4.1 Assignments

Variables

C is a popular high-level programming language. Some languages like C++, Java, C#, and Javascript have roots in C. A ***compiler*** converts a high-level language like C into assembly. In this section and others, the reader is assumed to know C.

In C, a variable represents a location in memory. An assignment like `x = 7;` assigns x's memory location with the value 7.

In assembly, that variable's location can be written to a register. The value 7 can be written to another register. The assignment can then be carried out using a store word instruction with those two registers.

PARTICIPATION ACTIVITY | 4.1.1: Assigning a value to a memory location.

Animation captions:

1. For int x, one can reserve a memory location (5000) for x. One can then create an addi instruction to write x's memory address into a register ($t0).
2. Seeing x = 7, one can write 7 into a register ($t1), then write that register's value to x's memory location (whose address is in $t0).
3. During execution, $t0 is written with 5000 (x's memory address).
4. addi writes $t1 with 7, and then sw stores $t1's value into memory address 5000 (x's location), thus achieving x = 7.

PARTICIPATION ACTIVITY | 4.1.2: Assigning a value to a memory location.

Implement the C by completing the assembly.

1) C

```
int x;
x = 9;
```

Assembly

```
addi $t2, $zero, 6500  # x's
address
addi $t1, $zero, 9
sw $t1, 0( _____ )
```


Check **Show answer**

2) C

```
int y;
y = 50;
```

Assembly

```
addi _____ , $zero, 6000  # y's
address
addi $t1, $zero, 50
            $t4)
```

Check **Show answer**

3) C

```
int y;
y = 99;
```

Assembly

```
addi $t0, $zero, 5000  # y's
address
__________
sw $t2, 0($t0)
```

Check **Show answer**

4) C

```
int z;
z = 555;
```

Assembly

```
addi $t3, $zero, 5000  # z's
address
__________
__________
```

NOTE: Use $t1.

Check **Show answer**

Assignments

In C, `y = x;` assigns variable y with the value of variable x. In assembly, that assignment requires first loading a register with x's value, then storing that register's value into y. naming

PARTICIPATION ACTIVITY 4.1.3: Assigning a variable with the value of another variable.

Animation captions:

1. Variables are associated with memory locations as before. x = 7 is implemented as before.
2. For C's y = x, the assembly instructions load x's value from memory into a register, then store that register's value into y.
3. During execution, x = 7 is carried out as before.
4. To assign y = x, first x's value is loaded into a register, then that value is stored into y.

PARTICIPATION ACTIVITY 4.1.4: Assigning a variable with the value of another variable.

Given int x and int y, and the initial assembly below, what C statement does the subsequent assembly carry out?

```
addi $t0, $zero, 5000   # x's address
addi $t1, $zero, 5004   # y's address
```


1)
```
addi $t2, $zero, 99
sw $t2, 0($t0)
```

- `x = 99;`
- `y = 99;`
- `x = y;`

2)
```
lw $t3, 0($t0)
sw $t3, 0($t1)
```

- `x = 5004;`
- `x = y;`
- `y = x;`

3)
```
lw $t3, 0($t1)
sw $t3, 0($t0)
```

- `x = y;`
- `y = x;`

4)
```
sw $t0, 0($t1)
```

- `x = y;`
- `y = x;`
- `y = 5000;`

CHALLENGE ACTIVITY 4.1.1: Variable assignments.

459784.3174716.qx3zqy7

(*naming) Good programming practice uses descriptive variable names like personAge; this material uses short names like x or y to keep focus on the other concepts being taught.

4.2 Expressions

Simple arithmetic expressions

In C, a statement may assign a variable with the result of a simple arithmetic expression, such as z = x + y. In assembly, x and y are loaded into registers, an arithmetic assembly instruction computes the expression's result, and the result is then stored into z.

PARTICIPATION ACTIVITY | 4.2.1: A simple C arithmetic operation in assembly.

Animation captions:

1. Assume x, y, and z are at 5000, 5004, and 5008, and those addresses are already in registers \$t0, \$t1, and \$t2. Assume x and y already have values 20 and 50.
2. To calculate C's z = x + y, first x and y are loaded into registers (\$t3 and \$t4).
3. Those two registers are added, and the sum written to another register (\$t5).
4. Finally, the sum is stored into z.

PARTICIPATION ACTIVITY | 4.2.2: Simple arithmetic expression: Variable plus variable.

Assume variable addresses in registers are

- x: \$t0
- y: \$t1
- z: \$t2

Indicate the assembly instructions to carry out: x = y + z. Each question below represents one instruction in a sequence.

1) Get y
   - ○ lw \$t3, 0(\$t0)
   - ○ lw \$t3, 0(\$t1)
   - ○ lw \$t3, 0(\$t2)

2) Get z
   - ○ lw \$t4, 0(\$t0)
   - ○ lw \$t4, 0(\$t1)
   - ○ lw \$t4, 0(\$t2)

3) Add y + z
   - ○ add \$t3, \$t3, \$t3
   - ○ add \$t3, \$t1, \$t2
   - ○ add \$t3, \$t3, \$t4

4) Assign x with y + z
   - ○ sw \$t0, 0(\$t3)
   - ○ sw \$t3, 0(\$t0)

PARTICIPATION ACTIVITY | 4.2.3: Simple arithmetic expression: Variable plus literal.

Assume variable addresses in registers are

- x: $t0
- y: $t1

1) Which instructions implement x = y + 5?

- lw $t2, 0($t1)
  lw $t3, 0($t5)
  add $t2, $t2, $t3
  sw $t2, 0($t0)

- lw $t2, 0($t1)
  addi $t2, $t2, 5
  sw $t2, 0($t0)
- lw $t2, 0($t1)
  add $t2, $t2, 5
  sw $t2, 0($t0)

2) Which instructions implement x = x * y?

- lw $t2, 0($t0)
  lw $t3, 0($t1)
  mul $t2, $t2, $t3
  sw $t2, 0($t0)
- lw $t2, 0($t0)
  lw $t3, 0($t1)
  add $t2, $t2, $t3
  sw $t2, 0($t0)
- Not possible.

