[BRANCH / JUMP]

MIPS Assembly Programming

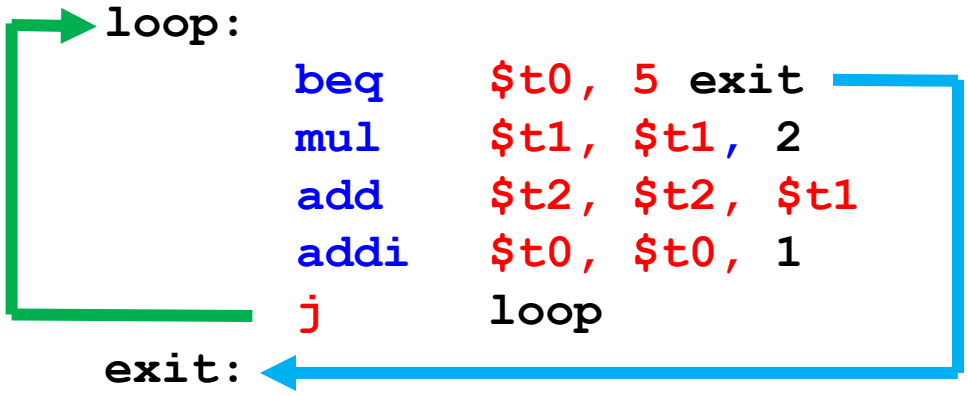
[Branch], [Function call]

MIPS Instruction Set

- Arithmetic
  - Algebraic (**`add, sub, mult, div`**) instructions
  - Logic (**`and, nor, or, xor`**) instructions
  - Shifting (**`sll, sra, srl`**) instructions
- Branch Instructions
  - Function Call instructions
  - Load and Store instructions.

Branch

- Branch instructions **Alter** (**change**) the order of program execution …
  - High Level Languages (HLL)
  - Assembly Language (AL).



In HLL (Java, C/C++)

- Constructs for altering the order of program execution within a procedure:
  - `if (...) ... else ...`
  - `while (...) ...`
  - `do ... while (...)`
  - `for (...) ...`
  - `goto ...` // C/C++ only - not Java
  - `switch (...) { case ... case ... case ... default ... }`

In Assembly Language

We only have two groups of similar instructions:

1. Unconditional Branch instruction
   - puts a new value into the program counter, causing the next instruction to be fetched from that location

2. Conditional Branch instructions
   - puts a new value into the program counter if and only if some condition is true

- Program Counter (PC) register - holds the address of the NEXT instruction to be fetched from memory and executed.
- In MIPS, a 4 is added to the pc after each fetched instruction (all MIPS instructions are one word = 4-bytes)

Branch instructions

- Unconditional branch instruction:
  - go to label (**`b`**)
- Conditional branch instructions:
  - if condition is true … then go to label (**`beq, bne`**)

Op	Operands	Description
`b`	*lab*	Unconditional branch to *lab*.
`beq`	*src1, src2, lab*	Branch to *lab* if *src1* $\equiv$ *src2*.
`bne`	*src1, src2, lab*	Branch to *lab* if *src1* $\neq$ *src2*.

Branch instructions

	Op	Operands	Description
	`b`	*lab*	Unconditional branch to *lab*.
	`beq`	*src1*, *src2*, *lab*	Branch to *lab* if *src1* $\equiv$ *src2*.
	`bne`	*src1*, *src2*, *lab*	Branch to *lab* if *src1* $\neq$ *src2*.
○	`bge(u)`	*src1*, *src2*, *lab*	Branch to *lab* if *src1* $\geq$ *src2*.
○	`bgt(u)`	*src1*, *src2*, *lab*	Branch to *lab* if *src1* $>$ *src2*.
○	`ble(u)`	*src1*, *src2*, *lab*	Branch to *lab* if *src1* $\leq$ *src2*.
○	`blt(u)`	*src1*, *src2*, *lab*	Branch to *lab* if *src1* $<$ *src2*.
○	`beqz`	*src1*, *lab*	Branch to *lab* if *src1* $\equiv 0$.
○	`bnez`	*src1*, *lab*	Branch to *lab* if *src1* $\neq 0$.
	`bgez`	*src1*, *lab*	Branch to *lab* if *src1* ≥ 0.
	`bgtz`	*src1*, *lab*	Branch to *lab* if *src1* > 0.
	`blez`	*src1*, *lab*	Branch to *lab* if *src1* ≤ 0.
	`bltz`	*src1*, *lab*	Branch to *lab* if *src1* < 0.
	`bgezal`	*src1*, *lab*	If *src1* ≥ 0, then put the address of the next instruction into `$ra` and branch to *lab*.
	`bgtzal`	*src1*, *lab*	If *src1* > 0, then put the address of the next instruction into `$ra` and branch to *lab*.
	`bltzal`	*src1*, *lab*	If *src1* < 0, then put the address of the next instruction into `$ra` and branch to *lab*.

