

## Assignment 11 - Write an assembler

12/6/2024

220 Points Possible

Attempt 1



In Progress

NEXT UP: Submit Assignment



Add Comment

Unlimited Attempts Allowed

Details

## 11 Assembly to Machine Code

In this final assignment, we will be creating an assembler capable of translating assembly programs into a format that you can run on your processor!



### Objectives

1. Demonstrate understanding of instruction formats
2. Demonstrate mastery of assembly
3. Understand the software hierarchy of assembly code, down to individual logic gates


### Overview

In this assignment, you will be making an assembler, capable of taking in assembly code and outputting the translated machine code. I am highly encouraging/requiring the use of Python for this assignment. If you want to do this assignment in another language, you must send me an email requesting approval to use that language (and also so I can plan out grading). Any programs submitted in a non-Python language, without being approved first will not be able to be graded.

### Setup

**Overview:** To begin, I have a starter file, which you can use (not required to), that gives an idea of what the structure of your program should look like.

Files:

- [starter.py \(https://usu.instructure.com/courses/759913/files/93295243?wrap=1\)](https://usu.instructure.com/courses/759913/files/93295243?wrap=1)    
([https://usu.instructure.com/courses/759913/files/93295243/download?download\\_frd=1](https://usu.instructure.com/courses/759913/files/93295243/download?download_frd=1))
- Computer.circ - Fully functioning Computer for you to test your programs on (will add after Assignment 10 has fully passed)

### Resources

Instruction Format Table:

Core Instruction Set	Syntax	format	op	funct
add	add	R	0000	010
subtract	sub	R	0000	110
and	and	R	0000	000
or	or	R	0000	001
set less than	slt	R	0000	111
load word	lw	I	0001	xxx
store word	sw	I	0010	xxx
branch on equal	beq	I	0011	xxx
jump	j	J	0100	xxx
branch not equal	bne	I	0110	xxx
jump register	jr	R	0111	xxx
Jump and Link	jal	J	1000	xxx
Display	display	R	1111	xxxx

Name	Bit Fields					Notes (16 bits total)
	bits 15-12	bits 11-9	bits 8-6	bits 5-3	bits 2-0	
R-Format	op	rs	rt	rd	funct	Arithmetic, logic
I-Format	op	rs	rt	address/ immediate (6 bits)		Load/store, branch, immediate
J-Format	op	target address (12 bits)				Jump

## Things to Know

**Overview:** Here are a few concepts that will be useful on this assignment (if you find more resources that are useful, please let me know and I can add to this list)

### Resources:

- **Dictionaries:** You will want to use these for creating the label and data tables.
  - <https://www.programiz.com/python-programming/dictionary> ↗ (<https://www.programiz.com/python-programming/dictionary>)
- **Writing to a file:** Very necessary for producing output files and reading input files:
  - <https://www.programiz.com/python-programming/examples/read-line-by-line> ↗ (<https://www.programiz.com/python-programming/examples/read-line-by-line>)
- **String formatting:** Useful for easily converting strings to binary/decimal/hexadecimal format
  - <https://www.pythonmorsels.com/string-formatting/> ↗ (<https://www.pythonmorsels.com/string-formatting/>)
- **Returning Multiple Items:** Useful for some of the functions you are building
  - <https://www.programiz.com/python-programming/examples/multiple-return-values> ↗ (<https://www.programiz.com/python-programming/examples/multiple-return-values>)
- **Pattern Matching:** Helpful for when you start encoding the instructions!
  - <https://peps.python.org/pep-0636/> ↗ (<https://peps.python.org/pep-0636/>)

## Task 1 - Setup and Preprocessing

**Overview:** Before you start translating into machine code, you need to prepare your file for translation. This includes removing any comments, and trimming extra whitespace to make assembling easier later.

**Starter Code:** The following starter code loads all lines of a file into a list of strings. Feel free to use this as a starter point for this program.

