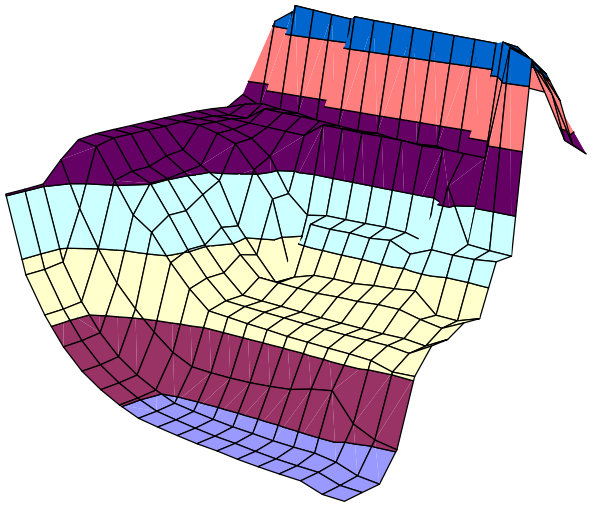
Computer Systems: A Programmer's Perspective, Second Edition

Instructor's Solution Manual, Version A [1]



Randal E. Bryant
David R. O'Hallaron

February 4, 2010

Chapter 1

Solutions to Homework Problems

This document contains solutions for the North American version of the Second Edition of the book (we call this "Version A"). A separate document contains solutions for the International versions (Version B).

The text uses two different kinds of exercises:

- *Practice Problems.* These are problems that are incorporated directly into the text, with explanatory solutions at the end of each chapter. Our intention is that students will work on these problems as they read the book. Each one highlights some particular concept.
- *Homework Problems.* These are found at the end of each chapter. They vary in complexity from simple drills to multi-week labs and are designed for instructors to give as assignments or to use as recitation examples.

This document gives the solutions to the homework problems.

1.1 Chapter 1: A Tour of Computer Systems

There are no homework problems in this chapter.

1.2 Chapter 2: Representing and Manipulating Information

Problem 2.57 Solution:

This exercise should be a straightforward variation on the existing code.

code/data/show-ans.c

```
1 void show_short(short x) {
2     show_bytes((byte_pointer) &x, sizeof(short));
3 }
4
```

```
 5 void show_long(long x) {
 6     show_bytes((byte_pointer) &x, sizeof(long));
 7 }
 8
 9 void show_double(double x) {
10     show_bytes((byte_pointer) &x, sizeof(double));
11 }
```

code/data/show-ans.c

Problem 2.58 Solution:

There are many ways to solve this problem. The basic idea is to create some multibyte datum with different values for the most and least-significant bytes. We then read byte 0 and determine which byte it is.

The following solution creates an `int` with value 1. We then access its first byte and convert it to an `int`. This byte will equal 0 on a big-endian machine and 1 on a little-endian machine.

code/data/show-ans.c

```
1 int is_little_endian(void) {
2     /* MSB = 0, LSB = 1 */
3     int x = 1;
4
5     /* Return MSB when big-endian, LSB when little-endian */
6     return (int) (* (char *) &x);
7 }
```

code/data/show-ans.c

Problem 2.59 Solution:

This is a simple exercise in masking and bit manipulation. It is important to mention that `~0xFF` is a way to generate a mask that selects all but the least significant byte that works for any word size.

```
(x & 0xFF) | (y & ~0xFF)
```

Problem 2.60 Solution:

Byte extraction and insertion code is useful in many contexts. Being able to write this sort of code is an important skill to foster.

code/data/rbyte-ans.c

```
1 unsigned replace_byte (unsigned x, int i, unsigned char b) {
2     int itimes8 = i << 3;
3     unsigned mask = 0xFF << itimes8;
4
5     return (x & ~mask) | (b << itimes8);
6 }
```

code/data/rbyte-ans.c

Problem 2.61 Solution:

These exercises require thinking about the logical operation `!` in a nontraditional way. Normally we think of it as logical negation. More generally, it detects whether there is any nonzero bit in a word. In addition, it gives practice with masking.

A. `!!x`

B. `!!~x`

C. `!!(x & 0xFF)`

D. `!!(~x & (0xFF << ((sizeof(int)-1)<<3)))`

Problem 2.62 Solution:

There are many solutions to this problem, but it is a little bit tricky to write one that works for any word size. Here is our solution:

code/data/shift-ans.c

```
1 int int_shifts_are_arithmetic() {
2     int x = -1; /* All 1's */
3     return (x >> 1) < 0;
4 }
```

code/data/shift-ans.c

The above code performs a right shift of a word in which all bits are set to 1. If the shift is arithmetic, the resulting word will still have all bits set to 1 and hence be negative.

Problem 2.63 Solution:

These problems are fairly tricky. They require generating masks based on the shift amounts. Shift value 0 must be handled as a special case, since otherwise we would be generating the mask by performing a left shift by 32.

code/data/rshift-ans.c

```
1 unsigned srl(unsigned x, int k)
2 {
3     /* Perform shift arithmetically */
4     unsigned xsra = (int) x >> k;
5     /* Make mask of low order w-k bits */
6     unsigned mask = k ? ((1 << (8*sizeof(int)-k)) - 1) : ~0;
7
8     return xsra & mask;
9 }
```

code/data/rshift-ans.c

code/data/rshift-ans.c

```
1 int sra(int x, int k)
2 {
3     /* Perform shift logically */
4     int xsrl = (unsigned) x >> k;
5     /* Make mask of high order k bits */
6     unsigned mask = k ? ~((1 << (8*sizeof(int)-k)) - 1) : 0;
7
8     return (x < 0) ? mask | xsrl : xsrl;
9 }
```

code/data/rshift-ans.c

Problem 2.64 Solution:

This problem is very simple, but it reinforces the idea of using different bit patterns as masks.

code/data/bits.c

```
1 /* Return 1 when any odd bit of x equals 1; 0 otherwise.  Assume w=32 */
2 int any_odd_one(unsigned x) {
3     /* Use mask to select odd bits */
4     return (x&0xAAAAAAAA) != 0;
5 }
```

code/data/bits.c

Problem 2.65 Solution:

This is a classic "bit puzzle" problem, and the solution is actually useful. The trick is to use the bit-level parallelism of the ^ operation to combine multiple bits at a time. The solution, using just 11 operations, is as follows:

code/data/bits.c

```
1 /* Return 1 when x contains an odd number of 1s; 0 otherwise.  Assume w=32 */
2 int odd_ones(unsigned x) {
3     /* Use bit-wise ^ to compute multiple bits in parallel */
4     /* Xor bits i and i+16 for 0 <= i < 16 */
5     unsigned p16 = (x  >> 16) ^ x;
6     /* Xor bits i and i+8 for 0 <= i < 8 */
7     unsigned p8 =  (p16>>  8) ^ p16;
8     /* Xor bits i and i+4 for 0 <= i < 4 */
9     unsigned p4 =  (p8 >>  4) ^ p8;
10    /* Xor bits i and i+2 for 0 <= i < 2 */
11    unsigned p2 =  (p4 >>  2) ^ p4;
12    /* Xor bits 0 and 1 */
13    unsigned p1 =  (p2 >>  1) ^ p2;
14    /* Answer is in least significant bit */
15    return p1 & 1;
16 }
```

code/data/bits.c

Problem 2.66 Solution:

The key idea is given in the hint. We can create a cascade of 1's to the right—first one, then two, then four, up to half the word size, using just 10 operations.

code/data/bits.c

```
/*
 * Generate mask indicating leftmost 1 in x.
 * For example 0xFF00 -> 0x8000, and 0x6600 --> 0x4000
 * If x = 0, then return 0.
 */
int leftmost_one(unsigned x) {
    /* First, convert to pattern of the form 0...011...1 */
    x |= (x>>1);
    x |= (x>>2);
    x |= (x>>4);
    x |= (x>>8);
    x |= (x>>16);
    /* Now knock out all but leading 1 bit */
    x ^= (x>>1);
    return x;
}
```

code/data/bits.c

Problem 2.67 Solution:

This problem illustrates some of the challenges of writing portable code. The fact that `1<<32` yields 0 on some 32-bit machines and 1 on others is common source of bugs.

A. The C standard does not define the effect of a shift by 32 of a 32-bit datum. On the SPARC (and many other machines), the expression `x << k` shifts by $k \bmod 32$, i.e., it ignores all but the least significant 5 bits of the shift amount. Thus, the expression `1 << 32` yields 1.

B. Compute `beyond_msb` as `2 << 31`.

C. We cannot shift by more than 15 bits at a time, but we can compose multiple shifts to get the desired effect. Thus, we can compute `set_msb` as `2 << 15 << 15`, and `beyond_msb` as `set_msb << 1`.

Problem 2.68 Solution:

Here is the code:

code/data/bits.c

```
/*
```

```
 2  * Mask with least signficant n bits set to 1
 3  * Examples: n = 6 --> 0x2F, n = 17 --> 0x1FFFF
 4  * Assume 1 <= n <= w
 5  */
 6 int lower_one_mask(int n) {
 7     /*
 8      * 2^n-1 has bit pattern 0...01..1 (n 1's)
 9      * But, we must avoid a shift by 32
10      */
11     return (2<<(n-1)) - 1;
12 }
```

code/data/bits.c

The code makes use of the trick that `(1<<n)-1` creates a mask of `n` ones. The only challenge is to avoid shifting by w when $n = w$. Instead of writing `1<<n`, we write `2<<(n-1)`. This code will not work for $n = 0$, but that's not a very useful case, anyhow.

Problem 2.69 Solution:

code/data/bits.c

```
 1 /*
 2  * Do rotating left shift.  Assume 0 <= n < w
 3  * Examples when x = 0x12345678:
 4  *    n=4 -> 0x23456781, n=20 -> 0x67812345
 5  */
 6 unsigned rotate_left(unsigned x, int n) {
 7     /* Mask all 1's when n = 0 and all 0's otherwise */
 8     int z_mask = -!n;
 9     /* Left w-n bits */
10     unsigned left  = x << n;
11     /* Right n bits */
12     unsigned right = x >> ((sizeof(unsigned)<<3)-n);
13     return (z_mask&x) | (~z_mask &(left|right));
14 }
```

code/data/bits.c

For the most part, this problem requires simple shifting and masking. We must treat the case of $n = 0$ as special, because we would otherwise attempt to shift by w. Instead, we generate this solution explicitly, and use masks of all ones and all zeros to select between the special and general case.

Problem 2.70 Solution:

The code is as follows:

code/data/bits.c

```
 1 /*
 2  * Return 1 when x can be represented as an n-bit, 2's complement number;
```

```
 3  * 0 otherwise
 4  * Assume 1 <= n <= w
 5  */
 6 int fits_bits(int x, int n) {
 7     /*
 8      * Use left shift then right shift
 9      * to sign extend from n bits to full int
10      */
11     int count = (sizeof(int)<<3)-n;
12     int leftright = (x << count) >> count;
13     /* See if still have same value */
14     return x == leftright;
15 }
```

code/data/bits.c

This code uses a common trick, demonstrated in Problem 2.23, of first shifting left by some amount k and then arithmetically shifting right by k. This has the effect of sign-extending from bit $w-k-1$ leftward.

Problem 2.71 Solution:

This problem highlights the difference between zero extension and sign extension.

A. The function does not perform any sign extension. For example, if we attempt to extract byte 0 from word `0xFF`, we will get 255, rather than -1.

B. The following code uses the trick shown in Problem 2.23 to isolate a particular range of bits and to perform sign extension at the same time. First, we perform a left shift so that the most significant bit of the desired byte is at bit position 31. Then we right shift by 24, moving the byte into the proper position and performing sign extension at the same time.

code/data/xbyte.c

```
1 int xbyte(packed_t word, int bytenum) {
2     int left = word << ((3-bytenum) << 3);
3     return left >> 24;
4 }
```

code/data/xbyte.c

Problem 2.72 Solution:

This code illustrates the hidden dangers of data type `size_t`, which is defined to be `unsigned` on most machines.

A. Since this one data value has type `unsigned`, the entire expression is evaluated according to the rules of unsigned arithmetic. As a result, the conditional expression will always succeed, since every value is greater or equal to 0.

B. The code can be corrected by rewriting the conditional test:

```
        if (maxbytes >= sizeof(val))
```

Problem 2.73 Solution:

Here is the solution.

code/data/bits.c

```
1 /* Addition that saturates to TMin or TMax */
2 int saturating_add(int x, int y) {
3     int sum = x + y;
4     int wm1 = (sizeof(int)<<3)-1;
5     /* In the following we create "masks" consisting of all 1's
6        when a condition is true, and all 0's when it is false */
7     int xneg_mask = (x >> wm1);
8     int yneg_mask = (y >> wm1);
9     int sneg_mask = (sum >> wm1);
10    int pos_over_mask = ~xneg_mask & ~yneg_mask & sneg_mask;
11    int neg_over_mask = xneg_mask & yneg_mask & ~sneg_mask;
12    int over_mask = pos_over_mask | neg_over_mask;
13    /* Choose between sum, INT_MAX, and INT_MIN */
14    int result =
15        (~over_mask & sum)|
16        (pos_over_mask & INT_MAX)|(neg_over_mask & INT_MIN);
17    return result;
18 }
```

code/data/bits.c

Logically, this code is a straightforward application of the overflow rules for two's complement addition. Avoiding conditionals, however, requires expressing the conditions in terms of masks consisting of all zeros or all ones.

Problem 2.74 Solution:

code/data/taddcheck.c

```
1 /* Determine whether arguments can be subtracted without overflow */
2 int tsub_ok(int x, int y) {
3     int diff = x-y;
4     int neg_over = x <  0 && y >= 0 && diff >= 0;
5     int pos_over = x >= 0 && y <  0 && diff <  0;
6     return !neg_over && !pos_over;
7 }
```

code/data/taddcheck.c

This is a straightforward application of the rules for addition, modified to change the conditions for argument `y`. This avoids the shortcoming of the proposed solution given in Problem 2.32.

Problem 2.75 Solution:

This problem requires a fairly deep understanding of two's complement arithmetic. Some machines only provide one form of multiplication, and hence the trick shown in the code here is actually required to implement the alternate form.

As seen in Equation 2.18 we have $x' \cdot y' = x \cdot y + (x_{w-1}y + y_{w-1}x)2^w + x_{w-1}y_{w-1}2^{2w}$. The final term has no effect on the $2w$-bit representation of $x' \cdot y'$, but the middle term represents a correction factor that must be added to the high order w bits. This is implemented as follows:

code/data/uhp-ans.c

```
1 unsigned unsigned_high_prod(unsigned x, unsigned y) {
2     unsigned p = (unsigned) signed_high_prod((int) x, (int) y);
3
4     if ((int) x < 0) /* x_{w-1} = 1 */
5         p += y;
6     if ((int) y < 0) /* y_{w-1} = 1 */
7         p += x;
8     return p;
9 }
```

code/data/uhp-ans.c

Problem 2.76 Solution:

Patterns of the kind shown here frequently appear in compiled code.

A. $K = 17$: `(x<<4) + x`
B. $K = -7$: `-(x<<3) + x`
C. $K = 60$: `(x<<6) - (x<<2)`
D. $K = -112$: `-(x<<7) + (x<<4)`

Problem 2.77 Solution:

The code follows the method described in 2.3.7 for dividing by a power of two using arithmetic right shift. The only challenge is to do correct biasing within the constraints of the coding rules.

code/data/bits.c

```
1 /* Divide by power of two.  Assume 0 <= k < w-1 */
2 int divide_power2(int x, int k) {
3     /* All 1's if x < 0 */
4     int mask = x>>((sizeof(int)<<3)-1);
5     int bias = mask & ((1<<k)-1);
6     return (x+bias)>>k;
7 }
```

code/data/bits.c

Problem 2.78 Solution:

This demonstrates the use of shifting for both multiplication and division. The only challenge is to compute the bias using the limited operations allowed by the coding rules.

code/data/bits.c

```
1 /* Compute 3*x/4 */
2 int mul3div4(int x) {
3     int mul3 = x + (x<<1);
4     int mul3_mask = mul3 >> ((sizeof(int)<<3)-1);
5     int bias = mul3_mask & 3;
6     return (mul3+bias)>>2;
7 }
```

code/data/bits.c

Problem 2.79 Solution:

The requirement that the function must not overflow makes this problem more challenging than Problem 2.78. The idea in our solution is to compute the lower 2 bits, including the bias separately, to derive a value `incr` that will be either 0, 1, or 2, that can be added to the remaining bits of `3*x`.

code/data/bits.c

```
1 /* Compute 3/4*x with no overflow */
2 int threefourths(int x) {
3     int xl2 = x & 0x3;
4     int xl1 = (x&1) << 1;
5     int x_mask = x >> ((sizeof(int)<<3)-1);
6     int bias = x_mask & 3;
7     int incr = (xl2+xl1+bias) >> 2;
8     int s2 = x >> 2;
9     int s1 = x >> 1;
10    return s1 + s2 + incr;
11 }
```

code/data/bits.c

Problem 2.80 Solution:

Bit patterns similar to these arise in many applications. Many programmers provide them directly in hexadecimal, but it would be better if they could express them in more abstract ways.

A. $1^{w-k}0^k$.

```
~((1 << k) - 1)
```

B. $0^{w-k-j}1^k0^j$.

```
((1 << k) - 1) << j
```

Problem 2.81 Solution:

These "C puzzle" problems are a great way to motivate students to think about the properties of computer arithmetic from a programmer's perspective. Our standard lecture on computer arithmetic starts by showing a set of C puzzles. We then go over the answers at the end.

A. `(x<y) == (-x>-y)`. No, Let `x` = $TMin_{32}$, `y` = 0.

B. `((x+y)<<4) + y-x == 17*y+15*x`. Yes, from the ring properties of two's complement arithmetic.

C. `~x+~y+1 == ~(x+y)`. Yes, $\tilde{}x+\tilde{}y+1 = (-x-1)+(-y-1)+1 = -(x+y)-1 = \tilde{}(x+y)$.

D. `(ux-uy) == -(unsigned)(y-x)`. Yes. Due to the isomorphism between two's complement and unsigned arithmetic.

E. `((x >> 2) << 2) <= x`. Yes. Right shift rounds toward minus infinity.

Problem 2.82 Solution:

This problem helps students think about fractional binary representations.

A. Letting V denote the value of the string, we can see that shifting the binary point k positions to the right gives a string $y.yyyyyy\cdots$, which has numeric value $Y+V$, and also value $V\times 2^k$. Equating these gives $V = \frac{Y}{2^k-1}$.

B. (a) For $y = 101$, we have $Y = 5$, $k = 3$, $V = \frac{5}{7}$.
   (b) For $y = 0110$, we have $Y = 6$, $k = 4$, $V = \frac{6}{15} = \frac{2}{5}$.
   (c) For $y = 010011$, we have $Y = 19$, $k = 6$, $V = \frac{19}{63}$.

Problem 2.83 Solution:

This problem helps students appreciate the property of IEEE floating point that the relative magnitude of two numbers can be determined by viewing the combination of exponent and fraction as an unsigned integer. Only the signs and the handling of ± 0 requires special consideration.

code/data/floatcomp-ans.c

```
1 int float_le(float x, float y) {
2     unsigned ux = f2u(x);
3     unsigned uy = f2u(y);
4     unsigned sx = ux >> 31;
5     unsigned sy = uy >> 31;
6
7     return
8         (ux<<1 == 0 && uy<<1 == 0) ||  /* Both are zero */
9         (sx && !sy) ||                 /* x < 0, y >=  0 */
10        (!sx && !sy && ux <= uy) ||    /* x >= 0, y >= 0 */
11        (sx && sy && ux >= uy);        /* x <  0, y <  0 */
12 }
```

code/data/floatcomp-ans.c

Problem 2.84 Solution:

Exercises such as this help students understand floating point representations, their precision, and their ranges.

A. The number 7.0 will have $E = 2$, $M = 1.11_2 = \frac{7}{4}$, $f = 0.11_2 = \frac{3}{4}$, and $V = 7$. The exponent bits will be $100\cdots01$ and the fraction bits will be $1100\cdots0$.

B. The largest odd integer that can be represented exactly will have a binary representation consisting of $n + 1$ ones. It will have $E = n$, $M = 1.11\cdots1_2 = 2 - 2^{-n}$, $f = 0.11\cdots1_2 = 1 - 2^{-n}$, and a value $V = 2^{n+1} - 1$. The bit representation of the exponent will be the binary representation of $n + 2^{k-1} - 1$. The bit representation of the fraction will be $11\cdots11$.

C. The reciprocal of the smallest positive normalized value will have value $V = 2^{2^{k-1}-2}$. It will have $E = 2^{k-1} - 2$, $M = 1$, and $f = 0$. The bit representation of the exponent will be $11\cdots101$. The bit representation of the fraction will be $00\cdots00$.

Problem 2.85 Solution:

This exercise is of practical value, since Intel-compatible processors perform all of their arithmetic in extended precision. It is interesting to see how adding a few more bits to the exponent greatly increases the range of values that can be represented.

Description	Extended precision	
	Value	Decimal
Smallest pos. denorm.	$2^{-63} \times 2^{-16382}$	3.64×10^{-4951}
Smallest pos. norm.	2^{-16382}	3.36×10^{-4932}
Largest norm.	$(2 - \epsilon) \times 2^{16383}$	1.19×10^{4932}

Problem 2.86 Solution:

We have found that working through floating point representations for small word sizes is very instructive. Problems such as this one help make the description of IEEE floating point more concrete.

Description	Hex	M	E	V
-0	`8000`	0	-62	-0
Smallest value > 2	`4001`	$\frac{257}{256}$	1	$\frac{257}{128}$
512	`4800`	1	72	—
Largest denormalized	`00FF`	$\frac{255}{256}$	-62	255×2^{-70}
$-\infty$	`FF00`	—	—	—
Number with hex representation `3BB0`	—	$\frac{27}{16}$	-4	$\frac{27}{256}$

Problem 2.87 Solution:

This problem tests a lot of concepts about floating-point representations, including the encoding of normalized and denormalized values, as well as rounding.

Format A		Format B		Comments
Bits	Value	Bits	Value	
`1 01111 001`	$\frac{-9}{8}$	`1 0111 0010`	$\frac{-9}{8}$	
`0 10110 011`	176	`0 1110 0110`	176	
`1 00111 010`	$\frac{-5}{1024}$	`1 0000 0101`	$\frac{-5}{1024}$	Norm $\rightarrow$ denorm
`0 00000 111`	$\frac{7}{131072}$	`0 0000 0001`	$\frac{1}{1024}$	Smallest positive denorm
`1 11100 000`	-8192	`1 1110 1111`	-248	Smallest number $> -\infty$
`0 10111 100`	384	`0 1111 0000`	$+\infty$	Round to ∞.

Problem 2.88 Solution:

This problem requires students to think of the relationship between `int`, `float`, and `double`.

A. `(float) x == (float) dx`. Yes. Converting to `float` could cause rounding, but both `x` and `dx` will be rounded in the same way.

B. `dx - dy == (double) (x-y)`. No. Let `x` = 0 and `y` = $TMin_{32}$.

C. `(dx + dy) + dz == dx + (dy + dz)`. Yes. Since each value ranges between $TMin_{32}$ and $TMax_{32}$, their sum can be represented exactly.

D. `(dx * dy) * dz == dx * (dy * dz)`. No. Let `dx` = $TMax_{32}$, `dy` = $TMax_{32} - 1$, `dz` = $TMax_{32} - 2$. (Not detected with Linux/GCC)

E. `dx / dx == dz / dz`. No. Let `x = 0, z = 1`.

Problem 2.89 Solution:

This problem helps students understand the relation between the different categories of numbers. Getting all of the cutoff thresholds correct is fairly tricky. Our solution file contains testing code.

code/data/fpwr2-ans.c

```
1 /* Compute 2**x */
2 float fpwr2(int x) {
3
4     unsigned exp, frac;
5     unsigned u;
6
7     if (x < -149) {
8         /* Too small.  Return 0.0 */
```

```
 9          exp  = 0;
10          frac = 0;
11      } else if (x < -126) {
12          /* Denormalized result */
13          exp  = 0;
14          frac = 1 << (x + 149);
15      } else if (x < 128) {
16          /* Normalized result. */
17          exp  = x + 127;
18          frac = 0;
19      } else {
20          /* Too big.  Return +oo */
21          exp  = 255;
22          frac = 0;
23      }
24      u = exp << 23 | frac;
25      return u2f(u);
26  }
```

code/data/fpwr2-ans.c

Problem 2.90 Solution:

This problem requires students to work from a bit representation of a floating point number to its fractional binary representation.

