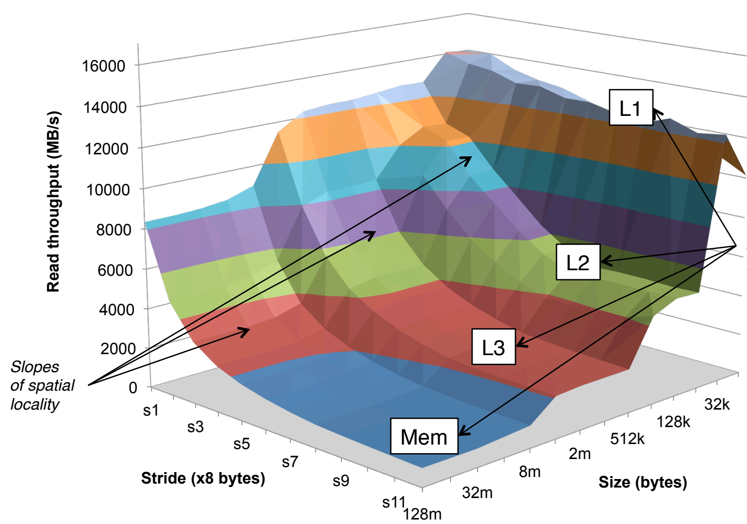
Computer Systems: A Programmer's Perspective, Third Edition

Instructor's Solution Manual [1]



Randal E. Bryant
David R. O'Hallaron

December 12, 2016

Solutions to Homework Problems

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.

Chapter 1: A Tour of Computer Systems

There are no homework problems in this chapter.

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

```
void show_short(short x) {
    show_bytes((byte_pointer) &x, sizeof(short));
}

void show_long(long x) {
    show_bytes((byte_pointer) &x, sizeof(long));
}

void show_double(double x) {
    show_bytes((byte_pointer) &x, sizeof(double));
}
```

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

```
int is_little_endian(void) {
    /* MSB = 0, LSB = 1 */
    int x = 1;

    /* Return MSB when big-endian, LSB when little-endian */
    return (int) (* (char *) &x);
}
```

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

```
unsigned replace_byte (unsigned x, int i, unsigned char b) {
    int itimes8 = i << 3;
    unsigned mask = 0xFF << itimes8;

    return (x & ~mask) | (b << itimes8);
}
```

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

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

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

```
unsigned srl(unsigned x, int k)
{
    /* Perform shift arithmetically */
    unsigned xsra = (int) x >> k;
    /* Make mask of low order w-k bits */
    unsigned mask = k ? ((1 << (8*sizeof(int)-k)) - 1) : ~0;

    return xsra & mask;
}
```

code/data/rshift-ans.c

code/data/rshift-ans.c

```
int sra(int x, int k)
{
    /* Perform shift logically */
    int xsrl = (unsigned) x >> k;
    /* Make mask of high order k bits */
    unsigned mask = k ? ~((1 << (8*sizeof(int)-k)) - 1) : 0;
```

```

    return (x < 0) ? mask | xsrl : xsrl;
}
```

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

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

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

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

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 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

```
/*
 * Mask with least signficant n bits set to 1
 * Examples: n == 6 --> 0x3F, n == 17 --> 0x1FFFF
 * Assume 1 <= n <= w
 */
int lower_one_mask(int n) {
    /*
     * 2^n-1 has bit pattern 0...01..1 (n 1's)
     * But, we must avoid a shift by 32
```

```
     */
    return (2<<(n-1)) - 1;
}
```

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 $\mathtt{n} = w$. Instead of writing `1<<n`, we write `2<<(n-1)`. This code will not work for $\mathtt{n} = 0$, but that's not a very useful case, anyhow.

Problem 2.69 Solution:

code/data/bits.c

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

code/data/bits.c

For the most part, this problem requires simple shifting and masking. We must treat the case of $\mathtt{n} = 0$ 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

```
/*
 * Return 1 when x can be represented as an n-bit, 2's complement number;
 * 0 otherwise
 * Assume 1 <= n <= w
 */
int fits_bits(int x, int n) {
    /*
     * Use left shift then right shift
     * to sign extend from n bits to full int
     */
```

```
    int count = (sizeof(int)<<3)-n;
    int leftright = (x << count) >> count;
    /* See if still have same value */
    return x == leftright;
}
```

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

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

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:

```
/* Must do signed comparison in event maxbytes < 0 */
if (maxbytes >= sizeof(val))
```

Problem 2.73 Solution:

Here is the solution.

code/data/bits.c

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

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

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

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

```
unsigned unsigned_high_prod(unsigned x, unsigned y) {
    unsigned p = (unsigned) signed_high_prod((int) x, (int) y);

    if ((int) x < 0) /* x_{w-1} = 1 */
        p += y;
    if ((int) y < 0) /* y_{w-1} = 1 */
        p += x;
    return p;
}
```

code/data/uhp-ans.c

Problem 2.76 Solution:

The solution to this problem can easily be adapted from the solution to Problem 2.30. The main point is to make sure students see that connection.

code/data/calloc.c

```
void *calloc(size_t nmemb, size_t size) {
    size_t asize =  nmemb * size;
    /* Check for overflow */
    if (nmemb == 0 || size == 0 || asize / nmemb != size)
        /* Error */
        return NULL;
    void *result = malloc(asize);
    if (result != NULL) {
        memset(result, 0, asize);
        return result;
    }
    return NULL;
}
```

code/data/calloc.c

Problem 2.77 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 2.78 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

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

code/data/bits.c

Problem 2.79 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

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

code/data/bits.c

Problem 2.80 Solution:

The requirement that the function must not overflow makes this problem more challenging than Problem 2.79. 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

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

code/data/bits.c

Problem 2.81 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 2.82 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 2.83 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 $y.y\,y\,y\,y\,y\,y\cdots$, 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 2.84 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

```
int float_le(float x, float y) {
    unsigned ux = f2u(x);
    unsigned uy = f2u(y);
    unsigned sx = ux >> 31;
    unsigned sy = uy >> 31;

    return
        (ux<<1 == 0 && uy<<1 == 0) ||  /* Both are zero */
        (sx && !sy) ||                 /* x < 0, y >=  0 */
        (!sx && !sy && ux <= uy) ||    /* x >= 0, y >= 0 */
        (sx && sy && ux >= uy);        /* x <  0, y <  0 */
}
```

code/data/floatcomp-ans.c

Problem 2.85 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 $100 \cdots 01$ and the fraction bits will be $1100 \cdots 0$.

B. The largest odd integer that can be represented exactly will have a binary representation consisting of $n + 1$ ones. It will have $E = n$, $M = 1.11 \cdots 1_2 = 2 - 2^{-n}$, $f = 0.11 \cdots 1_2 = 1 - 2^{-n}$, 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 $11 \cdots 11$.

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 $11 \cdots 101$. The bit representation of the fraction will be $00 \cdots 00$.

Problem 2.86 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 2.87 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. This particular format example is more compelling in that it is actually an IEEE standard.

Description	Hex	M	E	V	D
-0	`8000`	0	-14	-0	-0.0
Smallest value > 2	`4001`	$\frac{1025}{1024}$	1	1025×2^{-9}	2.00195
512	`6000`	1	9	512	512.0
Largest denormalized	`03FF`	$\frac{1023}{1024}$	-14	1023×2^{-24}	0.000061
$-\infty$	`FC00`	—	—	$-\infty$	$-\infty$
Number with hex representation `3BB0`	`3BB0`	$\frac{1968}{1024}$	-1	123×2^{-7}	0.961

Problem 2.88 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 2.89 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 2.90 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

```
/* Compute 2**x */
float fpwr2(int x) {

    unsigned exp, frac;
    unsigned u;

    if (x < -149) {
        /* Too small.  Return 0.0 */
        exp  = 0;
        frac = 0;
    } else if (x < -126) {
        /* Denormalized result */
        exp  = 0;
        frac = 1 << (x + 149);
    } else if (x < 128) {
        /* Normalized result. */
        exp  = x + 127;
        frac = 0;
    } else {
        /* Too big.  Return +oo */
        exp  = 255;
        frac = 0;
    }
    u = exp << 23 | frac;
    return u2f(u);
}
```

code/data/fpwr2-ans.c

Problem 2.91 Solution:

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

A. $\pi \approx \quad 11.0010010000111111011011_2$.

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

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

Problem 2.92 Solution:

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

code/data/float-functions.c

```
/* Compute -f.  If f is NaN, then return f. */
float_bits float_negate(float_bits f) {
    unsigned exp =  f>>23 & 0xFF;
    unsigned frac = f     & 0x7FFFFF;
    unsigned mask = 1 << 31;
    unsigned neg = f ^ mask;
    if (exp == 0xFF && frac != 0)
        /* NaN */
        return f;
    return neg;
}
```

code/data/float-functions.c

Problem 2.93 Solution:

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

code/data/float-functions.c

```
/* Compute |f|.  If f is NaN, then return f. */
float_bits float_absval(float_bits f) {
    unsigned exp =  f>>23 & 0xFF;
    unsigned frac = f     & 0x7FFFFF;
    unsigned mask = 1 << 31;
    unsigned absval = f & ~mask;
    if (exp == 0xFF && frac != 0)
        /* NaN */
        return f;
    return absval;
}
```

code/data/float-functions.c

Problem 2.94 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

```
/* Compute 2*f.  If f is NaN, then return f. */
float_bits float_twice(float_bits f) {
    unsigned sign = f>>31;
    unsigned exp =  f>>23 & 0xFF;
    unsigned frac = f     & 0x7FFFFF;
    if (exp == 0) {
        /* Denormalized.  Must double fraction */
        frac = 2*frac;
        if (frac > 0x7FFFFF) {
```

```
            /* Result normalized */
            frac = frac & 0x7FFFFF; /* Chop off leading bit */
            exp = 1;
        }
    } else if (exp < 0xFF) {
        /* Normalized.  Increase exponent */
        exp++;
        if (exp == 0xFF) {
            /* Infinity */
            frac = 0;
        }
    } else if (frac != 0) {
        /* NaN */
        return f;
    }
    /* Infinity does not require any changes */
    return (sign << 31) | (exp << 23) | frac;
}
```

code/data/float-functions.c

Problem 2.95 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

```
/* Compute 0.5*f.  If f is NaN, then return f. */
float_bits float_half(float_bits f) {
    unsigned sign = f>>31;
    unsigned exp =  f>>23 & 0xFF;
    unsigned frac = f     & 0x7FFFFF;
    /* Only roundup case will be when rounding to even */
    unsigned roundup = (frac & 0x3) == 3;
    if (exp == 0) {
        /* Denormalized.  Must halve fraction */
        frac = (frac >> 1) + roundup;
    } else if (exp < 0xFF) {
        /* Normalized.  Decrease exponent */
        exp--;
        if (exp == 0) {
            /* Denormalize, add back leading one */
            frac = (frac >> 1) + roundup + 0x400000;
        }
    } else if (frac != 0) {
        /* NaN */
        return f;
    }
    /* Infinity does not require any changes */
    return (sign << 31) | (exp << 23) | frac;
```

```
}
```

code/data/float-functions.c

Problem 2.96 Solution:

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

code/data/float-functions.c

```
/*
 * 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 2.97 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;
            exp--;
        }
        /* Now have Bit 31 = MSB (becomes implied leading one)
           Bits 8-30 are tentative fraction,
           Bits 0-7 require rounding.
        */
        residue = ai & 0xFF;
        frac = (ai >> 8) & 0x7FFFFF; /* 23 bits */
        if (residue > 0x80 || (residue == 0x80 && (frac & 0x1))) {
            /* Round up */
            frac ++;
            /* Might need to renormalize */
            if (frac > 0x7FFFFF) {
                frac = (frac & 0x7FFFFF) >> 1;
                exp++;
            }
        }
    }
    return (sign << 31) | (exp << 23) | frac;
}
```

code/data/float-functions.c

Chapter 3: Machine Level Representation of C Programs

Problem 3.58 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.

```
long decode2(long x, long y, long z)
{
    long t1 = y - z;
    long t2 = x * t1;
    long t3 = (t1 << 63) >> 63;
```

```
    long t4 = t3 ^ t2;

    return t4;
}
```

Problem 3.59 Solution:

A. Following the hint, we can compute $x \cdot y = 2^{128} x_h \cdot y_h + 2^{64} \cdot (x_l \cdot y_h + x_h \cdot y_l) + x_l \cdot y_l$. Since we are only interested in the lower 128 bits, we can ignore the term involving $x_h \cdot y_h$, and we can let s be the low-order 64 bits of $x_l \cdot y_h$, r be the low-order 64 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 $p_l = t_l$, and $p_h = r + s + t_h$.

