Project 6 - MIPS Functions - amfahe25 Aidan Fahey

collaboration.txt

```
I collaborated with Charlie
```

calc.S

```
1,4d0
< # calc.S implements a calculator-like program.
< # After initializing the program, it starts by printing a greeting with some
< # instructions. Then it interactively gets input and computes various results.
<
20,152d15
< ############# main() #######################################
< # Code for main(), which implements a calculator-like program.
< # It expects no parameters.
< # It returns nothing.
< # The C/C++ equivalent of this function is roughly...
< #   void main() {
< #       printString(welcome);
< #       while (true) {
< #           printString(helpmsg);
< #           ch = getchar();
< #           if (ch == 'p') {
< #               x = getnum()
< #               y = getnum()
< #               r = pow(x, y)
< #               printnum(r)
< #               printString("\n")
< #           } else if (ch == 'f') {
< #               x = getnum()
< #               y = getnum()
< #               fizzbuzz(x, y)
< #           } else if (ch == 'q') {
< #               break;
< #           }
< #       }
< #       printString(goodbye);
< #   }
< # It uses (clobbers) lots of registers, and calls lots of other functions which in turn
< #   also clobber various registers.
< # It creates a 32-byte stack frame to hold:
< #    A backup copy of $31, stored at 28(sp).
< #    A backup copy of user input x, stored at 24(sp).
< #    A backup copy of user input y, stored at 20(sp).
< #    The rest of the space in the stack frame is not used in any way.
< .data
< welcome: .asciz "Welcome to calculator!\n"
< goodbye: .asciz "All done, bye!\n"
< helpmsg: .asciz "[q]uit, [p]ow, or [f]izzbuzz?\n"
< newline: .asciiz "\n"
< stringforfunc: .asciz "Close, but no cigar"
< .text
< main:
<     # function prologue
<     addiu $29, $29, -32  # create space for this function's stack frame
<     sw $31, 28($29)      # store a backup copy of $31 into stack frame
<
<     la $4, welcome
<     jal printString      # call printString(welcome)
<     nop
<
< main_loop:
```

```
<     la $4, helpmsg
<     jal printString      # call printString(helpmsg)
<     nop
<
< try_again:
<     jal getchar          # call ch = getchar()
<     nop                  # result is in r2
<
<     li $3, 0x71            # ascii 'q'
<     beq $2, $3, exit_main  # if ch == 'q' then break out of loop
<
<     li $3, 0x70            # ascii 'p'
<     beq $2, $3, do_pow     # else if ch == 'p' then go to pow code
<
<     li $3, 0x66            # ascii 'f'
<     beq $2, $3, do_fizz    # else if ch == 'p' then go to pow code
<
<     j try_again            # else get another char and try again
<
< do_pow:
<     jal getnum             # x = getnum()
<     nop
<     sw $2, 24($29)         # store x in stack frame for safe keeping
<
<     jal getnum             # y = getnum()
<     nop
<     sw $2, 20($29)         # store y in stack frame for safe keeping
<
<     lw $4, 24($29)         # load x into arg0 register
<     lw $5, 20($29)         # load y into arg1 register
<     jal pow                # call r = pow(x, y)
<     nop
<
<     move $4, $2            # copy result r into arg0 register
<     jal printnum           # call printnum(r)
<     nop
<
<     la $4, newline
<     jal printString        # call printnum(newline)
<     nop
<
<     j main_loop            # go to top of main loop
<
< do_fizz:
<
<     # TODO (4): get user inputs for x and y, then call fizzbuzz
<     jal getnum             # x = getnum()
<     nop
<     sw $2, 24($29)         # store x in stack frame for safe keeping
<
<     jal getnum             # y = getnum()
<     nop
<     sw $2, 20($29)         # store y in stack frame for safe keeping
<
<     lw $4, 24($29)         # load x into arg0 register
<     lw $5, 20($29)         # load y into arg1 register
<     jal fizzbuzz           # call fizz buzz
<
<     j main_loop            # go to top of main loop
<
< do_print_many:
<
<     jal getnum             # x = getnum()
```

```
<     nop
<     sw $2, 24($29)         # store x in stack frame for safe keeping
<
<     lw $4, 24($29)         # load x into arg0 register
<     jal print_many
<
<     j main_loop
<
< exit_main:
<     la $4, goodbye
<     jal printString      # call printString(goodbye)
<     nop
<
<     # function epilogue
<     # note: by this point, the function result should be in r2
<     lw $31, 28($29)      # restore backup copy of $31 from our stack frame
<     addiu $29, $29, 32   # deallocate space used by our stack frame
<     jr $31               # return to whatever called this function
<
<
175,194d37
< .text
< printString:
<     # function prologue
<     addiu $29, $29, -32  # create space for this function's stack frame
<     sw $31, 28($29)      # store a backup copy of $31 into stack frame
<
<     # TODO (1): code for body of printString function goes here
