, to allocate a **`matrix[10][20]`** of integers:
  - **`matrix: .word 0:200 # 200 words (initialized to 0)`**
  - **`&matrix[1][5] = &matrix + (1×20 + 5)×4 = &matrix + 100`**

Element Addresses in a 2D Array

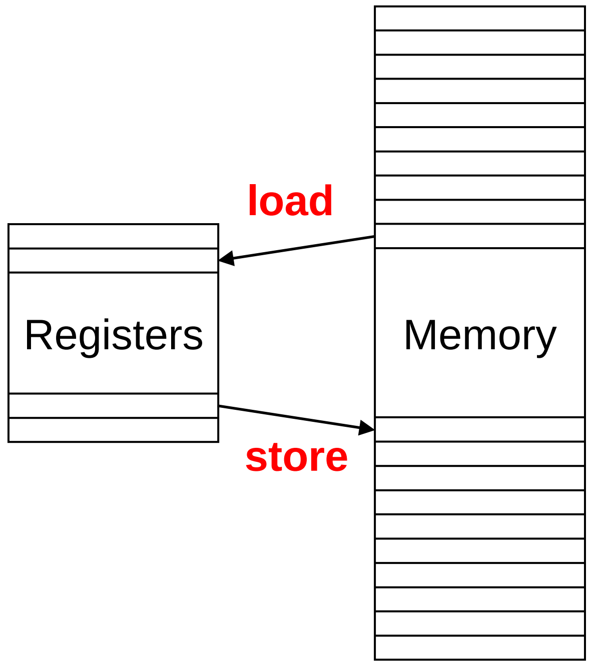
Address calculation is essential when programming in assembly



&matrix[i][j] = &matrix + (i×COLS + j) × Element_size

Load and Store Instructions

- Instructions that transfer data between memory & registers
- Programs include variables such as arrays and objects
- These variables are stored in memory
- Load Instruction:
  - Transfers data from memory to a register
- Store Instruction:
  - Transfers data from a register to memory
- Memory address must be specified by load and store



Load and Store Word

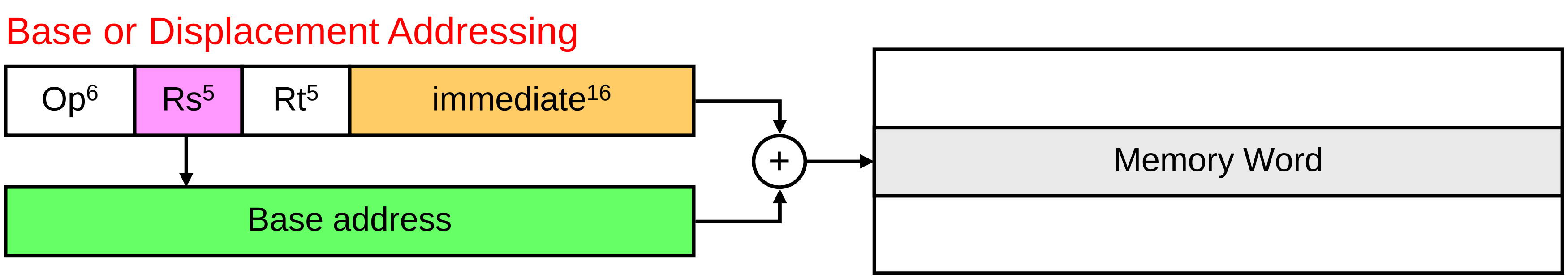
- Load Word Instruction (Word = 4 bytes in MIPS)

```
lw Rt, imm(Rs)     # Rt ← MEMORY[Rs+imm]
```

- Store Word Instruction

```
sw Rt, imm(Rs)     # Rt → MEMORY[Rs+imm]
```

- **Base / Displacement addressing** is used
  - Memory Address = Rs (**base**) + Immediate (**displacement**)
  - Immediate^{16} is sign-extended to have a signed displacement

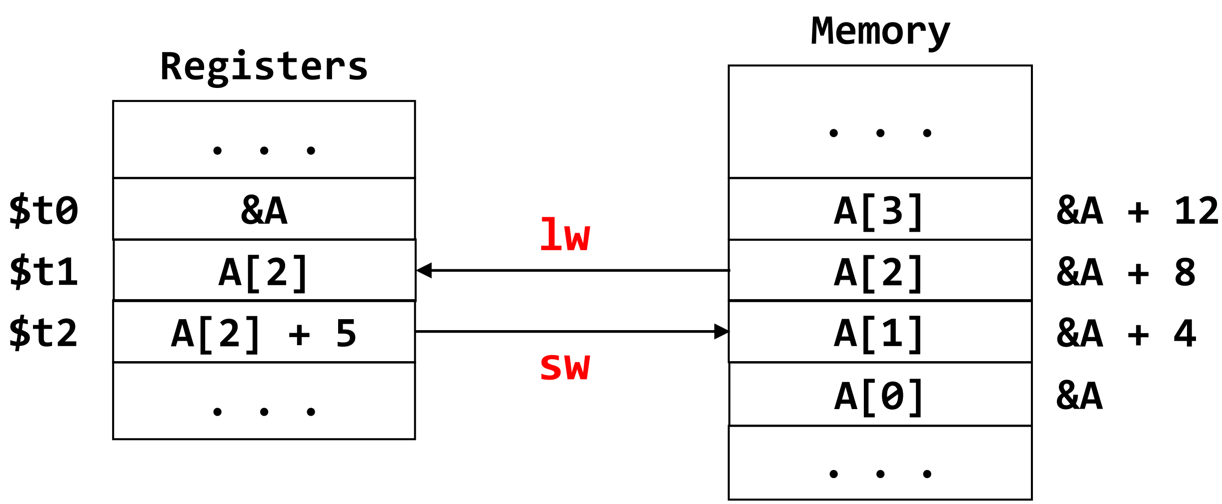


Example on Load & Store

- Translate: **A[1] = A[2] + 5** (**A** is an array of words)
- Given that the address of array **A** is stored in register **$t0**

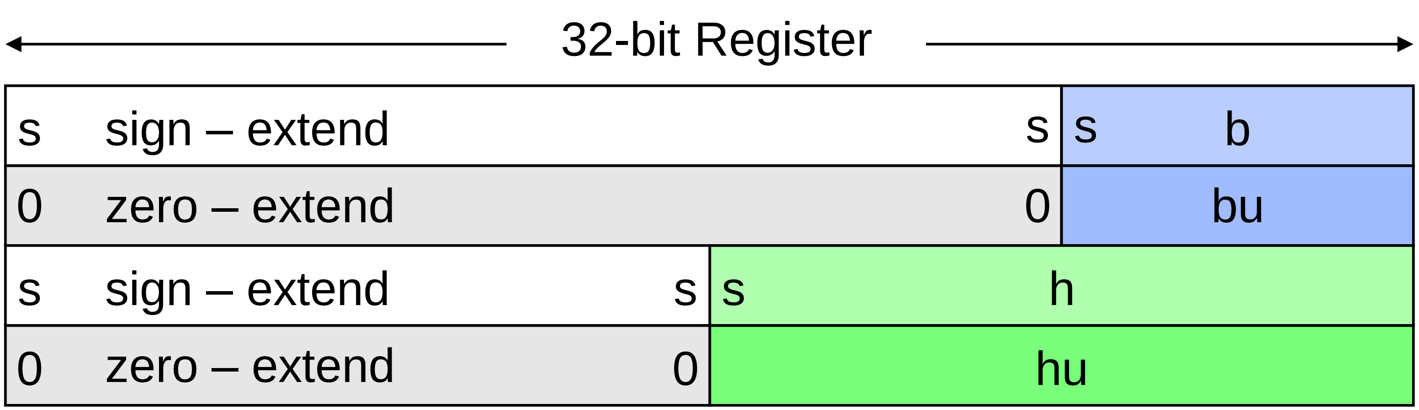
```
lw      $t1, 8($t0)      # $t1 = A[2]
addiu   $t2, $t1, 5      # $t2 = A[2] + 5
sw      $t2, 4($t0)      # A[1] = $t2
```

- Index of **A[2]** and **A[1]** should be multiplied by 4. Why?



