Computer Architecture
Tutorial 08

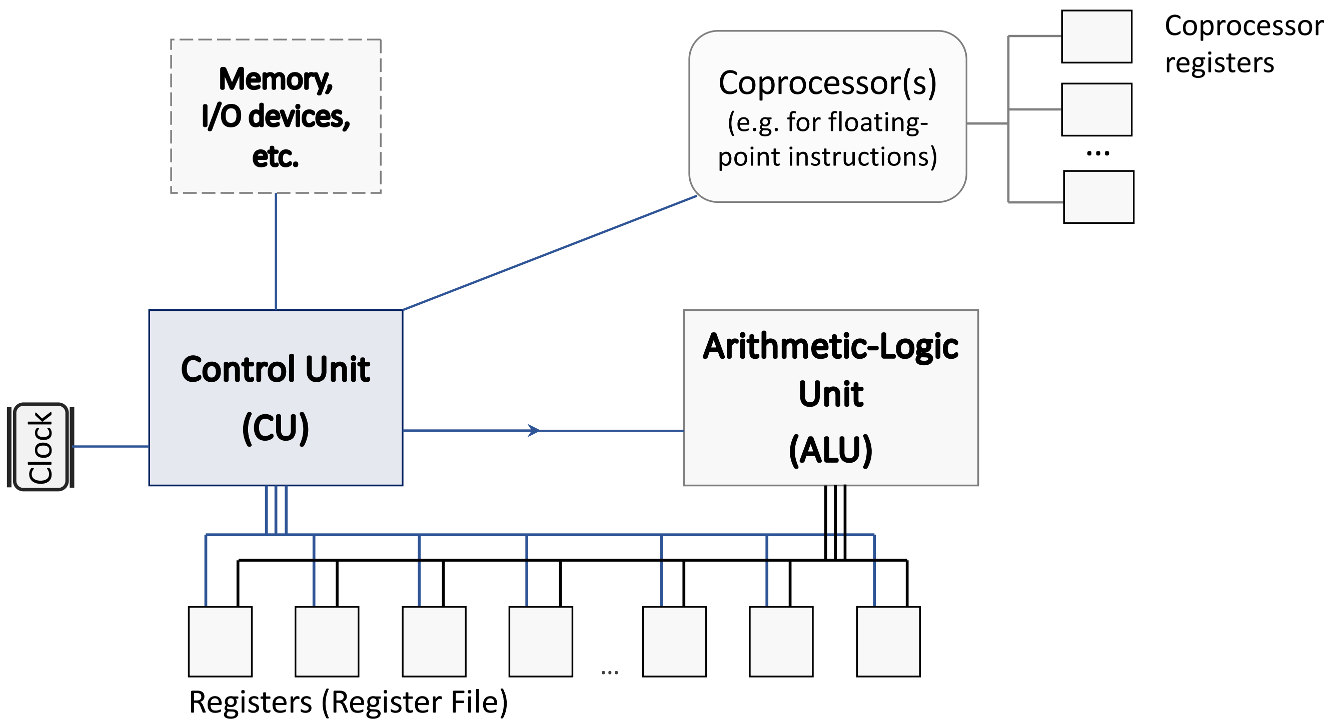
MIPS Instruction Set (Cont.)

Artem Burmyakov, Alexander Tormasov

October 14, 2021

INNOPOLIS UNIVERSITY

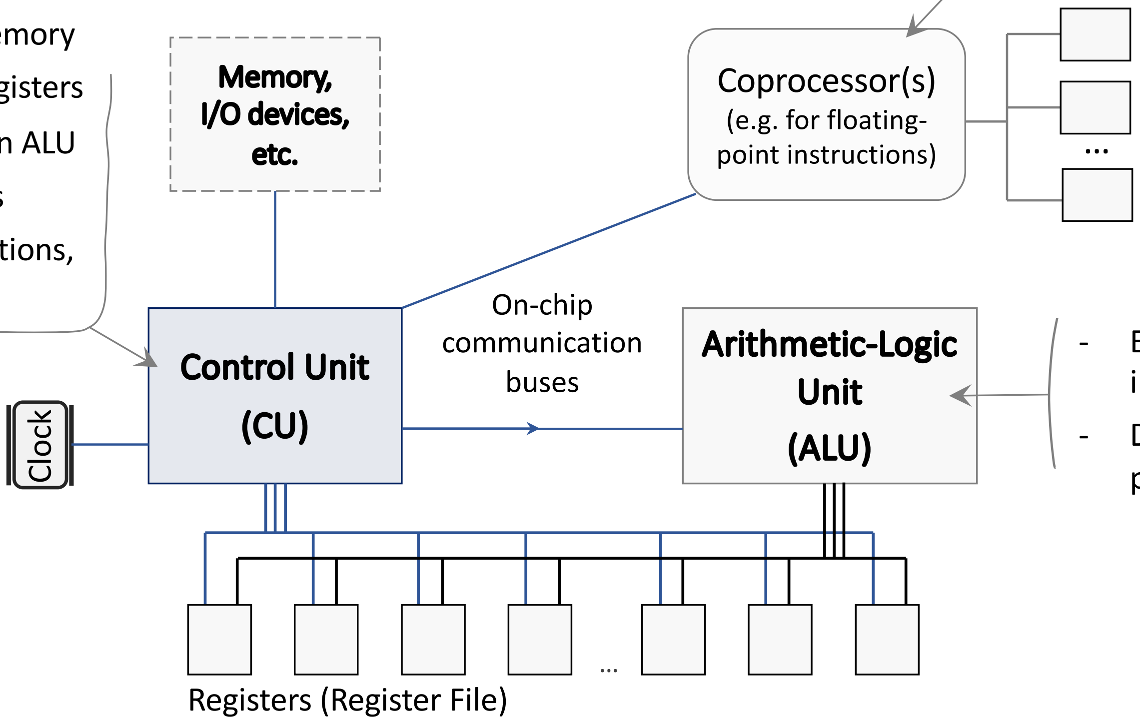
Recap: MIPS Architecture Layout



Recap: MIPS Architecture Layout

- Circuit optimized for a limited set of specific instructions (e.g. floating-point)
- Has its own registers

- Fetches instructions from memory
- Decodes instructions over registers
- Controls the execution flow in ALU
- Communicates to I/O devices
- Executes control flow instructions, e.g. jumps and branches



Coprocessor registers

- Executes arithmetic and logic instructions
- Designed for the minimized propagation delay