<     lui $8, 0x8000
<     printLoop:
<         lb $9, 0($4)
<         sb $9, 8($8)
<         addiu $4, $4, 1    # fixed by kwalsh, was addui
<         #bne $4, $0, printLoop # fixed by kwalsh, wrong condition
<         bne $9, $0, printLoop
<
<     # function epilogue
<     # note: by this point, the function result should be in r2
<     lw $31, 28($29)      # restore backup copy of $31 from our stack frame
<     addiu $29, $29, 32   # deallocate space used by our stack frame
<     jr $31               # return to whatever called this function
259,263d101
< .text
< pow:
<     # function prologue
<     addiu $29, $29, -32  # create space for this function's stack frame
<     sw $31, 28($29)      # store a backup copy of $31 into stack frame
265,285d102
<     # TODO (2): code for body of pow function goes here
<     li $2, 1
<     powLoop:
<         #mul $2, #2, $4
<         mul $2, $2, $4          # fixed by kwalsh
<         addiu $5, $5, -1
<         bne $5, $0, powLoop
<
<     # function epilogue
<     # note: by this point, the function result should be in r2
<     lw $31, 28($29)      # restore backup copy of $31 from our stack frame
<     addiu $29, $29, 32   # deallocate space used by our stack frame
<     jr $31               # return to whatever called this function
<
<
```

```
< ############# fizzbuzz(a, b) #######################################
< # Code for fizzbuzz(a, b), which counts from a to b (including a, including b),
< #   and for each number, prints the number and either "Fizz", "Buzz", or
< #   "FizzBuzz", depending on whether that number is divisible by 3, divisible by
< #   5, or divisible by both 3 and 5. one newline is printed after the output for
< #   each number.
298,370d114
< # It uses (clobbers) registers .... and it does / does't call other
< # functions....
< # It creates a ...-byte stack frame to hold:
< #    A backup copy of $31, stored at ...(sp).
< #    ... other things ? ...
< #    The rest of the space in the stack frame is not used in any way.
< .data
< # string constants needed by fizzbuzz go here
< fizz: .asciz "Fizz"
< buzz: .asciz "Buzz"
< fizzbuzz_kwalsh: .asciz "FizzBuzz"
< space: .asciz " "
< .text
< fizzbuzz:
<     # function prologue
<     addiu $29, $29, -32  # create space for this function's stack frame
<     sw $31, 28($29)      # store a backup copy of $31 into stack frame
<
<     # TODO (3): code for body of fizzbuzz function goes here
<     move $11, $4
<     move $12, $5
<     li $13, 3
<     li $14, 5
<     li $17, 15
<
<     fizzBuzzMain:
<         move $11, $4
<         jal printnum # fixed by kwalsh, was spelled incorrectly
<         nop
<         la $4, space
<         jal printString
<         nop
<
<         mod $15, $11, $13
<         mod $16, $11, $14
<         mod $18, $11, $17
<
<         beq $18, $0, printFizzBuzz
<         beq $15, $0, printFizz
<         beq $16, $0, printBuzz
<
<         printFizz:
<             la $4, fizz
<             jal printString
<             nop
<             jal endFizzBuzz
<
<         printBuzz:
<             la $4, buzz
<             jal printString
<             nop
<             jal endFizzBuzz
<
<         printFizzBuzz:
<             la $4, fizzbuzz_kwalsh
<             jal printString
```

```
<             nop
<             jal endFizzBuzz
<
<         endFizzBuzz:
<             la $4, newline
<             jal printString
<             nop
<             addiu $11, $11, 1
<             blt $11, $12, fizzBuzzMain  # fixed by kwalsh, added comma
<
<     # function epilogue
<     # note: there is no result, so we don't care what is in r2
<     lw $31, 28($29)      # restore backup copy of $31 from our stack frame
<     addiu $29, $29, 32   # deallocate space used by our stack frame
<     jr $31               # return to whatever called this function
<
<
539,565d282
<
<     # function epilogue
<     # note: by this point, the function result should be in r2
<     lw $31, 28($29)      # restore backup copy of $31 from our stack frame
<     addiu $29, $29, 32   # deallocate space used by our stack frame
<     jr $31               # return to whatever called this function
<
< ############# print_many(a) ######################################
< # Code for print_many(a), which prints a string the number of times specified by the user
< # It expects one arguments, the number of times to print the string
< # It returns nothing
< # Merry Christmas
< .text
< print_many:
<     # function prologue
<     addiu $29, $29, -32  # create space for this function's stack frame
<     sw $31, 28($29)      # store a backup copy of $31 into stack frame
<
<     move $15, $4
<
<     printManyLoop:
<         la $4, stringforfunc
<         jal printString
<         nop
<         addiu $15, $15, -1
<         #bne $15, $0, printManyLoop:
<         bne $15, $0, printManyLoop     # fixed by kwalsh, punctuation
```

```
Assembling calc.S ...
```

