Chapter 1 Solutions

n/a

Chapter 2 Solutions

2.1
```
lui  s0, 0x89ABD     # upper 20 bits, incremented by 1 to compensate for addi sign ext
addi s0, s0, 0xDEF   # DEF gets sign-extended to FFFFFDEF, bringing s0 to 89ABCDEF
```

2.3
```
add a4, a0, a2              # add low word
add a5, a1, a3              # add high word
bgeu a4, a0, done           # if low word of sum >= low word of an addend, there is no carry
addi a5, a5, 1              # increment high word if low word produces a carry
done:
```

2.5 0x00408093

2.7 sw tp, 4(a2)

2.9
```
matvecmul: # Base addresses of A, x, y in a0, a1, a2; m,n in a3, a4
  # I,j,sum in s0, s1, s2
  addi sp, sp, -12          # save s0-s2 on the stack
  sw s0, 0(sp)
  sw s1, 4(sp)
  sw s2, 8(sp)
  li s0, 0                  # i = 0
fori:
  bge s0, a3, donei                 # done with I loop if I >= m
  li s2, 0                  # sum = 0
  li s1, 0                  # j = 0
forj:
  bge s1, a4, donej                 # done with j loop if j >= n
  mul t0, s0, a4            # I * n
  add t0, t0, s1            # I * n + j
  slli t0, t0, 2            # multiply by 4 to convert from ints to bytes
  add t0, a0, t0            # &A[i*n+j]
  lw t0, 0(t0)              # A[i*n+j]
  slli t1, s1, 2            # multiply by 4 to convert j from ints to bytes
  addi t1, a1, t1           # &x[j]
  lw t1, 0(t1)              # x[j]
  mul t0, t0, t1            # A[i*n+j] * x[j]
  add s2, s2, t0            # sum = sum + A[i*n+j] * x[j]
  addi s1, s1, 1            # j = j + 1
  j forj
donej:
  slli t0, s0, 2            # multiply by 4 to convert I from ints to bytes
  add t0, a2, t0            # &y[i]
  sw s2, 0(t0)              # y[i] = sum
  addi s0, s0, 1            # i = i + 1
  j fori
donei:
  lw s2, 8(sp)              # restore s0-s2 from stack
  lw s1, 4(sp)
```

```
 lw s0, 0(sp)
 addi sp, sp, 12
 ret                              # return from matvecmul
```

2.11 This recursive function doesn't save the return address, so it doesn't go back to the caller when done. Fix by saving ra on the stack. n does not need to be saved because it is not used after the recursive call.

```
pow3:
 bne a0, zero, else  # check if n != 0
 li a0, 1        # return 1
 ret
else:
 addi sp, sp, -4
 sw ra, 0(sp)
 addi a0, a0, -1  # n-1
 jal pow3        # pow3(n-1)
 li t0, 3
 mul a0, a0, t0   # 3*pow(n-1)
 lw ra, 0(sp)
 addi sp, sp, 4
 ret
```

2.13

```
int f(int x[], int y[], int n) {
        int sum = 0;
        for (int i=0; i<n; i++)  sum = sum + x[i] * y[i];
        return sum;
}
```

2.15 The following program stores 0x00000001 into temp. The value at a byte offset of 0 is 0 for big endian or 1 for little-endian, so read and return this byte.

```
endian:
 la t1, temp              # &temp
 li t0, 1
 sw t0, 0(t1)             # temp = 0x00000001
 lb a0, 0(t1)             # a0 = 1 for little endian or 0 for big endian
 ret
```

2.17 RV64I are 32 bits long as well and are encoded identically to RV32I except for new instructions such as ld. Both RV32I and RV64I need the same number of bits to specify the operation, source, and destination registers, and immediate.

2.19 To support all 6 branches, the controller would need to determine if the branch is taken based on the appropriate flag (eq, lt, ltu, or their complements). In controller.sv of Code Example 2.15, replace the PCSrc logic with

```
case (Funct3[2:0])
 3'b000:  Flag = eq;  // beq
 3'b001:  Flag = ~eq; // bne
 3'b100:  Flag = lt; // blt
```

```
  3'b101:  Flag = ~lt // bge
  3'b110:  Flag = ltu; // bltu
  3'b111:  Flag = ~ltu; // bgeu
  default: Flag = 0;
endcase
assign PCSrc = Branch & Flag | Jump;
```

2.21 The least significant bit of controls is Jump. Hence, Jump is always 0. The jal at address 40 will not be taken, so the addi at 44 will put 1 in x2. Hence, the add at 48 will set x2 = (1 + 18) = 19, and the sw at 4C will write 19 to address 100.

2.

Chapter 3 Solutions

3.1 The sum is 0x12f.

3p1.S

```
.section .text.init
.globl rvtest_entry_point

rvtest_entry_point:
   li t0, 0x42
   li t1, 0xED
   add t2, t0, t1

self_loop:
   j self_loop
```

Makefile

```
TARGET = 3p1

$(TARGET).objdump: $(TARGET).elf
   riscv64-unknown-elf-objdump -D $(TARGET).elf > $(TARGET).objdump

$(TARGET).elf: $(TARGET).S Makefile
   riscv64-unknown-elf-gcc -g -o $(TARGET) -march=rv64gc -mabi=lp64 -mcmodel=medany \
      -nostartfiles -T../../link/link.ld $(TARGET).S -o $(TARGET).elf

clean:
   rm -f $(TARGET).elf $(TARGET).objdump
```

3p1.objdump

```
0000000080000000 <rvtest_entry_point>:
   80000000:    04200293          li      t0,66
   80000004:    0ed00313          li      t1,237
   80000008:    006283b3          add     t2,t0,t1

000000008000000c <self_loop>:
   8000000c:    a001              j    8000000c <self_loop>
```

Command line:

```
harris@chips:~/cvw/examples/exercises/3p1$ make
riscv64-unknown-elf-gcc -g -o 3p1 -march=rv64gc -mabi=lp64 -mcmodel=medany \
   -nostartfiles -T../../link/link.ld 3p1.S -o 3p1.elf
riscv64-unknown-elf-objdump -D 3p1.elf > 3p1.objdump
harris@chips:~/cvw/examples/exercises/3p1$ spike --isa rv64gc -d 3p1.elf
warning: tohost and fromhost symbols not in ELF; can't communicate with target
(spike)
core   0: 0x0000000000001000 (0x00000297) auipc   t0, 0x0
(spike)
core   0: 0x0000000000001004 (0x02028593) addi    a1, t0, 32
(spike)
core   0: 0x0000000000001008 (0xf1402573) csrr    a0, mhartid
(spike)
core   0: 0x000000000000100c (0x0182b283) ld      t0, 24(t0)
(spike)
core   0: 0x0000000000001010 (0x00028067) jr      t0
(spike)
core   0: >>>>  $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0_zmmul1p0_zaamo1p0_zalrsc1p0
core   0: 0x0000000080000000 (0x04200293) li      t0, 66
```

```
(spike)
core   0: 0x0000000080000004 (0x0ed00313) li      t1, 237
(spike)
core   0: 0x0000000080000008 (0x006283b3) add     t2, t0, t1
(spike) reg 0 t2
0x000000000000012f
```

3.3 Copy the files and change N to `.dword 6`. The sum 1+2+3+4+5+6 = 21 = 0x15, so change the first line of `sumtest.reference_output` to 15. The program takes 13 + 4N = 13 + 24 = 37 = 0x25 cycles to run, so change the second line to 25. Compile with `make` and simulate with `make sim`, and the signatures should match. The Spike trace is shown below. The program iterates through the `for` loop six times, adding to the running sum and incrementing the loop counter.

```
harris@chips:~/cvw/examples/exercises/3p3$ spike -d sumtest
(spike) run
core   0: 0x0000000000001000 (0x00000297) auipc   t0, 0x0
core   0: 0x0000000000001004 (0x02028593) addi    a1, t0, 32
core   0: 0x0000000000001008 (0xf1402573) csrr    a0, mhartid
core   0: 0x000000000000100c (0x0182b283) ld      t0, 24(t0)
core   0: 0x0000000000001010 (0x00028067) jr      t0
core   0: >>>>  rvtest_entry_point
core   0: 0x0000000080000000 (0x00005117) auipc   sp, 0x5
core   0: 0x0000000080000004 (0x20010113) addi    sp, sp, 512
core   0: 0x0000000080000008 (0x00004297) auipc   t0, 0x4
core   0: 0x000000008000000c (0xff828293) addi    t0, t0, -8
core   0: 0x0000000080000010 (0x0002b503) ld      a0, 0(t0)
core   0: 0x0000000080000014 (0xc0202c73) csrr    s8, instret
core   0: 0x0000000080000018 (0x02c000ef) jal     pc + 0x2c
core   0: >>>>  $xrv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0_zmmul1p0_zaamo1p0_zalrsc1p0
core   0: 0x0000000080000044 (0x00001141) c.addi  sp, -16
core   0: 0x0000000080000046 (0x0000e022) c.sdsp  s0, 0(sp)
core   0: 0x0000000080000048 (0x0000e426) c.sdsp  s1, 8(sp)
core   0: 0x000000008000004a (0x00004401) c.li    s0, 0
core   0: 0x000000008000004c (0x00004485) c.li    s1, 1
core   0: >>>>  for
core   0: 0x000000008000004e (0x00954563) blt     a0, s1, pc + 10
core   0: 0x0000000080000052 (0x00009426) c.add   s0, s1
core   0: 0x0000000080000054 (0x00000485) c.addi  s1, 1
core   0: 0x0000000080000056 (0x0000bfe5) c.j     pc - 8
core   0: >>>>  for
core   0: 0x000000008000004e (0x00954563) blt     a0, s1, pc + 10
core   0: 0x0000000080000052 (0x00009426) c.add   s0, s1
core   0: 0x0000000080000054 (0x00000485) c.addi  s1, 1
core   0: 0x0000000080000056 (0x0000bfe5) c.j     pc - 8
core   0: >>>>  for
core   0: 0x000000008000004e (0x00954563) blt     a0, s1, pc + 10
core   0: 0x0000000080000052 (0x00009426) c.add   s0, s1
core   0: 0x0000000080000054 (0x00000485) c.addi  s1, 1
core   0: 0x0000000080000056 (0x0000bfe5) c.j     pc - 8
core   0: >>>>  for
core   0: 0x000000008000004e (0x00954563) blt     a0, s1, pc + 10
core   0: 0x0000000080000052 (0x00009426) c.add   s0, s1
core   0: 0x0000000080000054 (0x00000485) c.addi  s1, 1
core   0: 0x0000000080000056 (0x0000bfe5) c.j     pc - 8
core   0: >>>>  for
core   0: 0x000000008000004e (0x00954563) blt     a0, s1, pc + 10
core   0: 0x0000000080000052 (0x00009426) c.add   s0, s1
core   0: 0x0000000080000054 (0x00000485) c.addi  s1, 1
core   0: 0x0000000080000056 (0x0000bfe5) c.j     pc - 8
core   0: >>>>  for
core   0: 0x000000008000004e (0x00954563) blt     a0, s1, pc + 10
```

```
core   0: 0x0000000080000052 (0x00009426) c.add   s0, s1
core   0: 0x0000000080000054 (0x00000485) c.addi  s1, 1
core   0: 0x0000000080000056 (0x0000bfe5) c.j     pc - 8
core   0: >>>>  for
core   0: 0x000000008000004e (0x00954563) blt     a0, s1, pc + 10
core   0: >>>>  done
core   0: 0x0000000080000058 (0x00008522) c.mv    a0, s0
core   0: 0x000000008000005a (0x00006402) c.ldsp  s0, 0(sp)
core   0: 0x000000008000005c (0x000064a2) c.ldsp  s1, 8(sp)
core   0: 0x000000008000005e (0x00000141) c.addi  sp, 16
core   0: 0x0000000080000060 (0x00008082) ret
core   0: 0x000000008000001c (0xc0202cf3) csrr    s9, instret
core   0: 0x0000000080000020 (0x418c8cb3) sub     s9, s9, s8
core   0: 0x0000000080000024 (0x00004297) auipc   t0, 0x4
core   0: 0x0000000080000028 (0xfe428293) addi    t0, t0, -28
core   0: 0x000000008000002c (0x00a2b023) sd      a0, 0(t0)
core   0: 0x0000000080000030 (0x0192b423) sd      s9, 8(t0)
core   0: >>>>  write_tohost
core   0: 0x0000000080000034 (0x00001317) auipc   t1, 0x1
core   0: 0x0000000080000038 (0xfcc30313) addi    t1, t1, -52
core   0: 0x000000008000003c (0x00004285) c.li    t0, 1
core   0: 0x000000008000003e (0x00533023) sd      t0, 0(t1)
core   0: >>>>  self_loop
core   0: 0x0000000080000042 (0x0000a001) c.j     pc + 0
```