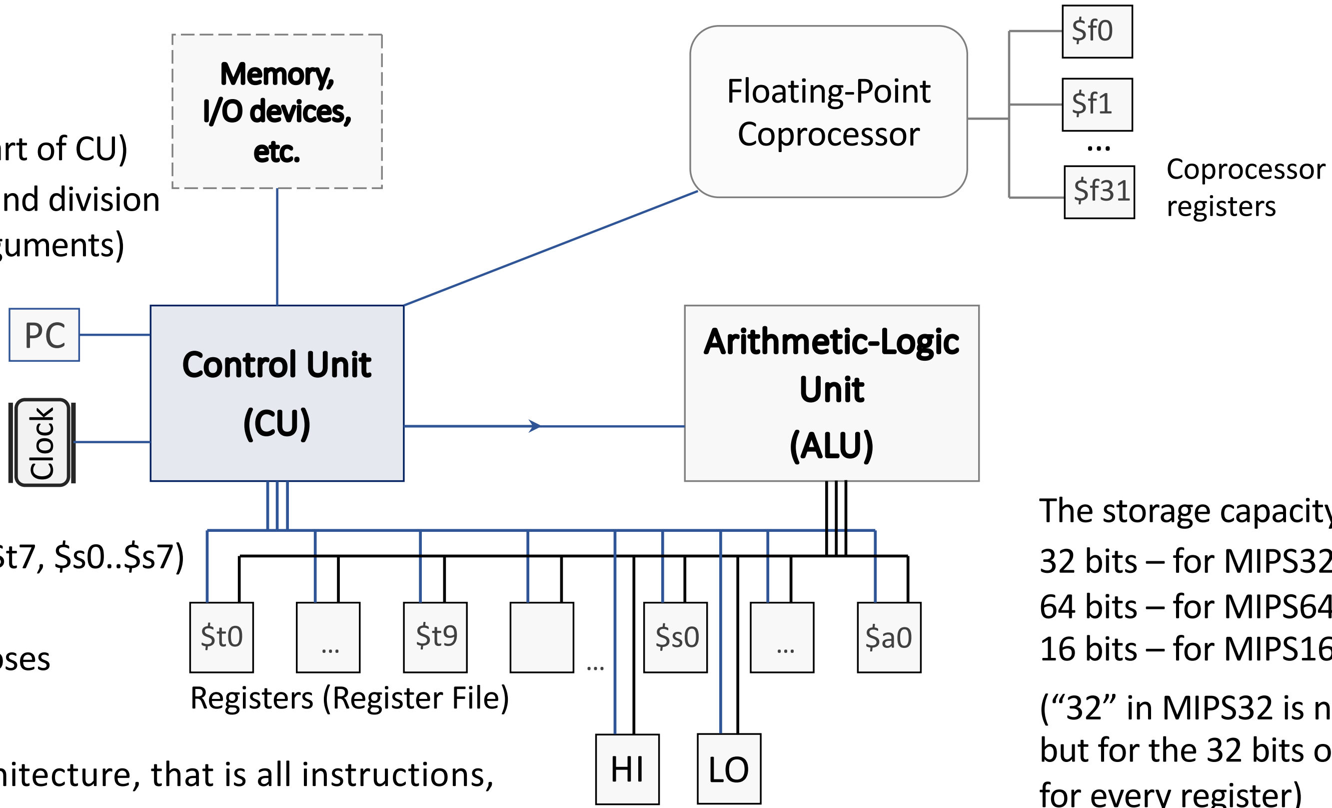
- Fast memory elements, directly connected to CU and ALU
- Store data for instruction to execute (e.g. instruction code, its input arguments), as well as the result of its execution
- Every register is reserved for a specific purpose (e.g. to store input arguments, or the result of computation)

Recap: MIPS Architecture Layout

MIPS is a RISC architecture

32 registers for floating-point instructions:
- Named by \$f0..\$f31
- Directly addressable
- Reserved for different purposes

3 special-purpose registers:
- PC – program counter (a part of CU)
- HI, LO – for multiplication and division instructions (for integer arguments)



32 general-purpose registers:
- Have their names (like \$t0..\$t7, \$s0..\$s7)
- Directly addressable
- Reserved for different purposes

The storage capacity of a register varies:

32 bits – for MIPS32;

64 bits – for MIPS64;

16 bits – for MIPS16

("32" in MIPS32 is not for 32 registers, but for the 32 bits of storage capacity for every register)

MIPS is a load/store architecture, that is all instructions, except for memory access, operate on registers

Data is retrived from these registers by using special functions mfhi and mflo

"Register spilling":
If the number of live variables exceeds the number of available registers, then the compiler spills some variables from registers into memory

Recap: 32 Directly Addressable MIPS Registers

Reg. Num	Reg. Name	Reg. Purpose
$0	**$zero**	Hardwired zero (0x00000000)
$1	**$at**	Assembler temporary
$2-$3	**$v0-$v1**	Codes of system calls; return values of system calls
$4-$7	**$a0-$a3**	Arguments for system calls
$8-$15	**$t0-$t7**	Registers for temporary values
$16-$23	**$s0-$s7**	Registers for variables
$24-$25	**$t8-$t9**	Additional registers for temporary values
$26-$27	**$k0-$k1**	Registers reserved for OS kernel
$28	**$gp**	Global pointer (to the next program instruction to be executed)
$29	**$sp**	Stack pointer
$30	**$fp**	Frame pointer
$31	**$ra**	Return address

Recap: Some MIPS Arithmetic Instructions

Arithmetic	add	**add $t0, $t1, $t2**	$t0 = $t1 + $t2
	subtract	**sub $t0, $t1, $t2**	$t0 = $t1 - $t2
	add immediate	**addi $t0, $t1, 26**	$t0 = $t1 + 26
	multiply	**mult $t1, $t2**	LO reg. – 32 least significant bits; HI reg. – 32 most significant bits
	divide integer	**div $t1, $t2**	LO – result of integer division HI reg. – division remainder

Recap: Some MIPS Arithmetic Instructions

Arithmetic	add	**add $t0, $t1, $t2**	$t0 = $t1 + $t2
	subtract	**sub $t0, $t1, $t2**	$t0 = $t1 - $t2
	add immediate	**addi $t0, $t1, 26**	$t0 = $t1 + 26
	multiply	**mult $t1, $t2**	LO reg. – 32 least significant bits; HI reg. – 32 most significant bits
	divide integer	**div $t1, $t2**	LO – result of integer division HI reg. – division remainder

Multiplication example:

```
li $t1, 13          # $t1 = 13
li $t2, 5           # $t2 = 5
mult $t1, $t2       # LO reg. contains result “65”
```

Recap: Some MIPS Arithmetic Instructions

Arithmetic	add	**add $t0, $t1, $t2**	$t0 = $t1 + $t2
	subtract	**sub $t0, $t1, $t2**	$t0 = $t1 - $t2
	add immediate	**addi $t0, $t1, 26**	$t0 = $t1 + 26
	multiply	**mult $t1, $t2**	LO reg. – 32 least significant bits; HI reg. – 32 most significant bits
	divide integer	**div $t1, $t2**	LO – result of integer division HI reg. – division remainder

