ENCM 369 Winter 2020 Lab 3 for the Week of February 3

Steve Norman
Department of Electrical & Computer Engineering
University of Calgary

February 2020

Lab instructions and other documents for ENCM 369 can be found at
`people.ucalgary.ca/~norman/encm369winter2020/`

Administrative details

Each student must hand in their own assignment

Later in the course, you will be allowed to work in pairs on some assignments.

Due Dates

The Due Date for this assignment is 3:30pm Friday, February 7.
The Late Due Date is 3:30pm Monday, February 10.

The penalty for handing in an assignment after the Due Date but before the Late Due Date is 3 marks. In other words, X/Y becomes (X–3)/Y if the assignment is late. There will be no credit for assignments turned in after the Late Due Date; they will be returned unmarked.

Marking scheme

A	1	mark
C	8	marks
E	8	marks
F	3	marks
total	20	marks

How to package and hand in your assignments

Please follow the instructions given in Lab 1.

Exercise A: Instructions that try to do bad things

Read This First

The point of this exercise is to help you understand what is going on when MARS programs fail to run to completion.

Here is a little background. Some instructions may ask for behaviour the processor can not deliver, depending on what is in the registers used by the instruction. Here are two examples:

- `lw $s1, 0($s0)` is attempted, but the address in `$s0` is not a multiple of four. With MIPS hardware, words cannot be read from (or written to) addresses that aren't multiples of four.
- `lw $s1, 0($s0)` is attempted, but the address in `$s0` is zero. Zero is a multiple of four, but programs don't have access to memory at address zero. (This is like trying to access data through a null pointer in C.)

In a real MIPS processor, what happens in the above cases—and many other out-of-the-ordinary situations—is that an *exception* occurs. An exception causes the program to be suspended while *exception handling* code is run. (This is a very vague description; exceptions will be covered in much more detail later in the course.)

In MARS, by default, this kind of exception causes immediate termination of a program, with the offending instruction highlighted in the text segment. That's quite useful for debugging—it lets you examine the exact state of registers and memory at the moment things went wrong.

What to Do

Copy all of the files for the directory `encm369w20lab03/exA`,

In MARS, try assembling and running each of the three programs in that directory. Before each run, use the Clear button to clear old messages from the Mars Messages tab.

Write down the messages that appear in the Mars Messages tab for each of the three program runs.

What to Hand In

Hand in a nicely organized list of all of the messages you were asked to write down.

Important notes about MARS programming

Please read this section carefully, because it has information you need to know to read and write assembly language code in Exercises B–F.

Avoid most pseudoinstructions

Except for `la`, which is quite hard to avoid using, please do not use pseudoinstructions in ENCM 369 lab exercises, unless you are explicitly told to do so. The reason for this is that sticking to real instructions helps you learn the real MIPS instruction set.

It's not always easy to know whether you are using a pseudoinstruction, because *many* mnemonics, including very common ones such as `lw`, `sw`, `add`, and `addi`, may be used in either real instructions or pseudoinstructions, depending on their operands.

To look for inadvertent use of pseudoinstructions in your MARS code, uncheck Permit extended (pseudo) instructions and formats in the MARS Settings menu before assembling your code. That way, use of pseudoinstructions will generate error messages. If you get error messages *only* for these kinds of instructions ...

- `la` pseudoinstructions
- pseudoinstructions in start-up code
- load and store instructions with implicit zero offsets

. . . go back to the Settings menu, check Permit extended (pseudo) instructions and formats, and try again to assemble your code. Otherwise, use the Edit tab to fix the lines that the MARS assembler has identified as pseudoinstructions.

Calling conventions

When you think about calling conventions, *always imagine a program with hundreds of procedures containing thousands of procedure calls!* When coding any one procedure, it is crucial to have a set of rules that allows you to make that procedure compatible with the rest of the program, without having to look at the code for any other procedure.

Even though programs you write in this course will only have a few relatively small procedures, you are required to follow the calling conventions.