#### Requirements:

- Create a preprocess function that takes in the **lines** variable, and performs the following:
  - Remove any comments from the code. This includes inline comments (comments after a line of code), and comments that take up their own line
  - Strip out any remaining whitespace from both the start and end of each line
  - Remove any lines containing only blank space
  - Return the preprocessed list of instructions. At this point, the list should only contain one string per instruction

#### Hints:

- Python's **strip()** and **split()** functions are helpful here

#### Test:

Use the following test assembly files and preprocessed line outputs to get an idea for what you should be doing in this step:

#### Test Input:

```
main:
# This is the start of my countdown program!
addi R1, R0, 0 # i = 0
addi R2, R0, 10 # stop at 10

loop:
    beq R1, R2, end_loop # Check to see if we've reached 10

    addi R1, R1, 1 #i += 1

    j loop

end_loop:
end_program:
```

#### Preprocessed Output:

```
['main:', 'addi R1, R0, 0', 'addi R2, R0, 10', 'loop:', 'beq R1, R2, end_loop', 'addi R1, R1, 1', 'j loop', 'end_loop:', 'end_program:']
```

## Task 2 - Create Data Table

**Overview:** Before tackling the program itself, it is helpful to construct a table/dictionary/mapping to keep track of the memory addresses in the **.data** section that each variable points to. We need to do this for two reasons. First, is that we need a file to load all of our data into RAM. Second, is we need to know which labels represent which memory address so that we can decode symbols within our program (for **lw** and **sw** instructions).

#### Requirements:

- Create a function, **build\_data\_table**. This function should take your program , and parse the **.data** section, creating a label table storing a **label -> memory address** mapping. This function should also remove the **.data** section from your source code.
  - **Inputs:**
    - **lines** - the preprocessed list of lines representing your program
  - **Outputs:**

- A mapping of label -> memory address. Assume that the memory addresses start at 0, and each variable is located 1 memory address apart.
- A list of each data value. This will be used to load a file containing your data into the **RAM** component in Logisim.
- The new instructions without the **.data** section
- The function should work with three types of programs:
  - A program with both a **.data** and **.text** header
  - A program with just a **.text** header
  - A program without any header at all (in this case, assume it is all source code and no data)
  - No other structures of programs need to be supported

### Test Inputs:

```
.data
word1: 10
word2: 20
word3: 30
result: 0

.text
main:
    # Load the values of word1, word2, and word3 into registers
    lw R0, word1
    lw R1, word2
    lw R2, word3

    # Add word1 and word2, and store the result in $t3
    add R3, R0, R1

    # Subtract word3 from the sum, and store the result in $t4
    sub R4, R3, R2

    # Add 10 to the result, and store the result in $t5
    addi R5, R4, 10

    # Store the final result in the 'result' variable
    sw R5, result
```

### Data Table 1:

```
{'word1': 0, 'word2': 1, 'word3': 2, 'result': 3}
```

### Data List 1:

```
[10, 20, 30, 0]
```

### Processed program 1:

```
['main:', 'lw R0, word1', 'lw R1, word2', 'lw R2, word3', 'add R3, R0, R1', 'sub R4, R3, R2', 'addi R5, R4, 10', 'sw R5, result']
```

### Test Input 2:

```
.text
# Start of program

main:
    # Load immediate values into registers
    li R1, 10
    li R2, 20
    li R3, 30

    # Add the values in R2 and R3, and store the result in R4
    add R4, R2, R3

    # Subtract the value in R4 from the value in R1, and store the result in R5
    sub R5, R4, R1

    # Add the result by 2, and store the result in R6
    addi R6, R5, 2

    # End of program
    j end
```

```
end:
```

**Data Table 2**

```
{}
```

**Data List 2**

```
[]
```

**Processed Program 2:**

```
['main:', 'li R1, 10', 'li R2, 20', 'li R3, 30', 'add R4, R2, R3', 'sub R5, R4, R1', 'addi R6, R5, 2', 'j end', 'end:']
```

**Test Input 3:**

```
# Load immediate values into registers
li R1, 10
li R2, 20
li R3, 30

# Add the values in R2 and R3, and store the result in R4
add R4, R2, R3

# Subtract the value in R4 from the value in R1, and store the result in R5
sub R5, R4, R1

# Add the result by 2, and store the result in R6
addi R6, R5, 2

# End of program
j end

end:
```

**Data Table 2**

```
{}
```

**Data List 2**

```
[]
```

**Processed Program 2:**

```
['li R1, 10', 'li R2, 20', 'li R3, 30', 'add R4, R2, R3', 'sub R5, R4, R1', 'addi R6, R5, 2', 'j end', 'end:']
```

## Task 3 - Create Label Table

**Overview:** Before translating the instructions in the program, it is helpful to construct a table to keep track of which lines of code each label refers to.