3.5 The disassembled code for matvecmul is shown below. A, x, y, m, and n are passed in a0, a1, a2, a3, and a4, respectively. The disassembly is shown below. Comments are added.

```
00000008000001c2 <matvecmul>:

// Matrix-vector multiplication y = Ax.
// A is an m rows x n columns matrix.
void matvecmul(int A[], int x[], int y[], int m, int n) {
# a0: A
# a1: x
# a2: y, later temporary
# a3: m
# a4: n
# t4: i*n
# a6: sum
            int i, j, sum;
            for (i=0; i<m; i = i + 1) {
   800001c2:        04d05663            blez        a3,done   i             # if m <= 0, leave
   800001c6:        8e32                mv          t3,a2                   # t3 = &y[0]
   800001c8:        068a                slli        a3,a3,0x2 # a3 = m*4 (convert to bytes)
   800001ca:        00d60f33            add         t5,a2,a3  # t5 = &y[m] for done detection
   800001ce:        00271313            slli        t1,a4,0x2 # t1 = n*4 (convert to bytes)
   800001d2:        932e                add         t1,t1,a1  # t1 = &x[n] for done detection
   800001d4:        4e81                li          t4,0                    # t4 = i*n = 0
   800001d6:        a805                j           bodyi
   800001d8 bodyj:
   800001d8:        002e9613            slli        a2,t4,0x2 # a2 = i*4 (convert to bytes)
   800001dc:        962a                add         a2,a2,a0  # a2 = &A[i*n+j]
                    sum = 0;
                    for (j=0; j<n; j = j + 1)
                              sum = sum + A[i*n+j] * x[j];
   800001de:        87ae                mv          a5,a1                   # a5 = &x[0]
                    sum = 0;
   800001e0:        4801                li          a6,0                    # a6 = sum = 0
                              sum = sum + A[i*n+j] * x[j];
   800001e2 bodyj:
   800001e2:        00062883            lw          a7,0(a2)  # a7 = A[i*n+j]
   800001e6:        4394                lw          a3,0(a5)  # a3 = x[j]
   800001e8:        031686bb            mulw        a3,a3,a7  # a3 = A[i*n+j] * x[j]
```

```
   800001ec:       0106883b        addw    a6,a3,a6  # sum = sum + [i*n+j] * x[j]
                   for (j=0; j<n; j = j + 1)
   800001f0:       0611            addi    a2,a2,4           # a2 = pointer to next element of A
   800001f2:       0791            addi    a5,a5,4           # a5 = pointer to next element of x
   800001f4:       fe6797e3        bne     a5,t1,bodyj       # repeat until last element of x
                   y[i] = sum;
   800001f8 afterj:
   800001f8:       010e2023        sw      a6,0(t3)   # y[i] = sum
           for (i=0; i<m; i = i + 1) {
   800001fc:       0e11            addi    t3,t3,4           # t3 = pointer to next element of y
   800001fe:       00ee8ebb        addw    t4,t4,a4   # t4 = t4 + n (for next i)
   80000202:       01ee0663        beq     t3,t5,donei       # repeat until last element of y
                   sum = 0;
   80000206 bodyi:
   80000206:       4801            li      a6,0              # a6 = sum = 0
                   for (j=0; j<n; j = j + 1)
   80000208:       fce048e3        bgtz    a4,bodyj  # if n < 0, leave
   8000020c:       b7f5            j       afterj
           }
}
   8000020e donei:
   8000020e:       8082            ret
```

The following commands run spike, go to the start of the matvecmul function, find the address of y, run matvecmul, and print the values of y, which are [0x7a 0x32] = [122 50] as expected.

```
harris@chips:~/cvw/examples/exercises/3p5$ spike -d matvecmul
(spike) until pc 0 800001c2
(spike) reg 0 a2
0x0000000080020f90
(spike) until pc 0 8000020e
(spike) mem 80020f90
0x0000007a00000032
```

Curious readers might study the code with -O2, which is optimized further.

3.7 The completed functions and results are shown below. The cycle count is 856 with -O and 792 with -O2.

```
// Add two Q1.31 fixed point numbers
int add_q31(int a, int b) {
   return a + b;
}

// Multiplly two Q1.31 fixed point numbers
int mul_q31(int a, int b) {
   long res = (long)a * (long)b;
   int result = res >> 31; // shift right to get the 32-bit result;
      //this is equivalent to shifting left by 1 and discarding the bottom 32 bits
   //printf("mul_q31: a = %x, b = %x, res = %lx, result = %x\n", a, b, res, result);
   return result;
}

// low pass filter x with coefficients c, result in y
// n is the length of x, m is the length of c
// inputs in Q1.31 format
void fir(int x[], int c[], int y[], int n, int m) {
   int i, j;
   for (j=0; j<n-m+1; j++) {
      y[j] = 0;
      for (i=0; i<m; i++)
         y[j] = add_q31(y[j], mul_q31(c[i], x[j-i+(m-1)]));
   }
}
harris@chips:~/cvw/examples/exercises/3p7$ make; spike fir
```

```
riscv64-unknown-elf-gcc -o fir -gdwarf-2 -O2\
  -march=rv64gc -mabi=lp64d -mcmodel=medany \
  -nostdlib -static -lm -fno-tree-loop-distribute-patterns \
  -T../../C/common/test.ld -I../../C/common \
  fir.c ../../C/common/crt.S ../../C/common/syscalls.c
riscv64-unknown-elf-objdump -S -D fir > fir.objdump
y[0] = 4fad3f2f
y[1] = 627c6236
y[2] = 4fad3f32
y[3] = 1e6f0e17
y[4] = e190f1eb
y[5] = b052c0ce
y[6] = 9d839dc6
y[7] = b052c0cb
y[8] = e190f1e6
y[9] = 1e6f0e12
y[10] = 4fad3f2f
y[11] = 627c6236
y[12] = 4fad3f32
y[13] = 1e6f0e17
y[14] = e190f1eb
y[15] = b052c0ce
y[16] = 9d839dc6
mcycle = 856
minstret = 863
```

3.9 The program with inline assembly is:

```
#include <stdio.h>  // supports printf
int main(void) {
   int a = 3;
   int b = 4;
   int c;
   // compute c = a + 2*b using inline assembly
   asm volatile("slli %0, %1, 1" : "=r" (c) : "r" (b));          // c = b << 1
   asm volatile("add %0, %1, %2" : "=r" (c) : "r" (a), "r" (c)); // c = a + c
   printf("c = %d\n", c);
}
```

The object dump show a and b are in a1 and a5. C is computed in a1, replacing a, and a5 is also used along the way because neither a nor b are needed later.

```
00000000800001c2 <main>:
#include <stdio.h>  // supports printf
int main(void) {
   800001c2:        1141                addi    sp,sp,-16
   800001c4:        e406                sd      ra,8(sp)
            int a = 3;
            int b = 4;
            int c;
            // compute c = a + 2*b using inline assembly
            asm volatile("slli %0, %1, 1" : "=r" (c) : "r" (b));            // c = b << 1
   800001c6:        4791                li      a5,4
   800001c8:        0786                slli    a5,a5,0x1
            asm volatile("add %0, %1, %2" : "=r" (c) : "r" (a), "r" (c)); // c = a + c
   800001ca:        458d                li      a1,3
   800001cc:        95be                add     a1,a1,a5

            printf("c = %d\n", c);
   800001ce:        2581                sext.w  a1,a1
   800001d0:        00000517            auipc   a0,0x0
   800001d4:        6d850513            addi    a0,a0,1752 # 800008a8 <atol+0x6a>
   800001d8:        516000ef            jal     800006ee <printf>
}
   800001dc:        4501                li      a0,0
   800001de:        60a2                ld      ra,8(sp)
   800001e0:        0141                addi    sp,sp,16
```

```
800001e2:       8082                ret
```

Spike gives

```
harris@chips:~/cvw/examples/exercises/3p9$ spike inline
c = 11
```

3.11 The following code is produced using RISC-V (32-bits) gcc 14.2.0. Without optimization, the compiler saves registers on the stack and restores them. It also keeps local variables in memory. With -O, it keeps variables in the a and t registers and does not save or restore them. It also replaces the multiplication of i*n+j with addition by n on each iteration. -O is smaller and faster, so there is little reason not to use it. -O2 is similar to -O.

No Optimization

```
matvecmul(int*, int*, int*, int, int):
        addi    sp,sp,-64
        sw      ra,60(sp)
        sw      s0,56(sp)
        addi    s0,sp,64
        sw      a0,-36(s0)
        sw      a1,-40(s0)
        sw      a2,-44(s0)
        sw      a3,-48(s0)
        sw      a4,-52(s0)
        sw      zero,-20(s0)
        j       .L2
.L5:
        sw      zero,-28(s0)
        sw      zero,-24(s0)
        j       .L3
.L4:
        lw      a4,-20(s0)
        lw      a5,-52(s0)
        mul     a4,a4,a5
        lw      a5,-24(s0)
        add     a5,a4,a5
        slli    a5,a5,2
        lw      a4,-36(s0)
        add     a5,a4,a5
        lw      a4,0(a5)
        lw      a5,-24(s0)
        slli    a5,a5,2
        lw      a3,-40(s0)
        add     a5,a3,a5
        lw      a5,0(a5)
        mul     a5,a4,a5
        lw      a4,-28(s0)
        add     a5,a4,a5
        sw      a5,-28(s0)
        lw      a5,-24(s0)
        addi    a5,a5,1
        sw      a5,-24(s0)
.L3:
        lw      a4,-24(s0)
        lw      a5,-52(s0)
        blt     a4,a5,.L4
        lw      a5,-20(s0)
        slli    a5,a5,2
        lw      a4,-44(s0)
        add     a5,a4,a5
        lw      a4,-28(s0)
        sw      a4,0(a5)
        lw      a5,-20(s0)
        addi    a5,a5,1
        sw      a5,-20(s0)
```

-O

```
matvecmul(int*, int*, int*, int, int):
        ble     a3,zero,.L1
        mv      t3,a2
        slli    a3,a3,2
        add     t5,a2,a3
        slli    t1,a4,2
        add     t1,a1,t1
        li      t4,0
        j       .L3
.L8:
        slli    a2,t4,2
        add     a2,a0,a2
        mv      a5,a1
        li      a6,0
.L4:
        lw      a3,0(a2)
        lw      a7,0(a5)
        mul     a3,a3,a7
        add     a6,a6,a3
        addi    a2,a2,4
        addi    a5,a5,4
        bne     a5,t1,.L4
.L6:
        sw      a6,0(t3)
        addi    t3,t3,4
        add     t4,t4,a4
        beq     t3,t5,.L1
.L3:
        li      a6,0
        bgt     a4,zero,.L8
        j       .L6
.L1:
        ret
```

-O2

```
matvecmul(int*, int*, int*, int, int):
        ble     a3,zero,.L1
        slli    a3,a3,2
        slli    t3,a4,2
        add     t5,a2,a3
        add     t3,a1,t3
        li      t4,0
.L3:
        bgt     a4,zero,.L5
        sw      zero,0(a2)
        addi    a2,a2,4
        add     t4,t4,a4
        bne     a2,t5,.L3
.L1:
        ret
.L13:
        sw      a7,0(a2)
        addi    a2,a2,4
        add     t4,t4,a4
        beq     a2,t5,.L12
.L5:
        slli    a6,t4,2
        add     a6,a0,a6
        mv      a5,a1
        li      a7,0
.L4:
        lw      a3,0(a6)
        lw      t1,0(a5)
        addi    a5,a5,4
        addi    a6,a6,4
        mul     a3,a3,t1
        add     a7,a7,a3
        bne     t3,a5,.L4
        j       .L13
.L12:
        ret
```

```
.L2:
        lw      a4,-20(s0)
        lw      a5,-48(s0)
        blt     a4,a5,.L5
        nop
        nop
        lw      ra,60(sp)
        lw      s0,56(sp)
        addi    sp,sp,64
        jr      ra
```