`bge, bgt, ble, blt` are pseudo-instructions

Branch instructions…

Examples

Branch if Equal; (**beq**)

Example-1

Branch if Equal; (**beq**)

```
        .text
        .globl main
main:
        li $t0, 3
        li $t1, 3
        beq  $t0, $t1, GO
        add  $t2, $t1, $t0

GO:
        mul  $t3, $t2, $t0

        li $v0, 10
        syscall
```

```
$t2 =  ?
$t3 =  ?
```

Branch if Equal; (**beq**)

```
        .text
        .globl main
main:
        li $t0, 3
        li $t1, 3
        beq  $t0, $t1, GO
        add  $t2, $t1, $t0

GO:
        mul  $t3, $t2, $t0

        li $v0, 10
        syscall
```

```
$t2 = 0
$t3 = 0
```

Branch if Equal; (**beq**)

Example-2

Branch if Equal; (**beq**)

```
        .text
        .globl main
main:
        li $t0, 3
        li $t1, 4
        beq  $t0, $t1, GO
        add  $t2, $t1, $t0

GO:
        mul  $t3, $t2, $t0

        li $v0, 10
        syscall
```

```
$t2 =  ?
$t3 =  ?
```

Branch if Equal; (**beq**)

```
        .text
        .globl main
main:
        li $t0, 3
        li $t1, 4
        beq  $t0, $t1, GO
        add  $t2, $t1, $t0

GO:
        mul  $t3, $t2, $t0

        li $v0, 10
        syscall
```

```
$t2 =  7
$t3 = 21
```

Branch if Equal; (**beq**)

Example-3

Branch if Equal; (**beq**)

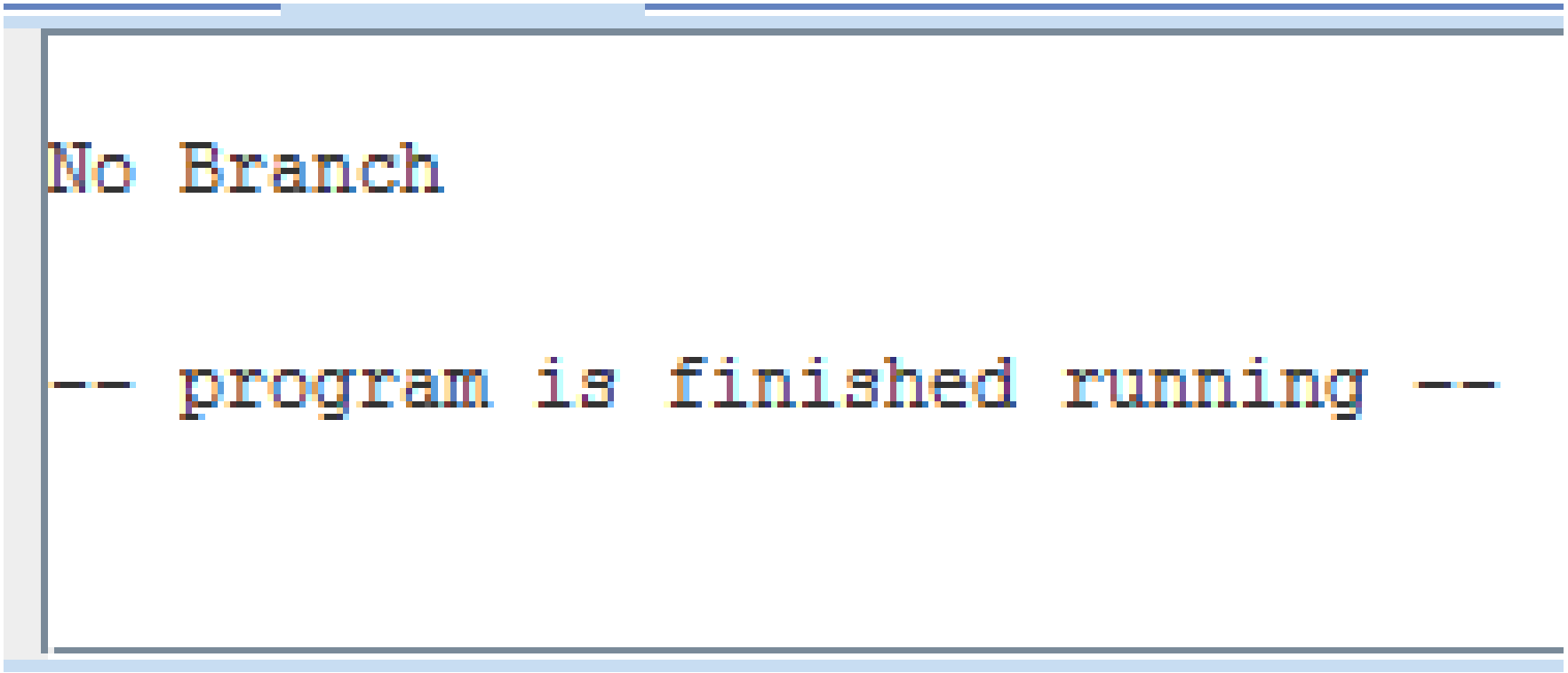
```
4               .text
5               .globl main
6   main:       li   $t0, 1
7               li   $t1, 2
8               beq  $t0, $t1, Branch
9               la   $a0,      BranchNo       # prints for no match
10              li   $v0, 4
11              syscall
12              li   $v0, 10                  # system call code for exit = 10
13              syscall                       # call operating sys
14  Branch:
15              la   $a0,      BranchYes      # prints for match
16              li   $v0, 4
17              syscall
18              li   $v0, 10                  # system call code for exit = 10
19              syscall                       # call operating sys
20
21              .data
22  BranchYes:  .asciiz  "Successful Branch  \n"
23  BranchNo:   .asciiz  "No Branch          \n"
```

What will be outputted?

