Control Flow and Arrays

COE 301

Computer Organization

Prof. Muhamed Mudawar

College of Computer Sciences and Engineering

King Fahd University of Petroleum and Minerals

Presentation Outline

- **Control Flow: Branch and Jump Instructions**
- **Translating If Statements and Boolean Expressions**
- Arrays
- Load and Store Instructions
- Translating Loops and Traversing Arrays
- Addressing Modes

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**:

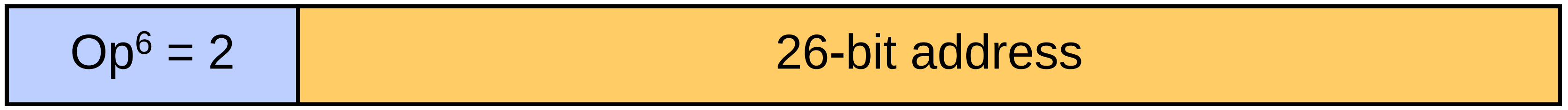
  If (branch is taken) **PC = PC + 4 + 4×offset** else **PC = PC+4**

Unconditional Jump Instruction

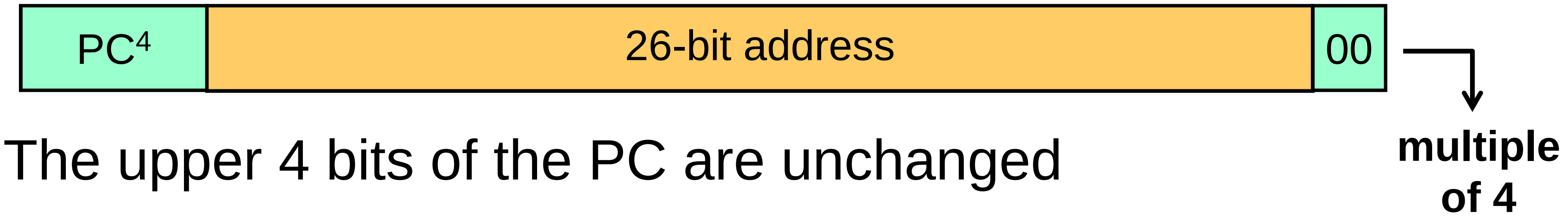
- The unconditional Jump instruction has the following syntax:

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

- The jump instruction is **always taken**
- The Jump instruction is of the J-type format:



- The jump instruction modifies the program counter PC:



- 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
sltu  Rd, Rs, Rt
slti  Rt, Rs, imm
sltiu Rt, Rs, imm
```

`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

Next . . .

- Control Flow: Branch and Jump Instructions
- Translating If Statements and Boolean Expressions
- **Arrays**
- **Load and Store Instructions**
- Translating Loops and Traversing Arrays
- Addressing Modes

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

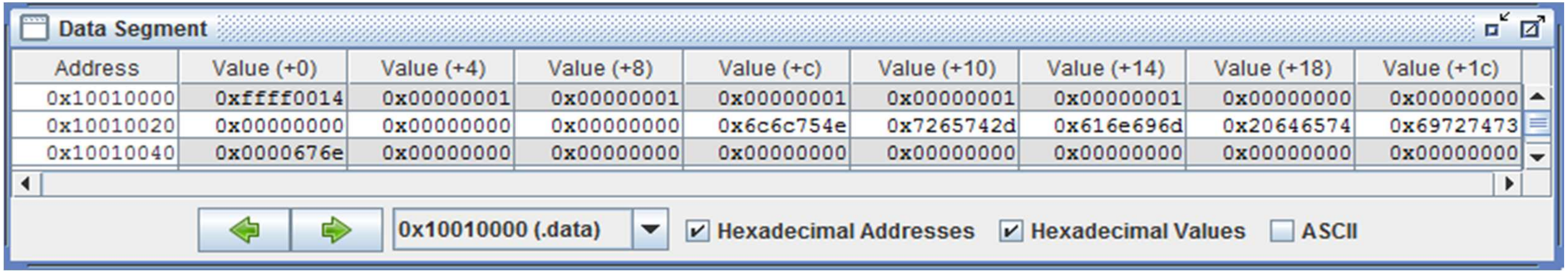