Multiplication example:

```
li  $t1, 13          # $t1 = 13
li  $t2, 5           # $t2 = 5
mult  $t1, $t2       # LO reg. contains result “65”
mflo  $t0            # move result from LO reg. to $t0
```

Recap: Some MIPS Arithmetic Instructions

Arithmetic	add	**add $t0, $t1, $t2**	$t0 = $t1 + $t2
	subtract	**sub $t0, $t1, $t2**	$t0 = $t1 - $t2
	add immediate	**addi $t0, $t1, 26**	$t0 = $t1 + 26
	multiply	**mult $t1, $t2**	LO reg. – 32 least significant bits; HI reg. – 32 most significant bits
	divide integer	**div $t1, $t2**	LO – result of integer division HI reg. – division remainder

Multiplication example:

li $t1, 13	# $t1 = 13
li $t2, 5	# $t2 = 5
mult $t1, $t2	# LO reg. contains result “65”
mflo $t0	# move result from LO reg. to $t0

Note:

- $t1 and $t2 might contain values up to 32 bits;
- For large values in $t1 and $t2, mult. result might exceed 32 bits
- Then the result is stored into 2 regs.: LO and HI

Recap: Some MIPS Arithmetic Instructions

Arithmetic	add	**add $t0, $t1, $t2**	$t0 = $t1 + $t2
	subtract	**sub $t0, $t1, $t2**	$t0 = $t1 - $t2
	add immediate	**addi $t0, $t1, 26**	$t0 = $t1 + 26
	multiply	**mult $t1, $t2**	LO reg. – 32 least significant bits; HI reg. – 32 most significant bits
	divide integer	**div $t1, $t2**	LO – result of integer division HI reg. – division remainder

Multiplication example:

```
li  $t1, 13          # $t1 = 13
li  $t2, 5           # $t2 = 5
mult  $t1, $t2       # LO reg. contains result "65"
mflo  $t0            # move result from LO reg. to $t0
```

Integer division example:

```
li  $t1, 13          # $t1 = 13
li  $t2, 5           # $t2 = 5
div  $t1, $t2        # LO reg. contains div. result "2"
                     # HI reg. contains remainder "3"
```

Recap: Some MIPS Arithmetic Instructions

Arithmetic	add	**add $t0, $t1, $t2**	$t0 = $t1 + $t2
	subtract	**sub $t0, $t1, $t2**	$t0 = $t1 - $t2
	add immediate	**addi $t0, $t1, 26**	$t0 = $t1 + 26
	multiply	**mult $t1, $t2**	LO reg. – 32 least significant bits; HI reg. – 32 most significant bits
	divide integer	**div $t1, $t2**	LO – result of integer division HI reg. – division remainder

Multiplication example:

```
li $t1, 13          # $t1 = 13
li $t2, 5           # $t2 = 5
mult $t1, $t2       # LO reg. contains result “65”
mflo $t0            # move result from LO reg. to $t0
```

Integer division example:

```
li $t1, 13          # $t1 = 13
li $t2, 5           # $t2 = 5
div $t1, $t2        # LO reg. contains div. result “2”
                    # HI reg. contains remainder “3”
mflo $t0            # move result “2” from LO reg. to $t0
mfhi $t3            # move remainder “3” from HI reg. to $t3
```

Recap: Some MIPS Arithmetic Instructions

Arithmetic	add	**add $t0, $t1, $t2**	$t0 = $t1 + $t2
	subtract	**sub $t0, $t1, $t2**	$t0 = $t1 - $t2
	add immediate	**addi $t0, $t1, 26**	$t0 = $t1 + 26
	multiply	**mult $t1, $t2**	LO reg. – 32 least significant bits; HI reg. – 32 most significant bits
	divide integer	**div $t1, $t2**	LO – result of integer division HI reg. – division remainder

Multiplication example:

```
li $t1, 13          # $t1 = 13
li $t2, 5           # $t2 = 5
mult $t1, $t2       # LO reg. contains result "65"
mflo $t0            # move result from LO reg. to $t0
```

Integer division example:

```
li $t1, 13          # $t1 = 13
li $t2, 5           # $t2 = 5
div $t1, $t2        # LO reg. contains div. result "2"
                    # HI reg. contains remainder "3"
mflo $t0            # move result "2" from LO reg. to $t0
mfhi $t3            # move remainder "3" from HI reg. to $t3
```

LO reg. – the result of the division
HI reg. – the remainder

Recap: Some MIPS Arithmetic Instructions

Arithmetic	add	**add $t0, $t1, $t2**	$t0 = $t1 + $t2
	subtract	**sub $t0, $t1, $t2**	$t0 = $t1 - $t2
	add immediate	**addi $t0, $t1, 26**	$t0 = $t1 + 26
	multiply	**mult $t1, $t2**	LO reg. – 32 least significant bits; HI reg. – 32 most significant bits
	divide integer	**div $t1, $t2**	LO – result of integer division HI reg. – division remainder
Logical (bitwise, bit by bit)	and	**and $t0, $t1, $t2**	$t0 = $t1 & $t2
	or	**or $t0, $t1, $t2**	$t0 = $t1 | $t2
	nor	**nor $t0, $t1, $t2**	$t0 = ~($t1 | $t2)
	and immediate	**andi $t0, $t1, 17**	$t0 = $t1 & 17
	or immediate	**ori $t0, $t1, 13**	$t0 = $t1 | 13
	shift left logical	**sll $t0, $t1, 4**	$t0 = $t1 << 4 (equivalent to "$\times 2^4$")
	shift right logical	**srl $t0, $t1, 6**	$t0 = $t1 >> 6 (equivalent to "$\div 2^6$")

Recap: Some MIPS Arithmetic Instructions

Arithmetic	add	**add $t0, $t1, $t2**	$t0 = $t1 + $t2
	subtract	**sub $t0, $t1, $t2**	$t0 = $t1 - $t2
	add immediate	**addi $t0, $t1, 26**	$t0 = $t1 + 26
	multiply	**mult $t1, $t2**	LO reg. – 32 least significant bits; HI reg. – 32 most significant bits
	divide integer	**div $t1, $t2**	LO – result of integer division HI reg. – division remainder
Logical (bitwise, bit by bit)	and	**and $t0, $t1, $t2**	$t0 = $t1 & $t2
	or	**or $t0, $t1, $t2**	$t0 = $t1 | $t2
	nor	**nor $t0, $t1, $t2**	$t0 = ~($t1 | $t2)
	and immediate	**andi $t0, $t1, 17**	$t0 = $t1 & 17
	or immediate	**ori $t0, $t1, 13**	$t0 = $t1 | 13
	shift left logical	**sll $t0, $t1, 4**	$t0 = $t1 << 4 (equivalent to "$\times 2^4$")
	shift right logical	**srl $t0, $t1, 6**	$t0 = $t1 >> 6 (equivalent to "$\div 2^6$")

