2 Solutions

Solution 2.1

2.1.1

a.	`add  f, g, h`
	`add  f, f, i`
	`add  f, f, j`

b.	`addi  f, h, 5` `addi  f, f, g`

2.1.2

a.	3
b.	2

2.1.3

a.	14
b.	10

2.1.4

a.	`f = g + h`
b.	`f = g + h`

2.1.5

a.	5
b.	5

Solution 2.2

2.2.1

a.	`add  f, f, f` `add  f, f, i`
b.	`addi f, j, 2` `add  f, f, g`

2.2.2

a.	2
b.	2

2.2.3

a.	6
b.	5

2.2.4

a.	`f += h;`
b.	`f = 1-f;`

2.2.5

a.	4
b.	0

Solution 2.3

2.3.1

a.	`add  f, f, g`
	`add  f, f, h`
	`add  f, f, i`
	`add  f, f, j`
	`addi f, f, 2`

b.	`addi f, f, 5` `sub  f, g, f`

2.3.2

a.	5
b.	2

2.3.3

a.	17
b.	–4

2.3.4

a.	`f = h - g;`
b.	`f = g - f - 1;`

2.3.5

a.	1
b.	0

Solution 2.4

2.4.1

a.	`lw  $s0, 16($s7)`
	`add  $s0, $s0, $s1`
	`add  $s0, $s0, $s2`

b.	`lw  $t0, 16($s7)`
	`lw  $s0, 0($t0)`
	`sub  $s0, $s1, $s0`

2.4.2

a.	3
b.	3

2.4.3

a.	4
b.	4

2.4.4

a.	`f += g + h + i + j;`
b.	`f = A[1];`

2.4.5

a.	no change
b.	no change

2.4.6

a.	5 as written, 5 minimally
b.	2 as written, 2 minimally

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: 12, 8, 4, 0
	Data: 1, 6, 4, 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: 16, 12, 8, 4, 0
	Data: 1, 2, 3, 4, 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: 12, 8, 4, 0	Data: 12, 34, 56, 78	Address: 12, 8, 4, 0	Data: 78, 56, 34, 12
b.	Address: 12, 8, 4, 0	Data: be, ad, f0, 0d	Address: 12, 8, 4, 0	Data: 0d, f0, ad, be

Solution 2.6

2.6.1

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

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

2.6.2

a.	3
b.	4

2.6.3

a.	4
b.	5

2.6.4

a.	`f = 2i + h;`
b.	`f = A[g - 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 2.9

2.9.1

a.	overflow
b.	no overflow

2.9.2

a.	overflow
b.	no overflow

2.9.3

a.	no overflow
b.	overflow

2.9.4

a.	no overflow
b.	no overflow

2.9.5

a.	1D100002
b.	6FFFB353

2.9.6

a.	487587842
b.	1879028563

Solution 2.10

2.10.1

a.	`sw  $t3, 4($s0)`
b.	`lw  $t0, 64($t0)`

2.10.2

a.	I-type
b.	I-type

2.10.3

a.	AE0B0004
b.	8D080040

2.10.4

a.	0x01004020
b.	0x8E690004

2.10.5

a.	R-type
b.	I-type

2.10.6

a.	op=0x0, rd=0x8, rs=0x8, rt=0x0, funct=0x0
b.	op=0x23, rs=0x13, rt=0x9, imm=0x4

Solution 2.11

2.11.1

a.	$1010\ 1110\ 0000\ 1011\ 1111\ 1111\ 1111\ 1100_{two}$
b.	$1000\ 1101\ 0000\ 1000\ 1111\ 1111\ 1100\ 0000_{two}$

2.11.2

a.	2920022012
b.	2366177216

2.11.3

a.	sw $t3, -4($s0)
b.	lw $t0, -64($t0)

2.11.4

a.	R-type
b.	I-type

2.11.5

a.	`add $v1, $at, $v0`
b.	`sw $a1, 4($s0)`

2.11.6

a.	0x00221820
b.	0xAD450004

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 $t0, $t1, $0`
b.	`lw $t1, 12($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.	0x55555550
b.	0xEADFEED0

2.13.3

a.	0x0000AAAA
b.	0x0000BFCD

2.13.4

a.	0x00015B5A
b.	0x00000000

2.13.5

a.	0x5b5a0000
b.	0x000000f0

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 2.15

2.15.1

a.	0x0000a581
b.	0x00ff5a66

2.15.2

a.	`nor   $t1, $t2, $t2` `and   $t1, $t1, $t3`
b.	`xor   $t1, $t2, $t3` `nor   $t1, $t1, $t1`

2.15.3

a.	`nor   $t1, $t2, $t2` `and   $t1, $t1, $t3`	000000 01010 01010 01001 00000 100111 000000 01001 01011 01001 00000 100100
b.	`xor   $t1, $t2, $t3` `nor   $t1, $t1, $t1`	000000 01010 01011 01001 00000 100110 000000 01001 01001 01001 00000 100111

2.15.4

a.	0x00000220
b.	0x00001234

2.15.5 Assuming $t1 = A, $t2 = B, $s1 = base of Array C

a.	`lw $t3, 0($s1)` `and $t1, $t2, $t3`

b.	`beq $t1, $0, ELSE`
	`add $t1, $t2, $0`
	`beq $0, $0, END`
	`ELSE: lw $t2, 0($s1)`
	`END:`

2.15.6

a.	`lw $t3, 0($s1)` `and $t1, $t2, $t3`	100011 10001 01011 0000000000000000 000000 01010 01011 01001 00000 100100

b.	`beq $t1, $0, ELSE`
	`add $t1, $t2, $0`
	`beq $0, $0, END`
	`ELSE: lw $t2, 0($s1)`
	`END:`
		000100 01001 00000 0000000000000010
		000000 01010 00000 01001 00000 100000
		000100 00000 00000 0000000000000001
		100011 10001 01010 0000000000000000

