Group - Jacob Tabalon, Maggi Savajiyani

Introduction

Our custom assembly language, called microMIPS, is an extension of MIPS that adds fun features and makes certain things easier. Deviating from MIPS, we decided to change the registers, changing registers from $t and $s to all be $r, making it easier to keep track of which registers are being used and making it more clear. Our focus was to mess around with binary a little bit using our special functions, giving our assembly language its own little flair. We have a couple of instructions that make certain tasks easier, for example printing integers, and also some instructions relating to time, but the brunt of our functions help manipulate numbers and perform operations on binary numbers. The main goal was to simplify MIPS in a way that makes it easier to use in some situations.

Basic Instructions:

Refer to MIPS Green Sheet

- MIPS Green Sheet [PDF]

Main thing changed from the green sheet is our registers

Group - Jacob Tabalon, Maggi Savajiyani

MIPS Reference Data ①

CORE INSTRUCTION SET

NAME	MNEMONIC	FORMAT	OPERATION (in Verilog)		OPCODE/FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	$0 / 20_{hex}$
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1)(2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		$0 / 21_{hex}$
And	and	R	R[rd] = R[rs] & R[rt]		$0 / 24_{hex}$
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr*4	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr*4	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+4;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		$0 / 08_{hex}$
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		$0 / 27_{hex}$
Or	or	R	R[rd] = R[rs] | R[rt]		$0 / 25_{hex}$
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		$0 / 2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2)(6)	b_{hex}
Set Less Than Unsigned	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	$0 / 2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		$0 / 00_{hex}$
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		$0 / 02_{hex}$
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	$0 / 22_{hex}$
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		$0 / 23_{hex}$

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Decimal
(1)	sll	add.*f*	00 0000	0
		sub.*f*	00 0001	1
j	srl	mul.*f*	00 0010	2
jal	sra	div.*f*	00 0011	3
beq	sllv	sqrt.*f*	00 0100	4
bne		abs.*f*	00 0101	5
blez	srlv	mov.*f*	00 0110	6
bgtz	srav	neg.*f*	00 0111	7
addi	jr		00 1000	8
addiu	jalr		00 1001	9
slti	movz		00 1010	10
sltiu	movn		00 1011	11
andi	syscall	round.w.*f*	00 1100	12
ori	break	trunc.w.*f*	00 1101	13
xori		ceil.w.*f*	00 1110	14
lui	sync	floor.w.*f*	00 1111	15
	mfhi		01 0000	16
(2)	mthi		01 0001	17
	mflo	movz.*f*	01 0010	18
	mtlo	movn.*f*	01 0011	19
emptyregs		ranmult	01 0100	20
flip		popcount	01 0101	21
sleep		rotr	01 0110	22
rand		while	01 0111	23
	mult		01 1000	24
	multu		01 1001	25
	div		01 1010	26
	divu		01 1011	27
push			01 1100	28
pop			01 1101	29
mod			01 1110	30
stimer			01 1111	31
lb	add	cvt.s.*f*	10 0000	32
lh	addu	cvt.d.*f*	10 0001	33
lwl	sub		10 0010	34
lw	subu		10 0011	35
lbu	and	cvt.w.*f*	10 0100	36
lhu	or		10 0101	37
lwr	xor		10 0110	38
	nor		10 0111	39
sb			10 1000	40
sh			10 1001	41
swl	slt		10 1010	42
sw	sltu		10 1011	43
			10 1100	44
			10 1101	45
swr			10 1110	46
cache			10 1111	47
ll	tge	c.f.*f*	11 0000	48
lwc1	tgeu	c.un.*f*	11 0001	49
lwc2	tlt	c.eq.*f*	11 0010	50
pref	tltu	c.ueq.*f*	11 0011	51
	teq	c.olt.*f*	11 0100	52
ldc1		c.ult.*f*	11 0101	53
ldc2	tne	c.ole.*f*	11 0110	54
		c.ule.*f*	11 0111	55
sc		c.sf.*f*	11 1000	56
swc1		c.ngle.*f*	11 1001	57
swc2		c.seq.*f*	11 1010	58
		c.ngl.*f*	11 1011	59
		c.lt.*f*	11 1100	60
sdc1		c.nge.*f*	11 1101	61
sdc2		c.le.*f*	11 1110	62
		c.ngt.*f*	11 1111	63

