

2

Solutions

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

2.2 `f = g+h+i`

2.3 `SUB X9, X3, X4` // compute `i-j`
`LSL X9, X9, #3` // multiply by 8 to convert the word offset to a byte offset
`ADD X11, X6, X9` // compute `&A[i-j]`
`LDUR X10, [X11, #0]` // load `A[i-j]`
`STUR X10, [X7, #64]` // store in `B[8]`

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

```
LSL X9, X0, #3    // X9 = f*8
ADD X9, X6, X9    // X9 = &A[f]
LSL X10, X1, #3   // X10 = g*8
ADD X10, X7, X10  // X10 = &B[g]
LDUR X0, [X9, #0] // f = A[f]
ADDI X11, X9, #8  // (*) X11 = X9 + 8 (i.e., X11 = &A[f+1])
LDUR X9, [X11, #0] // X9 = A[f+1]
ADD X9, X9, X0    // X9 = X9 + f (i.e., x9 = A[f+1] + A[f])
STUR X9, [X10, #0] // B[g] = X9 (i.e., B[g] = A[f+1] + A[f])
```

2.5 // Using LEV8

```
LSL X9, X0, #3    // X9 = f*8
ADD X9, X6, X9    // X9 = &A[f]
LSL X10, X1, #3   // X10 = g*8
ADD X10, X7, X10  // X10 = &B[g]
LDUR X0, [X9, #0] // f = A[f]
LDUR X9, [X9, #8] // X9 = A[f+1]
ADD X9, X9, X0    // X9 = X9 + f (i.e., X9 = A[f+1] + A[f])
STUR X9, [X10, #0] // B[g] = X9 (i.e., B[g] = A[f+1] + A[f])
```

// Using ARMv8

```
LDR X9, [X6, X0, LSL #3] // f = A[f]
ADD X10, X0, #1          // X10 = f+1
LDR X11, [X6, X10, LSL #3] // x11 = A[f+1]
ADD X12, X9, X11         // X12 = A[f] + A[f+1]
STR X12, [X7, X1, LSL #3] // B[g] = X12
```

2.6

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

2.7 2882400018

```

2.8 // Using LEGv8
      // X9 = A[i]
      LSL X9, X3, #3
      ADD X9, X6, X9
      LDUR X9, [X9, #0]

      // X10 = A[j]
      LSL X10, X4, #3
      ADD X10, X6, X10
      LDUR X10, [X10, #0]
      ADD X9, X9, X10           // compute A[i] + A[j]
      STUR X9, [X7, #64]      // store the result in B[8]

      // Using ARMv8

      LDR X9, [X6, X3, LSL #3] // X9 = A[i]
      LDR X10, [X6, X4, LSL #3] // X10 = A[j]
      ADD X9, X9, X10           // X9 = X9 + X10
      STUR X9, [X7, #64]      // B[8] = X9

```

2.9 $f = 2 * (&A)$ **2.10**

	type	opcode	rm	rn	rd/rt	imm./addr.
ADDI X9, X6, #8	I-type	580/0x244	–	6	9	8
ADD X10, X6, XZR	R-type	1112/0x458	31	6	10	–
STUR X10, [X9, #0]	D-type	1984/0x7c0	–	9	10	0
LDUR X9, [X9, #0]	D-type	1986/0x7c2	–	9	9	0
ADD X0, X9, X10	R-type	1112/0x458	10	9	0	–

2.11**2.11.1** 0x5000000000000000**2.11.2** overflow**2.11.3** 0xB000000000000000**2.11.4** no overflow**2.11.5** 0xD000000000000000**2.11.6** overflow

2.12**2.12.1** There is an overflow if $128 + X1 > 2^{63} - 1$.In other words, if $X1 > 2^{63} - 129$.There is also an overflow if $128 + X1 < -2^{63}$.In other words, if $X1 < -2^{63} - 128$ (which is impossible given the range of $X1$).**2.12.2** There is an overflow if $128 - X1 > 2^{63} - 1$.In other words, if $X1 < -2^{63} + 129$.There is also an overflow if $128 - X1 < -2^{63}$.In other words, if $X1 > 2^{63} + 128$ (which is impossible given the range of $X1$).**2.12.3** There is an overflow if $X1 - 128 > 2^{63} - 1$.In other words, if $X1 < 2^{63} + 127$ (which is impossible given the range of $X1$).There is also an overflow if $X1 - 128 < -2^{63}$.In other words, if $X1 < -2^{63} + 128$ **2.13** R-type: ADD X0, X0, X0**2.14** D-type: 0xf8020149 (1111 1000 0000 0010 0000 0001 0100 1001)**2.15** R-type

SUB X17, X13, X15

(110 0101 1000) (01111) (000000) (01101) (10001)

1100 1011 0000 1111 0000 0001 1011 0001

0xcb0f01b1

2.16 D-type

LDUR X3, [X12, #4]

(111 1100 0010) (0000000100) (00) (01100) (00011)

1111 1000 0100 0000 0100 0001 1000 0011

0xf8404183

2.17**2.17.1** The opcode would expand from 11 bits to 13.

