3.1 jal, jr: Subroutine instructions

Subroutines

A program often needs to perform the same operation for different data values. Ex: Determining the maximum of two values, converting a temperature from Fahrenheit to Celsius, etc. Instead of duplicating the instruction sequence for an operation multiple times, a programmer can use a subroutine. A ***subroutine*** is a sequence of instructions that performs a specific operation that can be called from anywhere within a program. A subroutine call causes the subroutine's statements to execute.

PARTICIPATION ACTIVITY 3.1.1: Subroutine for computing maximum of two values.

Animation captions:

1. Programs computes the maximum of DM[5000] and DM[5004], storing the result in DM[5008]. Then, the maximum of DM[5012] and DM[5016] is computed.
2. Redundant instructions for computing the maximum can be implemented as a subroutine. The CompMax subroutine is a sequence of instructions for computing the max of $t0 and $t1, and writing the max to $t2.
3. Calling the CompMax subroutine (instructions not shown) causes the subroutine's statements to execute. When the subroutine finishes, execution returns to the instruction after the subroutine call.
4. The CompMax subroutine can be called multiple times within the program.

PARTICIPATION ACTIVITY 3.1.2: Subroutines.

Refer to the animation above.

1) What label indicates the first instruction of the subroutine for computing the maximum value?

- CompMax
- MaxEnd

2) How many redundant instructions in the original code were moved to the subroutine.

- 5
- 10

3) A subroutine's instructions must be duplicated each time the subroutine is called.

- True
- False

4) A subroutine may have up to 1024 instructions.

- True
- False

Jump and link and jump register instructions

The ***jump and link*** (***jal***) instruction stores the address of the next instruction in register $ra, and then jumps to the instruction at the specified location. Ex: `jal CalcCube` stores the address of the instruction after the jal instruction in $ra, and continues execution with the instruction at CalcCube; CalcCube is the label for the first instruction of the subroutine. The ***$ra*** register (or ***return address register***) stores the instruction address to which a subroutine returns after executing. The ***jump register*** (***jr***) instruction jumps to the instruction at the address held in a register. Ex: `jr $ra` jumps to the instruction at the address held in register $ra. A programmer uses jal to call a subroutine, and jr to return from a subroutine.

PARTICIPATION ACTIVITY 3.1.3: Subroutine call using jal and jr instructions.

Animation captions:

1. The CalcCube subroutine computes the cube of the value held in $t0, writing the result to $t1.The addi instruction writes 3 to $t0.
2. The jal instruction writes the address of the next instruction, or 20, to $ra, then jumps to CalcCube. The label CalcCube specifies the subroutine's first instruction.
3. CalcCube computes the cube of $t0 as: $t0 * $t0 * $t0, writing the result to $t1.
4. jr returns from the subroutine by jumping to the instruction at the address held in $ra, or 20. The result of 27 is then stored in data memory at address 5000.

PARTICIPATION ACTIVITY 3.1.4: jal and jr instructions.

1) Write a jump and link instruction to call a subroutine named CalcTip.

Check Show answer

2) If the jal instruction below is located in instruction memory at address 200, what value is written to register $ra?


```
jal DetSpeed
```

Check Show answer

3) Using the $ra register, write an instruction to return from a subroutine named CalcTip.

Check Show answer

4) Assume $ra holds 116. If the jr instruction below is located in instruction memory at address 200, what is the address of the instruction executed after `jr $ra`?

Check Show answer

Arguments and return values

An ***argument*** is a value passed to a subroutine, that influences the subroutine's operations. A ***return value*** is a value returned from a subroutine. A simple subroutine may use specific registers for the argument and return value. Ex: The CalcCube subroutine above uses $t0 for the subroutine's argument and $t1 for the return value.

The assembly program below passes arguments to the CalcCube subroutine using $t0. The CalcCube subroutine returns the result using $t1. The program first passes 3 to the subroutine by writing 3 to register $t0. After executing the subroutine, $t1 holds the value 27, which is stored in data memory at address 5000. The program then passes 17 to the subroutine by writing 17 to $t0. The result of 4913 is then stored to data memory at address 5004.

Figure 3.1.1: Passing arguments to multiple CalcCube subroutine calls.

```
# Initialize registers for DM addresses
addi $t5, $zero, 5000
addi $t6, $zero, 5004

# Compute cube of 3
addi $t0, $zero, 3  # Pass argument of 3
jal CalcCube        # Call CalcCube
sw $t1, 0($t5)      # Store result to
DM[5000]

# Compute cube of 17
addi $t0, $zero, 17  # Pass argument of 17
jal CalcCube         # Call CalcCube
sw $t1, 0($t6)       # Store result to
DM[5004]
j Done

# CalcCube subroutine.
#    $t0 is subroutine argument
#    $t1 is subroutine return value
CalcCube:
   mul $t1, $t0, $t0
   mul $t1, $t1, $t0
   jr $ra            # Return from
subroutine

Done:
```


PARTICIPATION ACTIVITY 3.1.5: Subroutine arguments and return values.

The CalcEq subroutine below evaluates the equation: x * (y - z). Values for x, y, and z are passed to the subroutine as arguments.

```
CalcEq:
   sub $t5, $t1, $t2
   mul $t3, $t5, $t0
   jr $ra
```

1) Which register is used for x?

- $t0
- $t1
- $t2

2) Which register is used for the argument y?

- $t0
- $t1
- $t2

3) Which register is used for the argument

z?

- $t2
- $t3

4) Which register is used for the return value?

- $ra
- $t3

PARTICIPATION ACTIVITY 3.1.6: Create a subroutine.

Using the CompMax subroutine, complete the assembly program to compute the maximum of the three values in DM[5000], DM[5004], and DM[5008], storing the result in DM[5020].

1. Load $t0 and $t1 with DM[5000] and DM[5004], and call the CompMax subroutine.
2. Copy the result, which is held in $t2, into $t0. Load $t1 with DM[5008]. Call the CompMax subroutine.
3. Store the result, which is held in $t2, to DM[5020]

Assembly

```
# FIXME: Compute maximum of DM[5000],
# DM[5004], and DM[5008]
j Done

CompMax:
   slt $t3, $t0, $t1
   bne $t3, $zero, MaxIsT1
   add $t2, $zero, $t0
   j MaxEnd
   MaxIsT1: add $t2, $zero, $t1
   MaxEnd: jr $ra

Done:
```

Registers

$zero	0
$t0	0
$t1	0
$t2	0
$t3	0
$t4	0
$t5	0
$t6	0

+

Data memo

5000	12
5004	45
5008	33
5020	0

+

ENTER SIMULATION STEP RUN

More options

Table 3.1.1: Instruction summary: jal, jr.