A. $\pi \approx \quad 11.00100100001111111011011_2$.

B. $22/7 = 11.001001001001001001\cdots_2$.

C. They diverge in the ninth bit to the right of the binary point.

Problem 2.91 Solution:

This problem is relatively straightforward, and it provides a useful warmup for the more advanced problems.

code/data/float-functions.c

```
 1  /* Compute -f.  If f is NaN, then return f. */
 2  float_bits float_negate(float_bits f) {
 3      unsigned exp =  f>>23 & 0xFF;
 4      unsigned frac = f     & 0x7FFFFF;
 5      unsigned mask = 1 << 31;
 6      unsigned neg = f ^ mask;
 7      if (exp == 0xFF && frac != 0)
 8          /* NaN */
 9          return f;
10      return neg;
11  }
```

code/data/float-functions.c

Problem 2.92 Solution:

This problem is also relatively straightforward, and it provides a useful warmup for the more advanced problems.

code/data/float-functions.c

```
1 /* Compute |f|.  If f is NaN, then return f. */
2 float_bits float_absval(float_bits f) {
3     unsigned exp =  f>>23 & 0xFF;
4     unsigned frac = f     & 0x7FFFFF;
5     unsigned mask = 1 << 31;
6     unsigned absval = f & ~mask;
7     if (exp == 0xFF && frac != 0)
8         /* NaN */
9         return f;
10    return absval;
11 }
```

code/data/float-functions.c

Problem 2.93 Solution:

This problem is more difficult, since it requires dealing with the transition from denormalized to normalized numbers, and from normalized to infinity.

code/data/float-functions.c

```
1 /* Compute 2*f.  If f is NaN, then return f. */
2 float_bits float_twice(float_bits f) {
3     unsigned sign = f>>31;
4     unsigned exp =  f>>23 & 0xFF;
5     unsigned frac = f     & 0x7FFFFF;
6     if (exp == 0) {
7         /* Denormalized.  Must double fraction */
8         frac = 2*frac;
9         if (frac > 0x7FFFFF) {
10            /* Result normalized */
11            frac = frac & 0x7FFFFF; /* Chop off leading bit */
12            exp = 1;
13        }
14    } else if (exp < 0xFF) {
15        /* Normalized.  Increase exponent */
16        exp++;
17        if (exp == 0xFF) {
18            /* Infinity */
19            frac = 0;
20        }
21    } else if (frac != 0) {
```

```
22          /* NaN */
23          return f;
24      }
25      /* Infinity does not require any changes */
26      return (sign << 31) | (exp << 23) | frac;
27  }
```

code/data/float-functions.c

Problem 2.94 Solution:

This problem is still more difficult, since it requires dealing with the transition from normalized to denormalized numbers, and it also may require rounding.

code/data/float-functions.c

```
1  /* Compute 0.5*f.  If f is NaN, then return f. */
2  float_bits float_half(float_bits f) {
3      unsigned sign = f>>31;
4      unsigned exp =  f>>23 & 0xFF;
5      unsigned frac = f     & 0x7FFFFF;
6      /* Only roundup case will be when rounding to even */
7      unsigned roundup = (frac & 0x3) == 3;
8      if (exp == 0) {
9          /* Denormalized.  Must halve fraction */
10         frac = (frac >> 1) + roundup;
11     } else if (exp < 0xFF) {
12         /* Normalized.  Decrease exponent */
13         exp--;
14         if (exp == 0) {
15             /* Denormalize, add back leading one */
16             frac = (frac >> 1) + roundup + 0x400000;
17         }
18     } else if (frac != 0) {
19         /* NaN */
20         return f;
21     }
22     /* Infinity does not require any changes */
23     return (sign << 31) | (exp << 23) | frac;
24 }
```

code/data/float-functions.c

Problem 2.95 Solution:

This problem requires rounding and testing for out-of-range arguments.

code/data/float-functions.c

```
1  /*
2   * Compute (int) f.
```

```
 * If conversion causes overflow or f is NaN, return 0x80000000
 */
int float_f2i(float_bits f) {
    unsigned sign = f >> 31;
    unsigned exp = (f >> 23) & 0xFF;
    unsigned frac = f & 0x7FFFFF;
    /* Create normalized value with leading one inserted,
       and rest of significand in bits 8--30.
    */
    unsigned val = 0x80000000u + (frac << 8);
    if (exp < 127) {
        /* Absolute value is < 1 */
        return (int) 0;
    }
    if (exp > 158)
        /* Overflow */
        return (int) 0x80000000u;
    /* Shift val right */
    val = val >> (158 - exp);
    /* Check if out of range */
    if (sign) {
        /* Negative */
        return val > 0x80000000u ? (int) 0x80000000u : -(int) val;
    } else {
        /* Positive */
        return val > 0x7FFFFFFF ? (int) 0x80000000u : (int) val;
    }
}
```

code/data/float-functions.c

Problem 2.96 Solution:

This problem requires the most complex rounding computations of all the problems.

code/data/float-functions.c

```
/* Compute (float) i */
float_bits float_i2f(int i) {
    unsigned sign = (i < 0);
    unsigned ai = (i < 0) ? -i : i;
    unsigned exp = 127+31;
    unsigned residue;
    unsigned frac = 0;
    if (ai == 0) {
        exp = 0;
        frac = 0;
    } else {
        /* Normalize so that msb = 1 */
        while ((ai & (1<<31)) == 0) {
            ai = ai << 1;
```

```
15              exp--;
16          }
17          /* Now have Bit 31 = MSB (becomes implied leading one)
18             Bits 8-30 are tentative fraction,
19             Bits 0-7 require rounding.
20          */
21          residue = ai & 0xFF;
22          frac = (ai >> 8) & 0x7FFFFF; /* 23 bits */
23          if (residue > 0x80 || (residue == 0x80 && (frac & 0x1))) {
24              /* Round up */
25              frac ++;
26              /* Might need to renormalize */
27              if (frac > 0x7FFFFF) {
28                  frac = (frac & 0x7FFFFF) >> 1;
29                  exp++;
30              }
31          }
32      }
33      return (sign << 31) | (exp << 23) | frac;
34  }
```

code/data/float-functions.c

1.3 Chapter 3: Machine Level Representation of C Programs

Problem 3.54 Solution:

This is an example of a problem that requires students to reverse engineer the actions of GCC. We have found that reverse engineering is a good way to learn about both compilers and machine-level programs.

code/asm/decode2-ans.c

```
1 int decode2(int x, int y, int z)
2 {
3     int t1 = y - z;
4     int t2 = x * t1;
5     int t3 = (t1 << 31) >> 31;
6     int t4 = t3 ^ t2;
7
8     return t4;
9 }
```

code/asm/decode2-ans.c

Problem 3.55 Solution:

A. This problem is similar to Problem 3.12, except that it uses signed values rather than unsigned. As a consequence, it must sign extend argument `x` and then treat both arguments to the multiplcation as 64-bit values.

Let x and y denote the numbers being multiplied. We can view the 64 bits of argument `y` as being split into two 32-bit fields, and let y_h be the number having two's complement representation given by the high-order field and y_l be the number having unsigned representation given by the low-order field. We can then write $y = 2^{32} \cdot y_h + y_l$ (this can be derived from Equation 2.3.) We can do a similar partitioning of the value obtained by sign extending `x` to 64 bits, giving $x = 2^{32} \cdot x_h + x_l$.

We can therefore compute $x \cdot y = x_h \cdot y_h \cdot 2^{64} + (x_h \cdot y_l + x_l \cdot y_h) \cdot 2^{32} + x_l \cdot y_l$. Since we are only interested in the lower 64 bits, we can ignore the term involving $x_h \cdot y_h$, and we can let s be the low-order 32 bits of $x_l \cdot y_h$, r be the low-order 32 bits of $x_h \cdot y_l$, and t be the full 64-bit product $x_l \cdot y_l$, which we can split into high and low-order parts t_h and t_l. The final result has t_l as the low-order part, and $r + s + t_h$ as the high-order part.

Here is the annotated assembly code:

code/asm/hmulti.32sa

```
   dest at %ebp+8, x at %ebp+12, y at %ebp+16
 1   movl    16(%ebp), %esi              Get y_l
 2   movl    12(%ebp), %eax              Get x (= x_l)
 3   movl    %eax, %edx                  Copy x
 4   sarl    $31, %edx                   Sign extend to get x_h
 5   movl    20(%ebp), %ecx              Get y_h
 6   imull   %eax, %ecx                  Compute s = x_l*y_h
 7   movl    %edx, %ebx                  Copy x_h
 8   imull   %esi, %ebx                  Compute r = x_h*y_l
 9   addl    %ebx, %ecx                  Compute r+s
10   mull    %esi                        Compute t = x_l*y_l
11   leal    (%ecx,%edx), %edx           Add r+s to t_h
12   movl    8(%ebp), %ecx               Get dest
13   movl    %eax, (%ecx)                Store t_l
14   movl    %edx, 4(%ecx)               Store r+s+t_h
```

code/asm/hmulti.32sa

Problem 3.56 Solution:

One way to analyze assembly code is to try to reverse the compilation process and produce C code that would look "natural" to a C programmer. For example, we wouldn't want any `goto` statements, since these are seldom used in C. Most likely, we wouldn't use a `do-while` statement either. This exercise forces students to reverse the compilation into a particular framework. It requires thinking about the translation of `for` loops.

A. We can see that `result` must be in register `%edi`, since this value gets copied to `%eax` at the end of the function as the return value (line 13). We can see that `%esi` and `%ebx` get loaded with the values of `x` and `n` (lines 1 and 2), leaving `%edx` as the one holding variable `mask` (line 4.)

B. Register `%edi` (`result`) is initialized to -1 and `%edx` (`mask`) to 1.

C. The condition for continuing the loop (line 12) is that `mask` is nonzero.

D. The shift instruction on line 10 updates `mask` to be `mask << n`.

E. Lines 6–8 update `result` to be `result ^ (x&mask)`.

F. Here is the original code:

code/asm/for.c

```
1 int loop(int x, int n)
2 {
3     int result = -1;
4     int mask;
5     for (mask = 0x1; mask != 0; mask = mask << n) {
6         result ^= (x & mask);
7     }
8     return result;
9 }
```

code/asm/for.c

Problem 3.57 Solution:

This problem has a simple solution, but it took us a while to understand how simple it could be. It will require students to experiment with running GCC on different versions of their code.

The idea of our solution is to set up a local variable having value 0, and then using a conditional move to overwrite `xp` with the address of this variable when `xp` is null.

code/asm/cread.c

```
1 int cread_alt(int *xp) {
2     int zero = 0;
3     /* Replace null pointer */
4     if (!xp) xp = &zero;
5     return *xp;
6 }
```

code/asm/cread.c

Problem 3.58 Solution:

This problem requires students to reason about the code fragments that implement the different branches of a `switch` statement. For this code, it also requires understanding different forms of pointer dereferencing.

A. In line 34, register `%edx` is copied to register `%eax` as the return value. From this, we can infer that `%edx` holds `result`.

B. The original C code for the function is as follows:

code/asm/switchprob2.c

```
1 /* Enumerated type creates set of constants numbered 0 and upward */
2 typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;
3
4 int switch3(int *p1, int *p2, mode_t action)
5 {
6     int result = 0;
7     switch(action) {
8     case MODE_A:
9         result = *p1;
10        *p1 = *p2;
11        break;
12    case MODE_B:
13        *p2 += *p1;
14        result = *p2;
15        break;
16    case MODE_C:
17        *p2 = 15;
18        result = *p1;
19        break;
20    case MODE_D:
21        *p2 = *p1;
22        /* Fall Through */
23    case MODE_E:
24        result = 17;
25        break;
26    default:
27        result = -1;
28    }
29    return result;
30 }
```

code/asm/switchprob2.c

Problem 3.59 Solution:

This problem gives students practice analyzing disassembled code. The `switch` statement contains all the features one can imagine—cases with multiple labels, holes in the range of possible case values, and cases that fall through. The main trick is to use the jump table to identify the different entry points, and then analyze each block of code separately.

code/asm/switchbody-ans.c

```
1 int switch_prob(int x, int n)
2 {
3     int result = x;
4
5     switch(n) {
6     case 50:
7     case 52:
```

```
8         result <<= 2;
9         break;
10    case 53:
11        result >>= 2;
12        break;
13    case 54:
14        result *= 3;
15        /* Fall through */
16    case 55:
17        result *= result;
18        /* Fall through */
19    default:
20        result += 10;
21    }
22
23    return result;
24 }
```

code/asm/switchbody-ans.c

Problem 3.60 Solution:

This problem demonstrates that the same principles of nested array access extend beyond two levels.

A. Array element `A[i][j][k]` is located at address $x_A + 4(T(S \cdot i + j) + k)$.

B. We can see that the computations on lines 2–3 compute $11 \cdot j$, that line 4 computes $99 \cdot i$, and so line 7 reads from memory address $x_A + 4(11(9i + j) + k)$. From this we can conclude that $T = 11$ and $S = 9$. The function returns 1980, the number of bytes required to store array A, and so $4 \cdot R \cdot S \cdot T = 396 \cdot R = 1980$, giving $R = 5$.

Problem 3.61 Solution:

The code generated by GCC holds a value equal to $4n$ in register `%edi` that it uses to increment `Bptr`. Our strategy is to change the loop index `j` to increment by 4, so that we can use $4n$ as our stopping condition, eliminating the need to retrieve n from memory.

code/asm/varprod-ans.c

```
1 int var_prod_ele_opt (int n, int A[n][n], int B[n][n], int i, int k)
2 {
3     char *Arow = (char *) (A[i]);
4     char *Bptr = (char *) (&B[0][k]);
5     int result = 0;
6     int j4;
7     int n4 = 4*n;
8     for (j4 = 0; j4 < n4; j4+=4) {
9         int a = *(int *) (Arow + j4);
10        int b = *(int *) Bptr;
```

```
11          result += a * b;
12          Bptr += n4;
13      }
14      return result;
15  }
```

code/asm/varprod-ans.c

As can be seen, convincing GCC to follow this strategy requires some creative coding. We introduce a local variable `n4` to hold the scaled value of n, and use this as both the stopping value and the increment. This declaring both `Arow` and `Bptr` to be of type `char *`, so that the compiler will not scale the pointer arithmetic.

Another way to reduce the number of variables in the loop is to get rid of the parameter `j` altogether, using a pointer `Aptr` that steps through the successive elements of row i of `A`. We can use a pointer to element `A[i][n]` to determine when we have completed the iteration.

code/asm/varprod-ans.c

```
1  int var_prod_ele_opt_alt(int n, int A[n][n], int B[n][n], int i, int k)
2  {
3      int *Aptr = &A[i][0];
4      int *Bptr = &B[0][k];
5      int *Aend = &A[i][n];
6      int result = 0;
7      while (Aptr < Aend) {
8         result += *Aptr * *Bptr;
9         Aptr += 1;
10        Bptr += n;
11     }
12     return result;
13 }
```

code/asm/varprod-ans.c

Problem 3.62 Solution:

It might surprise students that they can understand the machine code implementation of a function based only on its inner loop. It's a useful strategy, since it avoids wading through lots of uninteresting code.

A. Line 7 increments register `%ebx` by 52, and so we can guess that $M = 52/4 = 13$. This is confirmed when we see that `%ebx` is used as a pointer to the successive elements in column j.

B. Line 8 checks the loop condition, and so we can surmise that `%edi` holds i and `%ecx` holds j.

C. Here is an optimized version of the C code:

code/asm/transpose.c

```
1  void transpose_opt(Marray_t A) {
2      int i, j;
```

```
3     for (i = 0; i < M; i++) {
4         int *Arow = &A[i][0];
5         int *Acol = &A[0][i];
6         for (j = 0; j < i; j++) {
7             int t1 = *Acol;
8             int t2 = Arow[j];
9             *Acol = t2;
10            Arow[j] = t1;
11            Acol += M;
12        }
13    }
14 }
```

code/asm/transpose.c

Problem 3.63 Solution:

The assembly code generated by version GCC used for the book is not very efficient. You might want to try compiling it with a different version.

This problem requires a fair amount of reverse engineering. Working backwards, you can see that the value of `E1(n)` is computed by the instructions leading up to line 5, while the value of `E2(n)` is computed by the instructions leading up to line 3.

code/asm/vararray.c

```
1 #define E1(n) (2*(n)+1)
2 #define E2(n) (3*(n))
```

code/asm/vararray.c

Problem 3.64 Solution:

This exercise both introduces new material and makes students spend time examining the IA32 stack structure and how it is accessed by machine code.

A. We can see that the value at offset 8 from `%ebp` is a pointer to the structure where the function is to fill in its results. The values at offsets 12 and 16 correspond to the fields `s1.a` and `s1.p`, respectively.

B. Starting from the top of stack, we can see that the first field points to the location allocated for the returned structure. The next two fields correspond to argument values `s1.a` and `s1.p`. The final two fields are where the result of `word_sum` is stored, and so are the values of `s2.sum` and `s2.diff`.

C. The general strategy is to pass the argument structure on the stack, just as any argument is passed. The callee then accesses the fields of its argument by offsets relative to `%ebp`.

D. The general strategy is for the caller to allocate space in its own stack from for the result structure, and then it passes a pointer to this structure as a hidden first argument to the function.

Problem 3.65 Solution:

This problem is like a puzzle, where a number of clues must be assembled to get the answer. It tests their understanding of structure layout, including the need to insert padding to satisfy alignment. The right way to solve the problem is to write out formulas for the offsets of the different fields in terms of A and B and then determine the possible solutions.

We can see from the assembly code that fields `t` and `u` of structure `str2` are at offsets 12 and 36, respectively. We can see that field `y` of structure `str1` is at offset 92. We can write the following equations for these offsets:

$$\begin{aligned} B + e_1 &= 12 \\ B + e_1 + 4 + 2B + e_2 &= 36 \\ 2AB + e_3 &= 92 \end{aligned}$$

where e_1, e_2, and e_3 represent amounts of padding we need to insert in the structures to satisfy alignment. We can also see that $e_1 \in \{0, 1, 2, 3\}$ and that $e_2, e_3 \in \{0, 2\}$.

From the first equation, we can see that $B \in \{9, 10, 11, 12\}$. We can rewrite the second equation to be

$$3B + e_1 + e_2 = 32$$

and we can see that this narrows the choices for B to either 9 or 10. Of these choices, only the combination $B = 9$, $A = 5$ can satisfy the third equation.

Problem 3.66 Solution:

This problem requires using a variety of skills to determine parameters of the structure. The code uses tricky memory address computations.

The analysis requires understanding data structure layouts, pointers, and address computations. Problems such as this one make good exercises for in-class discussion, such as during a recitation period. Try to convince students that these are "brain teasers." The answer can only be determined by assembling a number of different clues.

Here is a sequence of steps that leads to the answer:

1. Let us say that pointer `bp` has value p. Line 6 scales i by 5. Line 7 reads a value from address $p + 20i + 4$ and adds it to $5i$. We can surmise that this read corresponds to the expression `ap->idx`, from which we can deduce:

   - Structure `a_struct` is 20 bytes.
   - Pointer `ap` has value $p + 20i + 4$. Let us call this value q.
   - Field `idx` is at offset 0 within structure `a_struct`.

   Let j be the value of `ap->idx`.

2. Line 8 computes `bp->right`, and this field is at offset `0xb8` = 184. This implies that `CNT` has value 9.

3. Line 10 stores `n` at address $p + 20i + 4j + 8 = q + 4j + 4$. We can infer that field `x` is at offset 4 within `a_struct`.

This analysis leads us to the following answers:

A. `CNT` is 9.

B. The following is the declaration for `a_struct`.

code/asm/structprob-ans.c

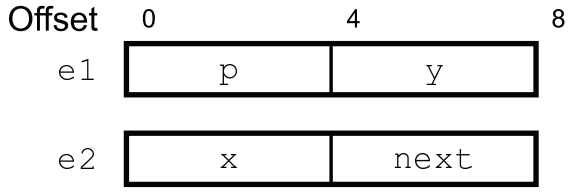
```
1 typedef struct {
2     int idx;
3     int x[4];
4 } a_struct;
```

code/asm/structprob-ans.c

Problem 3.67 Solution:

This is a very tricky problem. It raises the need for puzzle-solving skills as part of reverse engineering to new heights. It shows very clearly that unions are simply a way to associate multiple names (and types) with a single storage location.

A. The layout of the union is shown in the table that follows. As the table illustrates, the union can have either its "`e1`" interpretation (having fields `e1.p` and `e1.y`), or it can have its "`e2`" interpretation (having fields `e2.x` and `e2.next`).



B. It uses 8 bytes.

C. As always, we start by annotating the assembly code. In our annotations, we show multiple possible interpretations for some of the instructions, and then indicate which interpretation later gets discarded. For example, line 2 could be interpreted as either getting element `e1.y` or `e2.next`. In line 3, we see that the value gets used in an indirect memory reference, for which only the second interpretation of line 2 is possible.

code/asm/union2.32s

```
1   movl    8(%ebp), %edx     Get up
2   movl    4(%edx), %ecx     up->e1.y (no) or up->e2.next
3   movl    (%ecx), %eax      up->e2.next->e1.p or up->e2.next->e2.x (no)
4   movl    (%eax), %eax      *(up->e2.next->e1.p)
5   subl    (%edx), %eax      *(up->e2.next->e1.p) - up->e2.x
6   movl    %eax, 4(%ecx)     Store in up->e2.next->e1.y
```

code/asm/union2.32s

From this, we can generate C code as follows:

code/asm/union2.c

```
1 void proc (union ele *up)
2 {
3     up->e2.next->e1.y = *(up->e2.next->e1.p) - up->e2.x;
4 }
```

code/asm/union2.c

Problem 3.68 Solution:

This problem gets students in the habit of writing reliable code. As a general principle, code should not be vulnerable to conditions over which it has no control, such as the length of an input line. The following implementation uses the library function `fgets` to read up to `BUFSIZE` characters at a time.

code/asm/bufdemo.c

```
1 /* Read input line and write it back */
2 /* Code will work for any buffer size.  Bigger is more time-efficient */
3 #define BUFSIZE 64
4 void good_echo()
5 {
6     char buf[BUFSIZE];
7     int i;
8     while (1) {
9         if (!fgets(buf, BUFSIZE, stdin))
10            return;  /* End of file or error */
11        /* Print characters in buffer */
12        for (i = 0; buf[i] && buf[i] != '\n'; i++)
13            if (putchar(buf[i]) == EOF)
14                return; /* Error */
15        if (buf[i] == '\n') {
16            /* Reached terminating newline */
17            putchar('\n');
18            return;
19        }
20    }
21 }
```

code/asm/bufdemo.c

An alternative implementation is to use `getchar` to read the characters one at a time.

Problem 3.69 Solution:

This problem combines the topics of loops, data structures, and x86-64 code.

A. Here is the code

code/asm/tree64.c

```
1 /* Find the value in the leaf reached by following rightmost path */
2 long trace(tree_ptr tp) {
3     long val = 0;
4     while (tp) {
5         val = tp->val;
6         tp = tp->right;
7     }
8     return val;
9 }
```

code/asm/tree64.c

B. This code follows the rightmost branches in the tree and returns the value of the last node encountered. It returns 0 for an empty tree.

Problem 3.70 Solution:

This problem combines the topics of recursive functions, conditional moves, data structures, and x86-64 code.

A. Here is the code

code/asm/tree64.c

```
1 /* This looks for the max value in a tree */
2 long traverse(tree_ptr tp) {
3     if (!tp)
4         return LONG_MIN;
5     else {
6         long val = tp->val;
7         long lval, rval;
8         lval = traverse(tp->left);
9         if (lval > val)
10            val = lval;
11        rval = traverse(tp->right);
```

```
12          if (rval > val)
13              val = rval;
14          return val;
15      }
16  }
```

code/asm/tree64.c

B. This code finds the maximum value in a tree. It returns $TMin_{64}$ for an empty tree.

1.4 Chapter 4: Processor Architecture

Problem 4.43 Solution:

This problem further explores the semantics of this unusual instruction, which will become important when we implement the `pushl` instruction.

A. If we substitute `%esp` for *REG* in the code sequence we get

```
subl $4,%esp        Decrement stack pointer
movl %esp,(%esp)    Store REG on stack
```

which would imply that the decremented version of the stack pointer would be stored on the stack, which we know is not the case.

B. The following code sequence is correct for all registers, although harder to understand:

```
movl REG,-4(%esp)    Store REG at new top of stack
subl $4,%esp         Decrement stack pointer
```

Problem 4.44 Solution:

Implementing `popl` instruction will require great care, since it modifies two registers.

A. Substituting `%esp` for *REG* in the code sequence gives:

```
movl (%esp),%esp    Read %esp from stack
addl $4,%esp        Increment stack pointer
```

This code sequence would first read a new value of the stack pointer from memory and then increment it, yielding neither the value from memory nor anything related to the previous stack pointer.

B. As with Problem 4.43, we should reorder the two instructions:

```
addl $4,%esp          Increment stack pointer
movl -4(%esp), REG    Read REG from previous stack top
```

Problem 4.45 Solution:

This is a challenging exercise for those without much experience in writing assembly code. It's very important to provide students the instruction set simulator YIS to try out their code.