Assemble … GO

```
No Branch

-- program is finished running --
```



Branch if Greater Than (`bgt`)

Example-4

Branch if Greater Than (`bgt`)

Branches to target if `$t0` > `$t1`

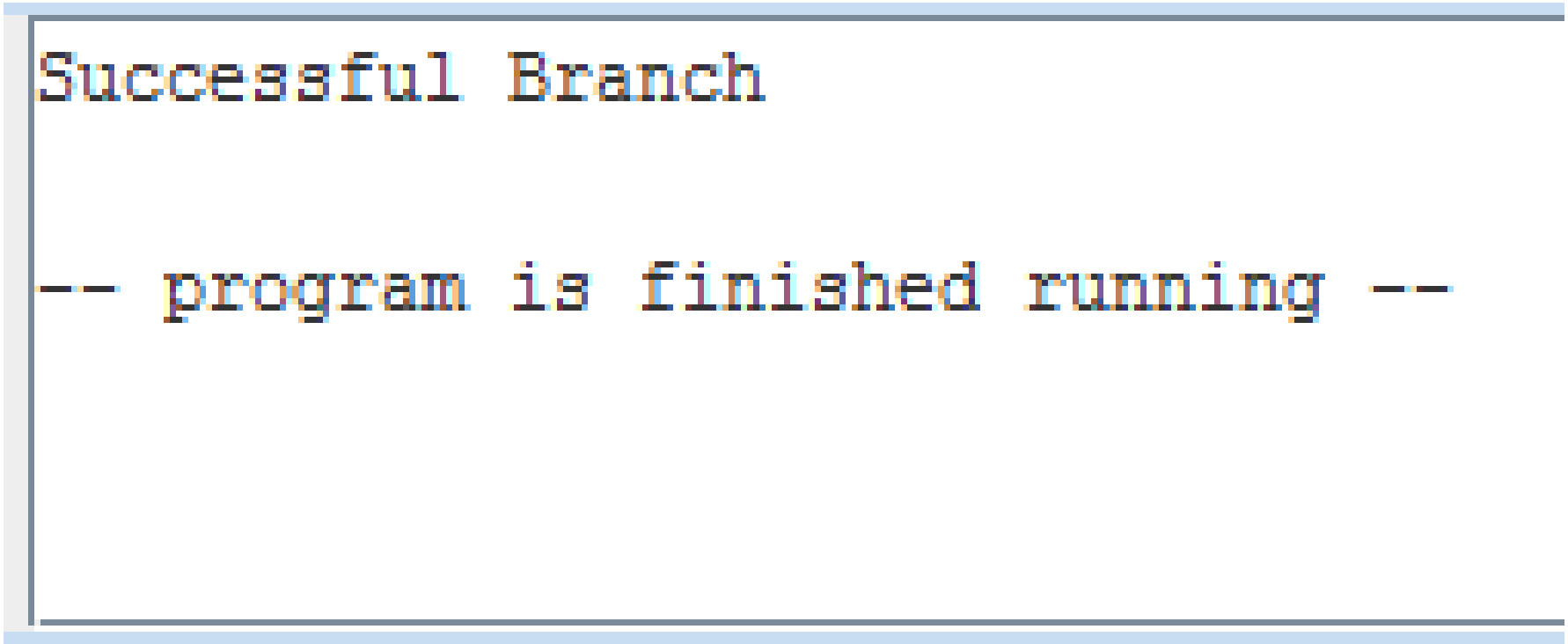
```
 4                .text
 5                .globl main
 6  main:         li   $t0, 2
 7                li   $t1, 1
 8                bgt  $t0, $t1, Branch
 9                la   $a0,      BranchNo       # prints for no match
10                li   $v0, 4
11                syscall
12                li   $v0, 10                  # system call code for exit = 10
13                syscall                       # call operating sys
14  Branch:
15                la   $a0,      BranchYes      # prints for match
16                li   $v0, 4
17                syscall
18                li   $v0, 10                  # system call code for exit = 10
19                syscall                       # call operating sys
20
21                .data
22  BranchYes:    .asciiz  "Successful Branch  \n"
23  BranchNo:     .asciiz  "No Branch          \n"
24
```

What will be outputted?

