2

Solutions

2.1
```
addi f, h, -5 (note, no subi)
add  f, f, g
```

2.2
```
f = g + h + i
```

2.3
```
sub $t0, $s3, $s4
add $t0, $s6, $t0
lw  $t1, 16($t0)
sw  $t1, 32($s7)
```

2.4
```
B[g] = A[f] + A[1+f];
```

2.5
```
add $t0, $s6, $s0
add $t1, $s7, $s1
lw  $s0, 0($t0)
lw  $t0, 4($t0)
add $t0, $t0, $s0
sw  $t0, 0($t1)
```

2.6

2.6.1
```
temp = Array[0];
temp2 = Array[1];
Array[0] = Array[4];
Array[1] = temp;
Array[4] = Array[3];
Array[3] = temp2;
```

2.6.2
```
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)
```

2.7

Little-Endian		Big-Endian	
Address	Data	Address	Data
12	ab	12	12
8	cd	8	ef
4	ef	4	cd
0	12	0	ab

2.8 2882400018

2.9
```
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]
```

2.10 `f = 2*(&A);`

2.11

	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	

2.12

2.12.1 50000000

2.12.2 overflow

2.12.3 B0000000

2.12.4 no overflow

2.12.5 D0000000

2.12.6 overflow

2.13

2.13.1 $128 + \times > 2^{31}-1$, $x > 2^{31}-129$ and $128 + x < -2^{31}$, $x < -2^{31} - 128$ (impossible)

2.13.2 $128 - x > 2^{31}-1$, $x < -2^{31}+129$ and $128 - x < -2^{31}$, $x > 2^{31} + 128$ (impossible)

2.13.3 $x - 128 < -2^{31}$, $x < -2^{31} + 128$ and $x - 128 > 2^{31} - 1$, $x > 2^{31} + 127$ (impossible)

2.14 r-type, add $s0, $s0, $s0

2.15 i-type, 0xAD490020

2.16 r-type, sub $v1, $v1, $v0, 0x00621822

2.17 i-type, lw $v0, 4($at), 0x8C220004

2.18

2.18.1 opcode would be 8 bits, rs, rt, rd fields would be 7 bits each

2.18.2 opcode would be 8 bits, rs and rt fields would be 7 bits each

2.18.3 more registers → more bits per instruction → could increase code size

more registers → less register spills → less instructions

more instructions → more appropriate instruction → decrease code size

more instructions → larger opcodes → larger code size

2.19

2.19.1 0xBABEFEF8

2.19.2 0xAAAAAAA0

2.19.3 0x00005545

2.20
```
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
```

2.21 nor $t1, $t2, $t2

2.22
```
lw  $t3, 0($s1)
sll $t1, $t3, 4
```

2.23 $t2 = 3

2.24 jump: no, beq: no

2.25

2.25.1 i-type

2.25.2
```
addi $t2, $t2, –1
beq  $t2, $0, loop
```

2.26

2.26.1 20

2.26.2
```
i = 10;
do {
   B += 2;
   i = i – 1;
} while ( i > 0)
```

2.26.3 5*N

2.27
```
        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
```

2.28 14 instructions to implement and 158 instructions executed

2.29
```
for (i=0; i<100; i++) {
   result += MemArray[s0];
   s0 = s0 + 4;
}
```

2.30
```
      addi $t1, $s0, 400
LOOP: lw   $s1, 0($t1)
      add  $s2, $s2, $s1
      addi $t1, $t1, -4
      bne  $t1, $s0, LOOP
```

2.31
```
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
```

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

2.33
```
after calling function fib:
old $sp ->   0x7ffffffc   ???
             -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
```

2.34

```
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
```

2.35 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`).

2.36 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.

2.37

```
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
```

2.38 0x00000011

2.39 Generally, all solutions are similar:

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

2.40 No, jump can go up to 0x0FFFFFFC.

2.41 No, range is 0x604 + 0x1FFFC = 0x0002 0600 to 0x604 - 0x20000 = 0xFFFE 0604.

2.42 Yes, range is 0x1FFFF004 + 0x1FFFC = 0x2001F000 to 0x1FFFF004 - 0x20000 = 1FFDF004

2.43

```
trylk: li   $t1,1
       ll   $t0,0($a0)
       bnez $t0,trylk
       sc   $t1,0($a0)
       beqz $t1,trylk
       lw   $t2,0($a1)
       slt  $t3,$t2,$a2
       bnez $t3,skip
       sw   $a2,0($a1)
skip:  sw   $0,0($a0)
```

2.44

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

2.45 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.

2.46

2.46.1 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.

2.46.2 107.04%, 113.43%

2.47

2.47.1 2.6

2.47.2 0.88

2.47.3 0.533333333