2 Solutions

Solution 2.1

2.1.1

a.	`sub  f, g, h`
b.	`addi f, h, -5  (note, no subi)` `add  f, f, g`

2.1.2

a.	1
b.	2

2.1.3

a.	−1
b.	0

2.1.4

a.	`f = f + 4`
b.	`f = g + h + i`

2.1.5

a.	5
b.	9

Solution 2.2

2.2.1

a.	`sub  f, g, f`
b.	`addi f, h, -2  (note no subi)` `add  f, f, i`

2.2.2

a.	1
b.	2

2.2.3

a.	1
b.	2

2.2.4

a.	`f += 4;`
b.	`f = i – (g + h);`

2.2.5

a.	5
b.	–1

Solution 2.3

2.3.1

a.	`sub  f, $0, f` `sub  f, f,  g`

b.	`sub  f, $0, f`
	`addi f, f, –5  (note, no subi)`
	`add  f, f,  g`

2.3.2

a.	2
b.	3

2.3.3

a.	–3
b.	–3

2.3.4

a.	`f += -4`
b.	`f += (g + h);`

2.3.5

a.	−3
b.	6

Solution 2.4

2.4.1

a.	`lw  $s0, 16($s6)`
	`sub $s0, $0,  $s0`
	`sub $s0, $s0, $s1`

b.	`sub $t0, $s3, $s4`
	`add $t0, $s6, $t0`
	`lw  $t1, 16($t0)`
	`sw  $t1, 32($s7)`

2.4.2

a.	3
b.	4

2.4.3

a.	3
b.	6

2.4.4

a.	`f = 2j + i + g;`
b.	`B[g] = A[f] + A[1+f];`

2.4.5

a.	`slli $s2, $s4, 1`
	`add  $s0, $s2, $s3`
	`add  $s0, $s0, $s1`

b.	`add  $t0, $s6, $s0`
	`add  $t1, $s7, $s1`
	`lw   $s0, 0($t0)`
	`lw   $t0, 4($t0)`
	`add  $t0, $t0, $s0`
	`sw   $t0, 0($t1)`

2.4.6

a.	5 as written, 5 minimally
b.	7 as written, 6 minimally

Solution 2.5

2.5.1

a.	Address
	20
	24
	28
	32
	34
		Data
		4
		5
		3
		2
		1
			`temp = Array[0];`
			`temp2 = Array[1];`
			`Array[0] = Array[4];`
			`Array[1] = Array[3];`
			`Array[3] = temp;`
			`Array[4] = temp2;`

b.	Address
	24
	38
	32
	36
	40
		Data
		2
		4
		3
		6
		1
			`temp = Array[0];`
			`temp2 = Array[1];`
			`Array[0] = Array[4];`
			`Array[1] = temp;`
			`Array[4] = Array[3];`
			`Array[3] = temp2;`

2.5.2

a.	Address
	20
	24
	28
	32
	34
		Data
		4
		5
		3
		2
		1
			`temp = Array[0];`
			`temp2 = Array[1];`
			`Array[0] = Array[4];`
			`Array[1] = Array[3];`
			`Array[3] = temp;`
			`Array[4] = temp2;`
				`lw   $t0, 0($s6)`
				`lw   $t1, 4($s6)`
				`lw   $t2, 16($s6)`
				`sw   $t2, 0($s6)`
				`lw   $t2, 12($s6)`
				`sw   $t2, 4($s6)`
				`sw   $t0, 12($s6)`
				`sw   $t1, 16($s6)`

b.	Address Data
	24 2
	38 4
	32 3
	36 6
	40 1
		`temp = Array[0];`
		`temp2 = Array[1];`
		`Array[0] = Array[4];`
		`Array[1] = temp;`
		`Array[4] = Array[3];`
		`Array[3] = temp2;`
			`lw $t0, 0($s6)`
			`lw $t1, 4($s6)`
			`lw $t2, 16($s6)`
			`sw $t2, 0($s6)`
			`sw $t0, 4($s6)`
			`lw $t0, 12($s6)`
			`sw $t0, 16($s6)`
			
			`sw $t1, 12($s6)`

2.5.3

a.	Address Data
	20 4
	24 5
	28 3
	32 2
	34 1
		`temp = Array[1];`
		`Array[1] = Array[5];`
		`Array[5] = temp;`
		`temp = Array[2];`
		`Array[2] = Array[4];`
		`temp2 = Array[3];`
		`Array[3] = temp;`
		`Array[4] = temp2;`
			`lw $t0, 0($s6)`
			`lw $t1, 4($s6)`
			`lw $t2, 16($s6)`
			`sw $t2, 0($s6)`
			`lw $t2, 12($s6)`
			`sw $t2, 4($s6)`
			`sw $t0, 12($s6)`
			
			`sw $t1, 16($s6)`
				8 MIPS instructions, +1 MIPS inst. for every non-zero offset lw/sw pair (11 MIPS inst.)

b.	Address Data
	24 2
	38 4
	32 3
	36 6
	40 1
		`temp = Array[3];`
		`Array[3] = Array[2];`
		`Array[2] = Array[1];`
		`Array[1] = Array[0];`
		`Array[0] = temp;`
			`lw $t0, 0($s6)`
			`lw $t1, 4($s6)`
			`lw $t2, 16($s6)`
			`sw $t2, 0($s6)`
			`sw $t0, 4($s6)`
			`lw $t0, 12($s6)`
			`sw $t0, 16($s6)`
			
			`sw $t1, 12($s6)`
				8 MIPS instructions, +1 MIPS inst. for every non-zero offset lw/sw pair (11 MIPS inst.)

