FIT1008 – Intro to Computer Science
Solutions for Tutorial 2

Semester 1, 2018

Exercise 1

- The stack segment is designed, seemingly backwards, to extend into lower addresses as it grows because this permits the area of memory that isn't allocated to your program to be as contiguous as possible (it extends from the top of the data segment up to the bottom of the stack segment). This means that it is more likely that a request for a large amount of dynamically allocated memory will succeed.

- It is a good idea to keep the text segment and data segment separate, rather than all together in memory: With the small programs we'll be writing here, there's not much of an argument for this, except that it makes programs easier to write because you don't need to insert instructions in your code to sidestep around non-executable data.

  For real machines, it's a good idea to do this because code is read-only and data is read/write. This has two effects:

  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.

  Both of these are easier to achieve if text and data are kept separate.

- There are many reasons for deciding to have 32 registers. 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 all 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.

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

```
1                   .data
2   prompt:         .asciiz "Enter two numbers: \n"
3   sumprompt:              .asciiz "Sum is "
4   dprompt:                .asciiz "Difference is "
5   newline:        .asciiz "\n"
6   a:              .word   0
7   b:              .word   0
8   s:              .word   0
9   d:              .word   0
10
11                  .text
12
13                  # print prompt and newline
14
15                  la      $a0, prompt
16                  addi    $v0, $0, 4
17                  syscall
18
19                  # read a
20
21                  addi    $v0, $0, 5
22                  syscall
23                  sw      $v0, a
24
25                  # read b
26
27                  addi    $v0, $0, 5
28                  syscall
29                  sw      $v0, b
30
31                  # compute s = a + b
32
33                  lw      $t0, a
34                  lw      $t1, b
35                  add $t0, $t0, $t1
36                  sw      $t0, s
37
38                  # compute d = a - b
39
40                  lw      $t0, a
```

```
41                  lw      $t1, b
42                  sub     $t0, $t0, $t1
43                  sw      $t0, d
44
45                  # print s
46
47                  la      $a0, sumprompt
48                  addi    $v0, $0, 4
49                  syscall
50
51                  lw      $t0, s
52                  add     $a0, $0, $t0
53                  addi    $v0, $0, 1
54                  syscall
55
56                  la      $a0, newline
57                  addi    $v0, $0, 4
58                  syscall
59
60                  # print d
61
62                  la      $a0, dprompt
63                  addi    $v0, $0, 4
64                  syscall
65
66                  lw      $t0, d
67                  add     $a0, $0, $t0
68                  addi    $v0, $0, 1
69                  syscall
70
71                  la      $a0, newline
72                  addi    $v0, $0, 4
73                  syscall
74
75                  # exit
76
77                  addi    $v0, $0, 10
78                  syscall
```

Exercise 3

PC	HI	LO	$o	$to	$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**

```
1       .text
2  main: addi $t0, $0, 62       # $t0 = 62
3        addi $t1, $0, -28      # $t1 = -28
4        addi $t2, $0, 20       # $t2 = 20
5        addi $t3, $0, 3        # $t3 = 3
6        add  $t4, $t1, $t0     # $t4 = $t1 + $t0
7        sub  $t4, $t4, $t2     # $t4 = $t4 - $t2
8        mult $t3, $t4          # LO = $t3*$t4
9        mflo $t5               # $t5 = LO
10       div  $t5, $t4          # LO = $t5/$t4; HI = $t5 % $t4
11       mflo $t6               # $t6 = LO
12       div  $t2, $t3          # LO = $t2/$t3; HI = $t2 % $t3
13       mfhi $t6               # $t6 = HI
```

Exercise 4

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`
- 00000000000000110010000010100000 `sll $t0, $a2, 5`
- 00001100000000000000000010101010 `jal 170`