COMPUTER ARCHITECTURE

2nd Year Computer science

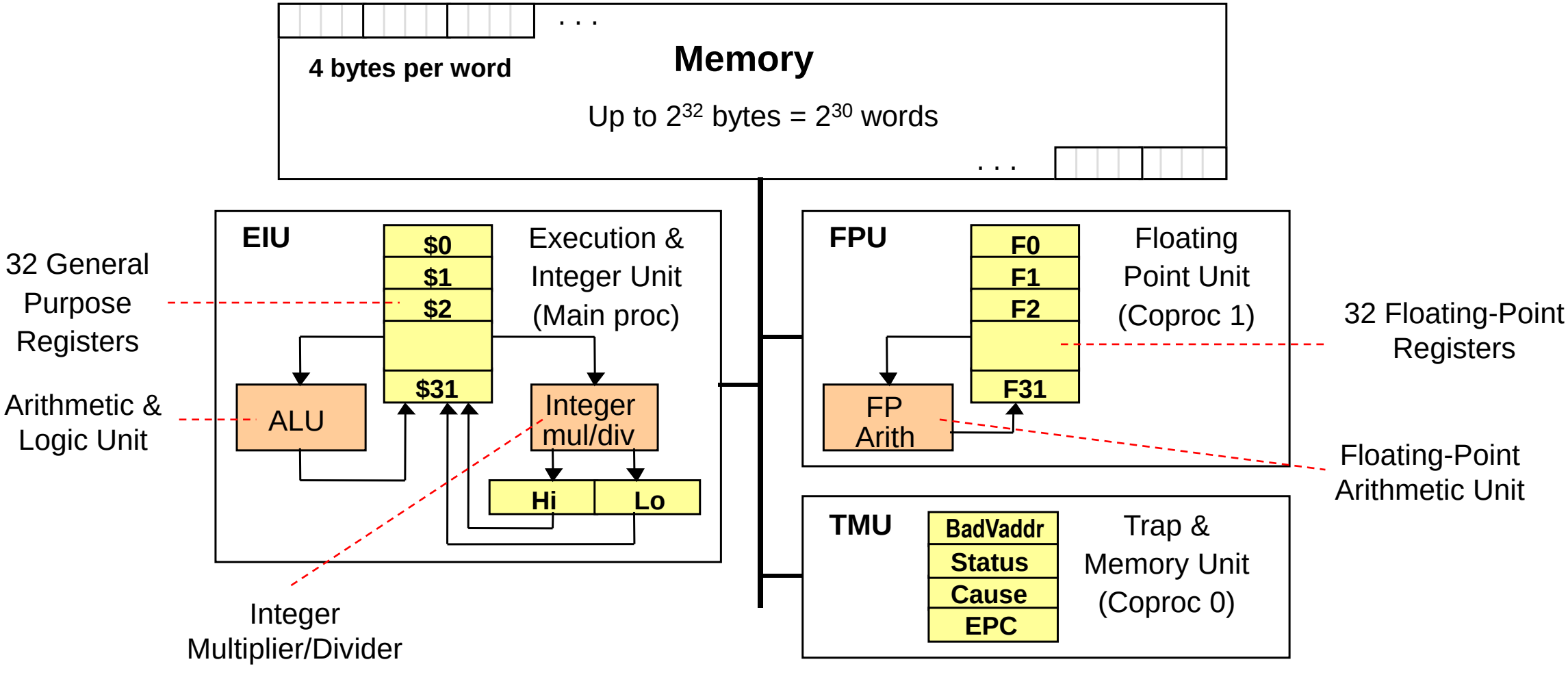
MIPS R3000 Assembly Language

Abdelhafid Boussouf University Center
2025-2026

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

Overview of the MIPS Architecture



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

.data

```
-----------------
-----------------
```

Data section

.text

```
-----------------
-----------------
li $v0, 10 syscall
```

```
# main program entry

# Exit program
```

Text section

.DATA & .TEXT 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

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)