2.5.4

a.	2882400018
b.	270544960

2.5.5

	Little-Endian		Big-Endian	
a.	Address 12 8 4 0	Data ab cd ef 12	Address 12 8 4 0	Data 12 ef cf ab
b.	Address 12 8 4 0	Data 10 20 30 40	Address 12 8 4 0	Data 40 30 20 10

Solution 2.6

2.6.1

a.

```
lw      $t0, 4($s7)             #  $t0 <-- B[1]
sub     $t0, $t0, $s1           #  $t0 <-- B[1] – g
add     $s0, $t0, $s2           #  f <-- B[1] –g + h
```

b.

```
sll     $t0, $s1, 2             #  $t0 <-- 4*g
add     $t0, $t0, $s7           #  $t0 <-- Addr(B[g])
lw      $t0, 0($t0)             #  $t0 <-- B[g]
addi    $t0, $t0, 1             #  $t0 <-- B[g]+1
sll     $t0, $t0, 2             #  $t0 <-- 4*(B[g]+1) = Addr(A[B[g]+1])
lw      $s0, 0($t0)             #  f <-- A[B[g]+1]
```

2.6.2

a.	3
b.	6

2.6.3

a.	5
b.	4

2.6.4

a.	`f = f - i;`
b.	`f = 2 * (&A);`

2.6.5

a.	$s0 = –30
b.	$s0 = 512

2.6.6

a.

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

b.

	Type	opcode	rs	rt	rd	immed
`addi $t0, $s6, 4`	I-type	8	22	8		4
`add $t1, $s6, $0`	R-type	0	22	0	9	
`sw $t1, 0($t0)`	I-type	43	8	9		0
`lw $t0, 0($t0)`	I-type	35	8	8		0
`add $s0, $t1, $t0`	R-type	0	9	8	16	

Solution 2.7

2.7.1

a.	613566756
b.	1606303744

2.7.2

a.	613566756
b.	1606303744

2.7.3

a.	24924924
b.	5FBE4000

2.7.4

a.	11111111111111111111111111111111
b.	10000000000

2.7.5

a.	FFFFFFFF
b.	400

2.7.6

a.	1
b.	FFFFFC00

Solution 2.8

2.8.1

a.	50000000, overflow
b.	0, no overflow

2.8.2

a.	B0000000, no overflow
b.	2, no overflow

2.8.3

a.	D0000000, overflow
b.	000000001, no overflow

2.8.4

a.	overflow
b.	overflow

2.8.5

a.	overflow
b.	overflow

2.8.6

a.	overflow
b.	overflow

Solution 2.9

2.9.1

a.	no overflow
b.	overflow

2.9.2

a.	no overflow
b.	no overflow

2.9.3

a.	no overflow
b.	no overflow

2.9.4

a.	overflow
b.	overflow

2.9.5

a.	94924924
b.	CFBE4000

2.9.6

a.	2492614948
b.	–809615360

Solution 2.10

2.10.1

a.	`add  $s0, $s0, $s0`
b.	`sub  $t1, $t2, $t3`

2.10.2

a.	r-type
b.	r-type

2.10.3

a.	2108020
b.	14B4822

2.10.4

a.	0x21080001
b.	0xAD490020

2.10.5

a.	i-type
b.	i-type

2.10.6

a.	op=0x8, rs=0x8, rs=0x8, imm=0x0
b.	op=0x2B, rs=0xA, rt=0x9, imm=0x20

Solution 2.11

2.11.1

a.	$0000\ 0001\ 0000\ 1000\ 0100\ 0000\ 0010\ 0000_{two}$
b.	$0000\ 0010\ 0101\ 0011\ 1000\ 1000\ 0010\ 0010_{two}$

2.11.2

a.	17317920
b.	39028770

2.11.3

a.	add $t0, $t0, $t0
b.	sub $s1, $s2, $s3

2.11.4

a.	r-type
b.	i-type

2.11.5

a.	`sub  $v1, $v1, $v0`
b.	`lw  $v0, 4($at)`

2.11.6

a.	0x00621822
b.	0x8C220004

Solution 2.12

2.12.1

	Type	opcode	rs	rt	rd	shamt	funct	
a.	r-type	6	7	7	7	5	6	total bits = 38
b.	r-type	8	5	5	5	5	6	total bits = 34

2.12.2

	Type	opcode	rs	rt	immed	
a.	i-type	6	7	7	16	total bits = 36
b.	i-type	8	5	5	16	total bits = 34

2.12.3

a.	more registers → more bits per instruction → could increase code size more registers → less register spills → less instructions
b.	more instructions → more appropriate instruction → decrease code size more instructions → larger opcodes → larger code size