Instruction	Format	Description	Example
jal	`jal JLabel`	Jump and link: Stores the address of the next instruction in register $ra, and continues execution with the instruction at JLabel.	`jal CalcTip`
jr	`jr $a`	Jump register: Causes execution to continue with the instruction at address $a.	`jr $t3`

CHALLENGE ACTIVITY 3.1.1: Call and create subroutines.

459784.3174716.qx3zqy7

3.2 Assembly program example: Subroutines

Assembly program

Subroutines enable programmers to write modular assembly programs. A subroutine has well-defined input and output, so a programmer can focus on developing a particular subroutine (or module) independently of other subroutines. Each subroutine should have easily-recognizable behavior, and the main behavior of the program should be easily understandable via a sequence of subroutine calls.

PARTICIPATION ACTIVITY 3.2.1: Modular program development with subroutine: Calculating employee pay.

Animation captions:

1. A program to calculate an employee's pay can be decomposed into four main steps.
2. Two subroutines can be used to calculate the overtime hours and to calculate the employee's pay. The CalcOvertimeHours and CalcPay subroutines can be developed independently.
3. Each subroutine has a well defined behavior.
4. The main program behavior consists mostly of subroutine calls.

PARTICIPATION ACTIVITY 3.2.2: Modular program development.

Refer to the animation above.

1) Modular development means to divide a program into separate modules (or subroutines) that can be developed and

tested separately.

- True
- False

2) The main program behavior only consists of jal instructions to call subroutines.

- True
- False

3) The CalcPay subroutine can be written before the CalcOvertimeHours subroutine.

- True
- False

Modular subroutine development

The subroutines for calculating the number of overtime hours and calculating the employee's pay can be developed separately.

Overtime is the number of hours worked beyond 40 hours in a single week. Ex: If an employee works 55 hours, the employee worked 15 hours of overtime. If an employee works 40 hours or fewer , then the employee worked zero overtime hours. The CalcOvertimeHours subroutine below calculates the number of overtime hours an employee has worked given the number of total hours the employee worked in a week. $t0 is used for the subroutine's argument, which is the number of hours worked in a week. $t1 is used for the subroutine's return value, which is the number of overtime hours.

Figure 3.2.1: CalcOvertimeHours subroutine calculates an employee's overtime hours.

```
CalcOvertimeHours:
   addi $t2, $zero, 40
   slt $t3, $t0, $t2
   bne $t3, $zero, NoOvertime
   # Overtime worked
   # Overtime hours is hours worked -
40
   sub $t1, $t0, $t2
   j ReturnOvertime
NoOvertime:
   # No overtime, so overtime hours
is 0
   addi $t1, $zero, 0
ReturnOvertime:
   jr $ra
```


PARTICIPATION ACTIVITY | 3.2.3: CalcOvertimeHours.

1) If $t0 holds 35, what is $t1 after the CalcOvertimeHours subroutines returns?

- 0
- 35
- 40

2) If $t0 holds 42, what is $t1 after the CalcOvertimeHours subroutines returns?

- 0
- 2
- 40

3) If the employee did not work overtime, which instruction writes 0 to the register for the return value?

- addi $t2, $zero, 40
- addi $t1, $zero, 0
- sub $t1, $t0, $t2

An employee is paid an hourly wage for the first 40 hours, and two times the hourly wage for overtime hours. The CalcPay subroutine below calculates an employee's weekly pay. An employee's hourly pay rate is $10/hour. The subroutine's uses $t0 for the total hours worked, $t1 for the employee hourly wage, and $t2 for the number of overtime hours. The subroutine returns the employee's pay using $t3.

Figure 3.2.2: CalcPay subroutine calculates an employee's pay.

```
CalcPay:
   # Calculate base pay
   mul $t3, $t0, $t1
   # Calculate overtime
pay
   mul $t4, $t2, $t1
   # Calculate total pay
   add $t3, $t3, $t4
   jr $ra
```


PARTICIPATION ACTIVITY 3.2.4: CalcPay subroutine.

1) Which register is used for the hourly wage?

- $t1
- $t2

2) The total pay is calculated as the total hours times the hourly wage plus the overtime hours times the hourly wage.
   - True
   - False

3) If $t0 holds 30, $t1 holds 10, and $t2 holds 0, what is $t3 after the subroutine returns?
   - 300
   - 600

4) If $t0 holds 55, $t1 holds 10, and $t2 holds 15, what is $t3 after the subroutine returns?
   - 550
   - 700

Main program behavior is a sequence of subroutine calls.

The program below calculates the pay for a single employee, where DM[5000] is the total hours worked by the employee, and the total pay is stored to DM[5040]. The program's main behavior consists of loading the hours worked, calling the CalcOvertimeHours subroutine to calculate the overtime hours, calling the CalcPay subroutine to calculate the pay, and storing the pay to memory.

Figure 3.2.3: Calculating pay for a single employee.

```
# Load hours worked from DM[5000]
addi $t6, $zero, 5000
lw $t0, 0($t6)
jal CalcOvertimeHours
# Overtime hours returned in $t1
# Copy $t1 to $t2
add $t2, $zero, $t1
# Initialize pay rate to $10/hour
addi $t1, $zero, 10
jal CalcPay
# Pay is returned in $t3
# Store pay to DM[5040]
addi $t6, $zero, 5040
sw $t3, 0($t6)
j Done

CalcOvertimeHours:
   addi $t2, $zero, 40
   slt $t3, $t0, $t2
   bne $t3, $zero, NoOvertime
   # Overtime worked
   # Overtime hours is 40 - hours
worked
   sub $t1, $t0, $t2
   j ReturnOvertime
NoOvertime:
   # No overtime, so overtime hours
is 0
   addi $t1, $zero, 0
ReturnOvertime:
   jr $ra

CalcPay:
   # Calculate base pay
   mul $t3, $t0, $t1
   # Calculate overtime pay
   mul $t4, $t2, $t1
   # Calculate total pay
   add $t3, $t3, $t4
   jr $ra

Done:
```


PARTICIPATION ACTIVITY 3.2.5: Using the subroutines to calculate employee pay.

Refer to the program above.

1) What does `add $t2, $zero, $t1` do?

- Initializes the total pay.
- Copies $t1 to $t2.

2) Which instruction writes the total hours worked argument for CalcPay?

- addi $t1, $zero, 10

- add $t2, $zero, $t1
- lw $t0, 0($t6)

3) What does `j Done` do?

- Calls the done subroutine.
- Jumps past the CalcOvertimeHour and CalcPay subroutines.

PARTICIPATION ACTIVITY | 3.2.6: Calculating pay for multiple employees.

Extend the program below to calculate pay for multiple employees, where each employee has a different hourly wage.