3.13 The objdump for strlen is shown below, with labels and comments added for clarity. The base address of str is passed in a0. strlen uses a registers for temporary values because they are not preserved across the function call. It copies the base address to a4 as a pointer walking through the string It checks if the 2 lsbs of the base address are 00 to see if str is word aligned to check length quickly word by word. If not, it goes to the slower checkbybyte routine. It checks if a word w contains a NULL (0) byte with some clever bit manipulation:
((w & 0x7f7f7f7f) + 0x7f7f7f7f) | w | 0x7f7f7f7f != 0xffffffff
If any byte of w is 0, that byte & 0x7f = 00. Then + 0x7f = 0x7f. Then or with that byte = 0x7f, or with 0x7f = 0x7f, so the byte does not equal 0xff. But if the byte is nonzero, that byte & 0x7f gives the bottom 7 bits of the byte. If the byte is between 0x01 and 0x7f, then the bottom 7 bits + 0x7f gives a number with a 1 in the most significant byte, which when OR'd with 0x7f gives 0xff. If the byte is between 0x80 and 0x8f, then it has a 1 in the most significant bit, which when OR'd with 0x7f gives 0xff. Hence, the result will be 0xffffffff if all of the bytes are nonzero.
This complicated method of checking by word takes 7 instructions per word for RV32 or per double-word for RV64. In contrast, a sequence of 4 or 8 lb, beqz instructions would take 8 or 16 instructions per word for RV32 or per doubleword for RV64. It is slightly faster (7/8) for RV32 and much faster (7/16) for RV64.
The routine increments a4 by 4 after each word. Once it gets to a word with a nonzero byte, it checks which word is nonzero, and computes a4 – a0 as a tentative length. It must correct this length by subtracting 4, 3, 2, or 1 based on which byte in the word was nonzero If it is the first (least significant) byte, done4 subtracts 4. If it is the second byte, done3 subtracts 3. Otherwise, some logic subtracts 2 or 1.
If str is not word aligned, checkbybyte checks for null one byte at a time until it either becomes word aligned (and uses the check by words) or finds a null and returns the length.

```
80000044 <strlen>:
80000044:       00357793    andi    a5,a0,3                 # a5 = 2 lsbs of str address
80000048:       872a        mv              a4,a0                   # copy address of str ptr=a4
8000004a:       ef9d        bnez    a5,checkbybyte          # check if str word aligned
8000004c setupword:
8000004c:       7f7f86b7    lui     a3,0x7f7f8
80000050:       f7f68693    addi    a3,a3,-129              # yes: a3 = 0x7f7f7f7f
80000054:       55fd        li              a1,-1                   # a1 = 0xffffffff
80000044 checkbyword:
80000056:       4310        lw              a2,0(a4)                # read next word w from ptr
80000058:       0711        addi    a4,a4,4                 # increment ptr by 1 word
8000005a:       00d677b3    and     a5,a2,a3            # strip out msb of each byte
8000005e:       97b6        add     a5,a5,a3            # add 0x7f7f7f7f
80000060:       8fd1        or              a5,a5,a2            # or with w
80000062:       8fd5        or              a5,a5,a3            # or with 0x7f7f7f7f
80000064:       feb789e3    beq     a5,a1,checkbyword # repeat if result=0xffffffff
                                                                # otherwise
80000068:       ffc74683    lbu     a3,-4(a4)           # check LSB
8000006c:       40a707b3    sub     a5,a4,a0            # a5 = a4 – a0
80000070:       ca8d        beqz    a3,done4            # if LSB = 0, return a5-4
80000072:       ffd74683    lbu     a3,-3(a4)           # check second byte
```

```
80000076:           c29d       beqz       a3,done3            # if 0, return a5 - 3
80000078:           ffe74503   lbu        a0,-2(a4)           # check third byte
8000007c:           00a03533   snez       a0,a0                       # a0 = (a0 != 0)
80000080:           953e       add        a0,a0,a5            # a0 = a5 + a0
80000082:           1579       addi       a0,a0,-2            # a0 = a0 - 2
80000084:           8082       ret
80000086 checkrealigned:
80000086:           d2f9       beqz       a3,setupword                # setup for checking by word
80000088 checkbybyte:
80000088:           00074783   lbu        a5,0(a4)            # read byte from ptr
8000008c:           0705       addi       a4,a4,1                     # increment ptr
8000008e:           00377693   andi       a3,a4,3                     # a3 = 2 lsbs of address
80000092:           fbf5       bnez       a5,checkrealigned   # continue if not null
80000094:           8f09       sub        a4,a4,a0            # if null, return a4 – a0 - 1
80000096:           fff70513   addi       a0,a4,-1
8000009a:           8082       ret
8000009c done3:
8000009c:           ffd78513   addi       a0,a5,-3            # decrement by 3
800000a0:           8082       ret
800000a2 done4:
800000a2:           ffc78513   addi       a0,a5,-4            # decrement by 4
800000a6:
```

3.15 The only change is the first line of sim in the Makefile:

```
riscv_sim_RV64 -T $(TARGET).signature.output --signature-granularity 8 $(TARGET)
```

Sail prints a trace of the instructions, and then confirms the signature matches

```
harris@chips:~/cvw/examples/exercises/3p15$ make sim
riscv_sim_RV64 sumtest -T sumtest.signature.output --signature-granularity 8
using sumtest.signature.output for test-signature output.
setting signature-granularity to 8 bytes
Running file sumtest.
ELF Entry @ 0x80000000
tohost located at 0x80001000
begin_signature: 0x80004008
end_signature: 0x80004018
CSR mstatus <- 0x0000000A00000000 (input: 0x0000000000000000)
mem[X,0x0000000000001000] -> 0x0297
mem[X,0x0000000000001002] -> 0x0000
[0] [M]: 0x0000000000001000 (0x00000297) auipc t0, 0x0
x5 <- 0x0000000000001000
mem[X,0x0000000000001004] -> 0x8593
mem[X,0x0000000000001006] -> 0x0202
[1] [M]: 0x0000000000001004 (0x02028593) addi a1, t0, 0x20
x11 <- 0x0000000000001020
mem[X,0x0000000000001008] -> 0x2573
mem[X,0x000000000000100A] -> 0xF140
…
[55] [M]: 0x000000008000003C (0x4285) c.li t0, 0x1
x5 <- 0x0000000000000001
mem[X,0x000000008000003E] -> 0x3023
mem[X,0x0000000080000040] -> 0x0053
[56] [M]: 0x000000008000003E (0x00533023) sd t0, 0x0(t1)
htif[0x0000000080001000] <- 0x0000000000000001
htif-syscall-proxy cmd: 0x000000000001
SUCCESS
diff --ignore-case sumtest.signature.output sumtest.reference_output || exit
echo "Signature matches! Success!"
Signature matches! Success!
```

Chapter 4 Solutions

4.1 `assign` statements are natural for modeling logic described with Boolean equations. `case` statements are natural for modeling logic described by truth tables.

4.3 The hardware implies a 64x32 RAM with four byte enables, built from four 64x8 single-port RAMs, each taking one of the byte enables. The generate loop produces one RAM for each byte, and the indexed part select selects the appropriate byte for each to write.

4.5 The `if/else` doesn't contain an `else` clause for when state is not `S0`, `S1`, or `S2`, so it is not combinational logic.

4.7 The code should use `==` for comparison rather than `=` for assignment.

4.9 The signals `State` and `state` are capitalized differently and are not the same signal. In the case, `L0AD` is misspelled with a zero instead of the letter `O`, which is hard to detect in some fonts.

4.11 No solution available

Chapter 5 Solutions

n/a

Chapter 6 Solutions

6.1 What constraints are essential to provide during RTL synthesis for SoC-level integration?

Timing constraints (like clock periods), area limits, power budgets, and I/O constraints are essential. These are usually provided in an SDC (Synopsys Design Constraints) file written usually in Tcl.

6.3 How does RTL coding style impact synthesis results

Poor RTL can infer unintended hardware. For example, using if instead of case may create large muxes. Clean, synthesizable coding styles lead to more predictable timing and smaller area.

6.5 How can synthesis tools optimize for low power?

Through techniques like clock gating, power-aware cell selection, and multi-voltage domains. Power constraints guide these optimizations during synthesis

6.7 Synthesis wallypipelinecore using Skywater 130nm at 200MHz and compare to the results in Section 6.3.

6.9 Compare a synthesis run with Skywater 130nm technology and examining the power report for the wallypipelined core using the default activity factor vs. using the activity using a vcd file from the program in Code Example 2.16.

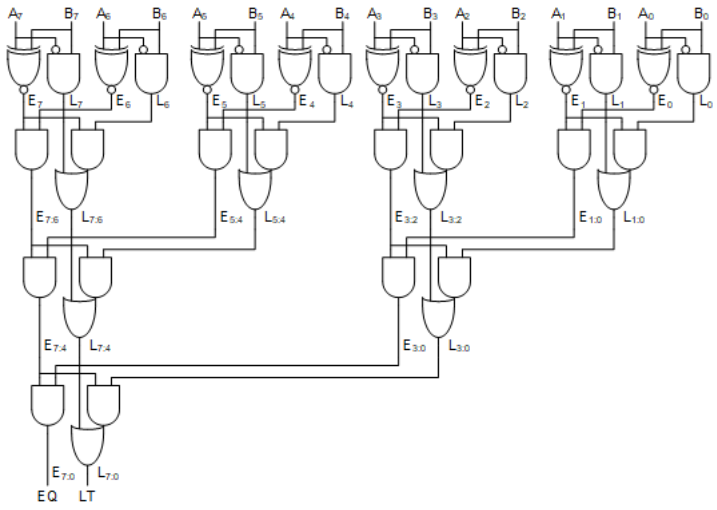
Chapter 7 Solutions

7.1 The integer register file has two read ports and one write port because R-type instructions such as add rd, rs1, rs2 need to read rs1 and rs2 and write rd.

7.3 The separate comparator is faster, allowing the IEU to resolve branches earlier in the Execute stage. Branches are often on the critical path.

7.5 A = 0x12345678. k = 12. Z = {31'b0, $A_{31:0}$} = 0x0000000012345678. The shift amount is k = 12. The first mux shifts right by 0, not 16, producing 0x000012345678. The second mux shifts right by 8, not 0, producing 0x0000123456. The third mux shifts right by 4, not 0, producing 0x000012345. The fourth mux shifts right by 0, not 2, producing 0x000012345. The final mux shifts right by 0, not 1, producing Y = 0x00012345.

7.7



7.9 Adr = 0x80003002. WD = 0x5678567856785678 (four copies of the bottom 16 bits). ByteMask = 0b00001100 to write to the second of the four halfwords because the least significant bits of Adr are 010.

7.11 Many solutions, similar to 7.15.

Chapter 8 Solutions

8.1 In a simple multitasking RTOS, the tasks run in user mode and the operating system runs in machine mode. With suitable PMP configuration, a misbehaving user task cannot access the memory of another task or lock up the system. The operating system controls access to shared peripherals and switches between tasks.