2.12.4

a.	17367058
b.	2903048210

2.12.5

a.	`sub  $t0, $t1, $0`
b.	`sw  $t1, 12($t0)`

2.12.6

a.	r-type, op=0x0, rt=0x9
b.	i-type, op=0x2B, rt=0x8

Solution 2.13

2.13.1

a.	0xBABEFEF8
b.	0x11D111D1

2.13.2

a.	0xAAAAAAA0
b.	0x00DD00D0

2.13.3

a.	0x00005545
b.	0x0000BA01

2.13.4

a.	0x00014B4A
b.	0x00000001

2.13.5

a.	0x4b4a0000
b.	0x00000000

2.13.6

a.	0x4b4bfffe
b.	0x0000003C

Solution 2.14

2.14.1

a.
```
lui   $t1, 0x003f
ori   $t1, $t0, 0xffe0
and   $t1, $t0, $t1
srl   $t1, $t1, 5
```

b.
```
lui   $t1, 0x003f
ori   $t1, $t0, 0xffe0
and   $t1, $t0, $t1
sll   $t1, $t1, 9
```

2.14.2

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

b.
```
andi  $t0, $t0, 0x000f
sll   $t0, $t0, 14
ori   $t1, $t1, 0x3fff
sll   $t1, $t1, 18
ori   $t1, $t1, 0x3fff
or    $t1, $t1, $t0
```

2.14.3

a.
```
srl   $t1, $t0, 28
sll   $t1, $t1, 29
```

b.
```
srl   $t0, $t0, 28
andi  $t0, $t0, 0x0007
sll   $t0, $t0, 14
ori   $t1, $t1, 0x7fff
sll   $t1, $t1, 17
ori   $t1, $t1, 0x3fff
or    $t1, $t1, $t0
```

2.14.4

a.
```
srl   $t0, $t0, 11
sll   $t0, $t0, 26
ori   $t2, $0,  0x03ff
sll   $t2, $t2, 16
ori   $t2, $t2, 0xffff
and   $t1, $t1, $t2
or    $t1, $t1, $t0
```

b.
```
srl   $t0, $t0, 11
sll   $t0, $t0, 26
srl   $t0, $t0, 12
ori   $t2, $0,  0xfff0
sll   $t2, $t2, 16
ori   $t2, $t2, 0x3fff
and   $t1, $t1, $t2
or    $t1, $t1, $t0
```

2.14.5

a.	`sll  $t0, $t0, 27`
	`ori  $t2, $0,  0x07ff`
	`sll  $t2, $t2, 16`
	`ori  $t2, $t2, 0xffff`
	`and  $t1, $t1, $t2`
	`or  $t1, $t1, $t0`

b.	`sll  $t0, $t0, 27`
	`srl  $t0, $t0, 13`
	`ori  $t2, $0,  0xfff8`
	`sll  $t2, $t2, 16`
	`ori  $t2, $t2, 0x3fff`
	`and  $t1, $t1, $t2`
	`or  $t1, $t1, $t0`

2.14.6

a.	`srl  $t0, $t0, 29`
	`sll  $t0, $t0, 30`
	`ori  $t2, $0,  0x3fff`
	`sll  $t2, $t2, 16`
	`ori  $t2, $t2, 0xffff`
	`and  $t1, $t1, $t2`
	`or  $t1, $t1, $t0`

b.	`srl  $t0, $t0, 29`
	`sll  $t0, $t0, 30`
	`srl  $t0, $t0, 16`
	`ori  $t2, $0,  0xffff`
	`sll  $t2, $t2, 16`
	`ori  $t2, $t2, 0x3fff`
	`and  $t1, $t1, $t2`
	`or  $t1, $t1, $t0`

Solution 2.15

2.15.1

a.	`0xff005a5a`
b.	`0x00ffffe7`

2.15.2

a.	`nor  $t1, $t2, $t2`
b.	`nor  $t1, $t3, $t3` `or  $t1, $t2, $t1`

2.15.3

a.	`nor  $t1, $t2, $t2`	`000000 01010 01010 01001 00000 100111`
b.	`nor  $t1, $t3, $t3` `or  $t1, $t2, $t1`	`000000 01011 01011 01001 00000 100111` `000000 01010 01001 01001 00000 100101`

2.15.4

a.	0xFFFFFFFF
b.	0x00012340

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

a.	`nor $t3, $t1, $t1` `or $t1, $t2, $t3`
b.	`lw $t3, 0($s1)` `sll $t1, $t3, 4`

2.15.6

a.	`nor $t3, $t1, $t1` `or $t1, $t2, $t3`	000000 01001 01001 01011 00000 100111 000000 01010 01011 01001 00000 100101
b.	`lw $t3, 0($s1)` `sll $t1, $t3, 4`	100011 10001 01011 0000000000000000 000000 00000 01011 01001 00100 000000

Solution 2.16

2.16.1

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

2.16.2

a.	none
b.	none

2.16.3

a.	Jump – No, Beq - No
b.	Jump – No, Beq - No

2.16.4

a.	$t2 = 2
b.	$t2 = 1

2.16.5

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

2.16.6

a.	jump – Yes, beq - no
b.	jump – no, beq - no

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.	i-type
b.	i-type

2.17.3

a.	`addi $t2, $t3, –5`
b.	`addi $t2, $t2, –1` `beq $t2, $0, loop`

2.17.4

a.	20
b.	20

2.17.5

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

b.	Same as part a.

