Manual for MIPS Disassembler

# File information

## Input File

In the demonstration the program reads a file called “input.txt.” The “input.txt” file consists of 7 lines of hex instruction. The hex must be in lowercase and numerical hex digits. The last line of the file must be a blank line (such that the last character is a new line character)

## Output File

The program writes to the “output.asm” file. The output is the disassembled version of the instruction code from the input file. The output lines correspond to the input instruction lines.

The instructions are written in the placement they would be in programing (i.e. add rd rs rt) with a “$” for the register numbers and commas where needed. The only difference is in the display of the register. Rather than the register name, the program calculates the register number (i.e. $a0 will output $4).

## Program Files

There are six program files needed for this project. The “MAIN\_DRIVER.asm” file is the main file that will be run. The five supporting files which store macros are “extractInstruction.asm”, “numberRegister.asm”, ”RFormat.asm”, “JFormat.asm”, and “IFormat.asm”. These files are included with the main file.

## Where the files are stored

The input, output, program files, and the MARS application must be in the same file directory (folder) path so that they can be run.

# Main File: “MAIN\_DRIVER.asm”

## .data section

Broken up into sections that contain necessary data about an action

1. Exit: the exit message that will show in the pop-up after the program is complete
2. File information: Prompts, error message, and spaces necessary when opening and reading from a file.
3. Opcode: error message for an unrecognizable opcode
4. Formatting: list of all the formatting details as asciiz data types

## .text section

### Main Subroutine:

1. Opening to write the file
   1. Opens the file with the necessary syscall (13) and the output file name
   2. Saves file descriptor in $s7 for future use
   3. Opening the file for writing at the beginning ensure that all necessary writes will be properly understood and written properly with no errors.
2. **start\_spot**: Retrieves the file name from the user
   1. Prints a prompt telling the user to enter the file name.
   2. Storing the answer into data space “filename”
   3. Goes though a loop that reads the filename byte-by-byte until it reaches the newline character. Proceeds to remove the newline character from “filename” so it is properly recognized.
3. Open the file
   1. Opening the file with syscall 13
      1. Flag register a1 for the read-only option for opening the file
   2. Errors in opening the file
      1. If there are errors in opening the file, the file descriptor will be negative. The program checks for this with a branch.
      2. **open\_error:** handles an error in opening the file
         1. prints an error message to the terminal
         2. plays a few music notes to represent error reaction
         3. Close the file
         4. Restart by asking the user for input (jump to start\_spot)
4. **read\_file**: Reads the file line by line and disassembles each instruction
   1. Loop variables
      1. Buffer –space in memory to act as the reading buffer (bytes read from here)
      2. Line – space in memory to keep track of the bytes that make a line
      3. Create a “cursor” in line buffer ($s3) to keep track of moving byte by byte
   2. **read\_line\_loop**: Loop to read a line – byte by byte
      1. Call the syscall (14) to read the file (given by user). Reading buffer is set to 1 byte for byte by byte reading.
      2. Stop the loop and close the file when the file descriptor is negative, indicating the end of the file
      3. Read byte
         1. The byte is a newline character, branch to **is\_line**
            1. Replace the new lines character with a NUL
            2. Reset the “cursor” to zero for next line
            3. Disassemble the instruction (call to get\_0pcode)
            4. Print a newline on the terminal and output file
            5. Loop back to read\_line\_loop
         2. The byte is not a newline character
            1. Save the placement of this byte with the line buffer

Store the current byte into equivalent spot in line buffer

* + - * 1. Move the placement of $s1 to mimic moving a byte in line buffer

1. Ending the program
   1. Closing the file
      1. With the syscall (16)
   2. Exiting the program
      1. Play the exit music
      2. Show a pop-up on the screen indicating the program is over
      3. Use syscall (10) to exit the program