Load and Store Byte and Halfword

- The MIPS processor supports the following data formats:
  - Byte = 8 bits, Half word = 16 bits, Word = 32 bits
- Load & store instructions for bytes and half words
  - lb = load byte, lbu = load byte unsigned, sb = store byte
  - lh = load half, lhu = load half unsigned, sh = store halfword
- Load expands a memory value to fit into a 32-bit register
- Store reduces a 32-bit register value to fit in memory



Load and Store Instructions

Instruction	Meaning	I-Type Format			
lb Rt, imm(Rs)	Rt $\leftarrow_1$ MEM[Rs+imm]	0x20	Rs	Rt	16-bit immediate
lh Rt, imm(Rs)	Rt $\leftarrow_2$ MEM[Rs+imm]	0x21	Rs	Rt	16-bit immediate
lw Rt, imm(Rs)	Rt $\leftarrow_4$ MEM[Rs+imm]	0x23	Rs	Rt	16-bit immediate
lbu Rt, imm(Rs)	Rt $\leftarrow_1$ MEM[Rs+imm]	0x24	Rs	Rt	16-bit immediate
lhu Rt, imm(Rs)	Rt $\leftarrow_2$ MEM[Rs+imm]	0x25	Rs	Rt	16-bit immediate
sb Rt, imm(Rs)	Rt $\rightarrow_1$ MEM[Rs+imm]	0x28	Rs	Rt	16-bit immediate
sh Rt, imm(Rs)	Rt $\rightarrow_2$ MEM[Rs+imm]	0x29	Rs	Rt	16-bit immediate
sw Rt, imm(Rs)	Rt $\rightarrow_4$ MEM[Rs+imm]	0x2b	Rs	Rt	16-bit immediate

- **Base / Displacement Addressing** is used
  - Memory Address = Rs (**Base**) + Immediate (**displacement**)
  - If Rs is $zero then Address = Immediate (**absolute**)
  - If Immediate is 0 then Address = Rs (**register indirect**)

Translating a WHILE Loop

- Consider the following WHILE loop:

  `i = 0; while (A[i] != value && i<n) i++;`

  Where **A** is an array of integers (4 bytes per element)

- Translate WHILE loop: **`$a0 = &A`**, **`$a1 = n`**, and **`$a2 = value`**

  `&A[i] = &A + i*4 = &A[i-1] + 4`

```
      li    $t0, 0          # $t0 = i = 0
loop: lw    $t1, 0($a0)     # $t1 = A[i]
      beq   $t1, $a2, done  # (A[i] == value)?
      beq   $t0, $a1, done  # (i == n)?
      addiu $t0, $t0, 1     # i++
      addiu $a0, $a0, 4     # $a0 = &A[i]
      j     loop            # jump backwards to loop
done: . . .
```

Copying a String

A string in C is an array of chars terminated with null char

```
i = 0;
do { ch = source[i]; target[i] = ch; i++; }
while (ch != '\0');
```

Given that: **`$a0 = &target`** and **`$a1 = &source`**

```
loop:
 lb     $t0, 0($a1)  # load byte: $t0 = source[i]
 sb     $t0, 0($a0)  # store byte: target[i]= $t0
 addiu  $a0, $a0, 1  # $a0 = &target[i]
 addiu  $a1, $a1, 1  # $a1 = &source[i]
 bnez   $t0, loop    # loop until NULL char
```

Initializing a Column of a Matrix

```
M = new int[10][5];      // allocate M on the heap
int i;
for (i=0; i<10; i++) { M[i][3] = i; }
```

```
# &M[i][3] = &M + (i*5 + 3) * 4 = &M + i*20 + 12
   li    $a0, 200        # $a0 = 10*5*4 = 200 bytes
   li    $v0, 9          # system call 9
   syscall               # allocate 200 bytes
   move  $t0, $v0        # $t0 = &M
   li    $t1, 0          # $t1 = i = 0
   li    $t2, 10         # $t2 = 10
L: sw    $t1, 12($t0)    # store M[i][3] = i
   addiu $t1, $t1, 1     # i++
   addiu $t0, $t0, 20    # $t0 = &M[i][3]
   bne   $t1, $t2, L     # if (i != 10) loop back
```

Jump and Branch Limits

- Jump Address Boundary = 2^{26} instructions = 256 MB
  - Jump cannot reach outside its 256 MB segment boundary
  - Upper 4 bits of PC are unchanged

Jump Target Address

PC^4	26-bit address	00

- Branch Address Boundary
  - Branch instructions use I-Type format (16-bit Offset)
  - PC-relative addressing:

PC^{30} + $Offset^{16}$ + 1	00

Branch Target address = PC + 4 × (1 + Offset)

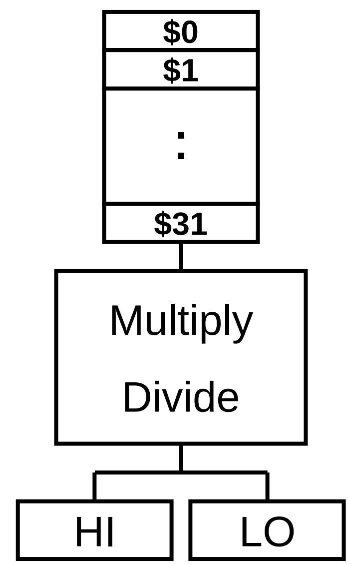
Count the number of instructions to skip starting at next instruction

Positive offset ➔ Forward branch, Negative offset ➔ Backward branch

Most branches are near : At most $\pm 2^{15}$ instructions can be skipped

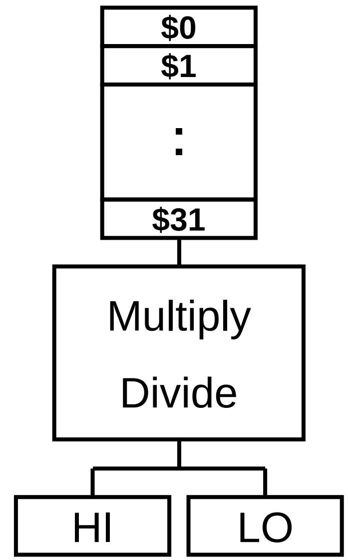
Integer Multiplication in MIPS

- Multiply instructions
  - `mult  Rs, Rt` **Signed multiplication**
  - `multu Rs, Rt` **Unsigned multiplication**
- 32-bit multiplication produces a 64-bit Product
- Separate pair of 32-bit registers
  - **HI = high-order 32-bit of product**
  - **LO = low-order 32-bit of product**
- MIPS also has a special `mul` instruction
  - `mul  Rd, Rs, Rt` `Rd = Rs × Rt`
  - Copy **LO** into destination register **Rd**
  - Useful when the product is small (32 bits) and **HI** is not needed