1. Modify the program to calculate pay for three employees. DM[5000], DM[5004], and DM[5008] are the total hours worked for the three employees. Store the employees' pay to DM[5040], DM[5044], and DM[5048], respectively.
2. Modify the program to load the employees' pay rates from DM[5020], DM[5024], and DM[5028], respectively.

3.3 Load and store with offsets

Load instruction with offset

An earlier section introduced the load instruction, which copies data from data memory into a register:

```
lw register 0(memory-address)
```

A memory address alone requires 32 bits, so cannot fit entirely within a 32-bit MIPS instruction. Thus, the memory address is held in a register.

Frequently, memory accesses are offsets from a base memory address, such as 5032 + 4, 5032 + 8, etc. Thus, the actual memory address is formed by adding a base memory address and an offset:

```
lw register offset(base-address)
```

Ex: If $t6 contains 5032, then `lw $t0, 4($t6)` copies the value in memory address 4 + 5032, or 5036, into $t0.

An ***offset*** is an amount added to a base address to form a final address. In MIPS, the offset is a 16-bit number so can range from -32,768 to 32,767.

PARTICIPATION ACTIVITY | 3.3.1: lw instruction with offset.

Animation captions:

1. The load instruction copies data from data memory into a register.

2. The memory address is the base memory address plus the offset. $t6 holds the base address (5032), and the offset is 4. So the memory address is 4 + 5032 = 5036.
3. The value in memory address 5036 is loaded into $t0.
4. Offsets may be useful for accessing sequential data, fields in a record, and more.

PARTICIPATION ACTIVITY 3.3.2: Load instruction.

1) Assuming $t6 holds 5032, what is the base address for:

```
lw $t5, 10($t6)
```

Check **Show answer**

2) Assuming $t6 holds 6044, what is the offset for:

```
lw $t4, 52($t6)
```

Check **Show answer**

3) Assuming $t6 holds 5072, from what memory address is the value loaded for:

```
lw $t5, 24($t6)
```

Check **Show answer**

4) Given the following register file and memory contents, what value is loaded into register $t3 by:

```
lw $t3, 20($t6)
```


Register file

$zero	0
$t0	
$t1	
$t2	
$t3	
$t4	40
$t5	5784
$t6	5780

Data memory D

	...
5796	24
5800	400
5804	30
5808	80
5812	-20
5816	17
	...

Check Show answer

5) Given the following register file and memory contents, what value is loaded into register $t3 by:

```
lw $t3, -4($t5)
```


Register file

$zero	0
$t0	
$t1	800
$t2	70
$t3	
$t4	
$t5	5020
$t6	1000

Data memory (DM)

	...
5004	20
5008	12
5012	30
5016	80
5020	
5024	70
5028	11
	...

Check Show answer

6) Assuming $t6 holds 5020, complete the load instruction to load register $t2 with data at memory location 5044 using $t6 as the base address.

```
lw $t2, ______($t6)
```

Check Show answer

7) Given the following register file, complete the load instruction to load register $t2 with data at memory location 5040, using an offset of 28.

Register file

$zero	0
$t0	
$t1	
$t2	
$t3	5000
$t4	5012
$t5	5040
$t6	5072


```
lw $t2, 28(______)
```

Check Show answer

8) Assuming $t5 holds 5000, write a load instruction that loads register $t4 with data at memory location 5048, using $t5 as the base address.

Check Show answer

9) Assuming $t5 holds 6000, write a load instruction that loads register $t3 with data at memory location 5960, using $t5 as the base address.

Check Show answer

CHALLENGE ACTIVITY | 3.3.1: Loading and storing from memory.

459784.3174716.qx3zqy7

Store with offset

An earlier section introduces a store instruction, which copies data from a register to memory. As with a load instruction, the memory address is formed by adding a base-address plus an offset.

```
sw register offset(base-address)
```

PARTICIPATION ACTIVITY | 3.3.3: Store instruction with offset.

1) Assuming $t3 holds 5132, which store instruction stores the value of register $t2 to data at memory location 5144 using $t3 as the base address?

- `sw $t2, 0($t3)`
- `sw $t2, $t3(12)`
- `sw $t2, 12($t3)`

2) Given the following register file, which instruction stores register $t3 to memory location 5084?

Register file

$zero	0

$t0	
$t1	
$t2	
$t3	17
$t4	5084
$t5	5076
$t6	5080

- ○ `sw $t3, 20($t4)`
- ○ `sw $t3, 8($t5)`
- ○ `sw $t3, 8($t6)`

3) Given the following register file, which instruction stores register $t3 to memory location 5984?

Register file

$zero	0
$t0	
$t1	
$t2	
$t3	17
$t4	5980
$t5	6000
$t6	6010

- ○ `sw $t3, -4($t4)`
- ○ `sw $t3, -16($t5)`
- ○ `sw $t3, -8($t6)`

PARTICIPATION ACTIVITY 3.3.4: Load, store, and memory.

1. Run the simulation step-by-step, observing register and memory values.
2. Load DM[5008]'s value into register $t3 using register $t4 as the base address and an offset of 8.
3. Add DM[5008]'s value to register $t1's value, which already holds the sum of DM[5000] and DM[5004].
4. Store the addition result in DM[5012].

Instruction format summary: lw and sw with offsets

In the condensed instruction format below, `C` is a literal value, like 20 or -4.

Table 3.3.1: Instruction summary: lw and sw with offset.

Instruction	Format	Description	Example
lw	`lw $a, C($b)`	Load word: Copies data from memory at address $b + C to register $a.	`lw $t3, 20($t6)`
sw	`sw $a, C($b)`	Store word: Copies data from register $a to memory at address $b + C.	`sw $t1, -4($t3)`

3.4 Subroutines and the program stack

Stack

A ***stack*** is a data structure in which items are inserted on or removed from the top of the stack. A stack ***push*** operation inserts an item on the top of the stack. A stack ***pop*** operation removes and returns the item at the top of the stack.

PARTICIPATION ACTIVITY 3.4.1: Stack push and pop operations.

Animation captions:

1. The value held in $t0 is pushed to the stack by copying the value to the top of the stack.
2. The stack grows each time an value is pushed, and the location of the stack top changes.
3. The value 20 is popped from the stack and copied to $t2. The stack shrinks each time an value is popped, and the location of the stack top changes.

PARTICIPATION ACTIVITY 3.4.2: Stack push and pop operations.

Type the stack as: 1, 2, 3

1) Given stack: 7, 5 (top is 7).

Type the stack after the following push operation:
Push 8 to stack

Check **Show answer**

2) Given stack: 34, 20 (top is 34).