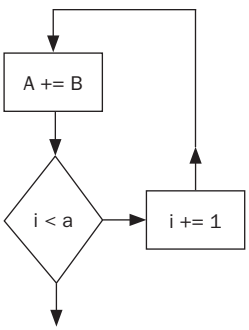
2.17.6

a.	3 × N
b.	5 × N

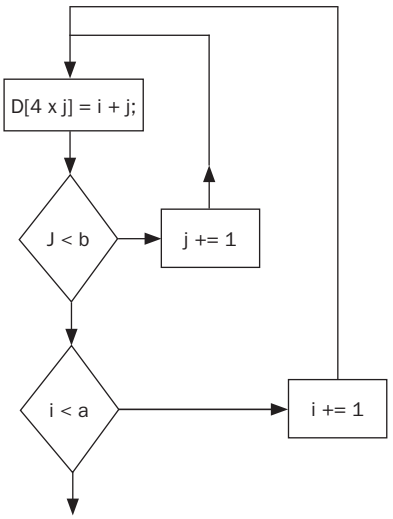
Solution 2.18

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: slt  $t2, $t0, $s0
      bne  $t2, $0, LOOP
```

b.

```
      addi $t0, $0, 0
      beq  $0, $0, TEST1
LOOP1:addi $t1, $0, 0
      beq  $0, $0, TEST2
LOOP2:add  $t3, $t0, $t1
      sll  $t2, $t1, 4
      add  $t2, $t2, $s2
      sw   $t3, ($t2)
      addi $t1, $t1, 1
TEST2:slt  $t2, $t1, $s1
      bne  $t2, $0, LOOP2
      addi $t0, $t0, 1
TEST1:slt  $t2, $t0, $s0
      bne  $t2, $0, LOOP1
```

2.18.3

a.	6 instructions to implement and infinite instructions executed
b.	14 instructions to implement and 158 instructions executed

2.18.4

a.	351
b.	601

2.18.5

a.

```
for(i=50; i>0; i--){
    result += MemArray[s0];
    result += MemArray[s0+1];
    s0 += 2;
}
```

b.

```
for (i=0; i<100; i++) {
    result += MemArray[s0];
    s0 = s0 + 4;
}
```

2.18.6

a.

```
      addi $t1, $s0, 400
LOOP: lw   $s1, 0($s0)
      add  $s2, $s2, $s1
      lw   $s1, 4($s0)
      add  $s2, $s2, $s1
      addi $s0, $s0, 8
      bne  $s0, $t1, LOOP
```

b.

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

Solution 2.19

2.19.1

a.

```
fib:    addi $sp, $sp, -12        # make room on stack
        sw   $ra, 8($sp)          # push $ra
        sw   $s0, 4($sp)          # push $s0
        sw   $a0, 0($sp)          # push $a0 (N)
        bgt  $a0, $0, test2       # if n>0, test if n=1
        add  $v0, $0, $0          # else fib(0) = 0
        j rtn                     #
test2: addi $t0, $0, 1            #
        bne  $t0, $a0, gen        # if n>1, gen
        add  $v0, $0, $t0         # else fib(1) = 1
        j rtn
gen:    subi $a0, $a0,1           # n-1
        jal  fib                  # call fib(n-1)
        add  $s0, $v0, $0         # copy fib(n-1)
        sub  $a0, $a0,1           # n-2
        jal  fib                  # call fib(n-2)
        add  $v0, $v0, $s0        # fib(n-1)+fib(n-2)
rtn:    lw   $a0, 0($sp)          # pop $a0
        lw   $s0, 4($sp)          # pop $s0
        lw   $ra, 8($sp)          # pop $ra
        addi $sp, $sp, 12         # restore sp
        jr   $ra

# fib(0) = 12 instructions, fib(1) = 14 instructions,
# fib(N) = 26 + 18N instructions for N >=2
```

b.

```
positive:
      addi $sp, $sp, -4
      sw   $ra, 0($sp)
      jal  addit
      addi $t1, $0, 1
      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
addit:
      add  $v0, $a0, $a1
      jr   $ra
# 13  instructions worst-case
```

2.19.2

a.

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

b.

```
positive:
      add  $t0, $a0, $a1
      addi $v0, $0, 1
      slt  $t2, $0, $t0
      bne  $t2, $0, exit
      addi $v0, $0, $0
exit:
      jr   $ra

# 6  instructions worst-case
```

2.19.3

a.

```
after calling function fib:
old $sp ->      0x7fffffc       ???
                 -4             contents of register $ra for fib(N)
                 -8             contents of register $s0 for fib(N)
$sp->           -12             contents of register $a0 for fib(N)

there will be N-1 copies of $ra, $s0, and $a0
```

b.