Reading assembly code from file: calc.S

Performing first pass: determining memory layout and addresses.

Performing second pass: encoding instructions and data.

Finished processing. Writing out code and data.

Writing assembled code to file: calc_code.txt

Writing assembled data to file: calc_data.txt

SUCCESS! No errors or warnings

Simulate calc with keyboard input [p5\n2\np2\n12\nq]

```
Loading logisim-style memory image 'calc_code.txt' ...
Loading logisim-style memory image 'calc_data.txt' ...
Executing MIPS code...
MIPS program output will be in PLAIN TEXT.
Simulator messages will be in PLAIN TEXT as well.
Note: Each character of user input is shown in curly-braces, e.g. {H}{i}{!}{\n}
Welcome to calculator!
<<NUL>>[q]uit, [p]ow, or [f]izzbuzz?
<<NUL>>{p}pEnter number: <<NUL>>{5}5{\n}
Enter number: <<NUL>>{2}2{\n}
25<<NUL>>
<<NUL>>[q]uit, [p]ow, or [f]izzbuzz?
<<NUL>>{p}pEnter number: <<NUL>>{2}2{\n}
Enter number: <<NUL>>{1}1{2}2{\n}
4096<<NUL>>
<<NUL>>[q]uit, [p]ow, or [f]izzbuzz?
<<NUL>>{q}qAll done, bye!
<<NUL>>
MIPS processor has halted due to apparent infinite-looping behavior.

MIPS code finishes with result $v0 = 0x00000071 (decimal 113, unsigned 113, char 'q').
MIPS processor executed approx. 1617 instructions in 236937 nsec at 6.6824 MHz.
Final state of registers:
    $0 = 0x00000000    $t0 = 0x80000000    $s0 = 0x00000000    $t8 = 0x00000000
   $at = 0x00000002    $t1 = 0x00000000    $s1 = 0x00000000    $t9 = 0x00000000
   $v0 = 0x00000071    $t2 = 0x0000000a    $s2 = 0x00000000    $k0 = 0x00000000
   $v1 = 0x00000071    $t3 = 0x00000000    $s3 = 0x00000000    $k1 = 0x00000000
   $a0 = 0x00000028    $t4 = 0x00000000    $s4 = 0x00000000    $gp = 0x00000000
   $a1 = 0x00000000    $t5 = 0x00000fd3    $s5 = 0x00000000    $sp = 0x00000ffc
   $a2 = 0x00000000    $t6 = 0x00000000    $s6 = 0x00000000    $fp = 0x00000000
   $a3 = 0x00000000    $t7 = 0x00000000    $s7 = 0x00000000    $ra = 0x0000001c
   $hi = 0x00000004    $pc = 0x00000020
   $lo = 0x00000000   $npc = 0x00000024
All data memory locations written by program:
00000080: ..31 3200 .... .... .... .... .... ....   12.

00000fb0: .... .... .... .... 0000 0398 .... ....          ....

00000fd0: .... ..34 3039 3600 0000 0104 .... ....     4096.....

00000ff0: 0000 000c 0000 0002 0000 001c .... ....  ...........
```