It helps a lot to first express the function using pointer code:

```
/* Bubble sort: Pointer version */
void bubble(int *data, int count) {
    /* Pointer to last element to check */
    int *p_end = data+count-1;
    while (data < p_end) {
        int *p = data;
        while (p < p_end) {
            int r = *p;
            int s = *(p+1);
            if (s < r) {
                /* Swap adjacent elements */
                *p = s;
                *(p+1) = r;
            }
            p++;
        }
        p_end--;
    }
}
```

Here is a complete program including the sort function and the testing code:

```
 1 # Execution begins at address 0
 2         .pos 0                  #
 3 init:   irmovl Stack, %esp      # Set up Stack pointer
 4         irmovl Stack, %ebp      # Set up base pointer
 5         jmp Main                # Execute main program
 6
 7 # Array of 6 elements
 8         .align 4
 9 array:  .long 0xdddd            # These values should get sorted
10         .long 0xeeee
11         .long 0xbbbb
12         .long 0xaaaa
13         .long 0xffff
14         .long 0xcccc
15         .long 0x0101            # This value should not change
16
17 Main:   irmovl $6,%eax
```

```
18          pushl %eax              # Push 6
19          irmovl array,%edx
20          pushl %edx              # Push array
21          call Bubble             # Bubble(array, 6)
22          halt
23
24  # void Bubble(int *data, int count)
25  Bubble: pushl %ebp
26          rrmovl %esp,%ebp
27          pushl %ebx              # Save registers
28          pushl %esi
29          pushl %edi
30          mrmovl 8(%ebp),%ecx     # ecx = data
31          mrmovl 12(%ebp),%edx    # edx = count
32          addl %edx,%edx
33          addl %edx,%edx          # count *= 4;
34          irmovl $4,%eax
35          subl %eax,%edx          # count -= 4;
36          addl %ecx,%edx          # edx = p_end
37          rrmovl %ecx,%eax
38          subl %edx,%eax          # data - p_end
39          jge Done
40  Loop1:  rrmovl %ecx,%ebx        # p = data
41  Loop2:  mrmovl (%ebx),%edi      # r = *p
42          mrmovl 4(%ebx),%esi     # s = *(p+1)
43          rrmovl %esi,%eax
44          subl %edi,%eax          # s-r
45          jge Skip                # Skip if s >= r
46          rmmovl %esi,(%ebx)      # *p = s
47          rmmovl %edi,4(%ebx)     # *(p+1) = r
48  Skip:
49          irmovl $4,%eax
50          addl %eax,%ebx          # p++
51          rrmovl %ebx,%eax
52          subl %edx,%eax          # p - p_end
53          jl Loop2
54          irmovl $4,%eax
55          subl %eax,%edx          # p_end--
56          rrmovl %ecx,%eax
57          subl %edx,%eax          # data - p_end
58          jl Loop1
59  Done:
60          popl %edi               # Restore registers
61          popl %esi
62          popl %ebx
63          rrmovl %ebp,%esp
64          popl %ebp
65          ret
66
67          .pos 0x100
```

```
68 Stack:  # The stack goes here and grows to lower addresses
```

Problem 4.46 Solution:

Our original code in the inner loop either updates the values in the array or keeps them the same:

```
        subl %edi,%eax          # s-r
        jge Skip                # Skip if s >= r
        rmmovl %esi,(%ebx)      # *p = s
        rmmovl %edi,4(%ebx)     # *(p+1) = r
Skip:
```

The modified version uses multiple conditional moves to either enable or disable a swap of variables `r` and `s`, and then it updates the array values unconditionally:

```
        subl %edi,%eax          # s-r
        cmovl %esi,%eax         # if old s < old r, t = s
        cmovl %edi,%esi         # if old s < old r, s = r
        cmovl %eax,%edi         # if old s < old r, r = t
        rmmovl %edi,(%ebx)      # *p = r
        rmmovl %esi,4(%ebx)     # *(p+1) = s
```

Problem 4.47 Solution:

This problem makes students carefully examine the tables showing the computation stages for the different instructions. The steps for `iaddl` are a hybrid of those for `irmovl` and `OPl`.

Stage	`iaddl V, rB`
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC + 1] valC $\leftarrow$ M_4[PC + 2] valP $\leftarrow$ PC + 6
Decode	valB $\leftarrow$ R[rB]
Execute	valE $\leftarrow$ valB + valC
Memory	
Write back	R[rB] $\leftarrow$ valE
PC update	PC $\leftarrow$ valP

Problem 4.48 Solution:

The `leave` instruction is fairly obscure, but working through its implementation makes it easier to understand the implementation of the `popl` instruction, one of the trickiest of the Y86 instructions.

Stage	`leave`
Fetch	icode : ifun $\leftarrow$ M_1[PC] valP $\leftarrow$ PC + 1
Decode	valA $\leftarrow$ R[%ebp] valB $\leftarrow$ R[%ebp]
Execute	valE $\leftarrow$ valB + 4
Memory	valM $\leftarrow$ M_4[valA]
Write back	R[%esp] $\leftarrow$ valE R[%ebp] $\leftarrow$ valM
PC update	PC $\leftarrow$ valP

Problem 4.49 Solution:

The following HCL code includes implementations of both the `iaddl` instruction and the `leave` instructions. The implementations are fairly straightforward given the computation steps listed in the solutions to problems 4.47 and 4.48. You can test the solutions using the test code in the `ptest` subdirectory. Make sure you use command line argument '`-i`.'

code/arch/seq-full-ans.hcl

```
1 ################################################################
2 #  HCL Description of Control for Single Cycle Y86 Processor SEQ   #
3 #  Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010       #
4 ################################################################
5
6 ## This is the solution for the iaddl and leave problems
7
8 ################################################################
9 #    C Include's.  Don't alter these                           #
10 ################################################################
11
12 quote '#include <stdio.h>'
13 quote '#include "isa.h"'
14 quote '#include "sim.h"'
15 quote 'int sim_main(int argc, char *argv[]);'
16 quote 'int gen_pc(){return 0;}'
17 quote 'int main(int argc, char *argv[])'
18 quote '  {plusmode=0;return sim_main(argc,argv);}'
19
20 ################################################################
21 #    Declarations.  Do not change/remove/delete any of these   #
22 ################################################################
23
24 ##### Symbolic representation of Y86 Instruction Codes #############
25 intsig INOP     'I_NOP'
26 intsig IHALT    'I_HALT'
27 intsig IRRMOVL  'I_RRMOVL'
```

```
28 intsig IIRMOVL  'I_IRMOVL'
29 intsig IRMMOVL  'I_RMMOVL'
30 intsig IMRMOVL  'I_MRMOVL'
31 intsig IOPL     'I_ALU'
32 intsig IJXX     'I_JMP'
33 intsig ICALL    'I_CALL'
34 intsig IRET     'I_RET'
35 intsig IPUSHL   'I_PUSHL'
36 intsig IPOPL    'I_POPL'
37 # Instruction code for iaddl instruction
38 intsig IIADDL   'I_IADDL'
39 # Instruction code for leave instruction
40 intsig ILEAVE   'I_LEAVE'
41
42 ##### Symbolic represenations of Y86 function codes                  #####
43 intsig FNONE    'F_NONE'        # Default function code
44
45 ##### Symbolic representation of Y86 Registers referenced explicitly #####
46 intsig RESP     'REG_ESP'       # Stack Pointer
47 intsig REBP     'REG_EBP'       # Frame Pointer
48 intsig RNONE    'REG_NONE'      # Special value indicating "no register"
49
50 ##### ALU Functions referenced explicitly                            #####
51 intsig ALUADD   'A_ADD'         # ALU should add its arguments
52
53 ##### Possible instruction status values                             #####
54 intsig SAOK     'STAT_AOK'              # Normal execution
55 intsig SADR     'STAT_ADR'      # Invalid memory address
56 intsig SINS     'STAT_INS'      # Invalid instruction
57 intsig SHLT     'STAT_HLT'      # Halt instruction encountered
58
59 ##### Signals that can be referenced by control logic ####################
60
61 ##### Fetch stage inputs               #####
62 intsig pc 'pc'                         # Program counter
63 ##### Fetch stage computations         #####
64 intsig imem_icode 'imem_icode'         # icode field from instruction memory
65 intsig imem_ifun  'imem_ifun'          # ifun field from instruction memory
66 intsig icode      'icode'             # Instruction control code
67 intsig ifun       'ifun'              # Instruction function
68 intsig rA         'ra'                # rA field from instruction
69 intsig rB         'rb'                # rB field from instruction
70 intsig valC       'valc'              # Constant from instruction
71 intsig valP       'valp'              # Address of following instruction
72 boolsig imem_error 'imem_error'       # Error signal from instruction memory
73 boolsig instr_valid 'instr_valid'      # Is fetched instruction valid?
74
75 ##### Decode stage computations        #####
76 intsig valA     'vala'                # Value from register A port
77 intsig valB     'valb'                # Value from register B port
```

```
78
79 ##### Execute stage computations        #####
80 intsig valE     'vale'                  # Value computed by ALU
81 boolsig Cnd     'cond'                  # Branch test
82
83 ##### Memory stage computations         #####
84 intsig valM     'valm'                  # Value read from memory
85 boolsig dmem_error 'dmem_error'         # Error signal from data memory
86
87
88 ################################################################
89 #    Control Signal Definitions.                               #
90 ################################################################
91
92 ################ Fetch Stage     ###################################
93
94 # Determine instruction code
95 int icode = [
96         imem_error: INOP;
97         1: imem_icode;          # Default: get from instruction memory
98 ];
99
100 # Determine instruction function
101 int ifun = [
102         imem_error: FNONE;
103         1: imem_ifun;           # Default: get from instruction memory
104 ];
105
106 bool instr_valid = icode in
107         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
108                IIADDL, ILEAVE,
109                IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
110
111 # Does fetched instruction require a regid byte?
112 bool need_regids =
113         icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
114                       IIADDL,
115                       IIRMOVL, IRMMOVL, IMRMOVL };
116
117 # Does fetched instruction require a constant word?
118 bool need_valC =
119         icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };
120
121 ################ Decode Stage    ###################################
122
123 ## What register should be used as the A source?
124 int srcA = [
125         icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
126         icode in { ILEAVE } : REBP;
127         icode in { IPOPL, IRET } : RESP;
```

```
128         1 : RNONE; # Don't need register
129 ];
130
131 ## What register should be used as the B source?
132 int srcB = [
133         icode in { IOPL, IRMMOVL, IMRMOVL  } : rB;
134         icode in { IIADDL  } : rB;
135         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
136         icode in { ILEAVE  } : REBP;
137         1 : RNONE;  # Don't need register
138 ];
139
140 ## What register should be used as the E destination?
141 int dstE = [
142         icode in { IRRMOVL } && Cnd : rB;
143         icode in { IIRMOVL, IOPL} : rB;
144         icode in { IIADDL } : rB;
145         icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
146         icode in { ILEAVE } : RESP;
147         1 : RNONE;  # Don't write any register
148 ];
149
150 ## What register should be used as the M destination?
151 int dstM = [
152         icode in { IMRMOVL, IPOPL } : rA;
153         icode in { ILEAVE } : REBP;
154         1 : RNONE;  # Don't write any register
155 ];
156
157 ################ Execute Stage   ###################################
158
159 ## Select input A to ALU
160 int aluA = [
161         icode in { IRRMOVL, IOPL } : valA;
162         icode in { IIRMOVL, IRMMOVL, IMRMOVL } : valC;
163         icode in { IIADDL } : valC;
164         icode in { ICALL, IPUSHL } : -4;
165         icode in { IRET, IPOPL } : 4;
166         icode in { ILEAVE } : 4;
167         # Other instructions don't need ALU
168 ];
169
170 ## Select input B to ALU
171 int aluB = [
172         icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
173                        IPUSHL, IRET, IPOPL } : valB;
174         icode in { IIADDL, ILEAVE } : valB;
175         icode in { IRRMOVL, IIRMOVL } : 0;
176         # Other instructions don't need ALU
177 ];
```

```
178
179 ## Set the ALU function
180 int alufun = [
181         icode == IOPL : ifun;
182         1 : ALUADD;
183 ];
184
185 ## Should the condition codes be updated?
186 bool set_cc = icode in { IOPL, IIADDL };
187
188 ################ Memory Stage    ###################################
189
190 ## Set read control signal
191 bool mem_read = icode in { IMRMOVL, IPOPL, IRET, ILEAVE };
192
193 ## Set write control signal
194 bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };
195
196 ## Select memory address
197 int mem_addr = [
198         icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
199         icode in { IPOPL, IRET } : valA;
200         icode in { ILEAVE } : valA;
201         # Other instructions don't need address
202 ];
203
204 ## Select memory input data
205 int mem_data = [
206         # Value from register
207         icode in { IRMMOVL, IPUSHL } : valA;
208         # Return PC
209         icode == ICALL : valP;
210         # Default: Don't write anything
211 ];
212
213 ## Determine instruction status
214 int Stat = [
215         imem_error || dmem_error : SADR;
216         !instr_valid: SINS;
217         icode == IHALT : SHLT;
218         1 : SAOK;
219 ];
220
221 ################ Program Counter Update ############################
222
223 ## What address should instruction be fetched at
224
225 int new_pc = [
226         # Call.  Use instruction constant
227         icode == ICALL : valC;
```

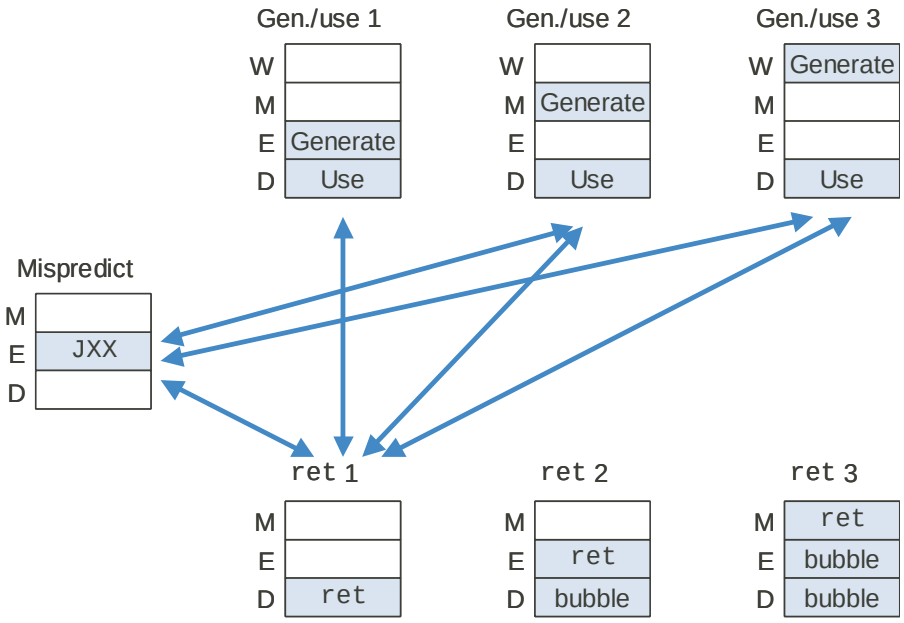


Figure 1.1: **Pipeline states for special control conditions.** The pairs connected by arrows can arise simultaneously.

```
228         # Taken branch.  Use instruction constant
229         icode == IJXX && Cnd : valC;
230         # Completion of RET instruction.  Use value from stack
231         icode == IRET : valM;
232         # Default: Use incremented PC
233         1 : valP;
234 ];
```

code/arch/seq-full-ans.hcl

Problem 4.50 Solution:

See the solution to Problem 4.49. When you test this code with the scripts in `ptest`, be sure to use the command line argument '`-l`.'

Problem 4.51 Solution:

This is a hard problem, because there are many possible combinations of special cases that can occur simultaneously. Figure 1.1 illustrates this problem. We can see that there are now three variants of generate/use cases, where the instruction in the execute, memory, or write-back stage is generating a value to be used by the instruction in the decode stage. The second and third generate/use cases can occur in combination with a mispredicted branch. In this case, we want to handle the misprediction, injecting bubbles into the decode and execute stages.

For cases where a misprediction does not occur, each of the generate/use conditions can occur in combination with the first `ret` pattern (where `ret` uses the value of `%esp`). In this case, we want to handle the data

hazard by stalling the fetch and and decode stages and injecting a bubble into the execute stage.

The test script `ctest.pl` in the `ptest` subdirectory generates tests that thoroughly test these possible control combinations.

The following shows the HCL code for the pipeline control logic.

code/arch/pipe-nobypass-ans.hcl

```
# Should I stall or inject a bubble into Pipeline Register F?
# At most one of these can be true.
bool F_bubble = 0;
bool F_stall =
        # Stall if either operand source is destination of
        # instruction in execute, memory, or write-back stages
        d_srcA != RNONE && d_srcA in
          { E_dstM, e_dstE, M_dstM, M_dstE, W_dstM, W_dstE } ||
        d_srcB != RNONE && d_srcB in
          { E_dstM, e_dstE, M_dstM, M_dstE, W_dstM, W_dstE } ||
        # Stalling at fetch while ret passes through pipeline
        IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register D?
# At most one of these can be true.
bool D_stall =
        # Stall if either operand source is destination of
        # instruction in execute, memory, or write-back stages
        # but not part of mispredicted branch
        !(E_icode == IJXX && !e_Cnd) &&
         (d_srcA != RNONE && d_srcA in
            { E_dstM, e_dstE, M_dstM, M_dstE, W_dstM, W_dstE } ||
          d_srcB != RNONE && d_srcB in
            { E_dstM, e_dstE, M_dstM, M_dstE, W_dstM, W_dstE });

bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
        # but not condition for a generate/use hazard
        !(d_srcA != RNONE && d_srcA in
           { E_dstM, e_dstE, M_dstM, M_dstE, W_dstM, W_dstE } ||
          d_srcB != RNONE && d_srcB in
           { E_dstM, e_dstE, M_dstM, M_dstE, W_dstM, W_dstE }) &&
          IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register E?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
        # Mispredicted branch
```

```
        (E_icode == IJXX && !e_Cnd) ||
          # Inject bubble if either operand source is destination of
          # instruction in execute, memory, or write back stages
          d_srcA != RNONE &&
            d_srcA in { E_dstM, e_dstE, M_dstM, M_dstE, W_dstM, W_dstE } ||
          d_srcB != RNONE &&
            d_srcB in { E_dstM, e_dstE, M_dstM, M_dstE, W_dstM, W_dstE };

# Should I stall or inject a bubble into Pipeline Register M?
# At most one of these can be true.
bool M_stall = 0;
# Start injecting bubbles as soon as exception passes through memory stage
bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };

# Should I stall or inject a bubble into Pipeline Register W?
bool W_stall = W_stat in { SADR, SINS, SHLT };
bool W_bubble = 0;
```

code/arch/pipe-nobypass-ans.hcl

Problem 4.52 Solution:

This problem is similar to Problem 4.49, but for the PIPE processor.

The following HCL code includes implementations of both the `iaddl` instruction and the `leave` instructions. You can test the solutions using the test code in the `ptest` subdirectory. Make sure you use command line argument '`-i`.'