Rm, Rn, and Rd would increase from 5 bits to 7 bits.

2.17.2 The opcode would expand from 10 bits to 12.

Rm and Rd would increase from 5 bits to 7 bits.

2.17.3 * 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.

2.18

2.18.1 0x1234567ababefef8

2.18.2 0x2345678123456780

2.18.3 0x545

2.19 It can be done in eight LEGv8 instructions:

```
// Create bit mask for bits 16 to 11 and apply it to X10
ORRI X12, XZR, #0x3F // (*)
LSL X12, X12, #11
AND X13, X10, X12

// Shift the masked bits to positions 31 to 26
LSL X13, X13, #15

// Create a mask to "zero out" positions 31 to 26 in X11
MOVZ X12, #0xFC00, LSL #16
EORI X12, X12, #-1 // (*) This is a NOT operation
AND X11, X11, X12

ORR X11, X11, X13
```

There are two main benefits to using the full ARMv8 instruction set:

(1) The bitmasks 0x1F800 and 0xFFFFFFFF03FFFFFF can be represented using ARM's complex encoding scheme for immediates.

(2) We can combine the shifts with the logical operations.

Using these ARMv8 features allows us to reduce the code to three instructions.

```
AND X13, X10, #0x1F800
AND X11, X11, #0xFFFFFFFF03FFFFFF
ORR X11, X11, X13, LSL #15
```

2.20 EORI X10, X11, #-1

In ARMv8, you would use

```
MVN X10, X11
```

2.21 LDUR X14, [X13, #0]

```
LSL X11, X14, #4
```

2.22 $X1 = 2$

2.23

2.23.1 [0x1ff00004,0x20100000] (Remember, the immediate value is added to PC+4.)

2.23.2 [0x18000004,0x28000000] (Remember, the immediate value is added to PC+4.)

2.24

2.24.1 The CB 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.

2.24.2 It can be done in three instructions.

```
SUBIS Rn, Rn, #1 // (*) Subtract 1 from Rn
B.PL loop      // if the result is >= 0, then continue with the loop.
ADDI Rn, Rn, #1 // (*) otherwise, exit the loop and
                // add back the 1 that shouldn't have been subtracted.
```

2.25

2.25.1 The final value of X0 is 20.

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

2.25.3 $5*N + 2$ instructions.

2.25.4 The final value of X0 is 22.

2.25.5 (Note the change from $>$ to $>=$ in the while loop)

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

2.25.6 The SUBIS instruction sets the condition flag.

2.25.7

```

    SUBIS X1, X1, #0 // (*)
LOOP: B.LE DONE
    SUBIS X1, X1, #1 // (*)
    ADDI X0, X0, #2 // (*)
    B LOOP

```

DONE:

Note: This optimization works only because the SUB is the last instruction in the loop that sets the condition code.

```

2.26   ORR X10, XZR, XZR // i = 0;
LOOPI:  SUBS XZR, X10, X0 // check i < a
        B.GE ENDI

        MOV X11, XZR, XZR // (!) j = 0;
LOOPJ:  SUBS XZR, X11, X1 // check j < b
        B.GE ENDJ

        ADD X12, X10, X11 // X12 = i + j
        LSL X13, X11, #5 // handles the 4*j as well as the *8 for the word size.
        STR X12, [X2, X13] // D[4*j] = X12 = i + j.

        ADDI X11, X11, #1 // (*) j++
        B LOOPJ
ENDJ:   ADDI X10, X10, #1 // (*) i++;
        B LOOPI
ENDI:

```

2.27 The code requires 13 ARMv8t instructions. When $a = 10$ and $b = 1$, this results in 143 instructions being executed.