### Disassembler Portion

1. **get\_0pcode:** Extract opcode of the instruction
   1. Read in first two bytes of the line and convert the ASCII values to their understandable hex value equivalents
      1. Converting: depending on whether the ASCII is a number or alphabet, subtract a preset constant that will equate to the hex equivalent
   2. Combine the two bytes and shift right to acquire the 6 desired opcode bits.
2. **format:** Instruction format : using the opcode, find the instruction format (R, I, J)
   1. R format : opcode is always 0
      1. Branch to the R\_format subroutine
   2. I format: have three different types based on the arrangement of operands
      1. Data – consists of the lw and sw instruction. They include the immediate value as the offset.
      2. Logic – the rs comes before the rt operand (lui, ori, addi, addui instructions)
      3. Branch – the rt comes before the rs operand (beq and bne
   3. J format: one of two options
      1. Opcode is 2 – j instruction
      2. Opcode is 3 – jal instruction
   4. Opcode not recognized
      1. Prints a error message to terminal and output file
      2. Returns to reading the file line by line
3. The instruction formats are modularized into the supporting files (R, I, J). These files have macros which are specified in the supporting files and called in main.asm. These macros are called such that they return to the original reading loop.
4. Extracting each instruction component is modularized in the supporting file “extractInstruction.asm.”

### Formatting

1. **printDollar:**
   1. prints a dollar sign “ $” to the terminal and the output file (appropriate buffer of 2)

Easy to implement this as a subroutine so both actions are completed in one call

* 1. return to caller

1. **printComma:** 
   1. prints a comma “ , “ to the terminal and the output file (appropriate buffer of 2)

Easy to implement this as a subroutine so both actions are completed in one call

* 1. return to caller

1. **printOpen:**
   1. prints an open parenthesis “( “ to the terminal and the output file (appropriate buffer 1)

Easy to implement this as a subroutine so both actions are completed in one call

* 1. return to caller

1. **printClose:**
   1. prints a close parenthesis “ )” to the terminal and the output file (appropriate buffer 2)

Easy to implement this as a subroutine so both actions are completed in one call

* 1. return to caller

### Song Note Details

1. **play\_exit\_song:** plays a series of 5 ascending notes using syscall 33 and the appropriate registers when the program is over
2. **play\_error\_song:** plays a series of 3 descending notes using syscall 33 and the appropriate registers when there is an error in opening the file

# Supporting File: “numberRegister.asm”

## .data

## Includes all the numbers ranging backwards from 9 to 0 (10 unique digits in total)

## .macro doNums

Int to ASCII Conversion

1. By getting the modulo (remainder after doing simple division), the integers are converted to its ascii equivalent, which can be printed to an output file

# Supporting File “RFormat.asm”

## .data

Includes the instruction names for most R type instructions

## .macro doR

1. Get the function code
   1. Key factor that distinguishes R format instructions
2. Search for special R instruction jr, srl/sll, syscall
   1. These do not display the regular patterns of the R format
3. Jr instruction: special because it only takes in the rs component
   1. Print jr
   2. Print the rs register
   3. Print out necessary formatting elements ($) where needed
   4. Return to the reading loop
4. Shift instructions: special because they include the shamt value and not an rs value
   1. Print srl or sll depending on the funct code
   2. Print rd
   3. Print rt
   4. Print shamt
   5. Print out necessary formatting elements ($ or ,) where needed
   6. Return to the reading loop
5. Syscall instruction: special because it only print syscall and no other value
   1. Print syscall message
   2. Return to the reading loop
6. Regular case
   1. Print the instruction by the “resolve\_funct subroutine” (listed later)
   2. Print rd
   3. Print rs
   4. Print rt
   5. Print out necessary formatting elements ($ or ,) where needed
   6. Return to the reading loop

## Resolving the R Format Instructions with the Funct code

1. **Macro functToInstruction**: takes the value of the funct code to match the corresponding R instruction
   1. Prepares the write to a file before the series of branch ($v0 and $a0 which are common among all the writes to the output file)
   2. Go through a series of branch statements to find the corresponding funct number
      1. Upon a match, branch to the corresponding instruction label
      2. Move the address of instruction name into $t9 for use in the original caller
      3. Print the instruction name to the output file (complete the call with loading $a1 with the address name and $a2 with the hard coded buffer)
   3. Return to the caller

# Supporting File: “JFormat.asm”

## .data

Includes the instruction names for J type instructions (j and jal)

## .macro doJ

1. Print j or jal depending on the opcode
2. Retrieve the address from the instruction
3. Add that to an address of 0x00400000
4. Print the address as a hexadecimal with a syscall (34)
5. Reser critical registers to zero (must be 0 for other algorithms to work properly)
6. Print out necessary formatting elements ($) where needed
7. Return to the reading loop