Sequences of arithmetic operations

Sometimes C statements write a variable several times. Intermediate results need not be stored into memory and may instead just be written to a register.

Figure 4.2.1: Intermediate writes to a variable need not be stored into memory.

Assume x's value is in $t3, y's value is in $t4, and z's address is in $t2.

C statements	Inefficient assembly	More efficient assembly

`z = x + y;`
`z = z + 1;`
	`add $t5, $t3, $t4  # $t5 = x + y`
	`sw $t5, 0($t2)     # Store into z`
	`lw $t5, 0($t2)     # Load z`
	`addi $t5, $t5, 1   # $t5 = z + 1`
	`sw $t5, 0($t2)     # Store into z`
		`add $t5, $t3, $t4  # $t5 = x + y`
		`addi $t5, $t5, 1   # $t5 = $t5 + 1`
		`sw $t5, 0($t2)     # Store into z`

In the assembly above, the intermediate result of x + y need not be stored into z, since that result in z would just be overwritten by the result of the next instruction (addi) that adds 1 and stores the new result into z.

PARTICIPATION ACTIVITY | 4.2.4: Intermediate results.

Given the following C that computes `z = x + x + y + 1;`

```
z = x + x;
z = z + y;
z = z + 1;
```


Which assembly instructions should be deleted from the following for efficiency?

```
1: add $t5, $t3, $t3 # $t5 = x + x
2: sw $t5, 0($t2)    # z = $t5
3: lw $t5, 0($t2)    # Load z
4: add $t5, $t5, $t4 # $t5 = z + y
5: sw $t5, 0($t2)    # z = $t5
6: lw $t5, 0($t2)    # Load z
7: addi $t5, $t5, 1  # $t5 = z + 1
8: sw $t5, 0($t2)    # z = $t5
```

1) `1: add $t5, $t3, $t3   # $t5 = x + x`

- Keep
- Delete

2) `2: sw $t5, 0($t2)   # z = $t5`

- Keep
- Delete

3) `3: lw $t5, 0($t2)    # Load z`

- Keep
- Delete

4) `4: add $t5, $t5, $t4   # $t5 = z + y`

- Keep
- Delete

5)
```
5: sw $t5, 0($t2)   # z = $t5
6: lw $t5, 0($t2)   # Load z
```

- Keep
- Delete

6) `7: addi $t5, $t5, 1   # $t5 = z + 1`

- Keep
- Delete

7) `8: sw $t5, 0($t2)   # z = $t5`

○ Keep

More complex expressions

A C statement with a simple expression having one operator, like `w = x + y`, can be converted to an assembly instruction, like `add $t5, $t3, $t4`. To convert a statement having a more complex expression, like `w = x + y + 3`, the statement may first be rewritten as several simpler statements, like `tmp1 = x + y` followed by `w = tmp1 + 3`. (tmp1 is a temporary variable used to enable such a rewrite). Each statement can then be converted to assembly.

Precedence rules should be obeyed, such as the * operator having higher precedence than +, and expressions within parentheses having higher precedence. For equal precedence operators, C specifies left-to-right evaluation. Ex: For x + y + 3, expression x + y should be computed first.

PARTICIPATION ACTIVITY | 4.2.5: A statement with a more complex expression can be rewritten as simpler statements, each then converted to assembly.

Animation captions:

1. A C statement whose expression has several operations can be rewritten as multiple statements, each with one operation, using a temporary variable (one or more).
2. Those simpler statements can be converted to assembly as before.
3. Obviously operator precedence must be respected. Multiply has higher precedence so should be computed before the add.
4. Likewise, expressions in parentheses have precedence.

PARTICIPATION ACTIVITY | 4.2.6: Rewriting a statement into statements with one-operator expressions.

Select the statements that are a correct rewrite involving one-operator expressions.

1) `w = x + y + z;`

○
```
tmp1 = x + y;
w = x + z;
```

○
```
tmp1 = x + y;
w = tmp1 + z;
```

2) `w = x + y - z;`

○
```
tmp1 = y - z;
w = x + tmp1;
```

○
```
tmp1 = x + y;
w = tmp1 - z;
```


3) `w = x + y * z;`

○
```
tmp1 = x + y;
w = tmp1 * z;
```

○
```
tmp1 = y * z;
w = x + tmp1;
```

4) `w = w + x + 1;`

```
tmp1 = w + x;
w = tmp1 + 1;
```

Not possible; w can't appear on the right.

5) `u = w + x + y + z;`

```
tmp1 = w + x;
tmp2 = tmp1 + y;
u = tmp2 + z;
```



```
tmp1 = w + x + y;
u = tmp1 + z;
```

6) `w = x * (y + z);`

```
u = x * tmp1;
tmp1 = y + z;
```

```
tmp1 = y + z;
w = x * tmp1;
```

7) `u = (w + x) * (y + z);`

```
tmp1 = w + x;
tmp2 = tmp1 * y;
u = tmp2 + z;
```

```
tmp1 = w + x;
tmp2 = y + z;
u = tmp1 * tmp2;
```

8) `u = w + (x * (y + z));`

```
tmp1 = w + x;
tmp2 = tmp1 * y;
u = tmp2 + z;
```

```
tmp1 = y + z;
tmp2 = x * tmp1;
u = w + tmp2;
```

CHALLENGE ACTIVITY 4.2.1: Arithmetic expressions.

459784.3174716.qx3zqy7

4.3 Conserving registers

Registers are limited, so computations should conserve registers, lest the computation require more registers than exist. If a value in a register is not read later, the register can be reused by writing another value.

PARTICIPATION ACTIVITY 4.3.1: Conserving registers.