Integer Division in MIPS

- Divide instructions
  - `div  Rs, Rt` **Signed division**
  - `divu Rs, Rt` **Unsigned division**
- Division produces quotient and remainder
- Separate pair of 32-bit registers
  - **HI = 32-bit remainder**
  - **LO = 32-bit quotient**
  - If divisor is 0 then result is **unpredictable**
- Moving data from **HI, LO** to MIPS registers
  - `mfhi Rd` (**Rd = HI**)
  - `mflo Rd` (**Rd = LO**)



Integer Multiply and Divide Instructions

Instruction	Meaning	Format					
`mult Rs, Rt`	HI, LO = Rs $\times_s$ Rt	Op = 0	Rs	Rt	0	0	0x18
`multu Rs, Rt`	HI, LO = Rs $\times_u$ Rt	Op = 0	Rs	Rt	0	0	0x19
`mul Rd, Rs, Rt`	Rd = Rs $\times_s$ Rt	0x1c	Rs	Rt	Rd	0	2
`div Rs, Rt`	HI, LO = Rs $/_s$ Rt	Op = 0	Rs	Rt	0	0	0x1a
`divu Rs, Rt`	HI, LO = Rs $/_u$ Rt	Op = 0	Rs	Rt	0	0	0x1b
`mfhi Rd`	Rd = HI	Op = 0	0	0	Rd	0	0x10
`mflo Rd`	Rd = LO	Op = 0	0	0	Rd	0	0x12
`mthi Rs`	HI = Rs	Op = 0	Rs	0	0	0	0x11
`mtlo Rs`	LO = Rs	Op = 0	Rs	0	0	0	0x13

$\times_s$ = Signed multiplication, $\times_u$ = Unsigned multiplication

$/_s$ = Signed division, $/_u$ = Unsigned division

NO arithmetic exception can occur

String to Integer Conversion

- Consider the conversion of string "91052" into an integer

`'9'`	`'1'`	`'0'`	`'5'`	`'2'`

- How to convert the string into an integer?
- Initialize: **sum = 0**
- Load each character of the string into a register
  - Check if the character is in the range: **`'0'`** to **`'9'`**
  - Convert the character into a **digit** in the range: **`0`** to **`9`**
  - Compute: **sum = sum * 10 + digit**
  - Repeat until end of string or a non-digit character is encountered
- To convert "91052", initialize sum to 0 then …
  - sum = 9, then 91, then 910, then 9105, then 91052

String to Integer Conversion Function

```
#--------------------------------------------------------------
# str2int:  Convert a string of digits into unsigned integer
# Input:    $a0 = address of null terminated string
# Output:   $v0 = unsigned integer value
#--------------------------------------------------------------
str2int:
      li      $v0, 0            # Initialize: $v0 = sum = 0
      li      $t0, 10           # Initialize: $t0 = 10
L1:   lb      $t1, 0($a0)       # load $t1 = str[i]
      blt     $t1, '0', done    # exit loop if ($t1 < '0')
      bgt     $t1, '9', done    # exit loop if ($t1 > '9')
      addiu   $t1, $t1, -48     # Convert character to digit
      mul     $v0, $v0, $t0     # $v0 = sum * 10
      addu    $v0, $v0, $t1     # $v0 = sum * 10 + digit
      addiu   $a0, $a0, 1       # $a0 = address of next char
      j       L1                # loop back
done: jr      $ra               # return to caller
```

Integer to String Conversion

- Convert an unsigned 32-bit integer into a string
- How to obtain the decimal digits of the number?
  - Divide the number by 10, Remainder = decimal digit (0 to 9)
  - Convert decimal digit into its ASCII representation ('0' to '9')
  - Repeat the division until the quotient becomes zero
  - Digits are computed **backwards** from least to most significant
- Example: convert 2037 to a string
  - Divide 2037/10 quotient = 203 remainder = 7 char = '7'
  - Divide 203/10 quotient = 20 remainder = 3 char = '3'
  - Divide 20/10 quotient = 2 remainder = 0 char = '0'
  - Divide 2/10 quotient = 0 remainder = 2 char = '2'

Integer to String Conversion Function

```
#--------------------------------------------------------------
# int2str:  Converts an unsigned integer into a string
# Input:    $a0 = value, $a1 = buffer address (12 bytes)
# Output:   $v0 = address of converted string in buffer
#--------------------------------------------------------------
int2str:
      li    $t0, 10          # $t0 = divisor = 10
      addiu $v0, $a1, 11     # start at end of buffer
      sb    $zero, 0($v0)    # store a NULL character
L2:   divu  $a0, $t0         # LO  = value/10, HI = value%10
      mflo  $a0              # $a0 = value/10
      mfhi  $t1              # $t1 = value%10
      addiu $t1, $t1, 48     # convert digit into ASCII
      addiu $v0, $v0, -1     # point to previous byte
      sb    $t1, 0($v0)      # store character in memory
      bnez  $a0, L2          # loop if value is not 0
      jr    $ra              # return to caller
```

Function Call and Return

- To execution a function, the **caller** does the following:
  - Puts the parameters in a place that can be accessed by the callee
  - Transfer control to the callee function
- To return from a function, the **callee** does the following:
  - Puts the results in a place that can be accessed by the caller
  - Return control to the caller, next to where the function call was made
- Registers are the fastest place to pass parameters and return results. The MIPS architecture uses the following:
  - **`$a0-$a3:`** four argument registers in which to pass parameters
  - **`$v0-$v1:`** two value registers in which to pass function results
  - **`$ra:`** return address register to return back to the caller

Function Call and Return Instructions

- **JAL** (**Jump-and-Link**) is used to call a function
  - Save return address in **$31 = PC+4** and jump to function
  - Register **$31** (**$ra**) is used by **JAL** as the **return address**
- **JR** (**Jump Register**) is used to return from a function
  - Jump to instruction whose address is in register Rs (PC = Rs)
- **JALR** (**Jump-and-Link Register**)
  - Save return address in Rd = PC+4, and
  - Call function whose address is in register Rs (PC = Rs)
  - Used to call functions whose addresses are known at runtime

Instruction	Meaning	Format					
`jal label`	`$31 = PC+4, j Label`	Op=3	26-bit address				
`jr Rs`	`PC = Rs`	Op=0	Rs	0	0	0	8
`jalr Rd, Rs`	`Rd = PC+4, PC = Rs`	Op=0	Rs	0	Rd	0	9

Example

- Consider the following **swap** function (written in C)
- Translate this function to MIPS assembly language

```c
void swap(int v[], int k)
{   int temp;
    temp = v[k]
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

Parameters:

$a0 = Address of **v[]**
$a1 = **k**, and
Return address is in **$ra**

```
swap:
 sll $t0,$a1,2       # $t0=k*4
 add $t0,$t0,$a0     # $t0=v+k*4
 lw  $t1,0($t0)      # $t1=v[k]
 lw  $t2,4($t0)      # $t2=v[k+1]
 sw  $t2,0($t0)      # v[k]=$t2
 sw  $t1,4($t0)      # v[k+1]=$t1
 jr  $ra             # return
```

Call / Return Sequence

- ❖ Suppose we call function swap as: **`swap(a,10)`**
  - ✧ Pass **address** of array **`a`** and **`10`** as arguments
  - ✧ Call the function swap saving **return address** in **`$31 = $ra`**
  - ✧ Execute function swap
  - ✧ Return control to the point of origin (return address)

Registers

	. . .
$a0=$4	addr a
$a1=$5	10
	. . .
$ra=$31	ret addr

Caller