Here is the annotated assembly code:

```
void store_prod(int128_t *dest, int64_t x, int64_t y)
dest in %rdi, x in %rsi, y in %rdx
store_prod:
  movq    %rdx, %rax         Copy y = yl
  cqto                       %rdx:%rax = yh:yl
  movq    %rsi, %rcx         Copy x = xl
  sarq    $63, %rcx          x < 0 ? -1 : 0 = xh
  imulq   %rax, %rcx         s = xl * yh
  imulq   %rsi, %rdx         r = xh * yl
  addq    %rdx, %rcx         r+s
  mulq    %rsi               %rdx:%rax = xl*yl = t
  addq    %rcx, %rdx         r+s+th
  movq    %rax, (%rdi)       Store pl
  movq    %rdx, 8(%rdi)      Store ph
  ret
```

Problem 3.60 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 `%rax`, since this value gets returned as the final value. Parameter `x` is passed in `%rdi`. Parameter `n` is passed in `%esi` and then copied into `%ecx`. Register `%edx` is initialized to 1. We can infer that `mask` must be `%rdx`.

B. They are initialized to 0 and 1, respectively.

C. The condition for continuing the loop is that `mask` is nonzero.

D. The `salq` instruction updates `mask` to be `mask << n`.

E. Variable `result` is updated to be `result | (x&mask)`.

F. Here is the original code:

```
long loop(long x, int n)
{
    long result = 0;
    long mask;
    for (mask = 0x1; mask != 0; mask = mask << n) {
        result |= (x & mask);
    }
    return result;
}
```

Problem 3.61 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.

```
long cread_fix(long *xp) {
    long zero = 0;
    if (!xp) xp = &zero;
    return *xp;
}
```

Problem 3.62 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.

The original C code for the function is as follows:

```
/* Enumerated type creates set of constants numbered 0 and upward */
typedef enum {MODE_A, MODE_B, MODE_C, MODE_D, MODE_E} mode_t;

long switch3(long *p1, long *p2, mode_t action)
{
    long result = 0;
    switch(action) {
    case MODE_A:
        result = *p2;
        *p2 = *p1;
        break;
    case MODE_B:
```

```
        *p1 += *p2;
        result = *p1;
        break;
    case MODE_C:
        *p1 = 59;
        result = *p2;
        break;
    case MODE_D:
        *p1 = *p2;
        /* Fall Through */
    case MODE_E:
        result = 27;
        break;
    default:
        result = 12;
    }
    return result;
}
```

Problem 3.63 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.

```
long switch_prob(long x, long n) {
    long result = x;
    switch(n) {
    case 60:
    case 62:
        result <<= 3;
        break;
    case 63:
        result >>= 3;
        break;
    case 64:
        result *= 15;
        /* Fall through */
    case 65:
        result *= result;
        /* Fall through */
    default:
        result += 75;
    }
    return result;
}
```

Problem 3.64 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 $x_{\mathrm{A}} + 8(T(S \cdot i + j) + k)$.

B. Consider the following annotated version of the assembly code:

```
   long store_ele(long i, long j, long k, long *dest)
   i in %rdi, j in %rsi, k in %rdx, dest in %rcx
1  store_ele:
2    leaq    (%rsi,%rsi,2), %rax     3 j
3    leaq    (%rsi,%rax,4), %rax     13 j
4    movq    %rdi, %rsi              Copy i
5    salq    $6, %rsi                64 i
6    addq    %rsi, %rdi              65 i
7    addq    %rax, %rdi              13 j + 65 i = 13(5 i + j)
8    addq    %rdi, %rdx              13(5 i + j) + k
9    movq    A(,%rdx,8), %rax        M[xA + 8 (13(5 i + j) + k)]
10   movq    %rax, (%rcx)
11   movl    $3640, %eax
12   ret
```

We can see that memory reference on line 9 indicate that $T = 13$ and $S = 5$. We can see on line 11 that the total array size is 3640 bytes. From this, we get $R = 3640/(8 \cdot 13 \cdot 5) = 7$.

Problem 3.65 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. We can see in the code that registers `%rdx` and `%rax` are being used as pointers. The only question is which one represents which matrix element. We can see in line 6 that `%rdx` gets incremented by 8. This must be a pointer to `A[i][j]`.

B. That leaves `%rax` as a pointer to `A[j][i]`.

C. Since `%rax` is incremented by 120 on each iteration (line 7), we must have $M = 120/8 = 15$.

Problem 3.66 Solution:

This problem requires some simple reverse engineering. The best approach is to annotate the assembly code:

```
long sum_col(long n, long A[NR(n)][NC(n)], long j)
n in %rdi, A in %rsi, j in %rdx
sum_col:
  leaq    1(,%rdi,4), %r8          4n+1
  leaq    (%rdi,%rdi,2), %rax      3n
  movq    %rax, %rdi               imax = Copy 3n
```

```
  testq   %rax, %rax              Test 3n
  jle     .L4                     If <= 0, goto done0
  salq    $3, %r8                 rowsize = 8(4n+1)
  leaq    (%rsi,%rdx,8), %rcx     Aptr = xA + 8j
  movl    $0, %eax                result = 0
  movl    $0, %edx                i = 0
.L3:                            loop:
  addq    (%rcx), %rax            result += *Aptr
  addq    $1, %rdx                i++
  addq    %r8, %rcx               Aptr += rowsize
  cmpq    %rdi, %rdx              Compare i:imax
  jne     .L3                     If !=, goto loop
  rep; ret                        Return
.L4:                            done0
  movl    $0, %eax                result = 0
  ret                             Return
```

From this, we can see that the successive elements in column j of A are spaced $8(4n+1)$ bytes apart, and that the stopping value for j is $3n$, corresponding to the following declarations:

```
#define NR(n) (3*(n))
#define NC(n) (4*(n)+1)
```

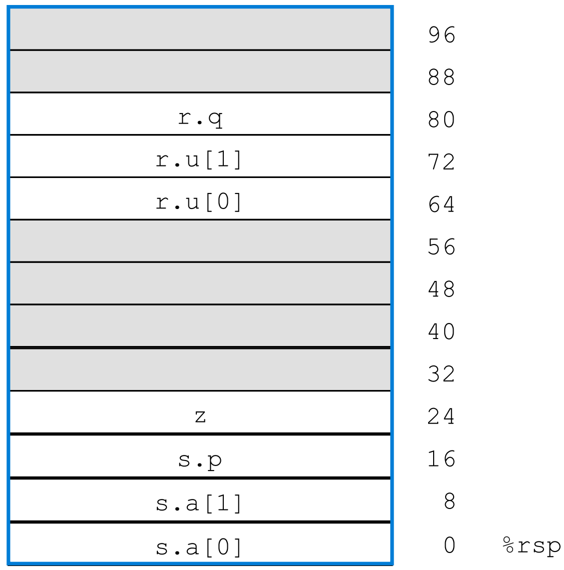
Problem 3.67 Solution:

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

The best way to solve this problem is to examine the code for `eval`, seeing how it sets up the call of `process`. Here is annotated code:

```
  long eval(long x, long y, long z)
  x in %rdi, y in %rsi, z in %rdx
eval:
  subq    $104, %rsp        Allocate 104 bytes on stack
  movq    %rdx, 24(%rsp)    Store z at %rsp+24
  leaq    24(%rsp), %rax    Compute &z
    Build s starting at %rsp
  movq    %rdi, (%rsp)      s.a[0] = x
  movq    %rsi, 8(%rsp)     s.a[1] = y
  movq    %rax, 16(%rsp)    s.p = &z
    Allocate space for r starting at %rsp+64
  leaq    64(%rsp), %rdi    Compute &r
  call    process           Call process
  movq    72(%rsp), %rax    Retrieve r.u[1]
  addq    64(%rsp), %rax    Add r.u[0]
  addq    80(%rsp), %rax    Add r.q
  addq    $104, %rsp        Deallocate stack space
  ret
```

From this, we can draw a diagram of the stack frame for `eval`:



With that information, the code for `process` is much easier to understand:

```
    strB process(strA s)
    Contents of s passed on stack at offsets 8, 16 and 24 from %rsp
    Pointer to storage for r passed in %rdi
    Return pointer to strB
process:
  movq    %rdi, %rax          Set return value
  movq    24(%rsp), %rdx      Retrieve s.p
  movq    (%rdx), %rdx        Get *s.p
  movq    16(%rsp), %rcx      Get s.a[1]
  movq    %rcx, (%rdi)        Copy to r.u[0]
  movq    8(%rsp), %rcx       Get s.a[0]
  movq    %rcx, 8(%rdi)       Copy to r.u[1]
  movq    %rdx, 16(%rdi)      Set r.q to *s.p
  ret
```

A. We can see that `eval` passes `s` to `process`, using 24 bytes at the top of the stack. It also stores argument `z` (in register `%rdx`) on the stack at offset 24.

B. Function `eval` allocates 24 bytes on the stack at offset 64 for the result structure `r`. It passes a pointer to this region as a hidden argument to `process` in register `%rdi`.

C. Function `process` accesses the fields of argument structure `s` on the stack. Since the `callq` instruction pushed an 8-byte return address, the fields of `s` start at offset 8.

D. Function `process` sets the fields of the result structure via the pointer passed as a hidden argument.

E. See above diagram.

F. Structure arguments are passed on the stack. When calling a function that will return a structure, the caller allocates space on its stack and passes a pointer to this region as a hidden argument to the function.