Animation captions:

1. This large expression is converted to four one-operation statements using several temporary (tmp) variables.
2. A straightforward conversion to assembly uses four registers for the tmp variables and u. If only $t5 and $t6 are available, a problem arises ($t7 and $t8 don' t exist).
3. Instead, $t5 can be reused. $t5 initially holds tmp1's 90, but that value is only read in the next instruction and not later. So that next instruction can overwrite tmp1 in $t5, using $t5 for tmp2.
4. Likewise for tmp2 and tmp3: Those values aren't used later, so the values can be overwritten. Above, the old values in $t5 are shown in gray, but in reality those values are overwritten so disappear entirely.

PARTICIPATION ACTIVITY 4.3.2: Conserving registers.

Assume no temporary values (tmp1, tmp2) are used by later instructions. Rewrite the instructions to reuse $t4, to conserve registers.

1)
```
tmp1 = x + y;
w = tmp1 + z;

add $t4, $t0, $t1
add $t5, $t4, $t2
sw $t5, 0($t3)
```

```
add $t4, $t0, $t1
```
[ ]
```
sw $t4, 0($t3)
```

Check Show answer

2)
```
tmp1 = w + x;
tmp2 = tmp1 + y;
w = tmp2 + 9;

add $t4, $t0, $t1
add $t5, $t4, $t2
addi $t6, $t5, 9
sw $t6, 0($t3)
```

```
add $t4, $t0, $t1
```
[ ]
```
sw $t4, 0($t3)
```

Check Show answer

3)
```
w = (w + x) * (y + z)

add $t4, $t0, $t1 # tmp1 = w +
x
add $t5, $t2, $t3 # tmp2 = y +
z
mul $t6, $t4, $t5 # tmp3 = tmp1
* tmp2
sw $t6, ...
```

For the above code, 3 registers are

used to hold temporary values. That number can be reduced to _____ .
Type 3, 2, or 1.

Check Show answer

CHALLENGE ACTIVITY | 4.3.1: Conserving registers.

459784.3174716.qx3zqy7

4.4 If-else

If statement

In C, an ***if*** statement executes substatements when the statement's expression is true, otherwise the substatements are skipped. If the expression is a comparison for equality, an if statement can be converted to a simple pattern of assembly instructions starting with a bne (branch on not equal) instruction.

PARTICIPATION ACTIVITY | 4.4.1: If statement in assembly.

Animation captions:

1. An if statement executes the "If substatements" when the expression is true, then continues with the statement that follows next. w = 50, then w = w + 1 so 51.
2. When the expression is false, the "If substatements" are skipped, and execution proceeds directly to the After statements. Assuming w was initially 0, then w = w + 1 yields 1.
3. The assembly starts with a branch instruction, followed by the If substatements, and finally the After statements.
4. If x equals y, the bne does not branch, but rather execution falls through to the If substatements, as desired. Then, execution proceeds to the After part.
5. If x does not equal y, bne branches directly to the After part, skipping the If substatements, as desired.

PARTICIPATION ACTIVITY | 4.4.2: If statement in assembly.

Variable values: x is $t0, y is $t1, and z is $t2. w's value should be in $t3. For the given C, select the correct assembly.

1)
```
if (x == y) {
   w = 0;
}
```

```
w = w + 5;
```

```
        beq  $t0, $t1,
After
        addi $t3, $zero, 0
After:  addi $t3, $t3, 5
```

```
        bne $t0, $t1,
After
        addi $t3, $zero, 0
After:  addi $t3, $t3, 5
```


2)
```
if (x == y) {
   w = z + 1;
   w = w + y;
}

w = w + 5;
```

```
        beq $t0, $t1,
After
        addi $t3, $t2, 1
        add $t3, $t3, $t1
After:  addi $t3, $t3, 5
```

```
        bne $t0, $t1,
After
        addi $t3, $t2, 1
After:  addi $t3, $t3, 5
```

```
        bne $t0, $t1,
After
        addi $t3, $t2, 1
        add $t3, $t3, $t1
After:  addi $t3, $t3, 5
```

3)
```
if (x == 0) {
   w = w + 10;
}

w = w + 5;
```

```
        beq $t0, $zero,
After
        addi $t3, $t3, 10
After:  addi $t3, $t3, 5
```

```
        bne $t0, $zero,
After
        addi $t3, $t3, 10
After:  addi $t3, $t3, 5
```


If-else statement

In C, an ***if-else*** statement executes one of two possible sets of substatements depending on an expression's value. When the expression is a comparison for equality, an if-else statement can be converted to a simple pattern in assembly, starting with a bne instruction.

PARTICIPATION ACTIVITY 4.4.3: If-else statement in assembly.

Animation captions:

1. If x and y are equal, the if branch executes.
2. Likewise for other equal values of x and y. But if x and y aren't equal, the else part executes instead.
3. For bne, if x and y aren't equal, the branch is taken. Thus, if x and y ARE equal, execution falls through to the If substatements, which is the instruction for w = 50.
4. The else part gets a label (in this case, Else). So if x and y are NOT equal, execution branches to this Else part, skipping the If substatements. Afterwards comes the After part.
5. One last detail: When the If substatements are done, execution should jump to the After part, to skip the Else part.
6. Execution when x and y are equal.
7. Execution when x and y are NOT equal.

PARTICIPATION ACTIVITY 4.4.4: If-else statement in assembly.

The first few questions list assembly instructions intended to implement the if-else statement, in sequence. Indicate whether the instructions are correct. Assume x and y values are in $t0 and $t1, and w's value should be in $t3.

```
if (x == y) {
   w = 0;
}
else {
   w = x;
}

w = w + 5;
```

1) `(1) bne $t0, $t1, After`

- Correct
- Incorrect

2) `(2) addi $t3, $zero, 0  # w = 0;`

- Correct
- Incorrect

3) `(3) Else: add $t3, $t0, $zero  # w = x;`

- Correct
- Incorrect

4) `(5) After: addi $t3, $t3, 5`

Correct

Incorrect

5) Suppose a programmer inserted the instruction `j After` after the Else substatement of `add $t3, $t0, $zero`. Is that jump instruction correct or incorrect?

- Correct
- Incorrect

6) Using the assembly pattern introduced above, does any assembly instruction branch to an If label?