Assemble … GO

```
Successful Branch

-- program is finished running --
```



Branch if Greater Than (`bgt`)

Example-5

Branch if Greater Than (`bgt`)

Branches to target if `$t0` > `$t1`

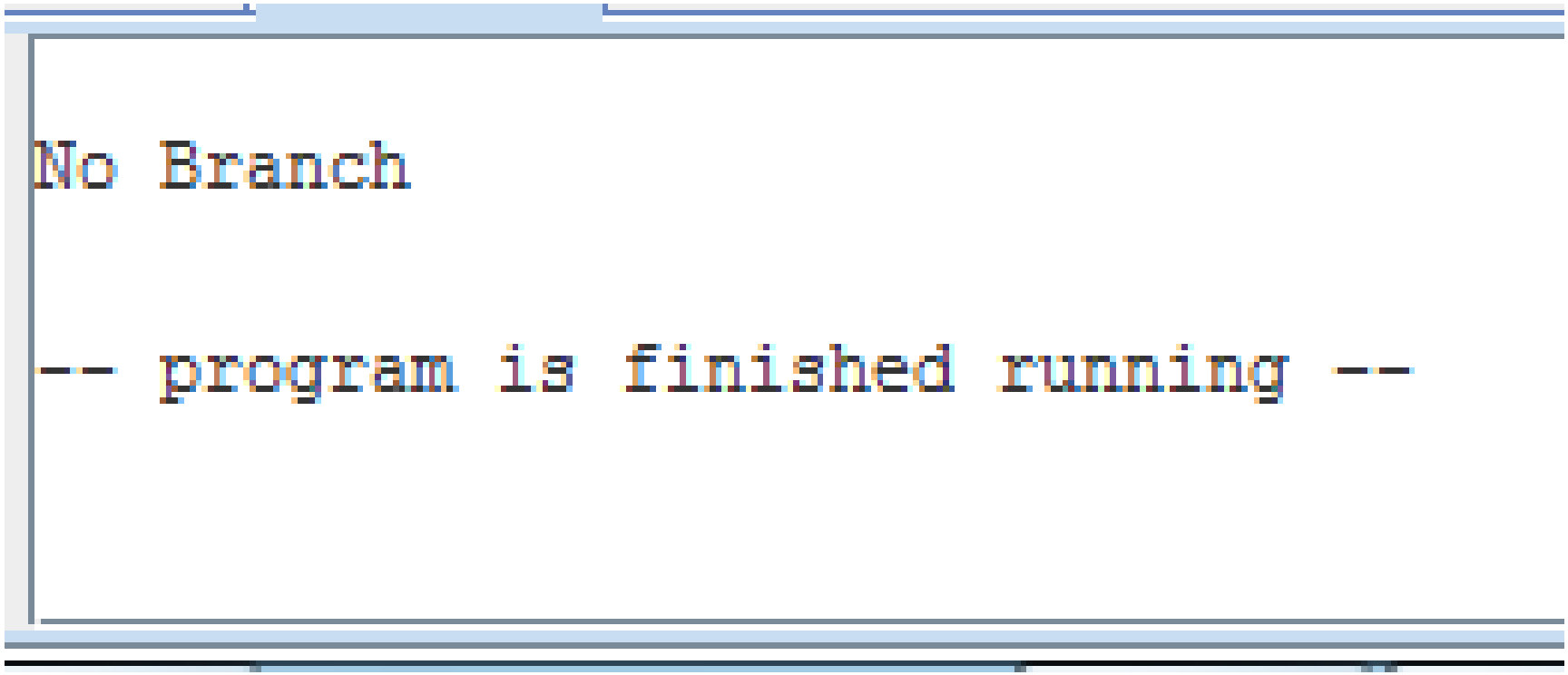
```
 4              .text
 5              .globl main
 6  main:       li   $t0, 1
 7              li   $t1, 2
 8              bgt  $t0, $t1, Branch
 9              la   $a0,      BranchNo      # prints for no match
10              li   $v0, 4
11              syscall
12              li   $v0, 10                 # system call code for exit = 10
13              syscall                      # call operating sys
14  Branch:
15              la   $a0,      BranchYes     # prints for match
16              li   $v0, 4
17              syscall
18              li   $v0, 10                 # system call code for exit = 10
19              syscall                      # call operating sys
20
21              .data
22  BranchYes:  .asciiz  "Successful Branch  \n"
23  BranchNo:   .asciiz  "No Branch          \n"
```

What will be outputted?

Assemble … GO

```
No Branch

-- program is finished running --
```



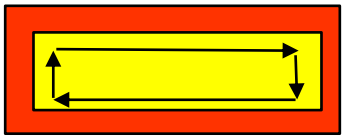
`j` = jump instruction

Unconditional Branch instruction

`j` = jump instruction

Op	Operands	Description
`j`	*label*	Jump to label *lab*.

```
loop:
```



```
j    loop
```

`beq` and **`j`** instructions

Example-J1 (1 loop)

beq and j

```
        .text
        .globl main
main:
        li      $t0, 1
        li      $t1, 1
        li      $t2, 0
loop:
        beq     $t0, 5 exit
        mul     $t1, $t1, 2
        add     $t2, $t2, $t1
        addi    $t0, $t0, 1
        j       loop
exit:
        move    $a0, $t2
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall
```

- Trace the example.
- What is the implemented expression by the loop?
- What is the result [$t2]?

beq branches to **exit** if $t0 = 5

5 minutes ...

