Week 04 Laboratory Sample Solutions

Objectives

- learning to run MIPS programs with spim, xspim or qtspim
- understanding MIPS I/O (syscalls)
- understanding MIPS control instructions (branch)

Preparation

Before the lab you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this lab called `lab04`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab04
$ cd lab04
$ 1521 fetch lab04
```

Or, if you're not working on CSE, you can download the provided code as a zip file or a tar file.

EXERCISE — INDIVIDUAL:

Do You MIPS me?

Write a MIPS assembler program `bad_pun.s`, which is equivalent to this C program:

```c
// A simple C program that attempts to be punny

#include <stdio.h>

int main(void) {
    printf("I MIPS you!\n");

    return 0;
}
```

For example:

```
$ 1521 spim -f bad_pun.s
I MIPS you!
```

> **HINT:**
>
> The i_love_mips.s lecture example would make a good starting point.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest bad_pun
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab04_bad_pun bad_pun.s
```

You must run `give` before **Monday 12 October 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `bad_pun.s`

```
#
# COMP1521 lab exercise sample solution
#
# A simple MIPS program that attempts to be punny
# Written 2/10/2019
# by Andrew Taylor (andrewt@unsw.edu.au)

main:
    la   $a0, string  # get address of string
    li   $v0, 4       # 4 is print string syscall
    syscall

    jr   $ra          # return

    .data
string:
    .asciiz "I MIPS you!\n"
```

EXERCISE — INDIVIDUAL:

MIPS Grading

In the files for this lab, you have been given `grade.s`, a MIPS assembler program which reads a number and always prints **FL**:

```
$ 1521 spim -f grade.s
Enter a mark: 100
FL
```

Add code to `grade.s` to make it equivalent to this C program:

```
// read a mark and print the corresponding UNSW grade

#include <stdio.h>

int main(void) {
    int mark;

    printf("Enter a mark: ");
    scanf("%d", &mark);

    if (mark < 50) {
        printf("FL\n");
    } else if (mark < 65) {
        printf("PS\n");
    } else if (mark < 75) {
        printf("CR\n");
    } else if (mark < 85) {
        printf("DN\n");
    } else {
        printf("HD\n");
    }

    return 0;
}
```

For example:

```
$ 1521 spim -f grade.s
Enter a mark: 42
FL
$ 1521 spim -f grade.s
Enter a mark: 72
CR
$ 1521 spim -f grade.s
Enter a mark: 89
HD
```

HINT:

Make sure you understand the odd_even.s lecture example.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest grade
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab04_grade grade.s
```

You must run `give` before **Monday 12 October 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `grade.s`

```
# Sample COMP1521 Lab solution
# read a mark and print the corresponding UNSW grade
main:
    la   $a0, prompt    # printf("Enter a mark: ");
    li   $v0, 4
    syscall

    li   $v0, 5         # scanf("%d", x);
    syscall

    la   $a0, fl
    blt  $v0, 50, print
    la   $a0, ps
    blt  $v0, 65, print
    la   $a0, cr
    blt  $v0, 75, print
    la   $a0, dn
    blt  $v0, 85, print
    la   $a0, hd

print:
    li   $v0, 4
    syscall
    jr   $ra            # return

    .data
prompt:
    .asciiz "Enter a mark: "
fl:
    .asciiz "FL\n"
ps:
    .asciiz "PS\n"
cr:
    .asciiz "CR\n"
dn:
    .asciiz "DN\n"
hd:
    .asciiz "HD\n"
```

EXERCISE — INDIVIDUAL:

MIPS Counting

In the files for this lab, you have been given a MIPS assembler program `count.s`, which reads a number and prints 42:

```
$ 1521 spim -f count.s
Enter a number: 13
42
```

Add code to `count.s` to make it equivalent to this C program:

```c
// read a number n and print the integers 1..n one per line

#include <stdio.h>

int main(void) {
    int number, i;

    printf("Enter number: ");
    scanf("%d", &number);

    i = 1;
    while (i <= number) {
        printf("%d\n", i);
        i = i + 1;
    }

    return 0;
}
```

For example:

```
$ 1521 spim -f count.s
Enter number: 4
1
2
3
4
$ 1521 spim -f count.s
Enter number: 10
1
2
3
4
5
6
7
8
9
10
```

HINT:

Make sure you understand the print10.s lecture example.

Started by choosing which register you will use for each variable.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest count
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab04_count count.s
```

You must run `give` before **Monday 12 October 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `count.s`

```
# COMP1521 lab sample solution
# read a number n and print the integers 1..n one per line

main:                           # int main(void) {
    # number in $t0; i in $t1

    la   $a0, prompt            # printf("Enter a number: ");
    li   $v0, 4
    syscall

    li   $v0, 5                 # scanf("%d", number);
    syscall
    move $t0, $v0

    li   $t1, 1

loop:                           # loop:
    bgt  $t1, $t0, end          # if ($t1 > $t0) goto end;

    move $a0, $t1               # printf("%d", i);
    li   $v0, 1
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $t1, $t1, 1            # i = i + 1
    j    loop                   # goto loop;

end:
    jr   $ra                    # return

    .data
prompt:
    .asciiz "Enter a number: "
```

EXERCISE — INDIVIDUAL:

MIPS 7-Eleven

In the files for this lab, you have been given `seven_eleven.s`, a MIPS assembler program which reads a number and prints 42:

```
$ 1521 spim -f seven_eleven.s
Enter a number: 13
42
```

Add code to `seven_eleven.s` to make it equivalent to this C program:

```
// Read a number and print positive multiples of 7 or 11 < n

#include <stdio.h>

int main(void) {
    int number, i;

    printf("Enter number: ");
    scanf("%d", &number);

    i = 1;
    while (i < number) {
        if (i % 7 == 0 || i % 11 == 0) {
            printf("%d\n", i);
        }
        i = i + 1;
    }

    return 0;
}
```

For example:

```
$ 1521 spim -f seven_eleven.s
Enter number: 15
7
11
14
$ 1521 spim -f seven_eleven.s
Enter number: 42
7
11
14
21
22
28
33
35
```

HINT:

Make sure you understand the odd_even.s and print_10.s lecture examples.

Start by choosing which register you will use for each variable.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest seven_eleven
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab04_seven_eleven seven_eleven.s
```

You must run `give` before **Monday 12 October 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `seven_eleven.s`

```
# Read a number and print positive multiples of  7 or 11 < n

main:                   # int main(void) {
                        # int number, i;  number in $t0, i in $t1

    la   $a0, prompt    # printf("Enter a number: ");
    li   $v0, 4
    syscall

    li   $v0, 5         # scanf("%d", number);
    syscall
    move $t0, $v0

    li   $t1, 1         # i = 1;

loop:                   # loop:
    bge  $t1, $t0, end  # if (i >= number) goto end;

    rem  $v0, $t1, 7    # if (i % 7 == 0) goto print
    beq  $v0, 0, print

    rem  $v0, $t1, 11   # if (i % 11 != 0) goto next
    bne  $v0, 0, next

print:
    move $a0, $t1       # printf("%d" i);
    li   $v0, 1
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

next:
    addi $t1, $t1, 1    # i = i + 1;

    j    loop           # goto loop;

end:
    jr   $ra            # return

    .data
prompt:
    .asciiz "Enter a number: "
```

CHALLENGE EXERCISE — INDIVIDUAL:

MIPS Tetrahedra

In the files for this lab, you have been given `tetrahedral.s`, a MIPS assembler program that reads a number and prints 42:

```
$ 1521 spim -f tetrahedral.s
Enter a number: 42
42
```

Add code to `tetrahedral.s` to make it equivalent to this C program:

```c
// Read a number n and print the first n tetrahedral numbers
// https://en.wikipedia.org/wiki/Tetrahedral_number

#include <stdio.h>

int main(void) {
    int i, j, n, total, how_many;

    printf("Enter how many: ");
    scanf("%d", &how_many);

    n = 1;

    while (n <= how_many) {
        total = 0;
        j = 1;

        while (j <= n) {
            i = 1;
            while (i <= j) {
                total = total + i;
                i = i + 1;
            }
            j = j + 1;
        }
        printf("%d\n", total);
        n = n + 1;
    }
    return 0;
}
```

For example:

```
$ 1521 spim -f tetrahedral.s
Enter number: 5
1
4
10
20
35
$ 1521 spim -f tetrahedral.s
Enter number: 12
1
4
10
20
35
56
84
120
165
220
286
364
```

HINT:

Make sure you understand the sum_100_squares.s lecture example.

Started by choosing which register you will use for each variable.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest tetrahedral
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab04_tetrahedral tetrahedral.s
```

You must run `give` before **Monday 12 October 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `tetrahedral.s`

```
# Read a number n and print the first n tetrahedral numbers
# https://en.wikipedia.org/wiki/Tetrahedral_number

# i in register $t0
# j in register $t1
# n in register $t2
# total in register $t3
# how_many in register $t4

main:

    la   $a0, prompt    # printf("Enter how many: ");
    li   $v0, 4
    syscall

    li   $v0, 5         # scanf("%d", number);
    syscall
    move $t4, $v0

    li   $t2, 1         # n = 1;

loop0:
    bgt  $t2, $t4, end0 # while (n <= how_many) {
    li   $t3, 0         # total = 0
    li   $t1, 0         # j = 1
loop1:
    bgt  $t1, $t2, print# while (j <= n) {
    li   $t0, 1         # i = 1

loop2:
    bgt  $t0, $t1, next # while (i <= j) {
    add  $t3, $t3, $t0  # total = total + i
    addi $t0, $t0, 1    # i = i + 1
    j    loop2

next:
    addi $t1, $t1, 1    # j = j + 1
    j    loop1

print:
    move $a0, $t3       # printf("%d", total);
    li   $v0, 1
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    addi $t2, $t2, 1    # n = n + 1
    j    loop0
end0:

    jr   $ra            # return

    .data
prompt:
    .asciiz "Enter how many: "
```

CHALLENGE EXERCISE — INDIVIDUAL:

Read & Execute MIPS Instructions

Write a MIPS assembler program `dynamic_load.s` which reads MIPS instructions as signed decimal integers until it reads the value -1, then executes the instructions.

dynamic_load.s should read instructions until it reads the value -1.

dynamic_load.s should then print a message, load the instructions, print another message, exactly as in the examples below.

For example, below is a tiny MIPS assembler program which prints 42. The comment on each line shows how the instruction is encoded, as a hexadecimal and as a signed decimal integer; it is this signed integer value that your program will read.

```
li $a0, 42  # 3404002a  -3422289878
li $v0, 1   # 34020001  -3422420991
syscall     # 0000000c           12
jr $ra      # 03e00008     65011720
```

This is what `dynamic_load.s` must do.

```
$ 1521 spim -f dynamic_load.s
Enter mips instructions as integers, -1 to finish:
-3422289878
-3422420991
12
65011720
-1
Starting executing instructions
42Finished executing instructions
```

The supplied files for the lab include files containing the instructions for some MIPS assembler programs from lectures. You can use these to test your program; for example:

```
$ cat add.instructions
-3422027759
-3421962215
19419168
663585
-3422420991
12
-3422289910
-3422420981
12
65011720
-1
$ 1521 spim -f dynamic_load.s <add.instructions
Enter mips instructions as integers, -1 to finish:
Starting executing instructions
42
Finished executing instructions
$ 1521 spim -f dynamic_load.s <print10.instructions
Enter mips instructions as integers, -1 to finish:
Starting executing instructions
1
2
3
4
5
6
7
8
9
10
Finished executing instructions
$ 1521 spim -f dynamic_load.s <sum_100_squares.instructions
Enter mips instructions as integers, -1 to finish:
Starting executing instructions
338350
Finished executing instructions
```

If you want to experiment with your own tests, this command will give you any MIPS program as integers.

```
$ 1521 mips_instructions 42.s
-3422289878
-3422420991
12
65011720
-1
```

If you want to try creating your own test cases, here is some MIPS assembler that prints a message without using initialized data:

```
# print a string without using pre-initialized data
# for the dynamic load challenge exercise

main:
    li   $a0, 'H'       # printf("%c", 'Hi');
    li   $v0, 11
    syscall

    li   $a0, 'i'       # printf("%c", 'i');
    li   $v0, 11
    syscall

    li   $a0, '\n'      # printf("%c", '\n');
    li   $v0, 11
    syscall

    jr   $ra
```

HINT:

This exercises involves writing only a small amount of code, but it is tricky to get right and will involve some thought and experimentation.