**Requirements:**

- Create a **build\_label\_table** function. This function should take the preprocessed lines, construct a mapping of label -> instruction number, and also remove the labels from the source code.
  - Labels should not count as an instruction. So if you are keeping a count of instructions, labels should not affect that count
  - The function should return a mapping of labels -> instruction number, as well as the modified list of instructions

**Test Input:**

```
main:
# This is the start of my countdown program!
addi R1, R0, 0 # i = 0
addi R2, R0, 10 # stop at 10

loop:
    beq R1, R2, end_loop # Check to see if we've reached 10

addi R1, R1, 1 #i += 1

j loop

end_loop:
end_program:
```

**Label Table:**

```
{'main': 0, 'loop': 2, 'end_loop': 5, 'end_program': 5}
```

**New Instructions:**


```
['addi R1, R0, 0', 'addi R2, R0, 10', 'beq R1, R2, end_loop', 'addi R1, R1, 1', 'j loop']
```

## Task 4 - Encode Instruction


**Overview:** In this task you will create a function that can convert an instruction into its binary format, and then use that function to translate the entire program into binary. This is the meat of the assembler

**Requirements:**

- Create a function, **encode\_instruction**
  - **Inputs:**
    - **line\_num** - the index or line number of the instruction to be processed
    - **instruction** - the string value containing the instruction to be processed (addi, R5, R2, 10, etc...)
    - **label\_table** - the label table you created in an earlier step
    - **data\_table** - the data table you created in an earlier step
  - **Outputs:**
    - The binary encoding of the line. This should be 16 bits (as a string), and should contain the relevant fields for the given instruction.
- The function should be able to handle to process the following instructions:
  - **add**
  - **sub**
  - **and**
  - **or**
  - **slt**
  - **addi**
  - **beq**
  - **bne**
  - **lw**
  - **sw**
  - **j**
  - **jr**
  - **jal**
  - **display**
- **lw** and **sw** should be able to support both of the following formats:
  - **lw R1, 10(R2)**

- **lw R3, result**
- If the second result is encountered, use **R0** as the base register, and determine the immediate to use for the instruction encoding.
- If an invalid instruction is given, an error should be raised: <https://docs.python.org/3/tutorial/errors.html> 
- Wrap this function in another function, **encode\_program**
  - **Inputs:**
    - **lines** - the list of lines that have been processed from the past two tasks
    - **label\_table** - the label table that was created earlier
    - **data\_table** - the data table that was created earlier
  - **Outputs:**
    - A list of binary encoded instructions

### Hints:

- Pattern matching or if/elif/else ladders are very helpful for this. Check out python's pattern matching documentation: <https://peps.python.org/pep-0636/>  (<https://peps.python.org/pep-0636/>)
- Make sure when processing immediate, you can handle negative/positive immediates (negatives must be represented using two's complement)
- The following helper functions could be useful
  - **register\_to\_binary** - takes the string form of a register, and converts it to a binary string
  - **dec\_to\_bin** - Converts a decimal number to a binary string, accounting for both negatives and positives
    - The `bin()` function can convert numbers to a binary string
    - Python's f-strings can also be helpful for this
      - Code snippet that will transform a negative number into the correct 16-bit two's complement form: `f"({1 << 16} + num:016b)"`
- For encoding labels into an immediate, use this as a guide
  - Branch instructions use the immediate as an offset for how many instructions forward or backwards to branch. For example, with the code:
 

```
...
47: beq R1, R1, goto_label
48: add R2, R3, R4
49: sub R3, R4, R5
--: goto_label:
50: addi R3, R3, 10
```

We want to branch to the `addi` instruction (instruction #50). So the immediate offset would be 2 (010). This works because we are branching two instructions down (line 49), and then the processor always does `PC+4`, taking us to the next line (50), which is where the branch instruction should take us.

- Jump instructions are jumps not branches, so their immediate value is just the instruction number that you want to jump to
- When there is a label in **lw** or **sw** instructions, the immediate value should just be the address of that label (which you stored in an earlier step).

### Test Input:

For the below example outputs, I separate the logical fields of the instruction by a space for easier visualization of the fields. You are not required to do this.