Problem 3.68 Solution:

This problem is like a puzzle, where a number of clues must be assembled to get the answer. It tests students' 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 8 and 12, respectively. We can see that field `y` of structure `str1` is at offset 184. We can write the following equations for these offsets:

$$\begin{aligned} B + e_1 &= 8 \\ B + e_1 + 4 + 2A + e_2 &= 32 \\ 4A \cdot B + e_3 &= 184 \end{aligned}$$

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 $e_1 \in \{0, 1, 2, 3\}$, $e_2 \in \{0, 2, 4, 6\}$, and $e_3 \in \{0, 4\}$.

From the first equation, we can see that $B \in \{5, 6, 7, 8\}$. We can substitute the first equation into the second to get

$$2A + e_2 = 20$$

which implies $A \in \{7, 9, 8, 10\}$. Finally, the third equation implies $A \cdot B \in \{45, 46\}$.

The only combination satisfying all three constraints is $A = 9$ and $B = 5$.

Problem 3.69 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.

Let us reexamine the disassembled code, with some annotations:

```
    void test(long i, b_struct *bp)
    i in %rdi, bp in %rsi
0000000000000000 <test>:
   0:   8b 8e 20 01 00 00       mov    0x120(%rsi),%ecx          Get bp->last
```

```
3    6:   03 0e                  add    (%rsi),%ecx              Add bp->first
4    8:   48 8d 04 bf            lea    (%rdi,%rdi,4),%rax
5    c:   48 8d 04 c6            lea    (%rsi,%rax,8),%rax
6   10:   48 8b 50 08            mov    0x8(%rax),%rdx           Get ap->idx
7   14:   48 63 c9               movslq %ecx,%rcx                Sign extend n
8   17:   48 89 4c d0 10         mov    %rcx,0x10(%rax,%rdx,8)   Store n
9   1c:   c3                     retq
```

Going deeper, observe that the `mov` instruction on line 6 performs an 8-byte read to dereference `ap->idx`. That implies that element `idx` in `a_struct` is a `long`, and therefore this structure has an 8-byte alignment requirement.

Let us say that `a_struct` is A bytes, and `CNT` is C. The code to retrieve `bp->last` references offset `0x120`, or 288_{10}. We can therefore see that the layout of `b_struct` must consist of 4 bytes for element `first`, 4 bytes of padding, 280 bytes for array of `a`, 4 bytes for `last`, and 4 bytes of additional padding. We can therefore conclude that $C \cdot A = 280$.

Let us say that pointer `bp` has value p. Then in retrieving `ap->idx` (line 6, the program references address $p + 8 + 40i$, implying that $A = 40$, and therefore $C = 7$.

We can therefore fill out the missing parts of the code with:

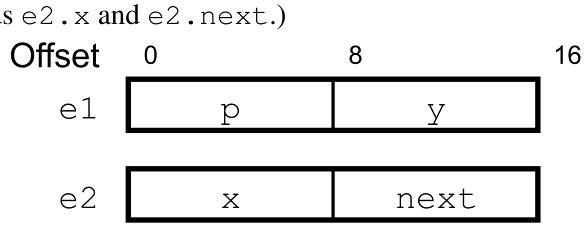
```
#define CNT 7

typedef struct {
    long idx;
    long x[4];
} a_struct;
```

Problem 3.70 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 diagram that follows. As the diagram 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 16 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 `up->e1.y` or `up->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.

```
   void proc (union ele *up)
   up in %rdi
1  proc:
2    movq    8(%rdi), %rax      Either 1) up->e1.y or 2) up->e2.next
3    movq    (%rax), %rdx       2) Either 3) up->e2.next->e1.p or 4) up->e2.next->e2.x
4    movq    (%rdx), %rdx       3) *(up->e2.next->e1.p)
5    subq    8(%rax), %rdx      Subtract up->e2.next->e1.y
6    movq    %rdx, (%rdi)       Store in up->e2.x
7    ret
```

From this, we can generate C code as follows:

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

Problem 3.71 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.

```
/* Read input line and write it back */
/* Code will work for any buffer size.  Bigger is more time-efficient */
#define BUFSIZE 64
void good_echo()
{
    char buf[BUFSIZE];
    int i;
    while (1) {
        if (!fgets(buf, BUFSIZE, stdin))
            return;  /* End of file or error */
        /* Print characters in buffer */
        for (i = 0; buf[i] && buf[i] != '\n'; i++)
            if (putchar(buf[i]) == EOF)
                return; /* Error */
        if (buf[i] == '\n') {
            /* Reached terminating newline */
            putchar('\n');
            return;
        }
    }
}
```

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

Problem 3.72 Solution:

This problem is intended to reinforce the lessons of Problem 3.49. Having a clear understanding of a solution to that problem makes this fairly straightforward.

A. The value of s_2 is computed by rounding $8n + 30$ down to the nearest multiple of 16, and subtracting this value from s_1. The amount of space allocated will be $8n + 24$ when n is odd and $8n + 16$ when n is even.

B. The value of p is computed by rounding s_2 up to the nearest multiple of 16.

C. The strategy in minimizing e_1 is to have s_1 be a value of the form $16a + 1$, so that it will be rounded up, and then choose whether n is odd or even according to which one has the least extra space. The strategy for maximizing e_1 is to have s_1 be a value of the form $16a$ so that we will have $e_2 = 0$, and to choose whether n is odd or even to have the most extra space. For this problem, the choices of odd vs. even are reversed from those of Problem 3.49:

n	s_1	s_2	p	e_1	e_2
6	2065	2001	2016	1	15
5	2064	2000	2000	24	0

D. We can see that s_2 is computed in a way that preserves whatever offset s_1 has with the nearest multiple of 16. We can also see that p will be aligned on a multiple of 16. As discussed in Section 3.9.3, this conforms to the requirement that any memory allocation function return an address that satisfies a 16-byte alignment.

Problem 3.73 Solution:

This problem gives students a chance to try their hand at writing functions directly in assembly code. As the code below shows, the only requirement in writing a function in assembly code, beyond the code for the function itself, is to declare the function to be global, using the assembly-code directive "`.globl`." GCC can assemble and link this code with C code.

```
.globl find_range_j
 int float_range_j(float x)
 x in %xmm0
find_range_j:
  vxorps  %xmm1, %xmm1, %xmm1       Set %xmm1 = 0
  vucomiss        %xmm1, %xmm0      Compare x:0
  jp nan
  jb neg
  ja pos
  movl $1,%eax                      Zero
  ret
neg:
  movl $0,%eax                      Negative
```

```
  ret
pos:                                    Positive
  movl $2,%eax
  ret
nan:
  movl $3,%eax                          NaN
  ret
```

Problem 3.74 Solution:

This code is a bit trickier. The conditional move instructions must be ordered correctly.

```
.globl find_range_c
 int float_range_c(float x)
 x in %xmm0
find_range_c:
  vxorps  %xmm1, %xmm1, %xmm1          Set %xmm1 = 0
  vucomiss        %xmm1, %xmm0         Compare x:0
  movl $1,%eax                         Default = ZERO
  movl $0,%edx
  cmovb %edx, %eax                     Negative
  movl $2,%edx
  cmova %edx, %eax                     Positive
  movl $3,%edx
  cmovp %edx, %eax                     NaN
  ret
```

Problem 3.75 Solution:

This problem tests students to reason and generalize from a small collection of examples.

A. Each complex number is passed as two arguments, giving the real and imaginary parts. In these cases, they are passed in successive XMM registers.

B. A complex value is returned with the real part in `%xmm0` and the imaginary part in `%xmm1`.

Chapter 4: Processor Architecture

Problem 4.45 Solution:

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

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

```
subq $8,%rsp          Decrement stack pointer
movq %rsp,(%rsp)      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:

```
movq  REG,-8(%rsp)     Store REG at new top of stack
subq $8,%rsp          Decrement stack pointer
```

Problem 4.46 Solution:

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

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

```
movq (%rsp),%rsp     Read %rsp from stack
addq $8,%rsp         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 4.45, we should reorder the two instructions:

```
addq $8,%rsp           Increment stack pointer
movq -8(%rsp), REG     Read REG from previous stack top
```

Problem 4.47 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:

code/arch/bubble.c

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

```
        }
        p_end--;
    }
}
```

code/arch/bubble.c

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

code/arch/bubble.ys

```
# Execution begins at address 0
        .pos 0                  #
init:
        irmovq stack, %rsp      # Set up stack pointer
        jmp main                # Execute main program

# Array of 6 elements
        .align 8
array:
        .quad 0xdddd            # These values should get sorted
        .quad 0xeeee
        .quad 0xbbbb
        .quad 0xaaaa
        .quad 0xffff
        .quad 0xcccc
        .quad 0x0101            # This value should not change

main:
        irmovq array,%rdi
        irmovq $6,%rsi
        call bubble             # bubble(array, 6)
        halt

# void bubble(long int *data, long int count)
# data in %rdi, count in %rsi
bubble:
        # During execution, have p_end in %rsi, p in %rax, 8 in %r8
        irmovq $8,%r8           # Constant 8
        addq %rsi,%rsi          # 2*count
        addq %rsi,%rsi          # 4*count
        addq %rsi,%rsi          # 8*count
        addq %rdi,%rsi          # data + count
        subq %r8,%rsi           # p_end = data + count - 1
        jmp test_outer          # Goto outer loop test
outer:
        rrmovq %rdi,%rax        # p = data
        jmp test_inner          # Goto inner loop test
inner:
        mrmovq (%rax), %rdx     # r = *p
        mrmovq 8(%rax), %rcx    # s = *(p+1)
```

```
        # Conditional swap code.  r in %rdx, s in %rcx
        rrmovq %rdx,%r9
        subq   %rcx,%r9         # r-s
        jle skip                # if r-s <= 0, don't swap
        rmmovq %rcx,(%rax)      # Swap
        rmmovq %rdx,8(%rax)
skip:
        # End of conditional swap code
        addq %r8,%rax           # p++
test_inner:
        rrmovq %rsi,%rdx        # p_end
        subq %rax,%rdx          # p_end - p
        jg inner                # if p_end-p > 0, goto inner
        subq %r8,%rsi           # p_end--
test_outer:
        rrmovq %rsi,%rdx        # p_end
        subq %rdi,%rdx          # p_end - data
        jg outer                # if p_end-data > 0, goto outer
        ret                     # Return

        .pos 0x200
stack:  # The stack goes here and grows to lower addresses
```

code/arch/bubble.ys

Problem 4.48 Solution:

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

code/arch/bubble.ys

```
        # Conditional swap code.  r in %rdx, s in %rcx
        rrmovq %rdx,%r9
        subq   %rcx,%r9         # r-s
        jle skip                # if r-s <= 0, don't swap
        rmmovq %rcx,(%rax)      # Swap
        rmmovq %rdx,8(%rax)
skip:
        # End of conditional swap code
```

code/arch/bubble.ys

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. This is expressed by the following code for the conditional swap:

code/arch/bubble-cmov.ys

```
        # Conditional swap code.  r in %rdx, s in %rcx
        rrmovq %rdx,%r9
        subq   %rcx,%r9         # r-s
```

```
        # Following transfers occur when r-s > 0 ==> s < r
        cmovg %rdx,%r9          # If r-s > 0, t = r
        cmovg %rcx,%rdx         # If r-s > 0, r = s
        cmovg %r9,%rcx          # if r-s > 0, s = t
        # Copy back (possibly swapped) values
        rmmovq %rdx,(%rax)      # *p = r
        rmmovq %rcx,8(%rax)     # *(p+1) = s
        # End of conditional swap code
```

code/arch/bubble-cmov.ys

Problem 4.49 Solution:

We can conditionally swap the two values r and s by computing value $d = r-s$, and then using a conditional move instruction to set register `%r9` to d when the swap condition holds and to 0 when it does not. Adding this value to r gives $r + (r - s) = s$ when the condition holds, and r when it does not. Subtracting this value from s gives $s - (r - s) = r$ when the condition holds, and s when it does not. This is expressed by the following code for the conditional swap:

code/arch/bubble-diff.ys

```
        # Conditional swap code.  r in %rdx, s in %rcx
        xorq %r10,%r10          # 0
        rrmovq %rdx,%r9         # r
        subq   %rcx,%r9         # d = r-s
        cmovle %r10,%r9         # r<s ? r-s : 0
        addq %r9,%rcx           # if r<s, s = s + d = r
        subq %r9,%rdx           # if r<s, r = r - d = s
        # Copy back (possibly swapped) values
        rmmovq %rdx,(%rax)      # *p = r
        rmmovq %rcx,8(%rax)     # *(p+1) = s
        # End of conditional swap code
```

code/arch/bubble-diff.ys

A similar result can be achieved by computing $d = r \text{ \^{} } s$.