```
  la    $a0, a
  li    $a1, 10
  jal   swap
# return here
  . . .
```

```
swap:
 sll $t0,$a1,2
 add $t0,$t0,$a0
 lw  $t1,0($t0)
 lw  $t2,4($t0)
 sw  $t2,0($t0)
 sw  $t1,4($t0)
 jr  $ra
```

Details of JAL and JR

Address	Instructions	Assembly Language
00400020	lui $1, 0x1001	la $a0, a
00400024	ori $4, $1, 0	
00400028	ori $5, $0, 10	ori $a1,$0,10
0040002C	jal 0x10000f	jal swap
00400030	. . .	# return here
		swap:
0040003C	sll $8, $5, 2	sll $t0, $a1, 2
00400040	add $8, $8, $4	add $t0, $t0, $a0
00400044	lw $9, 0($8)	lw $t1, 0($t0)
00400048	lw $10,4($8)	lw $t2, 4($t0)
0040004C	sw $10,0($8)	sw $t2, 0($t0)
00400050	sw $9, 4($8)	sw $t1, 4($t0)
00400054	jr $31	jr $ra

Pseudo-Direct Addressing

PC = imm26<<2

0x10000f << 2

= 0x0040003C

$31 **0x00400030**

Register **$31** is the return address register

Second Example

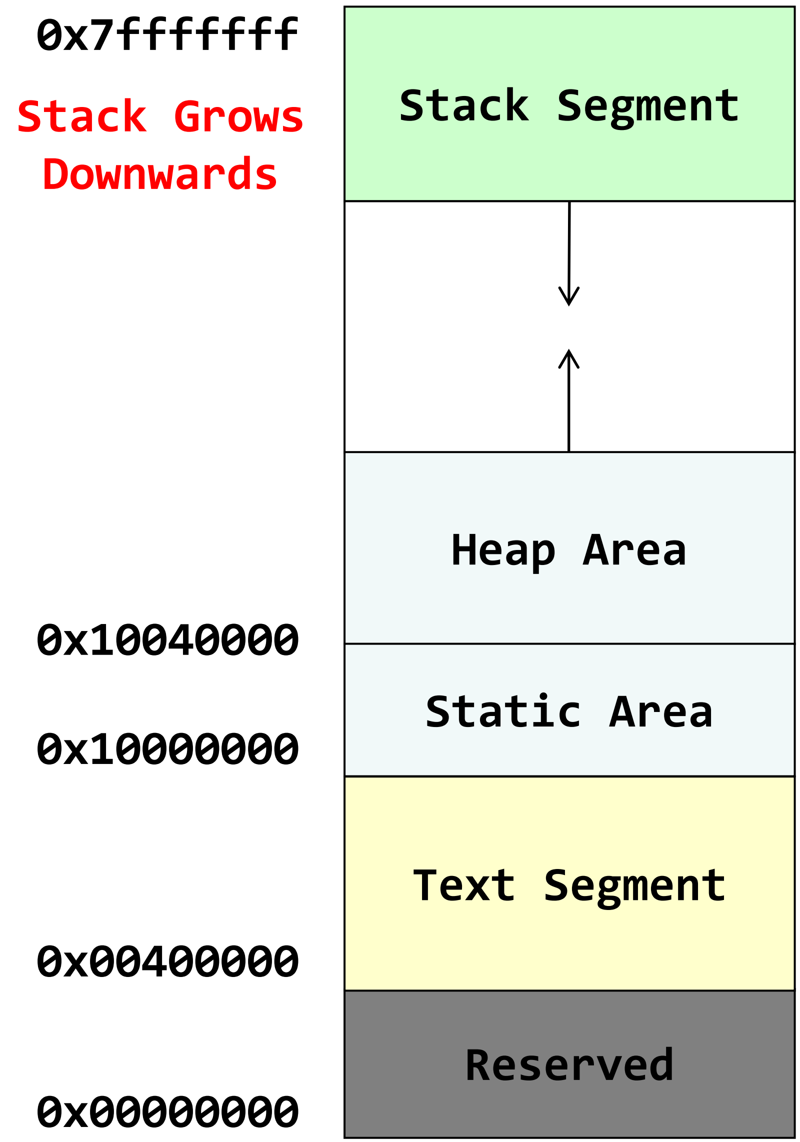
- Function **`tolower`** converts a capital letter to lowercase
- If parameter **`ch`** is not a capital letter then return **`ch`**

```
char tolower(char ch) {
  if (ch>='A' && ch<='Z')
    return (ch + 'a' - 'A');
  else
    return ch;
}
```