### Test 1:

```
main:
# This is the start of my countdown program!
addi R1,R0, 0 # i = 0
addi R2, R0,10 # stop at 10
```

```

loop:
beq R1, R2, end_loop # Check to see if we've reached 10

addi R1,R1,1 #i += 1

j loop

end_loop:
end_program:

```

**Output 1:**

```

0101 000 001 000000
0101 000 010 001010
0011 010 001 000010
0101 001 001 000001
0100 000000000010

```

**Test 2:**

```

main:
    lw      R0, 0(R1)      # Load word from memory address of R1 into R0
    lw      R1, 4(R2)      # Load word from memory address of R2 + 4 into R1

    add     R2, R0, R1     # Add R0 and R1, store result in R2
    sw     R2, -4(R3)      # Store word in R2 to memory address of R3 - 4

    sub     R3, R2, R0     # Subtract R0 from R2, store result in R3

    and     R4, R2, R3     # Logical AND of R2 and R3, store result in R4
    or      R5, R2, R3     # Logical OR of R2 and R3, store result in R5

    slt     R6, R4, R5     # Set R6 to 1 if R4 < R5, else 0

    addi    R7, R6, 5      # Add immediate value 5 to R6, store result in R7

    beq     R7, R6, equal  # If R7 equals R6, branch to label 'equal'
    j       continue      # Unconditional jump to label 'continue'

equal:
    addi    R7, R7, 1      # Just an example, increment R7 to show we were here

continue:
    bne     R7, R6, notequal # If R7 not equal to R6, branch to label 'notequal'
    j       end            # Jump to end, skipping the notequal section

notequal:
    sub     R7, R7, R6     # Example operation for not equal path

end:
    # End label, could be used for graceful exit
    # End of program. In real assembly, you might want to loop or exit.
    # Since this is for assembler testing, we'll end here.

```

**Output 2:**

```

0001 001 000 000000
0001 010 001 000100
0000 000 001 010 010
0010 011 010 111100
0000 010 000 011 110
0000 010 011 100 000
0000 010 011 101 001
0000 100 101 110 111
0101 110 111 000101
0011 110 111 000001
0100 000000001100
0101 111 111 000001
0110 110 111 000001
0100 000000001111
0000 111 110 111 110

```

**Test 3:**

```

main:                                # Main program label

    jal     printMessage             # Call the subroutine printMessage

```



```

# More main program code could go here

jr      R5          # Return from main (if in a larger system)

printMessage:        # Subroutine to print a message
# Code to print the message would go here.
# This is just a placeholder to demonstrate `jal` and `jr`.

jr      R7          # Return from subroutine

```

### Output 3:

```

1000 000000000010
0111 101 000 000 000
0111 111 000 000 000

```

### Test 4:

```

.data
var1: 10
var2: 20
var3: 30
var4: 40
var5: 50
result: 0

.text
main:
# Load variables into registers
lw R1, var1
lw R2, var2
lw R3, var3
lw R4, var4
lw R5, var5

# Perform arithmetic operations
add R6, R1, R2    # Add var1 and var2, store the result in R6
sub R7, R3, R4    # Subtract var4 from var3, store the result in R7

# Call the multiplication function to multiply var5 by 2
add R1, R5, R0    # Move var5 to R1 (first argument)
addi R2, R0, 2    # Set the second argument (multiplier) to 2
jal multiply      # Call the multiply function
add R6, R1, R0    # Move the result from R1 to R6

# Store the results back into memory
sw R6, result    # Store the sum of var1 and var2 in 'result'
sw R7, var4      # Update var4 with the difference of var3 and var4
sw R6, var5      # Update var5 with its doubled value

# Load the updated values back into registers
lw R4, var4
lw R5, var5

# Perform more arithmetic operations
add R6, R6, R4    # Add the sum of var1 and var2 with the updated var4
sub R7, R5, R6    # Subtract the doubled var5 from the updated var5

# Store the final results back into memory
sw R6, result    # Store the final sum in 'result'
sw R7, var5      # Update var5 with the final difference

# End of program
j end

multiply:
# Multiplication function
# Arguments: R1 - multiplicand, R2 - multiplier
# Returns: R1 - product
addi R3, R0, 0    # Initialize the product to 0
addi R4, R0, 0    # Initialize the loop counter to 0

multiply_loop:
beq R4, R2, multiply_end # If the loop counter equals the multiplier, end the loop
add R3, R3, R1        # Add the multiplicand to the product
addi R4, R4, 1        # Increment the loop counter
j multiply_loop       # Jump back to the start of the loop

multiply_end:
add R1, R3, R0    # Move the final product to R1
jr R7            # Return from the multiplication function