Type the stack after the following

two push operations:
Push 11 to stack
Push 4 to stack

Check Show answer

3) Given stack: 5, 9, 1 (top is 5)

What is $t1 after the following pop operation?
Pop stack to $t1

Check Show answer

4) Given stack: 5, 9, 1 (top is 5)

Type the stack after the following pop operation:
Pop stack to $t1

Check Show answer

5) Given stack: 2, 9, 5, 8, 1, 3 (top is 2).

What is $t2 after the following pop operations?
Pop stack to $t1
Pop stack to $t2

Check Show answer

6) Given stack: 41, 8 (top is 41)

Type the stack after the following pop operations:
Pop stack to $t0
Push 2 to stack
Push 15 to stack
Pop stack to $t5

Check Show answer

Program stack and stack pointer

The ***program stack*** is a stack used by a program to store data for subroutines. The ***stack pointer*** (***$sp***) register is used to hold the address of the top of the program stack. In MIPSzy, the $sp register is automatically initialized to the last data memory location, which is at address 8188. The MIPszy program stack is limited in size to 1KB, or 256 words. The stack grows toward decreasing memory addresses. Pushing a value to the stack first decrements $sp by 4 and then copies the value held in a register to data memory at address $sp. Popping a value from the stack first copies the top of the stack to a register and then increments $sp by 4.

Stack overflow

*A **stack overflow** occurs when the number of values pushed to the stack exceeds the size allocated for the stack. Ex: Pushing 1005 values to the MIPSzy results in a stack overflow, as the stack size is limited to 1000 entries. A processor may have special circuitry to detect a stack overflow, allowing the system to execute special operation to handle the overflow, such as terminating the program.*

PARTICIPATION ACTIVITY 3.4.3: Instructions for stack push and pop operations.

Animation captions:

1. The stack pointer register, $sp, is initialized to 8188, which is the address of the last location in data memory. $sp holds the address of memory location at the top of the stack.
2. To push $t0 to the stack, the addi instruction first decrements $sp by 4. 8188 + -4 is 8184, so the first item will be written to memory at address 8184.
3. The sw instruction copies the value held in $t0 to the top of the stack, which is at the memory address held by $sp, or 8184.
4. To pop a value from the stack to $t2, the lw instruction first copies the memory location at the address held by $sp to $t2. Then, the addi instruction increments $sp by 4.

PARTICIPATION ACTIVITY 3.4.4: MIPS program stack.

1) Complete the assembly to push the value held in $t3 to the stack.

```
sw $t3, 0($sp)
```

Check Show answer

2) Complete the assembly to pop a value from the top of the stack to $t4.

```
addi $sp, $sp, 4
```

Check Show answer

3) Given registers and stack content below, what is DM[8184] after the following instructions?

```
addi $sp, $sp, -4
sw $t0, 0($sp)
```


Register file

$zero	0
$t0	20
$t1	15
$t2	43
$t3	71
$sp	8188

Data memory (DM)

8176	0
8180	0
8184	0
8188	0

Check Show answer

4) Given registers and stack content below, what is $sp after the following instructions?

```
addi $sp, $sp, -4
sw $t2, 0($sp)
addi $sp, $sp, -4
sw $t3, 0($sp)
```

Register file

$zero	0
$t0	17
$t1	33
$t2	41
$t3	88
$sp	8188

Data memory (DM)

8176	0
8180	0
8184	25
8188	0

Check Show answer

5) Given registers and stack content below, what is $t3 after the following instructions?

```
lw $t2, 0($sp)
addi $sp, $sp, 4
lw $t3, 0($sp)
addi $sp, $sp, 4
```

Register file

$zero	0
$t0	17
$t1	33
$t2	41
$t3	88
$sp	8176

Data memory (DM)

8176	105
8180	47
8184	25
8188	0

Check Show answer

6) Given registers and stack content below, what is DM[8176] after the following instructions?

```
lw $t2, 0($sp)
addi $sp, $sp, 4
```

Register file

$zero	0
$t0	0
$t1	0
$t2	0
$t3	0
$sp	8176

Data memory (DM)

8176	105
8180	47
8184	25
8188	0

Check Show answer

Using the program stack for subroutines

Because registers are limited, a subroutine call can use the program stack for arguments and return values rather than directly using registers. The values stored in the program stack for a subroutine is called a ***stack frame***. A subroutine call using the program stack performs the following steps.

1. Push subroutine arguments to program stack
2. Reserve space on program stack for the return value.
3. Jump to the subroutine.
4. Subroutine performs task storing return value to the reserved program stack location.
5. Subroutine returns.
6. Pop return value from program stack.
7. Pop arguments from program stack.

PARTICIPATION ACTIVITY 3.4.5: Calling subroutine using program stack: CalcOvertimeHours.

Animation captions:

1. $t0 holds the subroutine argument, and is pushed to the stack by decrementing $sp and then storing $t0 to memory at address $sp.

2. To allocate space for the return value, the stack pointer is decremented but not value needed to the stack as the subroutine will store the return value.
3. The lw instruction loads the first argument from the stack at address 4($sp), which is 8180 + 4 or 8184.
4. Subroutine calculates overtime, which is 55 - 40 = 15, storing the result in $t4.
5. The sw instruction copies the return value to stack at address 0($sp), which is 8180. Then, the subroutines returns.
6. The return value is popped from the stack to $t1. Then, the argument is popped from the stack, but does not need to be copied to a register.

In the subroutine, an offset is used in lw or sw instructions to load arguments or store the return value. Ex: For a subroutine with 1 argument and 1 return value, `0($sp)` is the address for the return value, and `4($sp)` is the address for the argument.

PARTICIPATION ACTIVITY 3.4.6: Calling subroutine using program stack.

Assume the program stack is used for all subroutine arguments and return values.

1) What is the stack frame size for a subroutine with one argument and one return value?

- 1
- 2

2) What is the stack frame size for a subroutine with one argument and no return values?

- 1
- 2

3) For a subroutine with 2 arguments and a return value, which instruction loads the first argument to $t1.

- lw $t1, 0($sp)
- lw $t1, 4($sp)
- lw $t1, 8($sp)

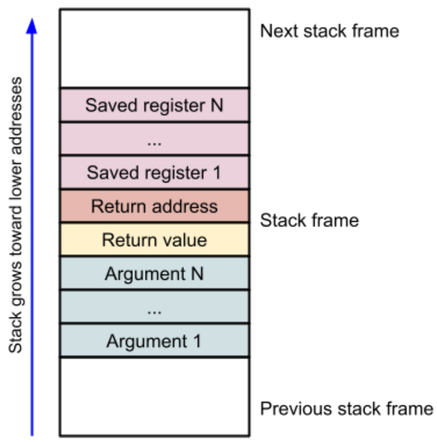
4) For a subroutine with 1 argument and a return value, which instruction stores $t4 to the stack entry allocated for the return value.