```
tolower:                        # $a0 = parameter ch
  blt   $a0, 'A', else          # branch if $a0 < 'A'
  bgt   $a0, 'Z', else          # branch if $a0 > 'Z'
  addi  $v0, $a0, 32            # 'a' – 'A' == 32
  jr    $ra                     # return to caller
else:
  move  $v0, $a0                # $v0 = ch
  jr    $ra                     # return to caller
```

The Stack Segment

- Every program has 3 segments when loaded into memory:
  - **Text segment**: stores machine instructions
  - **Data segment**: area used for static and dynamic variables
  - **Stack segment**: area that can be allocated and freed by functions
- The program uses only logical (virtual) addresses
- The actual (physical) addresses are managed by the OS

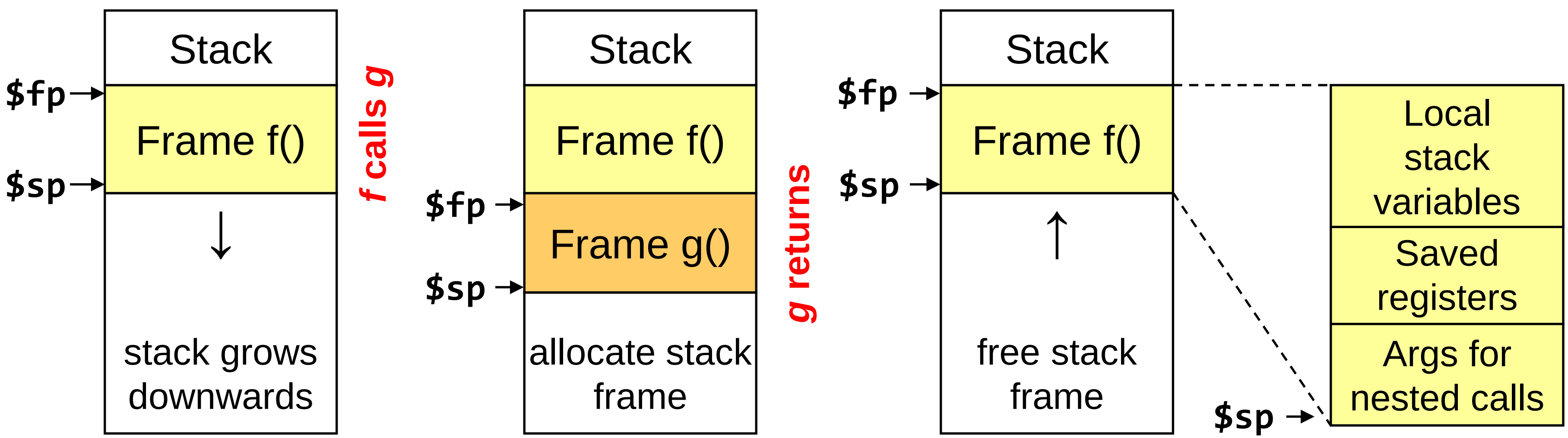


The Stack Segment (cont'd)

- The stack segment is used by functions for:
  - Passing parameters that cannot fit in registers
  - Allocating space for local variables
  - Saving registers across function calls
  - Implement recursive functions
- The stack segment is implemented via software:
  - The **`Stack Pointer $sp = $29`** (points to the top of stack)
  - The **`Frame Pointer $fp = $30`** (points to a stack frame)
- The stack pointer **`$sp`** is initialized to the base address of the stack segment, just before a program starts execution
- The MARS tool initializes register **`$sp`** to **`0x7fffeffc`**

Stack Frame

- **Stack frame** is an area of the stack containing …
  - Saved arguments, registers, local arrays and variables (if any)
- Called also the **activation frame**
- Frames are pushed and popped by adjusting …
  - Stack pointer `$sp = $29` (and sometimes frame pointer `$fp = $30`)
  - Decrement `$sp` to allocate stack frame, and increment to free



Leaf Function

- A leaf function does its work without calling any function
- Example of leaf functions are: **`swap`** and **`tolower`**
- A leaf function can freely modify some registers:
  - Argument registers: **`$a0 - $a3`**
  - Result registers: **`$v0 - $v1`**
  - Temporary registers: **`$t0 - $t9`**
  - These registers can be modified without saving their old values
- A leaf function does not need a stack frame if …
  - Its variables can fit in temporary registers
- A leaf function allocates a stack frame only if …
  - It requires additional space for its local variables

Non-Leaf Function

- A non-leaf function is a function that calls other functions
- A non-leaf function must allocate a stack frame
- Stack frame size is computed by the programmer (compiler)
- To allocate a stack frame of **`N`** bytes …
  - Decrement **`$sp`** by **`N`** bytes: **`$sp = $sp - N`**
  - **`N`** must be multiple of **`4`** bytes to have registers aligned in memory
  - In our examples, only register **`$sp`** will be used (**`$fp`** is not needed)
- Must save register **`$ra`** before making a function call
  - Must save **`$s0-$s7`** if their values are going to be modified
  - Other registers can also be preserved (if needed)
  - Additional space for local variables can be allocated (if needed)

Steps for Function Call and Return

- To make a function call …
  - Make sure that register `$ra` is saved before making a function call
  - Pass arguments in registers `$a0` thru `$a3`
  - Pass additional arguments on the stack (if needed)
  - Use the `JAL` instruction to make a function call (`JAL` modifies `$ra`)
- To return from a function …
  - Place the function results in `$v0` and `$v1` (if any)
  - Restore all registers that were saved upon function entry
    - Load the register values that were saved on the stack (if any)
  - Free the stack frame: `$sp = $sp + N` (stack frame = `N` bytes)
  - Jump to the return address: `jr $ra` (return to caller)

Preserving Registers

- The MIPS software specifies which registers must be preserved across a function call, and which ones are not

Must be Preserved	Not preserved
Return address: **$ra**	Argument registers: **$a0** to **$a3**
Stack pointer: **$sp**	Value registers: **$v0** and **$v1**
Saved registers: **$s0** to **$s7** and **$fp**	Temporary registers: **$t0** to **$t9**
Stack above the stack pointer	Stack below the stack pointer

- Caller saves register **$ra** before making a function call
- A callee function must preserve **$sp**, **$s0** to **$s7**, and **$fp**.
- If needed, the caller can save argument registers **$a0** to **$a3**. However, the callee function is free to modify them.

Example on Preserving Register

- A function **`f`** calls **`g`** twice as shown below. We don't know what **`g`** does, or which registers are used in **`g`**.
- We only know that function **`g`** receives two integer arguments and returns one integer result. Translate **`f`**:

```
int f(int a, int b) {
  int d = g(b, g(a, b));
  return a + d;
}
```

Translating Function f

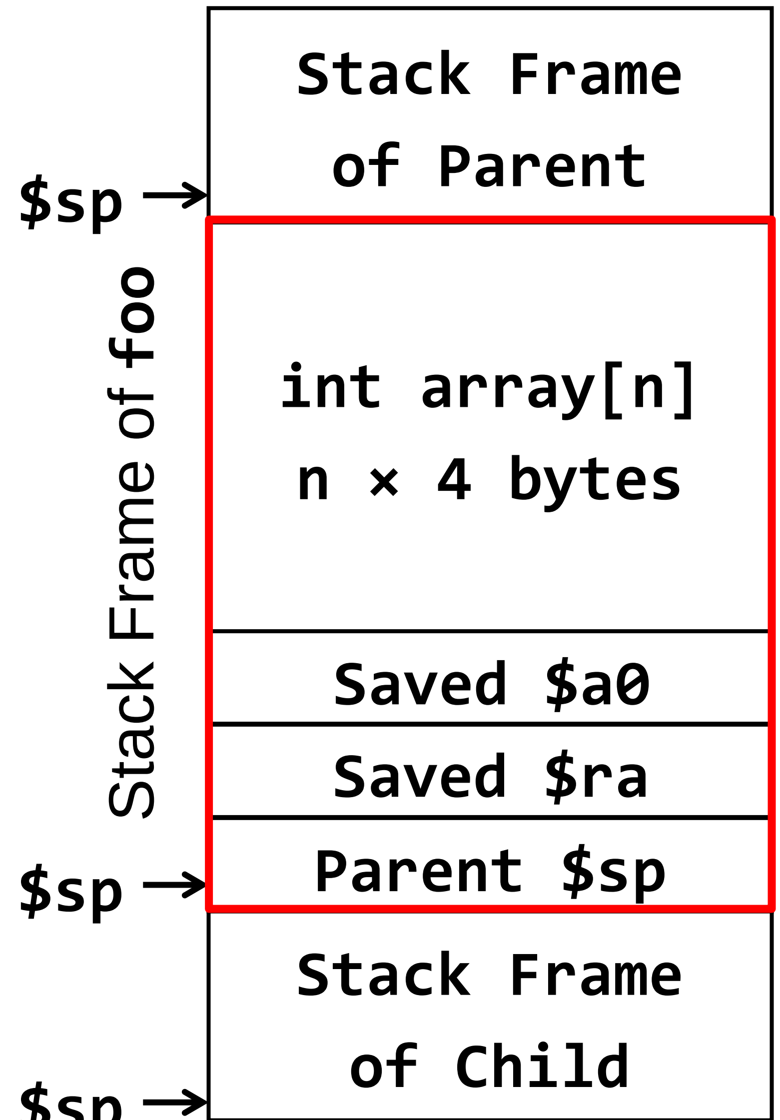
```
int f(int a, int b) {
  int d = g(b, g(a, b)); return a + d;
}
```

```
f:  addiu   $sp, $sp, -12       # allocate frame = 12 bytes
    sw      $ra, 0($sp)         # save $ra
    sw      $a0, 4($sp)         # save a (caller-saved)
    sw      $a1, 8($sp)         # save b (caller-saved)
    jal     g                   # call g(a,b)
    lw      $a0, 8($sp)         # $a0 = b
    move    $a1, $v0            # $a1 = result of g(a,b)
    jal     g                   # call g(b, g(a,b))
    lw      $a0, 4($sp)         # $a0 = a
    addu    $v0, $a0, $v0       # $v0 = a + d
    lw      $ra, 0($sp)         # restore $ra
    addiu   $sp, $sp, 12        # free stack frame
    jr      $ra                 # return to caller
```

Allocating a Local Array on the Stack

- In some languages, an array can be allocated on the stack
- The programmer (or compiler) must allocate a stack frame with sufficient space for the local array

```
void foo (int n) {
  // allocate on the stack
  int array[n];
  // generate random array
  random (array, n);
  // print array
  print  (array, n);
}
```



Translating Function foo

