1) Virtual memory allow the address to be read anywhere in the memory, it does not affect that much in modern processors. Top of the stack segment, last in first out. Stack will grow from bottom to top. Data is executed from top to bottom in the stack. It is executed continuosly, more organized in such way sot hat it will fill up the space and does not overlap each other.

FIT1008 Introduction to Computer Science (FIT2085 for Engineers)

Tutorial 2

Semester 1, 2019

Objectives of this tutorial

- To understand better the MIPS architecture.
- To be able to write simple MIPS programs and understand what happens when they are executed.
- To start looking into the format of MIPS instructions.

Remember 3)

You should have read the "Lecture-Tute-Prac Expectations"document that appears in Week 0. As it says there, the purpose of tutorials is not (just) to give you the answers to these questions, but to make you think further about the material covered during the lectures and, in particular, to generate discussion about some of the more interesting and challenging concepts.

You need to prepare answers to the starred tute questions before you come to the tute, and bring the written (possibly wrong) answers to the tute. If you are not prepared to do this, please don't come.

Tutes are often long and there will not be enough time to go through the entire tutorial, so make sure you tell your tutor if you are unsure about a particular question and would like to discuss it.

Exercise 1 *

1) Both heap and stack are dynamic (or non-static) space. The OS cannot determine whether if the stack or heap requires more memory. If certain object-oriented programs utilize more instance variables than local variables, then it would take up more heap than stack. The purpose of growing towards each other is for memory efficiency, if certain instance variables are not using anymore, the memory allocation will shrink in size, allowing more stack memory to be allocated.

Think about the particular decisions taken while designing the MIPS Architecture to answer the following questions:

2) It's easy to debug errors because text segment stores the user's code. Changing the user's code will automatically change the data in the data segment.

- Why do you think is the stack segment designed, seemingly backwards, to extend into lower addresses as it grows? In particular, why do the heap and stack segment grow towards each other, rather than in the same direction?
- Why do you think is a good idea to keep the text segment and data segment separate, rather than all together in memory?
- MIPS provides 32 general-purpose registers for your programs to use, presumably because that's the number that its designers thought best to supply. What decisions and tradeoffs do you think they made in arriving at this number? What is good/bad about a bigger number? And about a smaller one?
- What are the advantages of making all instructions with the same length? And the disadvantages?

Exercise 2 *

4) Advantages: It allow the system to read in a chunk of the length (chunk of 8 bits). No extra rows or hardware is needed to decode a sequence of binaries, but used the 8 bit to process the binaries.
Disadvantages: Unnecessary space is used by the zeros.

Consder the following Python code:

```
1  print("Enter two integers: ")
2  a = int(input())
3  b = int(input())
4
5  s = a + b
6
7  print("Sum is " + str(s))
```

A translation from high-level code (say in Python) into MIPS is **faithful**, if it is done by translating each line of code *independently* of the others (i.e., without optimizing the code by reusing the value of registers computed in previous instructions). While this can introduce considerable space/speed costs, it does make the translation easier, thus reducing the chances of committing mistakes. Since we do not want you to commit mistakes (and we want to be able to understand your code easily), we always ask you to do your translations faithfully.

Assume that somebody has faithfully translated the above code into (glaringly uncommented) MIPS assembly language. Unfortunately, they managed to get some of the lines out of order, obtaining the following MIPS code:

```
        .text

        .data

        la   $a0, sumprompt
        addi $v0, $0, 4
        syscall

        la   $a0, prompt
        addi $v0, $0, 4
        syscall


        addi $v0, $0, 5
        syscall
        sw   $v0, b

        lw   $a0, s
        addi $v0, $0, 1
        syscall

prompt:     .asciiz "Enter two integers: \n"
sumprompt:  .asciiz "Sum is "
newline:    .asciiz "\n"

        addi $v0, $0, 5
        syscall
        sw   $v0, a


        lw   $t0, a
        lw   $t1, b
        add  $t0, $t0, $t1
        sw   $t0, s

a:      .word   0
b:      .word   0
s:      .word   0

        la   $a0, newline
        addi $v0, $0, 4
        syscall

        addi $v0, $0, 10
        syscall
```

Reorder the above MIPS code and comment it so that it is a faithful translation of the Python code and makes sense. To make your life easier, the order within each block of code (where blocks are separated by blank lines) is correct. You just need to reorder the blocks.

Exercise 3 *

Consider the following (glaringly uncommented) MIPS code:

```
        .text
main:   addi $t0, $0, 62
        addi $t1, $0, -28
        addi $t2, $0, 20
        addi $t3, $0, 3
        add  $t4, $t1, $t0
        sub  $t4, $t4, $t2
        mult $t3, $t4
        mflo $t5
        div  $t5, $t4
```

```
11      mflo  $t6
12      div   $t2, $t3
13      mfhi  $t6
```

1. Prepare a table where each column corresponds to one of the following registers: **PC**, **HI**, **LO**, **$0**, **$t0**, **$t1**, **$t2**, **$t3**, **$t4**, **$t5** and **$t6**. For each instruction in the MIPS code, write in the appropriate table entry:
   - The value of the program counter, assuming that for line 2, PC has the value 0x00400000.
   - The value of the registers that participate in the instruction, and
   - A mark (e.g., *), to indicate a register's content has changed.
2. Comment each line of the code in a way that makes it more meaningful to you.