- Yes
- No

If-else-if

C programs commonly use a multi-branch form of an if-else statement. The assembly language pattern is similar to above, but requiring more labels. Each part's check has a branch (bne) to the subsequent part. Each part (except the last) ends with a jump to After.

Figure 4.4.1: If-else-if in assembly.

```
if (x == y) {
   w = w + 50;
}
else if (x == z) { //
Else1
   w = w + 60;
}
else {             //
Else2
   w = w + 70;
}
```

```
#
        bne $t0, $t1, Else1  # (x == y)
        addi $t3, $t3, 50    # w = w +
50;
        j After

Else1:  bne $t0, $t2, Else2  # (x == z)
        addi $t3, $t3, 60    # w = w +
60;
        j After

Else2:  addi $t3, $t3, 70    # w = w +
70;

After:
```

PARTICIPATION ACTIVITY | 4.4.5: If-else-if in assembly.

Find the error in the assembly, which is supposed to implement the if-else-if statement.

```
if (x == y) {
   w = w + 50;
}
else if (x == z) {
   w = w + 60;
}
else {
   w = w + 70;
}
```

1) bne $t0, $t1, After

addi $t3, $t3, 50
j After

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

Else2: addi $t3, $t3, 70

After:

PARTICIPATION ACTIVITY | 4.4.6: If-else-if in assembly.

Complete the missing assembly instructions to implement the C if-else-if code. $t0 has x's value, $t1 has y's, $t2 has z's. $t3 should get w's value.

```
if (x == z) {
   w = w + 50;
}
else if (x == y) {
   w = w + 60;
}
else {
   w = w + 70;
}
```

```
#
       (A) _____ $t0, $t2, Else1
        addi $t3, $t3, 50
        j After

(B) _____
        addi $t3, $t3, 60    # w = w + 60
        (C) ______

Else2:  addi $t3, $t3, 70    # w = w + 70

After:
```

1) (A)

$t0, $t2, Else1

Check **Show answer**

2) (B)

Check **Show answer**

3) (C)

Check Show answer

CHALLENGE ACTIVITY | 4.4.1: If-else in assembly.

459784.3174716.qx3zqy7

4.5 If-else expressions

Comparing variables for equal or not equal

An earlier section showed that x == y should use a bne instruction in assembly. Conversely, an x != y should use a beq instruction. In the assembly below, when x does not equal y, execution falls through beq to the If substatement, as desired. When x equals y, beq branches to After, skipping the If substatement.

Figure 4.5.1: != uses beq.

```
if (x != y) {
   w = w +
50;
}
```

```
#
      beq $t0, $t1, After
      addi $t3, $t3, 50 # If substatement
After:
```

PARTICIPATION ACTIVITY | 4.5.1: bne for == and beq for !=.

For the given C expression that completes the shown C, choose the correct assembly instruction to complete the shown assembly.

```
if ( _____ ) {
   w = w + 50;
}
```

```
#
      ___ ___ ___ After
      addi $t3, $t3, 50
After:
```


1) x == y

- beq $t0, $t1
- bne $t0, $t1

2) x != y

3) y == x

- beq $t0, $t1
- bne $t0, $t1
- beq $t0, $t1
- bne $t0, $t1

4) x == 0

- beq $t0, $zero
- bne $t0, $zero

5) x != 0

- beq $t0, $zero
- bne $t0, $zero

6) y != 0

- beq $zero, $t1
- bne $zero, $t1

Other comparisons

Common if expressions are not just equal or not equal, but also <, ≤, >, and ≥. MIPS pseudoinstructions exist for the latter four (discussed later in this section).

Table 4.5.1: MIPS branch instructions for various comparisons.

MIPS instruction	Example	Meaning
beq: Branch on equal	`beq $t0, $t1, L`	Branch if $t0 equals $t1
bne: Branch on not equal	`bne $t0, $t1, L`	Branch if $t0 does not equal $t1
blt: Branch on less than	`blt $t0, $t1, L`	Branch if $t0 < $t1
ble: Branch on less than or equal	`ble $t0, $t1, L`	Branch if $t0 ≤ $t1
bgt: Branch on greater than	`bgt $t0, $t1, L`	Branch if $t0 > $t1
bge: Branch on greater than or equal	`bge $t0, $t1, L`	Branch if $t0 ≥ $t1

(Above, L means Label, and, "$t0 equals $t1" actually means "$t0's value equals $t1's value")

An earlier section showed that an efficient pattern in assembly for if statements involving == or != uses the opposite comparison in assembly: == uses bne, != uses beq. Similarly, opposites should be used for the other comparisons.

Table 4.5.2: Comparison opposites.

Comparison	Opposite comparison
equal	not equal
not equal	equal
<	≥
≤	>
>	≤
≥	<

PARTICIPATION ACTIVITY 4.5.2: Various comparisons.

For each question's C expression that completes the given C, choose the correct assembly instruction to complete the assembly.

```
if ( _____ ) {
   w = w + 50;
}
```

```
#
   ____________
   addi $t3, $t3, 50
N:
```

1) x == y

- beq $t0, $t1, N
- bne $t0, $t1, N

2) x != y

- beq $t0, $t1, N
- bne $t0, $t1, N

3) x < y

- blt $t0, $t1, N
- bge $t0, $t1, N

4) x >= y

- blt $t0, $t1, N
- bge $t0, $t1, N

5) x > 0

- blt $t0, $zero, N

○ ble $t0, $zero, N

6) x <= 0

○ bgt $t0, $zero, N

○ bge $t0, $zero, N

Comparing with an expression rather than a variable

Earlier examples have compared with variables, like x == y. Sometimes an if statement in C compares with an expression, like (x - y) == z. To implement in assembly, one can first compute the expression and write the result to a register, and then compare with that register.

Figure 4.5.2: Comparing with an expression is done by first writing the expression's result to a register.

`if ((x - y) == z)`
`{`
`   w = w + 50;`
`}`
	`#`
	`      sub $t5, $t0, $t1  # $t5 = x - y`
	`      bne $t5, $t2, After # Compares (x - y)`
	`with z`
	`      addi $t3, $t3, 50  # If substatement`
	`After:`

PARTICIPATION ACTIVITY | 4.5.3: Comparing with expressions.

