2 Solutions

2.1
```
addi  $s0, $s2, -5
add   $s0, $s0, $s1
[addi f,h,-5 (note, no subi) add f,f,g]
```

2.2 `f = g+h+i`

2.3
```
sub   $t0,   $s3, $s4
sll   $t0,   $t0, 2
add   $t0,   $s6, $t0
lw    $t1,   0($t0)
sw    $t1,   32($s7)
```

2.4
```
B [g]= A[f] + A[f+1]
sll    $t0,   $s0,2       # t0 = f * 4
add    $t0,   $s6, $t0    # t0 = &A[f]
lw     $t1,   0($t0)      # s0 = A[f]
lw     $t2,   4($t0)      # s1 = A[1+f]
add    $t0,   $t1, $t2    # t0 = A[f] + A[1+f]
sll    $t1,   $s1,2       # t1 = g * 4
add    $t1,   $s7, $t1    # t1 = &B[g]
sw     $t0,   0($t1)      # B[g] = A[f] + A[1+f]
```

2.5

Little-Endian		Big-Endian	
Address	Data	Address	Data
3	ab	3	12
2	cd	2	ef
1	ef	1	cd
0	12	0	ab

2.6 2882400018

2.7
```
sll    $t0, $s3, 2      # $t0  <-- 4*i
add    $t0, $t0, $s6    # $t0  <-- Addr(A[i])
lw     $t0, 0($t0)      # $t0  <-- A[i]
sll    $t1, $s4, 2      # $t1  <-- 4*j
add    $t1, $t1, $s6    # $t1  <-- Addr(A[j])
lw     $t1, 0($t1)      # $t1  <-- A[j]
add    $t0,$t0,$t1      # t1   <-- A[i] + A[j]
sw     $t0,32($s7)      # B[8] <-- A[i] + A[j]
```

2.8
```
f = 2*(&A)
addi   $t0,   $s6, 4
add    $t1,   $s6, $0
sw     $t1,   0($t0)
lw     $t0,   0($t0)
add    $s0,   $t1, $t0
```

2.9

	type	opcode	rs	rt	rd	immed
addi $t0, $s6, 4	I-type	8	22	8		4
add $t1, $s6, $0	I-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	

	type	opcode, funct3,7	rs1	rs2	rd	Imm
addi x30. X 10.8	I-type	0x13, 0x0, –	10	–	30	8
addi x31. X 10.0	R-type	0x13, 0x0, –	10	–	31	0
sd x31. 0(x30)	S-type	0x23, 0x3, –	31	30	–	0
ld x30. 0(x30)	I-type	0x3, 0x3, –	30	–	30	0
add x5. X30. x31	R-type	0x33, 0x0, 0x0	30	31	5	–

2.10

2.10.1 `0x50000000`

2.10.2 `overflow`

2.10.3 `0xB0000000`

2.10.4 `no overflow`

2.10.5 `0xD0000000`

2.10.6 `overflow`

2.11

2.11.1 There is an overflow if 128 + $s1 > $2^{31} - 1$.
In other words, if $s1 > $2^{31} - 129$.
There is also an overflow if 128 + $s1 < -2^{31}.
In other words, if $s1 < $-2^{31} - 128$ (which is impossible given the range of $s1).

2.11.2 There is an overflow if 128 – $s1 > $2^{31} - 1$.
In other words, if $s1 < $-2^{31} + 129$.
There is also an overflow if 128 – $s1 < -2^{31}.
In other words, if $s1 > $2^{31} + 128$ (which is impossible given the range of $s1).

2.11.3 There is an overflow if $s1 − 128 > $2^{31} - 1$.
In other words, if $s1 > $2^{31} + 127$ (which is impossible given the range of $s1).
There is also an overflow if $s1 − 128 < -2^{31}.
In other words, if $s1 < $-2^{31} + 128$.

2.12 `R -type: add $s0, $s0, $s0`

2.13 `I-type: 0xAD490020`

2.14 `R-type: sub $v1, $v1, $v0, 0x00621822`

2.15 `I-type: lw $v0, 4($at), 0x8C220004`

2.16

2.16.1 The opcode would expand from 7 bits to 9.

The `rs1`, `rs2` , and rd fields would increase from 5 bits to 7 bits.

2.16.2 The opcode would expand from 7 bits to 9.

The`rs1` and `rd` fields would increase from 5 bits to 7 bits. This change does not affect the `imm` field *per se*, but it might force the ISA designer to consider shortening the immediate field to avoid an increase in overall instruction size.

2.16.3 * Increasing the size of each bit field potentially makes each instruction longer, potentially increasing the code size overall.

* However, increasing the number of registers could lead to less register spillage, which would reduce the total number of instructions, possibly reducing the code size overall.

2.17

2.17.1 `0xBABEFEF8`

2.17.2 `0xAAAAAAA0`

2.17.3 `0x00005545`

2.18 It can be done in eight MIPS instructions:

```
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
```

2.19 `nor $t1, $t2, $t2`

2.20
```
lw $t3, 0($s1)
sll $t1, $t3, 4
```

2.21 `$t2 = 3`

2.22

2.22.1 `[0x20000000,2FFFFFFC]`

2.22.2 `[1FFE0000,2001FFFC]`

2.23

2.23.1 The I-type instruction format would be most appropriate because it would allow the branch target to be encoded into the immediate field and the register address to be encoded into the rt field, whose datapath supports both read and write access the register file.

