# 2

# Solutions

**2.1**
```
addi x5, x7,-5
add  x5, x5, x6
[addi f,h,-5 (note, no subi) add f,f,g]
```

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

**2.3**
```
sub     x30, x28, x29    // compute i-j
slli    x30, x30, 3      // multiply by 8 to convert the
                            word offset to a byte offset
ld      x30, 0(x3)       // load A[i-j]
sd      x30, 64(x11)     // store in B[8]
```

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

slli    x30, x5, 3       // x30 = f*8
add     x30, x10, x30    // x30 = &A[f]
slli    x31, x6, 3       // x31 = g*8
add     x31, x11, x31    // x31 = &B[g]
ld      x5, 0(x30)       // f = A[f]
addi    x12, x30, 8      // x12 = &A[f]+8  (i.e. &A[f+1])
ld      x30, 0(x12)      // x30 = A[f+1]
add     x30, x30, x5     // x30 = A[f+1] + A[f]
sd      x30, 0(x31)      // B[g] = x30 (i.e. A[f+1] + A[f])
```

**2.5**

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

**2.6**
```
2882400018
```

**2.7**
```
slli    x28, x28, 3      // x28 = i*8
ld      x28, 0(x10)      // x28 = A[i]
slli    x29, x29, 3      // x29 = j*8
ld      x29, 0(x11)      // x29 = B[j]
add     x29, x28, x29    // Compute x29 = A[i] + B[j]
sd      x29, 64(x11)     // Store result in  B[8]
```

**2.8**
```
f = 2*(&A)
    addi   x30, x10, 8     // x30 = &A[1]
    addi   x31, x10, 0     // x31 = &A
    sd     x31, 0(x30)     // A[1] = &A
    ld     x30, 0(x30)     // x30 = A[1] = &A
    add    x5, x30, x31    // f = &A + &A = 2*(&A)
```

**2.9**

| | type | opcode, funct3,7 | rs1 | rs2 | rd | imm |
|---|---|---|---|---|---|---|
| `addi x30,x10,8` | I-type | 0x13, 0x0, -- | 10 | -- | 30 | 8 |
| `addi x31,x10,0` | 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** `0x5000000000000000`

**2.10.2** `overflow`

**2.10.3** `0xB000000000000000`

**2.10.4** `no overflow`

**2.10.5** `0xD000000000000000`

**2.10.6** `overflow`

**2.11**

**2.11.1** There is an overflow if $128 + x6 > 2^{63} - 1$.
In other words, if $x6 > 2^{63} - 129$.
There is also an overflow if $128 + x6 < -2^{63}$.
In other words, if $x6 < -2^{63} - 128$ (which is impossible given the range of x6).

**2.11.2** There is an overflow if $128 - x6 > 2^{63} - 1$.
In other words, if $x6 < -2^{63} + 129$.
There is also an overflow if $128 - x6 < -2^{63}$.
In other words, if $x6 > 2^{63} + 128$ (which is impossible given the range of x6).

**2.11.3** There is an overflow if $x6 - 128 > 2^{63} - 1$.
In other words, if $x6 < 2^{63} + 127$ (which is impossible given the range of x6).
There is also an overflow if $x6 - 128 < -2^{63}$.
In other words, if $x6 < -2^{63} + 128$.

**2.12** `R-type: add x1, x1, x1`

**2.13** S-type: 0x25F3023 (0000 0010 0101 1111 0011 0000 0010 0011)

**2.14** R-type: sub x6, x7, x5 (0x40538333: 0100 0000 0101 0011 1000 0011 0011 0011)

**2.15** I-type: ld x3, 4(x27) (0x4DB183: 0000 0000 0100 1101 1011 0001 1000 0011)

**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 12.

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** 0x1234567ababefef8

**2.17.2** 0x2345678123456780

**2.17.3** 0x545

**2.18** It can be done in eight RISC-V instructions:

```
addi x7, x0, 0x3f // Create bit mask for bits 16 to 11
slli x7, x7, 11   // Shift the masked bits
and  x28,x5, x7   // Apply the mask to x5
slli x7, x6, 15   // Shift the mask to cover bits 31
                     to 26
xori x7, x7, -1   // This is a NOT operation
and  x6, x6, x7   // "Zero out" positions 31 to
                     26 of x6
slli x28,x28,15   // Move selection from x5 into
                     positions 31 to 26
or   x6, x6, x28  // Load bits 31 to 26 from x28
```

**2.19** xori x5, x6, -1

**2.20**
```
ld   x6, 0(x17)
slli x6, x6, 4
```

**2.21** x6 = 2

**2.22**

**2.22.1** [0x1ff00000, 0x200FFFFE]

**2.22.2** [0x1FFFF000, 0x20000ffe]

**2.23**

**2.23.1** The UJ instruction format would be most appropriate because it would allow the maximum number of bits possible for the "loop" parameter, thereby maximizing the utility of the instruction.

**2.23.2** It can be done in three instructions:

```
loop:
     addi  x29, x29, -1   // Subtract 1 from x29
     bgt   x29, x0, loop  // Continue if x29 not
                             negative
     addi  x29, x29, 1    // Add back 1 that shouldn't
                             have been subtracted.
```

**2.24**

**2.24.1** The final value of xs is 20.

**2.24.2**
```
acc = 0;
i = 10;
while (i ! = 0) {
   acc += 2;
   i--;
}
```

**2.24.3** 4*N + 1 instructions.

**2.24.4** (Note: change condition ! = to > = in the while loop)

```
acc = 0;
i = 10;
while (i >= 0) {
   acc += 2;
   i--;
}
```

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

```
LOOPI:
      addi  x7,   x0, 0     // Init i = 0
      bge   x7,   x5, ENDI  // While i < a
      addi  x30,  x10, 0    // x30 = &D
      addi  x29,  x0, 0     // Init j = 0
LOOPJ:
      bge   x29,  x6, ENDJ  // While j < b
      add   x31,  x7, x29   // x31 = i+j
      sd    x31,  0(x30)    // D[4*j] = x31
      addi  x30,  x30, 32   // x30 = &D[4*(j+1)]
      addi  x29,  x29, 1    // j++
      jal   x0,   LOOPJ
ENDJ:
      addi  x7,   x7, 1     // i++;
      jal   x0,   LOOPI
ENDI:
```

**2.26** The code requires 13 RISC-V instructions. When a = 10 and b = 1, this results in 123 instructions being executed.

**2.27**

```
// This C code corresponds most directly to the given
assembly.

int i;
for (i = 0; i < 100; i++) {
  result += *MemArray;
  MemArray++;
}
return result;

// However, many people would write the code this way:

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

**2.28** The address of the last element of MemArray can be used to terminate the loop:

```
      add x29, x10, 800      // x29 = &MemArray[101]
LOOP:
      ld    x7,   0(x10)
      add   x5,   x5, x7
      addi  x10,  x10, 8
      blt   x10,  x29, LOOP  // Loop until MemArray points
                                to one-past the last element
```

**2.29**

```
      // IMPORTANT! Stack pointer must reamin a multiple
      of 16!!!!

fib:
      beq    x10,  x0, done  // If n==0, return 0
      addi   x5,   x0, 1
      beq    x10,  x5, done  // If n==1, return 1
      addi   x2,   x2, -16   // Allocate 2 words of stack
                                space
      sd     x1,   0(x2)     // Save the return address
      sd     x10,  8(x2)     // Save the current n
      addi   x10,  x10, -1   // x10 = n-1
      jal    x1,   fib       // fib(n-1)
      ld     x5,   8(x2)     // Load old n from the stack
      sd     x10,  8(x2)     // Push fib(n-1) onto the stack
      addi   x10,  x5, -2    // x10 = n-2
      jal    x1,   fib       // Call fib(n-2)
      ld     x5,   8(x2)     // x5 = fib(n-1)
      add    x10,  x10, x5   // x10 = fib(n-1)+fib(n-2)
// Clean up:
      ld     x1,   0(x2)     // Load saved return address
      addi   x2,   x2, 16    // Pop two words from the stack
done:
      jalr   x0,   x1
```

**2.30** [answers will vary]

## 2.31

```
// IMPORTANT! Stack pointer must remain a multiple of 16!!!
f:
  addi  x2, x2, -16   // Allocate stack space for 2 words
  sd    x1, 0(x2)     // Save return address
  add   x5, x12, x13  // x5 = c+d
  sd    x5, 8(x2)     // Save c+d on the stack
  jal   x1, g         // Call x10 = g(a,b)
  ld    x11, 8(x2)    // Reload x11= c+d from the stack
  jal   x1, g         // Call x10 = g(g(a,b), c+d)
  ld    x1, 0(x2)     // Restore return address
  addi  x2, x2, 16    // Restore stack pointer
  jalr  x0, x1
```