2.28 // 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;
```

```
2.29 // using LEGv8
      ADDI X10, X1, #792           // (*) i = MemArray + 99
LOOP: LDUR X11, [X10, #0]         // X11 = *MemArray
      ADD  X0, X0, X11            // result += X11
      SUBI X10, X10, #8           // (*)
      CMP  X10, X1               // (!)
      B.GE LOOP

      // Using the full ARMv8
      MOVZ X10, #99              // i = 99
LOOP: LDR  X11, [X1, X10, LSL #3] // X11 = MemArray[i]
      ADD  X0, X0, X11            // result += X11
      SUBS X10, X10, 1
      B.PL LOOP
```

```
2.30 FIB:
      CBZ X0, DONE               // return 0 if n == 0;

      CMPI X0, #1                // (*)
      B.EQ DONE                  // return 1 if n == 1;

      // IMPORTANT! Stack pointer must remain a multiple of 16!!!
      SUBI SP, SP, #32           // (*) make room on the stack
      STUR X0, [SP, #0]          // Preserve X0
      STUR X30, [SP, #8]         // preserve the return location
      STUR X19, [SP, #16]        // preserve X19

      //first recursive call
      SUBI X0, X0, #1            // (*) Place n-1 in X0
      BL FIB
      ADDI X19, X0, #0           // (*) place the return value in a preserved register

      // second recursive call
      LDUR X0, [SP, #0]          // retrieve a "safe/preserved" copy of X0
      SUBI X0, X0, #2            // (*) place n-2 in X0
      BL FIB

      //compute the return value
      ADD X0, X0, X19
```



```

//clean up
LDUR X30, [SP, #8]
LDUR X19, [SP, #16]
ADDI SP, SP, #32    // (*)

DONE:
BR X30

```

2.31 [answers will vary]

```

2.32 // IMPORTANT! Stack pointer must remain a multiple of 16!!!
F:
  SUBI SP, SP, #16    // (*) make room on the stack
  STUR X19, [SP, #0] // preserve X19
  STUR X30, [SP, #8] // preserve the return location

  ADD X19, X2, X3     // calculate c + d

  BL G                // call g(a, b). (This is the nested call.)

  MOV X1, X19         // (!) Move c+d into X1 for the "outer" call to g

  BL G                // call g(g(a, b), c+d);

// Clean up and return
LDUR X19, [SP, #0]
LDUR X30, [SP, #8]
ADDI SP, SP, #16 // (*)
BR X30

```

2.33 We can use the tail-call optimization for the second call to g, but then we must restore SP, X30, and X19 before that call. Thus, we save only one instruction: BR X30.

```

// IMPORTANT! Stack pointer must remain a multiple of 16!!!
F:
  SUBI SP, SP, #16    // (*) make room on the stack
  STUR X19, [SP, #0] // preserve X19
  STUR X30, [SP, #8] // preserve the return location

  ADD X19, X2, X3     // calculate c + d

// Clean up in preparation for tail call
LDUR X19, [SP, #0]
LDUR X30, [SP, #8]
ADDI SP, SP, #16    // (*)

B G // call g(g(a,b),c+d) with tail-call optimization

```

2.34 *We have no idea what the contents of X5 are, g can set X5 as it pleases.

*We don't know what the precise contents of X20 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 X30 are; but, we do know that it is equal to the return address set by the "BL F" instruction that invoked f.

2.35

```
A_TO_I:
    MOVZ X12, #1           // X12 stores whether the input is positive
    MOVZ X13, #0           // X13 stores our running total.
    MOVZ X14, #10          // X14 just holds the constant 10

    LDURB W10, [X0, #0]    // Load the first character
    CMPI X10, '-'          // (!) Check if it is a negative sign
    B.NE START            // If it is not, skip ahead and begin processing numbers
    MOVZ X12, #0           // X12 stores that the input is not positive
    ADDI X0, X0, #1        // (*) Increment X0 to point at the second character
    LDURB X10, [X0, #0]    // (*) load the 2nd character

START:
    CBZ X10, END           // if character is '\0', we're at end of the string.
    SUBIS X10, X10, '0'    // (*) calculate the integer value of the character
    B.MI NOT_DIGIT         // If the result is negative, then X10 is an ASCII
                          // character lower than '0'.
    CMPI X10, #9           // (!) Check if character is > 9
    B.GT NOT_DIGIT

    MUL X13, X13, X14      // X13 *= 10; account for place value
    ADD X13, X13, X10      // X13 += X10; account for the new digit

    // Note: The previous two instructions can be combined into MADD

    ADDI X0, X0, #1        // (*) move on to the next character
    LDURB X10, [X0, #0]    // (*)
    B START