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

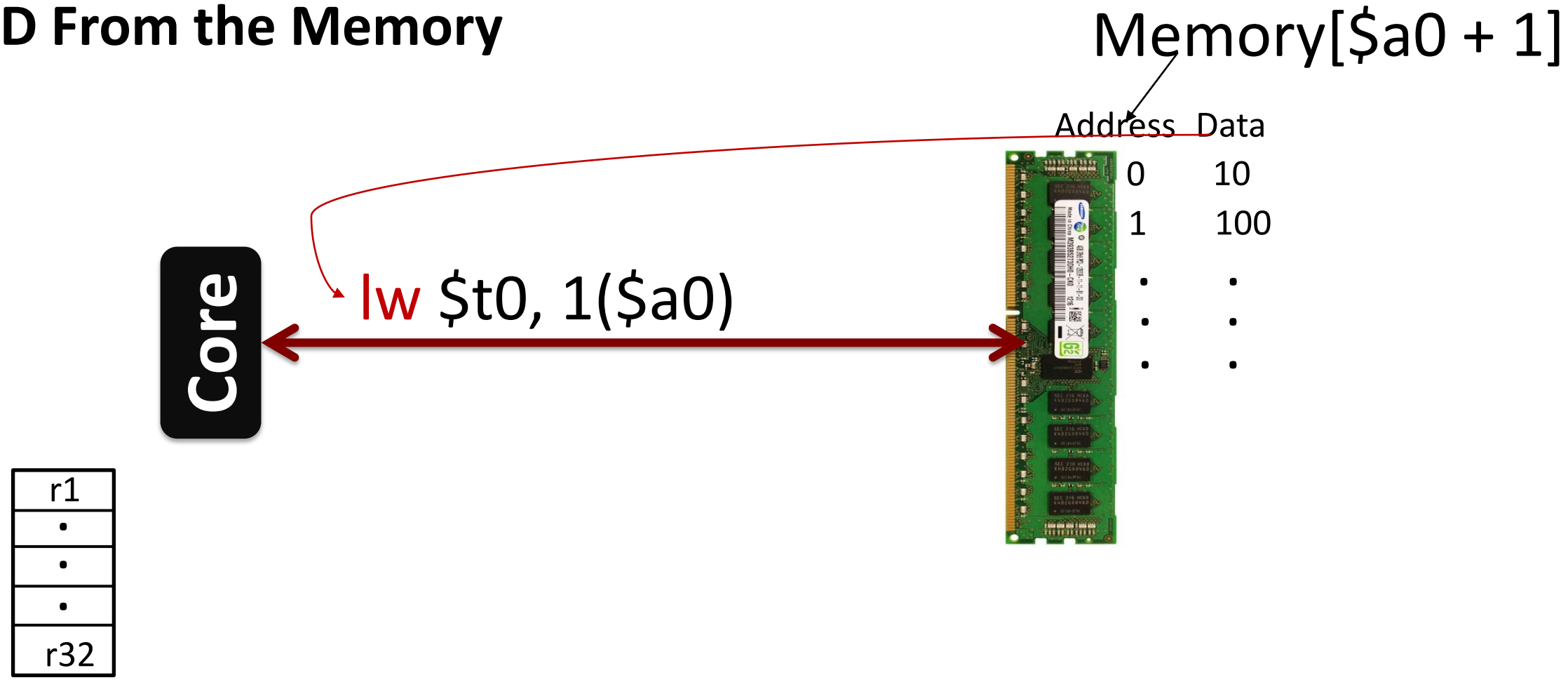
LOAD /STORE Instructions

load word	lw
load byte	lb
load byte unsigned	lbu
load half	lh
load half unsigned	lhu

load immediate	li
load address	la

LOAD /STORE Instructions

LOAD From the Memory



LOAD /STORE Instructions

LOAD From the Memory

- **Memory reads are called loads**
- **Mnemonic: load word (lw)**

Example: read a word of data at memory address 1 into $s3

- **Memory address calculation:**

  add the base address ($0) to the offset (1)