Group - Jacob Tabalon, Maggi Savajiyani

Unique Instructions

Name	Syntax	Binary Representation	Binary Representation (op/funct code)	Functionality
emptyregs	emptyregs	000000 | 0000 | 0000 | 0000 | 00000 | 000010100	20	Clears all general purpose registers ($r1-$r15) setting them to zero
flip (Bit Flip)	flip $rd, $rs	000000 | rs[4] | 0000 | rd[4] | 00000 | 000010101	21	Flips all bits in $rs and stores in $rd
sleep (Wait for N cycles)	sleep $rs	000000 | rs[4] | 0000 | 0000 | 00000 | 000010110	22	Pauses execution for $rs cycles
rand (Generate Random Number)	rand $rd, $rs	000000 | rs[4] | 0000 | rd[4] | 00000 | 000010111	23	Generates random number between 0 and $rs, stores in $rd
ranmult (Random Mult)	ranmult $rd	010100 | 0000 | 0000 | rd[4] | 00000 | 000000000	21	Multiplies the number in $rd by a random number 1-9, stores the multiplier in $rnd

popcount (Count Set Bits)	popcount $rd, $rs	010101 | rs[4] | 0000 | rd[4] | 00000 | 000000000	22	Counts number of '1' bits in $rs, stores in $rd
rotr (Rotate Right)	rotr $rd, $rs, $rt	010110 | rs[4] | rt[4] | rd[4] | 00000 | 000000000	23	Rotates bits in $rs right by $rt positions, stores in $rd
printint	printint $rs	010111 | rs[4] | 0000 | 0000 | 00000 | 000000000	24	Prints the number in $rs
push (Push to Stack)	push $rs	000000 | rs[4] | 0000 | 0000 | 00000 | 000011100	28	$sp = $sp - 4; Memory[$sp] = $rs
pop (Pop from Stack)	pop $rd	000000 | 0000 | 0000 | rd[4] | 00000 | 000011101	29	$rd = Memory[$sp]; $sp = $sp + 4
mod (Modulo)	mod $rs, $rt	000000 | rs[4] | rt[4] | rem[4] | 00000 | 000011110	30	Divides $rs by $rt and stores the remainder in $rem
stimer (Start/stop timer)	stimer	000000 | rs[4] | rt[4] | 0000 | 00000 | 000011111	31	Starts (or stops) the timer ($time) register from ticking

Group - Jacob Tabalon, Maggi Savajiyani

Registers

Name	Number	Use
$r0	0	Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for function results & expression evaluation
$a0-$a3	4-7	Arguments
$r1-$r15	8-23	Temporaries (Registers)
$time	24	Timer Register
$rem	25	Remainder (From Modulo)
$rnd	26	Stores Random Number
$status	27	Status Register
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Sample Programs

Program 1: Fun with bits

C:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main() {
    srand(time(NULL));
    int num = 100;
```

```
    int multiplier = rand() % 9 + 1;
    int result = num * multiplier;
    result = ~result;
    result >>= 3;
    printf("%d\n", result);
    result = 0
    printf(“%d\n”, result);
    return 0;

}
```

microMIPS:

// This short program takes 100, multiplies it by a random amount flips the bits, shifts the bits 3 to the right, counts the number of bits that are 1, prints it out, empties the registers, and prints the register again (now empty)

```
li $r1, 100
ranmult $r1
flip $r1, $r1
li $r2, 3
rotr $r1, $r1, $r2
popcount $r3, $r1
printint $r1
printint $r3
emptyregs
printint $r1
```

Machine Code:

```
00100000000010010000000001100100010100000000000001001000000000000000
01001000000100100000010101001000000000101000000000000000011010110010010
10100100100000000000010101010010000001011000000000000101111010010000000
00000000000000000000000000000000000000010100010111010010000000000000
00000000
```

Program 2: Stack + Modulo

Group - Jacob Tabalon, Maggi Savajiyani

This program starts a timer, generates a number, 1-10 and then takes that number modulo 10. If the remainder is not 0 it will add the result to the stack and then add 3 to the result, also pushing that to the stack. Regardless of if the remainder was 0 or not, it will wait for 0.1 seconds. Then, if there is something to pop out of the stack it will pop it out and print the value. Before stopping the timer and printing out the amount of time it took.

C:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>  // For sleep functions
#include <sys/time.h>  // For timing

#define STACK_SIZE 100

// Stack structure
typedef struct {
    int data[STACK_SIZE];
    int top;
} Stack;

// Initialize stack
void initStack(Stack *s) {
    s->top = -1;
}

// Check if stack is full
int isFull(Stack *s) {
    return s->top == STACK_SIZE - 1;
```

```
}

// Check if stack is empty
int isEmpty(Stack *s) {
    return s->top == -1;
}

// Push element onto stack
void push(Stack *s, int value) {
    if (isFull(s)) {
        return;
    }
    s->data[++(s->top)] = value;
    printf("Pushed %d onto the stack\n", value);
}

// Pop element from stack
int pop(Stack *s) {
    if (isEmpty(s)) {
        return -1;
    }
    return s->data[(s->top)--];
}

int main() {
    Stack stack;
    initStack(&stack);

    // Variables for timing
    struct timeval start_time, end_time;
    double elapsed_time;
```

```
    // Start the timer
    gettimeofday(&start_time, NULL);

    srand(time(NULL));

    // Generate random number between 1 and 10
    int random_num = rand() % 10 + 1;
    printf("Generated random number: %d\n", random_num);

    // Calculate modulo
    int result = random_num % 10;
    printf("Result of %d %% 10 = %d\n", random_num, result);
    int items_in_stack = 0;

    if (result != 0) {

        // Push result onto stack
        push(&stack, result);

        items_in_stack++;
        result += 3;
        result = result % 10;

        push(&stack, result);

    }

    // Sleep for 100 milliseconds/cycles
    usleep(100000);
```

```
    if (items_in_stack > 0) {
        // Demonstrate popping the value
        printf("Popped value from stack: %d\n", pop(&stack));
        items_in_stack--;
    }

    // Stop the timer
    gettimeofday(&end_time, NULL);
    elapsed_time = (end_time.tv_sec - start_time.tv_sec) +
                   (end_time.tv_usec - start_time.tv_usec) / 1000000.0;

    printf("Total execution time: %.6f seconds\n", elapsed_time);

    return 0;
}
```

microMIPS:

```
stimer
li $r2, 10
li $r4, 0
li $r8, 1
rand $r1, $r2
mod $r1, $r2 # modulo of our random number and 10
beq $rem, $r0, 6 # skips past 5 instructions if remainder is 0
push $rem
addi $r4, $r4, 1 # $r4 is tracking amount of items in stack
addi $rem, $rem, 3
mod $rem, $r2
push $rem
addi $r4, $r4, 1
sleep $r2 # sleeps for 100 cycles
```

beq $r0, $r4, 3 # if there are no items in stack skip 3 instructions

pop $r5

sub $r4, $r4, $r8

printint $r5 # printing value on top of stack

emptyregs

stimer

printint $time

Machine Code:

0000000000000000000000000000000010000000000101000000000000001010001000
0000001100000000000000000000010000000010000000000000000000100000010100
000001001000000101110000000100101010000000000000111100001001100100000000
0000000000010100000011001000000000000000000111000010000110001100000000000
00000001000000110010101000000000000001111000000011001000000000000000001110
0001000011000110000000000000000000100000001010000000000000000001011000010
00000001100000000000000000110000000000000000001101000000111010000000001100
1000001100000001000100101110110100000000000000000000000000000000000000
000000000010100000000000000000000000000000000000101111100000000000000000
000000000

Program 3: FizzBuzz

C:

```
#include <stdio.h>
int main() {
    int fizz;
    fizz = 3;
    int buzz;
    buzz = 5;
    int stop;
    stop = 100;
```

```
        int count;
        count = 1;
        int mod;
        mod = 0;
        int mod1;
        mod1 = 0;
        int zero;
        zero = 0;
        while (count <= stop) {
            mod = count % fizz;
            mod1 = count % buzz;
            if (mod == zero) {
                printf("Fizz\n");
                if (mod1 == zero) {
                    printf("FizzBuzz\n");
                }
            }
            if (mod1 == zero) {
                printf("Buzz\n");
            }
            if (mod != zero) {
                if (mod1 != zero) {
                    printf("%d\n", count)
                }
            }
            count = count + 1
        }
        return 0;
}
```

microMIPS:

```
.data
str1:    .asciiz    "Fizz\n"
str2:    .asciiz    "FizzBuzz\n"
str3:    .asciiz    "Buzz\n"
.text
addi $r0, $zero, 5000
addi $r1, $zero, 3
sw $r1, 0($r0)
addi $r2, $zero, 5004
```