code/arch/pipe-full-ans.hcl

```
################################################################
#    HCL Description of Control for Pipelined Y86 Processor    #
#    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010 #
################################################################

## This is the solution for the iaddl and leave problems

################################################################
#    C Include's.  Don't alter these                           #
################################################################

quote '#include <stdio.h>'
quote '#include "isa.h"'
quote '#include "pipeline.h"'
quote '#include "stages.h"'
quote '#include "sim.h"'
quote 'int sim_main(int argc, char *argv[]);'
quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'

################################################################
#    Declarations.  Do not change/remove/delete any of these   #
```

```
22 ##################################################################
23
24 ##### Symbolic representation of Y86 Instruction Codes #############
25 intsig INOP     'I_NOP'
26 intsig IHALT    'I_HALT'
27 intsig IRRMOVL  'I_RRMOVL'
28 intsig IIRMOVL  'I_IRMOVL'
29 intsig IRMMOVL  'I_RMMOVL'
30 intsig IMRMOVL  'I_MRMOVL'
31 intsig IOPL     'I_ALU'
32 intsig IJXX     'I_JMP'
33 intsig ICALL    'I_CALL'
34 intsig IRET     'I_RET'
35 intsig IPUSHL   'I_PUSHL'
36 intsig IPOPL    'I_POPL'
37 # Instruction code for iaddl instruction
38 intsig IIADDL   'I_IADDL'
39 # Instruction code for leave instruction
40 intsig ILEAVE   'I_LEAVE'
41
42 ##### Symbolic represenations of Y86 function codes            #####
43 intsig FNONE    'F_NONE'        # Default function code
44
45 ##### Symbolic representation of Y86 Registers referenced      #####
46 intsig RESP     'REG_ESP'           # Stack Pointer
47 intsig REBP     'REG_EBP'           # Frame Pointer
48 intsig RNONE    'REG_NONE'          # Special value indicating "no register"
49
50 ##### ALU Functions referenced explicitly ##########################
51 intsig ALUADD   'A_ADD'             # ALU should add its arguments
52
53 ##### Possible instruction status values                       #####
54 intsig SBUB     'STAT_BUB'      # Bubble in stage
55 intsig SAOK     'STAT_AOK'      # Normal execution
56 intsig SADR     'STAT_ADR'      # Invalid memory address
57 intsig SINS     'STAT_INS'      # Invalid instruction
58 intsig SHLT     'STAT_HLT'      # Halt instruction encountered
59
60 ##### Signals that can be referenced by control logic ##############
61
62 ##### Pipeline Register F ###########################################
63
64 intsig F_predPC 'pc_curr->pc'       # Predicted value of PC
65
66 ##### Intermediate Values in Fetch Stage ############################
67
68 intsig imem_icode  'imem_icode'      # icode field from instruction memory
69 intsig imem_ifun   'imem_ifun'       # ifun  field from instruction memory
70 intsig f_icode  'if_id_next->icode'  # (Possibly modified) instruction code
71 intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
```

```
intsig f_valC   'if_id_next->valc'   # Constant data of fetched instruction
intsig f_valP   'if_id_next->valp'   # Address of following instruction
boolsig imem_error 'imem_error'      # Error signal from instruction memory
boolsig instr_valid 'instr_valid'    # Is fetched instruction valid?

##### Pipeline Register D ########################################
intsig D_icode 'if_id_curr->icode'   # Instruction code
intsig D_rA 'if_id_curr->ra'         # rA field from instruction
intsig D_rB 'if_id_curr->rb'         # rB field from instruction
intsig D_valP 'if_id_curr->valp'     # Incremented PC

##### Intermediate Values in Decode Stage  ########################

intsig d_srcA    'id_ex_next->srca'  # srcA from decoded instruction
intsig d_srcB    'id_ex_next->srcb'  # srcB from decoded instruction
intsig d_rvalA 'd_regvala'           # valA read from register file
intsig d_rvalB 'd_regvalb'           # valB read from register file

##### Pipeline Register E ########################################
intsig E_icode 'id_ex_curr->icode'   # Instruction code
intsig E_ifun  'id_ex_curr->ifun'    # Instruction function
intsig E_valC  'id_ex_curr->valc'    # Constant data
intsig E_srcA  'id_ex_curr->srca'    # Source A register ID
intsig E_valA  'id_ex_curr->vala'    # Source A value
intsig E_srcB  'id_ex_curr->srcb'    # Source B register ID
intsig E_valB  'id_ex_curr->valb'    # Source B value
intsig E_dstE 'id_ex_curr->deste'    # Destination E register ID
intsig E_dstM 'id_ex_curr->destm'    # Destination M register ID

##### Intermediate Values in Execute Stage ########################
intsig e_valE 'ex_mem_next->vale'       # valE generated by ALU
boolsig e_Cnd 'ex_mem_next->takebranch' # Does condition hold?
intsig e_dstE 'ex_mem_next->deste'      # dstE (possibly modified to be RNONE)

##### Pipeline Register M                 ########################
intsig M_stat 'ex_mem_curr->status'     # Instruction status
intsig M_icode 'ex_mem_curr->icode'     # Instruction code
intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
intsig M_valA  'ex_mem_curr->vala'      # Source A value
intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
intsig M_valE  'ex_mem_curr->vale'      # ALU E value
intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
boolsig M_Cnd 'ex_mem_curr->takebranch' # Condition flag
boolsig dmem_error 'dmem_error'         # Error signal from instruction memory

##### Intermediate Values in Memory Stage ########################
intsig m_valM 'mem_wb_next->valm'       # valM generated by memory
intsig m_stat 'mem_wb_next->status'     # stat (possibly modified to be SADR)

##### Pipeline Register W ########################################
```

```
122 intsig W_stat 'mem_wb_curr->status'     # Instruction status
123 intsig W_icode 'mem_wb_curr->icode'     # Instruction code
124 intsig W_dstE 'mem_wb_curr->deste'      # Destination E register ID
125 intsig W_valE  'mem_wb_curr->vale'      # ALU E value
126 intsig W_dstM 'mem_wb_curr->destm'      # Destination M register ID
127 intsig W_valM  'mem_wb_curr->valm'      # Memory M value
128
129 ##################################################################
130 #    Control Signal Definitions.                                 #
131 ##################################################################
132
133 ################ Fetch Stage     ###################################
134
135 ## What address should instruction be fetched at
136 int f_pc = [
137         # Mispredicted branch.  Fetch at incremented PC
138         M_icode == IJXX && !M_Cnd : M_valA;
139         # Completion of RET instruction.
140         W_icode == IRET : W_valM;
141         # Default: Use predicted value of PC
142         1 : F_predPC;
143 ];
144
145 ## Determine icode of fetched instruction
146 int f_icode = [
147         imem_error : INOP;
148         1: imem_icode;
149 ];
150
151 # Determine ifun
152 int f_ifun = [
153         imem_error : FNONE;
154         1: imem_ifun;
155 ];
156
157 # Is instruction valid?
158 bool instr_valid = f_icode in
159         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
160           IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL, ILEAVE };
161
162 # Determine status code for fetched instruction
163 int f_stat = [
164         imem_error: SADR;
165         !instr_valid : SINS;
166         f_icode == IHALT : SHLT;
167         1 : SAOK;
168 ];
169
170 # Does fetched instruction require a regid byte?
171 bool need_regids =
```

```
172         f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
173                      IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };
174
175 # Does fetched instruction require a constant word?
176 bool need_valC =
177         f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };
178
179 # Predict next value of PC
180 int f_predPC = [
181         f_icode in { IJXX, ICALL } : f_valC;
182         1 : f_valP;
183 ];
184
185 ################ Decode Stage ######################################
186
187
188 ## What register should be used as the A source?
189 int d_srcA = [
190         D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
191         D_icode in { IPOPL, IRET } : RESP;
192         D_icode in { ILEAVE } : REBP;
193         1 : RNONE; # Don't need register
194 ];
195
196 ## What register should be used as the B source?
197 int d_srcB = [
198         D_icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL  } : D_rB;
199         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
200         D_icode in { ILEAVE } : REBP;
201         1 : RNONE;  # Don't need register
202 ];
203
204 ## What register should be used as the E destination?
205 int d_dstE = [
206         D_icode in { IRRMOVL, IIRMOVL, IOPL, IIADDL } : D_rB;
207         D_icode in { IPUSHL, IPOPL, ICALL, IRET, ILEAVE } : RESP;
208         1 : RNONE;  # Don't write any register
209 ];
210
211 ## What register should be used as the M destination?
212 int d_dstM = [
213         D_icode in { IMRMOVL, IPOPL } : D_rA;
214         D_icode in { ILEAVE } : REBP;
215         1 : RNONE;  # Don't write any register
216 ];
217
218 ## What should be the A value?
219 ## Forward into decode stage for valA
220 int d_valA = [
221         D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
```

```
222         d_srcA == e_dstE : e_valE;    # Forward valE from execute
223         d_srcA == M_dstM : m_valM;    # Forward valM from memory
224         d_srcA == M_dstE : M_valE;    # Forward valE from memory
225         d_srcA == W_dstM : W_valM;    # Forward valM from write back
226         d_srcA == W_dstE : W_valE;    # Forward valE from write back
227         1 : d_rvalA;  # Use value read from register file
228 ];
229
230 int d_valB = [
231         d_srcB == e_dstE : e_valE;    # Forward valE from execute
232         d_srcB == M_dstM : m_valM;    # Forward valM from memory
233         d_srcB == M_dstE : M_valE;    # Forward valE from memory
234         d_srcB == W_dstM : W_valM;    # Forward valM from write back
235         d_srcB == W_dstE : W_valE;    # Forward valE from write back
236         1 : d_rvalB;  # Use value read from register file
237 ];
238
239 ################ Execute Stage ######################################
240
241 ## Select input A to ALU
242 int aluA = [
243         E_icode in { IRRMOVL, IOPL } : E_valA;
244         E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IIADDL } : E_valC;
245         E_icode in { ICALL, IPUSHL } : -4;
246         E_icode in { IRET, IPOPL, ILEAVE } : 4;
247         # Other instructions don't need ALU
248 ];
249
250 ## Select input B to ALU
251 int aluB = [
252         E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
253                      IPUSHL, IRET, IPOPL, IIADDL, ILEAVE } : E_valB;
254         E_icode in { IRRMOVL, IIRMOVL } : 0;
255         # Other instructions don't need ALU
256 ];
257
258 ## Set the ALU function
259 int alufun = [
260         E_icode == IOPL : E_ifun;
261         1 : ALUADD;
262 ];
263
264 ## Should the condition codes be updated?
265 bool set_cc = E_icode in { IOPL, IIADDL } &&
266         # State changes only during normal operation
267         !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };
268
269 ## Generate valA in execute stage
270 int e_valA = E_valA;    # Pass valA through stage
271
```

```
272 ## Set dstE to RNONE in event of not-taken conditional move
273 int e_dstE = [
274         E_icode == IRRMOVL && !e_Cnd : RNONE;
275         1 : E_dstE;
276 ];
277
278 ################ Memory Stage ######################################
279
280 ## Select memory address
281 int mem_addr = [
282         M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
283         M_icode in { IPOPL, IRET, ILEAVE } : M_valA;
284         # Other instructions don't need address
285 ];
286
287 ## Set read control signal
288 bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET, ILEAVE };
289
290 ## Set write control signal
291 bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
292
293 ## Update the status
294 int m_stat = [
295         dmem_error : SADR;
296         1 : M_stat;
297 ];
298
299 ## Set E port register ID
300 int w_dstE = W_dstE;
301
302 ## Set E port value
303 int w_valE = W_valE;
304
305 ## Set M port register ID
306 int w_dstM = W_dstM;
307
308 ## Set M port value
309 int w_valM = W_valM;
310
311 ## Update processor status
312 int Stat = [
313         W_stat == SBUB : SAOK;
314         1 : W_stat;
315 ];
316
317 ################ Pipeline Register Control #########################
318
319 # Should I stall or inject a bubble into Pipeline Register F?
320 # At most one of these can be true.
321 bool F_bubble = 0;
```

```
322 bool F_stall =
323         # Conditions for a load/use hazard
324         E_icode in { IMRMOVL, IPOPL, ILEAVE } &&
325          E_dstM in { d_srcA, d_srcB } ||
326         # Stalling at fetch while ret passes through pipeline
327         IRET in { D_icode, E_icode, M_icode };
328
329 # Should I stall or inject a bubble into Pipeline Register D?
330 # At most one of these can be true.
331 bool D_stall =
332         # Conditions for a load/use hazard
333         E_icode in { IMRMOVL, IPOPL, ILEAVE } &&
334          E_dstM in { d_srcA, d_srcB };
335
336 bool D_bubble =
337         # Mispredicted branch
338         (E_icode == IJXX && !e_Cnd) ||
339         # Stalling at fetch while ret passes through pipeline
340         # but not condition for a load/use hazard
341         !(E_icode in { IMRMOVL, IPOPL, ILEAVE } && E_dstM in { d_srcA, d_srcB }) &&
342           IRET in { D_icode, E_icode, M_icode };
343
344 # Should I stall or inject a bubble into Pipeline Register E?
345 # At most one of these can be true.
346 bool E_stall = 0;
347 bool E_bubble =
348         # Mispredicted branch
349         (E_icode == IJXX && !e_Cnd) ||
350         # Conditions for a load/use hazard
351         E_icode in { IMRMOVL, IPOPL, ILEAVE } &&
352          E_dstM in { d_srcA, d_srcB};
353
354 # Should I stall or inject a bubble into Pipeline Register M?
355 # At most one of these can be true.
356 bool M_stall = 0;
357 # Start injecting bubbles as soon as exception passes through memory stage
358 bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };
359
360 # Should I stall or inject a bubble into Pipeline Register W?
361 bool W_stall = W_stat in { SADR, SINS, SHLT };
362 bool W_bubble = 0;
```

code/arch/pipe-full-ans.hcl

Problem 4.53 Solution:

See the solution to Problem 4.52. When you test this code with the scripts in `ptest`, be sure to use the command line argument '`-l`.'

Problem 4.54 Solution:

This problem requires changing the logic for predicting the PC value and the misprediction condition. It requires distinguishing between conditional and uncondiational branches. The complete HCL code is shown below. You should be able to detect whether the prediction logic is following the correct policy by doing performance checks as part of the testing with the scripts in the `ptest` directory. See the `README` file for documentation.

code/arch/pipe-nt-ans.hcl

```
1 ##################################################################
2 #    HCL Description of Control for Pipelined Y86 Processor      #
3 #    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010   #
4 ##################################################################
5
6 ## This is the solution for the branches not-taken problem
7
8 ##################################################################
9 #    C Include's.  Don't alter these                             #
10 ##################################################################
11
12 quote '#include <stdio.h>'
13 quote '#include "isa.h"'
14 quote '#include "pipeline.h"'
15 quote '#include "stages.h"'
16 quote '#include "sim.h"'
17 quote 'int sim_main(int argc, char *argv[]);'
18 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'
19
20 ##################################################################
21 #    Declarations.  Do not change/remove/delete any of these     #
22 ##################################################################
23
24 ##### Symbolic representation of Y86 Instruction Codes #############
25 intsig INOP     'I_NOP'
26 intsig IHALT    'I_HALT'
27 intsig IRRMOVL  'I_RRMOVL'
28 intsig IIRMOVL  'I_IRMOVL'
29 intsig IRMMOVL  'I_RMMOVL'
30 intsig IMRMOVL  'I_MRMOVL'
31 intsig IOPL     'I_ALU'
32 intsig IJXX     'I_JMP'
33 intsig ICALL    'I_CALL'
34 intsig IRET     'I_RET'
35 intsig IPUSHL   'I_PUSHL'
36 intsig IPOPL    'I_POPL'
37
38 ##### Symbolic represenations of Y86 function codes           #####
39 intsig FNONE    'F_NONE'        # Default function code
40
41 ##### Symbolic representation of Y86 Registers referenced     #####
42 intsig RESP     'REG_ESP'           # Stack Pointer
```

```
43 intsig RNONE    'REG_NONE'          # Special value indicating "no register"
44
45 ##### ALU Functions referenced explicitly ##########################
46 intsig ALUADD   'A_ADD'             # ALU should add its arguments
47 ## BNT: For modified branch prediction, need to distinguish
48 ## conditional vs. unconditional branches
49 ##### Jump conditions referenced explicitly
50 intsig UNCOND 'C_YES'               # Unconditional transfer
51
52 ##### Possible instruction status values                      #####
53 intsig SBUB     'STAT_BUB'      # Bubble in stage
54 intsig SAOK     'STAT_AOK'      # Normal execution
55 intsig SADR     'STAT_ADR'      # Invalid memory address
56 intsig SINS     'STAT_INS'      # Invalid instruction
57 intsig SHLT     'STAT_HLT'      # Halt instruction encountered
58
59 ##### Signals that can be referenced by control logic ##############
60
61 ##### Pipeline Register F ########################################
62
63 intsig F_predPC 'pc_curr->pc'       # Predicted value of PC
64
65 ##### Intermediate Values in Fetch Stage ###########################
66
67 intsig imem_icode  'imem_icode'     # icode field from instruction memory
68 intsig imem_ifun   'imem_ifun'      # ifun  field from instruction memory
69 intsig f_icode  'if_id_next->icode'  # (Possibly modified) instruction code
70 intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
71 intsig f_valC   'if_id_next->valc'   # Constant data of fetched instruction
72 intsig f_valP   'if_id_next->valp'   # Address of following instruction
73 boolsig imem_error 'imem_error'     # Error signal from instruction memory
74 boolsig instr_valid 'instr_valid'    # Is fetched instruction valid?
75
76 ##### Pipeline Register D ########################################
77 intsig D_icode 'if_id_curr->icode'   # Instruction code
78 intsig D_rA 'if_id_curr->ra'         # rA field from instruction
79 intsig D_rB 'if_id_curr->rb'         # rB field from instruction
80 intsig D_valP 'if_id_curr->valp'     # Incremented PC
81
82 ##### Intermediate Values in Decode Stage  #########################
83
84 intsig d_srcA    'id_ex_next->srca'  # srcA from decoded instruction
85 intsig d_srcB    'id_ex_next->srcb'  # srcB from decoded instruction
86 intsig d_rvalA 'd_regvala'           # valA read from register file
87 intsig d_rvalB 'd_regvalb'           # valB read from register file
88
89 ##### Pipeline Register E ########################################
90 intsig E_icode 'id_ex_curr->icode'   # Instruction code
91 intsig E_ifun  'id_ex_curr->ifun'    # Instruction function
92 intsig E_valC  'id_ex_curr->valc'    # Constant data
```

```
intsig E_srcA  'id_ex_curr->srca'    # Source A register ID
intsig E_valA  'id_ex_curr->vala'    # Source A value
intsig E_srcB  'id_ex_curr->srcb'    # Source B register ID
intsig E_valB  'id_ex_curr->valb'    # Source B value
intsig E_dstE 'id_ex_curr->deste'    # Destination E register ID
intsig E_dstM 'id_ex_curr->destm'    # Destination M register ID

##### Intermediate Values in Execute Stage ##########################
intsig e_valE 'ex_mem_next->vale'       # valE generated by ALU
boolsig e_Cnd 'ex_mem_next->takebranch' # Does condition hold?
intsig e_dstE 'ex_mem_next->deste'      # dstE (possibly modified to be RNONE)

##### Pipeline Register M                 ##########################
intsig M_stat 'ex_mem_curr->status'     # Instruction status
intsig M_icode 'ex_mem_curr->icode'     # Instruction code
intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
intsig M_valA  'ex_mem_curr->vala'      # Source A value
intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
intsig M_valE  'ex_mem_curr->vale'      # ALU E value
intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
boolsig M_Cnd 'ex_mem_curr->takebranch' # Condition flag
boolsig dmem_error 'dmem_error'         # Error signal from instruction memory

##### Intermediate Values in Memory Stage ###########################
intsig m_valM 'mem_wb_next->valm'       # valM generated by memory
intsig m_stat 'mem_wb_next->status'     # stat (possibly modified to be SADR)

##### Pipeline Register W ###########################################
intsig W_stat 'mem_wb_curr->status'     # Instruction status
intsig W_icode 'mem_wb_curr->icode'     # Instruction code
intsig W_dstE 'mem_wb_curr->deste'      # Destination E register ID
intsig W_valE  'mem_wb_curr->vale'      # ALU E value
intsig W_dstM 'mem_wb_curr->destm'      # Destination M register ID
intsig W_valM  'mem_wb_curr->valm'      # Memory M value

####################################################################
#    Control Signal Definitions.                                   #
####################################################################

################ Fetch Stage     ###################################

## What address should instruction be fetched at
int f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        # BNT: Changed misprediction condition
        M_icode == IJXX && M_ifun != UNCOND && M_Cnd : M_valE;
        # Completion of RET instruction.
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
```

```
143 ];
144
145 ## Determine icode of fetched instruction
146 int f_icode = [
147         imem_error : INOP;
148         1: imem_icode;
149 ];
150
151 # Determine ifun
152 int f_ifun = [
153         imem_error : FNONE;
154         1: imem_ifun;
155 ];
156
157 # Is instruction valid?
158 bool instr_valid = f_icode in
159         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
160           IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
161
162 # Determine status code for fetched instruction
163 int f_stat = [
164         imem_error: SADR;
165         !instr_valid : SINS;
166         f_icode == IHALT : SHLT;
167         1 : SAOK;
168 ];
169
170 # Does fetched instruction require a regid byte?
171 bool need_regids =
172         f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
173                      IIRMOVL, IRMMOVL, IMRMOVL };
174
175 # Does fetched instruction require a constant word?
176 bool need_valC =
177         f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
178
179 # Predict next value of PC
180 int f_predPC = [
181         # BNT: Revised branch prediction rule:
182         #   Unconditional branch is taken, others not taken
183         f_icode == IJXX && f_ifun == UNCOND : f_valC;
184         f_icode in { ICALL } : f_valC;
185         1 : f_valP;
186 ];
187
188 ################ Decode Stage ######################################
189
190
191 ## What register should be used as the A source?
192 int d_srcA = [
```

```
193         D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
194         D_icode in { IPOPL, IRET } : RESP;
195         1 : RNONE; # Don't need register
196 ];
197
198 ## What register should be used as the B source?
199 int d_srcB = [
200         D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
201         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
202         1 : RNONE;  # Don't need register
203 ];
204
205 ## What register should be used as the E destination?
206 int d_dstE = [
207         D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
208         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
209         1 : RNONE;  # Don't write any register
210 ];
211
212 ## What register should be used as the M destination?
213 int d_dstM = [
214         D_icode in { IMRMOVL, IPOPL } : D_rA;
215         1 : RNONE;  # Don't write any register
216 ];
217
218 ## What should be the A value?
219 ## Forward into decode stage for valA
220 int d_valA = [
221         D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
222         d_srcA == e_dstE : e_valE;    # Forward valE from execute
223         d_srcA == M_dstM : m_valM;    # Forward valM from memory
224         d_srcA == M_dstE : M_valE;    # Forward valE from memory
225         d_srcA == W_dstM : W_valM;    # Forward valM from write back
226         d_srcA == W_dstE : W_valE;    # Forward valE from write back
227         1 : d_rvalA;  # Use value read from register file
228 ];
229
230 int d_valB = [
231         d_srcB == e_dstE : e_valE;    # Forward valE from execute
232         d_srcB == M_dstM : m_valM;    # Forward valM from memory
233         d_srcB == M_dstE : M_valE;    # Forward valE from memory
234         d_srcB == W_dstM : W_valM;    # Forward valM from write back
235         d_srcB == W_dstE : W_valE;    # Forward valE from write back
236         1 : d_rvalB;  # Use value read from register file
237 ];
238
239 ################ Execute Stage ######################################
240
241 # BNT: When some branches are predicted as not-taken, you need some
242 # way to get valC into pipeline register M, so that
```

```
# you can correct for a mispredicted branch.
# One way to do this is to run valC through the ALU, adding 0
# so that valC will end up in M_valE

## Select input A to ALU
int aluA = [
        E_icode in { IRRMOVL, IOPL } : E_valA;
        # BNT: Use ALU to pass E_valC to M_valE
        E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX } : E_valC;
        E_icode in { ICALL, IPUSHL } : -4;
        E_icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
        E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                     IPUSHL, IRET, IPOPL } : E_valB;
        # BNT: Add 0 to valC
        E_icode in { IRRMOVL, IIRMOVL, IJXX } : 0;
        # Other instructions don't need ALU
];

## Set the ALU function
int alufun = [
        E_icode == IOPL : E_ifun;
        1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = E_icode == IOPL &&
        # State changes only during normal operation
        !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };

## Generate valA in execute stage
int e_valA = E_valA;    # Pass valA through stage

## Set dstE to RNONE in event of not-taken conditional move
int e_dstE = [
        E_icode == IRRMOVL && !e_Cnd : RNONE;
        1 : E_dstE;
];

################ Memory Stage ######################################

## Select memory address
int mem_addr = [
        M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
        M_icode in { IPOPL, IRET } : M_valA;
        # Other instructions don't need address
```

```
293 ];
294
295 ## Set read control signal
296 bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };
297
298 ## Set write control signal
299 bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
300
301 ## Update the status
302 int m_stat = [
303         dmem_error : SADR;
304         1 : M_stat;
305 ];
306
307 ## Set E port register ID
308 int w_dstE = W_dstE;
309
310 ## Set E port value
311 int w_valE = W_valE;
312
313 ## Set M port register ID
314 int w_dstM = W_dstM;
315
316 ## Set M port value
317 int w_valM = W_valM;
318
319 ## Update processor status
320 int Stat = [
321         W_stat == SBUB : SAOK;
322         1 : W_stat;
323 ];
324
325 ################ Pipeline Register Control ##########################
326
327 # Should I stall or inject a bubble into Pipeline Register F?
328 # At most one of these can be true.
329 bool F_bubble = 0;
330 bool F_stall =
331         # Conditions for a load/use hazard
332         E_icode in { IMRMOVL, IPOPL } &&
333          E_dstM in { d_srcA, d_srcB } ||
334         # Stalling at fetch while ret passes through pipeline
335         IRET in { D_icode, E_icode, M_icode };
336
337 # Should I stall or inject a bubble into Pipeline Register D?
338 # At most one of these can be true.
339 bool D_stall =
340         # Conditions for a load/use hazard
341         E_icode in { IMRMOVL, IPOPL } &&
342          E_dstM in { d_srcA, d_srcB };
```

```
343
344 bool D_bubble =
345         # Mispredicted branch
346         # BNT: Changed misprediction condition
347         (E_icode == IJXX && E_ifun != UNCOND && e_Cnd) ||
348         # Stalling at fetch while ret passes through pipeline
349         # but not condition for a load/use hazard
350         !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
351           IRET in { D_icode, E_icode, M_icode };
352
353 # Should I stall or inject a bubble into Pipeline Register E?
354 # At most one of these can be true.
355 bool E_stall = 0;
356 bool E_bubble =
357         # Mispredicted branch
358         # BNT: Changed misprediction condition
359         (E_icode == IJXX && E_ifun != UNCOND && e_Cnd) ||
360         # Conditions for a load/use hazard
361         E_icode in { IMRMOVL, IPOPL } &&
362          E_dstM in { d_srcA, d_srcB};
363
364 # Should I stall or inject a bubble into Pipeline Register M?
365 # At most one of these can be true.
366 bool M_stall = 0;
367 # Start injecting bubbles as soon as exception passes through memory stage
368 bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };
369
370 # Should I stall or inject a bubble into Pipeline Register W?
371 bool W_stall = W_stat in { SADR, SINS, SHLT };
372 bool W_bubble = 0;
```

code/arch/pipe-nt-ans.hcl

Problem 4.55 Solution:

This problem requires changing the logic for predicting the PC value and the misprediction condition. It's just a little bit more complex than Homework Problem 4.54. The complete HCL code is shown below. You should be able to detect whether the prediction logic is following the correct policy by doing performance checks as part of the testing with the scripts in the `ptest` directory. See the `README` file for documentation.