- sw $t4, 8($sp)
- sw $t4, 4($sp)
- sw $t4, 0($sp)

Saving return address and registers to the program stack

If a subroutine calls another subroutine, the value held in $ra must be saved before the second subroutine is called, because the jal for the second subroutine writes a new value to $ra. So, a subroutine that calls another subroutine will also push $ra to the program stack. Also, the value held in registers used by a subroutine may still be needed by code that called the subroutine. To avoid overwriting data, a subroutine can save the values held in any registers used by the subroutine to the program stack, and restore them before returning from the subroutine. The following shows the organization for a complete MIPSzy stack frame.

Figure 3.4.1: MIPSzy stack frame.



PARTICIPATION ACTIVITY 3.4.7: Saving and restoring register using the program stack.

Complete the CalcQuadFunc subroutine to save and restore any registers used by the subroutine using the program stack by determining the missing instructions for the highlighted lines.

```
# Computes x^2 + 2x
CalcQuadFunc:
   # Save $t0, $t1, and $t2 to stack
   addi $sp, $sp, -4
   sw $t0, 0($sp)
   addi $sp, $sp, -4
   sw $t1, 0($sp)
   (a)
   (b)
   lw $t0, 16($sp)    # Load argument from stack
   addi $t1, $zero, 2
   mul $t2, $t0, $t1  # Calculate 2*x
   mul $t1, $t0, $t0  # Calculate x*x
   add $t2, $t2, $t1  # Calculate x*x + 2x
   sw $t2, 12($sp)    # Copy return value to stack

   # Restore $t0, $t1, and $t2 from stack
   (c)
   addi $sp, $sp, 4
   lw $t1, 0($sp)
   addi $sp, $sp, 4
   (d)
   addi $sp, $sp, 4
   jr $ra
```


If unable to drag and drop, refresh the page.

addi $sp, $sp, -4   sw $t2, 0($sp)   lw $t2, 0($sp)   lw $t0, 0($sp)

(a)

(b)

(c)

(d)

Reset

PARTICIPATION ACTIVITY | 3.4.8: Saving and restoring registers using the program stack.

The CalcOvertimeHours subroutine uses registers $t1, $t2, $t3, and $t4. Modify the subroutine to save and restore those registers.

1. At the beginning of the subroutine, push the values held in $t1, $t2, $t3, and $t4 to the stack in that order.
2. Because more values have been pushed on the stack, the offsets for the location of the subroutine's argument and return value in the stack have changed. Modify the `lw $t1, 4($sp)` and `sw $t4, 0($sp)` instructions to use the correct offsets.
3. Just before the `jr $ra` instruction, restore the saved registers' values by popping 4 values from the stack into $t4, $t3, $t2, and $t1, in that order. Note the order is reversed from when the values were pushed on the stack.

CHALLENGE ACTIVITY | 3.4.1: Push and pop from stack.

459784.3174716.qx3zqy7

3.5 Machine instructions

Load, store, and add as 0s and 1s

A processor processes instructions in the form of 0's and 1's, each known as a ***machine instruction***. In MIPS, each machine instruction is 32 bits. Some bits, called the ***opcode*** ("operation code"), encode a machine instruction's operations like load word, store word, or add. Some machine instruction bits, each known as an ***operand***, indicate what register, address, or literal values are involved in an instruction .

PARTICIPATION ACTIVITY | 3.5.1: Representing MIPS assembly instructions as machine instructions.

Animation content:

undefined

Animation captions:

1. Each instruction is represented with 32 bits. For lw, the bits are divided into four fields. The first field is 6 bits for the opcode. lw's opcode is 100011.
2. 5 bits represent the destination register. $t0's encoding is 01000. Registers $t0..$t6 are encoded as 01000 to 01110.
3. The base register is represented with 5 bits as well. $t6's encoding is 01110.
4. And finally, the offset is represented with 16 bits. 0 is encoded in binary.
5. sw is similar, but with opcode 101011.
6. addi uses the same fields but for different purposes. (Here, the immediate of 15 is arbitrary).
7. For add, the 32 bits are divided into 6 fields. The first and last fields specify an add instruction.
8. The destination register uses the 4th field. The source registers use the 2nd and 3rd fields. The 5th field is unused.

PARTICIPATION ACTIVITY | 3.5.2: Machine instructions.

Refer to the above assembly and machine instructions.

1) lw's opcode is _____ .

- 000000
- 100011
- 101001

2) Registers like $t0 or $t6 are encoded using how many bits?

- 5
- 6

3) Register $t0 is encoded as _____ .

- 00000
- 01000
- 01110

4) In the lw machine instruction, the second field represents the _____ .

- destination register
- immediate
- base address

5) In the addi machine instruction, the immediate is represented by how many bits?

- 5
- 6
- 16

6) How many bits are used to denote that an instruction should perform an add?

- 6
- 12

7) In the add machine instruction, the second field is the destination register.

- True
- False

PARTICIPATION ACTIVITY | 3.5.3: Translating an lw assembly instruction to a MIPS machine instruction.

Finish translating this assembly instruction to a machine instruction:

```
lw $t6, 0($t2)
__a___  __b__  __c__  0000000000000000
```

1) a

Check Show answer

2) b

Check Show answer

3) c

Check Show answer

PARTICIPATION ACTIVITY | 3.5.4: Translating an add assembly instruction to a MIPS machine instruction.

Finish translating this assembly instruction to a machine instruction:

```
add $t6, $t5, $t4
__a___  __b__  __c__  __d__ 00000 __e___
```

1) a

Check Show answer

2) b

Check Show answer

3) c

Check Show answer

4) d

Check Show answer

5) e

Check Show answer

MIPSzy machine instructions

The table below shows MIPSzy's register encodings. The subsequent table shows all of MIPSzy's machine instructions, with opcodes in blue, registers in orange, green, and red, immediate values in purple, and unused bits in grey.

Table 3.5.1: MIPSzy register encodings.

Name	$zero	$t0	$t1	$t2	$t3	$t4	$t5	$t6
Encoding	00000	01000	01001	01010	01011	01100	01101	01110

Table 3.5.2: MIPSzy machine instructions.