Group - Jacob Tabalon, Maggi Savajiyani

```
addi $r3, $zero, 5
sw $r3, 0($r2)
addi $r4, $zero, 5008
addi $r5, $zero, 100
sw $r5, 0($r4)
addi $r6, $zero, 5012
addi $r7, $zero, 1
sw $r7, 0($r6)
addi $r8, $zero, 5016
addi $r9, $zero, 0
sw $r9, 0($r8)
addi $r10, $zero, 5020
addi $r11, $zero, 0
sw $r11, 0($r10)
addi $r12, $zero, 5024
addi $r13, $zero, 0
sw $r13, 0($r12)
WHILE1:
mod $r8, $r6, $r0
mod $r10, $r6, $r2
bne $r8, $r12, AFTER1
sw $a0, $str1
li $v0, 4
syscall
bne $r10, $r12, AFTER2
sw $a0, $str2
li $v0, 4
syscall
AFTER1:
AFTER2:
bne $r10, $r12, AFTER3
sw $a0, $str3
li $v0, 4
```

```
syscall
AFTER3:
beq $r8, $r12, AFTER4
beq $r10, $r12, AFTER5
sw $a0, $r6
li $v0, 10
syscall
AFTER4:
AFTER5:
addi $r6, $r6, 1
addi $r14, $zero, 0
sw $r14, 0($r12)
bgt $r6, $r4, WHILE1
```

Machine code:

00100000000000000010011100010000010000100100000000000000000
00111010110000001001000000000000000010000101000000000100111
00011000010000101100000000000000000010111010110101001011000000
00000000000000100001100000000001001110010000001000011010000000
00000000110010010101101100011010000000000000000001000011100000
00000010011100101000010000111100000000000000000001101011011100
11111000000000000000010001000000000001001110011000001000100
00100000000000000000000010101110000100010000000000000000010
00100100000000010011100111000010001001100000000000000000000
00101011100101001100000000000000000000010000011100000000000
01111000000010000011100000000011110

Compiler/Assembler/Disassembler also attached separately as their own pdfs.

Compiler:

Group - Jacob Tabalon, Maggi Savajiyani

```
memoryAddress = 5000
tRegister = 0
vars = dict()
statementNumber = 0
forStatementNumber = 0
ifStatements = 0
forStatements = 0
data = ""
datanum = 0

def getInstructionLine(varName):
    global memoryAddress, tRegister
    tRegisterName = f"$r{tRegister}"
    setVariableRegister(varName, tRegisterName)
    returnText = f"addi {tRegisterName}, $zero, {memoryAddress}"
    tRegister += 1
    memoryAddress += 4
    return returnText

def setVariableRegister(varName, tRegister):
    global vars
    vars[varName] = tRegister

def getVariableRegister(varName):
    global vars
    if varName in vars:
        return vars[varName]
    else:
        return "ERROR"

def getAssignmentLinesImmediateValue(val, varName):
```

```
    global tRegister
    outputText = f"""addi $r{tRegister}, $zero, {val}\nsw $r{tRegister},
0({getVariableRegister(varName)})"""
    tRegister += 1
    return outputText


def getAssignmentLinesVariable(varSource, varDest):
    global tRegister
    outputText = ""
    registerSource = getVariableRegister(varSource)
    outputText += f"lw $r{tRegister}, 0({registerSource})" + "\n"
    tRegister += 1
    registerDest = getVariableRegister(varDest)
    outputText += f"sw $r{tRegister-1}, 0({registerDest})"
    # tRegister += 1
    return outputText


def getIfStatement(expr):
    global tRegister, vars, statementNumber, ifStatements
    outputText = ""
    ifStatements += 1
    statementNumber += 1
    var1, expr, var2 = expr.split()
    varRegister1 = getVariableRegister(var1)
    varRegister2 = getVariableRegister(var2)
    if "==" in expr:
        outputText += f"bne {varRegister1}, {varRegister2},
AFTER{statementNumber}"
    elif ">=" in expr:
        outputText += f"blt {varRegister1}, {varRegister2},
AFTER{statementNumber}"
```

```
    elif "<=" in expr:
        outputText += f"bgt {varRegister1}, {varRegister2},
AFTER{statementNumber}"
    elif ">" in expr:
        outputText += f"ble {varRegister1}, {varRegister2},
AFTER{statementNumber}"
    elif "<" in expr:
        outputText += f"bge {varRegister1}, {varRegister2},
AFTER{statementNumber}"
    return outputText



def getAssignment(expr):
    outputText = ""
    var, _, var1, operation, var2 = line.split()
    var2 = var2.replace(";", "")
    if not var1.isdigit():
        if operation == "+":
            outputText += f"addi {getVariableRegister(var)},
{getVariableRegister(var1)}, {var2}"
            return outputText
    return ""


def whileLoop(expr):
    global forStatements, forStatementNumber, vars
    statement = ""
    forStatements = forStatements + 1
    forStatementNumber += 1
    var1, expr, var2 = expr.split()
    varRegister1 = getVariableRegister(var1)
    varRegister2 = getVariableRegister(var2)
    if "==" in expr:
```

```
        statement += f"bne {varRegister1}, {varRegister2},
WHILE{forStatementNumber}"
    elif ">=" in expr:
        statement += f"blt {varRegister1}, {varRegister2},
WHILE{forStatementNumber}"
    elif "<=" in expr:
        statement += f"bgt {varRegister1}, {varRegister2},
WHILE{forStatementNumber}"
    elif ">" in expr:
        statement += f"ble {varRegister1}, {varRegister2},
WHILE{forStatementNumber}"
    elif "<" in expr:
        statement += f"bge {varRegister1}, {varRegister2},
WHILE{forStatementNumber}"
