Solution 2.7

2.7.1

a.	−1391460350
b.	−19629

2.7.2

a.	2903506946
b.	4294947667

2.7.3

a.	AD100002
b.	FFFFB353

2.7.4

a.	01111111111111111111111111111111
b.	1111101000

2.7.5

a.	7FFFFFFF
b.	3E8

2.7.6

a.	80000001
b.	FFFFFC18

Solution 2.13

2.13.1

a.	0x57755778
b.	0xFEFFFEDE

2.13.2

a.	0x00005550
b.	0x0000EED0

2.13.3

a.	0x0000AAAA
b.	0x0000BFCD

2.13.4

a.	0x00015B5A
b.	0x000000D0

2.13.5

a.	0xEFEF0000
b.	0x00000000

2.13.6

a.	0xEFEFFFFF
b.	0x000000F0

Solution 2.19

2.19.1

a.

```
compare:
      addi $sp, $sp, -4
      sw   $ra, 0($sp)

      add  $s0, $a0, $0
      add  $s1, $a1, $0

      jal  sub
      addi $t1, $0, 1
      beq  $v0, $0, exit
      slt  $t2, $0, $v0
      bne  $t2, $0, exit
      addi $t1, $0, $0
exit:
      add  $v0, $t1, $0
      lw   $ra, 0($sp)
      addi $sp, $sp, 4
      jr   $ra

sub:
      sub  $v0, $a0, $a1
      jr   $ra
```

b.

```
fib_iter:
      addi $sp, $sp, -16
      sw   $ra, 12($sp)
      sw   $s0, 8($sp)
      sw   $s1, 4($sp)
      sw   $s2, 0($sp)

      add  $s0, $a0, $0
      add  $s1, $a1, $0
      add  $s2, $a2, $0

      add  $v0, $s1, $0,
      bne  $s2, $0, exit

      add  $a0, $s0, $s1
      add  $a1, $s0, $0
      add  $a2, $s2, -1
      jal  fib_iter
exit:
      lw   $s2, 0($sp)
      lw   $s1, 4($sp)
      lw   $s0, 8($sp)
      lw   $ra, 12($sp)
      addi $sp, $sp, 16
      jr   $ra
```

2.19.2

a.

```
compare:
      addi $sp, $sp, -4
      sw   $ra, 0($sp)

      sub  $t0, $a0, $a1
      addi $t1, $0, 1
      beq  $t0, $0, exit
      slt  $t2, $0, $t0
      bne  $t2, $0, exit
      addi $t1, $0, $0
exit:
      add  $v0, $t1, $0
      lw   $ra, 0($sp)
      addi $sp, $sp, 4
      jr   $ra
```

b. Due to the recursive nature of the code, not possible for the compiler to in-line the function call.

2.19.3

a.

```
after calling function compare:
old $sp =>      0x7ffffffc      ???
$sp =>          -4              contents of register $ra

after calling function sub:
old $sp =>      0x7ffffffc      ???
                -4              contents of register $ra
$sp =>          -8              contents of register $ra   #return to
                                compare
```

b.

```
after calling function fib_iter:
old $sp =>      0x7ffffffc      ???
                -4              contents of register $ra
                -8              contents of register $s0
                -12             contents of register $s1
$sp =>          -16             contents of register $s2
```

2.19.4

a.

```
f:  addi    $sp,$sp,-8
    sw      $ra,4($sp)
    sw      $s0,0($sp)
    move    $s0,$a2
    jal     func
    move    $a0,$v0
    move    $a1,$s0
    jal     func
    lw      $ra,4($sp)
    lw      $s0,0($sp)
    addi    $sp,$sp,8
    jr      $ra
```

b.	`f: addi $sp,$sp,-12`
	`sw $ra,8($sp)`
	`sw $s1,4($sp)`
	`sw $s0,0($sp)`
	`move $s0,$a1`
	`move $s1,$a2`
	`jal func`
	`move $a0,$s0`
	`move $a1,$s1`
	`move $s0,$v0`
	`jal func`
	`add $v0,$v0,$s0`
	`lw $ra,8($sp)`
	`lw $s1,4($sp)`
	`lw $s0,0($sp)`
	`addi $sp,$sp,12`
	`jr ra`

2.19.5

a.	We can use the tail-call optimization for the second call to `func`, but then we must restore $ra and $sp before that call. We save only one instruction (jr $ra).
b.	We can NOT use the tail call optimization here, because the value returned from `f` is not equal to the value returned by the last call to `func`.

2.19.6 Register $ra is equal to the return address in the caller function, registers $sp and $s3 have the same values they had when function `f` was called, and register $t5 can have an arbitrary value. For register $t5, note that although our function `f` does not modify it, function `func` is allowed to modify it so we cannot assume anything about the of $t5 after function `func` has been called.

Solution 2.25

2.25.1 Generally, all solutions are similar:

```
lui $t1, top_16_bits
ori $t1, $t1, bottom_16_bits
```

2.25.2 Jump can go up to 0x0FFFFFFC.

a.	no
b.	no

2.25.3 Range is 0x604 + 0x1FFFC = 0x0002 0600 to 0x604 − 0x20000 = 0xFFFE 0604.

a.	no
b.	yes

2.25.4 Range is 0x0042 0600 to 0x003E 0600.

a.	no
b.	no

2.25.5 Generally, all solutions are similar:

```
add  $t1, $zero, $zero       #clear $t1
addi $t2, $zero, top_8_bits  #set top 8b
sll  $t2, $t2, 24            #shift left 24 spots
or   $t1, $t1, $t2           #place top 8b into $t1
addi $t2, $zero, nxt1_8_bits #set next 8b
sll  $t2, $t2, 16            #shift left 16 spots
or   $t1, $t1, $t2           #place next 8b into $t1
addi $t2, $zero, nxt2_8_bits #set next 8b
sll  $t2, $t2, 24            #shift left 8 spots
or   $t1, $t1, $t2           #place next 8b into $t1
ori  $t1, $t1, bot_8_bits    #or in bottom 8b
```

2.25.6

a.	0x12345678
b.	0x12340000

2.25.7

a.	`t0 = (0x1234 << 16) | 0x5678;`
b.	`t0 = (t0 | 0x5678);` `t0 = 0x1234 << 16;`

Solution 2.39

2.39.1

a.	0.86 seconds
b.	0.78 seconds

2.39.2 Answer is no in all cases. Slows down the computer.

CCT = clock cycle time
ICa = instruction count (arithmetic)
ICls = instruction count (load/store)
ICb = instruction count (branch)

$$\begin{aligned}\text{new CPU time} = {} & 0.75 \times \text{old ICa} \times \text{CPIa} \times 1.1 \times \text{oldCCT} \\ & + \text{oldICls} \times \text{CPIls} \times 1.1 \times \text{oldCCT} \\ & + \text{oldICb} \times \text{CPIb} \times 1.1 \times \text{oldCCT}\end{aligned}$$

The extra clock cycle time adds sufficiently to the new CPU time such that it is not quicker than the old execution time in all cases.

2.39.3

a.	113.16%	121.13%
b.	106.85%	110.64%

2.39.4

a.	3
b.	2.65

2.39.5

a.	0.6
b.	1.07

2.39.6

a.	0.2
b.	0.716666667