Exercise 4 *

The Python code in one of the previous exercises has been expanded a little:

```
1   print("Enter two integers: ")
2   a = int(input())
3   b = int(input())
4
5   s = a + b
6   d = a - b
7
8   print("Sum is " + str(s))
9   print("Difference is " + str(d))
```

Add to the reordered and commented MIPS code you obtained for that exercise, the MIPS code needed to faithfully translate the new Python code.

Exercise 5

check opcode > check type diagram > check slots > convert instruction to binary

1. An assembler converts assembly language instructions into machine language (which in MIPS is encoded in 32-bit binary). Using the tables and diagram at the end of this tutorial, show the encoding of the following instructions.
   - addi **$t0**, **$zero**, 1 — 001000 00000 01000 (R08) 0000000000000001 [I-type]
   - add **$t2**, **$t0**, **$t1** — 000000 00000 00001 00010 00000 100000[R-type]
   - jr **$ra** — 001000 00000000000000000000100000 [J-type]
2. A disassembler converts the other way; that is, it takes binary machine language instructions and prints them in a human-readable format. Disassemblers are sometimes used by programmers to read programs when the original source code is not available.

   Using the same tables and diagram, determine the assembly language instructions that correspond to the following bit patterns.
   - 00100011101111011111111111110100 — addi $29, $29, -12 [I-type]
   - 00000010000100010001000000100101 — or $2, $16, $17 [J-type]
   - 00000000000001100100000101000000 — sll $8, $0, $6, 5 [J-type]
   - 00001100000000000000000000010101010 — jal 170 [J-type]

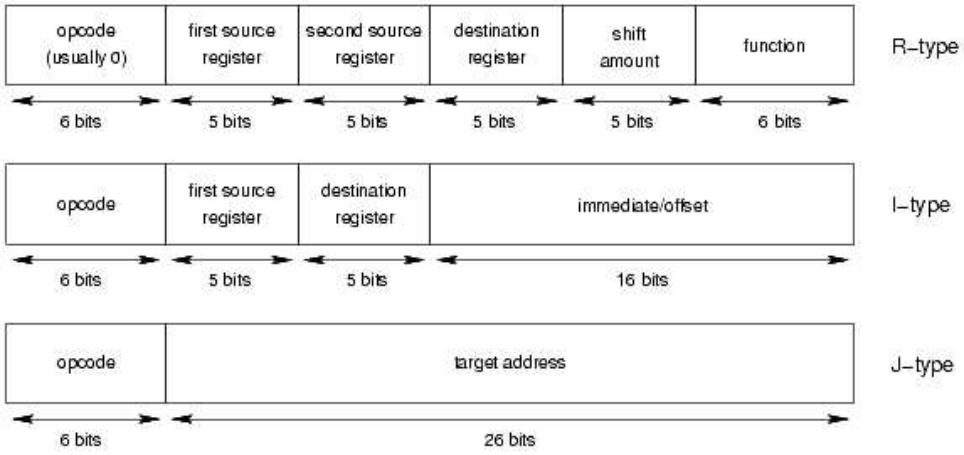
Appendix: MIPS Instruction formats

There are three main formats for MIPS instructions, depending on whether the operands are

- all registers (R-format),

- contain an immediate (constant) value (I-format), or
- contain a label (address).

These are illustrated in the following diagram:



Note that there are some exceptions: for instance, shift instructions sll, srl and sra are R-format instructions, with the first source register unused and the number of bits to shift stored in the "shift amount" field.

Opcode field encodings

This abridged table shows the bit patterns of instructions' opcodes (bits 31-26 of instruction word). The columns show the opcode field in binary, the opcode field in decimal, and the opcode instruction format, respectively.

```
000000  0       operation given     R-type
                in "Function" table
000010  2       j                   J-type
000011  3       jal                 J-type
000100  4       beq                 I-type
000101  5       bne                 I-type
001000  8       addi                I-type
001001  9       addiu               I-type
001010  10      slti                I-type
001011  11      sltiu               I-type
001100  12      andi                I-type
001101  13      ori                 I-type
001110  14      xori                I-type
001111  15      lui                 I-type
100000  32      lb                  I-type
100001  33      lh                  I-type
100011  35      lw                  I-type
100100  36      lbu                 I-type
100101  37      lhu                 I-type
101000  40      sb                  I-type
101001  41      sh                  I-type
101011  43      sw                  I-type
```

Function field encodings

This abridged table shows the bit patterns of the function field (bits 5-0 of instruction word) for R-type instructions, where bits 31-26 are 0. The columns show the function field in binary, the function field in decimal, and the opcode, respectively.

```
000000  0       sll
000010  2       srl
000011  3       sra
000100  4       sllv
000110  6       srlv
000111  7       srav
```

```
001000  8       jr
001001  9       jalr
001100  12      syscall
010000  16      mfhi
010010  18      mflo
011000  24      mult
011001  25      multu
011010  26      div
011011  27      divu
100000  32      add
100001  33      addu
100010  34      sub
100011  35      subu
100100  36      and
100101  37      or
100110  38      xor
100111  39      nor
101010  42      slt
101011  43      sltu
```

Register encodings

Each general purpose register has a corresponding number and is represented in an instruction by the binary representation of that number.