Project 1 Supplementary Materials

1. The list of instructions you need to support:

NO Pseudo-instructions or co-processor instructions required.

If there are any conflicts between the book and the official documents on instruction format, follow the book.

1. add
2. addu
3. addi
4. addiu
5. and
6. andi
7. clo
8. clz
9. div
10. divu
11. mult
12. multu
13. mul
14. madd
15. msub
16. maddu
17. msubu
18. nor
19. or
20. ori
21. sll
22. sllv
23. sra
24. srav
25. srl
26. srlv
27. sub
28. subu
29. xor
30. xori
31. lui

32. slt
33. sltu
34. slti
35. sltiu
36. beq
37. bgez
38. bgezal
39. bgtz
40. blez
41. bltzal
42. bltz
43. bne
44. j
45. jal
46. jalr
47. jr
48. teq
49. teqi
50. tne
51. tnei
52. tge
53. tgeu
54. tgei
55. tgeiu
56. tlt
57. tltu
58. tlti
59. tltiu
60. lb
61. lbu
62. lh
63. lhu
64. lw
65. lwl
66. lwr
67. ll
68. sb
69. sh
70. sw

71. swl
72. swr
73. sc
74. mfhi
75. mflo
76. mthi
77. mtlo
78. syscall

The detailed meanings of these instructions, and their format can be found in Appendix A.10. (Syscall on Page A-80.)

*For jumping and branching instructions, you need to support both labels and addresses (offsets).

*For instructions with trap (i.e. overflow trap), print out the error message and terminate the program execution.

2. Syscalls you need to support:

The syscalls you need to support are: **1, 4, 5, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17** in the following chart.

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $a0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $a0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $a0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

How syscall works?

Recall that you will read in a MIPS file as input in your project 1, and you will need to run the code (Read Project 1 instruction first if you do not understand). In your assembling part, you simply need to put the binary code: 00000000000000000000000000001100 in, all syscalls have the same machine code.

How do we distinguish them then? Syscalls are distinguished by checking the value stored in $v0 register. In your simulation part, when ever you see a syscall made (seeing this binary code: 00000000000000000000000000001100), you simply go and check what is stored in $v0. In other words, you can write a switch, with the value in $v0 being the cases. For each case, you simply implement its functionality. Using print_int as an example, in your `case: 1`, you can have `printf("%d", *a0);`. The argument column specifies the arguments this syscall takes (the integer to be printed is stored in $a0, for example).

3. The input MIPS code format

You will need to consider the following situations while reading the input MIPS file:

1. **There will only be .data and .text sections.**
2. There could be spaces or tabs before and after each line.
3. There could be spaces before and after each element within a line. e.g. `add    $t0,    $t1, $t2`.
4. There could be empty lines.
5. There could be comments after the line of code. There could also be a line with only comments. Comments are always following a "#".
6. Labels can be followed by a line of code, or can have it's own line. Labels are labeling the same line of code in both situations.

```
case1

label: add $t0, $t1, $t2

case2

label:
add $t0, $t1, $t2
```

4. The data types you need to support

The data types you need to support are:

1. ascii
2. asciiz
3. word
4. byte

5. half