```
foo:                        # $a0 = n
  sll    $t0, $a0, 2        # $t0 = n*4 bytes
  addiu  $t0, $t0, 12       # $t0 = n*4 + 12 bytes
  move   $t1, $sp           # $t1 = parent $sp
  subu   $sp, $sp, $t0      # allocate stack frame
  sw     $t1, 0($sp)        # save parent $sp
  sw     $ra, 4($sp)        # save $ra
  sw     $a0, 8($sp)        # save n
  move   $a1, $a0           # $a1 = n
  addiu  $a0, $sp, 12       # $a0 = $sp + 12 = &array
  jal    random             # call function random
  addiu  $a0, $sp, 12       # $a0 = $sp + 12 = &array
  lw     $a1, 8($sp)        # $a1 = n
  jal    print              # call function print
  lw     $ra, 4($sp)        # restore $ra
  lw     $sp, 0($sp)        # restore parent $sp
  jr     $ra                # return to caller
```

Remarks on Function foo

- Function starts by computing its frame size: **`$t0 = n×4 + 12`** bytes
  - Local array is **`n×4`** bytes and the saved registers are **`12`** bytes
- Allocates its own stack frame: **`$sp = $sp - $t0`**
  - Address of local stack array becomes: **`$sp + 12`**
- Saves parent **`$sp`** and registers **`$ra`** and **`$a0`** on the stack
- Function **foo** makes two calls to functions **`random`** and **`print`**
  - Address of the stack array is passed in **`$a0`** and **`n`** is passed in **`$a1`**
- Just before returning:
  - Function **foo** restores the saved registers: parent **`$sp`** and **`$ra`**
  - Stack frame is freed by restoring **`$sp`**: **`lw $sp, 0($sp)`**

Bubble Sort (Leaf Function)

```
void bubbleSort (int A[], int n) {
  int swapped, i, temp;
  do {
    n = n-1;
    swapped = 0;              // false
    for (i=0; i<n; i++) {
      if (A[i] > A[i+1]) {
        temp = A[i];          // swap A[i]
        A[i] = A[i+1];        // with A[i+1]
        A[i+1] = temp;
        swapped = 1;          // true
      }
    }
  } while (swapped);
}
```

Worst case Performance $O(n^2)$

Best case Performance $O(n)$

Translating Function Bubble Sort

```
bubbleSort:                     # $a0 = &A, $a1 = n
do:   addiu  $a1, $a1, -1       # n = n-1
      blez   $a1, L2            # branch if (n <= 0)
      move   $t0, $a0           # $t0 = &A
      li     $t1, 0             # $t1 = swapped = 0
      li     $t2, 0             # $t2 = i = 0
for:  lw     $t3, 0($t0)        # $t3 = A[i]
      lw     $t4, 4($t0)        # $t4 = A[i+1]
      ble    $t3, $t4, L1       # branch if (A[i] <= A[i+1])
      sw     $t4, 0($t0)        # A[i] = $t4
      sw     $t3, 4($t0)        # A[i+1] = $t3
      li     $t1, 1             # swapped = 1
L1:   addiu  $t2, $t2, 1        # i++
      addiu  $t0, $t0, 4        # $t0 = &A[i]
      bne    $t2, $a1, for      # branch if (i != n)
      bnez   $t1, do            # branch if (swapped)
L2:   jr     $ra                # return to caller
```

Example of a Recursive Function

```
int recursive_sum (int A[], int n) {
  if (n == 0) return 0;
  if (n == 1) return A[0];
  int sum1 = recursive_sum (&A[0], n/2);
  int sum2 = recursive_sum (&A[n/2], n – n/2);
  return sum1 + sum2;
}
```

- Two recursive calls
  - First call computes the sum of the first half of the array elements
  - Second call computes the sum of the 2^{nd} half of the array elements
- How to translate a recursive function into assembly?

Translating a Recursive Function

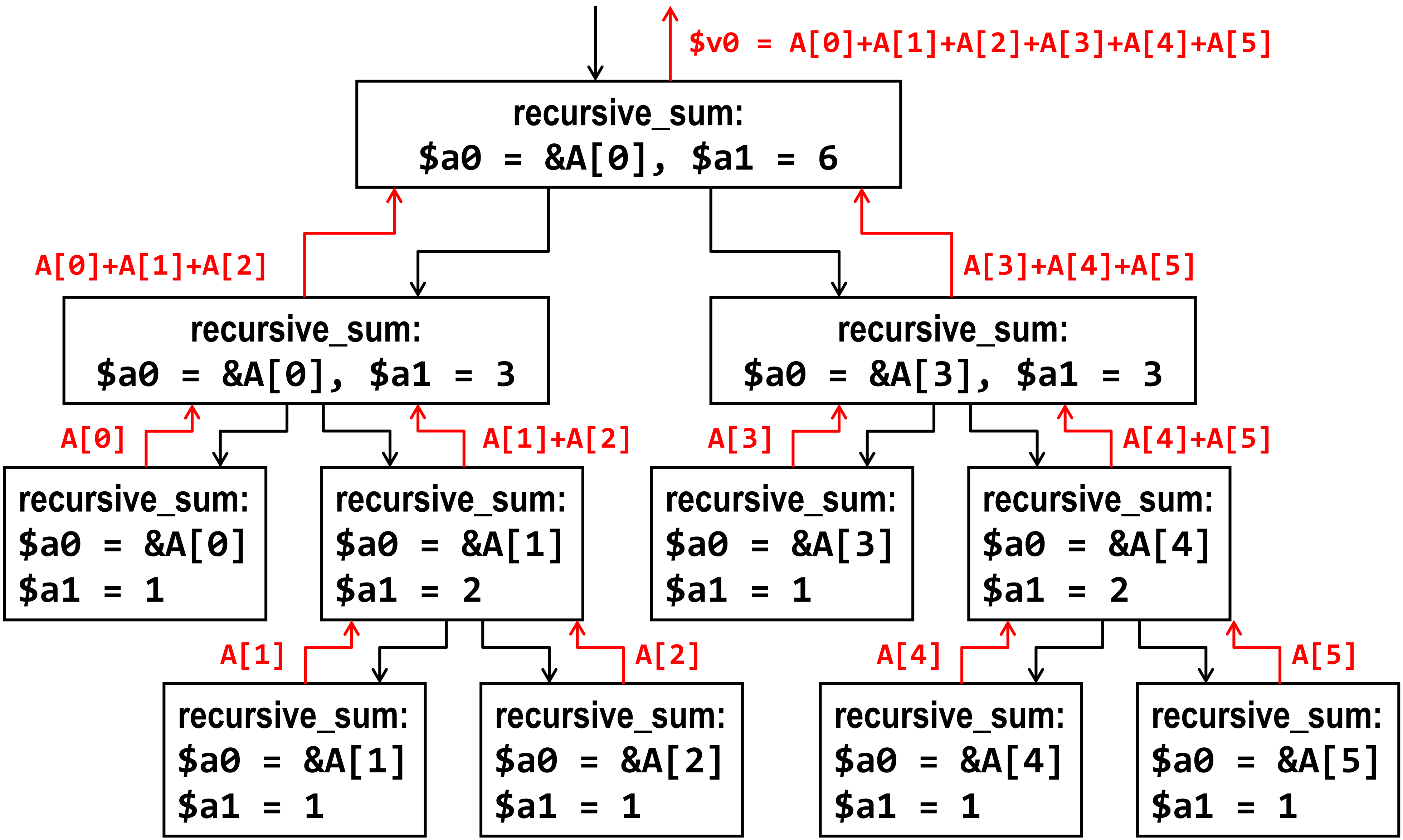
```
recursive_sum:                  # $a0 = &A, $a1 = n
    bnez   $a1, L1              # branch if (n != 0)
    li     $v0, 0
    jr     $ra                  # return 0
L1: bne    $a1, 1, L2           # branch if (n != 1)
    lw     $v0, 0($a0)          # $v0 = A[0]
    jr     $ra                  # return A[0]
L2: addiu  $sp, $sp, -12        # allocate frame = 12 bytes
    sw     $ra, 0($sp)          # save $ra
    sw     $s0, 4($sp)          # save $s0
    sw     $s1, 8($sp)          # save $s1
    move   $s0, $a0             # $s0 = &A (preserved)
    move   $s1, $a1             # $s1 = n  (preserved)
    srl    $a1, $a1, 1          # $a1 = n/2
    jal    recursive_sum        # first recursive call
```

Translating a Recursive Function (cont'd)