Solution 2.16

2.16.1

a.	$t2 = 1
b.	$t2 = 1

2.16.2

a.	all, 0x8000 to 0x7FFF
b.	0x8000 to 0xFFFE

2.16.3

a.	jump—no, beq—no
b.	jump—no, beq—no

2.16.4

a.	$t2 = 2
b.	$t2 = 2

2.16.5

a.	$t2 = 0
b.	$t2 = 1

2.16.6

a.	jump—yes, beq—no
b.	jump—yes, beq—yes

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)
```

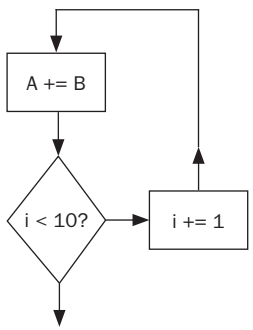
2.17.6

a.	5 × N + 3
b.	33 × N

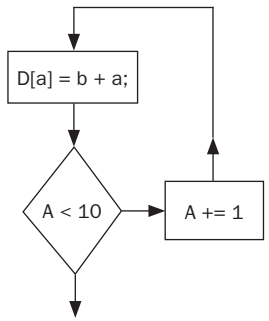
Solution 2.18

2.18.1

a.



b.



2.18.2

a.	`addi $t0, $0, 0`
	`beq $0, $0, TEST`
	`LOOP: add $s0, $s0, $s1`
	`addi $t0, $t0, 1`
	`TEST: slti $t2, $t0, 10`
	`bne $t2, $0, LOOP`

b.	`LOOP: slti $t2, $s0, 10`
	`beq $t2, $0, DONE`
	`add $t3, $s1, $s0`
	`sll $t2, $s0, 2`
	`add $t2, $s2, $t2`
	`sw $t3, ($t2)`
	`addi $s0, $s0, 1`
	`j LOOP`
	`DONE:`

2.18.3

a.	6 instructions to implement and 44 instructions executed
b.	8 instructions to implement and 2 instructions executed

2.18.4

a.	501
b.	301

2.18.5

a.	`for(i=100; i>0; i--){`
	`result += MemArray[s0];`
	`s0 += 1;`
	`}`

b.	`for(i=0; i<100; i+=2){`
	`result += MemArray[s0 + i];`
	`result += MemArray[s0 + i + 1];`
	
	`}`

2.18.6

a.	`addi $t1, $s0, 400`
	`LOOP: lw $s1, 0($s0)`
	`add $s2, $s2, $s1`
	`addi $s0, $s0, 4`
	`bne $s0, $t1, LOOP`

b.	already reduced to minimum instructions

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 =>      0x7fffffc      ???
$sp =>          -4             contents of register $ra

after calling function sub:
old $sp =>      0x7fffffc      ???
                -4             contents of register $ra
$sp =>          -8             contents of register $ra   #return to
                               compare
```

b.

```
after calling function fib_iter:
old $sp =>      0x7fffffc      ???
                -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 2.20

2.20.1

a.

```
FACT: addi  $sp, $sp, -8
      sw    $ra, 4($sp)
      sw    $a0, 0($sp)
      add   $s0, $0, $a0

      slti  $t0, $a0, 2
      beq   $t0, $0, L1

      addi  $v0, $0, 1
      addi  $sp, $sp, 8
      jr    $ra

L1:   addi  $a0, $a0, -1
      jal   FACT

      mul   $v0, $s0, $v0

      lw    $a0, 0($sp)
      lw    $ra, 4($sp)
      addi  $sp, $sp, 8
      jr    $ra
```

b.

```
FACT: addi  $sp, $sp, -8
      sw    $ra, 4($sp)
      sw    $a0, 0($sp)
      add   $s0, $0, $a0

      slti  $t0, $a0, 2
      beq   $t0, $0, L1

      addi  $v0, $0, 1
      addi  $sp, $sp, 8
      jr    $ra

L1:   addi  $a0, $a0, -1
      jal   FACT

      mul   $v0, $s0, $v0

      lw    $a0, 0($sp)
      lw    $ra, 4($sp)
      addi  $sp, $sp, 8
      jr    $ra
```

2.20.2

a. 25 MIPS instructions to execute nonrecursive vs. 45 instructions to execute (corrected version of) recursion

Nonrecursive version:

```
FACT:   addi  $sp, $sp, -4
        sw    $ra, 4($sp)
        add   $s0, $0, $a0
        add   $s2, $0, $1

LOOP:   slti  $t0, $s0, 2
        bne   $t0, $0, DONE
        mul   $s2, $s0, $s2
        addi  $s0, $s0, -1
        j LOOP

DONE:   add   $v0, $0, $s2
        lw    $ra, 4($sp)
        addi  $sp, $sp, 4
        jr    $ra
```

b. 25 MIPS instructions to execute nonrecursive vs. 45 instructions to execute (corrected version of) recursion

Nonrecursive version:

```
FACT:   addi  $sp, $sp, -4
        sw    $ra, 4($sp)
        add   $s0, $0, $a0
        add   $s2, $0, $1

LOOP:   slti  $t0, $s0, 2
        bne   $t0, $0, DONE
        mul   $s2, $s0, $s2
        addi  $s0, $s0, -1
        j LOOP

DONE:   add   $v0, $0, $s2
        lw    $ra, 4($sp)
        addi  $sp, $sp, 4
        jr    $ra
```

2.20.3

a.