Here are the rules seen so far in the course:

- Arguments are passed in `$a0`, `$a1`, `$a2`, and `$a3`, in that order.
- The return value, if there is one, goes in `$v0`.
- On exit from a procedure, `$sp` must contain the same address it contained on entry to that procedure.
- Nonleaf procedures should save `$ra` on the stack in the prologue and restore it in the epilogue. There is no need to do this in a leaf procedure.
- Any procedure, leaf or nonleaf, should save any s-registers it uses on the stack in the prologue and restore them in the epilogue.
- Nonleaf procedures must *not* rely on t-registers to maintain their values across procedure calls.
- Nonleaf procedures must separate incoming arguments from outgoing arguments. There are two ways to do this:
  - In the prologue, copy incoming arguments to stack slots.
  - In the prologue, copy incoming arguments to s-registers.

  There is *no need* to restore incoming argument values to a-registers in the epilogue.

How large should a stack frame be?

This question is more complicated than it might seem. The answer will depend on the context in which the question is asked!

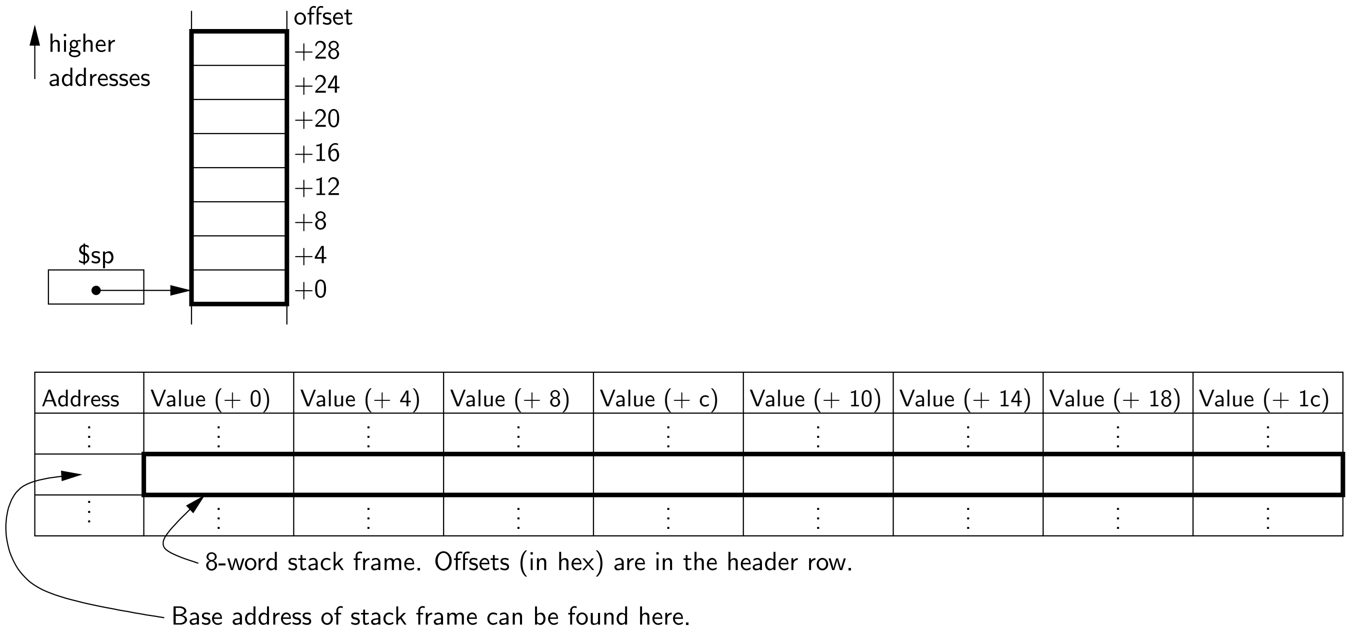
Examples in Section 6.4.6 of the textbook and in ENCM 369 lecture and tutorial examples: *The size of a stack frame should be just big enough to hold all the words that will be packed into it.* This is simple, and also allows for compact, easy-to-draw diagrams.

MARS programming in ENCM 369 labs: *The size of a stack frame should be a multiple of 8 words—so, a multiple of 32 bytes.* So, when deciding how big a stack frame should be, use the following rules:

number of words needed	frame size in bytes
1–8	32
9–16	64
17–24, 25–32, etc.	96, 128, etc.