```
after calling function positive:
old $sp ->      0x7fffffc      ???
$sp->           -4             contents of register $ra

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

2.19.4

a.

```
f:  addi    $sp,$sp,-12
    sw      $ra,8($sp)
    sw      $s1,4($sp)
    sw      $s0,0($sp)
    move    $s1,$a2
    move    $s0,$a3
    jal     func
    move    $a0,$v0
    add     $a1,$s0,$s1
    jal     func
    lw      $ra,8($sp)
    lw      $s1,4($sp)
    lw      $s0,0($sp)
    addi    $sp,$sp,12
    jr      $ra
```

b.

```
f:  addi    $sp,$sp,-4
    sw      $ra,0($sp)
    add     $t0,$a1,$a0
    add     $a1,$a3,$a2
    slt     $t1,$a1,$t0
    beqz    $t1,L
    move    $a0,$t0
    jal     func
    lw      $ra,0($sp)
    addi    $sp,$sp,4
    jr      ra
L:  move    $a0,$a1
    move    $a1,$t0
    jal     func
    lw      $ra,0($sp)
    addi    $sp,$sp,4
    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, $s0, $s1, and $sp before that call. We save only one instruction (jr $ra).

b. We can use the tail-call optimization for either call to `func` (when the condition for the if is true or false). This eliminates the need to save $ra and move the stack pointer, so we execute 5 fewer instructions (regardless of whether the if condition is true or not). The code of the function is 8 instructions shorter because we can eliminate both instances of the code that restores $ra and returns.

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 value of $t5 after function func has been called.

Solution 2.20

2.20.1

a.

```
FACT:   addi $sp, $sp, -8  # make room in stack for 2 more items
        sw   $ra, 4($sp)   # save the return address
        sw   $a0, 0($sp)   # save the argument n
        slti $t0, $a0, 1   # $t0 = $a0 x 2
        beq, $t0, $0, L1   # if $t0 = 0, goto L1
        add  $v0, $0, 1    # return 1
        add  $sp, $sp, 8   # pop two items from the stack
        jr   $ra           # return to the instruction after jal
L1:     addi $a0, $a0, -1  # subtract 1 from argument
        jal  FACT          # call fact(n-1)
        lw   $a0, 0($sp)   # just returned from jal: restore n
        lw   $ra, 4($sp)   # restore the return address
        add  $sp, $sp, 8   # pop two items from the stack
        mul  $v0, $a0, $v0 # return n*fact(n-1)
        jr   $ra           # return to the caller
```

b.

```
FACT:  addi $sp, $sp, -8     # make room in stack for 2 more items
       sw   $ra, 4($sp)      # save the return address
       sw   $a0, 0($sp)      # save the argument n
       slti $t0, $a0, 1      # $t0 = $a0 x 2
       beq, $t0, $0, L1      # if $t0 = 0, goto L1
       add  $v0, $0, 1       # return 1
       add  $sp, $sp, 8      # pop two items from the stack
       jr   $ra              # return to the instruction after jal
L1:    addi $a0, $a0, -1     # subtract 1 from argument
       jal  FACT             # call fact(n-1)
       lw   $a0, 0($sp)      # just returned from jal: restore n
       lw   $ra, 4($sp)      # restore the return address
       add  $sp, $sp, 8      # pop two items from the stack
       mul  $v0, $a0, $v0    # return n*fact(n-1)
       jr   $ra              # return to the caller
```

2.20.2

a. 25 MIPS instructions to execute non-recursive vs. 45 instructions to execute (corrected version of) recursion

Non-recursive 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 non-recursive vs. 45 instructions to execute (corrected version of) recursion

Non-recursive 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 non-recursive vs. 73 instructions to execute (corrected version of) recursion

Non-recursive 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 non-recursive vs. 73 instructions to execute (corrected version of) recursion

Non-recursive 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.

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

        addi $a0, $0, 10
        addi $a1, $0, 20
        jal  FUNC
        add  $t2, $v0 $0

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

FUNC:   lw   $t1, ($s0)   #assume $s0 has global variable base
        sub  $t0, $v0, $v1
        addi $v0, $t0, $t1
        jr   $ra
```

b.

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

        lw   $t1, ($s0)   #assume $s0 has global variable base
        addi $a0, $t1, 1
        jal  LEAF
        add  $t2, $v0 $0

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

LEAF:   addi  $v0, $a0, 1
        jr   $ra
```

2.21.2

a.

```
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
```

b.

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

global pointers:
0x10008000      100           my_global

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

global pointers:
0x10008000      101           my_global
```

2.21.3

a.

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

        addi $a0, $0, 10
        addi $a1, $0, 20
        jal  FUNC
        add  $t2, $v0 $0

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

FUNC:   lw   $t1, ($s0)   #assume $s0 has global variable base
        sub  $t0, $v0, $v1
        addi $v0, $t0, $t1
        jr   $ra
```

b.

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

        lw   $t1, ($s0)   #assume $s0 has global variable base
        addi $a0, $t1, 1
        jal  LEAF
        add  $t2, $v0 $0

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

LEAF:   addi  $v0, $a0, 1
        jr   $ra
```

2.21.4

a.	The return address of the function is in $ra, so the last instruction should be “jr $ra.”
b.	The tail call to g must use jr, not jal. If jal is used, it overwrites the return address so function g returns back to f, not to the original caller of f as intended.

2.21.5

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

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

2.21.6

a.	The function returns 101 (1000 is nonzero, so it returns 1+100).
b.	The function returns 500 (c+d is 1030, which is larger than 1, so the function returns g(b), which according to the problem statement is 500).