```
Recursive version
FACT:   addi  $sp, $sp, -8
        sw    $ra, 4($sp)
        sw    $a0, 0($sp)
        add   $s0, $0, $a0
HERE:   slti  $t0, $a0, 2
        beq   $t0, $0, L1
        addi  $v0, $0, 1
        addi  $sp, $sp, 8
        jr    $ra
L1:     addi  $a0, $a0, -1
        jal   FACT
        mul   $v0, $s0, $v0
        lw    $a0, 0($sp)
        lw    $ra, 4($sp)
        addi  $sp, $sp, 8
        jr    $ra

at label HERE, after calling function FACT with input of 4:
old $sp =>      0xnnnnnnnn      ???
                -4              contents of register $ra
$sp =>          -8              contents of register $a0

at label HERE, after calling function FACT with input of 3:
old $sp =>      0xnnnnnnnn      ???
                -4              contents of register $ra
                -8              contents of register $a0
                -12             contents of register $ra
$sp =>          -16             contents of register $a0

at label HERE, after calling function FACT with input of 2:
old $sp =>      0xnnnnnnnn      ???
                -4              contents of register $ra
                -8              contents of register $a0
                -12             contents of register $ra
                -16             contents of register $a0
                -20             contents of register $ra
$sp =>          -24             contents of register $a0

at label HERE, after calling function FACT with input of 1:
old $sp =>      0xnnnnnnnn      ???
                -4              contents of register $ra
                -8              contents of register $a0
                -12             contents of register $ra
                -16             contents of register $a0
                -20             contents of register $ra
                -24             contents of register $a0
                -28             contents of register $ra
$sp =>          -32             contents of register $a0
```

b.

```
Recursive version
FACT:   addi  $sp, $sp, -8
        sw    $ra, 4($sp)
        sw    $a0, 0($sp)
        add   $s0, $0, $a0
HERE:   slti  $t0, $a0, 2
        beq   $t0, $0, L1
        addi  $v0, $0, 1
        addi  $sp, $sp, 8
        jr    $ra
L1:     addi  $a0, $a0, -1
        jal   FACT
        mul   $v0, $s0, $v0
        lw    $a0, 0($sp)
        lw    $ra, 4($sp)
        addi  $sp, $sp, 8
        jr    $ra

at label HERE, after calling function FACT with input of 4:
old $sp =>      0xnnnnnnnn      ???
                -4              contents of register $ra
$sp =>          -8              contents of register $a0

at label HERE, after calling function FACT with input of 3:
old $sp =>      0xnnnnnnnn      ???
                -4              contents of register $ra
                -8              contents of register $a0
                -12             contents of register $ra
$sp =>          -16             contents of register $a0

at label HERE, after calling function FACT with input of 2:
old $sp =>      0xnnnnnnnn      ???
                -4              contents of register $ra
                -8              contents of register $a0
                -12             contents of register $ra
                -16             contents of register $a0
                -20             contents of register $ra
$sp =>          -24             contents of register $a0

at label HERE, after calling function FACT with input of 1:
old $sp =>      0xnnnnnnnn      ???
                -4              contents of register $ra
                -8              contents of register $a0
                -12             contents of register $ra
                -16             contents of register $a0
                -20             contents of register $ra
                -24             contents of register $a0
                -28             contents of register $ra
$sp =>          -32             contents of register $a0
```

2.20.4

a.

```
FIB:    addi  $sp, $sp, -12
        sw    $ra, 8($sp)
        sw    $s1, 4($sp)
        sw    $a0, 0($sp)

        slti  $t0, $a0, 3
        beq   $t0, $0, L1
        addi  $v0, $0, 1
        j     EXIT

L1:     addi  $a0, $a0, -1
        jal   FIB
        addi  $s1, $v0, $0
        addi  $a0, $a0, -1

        jal   FIB
        add   $v0, $v0, $s1

EXIT:   lw    $a0, 0($sp)
        lw    $s1, 4($sp)
        lw    $ra, 8($sp)
        addi  $sp, $sp, 12
        jr    $ra
```

b.

```
FIB:    addi  $sp, $sp, -12
        sw    $ra, 8($sp)
        sw    $s1, 4($sp)
        sw    $a0, 0($sp)

        slti  $t0, $a0, 3
        beq   $t0, $0, L1
        addi  $v0, $0, 1
        j     EXIT

L1:     addi  $a0, $a0, -1
        jal   FIB
        addi  $s1, $v0, $0
        addi  $a0, $a0, -1

        jal   FIB
        add   $v0, $v0, $s1

EXIT:   lw    $a0, 0($sp)
        lw    $s1, 4($sp)
        lw    $ra, 8($sp)
        addi  $sp, $sp, 12
        jr    $ra
```

2.20.5

a. 23 MIPS instructions to execute nonrecursive vs. 73 instructions to execute (corrected version of) recursion

Nonrecursive version:

```
FIB:    addi  $sp, $sp, -4
        sw    $ra, ($sp)
        addi  $s1, $0, 1
        addi  $s2, $0, 1
LOOP:   slti  $t0, $a0, 3
        bne   $t0, $0, EXIT
        add   $s3, $s1, $0
        add   $s1, $s1, $s2
        add   $s2, $s3, $0
        addi  $a0, $a0, -1
        j LOOP
EXIT:   add   $v0, s1, $0
        lw    $ra, ($sp)
        addi  $sp, $sp, 4
        jr    $ra
```

b. 23 MIPS instructions to execute nonrecursive vs. 73 instructions to execute (corrected version of) recursion

Nonrecursive version:

```
FIB:    addi  $sp, $sp, -4
        sw    $ra, ($sp)
        addi  $s1, $0, 1
        addi  $s2, $0, 1
LOOP:   slti  $t0, $a0, 3
        bne   $t0, $0, EXIT
        add   $s3, $s1, $0
        add   $s1, $s1, $s2
        add   $s2, $s3, $0
        addi  $a0, $a0, -1
        j LOOP
EXIT:   add   $v0, s1, $0
        lw    $ra, ($sp)
        addi  $sp, $sp, 4
        jr    $ra
```

2.20.6

a.