Problem 4.50 Solution:

This problem requires a bit of ingenuity to overcome the limitations of the Y86-64 instruction set. It helps reinforce some of the basic ideas covered in Chapter 3.

code/arch/switchv.ys

```
# Demonstration of switch statement implementation in Y86-64

.pos 0
init:   irmovq stack, %rsp      # Set up stack pointer
        call main
        halt

# Array in which to store results
```

```
        .align 8
        .quad -1         # Should not change
vals:
        .quad 0          # idx = -1.  Result = 0xddd
        .quad 0          # idx =  0.  Result = 0xaaa
        .quad 0          # idx =  1.  Result = 0xddd
        .quad 0          # idx =  2.  Result = 0xbbb
        .quad 0          # idx =  3.  Result = 0xccc
        .quad 0          # idx =  4.  Result = 0xddd
        .quad 0          # idx =  5.  Result = 0xbbb
        .quad 0          # idx =  6.  Result = 0xddd
        .quad -1         # Should not change

# Jump table
jtab:
        .quad case_0
        .quad case_def
        .quad case_2_5
        .quad case_3
        .quad case_def
        .quad case_2_5

# Code snippets
case_0:
        irmovq $0xaaa, %rax
        jmp done
case_2_5:
        irmovq $0xbbb, %rax
        jmp done
case_3:
        irmovq $0xccc, %rax
        jmp done
case_def:
        irmovq $0xddd, %rax
        jmp done

# long switchv(long idx)
# idx in %rdi
switchv:
        andq %rdi,%rdi           # Test idx
        jl case_def              # If < 0, default
        irmovq $5,%rdx
        subq %rdi,%rdx           # Compute 5-idx
        jl case_def              # If idx > 5, default
        addq %rdi,%rdi
        addq %rdi,%rdi
        addq %rdi,%rdi           # 8 * idx
        mrmovq jtab(%rdi),%rdx  # jtab[idx]
        pushq %rdx               # Put on top of stack
        ret                      # Goto jtab[idx]
```

```
done:
        ret                         # Return from switchv

main:
        pushq %rbx                  # Save %rbx
        xorq %rbx,%rbx              # i = 0
loop:
        irmovq $-1,%rdi
        addq %rbx,%rdi              # i+MINVAL
        call switchv
        rrmovq %rbx,%rdx
        addq %rdx,%rdx
        addq %rdx,%rdx
        addq %rdx,%rdx              # 8*i
        irmovq vals,%rdi
        addq %rdx,%rdi              # &vals[i]
        rmmovq %rax,(%rdi)          # vals[i] = switchv(idx)
        irmovq $1,%rdi
        addq %rdi,%rbx              # i++
        irmovq 8, %rdi
        subq %rbx,%rdi              # 8 - i
        jg loop
        popq %rbx
        ret

.pos 0x200
stack:
```

code/arch/switchv.ys

Problem 4.51 Solution:

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

Stage	`iaddq` V, rB
Fetch	icode : ifun $\leftarrow$ $M_1[PC]$ rA : rB $\leftarrow$ $M_1[PC+1]$ valC $\leftarrow$ $M_8[PC+2]$ valP $\leftarrow$ PC $+ 10$
Decode	valB $\leftarrow$ R[rB]
Execute	valE $\leftarrow$ valB + valC
Memory	
Write back	R[rB] $\leftarrow$ valE
PC update	PC $\leftarrow$ valP

Problem 4.52 Solution:

The following HCL code includes an implementation of the `iaddq` instruction. The implementation is fairly straightforward given the computation steps listed in the solution to problem 4.51. You can test the solution using the test code in the `ptest` subdirectory. Make sure you use command line argument ‘`-i`.’

code/arch/seq-full-ans.hcl

```
####################################################################
#  HCL Description of Control for Single Cycle Y86-64 Processor SEQ   #
#  Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2010       #
####################################################################

## This is the solution for the iaddq problem

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

quote '#include <stdio.h>'
quote '#include "isa.h"'
quote '#include "sim.h"'
quote 'int sim_main(int argc, char *argv[]);'
quote 'word_t gen_pc(){return 0;}'
quote 'int main(int argc, char *argv[])'
quote '  {plusmode=0;return sim_main(argc,argv);}'

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

##### Symbolic representation of Y86-64 Instruction Codes #############
wordsig INOP    'I_NOP'
wordsig IHALT   'I_HALT'
wordsig IRRMOVQ 'I_RRMOVQ'
wordsig IIRMOVQ 'I_IRMOVQ'
wordsig IRMMOVQ 'I_RMMOVQ'
wordsig IMRMOVQ 'I_MRMOVQ'
wordsig IOPQ    'I_ALU'
wordsig IJXX    'I_JMP'
wordsig ICALL   'I_CALL'
wordsig IRET    'I_RET'
wordsig IPUSHQ  'I_PUSHQ'
wordsig IPOPQ   'I_POPQ'
# Instruction code for iaddq instruction
wordsig IIADDQ  'I_IADDQ'

##### Symbolic represenations of Y86-64 function codes                  #####
wordsig FNONE    'F_NONE'        # Default function code

##### Symbolic representation of Y86-64 Registers referenced explicitly #####
wordsig RRSP     'REG_RSP'      # Stack Pointer
```

```
wordsig RNONE    'REG_NONE'     # Special value indicating "no register"

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

##### Possible instruction status values                            #####
wordsig SAOK    'STAT_AOK'      # Normal execution
wordsig SADR    'STAT_ADR'      # Invalid memory address
wordsig SINS    'STAT_INS'      # Invalid instruction
wordsig SHLT    'STAT_HLT'      # Halt instruction encountered

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

##### Fetch stage inputs                #####
wordsig pc 'pc'                         # Program counter
##### Fetch stage computations          #####
wordsig imem_icode 'imem_icode'         # icode field from instruction memory
wordsig imem_ifun  'imem_ifun'          # ifun field from instruction memory
wordsig icode     'icode'               # Instruction control code
wordsig ifun      'ifun'                # Instruction function
wordsig rA        'ra'                  # rA field from instruction
wordsig rB        'rb'                  # rB field from instruction
wordsig valC      'valc'                # Constant from instruction
wordsig valP      'valp'                # Address of following instruction
boolsig imem_error 'imem_error'         # Error signal from instruction memory
boolsig instr_valid 'instr_valid'       # Is fetched instruction valid?

##### Decode stage computations         #####
wordsig valA    'vala'                  # Value from register A port
wordsig valB    'valb'                  # Value from register B port

##### Execute stage computations        #####
wordsig valE    'vale'                  # Value computed by ALU
boolsig Cnd     'cond'                  # Branch test

##### Memory stage computations         #####
wordsig valM    'valm'                  # Value read from memory
boolsig dmem_error 'dmem_error'         # Error signal from data memory


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

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

# Determine instruction code
word icode = [
        imem_error: INOP;
        1: imem_icode;          # Default: get from instruction memory
```

```
];

# Determine instruction function
word ifun = [
        imem_error: FNONE;
        1: imem_ifun;           # Default: get from instruction memory
];

bool instr_valid = icode in
        { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
               IIADDQ,
               IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };

# Does fetched instruction require a regid byte?
bool need_regids =
        icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                     IIADDQ,
                     IIRMOVQ, IRMMOVQ, IMRMOVQ };

# Does fetched instruction require a constant word?
bool need_valC =
        icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL, IIADDQ };

################ Decode Stage    ###################################

## What register should be used as the A source?
word srcA = [
        icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : rA;
        icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];

## What register should be used as the B source?
word srcB = [
        icode in { IOPQ, IRMMOVQ, IMRMOVQ  } : rB;
        icode in { IIADDQ  } : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
word dstE = [
        icode in { IRRMOVQ } && Cnd : rB;
        icode in { IIRMOVQ, IOPQ} : rB;
        icode in { IIADDQ } : rB;
        icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];

## What register should be used as the M destination?
```

```
word dstM = [
	icode in { IMRMOVQ, IPOPQ } : rA;
	1 : RNONE;  # Don't write any register
];

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

## Select input A to ALU
word aluA = [
	icode in { IRRMOVQ, IOPQ } : valA;
	icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : valC;
	icode in { IIADDQ } : valC;
	icode in { ICALL, IPUSHQ } : -8;
	icode in { IRET, IPOPQ } : 8;
	# Other instructions don't need ALU
];

## Select input B to ALU
word aluB = [
	icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
		      IPUSHQ, IRET, IPOPQ } : valB;
	icode in { IIADDQ } : valB;
	icode in { IRRMOVQ, IIRMOVQ } : 0;
	# Other instructions don't need ALU
];

## Set the ALU function
word alufun = [
	icode == IOPQ : ifun;
	1 : ALUADD;
];

## Should the condition codes be updated?
bool set_cc = icode in { IOPQ, IIADDQ };

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

## Set read control signal
bool mem_read = icode in { IMRMOVQ, IPOPQ, IRET};

## Set write control signal
bool mem_write = icode in { IRMMOVQ, IPUSHQ, ICALL };

## Select memory address
word mem_addr = [
	icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : valE;
	icode in { IPOPQ, IRET } : valA;
	# Other instructions don't need address
];
```

```
## Select memory input data
word mem_data = [
        # Value from register
        icode in { IRMMOVQ, IPUSHQ } : valA;
        # Return PC
        icode == ICALL : valP;
        # Default: Don't write anything
];

## Determine instruction status
word Stat = [
        imem_error || dmem_error : SADR;
        !instr_valid: SINS;
        icode == IHALT : SHLT;
        1 : SAOK;
];

################ Program Counter Update ############################

## What address should instruction be fetched at

word new_pc = [
        # Call.  Use instruction constant
        icode == ICALL : valC;
        # Taken branch.  Use instruction constant
        icode == IJXX && Cnd : valC;
        # Completion of RET instruction.  Use value from stack
        icode == IRET : valM;
        # Default: Use incremented PC
        1 : valP;
];
```

code/arch/seq-full-ans.hcl

Problem 4.53 Solution:

This is a hard problem, because there are many possible combinations of special cases that can occur simultaneously. Figure 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 `%rsp`). 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.