    vars[forStatements] = statement
    return f"WHILE{forStatementNumber}:"

def modulo(expr):
    outputText = ""
    expr = expr.strip()
    expr = expr.replace(";", "")
    var1, equal, var2, percent, var3 = expr.split(" ")
    varRegister1 = getVariableRegister(var1)
    varRegister2 = getVariableRegister(var2)
    varRegister3 = getVariableRegister(var3)
    outputText += f"mod {varRegister1}, {varRegister2}, {varRegister3}"
    return(outputText)

def printf(expr):
    global data, vars, datanum
    outputText = ""
    datanum += 1
```

```
    if "%d" in expr:
        expr.strip()
        expr =
expr.replace("(","").replace(")","").replace("{","").replace("\n", "")
        _, var1 = expr.split(", ")
        var1 = var1.replace(" ", "")
        varRegister1 = getVariableRegister(var1)
        outputText += f"sw $a0, {varRegister1}" + "\n"
        outputText += f"li $v0, 10" + "\n"
        outputText += "syscall"
        return outputText
    else:
        expr.strip()
        expr =
expr.replace("(","").replace(")","").replace("{","").replace("\n", "")
        _, expr, _ = expr.split('"')
        outputText += f"sw $a0, $str{datanum}" + "\n"
        outputText += f"li $v0, 4" + "\n"
        outputText += "syscall"
        data += f'str{datanum}:     .asciiz     "{expr}"'
        data += "\n"
        return outputText


f = open("program7.c", "r")
lines = f.readlines()
outputText = ""
data = ""
for line in lines:
    # if line.startswith("if "):
    if "if" in line:
        _, expr = line.split("if ")
```

```
        expr = expr.replace("(","").replace(")","").replace("{","")
        outputText += getIfStatement(expr) + "\n"
    elif "else if" in line:
    # elif line.startswith("else if "):
        _, expr = line.split("if ")
        expr = expr.replace("(","").replace(")","").replace("{","")
        outputText += getIfStatement(expr) + "\n"
    # while loop
    elif line.startswith("while"):
        _, expr = line.split("while ")
        expr = expr.replace("(","").replace(")","").replace("{","")
        outputText += whileLoop(expr) + "\n"
    # end for/if
    # elif line.startswith("}"):
    elif "}" in line:
        if ifStatements > 0:
            outputText += f"AFTER{statementNumber}:" + "\n"
            ifStatements -= 1
        elif forStatements > 0:
            outputText += vars[forStatements] + "\n"
    # int declarations
    elif line.startswith("int "):
        _, var = line.split()
        var = var.strip(";")
        outputText += getInstructionLine(var) + "\n"
    # assignments
    elif "printf" in line:
        outputText += printf(line) + "\n"
    elif "%" in line:
        outputText += modulo(line) + "\n"
    elif "=" in line:
```

```
        if len(line.split()) == 3:
            varName, _, val = line.split()
        else:
            outputText += getAssignment(line) + "\n";
        val = val.strip(";")
        if val.isdigit():
            # immediately value assignments
            outputText += getAssignmentLinesImmediateValue(val, varName) +
"\n"
        else:
            # variable assignments
            outputText += getAssignmentLinesVariable(val, varName) + "\n"
    else:
        pass
outputText = ".data" + "\n" + data + ".text" + "\n" + outputText
outputFile = open("output7.asm", "w")
outputFile.write(outputText)
```