```
recursive version
FIB:    addi  $sp, $sp, -12
        sw    $ra, 8($sp)
        sw    $s1, 4($sp)
        sw    $a0, 0($sp)

HERE:   slti  $t0, $a0, 3
        beq   $t0, $0, L1
        addi  $v0, $0, 1
        j     EXIT

L1:     addi  $a0, $a0, -1
        jal   FIB
        addi  $s1, $v0, $0
        addi  $a0, $a0, -1

        jal   FIB
        add   $v0, $v0, $s1

EXIT:   lw    $a0, 0($sp)
        lw    $s1, 4($sp)
        lw    $ra, 8($sp)
        addi  $sp, $sp, 12
        jr    $ra

at label HERE, after calling function FIB with input of 4:
old $sp =>      0xnnnnnnnn      ???
                -4              contents of register $ra
                -8              contents of register $s1
$sp =>          -12             contents of register $a0
```

b.

```
recursive version
FIB:    addi  $sp, $sp, -12
        sw    $ra, 8($sp)
        sw    $s1, 4($sp)
        sw    $a0, 0($sp)

HERE:   slti  $t0, $a0, 3
        beq   $t0, $0, L1
        addi  $v0, $0, 1
        j     EXIT

L1:     addi  $a0, $a0, -1
        jal   FIB
        addi  $s1, $v0, $0
        addi  $a0, $a0, -1

        jal   FIB
        add   $v0, $v0, $s1

EXIT:   lw    $a0, 0($sp)
        lw    $s1, 4($sp)
        lw    $ra, 8($sp)
        addi  $sp, $sp, 12
        jr    $ra

at label HERE, after calling function FIB with input of 4:
old $sp =>      0xnnnnnnnn      ???
                -4              contents of register $ra
                -8              contents of register $s1
$sp =>          -12             contents of register $a0
```

Solution 2.21

2.21.1

a.

```
after entering function main:
old $sp =>      0x7fffffc      ???
$sp =>          -4             contents of register $ra

after entering function leaf_function:
old $sp =>      0x7fffffc      ???
                -4             contents of register $ra
$sp =>          -8             contents of register $ra (return to main)
```

b.

```
after entering function main:
old $sp =>      0x7fffffc      ???
$sp =>          -4             contents of register $ra

after entering function my_function:
old $sp =>      0x7fffffc      ???
                -4             contents of register $ra
$sp =>          -8             contents of register $ra (return to main)

global pointers:
0x10008000      100            my_global
```

2.21.2

a.

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

        addi $a0, $0, 1
        jal  LEAF

        lw   $ra, ($sp)
        addi $sp, $sp, 4
        jr   $ra

LEAF:   addi $sp, $sp, -8
        sw   $ra, 4($sp)
        sw   $s0, 0($sp)

        addi $s0, $a0, 1
        slti $t2, 5, $a0
        bne  $t2, $0, DONE
        add  $a0, $s0, $0
        jal  LEAF

DONE:   add  $v0, $s0, $0
        lw   $s0, 0($sp)
        lw   $ra, 4($sp)
        addi $sp, $sp, 8
        jr   $ra
```

b.

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

        addi $a0, $0, 10
        addi $t1, $0, 20
        lw   $a1, ($s0)   #assume $s0 has global variable base
        jal  FUNC
        add  $t2, $v0 $0

        lw   $ra, ($sp)
        addi $sp, $sp, 4
        jr   $ra

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

2.21.3

a.

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

        addi $a0, $0, 1
        jal  LEAF

        lw   $ra, ($sp)
        addi $sp, $sp, 4
        jr   $ra

LEAF:   addi $sp, $sp, -8
        sw   $ra, 4($sp)
        sw   $s0, 0($sp)

        addi $s0, $a0, 1
        slti $t2, 5, $a0
        bne  $t2, $0, DONE
        add  $a0, $s0, $0
        jal  LEAF

DONE:   add  $v0, $s0, $0
        lw   $s0, 0($sp)
        lw   $ra, 4($sp)
        addi $sp, $sp, 8
        jr   $ra
```

b.

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

        addi $a0, $0, 10
        addi $t1, $0, 20
        lw   $a1, ($s0)   #assume $s0 has global variable base
        jal  FUNC
        add  $t2, $v0 $0

        lw   $ra, ($sp)
        addi $sp, $sp, 4
        jr   $ra

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

2.21.4

a.	Register $s0 is used to hold a temporary result without saving $s0 first. To correct this problem, $t0 (or $v0) should be used in place of $s0 in the first two instructions. Note that a sub-optimal solution would be to continue using $s0, but add code to save/restore it.
b.	The two addi instructions move the stack pointer in the wrong direction. Note that the MIPS calling convention requires the stack to grow down. Even if the stack grew up, this code would be incorrect because $ra and $s0 are saved according to the stack-grows-down convention.

2.21.5

a.	`int f(int a, int b, int c, int d){`
	`  return 2*(a-d)+c-b;`
	`}`

b.	`int f(int a, int b, int c){`
	`  return g(a,b)+c;`
	`}`

2.21.6

a.	The function returns 842 (which is 2 × (1 – 30) + 1000 – 100)
b.	The function returns 1500 (g(a, b) is 500, so it returns 500 + 1000)

Solution 2.22

2.22.1

a.	`65 20 98 121 116 101`
b.	`99 111 109 112 117 116 101 114`

2.22.2

a.	`U+0041, U+0020, U+0062, U+0079, U+0074, U+0065`
b.	`U+0063, U+006f, U+006d, U+0070, U+0075, U+0074, U+0065, U+0072`

2.22.3

a.	`add`
b.	`shift`

Solution 2.23

2.23.1

a.