Implement the C by completing the assembly. Assume $t0 has x's value, $t1 has y's, $t2 has z's.

1) C

```
if ((x - y) == z) {
   w = w + 50;
}
```

Assembly

```
#
      sub $t5, $t0, $t1
      bne ___, $t2, After
      addi $t3, $t3, 50
After:
```


Check **Show answer**

2) C

```
if ((x + y) == z) {
   w = w + 50;
}
```

Assembly

```
#
      ___ ___, $t0, $t1
      bne $t5, $t2, After
      addi $t3, $t3, 50 # If
substatement
After:
```

Check Show answer

3) C

```
if ((x + y) > z) {
   w = w + 50;
}
```

Assembly

```
#
      add $t5, $t0, $t1
      ___ ___, $t2, After
      addi $t3, $t3, 50 # If
substatement
After:
```

Check Show answer

4) C

```
if ((x + y + 9) != z) {
   w = w + 50;
}
```

Assembly

```
#
      add $t5, $t0, $t1

      ___ ___ ___ ___
      beq  $t5, $t2, After
      addi $t3, $t3, 50 # If
substatement
After:
```

Check Show answer

5) C

```
if ((x + y) == (x * y) {
   w = w + 50;
}
```


Assembly

```
#
      add $t4, $t0, $t1
      mul $t5, $t0, $t1
      ___ ___, $t5, After
      addi $t3, $t3, 50 # If
substatement
After:
```

Check Show answer

6) C

```
if ((x + 3) >= (x * (y - 1)) {
   w = w + 50;
}
```

Assembly

```
#
      addi $t4, $t0, 3  # x + 3
      sub $t5, $t1, 1   # y - 1
      mul $t5, $t0, $t5 #
x*(y-1)
      ___ ___ ___ ____
      addi $t3, $t3, 50 # If
substatement
After:
```


Check Show answer

CHALLENGE ACTIVITY 4.5.1: If-else expressions in assembly.

459784.3174716.qx3zqy7

4.6 Loops

While loops

In C, a ***while loop*** has an expression and substatements. If the expression is true, the substatements execute, and then execution jumps back to check the expression again. Each execution of a loop's substatements is called an ***iteration***.

A while loop can be converted to assembly using a pattern similar to an if statement's pattern, but with a jump back after the substatements.

PARTICIPATION ACTIVITY 4.6.1: While loop in assembly.

Animation captions:

1. A while loop executes its substatements when the expression is true, then jumps back to the expression. When the expression is false, execution proceeds to the statements after.
2. Like an if statement, the assembly starts with an instruction that branches past the substatements when the expression is false, executing the substatements when true.
3. However the substatements are followed by a jump back to the loop's start, to check the expression again.
4. The assembly implements the desired looping behavior.

PARTICIPATION ACTIVITY | 4.6.2: While loop in assembly.

Implement the C by completing the assembly. Assume $t0 has x's value, $t1 has y's value, and $t2 has 2.

C	Assembly

<pre>while (x <= y) {
   x = x * 2;
}
 	
y = y + 3;</pre>
	<pre>While: (a) ___ $t0, $t1, After
	        (b) ___ $t0, $t0, $t2
	        (c) _____
	
	(d)___: addi $t1, $t1, 3</pre>

1) (a)

Check Show answer

2) (b)

Check Show answer

3) (c)

Check Show answer

4) (d)

Check Show answer

For loops

In C, a ***for loop*** has four parts: substatements, and three preceding parts of an initialization, an expression, and an update. A for loop is merely a convenient representation of a common form of while loop. Thus, to implement in assembly, one can convert the for loop to a while loop, and then implement the while loop as above.

Figure 4.6.1: For loop first converted to while loop, then to assembly.

`// for (Init; Expr; Update)`
`for (i = 0;  i < y;  i = i + 1) {`
`   w = w + 50; // Substmts`
`}`
	`i = 0;          // Init`
	`while (i < y) {  // Expr`
	`   w = w + 50;   // Substmts`
	`   i = i + 1;    // Update`
	`}`
		`addi $t0, $zero, 0  # i = 0`
		`While: bge  $t0, $t1, After # while (i < y)`
		`       addi $t3, $t3, 50   # w = w + 50`
		`       addi $t0, $t0, 1    # i = i + 1`
		`       j While`
		`After:`

In the assembly above, assume $t0 is i, $t1 is y, and $t3 is w.

PARTICIPATION ACTIVITY 4.6.3: For loop as a while loop.

Arrange the statements to implement the for loop using a while loop.

```
for (i = 50; i >= 0; i = i - 1) {
    x = x + y;
    y = y + 2;
}
```

If unable to drag and drop, refresh the page.

```
   i = i - 1;
}
```

```
x = x + y;
y = y + 2;
```

```
i = 50;
```

```
while (i >= 0) {
```

(1)

(2)

(3)

(4)

Reset

CHALLENGE ACTIVITY 4.6.1: Loops in assembly.

459784.3174716.qx3zqy7

4.7 Functions

Functions using registers only

In C, a ***function*** is a named group of statements that performs a specific operation. A ***function call*** involves passing arguments to the function's parameters, executing the function's statements, and returning the function's return value. For a function with a few arguments, like 1 or 2, registers can be used to pass arguments to the function and return a value from the function. This material assumes $t0 is used for the first argument, $t1 for the second argument, if needed, and $t2 for the return value, if needed.

A function definition can be converted to an assembly subroutine that assumes $t0 and $t1 hold the arguments, and writes the return value to $t2 before returning. A function call is converted to assembly following a simple pattern that assigns $t0 and $t1 with the arguments, jumps to the subroutine, and reads the return value from $t2.

PARTICIPATION ACTIVITY | 4.7.1: Functions in assembly.