  `.type` directive specifies the size of each array element

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

- Examples of static array definitions:

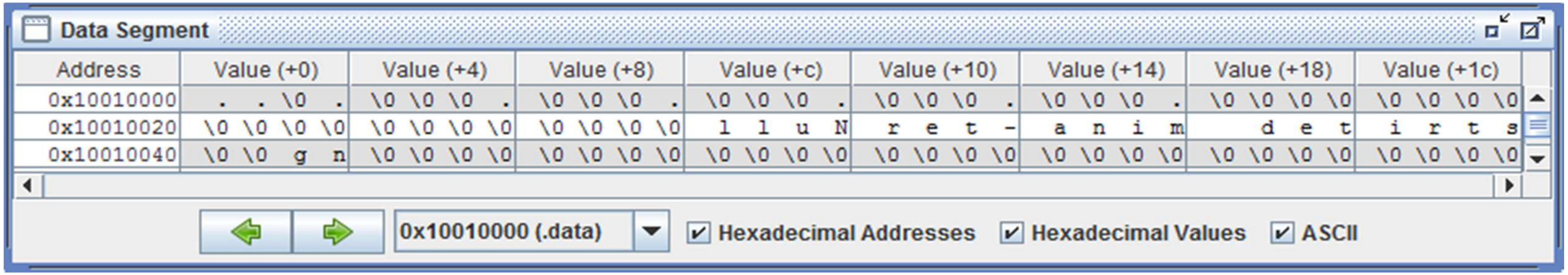
```
arr1: .half 20, -1   # array of 2 half words
arr2: .word 1:5      # array of 5 words (value=1)
arr3: .space 20      # array of 20 bytes
str1: .asciiz "Null-terminated string"
```

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