Solution 2.22

2.22.1

a.	`68 65 6C 6C 6F 20 77 6F 72 6C 64`
b.	`48 49 50 51 52 53 54 55 56 57`

2.22.2

a.	`U+0038, U+0020, U+0062, U+0069, U+0074, U+0073`
b.	`U+0030, U+0031, U+0032, U+0033, U+0034, U+0035, U+0036, U+0037, U+0038, U+0039`

2.22.3

a.	`ADD`
b.	`MIPS`

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.	0x0012ffff

2.24.2

a.	0x00000080
b.	0x00800000

2.24.3

a.	0x00000011
b.	0x00115555

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.	yes

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

a.	no
b.	no

2.25.4 Range is 0x1FFFF004 + 0x1FFFC = 0x2001F000 to 0x1FFFF004 – 0x20000 = 1FFDF004

a.	yes
b.	no

2.25.5 Generally, all solutions are similar:

```
add  $t1, $0, $0             #clear $t1
addi $t2, $0, 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, $0, 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, $0, 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.	0x00000000

2.25.7

a.	t0 = (0x1234 << 16) + 0x5678;
b.	t0 = (0x1234 << 16) && 0x5678;

Solution 2.26

2.26.1 Branch range is 0x00020000 to 0xFFFE0004.

a.	one branch
b.	one branch

2.26.2

a.	one
b.	can't be done

2.26.3 Branch range is 0x00000200 to 0xFFFFFE04.

a.	256 branches
b.	one branch

2.26.4

a.	branch range is 16× smaller
b.	branch range is 4× smaller

2.26.5

a.	no change
b.	jump to addresses 0 to 2^{26} instead of 0 to 2^{28}, assuming the PC<0x08000000

2.26.6

a.	rs field now 7 bits
b.	no change

Solution 2.27

2.27.1

a.	MIPS lw/sw instructions: lw $t0, 8($t1)
b.	jump

2.27.2

a.	i-type
b.	j-type

2.27.3

a.	+ allows memory from (base +/– 2^{15}) addresses to be loaded without changing the base – max size of 64 kB memory array without having to use multiple base addresses
b.	+ large jump range – jump range not as large as jump-register – can only access 1/16th of the total addressable space

2.27.4

a.	0x00400000 beq $s0, $0, FAR ... 0x00403100 FAR: addi $s0, $s0, 1	0x12000c3c 0x22100001
b.	0x00000100 j AWAY ... 0x04000010 AWAY: addi $s0, $s0, 1	0x09000004 0x22100001

2.27.5

a.

```
        addi  $t0, $0, 0x31
        sll   $t0, $t0, 8
        beq   $s0, $0, TEMP
        ...
TEMP:   jr    $t0
```

b.

```
        addi  $s0, $0, 0x4
        sll   $s0, $s0, 24
        ori   $s0, $s0, 0x10
        jr    $s0
        ...
        addi  $s0, $s0, 1
```

2.27.6

a.	2
b.	3

Solution 2.28

2.28.1

a.	3 instructions

2.28.2

a.	The location specified by the LL instruction is different than the SC instruction; hence, the operation of the store conditional is undefined.

2.28.3

a.

```
try:  MOV     R3,R4
      LL      R2,0(R1)
      ADDI    R2, R2, 1
      SC      R3,0(R1)
      BEQZ    R3,try
      MOV     R4,R2
```

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)	1	1	2	99	99	40
ll $t1, 0($s1)		2	99	2	99	99	40
	sc $t0, 0($s1)	3	99	2	40	99	1
sc $t0, 0($s1)		4	99	0	40	99	1

b.

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)		1	99	2	99	30	40
	ll $t1, 0($s1)	2	99	2	99	99	40
	addi $t1,$t1,1	3	99	2	99	100	40
	sc $t0, 0($s1)	4	99	2	100	100	1
sc $t0, 0($s1)		5	99	0	100	100	1

Solution 2.29

2.29.1 The critical section can be implemented as:

```
comment:  Not sure what this is...

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

       operation

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

Where operation is implemented as:

a.

```
       lw   $t0,0($a1)
       slt  $t1,$t0,$a2
       bne  $t1,$0,skip
       sw   $a2,0($a1)
skip:
```

b.

```
       lw   $t0,0($a1)
       blez $t0,skip
       sle  $t1,$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)
       sle  $t1,$t0,$a2
       bnez $t1,skip
       mov  $t0,$a2
       sc   $t0,0($a1)
       beqz $t0,try
skip:
```

b.
```
try:   ll   $t0,0($a1)
       blez $t0,skip
       sle  $t1,$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 if it uses a register whose value is no longer needed. We have:

	Lock-based	Direct LL/SC implementation
a.	6 + 3	3
b.	6 + 2	2

2.29.4

a.	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.
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 its 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 an LL and a 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 $t0, $0, $0`
b.	`add $t0, $0, large` `beq $t1, $t0, LOOP`

2.30.2

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

Solution 2.31

2.31.1

a.

	Text Size	0x440
	Data Size	0x90
Text	Address	Instruction
	0x00400000	lbu $a0, 8000($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
	…	…
	0x00400140	sw $a0, 8040($gp)
	0x00400144	jmp 0x04002C0
	…	…
	0x004002C0	jal 0x0400000
	…	…
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:
lw      $v0,0($a0)
lw      $v1,0($a1)
sw      $v1,0($a0)
sw      $v0,0($a1)
jr      $ra
```

b.

```
swap:
lw      $t0,0($a0)
lw      $t1,0($a1)
add     $t0,$t0,$t1
sub     $t1,$t0,$t1
sub     $t0,$t0,$t1
sw      $t0,0($a0)
sw      $t1,0($a1)
jr      $ra
```

2.32.2

a.	Pass the address of v[j] and of v[j+1] to swap. Because the address of v[j] 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 "mov $a0,$t2" and "addi $a1,$t2,4."
b.	Pass the address of v[j] and of v[j+1] to swap. Because the address of v[j] 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 "mov $a0,$t2" and "addi $a1,$t2,4."