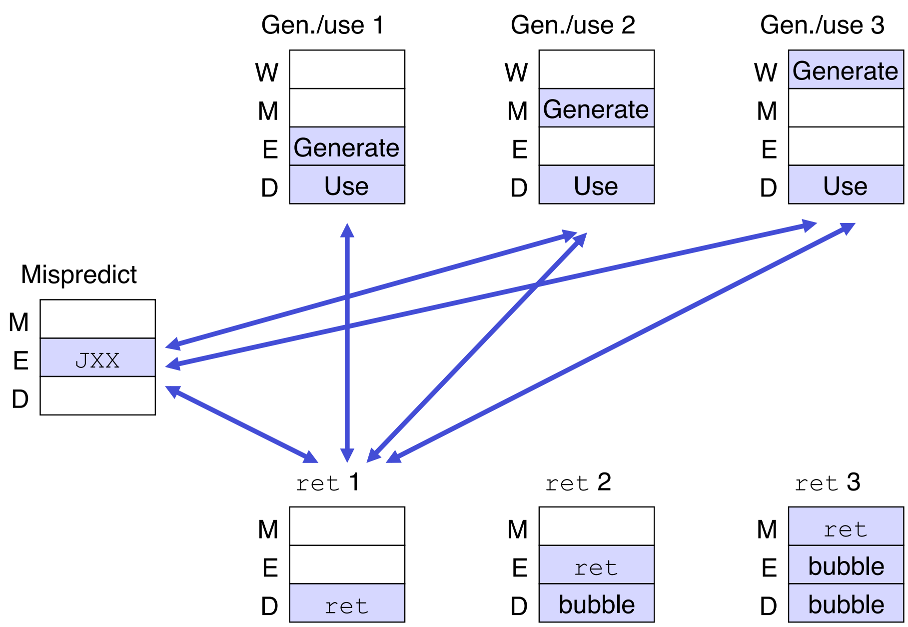


Figure 1: **Pipeline states for special control conditions.** The pairs connected by arrows can arise simultaneously.

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 { IMRMOVQ, IPOPQ } && 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 4.54 Solution:

This problem is similar to Problem 4.52, but for the PIPE processor.

The following HCL code include an implementation of the `iaddq` instruction. You can test the solution 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 Y86-64 Processor     #
#    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2014     #
####################################################################
```

```

## This is the solution for the iaddq 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 Y86-64 Instruction Codes #############
wordsig INOP    'I_NOP'
wordsig IHALT   'I_HALT'
wordsig IRRMOVQ 'I_RRMOVQ'
wordsig IIRMOVQ 'I_IRMOVQ'
wordsig IRMMOVQ 'I_RMMOVQ'
wordsig IMRMOVQ 'I_MRMOVQ'
wordsig IOPQ    'I_ALU'
wordsig IJXX    'I_JMP'
wordsig ICALL   'I_CALL'
wordsig IRET    'I_RET'
wordsig IPUSHQ  'I_PUSHQ'
wordsig IPOPQ   'I_POPQ'
# Instruction code for iaddq instruction
wordsig IIADDQ  'I_IADDQ'

##### Symbolic represenations of Y86-64 function codes            #####
wordsig FNONE    'F_NONE'        # Default function code

##### Symbolic representation of Y86-64 Registers referenced      #####
wordsig RRSP     'REG_RSP'           # Stack Pointer
wordsig RNONE    'REG_NONE'          # Special value indicating "no register"

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

##### Possible instruction status values                       #####
wordsig SBUB    'STAT_BUB'      # Bubble in stage
wordsig SAOK    'STAT_AOK'      # Normal execution
wordsig SADR    'STAT_ADR'      # Invalid memory address
wordsig SINS    'STAT_INS'      # Invalid instruction
```

```
wordsig SHLT    ’STAT_HLT’      # Halt instruction encountered

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

##### Pipeline Register F ##########################################

wordsig F_predPC ’pc_curr->pc’       # Predicted value of PC

##### Intermediate Values in Fetch Stage ###########################

wordsig imem_icode  ’imem_icode’      # icode field from instruction memory
wordsig imem_ifun   ’imem_ifun’       # ifun  field from instruction memory
wordsig f_icode ’if_id_next->icode’  # (Possibly modified) instruction code
wordsig f_ifun  ’if_id_next->ifun’   # Fetched instruction function
wordsig f_valC  ’if_id_next->valc’   # Constant data of fetched instruction
wordsig 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 ##########################################
wordsig D_icode ’if_id_curr->icode’   # Instruction code
wordsig D_rA ’if_id_curr->ra’        # rA field from instruction
wordsig D_rB ’if_id_curr->rb’        # rB field from instruction
wordsig D_valP ’if_id_curr->valp’     # Incremented PC

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

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

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

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

##### Pipeline Register M                  #########################
wordsig M_stat ’ex_mem_curr->status’     # Instruction status
```

```
wordsig M_icode 'ex_mem_curr->icode'    # Instruction code
wordsig M_ifun  'ex_mem_curr->ifun'     # Instruction function
wordsig M_valA  'ex_mem_curr->vala'      # Source A value
wordsig M_dstE 'ex_mem_curr->deste'     # Destination E register ID
wordsig M_valE  'ex_mem_curr->vale'      # ALU E value
wordsig 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 ##########################
wordsig m_valM 'mem_wb_next->valm'      # valM generated by memory
wordsig m_stat 'mem_wb_next->status'    # stat (possibly modified to be SADR)

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

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

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

## What address should instruction be fetched at
word 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
word f_icode = [
        imem_error : INOP;
        1: imem_icode;
];

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

# Is instruction valid?
```

```
bool instr_valid = f_icode in
        { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
          IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IIADDQ };

# Determine status code for fetched instruction
word f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];

# Does fetched instruction require a regid byte?
bool need_regids =
        f_icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                     IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ };

# Does fetched instruction require a constant word?
bool need_valC =
        f_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL, IIADDQ };

# Predict next value of PC
word f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];

################ Decode Stage ######################################


## What register should be used as the A source?
word d_srcA = [
        D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : D_rA;
        D_icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];

## What register should be used as the B source?
word d_srcB = [
        D_icode in { IOPQ, IRMMOVQ, IMRMOVQ, IIADDQ  } : D_rB;
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
word d_dstE = [
        D_icode in { IRRMOVQ, IIRMOVQ, IOPQ, IIADDQ } : D_rB;
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];
```

```

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

## What should be the A value?
## Forward into decode stage for valA
word 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
];

word 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
word aluA = [
	E_icode in { IRRMOVQ, IOPQ } : E_valA;
	E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IIADDQ } : E_valC;
	E_icode in { ICALL, IPUSHQ } : -8;
	E_icode in { IRET, IPOPQ } : 8;
	# Other instructions don't need ALU
];

## Select input B to ALU
word aluB = [
	E_icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
		     IPUSHQ, IRET, IPOPQ, IIADDQ } : E_valB;
	E_icode in { IRRMOVQ, IIRMOVQ } : 0;
	# Other instructions don't need ALU
];

## Set the ALU function
word alufun = [
	E_icode == IOPQ : E_ifun;
```

```
        1 : ALUADD;
];

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

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

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

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

## Select memory address
word mem_addr = [
        M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : M_valE;
        M_icode in { IPOPQ, IRET } : M_valA;
        # Other instructions don't need address
];

## Set read control signal
bool mem_read = M_icode in { IMRMOVQ, IPOPQ, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVQ, IPUSHQ, ICALL };

## Update the status
word m_stat = [
        dmem_error : SADR;
        1 : M_stat;
];

## Set E port register ID
word w_dstE = W_dstE;

## Set E port value
word w_valE = W_valE;

## Set M port register ID
word w_dstM = W_dstM;

## Set M port value
word w_valM = W_valM;

```

```
## Update processor status
word Stat = [
        W_stat == SBUB : SAOK;
        1 : W_stat;
];

################ Pipeline Register Control #########################

# 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 =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB } ||
        # 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 =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB };

bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&
          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) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB};