Animation captions:

1. A function call passes arguments to the function's parameters, executes the function's statements, and returns the function's return value.
2. Register $t0 is used for the first parameter x, $t1 for the second parameter y, and $t2 for the return value.
3. The assembly starts with the subroutine label, which uses the function's name Max. The subroutine ends with the label MaxEnd and a jr instruction that jumps to the return address is held in $ra.
4. The function's statement are converted to assembly. Statements that use the function's parameters x and y will use registers $t0 and $t1.
5. Returning from the function writes the return value to $t2 and jumps to the label for the subroutine's end, MaxEnd.
6. The function call is converted to instructions that write the arguments w and 20 to $t0 and $t1 followed by a jal that jumps to the subroutine. When the subroutine returns, the return value will be in $t2.

MIPS argument and return value registers

MIPS, having more registers than MIPSzy, reserves registers $a0 to $a3 for a subroutine's arguments and $v0 and $v1 for the return value.

PARTICIPATION ACTIVITY | 4.7.2: Functions using registers.

Implement the C by completing the assembly. Assume $t0 is used for the first parameter, $t1 for the second parameter, and $t2 for the return value.

`int CalcFunc(int aVal, int bVal) {`
`   return aVal * 4 + bVal;`
`}`
	`(a) _____:`
	`   addi $t3, $zero, 4`
	`   mul $t2, (b) ___, $t3`
	`   add (c) ___, $t2, $t1`
	`   CalcFuncEnd: (d) _____`

1) (a)

Check Show answer

2) (b)

Check Show answer

3) (c)

Check Show answer

4) (d)

Check Show answer

Functions using the stack

The program stack can be used to pass arguments to and return values from a function. In converting the function definition to assembly, a lw instruction is used to load a function argument, and a sw instruction is used to store to the return value. The location of the arguments and return value depends on the number of function parameters. Ex: For a function with 2 parameters and a return value, 0($sp) is the address for the return value, 4($sp) is for the second parameter, and 8($sp) for the first parameter.

Figure 4.7.1: Function definition converted to assembly using program stack.

<pre>int Max(int x, int y)
{
  if (x > y) {
     return x;
  }
  else {
     return y;
  }
}</pre>
	<pre>Max:
	   lw $t0, 8($sp) # Load x from stack
	   lw $t1, 4($sp) # Load y from stack
	   slt $t3, $t0, $t1
	   bne $t3, $zero, MaxIsY
	   sw $t0, 0($sp) # Store return value to
	stack
	   j MaxEnd
	   MaxIsY: sw $t1, 0($sp) # Store return value to
	stack
	   MaxEnd: jr $ra</pre>

Table 4.7.1: Example stack addresses for various functions.

Function	Stack frame
`void OutLen(int feet, int inches)`	`0($sp)`: inches `4($sp)`: feet
`int CompAvg(int a, int b, int c)`	`0($sp)`: return value `4($sp)`: c `8($sp)`: b `12($sp)`: a
`int CalcSum(int w, int x, int y, int z)`	`0($sp)`: return value `4($sp)`: z `8($sp)`: y `12($sp)`: x `16($sp)`: w

PARTICIPATION ACTIVITY 4.7.3: Function using the program stack.

Implement the C by completing the assembly.

`int ConvFtToIn(int feet, int inches) {`
`    return feet * 12 + inches;`
`}`
	`ConvFtToIn:`
	`   lw $t0, (a) _____`
	`   lw  (b) _____`
	`   addi $t2, $zero, 12`
	`   mul $t3, $t0, $t2`
	`   add $t2, $t3, $t1`
	`   (c) _____`
	`   ConvFtToInEnd: jr $ra`

1) (a)

Check Show answer

2) (b)

Check Show answer

3) (c)

Check Show answer

A function call using the program stack is converted to instructions that push each argument to the stack, reserving a stack location for the return value, jumping to the subroutine, and popping the return value and arguments afterwards.

Figure 4.7.2: Function call in assembly using program stack.

Assume w is held in $t0 and 20 is held in $t1.

`z = Max(w, 20);`	`addi $sp, $sp, -4`
	`sw $t0, 0($sp)        # Push w to stack`
	`addi $sp, $sp, -4`
	`sw $t1, 0($sp)        # Push 20 to stack`
	`addi $sp, $sp, -4     # Make space for return value`
	`jal Max               # Jump to Max subroutine`
	`lw $t2, 0($sp)        # Pop return value to $t2`
	`addi $sp, $sp, 4`
	`addi $sp, $sp, 4      # Pop argument from stack`
	`addi $sp, $sp, 4      # Pop argument from stack`

PARTICIPATION ACTIVITY 4.7.4: Function call using the program stack.

Detect the error in the assembly for each function call. Assume $t0 holds xVal and $t1 holds yVal. Assume zVal is located at memory address 5004.

1) C

```
PrintVals(xVal, yVal);
```

Assembly

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

addi $sp, $sp, -4
sw $t1, 0($sp)
addi $sp, $sp, -4

jal PrintVals
addi $sp, $sp, 4
addi $sp, $sp, 4
```


2) C

```
zVal = ConvKmToMiles(yVal);
```

Assembly

```
addi $sp, $sp, -4sw $t1, 0($sp)

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

addi $sp, $sp, -4

jal ConvKmToMiles
lw $t4, 0($sp)
addi $sp, $sp, 4
addi $sp, $sp, 4
addi $t6, $zero, 5004
sw $t4, 0($t6)
```

3) C

```
zVal = CompMin(100, xVal);
```

Assembly

```
addi $t3, $zero, 100
addi $sp, $sp, -4
sw $t3, 0($sp)

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

addi $sp, $sp, -4
jal CompMin
lw $t4, 4($sp)

addi $sp, $sp, 4
addi $sp, $sp, 4
addi $sp, $sp, 4
addi $t6, $zero, 5004
sw $t4, 0($t6)
```


Functions with local variables

In C, A ***local variable*** declared in a function has a scope limited to the function, meaning the variable only exists when the function executes. A local variable can be stored in the program stack. In assembly, each variable declaration is implemented by pushing a new value to the stack. Before the function returns, the variable is popped from the stack.

PARTICIPATION ACTIVITY 4.7.5: Function with local variable.

Animation captions:

1. The variable declaration is converted to an instruction that pushes the initial value to the stack. For w, the value 0 is pushed to the stack.
2. Pushing variables to the stack changes the stack offsets for the arguments and return value.
3. The function statements are converted to assembly instructions (not shown here).
4. A return statement is converted to an instruction that copies w's value to the stack location reserved for the return value. Since, w is still on the stack, the return value's location is memory address $sp + 4.
5. w is only needed in the function, so w must be popped from the stack before the jr instruction.

PARTICIPATION ACTIVITY | 4.7.6: Function with local variables.

Consider the following C function.