2.32.3

a.

```
swap:
lb      $v0,0($a0) ; Byte-sized load
lb      $v1,0($a1)
sb      $v1,0($a0) ; Byte-sized store
sb      $v0,0($a1)
jr      $ra
```

b.

```
swap:
lb      $t0,0($a0)  ; Byte-sized load
lb      $t1,0($a1)
add     $t0,$t0,$t1
sub     $t1,$t0,$t1
sub     $t0,$t0,$t1
sb      $t0,0($a0)  ; Byte-sized store
sb      $t1,0($a1)
jr      $ra
```

2.32.4

a.	No change to saving/restoring code is needed because the same s-registers are used in the modified sort() code.
b.	No change. This modification affects array address computation and load/store instructions. We still need to use the same s-registers which need to be saved/restored.

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.	The number of instructions in sort() is unchanged. The swap() function is changed, but it is never executed when sorting an already-sorted array. As a result, we execute exactly the same number of instructions.
b.	The only change in the number of instructions is that sll instructions can be eliminated in both sort() and swap(). When sorting an already-sorted array, swap() is never executed, and the inner loop in sort() always exits during its first iteration, so we save one sll 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.	The number of instructions in sort() is unchanged. However, the swap() function now has only 5 instructions (instead of 7) so we now execute 90 instructions fewer.

b.	One fewer instruction is executed each time v[j] is needed to check the "v[j]>v[j+1]" condition for the inner loop. This happens a total of 45 times. Also, swap() now has one instruction less (no sll is needed), so there we also execute a total of 45 fewer instructions. Overall, we execute 90 instructions fewer.

Solution 2.33

2.33.1

a.

```
copy: move $t0,$0
loop: beq  $t0,$a2,done
      sll  $t1,$t0,2
      add  $t2,$t1,$a1
      lw   $t2,0($t2)
      add  $t1,$t1,$a0
      sw   $t2,0($t1)
      addi $t0,$t0,1
      b    loop
done: jr   $ra
```

b.

```
shift:  move $t0,$0
        addi $t1,$a1,-1
loop:   beq  $t0,$t1,done
        sll  $t2,$t0,2
        add  $t2,$t2,$a0
        lw   $t3,4($t2)
        sw   $t3,0($t2)
        addi $t0,$t0,1
        b    loop
done:   jr $ra
```

2.33.2

a.

```
void copy(int *a, int *b, int n){
  int *p,*q;
  for(p=a,q=b;p!=a+n;p++,q++)
    *p=*q;
}
```

b.

```
void shift(int *a, int n){
  int *p;
  for(p=a;p!=a+n-1;p++)
    *p=*(p+1);
}
```

2.33.3

a.

```
copy: move $t0,$a0
      move $t1,$a1
      sll  $t2,$a2,2
      add  $t2,$t2,$a0
loop: beq  $t0,$t2,done
      lw   $t3,0($t1)
      sw   $t3,0($t0)
      addi $t0,$t0,4
      addi $t1,$t1,4
      b    loop
done: jr   $ra
```

b.

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

2.33.4

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

2.33.5

	Array-based	Pointer-based
a.	3	4
b.	4	3

2.33.6 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.	`add  $s0, $s1, $s2` `# no equivalent to ADC in MIPS`

b.	`addi $t0, $0, 4`
	`beq  $s0, $t0, LABEL`
	`add  $s1, $s1, $s0`

2.34.2

a.	ADD, ADC — both ARM register-register instruction format
b.	CMP, ADDNE — both ARM register-register instruction format

2.34.3

a.	`ORR   r0, 0`
	`NOT   r4, r0`
	`AND   r1, r4`

b.	`ROR   r1, r2, #16`

2.34.4

a.	ORR, NOT, AND — all ARM register-register instruction format
b.	ROR — an ARM register-register instruction format

Solution 2.35

2.35.1

a.	register + offset (displacement or based)
b.	rregister + offset and update register

2.35.2

a.	`addi $s1, $s1, 4` `lw   $s0, 4($s1)`

b.	`lw   $s1, 0($s0)`
	`lw   $s2, 4($s0)`
	`lw   $s3, 8($s0)`
	`addi $s0, $s0, 12`

2.35.3

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

b.	`addu $s0, $s0, $s1 # add lower words`
	`sltu $t0, $s0, $s1 # find sign bit`
	`addu $t0, $t0, $s2 # add sign bit to upper word`
	`addu $s2, $t0, $s3 # add upper words`