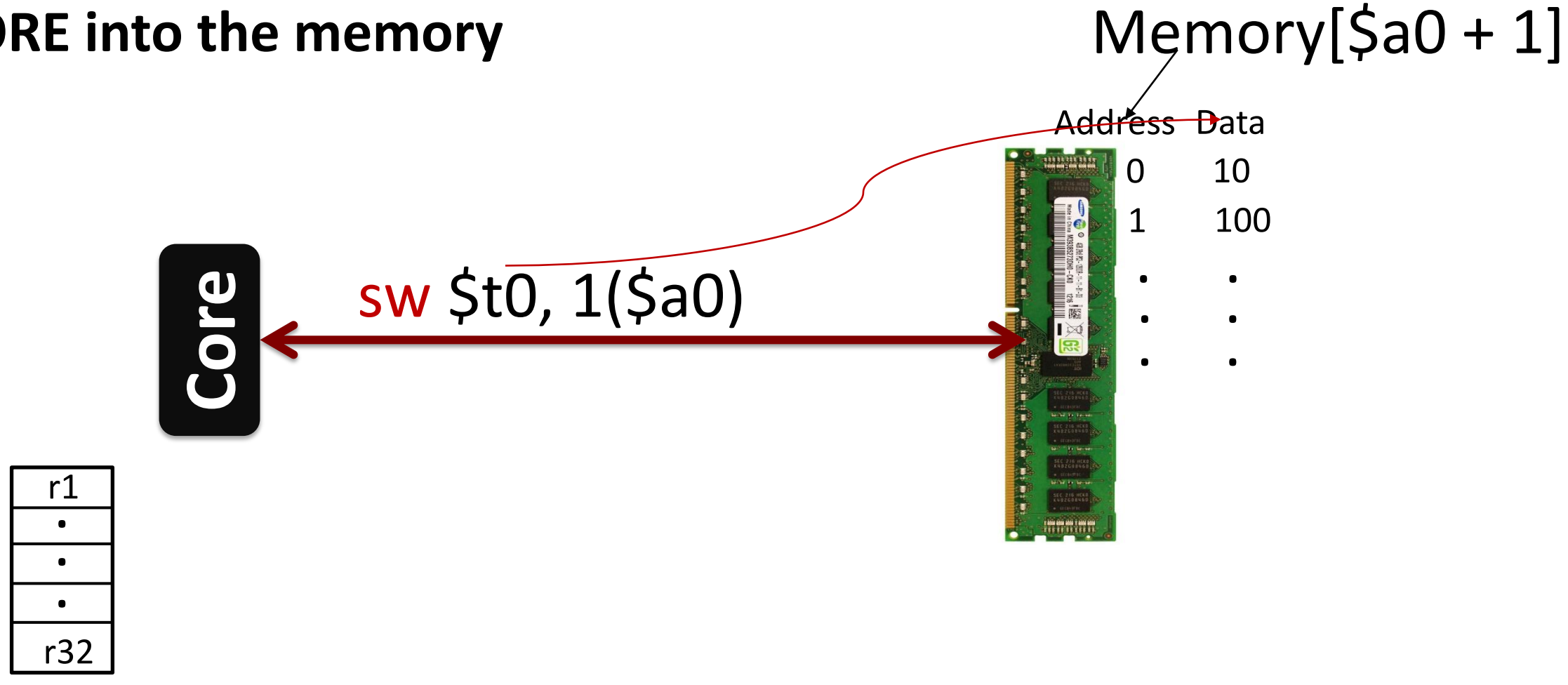
  - address = ($0 + 1) = 1
  - `$s3` holds the value **`0xF2F1AC07`** after the instruction completes

- **Any register may be used to store the base address**

```
lw $s3, 1($0)  # read memory word 1 into $s3
```

LOAD /STORE Instructions

STORE into the memory



LOAD /STORE Instructions

STORE into the memory

- **Memory writes are called stores**
  - **Mnemonic: store word (sw)**
  - **Example: Write (store) the value held in `$t4` into memory address 7**
  - **Memory address calculation:**
    - add the base address ($0) to the offset (7)
    - address = ($0 + 7) = 7
    - Offset can be written in decimal (default) or hexadecimal
  - **Any register may be used to store the base address**

```
sw $t4, 0x7($0)  # write the value
                          # to memory word 7
```

LOAD /STORE Instructions

STORE into the memory

- Li - Load immediate -

  Li Rdest, Imm

  Exemple:

  Li $t0, 23

- La - Load address-
- **La Rdest, adress**
- Copy of Register

  Move Rdest, Rsrc

```
li    $t0, 42          # $t1 =42
move  $t1, $t0
```

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	

Reading and Printing an Integer

```
###################### Code segment ##########################
.text
  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
  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
```

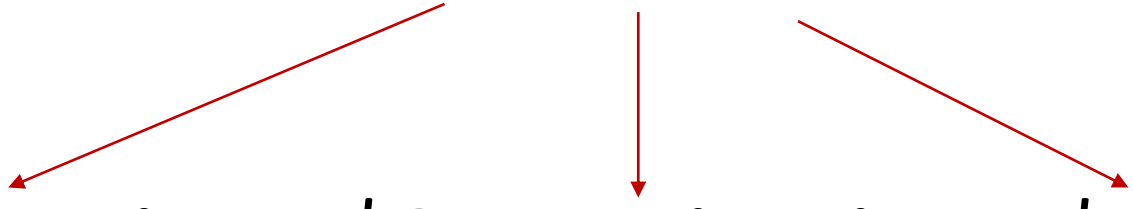
Arithmetic instructions

Instruction	Meaning
`add $t1, $t2, $t3`	`$t1 = $t2 + $t3`
`addu $t1, $t2, $t3`	`$t1 = $t2 + $t3`
`sub $t1, $t2, $t3`	`$t1 = $t2 - $t3`
`subu $t1, $t2, $t3`	`$t1 = $t2 - $t3`