Assembly	Machine
lw $t0, 0($t1)	100011 01001 01000 0000000000000000
sw $t0, 0($t1)	101011 01001 01000 0000000000000000
addi $t0, $t1, 15	001000 01001 01000 0000000000001111
add $t0, $t1, $t2	000000 01001 01010 01000 00000 100000
sub $t0, $t1, $t2	000000 01001 01010 01000 00000 100010
mult $t1, $t2	000000 01001 01010 00000 00000 011000
mflo $t0	000000 00000 00000 01000 00000 010010
beq $t1, $t2, BLabel	000100 01001 01010 0000000000000010
bne $t1, $t2, BLabel	000101 01001 01010 0000000000000010
slt $t0, $t1, $t2	000000 01001 01010 01000 00000 101010
j JLabel	000010 00000000000000000000000101
jal JLabel	000011 00000000000000000000000101
jr $t1	000000 01001 00000 00000 00000 001000

Assume BLabel becomes an immediate of 2, and JLabel 5. Creating immediates for branches/jumps is in another section.

$t0, $t1, and $t2 are used for registers. Other registers could be used.

addi's immediate value is shown as 15. That value is arbitrary.

PARTICIPATION ACTIVITY 3.5.5: MIPSzy machine instructions.

1) Different MIPSzy instructions have different numbers of bits.

- True
- False

2) addi uses _____ bits for the immediate value.

- 5
- 16

3) For a sub instruction, the first 6 bits are 000000, and the last 6 bits are _____ .

- 100000
- 100010

4) add and sub make use of all 32 bits.

- True
- False

5) For an slt instruction, the first 6 bits are 000000, and the last 6 bits are _____ .

- 101010
- 000000

6) For a j instruction, the immediate is _____ bits.

- 26
- 32

PARTICIPATION ACTIVITY 3.5.6: Translating an addi instruction to a MIPS machine instruction.

Finish translating this assembly instruction to a machine instruction:

```
addi $t3, $t4, 7
__a___  __b__  __c__  0000000000___d___
```

1) a

Check Show answer

2) b

Check Show answer

3) c

Check Show answer

4) d (just the last 6 bits of the immediate field).

Check Show answer

Executable files

Assembly instructions can be translated to machine instructions. An ***executable file*** contains the 0's and 1's of a program's machine instructions and can be loaded into an instruction memory and then executed (run). The 0's and 1's are placed into a processor's instruction memory, and the processor then executes each machine instruction.

PARTICIPATION ACTIVITY 3.5.7: Assembly instructions can be translated to machine instructions (0's and 1's) representing an executable file.

Animation captions:

1. This program doubles the value in data memory location 5000. Each instruction is translated to machine code.
2. The machine instructions are written into a file consisting of 0's and 1's, called an executable file.
3. To run a program, the machine instructions are placed into a processor's instruction memory. The CPU then executes each instruction.

PARTICIPATION ACTIVITY 3.5.8: Executable files.

1) An assembly program is placed into a processor's instruction memory.
   - True
   - False

2) An executable file is convenient for humans to read.
   - True
   - False

3) A processor executes each machine instruction one at a time (conceptually).
   - True
   - False

3.6 Jump/branch immediates

Most parts of an assembly instruction are straightforwardly translatable to a machine instruction field. An operation like addi has a specific opcode like 00100, each register, like $t1, has a specific encoding like 01001, etc. However, the immediate field for a jump or branch instruction is more involved.

Jump immediates

A jump assembly instruction (j or jal) jumps to a label representing an address in instruction memory. An address is 32 bits, but a jump machine instruction only has a 26-bit immediate field. Because an address is always a multiple of 4, the rightmost two bits are always 00, and thus are omitted, meaning those 26 bits represent 28 bits. To form a 32-bit address, the processor copies the uppermost 4 bits of the jump instruction's address. Thus, a jump instruction can only jump to addresses whose uppermost 4 bits are the same as those of the jump instruction.

Determining a label's address is discussed in another section on assemblers.

PARTICIPATION ACTIVITY 3.6.1: Determining the immediate field for a jump instruction.

Animation captions:

1. In this program, the jump is to address 48.
2. 48 is 110000 in binary. 0s are prepended to form 28 bits.
3. The rightmost two bits are always 00 (addresses are multiples of 4) and can be omitted. The j machine instruction is thus the opcode's 6 bits plus those 26 immediate bits.
4. When executing the program, given that immediate, the CPU will append 00 and prepend the j instruction address's leftmost four bits to form the actual 32-bit address.

PARTICIPATION ACTIVITY 3.6.2: Jump instruction immediate.

In the binary addresses below, the .. means enough 0's to form the indicated number of bits.

1) For instruction `j Label`, the address of Label is determined to be 40. What is the machine instruction's immediate field? (Each choice is 26 bits)

- 0..0101000
- 0..01010
- 0..0101

2) For instruction `j Label`, the address of Label is determined to be 2004. What is the machine instruction's immediate field? (Each choice is 26 bits) (Hint: 2004 is 0..011111010100 in binary).

- 0..011111010100
- 0..0111110101
- 0..01111101010000

3) Instruction `j Label` is at address 0000..1000 (32 bits). Given an immediate field of 0..0111 (26 bits), what address will the CPU construct?

- 0000..0111 (32 bits)
- 0000..011100 (26 bits)
- 0000..011100 (32 bits)

4) Instruction `j Label` is at address 0110..1000 (32 bits). Given an immediate field of 0..0111 (26 bits), what address will the CPU construct?

- 0000..011100 (26 bits)
- 0000..011100 (32 bits)
- 0110..011100 (32 bits)

Branch immediates

A branch assembly instruction (beq or bne) also branches to a label representing an address, but a branch's machine instruction only has a 16-bit immediate field. 16 bits is too few to specify an actual address. Thus, that 16-bit field instead indicates an offset from the next instruction's address. Offsets must be a multiple of 4, so the rightmost two bits are always 00, and thus omitted. Branches can thus only be to addresses reachable by an 18-bit offset from the current address, meaning a range of -2^{17} (-131,072) to $+2^{17}$-4 (131,068). That limited range is OK because branches are usually to nearby addresses.

PARTICIPATION ACTIVITY | 3.6.3: Determining the immediate field for a branch instruction.

Animation captions:

1. In this program, the branch is to the instruction at address 48.
2. Branches use offsets. The offset is computed from the next instruction's address, or 20. The offset is thus 48 - 20 = 28.
3. 28 in binary is 11100. 0's are prepending to yield 18 bits. The rightmost 2 bits are always 00 (due to word alignment) so are omitted. The remaining 16 bits form the branch's immediate field.
4. When executing the program, the CPU reconstructs the address by appending 00, then adding the field to the next instruction's 32-bit address.

Why is the offset from the next instruction's address?

The processor's hardware keeps track of the current instruction's address using a hardware component called a program counter (PC). When fetching the current machine instruction from memory, the processor immediately adds 4 to the PC, in preparation for fetching the next instruction. If the current instruction branches, the offset is added to that PC + 4 value.

An address specified as an offset to the PC is called a ***PC-relative address***

PARTICIPATION ACTIVITY | 3.6.4: Branch instruction immediate.