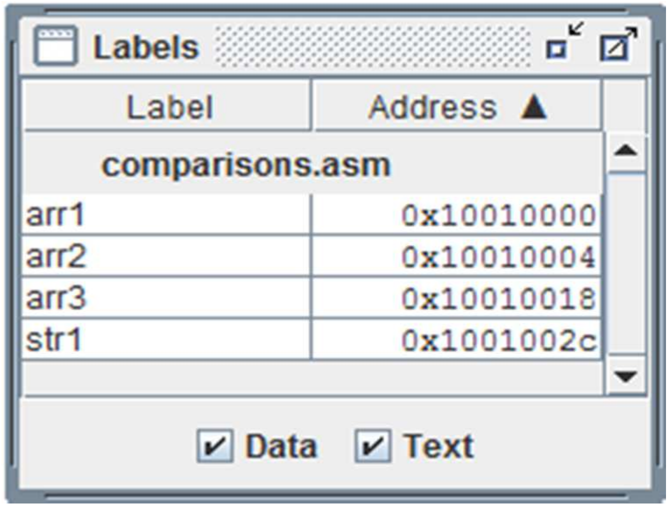
0x10010000 (.data) ✓ Hexadecimal Addresses ✓ Hexadecimal Values ☐ ASCII


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

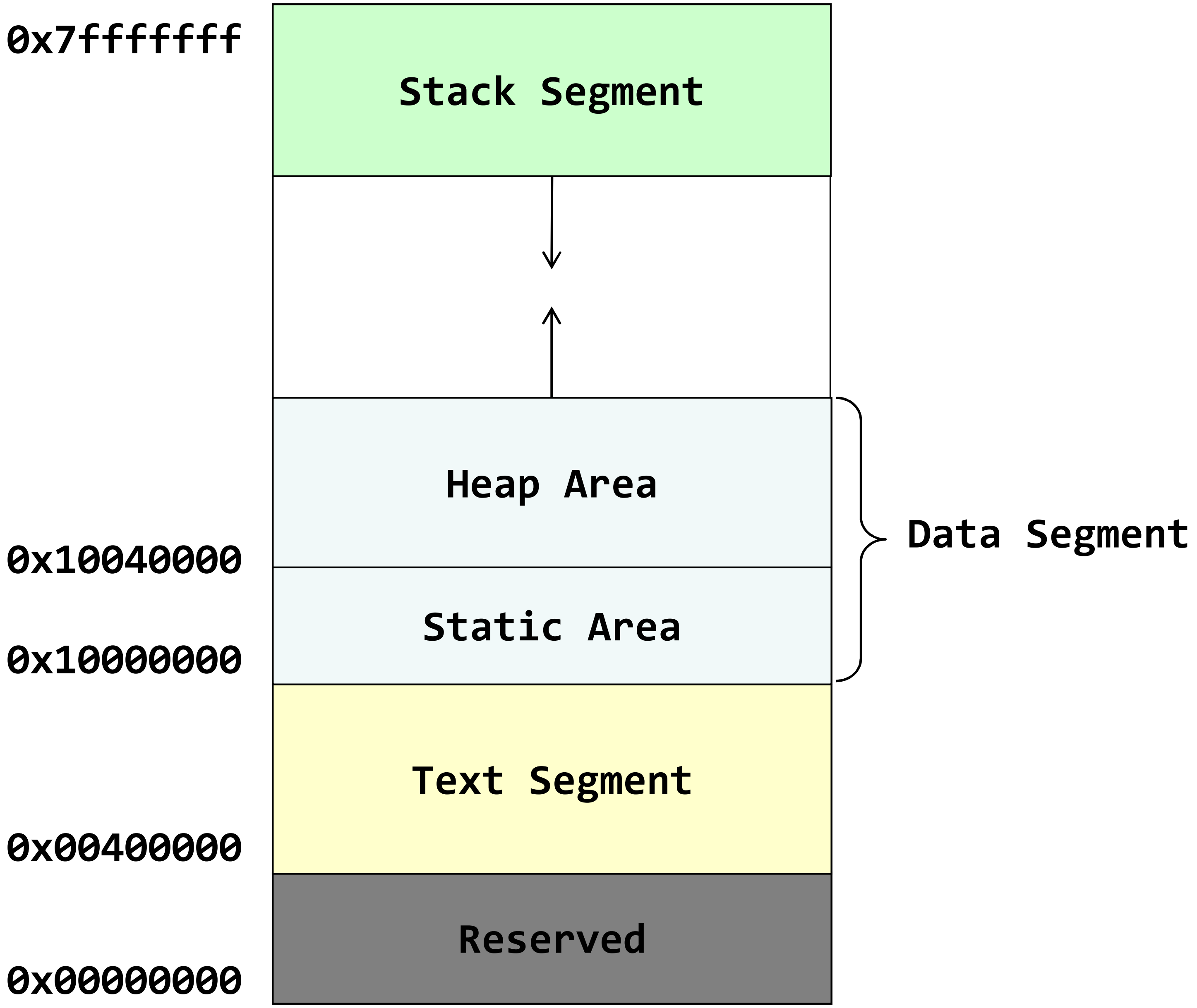
✓ Data ✓ Text

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`

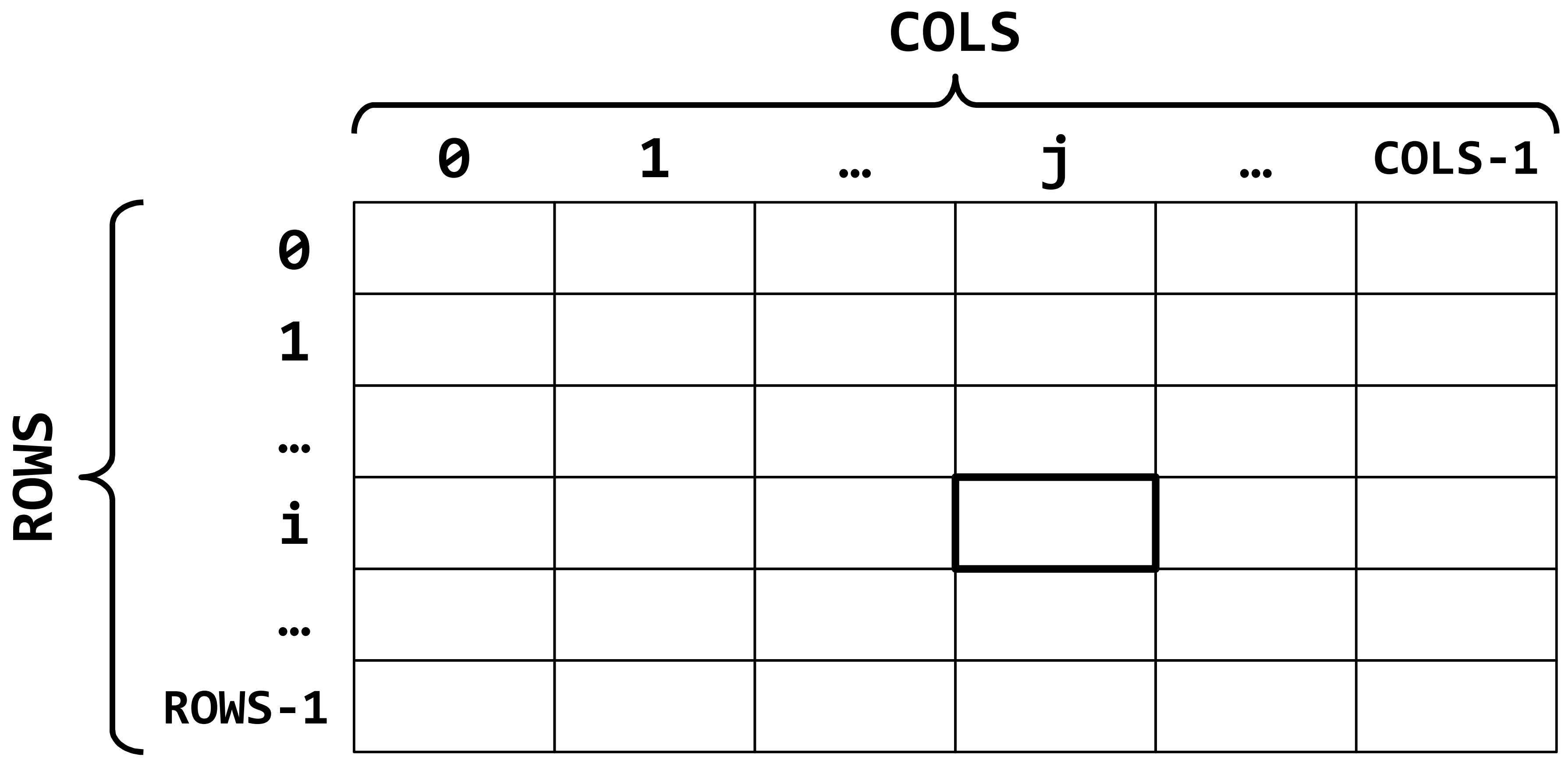
- For example, to allocate a **`matrix[10][20]`** of integers:

  `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

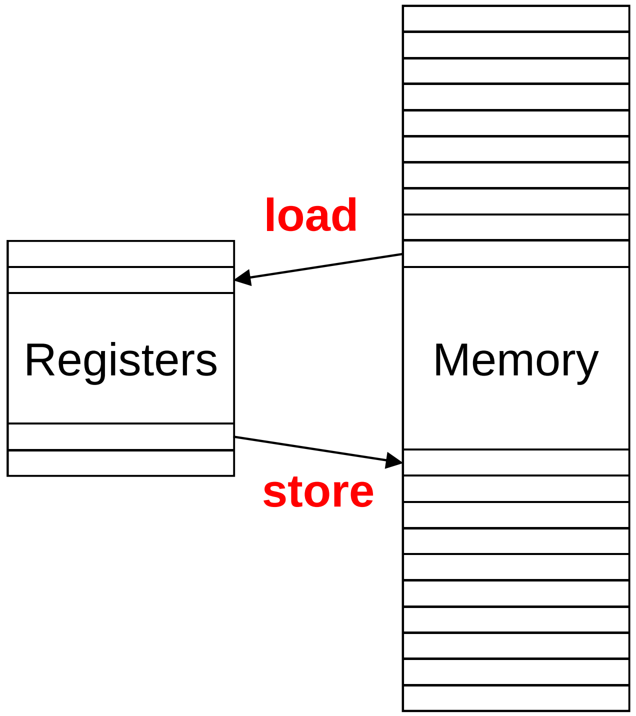