- 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**

Arithmetic instructions

add $0, $1, $2



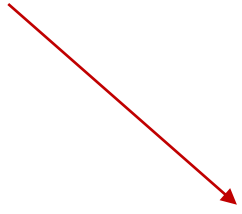
add: operation, $0: Destination, $1 & $2: Source(s)

Most of the arithmetic/logical: two sources and one destination

Arithmetic instructions

Constants and Immediate

x=x+10



No need of a register

addi $s0, $s0, 10

i: immediate, for constants

constant: **16 bits**.

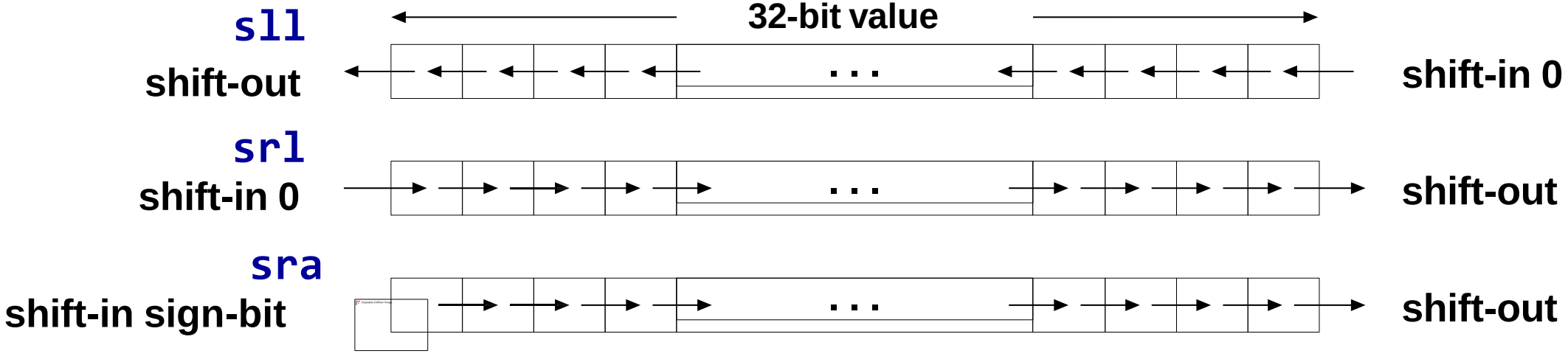
Arithmetic instructions

- 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)
```

Shift Instructions

- 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



Logic Bitwise Instructions

Instruction	Meaning
`and $t1, $t2, $t3`	`$t1 = $t2 & $t3`
`or $t1, $t2, $t3`	`$t1 = $t2 | $t3`
`xor $t1, $t2, $t3`	`$t1 = $t2 ^ $t3`
`nor $t1, $t2, $t3`	`$t1 = ~($t2|$t3)`

- 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
```

Branching

- **Allows a program to execute instructions out of sequence**
- ***Conditional branches***
  - branch if equal: **`beq`**
  - branch if not equal: **`bne`**
- ***Unconditional branches***
  - jump: **`j,b`**
  - jump register: **`jr`**
  - jump and link: **`jal`**

Conditional Branching

beq $s0, $s1, label	if $s0==$s1	goto label
bne $s0, $s1, label	if $s0!=$s1	goto label
bge $s0, $s1, label	if $s0>=$s1	goto label
bgt $s0, $s1, label	if $s0> $s1	goto label
ble $s0, $s1, label	if $s0<=$s1	goto label
blt $s0, $s1, label	if $s0<$s1	goto label
bgez $s0, label	if $s0>=0	goto label
bgtz $s0, label	if $s0> 0	goto label
blez $s0, label	if $s0<=0	goto label
bltz $s0, label	if $s0< 0	goto label
bnez $s0, label	if $s0!=0	goto label
beqz $s0, label	if $s0==0	goto label

Conditional Branching (beq)

```
# MIPS assembly
 addi   $s0,  $0, 4
 addi   $s1,  $0, 1
 sll    $s1,  $s1,   2
 beq    $s0,  $s1,   target
 addi   $s1,  $s1,   1
 sub    $s1,  $s1,   $s0

target:
 add  $s1, $s1, $s0
```

Blackboard

Labels indicate instruction locations in a program. They cannot use reserved words and must be followed by a colon (:).