```
MAIN:   addi $sp, $sp, -4
        sw   $ra, ($sp)
        add  $t6, $0, 0x30  # '0'
        add  $t7, $0, 0x39  # '9'
        add  $s0, $0, $0
        add  $t0, $a0, $0

LOOP:   lb   $t1, ($t0)
        slt  $t2, $t1, $t6
        bne  $t2, $0, DONE
        slt  $t2, $t7, $t1
        bne  $t2, $0, DONE
        sub  $t1, $t1, $t6

        beq  $s0, $0, FIRST
        mul  $s0, $s0, 10
FIRST:  add  $s0, $s0, $t1
        addi $t0, $t0, 1
        j LOOP

DONE:   add  $v0, $s0, $0
        lw   $ra, ($sp)
        addi $sp, $sp, 4
        jr   $ra
```

b.

```
MAIN:   addi $sp, $sp, -4
        sw   $ra, ($sp)
        add  $t4, $0, 0x41  # 'A'
        add  $t5, $0, 0x46  # 'F'
        add  $t6, $0, 0x30  # '0'
        add  $t7, $0, 0x39  # '9'
        add  $s0, $0, $0
        add  $t0, $a0, $0

LOOP:   lb   $t1, ($t0)
        slt  $t2, $t1, $t6
        bne  $t2, $0, DONE
        slt  $t2, $t7, $t1
        bne  $t2, $0, HEX
        sub  $t1, $t1, $t6
        j DEC
HEX:    slt  $t2, $t1, $t4
        bne  $t2, $0, DONE
        slt  $t2, $t5, $t1
        bne  $t2, $0, DONE
        sub  $t1, $t1, $t4
        addi $t1, $t1, 10
DEC:    beq  $s0, $0, FIRST
        mul  $s0, $s0, 10
FIRST:  add  $s0, $s0, $t1
        addi $t0, $t0, 1
        j LOOP

DONE:   add  $v0, $s0, $0
        lw   $ra, ($sp)
        addi $sp, $sp, 4
        jr   $ra
```

Solution 2.24

2.24.1

a.	0x00000012
b.	0x12ffffff

2.24.2

a.	0x00000080
b.	0x80000000

2.24.3

a.	0x00000011
b.	0x11555555

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 2.26

2.26.1 Branch range is 0x00020000 to 0xFFFE0004.

a.	one branch
b.	three branches

2.26.2

a.	one
b.	can't be done

2.26.3 Branch range is 0x00000200 to 0xFFFFFE04.

a.	eight branches
b.	512 branches

2.26.4

a.	branch range is 16x larger
b.	branch range is 16x smaller

2.26.5

a.	no change
b.	jump to addresses 0 to 2^{12} instead of 0 to 2^{28}, assuming the PC<0x08000000

2.26.6

a.	rs field now 3 bits
b.	no change

Solution 2.27

2.27.1

a.	jump register
b.	beq

2.27.2

a.	R-type
b.	I-type

2.27.3

a.	+ can jump to any 32b address – need to load a register with a 32b address, which could take multiple cycles
b.	+ allows the PC to be set to the current PC + 4 +/– BranchAddr, supporting quick forward and backward branches – range of branches is smaller than large programs

2.27.4

a.	0x00000000 0x00000004	lui $s0, 100 ori $s0, $s0, 40	0x3c100100 0x36100028
b.	0x00000100 0x00000104	addi $t0, $0, 0x0000 lw $t1, 0x4000($t0)	0x20080000 0x8d094000

2.27.5

a.	`addi  $s0, $zero, 0x80`
	`sll   $s0, $s0, 17`
	`ori   $s0, $s0, 40`

b.	`addi  $t0, $0, 0x0040`
	`sll   $t0, $t0, 8`
	`lw    $t1, 0($t0)`

2.27.6

a.	1
b.	1

Solution 2.28

2.28.1

a.	4 instructions

2.28.2

a.	One of the locations specified by the LL instruction has no corresponding SC instruction.

2.28.3

a.	`try:  MOV     R3,R4`
	`       MOV     R6,R7`
	`       LL      R2,0(R2)`
	`       # adjustment or test code here`
	`       SC      R3,0(R2)`
	`       BEQZ    R3,try`
	`try2:`
	`       LL      R5,0(R1)`
	`       # adjustment or test code here`
	`       SC      R6,0(R1)`
	`       BEQZ    R6,try2`
	`       MOV     R4,R2`
	`       MOV     R7,R5`

2.28.4

a.

Processor 1	Processor 2	Cycle	Processor 1		Mem	Processor 2	
			$t1	$t0	($s1)	$t1	$t0
		0	1	2	99	30	40
ll $t1, 0($s1)	ll $t1, 0($s1)	1	99	2	99	99	40
sc $t0, 0($s1)		2	99	1	2	99	40
	sc $t0, 0($s1)	3	99	1	2	99	0

b.

Processor 1	Processor 2	Cycle	Processor 1			Mem	Processor 2		
			$s4	$t1	$t0	($s1)	$s4	$t1	$t0
		0	2	3	4	99	10	20	30
	try: add $t0, $0, $s4	1	2	3	4	99	10	20	10
try: add $t0, $0, $s4	ll $t1, 0($s1)	2	2	3	2	99	10	99	10
ll $t1, 0($s1)		3	2	99	2	99	10	99	10
sc $t0, 0($s1)		4	2	99	1	2	10	99	10
beqz $t0, try	sc $t0, 0($s1)	5	2	99	1	2	10	99	0
add $s4, $0, $t1	beqz $t0, try	6	99	99	1	2	10	99	0

Solution 2.29

2.29.1 The critical section can be implemented as:

```
trylk: li   $t1,1
       ll   $t0,0($a0)
       bnez $t0,trylk
       sc   $t1,0($a0)
       beqz $t1,trylk

       operation

       sw   $zero,0($a0)
```

Where operation is implemented as:

a.

```
       lw   $t0,0($a1)
       add  $t0,$t0,$a2
       sw   $t0,0($a1)
```

b.