8.3 Endian mode in RV32 is controlled by mstatush.MBE (bit 5).

```
li t0, 32                    # 1 in bit 5
csrs mstatush, t0    # set bit 5 (mstatush.MBE)
```

8.5

```
rvtest_entry_point:

   # set up trap trap_handler
   la t0, trap_handler # address of trap trap_handler
   csrw mtvec, t0      # mtvec = pointer to trap handler
   la t0, trapstack    # address of trap stack
   csrw mscratch, t0   # mscratch = pointer to trap stack

   lw t0, 1(zero)      # cause access or misaligned load fault to invoke trap handler

self_loop:
   j self_loop


trap_handler:
   csrrw tp, mscratch, tp  # swap tp and mscratch to put a trap stack pointer in tp
   sw t0, 0(tp)            # save t0 on trap stack
   csrr t0, mepc           # read mepc
   addi t0, t0, 4          # mepc + 4
   csrw mepc, t0           # mepc = mpec + 4 (return to next instruction)
   lw t0, 0(tp)            # restore t0 from trap stack
   csrrw tp, mscratch, tp  # restore tp and trap stack pointer
   mret

trapstack:
   .word 0                 # room to save a register
```

8.7

```
rvtest_entry_point:

   # set up trap trap_handler
   la t0, trap_handler # address of trap trap_handler
   csrw mtvec, t0      # mtvec = pointer to trap handler
   la t0, trapstack    # address of trap stack
   csrw mscratch, t0   # mscratch = pointer to trap stack

   la t0, destination  # get address to make a load
   lw t0, 3(t0)        # misaligned load will invoke trap handler
   # should return 0x23456789 in t0

self_loop:
   j self_loop


trap_handler:
   csrrw tp, mscratch, tp  # swap tp and mscratch to put a trap stack pointer in tp

   # save all registers on trap stack.  We will need to index into them to find the arguments to emulate multiply
```

```
sw x0, 0(tp)          # x0 is 0, but we might want to use it
sw x1, 4(tp)
sw x2, 8(tp)
sw x3, 12(tp)
sw x4, 16(tp)
sw x5, 20(tp)
sw x6, 24(tp)
sw x7, 28(tp)
sw x8, 32(tp)
sw x9, 36(tp)
sw x10, 40(tp)
sw x11, 44(tp)
sw x12, 48(tp)
sw x13, 52(tp)
sw x14, 56(tp)
sw x15, 60(tp)
sw x16, 64(tp)
sw x17, 68(tp)
sw x18, 72(tp)
sw x19, 76(tp)
sw x20, 80(tp)
sw x21, 84(tp)
sw x22, 88(tp)
sw x23, 92(tp)
sw x24, 96(tp)
sw x25, 100(tp)
sw x26, 104(tp)
sw x27, 108(tp)
sw x28, 112(tp)
sw x29, 116(tp)
sw x30, 120(tp)
sw x31, 124(tp)

csrr t0, mcause        # check cause of trap
li t1, 4              # cause 4 is misaligned load
bne t0, t1, trap_return # return for any other cause

# check if instruction is lw (op=0000011, funct3 = 010)
csrr t0, mepc          # address of faulting instruction
lw t3, 0(t0)          # fetch the instruction.  It must have been a load.
srli t1, t3, 12        # get funct3 field (instr[14:12])
andi t1, t1, 7         # mask off other bits.
xori t1, t1, 0b010     # should produce 0 if funct3 = 010
bnez t1, trap_return    # return if any other kind of load

# emulate lw by performing four byte loads
csrr t0, mtval         # address of load instruction
lbu t1, 0(t0)          # read zeroth byte
lbu t2, 1(t0)          # read the first byte
slli t2, t2, 8         # shift into position
or t1, t1, t2          # merge with zeroth byte
lbu t2, 2(t0)          # read the second byte
slli t2, t2, 16        # shift into position
or t1, t1, t2          # merge with previous two bytes
lbu t2, 3(t0)          # read the third byte
slli t2, t2, 24        # shift into position
or t2, t1, t2          # merge with previous three bytes

# find rd and put result there
srli t1, t3, 7         # extract rd from instr[11:7]
andi t1, t1, 31        # mask off other bits
slli t1, t1, 2         # multiply rd by 4 to make it a word index
add t1, tp, t1         # find location of rd on trap stack
sw t2, 0(t1)           # store result into rd storage on trap stack

# return to next instruction
```

```
trap_return:
   csrr t0, mepc          # read mepc
   addi t0, t0, 4         # mepc + 4
   csrw mepc, t0          # mepc = mpec + 4 (return to next instruction)
   # restore all of the registers from the trap stack (rd could be in any one)
   lw x1, 4(tp)
   lw x2, 8(tp)
   lw x3, 12(tp)
   lw x4, 16(tp)
   lw x5, 20(tp)
   lw x6, 24(tp)
   lw x7, 28(tp)
   lw x8, 32(tp)
   lw x9, 36(tp)
   lw x10, 40(tp)
   lw x11, 44(tp)
   lw x12, 48(tp)
   lw x13, 52(tp)
   lw x14, 56(tp)
   lw x15, 60(tp)
   lw x16, 64(tp)
   lw x17, 68(tp)
   lw x18, 72(tp)
   lw x19, 76(tp)
   lw x20, 80(tp)
   lw x21, 84(tp)
   lw x22, 88(tp)
   lw x23, 92(tp)
   lw x24, 96(tp)
   lw x25, 100(tp)
   lw x26, 104(tp)
   lw x27, 108(tp)
   lw x28, 112(tp)
   lw x29, 116(tp)
   lw x30, 120(tp)
   lw x31, 124(tp)
   csrrw tp, mscratch, tp  # restore tp and trap stack pointer
   mret

destination:
   .dword 0x0123456789ABCDEF   # fill destination with some stuff

trapstack:
   .fill 32, 4            # room to save registers
```

8.9 Unavailable

8.11 See Section 8.4.4 and src/privileged/trap.sv

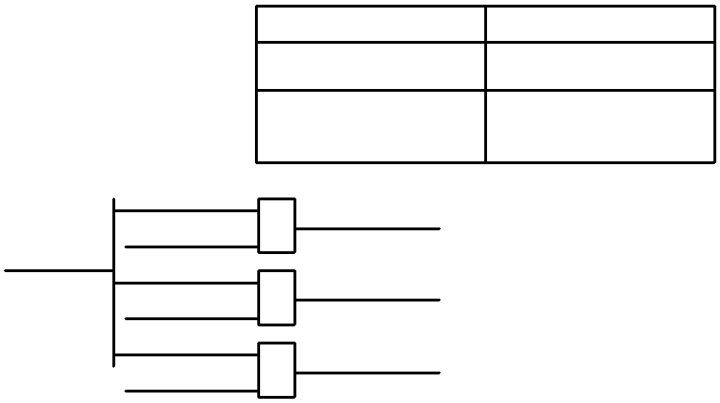
8.13 The PCF register holds the program counter. An adder adds 4 to find the next PC (PCPlus4). The PC logic chooses the next PC from PCPlus4 or various other possibilities:

- If reset is asserted, choose the RESET_VECTOR
- If a trap occurs, go to STVEC or MTVEC depending on the privilege mode (SelMTVEC). Possibly load the bottom 6 bits with the shifted Cause for vectored interrupts.
- If a return instruction is executed, choose MEPC or SPEC depending on the privilege mode (mretM).

- If a CSR write or a fence occurs in the Memory stage, the pipeline is flushed, and the PC is reloaded to restart at the following instruction (PCE, Execute stage).
- If a branch or jump is taken (PCSrcE), pick the target address computed by the IEU (IEUAdrE).

Chapter 9 Solutions

9.1 The table shows the lower 16 bits of a 32-bit address for each device's address range. The bits highlighted in blue are common for all addresses in the range and indicate the subset of the addresses to use when decoding. This encoding scheme only works if the address range is contiguous and the last address ends in all ones.



9.3 APB, AHB, and AXI are three bus architectures from ARM's AMBA bus family that target different parts of the system architecture. APB is the simplest with the lowest performance. APB transactions are not pipelined and intended for low performance peripherals such are UART and I2C. Its design facilitates low area and power implementations.

AHB is a high-performance bus that pipelines two transactions by overlapping the address and data phases of the bus transactions. This alone doubles potential throughput of AHB over APB. However, pipelined operations increase bus traffic which increases power. Modestly more area is needed to implement AHB.

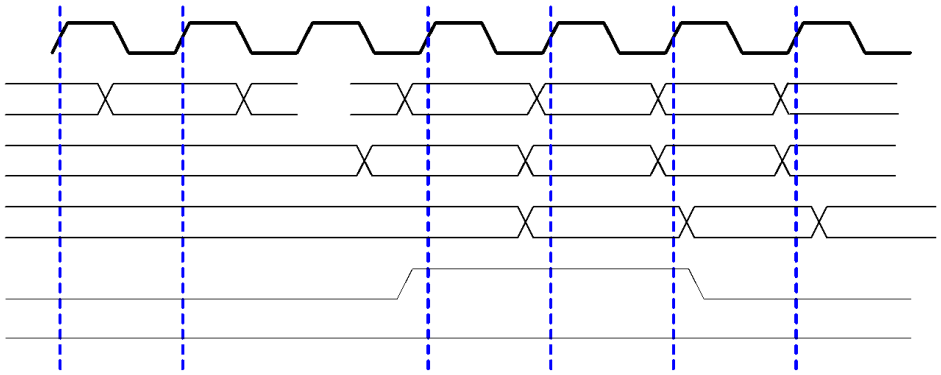
AHB transactions must be completed in the same order they are issued. In system with multiple managers, and a fast and slow peripherals, this is a major bottleneck. AXI separates the requesting address phase, data phase, and write response phases into separate channels and then assigns each transactions a unique identifier. This allows bus requests to reorder as they pass through a complex network. A high performance CPU can issue many requests without waiting for a response. This allows AXI to dramatically increase practical bandwidth for speculative CPUs. The tradeoffs are increased areas for each of the AXI channels and increased power consumption. Complexity is another concern. AXI is harder to verify compared to AHB.

9.5 Eight back-to-back single transfers with 1 wait state inserted between each transfer effectively cuts the transfer bandwidth in half and doubles the transfer time. It takes 17 cycles to transfer all 8 transfers. The effective bandwidth is 8 bytes / 2 cycles * 1 GHz = 4×10^9 bytes / s. Burst transfer only pays the 1 cycle wait at the start of the data phase. The 8-beat transaction takes 10 cycles. The next transaction's address phase will overlap with the 10^{th} cycle, such that each transaction will effectively take 9 cycles. 64 bytes are

transferred every 9 cycles with an effective bandwidth = 8 beats * 8 bytes/beat / 9 cycles * 1 GHz = 7.1×10^9 bytes / s.

9.7 IROM and DTIM are advantageous because they are small, fast, and local to the microcontroller core. The main application can run from the IROM and store frequently used data in the DTIM. However, not all applications will fit inside the IROM and DTIM. Applications can be swapped between the slower external memory and IROM/DTIM. The main disadvantage of the IROM is it is fixed at design time and cannot be updated. The other disadvantage is they are constrained by the size of SRAM that can fit on the chip.

9.9



9.11 BusCommittedM prevents the CPU from trapping interrupts while a bus transaction is in progress. Once a bus transaction is started it cannot be interrupted and must complete its operation. An interrupt would trap and flush the instructions in the Fetch and Memory stages and require flushing the bus transactions. If a store instruction is flushed and BusCommittedM didn't exist the bus would need to cancel a potentially inflict transaction. The problem is the CPU's bus controller does not know if the write to memory has occurred or not. For memory loads the problem is less problematic because the memory is not changed. However, for non-idempotent read such as a peripheral register the problem still exists. BusCommittedM delays the trap until the next instruction before the bus transaction can start.

Chapter 10 Solutions

10.1 Programs tend to access nearby memory addresses, which we call spatial locality. For example, programs execute instructions sequentially so the next instruction in memory is likely to be fetched and executed. Similarly, if a program reads an element from an array, it will likely read the next few elements. Cache lines hold n bytes of continuous data so they will contain multiple adjacent instructions or data. The length of the cache line changes impacts how much data the cache fetches when the access misses. Increasing the length increases the potential spatial locality and will increase hit rate up to a certain length. However, for a given cache capacity a longer line means fewer cache sets.

10.3 The cache capacity C = 32 bytes, the cache line length L = 4 bytes, and the number of ways N = 1. The number of lines = C / L = 32 / 4 = 8 lines. The number of sets = C / (L × N) = 32 / (4 × 1) = 8 lines.

The cache line length is 4 bytes, the line offset is 2 bits long, adr[1:0]. The set is 8 lines, 3 bits long, adr[4:2]. And Finaly, the tag is remaining bits (assume a 16-bit address), adr[15:5]. To compute the hit rate first find where in the cache each address will live. 0x0 -> set 0, 0x4 -> set 1, 0x8 -> set 2, 0xC ->3, 0x40 -> set 0, 0x44 -> set 1, 0x48 -> set 2, 0x4c -> set 3, 0x10 -> set 4, 0x14 -> set 5, 0x18 -> set 6, 0x1C -> set 7.

The first access to 0x0, 0x4, 0x8, and 0xC miss due to compulsory miss. 0x40, 0x44, 0x48, and 0x4C miss because they conflict with the same sets as the previous 4 accesses. 0x10, 0x14, 0x18, and 0x1C miss the first loop as well, but will hit on the next 4 iterations because they do not conflict with any other accesses. Only 16 out of 60 access hit.

10.5 $0.04 \times (150 + 0.65) + 0.96 \times 0.65 + 0.5 \times 0.05 \times 150 = 6.986 + 7.2 = 14.186$

10.7 No solution provided.

10.9 The cache replacement policy maintains a history of the way access patterns within a given set and then decides which way to evict when the cache is set is full and a new line must be inserted into the cache. The cache updates the replacement policy on every memory access and uses the replacement policy and eviction.