Using `beq` and `j`

```
1
2           .text
3           .globl main
4   main:
5           li      $t0, 1          # loads 1 to t0
6           li      $t1, 1          # loads 1 to t1
7           li      $t2, 0          # loads 0 to t2
8   loop:
9           beq     $t0, 5 exit     # if the value of 5 is in t0, go to exit
10          mul     $t1, $t1, 2     # multiply t1 by 2 and store it to t1
11          add     $t2, $t2, $t1   # adds t1 and t2 and stores it to t2
12          addi    $t0, $t0, 1     # adds t0 and 1 and stores it to t0 -- uses the loop to increments from 1 to 5
13          j       loop            # jumps to loop
14  exit:
15          move    $a0, $t2        # moves the value in t2 into a0
16          li      $v0, 1          # prints the value in a0
17          syscall
18          li      $v0, 10         #exit
19          syscall
```

Trace the program

Initial values:

```
$t0=1;$t1=1;$t2=0
```

In the loop:

```
$t1 = $t1 x 2
$t2 = $t2 + $t1
$t0 = $t0 + 1
```

```
loop:
        beq     $t0, 5 exit
        mul     $t1, $t1, 2
        add     $t2, $t2, $t1
        addi    $t0, $t0, 1
        j       loop
```

step-1:

```
$t1 = 1x2 = 2
$t2 = 0+2 = 2
$t0 = 1+1 = 2
```

step-2:

```
$t1 = 2x2 = 4
$t2 = 4+2 = 6
$t0 = 2+1 = 3
```

step-3:

```
$t1 = 4x2 =  8
$t2 = 6+8 = 14
$t0 = 3+1 =  4
```

step-4:

```
$t1 = 8x2    = 16
$t2 = 16+14 = 30
$t0 = 4+1    =  5
```

Trace … another way

$t0	$t1	$t2
1	?	?
2	?	?
3	?	?
4	?	?

```
        .text
        .globl main
main:
        li      $t0, 1
        li      $t1, 1
        li      $t2, 0
loop:
        beq     $t0, 5 exit
        mul     $t1, $t1, 2
        add     $t2, $t2, $t1
        addi    $t0, $t0, 1
        j       loop
exit:
        move    $a0, $t2
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall
```

Trace …

$t0	$t1	$t2
1	?	?

```
        .text
        .globl main
main:
        li      $t0, 1
        li      $t1, 1
        li      $t2, 0
loop:
        beq     $t0, 5 exit
        mul     $t1, $t1, 2
        add     $t2, $t2, $t1
        addi    $t0, $t0, 1
        j       loop
exit:
        move    $a0, $t2
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall
```

Trace …

$t0	$t1	$t2
1	2^1	2
2	?	?
3		
4		

```
        .text
        .globl main
main:
        li      $t0, 1
        li      $t1, 1
        li      $t2, 0
loop:
        beq     $t0, 5 exit
        mul     $t1, $t1, 2
        add     $t2, $t2, $t1
        addi    $t0, $t0, 1
        j       loop
exit:
        move    $a0, $t2
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall
```

Trace …

$t0	$t1	$t2
1	2^1	2
2	2^2	6
3	?	?
4		

```
        .text
        .globl main
main:
        li      $t0, 1
        li      $t1, 1
        li      $t2, 0
loop:
        beq     $t0, 5 exit
        mul     $t1, $t1, 2
        add     $t2, $t2, $t1
        addi    $t0, $t0, 1
        j       loop
exit:
        move    $a0, $t2
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall
```

Trace …

$t0	$t1	$t2
1	2^1	2
2	2^2	6
3	2^3	14
4	?	?

```
        .text
        .globl main
main:
        li      $t0, 1
        li      $t1, 1
        li      $t2, 0
loop:
        beq     $t0, 5 exit
        mul     $t1, $t1, 2
        add     $t2, $t2, $t1
        addi    $t0, $t0, 1
        j       loop
exit:
        move    $a0, $t2
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall
```

Trace …

$t0	$t1	$t2
1	2^1	2
2	2^2	6
3	2^3	14
4	2^4	30

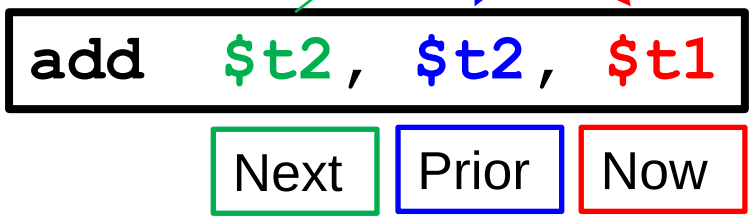
```
        .text
        .globl main
main:
        li      $t0, 1
        li      $t1, 1
        li      $t2, 0
loop:
        beq     $t0, 5 exit
        mul     $t1, $t1, 2
        add     $t2, $t2, $t1
        addi    $t0, $t0, 1
        j       loop
exit:
        move    $a0, $t2
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall
```

Loop Mathematical formula?