Here is why: If you follow the above rule, and if the address in `$sp` is a multiple of 32 when `main` starts, then every stack frame will occupy exactly one or two (or more) rows in the MARS Data Segment window. This "wastes" some stack space but makes it relatively easy to locate words within stack frames, as illustrated in Figure 1.

Figure 1: Two ways to visualize an 8-word stack frame. Above: How it would be drawn in a lecture or textbook example. Below: How it would appear in the MARS Data Segment window, assuming that *all* stack frame sizes are multiples of 8 words.



How to allocate local variables

It was suggested early in the course that s-registers are to be allocated for local variables of `int` or pointer type. *In fact, this suggestion is too simple*—in some cases it just won't work, and in others it leads to minor inefficiencies. Here is a better set of rules:

- In a *nonleaf* procedure, the best choice for a local variable of `int` or pointer type is usually an s-register.
- In a *leaf* procedure, the best choice for a local variable of `int` or pointer type is usually a t-register. By choosing a t-register, you avoid the need to save and restore its value to and from the stack.
- But some local variables can't be allocated in registers at all. An array must be in memory. An individual `int` variable whose address is taken with the C `&` operator must be in memory. If a local variable of a procedure must be in memory, space should be allocated within that procedure's stack frame for that variable. (This item is relevant to Lab 4 more than Lab 3; in Lab 3 you won't have to put any local variables in memory.)

Global symbols and the `.globl` directive

The assembler directive

```
            .globl  foo
```

says that `foo` is to be treated as a *global symbol*.

In a real assembly language development environment global symbols are needed when programs are built from multiple source files. (*Source files* in software development are the text files edited by programmers as they work on a program. So in C development, the source files are the `.c` and `.h` files produced by programmers, but in assembly language development, source files are text files containing assembly language code—for MARS, `.asm` files.) Code and data labeled with global symbols can be accessed from all source files belonging to the program. This is necessary when a procedure in one source file calls a procedure in another source file, and when a procedure in one source file accesses an external variable set up in another source file.

MARS allows you to build programs from multiple source files, and in that case, global symbols are necessary. However, for the small programs you will work with in ENCM 369, it is easier to put all the code for a program in a single `.asm` file.

But there is another good reason to make some of your MARS symbols global. In the MARS Labels window, global symbols are listed separately from non-global symbols. It is helpful to make your labels for procedures and external variables global, because the addresses for those labels are probably more interesting than addresses of various labelled instructions in loops and if-statements.

Exercise B: Looking at memory accesses in MARS

Read This First

There are no marks for this exercise, but doing it will show you some things about MARS that will help with many later lab exercises.

What to Do

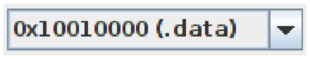
Copy the file `encm369w20lab03/exB/ex3B.asm`

Bring the file `ex3B.asm` into the MARS Edit tab, and assemble it.

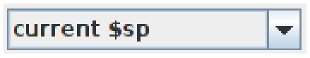
In the Execute tab, put a check in the Bkpt (breakpoint) column beside the instruction at address `0x00400088`, in the Text Segment window. This is the `sw` instruction in `main` that copies the incoming `$ra` value to the stack frame of `main`. Click the Run icon to start the program; you should find the program paused at the breakpoint you just set.

Click the Single-Step icon so that the processor will read and execute the `sw` instruction.

You should see that the control you've previously seen as


0x10010000 (.data)

has changed to

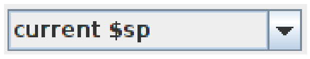

current $sp

That change was automatic as a result of the `sw`, but sometimes you may need to switch manually back and forth between viewing memory starting at `0x10010000` and at current $sp.

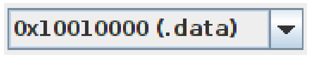
You should also see that `0x0040000c`, the return address from `main` back to the start-up code, has been copied from `$ra` into the stack at an offset of 20 bytes (0x14 bytes) from where `$sp` points.