Simulate calc with keyboard input [f10\n30\nq]

```
Loading logisim-style memory image 'calc_code.txt' ...
Loading logisim-style memory image 'calc_data.txt' ...
Executing MIPS code...
MIPS program output will be in PLAIN TEXT.
Simulator messages will be in PLAIN TEXT as well.
Note: Each character of user input is shown in curly-braces, e.g. {H}{i}{!}{\n}
```

```
Welcome to calculator!
<<NUL>>[q]uit, [p]ow, or [f]izzbuzz?
<<NUL>>{f}fEnter number: <<NUL>>{1}1{0}0{\n}
Enter number: <<NUL>>{3}3{0}0{\n}
10<<NUL>> <<NUL>>Buzz<<NUL>>
<<NUL>>73<<NUL>> <<NUL>>Fizz<<NUL>>
<<NUL>>[q]uit, [p]ow, or [f]izzbuzz?
<<NUL>>{q}qAll done, bye!
<<NUL>>
MIPS processor has halted due to apparent infinite-looping behavior.

MIPS code finishes with result $v0 = 0x00000071 (decimal 113, unsigned 113, char 'q').
MIPS processor executed approx. 1210 instructions in 173264 nsec at 6.6983 MHz.
Final state of registers:
    $0 = 0x00000000    $t0 = 0x80000000    $s0 = 0x00000003    $t8 = 0x00000000
   $at = 0x0000002c    $t1 = 0x00000000    $s1 = 0x0000000f    $t9 = 0x00000000
   $v0 = 0x00000071    $t2 = 0x0000000a    $s2 = 0x0000000d    $k0 = 0x00000000
   $v1 = 0x00000071    $t3 = 0x0000004a    $s3 = 0x00000000    $k1 = 0x00000000
   $a0 = 0x00000028    $t4 = 0x0000001e    $s4 = 0x00000000    $gp = 0x00000000
   $a1 = 0x0000001e    $t5 = 0x00000fb5    $s5 = 0x00000000    $sp = 0x00000ffc
   $a2 = 0x00000000    $t6 = 0x00000005    $s6 = 0x00000000    $fp = 0x00000000
   $a3 = 0x00000000    $t7 = 0x00000049    $s7 = 0x00000000    $ra = 0x0000001c
   $hi = 0x0000000d    $pc = 0x00000020
   $lo = 0x00000004   $npc = 0x00000024
All data memory locations written by program:
00000080: ..33 3000 .... .... .... .... .... ....   30.

00000f90: .... .... .... .... 0000 0398 .... ....          ....

00000fb0: .... .... ..37 3300 0000 0254 .... ....       73....T

00000fd0: .... .... .... .... 0000 0104 .... ....          ....

00000ff0: 0000 001e 0000 000a 0000 001c .... ....  ............
```