Conditional Branching (beq)

```
# MIPS assembly
 addi    $s0,  $0, 4                 #  $s0  =  0 + 4 = 4
 addi    $s1,  $0, 1                 #  $s1  =  0 + 1 = 1
 sll     $s1,  $s1,   2              #  $s1  =  1 << 2 = 4
 beq     $s0,  $s1,   target         # branch is taken
 addi    $s1,  $s1,   1              # not executed
 sub     $s1,  $s1,   $s0            # not executed

target: add                         #  label
       $s1,     $s1,   $s0          #  $s1 = 4 + 4 = 8
```

Labels indicate instruction locations in a program. They cannot use reserved words and must be followed by a colon (:).

The Branch Not Taken (bne)

```
#  MIPS   assembly
   addi     $s0, $0, 4                  #  $s0  =  0 + 4 = 4
   addi     $s1, $0, 1                  #  $s1  =  0 + 1 = 1
   sll      $s1, $s1,     2             #  $s1  =  1 << 2 = 4
   bne    $s0, $s1, target    # branch not taken
   addi   $s1,       $s1,  1            #  $s1  =  4  +  1  =  5
   sub    $s1,       $s1,  $s0          #  $s1  =  5  -  4  =  1

target:
   add    $s1,       $s1,  $s0          #  $s1  =  1  +  4  =  5
```

Unconditional Branching / Jumping (j)

```
#  MIPS   assembly
        addi   $s0, $0,       4          # $s0 = 4
        addi   $s1, $0,       1         #  $s1 = 1
        j      target                    #  jump to target
      sra          $s1,   $s1,  2        #  not  executed
      addi         $s1,   $s1,  1        #  not  executed
      sub          $s1,   $s1,    $s0    #  not  executed

  target:
      add          $s1,   $s1,    $s0    #  $s1  = 1 + 4 = 5
```

Unconditional Branching (jr)

```
# MIPS assembly
0x00002000      addi   $s0,    $0,  0x2010        # load 0x2010 to $s0
0x00002004      jr     $s0                        # jump to $s0
0x00002008      addi   $s1,    $0,  1             # not executed
0x0000200C      sra    $s1,   $s1, 2             # not executed
0x00002010      lw     $s3,   44($s1)            # program continues
```

High-Level Code Constructs

- `if` statements
- `if/else` statements
- `while` loops
- `for` loops

If Statement

High-level code

```
if (i == j) f =
  g + h;

f = f - i;
```

MIPS assembly code

```
# $s0 = f, $s1 = g, $s2 = h #
$s3 = i, $s4 = j
```

If / Else Statement

High-level code

```
if (i   == j)
    f = g + h;
else
    f = f  -  i;
```

MIPS assembly code

```
#  $s0  = f, $s1    = g,   $s2 = h
#  $s3  = i, $s4    = j
        bne $s3,    $s4,   L1
        add $s0,    $s1,   $s2
        j   done
L1:     sub  $s0,   $s0,   $s3
done:
```

- **Notice that the assembly tests for the opposite case (i != j) than the test in the high-level code (i == j)**

While Loops

High-level code

```
// determines the power
// of x    such that     2x = 128
int pow    = 1;
int x      = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x
```

While Loops

High-level code

```
// determines the power
// of x such that 2x = 128
int pow = 1;
int x   = 0;

while (pow != 128) { pow
  = pow * 2;
  x = x + 1;
}
```

MIPS assembly code

```
# $s0 = pow, $s1 = x

        addi $s0, $0, 1 add
             $s1, $0, $0 addi
        $t0, $0, 128
while:  beq  $s0, $t0, done sll
             $s0, $s0, 1 addi
        $s1, $s1, 1
        j    while
done:
```

- **Notice that the assembly tests for the opposite case (pow == 128) than the test in the high-level code (pow != 128)**

For Loops

The general form of a for loop is:

```
for (initialization; condition; loop operation)
  loop body
```

- **`initialization`: executes before the loop begins**
- **`condition`: is tested at the beginning of each iteration**
- **`loop operation`: executes at the end of each iteration**
- **`loop body`: executes each time the condition is met**

For Loops

High-level code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i = 0; i != 10; i = i+1) { sum
  = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
```

For Loops

High-level code