```

```
end:
```

**Output 4:**

```
0001 000 001 000000
0001 000 010 000001
0001 000 011 000010
0001 000 100 000011
0001 000 101 000100
0000 001 010 110 010
0000 011 100 111 110
0000 101 000 001 010
0101 000 010 000010
1000 000000010101
0000 001 000 110 010
0010 000 110 000101
0010 000 111 000011
0010 000 110 000100
0001 000 100 000011
0001 000 101 000100
0000 110 100 110 010
0000 101 110 111 110
0010 000 110 000101
0010 000 111 000100
0100 000000011101
0101 000 011 000000
0101 000 100 000000
0011 010 100 000011
0000 011 001 011 010
0101 100 100 000001
0100 000000010111
0000 011 000 001 010
0111 111 000 000 000
```

**Task 5 - Prepare for Logisim**

**Overview:** The last step of our assembler is to convert our binary encodings into a format suitable for Logisim and write it to an output file

**Requirements:**

- Create a function, **post\_process**
  - **Inputs:**
    - **lines:** a list of binary strings obtained from the prior steps
  - **Outputs:**
    - A list of hexadecimal strings, which are the result of converting the binary strings to hexadecimal
- Write the hex program to an output file
  - The first line of the output file must be: v3.0 hex words addressed
  - The next line of the output file should start with 00: and be followed by each hex instruction with a space in between
- Write the list of data to an output file
  - The first line of the output file must be: v3.0 hex words addressed
  - The next line of the output file should start with 00: and be followed by each data value (converted to hex) with a space in between
- 

**Test.asm:**

```
main:
# This is the start of my countdown program!
addi R1, R0, 0 # i = 0
addi R2, R0, 10 # stop at 10

loop:
    beq R1, R2, end_loop # Check to see if we've reached 10

    addi R1, R1, 1 # i += 1

j loop
```

```
end_loop:
end_program:
```

**Program.hex:**

```
v3.0 hex words addressed
00: 5040 508a 3442 5241 4002
```

**Test 2:**

```
.data
var1: 10
var2: 20
var3: 30
var4: 40
var5: 50
result: 0

.text
main:
    # Load variables into registers
    lw R1,var1
    lw R2, var2
    lw R3, var3
    lw R4,var4
    lw R5, var5

    # Perform arithmetic operations
    add R6, R1, R2    # Add var1 and var2, store the result in R6
    sub R7, R3, R4    # Subtract var4 from var3, store the result in R7

    # Call the multiplication function to multiply var5 by 2
    add R1, R5, R0    # Move var5 to R1 (first argument)
    addi R2, R0, 2    # Set the second argument (multiplier) to 2
    jal multiply      # Call the multiply function
    add R6, R1, R0    # Move the result from R1 to R6

    # Store the results back into memory
    sw R6, result    # Store the sum of var1 and var2 in 'result'
    sw R7, var4      # Update var4 with the difference of var3 and var4
    sw R6, var5      # Update var5 with its doubled value

    # Load the updated values back into registers
    lw R4,var4
    lw R5, var5

    # Perform more arithmetic operations
    add R6, R6, R4    # Add the sum of var1 and var2 with the updated var4
    sub R7, R5, R6    # Subtract the doubled var5 from the updated var5

    # Store the final results back into memory
    sw R6, result    # Store the final sum in 'result'
    sw R7, var5      # Update var5 with the final difference

    # End of program
    j end

multiply:
    # Multiplication function
    # Arguments: R1 - multiplicand, R2 - multiplier
    # Returns: R1 - product
    addi R3, R0, 0    # Initialize the product to 0
    addi R4, R0, 0    # Initialize the loop counter to 0

    multiply_loop:
        beq R4,R2,multiply_end    # If the loop counter equals the multiplier, end the loop
        add R3, R3,R1              # Add the multiplicand to the product
        addi R4, R4, 1             # Increment the loop counter
        j multiply_loop            # Jump back to the start of the loop

    multiply_end:
        add R1, R3, R0             # Move the final product to R1
        jr R7                      # Return from the multiplication function

end:
```

**Program.hex:**

```
v3.0 hex words addressed
00: 1040 1081 10c2 1103 1144 02b2 073e 0a0a 5082 8015 0232 2185 21c3 2184 1103 1144 0d32 0bbe 2185 21c4 401d 50c0 5100 3503 065a 590
```

```
1 4017 060a 7e00
```