2.23.2 It can be done in three instructions:
```
Loop:
     addi   $s0,$s0,-1   # subtract 1 from $s0
     bltz   $s0,loop     # continue if $s0 not
                           negative
     addi   $s0,$s0,1    # add back 1 that shouldn't
                           have been subtracted
```

2.24

2.24.1 The final value of `$s2` is `20`.

2.24.2
```
i = 10;
do {
   B += 2;
   i = i - 1;
} while (i > 0)
```

2.24.3 `5*N` instructions.

2.25 The C code can be implemented in RISC-V assembly as follows.

```
      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
```

2.26 Answers will vary, but should require approximately 14 MIPS instructions, and when a = 10 and b = 1, should result in approximately 158 instructions being executed.

2.27
```
// Assume MemArray is an array of 32-bit words.
for (i=0; i<100; i++) {
  result += MemArray[i];
}
return result;
```

2.28
```
      addi   $t1,   $s0, 400
LOOP: lw     $s1,   0($t1)
      add    $s2,   $s2, $s1
      addi   $t1,   $t1, -4
      bne    $t1,   $s0, LOOP
```

2.29

```
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
done:
jalr x0, x1
```

2.30

```
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
```

2.31
```
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
```

2.32 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).

2.33 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.

2.34
```
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
```

2.35

2.35.1 `0x11`

2.35.2 `0x44`

2.36 Generally, all solutions are similar:

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

2.37
```
setmax:
    ll $t0,0($a0) # $t0 = *shvar
    sub $t1,$t0,$a1 # $t1 = *shvar-x
    bgez $t1,skip # if result is zero or positive, then
    x <= *shvar, so keep original value of *shvar, else
    replace with x
    add $t0,$0,$a1
skip:
    sc $t0,0($a0) # try to store result to shvar
    beqz $t0,setmax # repeat if operation was not atomic
```

2.38 When two processors A and B begin executing this routine at the same time, at most one of them will execute the store-conditional instruction successfully, while the other will be forced to retry the operation. If processor A's store-conditional successds initially, then B will re-enter the try block, and it will see the new value of shvar written by A when it fi nally succeeds. The hardware guarantees that both processors will eventually execute the code completely.

2.39

2.39.1 No. The resulting machine would be slower overall.
Current CPU requires (num arithmetic * 1 cycle) + (num load/store * 10 cycles) + (num branch/jump * 3 cycles) = 500 * 10^6 * 1 + 300 * 10^6 * 10 + 100 * 10^6 * 3 = 3800 * 10^6 cycles.
The new CPU requires (.75 * num arithmetic * 1 cycle) + (num load/store * 10 cycles) + (num branch/jump * 3 cycles) = 375 * 10^6 * 1 + 300 * 10^6 * 10 + 100 * 10^6 * 3 = 3675 * 10^6 cycles.
However, given that each of the new CPU's cycles is 10% longer than the original CPU's cycles, the new CPU's 3675 * 10^6 cycles will take as long as 4042.5 * 10^6 cycles on the original CPU.

2.39.2 If we double the performance of arithmetic instructions by reducing their CPI to 0.5, then the the CPU will run the reference program in (500 * .5) + (300 * 10) + 100 * 3 = 3550 cycles. This represents a speedup of 1.07.

If we improve the performance of arithmetic instructions by a factor of 10 (reducing their CPI to 0.1), then the the CPU will run the reference program in (500 * .1) + (300 * 10) + 100 *3 = 3350 cycles. This represents a speedup of 1.13.

2.40

2.40.1 Take the weighted average: 0.7 * 2 + 0.1 * 6 + 0.2 * 3 = 2.6

2.40.2 For a 25% improvement, we must reduce the CPU to 2.6 * .75 = 1.95. Thus, we want 0.7 * x + 0.1 * 6 + 0.2 * 3 < = 1.95. Solving for x shows that the arithmetic instructions must have a CPI of at most 1.07.

2.40.3 For a 50% improvement, we must reduce the CPU to 2.6 * .5 = 1.3. Thus, we want 0.7 * x + 0.1 * 6 + 0.2 * 3 < = 1.3. Solving for x shows that the arithmetic instructions must have a CPI of at most 0.14

2.41
```
lwr   $t1, $s0($s6), 2   # t1 = A[f]
add   $s0, $s0, 1
lwr   $t2, $s0($s6), 2   # t2 = A[1+f]
add   $t0, $t1, $t2      # t0 = A[f] + A[1+f]
swr   $t0, $s1($s7), 2   # B[g] = A[f] + A[1+f] (don't
                           need scaled store here)
```

2.42
```
lwr   $t0, $s3($s6),2    # $t0 <-- A[i]
lwr   $t1, $s4($s6),2    # $t1 <-- A[j]
add   $t0, $t0,$t1       # t1 <-- A[i] + A[j]
sw    $t0,32($s7)        # B[8] <-- A[i] + A[j]
```