```
// add the numbers from 0 to 9
int sum = 0;
int i;

for (i = 0; i != 10; i = i+1) { sum
  = sum + i;
}
```

MIPS assembly code

```
# $s0 = i, $s1 = sum
        addi $s1, $0, 0 add
             $s0, $0, $0
        addi $t0, $0, 10
for:    beq  $s0, $t0, done add
             $s1, $s1, $s0 addi
        $s0, $s0, 1
        j    for
done:
```

- **Notice that the assembly tests for the opposite case (i == 10) than the test in the high-level code (i != 10)**

Less Than Comparisons

High-level code

```
// add the powers       of  2 from  1
// to 100
int sum = 0;
int i;

for (i = 1;      i < 101;    i =  i*2)   {
  sum = sum      + i;
}
```

MIPS assembly code

```
#  $s0       = i, $s1 = sum
             addi $s1, $0,   0
          addi $s0,      $0,  1
          addi $t0,      $0,  101
loop:     slt    $t1,    $s0, $t0
          beq    $t1,    $0, done
          add    $s1,    $s1, $s0
          sll j  $s0,    $s0, 1
                 loop
done:
```

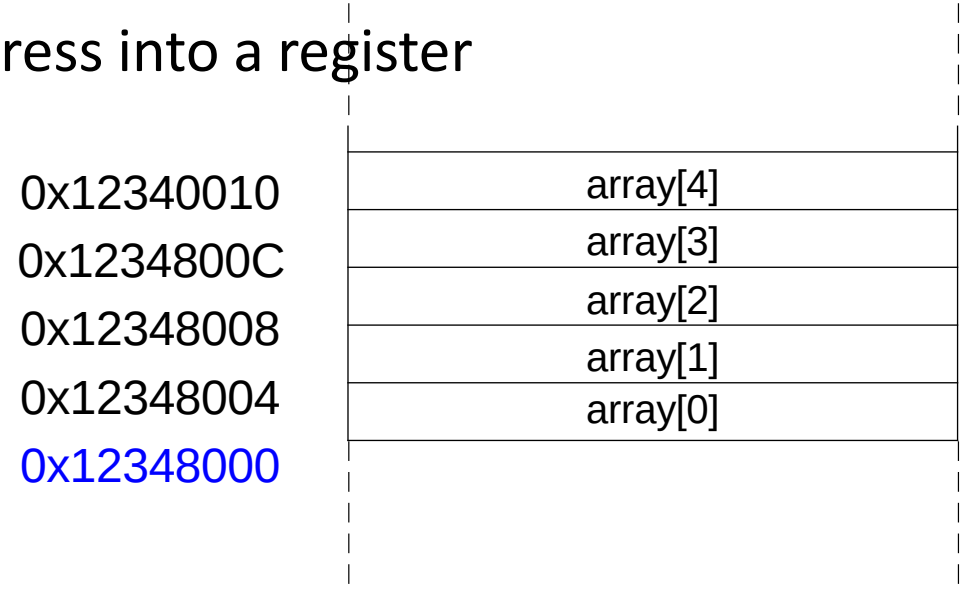
- **$t1 = 1 if i < 101**

Arrays

- Useful for accessing large amounts of similar data
- Array element: accessed by index
- Array size: number of elements in the array

Arrays

- **5-element array**
- **Base address = `0x12348000`**
  (address of the first array element, array[0])
- **First step in accessing an array:**
  - Load base address into a register



Arrays

High-level code

```
// high-level code
  int array[5];
  array[0] = array[0] * 2;
  array[1] = array[1] * 2;
```

MIPS Assembly code

```
# MIPS assembly code
# array base address = $s0

# Initialize $s0 to 0x12348000
```

Arrays

High-level code

```
// high-level code
  int array[5];
  array[0] = array[0] * 2;
  array[1] = array[1] * 2;
```

MIPS Assembly code

```
# MIPS assembly code
# array base address = $s0

# Initialize $s0 to 0x12348000
 lui    $s0,  0x1234          #  upper   $s0
 ori    $s0,  $s0, 0x8000     #  lower   $s0
```

Arrays

High-level code

```
// high-level code
  int array[5];
  array[0] = array[0] * 2;
  array[1] = array[1] * 2;
```

MIPS Assembly code

```
# MIPS assembly code
# array base address = $s0

# Initialize $s0 to 0x12348000
lui  $s0, 0x1234          # upper $s0
ori   $s0, $s0, 0x8000 # lower $s0

lw     $t1,   0($s0)      #  $t1=array[0]
sll    $t1,   $t1, 1      #  $t1=$t1*2
sw     $t1,   0($s0)      #  array[0]=$t1

lw      $t1,  4($s0)       # $t1=array[1]
sll     $t1,  $t1, 1       # $t1=$t1*2
sw      $t1,  4($s0)       # array[1]=$t1
```

Arrays Using For Loops

High-level code

