Solution 2.5

2.5.1

a.	Address / Data
	12 / 1
	8 / 6
	4 / 4
	0 / 2
		`temp = Array[3];`
		`Array[3] = Array[2];`
		`Array[2] = Array[1];`
		`Array[1] = Array[0];`
		`Array[0] = temp;`

b.	Address / Data
	16 / 1
	12 / 2
	8 / 3
	4 / 4
	0 / 5
		`temp = Array[4];`
		`Array[4] = Array[0];`
		`Array[0] = temp;`
		`temp = Array[3];`
		`Array[3] = Array[1];`
		`Array[1] = temp;`

2.5.2

a.	Address / Data
	12 / 1
	8 / 6
	4 / 4
	0 / 2
		`temp = Array[3];`
		`Array[3] = Array[2];`
		`Array[2] = Array[1];`
		`Array[1] = Array[0];`
		`Array[0] = temp;`
			`lw $t0, 12($s6)`
			`lw $t1, 8($s6)`
			`sw $t1, 12($s6)`
			`lw $t1, 4($s6)`
			`sw $t1, 8($s6)`
			`lw $t1, 0($s6)`
			`sw $t1, 4($s6)`
			`sw $t0, 0($s6)`

b.	Address / Data
	16 / 1
	12 / 2
	8 / 3
	4 / 4
	0 / 5
		`temp = Array[4];`
		`Array[4] = Array[0];`
		`Array[0] = temp;`
		
		`temp = Array[3];`
		`Array[3] = Array[1];`
		`Array[1] = temp;`
			`lw $t0, 16($s6)`
			`lw $t1, 0($s6)`
			`sw $t1, 16($s6)`
			`sw $t0, 0($s6)`
			
			`lw $t0, 12($s6)`
			`lw $t1, 4($s6)`
			`sw $t1, 12($s6)`
			`sw $t0, 4($s6)`

2.5.3

a.	Address / Data
	12 / 1
	8 / 6
	4 / 4
	0 / 2
		`temp = Array[3];`
		`Array[3] = Array[2];`
		`Array[2] = Array[1];`
		`Array[1] = Array[0];`
		`Array[0] = temp;`
			`lw $t0, 12($s6)`
			`lw $t1, 8($s6)`
			`sw $t1, 12($s6)`
			`lw $t1, 4($s6)`
			`sw $t1, 8($s6)`
			`lw $t1, 0($s6)`
			`sw $t1, 4($s6)`
			`sw $t0, 0($s6)`
				8 mips instructions, +1 mips inst. for every non-zero offset lw/sw pair (11 mips inst.)

b.	Address / Data
	16 / 1
	12 / 2
	8 / 3
	4 / 4
	0 / 5
		`temp = Array[4];`
		`Array[4] = Array[0];`
		`Array[0] = temp;`
		
		`temp = Array[3];`
		`Array[3] = Array[1];`
		`Array[1] = temp;`
			`lw $t0, 16($s6)`
			`lw $t1, 0($s6)`
			`sw $t1, 16($s6)`
			`sw $t0, 0($s6)`
			
			`lw $t0, 12($s6)`
			`lw $t1, 4($s6)`
			`sw $t1, 12($s6)`
			`sw $t0, 4($s6)`
				8 mips instructions, +1 mips inst. for every non-zero offset lw/sw pair (11 mips inst.)

2.5.4

a.	305419896
b.	3199070221

2.5.5

	Little-Endian		Big-Endian	
a.	Address	Data	Address	Data
	12	12	12	78
	8	34	8	56
	4	56	4	34
	0	78	0	12
b.	Address	Data	Address	Data
	12	be	12	0d
	8	ad	8	f0
	4	f0	4	ad
	0	0d	0	be

Solution 2.6

2.6.1

a.	`lw  $s0, 4($s7)`
	`sub  $s0, $s0, $s1`
	`add  $s0, $s0, $s2`

b.	`sll  $s1, $s1, 2`
	`add  $t0, $s7, $s1`
	`lw  $t0, 0($t0)`
	`sll  $t0, $t0, 2`
	`add  $t0, $t0, $s6`
	`lw  $s0, 4($t0)`

2.6.2

a.	3
b.	6

2.6.3

a.	4
b.	5

2.6.4

a.	`f = 2i + h;`
b.	`f = A[(g/4)-3];`

2.6.5

a.	$s0 = 110
b.	$s0 = 300

2.6.6

a.