Assembler:

```
import sys
import os



op_codes = {
    "add": "000000",
    "sub": "000000",
    "and:": "000000",
    "or:": "000000",
    "slt": "000000",
    "lw": "100011",
```

```
    "sw": "101011",
    "beq": "000100",
    "bne": "000100",
    "emptyregs" : "000000",
    "flip" : "000000",
    "sleep" : "000000",
    "rand" : "000000",
    "push" : "000000",
    "pop" : "000000",
    "mod" : "000000",
    "stimer" : "000000",
    "addi" : "001000",
    "li" : "001000",
    "ranmult" : "010100",
    "popcount" : "010101",
    "rotr" : "010110",
    "printint" : "010111",
}
func_codes = {
    "add": "100000",
    "sub": "100010",
    "and:": "100100",
    "or:": "100101",
    "slt": "101010",
    "emptyregs" : "010100",
    "flip" : "010101",
    "sleep" : "010110",
    "rand" : "010111",
    "push" : "011100",
    "pop" : "011101",
    "mod" : "011110",
```

```
    "stimer" : "000000",
    "addi" : "000000",
    "li" : "001000",
    "ranmult" : "000000",
    "popcount" : "000000",
    "rotr" : "000000",
    "printint" : "000000",


}
registers = {
    "$r0": "00000",
    "$r1": "01001",
    "$r2": "01010",
    "$r3": "01011",
    "$r4": "01100",
    "$r5": "01101",
    "$r6": "01110",
    "$r7": "01111",
    "$r8": "10000",
    "$r9": "10001",
    "$r10": "10010",
    "$r11": "10011",
    "$r12": "10100",
    "$r13": "10101",
    "$r14": "10110",
    "$r15": "10111",
    "$time": "11000",
    "$rem": "11001",
    "$rnd": "11010",
    "$status": "11011",
    "$gp": "11100",
```

```
    "$sp": "11101",
    "$fp": "11110",
    "$ra": "11111",
    "$v0": "00010"

}

labels = dict()

shift_logic_amount = "00000"

line_address = 0



def interpret_line(mips_file: str):
    global line_address
    input_file = open(mips_file, "r")
    output_file = open("program2.bin", "w")
    for instruction in input_file:
        bin = assemble(instruction)
        line_address += 4
        output_file.write(bin)



def assemble(line):
    line = line.split("#")[0].strip()

    if not line:
        return

    parts = line.split(" ")
```

```
    op_code = parts[0]

    if op_code in ["emptyregs", "stimer"]:
        return (
            "00000000000000000000000000" + func_codes[op_code]
        )

    elif op_code in ["push", "sleep", "printint"]:

        rs = parts[1]

        return (
            op_codes[op_code]
            + registers[rs]
            + "00000"
            + "00000"
            + "00000"
            + func_codes[op_code]
        )

    elif op_code in ["flip", "rand", "popcount"]:
        rd, rs = (
            parts[1].replace(",", ""),
            parts[2].replace(",", ""),
        )

        return (
            op_codes[op_code]
            + registers[rs]
            + "00000"
            + registers[rd]
```

```
            + shift_logic_amount
            + func_codes[op_code]
        )

    elif op_code in ["ranmult", "pop"]:
        rd = parts[1]
        return (
            op_codes[op_code]
            + "00000"
            + "00000"
            + registers[rd]
            + "00000"
            + func_codes[op_code]
        )

    elif op_code in ["mod", "pop"]:
        rs, rt = (
            parts[1].replace(",", ""),
            parts[2].replace(",", ""),
        )
        return (
            op_codes[op_code]
            + registers[rs]
            + registers[rt]
            + "00000"
            + "00000"
            + func_codes[op_code]
        )

    elif op_code in ["addi"]:
        rs, rt, immediate = (
```

```
            parts[1].replace(",", ""),
            parts[2].replace(",", ""),
            parts[3].replace(",", ""),
        )

        return (
            op_codes[op_code]
            + registers[rs]
            + registers[rt]
            + bin(int(immediate)).replace("0b", "").zfill(16)
        )
    elif op_code in ["li"]:
        rs, immediate = (
            parts[1].replace(",", ""),
            parts[2].replace(",", ""),
        )

        return (
            op_codes[op_code]
            + "00000"
            + registers[rs]
            + bin(int(immediate)).replace("0b", "").zfill(16)
        )

    elif op_code in func_codes:
        rd, rs, rt = (
            parts[1].replace(",", ""),
            parts[2].replace(",", ""),
            parts[3].replace(",", ""),
        )
        return (
```

```
            op_codes[op_code]
            + registers[rs]
            + registers[rt]
            + registers[rd]
            + shift_logic_amount
            + func_codes[op_code]
        )

    if op_code in ["lw", "sw", "beq"]:
        if op_code == "lw" or op_code == "sw":
            rt = parts[1].replace(",", "")
            offset, rs = parts[2].replace(")", "").split("(")
            offset_bin = bin(int(offset)).replace("0b", "").zfill(16)
            return op_codes[op_code] + registers[rs] + registers[rt] +
offset_bin
        else:
            rs, rt, offset = (
                parts[1].replace(",", ""),
                parts[2].replace(",", ""),
                parts[3].replace(",", ""),
            )
            offset_bin = bin(int(offset)).replace("0b", "").zfill(16)
            return op_codes[op_code] + registers[rs] + registers[rt] +
offset_bin

    # handles labels
    op_code = op_code.replace(":", "")



if __name__ == "__main__":
```

```
    # mips_file = sys.argv[1]
    mips_file = "test.mips"
    interpret_line(mips_file)
```