```
// high-level code int
arr[1000];  int i;

for (i = 0; i < 1000; i = i + 1)
    arr[i] = arr[i] * 8;
```

MIPS Assembly code

```
# $s0 = array base, $s1 = i
lui  $s0, 0x23B8          # upper $s0
ori   $s0, $s0, 0xF000 # lower $s0
```

Arrays Using For Loops

High-level code

```
// high-level code int
arr[1000];  int i;

for (i = 0; i < 1000; i = i + 1)
    arr[i] = arr[i] * 8;
```

MIPS Assembly code

```
# $s0 = array base, $s1 = i
lui  $s0, 0x23B8          # upper $s0
ori   $s0, $s0, 0xF000 # lower $s0

addi $s1, $0, 0     # i = 0
addi $t2, $0, 1000 # $t2 = 1000

Loop:
slt  $t0, $s1, $t2 # i < 1000?
beq    $t0,  $0, done     # if not done
sll    $t0,  $s1, 2       # $t0=i * 4
add    $t0,  $t0, $s0     # addr of arr[i]
lw     $t1,  0($t0)       # $t1=arr[i]
sll    $t1,  $t1, 3       # $t1=arr[i]*8
sw     $t1,  0($t0)       # arr[i] = $t1
addi  $s1,   $s1, 1       # i = i + 1
j      Loop               # repeat
done:
```

Procedures

- **Definitions**
  - ***Caller:*** calling procedure (in this case, main)
  - ***Callee:*** called procedure (in this case, sum)

```
// High level code
void main()
{
  int y;
  y = sum(42, 7);
  ...
}

int sum(int a, int b)
{
  return (a + b);
}
```

Procedure Calling Conventions

- **Caller:**
  - passes arguments to callee
  - jumps to the callee
- **Callee:**
  - performs the procedure
  - returns the result to caller
  - returns to the point of call
  - must not overwrite registers or memory needed by the caller

MIPS Procedure Calling Conventions

- **Call procedure:**
  - jump and link (`jal`)
- **Return from procedure:**
  - jump register (`jr`)
- **Argument values:**
  - `$a0 - $a3`
- **Return value:**
  - `$v0`

Procedure Calls

High-level code

```
int main() {
  simple(); a =
  b + c;
}

void simple() {
  return;
}
```

MIPS Assembly code

```
0x00400200 main: jal simple
0x00400204       add $s0,$s1,$s2



...
0x00401020 simple: jr $ra
```

- ***void*** **means that simple doesn't return a value**

Procedure Calls

High-level code

```
int main() {
  simple(); a =
  b + c;
}

void simple() {
  return;
}
```

MIPS Assembly code

```
0x00400200 main: jal simple
0x00400204       add $s0,$s1,$s2


...
0x00401020 simple: jr $ra
```

- **`jal`: jumps to `simple` and saves PC+4 in the return address register (`$ra`)**
  - In this case, `$ra` = `0x00400204` after `jal` executes
- **`jr $ra`: jumps to address in `$ra`**
  - in this case jump to address `0x00400204`

Input Arguments and Return Values

- **MIPS conventions:**
  - Argument values: `$a0 - $a3`
  - Return value: `$v0`

Input Arguments and Return Values

```
// High-level code
int main()
{
  int y;
  ...
  // 4 arguments
  y = diffofsums(2, 3, 4, 5);
  ...
}
int diffofsums(int f, int g,
  int h, int i)
{
  int result;
  result = (f + g) - (h + i);
  return result; // return value
}
```

```
# MIPS assembly code #
$s0 = y

  main:
    ...
  addi  $a0,  $0,  2      # argument 0 = 2
  addi  $a1,  $0,  3      # argument 1 = 3
  addi  $a2,  $0,  4      # argument 2 = 4
  addi $a3, $0, 5         # argument 3 = 5 #
  jal   diffofsums        call procedure
  add   $s0, $v0, $0      # y = returned value

# $s0 = result
diffofsums:
  add $t0, $a0, $a1       # $t0 = f + g #
  add $t1, $a2, $a3       $t1 = h + i
  sub $s0, $t0, $t1       # result = (f + g) - (h + i) #
  add $v0, $s0, $0 jr     put return value in $v0
      $ra                 # return to caller
```

Input Arguments and Return Values