# 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-full-ans.hcl

Problem 4.55 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

```
####################################################################
#    HCL Description of Control for Pipelined Y86-64 Processor     #
#    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2014     #
####################################################################

## This is the solution for the branches not-taken 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 Y86-64 Instruction Codes #############
wordsig INOP    'I_NOP'
wordsig IHALT   'I_HALT'
wordsig IRRMOVQ 'I_RRMOVQ'
wordsig IIRMOVQ 'I_IRMOVQ'
wordsig IRMMOVQ 'I_RMMOVQ'
wordsig IMRMOVQ 'I_MRMOVQ'
wordsig IOPQ    'I_ALU'
wordsig IJXX    'I_JMP'
wordsig ICALL   'I_CALL'
wordsig IRET    'I_RET'
wordsig IPUSHQ  'I_PUSHQ'
```

```
wordsig IPOPQ   ’I_POPQ’

##### Symbolic represenations of Y86-64 function codes            #####
wordsig FNONE    ’F_NONE’        # Default function code

##### Symbolic representation of Y86-64 Registers referenced      #####
wordsig RRSP     ’REG_RSP’           # Stack Pointer
wordsig RNONE    ’REG_NONE’          # Special value indicating "no register"

##### ALU Functions referenced explicitly ##########################
wordsig ALUADD  ’A_ADD’              # ALU should add its arguments
## BNT: For modified branch prediction, need to distinguish
## conditional vs. unconditional branches
##### Jump conditions referenced explicitly
wordsig UNCOND ’C_YES’               # Unconditional transfer

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

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

##### Pipeline Register F ##########################################

wordsig F_predPC ’pc_curr->pc’       # Predicted value of PC

##### Intermediate Values in Fetch Stage ###########################

wordsig imem_icode  ’imem_icode’      # icode field from instruction memory
wordsig imem_ifun   ’imem_ifun’       # ifun  field from instruction memory
wordsig f_icode ’if_id_next->icode’  # (Possibly modified) instruction code
wordsig f_ifun  ’if_id_next->ifun’   # Fetched instruction function
wordsig f_valC  ’if_id_next->valc’   # Constant data of fetched instruction
wordsig 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 ##########################################
wordsig D_icode ’if_id_curr->icode’   # Instruction code
wordsig D_rA ’if_id_curr->ra’        # rA field from instruction
wordsig D_rB ’if_id_curr->rb’        # rB field from instruction
wordsig D_valP ’if_id_curr->valp’     # Incremented PC

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

wordsig d_srcA   ’id_ex_next->srca’  # srcA from decoded instruction
wordsig d_srcB   ’id_ex_next->srcb’  # srcB from decoded instruction
```

```
wordsig d_rvalA 'd_regvala'          # valA read from register file
wordsig d_rvalB 'd_regvalb'          # valB read from register file

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

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

##### Pipeline Register M                  #########################
wordsig M_stat 'ex_mem_curr->status'     # Instruction status
wordsig M_icode 'ex_mem_curr->icode'    # Instruction code
wordsig M_ifun  'ex_mem_curr->ifun'     # Instruction function
wordsig M_valA  'ex_mem_curr->vala'      # Source A value
wordsig M_dstE 'ex_mem_curr->deste'     # Destination E register ID
wordsig M_valE  'ex_mem_curr->vale'      # ALU E value
wordsig 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 ##########################
wordsig m_valM 'mem_wb_next->valm'      # valM generated by memory
wordsig m_stat 'mem_wb_next->status'    # stat (possibly modified to be SADR)

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

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

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

## What address should instruction be fetched at
word 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;
];

## Determine icode of fetched instruction
word f_icode = [
	imem_error : INOP;
	1: imem_icode;
];

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

# Is instruction valid?
bool instr_valid = f_icode in
	{ INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
	  IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };

# Determine status code for fetched instruction
word f_stat = [
	imem_error: SADR;
	!instr_valid : SINS;
	f_icode == IHALT : SHLT;
	1 : SAOK;
];

# Does fetched instruction require a regid byte?
bool need_regids =
	f_icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
		     IIRMOVQ, IRMMOVQ, IMRMOVQ };

# Does fetched instruction require a constant word?
bool need_valC =
	f_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };

# Predict next value of PC
word f_predPC = [
	# BNT: Revised branch prediction rule:
	#   Unconditional branch is taken, others not taken
	f_icode == IJXX && f_ifun == UNCOND : f_valC;
	f_icode in { ICALL } : f_valC;
	1 : f_valP;
```

```
];

################ Decode Stage ######################################


## What register should be used as the A source?
word d_srcA = [
        D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : D_rA;
        D_icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don’t need register
];

## What register should be used as the B source?
word d_srcB = [
        D_icode in { IOPQ, IRMMOVQ, IMRMOVQ  } : D_rB;
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don’t need register
];

## What register should be used as the E destination?
word d_dstE = [
        D_icode in { IRRMOVQ, IIRMOVQ, IOPQ} : D_rB;
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don’t write any register
];

## What register should be used as the M destination?
word d_dstM = [
        D_icode in { IMRMOVQ, IPOPQ } : D_rA;
        1 : RNONE;  # Don’t write any register
];

## What should be the A value?
## Forward into decode stage for valA
word 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
];

word 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 #####################################

# BNT: When some branches are predicted as not-taken, you need some
# 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
word aluA = [
        E_icode in { IRRMOVQ, IOPQ } : E_valA;
        # BNT: Use ALU to pass E_valC to M_valE
        E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX } : E_valC;
        E_icode in { ICALL, IPUSHQ } : -8;
        E_icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];

## Select input B to ALU
word aluB = [
        E_icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                     IPUSHQ, IRET, IPOPQ } : E_valB;
        # BNT: Add 0 to valC
        E_icode in { IRRMOVQ, IIRMOVQ, IJXX } : 0;
        # Other instructions don't need ALU
];

## Set the ALU function
word alufun = [
        E_icode == IOPQ : E_ifun;
        1 : ALUADD;
];

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

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

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

```
################ Memory Stage ######################################

## Select memory address
word mem_addr = [
	M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : M_valE;
	M_icode in { IPOPQ, IRET } : M_valA;
	# Other instructions don't need address
];

## Set read control signal
bool mem_read = M_icode in { IMRMOVQ, IPOPQ, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVQ, IPUSHQ, ICALL };

## Update the status
word m_stat = [
	dmem_error : SADR;
	1 : M_stat;
];

## Set E port register ID
word w_dstE = W_dstE;

## Set E port value
word w_valE = W_valE;

## Set M port register ID
word w_dstM = W_dstM;

## Set M port value
word w_valM = W_valM;

## Update processor status
word Stat = [
	W_stat == SBUB : SAOK;
	1 : W_stat;
];

################ Pipeline Register Control #########################

# 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 =
	# Conditions for a load/use hazard
	E_icode in { IMRMOVQ, IPOPQ } &&
	 E_dstM in { d_srcA, d_srcB } ||
	# 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 =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB };

bool D_bubble =
        # Mispredicted branch
        # BNT: Changed misprediction condition
        (E_icode == IJXX && E_ifun != UNCOND && e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&
          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
        # BNT: Changed misprediction condition
        (E_icode == IJXX && E_ifun != UNCOND && e_Cnd) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB};

# 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-nt-ans.hcl

Problem 4.56 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 4.55. 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

```
####################################################################
```

```
#    HCL Description of Control for Pipelined Y86-64 Processor     #
#    Copyright (C) Randal E. Bryant, David R. O’Hallaron, 2014     #
####################################################################

## BBTFNT: This is the solution for the backward taken, forward
## not-taken branch prediction 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 Y86-64 Instruction Codes #############
wordsig INOP    ’I_NOP’
wordsig IHALT   ’I_HALT’
wordsig IRRMOVQ ’I_RRMOVQ’
wordsig IIRMOVQ ’I_IRMOVQ’
wordsig IRMMOVQ ’I_RMMOVQ’
wordsig IMRMOVQ ’I_MRMOVQ’
wordsig IOPQ    ’I_ALU’
wordsig IJXX    ’I_JMP’
wordsig ICALL   ’I_CALL’
wordsig IRET    ’I_RET’
wordsig IPUSHQ  ’I_PUSHQ’
wordsig IPOPQ   ’I_POPQ’

##### Symbolic represenations of Y86-64 function codes            #####
wordsig FNONE    ’F_NONE’        # Default function code

##### Symbolic representation of Y86-64 Registers referenced      #####
wordsig RRSP     ’REG_RSP’            # Stack Pointer
wordsig RNONE    ’REG_NONE’           # Special value indicating "no register"

##### ALU Functions referenced explicitly ##########################
wordsig ALUADD  ’A_ADD’               # ALU should add its arguments
## BBTFNT: For modified branch prediction, need to distinguish
## conditional vs. unconditional branches
##### Jump conditions referenced explicitly
wordsig UNCOND ’C_YES’                # Unconditional transfer
```

```

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

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

##### Pipeline Register F ##########################################

wordsig F_predPC 'pc_curr->pc'       # Predicted value of PC

##### Intermediate Values in Fetch Stage ###########################

wordsig imem_icode  'imem_icode'      # icode field from instruction memory
wordsig imem_ifun   'imem_ifun'       # ifun  field from instruction memory
wordsig f_icode 'if_id_next->icode'  # (Possibly modified) instruction code
wordsig f_ifun  'if_id_next->ifun'   # Fetched instruction function
wordsig f_valC  'if_id_next->valc'   # Constant data of fetched instruction
wordsig 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 ##########################################
wordsig D_icode 'if_id_curr->icode'   # Instruction code
wordsig D_rA 'if_id_curr->ra'        # rA field from instruction
wordsig D_rB 'if_id_curr->rb'        # rB field from instruction
wordsig D_valP 'if_id_curr->valp'     # Incremented PC

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

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

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

##### Intermediate Values in Execute Stage #########################
```

```
wordsig e_valE ’ex_mem_next->vale’      # valE generated by ALU
boolsig e_Cnd ’ex_mem_next->takebranch’ # Does condition hold?
wordsig e_dstE ’ex_mem_next->deste’      # dstE (possibly modified to be RNONE)

##### Pipeline Register M                  ##########################
wordsig M_stat ’ex_mem_curr->status’     # Instruction status
wordsig M_icode ’ex_mem_curr->icode’    # Instruction code
wordsig M_ifun  ’ex_mem_curr->ifun’     # Instruction function
wordsig M_valA  ’ex_mem_curr->vala’      # Source A value
wordsig M_dstE ’ex_mem_curr->deste’     # Destination E register ID
wordsig M_valE  ’ex_mem_curr->vale’      # ALU E value
wordsig 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 ##########################
wordsig m_valM ’mem_wb_next->valm’      # valM generated by memory
wordsig m_stat ’mem_wb_next->status’    # stat (possibly modified to be SADR)

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

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

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

## What address should instruction be fetched at
word f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        # BBTFNT: Mispredicted forward branch.  Fetch at target (now in valE)
        M_icode == IJXX && M_ifun != UNCOND && M_valE >= M_valA
           && M_Cnd : M_valE;
        # BBTFNT: Mispredicted backward branch.
        #    Fetch at incremented PC (now in valE)
        M_icode == IJXX && M_ifun != UNCOND && M_valE < M_valA
          && !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
```

```
word f_icode = [
        imem_error : INOP;
        1: imem_icode;
];

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

# Is instruction valid?
bool instr_valid = f_icode in
        { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
          IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };

# Determine status code for fetched instruction
word f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];

# Does fetched instruction require a regid byte?
bool need_regids =
        f_icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                     IIRMOVQ, IRMMOVQ, IMRMOVQ };

# Does fetched instruction require a constant word?
bool need_valC =
        f_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };

# Predict next value of PC
word f_predPC = [
        f_icode in { ICALL } : f_valC;
        f_icode == IJXX && f_ifun == UNCOND : f_valC; # Unconditional branch
        f_icode == IJXX && f_valC < f_valP : f_valC; # Backward branch
        # BBTFNT: Forward conditional branches will default to valP
        1 : f_valP;
];

################ Decode Stage ######################################


## What register should be used as the A source?
word d_srcA = [
        D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : D_rA;
        D_icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
```

```
];

## What register should be used as the B source?
word d_srcB = [
        D_icode in { IOPQ, IRMMOVQ, IMRMOVQ  } : D_rB;
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don’t need register
];

## What register should be used as the E destination?
word d_dstE = [
        D_icode in { IRRMOVQ, IIRMOVQ, IOPQ} : D_rB;
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don’t write any register
];

## What register should be used as the M destination?
word d_dstM = [
        D_icode in { IMRMOVQ, IPOPQ } : D_rA;
        1 : RNONE;  # Don’t write any register
];

## What should be the A value?
## Forward into decode stage for valA
word 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
];

word 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 #####################################

# BBTFNT: When some branches are predicted as not-taken, you need some
# 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
word aluA = [
	E_icode in { IRRMOVQ, IOPQ } : E_valA;
	# BBTFNT: Use ALU to pass E_valC to M_valE
	E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX } : E_valC;
	E_icode in { ICALL, IPUSHQ } : -8;
	E_icode in { IRET, IPOPQ } : 8;
	# Other instructions don't need ALU
];

## Select input B to ALU
word aluB = [
	E_icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
		     IPUSHQ, IRET, IPOPQ } : E_valB;
	# BBTFNT: Add 0 to valC
	E_icode in { IRRMOVQ, IIRMOVQ, IJXX } : 0;
	# Other instructions don't need ALU
];

## Set the ALU function
word alufun = [
	E_icode == IOPQ : E_ifun;
	1 : ALUADD;
];

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

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

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

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

## Select memory address
word mem_addr = [
	M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : M_valE;
	M_icode in { IPOPQ, IRET } : M_valA;
	# Other instructions don't need address
];

## Set read control signal
```

```
bool mem_read = M_icode in { IMRMOVQ, IPOPQ, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVQ, IPUSHQ, ICALL };

## Update the status
word m_stat = [
        dmem_error : SADR;
        1 : M_stat;
];

## Set E port register ID
word w_dstE = W_dstE;

## Set E port value
word w_valE = W_valE;

## Set M port register ID
word w_dstM = W_dstM;

## Set M port value
word w_valM = W_valM;

## Update processor status
word Stat = [
        W_stat == SBUB : SAOK;
        1 : W_stat;
];

################ Pipeline Register Control #########################

# 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 =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB } ||
        # 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 =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB };

bool D_bubble =
        # Mispredicted branch
```

```
        # BBTFNT: Changed misprediction condition
        (E_icode == IJXX && E_ifun != UNCOND &&
          (E_valC < E_valA && !e_Cnd || E_valC >= E_valA && e_Cnd)) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&
          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
        # BBTFNT: Changed misprediction condition
        (E_icode == IJXX && E_ifun != UNCOND &&
          (E_valC < E_valA && !e_Cnd || E_valC >= E_valA && e_Cnd)) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         E_dstM in { d_srcA, d_srcB};

# 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-btfnt-ans.hcl

Problem 4.57 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:

$$\text{E\_icode} \in \{\text{IMRMOVQ}, \text{IPOPQ}\} \;\&\&\; (\text{E\_dstM} = \text{d\_srcB} \;||\; \text{E\_dstM} = \text{d\_srcA} \;\&\&\; !\,\text{D\_icode} \in \{\text{IRMMOVQ}, \text{IPUSHQ}\})$$

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

code/arch/pipe-lf-ans.hcl