Loop mathematical formula …

$t0	$t1	$t2
1	2^1	2
2	2^2	6
3	2^3	14
4	2^4	30

$30 = 2^4 + 14$



Loop mathematical formula …

$t0	$t1	$t2
1	2^1	2
2	2^2	6
3	2^3	14
4	2^4	30

$30 = 2^4 + 2^3 + 6$

$30 = 2^4 + 14$

```
add   $t2, $t2, $t1
```

Loop mathematical formula …

$t0	$t1	$t2
1	2^1	2
2	2^2	6
3	2^3	14
4	2^4	30

$30 = 2^4 + 2^3 + 2^2 + 2^1$

$30 = 2^4 + 2^3 + 6$

$30 = 2^4 + 14$

```
add   $t2, $t2, $t1
```

Loop mathematical formula …

$t0	$t1	$t2
1	2^1	2
2	2^2	6
3	2^3	14
4	2^4	30

$30 = 2^4 + 2^3 + 2^2 + 2^1$

$30 = 2^4 + 2^3 + 2^2 + 2$

$30 = 2^4 + 2^3 + 6$

$30 = 2^4 + 14$

```
add   $t2, $t2, $t1
```

Loop recursion formula?

Final loop mathematical formula?

Final loop mathematical formula

$$\sum_{i=1}^{4} 2^i$$

$$30 = 2^4 + 2^3 + 2^2 + 2^1$$

Registers

Name	Number	Value
$zero	0	0
$at	1	5
$v0	2	10
$v1	3	0
$a0	4	30
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	5
$t1	9	16
$t2	10	30
$t3	11	0
$t4	12	0
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194364
hi		0
lo		16

Therefore

- To implement a single-summation, 1 loop is needed

Single-Summation (1 loop)

Example-J2

- Trace the example.
- What is the result **(Console)**?
- What is the implemented expression by the loop?

```
        .text
        .globl main
main:

        li    $t0, 1
        li    $t1, 2
        li    $t2, 0
loop:
        beq    $t0, 5, exit
        mul    $t1, $t1, 2
        add    $t2, $t2, $t1
        addi   $t0, $t0, 1
        j loop

exit:
        move    $a0, $t2
        li      $v0, 1
        syscall
        li      $v0, 10
        syscall
```

5 minutes ...

The result is: 60

$$\sum_{i=1}^{4} 2^{i+1}$$

(1 loop) Mathematical formula

Therefore

- To implement a single-summation, 1 loop is needed

Double-Summation (2 loops)

Example-J3

Double-Summation (2 loops)

$$\sum_{a=1}^{3} \sum_{b=1}^{3} (a + b + 1)$$

(2 loops) Mathematical formula

(1 + 1 + 1) + (1 + 2 + 1) + (1 + 3 + 1) +
(2 + 1 + 1) + (2 + 2 + 1) + (2 + 3 + 1) +
(3 + 1 + 1) + (3 + 2 + 1) + (3 + 3 + 1) = 45

MIPS code

```
1
2               .text
3               .globl main
4       main:
5               li $t0, 1
6               li $t1, 1
7       loopb:
8               beq $t1, 4, loopa
9               add $t2, $t1, $t0
10              addi $t2, $t2, 1
11              add $t3, $t3, $t2
12              addi $t1, $t1, 1
13              j loopb
14      loopa:
15              addi $t0, $t0, 1
16              beq $t0, 4, end
17              li $t1, 1
18              j loopb
19      end:
20              move $a0, $t3
21              li $v0, 1
22              syscall
23
24              li $v0, 10
25              syscall
```

The code loads registers **`$t0`** and **`$t1`** both with a value of 1. The code then enters a loop which checks if **`$t1`** is 4, adds the values of **`$t0`** and **`$t1`** and loads it into register **`$t2`**, and adds 1 to it. It then takes the sum of **`$t2`** and **`$t3`** and adds it to 1. If **`$t1`** is 4, the code jumps to the outer loop, which increases **`$t0`**, checks if **`$t0`** is equal to 4, resets **`$t1`** to 1, and then jumps back into the inner loop. If **`$t0`** is 4, the code jumps to the exit branch, which outputs the result of the formula and exits the program.

45

Therefore

- To implement a double-summation, 2 nested loops are needed
- … to implement a triple-summation, 3 nested loops are needed
- … to implement a quad-summation, 4 nested loops are needed …

`bgt, srl` and `j`

Example-J4

bgt, srl, j

- Trace the example (initially `$t0 = 0`).
- What is the output ?

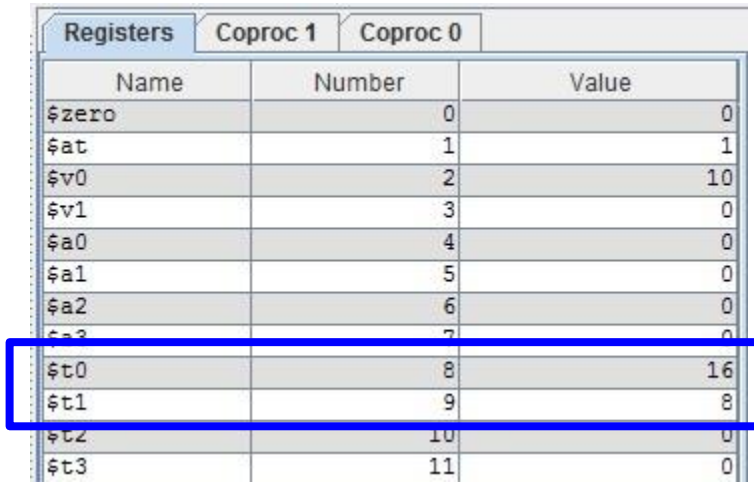
```
      .text
      .globl main
main:
      bgt  $t0, 15, go
      addi $t0, $t0, 1
      srl  $t1, $t0, 1
      j main
go:
      li $v0,10
      syscall
```