code/arch/pipe-btfnt-ans.hcl

```
1 ##################################################################
2 #    HCL Description of Control for Pipelined Y86 Processor      #
3 #    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010   #
4 ##################################################################
5
6 ## BBTFNT: This is the solution for the backward taken, forward
7 ## not-taken branch prediction problem
8
```

```
 9 ##################################################################
10 #    C Include's.  Don't alter these                             #
11 ##################################################################
12
13 quote '#include <stdio.h>'
14 quote '#include "isa.h"'
15 quote '#include "pipeline.h"'
16 quote '#include "stages.h"'
17 quote '#include "sim.h"'
18 quote 'int sim_main(int argc, char *argv[]);'
19 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'
20
21 ##################################################################
22 #    Declarations.  Do not change/remove/delete any of these     #
23 ##################################################################
24
25 ##### Symbolic representation of Y86 Instruction Codes #############
26 intsig INOP     'I_NOP'
27 intsig IHALT    'I_HALT'
28 intsig IRRMOVL  'I_RRMOVL'
29 intsig IIRMOVL  'I_IRMOVL'
30 intsig IRMMOVL  'I_RMMOVL'
31 intsig IMRMOVL  'I_MRMOVL'
32 intsig IOPL     'I_ALU'
33 intsig IJXX     'I_JMP'
34 intsig ICALL    'I_CALL'
35 intsig IRET     'I_RET'
36 intsig IPUSHL   'I_PUSHL'
37 intsig IPOPL    'I_POPL'
38
39 ##### Symbolic represenations of Y86 function codes            #####
40 intsig FNONE    'F_NONE'        # Default function code
41
42 ##### Symbolic representation of Y86 Registers referenced      #####
43 intsig RESP     'REG_ESP'           # Stack Pointer
44 intsig RNONE    'REG_NONE'          # Special value indicating "no register"
45
46 ##### ALU Functions referenced explicitly ##########################
47 intsig ALUADD   'A_ADD'             # ALU should add its arguments
48 ## BBTFNT: For modified branch prediction, need to distinguish
49 ## conditional vs. unconditional branches
50 ##### Jump conditions referenced explicitly
51 intsig UNCOND 'C_YES'               # Unconditional transfer
52
53 ##### Possible instruction status values                       #####
54 intsig SBUB     'STAT_BUB'      # Bubble in stage
55 intsig SAOK     'STAT_AOK'      # Normal execution
56 intsig SADR     'STAT_ADR'      # Invalid memory address
57 intsig SINS     'STAT_INS'      # Invalid instruction
58 intsig SHLT     'STAT_HLT'      # Halt instruction encountered
```

```
59
60 ##### Signals that can be referenced by control logic ##############
61
62 ##### Pipeline Register F ########################################
63
64 intsig F_predPC 'pc_curr->pc'        # Predicted value of PC
65
66 ##### Intermediate Values in Fetch Stage #############################
67
68 intsig imem_icode  'imem_icode'      # icode field from instruction memory
69 intsig imem_ifun   'imem_ifun'       # ifun  field from instruction memory
70 intsig f_icode  'if_id_next->icode'  # (Possibly modified) instruction code
71 intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
72 intsig f_valC   'if_id_next->valc'   # Constant data of fetched instruction
73 intsig f_valP   'if_id_next->valp'   # Address of following instruction
74 boolsig imem_error 'imem_error'      # Error signal from instruction memory
75 boolsig instr_valid 'instr_valid'    # Is fetched instruction valid?
76
77 ##### Pipeline Register D ########################################
78 intsig D_icode 'if_id_curr->icode'   # Instruction code
79 intsig D_rA 'if_id_curr->ra'         # rA field from instruction
80 intsig D_rB 'if_id_curr->rb'         # rB field from instruction
81 intsig D_valP 'if_id_curr->valp'     # Incremented PC
82
83 ##### Intermediate Values in Decode Stage  ##########################
84
85 intsig d_srcA    'id_ex_next->srca'  # srcA from decoded instruction
86 intsig d_srcB    'id_ex_next->srcb'  # srcB from decoded instruction
87 intsig d_rvalA 'd_regvala'           # valA read from register file
88 intsig d_rvalB 'd_regvalb'           # valB read from register file
89
90 ##### Pipeline Register E ########################################
91 intsig E_icode 'id_ex_curr->icode'   # Instruction code
92 intsig E_ifun  'id_ex_curr->ifun'    # Instruction function
93 intsig E_valC  'id_ex_curr->valc'    # Constant data
94 intsig E_srcA  'id_ex_curr->srca'    # Source A register ID
95 intsig E_valA  'id_ex_curr->vala'    # Source A value
96 intsig E_srcB  'id_ex_curr->srcb'    # Source B register ID
97 intsig E_valB  'id_ex_curr->valb'    # Source B value
98 intsig E_dstE 'id_ex_curr->deste'    # Destination E register ID
99 intsig E_dstM 'id_ex_curr->destm'    # Destination M register ID
100
101 ##### Intermediate Values in Execute Stage ##########################
102 intsig e_valE 'ex_mem_next->vale'       # valE generated by ALU
103 boolsig e_Cnd 'ex_mem_next->takebranch' # Does condition hold?
104 intsig e_dstE 'ex_mem_next->deste'      # dstE (possibly modified to be RNONE)
105
106 ##### Pipeline Register M                ##########################
107 intsig M_stat 'ex_mem_curr->status'     # Instruction status
108 intsig M_icode 'ex_mem_curr->icode'     # Instruction code
```

```
109 intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
110 intsig M_valA  'ex_mem_curr->vala'      # Source A value
111 intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
112 intsig M_valE  'ex_mem_curr->vale'      # ALU E value
113 intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
114 boolsig M_Cnd 'ex_mem_curr->takebranch' # Condition flag
115 boolsig dmem_error 'dmem_error'         # Error signal from instruction memory
116
117 ##### Intermediate Values in Memory Stage ##########################
118 intsig m_valM 'mem_wb_next->valm'       # valM generated by memory
119 intsig m_stat 'mem_wb_next->status'     # stat (possibly modified to be SADR)
120
121 ##### Pipeline Register W ###################################################
122 intsig W_stat 'mem_wb_curr->status'     # Instruction status
123 intsig W_icode 'mem_wb_curr->icode'     # Instruction code
124 intsig W_dstE 'mem_wb_curr->deste'      # Destination E register ID
125 intsig W_valE  'mem_wb_curr->vale'      # ALU E value
126 intsig W_dstM 'mem_wb_curr->destm'      # Destination M register ID
127 intsig W_valM  'mem_wb_curr->valm'      # Memory M value
128
129 ##############################################################################
130 #    Control Signal Definitions.                                             #
131 ##############################################################################
132
133 ################ Fetch Stage     ###################################
134
135 ## What address should instruction be fetched at
136 int f_pc = [
137         # Mispredicted branch.  Fetch at incremented PC
138         # BBTFNT: Mispredicted forward branch.  Fetch at target (now in valE)
139         M_icode == IJXX && M_ifun != UNCOND && M_valE >= M_valA
140            && M_Cnd : M_valE;
141         # BBTFNT: Mispredicted backward branch.
142         #    Fetch at incremented PC (now in valE)
143         M_icode == IJXX && M_ifun != UNCOND && M_valE < M_valA
144           && !M_Cnd : M_valA;
145         # Completion of RET instruction.
146         W_icode == IRET : W_valM;
147         # Default: Use predicted value of PC
148         1 : F_predPC;
149 ];
150
151 ## Determine icode of fetched instruction
152 int f_icode = [
153         imem_error : INOP;
154         1: imem_icode;
155 ];
156
157 # Determine ifun
158 int f_ifun = [
```

```
159         imem_error : FNONE;
160         1: imem_ifun;
161 ];
162
163 # Is instruction valid?
164 bool instr_valid = f_icode in
165         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
166           IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
167
168 # Determine status code for fetched instruction
169 int f_stat = [
170         imem_error: SADR;
171         !instr_valid : SINS;
172         f_icode == IHALT : SHLT;
173         1 : SAOK;
174 ];
175
176 # Does fetched instruction require a regid byte?
177 bool need_regids =
178         f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
179                      IIRMOVL, IRMMOVL, IMRMOVL };
180
181 # Does fetched instruction require a constant word?
182 bool need_valC =
183         f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
184
185 # Predict next value of PC
186 int f_predPC = [
187         f_icode in { ICALL } : f_valC;
188         f_icode == IJXX && f_ifun == UNCOND : f_valC; # Unconditional branch
189         f_icode == IJXX && f_valC < f_valP : f_valC; # Backward branch
190         # BBTFNT: Forward conditional branches will default to valP
191         1 : f_valP;
192 ];
193
194 ################ Decode Stage ######################################
195
196
197 ## What register should be used as the A source?
198 int d_srcA = [
199         D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
200         D_icode in { IPOPL, IRET } : RESP;
201         1 : RNONE; # Don't need register
202 ];
203
204 ## What register should be used as the B source?
205 int d_srcB = [
206         D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
207         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
208         1 : RNONE;  # Don't need register
```

```
209 ];
210
211 ## What register should be used as the E destination?
212 int d_dstE = [
213         D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
214         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
215         1 : RNONE;  # Don't write any register
216 ];
217
218 ## What register should be used as the M destination?
219 int d_dstM = [
220         D_icode in { IMRMOVL, IPOPL } : D_rA;
221         1 : RNONE;  # Don't write any register
222 ];
223
224 ## What should be the A value?
225 ## Forward into decode stage for valA
226 int d_valA = [
227         D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
228         d_srcA == e_dstE : e_valE;    # Forward valE from execute
229         d_srcA == M_dstM : m_valM;    # Forward valM from memory
230         d_srcA == M_dstE : M_valE;    # Forward valE from memory
231         d_srcA == W_dstM : W_valM;    # Forward valM from write back
232         d_srcA == W_dstE : W_valE;    # Forward valE from write back
233         1 : d_rvalA;  # Use value read from register file
234 ];
235
236 int d_valB = [
237         d_srcB == e_dstE : e_valE;    # Forward valE from execute
238         d_srcB == M_dstM : m_valM;    # Forward valM from memory
239         d_srcB == M_dstE : M_valE;    # Forward valE from memory
240         d_srcB == W_dstM : W_valM;    # Forward valM from write back
241         d_srcB == W_dstE : W_valE;    # Forward valE from write back
242         1 : d_rvalB;  # Use value read from register file
243 ];
244
245 ################ Execute Stage ######################################
246
247 # BBTFNT: When some branches are predicted as not-taken, you need some
248 # way to get valC into pipeline register M, so that
249 # you can correct for a mispredicted branch.
250 # One way to do this is to run valC through the ALU, adding 0
251 # so that valC will end up in M_valE
252
253 ## Select input A to ALU
254 int aluA = [
255         E_icode in { IRRMOVL, IOPL } : E_valA;
256         # BBTFNT: Use ALU to pass E_valC to M_valE
257         E_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX } : E_valC;
258         E_icode in { ICALL, IPUSHL } : -4;
```

```
259         E_icode in { IRET, IPOPL } : 4;
260         # Other instructions don't need ALU
261 ];
262
263 ## Select input B to ALU
264 int aluB = [
265         E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
266                      IPUSHL, IRET, IPOPL } : E_valB;
267         # BBTFNT: Add 0 to valC
268         E_icode in { IRRMOVL, IIRMOVL, IJXX } : 0;
269         # Other instructions don't need ALU
270 ];
271
272 ## Set the ALU function
273 int alufun = [
274         E_icode == IOPL : E_ifun;
275         1 : ALUADD;
276 ];
277
278 ## Should the condition codes be updated?
279 bool set_cc = E_icode == IOPL &&
280         # State changes only during normal operation
281         !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };
282
283 ## Generate valA in execute stage
284 int e_valA = E_valA;    # Pass valA through stage
285
286 ## Set dstE to RNONE in event of not-taken conditional move
287 int e_dstE = [
288         E_icode == IRRMOVL && !e_Cnd : RNONE;
289         1 : E_dstE;
290 ];
291
292 ################ Memory Stage ######################################
293
294 ## Select memory address
295 int mem_addr = [
296         M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
297         M_icode in { IPOPL, IRET } : M_valA;
298         # Other instructions don't need address
299 ];
300
301 ## Set read control signal
302 bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };
303
304 ## Set write control signal
305 bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
306
307 ## Update the status
308 int m_stat = [
```

```
309         dmem_error : SADR;
310         1 : M_stat;
311 ];
312
313 ## Set E port register ID
314 int w_dstE = W_dstE;
315
316 ## Set E port value
317 int w_valE = W_valE;
318
319 ## Set M port register ID
320 int w_dstM = W_dstM;
321
322 ## Set M port value
323 int w_valM = W_valM;
324
325 ## Update processor status
326 int Stat = [
327         W_stat == SBUB : SAOK;
328         1 : W_stat;
329 ];
330
331 ################ Pipeline Register Control ##########################
332
333 # Should I stall or inject a bubble into Pipeline Register F?
334 # At most one of these can be true.
335 bool F_bubble = 0;
336 bool F_stall =
337         # Conditions for a load/use hazard
338         E_icode in { IMRMOVL, IPOPL } &&
339          E_dstM in { d_srcA, d_srcB } ||
340         # Stalling at fetch while ret passes through pipeline
341         IRET in { D_icode, E_icode, M_icode };
342
343 # Should I stall or inject a bubble into Pipeline Register D?
344 # At most one of these can be true.
345 bool D_stall =
346         # Conditions for a load/use hazard
347         E_icode in { IMRMOVL, IPOPL } &&
348          E_dstM in { d_srcA, d_srcB };
349
350 bool D_bubble =
351         # Mispredicted branch
352         # BBTFNT: Changed misprediction condition
353         (E_icode == IJXX && E_ifun != UNCOND &&
354           (E_valC < E_valA && !e_Cnd || E_valC >= E_valA && e_Cnd)) ||
355         # Stalling at fetch while ret passes through pipeline
356         # but not condition for a load/use hazard
357         !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
358           IRET in { D_icode, E_icode, M_icode };
```

```
359
360 # Should I stall or inject a bubble into Pipeline Register E?
361 # At most one of these can be true.
362 bool E_stall = 0;
363 bool E_bubble =
364         # Mispredicted branch
365         # BBTFNT: Changed misprediction condition
366         (E_icode == IJXX && E_ifun != UNCOND &&
367           (E_valC < E_valA && !e_Cnd || E_valC >= E_valA && e_Cnd)) ||
368         # Conditions for a load/use hazard
369         E_icode in { IMRMOVL, IPOPL } &&
370          E_dstM in { d_srcA, d_srcB};
371
372 # Should I stall or inject a bubble into Pipeline Register M?
373 # At most one of these can be true.
374 bool M_stall = 0;
375 # Start injecting bubbles as soon as exception passes through memory stage
376 bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };
377
378 # Should I stall or inject a bubble into Pipeline Register W?
379 bool W_stall = W_stat in { SADR, SINS, SHLT };
380 bool W_bubble = 0;
```

code/arch/pipe-btfnt-ans.hcl

Problem 4.56 Solution:

This is an interesting problem. It gives students the experience of improving the pipeline performance. It might be interesting to have them test the program on code that copies an array from one part of memory to another, comparing the CPE with and without load bypassing.

When testing the code with the scripts in `ptest`, be sure to do the performance checks. See the instructions in the `README` file for this directory.

A. Here's the formula for a load/use hazard:

$$\text{E\_icode} \in \{\text{IMRMOVL}, \text{IPOPL}\} \;\&\&\; (\text{E\_dstM} = \text{d\_srcB} \;||\; \text{E\_dstM} = \text{d\_srcA} \;\&\&\; !\,\text{D\_icode} \in \{\text{IRMMOVL}, \text{IPUSHL}\})$$

B. The HCL code for the control logic is shown below:

code/arch/pipe-lf-ans.hcl

```
1 ##################################################################
2 #    HCL Description of Control for Pipelined Y86 Processor      #
3 #    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010   #
4 ##################################################################
5
```

```
## This is the solution to the load-forwarding problem

################################################################
#    C Include's.  Don't alter these                           #
################################################################

quote '#include <stdio.h>'
quote '#include "isa.h"'
quote '#include "pipeline.h"'
quote '#include "stages.h"'
quote '#include "sim.h"'
quote 'int sim_main(int argc, char *argv[]);'
quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'

################################################################
#    Declarations.  Do not change/remove/delete any of these   #
################################################################

##### Symbolic representation of Y86 Instruction Codes #############
intsig INOP     'I_NOP'
intsig IHALT    'I_HALT'
intsig IRRMOVL  'I_RRMOVL'
intsig IIRMOVL  'I_IRMOVL'
intsig IRMMOVL  'I_RMMOVL'
intsig IMRMOVL  'I_MRMOVL'
intsig IOPL     'I_ALU'
intsig IJXX     'I_JMP'
intsig ICALL    'I_CALL'
intsig IRET     'I_RET'
intsig IPUSHL   'I_PUSHL'
intsig IPOPL    'I_POPL'

##### Symbolic represenations of Y86 function codes            #####
intsig FNONE    'F_NONE'        # Default function code

##### Symbolic representation of Y86 Registers referenced      #####
intsig RESP     'REG_ESP'           # Stack Pointer
intsig RNONE    'REG_NONE'          # Special value indicating "no register"

##### ALU Functions referenced explicitly ##########################
intsig ALUADD   'A_ADD'             # ALU should add its arguments

##### Possible instruction status values                       #####
intsig SBUB     'STAT_BUB'      # Bubble in stage
intsig SAOK     'STAT_AOK'      # Normal execution
intsig SADR     'STAT_ADR'      # Invalid memory address
intsig SINS     'STAT_INS'      # Invalid instruction
intsig SHLT     'STAT_HLT'      # Halt instruction encountered

##### Signals that can be referenced by control logic ##############
```

```
56
57 ##### Pipeline Register F ########################################
58
59 intsig F_predPC 'pc_curr->pc'        # Predicted value of PC
60
61 ##### Intermediate Values in Fetch Stage ############################
62
63 intsig imem_icode  'imem_icode'      # icode field from instruction memory
64 intsig imem_ifun   'imem_ifun'       # ifun  field from instruction memory
65 intsig f_icode  'if_id_next->icode'  # (Possibly modified) instruction code
66 intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
67 intsig f_valC   'if_id_next->valc'   # Constant data of fetched instruction
68 intsig f_valP   'if_id_next->valp'   # Address of following instruction
69 boolsig imem_error 'imem_error'      # Error signal from instruction memory
70 boolsig instr_valid 'instr_valid'    # Is fetched instruction valid?
71
72 ##### Pipeline Register D ########################################
73 intsig D_icode 'if_id_curr->icode'   # Instruction code
74 intsig D_rA 'if_id_curr->ra'         # rA field from instruction
75 intsig D_rB 'if_id_curr->rb'         # rB field from instruction
76 intsig D_valP 'if_id_curr->valp'     # Incremented PC
77
78 ##### Intermediate Values in Decode Stage  #########################
79
80 intsig d_srcA    'id_ex_next->srca'  # srcA from decoded instruction
81 intsig d_srcB    'id_ex_next->srcb'  # srcB from decoded instruction
82 intsig d_rvalA 'd_regvala'           # valA read from register file
83 intsig d_rvalB 'd_regvalb'           # valB read from register file
84
85 ##### Pipeline Register E ########################################
86 intsig E_icode 'id_ex_curr->icode'   # Instruction code
87 intsig E_ifun  'id_ex_curr->ifun'    # Instruction function
88 intsig E_valC  'id_ex_curr->valc'    # Constant data
89 intsig E_srcA  'id_ex_curr->srca'    # Source A register ID
90 intsig E_valA  'id_ex_curr->vala'    # Source A value
91 intsig E_srcB  'id_ex_curr->srcb'    # Source B register ID
92 intsig E_valB  'id_ex_curr->valb'    # Source B value
93 intsig E_dstE 'id_ex_curr->deste'    # Destination E register ID
94 intsig E_dstM 'id_ex_curr->destm'    # Destination M register ID
95
96 ##### Intermediate Values in Execute Stage #########################
97 intsig e_valE 'ex_mem_next->vale'       # valE generated by ALU
98 boolsig e_Cnd 'ex_mem_next->takebranch' # Does condition hold?
99 intsig e_dstE 'ex_mem_next->deste'      # dstE (possibly modified to be RNONE)
100
101 ##### Pipeline Register M                ##########################
102 intsig M_stat 'ex_mem_curr->status'     # Instruction status
103 intsig M_icode 'ex_mem_curr->icode'     # Instruction code
104 intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
105 intsig M_valA  'ex_mem_curr->vala'      # Source A value
```

```
106 intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
107 intsig M_valE  'ex_mem_curr->vale'      # ALU E value
108 intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
109 boolsig M_Cnd 'ex_mem_curr->takebranch' # Condition flag
110 boolsig dmem_error 'dmem_error'         # Error signal from instruction memory
111 ## LF: Carry srcA up to pipeline register M
112 intsig M_srcA 'ex_mem_curr->srca'       # Source A register ID
113
114 ##### Intermediate Values in Memory Stage ##########################
115 intsig m_valM 'mem_wb_next->valm'       # valM generated by memory
116 intsig m_stat 'mem_wb_next->status'     # stat (possibly modified to be SADR)
117
118 ##### Pipeline Register W ###########################################
119 intsig W_stat 'mem_wb_curr->status'     # Instruction status
120 intsig W_icode 'mem_wb_curr->icode'     # Instruction code
121 intsig W_dstE 'mem_wb_curr->deste'      # Destination E register ID
122 intsig W_valE  'mem_wb_curr->vale'      # ALU E value
123 intsig W_dstM 'mem_wb_curr->destm'      # Destination M register ID
124 intsig W_valM  'mem_wb_curr->valm'      # Memory M value
125
126 ####################################################################
127 #    Control Signal Definitions.                                   #
128 ####################################################################
129
130 ################ Fetch Stage     ###################################
131
132 ## What address should instruction be fetched at
133 int f_pc = [
134         # Mispredicted branch.  Fetch at incremented PC
135         M_icode == IJXX && !M_Cnd : M_valA;
136         # Completion of RET instruction.
137         W_icode == IRET : W_valM;
138         # Default: Use predicted value of PC
139         1 : F_predPC;
140 ];
141
142 ## Determine icode of fetched instruction
143 int f_icode = [
144         imem_error : INOP;
145         1: imem_icode;
146 ];
147
148 # Determine ifun
149 int f_ifun = [
150         imem_error : FNONE;
151         1: imem_ifun;
152 ];
153
154 # Is instruction valid?
155 bool instr_valid = f_icode in
```

```
156         { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
157           IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };
158
159 # Determine status code for fetched instruction
160 int f_stat = [
161         imem_error: SADR;
162         !instr_valid : SINS;
163         f_icode == IHALT : SHLT;
164         1 : SAOK;
165 ];
166
167 # Does fetched instruction require a regid byte?
168 bool need_regids =
169         f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
170                      IIRMOVL, IRMMOVL, IMRMOVL };
171
172 # Does fetched instruction require a constant word?
173 bool need_valC =
174         f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
175
176 # Predict next value of PC
177 int f_predPC = [
178         f_icode in { IJXX, ICALL } : f_valC;
179         1 : f_valP;
180 ];
181
182 ################ Decode Stage ######################################
183
184
185 ## What register should be used as the A source?
186 int d_srcA = [
187         D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
188         D_icode in { IPOPL, IRET } : RESP;
189         1 : RNONE; # Don't need register
190 ];
191
192 ## What register should be used as the B source?
193 int d_srcB = [
194         D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
195         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
196         1 : RNONE;  # Don't need register
197 ];
198
199 ## What register should be used as the E destination?
200 int d_dstE = [
201         D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
202         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
203         1 : RNONE;  # Don't write any register
204 ];
205
```

```
## What register should be used as the M destination?
int d_dstM = [
        D_icode in { IMRMOVL, IPOPL } : D_rA;
        1 : RNONE;  # Don't write any register
];

## What should be the A value?
## Forward into decode stage for valA
int d_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == e_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];

int d_valB = [
        d_srcB == e_dstE : e_valE;    # Forward valE from execute
        d_srcB == M_dstM : m_valM;    # Forward valM from memory
        d_srcB == M_dstE : M_valE;    # Forward valE from memory
        d_srcB == W_dstM : W_valM;    # Forward valM from write back
        d_srcB == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalB;  # Use value read from register file
];

################ Execute Stage ######################################

## Select input A to ALU
int aluA = [
        E_icode in { IRRMOVL, IOPL } : E_valA;
        E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
        E_icode in { ICALL, IPUSHL } : -4;
        E_icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
        E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                     IPUSHL, IRET, IPOPL } : E_valB;
        E_icode in { IRRMOVL, IIRMOVL } : 0;
        # Other instructions don't need ALU
];

## Set the ALU function
int alufun = [
        E_icode == IOPL : E_ifun;
        1 : ALUADD;
```

```
256 ];
257
258 ## Should the condition codes be updated?
259 bool set_cc = E_icode == IOPL &&
260         # State changes only during normal operation
261         !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };
262
263 ## Generate valA in execute stage
264 ## LB: With load forwarding, want to insert valM
265 ##   from memory stage when appropriate
266 int e_valA = [
267         # Forwarding Condition
268         M_dstM == E_srcA && E_icode in { IPUSHL, IRMMOVL } : m_valM;
269         1 : E_valA;  # Use valA from stage pipe register
270 ];
271
272 ## Set dstE to RNONE in event of not-taken conditional move
273 int e_dstE = [
274         E_icode == IRRMOVL && !e_Cnd : RNONE;
275         1 : E_dstE;
276 ];
277
278 ################ Memory Stage ######################################
279
280 ## Select memory address
281 int mem_addr = [
282         M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : M_valE;
283         M_icode in { IPOPL, IRET } : M_valA;
284         # Other instructions don't need address
285 ];
286
287 ## Set read control signal
288 bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };
289
290 ## Set write control signal
291 bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
292
293 ## Update the status
294 int m_stat = [
295         dmem_error : SADR;
296         1 : M_stat;
297 ];
298
299 ## Set E port register ID
300 int w_dstE = W_dstE;
301
302 ## Set E port value
303 int w_valE = W_valE;
304
305 ## Set M port register ID
```

```
306 int w_dstM = W_dstM;
307
308 ## Set M port value
309 int w_valM = W_valM;
310
311 ## Update processor status
312 int Stat = [
313         W_stat == SBUB : SAOK;
314         1 : W_stat;
315 ];
316
317 ################ Pipeline Register Control ##########################
318
319 # Should I stall or inject a bubble into Pipeline Register F?
320 # At most one of these can be true.
321 bool F_bubble = 0;
322 bool F_stall =
323         # Conditions for a load/use hazard
324         E_icode in { IMRMOVL, IPOPL } &&
325          (E_dstM == d_srcB ||
326           (E_dstM == d_srcA && !D_icode in { IRMMOVL, IPUSHL })) ||
327         # Stalling at fetch while ret passes through pipeline
328         IRET in { D_icode, E_icode, M_icode };
329
330 # Should I stall or inject a bubble into Pipeline Register D?
331 # At most one of these can be true.
332 bool D_stall =
333         # Conditions for a load/use hazard
334         E_icode in { IMRMOVL, IPOPL } &&
335         E_icode in { IMRMOVL, IPOPL } &&
336          (E_dstM == d_srcB ||
337           (E_dstM == d_srcA && !D_icode in { IRMMOVL, IPUSHL }));
338
339 bool D_bubble =
340         # Mispredicted branch
341         (E_icode == IJXX && !e_Cnd) ||
342         # Stalling at fetch while ret passes through pipeline
343         # but not condition for a load/use hazard
344         !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
345           IRET in { D_icode, E_icode, M_icode };
346
347 # Should I stall or inject a bubble into Pipeline Register E?
348 # At most one of these can be true.
349 bool E_stall = 0;
350 bool E_bubble =
351         # Mispredicted branch
352         (E_icode == IJXX && !e_Cnd) ||
353         # Conditions for a load/use hazard
354         E_icode in { IMRMOVL, IPOPL } &&
355          (E_dstM == d_srcB ||
```

```
356           (E_dstM == d_srcA && !D_icode in { IRMMOVL, IPUSHL }));
357
358 # Should I stall or inject a bubble into Pipeline Register M?
359 # At most one of these can be true.
360 bool M_stall = 0;
361 # Start injecting bubbles as soon as exception passes through memory stage
362 bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };
363
364 # Should I stall or inject a bubble into Pipeline Register W?
365 bool W_stall = W_stat in { SADR, SINS, SHLT };
366 bool W_bubble = 0;
```

code/arch/pipe-lf-ans.hcl

Problem 4.57 Solution:

This is a hard problem. It requires carefully thinking through the design and taking care of many details. It's fun to see the working pipeline in operation, though. It also gives some insight into how more complex instructions are implemented in a pipelined system. For example, Intel's implementation of the i486 processor uses a pipeline where some instructions require multiple cycles in the decode cycle to handle the complex address computations. Controlling this requires a mechanism similar to what we present here.