```
int CalcBonus(int totSales, int salesGoal) {
   int maxBonus = 100;
   int extraSales = 0;
   int bonus = 0;

   extraSales = totSales - salesGoal;
   if (extraSales > 0) {
       bonus = extraSales * 10;
   }

   if (bonus > maxBonus) {
      bonus = maxBonus;
   }

   return bonus;
}
```

1) If all local variables are allocated to the stack, how many elements will the stack frame contain?

Check Show answer

2) Complete the assembly for the maxBonus declaration?

```
addi $t0, $zero, 

addi $sp, $sp, -4

sw $t0, 0($sp)
```


Check Show answer

3) Complete the assembly for the extraSales declaration?

```
addi $sp, $sp, -4
sw
```

Check Show answer

4) Complete the assembly for the statement, assuming $t1 holds totSales and $t2 holds salesGoal.

```
extraSales = totSales -
salesGoal;
```

```
sub $t4, $t1, $t2
sw $t4,      ($sp)
```

Check Show answer

CHALLENGE ACTIVITY | 4.7.1: Functions in assembly.

459784.3174716.qx3zqy7

4.8 Arrays and strings

Arrays

In C, an ***array*** is a variable consisting of a sequence of ***elements***. Ex: int x[4] defines 4 elements, accessed as x[0], x[1], x[2], and x[3]. An array's elements are stored sequentially in memory, with a starting address known as the ***base address*** (or just base). So if x's base is 5000, then x[0] is at 5000, x[1] 5004, x[2] 5008, and x[3] 5012 (recalling word addresses increment by 4).

In assembly, accessing element x[i] requires calculating the element's address as: base + 4*i. Ex: If x's base is 5000, then x[2]'s address is 5000 + 4*2 = 5008.

PARTICIPATION ACTIVITY | 4.8.1: Declaring an array, and calculating an element's address.

Animation captions:

1. The C statements can be implemented as assembly instructions.
2. x's base will be 5000. That base is stored in x (which happens to be at 7040).
3. To access x[2], x[2]'s address must calculated by loading the base address from x, then adding 2*4 or 8, yielding 5000 + 8 or 5008.
4. 20 can then be stored into that address for x[2].

PARTICIPATION ACTIVITY | 4.8.2: Arrays in assembly.

Consider the above animation.

1) int x[4] defines an array of how many elements?

- 3
- 4

2) x's base address is _____ .

- 5000
- 7040

3) x's base address is stored at address _____ .

- 5000
- 7040

4) Which instruction is used to get x's base address, to begin calculating an element's address?

- `lw $t1, 0($t0)`
- `sw $t1, 0($t0)`

5) The calculation for x[1] would add what to the base address 5000?

- 1
- 4
- 8

6) At what address is x[0]?

- 5000
- 5001
- 5004

7) Given another array declared as int z[300] with base address 6000, at what address is element z[100]?

- 6100
- 6400
- 7200

8) Which instructions write the address of x[1] into $t1?

- addi $t6, $zero, 7040
  lw $t0, 0($t6)
  addi $t1, $t0, 4

9) Assuming x[1]'s address is in $t1, which instruction writes $t6 with x[1]'s value?

- addi $t6, $zero, 7040
  lw $t0, 0($t6)
  addi $t1, $t0, 1
- add $t6, $t1, $zero
- lw $t6, 0($t1)

Arrays and loops

One benefit of an array versus one variable per element is efficient handling in loops, as shown below.

Figure 4.8.1: Array example in C.

Assume int x[51] and int i.

```
i = 0;
while (i <= 50) {
   x[i] = i * i;
   i = i + 1;
}
// x will be 0, 1, 4, 9, ...,
2500
```

Figure 4.8.2: Above array example in assembly.

Assume: $t0 has x's base of 5000, $t1 has 50, and $t2 has 4.