`bgt` branches to `go` if `$t0` > 15

```
$t0 = ?
$t1 = ?
```

bgt, j

```
        .text
        .globl main
main:
        bgt  $t0, 15, go
        addi $t0, $t0, 1
        srl  $t1, $t0, 1
        j main
go:
        li $v0,10
        syscall
```



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	1
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	16
$t1	9	8
$t2	10	0
$t3	11	0

```
$t0 = 16
$t1 =  8
```

`bne` and **`j`**

Example-J5

bne, j

- Trace the example.
- What is the implemented expression?

```
        .text
        .globl main
main:

        li      $t0, 3
        li      $t1, 4
        li      $t2, 5
        add     $t3, $t1, $t2
One:
        bne     $t0, $t1, Two
        add     $t4, $t2, $t0
        j Exit
Two:
        sub     $t4, $t2, $t0
Exit:
        li,     $v0, 10
        syscall
```

$t4 = ?

Implements the: `if-else` statement

```
        .text
        .globl main
main:

        li      $t0, 3
        li      $t1, 4
        li      $t2, 5
        add     $t3, $t1, $t2
One:
        bne     $t0, $t1, Two
        add     $t4, $t2, $t0
        j Exit
Two:
        sub     $t4, $t2, $t0
Exit:
        li,     $v0, 10
        syscall
```

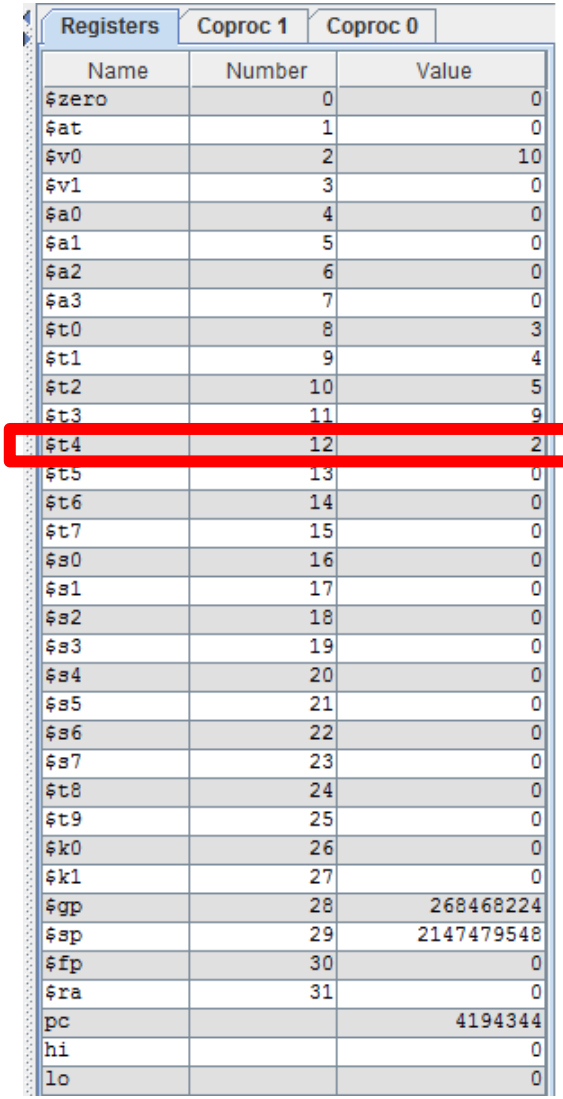
```
if: ($t0 ≠ $t1)
      $t4 = $t2-$t0;
else:
      $t4 = $t2+$t0;
```

$t4 = 2

Assemble … Go

```
        .text
        .globl main
main:

        li      $t0, 3
        li      $t1, 4
        li      $t2, 5
        add     $t3, $t1, $t2
One:
        bne     $t0, $t1, Two
        add     $t4, $t2, $t0
        j Exit
Two:
        sub     $t4, $t2, $t0
Exit:
        li,     $v0, 10
        syscall
```



Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	3
$t1	9	4
$t2	10	5
$t3	11	9
$t4	12	2
$t5	13	0
$t6	14	0
$t7	15	0
$s0	16	0
$s1	17	0
$s2	18	0
$s3	19	0
$s4	20	0
$s5	21	0
$s6	22	0
$s7	23	0
$t8	24	0
$t9	25	0
$k0	26	0
$k1	27	0
$gp	28	268468224
$sp	29	2147479548
$fp	30	0
$ra	31	0
pc		4194344
hi		0
lo		0