```
# $s0 = result
diffofsums:
   add   $t0,    $a0,  $a1     # $t0 = f + g
   add   $t1,    $a2,  $a3     # $t1 = h + i
   sub   $s0,    $t0,  $t1     # result = (f + g) - (h + i)
   add   $v0,    $s0,  $0      # put return value in $v0
   jr    $ra                   # return to caller
```

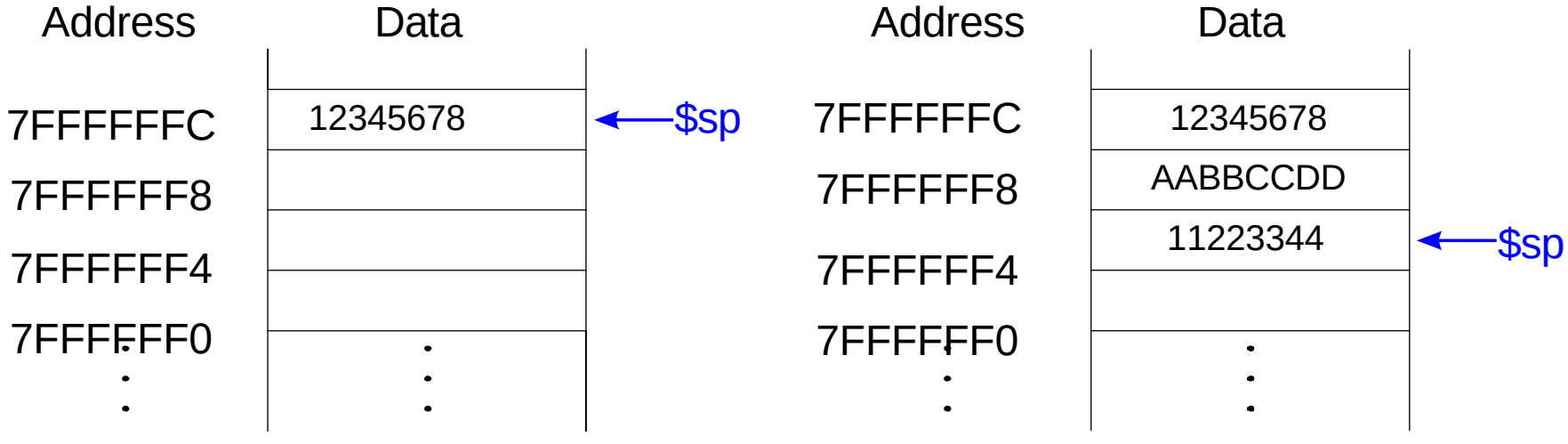
- `diffofsums` **overwrote 3 registers: `$t0`, `$t1`, and `$s0`**
- `diffofsums` **can use the *stack* to temporarily store registers (comes next)**

The Stack

- **Memory used to temporarily save variables**
- **Like a stack of dishes, last-in-first- out (LIFO) queue**
- ***Expands:*** **uses more memory when more space is needed**
- ***Contracts:*** **uses less memory when the space is no longer needed**

The Stack

- **Grows down (from higher to lower memory addresses)**
- **Stack pointer: `$sp`, points to top of the stack**



How Procedures use the Stack

- **Called procedures must have no other unintended side effects**

- **But `diffofsums` overwrites 3 registers: `$t0`, `$t1`, `$s0`**

```
# MIPS assembly # $s0 = result
diffofsums:

  add   $t0,   $a0,  $a1     # $t0 = f + g
  add   $t1,   $a2,  $a3     # $t1 = h + i
  sub   $s0,   $t0,  $t1     # result = (f + g) - (h + i)
  add   $v0,   $s0,  $0      # put return value in $v0
  jr    $ra                  # return to caller
```

Storing Register Values on the Stack

```
# $s0 = result
diffofsums:
   addi   $sp,   $sp, -12       # make space on stack
                                # to store 3 registers
   sw     $s0,   8($sp)         # save $s0 on stack
   sw     $t0,   4($sp)         # save $t0 on stack
   sw     $t1,   0($sp)         # save $t1 on stack
   add    $t0,   $a0, $a1       # $t0 = f + g
   add    $t1,   $a2, $a3       # $t1 = h + i
   sub    $s0,   $t0, $t1       # result = (f + g) - (h + i)
   add    $v0,   $s0, $0        # put return value in $v0
   lw     $t1,   0($sp)         # restore $t1 from stack
   lw     $t0,   4($sp)         # restore $t0 from stack
   lw     $s0,   8($sp)         # restore $s0 from stack
   addi   $sp,   $sp, 12        # deallocate stack space
   jr     $ra                   # return to caller
```