**2.32** We can use the tail-call optimization for the second call to `g`, saving one instruction:

```
// IMPORTANT! Stack pointer must remain a multiple of 16!!!
f:
  addi  x2, x2, -16   // Allocate stack space for 2 words
  sd    x1, 0(x2)     // Save return address
  add   x5, x12, x13  // x5 = c+d
  sd    x5, 8(x2)     // Save c+d on the stack
  jal   x1, g         // Call x10 = g(a,b)
  ld    x11, 8(x2)    // Reload x11 = c+d from the stack
  ld    x1, 0(x2)     // Restore return address
  addi  x2, x2, 16    // Restore stack pointer
  jal   x0, g         // Call x10 = g(g(a,b), c+d)
```

**2.33** *We have no idea what the contents of `x10-x14` are, g can set them as it pleases.

*We don't know what the precise contents of `x8` and `sp` are; but we do know that they are identical to the contents when `f` was called.

*Similarly, we don't know what the precise contents of `x1` are; but, we do know that it is equal to the return address set by the "`jal x1, f`" instruction that invoked `f`.

## 2.34

```
a_to_i:
  addi   x28,  x0, 10     # Just stores the constant 10
  addi   x29,  x0, 0      # Stores the running total
  addi   x5,   x0, 1      # Tracks whether input is positive
                            or negative
  # Test for initial '+' or '-'
  lbu    x6, 0(x10)       # Load the first character
  addi   x7, x0, 45       # ASCII '-'
  bne    x6, x7, noneg
  addi   x5, x0, -1       # Set that input was negative
  addi   x10, x10, 1      # str++
  jal    x0, main_atoi_loop

noneg:
  addi   x7, x0, 43       # ASCII '+'
  bne    x6, x7, main_atoi_loop
  addi   x10, x10, 1      # str++

main_atoi_loop:
  lbu    x6, 0(x10)       # Load the next digit
  beq    x6, x0, done     # Make sure next char is a digit,
                            or fail
  addi   x7, x0, 48       # ASCII '0'
  sub    x6, x6, x7
  blt    x6, x0, fail     # *str < '0'
  bge    x6, x28, fail    # *str >= '9'

  # Next char is a digit, so accumulate it into x29

  mul    x29, x29, x28    # x29 *= 10
  add    x29, x29, x6     # x29 += *str - '0'
  addi   x10, x10, 1      # str++
  jal    x0, main_atoi_loop

done:
  addi   x10, x29, 0      # Use x29 as output value
  mul    x10, x10, x5     # Multiply by sign
  jalr   x0, x1           # Return result

fail:
  addi   x10, x0, -1
  jalr   x0, x1
```

## 2.35

**2.35.1** 0x11

**2.35.2** 0x88

**2.36**
```
lui   x10,  0x11223
addi  x10,  x10, 0x344
slli  x10,  x10, 32
lui   x5,   0x55667
addi  x5,   x5, 0x788
add   x10,  x10, x5
```

**2.37**
```
setmax:
    try:
    lr.d  x5,  (x10)        # Load-reserve *shvar
    bge   x5,  x11, release # Skip update if *shvar > x
    addi  x5,  x11, 0
release:
    sc.d  x7,  x5, (x10)
    bne   x7,  x0, try      # If store-conditional failed,
                              try again
    jalr  x0,  x1
```

**2.38** When two processors A and B begin executing this loop 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 loop. 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 finally 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*1 + 300*10 + 100*3 = 3800 cycles.
The new CPU requires (.75*num arithmetic * 1 cycle) + (num load/store * 10 cycles) + (num branch/jump * 3 cycles) = 375*1 + 300*10 + 100*3 = 3675 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 cycles will take as long as 4042.5 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

```
ldr   x28, x5(x10), 3      // Load x28=A[f]
addi  x5, x5, 1            // f++
ldr   x29, x5(x10), 3      // Load x29=A[f+1]
add   x29, x29, x28        // Add x29 = A[f] + A[f+1]
sdr   x12, x6(x11), 3      // Store B[g] = x29
```

**2.42**

```
ldr   x28, x28, (x10), 3   // Load x28=A[i]
ldr   x29, x29, (x11), 3   // Load x29=B[j]
add   x29, x28, x29
sd    x29, 64(x11)         // Store B[8]=x29 (don't
                              need scaled store here)
```