The complete HCL is shown below:

code/arch/pipe-1w-ans.hcl

```
1 ##################################################################
2 #    HCL Description of Control for Pipelined Y86 Processor      #
3 #    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010   #
4 ##################################################################
5
6 ## This is a solution to the single write port problem
7 ## Overall strategy:  IPOPL passes through pipe,
8 ## treated as stack pointer increment, but not incrementing the PC
9 ## On refetch, modify fetched icode to indicate an instruction "IPOP2",
10 ## which reads from memory.
11
12 ##################################################################
13 #    C Include's.  Don't alter these                             #
14 ##################################################################
15
16 quote '#include <stdio.h>'
17 quote '#include "isa.h"'
18 quote '#include "pipeline.h"'
19 quote '#include "stages.h"'
20 quote '#include "sim.h"'
21 quote 'int sim_main(int argc, char *argv[]);'
22 quote 'int main(int argc, char *argv[]){return sim_main(argc,argv);}'
23
24 ##################################################################
25 #    Declarations.  Do not change/remove/delete any of these     #
```

```
26 ##################################################################
27
28 ##### Symbolic representation of Y86 Instruction Codes #############
29 intsig INOP     'I_NOP'
30 intsig IHALT    'I_HALT'
31 intsig IRRMOVL  'I_RRMOVL'
32 intsig IIRMOVL  'I_IRMOVL'
33 intsig IRMMOVL  'I_RMMOVL'
34 intsig IMRMOVL  'I_MRMOVL'
35 intsig IOPL     'I_ALU'
36 intsig IJXX     'I_JMP'
37 intsig ICALL    'I_CALL'
38 intsig IRET     'I_RET'
39 intsig IPUSHL   'I_PUSHL'
40 intsig IPOPL    'I_POPL'
41 # 1W: Special instruction code for second try of popl
42 intsig IPOP2    'I_POP2'
43
44 ##### Symbolic represenations of Y86 function codes            #####
45 intsig FNONE    'F_NONE'        # Default function code
46
47 ##### Symbolic representation of Y86 Registers referenced      #####
48 intsig RESP     'REG_ESP'           # Stack Pointer
49 intsig RNONE    'REG_NONE'          # Special value indicating "no register"
50
51 ##### ALU Functions referenced explicitly ##########################
52 intsig ALUADD   'A_ADD'             # ALU should add its arguments
53
54 ##### Possible instruction status values                       #####
55 intsig SBUB     'STAT_BUB'      # Bubble in stage
56 intsig SAOK     'STAT_AOK'      # Normal execution
57 intsig SADR     'STAT_ADR'      # Invalid memory address
58 intsig SINS     'STAT_INS'      # Invalid instruction
59 intsig SHLT     'STAT_HLT'      # Halt instruction encountered
60
61 ##### Signals that can be referenced by control logic ##############
62
63 ##### Pipeline Register F ###########################################
64
65 intsig F_predPC 'pc_curr->pc'        # Predicted value of PC
66
67 ##### Intermediate Values in Fetch Stage ############################
68
69 intsig imem_icode  'imem_icode'      # icode field from instruction memory
70 intsig imem_ifun   'imem_ifun'       # ifun  field from instruction memory
71 intsig f_icode  'if_id_next->icode'  # (Possibly modified) instruction code
72 intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
73 intsig f_valC   'if_id_next->valc'   # Constant data of fetched instruction
74 intsig f_valP   'if_id_next->valp'   # Address of following instruction
75 ## 1W: Provide access to the PC value for the current instruction
```

```
76 intsig f_pc     'f_pc'               # Address of fetched instruction
77 boolsig imem_error 'imem_error'      # Error signal from instruction memory
78 boolsig instr_valid 'instr_valid'    # Is fetched instruction valid?
79
80 ##### Pipeline Register D ########################################
81 intsig D_icode 'if_id_curr->icode'   # Instruction code
82 intsig D_rA 'if_id_curr->ra'         # rA field from instruction
83 intsig D_rB 'if_id_curr->rb'         # rB field from instruction
84 intsig D_valP 'if_id_curr->valp'     # Incremented PC
85
86 ##### Intermediate Values in Decode Stage  ########################
87
88 intsig d_srcA    'id_ex_next->srca'  # srcA from decoded instruction
89 intsig d_srcB    'id_ex_next->srcb'  # srcB from decoded instruction
90 intsig d_rvalA 'd_regvala'           # valA read from register file
91 intsig d_rvalB 'd_regvalb'           # valB read from register file
92
93 ##### Pipeline Register E ########################################
94 intsig E_icode 'id_ex_curr->icode'   # Instruction code
95 intsig E_ifun  'id_ex_curr->ifun'    # Instruction function
96 intsig E_valC  'id_ex_curr->valc'    # Constant data
97 intsig E_srcA  'id_ex_curr->srca'    # Source A register ID
98 intsig E_valA  'id_ex_curr->vala'    # Source A value
99 intsig E_srcB  'id_ex_curr->srcb'    # Source B register ID
100 intsig E_valB  'id_ex_curr->valb'    # Source B value
101 intsig E_dstE 'id_ex_curr->deste'    # Destination E register ID
102 intsig E_dstM 'id_ex_curr->destm'    # Destination M register ID
103
104 ##### Intermediate Values in Execute Stage #########################
105 intsig e_valE 'ex_mem_next->vale'       # valE generated by ALU
106 boolsig e_Cnd 'ex_mem_next->takebranch' # Does condition hold?
107 intsig e_dstE 'ex_mem_next->deste'      # dstE (possibly modified to be RNONE)
108
109 ##### Pipeline Register M                 ########################
110 intsig M_stat 'ex_mem_curr->status'     # Instruction status
111 intsig M_icode 'ex_mem_curr->icode'     # Instruction code
112 intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
113 intsig M_valA  'ex_mem_curr->vala'      # Source A value
114 intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
115 intsig M_valE  'ex_mem_curr->vale'      # ALU E value
116 intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
117 boolsig M_Cnd 'ex_mem_curr->takebranch' # Condition flag
118 boolsig dmem_error 'dmem_error'         # Error signal from instruction memory
119
120 ##### Intermediate Values in Memory Stage ##########################
121 intsig m_valM 'mem_wb_next->valm'       # valM generated by memory
122 intsig m_stat 'mem_wb_next->status'     # stat (possibly modified to be SADR)
123
124 ##### Pipeline Register W ########################################
125 intsig W_stat 'mem_wb_curr->status'     # Instruction status
```

```
intsig W_icode 'mem_wb_curr->icode'     # Instruction code
intsig W_dstE 'mem_wb_curr->deste'      # Destination E register ID
intsig W_valE  'mem_wb_curr->vale'      # ALU E value
intsig W_dstM 'mem_wb_curr->destm'      # Destination M register ID
intsig W_valM  'mem_wb_curr->valm'      # Memory M value

################################################################
#    Control Signal Definitions.                               #
################################################################

################ Fetch Stage     ###################################

## What address should instruction be fetched at
int f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        M_icode == IJXX && !M_Cnd : M_valA;
        # Completion of RET instruction.
        W_icode == IRET : W_valM;
        # Default: Use predicted value of PC
        1 : F_predPC;
];

## Determine icode of fetched instruction
## 1W: To split ipopl into two cycles, need to be able to
## modify value of icode,
## so that it will be IPOP2 when fetched for second time.
int f_icode = [
        imem_error : INOP;
        ## Can detected refetch of ipopl, since now have
        ## IPOPL as icode for instruction in decode.
        imem_icode == IPOPL && D_icode == IPOPL : IPOP2;
        1: imem_icode;
];

# Determine ifun
int f_ifun = [
        imem_error : FNONE;
        1: imem_ifun;
];

# Is instruction valid?
bool instr_valid = f_icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
          IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IPOP2 };

# Determine status code for fetched instruction
int f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
```

```
176         1 : SAOK;
177 ];
178 
179 # Does fetched instruction require a regid byte?
180 bool need_regids =
181         f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
182                      IPOP2,
183                      IIRMOVL, IRMMOVL, IMRMOVL };
184 
185 # Does fetched instruction require a constant word?
186 bool need_valC =
187         f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };
188 
189 # Predict next value of PC
190 int f_predPC = [
191         f_icode in { IJXX, ICALL } : f_valC;
192         ## 1W: Want to refetch popl one time
193         # (on second time f_icode will be IPOP2). Refetch popl
194         f_icode == IPOPL : f_pc;
195         1 : f_valP;
196 ];
197 
198 ################ Decode Stage ######################################
199 
200 ## W1: Strategy.  Decoding of popl rA should be treated the same
201 ## as would iaddl $4, %esp
202 ## Decoding of pop2 rA treated same as mrmovl -4(%esp), rA
203 
204 ## What register should be used as the A source?
205 int d_srcA = [
206         D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
207         D_icode in { IPOPL, IRET } : RESP;
208         1 : RNONE; # Don't need register
209 ];
210 
211 ## What register should be used as the B source?
212 int d_srcB = [
213         D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
214         D_icode in { IPUSHL, IPOPL, ICALL, IRET, IPOP2 } : RESP;
215         1 : RNONE;  # Don't need register
216 ];
217 
218 ## What register should be used as the E destination?
219 int d_dstE = [
220         D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
221         D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
222         1 : RNONE;  # Don't write any register
223 ];
224 
225 ## What register should be used as the M destination?
```

```
226 int d_dstM = [
227         D_icode in { IMRMOVL, IPOP2 } : D_rA;
228         1 : RNONE;  # Don't write any register
229 ];
230
231 ## What should be the A value?
232 ## Forward into decode stage for valA
233 int d_valA = [
234         D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
235         d_srcA == e_dstE : e_valE;    # Forward valE from execute
236         d_srcA == M_dstM : m_valM;    # Forward valM from memory
237         d_srcA == M_dstE : M_valE;    # Forward valE from memory
238         d_srcA == W_dstM : W_valM;    # Forward valM from write back
239         d_srcA == W_dstE : W_valE;    # Forward valE from write back
240         1 : d_rvalA;  # Use value read from register file
241 ];
242
243 int d_valB = [
244         d_srcB == e_dstE : e_valE;    # Forward valE from execute
245         d_srcB == M_dstM : m_valM;    # Forward valM from memory
246         d_srcB == M_dstE : M_valE;    # Forward valE from memory
247         d_srcB == W_dstM : W_valM;    # Forward valM from write back
248         d_srcB == W_dstE : W_valE;    # Forward valE from write back
249         1 : d_rvalB;  # Use value read from register file
250 ];
251
252 ################ Execute Stage ######################################
253
254 ## Select input A to ALU
255 int aluA = [
256         E_icode in { IRRMOVL, IOPL } : E_valA;
257         E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
258         E_icode in { ICALL, IPUSHL, IPOP2 } : -4;
259         E_icode in { IRET, IPOPL } : 4;
260         # Other instructions don't need ALU
261 ];
262
263 ## Select input B to ALU
264 int aluB = [
265         E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
266                      IPUSHL, IRET, IPOPL, IPOP2 } : E_valB;
267         E_icode in { IRRMOVL, IIRMOVL } : 0;
268         # Other instructions don't need ALU
269 ];
270
271 ## Set the ALU function
272 int alufun = [
273         E_icode == IOPL : E_ifun;
274         1 : ALUADD;
275 ];
```

```
276
277 ## Should the condition codes be updated?
278 bool set_cc = E_icode == IOPL &&
279         # State changes only during normal operation
280         !m_stat in { SADR, SINS, SHLT } && !W_stat in { SADR, SINS, SHLT };
281
282 ## Generate valA in execute stage
283 int e_valA = E_valA;    # Pass valA through stage
284
285 ## Set dstE to RNONE in event of not-taken conditional move
286 int e_dstE = [
287         E_icode == IRRMOVL && !e_Cnd : RNONE;
288         1 : E_dstE;
289 ];
290
291 ################ Memory Stage ######################################
292
293 ## Select memory address
294 int mem_addr = [
295         M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL, IPOP2 } : M_valE;
296         M_icode in { IRET } : M_valA;
297         # Other instructions don't need address
298 ];
299
300 ## Set read control signal
301 bool mem_read = M_icode in { IMRMOVL, IPOP2, IRET };
302
303 ## Set write control signal
304 bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };
305
306 ## Update the status
307 int m_stat = [
308         dmem_error : SADR;
309         1 : M_stat;
310 ];
311
312 ################ Write back stage ##################################
313
314 ## 1W: For this problem, we introduce a multiplexor that merges
315 ## valE and valM into a single value for writing to register port E.
316 ## DO NOT CHANGE THIS LOGIC
317 ## Merge both write back sources onto register port E
318 ## Set E port register ID
319 int w_dstE = [
320         ## writing from valM
321         W_dstM != RNONE : W_dstM;
322         1: W_dstE;
323 ];
324
325 ## Set E port value
```

```
326 int w_valE = [
327         W_dstM != RNONE : W_valM;
328         1: W_valE;
329 ];
330
331 ## Disable register port M
332 ## Set M port register ID
333 int w_dstM = RNONE;
334
335 ## Set M port value
336 int w_valM = 0;
337
338 ## Update processor status
339 int Stat = [
340         W_stat == SBUB : SAOK;
341         1 : W_stat;
342 ];
343
344 ################ Pipeline Register Control ##########################
345
346 # Should I stall or inject a bubble into Pipeline Register F?
347 # At most one of these can be true.
348 bool F_bubble = 0;
349 bool F_stall =
350         # Conditions for a load/use hazard
351         E_icode in { IMRMOVL, IPOP2 } &&
352          E_dstM in { d_srcA, d_srcB } ||
353         # Stalling at fetch while ret passes through pipeline
354         IRET in { D_icode, E_icode, M_icode };
355
356 # Should I stall or inject a bubble into Pipeline Register D?
357 # At most one of these can be true.
358 bool D_stall =
359         # Conditions for a load/use hazard
360         E_icode in { IMRMOVL, IPOP2 } &&
361          E_dstM in { d_srcA, d_srcB };
362
363 bool D_bubble =
364         # Mispredicted branch
365         (E_icode == IJXX && !e_Cnd) ||
366         # Stalling at fetch while ret passes through pipeline
367         # but not condition for a load/use hazard
368         # 1W: Changed Load/Use condition
369         !(E_icode in { IMRMOVL, IPOP2 } && E_dstM in { d_srcA, d_srcB }) &&
370           IRET in { D_icode, E_icode, M_icode };
371
372 # Should I stall or inject a bubble into Pipeline Register E?
373 # At most one of these can be true.
374 bool E_stall = 0;
375 bool E_bubble =
```

```
376         # Mispredicted branch
377         (E_icode == IJXX && !e_Cnd) ||
378         # Conditions for a load/use hazard
379         E_icode in { IMRMOVL, IPOP2 } &&
380          E_dstM in { d_srcA, d_srcB};
381
382 # Should I stall or inject a bubble into Pipeline Register M?
383 # At most one of these can be true.
384 bool M_stall = 0;
385 # Start injecting bubbles as soon as exception passes through memory stage
386 bool M_bubble = m_stat in { SADR, SINS, SHLT } || W_stat in { SADR, SINS, SHLT };
387
388 # Should I stall or inject a bubble into Pipeline Register W?
389 bool W_stall = W_stat in { SADR, SINS, SHLT };
390 bool W_bubble = 0;
```

code/arch/pipe-1w-ans.hcl

Problem 4.58 Solution:

The code with a conditional jump does much better than the code with conditional data transfers. For an array with 6 elements, the conditional data transfer code requires 44 more clock cycles. Examining the code more carefully, we can see that the jump instruction in one requires one cycle when the branch is taken and three cycles when it is not. On the other hand, the three conditional data transfer instructions always require three cycles. Sorting n elements requires $n(n-1)$ iterations of the inner loop, and so it appears that of the 30 iterations for our example, the branch was taken 22 times and not taken 8 times.

1.5 Chapter 5: Optimizing Program Performance

Problem 5.15 Solution:

This problem gives students a chance to examine machine code and perform a detailed analysis of its execution timing.

A. See Figure 1.2.

B. The critical path is formed by the addition operation updating variable `sum`. This puts a lower bound on the CPE equal to the latency of floating-point addition.

C. For integer data, the lower bound would be just 1.00. Some other resource constraint is limiting the performance.

D. The multiplication operations have longer latencies, but these are not part of a critical path of dependencies, and so they can just be pipelined through the multiplier.

Problem 5.16 Solution:

This problem gives practice applying loop unrolling.

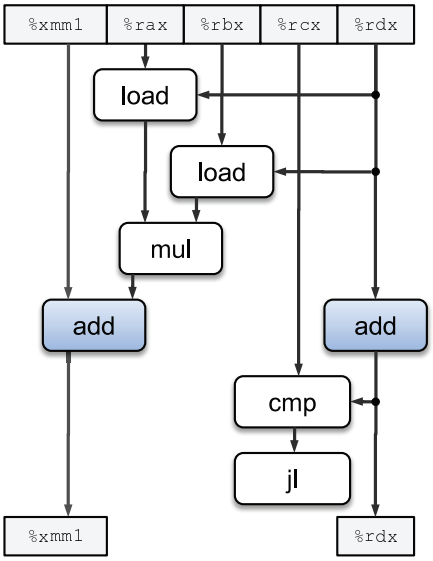


Figure 1.2: **Data-flow for function `inner4`.** The multiplication operation is not on any critical path.

```
1 void inner5(vec_ptr u, vec_ptr v, data_t *dest)
2 {
3     long int i;
4     int length = vec_length(u);
5     int limit = length-3;
6     data_t *udata = get_vec_start(u);
7     data_t *vdata = get_vec_start(v);
8     data_t sum = (data_t) 0;
9
10    /* Do four elements at a time */
11    for (i = 0; i < limit; i+=4) {
12        sum = sum
13            + udata[i]  * vdata[i]
14            + udata[i+1] * vdata[i+1]
15            + udata[i+2] * vdata[i+2]
16            + udata[i+3] * vdata[i+3];
17    }
18
19    /* Finish off any remaining elements */
20    for (; i < length; i++) {
21        sum += udata[i] * vdata[i];
22    }
23    *dest = sum;
24 }
```

A. We must perform two loads per element to read values for `udata` and `vdata`. There is only one unit to perform these loads, and it requires one cycle.

B. The performance for floating point is still limited by the 3 cycle latency of the floating-point adder.

Problem 5.17 Solution:

This exercise gives students a chance to introduce multiple accumulators..

```
1 void inner6(vec_ptr u, vec_ptr v, data_t *dest)
2 {
3     long int i;
4     int length = vec_length(u);
5     int limit = length-3;
6     data_t *udata = get_vec_start(u);
7     data_t *vdata = get_vec_start(v);
8     data_t sum0 = (data_t) 0;
9     data_t sum1 = (data_t) 0;
10    data_t sum2 = (data_t) 0;
11    data_t sum3 = (data_t) 0;
12
13    /* Do four elements at a time */
14    for (i = 0; i < limit; i+=4) {
15        sum0 += udata[i]   * vdata[i];
16        sum1 += udata[i+1] * vdata[i+1];
17        sum2 += udata[i+2] * vdata[i+2];
18        sum3 += udata[i+3] * vdata[i+3];
19    }
20
21    /* Finish off any remaining elements */
22    for (; i < length; i++) {
23        sum0 = sum0 + udata[i] * vdata[i];
24    }
25    *dest = sum0 + sum1 + sum2 + sum3;
26 }
```

A. For each element, we must perform two loads with a unit that can only load one value per clock cycle.

B. With IA32, the limited set of registers causes some of the accumulator values to be spilled into memory.

Problem 5.18 Solution:

This gives students a chance to experiment with reassociation.

```
1 void inner7(vec_ptr u, vec_ptr v, data_t *dest)
2 {
3     long int i;
4     int length = vec_length(u);
5     int limit = length-3;
6     data_t *udata = get_vec_start(u);
```

```
    data_t *vdata = get_vec_start(v);
    data_t sum = (data_t) 0;

    /* Do four elements at a time */
    for (i = 0; i < limit; i+=4) {
        sum +=
            ((udata[i]  * vdata[i]
              + udata[i+1] * vdata[i+1])
             + (udata[i+2] * vdata[i+2]
                + udata[i+3] * vdata[i+3]));
    }

    /* Finish off any remaining elements */
    for (; i < length; i++) {
        sum += udata[i] * vdata[i];
    }
    *dest = sum;
}
```

Problem 5.19 Solution:

Our fastest version writes two words of type `long unsigned` on every iteration. The code was made more complex by the need to perform an initial alignment and to avoid creating a negative upper bound in the main loop, since it uses unsigned arithmetic.

```
/* Pack characters into long, and do 2 per loop */
void *align_pack_2_memset(void *s, int c, size_t n)
{
    unsigned long word = 0;
    unsigned char *schar;
    unsigned long *slong;
    size_t cnt = 0;
    int i;

    /* Need to avoid negative numbers with size_t data */
    if (n < 3*sizeof(word))
        return basic_memset(s, c, n);

    /* Pack repeated copies of c into word */
    for (i = 0; i < sizeof(word); i++)
        word = (word << 8) | (c & 0xFF);

    /* Step through characters until get alignment */
    schar = (unsigned char *) s;
    while ((unsigned long) schar % sizeof(word) != 0) {
        *schar++ = (unsigned char) c;
        cnt++;
    }

```

```
25      /* Step through 8 or 16 characters at a time */
26      slong = (unsigned long *) schar;
27      while (cnt < n-2*sizeof(word)+1) {
28          *slong = word;
29          *(slong+1) = word;
30          cnt += 2*sizeof(word);
31          slong += 2;
32      }
33
34      /* Finish off single characters */
35      schar = (unsigned char *) slong;
36      while (cnt < n) {
37          *schar++ = (unsigned char) c;
38          cnt++;
39      }
40      return s;
41  }
```

Problem 5.20 Solution:

Polynomial evaluation provides a wealth of opportunities for rearranging and reorganizing expressions. We identified three fundamentally different methods of writing faster versions of a polynomial evaluation routine. We show the 4-way versions for each of them.

The first method is a variant on direct evaluation with reassociation. On our machine, this function achieved a CPE of 2.25. A version with 8-way unrolling achieved a CPE of 1.45.

```
1  /* Reassociate */
2  double poly4(double a[], double x, int degree)
3  {
4      long int i;
5      double result = a[0];
6      double x2 = x*x;
7      double x3 = x2*x;
8      double x4 = x2*x2;
9      double xpwr = x;
10     for (i = 1; i <= degree-3; i+=4) {
11         double nxpwr = x4 * xpwr;
12         result += ((a[i] + x*a[i+1]) + (x2*a[i+2] + x3*a[i+3])) * xpwr;
13         xpwr = nxpwr;
14     }
15     for (; i <= degree; i++) {
16         result += a[i] * xpwr;
17         xpwr = x * xpwr;
18     }
19     return result;
20 }
```

The second method is a variant on direct evaluation with multiple accumulators. It combines the accumulated sums using Horner's method. This approach seems like it should achieve near-optimal results, but the

4-way version has a CPE of 2.15, and the 8-way version has a CPE of 1.76.

```
1 /* Accumulate multiple values in parallel */
2 double poly4x(double a[], double x, int degree)
3 {
4     long int i;
5     double result0 = a[0];
6     double result1 = 0;
7     double result2 = 0;
8     double result3 = 0;
9     double x4 =      (x*x)*(x*x);
10    double xpwr = x;
11    for (i = 1; i <= degree-3; i+=4) {
12        double nxpwr = x4 * xpwr;
13        result0 += a[i] * xpwr;
14        result1 += a[i+1] * xpwr;
15        result2 += a[i+2] * xpwr;
16        result3 += a[i+3] * xpwr;
17        xpwr = nxpwr;
18    }
19    result0 += x*(result1 + x*(result2 + x*result3));
20    for (; i <= degree; i++) {
21        result0 += a[i] * xpwr;
22        xpwr = x * xpwr;
23    }
24    return result0;
25 }
```

The final method is a variant on Horner's method with reassociation. This function gave a CPE of 2.09.

```
1 /* Apply Horner's method */
2 double poly4h(double a[], double x, int degree)
3 {
4     long int i;
5     double x2 = x*x;
6     double x3 = x2*x;
7     double x4 = x2*x2;
8     double result = a[degree];
9     for (i = degree-1; i >= 3; i-=4)
10        result = ((a[i-3] + x*a[i-2]) + (x2*a[i-1] + x3*a[i])) + x4*result;
11    for (; i >= 0; i--)
12        result = a[i] + x*result;
13    return result;
14 }
```

Problem 5.21 Solution:

Prefix sum allows for interesting optimizations. Making use of reassociation requires some cleverness.