	Type	opcode	rs	rt	rd	immed
`add $s0, $s0, $s1`	R-type	0	16	17	16	
`add $s0, $s3, $s2`	R-type	0	19	18	16	
`add $s0, $s0, $s3`	R-type	0	16	19	16	

b.

	Type	opcode	rs	rt	rd	immed
`addi $s6, $s6, -20`	I-type	8	22	22		–20
`add  $s6, $s6, $s1`	R-type	0	22q	17	22	
`lw  $s0, 8($s6)`	I-type	35	22	16		8

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 2.8

2.8.1

a.	7FFFFFFF, no overflow
b.	80000000, overflow

2.8.2

a.	60000001, no overflow
b.	0, no overflow

2.8.3

a.	EFFFFFFF, overflow
b.	C0000000, overflow

2.8.4

a.	overflow
b.	no overflow

2.8.5

a.	no overflow
b.	no overflow

2.8.6

a.	overflow
b.	no overflow

Solution 2.12

2.12.1

	Type	opcode	rs	rt	rd	shamt	funct	
a.	R-type	6	3	3	3	5	6	total bits = 26
b.	R-type	6	5	5	5	5	6	total bits = 32

2.12.2

	Type	opcode	rs	rt	immed	
a.	I-type	6	3	3	16	total bits = 28
b.	I-type	6	5	5	10	total bits = 26

2.12.3

a.	less registers → less bits per instruction → could reduce code size less registers → more register spills → more instructions
b.	smaller constants → more lui instructions → could increase code size smaller constants → smaller opcodes → smaller code size

2.12.4

a.	17367056
b.	2366177298

2.12.5

a.	add $zero, $t1, $t0 (若funct=32)
b.	lw $t1, 18($t0)

2.12.6

a.	R-type, op=0×0, rt=0×9
b.	I-type, op=0×23, rt=0×8

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 2.14

2.14.1

a.	`add  $t1, $t0, $0`
	`srl  $t1, $t1, 5`
	`andi $t1, $t1, 0x0001ffff`

b.	`add  $t1, $t0, $0`
	`sll  $t1, $t1, 10`
	`andi $t1, $t1, 0xffff8000`

2.14.2

a.	`add  $t1, $t0, $0` `andi $t1, $t1, 0x0000000f`

b.	`add  $t1, $t0, $0`
	`srl  $t1, $t1, 14`
	`andi $t1, $t1, 0x0003c000`

2.14.3

a.	`add  $t1, $t0, $0` `srl  $t1, $t1, 28`

b.	`add  $t1, $t0, $0`
	`srl  $t1, $t1, 14`
	`andi $t1, $t1, 0x0001c000`

2.14.4

a.	`add  $t2, $t0, $0`
	`srl  $t2, $t2, 11`
	`and  $t2, $t2, 0x0000003f`
	`and  $t1, $t1, 0xffffffc0`
	`ori  $t1, $t1, $t2`

b.	`add  $t2, $t0, $0`
	`sll  $t2, $t2, 3`
	`and  $t2, $t2, 0x000fc000`
	`and  $t1, $t1, 0xfff03fff`
	`ori  $t1, $t1, $t2`

2.14.5

a.	`add $t2, $t0, $0`
	`and $t2, $t2, 0x0000001f`
	`and $t1, $t1, 0xffffffe0`
	`ori $t1, $t1, $t2`

b.	`add $t2, $t0, $0`
	`sll $t2, $t2, 14`
	`and $t2, $t2, 0x0007c000`
	`and $t1, $t1, 0xfff83fff`
	`ori $t1, $t1, $t2`