**Data.hex:**

```
v3.0 hex words addressed
00: 000a 0014 001e 0028 0032 0000
```

**Test 3 (Pong):**

```
.data
screenStart: 16384      # 0x4000
screenEnd: 24576        # 0x6000
ballColor: 65535        # 0xFFFF (white in RGB565)
rowShift: 128
delay: 2500
myDelay: 10000
xVelocity: 1
yVelocity: 1
xEdge: 126
yEdge: 62

# Should start on row 31, column 64-65
# 31 * 128 + 63
startPix: 20415
ballPix: 20415

# right paddle started at 110 (128 - 18)
rightPaddleStart: 19566
rightPaddlePix: 19566

# Left paddle starts at
leftPaddleStart: 19474
leftPaddlePix: 19474

keyboardMem: 24576
upArrow: 38
downArrow: 40
wKey: 87
sKey: 83
spaceKey: 32

leftScore: 0
rightScore: 1

.text
j main

# Delays for 10000 ticks
debugLoop:
    lw R1, myDelay
    myDebugLoop:
        addi R1, R1, -1

        bne R1, R0, myDebugLoop

    jr R7

# This will clear the screen, reset the positions of the paddles/ball and wait for the user to press start
resetGame:
    # Increment SP
    addi R6, R6, -1
    sw R7, 0(R6)

    # Call the clear Screen function
    jal clearScreen

    # Reset ball/paddle position
    lw R1, startPix
    sw R1, ballPix

    lw R1, rightPaddleStart
    sw R1, rightPaddlePix

    lw R1, leftPaddleStart
    sw R1, leftPaddlePix

    # Then, draw everything to the screen
    lw R1, ballColor
    jal drawBall
    jal drawPaddle
    display
```

```

# Finally, run a delay loop waiting for the user
lw R1, spaceKey
userResetLoop:
    lw R2, keyboardMem
    lw R2, 0(R2)
    bne R1, R2, userResetLoop

# Load the return address and return
lw R7, 0(R6)
addi R6, R6, 1
jr R7

clearScreen:
    lw R1, screenStart
    lw R2, screenEnd
clearLoop:
    beq R1, R2, clearLoopEnd
    sw R0, 0(R1)      # Write black (0) to current pixel
    addi R1, R1, 1
    j clearLoop

clearLoopEnd:

    jr R7

# R1 (arg)= pixel
# R1 (return value) = column
# R2 (return value) = row of that pixel
getRowCol:

    # Subtract the start pixel
    lw R2, screenStart
    sub R1, R1, R2

    # Run a loop to get the row/column
    addi R2, R0, 0 # Row
    lw R3, rowShift
    modulus_loop:
        # Check to see if R1 < 128. If so, then
        slt R4, R1, R3
        bne R4, R0, modulus_loop_end

        addi R2, R2, 1 # increment row counter
        sub R1, R1, R3 # subtract 128 from r1

        j modulus_loop

    modulus_loop_end:

    jr R7

# Returns if the ball is touching any walls and reverses velocity if needed
checkBounce:
    # First, store the return address on the stack
    addi R6, R6, -1
    sw R7, 0(R6)

    # Get the row/column of the ball
    lw R1, ballPix
    jal getRowCol

    # Check collisions

    # Left Wall
    beq R1, R0, hitLeft

    # Right Wall
    lw R3, xEdge
    beq R1, R3, hitRight

    # Top Wall
    beq R2, R0, hitTop

    # Bottom Wall
    lw R3, yEdge
    beq R2, R3, hitBottom

    # Right Paddle (2 pixels to the right of the ball)
    lw R1, ballPix
    lw R2, 2(R1)
    lw R3, ballColor
    beq R3, R2, hitRightPaddle