```
       lw   $t0,0($a1)
       sge  $t1,$t0,$a2
       bnez $t1,skip
       sw   $a2,0($a1)
skip:
```

2.29.2 The entire critical section is now:

a.
```
try:  ll   $t0,0($a1)
      add  $t0,$t0,$a2
      sc   $t0,0($a1)
      beqz $t0,try
```

b.
```
try:  ll   $t0,0($a1)
      sge  $t1,$t0,$a2
      bnez $t1,skip
      mov  $t0,$a2
      sc   $t0,0($a1)
      beqz $t0,try
skip:
```

2.29.3 The code that directly uses `ll/sc` to update `shvar` avoids the entire lock/unlock code. When SC is executed, this code needs 1) one extra instruction to check the outcome of SC, and 2) if the register used for SC is needed again we need an instruction to copy its value. However, these two additional instructions may not be needed, e.g., if SC is not on the best-case path or f it uses a register whose value is no longer needed. We have:

	Lock-based	Direct LL/SC implementation
a.	6+3	4
b.	6+3	3

2.29.4

a.	Both processors attempt to execute SC at the same time, but one of them completes the write first. The other's SC detects this and its SC operation fails.
b.	It is possible for one or both processors to complete this code without ever reaching the SC instruction. If only one executes SC, it completes successfully. If both reach SC, they do so in the same cycle, but one SC completes first and then the other detects this and fails.

2.29.5 Every processor has a different set of registers, so a value in a register cannot be shared. Therefore, shared variable `shvar` must be kept in memory, loaded each time their value is needed, and stored each time a task wants to change the value of a shared variable. For local variable x there is no such restriction. On the contrary, we want to minimize the time spent in the critical section (or between the LL and SC, so if variable x is in memory it should be loaded to a register before the critical section to avoid loading it during the critical section.

2.29.6 If we simply do two instances of the code from 2.29.2 one after the other (to update one shared variable and then the other), each update is performed atomically, but the entire two-variable update is not atomic, i.e., after the update to the first variable and before the update to the second variable, another process can perform its own update of one or both variables. If we attempt to do two LLs

(one for each variable), compute their new values, and then do two SC instructions (again, one for each variable), the second LL causes the SC that corresponds to the first LL to fail (we have a LL and SC with a non-register-register instruction executed between them). As a result, this code can never successfully complete.

Solution 2.30

2.30.1

a.	`add $t1, $t2, $0`
b.	`add $t0, $0, small` `beq $t1, $t0, LOOP`

2.30.2

a.	Yes. The address of v is not known until the data segment is built at link time.
b.	No. The branch displacement does not depend on the placement of the instruction in the text segment.

Solution 2.31

2.31.1

a.

	Text Size	0x440
	Data Size	0x90
Text	Address	Instruction
	0x00400000	`lw $a0, 0x8000($gp)`
	0x00400004	`jal 0x0400140`
	…	…
	0x00400140	`sw $a1, 0x8040($gp)`
	0x00400144	`jal 0x0400000`
	…	…
Data	0x10000000	(X)
	…	…
	0x10000040	(Y)

b.

	Text Size	0x440
	Data Size	0x90
Text	Address	Instruction
	0x00400000	lui $at, 0x1000
	0x00400004	ori $a0, $at, 0
	0x00400008	jal 0x0400140
	…	…
	0x00400140	sw $a0, 8040($gp)
	0x00400144	jmp 0x04002C0
	…	…
	0x004002C0	jr $ra
	…	…
Data	0x10000000	(X)
	…	…
	0x10000040	(Y)

2.31.2 0x8000 data, 0xFC00000 text. However, because of the size of the `beq` immediate field, 218 words is a more practical program limitation.

2.31.3 The limitation on the sizes of the displacement and address fields in the instruction encoding may make it impossible to use branch and jump instructions for objects that are linked too far apart.

Solution 2.32

2.32.1

a.

```
swap:
sll     $t0,$a1,2
add     $t0,$t0,$a0
lw      $t2,0($t0)
sll     $t1,$a2,2
add     $t1,$t1,$a0
lw      $t3,0($t1)
sw      $t3,0($t0)
sw      $t2,0($t1)
jr      $ra
```

b.

```
swap:
lw      $t0,0($a0)
lw      $t1,4($a0)
sw      $t1,0($a0)
sw      $t0,4($a0)
jr      $ra
```

2.32.2

a.	Pass j+1 as a third parameter to swap. We can do this by adding an "addi $a2,$a1,1" instruction right before "jal swap".
b.	Pass the address of v[j] to swap. Since that address is already in $t2 at the point when we want to call swap, we can replace the two parameter-passing instructions before "jal swap" with a simple "mov $a0,$t2".

2.32.3

a.

```
swap:
add     $t0,$t0,$a0 ; No sll
lb      $t2,0($t0)  ; Byte-sized load
add     $t1,$t1,$a0 ; No sll
lb      $t3,0($t1)
sb      $t3,0($t0)  ; Byte-sized store
sb      $t2,0($t1)
jr      $ra
```

b.

