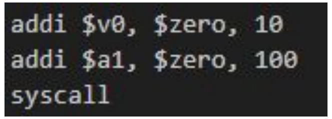
microMIPS

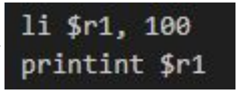
Jacob Tabalon & Maggi Savajiyani

Design Philosophy

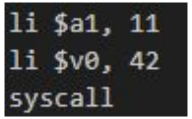
Our goal with microMIPS was to create a form of MIPS that made certain tedious tasks easier, such as printing an integer or generating a random number. The goal was mainly to remove intermediate steps such as syscall and give us more control over where we wanted our data stored. Giving us more control over where we want our arguments and output was important to us. Another thing we wanted to change was making using a stack much easier, so we implemented some instructions that make a stack much more clear to use. Our hope was to make some long MIPS code look shorter/more readable if possible.



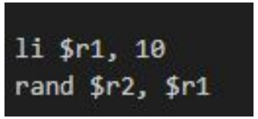
```
addi $v0, $zero, 10
addi $a1, $zero, 100
syscall
```



```
li $r1, 100
printint $r1
```



```
li $a1, 11
li $v0, 42
syscall
```

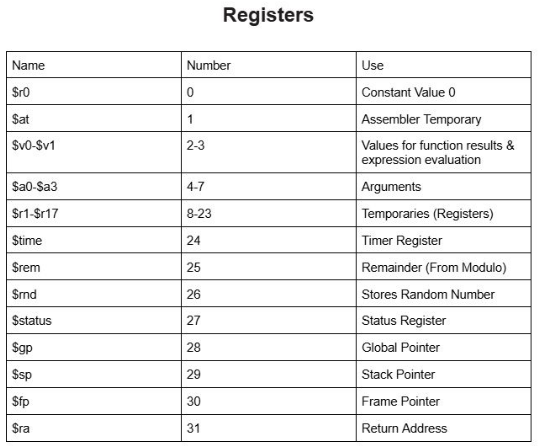


```
li $r1, 10
rand $r2, $r1
```

Changes in Registers

We wanted to simplify some of the registers in MIPS, namely the $t0-$t7 and $s0-$s7 and combine them into $r”number” making it easier to keep track of which registers you have used and have not used. There are also a couple of different registers that we have making it possible to use some of

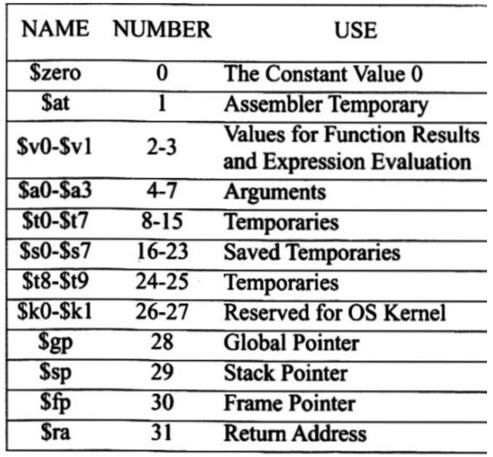
microMIPS



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
$r1-$r17	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

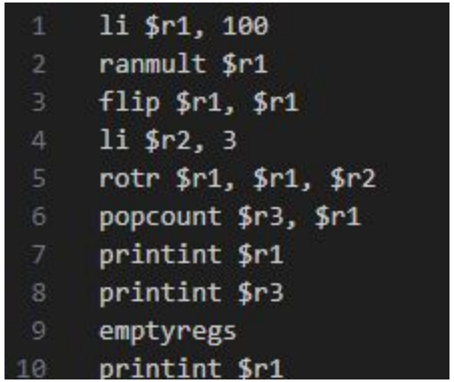
MIPS



NAME	NUMBER	USE
$zero	0	The Constant Value 0
$at	1	Assembler Temporary
$v0-$v1	2-3	Values for Function Results and Expression Evaluation
$a0-$a3	4-7	Arguments
$t0-$t7	8-15	Temporaries
$s0-$s7	16-23	Saved Temporaries
$t8-$t9	24-25	Temporaries
$k0-$k1	26-27	Reserved for OS Kernel
$gp	28	Global Pointer
$sp	29	Stack Pointer
$fp	30	Frame Pointer
$ra	31	Return Address