`bge` and **`j`**

Example-J6

bge, j

- Trace the example.
- What is the implemented function?

```
        .text
        .globl main
main:
        li   $t0, 0
        li   $t1, 1
loop:   bge  $t1, 10, exit
        add  $t0, $t0, $t1
        addi $t1, $t1, 2
        j    loop
exit:
        li $v0, 10
        syscall
```

$t0 = ?

bge, j

```
        .text
        .globl main
main:
        li   $t0, 0
        li   $t1, 1
loop:   bge  $t1, 10, exit
        add  $t0, $t0, $t1
        addi $t1, $t1, 2
        j    loop
exit:
        li $v0, 10
        syscall
```

Sum of odd numbers from 1-10

$t0 = 25

MIPS Instruction Set

- Arithmetic
  - Algebraic (**`add, sub, mult, div`**) instructions
  - Logic (**`and, nor, or, xor`**) instructions
  - Shifting (**`sll, sra, srl`**) instructions
- Branch Instructions
- Function Call Instructions
- Load and Store Instructions.

jal … **jr** instructions

Jal x = **J**ump **a**nd **l**ink to **x**

Jr x = **J**ump **r**eturn to **x**

Jump

Op	Operands	Description
`j`	*label*	Jump to label *lab*.
`jr`	*src1*	Jump to location *src1*.
`jal`	*label*	Jump to label *lab*, and store the address of the next instruction in `$ra`.
`jalr`	*src1*	Jump to location *src1*, and store the address of the next instruction in `$ra`.

Methods (Subroutines) in MIPS using: **`jal ... jr`**

Example-M1

```
        .text
        .globl main
main:
        li    $a0, 2
        li    $a1, 3
        jal   math-x              # call method
        move  $t2, $v0            # get result
        li    $v0, 10
        syscall
#.....................................
```

Main

Call method:
math-x

jal math-x = **j**ump **a**nd **l**ink to **math-x**

```
        .text
        .globl main
main:
        li    $a0, 2
        li    $a1, 3
        jal   math-x            # call method
        move  $t2, $v0          # get result
        li    $v0, 10
        syscall
#.....................................
        .text
        .globl math-x
math-x:
        mul   $t0, $a0, $a0
        mul   $t1, $a1, $a1
        add   $v0, $t0, $t1
        jr    $ra               # return to $v0
```

Main

(Method) Subroutine

`$vo-$v1` function return values

`$a0-$a3` function arguments

`jal math-x` = jump to label `math-x`, and store the address of the next instruction in the return address register: `$ra` (sets `$ra` to PC+4)

`jr $ra` = jump return to `$ra`

```
        .text
        .globl main
main:
        li    $a0, 2
        li    $a1, 3
        jal   math-x              # call method
        move  $t2, $v0            # get result
        li    $v0, 10
        syscall
#.....................................
        .text
        .globl math-x
math-x:
         mul   $t0, $a0, $a0
         mul   $t1, $a1, $a1
         add   $v0, $t0, $t1
         jr    $ra                 # return to $v0
```

Main

(Method) Subroutine

```
$t0 =   ?
$t1 =   ?
$t2 =   ?
```

```
        .text
        .globl main
main:
        li    $a0, 2
        li    $a1, 3
        jal   math-x              # call method
        move  $t2, $v0            # get result
        li    $v0, 10
        syscall
#.....................................
        .text
        .globl math-x
math-x:
         mul   $t0, $a0, $a0
         mul   $t1, $a1, $a1
         add   $v0, $t0, $t1
         jr    $ra                 # return to $v0
```

Registers

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	2
$a1	5	3
$a2	6	0
$a3	7	0
$t0	8	4
$t1	9	9
$t2	10	13

```
$t0 =  4
$t1 =  9
$t2 = 13
```

Example-M2

```
        .text
        .globl main
main:
        li      $t0, 4
        li      $t1, 9
        li      $t2, 16
        jal     method

        li      $v0, 10
        syscall
#-------------------------------
        .text
        .globl method
method:
        add     $t3, $t1, $t0
        nop
        nop
        add     $t3, $t3, $t2
        jr      $ra
```

Main

(Method) Subroutine

$t3 = ?

```
        .text
        .globl main
main:
        li      $t0, 4
        li      $t1, 9
        li      $t2, 16
        jal     method

        li      $v0, 10
        syscall
#-------------------------------
        .text
        .globl method
method:
        add     $t3, $t1, $t0
        nop
        nop
        add     $t3, $t3, $t2
        jr      $ra
```

Main

(Method) Subroutine

Registers

Name	Number	Value
$zero	0	0
$at	1	0
$v0	2	10
$v1	3	0
$a0	4	0
$a1	5	0
$a2	6	0
$a3	7	0
$t0	8	4
$t1	9	9
$t2	10	16
$t3	11	29

$t3 = 29

Notes …

- **jal** stands for **j**ump **a**nd **l**ink, it is the way of jumping to another location while retaining the memory location of where you jumped from…

- **jr $ra** stands for **j**ump **r**eturn and is used to jump back to the register **$ra** (**r**eturn **a**ddress)

- **nop** stands for **n**o-**op**eration, it is used to prevent the "processor" from beginning another operation (**used in pipelining processing**).