10.11 No solution provided.

10.13 No solution provided.

Chapter 11 Solutions

11.1 Virtual memory gives each process its own independent address space, provides protection and security, uses the disk to give the effect of having more memory than the amount of DRAM, and simplifies dealing with relocating programs and handling fragmentation. It requires address translation, which is usually cached in a TLB and handled by a hardware page table walker on a TLB miss.

11.3

a) The address in binary is 0000000000000000 000000000 000000000 000000000 001000101 011001111000. Hence, the sign extension is 0, VPN3 = 0, VPN2 = 0, and VPN1 = 0, which is a megapage with PPN 0x90000, so the base address is 0x90000000. The megapage offset is 45678, so the physical address is 0x90045678.

b) The binary address is 0000000000000000 000000000 000000000 000000001 000000001 111111111100. Hence, the sign extension is 0, VPN3 = 0, VPN2 = 0, VPN1 = 1, VPN0 = 1, which is a kilopage at base address 80200000. The offset is FFC, so the physical address is 0x80200FFC.

c) The binary address start is 0000000000000000 000000010 . Hence the sign extension is 0, VPN3 = 2, which is a terapage at base address of 0x0000000. The remainder of the bits are the terapage offset, 0x0081234560, so the physical address is 0x81234560.

d) The binary address is 1111111111111111 111111111 111111111 111111111 111111111 111011101101. Thus, the sign extension is all 1s, VPN3 = 511, VPN2 = 511, VPN1 = 511, VPN0 = 511, which is a kilopage at base address 80201000. Adding the page offset of EED gives physical address 0x80201EED.

11.5 satp has a mode bit of 1, ASID = 0, and PPN of 80500, so the page table begins at address 0x80500000. The megapage at virtual address 0x80000000 has VPN1 = 1000000000 = 0x200. The kilopage at virtual address 0x80803000 has VPN1 = 1000000010 = 0x202, and VPN0 = 0000000011 = 0x003. Each PTE is 32 bits long, so the megapage level 1 entry is at an address of 0x80500000 + 0x200 * 4 = 0x80500800. It has a PPN of 0x080000, and U = X = W = R = V = 1, so the PTE is 0x2000001F. The kilopage has a level 1 entry at 0x80500000 + 0x202 * 4 = 0x80500808. It points to 0x80501000, so the PPN is 0x080501, with V = 1, and X=W=R=0 to indicate it is nonleaf. Hence the PTE is 0x20140401. Finally, the kilopage has a level 0 entry at 80501000 + 0x003 * 4 = 0x8050100C. The page has a PPN of 000001, with V = R = W = 1, so the PTE is 0x00000407. In summary, the page table has the following entries

Address	Value
80500800	2000001F
80500808	20140401
8050100C	00000407

11.7 If the L1$ way size is larger than the page size, then the upper bits of the physical address needed to access the cache will not be available immediately. The processor could wait

until address translation occurs before reading the cache, which increases latency. Or it could play tricks like page coloring to work around the problem.

11.9 Consider a UART peripheral that sends a byte over a serial port every time a byte is written to a memory-mapped transmit data register. Writing the same byte twice causes the byte to be sent twice, which is not intended behavior. Hence, the peripheral memory location should be non-idempotent.

11.11 Use the following regions. Region 0 grants RWX access to 80000000-87FFFFFF to everyone. Region 1 grants RX access to 1000-1FFF to everyone. Region 2 makes all other addresses inaccessible in supervisor and user mode. Region 3 and 4 grant RW access in machine mode to 10000000-10000107. Region 5 makes all other addresses inaccessible in machine mode.

Reg	pmpaddr	pmpcfg	L	A	X	W	R	Comments
0	20FFFFFF	1F	0	NAPOT	1	1	1	80000000-87FFFFFF RWX
1	000000BF	1B	0	NAPOT	0	1	1	1000-1FFF RX
2	FFFFFFFF	18	0	NAPOT	0	0	0	All addresses inaccessible in SU mode
3	04000000	00	0	OFF	0	0	0	10000000
4	04000042	8B	1	TOR	0	1	1	10000000-10000107: RW accessible in M
5	FFFFFFFF	98	1	NAPOT	0	0	0	All other addresses inaccessible in M

Chapter 12 Solutions

12.1 The eight blocks are cache, ahbcacheinterface, HPTW, MMU, endian swap (ES), subword read (SWR), subword write (SWW), a second ES, and the Atomic unit. The cache stores frequently access data memory. The ahbcacheinterface connects the cache to the AHB bus. The HPTW is the hardware page table walker. When either the ITLB or DTLB miss, the HPTW state machine walks the current page tables in memory and fills the corresponding TLB. The two ES modules swap byte endianness for loads or stores if big endian mode is enabled. Finally, SWR and SWW mux the sub word or byte within the cache's native word size.

12.3 A store hit begin with IEUAdrE going to the cache's internal tag SRAM. The clock rises and the SRAM samples the address and outputs the tag. IEUAdrM muxes inside the HPTW to IHAdrM. The MMU then translates the address to the physical address with its DTLB. The PMP and PMA check the physical address PAdrM for access violations and the cache compares it to the Tag to check for a Hit. The WriteDataM is aligned by SWW and not swapped by SW because it is a little-endian store. Finally, the LittleEndianWriteDataM is written into the cache SRAM and the dirty bit set.

12.5 The EBU requires a separate beat counter because AHB does not transmit which beat of the transaction is currently being transmitted. Instead, it transmits the current SEQ address and then stops the transaction. Because the EBU needs to know when the transaction will end it uses another counter to keep track of how much of the bus is transmitted.

12.7 Solution on rosethompson's cvw ahbopt branch, https://github.com/rosethompson/cvw/blob/ahbopt/src/ebu/ahbcacheinterface.sv

12.9 No solution provided

Chapter 13 Solutions

13.1 Static branch predictors use profiling to measure an application's branch behavior and then embed the expected branch outcomes into the program so that the CPU can make an optimized guess. Dynamic branch prediction uses live past branch behavior to estimate the future branch outcomes. Static branch predictors require cooperation between the ISA and compiler, while dynamic branch predictors can be invisible to the ISA. A dynamic branch predictor records branch outcomes as the CPU executes branches and then computes the expected branch outcome from this branch history.

13.3 The outer branch is executed 100 times and the first time are predicted wrong, so it's prediction accuracy is 99%. The inner branch br2 is always predicted wrong on the first and last time and it is executed 4 x 100 = 400 so its accuracy is 200/400 = 50%. The inner loop is more complicated. a and b both need to be odd for the fuct1 to be called. As the inner loop executes when a is even, b will not matter and fuct1 will be called and the branch will be taken. If a is odd, then as b oscillates between even and odd, the branch will also oscillate between taken and not taken. This means when a is odd br3 will be predicted correctly every time except the first. When it is even it will be predicted wrong every time. The prediction accuracy is 50%.

13.5 No solution provided

13.7 The BTA does not have the correct address for return instructions because the return address changes based on where the function was called. The return address stack (RAS) is a shadow stack inside in the branch predictor that records the last N return addresses. Each jal instruction (function call) pushes the return address onto the stack and each jalr ra (function return) pops the stack. Combined with the instruction class prediction the RAS reliably predicts the return addresses.

13.9 Branches are resolved in three ways. When the fetched program counter PCD is correct PCD == PCCorrectE then no action is taken. When it is incorrect the CPU needs to look at the branch direction PCSrcE. If it is not taken PCSrcE = 0, then PCNextF = PCE + 4. If it is taken, PCSrcE = 1 and PCNextF = IEUAdrE. In both cases the Pipeline flushes the Fetch and Decode stages.

13.11 The Fetch and Decode stages are flushed on a branch misprediction.

Chapter 14 Solutions

14.1 According to Table I.6, c.li is a CI type instruction with op = 01 and instr 15:13 = 010. The rd field is x5 = 00101, and imm = 13 = 001101. Hence, the machine language is 0100_0010_1011_0101 = 0x42b5

14.2 According to Table I.6, c.li translates to addi t0, x0, 13.

14.3 c.slli x7, 3

14.4 According to Figure I.2, c.li has a 6-bit immediate. It is sign-extended, so it is in the range [-32, 31].

14.5 beqz is a CB instruction, with an 8-bit immediate. The immediate is doubled (because no addresses are odd), giving a range [-256, 254], and a most positive offset of 254.

14.6 The instruction 0x03D00C13 is fetched from 0x31E. This address wraps a cache line boundary so half has to be fetched from each line.

Chapter 15 Solutions

15.1 The `__mulsi3` function takes the multiplicand and multiplier in `a0` and `a1`, respectively. It returns the lowest XLEN bits of the product in `a0`. It uses a shift and add process, adding the multiplicand to the running result if the lsb of the multiplier is 1, then shifting the multiplier right and the multiplicand left and repeating until the multiplier is 0.

Chapter 16 Solutions

16.1 The Ideal Gas constant can be represented using finite precision in U4.12 as $1000.010100001000 = 2^3 + 2^{-2} + 2^{-4} + 2^{-9} = 8.314453125$. Unfortunately, using finite precision Avogadro's number cannot be represented in U4.12 as the largest fractional integer available is $2^3 = 8$. However, for the Ideal Gas constant in single-precision using $Sign = 0$, biased exponent = $130 = 1000\,0010 = 2^{130-127} = 2^3$, fraction/mantissa $= 329,728 = 0A\,1000 = 1 + 0.039306640625$. The single-precision number is $(1)^0 \times 2^3 \times 1.039306640625 = 8.314453125$. For Avogadro's number in single precision is $Sign = 0$, biased exponent = $205 = 1100\,1101 = 2^{205-127} = 2^{78}$, fraction/mantissa = $8,325,120 = FE\,1000 = 1 + 0.992431640625$. The single precision number is $(1)^0 \times 2^{78} \times 1.992431640625 = 6.021755 \times 10^{23}$.

15.3 C790 = 1 10001 1110010000. Unbiased exponent = 17-15 = 2.

$-1.111001 \cdot 2^2 = -111.1001 = -7.5625$

16.5 (a) 1.0 + -(largest subnorm) rounds up to 1.0, down to $1.1111\ldots \cdot 2^{-1}$

3F800000 + 807FFFFF = 3F800001 in RNE, 3F7FFFFF in RZ

(b) Subtract two numbers that are nearly equal.

$1.00000001 \cdot 2^{10} - 1.11111101 \cdot 2^9 = 1.01 \cdot 2^3$

0 10001001 00000001 – 0 10001000 11111101 = 44808000 – 447E8000 = 41200000

c) Multiply two numbers to get an inexact subnormal number.

$1.01 \cdot 2^{-49} * 1.0 \cdot 2^{-100} = 1.01 \cdot 2^{-149}$, rounds to the smallest subnorm $1.0 \cdot 2^{-149}$.

0 01001110 01000… * 0 00011011 0000. This is tiny and inexact so it sets the underflow flag.

d) $(1+2^{-23}) * (1+2^{-23}) - 1.0 = 1 + 2^{-23} + 2^{-23} + 2^{-46} - 1 = 2^{-22} + 2^{-46}$. This rounds to 2^{-22} and is inexact. Biased exponent = 105: 0 01101001 0000… = 34800000

3F800001 * 3F800001 – 3F800000 = 34800000, inexact.

16.7 X = 0x3e00 = $1.1 \cdot 2^0$. Y = 0x3f00 = $1.11 \cdot 2^0$. Z = 0xc100 = $-1.01 \cdot 2^1$.

Pm = 1.1 x 1.11 = 10.101
Pe = 0 + 0 = 0
Acnt = 0 – 1 = -1
Zm = -1.01 >> -1 = 10.1
Sm = 10.101 – 10.1 = 0.001
Mcnt = 3
Mm = 0.001 << 3 = 1.0

Me = 0 – 3 = -3
R = 1.0×2^{-3} (exact, no rounding needed)

W = $1.0 \cdot 2^{-3}$ = 0x300 = 0.125

16.9 Viewed as 7-bit fractions, X = 249/128, D = 177/128. Q = X / D = 249/177 = 1.40677966 = 180.07 / 128. This should round to Q = 1.0110100.