```
srl     $t0, $s1, 1          # $t0 = n/2
sll     $t1, $t0, 2          # $t1 = (n/2) * 4
addu    $a0, $s0, $t1        # $a0 = &A[n/2]
subu    $a1, $s1, $t0        # $a1 = n – n/2
move    $s0, $v0             # $s0 = sum1 (preserved)
jal     recursive_sum        # second recursive call
addu    $v0, $s0, $v0        # $v0 = sum1 + sum2
lw      $ra, 0($sp)          # restore $ra
lw      $s0, 4($sp)          # restore $s0
lw      $s1, 8($sp)          # restore $s1
addiu   $sp, $sp, 12         # free stack frame
jr      $ra                  # return to caller
```

❖ **$ra**, **$s0**, and **$s1** are preserved across recursive calls

Illustrating Recursive Calls



MIPS Floating Point Coprocessor

- Called Coprocessor 1 or the Floating Point Unit (FPU)
- 32 separate floating point registers: **$f0, $f1, …, $f31**
- FP registers are 32 bits for single precision numbers
- Even-odd register pair form a double precision register
- Use the even number for double precision registers
  - **$f0, $f2, $f4, …, $f30** are used for double precision
- Separate FP instructions for single/double precision
  - Single precision: **add.s, sub.s, mul.s, div.s** **(.s extension)**
  - Double precision: **add.d, sub.d, mul.d, div.d** **(.d extension)**
- FP instructions are more complex than the integer ones
  - Take more cycles to execute

Floating-Point Arithmetic Instructions

Instruction	Meaning	Op[6]	fmt[5]	ft[5]	fs[5]	fd[5]	func[6]
add.s $f5,$f3,$f4	$f5 = $f3 + $f4	0x11	0x10	$f4	$f3	$f5	0
sub.s $f5,$f3,$f4	$f5 = $f3 – $f4	0x11	0x10	$f4	$f3	$f5	1
mul.s $f5,$f3,$f4	$f5 = $f3 × $f4	0x11	0x10	$f4	$f3	$f5	2
div.s $f5,$f3,$f4	$f5 = $f3 / $f4	0x11	0x10	$f4	$f3	$f5	3
sqrt.s $f5,$f3	$f5 = sqrt($f3)	0x11	0x10	0	$f3	$f5	4
abs.s $f5,$f3	$f5 = abs($f3)	0x11	0x10	0	$f3	$f5	5
neg.s $f5,$f3	$f5 = –($f3)	0x11	0x10	0	$f3	$f5	7
add.d $f6,$f2,$f4	$f6,7 = $f2,3 + $f4,5	0x11	0x11	$f4	$f2	$f6	0
sub.d $f6,$f2,$f4	$f6,7 = $f2,3 – $f4,5	0x11	0x11	$f4	$f2	$f6	1
mul.d $f6,$f2,$f4	$f6,7 = $f2,3 × $f4,5	0x11	0x11	$f4	$f2	$f6	2
div.d $f6,$f2,$f4	$f6,7 = $f2,3 / $f4,5	0x11	0x11	$f4	$f2	$f6	3
sqrt.d $f6,$f2	$f6,7 = sqrt($f2,3)	0x11	0x11	0	$f2	$f6	4
abs.d $f6,$f2	$f6,7 = abs($f2,3)	0x11	0x11	0	$f2	$f6	5
neg.d $f6,$f2	$f6,7 = –($f2,3)	0x11	0x11	0	$f2	$f6	7

Floating-Point Load and Store

- Separate floating-point load and store instructions
  - **lwc1:** load word coprocessor 1
  - **ldc1:** load double coprocessor 1
  - **swc1:** store word coprocessor 1
  - **sdc1:** store double coprocessor 1

General purpose register is used as the **address** register

Instruction	Meaning	Op^6	rs^5	ft^5	$Immediate^{16}$
lwc1 $f2, 8($t0)	$f2 $\leftarrow_4$ Mem[$t0+8]	0x31	$t0	$f2	8
swc1 $f2, 8($t0)	$f2 $\rightarrow_4$ Mem[$t0+8]	0x39	$t0	$f2	8
ldc1 $f2, 8($t0)	$f2,3 $\leftarrow_8$ Mem[$t0+8]	0x35	$t0	$f2	8
sdc1 $f2, 8($t0)	$f2,3 $\rightarrow_8$ Mem[$t0+8]	0x3d	$t0	$f2	8

Data Movement Instructions

- Moving data between general purpose and FP registers
  - **mfc1:** move from coprocessor 1 (to a general purpose register)
  - **mtc1:** move to coprocessor 1 (from a general purpose register)
- Moving data between FP registers
  - **mov.s:** move single precision float
  - **mov.d:** move double precision float = even/odd pair of registers

Instruction	Meaning	Op⁶	fmt⁵	rt⁵	fs⁵	fd⁵	func
mfc1 $t0, $f2	$t0 = $f2	0x11	0	$t0	$f2	0	0
mtc1 $t0, $f2	$f2 = $t0	0x11	4	$t0	$f2	0	0
mov.s $f4, $f2	$f4 = $f2	0x11	0x10	0	$f2	$f4	6
mov.d $f4, $f2	$f4,5 = $f2,3	0x11	0x11	0	$f2	$f4	6

Convert Instructions

- Convert instruction: **cvt.x.y**
  - Convert the **source** format **y** into **destination** format **x**
- Supported Formats:
  - Single-precision float = **.s**
  - Double-precision float = **.d**
  - Signed integer word = **.w** (in a floating-point register)

Instruction	Meaning	Op6	fmt5		fs5	fd5	func
cvt.s.w $f2,$f4	$f2 = W2S($f4)	0x11	0x14	0	$f4	$f2	0x20
cvt.s.d $f2,$f4	$f2 = D2P($f4,5)	0x11	0x11	0	$f4	$f2	0x20
cvt.d.w $f2,$f4	$f2,3 = W2D($f4)	0x11	0x14	0	$f4	$f2	0x21
cvt.d.s $f2,$f4	$f2,3 = S2D($f4)	0x11	0x10	0	$f4	$f2	0x21
cvt.w.s $f2,$f4	$f2 = S2W($f4)	0x11	0x10	0	$f4	$f2	0x24
cvt.w.d $f2,$f4	$f2 = D2W($f4,5)	0x11	0x11	0	$f4	$f2	0x24

Floating-Point Compare and Branch

- Floating-Point unit has eight condition code **cc** flags
  - Set to 0 (false) or 1 (true) by any comparison instruction
- Three comparisons: **eq** (equal), **lt** (less than), **le** (less or equal)
- Two branch instructions based on the condition flag

Instruction	Meaning	Op^6	fmt^5	ft^5	fs^5		func
`c.eq.s cc $f2,$f4`	cc = ($f2 == $f4)	0x11	0x10	$f4	$f2	cc	0x32
`c.eq.d cc $f2,$f4`	cc = ($f2,3 == $f4,5)	0x11	0x11	$f4	$f2	cc	0x32
`c.lt.s cc $f2,$f4`	cc = ($f2 < $f4)	0x11	0x10	$f4	$f2	cc	0x3c
`c.lt.d cc $f2,$f4`	cc = ($f2,3 < $f4,5)	0x11	0x11	$f4	$f2	cc	0x3c
`c.le.s cc $f2,$f4`	cc = ($f2 <= $f4)	0x11	0x10	$f4	$f2	cc	0x3e
`c.le.d cc $f2,$f4`	cc = ($f2,3 <= $f4,5)	0x11	0x11	$f4	$f2	cc	0x3e
`bc1f cc Label`	branch if (cc == 0)	0x11	8	cc,0	16-bit Offset		
`bc1t cc Label`	branch if (cc == 1)	0x11	8	cc,1	16-bit Offset		

Example 1: Area of a Circle

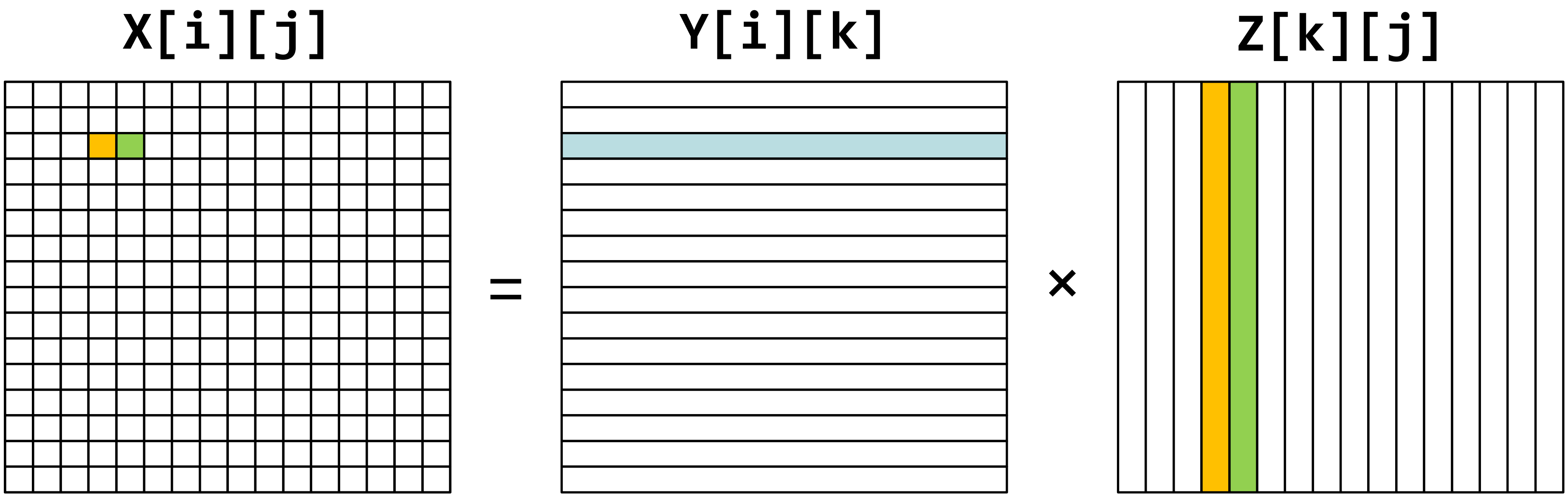
```
.data
  pi:     .double             3.141592653589793
  msg:    .asciiz             "Circle Area = "
.text
main:
  ldc1    $f2, pi             # $f2,3 = pi
  li      $v0, 7              # read double (radius)
  syscall                     # $f0,1 = radius
  mul.d   $f12, $f0, $f0      # $f12,13 = radius*radius
  mul.d   $f12, $f2, $f12     # $f12,13 = area
  la      $a0, msg
  li      $v0, 4              # print string (msg)
  syscall
  li      $v0, 3              # print double (area)
  syscall                     # print $f12,13
```

Example 2: Matrix Multiplication