Here are the versions that use 2 and 3-way unrolling. We have ordered the statements in a way that results in a better instruction schedule.

```
1  /* Compute prefix sum of vector a */
2  /* Use 2-way loop unrolling + lookahead */
3  void psum2a(float a[], float p[], int cnt)
4  {
5      long int i;
6      /* last_val holds p[i-1]; val holds p[i+1] */
7      float last_val, val;
8      last_val = p[0] = a[0];
9      for (i = 1; i < cnt-1; i+=2) {
10         float a01 = a[i] + a[i+1];
11         val    = last_val + a01;
12         p[i]   = last_val + a[i];
13         p[i+1] = val;
14         last_val = val;
15     }
16     /* Finish remaining elements */
17     for (; i < cnt; i++) {
18         val  = last_val + a[i];
19         p[i] = val;
20         last_val = val;
21     }
22 }
```

```
1  /* Compute prefix sum of vector a */
2  /* Use 3-way loop unrolling + lookahead */
3  void psum3a(float a[], float p[], int cnt)
4  {
5      long int i;
6      /* last_val holds p[i-1]; val holds p[i+2] */
7      float last_val, val;
8      last_val = p[0] = a[0];
9      for (i = 1; i < cnt-2; i+=3) {
10         p[i]   = last_val + a[i];
11         float a01   = a[i] + a[i+1];
12         p[i+1] = last_val + a01;
13         float a012  = a01  + a[i+2];
14         val    = last_val + a012;
15         p[i+2] = val;
16         last_val = val;
17     }
18     /* Finish remaining elements */
19     for (; i < cnt; i++) {
20         val  = last_val + a[i];
21         p[i] = val;
22         last_val = val;
23     }
```

```
24 }
```

Problem 5.22 Solution:

This problem is a simple application of Amdahl's law. Speeding up part B by 3 gives an overall speedup of $1/(.2+.3/3+.5) = 1.25$. Speeding up part C by 1.5 gives an overall speedup of $1/(.2+.3+.5/1.5) = 1.2$. So the best strategy is to optimize part B.

1.6 Chapter 6: The Memory Hierarchy

Problem 6.23 Solution:

This is a thought problem to help the students understand the geometry factors that determine the capacity of a disk. Let r be the radius of the platter and xr be the radius of the hole. The number of bits/track is proportional to $2\pi xr$ (the circumference of the innermost track), and the number of tracks is proportional to $(r - xr)$. Thus, the total number of bits is proportional to $2\pi xr(r - xr)$. Setting the derivative to zero and solving for x gives $x = 1/2$. In words, the radius of the hole should be 1/2 the radius of the platter to maximize the bit capacity.

Problem 6.24 Solution:

The average rotational latency (in ms) is

$$\begin{aligned} T_{avg\,rotation} &= \textit{1/2} \times T_{max\,rotation} \\ &= \textit{1/2} \times (\textit{60 secs / 15,000 RPM}) \times \textit{1000 ms/sec} \\ &\approx \textit{2 ms}. \end{aligned}$$

The average transfer time is

$$\begin{aligned} T_{avg\,transfer} &= (\textit{60 secs / 15,000 RPM}) \times \textit{1/800 sectors/track} \times \textit{1000 ms/sec} \\ &\approx \textit{0.005 ms}. \end{aligned}$$

Putting it all together, the total estimated access time is

$$\begin{aligned} T_{access} &= T_{avg\,seek} + T_{avg\,rotation} + T_{avg\,transfer} \\ &= \textit{4 ms} + \textit{2 ms} + \textit{0.005 ms} \\ &= \textit{6.005 ms}. \end{aligned}$$

Problem 6.25 Solution:

This is a good check of the student's understanding of the factors that affect disk performance. It's a nice model problem that can be easily changed from term to term. First we need to determine a few basic properties of the file and the disk. The file consists of 4,000 512-byte logical blocks. For the disk, $T_{avg\,seek}$ = 4 ms, $T_{max\,rotation}$ = 4 ms, and $T_{avg\,rotation}$ = 2 ms.

A. *Best case:* In the optimial case, the blocks are mapped to contiguous sectors, on the same cylinder, that can be read one after the other without moving the head. Once the head is positioned over the first sector it takes 4 full rotations (1,000 sectors per rotation) of the disk to read all 4,000 blocks. So the total time to read the file is $T_{avg\,seek} + T_{avg\,rotation} + 4 * T_{max\,rotation} = 4 + 2 + 16 = 22$ ms.

B. *Random case:* In this case, where blocks are mapped randomly to sectors, reading each of the 4,000 blocks requires $T_{avg\,seek} + T_{avg\,rotation}$ ms, so the total time to read the file is $(T_{avg\,seek} + T_{avg\,rotation}) * 4,000 = 24,000$ ms (24 seconds).

Problem 6.26 Solution:

This problem gives the students more practice in working with address bits. Some students hit a conceptual wall with this idea of partitioning address bits. In our experience, having them do these kinds of simple drills is helpful.

	m	C	B	E	S	t	s	b
1.	32	1024	4	4	64	24	6	2
2.	32	1024	4	256	1	30	0	2
3.	32	1024	8	1	128	22	7	3
4.	32	1024	8	128	1	29	0	3
5.	32	1024	32	1	32	22	5	5
6.	32	1024	32	4	8	24	3	5

Problem 6.27 Solution:

Here's another set of simple drills that require the students to manipulate address bits. These are a bit harder than the previous problem because they test different relationships betweeen sizes of things and the number of bits.

Cache	m	C	B	E	S	t	s	b
1.	32	2048	8	1	256	21	8	3
2.	32	2048	4	4	128	23	7	2
3.	32	1024	2	8	64	25	6	1
4.	32	1024	32	2	16	23	4	5

Problem 6.28 Solution:

This is an inverse cache indexing problem (akin to Problem 6.30) that requires the students to work backwards from the contents of the cache to derive a set of addresses that hit in a particular set. Students must know cache indexing cold to solve this style of problem.

A. Set 1 contains two valid lines: Line 0 and Line 1. Line 0 has a tag of `0x45`. There are four bytes in each block, and thus four addresses will hit in Line 0. These addresses have the binary form `0 1000 1010 01xx`. Thus, the following four hex addresses will hit in Line 0 of Set 1: `0x08a4`, `0x08a5`, `0x08a6`, and `0x08a7`. Similarly, the following four addresses will hit in Line 1 of Set 1: `0x0704`, `0x0705`, `0x0706`, `0x0707`.

B. Set 6 contains one valid line with a tag of `0x91`. Since there is only one valid line in the set, four addresses will hit. These addresses have the binary form `1 0010 0011 10xx`. Thus, the four hex addresses that hit in Set 6 are: `0x1238`, `0x1239`, `0x123a`, and `0x123b`.

Problem 6.29 Solution:

Another inverse cache indexing problem, using the same cache as the previous problem.

A. Set 2 contains no valid lines, so no addresses will hit.

B. Set 4 contains two valid lines: Line 0 and Line 1. Line 0 has a tag of `0xC7`. There are four bytes in each block, and thus four addresses will hit in Line 0. These addresses have the binary form `1 1000 1111 00xx`. Thus, the following four hex addresses will hit in Line 0 of Set 4: `0x18f0`, `0x18f1`, `0x18f2`, and `0x18f3`.

   Similarly, the following four addresses will hit in Line 1 of Set 4: `0x00b0`, `0x00b1`, `0x00b2`, `0x00b3`.

C. Set 5 contains one valid line, Line 0, with a tag of `0x71`. Addresses that hit in this line have the binary form `0 1110 0011 01xx`. Thus, the following four hex addresses will hit in Line 0 of Set 5: `0x0e34`, `0x0e35`, `0x0e36`, `0x0e37`.

D. Set 7 contains one valid line, Line 1, with a tag of `0xde`. Addresses that hit in this line have the binary form `1 1011 1101 11xx`. Thus, the following four hex addresses will hit in Line 1 of Set 7: `0x1bdc`, `0x1bdd`, `0x1bde`, `0x1bdf`.

Problem 6.30 Solution:

This problem is a straightforward test of the student's ability to work through some simple cache translation and lookup operations.

A. CT: [11–4], CI: [3–2], CO: [1–0]

B.

Operation	Address	Hit?	Read Value (or Unknown)
Read	0x834	No	Unknown
Write	0x836	Yes	(not applicable)
Read	0xFFD	Yes	C0

Problem 6.31 Solution:

This is the first in a series of four related problems on basic cache operations. This first problem sets the stage and serves as a warmup. The next three problems use the cache defined in this first problem.

A. Cache size: $C = 128$ bytes.

B. Address fields: CT: [12-5] CI: [4-2] CO: [1-0]

Problem 6.32 Solution:

Address `0x071A`

A. Address format (one bit per box):

12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	1	0	0	0	1	1	0	1	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. Memory reference:

Parameter	Value
Block Offset (CO)	0x2
Index (CI)	0x6
Cache Tag (CT)	0x38
Cache Hit? (Y/N)	Y
Cache Byte returned	0xEB

Problem 6.33 Solution:

Address `0x16E8`

A. Address format (one bit per box):

12	11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	1	0	1	1	1	0	1	0	0	0
CT	CT	CT	CT	CT	CT	CT	CT	CI	CI	CI	CO	CO

B. Memory reference:

Parameter	Value
Block Offset (CO)	0x0
Index (CI)	0x2
Cache Tag (CT)	0xB7
Cache Hit? (Y/N)	N
Cache Byte returned	–

Problem 6.34 Solution:

There are two valid lines in Set 2, the first with a tag of `0xBC`, and the second with a tag of `0xB6`. The addresses that hit in the first line have the binary form `1 0111 1000 10xx`, which corresponds to the address range of `0x1788` – `0x178b`. Similarly, the addresses that hit in the second line have the binary form 1 0110 1100 10xx, and thus an address range of `0x16c8` – `0x16cb`.

Problem 6.35 Solution:

This problem is tougher than it looks. The approach is similar to the solution to Problem 6.18. The cache is not large enough to hold both arrays. References to cache lines for one array evict recently loaded cache lines from the other array.

`dst` array				
	col 0	col 1	col 2	col 3
row 0	m	m	m	m
row 1	m	m	m	m
row 2	m	m	m	m
row 3	m	m	m	m

`src` array				
	col 0	col 1	col 2	col 3
row 0	m	m	h	m
row 1	m	h	m	h
row 2	m	m	h	m
row 3	m	h	m	h

Problem 6.36 Solution:

In this case, the cache is large enough to hold both arrays, so the only misses are the initial cold misses.

`dst` array				
	col 0	col 1	col 2	col 3
row 0	m	h	h	h
row 1	m	h	h	h
row 2	m	h	h	h
row 3	m	h	h	h

`src` array				
	col 0	col 1	col 2	col 3
row 0	m	h	h	h
row 1	m	h	h	h
row 2	m	h	h	h
row 3	m	h	h	h

Problem 6.37 Solution:

This style of problem (and the ones that follow) requires a practical high-level analysis of the cache behavior, rather than the more tedious step-by-step analysis that we use when we are first teaching students how caches work. We always include a problem of this type on our exams because it tests a skill the students will need as working programmers: the ability to look at code and get a feel for how well it uses the caches. This particular problem is a nice introduction to this type of high-level analysis.

A. Case 1: Assume the cache is 512-bytes, direct-mapped, with 16-byte cache blocks. What is the miss rate? In this case, each access to `x[1][i]` conflicts with previous access to `x[0][i]`, so the miss rate is 100%.

B. Case 2: What is the miss rate if we double the cache size to 1024 bytes? If we double the cache size, then the entire array fits in the cache, so the only misses are the cold (compulsary) misses for each new block. Since each block holds four array items, the miss rate is 25%.

C. Case 3: Now assume the cache is 512 bytes, 2-way set associative using an LRU replacement policy, with 16-byte cache blocks. What is the cache miss rate? Increasing the associativity removes the conflict misses that occurred in the direct mapped cache of Case 1. The only misses are the cold misses when each block is loaded, so the miss rate is 25%.

D. For Case 3, will a larger cache size help to reduce the miss rate? No. Even if the cache were infinitely large, we would still have the compulsary misses required to load each new cache block.

E. For Case 3, will a larger block size help to reduce the miss rate? Yes. A larger block size would reduce the number of compulsary misses by an amount inversely proportional to the increase. For example, if we doubled the block size, we decrease the miss rate by half.

Problem 6.38 Solution:

Here are the miss rates for the different functions and values of `N`:

Function	`N = 64`	`N = 60`
`sumA`	25%	25%
`sumB`	100%	25%
`sumC`	50%	25%

Problem 6.39 Solution:

In this problem, each cache line holds two 16-byte `point_color` structures. The `square` array is $256 \times 16 = 4096$ bytes and the cache is 2048 bytes, so the cache can only hold half of the array. Since the code employs a row-wise stride-1 reference pattern, the miss pattern for each cache line is a miss, followed by 7 hits.

A. What is the total number of writes? 1024 writes.

B. What is the total number of writes that miss in the cache? 128 misses.

C. What is the miss rate? $128/1024 = 12.5\%$.

Problem 6.40 Solution:

Since the cache cannot hold the entire array, the column-wise scan of the second half of the array evicts the lines loaded during the scan of the first half. So for every structure, we have a miss followed by 3 hits.

A. What is the total number of writes? 1024 writes.

B. What is the total number of writes that miss in the cache? 256 writes.

C. What is the miss rate? $256/1024 = 25\%$.

Problem 6.41 Solution:

Both loops access the array in row-major order. The first loop performs 256 writes. Since each cache line holds two structures, half of these references hit and half miss. The second loop performs a total of 768 writes. For each pair of structures, there is an initial cold miss, followed by 5 hits. So this loop experiences a total of 128 misses. Combined, there are $256 + 768 = 1024$ writes, and $128 + 128 = 256$ misses.

A. What is the total number of writes? 1024 writes.

B. What is the total number of writes that miss in the cache? 256 writes.

C. What is the miss rate? $256/1024 = 25\%$.

Problem 6.42 Solution:

Each `pixel` structure is 4 bytes, so each 4-byte cache line holds exactly one structure. For each structure, there is a miss, followed by three hits, for a miss rate of 25%.

Problem 6.43 Solution:

This code visits the array of `pixel` structures in row-major order. The cache line holds exactly one structure. Thus, for each structure we have a miss, followed by three hits, for a miss rate of 25%.

Problem 6.44 Solution:

In this code each loop iteration zeros the entire 4-byte structure by writing a 4-byte integer zero. Thus, although there are only 640×480 writes, each of these writes misses. Thus, the miss rate is 100%.

Problem 6.44 Solution:

In this code each loop iteration zeros the entire 4-byte structure by writing a 4-byte integer zero. Thus, although there are only 640×480 writes, each of these writes misses. Thus, the miss rate is 100%.

Problem 6.45 Solution:

Solution approach: Use the `mountain` program to generate a graph similar to Figure 6.44, which shows a slice through the mountain with constant stride and varying working set size. Do the same analysis we did in the text. Each relatively flat region of the graph corresponds to a different level in the hierarchy. As working set size increases, a transition from one flat region to another at size x indicates a cache size of x.

Problem 6.46 Solution:

No solution provided.

Problem 6.47 Solution:

No solution provided.

1.7 Chapter 7: Linking

Problem 7.6 Solution:

This problem builds on Problem 7.1 by adding some functions and variables that are declared with the `static` attribute. The main idea for the students to understand is that static symbols are local to the module that defines them, and are not visible to other modules.

Symbol	swap.o .symtab entry?	Symbol type	Module where defined	Section
buf	yes	extern	main.o	.data
bufp0	yes	global	swap.o	.data
bufp1	yes	local	swap.o	.bss
swap	yes	global	swap.o	.text
temp	no	—	—	—
incr	yes	local	swap.o	.text
count	yes	local	swap.o	.data

Problem 7.7 Solution:

This is a good example of the kind of silent nasty bugs that can occur because of quirks in the linker's symbol resolution algorithm. The programming error in this case is due to the fact that both modules define a weak global symbol `x`, which is then resolved silently by the linker (Rule 3). We can fix the bug by simply defining `x` with the `static` attribute, which turns it into a local linker symbol, and thus limits its scope to a single module:

```
1 static double x;
2
3 void f() {
4       x = -0.0;
5 }
```

Problem 7.8 Solution:

This is another problem in the spirit of Problem 7.2 that tests the student's understanding of how the linker resolves global symbols, and the kinds of errors that can result if they are not careful.

A. Because Module 2 defines `main` with the `static` attribute, it is a local symbol, and thus there are no multiply-defined global symbols. Each module refers to its own definition of `main`. This is an important idea; make sure students understand the impact of the `static` attribute and how it limits the scope of function and variable symbols.

   (a) `REF(main.1) --> DEF(main.1)`

   (b) `REF(main.2) --> DEF(main.2)`

B. Here we have two weak definitions of `x`, so the symbol resolution in this case is UNKNOWN (Rule 3).

C. This is an ERROR, since there are two strong definitions of `x` (Rule 1).

Problem 7.9 Solution:

This problem is a nice example of why it pays to have a working understanding of linkers. The output of the program is incomprehensible until you realize that linkers are just dumb symbol resolution and relocation machines. Because of Rule 2, the strong symbol associated with the function `main` in `m1.o` overrides

the weak symbol associated with the variable `main` in `m2.o`. Thus, the reference to variable `main` in `m2` resolves to the value of symbol `main`, which in this case is the address of the first byte of function `main`. This byte contains the hex value `0x55`, which is the binary encoding of `pushl %ebp`, the first instruction in procedure `main`!

Problem 7.10 Solution:

These are more drills, in the spirit of Problem 7.3, that help the students understand how linkers use static libraries when they resolve symbol references.

A. `gcc p.o libx.a`

B. `gcc p.o libx.a liby.a libx.a`

C. `gcc p.o libx.a liby.a libx.a libz.a`

Problem 7.11 Solution:

This problem is a sanity check to make sure the students understand the difference between `.data` and `.bss`, and why the distinction exists in the first place. The first part of the runtime data segment is initialized with the contents of the `.data` section in the object file. The last part of the runtime data segment is `.bss`, which is always initialized to zero, and which doesn't occupy any actual space in the executable file. Thus the discrepancy between the runtime data segment size and the size of the chunk of the object file that initializes it.

Problem 7.12 Solution:

This problem tests whether the students have grasped the concepts of relocation records and relocation. The solution approach is to mimic the behavior of the linker: use the relocation records to identify the locations of the references, and then either compute the relocated absolute addresses using the algorithm in Figure 7.9, or simply extract them from the relocated instructions in Figure 7.10. There are a couple of things to notice about the relocatable object file in Figure 7.19:

- The `movl` instruction in line 8 contains *two* references that need to be relocated.
- The instructions in lines 5 and 8 contain references to `buf[1]` with an initial value of `0x4`. The relocated addresses are computed as `ADDR(buf) + 4`.

Line # in Fig.7.10	Address	Value
15	`0x80483cb`	`0x004945c`
16	`0x80483d0`	`0x0049458`
18	`0x80483d8`	`0x0049548`
18	`0x80483dc`	`0x0049458`
23	`0x80483e7`	`0x0049548`

Problem 7.13 Solution:

The next two problems require the students to derive the relocation records from the C source and the disassembled relocatable. The best solution approach is to learn how to use `objdump` and then use `objdump` to extract the relocation records from the executable.

A. Relocation entries for the `.text` section:

```
1 RELOCATION RECORDS FOR [.text]:
2 OFFSET   TYPE              VALUE
3 00000012 R_386_PC32        p3
4 00000019 R_386_32          xp
5 00000021 R_386_PC32        p2
```

B. Relocation entries for `.data` section:

```
1 RELOCATION RECORDS FOR [.data]:
2 OFFSET   TYPE              VALUE
3 00000004 R_386_32          x
```

Problem 7.14 Solution:

A. Relocation entries for the `.text` section:

```
1 RELOCATION RECORDS FOR [.text]:
2 OFFSET   TYPE              VALUE
3 00000011 R_386_32          .rodata
```

B. Relocation entries for the `.rodata` section:

```
1 RELOCATION RECORDS FOR [.rodata]:
2 OFFSET   TYPE              VALUE
3 00000000 R_386_32          .text
4 00000004 R_386_32          .text
5 00000008 R_386_32          .text
6 0000000c R_386_32          .text
7 00000010 R_386_32          .text
8 00000014 R_386_32          .text
```

Problem 7.15 Solution:

A. On our system, `libc.a` has 1082 members and `libm.a` has 373 members.

```
unix> ar -t /usr/lib/libc.a | wc -l
    1082
unix> ar -t /usr/lib/libm.a | wc -l
    373
```

B. Interestingly, the code in the `.text` section is identical, whether a program is compiled using `-g` or not. The difference is that the "`-O2 -g`" object file contains debugging info in the `.debug` section, while the "`-O2`" version does not.

C. On our system, the `gcc` driver uses the standard C library (`libc.so.6`) and the dynamic linker (`ld-linux.so.2`):

```
linux> ldd /usr/local/bin/gcc
   libc.so.6 => /lib/libc.so.6 (0x4001a000)
   /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

1.8 Chapter 8: Exceptional Control Flow

Problem 8.9 Solution:

Process pair	Concurrent?
AB	n
AC	y
AD	y
BC	y
BD	y
CD	y

Problem 8.10 Solution:

A. Called once, returns twice: `fork`

B. Called once, never returns: `execve` and `longjmp`.

C. Called once, returns one or more times: `setjmp`.

Problem 8.11 Solution:

This program has the same process graph as the program in Figure 8.16(c). There are a total of four processes, each of which prints a single "hello" line. Thus, the program prints four "hello" lines.

Problem 8.12 Solution:

This program has the same process graph as the program in Figure 8.16(c). There are four processes, each of which prints one "hello" line in `doit` and one "hello" line in `main` after it returns from `doit`. Thus, the program prints a total of eight "hello" lines.

Problem 8.13 Solution:

This problem is a simple variant of Problem 8.2. The parent process prints

```
x=4
x=3
```

and the child process prints

```
x=2
```

Thus, any of the following sequences represents a possible output:

```
x=4       x=4       x=2
x=3       x=2       x=4
x=2       x=3       x=3
```

Problem 8.14 Solution:

The program consists of three processes: the original parent, its child, and its grandchild. Each of these processes executes a single `printf` and then terminates. Thus, the program prints three "hello" lines.

Problem 8.15 Solution:

This program is identical to the program in Problem 8.14, except that the call to `exit` in line 8 has been replaced by a `return` statement. The process hierarchy is identical, consisting of a parent, a child, and a grandchild. And as before, the parent executes a single `printf`. However, because of the `return` statement, the child and grandchild each execute two `printf` statements. Thus, the program prints a total of five output lines.

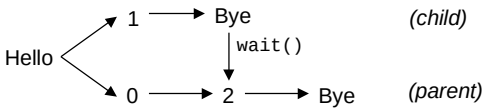
Problem 8.16 Solution:

The parent initializes `counter` to 1, then creates the child, which decrements `counter` and terminates. The parent waits for the child to terminate, then increments `counter` and prints the result. Remember, each process has its own separate address space, so the decrement by the child has no impact on the parent's copy of `counter`. Thus the output is:

```
counter = 2
```

Problem 8.17 Solution:

This problem is a nice way to check the students' understanding of the interleaved execution of processes. It also their first introduction to the idea of synchronization. In this case, the `wait` function in the parent will not complete until the child has terminated. The key idea is that any topological sort of the following DAG is a possible output:

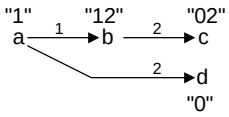


Thus, there are only three possible outcomes (each column is an outcome):

```
Hello   Hello   Hello
1       1       0
Bye     0       1
0       Bye     Bye
2       2       2
Bye     Bye     Bye
```

Problem 8.18 Solution:

This problem really tests the students' understanding of concurrent process execution. The most systematic solution approach is to draw the process hierarchy, labeling each node with the output of the corresponding process:



For each process, the kernel preserves the ordering of its `printf` statements, but otherwise can interleave the statements arbitrarily. Thus, any topological sort of the following DAG represents a possible output:

$$1 \quad 1 \rightarrow 2 \quad 0 \rightarrow 2 \quad 0$$

A. 112002 (possible)

B. 211020 (not possible)

C. 102120 (possible)

D. 122001 (not possible)

E. 100212 (possible)

Problem 8.19 Solution:

This function calls the `fork` function a total of $2^n - 1$ times. The parent and each child print a line of output, for a total of 2^n lines of output.

Problem 8.20 Solution:

This is an easy problem for students who understand the `execve` function and the structure of the `argv` and `envp` arrays. Notice that a correct solution must pass a pointer to the `envp` array (the global `environ` pointer on our system) to correctly mimic the behavior of `/bin/ls`.

code/ecf/myls-ans.c

```
1 #include "csapp.h"
2
3 int main(int argc, char **argv) {
4   Execve("/bin/ls", argv, environ);
5   exit(0);
6 }
```

code/ecf/myls-ans.c

Problem 8.21 Solution:

Drawing the process graph reveals that this program has only two possible output sequences: “abc” or “bac”.

Problem 8.22 Solution:

The `system` man page provides a basic template for implementing the `mysystem` function. The version the students implement for this problem requires somewhat different return code processing.

code/ecf/mysystem-ans.c

```
1 #include "csapp.h"
2
3 int mysystem(char *command)
4 {
5     pid_t pid;
6     int status;
7
8     if (command == NULL)
9         return -1;
10
11    if ((pid = fork()) == -1)
12        return -1;
13
14    if (pid == 0) { /* child */
15        char *argv[4];
16        argv[0] = "sh";
17        argv[1] = "-c";
18        argv[2] = command;
19        argv[3] = NULL;
20        execve("/bin/sh", argv, environ);
21        exit(-1); /* control should never reach here */
22    }
23
24    /* parent */
25    while (1) {
26        if (waitpid(pid, &status, 0) == -1) {
27            if (errno != EINTR) /* restart waitpid if interrupted */
28                return -1;
29        }
30        else {
```

```
31              if (WIFEXITED(status))
32                  return WEXITSTATUS(status);
33              else
34                  return status;
35          }
36      }
37  }
```

code/ecf/mysystem-ans.c

Problem 8.23 Solution:

Signals cannot be used to count events in other processes because signals are not queued. Solving this problem requires inter-process communication (IPC) mechanisms (not discussed in the text), or threads, which are discussed in Chapter 12.

Problem 8.24 Solution:

This is a nontrivial problem that teaches the students how a parent process can use the `wait` or `waitpid` function to determine a child's termination status.

code/ecf/waitprob2-ans.c

```
1  #include "csapp.h"
2
3  #define NCHILDREN 2
4
5  int main()
6  {
7      int status, i;
8      pid_t pid;
9      char buf[MAXLINE];
10
11     for (i = 0; i < NCHILDREN; i++) {
12         pid = Fork();
13         if (pid == 0)  /* child */
14             /* child attempts to modify first byte of main */
15             *(char *)main = 1;
16     }
17
18     /* parent waits for all children to terminate */
19     while ((pid = wait(&status)) > 0) {
20         if (WIFEXITED(status))
21             printf("child %d terminated normally with exit status=%d\n",
22                    pid, WEXITSTATUS(status));
23         else
24             if (WIFSIGNALED(status)) {
25                 sprintf(buf, "child %d terminated by signal %d",
26                         pid, WTERMSIG(status));
27                 psignal(WTERMSIG(status), buf);
28             }
```

```
29     }
30     if (errno != ECHILD)
31         unix_error("wait error");
32
33     return 0;
34 }
```

code/ecf/waitprob2-ans.c

Problem 8.25 Solution:

This is a beautiful little problem that shows students the interaction between two different forms of exceptional control flow: signals and nonlocal jumps.

code/ecf/tfgets-ans.c

```
1 #include "csapp.h"
2
3 static sigjmp_buf env;
4
5 static void handler(int sig)
6 {
7     Alarm(0);
8     siglongjmp(env, 1);
9 }
10
11 char *tfgets(char *s, int size, FILE *stream)
12 {
13     Signal(SIGALRM, handler);
14
15     Alarm(5);
16     if (sigsetjmp(env, 1) == 0)
17         return(Fgets(s, size, stream)); /* return user input */
18     else
19         return NULL;  /* return NULL if fgets times out */
20 }
21
22 int main()
23 {
24     char buf[MAXLINE];
25
26     while (1) {
27         bzero(buf, MAXLINE);
28         if (tfgets(buf, sizeof(buf), stdin) != NULL)
29             printf("read: %s", buf);
30         else
31             printf("timed out\n");
32     }
33     exit(0);
34 }
```

code/ecf/tfgets-ans.c

Problem 8.26 Solution:

Writing a simple shell with job control is a fascinating project that ties together many of the ideas in this chapter. The distribution of the Shell Lab on the CS:APP2 Instructor Site

```
http://csapp2.cs.cmu.edu
```

provides the reference solution.

1.9 Chapter 9: Virtual Memory

Problem 9.11 Solution:

The following series of address translation problems give the students more practice with translation process. These kinds of problems make excellent exam questions because they require deep understanding, and they can be endlessly recycled in slightly different forms.