```

```

# Check to see if 1 pixels to the left of the ball is white
lw R1, ballPix
lw R2, -1(R1)
lw R3, ballColor
beq R3, R2, hitLeftPaddle

j bounce_done

# If we hit the left wall, add 1 to left score and reset the ball
hitLeft:
    lw R1, leftScore
    addi R1, R1, 1
    sw R1, leftScore
    jal resetGame
    j bounce_done

hitRight:
    lw R1, rightScore
    addi R1, R1, 1
    sw R1, rightScore
    jal resetGame
    j bounce_done

hitTop:
    addi R1, R0, 1
    sw R1, yVelocity
    j bounce_done

hitBottom:
    addi R1, R0, -1
    sw R1, yVelocity
    j bounce_done

hitRightPaddle:
    addi R1, R0, -1
    sw R1, xVelocity
    j bounce_done

hitLeftPaddle:
    addi R1, R0, 1
    sw R1, xVelocity
    j bounce_done

bounce_done:
    lw R7, 0(R6)
    addi R6, R6, 1
    jr R7

# Moves the ball in it's direction
moveBall:
    # First, add the x velocity to the ball's position
    lw R1, xVelocity
    lw R2, ballPix
    add R2, R2, R1 #pix = pix + velocity

    # Then, check whether the y velocity is -1 or 1
    addi R1, R0, 1
    lw R3, yVelocity
    lw R4, rowShift
    # if yvelocity == 1, then add the row shift, otherwise subtract
    bne R3, R1, moveUp
    # By default if R3 == R1, we move down
    moveDown:
        add R2, R2, R4
        j moveDone

    moveUp:
        sub R2, R2, R4

    moveDone:

    # Store the new ball pixel in memory
    sw R2, ballPix

    jr R7

# Checks for a keyboard input and moves the paddle if so
movePaddles:
    lw R1, keyboardMem
    lw R1, 0(R1)

    lw R2, upArrow
    beq R1, R2, rightUp

    lw R2, downArrow

```

```

    beq R1, R2, rightDown

    lw R2, wKey
    beq R1, R2, leftUp

    lw R2, sKey
    beq R1, R2, leftDown

    j leaveMovePaddle

rightUp:
    lw R1, rightPaddlePix
    lw R2, rowShift
    sub R1, R1, R2
    sw R1, rightPaddlePix
    j leaveMovePaddle

rightDown:
    lw R1, rightPaddlePix
    lw R2, rowShift
    add R1, R1, R2
    sw R1, rightPaddlePix
    j leaveMovePaddle

leftUp:
    lw R1, leftPaddlePix
    lw R2, rowShift
    sub R1, R1, R2
    sw R1, leftPaddlePix
    j leaveMovePaddle

leftDown:
    lw R1, leftPaddlePix
    lw R2, rowShift
    add R1, R1, R2
    sw R1, leftPaddlePix
    j leaveMovePaddle

leaveMovePaddle:

jr R7

# R1 = ball color
drawBall:
    # Load the current position of the ball into R2
    lw R2, ballPix

    # Draw the top 2 pixels
    sw R1, 0(R2)
    sw R1, 1(R2)

    # Shift the address down by a row
    lw R3, rowShift
    add R2, R2, R3

    # Draw the bottom 2 pixels
    sw R1, 0(R2)
    sw R1, 1(R2)

    jr R7

# R1 = color
drawPaddle:
    # First, draw the right paddle
    lw R2, rightPaddlePix
    lw R5, leftPaddlePix

    # Then, draw 5 offsets across and increas
    addi R3, R0, 14
paddleLoop:
    sw R1, 0(R2)
    sw R1, 0(R5)
    sw R1, 1(R2)
    sw R1, 1(R5)
    sw R1, 2(R2)
    sw R1, 2(R5)

```

```

    # Then, shift R2 down by 128
    lw R4, rowShift
    add R2, R2, R4
    add R5, R5, R4

    addi R3, R3, -1

    bne R3, R0, paddleLoop

jr R7

# Delays by the specified delay time
delay:
    addi R1, R0, 0
    lw R2, delay
delayLoop:
    beq R1, R2, delayLoopEnd
    addi R1, R1, 1
    j delayLoop

delayLoopEnd:
jr R7

main:
    # Clear the screen
    jal resetGame

drawLoop:

    # Check for a bounce
    jal checkBounce

    # First, clear the ball
    addi R1, R0, 0
    jal drawBall

    # Clear paddles
    addi R1, R0, 0
    jal drawPaddle

    # Then, move the ball
    jal moveBall

    # Move paddle
    jal movePaddles

    # Then, draw the ball again
    lw R1, ballColor
    jal drawBall

    # Draw the paddle
    lw R1, ballColor
    jal drawPaddle

    # Draw the current score as a series of boxes

    # Display
    display

    j drawLoop

exit:
    j exit

# Clear screen