In binary addresses below, the .. means enough 0's to form the indicated number of bits.

1) For instruction `beq $t0, $t1, Label`, beq's next instruction's address is 40, and Label's address is 60. What is the offset in decimal, before any adjustments for filling the 16-bit immediate field?

- 20
- 40
- 60

2) For instruction `beq $t0, $t1, Label`, the beq instruction's address is 40, and Label's address is 60. What is the offset in decimal, before any adjustments for filling the 16-bit immediate field?

- 16
- 20
- 24

3) For a branch instruction, the offset is determined to be 32 (before adjustments). What will the 16-bit immediate field be?

- 0..010
- 0..01000
- 0..0100000

4) For instruction `beq $t0, $t1, Label`, the beq instruction's address is 40, and Label's address is 60. What is the machine instructions 16-bit immediate field?

- 0..010000
- 0..0100
- 0..0101

5) For instruction `beq $t0, $t1, Label`, the beq instruction's address is 40, and Label's address is 32. What is the offset in decimal, before any adjustments for filling the 16-bit field?

- 4
- -8
- -12

CHALLENGE ACTIVITY | 3.6.1: Jump/branch immediates.

459784.3174716.qx3zqy7

3.7 Assemblers

An ***assembler*** is a program that converts assembly instructions into machine instructions (0s and 1s). The assembler's three main tasks are:

1. Replacing pseudoinstructions with native instructions
2. Determining each label's memory address
3. Generating machine instructions for the assembly instructions

Replacing pseudoinstructions

A ***pseudoinstruction*** is an assembly instruction that must be replaced by one or more native instructions before being executed. The assembler does such replacement.

PARTICIPATION ACTIVITY | 3.7.1: An assembler's first task is replacing pseudoinstructions by native instructions.

Animation captions:

1. Pseudoinstructions ease the programmer's job by providing easy-to-use instructions for common operations, like blt and mul, which don't exist natively in MIPS.
2. The assembler translates each pseudoinstruction to equivalent native instructions.

PARTICIPATION ACTIVITY | 3.7.2: Replacing pseudoinstructions by native instructions.

Refer to the animation above.

1) blt is replaced by how many instructions?

- 1
- 2

2) mul is replaced by how many instructions?

- 1
- 2

3) The program with pseudoinstructions has 6 instructions. The program with native instructions has how many instructions?

- 6
- 8

Determining label addresses

Assembly instructions use labels for branches/jumps, but machine instructions use numerical offsets or addresses. Thus, a task of the assembler is to create a ***symbol table***, which lists an address for each label.

PARTICIPATION ACTIVITY | 3.7.3: An assembler's second task is to determine label addresses, kept in a symbol table.

Animation captions:

1. An assembler keeps track of each instruction's address, ignoring blank lines and comment lines.
2. When a label is defined, the assembler updates the symbol table with that label's address.
3. If a label appears on a line before an instruction (which is OK), the address is that of the instruction.

PARTICIPATION ACTIVITY | 3.7.4: Label addresses: Symbol table.

Consider the following assembly. Assume this portion of the program will be placed in instruction memory starting at address 200.

```
# Branch example
        bne $t0, $t1, Else1
        addi $t3, $t3, 50
        j Cont

Else1:  bne $t0, $t2, Else2
        addi $t3, $t3, 60
        j Cont

Else2:  addi $t3, $t3, 70

Cont:
      ...
```

1) What is the address of `bne $t0, $t1, Else1`?

Check Show answer

2) What is the address of `addi $t3, $t3, 50`?

Check Show answer

3) What is the address of label Else1?

Check Show answer

4) What is the address of label Else2?

Check Show answer

5) What is the address of label Cont?

Check Show answer

6) For the shown program, how many symbols will be in the symbol table?

Check Show answer

7) When examining the program from top to bottom, is the address of label Else1 known at the first instruction `bne $t0, $t1, Else1`? Type yes or no.

Check Show answer

8) When examining the program from top to bottom, is the address of label Cont known at the third instruction `j Cont`? Type yes or no.

Check Show answer

Generating machine instructions

After pseudoinstructions have been replaced by native instructions, and all label addresses are determined, each native instruction can be translated to a machine instruction.

PARTICIPATION ACTIVITY | 3.7.5: An assembler replaces pseudoinstructions by native ones, puts determined label addresses in a symbol table, and finally generates machine instructions.

Animation content:

undefined

Animation captions:

1. The assembler's first task is replacing pseudoinstructions with native instructions.
2. The assembler's second task is creating a symbol table, which lists each label's address.
3. Now the assembler can generate machine instructions. slt is easy.
4. For bne, the symbol table shows label Neg's address as 16 (10000 in binary). The machine instruction is generated using an offset 16 - 8 = 8. The rightmost 00 are omitted to yield 0..010 for the immediate.
5. The symbol table shows j's Cont label address as 28 (11100 in binary). That address is made into immediate 0000...111 (right 00 omitted, 0000.. prepended.
6. The remaining machine instructions are generated straightforwardly. Note that the assembly labels are no longer present in the machine instructions.

Above, the spaces in the machine instructions do not really exist, and are shown for readability only.

PARTICIPATION ACTIVITY | 3.7.6: Assembler.

Consider the animation above.

1) The assembler replaced the mul pseudoinstruction by two native instructions.

- True
- False

2) The assembler determined the instruction memory address of labels Neg and Cont.

- True
- False

3) Given a program consisting of native assembly instructions and a symbol table, the assembler could convert each native assembly instruction to a machine instruction one at a time.

- True
- False

4) A reasonable alternative approach is to generate the symbol table before replacing pseudoinstructions by native instructions, using each pseudoinstruction's address.

- True
- False

5) The above assembler made three passes over assembly: One to replace pseudoinstructions, one to create a symbol table, and one to convert each assembly instruction to a machine instruction.

- True
- False

PARTICIPATION ACTIVITY | 3.7.7: Machine instructions.

3.8 Flowcharts and assembly programming

Flowchart basics

A ***flowchart*** is a graphical depiction of a program. A flowchart typically uses an oval to indicate a program's start and end, a rectangle for processing, and a diamond for a decision.

PARTICIPATION ACTIVITY | 3.8.1: A flowchart graphically depicts a program.

Animation captions:

1. A flowchart represents a program's behavior. An oval represents the starting point. A rectangle represents processing, like getting a value (age) or setting a value (discount).
2. A diamond represents a decision. If age > 60 is true, then set discount with 15.
3. If age > 60 is false, do nothing. Both decision paths rejoin, and finally price is updated with the calculated discount. Stop is represented by an oval.
4. One can mentally "execute" the flowchart. If age > 60 is true, the original price of 100 is discounted by 15. (If false, price would have stayed at 100).