```
A. 00 0010 0111 1100

B.     VPN:            0x9
       TLBI:           0x1
       TLBT:           0x2
       TLB hit?        N
       page fault?     N
       PPN:            0x17

C.     0101 1111 1100

D.     CO:             0x0
       CI:             0xf
       CT:             0x17
       cache hit?      N
       cache byte?     -
```

Problem 9.12 Solution:

```
A. 00 0011 1010 1001

B.     VPN:            0xe
       TLBI:           0x2
       TLBT:           0x3
       TLB hit?        N
       page fault?     N
```

```
        PPN:            0x11

C.      0100 0110 1001

D.      CO:             0x1
        CI:             0xa
        CT:             0x11
        cache hit?      N
        cache byte?     -
```

Problem 9.13 Solution:

```
A. 00 0000 0100 0000

B.      VPN:            0x1
        TLBI:           0x1
        TLBT:           0x0
        TLB hit?        N
        page fault?     Y
        PPN:            -

C.      n/a

D.      n/a
```

Problem 9.14 Solution:

This problem has a kind of "gee whiz!" appeal to students when they realize that they can modify a disk file by writing to a memory location. The template is given in the solution to Problem 9.5. The only tricky part is to realize that changes to memory-mapped objects are not reflected back unless they are mapped with the MAP_SHARED option.

code/vm/mmapwrite-ans.c

```
 1 #include "csapp.h"
 2
 3 /*
 4  * mmapwrite - uses mmap to modify a disk file
 5  */
 6 void mmapwrite(int fd, int len)
 7 {
 8     char *bufp;
 9
10     bufp = Mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
11     bufp[0] = 'J';
12 }
```

```
13
14 /* mmapwrite driver */
15 int main(int argc, char **argv)
16 {
17     int fd;
18     struct stat stat;
19
20     /* check for required command line argument */
21     if (argc != 2) {
22         printf("usage: %s <filename>\n", argv[0]);
23         exit(0);
24     }
25
26     /* open the input file and get its size */
27     fd = Open(argv[1], O_RDWR, 0);
28     fstat(fd, &stat);
29     mmapwrite(fd, stat.st_size);
30     exit(0);
31 }
```

code/vm/mmapwrite-ans.c

Problem 9.15 Solution:

This is another variant of Problem 9.6.

Request	Block size (decimal bytes)	Block header (hex)
`malloc(3)`	8	`0x9`
`malloc(11)`	16	`0x11`
`malloc(20)`	24	`0x19`
`malloc(21)`	32	`0x21`

Problem 9.16 Solution:

This is a variant of Problem 9.7. The students might find it interesting that optimized boundary tags coalescing scheme, where the allocated blocks don't need a footer, has the same minimum block size (16 bytes) for either alignment requirement.

Alignment	Allocated block	Free block	Minimum block size (bytes)
Single-word	Header and footer	Header and footer	20
Single-word	Header, but no footer	Header and footer	16
Double-word	Header and footer	Header and footer	24
Double-word	Header, but no footer	Header and footer	16

Problem 9.17 Solution:

This is a really interesting problem for students to work out. At first glance, the solution appears trivial. You define a global roving pointer (`void *rover`) that points initially to the front of the list, and then perform the search using this rover:

code/vm/malloc2-ans.c

```
1 static void *find_fit(size_t asize)
2 {
3     char *oldrover;
4
5     oldrover = rover;
6
7     /* search from the rover to the end of list */
8     for ( ; GET_SIZE(HDRP(rover)) > 0; rover = NEXT_BLKP(rover))
9         if (!GET_ALLOC(HDRP(rover)) && (asize <= GET_SIZE(HDRP(rover))))
10            return rover;
11
12    /* search from start of list to old rover */
13    for (rover = heap_listp; rover < oldrover; rover = NEXT_BLKP(rover))
14        if (!GET_ALLOC(HDRP(rover)) && (asize <= GET_SIZE(HDRP(rover))))
15            return rover;
16
17    return NULL;  /* no fit found */
18 }
```

code/vm/malloc2-ans.c

However, the interaction with coalescing introduces a subtlety that is easy to overlook. Suppose that the rover is pointing at an allocated block b when the application makes a request to free b. If the previous block is free, then it will be coalesced with b, and the rover now points to garbage in the middle of a free block. Eventually, the allocator will either allocate a non-disjoint block or crash. Thus, a correct solution must anticipate this situation when it coalesces, and adjust the rover to point to new coalesced block:

code/vm/malloc2-ans.c

```
1 static void *coalesce(void *bp)
2 {
3     int prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
4     int next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
5     size_t size = GET_SIZE(HDRP(bp));
6
7     if (prev_alloc && next_alloc) {            /* Case 1 */
8         return bp;
9     }
10
11    else if (prev_alloc && !next_alloc) {      /* Case 2 */
12        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
13        PUT(HDRP(bp), PACK(size, 0));
14        PUT(FTRP(bp), PACK(size,0));
15    }
16
17    else if (!prev_alloc && next_alloc) {      /* Case 3 */
18        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
19        PUT(FTRP(bp), PACK(size, 0));
20        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
21        bp = PREV_BLKP(bp);
```

```
22      }
23
24      else {                                      /* Case 4 */
25          size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
26              GET_SIZE(FTRP(NEXT_BLKP(bp)));
27          PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
28          PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
29          bp = PREV_BLKP(bp);
30      }
31
32      /* Make sure the rover isn't pointing into the free block */
33      /* that we just coalesced */
34      if ((rover > (char *)bp) && (rover < NEXT_BLKP(bp)))
35          rover = bp;
36
37      return bp;
38  }
```

code/vm/malloc2-ans.c

Interestingly, when we benchmark the implicit allocator in Section 9.9.12 on a collection of large traces, we find that next fit improves the average throughput by more than a factor of 10, from 10K requests/sec to a respectable 139K requests/sec. However, the memory utilization of next fit (80%) is worse than first fit (99%). By contrast, the C standard library's GNU `malloc` package, which uses a complicated segregated storage scheme, runs at 119K requests/sec on the same set of traces.

Problem 9.18 Solution:

No solution yet.

Problem 9.19 Solution:

Here are the true statements. The observation about the equivalence of first fit and best fit when the list is ordered is interesting.

1. (a) In a buddy system, up to 50% of the space can be wasted due to internal fragmentation.
2. (d) Using the first-fit algorithm on a free list that is ordered according to increasing block sizes is equivalent to using the best-fit algorithm.
3. (b) Mark-and-sweep garbage collectors are called conservative if they treat everything that looks like a pointer as a pointer,

Problem 9.20 Solution:

This one of our favorite labs. See the CS:APP Instructor's Web page for a turnkey solution, including solution implementation and autograders.

1.10 Chapter 10: I/O

Problem 10.6 Solution:

On entry, descriptors 0-2 are already open. The `open` function always returns the lowest possible descriptor, so the first two calls to `open` return descriptors 3 and 4. The call to the `close function` frees up descriptor 4, so the final call to `open` returns descriptor 4, and thus the output of the program is "`fd2 = 4`".

Problem 10.7 Solution:

code/io/cpfile1-ans.c

```
1 #include "csapp.h"
2
3 int main(int argc, char **argv)
4 {
5     int n;
6     char buf[MAXBUF];
7
8     while((n = Rio_readn(STDIN_FILENO, buf, MAXBUF)) != 0)
9         Rio_writen(STDOUT_FILENO, buf, n);
10    exit(0);
11 }
```

code/io/cpfile1-ans.c

Problem 10.8 Solution:

The solution is nearly identical to Figure 10.10, calling `fstat` instead of `stat`.

code/io/fstatcheck-ans.c

```
1 #include "csapp.h"
2
3 int main (int argc, char **argv)
4 {
5     struct stat stat;
6     char *type, *readok;
7     int size;
8
9     if (argc != 2) {
10        fprintf(stderr, "usage: %s <fd>\n", argv[0]);
11        exit(0);
12    }
13    Fstat(atoi(argv[1]), &stat);
14    if (S_ISREG(stat.st_mode))     /* Determine file type */
15        type = "regular";
16    else if (S_ISDIR(stat.st_mode))
17        type = "directory";
18    else if (S_ISCHR(stat.st_mode))
```

```
19          type = "character device";
20      else
21          type = "other";
22
23      if ((stat.st_mode & S_IRUSR)) /* Check read access */
24          readok = "yes";
25      else
26          readok = "no";
27
28      size = stat.st_size; /* check size */
29
30      printf("type: %s, read: %s, size=%d\n",
31             type, readok, size);
32
33      exit(0);
34  }
```

code/io/fstatcheck-ans.c

Problem 10.9 Solution:

Before the call to `execve`, the child process opens `foo.txt` as descriptor 3, redirects `stdin` to `foo.txt`, and then (here is the kicker) closes descriptor 3:

```
if (Fork() == 0) { /* child */
    fd = Open(''foo.txt'', O_RDONLY, 0); /* fd == 3 */
    Dup2(fd, STDIN_FILENO);
    Close(fd);
    Execve(''fstatcheck'', argv, envp);
}
```

When `fstatcheck` begins running in the child, there are exactly three open files, corresponding to descriptors 0, 1, and 2, with descriptor 1 redirected to `foo.txt`.

Problem 10.10 Solution:

The purpose of this problem is to give the students additional practice with I/O redirection. The trick is that if the user asks us to copy a file, we redirect standard input to that file before running the copy loop. The redirection allows the same copy loop to be used for either case.

code/io/cpfile2-ans.c

```
1  #include "csapp.h"
2
3  int main(int argc, char **argv)
4  {
5      int n;
6      rio_t rio;
7      char buf[MAXLINE];
8
9      if ((argc != 1) && (argc != 2) ) {
```

```
10          fprintf(stderr, "usage: %s <infile>\n", argv[0]);
11          exit(1);
12      }
13
14      if (argc == 2) {
15          int fd;
16          if ((fd = Open(argv[1], O_RDONLY, 0)) < 0) {
17              fprintf(stderr, "Couldn't read %s\n", argv[1]);
18              exit(1);
19          }
20          Dup2(fd, STDIN_FILENO);
21          Close(fd);
22      }
23
24      Rio_readinitb(&rio, STDIN_FILENO);
25      while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
26          Rio_writen(STDOUT_FILENO, buf, n);
27      exit(0);
28  }
```

code/io/cpfile2-ans.c

1.11 Chapter 11: Network Programming

Problem 11.6 Solution:

There is no unique solution. The problem has several purposes. First, we want to make sure students can compile and run Tiny. Second, we want students to see what a real browser request looks like and what the information contained in it means.

Problem 11.7 Solution:

Solution outline: This sounds like it might be difficult, but it is really very simple. To a Web server, all content is just a stream of bytes. Simply add the MIME type `video/mpg` to the `get_filetype` function in Figure 11.33.

Problem 11.8 Solution:

Solution outline: Install a SIGCHLD handler in the main routine and delete the call to `wait` in `serve_dynamic`.

Problem 11.9 Solution:

Solution outline: Allocate a buffer, read the requested file into the buffer, write the buffer to the descriptor, and then free the buffer.

Problem 11.10 Solution:

No solution yet.

Problem 11.11 Solution:

Solution outline: HEAD is identical to GET, except that it does not return the response body.

Problem 11.12 Solution:

No solution yet.

Problem 11.13 Solution:

Solution outline: Install the SIG_IGN handler for SIGPIPE, and write a wrapper function `rio_writenp` that returns 0 when it encounters an EPIPE error. To be more efficient, Tiny can check the return code after each write and return to the main routine when it gets a zero.

1.12 Chapter 12: Concurrency

Problem 12.16 Solution:

This purpose of this problem is get the student's feet wet with a simple threaded program.

code/conc/hellon-ans.c

```
1 #include "csapp.h"
2
3 void *thread(void *vargp);
4
5 int main(int argc, char **argv)
6 {
7     pthread_t *tid;
8     int i, n;
9
10    if (argc != 2) {
11        fprintf(stderr, "usage: %s <nthreads>\n", argv[0]);
12        exit(0);
13    }
14    n = atoi(argv[1]);
15    tid = Malloc(n * sizeof(pthread_t));
16
17    for (i = 0; i < n; i++)
18        Pthread_create(&tid[i], NULL, thread, NULL);
19    for (i = 0; i < n; i++)
20        Pthread_join(tid[i], NULL);
21    exit(0);
22 }
23
24 /* thread routine */
25 void *thread(void *vargp)
26 {
27    printf("Hello, world!\n");
28    return NULL;
```

```
29 }
```

code/conc/hellon-ans.c

Problem 12.17 Solution:

This is the student's first introduction to the many synchronization problems that can arise in threaded programs.

A. The problem is that the main thread calls `exit` without waiting for the peer thread to terminate. The `exit` call terminates the entire process, including any threads that happen to be running. So the peer thread is being killed before it has a chance to print its output string.

B. We can fix the bug by replacing the `exit` function with `pthread_exit`, which waits for outstanding threads to terminate before it terminates the process.

Problem 12.18 Solution:

The idea here is to check whether students understand the notions of safe and unsafe trajectories, where trajectories that skirt the critical region are safe, and those that cross the critical region are unsafe.

A. $H_2, L_2, U_2, H_1, L_1, S_2, U_1, S_1, T_1, T_2$: unsafe

B. $H_2, H_1, L_1, U_1, S_1, L_2, T_1, U_2, S_2, T_2$: safe

C. $H_1, L_1, H_2, L_2, U_2, S_2, U_1, S_1, T_1, T_2$: unsafe

Problem 12.19 Solution:

The idea is to use another mutex in the writer as a gateway or holding area that allows only a single writer at a time to be waiting on the mutex that protects the critical section. So when a writer finishes its critical section and executes the V, the only other threads that can be restarted are readers.

code/conc/rw2-ans.c

```
 1 #include "csapp.h"
 2
 3 /* Global variables */
 4 int readcount;      /* Initially = 0 */
 5 sem_t mutex, w, wg; /* All initially = 1 */
 6
 7 void reader(void)
 8 {
 9     while (1) {
10         P(&mutex);
11         readcount++;
12         if (readcount == 1)
13             P(&w);
14         V(&mutex);
```

```
15
16          /* Critical section: */
17          /* Reading happens   */
18
19          P(&mutex);
20          readcount--;
21          if (readcount == 0)
22              V(&w);
23          V(&mutex);
24      }
25  }
26
27  void writer(void)
28  {
29      while (1) {
30          P(&wg);
31          P(&w);
32
33          /* Critical section: */
34          /* Writing happens   */
35
36          V(&w);
37          V(&wg);
38      }
39  }
```

code/conc/rw2-ans.c

Problem 12.20 Solution:

Here is an elegant solution due to Henri Casanova. The idea is to using a counting semaphore initialized to N. Each reader must acquire 1 resource to be able to read, thus N concurrent readers are allowd. Similarly, each writer must acquire N resources to be able to write, and therefore only writer can be executing at a time, and when a writer is executing, no other readers can be executing. A mutex in the writer ensures that only one writer at a time is busy accumulating resources.

code/conc/rw3-ans.c

```
1  #include "csapp.h"
2
3  #define N 10
4
5  /* Global variables */
6  sem_t sem;    /* Initially = N */
7  sem_t wmutex; /* Initially = 1 */
8
9  void reader(void)
10 {
11     while (1) {
12         P(&sem);
13
```

```
14          /* Critical section: */
15          /* Reading happens   */
16
17          V(&sem);
18      }
19  }
20
21  void writer(void)
22  {
23      int i;
24
25      while (1) {
26          P(&wmutex);
27          for (i=0; i<N; i++)
28              P(&sem);
29          V(&wmutex);
30
31          /* Critical section: */
32          /* Writing happens   */
33
34          for (i=0; i<N; i++)
35              V(&sem);
36      }
37  }
```

code/conc/rw3-ans.c

Problem 12.21 Solution:

See the solution in Courtois et al, Concurrent Control with Readers and Writers CACM, Oct, 1971.

Problem 12.22 Solution:

No solution provided.

Problem 12.23 Solution:

No solution provided.

Problem 12.24 Solution:

Each of the `Rio` functions is passed a pointer to a buffer, and then operates exclusively on this buffer and local stack variables. If they are invoked properly by the calling function, such that none of the buffers are shared, then they are reentrant. This is a good example of the class of implicit reentrant functions.

Problem 12.25 Solution:

The `echo_cnt` function is thread-safe because (a) It protects accesses to the shared global `byte_cnt` with a mutex, and (b) All of the functions that it calls, such as `rio_readline` and `rio_writen`, are thread-safe. However, because of the shared variable, `echo_cnt` is not reentrant.

Problem 12.26 Solution:

No solution provided.

Problem 12.27 Solution:

The problem occurs because you must close the same descriptor twice in order to avoid a memory leak. Here is the deadly race: The peer thread that closes the connection completes the first close operation, thus freeing up descriptor k, and then is swapped out. A connection request arrives while the main thread is blocked in `accept` which returns a connected descriptor of k, the smallest available descriptor. The main thread is swapped out, and the the peer thread runs again, completing its second close operation, which closes descriptor k again. When the main thread runs again, the connected descriptor it passes to the peer thread is closed!

Problem 12.28 Solution:

Interestingly, as long as you lock the mutexes in the correct order, the order in which you release the mutexes has no affect on the deadlock-freedom of the program.

Problem 12.29 Solution:

Thread 1 holds mutex pairs (a, b) and (a, c) simultaneously, but not mutex pair (b,c), while Thread 2 holds mutex pair (c,b) simultaneously, not the other two. Since the sets are disjoint, there is no deadlock potential, even though Thread 2 locks its mutexes in the wrong order. Drawing the progress graph is a nice visual way to confirm this.

Problem 12.30 Solution:

A. Thread 1 holds (a, b) and (a, c) simultaneously. Thread 2 holds (b, c) simultaneously. Thread 3 holds (a, b) simultaneously.

B. Thread 1 locks all of its mutexes in order, so it is OK. Thread 2 does not violate the lock ordering with respect to (b, c) because it is the only thread that hold this pair of locks simultaneously. Thread 3 locks (b, c) out of order, but this is OK because it doesn't hold those locks simultaneously. However, locking (a,b) out of order is a problem, because Thread 1 also needs to hold that pair simultaneously.

C. Swapping the `P(b)` and `P(a)` statements will break the deadlock.

The next three problems give the students an interesting contrast in concurrent programming with processes, select, and threads.

Problem 12.31 Solution:

A version of `tfgets` based on processes:

code/conc/tfgets-proc-ans.c

```
1 #include "csapp.h"
2 #define TIMEOUT 5
3
4 static sigjmp_buf env; /* buffer for non-local jump */
```

```
 5 static char *str;      /* global to keep gcc -Wall happy */
 6
 7 /* SIGCHLD signal handler */
 8 static void handler(int sig)
 9 {
10     Wait(NULL);
11     siglongjmp(env, 1);
12 }
13
14 char *tfgets(char *s, int size, FILE *stream)
15 {
16     pid_t pid;
17
18     str = NULL;
19
20     Signal(SIGCHLD, handler);
21
22     if ((pid = Fork()) ==  0) { /* child */
23         Sleep(TIMEOUT);
24         exit(0);
25     }
26     else {  /* parent */
27         if (sigsetjmp(env, 1) == 0) {
28             str = fgets(s, size, stream);
29             Kill(pid, SIGKILL);
30             pause();
31         }
32         return str;
33     }
34 }
```

code/conc/tfgets-proc-ans.c

Problem 12.32 Solution:

A version of `tfgets` based on I/O multiplexing:

code/conc/tfgets-select-ans.c

```
 1 #include "csapp.h"
 2
 3 #define TIMEOUT 5
 4
 5 char *tfgets(char *s, int size, FILE *stream)
 6 {
 7     struct timeval tv;
 8     fd_set rfds;
 9     int retval;
10
11     FD_ZERO(&rfds);
12     FD_SET(0, &rfds);
```

```
13
14      /* Wait for 5 seconds for stdin to be ready */
15      tv.tv_sec = 5;
16      tv.tv_usec = 0;
17      retval = select(1, &rfds, NULL, NULL, &tv);
18      if (retval)
19          return fgets(s, size, stream);
20      else
21          return NULL;
22  }
```

code/conc/tfgets-select-ans.c

Problem 12.33 Solution:

A version of `tfgets` based on threads:

code/conc/tfgets-thread-ans.c

```
1  #include "csapp.h"
2  #define TIMEOUT 5
3
4  void *fgets_thread(void *vargp);
5  void *sleep_thread(void *vargp);
6
7  char *returnval;  /* fgets output string */
8  typedef struct {  /* fgets input arguments */
9      char *s;
10     int size;
11     FILE *stream;
12 } args_t;
13
14 char *tfgets(char *str, int size, FILE *stream)
15 {
16     pthread_t fgets_tid, sleep_tid;
17     args_t args;
18
19     args.s = str;
20     args.size = size;
21     args.stream = stdin;
22     returnval = NULL;
23     Pthread_create(&fgets_tid, NULL, fgets_thread, &args);
24     Pthread_create(&sleep_tid, NULL, sleep_thread, &fgets_tid);
25     Pthread_join(fgets_tid, NULL);
26     return returnval;
27 }
28
29 void *fgets_thread(void *vargp)
30 {
31     args_t *argp = (args_t *)vargp;
32     returnval = fgets(argp->s, argp->size, stdin);
```

```
33      return NULL;
34  }
35
36  void *sleep_thread(void *vargp)
37  {
38      pthread_t fgets_tid = *(pthread_t *)vargp;
39      Pthread_detach(Pthread_self());
40      Sleep(TIMEOUT);
41      pthread_cancel(fgets_tid);
42      return NULL;
43  }
```

code/conc/tfgets-thread-ans.c

Problem 12.34 Solution:

No solution provided.

Problem 12.35 Solution:

No solution provided.

Problem 12.36 Solution:

No solution provided.

Problem 12.37 Solution:

No solution provided.

Problem 12.38 Solution:

No solution provided.

Problem 12.39 Solution:

No solution provided.