Slowly single-step through the rest of program execution. Here are some things to watch for:

- The instruction `jal fill` writes `0x00400060` into the PC to get `fill` started, and writes `0x0040009c` into `$ra` to provide a path back from `fill` to exactly the right place in `main`.
- `fill` writes a few words into the `.data` area. Note the automatic change from

  back to

  If you want to see the stack again, you will have to operate the above control manually.
- The `lw` instruction near the end of `main` copies `0x0040000c` back into `$ra` to allow a return back to the start-up code.

What to Hand In

Nothing.

Exercise C: Translating a simple program with procedure calls

What to Do

Make copies of the files in `encm369w20lab03/exC`
Study the program in `functions.c`.

Follow the instructions found in the comments in the C code. To get started, make a copy of the file `stub2.asm`, using `functions.asm` as the name for your copy.

Your assembly language procedures must follow all calling conventions introduced so far in ENCM 369.

Note that `main` is to be treated *just like any other procedure.* That means if it uses s-registers, it must save the old s-register values in the prologue and restore them in the epilogue. It does *not* matter that you can tell from reading the start-up and clean-up code that this save-and-restore is unnecessary—the point is to get as much practice as possible following the usual conventions for coding procedures.

What to Hand In

Hand in a printout of your completed MARS program. Make sure you have printed it in a way that will be *easy* for your teaching assistants to read.

Exercise D: Detailed study of an assembly-language program

Read This First

There are no marks for this exercise, because as you will see, it is easy to check all the answers by running the program in MARS. *However, it is a very important exercise. You can expect to find a similar (but shorter and simpler) problem on Midterm #1.*

This exercise is designed to help solidify your understanding of exactly how registers and stack memory are used to allow assembly-language procedures to work together.

What to Do, Part I

Do not use MARS at all in this part of the exercise!

Print the figures on pages 9 and 10 on separate pieces of paper (not back-to-back on a single sheet).

The assembly-language code in Figure 3 is a correct (carefully tested) translation of the C code in the same Figure.

The program will pass through POINT ONE three times. You are asked to determine the state of the program the *third time* it gets to POINT ONE.

Specifically, your goal is to fill in *all* of the empty boxes for registers and memory words in Figure 4 with either *numbers* or the word "unused" to indicate that the program has neither read nor written a particular memory word.

For any given register or memory word, use either base ten or hexadecimal notation, whichever is more convenient. (For example, the numbers 200 and 1000 will appear in memory words somewhere; there is no need to convert either of them to hex.)

To get you started, the stack frame of `main` has been filled in for you. Note the saved `$ra` and `$s0` values, and the six unused words in the 8-word stack frame.

Here are some important hints:

- It is possible to solve the problem by ignoring the C code, and tracing the effect of every assembly-language instruction starting from the first instruction of `main`. However, that method is slow and painful.

  It is much better to use clues given in the C code.

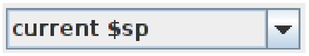
  Example: You can see that `main` calls `procA`, `procA` calls `procB`, and `procB` calls `procC` from within a loop. Also, `procC` is leaf, so probably won't have a stack frame at all. With that information gained from the C code, you can then look at the *prologues* of the assembly language procedures to determine sizes and layouts of stack frames, and mark out the frames of `procA` and `procB` on the diagram of the stack.

  Another example: You can use the C code to determine which words in the array `gg` are pointed to by `p` and `q`. You can see from the assembly language prologue for `procB` that `$s0` and `$s1` are used for `p` and `q`. Because `procC` does not use `$s0` or `$s1`, finding the addresses in `p` and `q` will give you the contents of `$s0` and `$s1` at POINT ONE.

- See Figure 2 for a good method of finding procedure return addresses.

What to Do, Part II

Make copies of the files in `encm369w20lab03/exD` and check your Part I solution using MARS. Set a breakpoint on the first `sll` instruction in `procC`, and run the program until that breakpoint has been hit for the third time. Use


current $sp

to look at the stack, and

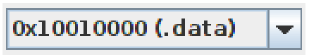

0x10010000 (.data)