```

## Program.hex

v3.0 hex words addressed

```

00: 409f 1045 527f 607e 7e00 5dbf 2dc0 8019 104a 204b 104c 204d 104e 204f 1042 8082 808a f000 1055 1090 1480 647d 1dc0 5d81 7e00 104
0 1081 3443 2200 5241 401b 7e00 1080 028e 5080 10c3 02e7 6103 5481 02ce 4024 7e00 5dbf 2dc0 104b 8020 304e 10c8 3651 3095 10c9 3696
104b 1282 10c2 34d5 104b 12bf 10c2 34d4 4053 1056 5241 2056 8005 4053 1057 5241 2057 8005 4053 5041 2047 4053 507f 2047 4053 507f 20
46 4053 5041 2046 4053 1dc0 5d81 7e00 1046 108b 0452 5041 10c7 1103 62c2 0512 4060 0516 208b 7e00 1050 1240 1091 3447 1092 344a 1093
344d 1094 3450 4081 104d 1083 028e 204d 4081 104d 1083 028a 204d 4081 104f 1083 028e 204f 4081 104f 1083 028a 204f 4081 7e00 108b 24

```



```
40 2441 10c3 04d2 2440 2441 7e00 108d 114f 50ce 2440 2a40 2441 2a41 2442 2a42 1103 0512 0b2a 56ff 60f5 7e00 5040 1084 3442 5241 409b
7e00 8005 802a 5040 8082 5040 808a 8056 8062 1042 8082 1042 808a f000 40a0 40ad
```

## Data.hex

v3.0 hex words addressed

```
00: 4000 6000 ffff 0080 09c4 2710 0001 0001 007e 003e 4fbf 4fbf 4c6e 4c6e 4c12 4c12 6000 0026 0028 0057 0053 0020 0000 0001
```

## Task 6 - Writing your Own Program

**Overview:** To tie it all together, write a program that makes use of the keyboard/screen and assemble it. This task is very much up to you! You could make a game, a calculator, or anything interesting!

### Requirements:

- Your program must include the following elements
  - Loading or Storing from/to data in the **.data** section
  - At least one function (can be leaf or non-leaf)
  - At least one loop
  - Has some kind of graphical output
  - Must be non-trivial
    - This is a very loose requirement, but as an example of a "trivial" program, writing a program that draws a single pixel to the screen would not qualify

### Testing:

Once you write your code, you should be able to assemble it and test it on the CPU linked at the beginning! Load the **data.hex** file into **RAM** and the **program.hex** into **ROM**, and you should be able to run your own-program on the hardware you've been building all semester!



## Submission

Submit the following two files:

- assembler.py** - A python file containing your assembler program
  - This file should produce a **program.hex** and **data.hex** file when run with proper input
- your\_program.asm** - An assembly file (can be named anything) containing your program for task 6

### View Rubric

#### Assignment 12

Criteria	Ratings			Pts
Task 1: Preprocessing <a href="#">view longer description</a>	10 pts Full Marks	5 pts Half Marks Only 1 criteria is met	0 pts No Marks	/ 10 pts
Task 2: Data Table <a href="#">view longer description</a>	15 pts Full Marks	10 pts Some Errors Only 2 criteria met	5 pts Errors Only 1 criteria met	0 pts No Marks / 15 pts
Task 3: Label Table <a href="#">view longer description</a>	20 pts Full Marks	10 pts Half Marks 1 criteria met	0 pts No Marks	/ 20 pts

Assignment 12

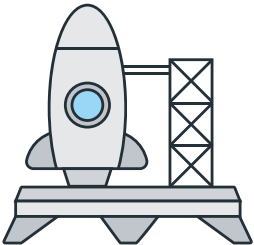
Criteria	Ratings		Pts
Task 4: Encoding Instructions <a href="#">view longer description</a>	90 pts Full Marks	0 pts No Marks	/ 90 pts
Task 5: Preparing for Logisim <a href="#">view longer description</a>	20 pts Full Marks	0 pts No Marks	/ 20 pts
Task 6: Your Own Program <a href="#">view longer description</a>	30 pts Full Marks	0 pts No Marks	/ 30 pts
Overall correctness of Assembler <a href="#">view longer description</a>	35 pts Full Marks	0 pts No Marks	/ 35 pts
			Total Points: 0

Choose a submission type

Upload

Office 365

More



Choose a file to upload  
File permitted: PY, ASM, ZIP

or

 Canvas Files



<https://usu.instructure.com/courses/759913/modules/items/6556308>



<https://usu.instructure.com/courses/759913/modules/items/6555>