PARTICIPATION ACTIVITY | 3.8.2: Flowchart basics.

Refer to the flowchart above.

1) The top oval is the program's starting point.

- True
- False

2) The first processing box sets discount =

_____ .

- 0
- 15

3) If age is 25, the last processing box will compute price = price - _____ .

- 0
- 15

Flowcharting before programming

Some assembly-language programmers begin by creating a flowchart, and then implement the flowchart as an assembly program. Below, the assembly program uses ***assembly pseudocode***: informal code that is easy to read, and can be straightforwardly converted to a particular assembly language.

PARTICIPATION ACTIVITY | 3.8.3: Creating a flowchart first, then implementing the flowchart as an assembly program (pseudocode).

Animation captions:

1. Programmer creates flowchart to compute x to the power of y, then mentally executes flowchart to ensure correctness.
2. This flowchart describes a loop. When i reaches 0, the loop exits. The loop executes y times.
3. The programmer then implements the flowchart as assembly instructions. (Pseudocode is shown here). The first process box is implemented as instructions.
4. The decision box is implemented by a branch instruction. If i is 0 is true, branch to End. If false, execution falls through to Calc.
5. Calc implements the process box's instructions, then jumps back to decision, labeled Check.

PARTICIPATION ACTIVITY | 3.8.4: Implementing a flowchart as an assembly program.

Refer to the flowchart above.

1) The decision box is labeled: _____ .

Check **Show answer**

2) At the decision box, if i is greater than 0, the assembly program goes to label: _____ .

Check **Show answer**

3) At the decision box, if i is 0, the

assembly program goes to label:
_____ .

Check Show answer

4) For y = 3, how many times will the "jump" instruction execute?

Check Show answer

3.9 MIPSzy instruction summary

Table 3.9.1: MIPSzy Instruction summary.

Instruction	Format	Description	Example
lw	`lw $a, 0($b)`	Load word: Copies data from memory at address $b to register $a.	`lw $t3, 0($t6)`
sw	`sw $a, 0($b)`	Store word: Copies data from register $a to memory at address $b.	`sw $t1, 0($t3)`
lw (with offset)	`lw $a, C($b)`	Load word: Copies data from memory at address $b + C to register $a.	`lw $t3, 20($t6)`
sw (with offset)	`sw $a, C($b)`	Store word: Copies data from register $a to memory at address $b + C.	`sw $t1, -4($t3)`
addi	`addi $a, $b, C`	Add immediate: Adds register $b and the immediate value C, and writes the sum into register $a.	`addi $t3, $t2, 7`
add	`add $a, $b, $c`	Add: Computes the sum of registers $b and $c, and writes the sum into register $a.	`add $t4, $t1, $t2`
sub	`sub $a, $b, $c`	Subtract: Subtracts $c from $b, and writes the difference into register $a.	`sub $t3, $t2, $t5`
		Multiply:	

mul	`mul $a, $b, $c`	Multiplies register \$b and \$c, and writes the lower 32-bits of the product into register \$a. mul is a pseudoinstruction implemented using mult and mflo.	`mul $t3, $t2, $t1`
mult	`mult $a, $b`	Multiply: Multiplies register \$a and \$b, writing the 64-bit result to special register \$LO and \$HI.	`mult $t3, $t5`
mflo	`mflo $a`	Move from LO register: Copies value held in special register \$LO to register \$a.	`mflo $t2`
beq	`beq $a, $b, BLabel`	Branch on equal: Branches to the instruction at BLabel if the values held in \$a and \$b are equal. Otherwise, instruction immediately after beq is executed.	`beq $t3, $t2, SumEq5`
bne	`bne $a, $b, BLabel`	Branch on not equal: Branches to the instruction at BLabel if the values held in \$a and \$b are not equal. Otherwise, instruction immediately after bne is executed.	`bne $t4, $t5, GuessNeqCorrect`
		Set on less than: Write 1 to register \$a if value held in register \$b is less	

©zyBooks 05/17/23 13:46 1587358
Oscar Benitez
UCMERCEDCSE031ChandrasekharSpring2023

slt	`slt $a, $b, $c`	register $b is less than value held in register $c, and otherwise writes 0.	`slt $t1, $t5, $t6`
j	`j JLabel`	Jump: Causes execution to continue with the instruction at JLabel.	`j CalcTip`
jal	`jal JLabel`	Jump and link: Stores the address of the next instruction to register $ra, but continues execution with the instruction at JLabel.	`jal CalcTip`
jr	`jr $a`	Jump register: Causes execution to continue with the instruction at address $a.	`jr $t3`

Table 3.9.2: MIPSzy machine instructions.

Assembly	Machine
lw $t0, 0($t1)	100011 01001 01000 0000000000000000
sw $t0, 0($t1)	101011 01001 01000 0000000000000000
addi $t0, $t1, 15	001000 01001 01000 0000000000001111
add $t0, $t1, $t2	000000 01001 01010 01000 00000 100000
sub $t0, $t1, $t2	000000 01001 01010 01000 00000 100010
mult $t1, $t2	000100 01001 01010 00000 00000 011000
mflo $t0	000000 00000 00000 01000 00000 010010
beq $t1, $t2, BLabel	000100 01001 01010 0000000000000010
bne $t1, $t2, BLabel	000101 01001 01010 0000000000000010

bne $t1, $t2, BLabel	000101 01001 01010 0000000000000010
slt $t0, $t1, $t2	000000 01001 01010 01000 00000 101010
j JLabel	000010 00000000000000000000000101
jal JLabel	000011 00000000000000100000000101
jr $t1	000000 01001 00000 00000 00000 001000

Assume BLabel becomes an immediate of 2, and JLabel 5. Creating immediates for branches/jumps is in another section.

$t0, $t1, and $t2 are used for registers. Other registers could be used.

addi's immediate value is shown as 3. That value is arbitrary.

Exploring further:

- MIPS Processor
- Computer Organization and Design (6e) - Interactive Version (MIPS)

3.10 MIPSzy simulators

MIPSzy programming window

The MIPSzy simulator below supports the full MIPSzy instruction set. The simulator also supports displaying any MIPSzy register or valid memory address throughout execution.

PARTICIPATION ACTIVITY 3.10.1: MIPSzy simulator.

MIPSzy simulator with input and output

The MIPSzy simulator below additionally supports reading from input and writing to output.

PARTICIPATION ACTIVITY 3.10.2: MIPSzy simulator with input and output.

MIPSzy simulator with assembler

The MIPSzy simulator below includes an assembler that converts assembly instructions into machine instructions (0s and 1s).

PARTICIPATION ACTIVITY 3.10.3: MIPSzy simulator with assembler.