```
####################################################################
#    HCL Description of Control for Pipelined Y86-64 Processor     #
#    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2014     #
####################################################################

## 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 Y86-64 Instruction Codes #############
wordsig INOP    'I_NOP'
wordsig IHALT   'I_HALT'
wordsig IRRMOVQ 'I_RRMOVQ'
wordsig IIRMOVQ 'I_IRMOVQ'
wordsig IRMMOVQ 'I_RMMOVQ'
wordsig IMRMOVQ 'I_MRMOVQ'
wordsig IOPQ    'I_ALU'
wordsig IJXX    'I_JMP'
wordsig ICALL   'I_CALL'
wordsig IRET    'I_RET'
wordsig IPUSHQ  'I_PUSHQ'
wordsig IPOPQ   'I_POPQ'

##### Symbolic represenations of Y86-64 function codes            #####
wordsig FNONE    'F_NONE'        # Default function code

##### Symbolic representation of Y86-64 Registers referenced      #####
wordsig RRSP     'REG_RSP'            # Stack Pointer
wordsig RNONE    'REG_NONE'           # Special value indicating "no register"

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

##### Possible instruction status values                     #####
wordsig SBUB    'STAT_BUB'      # Bubble in stage
wordsig SAOK    'STAT_AOK'      # Normal execution
```

```
wordsig SADR    ’STAT_ADR’      # Invalid memory address
wordsig SINS    ’STAT_INS’      # Invalid instruction
wordsig SHLT    ’STAT_HLT’      # Halt instruction encountered

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

##### Pipeline Register F ##########################################

wordsig F_predPC ’pc_curr->pc’       # Predicted value of PC

##### Intermediate Values in Fetch Stage ###########################

wordsig imem_icode  ’imem_icode’      # icode field from instruction memory
wordsig imem_ifun   ’imem_ifun’       # ifun  field from instruction memory
wordsig f_icode ’if_id_next->icode’  # (Possibly modified) instruction code
wordsig f_ifun  ’if_id_next->ifun’   # Fetched instruction function
wordsig f_valC  ’if_id_next->valc’   # Constant data of fetched instruction
wordsig 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 ##########################################
wordsig D_icode ’if_id_curr->icode’   # Instruction code
wordsig D_rA ’if_id_curr->ra’        # rA field from instruction
wordsig D_rB ’if_id_curr->rb’        # rB field from instruction
wordsig D_valP ’if_id_curr->valp’     # Incremented PC

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

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

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

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

```
##### Pipeline Register M                  #########################
wordsig M_stat ’ex_mem_curr->status’     # Instruction status
wordsig M_icode ’ex_mem_curr->icode’    # Instruction code
wordsig M_ifun  ’ex_mem_curr->ifun’     # Instruction function
wordsig M_valA  ’ex_mem_curr->vala’      # Source A value
wordsig M_dstE ’ex_mem_curr->deste’     # Destination E register ID
wordsig M_valE  ’ex_mem_curr->vale’      # ALU E value
wordsig 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
## LF: Carry srcA up to pipeline register M
wordsig M_srcA ’ex_mem_curr->srca’      # Source A register ID

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

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

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

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

## What address should instruction be fetched at
word 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
word f_icode = [
	imem_error : INOP;
	1: imem_icode;
];

# Determine ifun
word f_ifun = [
	imem_error : FNONE;
```

```
        1: imem_ifun;
];

# Is instruction valid?
bool instr_valid = f_icode in
        { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
          IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ };

# Determine status code for fetched instruction
word f_stat = [
        imem_error: SADR;
        !instr_valid : SINS;
        f_icode == IHALT : SHLT;
        1 : SAOK;
];

# Does fetched instruction require a regid byte?
bool need_regids =
        f_icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                     IIRMOVQ, IRMMOVQ, IMRMOVQ };

# Does fetched instruction require a constant word?
bool need_valC =
        f_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };

# Predict next value of PC
word f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        1 : f_valP;
];

################ Decode Stage ######################################


## What register should be used as the A source?
word d_srcA = [
        D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : D_rA;
        D_icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];

## What register should be used as the B source?
word d_srcB = [
        D_icode in { IOPQ, IRMMOVQ, IMRMOVQ  } : D_rB;
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
word d_dstE = [
```

```
        D_icode in { IRRMOVQ, IIRMOVQ, IOPQ} : D_rB;
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];

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

## What should be the A value?
## Forward into decode stage for valA
word 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
];

word 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
word aluA = [
        E_icode in { IRRMOVQ, IOPQ } : E_valA;
        E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : E_valC;
        E_icode in { ICALL, IPUSHQ } : -8;
        E_icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];

## Select input B to ALU
word aluB = [
        E_icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                     IPUSHQ, IRET, IPOPQ } : E_valB;
        E_icode in { IRRMOVQ, IIRMOVQ } : 0;
        # Other instructions don't need ALU
];
```

```
## Set the ALU function
word alufun = [
        E_icode == IOPQ : E_ifun;
        1 : ALUADD;
];

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

## Generate valA in execute stage
## LB: With load forwarding, want to insert valM
##   from memory stage when appropriate
word e_valA = [
        # Forwarding Condition
        M_dstM == E_srcA && E_icode in { IPUSHQ, IRMMOVQ } : m_valM;
        1 : E_valA;  # Use valA from stage pipe register
];

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

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

## Select memory address
word mem_addr = [
        M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ } : M_valE;
        M_icode in { IPOPQ, IRET } : M_valA;
        # Other instructions don't need address
];

## Set read control signal
bool mem_read = M_icode in { IMRMOVQ, IPOPQ, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVQ, IPUSHQ, ICALL };

## Update the status
word m_stat = [
        dmem_error : SADR;
        1 : M_stat;
];

## Set E port register ID
word w_dstE = W_dstE;
```

```
## Set E port value
word w_valE = W_valE;

## Set M port register ID
word w_dstM = W_dstM;

## Set M port value
word w_valM = W_valM;

## Update processor status
word Stat = [
        W_stat == SBUB : SAOK;
        1 : W_stat;
];

################ Pipeline Register Control #########################

# 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 =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         (E_dstM == d_srcB ||
          (E_dstM == d_srcA && !D_icode in { IRMMOVQ, IPUSHQ })) ||
        # 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 =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
        E_icode in { IMRMOVQ, IPOPQ } &&
         (E_dstM == d_srcB ||
          (E_dstM == d_srcA && !D_icode in { IRMMOVQ, IPUSHQ }));

bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }) &&
          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) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOPQ } &&
         (E_dstM == d_srcB ||
          (E_dstM == d_srcA && !D_icode in { IRMMOVQ, IPUSHQ }));

# 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-lf-ans.hcl

Problem 4.58 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

```
####################################################################
#    HCL Description of Control for Pipelined Y86-64 Processor     #
#    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2014     #
####################################################################

## This is a solution to the single write port problem
## Overall strategy:  IPOPQ passes through pipe,
## treated as stack pointer increment, but not incrementing the PC
## On refetch, modify fetched icode to indicate an instruction "IPOP2",
## which reads from memory.

####################################################################
#    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 Y86-64 Instruction Codes #############
wordsig INOP    'I_NOP'
wordsig IHALT   'I_HALT'
wordsig IRRMOVQ 'I_RRMOVQ'
wordsig IIRMOVQ 'I_IRMOVQ'
wordsig IRMMOVQ 'I_RMMOVQ'
wordsig IMRMOVQ 'I_MRMOVQ'
wordsig IOPQ    'I_ALU'
wordsig IJXX    'I_JMP'
wordsig ICALL   'I_CALL'
wordsig IRET    'I_RET'
wordsig IPUSHQ  'I_PUSHQ'
wordsig IPOPQ   'I_POPQ'
# 1W: Special instruction code for second try of popq
wordsig IPOP2   'I_POP2'

##### Symbolic represenations of Y86-64 function codes            #####
wordsig FNONE    'F_NONE'        # Default function code

##### Symbolic representation of Y86-64 Registers referenced      #####
wordsig RRSP     'REG_RSP'           # Stack Pointer
wordsig RNONE    'REG_NONE'          # Special value indicating "no register"

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

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

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

##### Pipeline Register F ##########################################

wordsig F_predPC 'pc_curr->pc'       # Predicted value of PC

##### Intermediate Values in Fetch Stage ###########################

wordsig imem_icode  'imem_icode'      # icode field from instruction memory
wordsig imem_ifun   'imem_ifun'       # ifun  field from instruction memory
```

```
wordsig f_icode 'if_id_next->icode'  # (Possibly modified) instruction code
wordsig f_ifun  'if_id_next->ifun'   # Fetched instruction function
wordsig f_valC  'if_id_next->valc'   # Constant data of fetched instruction
wordsig f_valP  'if_id_next->valp'   # Address of following instruction
## 1W: Provide access to the PC value for the current instruction
wordsig f_pc    'f_pc'               # Address of fetched instruction
boolsig imem_error 'imem_error'      # Error signal from instruction memory
boolsig instr_valid 'instr_valid'    # Is fetched instruction valid?