Recap: MIPS System Calls for External Interaction Available in MARS Simulator*

Code in \$v0	Purpose	Arguments/Return values
1	Print integer	\$a0 contains value to be printed
2	Print float	\$f12 contains float value to be printed
3	Print double	\$f12 contains double value to be printed
4	Print string	\$a0 contains the memory address of a null-terminated string; \$a1 – a string size in bytes (optional)
5	Read integer	\$v0 contains value, that was read
6	Read float	\$f0 contains the input float
7	Read double	\$f0 contains the input double
8	Read string	\$a0 - the address of an input buffer, where the string is stored; \$a1 – the number of characters to read
10	Exit program	none

*MARS Simulator acts similar to an Operating System in this case

Recap: MIPS System Calls for External Interaction Available in MARS Simulator

Code in $v0	Purpose	Arguments/Return values
1	Print integer	$a0 contains value to be printed
2	Print float	$f12 contains float value to be printed
3	Print double	$f12 contains double value to be printed
4	Print string	$a0 contains the memory address of a null-terminated string; $a1 – a string size in bytes (optional)
5	Read integer	$v0 contains value, that was read
6	Read float	$f0 contains the input float
7	Read double	$f0 contains the input double
8	Read string	$a0 - the address of an input buffer, where the string is stored; $a1 – the number of characters to read
10	Exit program	none

Example of string printing:

```
.data
    msg:      .asciiz "Hello!"    # string constant to be printed
.text                 # program starts next
main:
    li $v0, 4          # syscall code to print string
    la $a0, msg        # load memory address of the string beginning
    syscall            # syscall invocation, to print message
```

Recap: MIPS System Calls for External Interaction Available in MARS Simulator

Code in $v0	Purpose	Arguments/Return values
1	Print integer	$a0 contains value to be printed
2	Print float	$f12 contains float value to be printed
3	Print double	$f12 contains double value to be printed
4	Print string	$a0 contains the memory address of a null-terminated string; $a1 – a string size in bytes (optional)
5	Read integer	$v0 contains value, that was read
6	Read float	$f0 contains the input float
7	Read double	$f0 contains the input double
8	Read string	$a0 - the address of an input buffer, where the string is stored; $a1 – the number of characters to read
10	Exit program	none

More MIPS System Calls for External Interaction in MARS (just for your reference)

Code in $v0	Purpose	Arguments/Return values
1	Print integer	$a0 contains value to be printed
2	Print float	$f12 contains float value to be printed
3	Print double	$f12 contains double value to be printed
4	Print string	$a0 contains the memory address of a null-terminated string; $a1 – a string size in bytes (optional)
5	Read integer	$v0 contains value, that was read
6	Read float	$f0 contains the input float
7	Read double	$f0 contains the input double
8	Read string	$a0 - the address of an input buffer, where the string is stored; $a1 – the number of characters to read
9	Allocate Head memory	$a0 - number of bytes to allocate; $v0 – the beginning address of the allocated memory block
10	Exit program	none
11	Print character	$a0 – character to print
12	Read character	$v0 – character that was read
13, 14, 15	Open, read, and write file	$a0, $a1, $a2 (meaning differ based on a syscall code)
17	Terminate with return value	$a0 – value to return

More MIPS System Calls for External Interaction in MARS (just for your reference)

Code in $v0	Purpose	Arguments/Return values
1	Print integer	$a0 contains value to be printed
2	Print float	$f12 contains float value to be printed
3	Print double	$f12 contains double value to be printed
4	Print string	$a0 contains the memory address of a null-terminated string; $a1 – a string size in bytes (optional)
5	Read integer	$v0 contains value, that was read
6	Read float	$f0 contains the input float
7	Read double	$f0 contains the input double
8	Read string	$a0 - the address of an input buffer, where the string is stored; $a1 – the number of characters to read
9	Allocate Head memory	$a0 - number of bytes to allocate; $v0 – the beginning address of the allocated memory block
10	Exit program	none
11	Print character	$a0 – character to print
12	Read character	$v0 – character that was read
13, 14, 15	Open, read, and write file	$a0, $a1, $a2 (meaning differ based on a syscall code)
17	Terminate with return value	$a0 – value to return

Many more system calls are available (up to 59, based on the MARS version)

MIPS Program Structure

```
.data       # contains variable declarations (int. variables, strings, etc.)
```

MIPS Program Structure

```
.data
    intVar:        .word        # 1 word = 4 bytes = 32 bits
```

MIPS Program Structure

```
.data
    intVar:         .word         # 1 word = 4 bytes = 32 bits
    charVar:        .byte         # assumption: 1 char takes 1 byte
```

MIPS Program Structure

```
.data
    intVar:          .word              # 1 word = 4 bytes = 32 bits
    charVar:         .byte              # assumption: 1 char takes 1 byte
    stringConst:     .asciiz "Hello!"   # string constant
```

MIPS Program Structure

```
.data
    intVar:         .word               # 1 word = 4 bytes = 32 bits
    charVar:        .byte               # assumption: 1 char takes 1 byte
    stringConst:    .asciiz “Hello!”    # string constant
    arrayVar:       .space 20           # array of consecutive bytes; space size is in bytes; used for string as well
```

MIPS Program Structure

```
.data
    intVar:          .word              # 1 word = 4 bytes = 32 bits
    charVar:         .byte              # assumption: 1 char takes 1 byte
    stringConst:     .asciiz “Hello!”   # string constant
    arrayVar:        .space 20          # array of consecutive bytes; space size is in bytes; used for string as well
    16bitIntVar:     .half              # half = half a word = 2 bytes = 16 bits
```

MIPS Program Structure

```
.data
    intVar:         .word            # 1 word = 4 bytes = 32 bits
    charVar:        .byte            # assumption: 1 char takes 1 byte
    stringConst:    .asciiz "Hello!" # string constant
    arrayVar:       .space 20        # array of consecutive bytes; space size is in bytes; used for string as well
    16bitIntVar:    .half            # half = half a word = 2 bytes = 16 bits
    floatVar:       .float
    doubleVar:      .double
```

MIPS Program Structure

```
.data
        intVar:         .word               # 1 word = 4 bytes = 32 bits
        charVar:        .byte               # assumption: 1 char takes 1 byte
        stringConst:    .asciiz “Hello!”    # string constant
        arrayVar:       .space 20           # array of consecutive bytes; space size is in bytes; used for string as well
        16bitIntVar:    .half               # half = half a word = 2 bytes = 16 bits
        floatVar:       .float
        doubleVar:      .double
```

Other data types are available, based on the MIPS version

MIPS Program Structure

```
.data
    intVar:         .word           # 1 word = 4 bytes = 32 bits
    charVar:        .byte           # assumption: 1 char takes 1 byte
    stringConst:    .asciiz "Hello!"    # string constant
    arrayVar:       .space 20       # array of consecutive bytes; space size is in bytes; used for string as well
.text       # program text starts here
```

