FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 4

Semester 1, 2019

Objectives of this tutorial

- To understand memory diagrams.
- To understand the function calling and returning convention in MIPS.
- To be able to write simple MIPS functions and translate them from a higher level language.

Exercise 1 *

Consider the following (not very useful) Python code:

```python
def function(x,y):
    ## HERE
    if x >= y:
        return x - y
    return 0

def main():
     x = function(27,2)
     print(x)
```

(a) Draw a memory diagram at the time `## HERE` is found (that is, right before the if-then executes). Ensure the diagram includes the appropriate registers and (if you have time) the appropriate addresses for all memory segments needed (have a look at the examples I gave in the lecture slides to figure out what appropriate addresses are). Mark the stack frames (if any) on the stack.

(b) Translate the above program into MIPS. Try to make your translation as faithful as possible (although the call to print will be done via syscall).

Exercise 2 *

Consider the following Python code:

```python
def even_product(a_list):
    product = 1
    i = len(a_list)
    ## HERE
    while i != 0:
        if i%2 == 0:
            product *= a_list[i]
        i -= 1
    return product
```

Draw a memory diagram at the time `## HERE` is found, assuming **even_product** was called with a list that was just read from input (say [3,-4]). Again, ensure the diagram includes the appropriate registers and (if you have time) the appropriate addresses for all memory segments needed, and mark the stack frames (if any) on the stack.

Exercise 3 *

Consider each of the steps performed as part of the function call/return convention.

1. Caller: saves temporary registers by pushing their values on to the stack
2. Caller: prepares the arguments by pushing them on to the stack

3. Caller: calls the function using the `jal` instruction
4. Callee: saves `$ra` by pushing its value on the stack
5. Callee: saves `$fp` by pushing its value on the stack
6. Callee: copies `$sp` to `$fp`
7. Callee: allocates local variables by reserving enough space onto the stack
8. Callee: if there is a return value, stores it in `$v0`
9. Callee: deallocates local variables by popping the previously pushed space
10. Callee: restores `$fp` by popping its saved value off the stack
11. Callee: restores `$ra` by popping its saved value off the stack
12. Callee: returns using the `jr $ra` instruction
13. Caller: clears the function arguments by popping their allocated space off the stack
14. Caller: restores temporary registers by popping their values off the stack
15. Caller: uses the return value `$v0` if necessary

For each of the steps above:

- Explain the rationale behind the step i.e., explain what is the step trying to achieve in terms of functionality (e.g., needed to allow more than one functions to be called, needed to be able to pass a non-fixed number of parameters, etc).
- Discuss whether the step **must** be performed by the caller/callee or whether this is just a matter of convention (i.e., someone had to do it).

Exercise 4

Translate the above Python program into MIPS. Try to make your translation as faithful as possible (although the call to **len** will be done by directly accessing the length stored in the MIPS array.

Exercise 5

(a) The function calling convention given in lectures typically has functions accessing their first parameter at `8($fp)`, the second at `12($fp)`, the third at `16($fp)`, and so on.
Is this **order** necessary? In other words, would it be possible to have the last parameter at `8($fp)` instead, and the second-last at `12($fp)`, and so on (provided that all functions are changed to agree with this new convention)?

(b) Why do you seldom see functions accessing the memory at address `4($fp)`? (Hint: consult a memory diagram.)