2.14.6

a.	`add $t2, $t0, $0`
	`srl $t2, $t2, 29`
	`and $t2, $t2, 0x00000003`
	`and $t1, $t1, 0xfffffffc`
	`ori $t1, $t1, $t2`

b.	`add $t2, $t0, $0`
	`srl $t2, $t2, 15`
	`and $t2, $t2, 0x0000c000`
	`and $t1, $t1, 0xffff3fff`
	`ori $t1, $t1, $t2`

Solution 2.17

2.17.1 The answer is really the same for all. All of these instructions are either supported by an existing instruction, or sequence of existing instructions. Looking for an answer along the lines of, "these instructions are not common, and we are only making the common case fast".

2.17.2

a.	could be either R-type of I-type
b.	R-type

2.17.3

a.	`ABS:  sub  $t2,$zero,$t3  # t2 = - t3`
	`  ble  $t3,$zero,done  # if t3 < 0, result is t2`
	`  add  $t2,$t3,$zero  # if t3 > 0, result is t3`
	`DONE:`

b.	`slt $t1, $t3, $t2`

2.17.4

a.	20
b.	200

2.17.5

a.	`i = 10;`
	`do {`
	`  B += 2;`
	`  i = i - 1;`
	`} while (i > 0)`

b.	`i = 10;`
	`do {`
	`  temp = 10;`
	`  do {`
	`  B += 2;`
	`  temp = temp - 1;`
	`  } while (temp > 0)`
	`  i = i - 1;`
	`} while (i > 0)`

2.17.6

a.	$5 \times N + 3$
b.	$33 \times N$

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 2.33

2.33.1

a.

```
find:  move $v0,$zero
loop:  beq  $v0,$a1,done
       sll  $t0,$v0,2
       add  $t0,$t0,$a0
       lw   $t0,0($t0)
       bne  $t0,$a2,skip
       jr   $ra
skip:  addi $v0,$v0,1
       b    loop
done:  li   $v0,-1
       jr   $ra
```

b.

```
count: move $v0,$zero
       move $t0,$zero
loop:  beq  $t0,$a1,done
       sll  $t1,$t0,2
       add  $t1,$t1,$a0
       lw   $t1,0($t1)
       bne  $t1,$a2,skip
       addi $v0,$v0,1
skip:  addi $t0,$t0,1
       b    loop
done:  jr   $ra
```

2.33.2

a.

```
int find(int *a, int n, int x){
  int *p;
  for(p=a;p!=a+n;p++)
    if(*p==x)
      return p-a;
  return -1;
}
```

b.

```
int count(int *a, int n, int x){
  int res=0;
  int *p;
  for(p=a;p!=a+n;p++)
    if(*p==x)
      res=res+1;
  return res;
}
```

2.33.3

a.

```
find:  move $t0,$a0
       sll  $t1,$a1,2
       add  $t1,$t1,$a0
loop:  beq  $t0,$t1,done
       lw   $t2,0($t0)
       bne  $t2,$a2,skip
       sub  $v0,$t0,$a0
       srl  $v0,$v0,2
       jr   $ra
skip:  addi $t0,$t0,4
       b loop
done:  li   $v0,-1
       jr   $ra
```

b.

```
find:  move $v0,$zero
       move $t0,$a0
       sll  $t1,$a1,2
       add  $t1,$t1,$a0
loop:  beq  $t0,$t1,done
       lw   $t2,0($t0)
       bne  $t2,$a2,skip
       addi $v0,$v0,1
skip:  addi $t0,$t0,4
       b    loop
done:  jr   $ra
```

2.33.4

	Array-based	Pointer-based
a.	7	5
b.	8	6

2.33.5

	Array-based	Pointer-based
a.	1	3
b.	2	3

2.33.6 Nothing would change. The code would change to save all t-registers we use to the stack, but this change is outside the loop body. The loop body itself would stay exactly the same.

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%	126%
b.	106.85%	113%

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