NOT_DIGIT: MOV X0, #-1 // (!)
BR X30

END: CBNZ X12, POS
SUB X0, X12, X13          // Remember, if input is negative, X12 is 0
BR X30

POS: MOV X0, X13          // (!)
BR X30
```

2.36**2.36.1** 0x11**2.36.2** 0x88

2.37 MOVZ X0, #0x7788
 MOVK X0, #0x5566, LSL #16
 MOVK X0, #0x3344, LSL #32
 MOVK X0, #0x1122, LSL #48

2.38 // acquire the lock
 MOVZ X11, #1
 TRYLOCK: LDXR X10, [X0, #0]
 CBNZ X10, TRYLOCK // try again if lock is held.
 STXR X12, X11, [X0, #0] // (*) try to claim the lock
 CBNZ X12, TRYLOCK // try again if claim fails

 // set the shared variable
 LDUR X13, [X1, #0]
 CMP X13, X2 // (!)
 B.GE RELEASE
 STUR X2, [X1, #0]

 // release the lock
 RELEASE: STUR XZR, [X0, #0]

2.39 // acquire lock
 TRYLOCK: LDXR X10, [X0, #0]
 CMP X10, X1 // (!)
 B.GE NOUPDATE
 STXR X12, X1, [X0, #0] // (*)
 CBNZ X12, TRYLOCK // try again if lock is held.

 NOUPDATE:

2.40 It is possible for one or both processors to complete this code without ever reaching the STXR instruction. If at most one processor reaches the STXR instruction, the code completes successfully. If both processors reach the STXR instruction, one will necessarily (by hardware design) complete first; the other processor will detect this and fail.

2.41

2.41.1 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 $(.75 * \text{num arithmetic} * 1 \text{ cycle}) + (\text{num load/store} * 10 \text{ cycles}) + (\text{num branch/jump} * 3 \text{ cycles}) = 375 * 1 + 300 * 10 + 100 * 3 = 3675$ 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.

2.41.2 If we double the performance of arithmetic instructions by reducing their CPI to 0.5, then 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 CPU will run the reference program in $(500 * .1) + (300 * 10) + 100 * 3 = 3350$ cycles. This represents a speedup of 1.13.

2.42

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

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

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

2.43

```
LDR X9, [X6, X0, LSL #3]    // f = A[f]
ADD X10, X0, 1              // X10 = f+1
LDR X11, [X6, X10, LSL #3]  // X11 = A[f+1]
ADD X12, X9, X11            // X12 = A[f] + A[f+1]
STR X12, [X7, X1, LSL #3]   // B[g] = X12
```

2.44

```
LDR X9, [X6, X3, LSL #3]    // X9 = A[i]
LDR X10, [X6, X4, LSL #3]   // X10 = A[j]
ADD X9, X9, X10             // X9 = X9 + X10
STUR X9, [X7, #64]         // B[8] = X9
```

2.45

```
AND X13, X10, #0x1F800
AND X11, X11, #0xFFFFFFFF03FFFFFF
ORR X11, X11, X13, LSL #15
```

2.46 `EORI X10, X11, #-1` won't work in ARMv8 because -1 can't be represented using ARMv8's complex 12-bit encoding. Most developers would use `MVN X10, X11`, which is a pseudoinstruction for `ORN X10, XZR, X11`.

```
2.47  ADD X10, X1, #800           // i = &MemArray[100]
LOOP:  LDR X11, [X10, #-8]!       // X11 = *(--i)
      ADD X0, X0, X11           // result += X11
      CMP X10, X1               // (!) continue while i > MemArray
      B.GT LOOP                 // (In other words, stop when i reaches MemArray)
```

```
2.48  ADD X10, X1, #800           // stop = &MemArray[100]
LOOP:  LDR X11, [X1], #8         // X11 = *(MemArray++)
      ADD X0, X0, X11           // result += X11
      SUBS X1, XZR, X10         // continue while MemArray < stop
      B.LT LOOP
```

```
2.49  ADD X10, X0, X1, LSL #3    // stop = &array[length]
      LDR X11, [X0], #8         // max = *array
      B ENTER                   // skip to the comparison (in case length == 1)
LOOP:  LDR X12, [X0], #8         // load next value in the array (temp = *array)
      CMP X11, X12              // (!) compare temp to max thus far
      CSEL X11, X11, X12, GT    // update max, if necessary
ENTER: CMP X0, X10              // (!) keep going of array < stop
      B.LT LOOP
      ORR X0, XZR, X11          // put max in X0 for return
      BR X30
```