MIPS Program Structure

```
.data
    intVar:         .word             # 1 word = 4 bytes = 32 bits
    charVar:        .byte             # assumption: 1 char takes 1 byte
    stringConst:    .asciiz "Hello!"  # string constant
    arrayVar:       .space 20         # array of consecutive bytes; space size is in bytes; used for string as well
.text     # program text starts here
main:     # labeled block of code; can be any other name, e.g. "__start"
```

MIPS Program Structure

```
.data
    intVar:        .word             # 1 word = 4 bytes = 32 bits
    charVar:       .byte             # assumption: 1 char takes 1 byte
    stringConst:   .asciiz "Hello!"  # string constant
    arrayVar:      .space 20         # array of consecutive bytes; space size is in bytes; used for string as well
.text     # program text starts here
main:     # labeled block of code; can be any other name, e.g. "__start"
# program instructions follow
...
```

MIPS Program Structure

```
.data
    intVar:         .word           # 1 word = 4 bytes = 32 bits
    charVar:        .byte           # assumption: 1 char takes 1 byte
    stringConst:    .asciiz "Hello!"    # string constant
    arrayVar:       .space 20       # array of consecutive bytes; space size is in bytes; used for string as well
.text       # program text starts here
main:       # labeled block of code; can be any other name, e.g. "__start"
# program instructions follow
…
```

MIPS Program Structure

```
.data
    intVar:         .word           # 1 word = 4 bytes = 32 bits
    charVar:        .byte           # assumption: 1 char takes 1 byte
    stringConst:    .asciiz “Hello!”    # string constant
    arrayVar:       .space 20       # array of consecutive bytes; space size is in bytes; used for string as well
.text     # program text starts here
.globl    main      # labeled block(s) of code, to be available to other programs
main:     # labeled block of code; can be any other name, e.g. “__start”
# program instructions follow
…
```

Example: Computation of a Logical Expression in MIPS Assembly

$$c = (a < b) \;||\; \big((a + b) == 10\big)$$

Example: Computation of a Logical Expression in MIPS Assembly

Registers assignment

$$c = (a < b) \,||\, \big((a + b) == 10\big)$$

$s0

Example: Computation of a Logical Expression in MIPS Assembly

Registers assignment

$$c = (a < b) \;||\; \big((a + b) == 10\big)$$

`$s0` `$s1`

Example: Computation of a Logical Expression in MIPS Assembly

Registers assignment

$$c = (a < b) \;||\; \big((a + b) == 10\big)$$

$s2

$s0

$s1

Example: Computation of a Logical Expression in MIPS Assembly

$$c = (a < b) \,||\, \left(\overbrace{\color{red}(a + b)}^{\color{red}\$t0} == 10\right)$$

\$s2 \$s0 \$s1

```
add $t0, $s0, $s1       # syscall code to print string
```

Example: Computation of a Logical Expression in MIPS Assembly

$$\underset{\$s2}{c} = (\underset{\$s0}{a} < \underset{\$s1}{b}) \,||\, \overbrace{(\overbrace{(\underset{\$s0}{a} + \underset{\$s1}{b})}^{\$t0} {\color{red}{==}} 10)}^{{\color{red}\$t1}}$$

```
add $t0, $s0, $s1        # syscall code to print string
```

Example: Computation of a Logical Expression in MIPS Assembly

$$c = (a < b) \;||\; \left(\overbrace{\overbrace{(a+b)}^{\$t0} == 10}^{\$t1}\right)$$

$s2 $s0 $s1 $t3

```
add $t0, $s0, $s1        # syscall code to print string
li $t3, 10               # load const. 10 into $t3
```

Example: Computation of a Logical Expression in MIPS Assembly

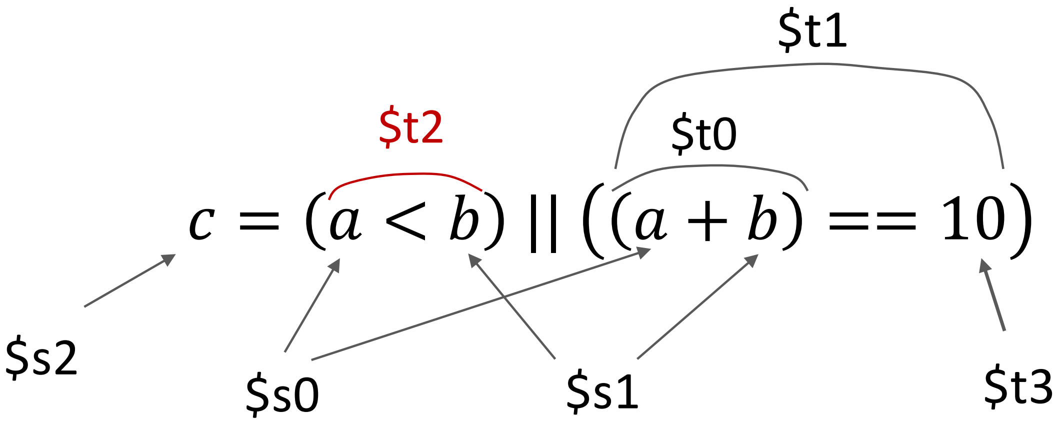
$$c = (a < b) \;||\; \overbrace{\Big(\overbrace{(a + b)}^{\$t0} == 10\Big)}^{\$t1}$$

\$s2 \$s0 \$s1 \$t3

```
add $t0, $s0, $s1      # syscall code to print string
li $t3, 10             # load const. 10 into $t3
seq $t1, $t0, $t3      # $t1 = 1, if $t0 ==$t3, and $t1=0 otherwise
```

seq = set to “1” if equal (set to “0” otherwise)

Example: Computation of a Logical Expression in MIPS Assembly



add \$t0, \$s0, \$s1	# syscall code to print string
li \$t3, 10	# load const. 10 into \$t3
seq \$t1, \$t0, \$t3	# \$t1 = 1, if \$t0 ==\$t3, and \$t1=0 otherwise
slt \$t2, \$s0, \$s1	# \$t2 = 1, if \$s0 < \$s1, and \$t2=0 otherwise

seq = set to “1” if equal (set to “0” otherwise)

slt = set to “1” if less than

Example: Computation of a Logical Expression in MIPS Assembly

$$\underset{\$s2}{c} = \overbrace{(\underset{\$s0}{a} < \underset{\$s1}{b})}^{\$t2} \,||\, \left(\overbrace{\overbrace{(\underset{\$s0}{a} + \underset{\$s1}{b})}^{\$t0} == \underset{\$t3}{10}}^{\$t1}\right)$$

```
add $t0, $s0, $s1      # syscall code to print string
li $t3, 10             # load const. 10 into $t3
seq $t1, $t0, $t3      # $t1 = 1, if $t0 ==$t3, and $t1=0 otherwise
slt $t2, $s0, $s1      # $t2 = 1, if $s0 < $s1, and $t2=0 otherwise
or $s2, $t2, $t1       # $s2 = $t2 OR $t1
```

seq = set to “1” if equal (set to “0” otherwise)

slt = set to “1” if less than

Jump Instructions for the execution flow control

Jump Instructions for the execution flow control

Infinite loop example with unconditional jump (j):

```
main:
    move $t0, $zero            # set $t0 = 0
```