Assembler

Our assembler was coded in python, making sure to add support for our newly coded instructions and also for the most commonly used MIPS instructions. We created some sample programs and made sure that our assembler ran properly converting our instructions alongside MIPS instructions into our form of machine code, also in 32 bit similarly to MIPS. It handles our custom opcodes and func codes, correctly assembling to our machine code. Because our language is not entirely like MIPS, we had to add support for new formats for arguments such as “instruction $rd” and “instruction $rd, $rs”.



```
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
```

Assembles to

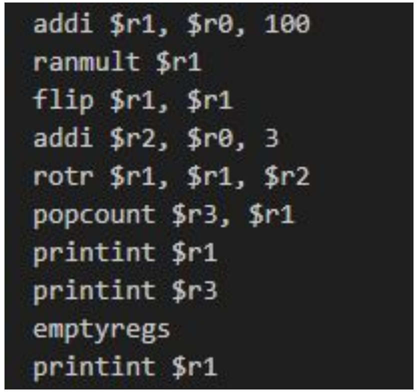
```
001000000000100100000000011001000
101000000000000001001000000000000
000001001000000100100000010101001
000000000101000000000000000110101
100100101010010010000000000001010
101001000000101100000000000010111
010010000000000000000000000101110
101100000000000000000000000000000
000000000000000000010100010111010
01000000000000000000000
```

Disassembler

Our disassembler was also coded in python. Since our language is closely related to MIPS, we just had to add our new op codes and func codes. We also had to add support for our new instruction formats similarly to our assembler. It makes sure that functionality is retained even with pseudo instructions. For example, our original microMIPS code shown on the last slide used li instead of addi, but after disassembling the code it becomes addi.

```
001000000000100100000000011001000
101000000000000001001000000000000
000001001000000100100000010101001
000000000101000000000000000110101
100100101010010010000000000001010
101001000000101100000000000010111
010010000000000000000000000101110
101100000000000000000000000000000
000000000000000000010100010111010
01000000000000000000000
```

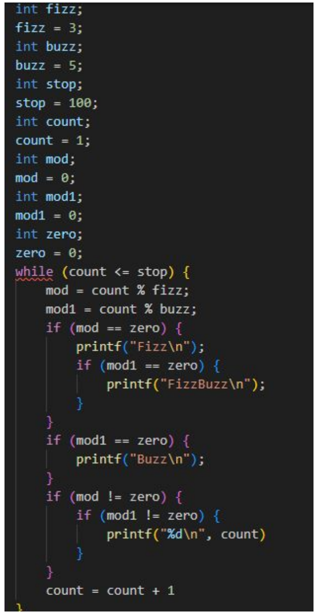
Disassembles to →



```
addi $r1, $r0, 100
ranmult $r1
flip $r1, $r1
addi $r2, $r0, 3
rotr $r1, $r1, $r2
popcount $r3, $r1
printint $r1
printint $r3
emptyregs
printint $r1
```

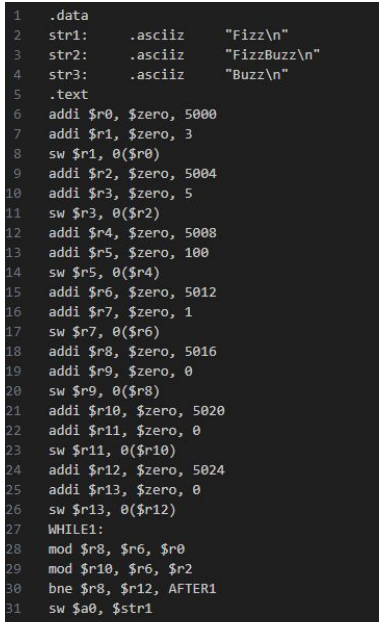
Compiler

For our compiler, we built off of the existing python compiler made in class changing the available registers and adding functionality for the FizzBuzz program. An example of the compiler for fizzbuzz working is shown below:

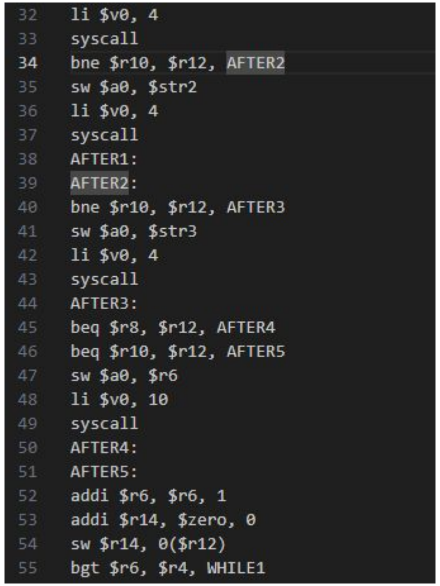


```
int fizz;
fizz = 3;
int buzz;
buzz = 5;
int stop;
stop = 100;
int count;
count = 1;
int mod;
mod = 0;
int mod1;
mod1 = 0;
int zero;
zero = 0;
while (count <= stop) {
    mod = count % fizz;
    mod1 = count % buzz;
    if (mod == zero) {
        printf("Fizz\n");
        if (mod1 == zero) {
            printf("FizzBuzz\n");
        }
    }
    if (mod1 == zero) {
        printf("Buzz\n");
    }
    if (mod != zero) {
        if (mod1 != zero) {
            printf("%d\n", count)
        }
    }
    count = count + 1
}
```

Compiles to



```
.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
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
```

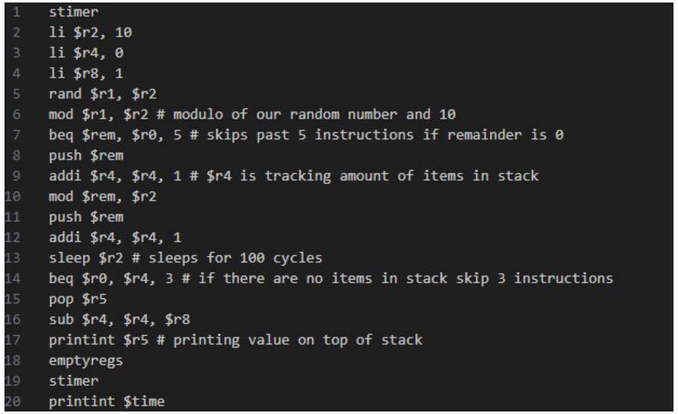


```
li $v0, 4
syscall
bne $r10, $r12, AFTER2
sw $a0, $str2
li $v0, 4
syscall
AFTER1:
AFTER2:
bne $r10, $r12, AFTER3
sw $a0, $str3
li $v0, 4
syscall
AFTER3:
beq $r8, $r12, AFTER4
beq $r10, $r12, AFTER5
sw $a0, $r6
li $v0, 10
syscall
AFTER4:
AFTER5:
addi $r6, $r6, 1
addi $r14, $zero, 0
sw $r14, 0($r12)
bgt $r6, $r4, WHILE1
```

Sample Programs

Our first sample program was FizzBuzz, showcasing that our compiler could properly compile FizzBuzz and that our assembler and disassembler could take the compiled code and turn it into machine code and back. Our second and third programs aimed to show the functionality of special functions, utilizing them to have some fun with numbers messing with their bits and showing what changes when using our functions. Sample programs 1 and 2 were shown in the compiler/assembler/disassembler slides. Our third program mainly focused on using the stack instructions that we created and messing around more with the random and modulo instructions while tracking time taken using our timer alongside a sleep buffer.

Sample Program 3

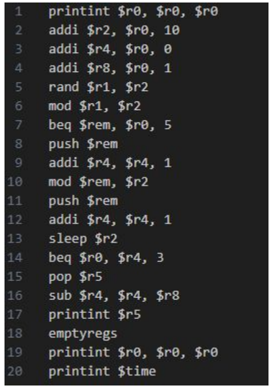


```
stimer
li $r2, 10
li $r4, 0
li $r8, 1
rand $r1, $r2
mod $r1, $r2 # modulo of our random number and 10
beq $rem, $r0, 5 # skips past 5 instructions if remainder is 0
push $rem
addi $r4, $r4, 1 # $r4 is tracking amount of items in stack
mod $rem, $r2
push $rem
addi $r4, $r4, 1
sleep $r2 # sleeps for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

Assembles to