Initialization	D	0001.0110001	
	-D = ~D + 1	1110.1001111	
Step 0	WS_{-1} = X	0001.1111001	
	WC-1	0000.0000000	
	$-q_0D$	1110.1001111	Wmsbs = 0001, so q_0 = 1
	sum	1111.0110110	<< 1
	carry	0001.0010010	<< 1
	Check	W = sum+carry = 0000.1001000 X = QD+W = (1.0)(1.0110001) + 0.1001000 = 1.1111001	
Step 1	WS0	1110.1101100	
	WC0	0010.0100100	Wmsbs = 0000, so q_1 = 1
	$-q_1D$	1110.1001111	
	sum	0010.0000111	<<1
	carry	1101.1011000	<<1
	Check	W = sum + carry = 1111.1011111 X = QD+W = (1.1)(1.0110001) + 1111.1011111 = 1.1111001	
Step 2	WS1	0100.0001110	
	WC1	1011.0110000	Wsbs = 1111, so q2 = 0
	-q2D	0000.0000000	
	sum	1111.0111110	
	carry	0000.0000000	
	Check	W = sum + carry = 1111.0111110	

16.11 Repeat Example F.7 to perform radix-2 recurrence division of 1.1111001 / 1.0110001.

Chapter 17 Solutions

Chapter 18 Solutions

18.1 bext a2, a0, a1

bseti a2, a0, 5 # Sets bit 5 of a0, stores in a2

18.3 ctz a1, a0

18.5 Write a program to compute the multiplicative inverse in GF(2^8)? Give an example by computing the multiplicative inverse of $x^6 + x + 1$ for the irreducible polynomial of $m(x) = x^8 + x^4 + x^3 + x + 1$. Also, write a RV32 implementation and show the results using HTIF as Chapter 3.3.10.

Use the Extended Euclidean Algorithm (EEA) over polynomials modulo an irreducible polynomial. Hardware versions use lookup tables or loop-based inversion logic. Using the example given, one can compute the multiplicative inverse of 0x43 (i.e., ($x^6 + x + 1$) in GF(2^8) modulo the AES irreducible polynomial $m(x) = x^8 + x^4 + x^3 + x + 1$) (i.e., 0x11B). Initially, we assume: $r_0 = 0x11B$ and $r_1 = 0x43$. The final value before the remainder becomes 0 when $s_0 = 0xCA$. One can verify this with $0x43 \cdot 0xCA \; mod \; 0x11B = 0x01$.

The multiplicative inverse uses the EEA algorithm is performing division over GF(2^8). The power of cryptography is that it can perform this computation with XOR and bit shifts instead of add/subtract and divide. That is, we are finding the inverse of $a \; mod \; m(x)$ such that $a \cdot a^{-1} = 1 \; mod \; m(x)$. To compute the inverse using the EEA algorithm, you set $s_0 = 0$, $s_1 = 1$, $r_0 = m(x)$, and $r_1 = a$. The main computation then performs division like regular division explained in Chapter 15. This equation is somewhat confusing but uses the Bézout's identity to help solve things easier by computing the difference in the degree of the modulus and the input polynomial. This algorithm is called the extended Euclidean algorithm as it computes not only the $gcd(a, b)$ but x, y such that $a \cdot x + b \cdot y = gcd(a, b)$ (i.e., using the Euclidean algorithm with an extra computation).

The algorithm to compute this is listed in many textbooks but amounts to the following given that we can put the polynomial in a form in GF(2) for easy symbolic computation. The most often method is in an array where the coefficients are ordered from highest to lowest degree. For example, $p(x) = x^3 + x + 1$ would be encoded as: 1011_2 or 11_{10}. Then, the algorithm to compute the EEA over GF(28) is performed as follows (the algorithm is also found in the GitHub repository in the README within the examples subdirectory):

Algorithm 1 Extended Euclidean Algorithm in GF(2^8)

Require: Input value a, modulus m (usually $0x11B$)
Ensure: Return s_0 such that $s_0 \cdot a \equiv 1 \pmod{m}$ if inverse exists

$r_0 \leftarrow m$
$r_1 \leftarrow a$
$s_0 \leftarrow 0$
$s_1 \leftarrow 1$
while $r_1 \neq 0$ **do**
    $d_0 \leftarrow \deg(r_0)$
    $d_1 \leftarrow \deg(r_1)$
    shift $\leftarrow d_0 - d_1$
    if shift < 0 **then**
        Swap $r_0 \leftrightarrow r_1$
        Swap $s_0 \leftrightarrow s_1$
        shift $\leftarrow -$shift
    end if
    $r_0 \leftarrow r_0 \oplus (r_1 \ll \text{shift})$
    $s_0 \leftarrow s_0 \oplus (s_1 \ll \text{shift})$
    Mask r_0 and s_0 to 9 bits: $r_0 \leftarrow r_0 \,\&\, 0x1FF$, $s_0 \leftarrow s_0 \,\&\, 0x1FF$
end while
if $r_0 = 1$ **then**
    return s_0 ▷ Inverse found
else
    return error ▷ No inverse exists
end if

The RV32 program can be computed simply by the following assembly as well as using HTIF to output the result of 0x67 for the given polynomial and modulus. This implementation takes 561 instructions to complete. Finite Field arithmetic is sometimes challenging with programs as there can be overflow since multiplication is carryless. Therefore, this code masks off 9 bits to stay in GF(2^8) and allow the computation to proceed.

```
.text

.global rvtest_entry_point
rvtest_entry_point:

    li a0, 0x43       # Input value to invert in GF(2^8)
    li t0, 0x11B      # Modulus: irreducible poly x^8 + x^4 + x^3 + x + 1

    # Initialize Extended Euclidean variables: a*x+b*y=gcd(a,b)
    li t1, 0          # s0 = 0 (old x)
    li t2, 1          # s1 = 1 (new x)
    mv t3, t0         # r0 = modulus
    mv t4, a0         # r1 = input value

    csrr s8, instret  # count instructions at beginning

    # --- Register usage ---
    # a0: Input / Final result (return value)
    # t0: Modulus (0x11B)
    # t1: s0 (previous x value)
    # t2: s1 (current x value)
    # t3: r0 (previous remainder)
    # t4: r1 (current remainder)
    # t5: degree(r0)
    # t6: degree(r1)
    # a1: argument to gf_degree
    # a2: shift amount
    # a3: temporary for SLT result
    # a4/a5: shift results, temporaries
    # s8: initial seed of instruction count (instret)
    # s9: final value of instruction count (instret)
```

```
# ---------------------------------------
# Main loop: Extended Euclidean Division
# ---------------------------------------
inv_loop:
   beqz t4, fail            # If r1 == 0, input is not invertible → fail

   # Compute degree of r0
   mv a1, t3                # Set a1 = r0 = (modulus m)
   call gf_degree           # Compute deg(r0)
   mv t5, a0                        # degree(r0)

   # Compute degree of r1
   mv a1, t4                # Set a1 = r1 = (input a)
   call gf_degree           # Compute deg(r1)
   mv t6, a0                        # degree (r1)

   # Compute shift = deg(r0) - deg(r1)
   sub a2, t5, t6           # alignment r1 with highest term in r0
   slt a3, a2, zero         # Check if shift < 0
   bnez a3, swap_and_negate

   # Perform r0 ^= r1 << shift
   # Polynomial subtraction in GF(2)
   sll a4, t4, a2
   xor t3, t3, a4
   andi t3, t3, 0x1FF       # Mask off 9 bits to stay in GF(2^8)

   # Update s0: s0 ^= s1 << shift - Update Bezout coefficient
   sll a5, t2, a2
   xor t1, t1, a5
   andi t1, t1, 0x1FF               # Mask off 9 bits to stay in GF(2^8)

   # Swap (r0, r1)
   mv a4, t3
   mv t3, t4
   mv t4, a4

   # Swap (s0, s1)
   mv a4, t1
   mv t1, t2
   mv t2, a4

   # Check if r0 == 1, then we are done
   li t5, 1
   beq t3, t5, done

   j inv_loop

# -------------------------------------------
# Case: shift < 0 → negate, then apply shift
# -------------------------------------------
swap_and_negate:
   # Swap r0 <-> r1
   mv a4, t3
   mv t3, t4
   mv t4, a4

   # Swap s0 <-> s1
   mv a4, t1
   mv t1, t2
   mv t2, a4

   # shift = -shift
   sub a2, zero, a2

   # r0 ^= r1 << shift
```

```
    sll a4, t4, a2
    xor t3, t3, a4
    andi t3, t3, 0x1FF          # Mask off to 9 bits again

    # s0 ^= s1 << shift
    sll a5, t2, a2
    xor t1, t1, a5
    andi t1, t1, 0x1FF

    # Final swap to maintain invariant
    mv a4, t3
    mv t3, t4
    mv t4, a4

    mv a4, t1
    mv t1, t2
    mv t2, a4

    # Check if r0 == 1
    li t5, 1
    beq t3, t5, done

    j inv_loop

# --------------------------------------
# Helper: compute degree of a1 (MSB set)
# --------------------------------------
gf_degree:
    li t0, 8                    # Start checking from MSB down
deg_loop:
    srl a0, a1, t0              # Right shift a1 by t0
    andi a0, a0, 1              # Isolate LSB
    bnez a0, done_deg           # If bit is set, we're done
    addi t0, t0, -1
    bgez t0, deg_loop           # Check next bit
    li a0, 0                    # Nothing set
    ret
done_deg:
    mv a0, t0
    ret

# ---------------------
# Finalize inverse
# ---------------------
done:
    mv a0, t1                   # Result is a0
    andi a0, a0, 0xFF           # Mask to 8 bits
    csrr s9, instret            # count instructions at end
    sub s9, s9, s8              # get number of #instructions executed

write_tohost:                   # HTIF stuff
    la t1, tohost
    li t0, 1                    # 1 for success, 3 for failure
    sw t0, 0(t1)                # send success code

    la t0, begin_signature      # Address of signature
    sw a0, 0(t0)                # Store result (i.e., a0)
    sw s9, 4(t0)                # record #instructions executed

fail:
    li a0, 0xff                 # Signal failure: no inverse exists

self_loop:
    j self_loop

.section .tohost
tohost:                         # write to HTIF
```

```
   .word 0
fromhost:
   .word 0

.data
.EQU XLEN,32
begin_signature:
   .fill 2*(XLEN/32),4,0xdeadbeef
end_signature:


.bss
            .space 512
```

18.7 Write a similar program as Exercise 18.6 but for SHA-256 with the message "Go Wally!". Show that you get the correct output. Report how many instructions were executed.