```
void mm (int n, float X[n][n], Y[n][n], Z[n][n]) {
  for (int i=0; i!=n; i=i+1) {
    for (int j=0; j!=n; j=j+1) {
      float sum = 0.0;
      for (int k=0; k!=n; k=k+1) {
        sum = sum + Y[i][k] * Z[k][j];
      }
      X[i][j] = sum;
    }
  }
}
```

- ❖ Matrix size is passed in **`$a0 = n`**
- ❖ Matrix addresses in **`$a1 = &X`**, **`$a2 = &Y`**, and **`$a3 = &Z`**
- ❖ What is the MIPS assembly code for the procedure?

Access Pattern for Matrix Multiply



Matrix X is accessed by row.

Matrix Y is accessed by row.

Matrix Z accessed by column.

```
&X[i][j] = &X + (i*n + j)*4 = &X[i][j-1] + 4
&Y[i][k] = &Y + (i*n + k)*4 = &Y[i][k-1] + 4
&Z[k][j] = &Z + (k*n + j)*4 = &Z[k-1][j] + 4*n
```

Matrix Multiplication Procedure (1 of 3)

```
# arguments $a0=n, $a1=&X, $a2=&Y, $a3=&Z
mm: sll    $t0, $a0, 2      # $t0 = n*4 (row size)
    li     $t1, 0           # $t1 = i = 0

# Outer for (i = . . . )  loop starts here
L1: li     $t2, 0           # $t2 = j = 0

# Middle for (j = . . . ) loop starts here
L2: li     $t3, 0           # $t3 = k = 0
    move   $t4, $a2         # $t4 = &Y[i][0]
    sll    $t5, $t2, 2      # $t5 = j*4
    addu   $t5, $a3, $t5    # $t5 = &Z[0][j]
    mtc1   $zero, $f0       # $f0 = sum = 0.0
```

Matrix Multiplication Procedure (2 of 3)

```
# Inner for (k = . . . ) loop starts here
# $t3 = k, $t4 = &Y[i][k], $t5 = &Z[k][j]
L3: lwc1   $f1, 0($t4)     # load $f1 = Y[i][k]
    lwc1   $f2, 0($t5)     # load $f2 = Z[k][j]
    mul.s  $f3, $f1, $f2   # $f3 = Y[i][k]*Z[k][j]
    add.s  $f0, $f0, $f3   # sum = sum + $f3
    addiu  $t3, $t3, 1     # k = k + 1
    addiu  $t4, $t4, 4     # $t4 = &Y[i][k]
    addu   $t5, $t5, $t0   # $t5 = &Z[k][j]
    bne    $t3, $a0, L3    # loop back if (k != n)
# End of inner for loop
```

Matrix Multiplication Procedure (3 of 3)

```
    swc1   $f0, 0($a1)      # store X[i][j] = sum
    addiu  $a1, $a1, 4      # $a1 = &X[i][j]
    addiu  $t2, $t2, 1      # j = j + 1
    bne    $t2, $a0, L2     # loop L2 if (j != n)
# End of middle for loop

    addu   $a2, $a2, $t0    # $a2 = &Y[i][0]
    addiu  $t1, $t1, 1      # i = i + 1
    bne    $t1, $a0, L1     # loop L1 if (i != n)
# End of outer for loop

    jr     $ra              # return to caller
```