```
Line  #
1         addi $t3, $zero, 0  # i = 0;
2     While:
3         bgt $t3, $t1, After  # while (i <= 50)
4         mul $t4, $t3, $t2    # $t4 = i * 4
5         add $t4, $t0, $t4    # $t4 = x's base +
      i*4
6         mul $t5, $t3, $t3    # $t5 = i * i
7         sw $t5, 0($t4)       # x[i] = i * i;
8         addi $t3, $t3, 1     # i = i + 1;
9         j While
10
11    After:
```


PARTICIPATION ACTIVITY 4.8.3: Arrays and loops.

Consider the figure above showing assembly.

1) In the first iteration, i ($t3) is 0. What is $t4 after line 4 executes?

- 0
- 4

2) In the first iteration, what is $t4 after line 5 executes?
   - 0
   - 5000
   - 5004

3) In the second iteration, what element is being written?
   - x[0]
   - x[1]
   - x[2]

4) In the second iteration, what address is calculated in line 5?
   - 5000
   - 5004
   - 5008

5) In the last iteration, i will be 50. What address will the sw instruction store into?
   - 2500
   - 50
   - 200
   - 5200

6) Suppose the array was int x[100] rather than int x[50]. How many of the shown loop instructions need to be modified?
   - 0
   - 1
   - 2

CHALLENGE ACTIVITY | 4.8.1: Arrays in assembly.

459784.3174716.qx3zqy7

Strings

In C, a ***string*** is an array of characters. Each character is stored as a number, being the character's ASCII value. The last element in a C string is always the ***null character*** '\0', whose ASCII value is 0.

PARTICIPATION ACTIVITY | 4.8.4: A C string is an array of ASCII values, ending with 0.

Animation captions:

1. In C, a string is an array of characters, each an ASCII number. Suppose string s has base 5100. For "Hi", H is 72, and i is 105, which are stored in s' first two elements.
2. In C, the last character in a string array is the null character '\0', whose ASCII value is 0. An extra element is always required, so "Hi" has 3 elements, not just 2.
3. The ending null character allows a loop to detect the string's end, without knowing the array's size.

A programmer can leave the array size blank, as in `char myStr[] = "Hi";`. The compiler will create an array with the appropriate number of elements, in this case 3, with the last element being the null character.

A character is 8 bits (one byte), while a memory word is 32 bits. Thus, in MIPS, a character array is stored with four characters per word, with each successive element having an address incremented by 1 (not 4). MIPS has instructions lb (load byte) and sb (store byte) to access bytes within a word. However, for simplicity of introduction, MIPSzy only has lw (load word) and sw (store word), and thus packing four characters per word is not discussed here.

PARTICIPATION ACTIVITY 4.8.5: C strings.

1) For char s[4] = "Hey", what character is s[1]?

Check **Show answer**

2) For char s[] = "Hiya", a compiler creates an array with how many elements?

Check **Show answer**

3) For char s[] = "a0b1", what is the value stored in s[1]? (Note: Use an ASCII lookup table on the web).

Check **Show answer**

4) For char s[] = "0123", what is the ASCII value of s[4]?

Check **Show answer**

5) char s[] = "1234567" requires 8

words in MIPSzy but only _____
words in MIPS.

Check Show answer

4.9 Compilers

Basic compiler operations

A ***compiler*** is a program that converts a program in a high-level language, like C, C++, or Java, into assembly instructions.

A modern compiler typically has three parts:

- A compiler's ***front end*** ensures the program is valid according to the language's rules, and converts the program to an intermediate representation (IR). Ex: `y = 6 * y;` is valid in C, but `y = 6y;` is not.
- A compiler's ***optimizations*** simplify the intermediate representation.
- A compiler's ***back end*** converts the intermediate representation to a processor's assembly instructions.

PARTICIPATION ACTIVITY | 4.9.1: A compiler converts a high-level language program into assembly instructions.

Animation captions:

1. A compiler converts a high-level program, like C, C++, or Java, into assembly instructions.
2. Modern compilers typically have three parts. The front end ensures the program is valid, then generates an intermediate representation of the program.
3. The optimizations seek to simplify the IR. Here, optimization modifies the IR to reduce the number of registers needed.
4. The back end converts the IR to a specific processor's assembly instructions.

PARTICIPATION ACTIVITY | 4.9.2: Compiler.

Which compiler would be responsible for the following tasks?

1) Implement the operation `x = y * 2` as the instruction `add $t3, $t2, $t2`.

- Front end
- Optimization
- Back end

2) Detect and report a syntax error for the following C statement.

`if numVal < 5 {`

- Front end
- Optimization
- Back end

3) Determine which MIPS register should hold the result of the operation `w = x + 100`.

- Front end
- Optimization
- Back end

4) Reduce the number of operations needed to implement the following.

```
x = (a * b)
y = (a * c)
z = x + y
```

- Front end
- Optimization
- Back end

Memory organization

Program memory refers to all memory contents used by a program, including both instructions and data. Program memory is typically organized into several regions (or segments):

- The ***code*** region contains a program's instructions. The code region is also called the ***text*** region.
- The ***static*** region contains global variables (variables defined outside any function) and static local variables (variables defined inside functions starting with the keyword "static").
- The ***stack*** contains values used to call functions and may also contain a function's local variables.
- The ***heap*** contains all dynamically allocated memory.. Ex: The malloc() function allocates memory in the heap, and the free() function deallocates memory in the heap.

When compiling a C program, the compiler determines to which memory region each variable should be located, by determining a specific memory address for each variable, and generating the instructions to initialize the variable, if needed.

PARTICIPATION ACTIVITY | 4.9.3: Program memory organization.

Animation content:

undefined

Animation captions:

1. Program memory is organized into four regions: code, static, stack, and heap.
2. The compiler selects a memory location in the static region for each global variable (and static

variable). currVals is assigned to location 4096, maxVals to 4100, and minVal to 4104.

3. If a variable is initialized, the compiler generates assembly instructions that store the initial value to variable's memory location. currVals is initialized to 0 by storing 0 to memory location 4096.
4. The program's instructions are contained within the code region, which corresponds to the instruction memory.
5. The compiler compiles each subroutine and finds a location in the code region for the subroutines's instructions. A subroutine's local variable can be implemented using registers or the stack.

PARTICIPATION ACTIVITY 4.9.4: Program memory organization.

Refer to the animation above.

1) In what memory location is minVal located?

- 4096
- 4100
- 4104

2) CalcMax's local variable max will be contained in the static region.

- True
- False

3) Which instructions might the compiler generate to initialize maxVals to 7?

```
addi $t2, $zero, 7
sw $t2, 0($t1)
```

```
addi $t1, $zero, 4100
addi $t2, $zero, 7
sw $t2, 0($t1)
```

```
addi $t1, $zero, 4100
addi $t2, $zero, -1
sw $t2, 0($t1)
```

4) For the following function, in which region would the variable lastVal be located?


```
int DiffFromLast(int newVal) {
   static int lastVal = 0;
   int valDiff = 0;

   valDiff = lastVal - newVal;
   lastVal = newVal;

   return valDiff;
}
```

- Code
- Stack

data and bss regions

*The static region is often divided into two regions: data and bss. The **data** region contains data that is initialized. Ex: A global variable declaration `int currSize = 4;` initializes the variable to 4, so would be allocated to the data region. The **bss** region (short for block started by symbol -- a term carried over from early assemblers) contains data that is uninitialized. Ex: A static variable declared as `static int lastReading;` does not initialize the variable to 4, so would be allocated to the bss region.*