FIT1008 Introduction to Computer Science (FIT2085 for Engineers)
Tutorial 2 solutions
Semester 1, 2019

Exercise 1

- Memory is allocated dynamically at run-time in both the heap and stack segments. These two segments grow towards each other, rather than in the same direction, so that the area of memory that is not allocated to your program (i.e., that is free) can be as contiguous as possible (it extends from the top of the data segment up to the bottom of the stack segment). As a result, it is more likely that a request for a large amount of dynamically allocated memory will succeed. Think of what would happen if you split the memory into two, giving X bytes to the heap, and Y bytes. If the heap consumed its X bytes, you would be out of memory even if some of the Y bytes were free, and viceversa. Making them grow towards each other avoids this.
- There are many reasons to keep the text segment and data segment separate, rather than all together in memory:
  - It makes it easier to program, as you don't need to insert instructions in your code to sidestep around non-executable data.
  - it makes it efficient too, as updates to the PC often simply mean incrementing it by 4

  In addition, for those who want to know more, the following might be interesting (would not expect you to be able to come up with this if you didn't know it before):
  - For a multi-tasking system, where many copies of the same program (say, an editor) can run at the same time, you need to store just one copy of the program's code, and as many copies of the program's data as necessary (once for each instance). (Sophisticated memory mapping of modern computers makes this relatively easy to do.) This brings a great saving in space.
  - For embedded systems, where the program doesn't change, code can be kept in read-only memory (ROM), which is cheaper to manufacture.
- There are also many reasons for deciding to have 32 registers in a computer where the word size is 32 bits:
  - Clearly, it makes sense to make the number of registers a power of two, since the register numbers are going to be encoded in binary instructions. It's thus apparent that the other choices were 16 (or 8 or even less), which were deemed too few, and 64 (or 128 or more), which were deemed too many.
  - 16 registers are probably too few because some complex code is likely to require more than ten or so registers (the remaining six-ish are going to be required to keep the system going - such as `$at, $gp, $sp, $ra, $k0` and `$k1` in real MIPS machines). This is especially true given the demarcations the MIPS register usage convention dictates regarding temporaries, arguments, return values and so on. Some machines get by fine with only 16 registers (Motorola 68000 does, for instance, and Intel 80x86 has even fewer), but it is harder for compilers to write good code for these machines. This is part of the reason for the rigidly defined register roles in MIPS: it makes compilers easier to write.
  - It follows from the above this that if 16 isn't desirable, 8 registers is right out.
  - In the other direction, 64 registers would be a possibility, but it's likely that most programs are not nearly complex enough to need that many registers. There's also the problem of real estate on the chip; twice as many registers needs twice the area. Interrupt handlers have to save all register values as they begin so that they can be restored at the end of the interrupt; this would take twice as long to perform. More importantly, instructions would need six bits to refer to each register, which means that the encodings would expand (or the number of bits available to store immediate values would shrink from 16 to an awkward 14 bits). To load a 32-bit constant would take three instructions, not two (for those interested, lookup the lui

instruction). Consequently, code size would bloat and, as a result, run slower for almost no gain.
    - This being the case, 128 registers or more are probably overkill.

  In summary, 32 registers is a compromise between these two arguments: it's big enough to be roomy, without being excessively so, and it's small enough to not impinge on efficiency, without being inconvenient.

- Having all instructions the same length has advantages from the point of view of the fetch-execute cycle. If all instructions are the same length, then only a single memory access is needed to completely fetch an instruction, since its length is known ahead of time. If instructions were of different lengths, then it would be necessary to fetch only the first part of the instruction, and from its bit pattern determine the number of bytes remain in the instruction. This means that memory accesses are required in the decode stage and possibly the execute stage too. This is likely to make the computer's hardware more complex and possibly slower.
  On the flip side, if all instructions are the same length, then this length is dictated by the most complex instruction that the computer is capable of performing. This means that either all instructions are very simple, or the instruction size is wasteful of bits in very simple instructions, which may have been able to be encoded in fewer bits. This tradeoff means that for instructions of MIPS' complexity, code size is about 1/3 larger than for a computer with variable-sized instructions (e.g., Intel x86).

Exercise 2

A possible reordered and commented code is as follows:

```
        .data

prompt:     .asciiz "Enter two numbers: \n"
newline:    .asciiz "\n"
sumprompt:  .asciiz "Sum is "
a:      .word   0
b:      .word   0
s:      .word   0

        .text

        # print prompt
        la   $a0, prompt       # load prompt address into $a0
        addi $v0, $0, 4        # $v0=4
        syscall                # print prompt string

        # read a
        addi $v0, $0, 5        # $v0=5
        syscall                # read int value
        sw   $v0, a            # a = int value just read

        # read b
        addi $v0, $0, 5        # $v0=5
        syscall                # read int value
        sw   $v0, b            # b = int value just read

        # compute s = a + b
        lw   $t0, a            # $t0 = a
        lw   $t1, b            # $t1 = b (faithful: no reuse of $v0)
        add  $t0, $t0, $t1     # t1 = a + b
        sw   $t0, s            # s = a + b

        # print sumprompt
        la   $a0, sumprompt    # load sumprompt address into $a0
        addi $v0, $0, 4        # $v0=4
        syscall                # print sumprompt string

        # print s
```

```
        lw   $a0, s            # $a0 = s
        addi $v0, $0, 1        # $v0 = 1
        syscall                # print s integer

        # print newline
        la   $a0, newline      #load  newline address into $a0
        addi $v0, $0, 4        # $v0 = 4
        syscall                #print newline string

        # exit program
        addi $v0, $0, 10       # $v0 = 10
        syscall                #exit
```

I don't mind if you used a different order for the assembler directives that define the global variables and the string constants (as long as they all appear in between .data and .text), since each is easily identified by the name. However, the order of the rest is important for a faithful translation of the code, so please get accustomed to do faithful translations.