Jump Instructions for the execution flow control

Infinite loop example with unconditional jump (j):

```
main:
    move $t0, $zero          # set $t0 = 0
my_loop:                     # labeled block of code
```

Jump Instructions for the execution flow control

Infinite loop example with unconditional jump (j):

```
main:
    move $t0, $zero            # set $t0 = 0
my_loop:                       # labeled block of code
    addi $t0, $t0, 1           # increment $t0 by 1
```

Jump Instructions for the execution flow control

Infinite loop example with unconditional jump (j):

```
main:
    move $t0, $zero          # set $t0 = 0
my_loop:                     # labeled block of code
    addi $t0, $t0, 1         # increment $t0 by 1
    j my_loop
```

Jump Instructions for the execution flow control

Infinite loop example with unconditional jump (j):

```
main:
    move $t0, $zero        # set $t0 = 0
my_loop:                   # labeled block of code
    addi $t0, $t0, 1       # increment $t0 by 1
    j my_loop
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

Jump Instructions for the execution flow control

Infinite loop example with unconditional jump (j):

```
1 main:
2     move $t0, $zero        # set $t0 = 0
3 my_loop:                   # labeled block of code
4     addi $t0, $t0, 1       # increment $t0 by 1
5     j my_loop
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Jump Instructions for the execution flow control

Infinite loop example with unconditional jump (j):

```
1  main:
2      move $t0, $zero          # set $t0 = 0
3  my_loop:                     # labeled block of code
4      addi $t0, $t0, 1         # increment $t0 by 1
5      j my_loop
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Jump Instructions for the execution flow control

Infinite loop example with unconditional jump (j):

```
1  main:
2      move $t0, $zero          # set $t0 = 0
3  my_loop:                     # labeled block of code
4      addi $t0, $t0, 1         # increment $t0 by 1
5      j my_loop    ...
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

This process continues infinitely

Jump Instructions

Infinite loop example with unconditional jump (j):

```
1  main:
2      move $t0, $zero          # set $t0 = 0
3  my_loop:                     # labeled block of code
4      addi $t0, $t0, 1         # increment $t0 by 1
5      j my_loop   …
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

```
1  main:
2      move $t0, $zero          # set $t0 = 0
```

Jump Instructions

Infinite loop example with unconditional jump (j):

```
1  main:
2      move $t0, $zero        # set $t0 = 0
3  my_loop:                   # labeled block of code
4      addi $t0, $t0, 1       # increment $t0 by 1
5      j my_loop   …
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

```
1  main:
2      move $t0, $zero        # set $t0 = 0
3      jal my_func            # jump to my_func block
```

Jump Instructions

Infinite loop example with unconditional jump (j):

```
1 main:
2     move $t0, $zero        # set $t0 = 0
3 my_loop:                   # labeled block of code
4     addi $t0, $t0, 1       # increment $t0 by 1
5     j my_loop   ...
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

```
main:
    move $t0, $zero        # set $t0 = 0
    jal my_func            # jump to my_func block
 ...
my_func:                   # labeled block of code
```

Jump Instructions

Infinite loop example with unconditional jump (j):

1	**main:**	
2	**move** $t0, $zero	# set $t0 = 0
3	**my_loop:**	# labeled block of code
4	**addi** $t0, $t0, 1	# increment $t0 by 1
5	**j my_loop** …	

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

1	**main:**	
2	**move** $t0, $zero	# set $t0 = 0
3	**jal my_func**	# jump to my_func block
4	…	
5	**my_func:**	# labeled block of code
6	**addi** $t0, $t0, 1	# increment $t0 by 1

Jump Instructions

Infinite loop example with unconditional jump (j):

```
1  main:
2      move $t0, $zero          # set $t0 = 0
3  my_loop:                     # labeled block of code
4      addi $t0, $t0, 1         # increment $t0 by 1
5      j my_loop      …
```

At the first iteration, the instructions are executed sequentially ("my_loop" label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

```
1  main:
2      move $t0, $zero          # set $t0 = 0
3      jal my_func              # jump to my_func block
4   …
5  my_func:                     # labeled block of code
6      addi $t0, $t0, 1         # increment $t0 by 1
7      jr $ra                   # return to line 4
```

Jump Instructions

Infinite loop example with unconditional jump (j):

```
1  main:
2      move $t0, $zero           # set $t0 = 0
3  my_loop:                      # labeled block of code
4      addi $t0, $t0, 1          # increment $t0 by 1
5      j my_loop   …
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

```
1  main:
2      move $t0, $zero           # set $t0 = 0
3      jal my_func               # jump to my_func block
4   …
5  my_func:                      # labeled block of code
6      addi $t0, $t0, 1          # increment $t0 by 1
7      jr $ra                    # return to line 4
```

When reaching “jal” instruction, before executing the jump, the address of the next instruction after the jump instruction is stored into $ra (“return address”)

Jump Instructions

Infinite loop example with unconditional jump (j):

1	**main:**
2	**move** $t0, $zero # set $t0 = 0
3	**my_loop:** # labeled block of code
4	**addi** $t0, $t0, 1 # increment $t0 by 1
5	**j my_loop** …

At the first iteration, the instructions are executed sequentially ("my_loop" label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

1	**main:**
2	**move** $t0, $zero # set $t0 = 0
3	**jal my_func** # jump to my_func block
4	…
5	**my_func:** # labeled block of code
6	**addi** $t0, $t0, 1 # increment $t0 by 1
7	**jr** $ra # return to line 4

When reaching "jal" instruction, before executing the jump, the address of the next instruction after the jump instruction is stored into $ra ("return address")

Then jump occurs to a block with a specified label ("jump delay" is imposed, exceeding 1 CPU cycle)

Jump Instructions

Infinite loop example with unconditional jump (j):

```
1 main:
2     move $t0, $zero        # set $t0 = 0
3 my_loop:                   # labeled block of code
4     addi $t0, $t0, 1       # increment $t0 by 1
5     j my_loop    …
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

```
1 main:
2     move $t0, $zero        # set $t0 = 0
3     jal my_func            # jump to my_func block
4   …
5 my_func:                   # labeled block of code
6     addi $t0, $t0, 1       # increment $t0 by 1
7     jr $ra                 # return to line 4
```

When reaching “jal” instruction, before executing the jump, the address of the next instruction after the jump instruction is stored into $ra (“return address”)

Then jump occurs to a block with a specified label (“jump delay” is imposed, exceeding 1 CPU cycle)

Jump Instructions

Infinite loop example with unconditional jump (j):