```
0000000000000000000000000000000000010000000001010000000000000001
0100010000000000110000000000000000000000100000000100000000000000
0000010000000010100000000100100000010111000000001001010100000000
0000111100001001100100000000000000000000101000000110010000000000
0000000111000010000110001100000000000000000001000000110010101000
0000000000111100000000110010000000000000000000111000010000110001 1
0000000000000000010000000101000000000000000000001011000010000000
0110000000000000000011000000000000000000110100000011101000000001
1001000000110000000010001001011101101000000000000000000000000000
0000000000000000000000101000000000000000000000000000000000000001
0111110000000000000000000000000
```

Disassembles to

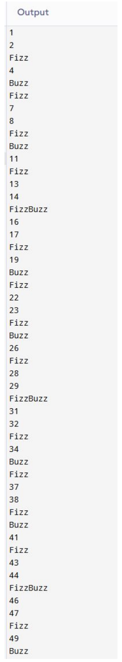


```
printint $r0, $r0, $r0
addi $r2, $r0, 10
addi $r4, $r0, 0
addi $r8, $r0, 1
rand $r1, $r2
mod $r1, $r2
beq $rem, $r0, 5
push $rem
addi $r4, $r4, 1
mod $rem, $r2
push $rem
addi $r4, $r4, 1
sleep $r2
beq $r0, $r4, 3
pop $r5
sub $r4, $r4, $r8
printint $r5
emptyregs
printint $r0, $r0, $r0
printint $time
```

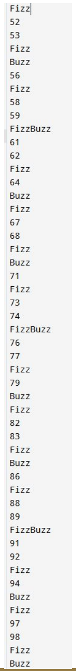
Simulator/Outputs

We did not create a simulator, but our outputs can be seen in our C code.

Program 1: FizzBuzz

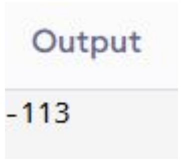


```
Output

1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
Fizz
22
23
Fizz
Buzz
26
Fizz
28
29
FizzBuzz
31
32
Fizz
34
Buzz
Fizz
37
38
Fizz
Buzz
41
Fizz
43
44
FizzBuzz
46
47
Fizz
49
Buzz
```

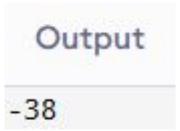


```
Fizz
52
53
Fizz
Buzz
56
Fizz
58
59
FizzBuzz
61
62
Fizz
64
Buzz
Fizz
67
68
Fizz
Buzz
71
Fizz
73
74
FizzBuzz
76
77
Fizz
79
Buzz
Fizz
82
83
Fizz
Buzz
86
Fizz
88
89
FizzBuzz
91
92
Fizz
94
Buzz
Fizz
97
98
Fizz
Buzz
```

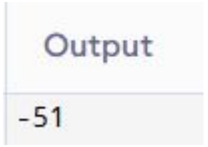
Program 2: Binary Fun



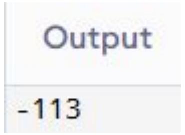
```
Output

-113
```



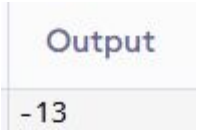
```
Output

-38
```



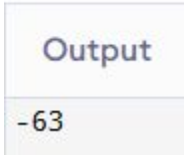
```
Output

-51
```



```
Output

-113
```

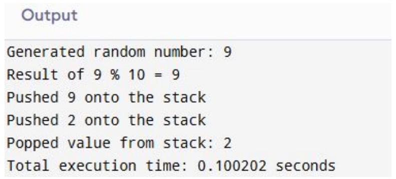


```
Output

-13
```

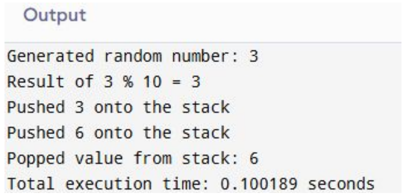


```
Output

-63
```

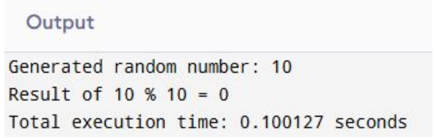
Program 3: Working with stacks (extra print statements for tracking)



```
Output

Generated random number: 9
Result of 9 % 10 = 9
Pushed 9 onto the stack
Pushed 2 onto the stack
Popped value from stack: 2
Total execution time: 0.100202 seconds
```



```
Output

Generated random number: 3
Result of 3 % 10 = 3
Pushed 3 onto the stack
Pushed 6 onto the stack
Popped value from stack: 6
Total execution time: 0.100189 seconds
```



```
Output

Generated random number: 10
Result of 10 % 10 = 0
Total execution time: 0.100127 seconds
```

(when 10 is generated, a lot of instructions get skipped on purpose)