Exercise 3

The following table provides appropriate values for the requested registers. Bold indicates the value of the register has changed w.r.t. its previous use.

PC	HI	LO	$0	$t0	$t1	$t2	$t3	$t4	$t5	$t6
0x0040000			0	**62**						
0x0040004			0		**-28**					
0x0040008			0			**20**				
0x004000c			0				**3**			
0x0040010				62	-28			**34**		
0x0040014						20		**14**		
0x0040018	**0**	**42**					3	14		
0x004001c		42							**42**	
0x0040020	**0**	**3**						14	42	
0x0040024		3								**3**
0x0040028	**2**	**6**				20	3			
0x004002c	2									**2**

A possible way to comment the code is as follows:

```
      .text
main: addi $t0, $0, 62       # $t0 = 62
      addi $t1, $0, −28      # $t1 = −28
      addi $t2, $0, 20       # $t2 = 20
      addi $t3, $0, 3        # $t3 = 3
      add  $t4, $t1, $t0     # $t4 = $t1 + $t0 = 34
      sub  $t4, $t4, $t2     # $t4 = $t4 − $t2 = 14
      mult $t3, $t4          # LO = $t3∗$t4 = 42; HI = 0 (no overflow)
      mflo $t5               # $t5 = LO = 42
      div  $t5, $t4          # LO = $t5//$t4 = 3; HI = $t5 % $t4 = 0
      mflo $t6               # $t6 = LO = 0
      div  $t2, $t3          # LO = $t2//$t3 = 6; HI = $t2 % $t3 = 2
      mfhi $t6               # $t6 = HI = 2
```

Exercise 4

A faithful translation of the extended Python code could be as follows:

```
        .data
prompt:     .asciiz "Enter two numbers: \n"
newline:    .asciiz "\n"
sumprompt:  .asciiz "Sum is "
dprompt:    .asciiz "Difference is "
a:      .word   0
b:      .word   0
s:      .word   0
d:      .word   0

        .text

        # print prompt
        la   $a0, prompt        # load prompt address into $a0
        addi $v0, $0, 4         # $v0=4
        syscall                 # print prompt string

        # read a
        addi $v0, $0, 5         # $v0=5
        syscall                 # read int value
        sw   $v0, a             # a = int value just read

        # read b
        addi $v0, $0, 5         # $v0=5
        syscall                 # read int value
        sw   $v0, b             # b = int value just read

        # compute s = a + b
        lw   $t0, a             # $t0 = a
        lw   $t1, b             # $t1 = b (faithful: no reuse of $v0)
        add $t0, $t0, $t1       # t1 = a + b
        sw   $t0, s             # s = a + b

        # compute d = a - b
        lw   $t0, a             # $t0 = a (faithful: no reuse of $t0)
        lw   $t1, b             # $t1 = b
        sub $t0, $t0, $t1       # $t0 = a - b
        sw   $t0, d             # d = a-b

        # print sumprompt
        la   $a0, sumprompt     # load sumprompt address into $a0
        addi $v0, $0, 4         # $v0=4
        syscall                 # print sumprompt string

        # print s
        lw   $a0, s             # $a0 = s
        addi $v0, $0, 1         # $v0 = 1
        syscall                 # print s integer

        # print newline
        la   $a0, newline       #load  newline address into $a0
        addi $v0, $0, 4         # $v0 = 4
        syscall                 #print newline string

        # print dprompt
        la   $a0, dprompt       #load dprompt address into $a0
        addi $v0, $0, 4         # $v0 = 4
        syscall                 #print dprompt string

        # print d
        lw   $a0, d             # $a0 = d
        addi $v0, $0, 1         # $v0 = 1
        syscall                 # print d integer
```

```
        # print newline
        la  $a0, newline      #load newline address into $a0
        addi $v0, $0, 4       # $v0 = 4
        syscall               # print newline string

        # exit program
        addi $v0, $0, 10      # $v0 = 10
        syscall               #exit
```

Exercise 5   Assembling and disassembling

The encoding of the following instructions is:

- addi $t0, $zero, 1 → I-format: 001000 00000 01000 0000000000000001
- add $t2, $t0, $t1→ R-format: 000000 01000 01001 01010 00000 100000
- jr $ra → R-format: 000000 11111 00000 00000 00000 001000

The assembly language instructions that correspond to the following bit patterns are:

- 00100011101111011111111111110100 addi $sp, $sp, −12
- 00000010000100010001000000100101 or $v0, $s0, $s1
- 00000000000001100100000101000000 sll $t0, $a2, 5
- 00001100000000000000000010101010 jal 170 which is jal 0x000000AA

As you will see later, the immediate values (-12, 5, and 170) might require special interpretation as unsigned, signed, address, offset, etc. For thise who have not seen signed integer representation, don't worry, you will not need this for the unit.

For those who have noticed there is an `addiu` instruction and are wondering whether it means unsigned (as it usually means with other instructions) and therefore the `-12` is wrong, you will be interested in knowing that in this particular case itf **does not mean unsigned**, since `addiu` performed sign extension. The only difference with `addi` is that `addiu` does not check for overflow.