Address calculation is essential when programming in assembly



`&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)

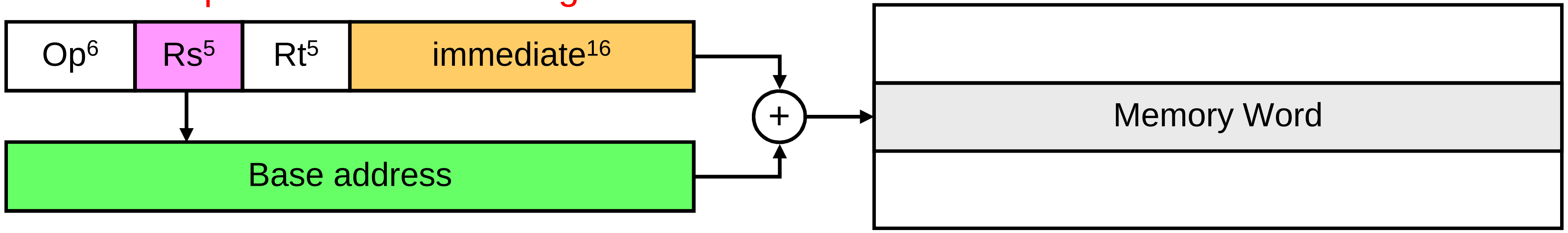
```
lw Rt, imm(Rs)     # Rt ← MEMORY[Rs+imm]
```

- 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

Base or Displacement Addressing

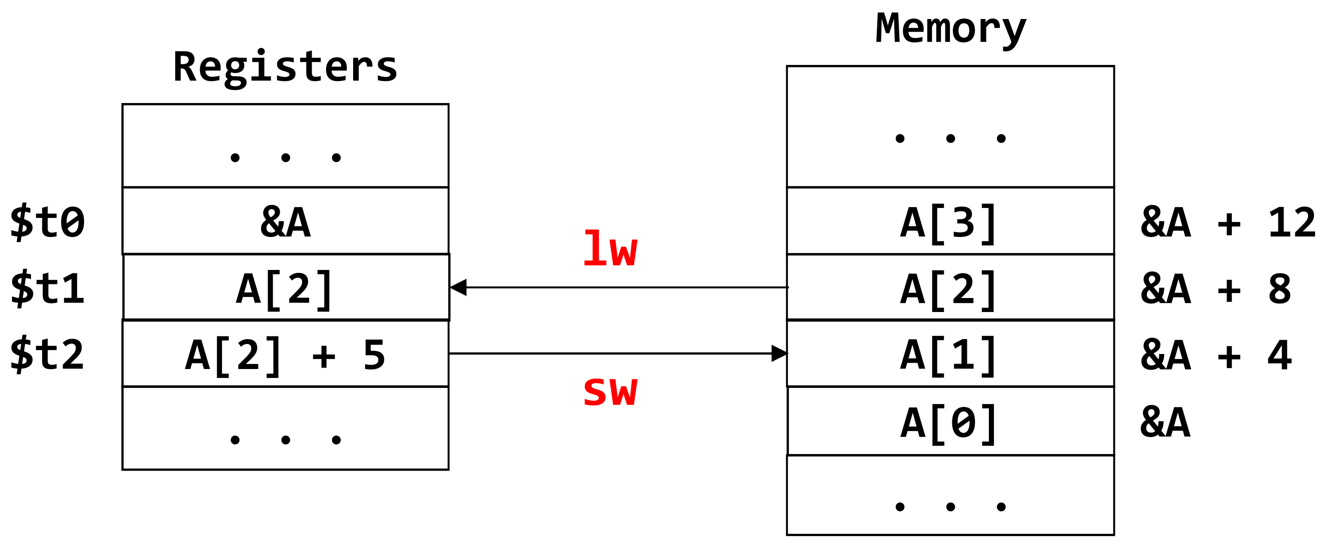


Example on Load & Store

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

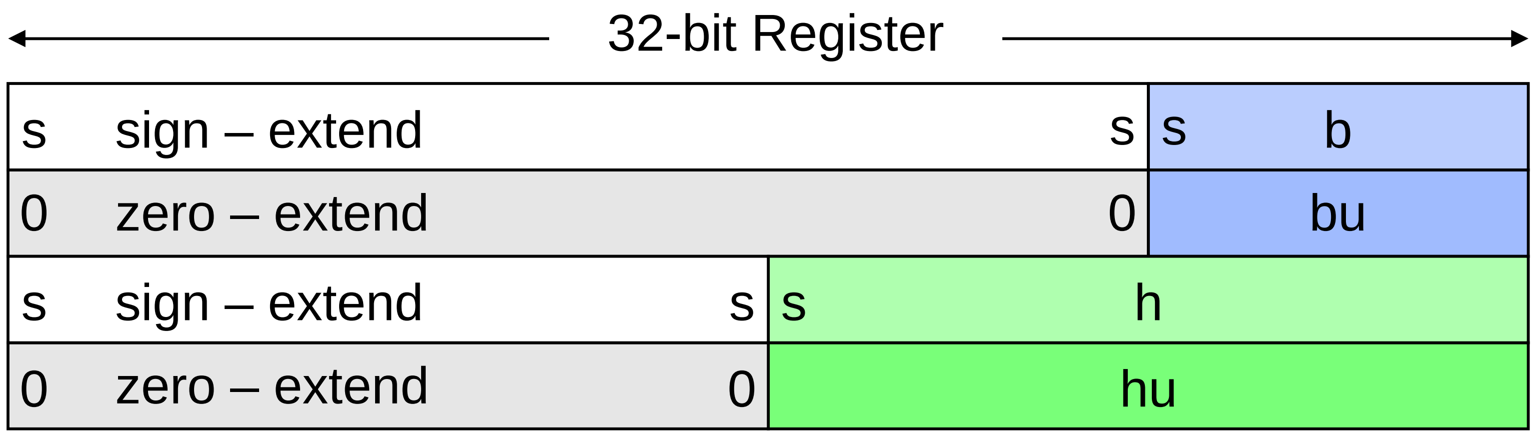
```
lw      $t1, 8($t0)     # $t1 = A[2]
addiu   $t2, $t1, 5     # $t2 = A[2] + 5
sw      $t2, 4($t0)     # A[1] = $t2
```

- 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**)

Next . . .

- ❖ Control Flow: Branch and Jump Instructions
- ❖ Translating If Statements and Boolean Expressions
- ❖ Arrays
- ❖ Load and Store Instructions
- ❖ **Translating Loops and Traversing Arrays**
- ❖ **Addressing Modes**

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
```