```
1  main:
2      move $t0, $zero        # set $t0 = 0
3  my_loop:                   # labeled block of code
4      addi $t0, $t0, 1       # increment $t0 by 1
5      j my_loop   …
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

```
1  main:
2      move $t0, $zero        # set $t0 = 0
3      jal my_func            # jump to my_func block
4   …
5  my_func:                   # labeled block of code
6      addi $t0, $t0, 1       # increment $t0 by 1
7      jr $ra                 # return to line 4
```

When reaching “jal” instruction, before executing the jump, the address of the next instruction after the jump instruction is stored into $ra (“return address”)

Then jump occurs to a block with a specified label (“jump delay” is imposed, exceeding 1 CPU cycle)

Jump Instructions

Infinite loop example with unconditional jump (j):

```
1  main:
2      move $t0, $zero          # set $t0 = 0
3  my_loop:                     # labeled block of code
4      addi $t0, $t0, 1         # increment $t0 by 1
5      j my_loop    …
```

At the first iteration, the instructions are executed sequentially ("my_loop" label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

```
1  main:
2      move $t0, $zero          # set $t0 = 0
3      jal my_func              # jump to my_func block
4  …
5  my_func:                     # labeled block of code
6      addi $t0, $t0, 1         # increment $t0 by 1
7      jr $ra                   # return to line 4
```

When reaching "jal" instruction, before executing the jump, the address of the next instruction after the jump instruction is stored into $ra ("return address")

Then jump occurs to a block with a specified label ("jump delay" is imposed, exceeding 1 CPU cycle)

When reaching instruction "jr $ra" (jump return), program flow goes to the instruction with address in register $ra

Jump Instructions

Infinite loop example with unconditional jump (j):

```
1  main:
2      move $t0, $zero          # set $t0 = 0
3  my_loop:                     # labeled block of code
4      addi $t0, $t0, 1         # increment $t0 by 1
5      j my_loop   …
```

At the first iteration, the instructions are executed sequentially (“my_loop” label makes no effect)

When unconditional jump instruction is reached, the program counter (PC) sets the next instruction to execute back to line 4

Instructions 4,5 are executed again

Loop with jump and link (jal) example:

```
1  main:
2      move $t0, $zero          # set $t0 = 0
3      jal my_func              # jump to my_func block
4  …
5  my_func:                     # labeled block of code
6      addi $t0, $t0, 1         # increment $t0 by 1
7      jr $ra                   # return to line 4
```

When reaching “jal” instruction, before executing the jump, the address of the next instruction after the jump instruction is stored into $ra (“return address”)

Then jump occurs to a block with a specified label (“jump delay” is imposed, exceeding 1 CPU cycle)

When reaching instruction “jr $ra” (jump return), program flow goes to the instruction with address in register $ra

Branch Instructions: Array Filling Example

```
.data:
```

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10     # allocate 10 bytes (array size)
```

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10     # allocate 10 bytes (array size)
.text                      # program instructions start next
main:                      # labeled block of code
```

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10     # allocate 10 bytes (array size)
.text                      # program instructions start next
main:                      # labeled block of code
    li $t3, 10             # $t3 = 10 (size of the array)
```

$t3 – the num. of elements to be filled

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10      # allocate 10 bytes (array size)
.text                       # program instructions start next
main:                       # labeled block of code
    li $t3, 10              # $t3 = 10 (size of the array)
    la $t0, myArray         # load the address of array 1st element
```

$t3 – the num. of elements to be filled

$t0 – the address of element to be filled

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10     # allocate 10 bytes (array size)
.text                      # program instructions start next
main:                      # labeled block of code
    li $t3, 10             # $t3 = 10 (size of the array)
    la $t0, myArray        # load the address of array 1st element
    li $t1, 1              # value to be filled inside the array
```

$t3 – the num. of elements to be filled

$t0 – the address of element to be filled

$t1 – int. value to be copied into element

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10    # allocate 10 bytes (array size)
.text                     # program instructions start next
main:                     # labeled block of code
    li $t3, 10            # $t3 = 10 (size of the array)
    la $t0, myArray       # load the address of array 1st element
    li $t1, 1             # value to be filled inside the array
fill_array:               # labeled block of code
```

$t3 – the num. of elements to be filled

$t0 – the address of element to be filled

$t1 – int. value to be copied into element

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10    # allocate 10 bytes (array size)
.text                     # program instructions start next
main:                     # labeled block of code
    li $t3, 10            # $t3 = 10 (size of the array)
    la $t0, myArray       # load the address of array 1st element
    li $t1, 1             # value to be filled inside the array
fill_array:               # labeled block of code
    sb $t1, 0($t0)        # put in memory cell with address $t0 and offset 0
                          # the value from $t1 (which is “1”)
```

$t3 – the num. of elements to be filled

$t0 – the address of element to be filled

$t1 – int. value to be copied into element

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10      # allocate 10 bytes (array size)
.text                       # program instructions start next
main:                       # labeled block of code
    li $t3, 10              # $t3 = 10 (size of the array)
    la $t0, myArray         # load the address of array 1st element
    li $t1, 1               # value to be filled inside the array
fill_array:                 # labeled block of code
    sb $t1, 0($t0)          # put in memory cell with address $t0 and offset 0
                            # the value from $t1 (which is "1")
    addi $t0, $t0, 1        # increment the address in $t0 by 1 byte
                            # (1 byte - the size of an array element)
```

$t3 – the num. of elements to be filled

$t0 – the address of element to be filled

$t1 – int. value to be copied into element

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10     # allocate 10 bytes (array size)
.text                      # program instructions start next
main:                      # labeled block of code
    li $t3, 10             # $t3 = 10 (size of the array)
    la $t0, myArray        # load the address of array 1st element
    li $t1, 1              # value to be filled inside the array
fill_array:                # labeled block of code
    sb $t1, 0($t0)         # put in memory cell with address $t0 and offset 0
                           # the value from $t1 (which is "1")
    addi $t0, $t0, 1       # increment the address in $t0 by 1 byte
                           # (1 byte - the size of an array element)
    addi $t3, $t3, -1      # decrement the number of elements to be filled
```

$t3 – the num. of elements to be filled

$t0 – the address of element to be filled

$t1 – int. value to be copied into element

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10      # allocate 10 bytes (array size)
.text                       # program instructions start next
main:                       # labeled block of code
    li $t3, 10              # $t3 = 10 (size of the array)
    la $t0, myArray         # load the address of array 1st element
    li $t1, 1               # value to be filled inside the array
fill_array:                 # labeled block of code
    sb $t1, 0($t0)          # put in memory cell with address $t0 and offset 0
                            # the value from $t1 (which is "1")
    addi $t0, $t0, 1        # increment the address in $t0 by 1 byte
                            # (1 byte - the size of an array element)
    addi $t3, $t3, -1       # decrement the number of elements to be filled
    bgtz $t3, fill_array    # branch if $t3 > 0
```