Figure 2: A quick way to determine return addresses. In this example the return address from `procA` back to `main` is calculated—it's `0x0040_0150`. Notes: skip lines without instructions; count two real instructions for the pseudoinstruction `la` *GPR*, *label*.

```
main:
    addi    $sp, $sp, -32       # 0x0040_012c
    sw      $ra, 4($sp)         #       _0130
    sw      $s0, 0($sp)         #       _0134

    addi    $s0, $zero, 1000    #       _0138
    addi    $a0, $zero, 200     #       _013c
    la      $a1, gg             #  _0140, _0144
    addi    $a2, $zero, 3       #       _0148
    jal     procA               #       _014c
    add     $s0, $s0, $v0       #       _0150
    add     $v0, $zero, $zero

    lw      $s0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 32
    jr      $ra
```

to look at the array gg. It might (or might not, depending on exactly how you use MARS) be helpful to use

to get all three stack frames visible at the same time. Uncheck

if you would like to see numbers in base ten; check it again for hexadecimal display.

What to Hand In

Nothing.

Figure 3: C and assembly language code listings for Exercise D.

```c
int procC(int x)
{
    // POINT ONE

    return 8 * x + 2 * x;
}

void procB(int *p, int *q)
{
    while (p != q) {
        *p = procC(*p);
        p++;
    }
}

int procA(int s, int *a, int n)
{
    int k;
    k = n - 1;
    procB(a, a + n);
    while (k >= 0) {
        s += a[k];
        k--;
    }
    return s;
}

int gg[] = { 2, 3, 4 };

int main(void)
{
    int mv;
    mv = 1000;
    mv += procA(200, gg, 3);
    return 0;
}
```

```
    .text
    .globl  procC
procC:
    # POINT ONE

    sll     $t0, $a0, 3
    sll     $t1, $a0, 1
    add     $v0, $t0, $t1
    jr      $ra

    .text
    .globl  procB
procB:
    addi    $sp, $sp, -32
    sw      $ra, 8($sp)
    sw      $s1, 4($sp)
    sw      $s0, 0($sp)
    add     $s0, $a0, $zero
    add     $s1, $a1, $zero

L1:
    beq     $s0, $s1, L2
    lw      $a0, ($s0)
    jal     procC
    sw      $v0, ($s0)
    addi    $s0, $s0, 4
    j       L1
L2:
    lw      $s0, 0($sp)
    lw      $s1, 4($sp)
    lw      $ra, 8($sp)
    addi    $sp, $sp, 32
    jr      $ra
```

```
    .text
    .globl  procA
procA:
    addi    $sp, $sp, -32
    sw      $ra, 16($sp)
    sw      $s3, 12($sp)
    sw      $s2, 8($sp)
    sw      $s1, 4($sp)
    sw      $s0, 0($sp)
    add     $s0, $a0, $zero
    add     $s1, $a1, $zero
    add     $s2, $a2, $zero

    addi    $s3, $s2, -1
    add     $a0, $s1, $zero
    sll     $t0, $s2, 2
    add     $a1, $s1, $t0
    jal     procB
L3: slt     $t0, $s3, $zero
    bne     $t0, $zero, L4
    sll     $t1, $s3, 2
    add     $t2, $s1, $t1
    lw      $t3, ($t2)
    add     $s0, $s0, $t3
    addi    $s3, $s3, -1
    j       L3
L4: add     $v0, $s0, $zero

    lw      $s0, 0($sp)
    lw      $s1, 4($sp)
    lw      $s2, 8($sp)
    lw      $s3, 12($sp)
    lw      $ra, 16($sp)
    addi    $sp, $sp, 32
    jr      $ra

    .data
    .globl  gg
gg: .word   2, 3, 4

    .text
    .globl  main
main:
    addi    $sp, $sp, -32
    sw      $ra, 4($sp)
    sw      $s0, 0($sp)

    addi    $s0, $zero, 1000
    addi    $a0, $zero, 200
    la      $a1, gg
    addi    $a2, $zero, 3
    jal     procA
    add     $s0, $s0, $v0
    add     $v0, $zero, $zero

    lw      $s0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 32
    jr      $ra
```

