FIT1008 – Intro to Computer Science
Tutorial 2

Semester 1, 2018

Objectives of this tutorial

- To understand the MIPS architecture.
- To be able to write simple MIPS program and understand what happens when they are executed.
- To understand the format of MIPS instructions.

Exercise 1

Think about the particular decisions taken while designing the MIPS Architecture.

- Suggest a reason why the stack segment is designed, seemingly backwards, to extend into lower addresses as it grows. In particular, why the heap and stack segment grow towards each other, rather than in the same direction.
- Suggest a reason why it 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 advantages are there to having all instructions the same length? What disadvantages are there?

Exercise 2

Translate the following Python code into MIPS assembly language. Try to make your translation as faithful as possible, that is, do not try to optimize the code.

```
1  print("Enter two numbers: ")
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))
```

Exercise 3

Consider the following MIPS code:

```
1        .text
2  main: addi $t0, $0, 62
3        addi $t1, $0, -28
4        addi $t2, $0, 20
5        addi $t3, $0, 3
6        add  $t4, $t1, $t0
7        sub  $t4, $t4, $t2
8        mult $t3, $t4
9        mflo $t5
10       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**, **$0**, **$to**, **$t1**, **$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. *Note: For line 2, PC has the value 0x00400000.*
   - the value of the registers that participate in the instruction, and
   - indicate which register's content changed.
2. Comment each line of the code.

Exercise 4

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`
   - `add $t2, $t0, $t1`
   - `jr $ra`

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.

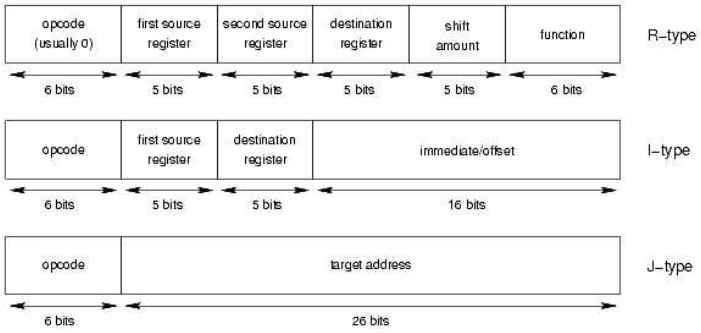
   - **00100011101111011111111111110100**
   - **00000010000100010001000000100101**
   - **00000000000001100100000101000000**
   - **00001100000000000000000000010101010**

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.