2.35.4

a.	4 ARM vs. 4 MIPS instructions
b.	2 ARM vs. 4 MIPS instructions

2.35.5

a.	ARM 0.67 times as fast as MIPS
b.	ARM 1.33 times as fast as MIPS

Solution 2.36

2.36.1

a.	`srl $s1, $s1, 4` `add $s3, $s2, $s1`
b.	`add $s3, $s2, $s1`

2.36.2

a.	`add $s3, $s2, $0`
b.	`addi $s3, $s2, 8`

2.36.3

a.	`srl $s1, $s1, 4` `add $s3, $s2, $s1`
b.	`add $s3, $s2, $s1`

2.36.4

a.	`ADD r3, r2, #2`
b.	`SUBS r3, r2, -1`

Solution 2.37

2.37.1

a.	START: mov eax, 3
	push eax
	mov eax, 4
	mov ecx, 4
	add eax, ecx
	pop ecx
	add eax, ecx
		eax = (4 + 4) + 3

b.	START: mov ecx, 100
	mov eax, 0
	LOOP: add eax, ecx
	dec ecx
	cmp ecx, 0
	jne LOOP
	DONE:
		ebx = 0;
		for (i=100; i>0; i--)
		ebx += i

2.37.2

a.	START: addi $s0, $0, 3
	addi $sp, $sp, −4
	sw $s0, 0($sp)
	addi $s0, $0, 4
	addi $s2, $0, 4
	add $s0, $s0, $s2
	lw $s2, 0($sp)
	addi $sp, $sp, 4
	add $s0, $s0, $s2

b.	START: add $s0, $0, $0
	addi $s2, $0, 100
	LOOP: add $s0, $s0, $s2
	addi $s2, $s2, −1
	bne $s2, $0, LOOP

2.37.3

a.	push eax	5,3
b.	test eax, 0x00200010	7, 1, 8, 32

2.37.4

a.	sw $a0, 0($sp)
b.	addi $t0, $0, 0x00200010 and $t1, $s0, $t0 slt $t2, $t1, $0

Solution 2.38

2.38.1

a.	This instruction copies ECX elements, where each element is 2 bytes in size, from an array pointed to by ESI to an array pointer by EDI.
b.	This instruction finds the first occurrence of a byte (given in AL) in an array pointed to by EDI. The search stops when the byte is found, or when the entire length of the array (specified in ECX) is searched. For example, the C library function strlen can easily be implemented using this instruction.

2.38.2

a.	`loop: lh $t0,0($a2)`
	`sh $t0,0($a1)`
	`addi $a0,$a0,-1`
	`addi $a1,$a1,2`
	`addi $a2,$a2,2`
	`bnez $a0,loop`

b.	`loop: lb $t0,0($a1)`
	`beq $t0,$a3,done`
	`addi $a0,$a0,-1`
	`addi $a1,$a1,1`
	`bnez $a0,loop`
	`done:`

2.38.3

	x86	MIPS	Speedup
a.	5	6	1.2
b.	3	5	1.67

2.38.4

	MIPS Code	Code Size Comparison

a.	`f: slt $t0,$a1,$a0`
	`beqz $t0,S`
	`move $v0,$a2`
	`jr $ra`
	`S: move $v0,$a3`
	`jr $ra`
		MIPS: 6 × 4 = 24 bytes
		×86: 25 bytes

b.	`f: beqz $a1,D`
	`move $t0,zero`
	`move $t1,$a0`
	`L: addi $t0,$t0,1`
	`sw $0,0($t1)`
	`addi $t1,$t1,4`
	`bne $t0,$a1,L`
	`D: jr $ra`
		MIPS: 8 × 4 = 32 bytes
		×86: 31 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 speedup is well below 1):

	MIPS Cycles	x86 Cycles	Speedup
a.	4	15	0.27
b.	2	13	0.15

Solution 2.39

2.39.1

a.	0.76 seconds
b.	2.86 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.	107.04%	113.43%
b.	107.52%	114.4%

2.39.4

a.	2.6
b.	3.7

2.39.5

a.	0.88
b.	0.26

2.39.6

a.	0.533333333
b.	not possible

Solution 2.40

2.40.1

a.	In the first iteration $t0 points to a[0] and the lw fetches a[0] as intended. In the second iteration $t0 points to the next byte and the lw uses a non-aligned address and causes a bus error. Note that the computation for $t1 (address of a[n]) does not cause a bus error because that address is not actually used to access memory.
b.	In the very first iteration $0 is 0, and the address of the first lw is one byte into a[0] instead of a[1]. This means this access is non-aligned and causes a bus error.

2.40.2

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

2.40.3

a.

```
f: move $v0,$0
   move $t0,$a0
   sll  $t1,$a1,2      ; We must multiply n by 4 to get the address
   add  $t1,$t1,$a0    ; of the end of array a
L: lw   $t2,0($t0)
   bne  $t2,$a2,S
   addi $v0,$v0,1
S: addi $t0,$t0,4      ; Move to next element in a
   bne  $t0,$t1,L
   jr   $ra
```

b.

```
f:  move $t0,$0
    addi $t1,$a1,-1
L:  sll  $t2,$t0,2       ; We must multiply the index by 4 before we
    add  $t2,$t2,$a0     ; add it to a[] to form the address for lw
    lw   $t3,4($t2)      ; The offset of a[i+1] from a[i] is 4, not 1
    sw   $t3,0($t2)
    addi $t0,$t0,1
    bne  $t0,$t1,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.	10, 11, 12, 13, 14
b.	100, 102, 104, 106, 108

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()` 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.	10 5 0xffe8 7 1
b.	100 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.	1, 5, 7, 10, 0xffe8
b.	1, 5, 7, 100, 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 from 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=5, v=7 So v is a 5-element array of integers that begins at address 7
b.	n=5, v=7 So v is a 5-element array of integers that begins at address 7

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).