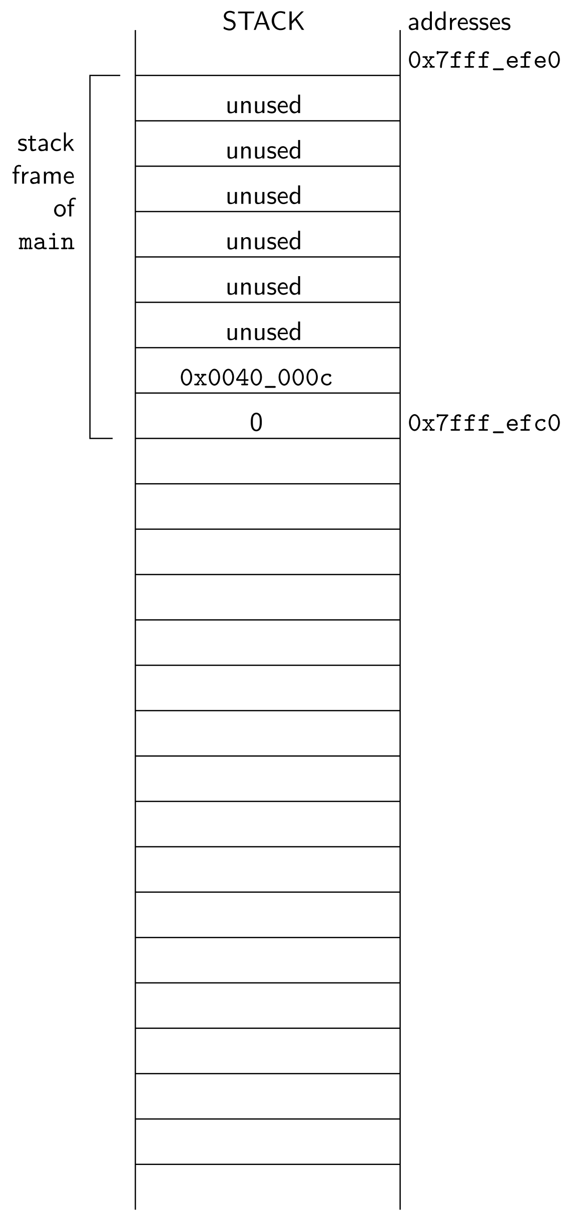
Figure 4: Worksheet for Exercise D.

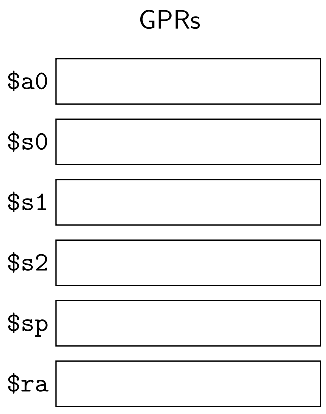
Addresses of procedures and data:

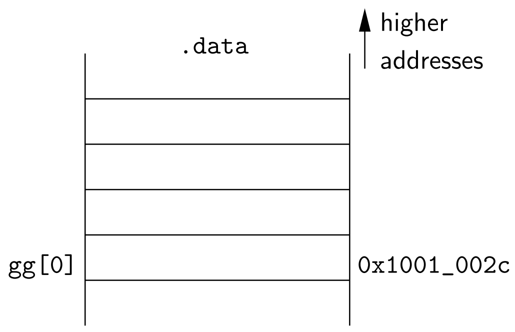
label	address
procC	0x0040_0060
procB	0x0040_0070
procA	0x0040_00b4
main	0x0040_012c
gg	0x1001_002c

Contents of some GPRs when `main` starts:

GPR	contents
$s0-$s7	all 0
$sp	0x7fff_efe0
$ra	0x0040_000c







Exercise E: More practice with procedures

What to Do

Copy the C source file in `encm369w20lab03/exE` and study the program.

Follow the instructions found in the comments in the C code. You can start by making a copy of `stub2.asm` from Exercise C.

Your assembly language procedures must follow all calling conventions introduced so far in ENCM 369.

What to Hand In

Hand in a printout of your completed MARS program.

Exercise F: A function to swap contents of integer variables

What to Do

Copy the files from `encm369w20lab03/exF`

Study the program in `swap.c`, then follow the instructions found in the comments in the C code. Your assembly language procedures must follow all calling conventions introduced so far in ENCM 369.

What to Hand In

Hand in a printout of your completed MARS program.