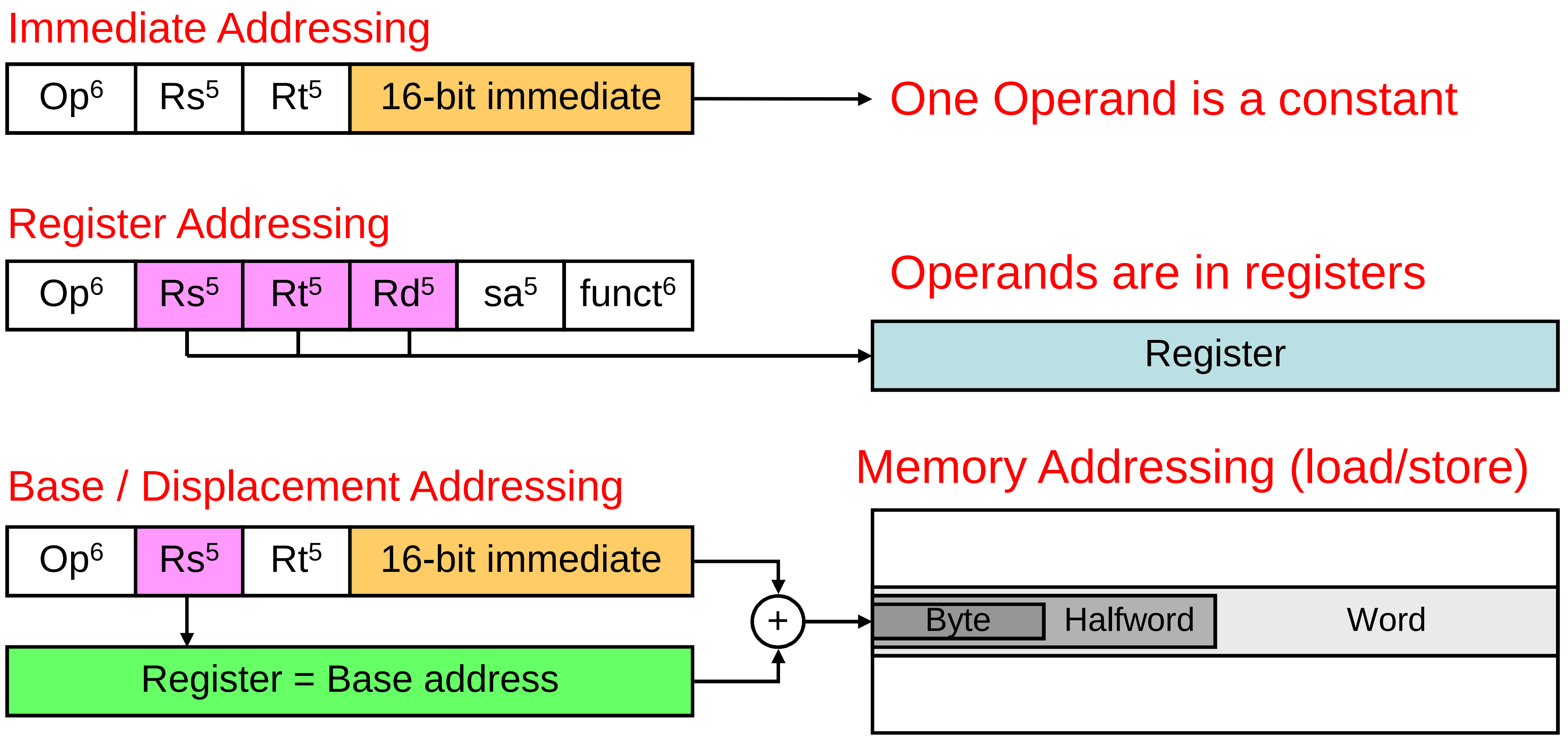
Initializing a Column of a Matrix

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

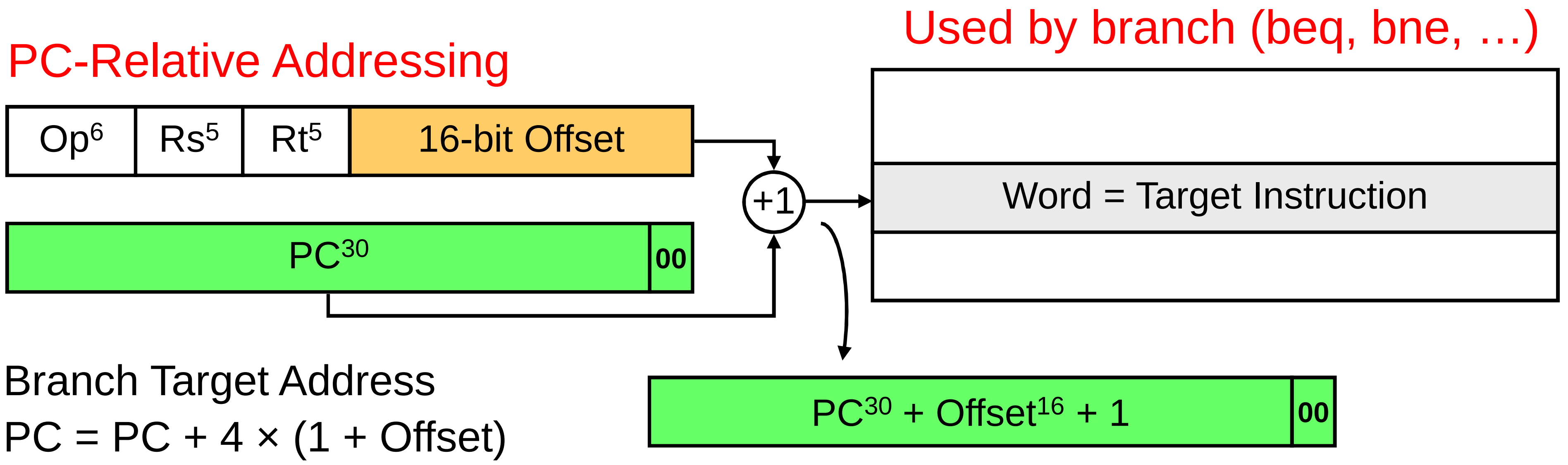
```
# &M[i][3] = &M + (i*5 + 3) * 4 = &M + i*20 + 12
   li    $a0, 200          # $a0 = 10*5*4 = 200 bytes
   li    $v0, 9            # system call 9
   syscall                 # allocate 200 bytes
   move  $t0, $v0          # $t0 = &M
   li    $t1, 0            # $t1 = i = 0
   li    $t2, 10           # $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
```