NOTE:

The input to `dynamic_load.s` will be between 1 and 1024 signed decimal integers: spim can't read hexadecimal.

You can assume the instructions don't use initialized data. This mean printing strings in the normal way won't work.

You can assume the instructions read in execute a `jr $ra` instruction to terminate.

You can assume the instructions don't use jump instructions, except for `jr $ra`.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 1521 autotest dynamic_load
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ give cs1521 lab04_dynamic_load dynamic_load.s
```

You must run `give` before **Monday 12 October 21:00** to obtain the marks for this lab exercise. Note that this is an individual exercise, the work you submit with `give` must be entirely your own.

Sample solution for `dynamic_load.s`

```
main:
    la   $t0, dynamic_code # t0 contains the address we store the next instruction

    la   $a0, prompt    # printf("Enter mips instructions as integers, -1 to finish");
    li   $v0, 4
    syscall

loop:
    li   $v0, 5          # scanf("%d", &numbers[i]);
    syscall              #
    beq  $v0, -1, end    #
    sw   $v0, ($t0)      # store entered number in  extra_code
    addi $t0, $t0, 4     # next location to store an instruction
    j    loop

end:
    li   $v0, 65011720   # store jr $ra at finish of extra_code
    sw   $v0, ($t0)      # in case execution reaches there

    la   $a0, start      # printf("Starting executing instructions\n");
    li   $v0, 4
    syscall

    move $s0, $ra        # save our return address in s0

    la   $t0, dynamic_code # jump to the code we've read in
    jal  $t0

    la   $a0, finish     # printf("Finished executing instructions\n");
    li   $v0, 4
    syscall

    jr   $s0             # return using saved return address in $s0

dynamic_code:            # the code segment in SPIM is write-able allowing
.space 4096              # us to write instruction here to be executed
                         # often code segments are configured to be read-only
                         #
                         # we can't put the code in the data segment
                         # because that is configured to be not executable on SPIM
                         # and this is typical to avoid security exploits & weird bugs
.data

prompt:
    .asciiz "Enter mips instructions as integers, -1 to finish:\n"
start:
    .asciiz "Starting executing instructions\n"
finish:
    .asciiz "Finished executing instructions\n"
```

Alternative solution for `dynamic_load.s`

```
.data
    GREET: .asciiz "Enter mips instructions as integers, -1 to finish:\n"
    START: .asciiz "Starting executing instructions\n"
    STOP:  .asciiz "Finished executing instructions\n"
.text
.globl main
main:
    sw   $fp, -4($sp)
    la   $fp, -4($sp)
    sw   $ra, -4($fp) # $ra is the only register that we need after dynamic code
    la   $sp, unsafe# requisition $sp to point into the .text area

    li   $v0, 4   # puts()
    la   $a0, GREET
    syscall

next_input:
    li   $v0, 5   # scanf("%d")
    syscall

    beq  $v0, -1, end_input

    sw   $v0, 0($sp)# save the dynamic instruction into memory at the location pointed to by $sp
    la   $sp, 4($sp)# $sp ins't in the stack, it's in the text, so grow up not down

    j    next_input
end_input:

    li   $v0, 4   # puts()
    la   $a0, START
    syscall

    jal  unsafe# goto dynamic code, and allow dynamic code to return to here

    li   $v0, 4  # puts()
    la   $a0, STOP
    syscall

    lw   $ra, -4($fp)# $fp is assumed to be in the same place after dynamic code as it was before
    la   $sp,  4($fp)
    lw   $fp,   ($fp)
    jr   $ra
unsafe:
```

Submission

When you are finished each exercises make sure you submit your work by running `give`.

You can run `give` multiple times. Only your last submission will be marked.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

Remember you have until **Mon Oct 12 21:00:00 2020** to submit your work.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

You check the files you have submitted here.

Automarking will be run by the lecturer several days after the submission deadline, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run by the lecturer you can view your results here. The resulting mark will also be available via give's web interface.

Lab Marks

When all components of a lab are automarked you should be able to view the the marks via give's web interface or by running this command on a CSE machine:

```
$ 1521 classrun -sturec
```