This is a fun exercise and similar to Exercise 18.6 making sure the message, K and H constants are properly put in little endian format is important. The number of instructions executed involves
$6 + (7\times16) + 3 + (15\times(64 - 16)) + 22 + (32\times64) + 26 = 2,937$ instructions, where 6 instructions are initially used to set up the counter t, 3 instructions are used to set up the counter for W[16] to W[63], 22 instructions to store the initial the working variables a though h, and 26 instructions to store the final result. The K extension does not have instructions to compute the complete Secure Hash mainly because of the complexity of the compression function. Consequently, this program only uses the sha256sig1, sha256sig0, sha256sum0, and sha256sum0 instructions to compute the σ_1, σ_0, Σ_1, and Σ_0, respectively.

```
.section .text
.global _start

_start:
   la sp, topofstack                    # Initialize stack pointer
   # Load initial hash state into a0–a7
   la t0, H
   lw a0, 0(t0)
   lw a1, 4(t0)
   lw a2, 8(t0)
   lw a3, 12(t0)
   lw a4, 16(t0)
   lw a5, 20(t0)
   lw a6, 24(t0)
   lw a7, 28(t0)

   # Store initial state in signature
   la s11, begin_signature
   csrr s10, instret                    # count instructions at beginning

   la s0, M                             # Load point to padded message
   la s1, W                             # See NIST 180-4 6.2.2
   li t1, 0                             # Initialize counter (t)

load_loop:
   lw t2, 0(s0)
   sw t2, 0(s1)
```

```
    addi s0, s0, 4                  # increment source M ptr
    addi s1, s1, 4                  # increment destination W ptr
    addi t1, t1, 1              # increment t
    li t3, 16
    blt t1, t3, load_loop

    # Let's now compute W[16] to W[63]
    la s1, W
    li t1, 16

gen_loop:
    slli t2, t1, 2              # addr{W[16]} <- 16 * 4 (XLEN=32
)
    add t3, s1, t2              # Compute addr{W[16]}

    lw t4, -8(t3)               # W_{t-2}
    lw t5, -28(t3)              # W_{t-7}
    lw t6, -60(t3)              # W_{t-15}
    lw t0, -64(t3)              # W_{t-16}

    sha256sig1 t4, t4               # Sigma1 transformation
    sha256sig0 t6, t6               # Sigma0 transformation

    add t4, t4, t5              # temp = Sigma1+W_{t-7}
    add t4, t4, t6              # temp = temp + Sigma0
    add t4, t4, t0              # Message Schedule W_t

    sw t4, 0(t3)                # Store W_t

    addi t1, t1, 1
    li t5, 64
    blt t1, t5, gen_loop

    # Working variables: s2–s9 = a-h
    addi sp, sp, -64                # Allocate 64 bytes on the stack
    mv s2, a0                       # a
    mv s3, a1                       # b
    mv s4, a2                       # c
    mv s5, a3                       # d
    mv s6, a4                       # e
    mv s7, a5                       # f
    mv s8, a6                       # g
    mv s9, a7                       # h

    sw s2,  0(sp)               # Save a on the stack
    sw s3,  4(sp)               # Save b on the stack
    sw s4,  8(sp)               # Save c on the stack
    sw s5, 12(sp)               # Save d on the stack
    sw s6, 16(sp)               # Save e on the stack
    sw s7, 20(sp)               # Save f on the stack
    sw s8, 24(sp)               # Save g on the stack
    sw s9, 28(sp)               # Save h on the stack

    li t0, 0                    # schedule counter
    la t1, K
    la t2, W

# Compression Function
sha_round:
    slli t3, t0, 2
    add t4, t1, t3
    lw t5, 0(t4)                    # K[t]

    add t4, t2, t3
    lw t6, 0(t4)                    # W[t]

    sha256sum1 a0, s6               # Sum1 transformation
```

```
    and a1, s6, s7                  # a1 = x & y
    not a2, s6                      # a2 = ~x
    and a2, a2, s8                          # a2 = ~x & z
    xor a1, a1, a2
    add a2, a0, a1
    add a2, a2, s9
    add a2, a2, t5                          # Ch = a1 = (x & y) ^ (~x & z) (
4 instructions)
    add a2, a2, t6                          # T1 = h + SUM1(e) + Ch(e,f,g) +
 K[i] + W[i]

    sha256sum0 a3, s2                       # Sum0 transformation
    and a4, s2, s3              # a4 = a & b
    and a5, s2, s4              # a5 = a & c
    xor a4, a4, a5              # a4 = (a & b) ^ (a & c)
    and a5, s3, s4              # a5 = b & c
    xor a4, a4, a5              # a4 = Maj(a, b, c) (5 instructions)
    add a5, a3, a4                          # T2 = SUM0(a) + Maj(a,b,c)

    # Compression Function Shift
    mv s9, s8                               # h
    mv s8, s7                               # g
    mv s7, s6                               # f
    add s6, s5, a2                          # e = d+T1
    mv s5, s4                               # d
    mv s4, s3                               # c
    mv s3, s2                               # b
    add s2, a2, a5                          # a = T1+T2

    addi t0, t0, 1                  # Modify branch ctr for t
    li a7, 64
    blt t0, a7, sha_round

    # Restore eight working variables
    lw a0,  0(sp)                   # Restore a0 (a)
    lw a1,  4(sp)                   # Restore a1 (b)
    lw a2,  8(sp)                   # Restore a2 (c)
    lw a3, 12(sp)                           # Restore a3 (d)
    lw a4, 16(sp)                           # Restore a4 (e)
    lw a5, 20(sp)                           # Restore a5 (f)
    lw a6, 24(sp)                           # Restore a6 (g)
    lw a7, 28(sp)                           # Restore a7 (h)
    addi sp, sp, 64                         # Deallocate stack space

    # Intermediate Hash (e.g., H_0^i = a + H_0^{i-1})
    add a0, a0, s2                          # H0
    add a1, a1, s3                          # H1
    add a2, a2, s4                          # H2
    add a3, a3, s5                          # H3
    add a4, a4, s6                          # H4
    add a5, a5, s7                          # H5
    add a6, a6, s8                          # H6
    add a7, a7, s9                          # H7

    # Print out hash value or digest
    sw a0, 0(s11)
    sw a1, 4(s11)
    sw a2, 8(s11)
    sw a3, 12(s11)
    sw a4, 16(s11)
    sw a5, 20(s11)
    sw a6, 24(s11)
    sw a7, 28(s11)
    addi s11, s11, 32

# Finalize
done:
```

```
    csrr s9, instret                    # Read instruction count
    sub s9, s9, s10                     # get number of #instructions ex
ecuted
    sw s9, 0(s11)                       # Print instruction count

write_tohost:
    la t1, tohost
    li t0, 1                            # Success code
    sw t0, 0(t1)                        # Send success code

self_loop:
    j self_loop

.section .tohost
tohost:
    .word 0
fromhost:
    .word 0

.data
.align 4

.EQU XLEN,32
begin_signature:
    .fill (8*1+1)*(XLEN/32),4,0xdeadbeef
end_signature:

# Initial SHA-256 hash values (NIST 180-4 5.3.3)
H:
    .word 0x6a09e667, 0xbb67ae85, 0x3c6ef372, 0xa54ff53a
    .word 0x510e527f, 0x9b05688c, 0x1f83d9ab, 0x5be0cd19

# K constants for SHA-256 (NIST 180-4 4.2.2)
K:
    .word 0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5
    .word 0x3956c25b, 0x59f111f1, 0x923f82a4, 0xab1c5ed5
    .word 0xd807aa98, 0x12835b01, 0x243185be, 0x550c7dc3
    .word 0x72be5d74, 0x80deb1fe, 0x9bdc06a7, 0xc19bf174

    .word 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc
    .word 0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da
    .word 0x983e5152, 0xa831c66d, 0xb00327c8, 0xbf597fc7
    .word 0xc6e00bf3, 0xd5a79147, 0x06ca6351, 0x14292967

    .word 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc, 0x53380d13
    .word 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85
    .word 0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3
    .word 0xd192e819, 0xd6990624, 0xf40e3585, 0x106aa070

    .word 0x19a4c116, 0x1e376c08, 0x2748774c, 0x34b0bcb5
    .word 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f, 0x682e6ff3
    .word 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208
    .word 0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2

# One padded 512-bit block for message "Go Wally!"
M:
    .word 0x476f2057, 0x616c6c79, 0x21800000, 0x00000000
    .word 0x00000000, 0x00000000, 0x00000000, 0x00000000
    .word 0x00000000, 0x00000000, 0x00000000, 0x00000000
    .word 0x00000000, 0x00000000, 0x00000000, 0x00000048

# Space for W[0..63] (64 x 4 bytes)
W:
     .skip 256

.bss
    .space 512
```

topofstack:

18.9 Compute the AES-128 encryption of a block in RV64 in C. Use the methods described in Section 3.4.1 to compute the number of cycles and instructions and verify the results.