Addressing Modes

- Where are the operands?
- How memory addresses are computed?

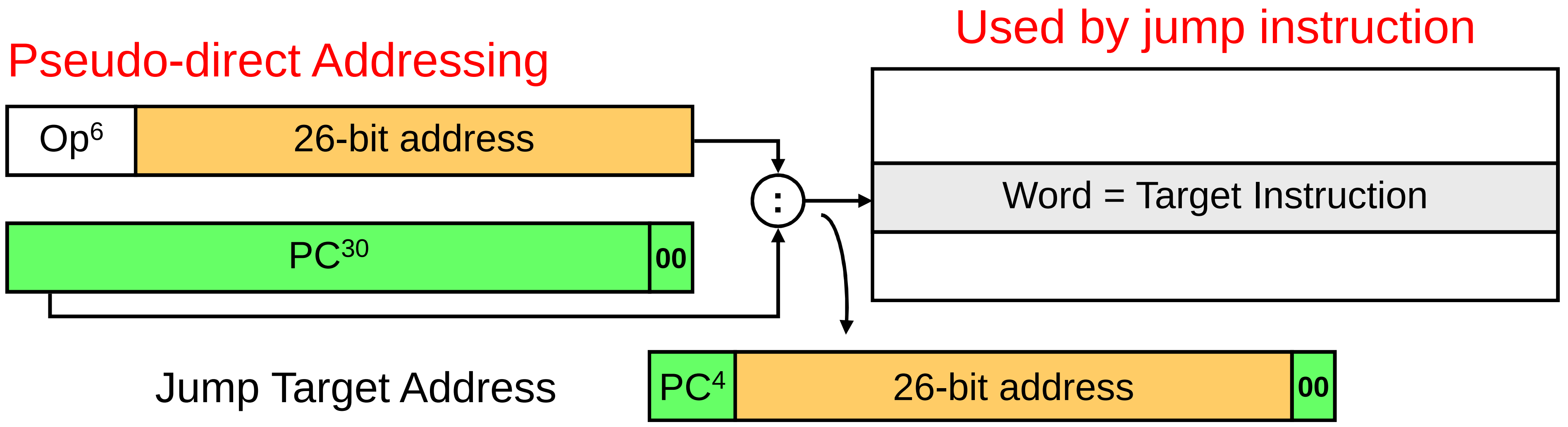


Branch / Jump Addressing Modes



Branch Target Address

PC = PC + 4 × (1 + Offset)

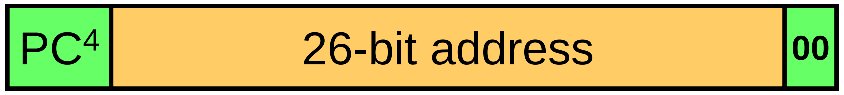


Jump Target Address

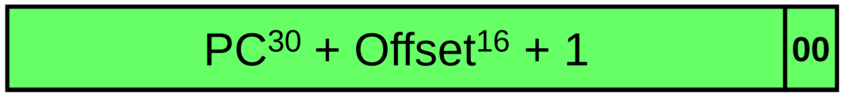
Jump and Branch Limits

- 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



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



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

Summary of RISC Design

- All instructions are of the same size
- Few instruction formats
- All arithmetic and logic operations are register to register
  - Operands are read from registers
  - Result is stored in a register
- General purpose registers for data and memory addresses
- Memory access only via load and store instructions
  - Load and store: bytes, half words, and words
- Few simple addressing modes