Disassembler:

```
import sys


op_codes = {
    "000000": "add",
    "000000": "sub",
    "000000": "and",
    "000000": "or",
    "000000": "slt",
    "100011": "lw",
    "101011": "sw",
    "000100": "beq",
    "000101": "bne",
    "000000": "emptyregs",
    "000000": "flip",
    "000000": "sleep",
    "000000": "rand",
    "000000": "push",
    "000000": "pop",
    "000000": "mod",
    "000000": "stimer",
    "010100": "ranmult",
    "010101": "popcount",
    "010110": "rotr",
    "001000" : "addi",
    "010111" : "printint",
}
func_codes = {
    "100000": "add",
    "100010": "sub",
```

```
    "100100": "and",
    "100101": "or",
    "101010": "slt",
    "000000": "ranmult",
    "000000": "popcount",
    "000000": "rotr",
    "010100": "emptyregs",
    "010101": "flip",
    "010110": "sleep",
    "010111": "rand",
    "011100": "push",
    "011101": "pop",
    "011110": "mod",
    "011111": "stimer",
    "000000" : "addi",
    "000000" : "printint",


}
registers = {
    "00000": "$r0",
    "01001": "$r1",
    "01010": "$r2",
    "01011": "$r3",
    "01100": "$r4",
    "01101": "$r5",
    "01110": "$r6",
    "01111": "$r7",
    "10000": "$r8",
    "10001": "$r9",
    "10010": "$r10",
    "10011": "$r11",
    "10100": "$r12",
    "10101": "$r13",
    "10110": "$r14",
    "10111": "$r15",
    "00010" : "$v0",
    "11001": "$rem",
    "11010": "$rnd",
    "11011": "$status",
```

```
    "11100": "$gp",
    "11101": "$sp",
    "11110": "$fp",
    "11111": "$ra",
    "11000": "$time",
}

labels = dict()

def handle_lines(bin_file: str):
    input_file = open(bin_file, "r")
    line = input_file.readlines()[0].strip()
    mips_instructions = bin_to_mips(line)
    output_file = open("BACK_TO_MIPS.txt", "w")
    for instruction in mips_instructions:
        output_file.write(instruction)
        output_file.write("\n")


def bin_to_mips(line):
    global labels
    mips = []
    bit_string = ""
    for i in range(0, len(line)):
        bit_string += line[i]
        if len(bit_string) == 32:
            op_code = bit_string[0:6]
            print(op_code)

            if op_code == "000000":
                rs, rt, rd, shift, func_code = (
                    bit_string[6:11],
                    bit_string[11:16],
                    bit_string[16:21],
                    bit_string[21:26],
                    bit_string[26:32],
                )
                if func_codes[func_code] == "pop":
                    mips.append(
                        f"{func_codes[func_code]} {registers[rd]}"
```

```
                    )
                elif func_codes[func_code] == "push":
                    mips.append(
                        f"{func_codes[func_code]} {registers[rs]}"
                    )
                elif func_codes[func_code] == "mod":
                    mips.append(
                        f"{func_codes[func_code]} {registers[rs]},
{registers[rt]}"
                    )
                elif func_codes[func_code] == "stimer":
                    mips.append(
                        f"{func_codes[func_code]}"
                    )
                elif func_codes[func_code] == "emptyregs":
                    mips.append(
                        f"{func_codes[func_code]}"
                    )
                elif func_codes[func_code] == "flip":
                    mips.append(
                        f"{func_codes[func_code]} {registers[rd]},
{registers[rs]}"
                    )
                elif func_codes[func_code] == "rand":
                    mips.append(
                        f"{func_codes[func_code]} {registers[rd]},
{registers[rs]}"
                    )
                elif func_codes[func_code] == "sleep":
                    mips.append(
                        f"{func_codes[func_code]} {registers[rs]}"
                    )
                else:
                    mips.append(
                        f"{func_codes[func_code]} {registers[rd]},
{registers[rs]}, {registers[rt]}"
                    )
            elif op_codes[op_code] == "addi":
                rs, rd, immediate = (
                    bit_string[6:11],
```

```
                    bit_string[11:16],
                    bit_string[17:32]
                )
                mips.append(
                    f"{op_codes[op_code]} {registers[rd]},
{registers[rs]}, {int(immediate, 2)}"
                )
            elif op_codes[op_code] == "beq":
                print("test")
                rs, rt, offset = bit_string[6:11], bit_string[11:16],
bit_string[16:32]
                mips.append(
                    f"{op_codes[op_code]} {registers[rs]},
{registers[rt]}, {int(offset, 2)}"
                )

            elif op_codes[op_code] == "printint":
                rs, rt, rd, shift, func_code = (
                    bit_string[6:11],
                    bit_string[11:16],
                    bit_string[16:21],
                    bit_string[21:26],
                    bit_string[26:32],
                )
                mips.append(
                    f"{func_codes[func_code]} {registers[rs]}"
                )

            elif op_code not in ["100011", "101011"]:
                rs, rt, rd, shift, func_code = (
                    bit_string[6:11],
                    bit_string[11:16],
                    bit_string[16:21],
                    bit_string[21:26],
                    bit_string[26:32],
                )
                if op_codes[op_code] == "ranmult":
                    mips.append(
                        f"{op_codes[op_code]} {registers[rd]}"
                    )
```

Group - Jacob Tabalon, Maggi Savajiyani

```
                elif op_codes[op_code] == "popcount":
                    mips.append(
                        f"{op_codes[op_code]} {registers[rd]},
{registers[rs]}"
                    )
                elif op_codes[op_code] == "rotr":
                    mips.append(
                        f"{op_codes[op_code]} {registers[rd]},
{registers[rs]}, {registers[rt]}"
                    )

            elif op_code in ["100011", "101011"]:
                rs, rt, offset = bit_string[6:11], bit_string[11:16],
bit_string[16:32]
                mips.append(
                    f"{op_codes[op_code]} {registers[rt]}, {int(offset,
2)}({registers[rs]})"
                )



            bit_string = ""
    return mips


if __name__ == "__main__":
    # handle_lines(sys.argv[1])
    handle_lines("program2.bin")
```