There are many available implementations in C for AES-128. For this problem, a custom aes128.c program is created. Using the techniques discussed in Section 3.4.1 and using the -O3 flag, this amounts to 5,991 instructions with RV64.

```
// Implementation: Key Expansion
enum keySize {
  SIZE_16 = 16,
  SIZE_24 = 24,
  SIZE_32 = 32
};

// main Function Prototypes
void KeySchedule(unsigned char *word, int iteration);
unsigned char getRconValue(unsigned char num);
void rotate(unsigned char *word);
// AES Encryption Function Prototypes
void subBytes(unsigned char *state);
void shiftRows(unsigned char *state);
void shiftRow(unsigned char *state, unsigned char nbr);
void addRoundKey(unsigned char *state, unsigned char *roundKey);
unsigned char galois_multiplication(unsigned char a, unsigned char b);
void mixColumns(unsigned char *state);
void mixColumn(unsigned char *column);
void aes_cipher(unsigned char *state, unsigned char *roundKey);
void createRoundKey(unsigned char *expandedKey, unsigned char *roundKey);
void aes_main(unsigned char *state, unsigned char *expandedKey, int nbrRounds);
char aes_encrypt(unsigned char *input, unsigned char *output, unsigned char *key, enum keySize size);
// AES Decryption Function Prototypes
void invSubBytes(unsigned char *state);
void invShiftRows(unsigned char *state);
void invShiftRow(unsigned char *state, unsigned char nbr);
void invMixColumns(unsigned char *state);
void invMixColumn(unsigned char *column);
void aes_invRound(unsigned char *state, unsigned char *roundKey);
void aes_invCipher(unsigned char *state, unsigned char *expandedKey, int nbrRounds);
char aes_decrypt(unsigned char *input, unsigned char *output, unsigned char *key, enum keySize size);

static int verifyChar(int n, const volatile char* test, const char* verify) {
   int i;
   // Unrolled for faster verification
   for (i = 0; i < n / 2 * 2; i += 2)
   {
      char t0 = test[i], t1 = test[i + 1];
      char v0 = verify[i], v1 = verify[i + 1];
      if (t0 != v0) return i + 1;
      if (t1 != v1) return i + 2;
   }
   if (n % 2 != 0 && test[n - 1] != verify[n - 1])
      return n;
   return 0;
}



// Helper function to print AES state
void printState(const unsigned char *state) {
  for (int col = 0; col < 4; col++) {
    printf("%02x %02x %02x %02x ",
         state[col], state[col + 4], state[col + 8], state[col + 12]);
```

```
  }
  printf("\n");
}

unsigned char getSBoxValue(unsigned char num) {
  return sbox[num];
}

unsigned char getSBoxInvert(unsigned char num) {
  return rsbox[num];
}

// left circular rotation (i.e., byte-wise rotate left) on a 4-byte word
void rotate(unsigned char *word) {
  unsigned char temp = word[0];
  word[0] = word[1];
  word[1] = word[2];
  word[2] = word[3];
  word[3] = temp;
}

unsigned char getRconValue(unsigned char num) {
  return Rcon[num];
}

// Key Schedule: RotWord → SubWord → XOR with Rcon
void KeySchedule(unsigned char *word, int iteration) {
  int i;

  // rotate the 32-bit word 8 bits to the left
  rotate(word);
  // apply S-Box substitution on all 4 parts of the 32-bit word
  for (i = 0; i < 4; ++i) {
    word[i] = getSBoxValue(word[i]);
  }
  // XOR the output of the rcon operation with i to the first part (leftmost) only
  word[0] = word[0] ^ getRconValue(iteration);
}

// Rijndael's key expansion:  expands a 128, 192, 256 key into a 176, 208, 240 bytes key
void KeyExpansion(unsigned char *expandedKey, unsigned char *key,
                  enum keySize size, unsigned int expandedKeySize) {

  int currentSize = 0;
  int rconIteration = 1;
  int i;
  unsigned char t[4] = {0}; // temporary 4-byte variable

  // set the 16, 24,32 bytes of the expanded key to the input key
  for (i = 0; i < size; i++)
    expandedKey[i] = key[i];
  currentSize += size;
  while (currentSize < expandedKeySize) {
    // assign the previous 4 bytes to the temporary value t
    for (i = 0; i < 4; i++) {
      t[i] = expandedKey[(currentSize - 4) + i];
    }
    // every 16, 24, 32 bytes we apply the core schedule to t
    // and increment rconIteration afterwards
    if (currentSize % size == 0) {
      KeySchedule(t, rconIteration++);
    }
    // For 256-bit keys, we add an extra sbox to the calculation
    if (size == SIZE_32 && ((currentSize % size) == 16)) {
      for (i = 0; i < 4; i++)
        t[i] = getSBoxValue(t[i]);
    }
```

```
    // We XOR t with the four-byte block 16, 24, 32 bytes before the new expanded key.
    // This becomes the next four bytes in the expanded key.
    for (i = 0; i < 4; i++) {
      expandedKey[currentSize] = expandedKey[currentSize - size] ^ t[i];
      currentSize++;
    }
  }
}

void subBytes(unsigned char *state) {
  for (int i = 0; i < 16; i++) {
    state[i] = getSBoxValue(state[i]);
  }
}

void shiftRows(unsigned char *state) {
  for (int row = 0; row < 4; row++) {
    shiftRow(state + row * 4, row);
  }
}

void shiftRow(unsigned char *state, unsigned char nbr) {
  unsigned char temp[4];

  // Perform rotation directly
  for (int i = 0; i < 4; i++) {
    temp[i] = state[(i + nbr) % 4];
  }

  // Copy result back to state
  for (int i = 0; i < 4; i++) {
    state[i] = temp[i];
  }
}

void addRoundKey(unsigned char *state, unsigned char *roundKey) {
  int i;
  for (i = 0; i < 16; i++)
    state[i] = state[i] ^ roundKey[i];
}

// Based on work by Tom St. Denis/Simon Johnson
// https://www.amazon.com/Cryptography-Developers-Tom-St-Denis/dp/1597491047
unsigned char gfmul(unsigned char a, unsigned char b) {
  unsigned char p = 0;
  for (unsigned char counter = 0; counter < 8; counter++) {
    if (b & 1)
      p ^= a;
    a = (a << 1) ^ ((a & 0x80) ? 0x1b : 0);
    b >>= 1;
  }
  return p;
}

// Section 5.1.3 of FIPS 197
void mixColumns(unsigned char *state) {
  unsigned char column[4];

  for (int col = 0; col < 4; col++) {
    // Extract the column from the state
    for (int row = 0; row < 4; row++) {
      column[row] = state[row * 4 + col];
    }

    // Mix the column
    mixColumn(column);
```

```
    // Store the mixed column back into the state
    for (int row = 0; row < 4; row++) {
      state[row * 4 + col] = column[row];
    }
  }
}

void mixColumn(unsigned char *column) {
  unsigned char a[4], b[4];

  for (int i = 0; i < 4; i++) {
    a[i] = column[i];
    b[i] = gfmul(column[i], 2);
  }

  column[0] = b[0] ^ a[3] ^ a[2] ^ gfmul(a[1], 3);
  column[1] = b[1] ^ a[0] ^ a[3] ^ gfmul(a[2], 3);
  column[2] = b[2] ^ a[1] ^ a[0] ^ gfmul(a[3], 3);
  column[3] = b[3] ^ a[2] ^ a[1] ^ gfmul(a[0], 3);
}

// The rounds in the specifcation of aes_cipher() are composed of the following 4 byte-oriented
// transformations on the state (Section 5.1 of FIPS 197) - outputs hex after each step
void aes_cipher(unsigned char *state, unsigned char *roundKey) {
  subBytes(state);

  shiftRows(state);

  mixColumns(state);

  addRoundKey(state, roundKey);
}

void createRoundKey(unsigned char *expandedKey, unsigned char *roundKey) {
  int i, j;
  // iterate over the columns
  for (i = 0; i < 4; i++) {
    // iterate over the rows
    for (j = 0; j < 4; j++)
      roundKey[(i + (j * 4))] = expandedKey[(i * 4) + j];
  }
}

void aes_main(unsigned char *state, unsigned char *expandedKey, int nbrRounds) {
  unsigned char roundKey[16];

  // Initial round key
  createRoundKey(expandedKey, roundKey);
  addRoundKey(state, roundKey);

  for (int i = 1; i < nbrRounds; i++) {
    createRoundKey(expandedKey + 16 * i, roundKey);
    aes_cipher(state, roundKey);
  }

  // Final round (no MixColumns)
  createRoundKey(expandedKey + 16 * nbrRounds, roundKey);
  subBytes(state);

  shiftRows(state);

  addRoundKey(state, roundKey);
}

char aes_encrypt(unsigned char *input, unsigned char *output,
                 unsigned char *key, enum keySize size) {
```

```
  int nbrRounds;
  int expandedKeySize;
  unsigned char block[16];

  // Determine number of rounds based on key size
  switch (size) {
    case SIZE_16: nbrRounds = 10; break;
    case SIZE_24: nbrRounds = 12; break;
    case SIZE_32: nbrRounds = 14; break;
    default: return ERROR_AES_UNKNOWN_KEYSIZE;
  }

  expandedKeySize = 16 * (nbrRounds + 1);
  unsigned char expandedKey[expandedKeySize];

  // Map input (row-major to column-major for AES)
  for (int row = 0; row < 4; row++) {
    for (int col = 0; col < 4; col++) {
      block[row + 4 * col] = input[4 * row + col];
    }
  }

  // Expand the key
  KeyExpansion(expandedKey, key, size, expandedKeySize);
  // Encrypt the block
  aes_main(block, expandedKey, nbrRounds);
  // Map block back to output (column-major to row-major)
  for (int row = 0; row < 4; row++) {
    for (int col = 0; col < 4; col++) {
      output[4 * row + col] = block[row + 4 * col];
    }
  }

  return SUCCESS;
}

void invSubBytes(unsigned char *state) {
  for (int i = 0; i < 16; i++) {
    state[i] = getSBoxInvert(state[i]);
  }
}

void invShiftRows(unsigned char *state) {
  for (int row = 0; row < 4; row++) {
    invShiftRow(state + row * 4, row);
  }
}

void invShiftRow(unsigned char *state, unsigned char nbr) {
  unsigned char temp[4];

  // Perform rotation to the right by `nbr` positions
  for (int i = 0; i < 4; i++) {
    temp[i] = state[(i - nbr + 4) % 4];
  }

  // Copy back the rotated values
  for (int i = 0; i < 4; i++) {
    state[i] = temp[i];
  }
}

void invMixColumns(unsigned char *state) {
  unsigned char column[4];

  for (int col = 0; col < 4; col++) {
    // Extract one column (4 bytes from each row)
```

```
    for (int row = 0; row < 4; row++) {
      column[row] = state[row * 4 + col];
    }

    // Apply inverse MixColumn transformation
    invMixColumn(column);

    // Store transformed column back into the state
    for (int row = 0; row < 4; row++) {
      state[row * 4 + col] = column[row];
    }
  }
}

void invMixColumn(unsigned char *column) {
  unsigned char a[4];

  for (int i = 0; i < 4; i++)
    a[i] = column[i];

  unsigned char a0_14 = gfmul(a[0], 14);
  unsigned char a1_11 = gfmul(a[1], 11);
  unsigned char a2_13 = gfmul(a[2], 13);
  unsigned char a3_9  = gfmul(a[3],  9);

  unsigned char a0_9  = gfmul(a[0],  9);
  unsigned char a1_14 = gfmul(a[1], 14);
  unsigned char a2_11 = gfmul(a[2], 11);
  unsigned char a3_13 = gfmul(a[3], 13);

  unsigned char a0_13 = gfmul(a[0], 13);
  unsigned char a1_9  = gfmul(a[1],  9);
  unsigned char a2_14 = gfmul(a[2], 14);
  unsigned char a3_11 = gfmul(a[3], 11);

  unsigned char a0_11 = gfmul(a[0], 11);
  unsigned char a1_13 = gfmul(a[1], 13);
  unsigned char a2_9  = gfmul(a[2],  9);
  unsigned char a3_14 = gfmul(a[3], 14);

  column[0] = a0_14 ^ a1_11 ^ a2_13 ^ a3_9;
  column[1] = a0_9  ^ a1_14 ^ a2_11 ^ a3_13;
  column[2] = a0_13 ^ a1_9  ^ a2_14 ^ a3_11;
  column[3] = a0_11 ^ a1_13 ^ a2_9  ^ a3_14;
}

void aes_invRound(unsigned char *state, unsigned char *roundKey) {

  invShiftRows(state);
  invSubBytes(state);
  addRoundKey(state, roundKey);
  invMixColumns(state);
}

void aes_invCipher(unsigned char *state, unsigned char *expandedKey, int nbrRounds) {
  unsigned char roundKey[16];

  // Initial round key for final round
  createRoundKey(expandedKey + 16 * nbrRounds, roundKey);
  addRoundKey(state, roundKey);

  // Inverse rounds (excluding final round)
  for (int round = nbrRounds - 1; round > 0; round--) {
    createRoundKey(expandedKey + 16 * round, roundKey);
    aes_invRound(state, roundKey);
  }
```

```
  // Final inverse round (no invMixColumns)
  createRoundKey(expandedKey, roundKey);
  invShiftRows(state);
  invSubBytes(state);
  addRoundKey(state, roundKey);
}

char aes_decrypt(unsigned char *input, unsigned char *output,
                 unsigned char *key, enum keySize size) {
  int nbrRounds;
  int expandedKeySize;
  unsigned char block[16];

  // Determine the number of rounds based on key size
  switch (size) {
    case SIZE_16: nbrRounds = 10; break;
    case SIZE_24: nbrRounds = 12; break;
    case SIZE_32: nbrRounds = 14; break;
    default: return ERROR_AES_UNKNOWN_KEYSIZE;
  }

  expandedKeySize = 16 * (nbrRounds + 1);
  unsigned char expandedKey[expandedKeySize];

  // Transpose input (column-major block for AES)
  for (int row = 0; row < 4; row++) {
    for (int col = 0; col < 4; col++) {
      block[row + 4 * col] = input[4 * row + col];
    }
  }

  // Key expansion
  KeyExpansion(expandedKey, key, size, expandedKeySize);

  // Decrypt
  aes_invCipher(block, expandedKey, nbrRounds);

  // Transpose back into output (row-major)
  for (int row = 0; row < 4; row++) {
    for (int col = 0; col < 4; col++) {
      output[4 * row + col] = block[row + 4 * col];
    }
  }

  return SUCCESS;
}

int main(int argc, char *argv[]) {

  // FIPS 197-2 Appendix A/B results for ciphertext
  unsigned char expected[16] = {
     0x39, 0x25, 0x84, 0x1d, 0x02, 0xdc, 0x09, 0xfb,
     0xdc, 0x11, 0x85, 0x97, 0x19, 0x6a, 0x0b, 0x32};

  // Rounds: Number of AES rounds
  // Words/Key: Number of 32-bit words per round key (Nb = 4 for AES)
  // Round Keys: Total words in expanded key = Nb × (Rounds + 1)
  // KeyExp (B): Key expansion size in bytes = Round Keys × 4
  // Block (B): Block size in bytes = 128 bits / 8 = 16

  //+-----------+ Block Size | Rounds | Words/Key | Round Keys | KeyExp (B) | Block (B) |
  //|-----------|------------|--------|-----------|------------|------------|-----------|
  //| AES-128   | 128 bits   | 10     | 4         | 44         | 176        | 16        |
  //| AES-192   | 128 bits   | 12     | 4         | 52         | 208        | 16        |
  //| AES-256   | 128 bits   | 14     | 4         | 60         | 240        | 16        |
  //+-----------+------------+--------+-----------+------------+------------+-----------+
```

```
  // the expanded keySize to store full set of round keys (change according to table)
  int expandedKeySize = 176;

  // the cipher key size defined on Line 81 (change for different AES key)
  enum keySize size = SIZE_16;

  unsigned char expandedKey[expandedKeySize];

  // the cipher key (FIPS 197 example (page 28) 128-bit Cipher Key in Appendix A)
  unsigned char key[16] = {0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
                           0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};

  // (FIPS 197 example (page 34) in Appendix B)
  // AES uses an internal structure called the state, which is a 4x4 byte matrix (16 bytes total).
  // AES operates on 128-bit blocks (16 bytes), always — regardless of key size (128, 192, 256).
  unsigned char plaintext[16] = {0x32, 0x43, 0xf6, 0xa8, 0x88, 0x5a, 0x30, 0x8d,
                                 0x31, 0x31, 0x98, 0xa2, 0xe0, 0x37, 0x07, 0x34};

  unsigned char ciphertext[16];
  unsigned char decryptedtext[16];
  int i;

  KeyExpansion(expandedKey, key, size, expandedKeySize);

  // AES Encryption


      setStats(1);        // record initial mcycle and minstret
  aes_encrypt(plaintext, ciphertext, key, size);
  setStats(0);        // record elapsed mcycle and minstret
  return verifyChar(16, ciphertext, expected);

  printf("\nCiphertext (hex format):\n");
  for (i = 0; i < 16; i++) {
    printf("%02x%c", ciphertext[i], ((i + 1) % 16) ? ' ' : '\n');
  }

  // AES Decryption
  aes_decrypt(ciphertext, decryptedtext, key, size);
  printf("\nDecrypted text (hex format):\n");
  for (i = 0; i < 16; i++) {
    printf("%2.2x%c", decryptedtext[i], ((i + 1) % 16) ? ' ' : '\n');
  }

  return 0;
}
```

Chapter 19 Solutions

n/a

Chapter 20 Solutions

n/a

Chapter 21 Solutions

n/a

Chapter 22 Solutions

n/a

Chapter 23 Solutions

n/a