$t3 – the num. of elements to be filled

$t0 – the address of element to be filled

$t1 – int. value to be copied into element

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10     # allocate 10 bytes (array size)
.text                      # program instructions start next
main:                      # labeled block of code
    li $t3, 10             # $t3 = 10 (size of the array)
    la $t0, myArray        # load the address of array 1st element
    li $t1, 1              # value to be filled inside the array
fill_array:                # labeled block of code
    sb $t1, 0($t0)         # put in memory cell with address $t0 and offset 0
                           # the value from $t1 (which is "1")
    addi $t0, $t0, 1       # increment the address in $t0 by 1 byte
                           # (1 byte - the size of an array element)
    addi $t3, $t3, -1      # decrement the number of elements to be filled
    bgtz $t3, fill_array   # branch if $t3 > 0
```

$t3 – the num. of elements to be filled

$t0 – the address of element to be filled

$t1 – int. value to be copied into element

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10      # allocate 10 bytes (array size)
.text                       # program instructions start next
main:                       # labeled block of code
    li $t3, 10              # $t3 = 10 (size of the array)
    la $t0, myArray         # load the address of array 1st element
    li $t1, 1               # value to be filled inside the array
fill_array:                 # labeled block of code
    sb $t1, 0($t0)          # put in memory cell with address $t0 and offset 0
                            # the value from $t1 (which is "1")
    addi $t0, $t0, 1        # increment the address in $t0 by 1 byte
                            # (1 byte - the size of an array element)
    addi $t3, $t3, -1       # decrement the number of elements to be filled
    bgtz $t3, fill_array    # branch if $t3 > 0
```

$t3 – the num. of elements to be filled

$t0 – the address of element to be filled

$t1 – int. value to be copied into element

At some point, branch condition stops holding, and program terminates

Branch Instructions: Array Filling Example

```
.data:
    myArray: .space 10     # allocate 10 bytes (array size)
.text                      # program instructions start next
main:                      # labeled block of code
    li $t3, 10             # $t3 = 10 (size of the array)
    la $t0, myArray        # load the address of array 1st element
    li $t1, 1              # value to be filled inside the array
fill_array:                # labeled block of code
    sb $t1, 0($t0)         # put in memory cell with address $t0 and offset 0
                           # the value from $t1 (which is "1")
    addi $t0, $t0, 1       # increment the address in $t0 by 1 byte
                           # (1 byte - the size of an array element)
    addi $t3, $t3, -1      # decrement the number of elements to be filled
    bgtz $t3, fill_array   # branch if $t3 > 0
```

Some types of branch instructions:

bgtz – branch on greater than 0

beq – branch on equal

begz – branch on equal or greater than 0

bne – branch on not equal

Types of MIPS Processor Instructions: R, I, and J

All MIPS instructions are 32 bit long in their binary representation;
The difference is in the number of fields, their meaning, and sizes

R-type

(“register”, or “regular”)

opcode	rs	rt	rd	shamt	funct

add, sub, sll, srl

I-type

(“immediate”)

opcode	rs	rt	constant value or memory address

addi, lw, sw

J-type

(“jump”)

opcode	label address to jump to (26 bits)

j, jr, jal

Reading and Printing a String: Correction to the Example from the Last Tutorial

```
. data
    msg:        .asciiz “Enter your string: ”        # message asking to input a string
    inputStr:   .space 10                            # array of 10 bytes, to store input string
.text:
main:
    li $v0, 4                 # syscall code to print message asking to input a string
    la $a0, msg1              # load memory address of the beginning of a string
    syscall                   # syscall invocation, to print message
    li $v0, 8                 # syscall code to read string from user
    la $a0, inputStr          # memory address, where to start writing an input string
    li $a1, 10                # the maximum size of an input string
    syscall                   # syscall to read string; $v0 now “contains” a user string
    move $a0, $v0             # move string address from $v0 to $a0
    li $v0, 4                 # syscall code to print a string
    li $a1, 10                # the size of a string to be printed
    syscall                   # printing string
    li $v0, 10                # syscall code to terminate the program
    syscall                   # termination
```

MIPS Instruction Set

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1,$s2,$s3`	`$s1` = `$s2` + `$s3`	Three register operands
	subtract	`sub $s1,$s2,$s3`	`$s1` = `$s2` – `$s3`	Three register operands
	add immediate	`addi $s1,$s2,20`	`$s1` = `$s2` + 20	Used to add constants
Data transfer	load word	`lw $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Word from memory to register
	store word	`sw $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Word from register to memory
	load half	`lh $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	load half unsigned	`lhu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Halfword memory to register
	store half	`sh $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Halfword register to memory
	load byte	`lb $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	load byte unsigned	`lbu $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Byte from memory to register
	store byte	`sb $s1,20($s2)`	Memory[`$s2` + 20] = `$s1`	Byte from register to memory
	load linked word	`ll $s1,20($s2)`	`$s1` = Memory[`$s2` + 20]	Load word as 1st half of atomic swap
	store condition. word	`sc $s1,20($s2)`	Memory[`$s2`+20]=`$s1`;`$s1=0 or 1`	Store word as 2nd half of atomic swap
	load upper immed.	`lui $s1,20`	`$s1` = 20 * 2^{16}	Loads constant in upper 16 bits
Logical	and	`and $s1,$s2,$s3`	`$s1` = `$s2` & `$s3`	Three reg. operands; bit-by-bit AND
	or	`or $s1,$s2,$s3`	`$s1` = `$s2` | `$s3`	Three reg. operands; bit-by-bit OR
	nor	`nor $s1,$s2,$s3`	`$s1` = ~ (`$s2` | `$s3`)	Three reg. operands; bit-by-bit NOR
	and immediate	`andi $s1,$s2,20`	`$s1` = `$s2` & 20	Bit-by-bit AND reg with constant
	or immediate	`ori $s1,$s2,20`	`$s1` = `$s2` | 20	Bit-by-bit OR reg with constant
	shift left logical	`sll $s1,$s2,10`	`$s1` = `$s2` << 10	Shift left by constant
	shift right logical	`srl $s1,$s2,10`	`$s1` = `$s2` >> 10	Shift right by constant
Conditional branch	branch on equal	`beq $s1,$s2,25`	if (`$s1` == `$s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1,$s2,25`	if (`$s1`!= `$s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for `beq`, `bne`
	set on less than unsigned	`sltu $s1,$s2,$s3`	if (`$s2` < `$s3`) `$s1` = 1; else `$s1` = 0	Compare less than unsigned
	set less than immediate	`slti $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant
	set less than immediate unsigned	`sltiu $s1,$s2,20`	if (`$s2` < 20) `$s1` = 1; else `$s1` = 0	Compare less than constant unsigned
Unconditional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call