```
swap:
lb      $t0,0($a0)  ; Byte-sized load
lb      $t1,1($a0)  ; Offset is 1, not 4
sb      $t1,0($a0)  ; Byte-sized store
sb      $t0,1($a0)
jr      $ra
```

2.32.4

a.	Yes, we must save the additional s-registers. Also, the code for sort() in Figure 2.27 is using 5 t-registers and only 4 s-registers remain. Fortunately, we can easily reduce this number, e.g., by using t1 instead of t0 for loop comparisons.
b.	No change to saving/restoring code is needed because the same s-registers are used in the modified sort() code.

2.32.5 When the array is already sorted, the inner loop always exits in its first iteration, as soon as it compares v[j] with v[j+1]. We have:

a.	We need 4 more instructions to save and 4 more to restore registers. The number of instructions in the rest of the code is the same, so there are exactly 8 more instructions executed in the modified sort(), regardless of how large the array is.
b.	One fewer instruction is executed in each iteration of the inner loop. Because the array is already sorted, the inner loop always exits during its first iteration, so we save one instruction per iteration of the outer loop. Overall, we execute 10 instructions fewer.

2.32.6 When the array is sorted in reverse order, the inner loop always executes the maximum number of iterations and swap is called in each iteration of the inner loop (a total of 45 times). We have:

a.	This change only affects the number of instructions needed to save/restore registers in swap(), so the answer is the same as in Problem When the array is already sorted, the inner loop always exits in its first iteration, as soon as it compares v[j] with v[j+1]. We have:.

b.	One fewer instruction is executed each time the "j>=0" condition for the inner loop is checked. This condition is checked a total of 55 times (whenever swap is called, plus a total of 10 times to exit the inner loop once in each iteration of the outer loop), so we execute 55 instructions fewer.

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 2.34

2.34.1

a.	`addi $s0, $0, 10`
	`LOOP: add $s0, $s0, $s1`
	`addi $s0, $s0, -1`
	`bne $s0, $0, LOOP`

b.	`sll $s1, $s2, 28`
	`srl $s2, $s2, 4`
	`or $s1, $s1, $s2`

2.34.2

a.	ADD, SUBS, MOV—all ARM register-register instruction format BNE—an ARM branch instruction format
b.	ROR—an ARM register-register instruction format

2.34.3

a.	`CMP r0, r1` `BMI FARAWAY`
b.	`ADD r0, r1, r2`

2.34.4

a.	CMP—an ARM register-register instruction format BMI—an ARM branch instruction format
b.	ADD—an ARM register-register instruction format

Solution 2.35

2.35.1

a.	register operand
b.	register + offset and update register

2.35.2

a.	`lw $s0, ($s1)`

b.	`lw $s1, ($s0)`
	`lw $s2, 4($s0)`
	`lw $s3, 8($s0)`

2.35.3

a.	`addi $s0, $0, TABLE1`
	`addi $s1, $0, 100`
	`xor  $s2, $s2, $s2`
	`ADDLP: lw   $s4, ($s0)`
	`addi $s2, $s2, 4`
	`addi $s0, $s0, 4`
	`addi $s1, $s1, -1`
	`bne  $s1, $0, ADDLP`

b.	`sll  $s1, $s2, 28`
	`srl  $s2, $s2, 4`
	`or   $s1, $s1, $s2`

2.35.4

a.	8 ARM vs. 8 MIPS instructions
b.	1 ARM vs. 3 MIPS instructions

2.35.5

a.	ARM 0.67 times as fast as MIPs
b.	ARM 2 times as fast as MIPs

Solution 2.36

2.36.1

a.	`sll  $s1, $s1, 3` `add  $s3, $s2, $s1`

b.	`sll  $s4, $s1, 29`
	`srl  $s1, $s1, 3`
	`or   $s1, $s1, $s4`
	`add  $s3, $s2, $s1`

2.36.2

a.	`addi $s3, $s2, 64`
b.	`addi $s3, $s2, 64`

2.36.3

a.	`sll  $s1, $s1, 3` `add  $s3, $s2, $s1`

b.	`sll  $s4, $s1, 29`
	`srl  $s1, $s1, 3`
	`or   $s1, $s1, $s4`
	`add  $s3, $s2, $s1`

2.36.4

a.	`add  r3, r2, #1`
b.	`add  r3, r2, 0x8000`

Solution 2.37

2.37.1

a.	`mov edx, [esi+4*ebx]`	`edx=memory(esi+4*ebx)`

b.	`START: mov  ax, 00101100b`
	`mov  cx, 00000011b`
	`mov  bx, 11110000b`
	`and  ax, bx`
	`or   ax, cx`
		`char ax = 00101100b;`
		`char bx = 11110000b;`
		`char cx = 00000011b;`
		`ax = ax && bx;`
		`ax = ax || cx;`

2.37.2

a.	`sll  $s2, $s2, 2`
	`add  $s4, $s4, $s2`
	`lw   $s3, ($s4)`

b.	`START: addi $s0, $0, 0x2c`
	`addi $s2, $0, 0x03`
	`addi $s1, $0, 0xf0`
	`and  $s0, $s0, $s1`
	`or   $s0, $s0, $s2`

2.37.3

a.	`mov edx, [esi+4*ebx]`	6, 1, 1, 8, 8
b.	`add  eax, 0x12345678`	4, 4, 1, 32

2.37.4

a.	`addi $t0, $0, 2`
	`sll $a0, $a0, $t0`
	`add $a0, $a0, $a1`
	`lw  $v0, 0($a0)`

b.	`lui $a0, 0x1234` `ori $a0, 0x5678`

Solution 2.38

2.38.1

a.	This instruction copies ECX bytes from an array pointed to by ESI to an array pointer by EDI. An example C library function that can easily be implemented using this instruction is memcpy.
b.	This instruction copies ECX elements, where each element is 4 bytes in size, from an array pointed to by ESI to an array pointer by EDI.

2.38.2

a.	`loop: lb $t0,0($a2)`
	`sb $t0,0($a1)`
	`addi $a0,$a0,-1`
	`addi $a1,$a1,1`
	`addi $a2,$a2,1`
	`bnez $a0,loop`

b.	`loop: lw $t0,0($a2)`
	`sw $t0,0($a1)`
	`addi $a0,$a0,-1`
	`addi $a1,$a1,4`
	`addi $a2,$a2,4`
	`bnez $a0,loop`

2.38.3

	x86	MIPS	Speed-up
a.	5	6	1.2
b.	5	6	1.2

2.38.4

	MIPS code	Code size comparison
a.	`f: add $v0,$a0,$a1` `jr $ra`	MIPS: 2 × 4 = 8 bytes x86: 11 bytes

b.	`f: lw $t0,0($a0)`
	`lw $t1,0($a1)`
	`add $t0,$t0,$t1`
	`sw $t0,0($a0)`
	`sw $t0,0($a1)`
	`jr $ra`
		MIPS: 6 × 4 = 24 bytes
		x86: 19 bytes

2.38.5 In MIPS, we fetch the next two consecutive instructions by reading the next 8 bytes from the instruction memory. In x86, we only know where the second instruction begins after we have read and decoded the first one, so it is more difficult to design a processor that executes multiple instructions in parallel.

2.38.6 Under these assumptions, using x86 leads to a significant slowdown (the speed-up is well below 1):

	MIPS Cycles	x86 Cycles	Speed-up
a.	2	11	0.18
b.	6	19	0.32

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)

new CPU time = 0.75 × old ICa × CPIa × 1.1 × oldCCT
+ oldICls × CPIls × 1.1 × oldCCT
+ oldICb × CPIb × 1.1 × oldCCT

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

Solution 2.40

2.40.1

a.	In the first iteration $t0 is 0 and the lw fetches a[0]. After that $t0 is 1, the lw uses a non-aligned address triggers a bus error.
b.	In the first iteration $t0 and $t1 point to a[0] b[0], so the lw and sw instructions access a[0], b[0], and then a[0] as intended. In the second iteration $t0 and $t1 point to the next byte in a[0] and b[1], respectively, instead of pointing to a[1] and b[1]. Thus the first lw uses a non-aligned address and causes a bus error. Note that the computation for $t2 (address of a[n]) does not cause a bus error because that address is not actually used to access memory.

2.40.2

a.	Yes, assuming that x is a sign-extended byte value between –128 and 127. If x is simply a byte value between 0 and 255, the function procedure only works if neither x nor array a contain values outside the range of 0..127.
b.	Yes.

2.40.3

a.