# Supporting File: “IFormat.asm”

## .data

Includes the instruction names for most I type instructions

## .macro dataITType

For data transfer

1. Print sw or lw depending on the opcode
2. Prints rt
3. Prints the immediate
4. Prints rs
5. Reset critical registers to zero
6. Print out necessary formatting elements ($, (), or ,) where needed
7. Returns to the reading loop

## .macro branchIType

1. Print the instruction depending on the opcode
2. Print rt
3. Print immediate value
4. Reset critical registers to zero
5. Print out necessary formatting elements ($ or ,) where needed
6. Returns to the reading loop

## .macro logicIType

1. Print the instructio depending on the opcode
2. Print rt
3. Print rs
   1. Exception: if the opcode read lui, the rs is branched over
4. Print immediate value
5. Reset critical registers to zero
6. Print out necessary formatting elements ($ or ,) where needed
7. Returns to the reading loop

.macro opToInstruction

1. **resolve\_opcode**: takes the value of the opcode code to match the corresponding I instruction
   1. Prepares the write to a file before the series of branch ($v0 and $a0 which are common among all the writes to the output file)
   2. Go through a series of branch statements to find the corresponding funct number
      1. Upon a match, branch to the corresponding instruction label
      2. Move the address of instruction name into $t9 for use in the original caller
      3. Print the instruction name to the output file (complete the call with loading $a1 with the address name and $a2 with the hard coded buffer)
   3. Return to the caller

# Supporting File: “extractInstruction.asm”

## .macro rsExtract

Retrieve rs value

1. Read in first two bytes and convert to hexadecimal value
2. Join the two into one 8-bit number
3. Add the with 0011 1110 to mask the values that don’t make up the rs value
4. Shift right by 1 for the rs code
5. Return to caller

## .macro rtExtract

Retrieve rt value

1. Read in second and third and convert to hexadecimal value
2. Join the two into one 8-bit number
3. Add the with 0001 1111 to mask the values that don’t make up the rt value
4. Return to caller

## .macro rdExtract

Retrieve rd value

1. Read in fourth and fifth byte and convert to hexadecimal value
2. Join the two into one 8-bit number
3. Add the with 1111 1000 to mask the values that don’t make up the rd value
4. Shift right by 3 for the rs code
5. Return to caller

## .macro shamtExtract

Retrieve shamt value

1. Read in fifth and sixth byte and convert to hexadecimal value
2. Join the two into one 8-bit number
3. Add the with 0111 1100 to mask the values that don’t make up the shamt value
4. Shift right by 2 for the shamt code
5. Return to caller

## .macro functExtract

Retrieve funct value

1. Read in sixth and seventh byte and convert to hexadecimal value
2. Join the two into one 8-bit number
3. Add the with 0011 1111 to mask the values that don’t make up the funct value
4. Return to caller

## .macro immExtract

Retrieve immediate value

1. Have one main variable that holds the immediate value
2. Go through a loop that reads the last four bytes of the instruction
   1. End the loop when you reach the 8th byte
   2. Convert the ACII value to the equivalent hex value
   3. Extend the immediate variable by 4 bits
   4. Append the byte to the immediate variable
   5. Increment loop variables

## .macro addrExtract

Retrieve jump address

1. Have one main variable that holds the address value
2. Extract the last two bits of the 2nd byte and append onto the main address variable
3. Go through a loop that reads the last six bytes of the instruction
   1. End the loop when you reach the 8th byte
   2. Convert the ACII value to the equivalent hex value
   3. Extend the address variable by 4 bits
   4. Append the byte to the address variable
   5. Increment loop variables

# Running the Program

1. Open the “MAIN\_DRIVER.asm” file in same folder as the MARS application and other supporting files
2. Assemble the Program
3. Run the Program
4. Play with the input by entering an incorrect file name
5. You will get an error check that will play music
6. Repeats the prompt for you to re-enter a new file name
7. Enter in “input.txt” when asked for file name
8. Outputs are written both on terminal and “output.asm”
   1. Music is played and there will be a pop-up message