```
f:  move $v0,$zero
    move $t0,$zero
L:  sll  $t1,$t0,2     ; We must multiply the index by 4 before we
    add  $t1,$t1,$a0   ; add it to a[] to form the address for lw
    lw   $t1,0($t1)
    bne  $t1,$a2,S
    addi $v0,$v0,1
S:  addi $t0,$t0,1
    bne  $t0,$a1,L
    jr   $ra
```

b.

```
f: move $t0,$a0
   move $t1,$a1
   sll  $t2,$a2,2      ; We must multiply n by 4 to get the address
   add  $t2,$t2,$a0    ; of the end of array a
L: lw   $t3,0($t0)
   lw   $t4,0($t1)
   add  $t3,$t3,$t4
   sw   $t3,0($t0)
   addi $t0,$t0,4      ; Move to next element in a
   addi $t1,$t1,4      ; Move to next element in b
   bne  $t0,$t2,L
   jr   $ra
```

2.40.4 At the exit from `my_alloc`, the $sp register is moved to "free" the memory that is returned to main. Then `my_init()` writes to this memory to initialize it. Note that neither `my_init` nor main access the stack memory in any other way until `sort()` is called, so the values at the point where `sort()` is called are still the same as those written by `my_init`:

a.	0, 0, 0, 0, 0
b.	5, 4, 3, 2, 1

2.40.5 In main, register $s0 becomes 5, then `my_alloc` is called. The address of the array `v` "allocated" by `my_alloc` is 0xffe8, because in `my_alloc` $sp was saved at 0xfffc, and then 20 bytes (4 × 5) were reserved for array arr ($sp was decremented by 20 to yield 0xffe8). The elements of array `v` returned to main are thus a[0] at 0xffe8, a[1] at 0xffec, a[2] at 0xfff0, a[3] at 0xfff4, and a[4] at 0xfff8. After `my_alloc` returns, $sp is back to 0x10000. The value returned from `my_alloc` is 0xffe8 and this address is placed into the $s1 register. The `my_init` function does not modify $sp, $s0, $s1, $s2, or $s3. When `sort()` begins to execute, $sp is 0x1000, $s0 is 5, $s1 is 0xffe7, and $s2 and $s3 keep their original values of −10 and 1, respectively. The sort (0 procedure then changes $sp to 0xffec (0x1000 minus 20), and writes $s0 to memory at address 0xffec (this is where a[1] is, so a[1] becomes 5), writes $s1 to memory at address 0xfff0 (this is where a[2] is, so a[2] becomes 0xffe8), writes $s2 to memory address 0xfff4 (this is where a[3] is, so a[3] becomes −10), writes $s3 to memory address 0xfff8 (this is where a[4] is, so a[4] becomes 1), and writes the return address to 0xfffc, which does not affect values in array `v`. Now the values of array `v` are:

a.	`0 5 0xffe8 7 1`
b.	`5 5 0xffe8 7 1`

2.40.6 When the `sort()` procedure enters its main loop, the elements of array `v` are sorted without any interference from other stack accesses. The resulting sorted array is

a.	`0, 1, 5, 7, 0xffe8`
b.	`1, 5, 5, 7, 0xffe8`

Unfortunately, this is not the end of the chaos caused by the original bug in `my_alloc`. When the `sort()` function begins restoring registers, $ra is read the (luckily) unmodified location where it was saved. Then $s0 is read from memory at address 0xffec (this is where a[1] is), $s1 is read from address 0xfff0 (this is where a[2] is), $s2 is read from address 0xfff4 (this is where a[3] is), and $s3 is read from address 0xfff8 (this is where a[4] is). When `sort()` returns to `main()`, registers $s0 and $s1 are supposed to keep n and the address of array `v`. As a result, after `sort()` returns to `main()`, n and `v` are:

a.	n=1, v=5 So v is a 1-element array of integers that begins at address 5
b.	n=5, v=5 So v is a 5-element array of integers that begins at address 5

If we were to actually attempt to access (e.g., print out) elements of array `v` in the `main()` function after this point, the first lw would result in a bus error due to non-aligned address. If MIPS were to tolerate non-aligned accesses, we would print out whatever values were at the address `v` points to (note that this is not the same address to which `my_init` wrote its values).