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

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

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

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

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

##### Pipeline Register M                 #########################
wordsig M_stat 'ex_mem_curr->status'     # Instruction status
wordsig M_icode 'ex_mem_curr->icode'    # Instruction code
wordsig M_ifun  'ex_mem_curr->ifun'     # Instruction function
wordsig M_valA  'ex_mem_curr->vala'      # Source A value
wordsig M_dstE 'ex_mem_curr->deste'     # Destination E register ID
wordsig M_valE  'ex_mem_curr->vale'      # ALU E value
wordsig 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 ##########################
```

```
wordsig m_valM 'mem_wb_next->valm'      # valM generated by memory
wordsig m_stat 'mem_wb_next->status'    # stat (possibly modified to be SADR)

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

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

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

## What address should instruction be fetched at
word 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 ipopq into two cycles, need to be able to
## modify value of icode,
## so that it will be IPOP2 when fetched for second time.
word f_icode = [
        imem_error : INOP;
        ## Can detected refetch of ipopq, since now have
        ## IPOPQ as icode for instruction in decode.
        imem_icode == IPOPQ && D_icode == IPOPQ : IPOP2;
        1: imem_icode;
];

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

# Is instruction valid?
bool instr_valid = f_icode in
        { INOP, IHALT, IRRMOVQ, IIRMOVQ, IRMMOVQ, IMRMOVQ,
          IOPQ, IJXX, ICALL, IRET, IPUSHQ, IPOPQ, IPOP2 };
```

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

# Does fetched instruction require a regid byte?
bool need_regids =
        f_icode in { IRRMOVQ, IOPQ, IPUSHQ, IPOPQ,
                     IPOP2,
                     IIRMOVQ, IRMMOVQ, IMRMOVQ };

# Does fetched instruction require a constant word?
bool need_valC =
        f_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ, IJXX, ICALL };

# Predict next value of PC
word f_predPC = [
        f_icode in { IJXX, ICALL } : f_valC;
        ## 1W: Want to refetch popq one time
        # (on second time f_icode will be IPOP2). Refetch popq
        f_icode == IPOPQ : f_pc;
        1 : f_valP;
];

################ Decode Stage ######################################

## W1: Strategy.  Decoding of popq rA should be treated the same
## as would iaddq $8, %rsp
## Decoding of pop2 rA treated same as mrmovq -8(%rsp), rA

## What register should be used as the A source?
word d_srcA = [
        D_icode in { IRRMOVQ, IRMMOVQ, IOPQ, IPUSHQ  } : D_rA;
        D_icode in { IPOPQ, IRET } : RRSP;
        1 : RNONE; # Don't need register
];

## What register should be used as the B source?
word d_srcB = [
        D_icode in { IOPQ, IRMMOVQ, IMRMOVQ  } : D_rB;
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET, IPOP2 } : RRSP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
word d_dstE = [
        D_icode in { IRRMOVQ, IIRMOVQ, IOPQ} : D_rB;
```

```
        D_icode in { IPUSHQ, IPOPQ, ICALL, IRET } : RRSP;
        1 : RNONE;  # Don't write any register
];

## What register should be used as the M destination?
word d_dstM = [
        D_icode in { IMRMOVQ, IPOP2 } : D_rA;
        1 : RNONE;  # Don't write any register
];

## What should be the A value?
## Forward into decode stage for valA
word 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
];

word 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
word aluA = [
        E_icode in { IRRMOVQ, IOPQ } : E_valA;
        E_icode in { IIRMOVQ, IRMMOVQ, IMRMOVQ } : E_valC;
        E_icode in { ICALL, IPUSHQ, IPOP2 } : -8;
        E_icode in { IRET, IPOPQ } : 8;
        # Other instructions don't need ALU
];

## Select input B to ALU
word aluB = [
        E_icode in { IRMMOVQ, IMRMOVQ, IOPQ, ICALL,
                     IPUSHQ, IRET, IPOPQ, IPOP2 } : E_valB;
        E_icode in { IRRMOVQ, IIRMOVQ } : 0;
        # Other instructions don't need ALU
];
```

```
## Set the ALU function
word alufun = [
	E_icode == IOPQ : E_ifun;
	1 : ALUADD;
];

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

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

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

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

## Select memory address
word mem_addr = [
	M_icode in { IRMMOVQ, IPUSHQ, ICALL, IMRMOVQ, IPOP2 } : M_valE;
	M_icode in { IRET } : M_valA;
	# Other instructions don't need address
];

## Set read control signal
bool mem_read = M_icode in { IMRMOVQ, IPOP2, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVQ, IPUSHQ, ICALL };

## Update the status
word m_stat = [
	dmem_error : SADR;
	1 : M_stat;
];

################ Write back stage ##################################

## 1W: For this problem, we introduce a multiplexor that merges
## valE and valM into a single value for writing to register port E.
## DO NOT CHANGE THIS LOGIC
## Merge both write back sources onto register port E
## Set E port register ID
word w_dstE = [
	## writing from valM
```

```
        W_dstM != RNONE : W_dstM;
        1: W_dstE;
];

## Set E port value
word w_valE = [
        W_dstM != RNONE : W_valM;
        1: W_valE;
];

## Disable register port M
## Set M port register ID
word w_dstM = RNONE;

## Set M port value
word w_valM = 0;

## Update processor status
word Stat = [
        W_stat == SBUB : SAOK;
        1 : W_stat;
];

################ Pipeline Register Control #########################

# 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 =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOP2 } &&
         E_dstM in { d_srcA, d_srcB } ||
        # 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 =
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOP2 } &&
         E_dstM in { d_srcA, d_srcB };

bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Cnd) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        # 1W: Changed Load/Use condition
        !(E_icode in { IMRMOVQ, IPOP2 } && E_dstM in { d_srcA, d_srcB }) &&
          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) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVQ, IPOP2 } &&
         E_dstM in { d_srcA, d_srcB};

# 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-1w-ans.hcl

Problem 4.59 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.

Chapter 5: Optimizing Program Performance

Problem 5.13 Solution:

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

A. See Figure 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.

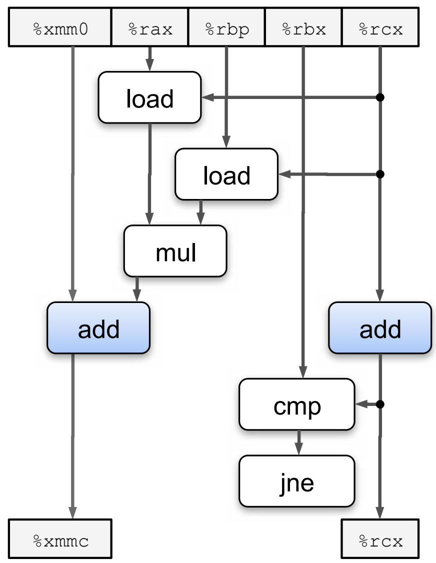


Figure 2: **Data-flow for function `inner4`.** The multiplication operation is not on any critical path.

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 5.14 Solution:

This problem gives practice applying loop unrolling.

```
/* Inner Product.  6 X 1 unrolling */
void inner_u6x1(vec_ptr u, vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(u);
    long limit = length-5;
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t sum = (data_t) 0;
    /* Do 6 elements at a time */
    for (i = 0; i < limit; i+=6) {
        sum = sum + udata[i] * vdata[i]
            + udata[i+1] * vdata[i+1]
            + udata[i+2] * vdata[i+2]
            + udata[i+3] * vdata[i+3]
            + udata[i+4] * vdata[i+4]
            + udata[i+5] * vdata[i+5];
    }
    /* Finish off any remaining elements */
    for (; i < length; i++) {
        sum = sum + udata[i] * vdata[i];
```

```
    }
    *dest = sum;
}
```

A. We must perform two loads per element to read values for `udata` and `vdata`. Since the machine has two functional units that can do this operation, the CPE cannot be less than 1.0.

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

Problem 5.15 Solution:

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

```
/* Inner Product.  6 X 6 unrolling */
void inner_u6x6(vec_ptr u, vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(u);
    long limit = length-5;
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t sum0 = (data_t) 0;
    data_t sum1 = (data_t) 0;
    data_t sum2 = (data_t) 0;
    data_t sum3 = (data_t) 0;
    data_t sum4 = (data_t) 0;
    data_t sum5 = (data_t) 0;
    /* Do 6 elements at a time */
    for (i = 0; i < limit; i+=6) {
        sum0 = sum0 + udata[i]   * vdata[i];
        sum1 = sum1 + udata[i+1] * vdata[i+1];
        sum2 = sum2 + udata[i+2] * vdata[i+2];
        sum3 = sum3 + udata[i+3] * vdata[i+3];
        sum4 = sum4 + udata[i+4] * vdata[i+4];
        sum5 = sum5 + udata[i+5] * vdata[i+5];
    }
    /* Finish off any remaining elements */
    for (; i < length; i++) {
        sum0 = sum0 + udata[i] * vdata[i];
    }
    *dest = (sum0 + sum1 + sum2) + (sum3 + sum4 + sum5);
}
```

Both the pair of load units, as well as the single floating-point adder impose throughput lower bounds of 1.00.

Problem 5.16 Solution:

This gives students a chance to experiment with reassociation.

```
/* Inner Product.  6 X 1a unrolling */
void inner_u6x1a(vec_ptr u, vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(u);
    long limit = length-5;
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t sum = (data_t) 0;
    /* Do 6 elements at a time */
    for (i = 0; i < limit; i+=6) {
        sum = sum +
            ((udata[i]   * vdata[i]
              + udata[i+1] * vdata[i+1])
             + (udata[i+2] * vdata[i+2]
                + udata[i+3] * vdata[i+3]))
            + ((udata[i+4] * vdata[i+4])
               + udata[i+5] * vdata[i+5]);
    }
    /* Finish off any remaining elements */
    for (; i < length; i++) {
        sum = sum + udata[i] * vdata[i];
    }
    *dest = sum;
}
```

Problem 5.17 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;
    long 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++;
    }

    /* Step through 2*sizeof(word) characters at a time */
    slong = (unsigned long *) schar;
    while (cnt < n-2*sizeof(word)+1) {
        *slong = word;
        *(slong+1) = word;
        cnt += 2*sizeof(word);
        slong += 2;
    }

    /* Finish off single characters */
    schar = (unsigned char *) slong;
    while (cnt < n) {
        *schar++ = (unsigned char) c;
        cnt++;
    }
    return s;
}
```

Problem 5.18 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 1.26 A version with 8-way unrolling achieved a CPE of 1.11.

```
/* Reassociate */
double poly4(double a[], double x, long degree)
{
    long i;
    double result = a[0];
    double x2 = x*x;
    double x3 = x2*x;
    double x4 = x2*x2;
    double xpwr = x;
    for (i = 1; i <= degree-3; i+=4) {
        double nxpwr = x4 * xpwr;
        result += ((a[i] + x*a[i+1]) + (x2*a[i+2] + x3*a[i+3])) * xpwr;
        xpwr = nxpwr;
    }
    for (; i <= degree; i++) {
```

```
        result += a[i] * xpwr;
        xpwr = x * xpwr;
    }
    return result;
}
```

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 1.26, and the 8-way version has a CPE of 1.08.

```
/* Accumulate multiple values in parallel */
double poly4x(double a[], double x, long degree)
{
    long i;
    double result0 = a[0];
    double result1 = 0;
    double result2 = 0;
    double result3 = 0;
    double x4 =      (x*x)*(x*x);
    double xpwr = x;
    for (i = 1; i <= degree-3; i+=4) {
        double nxpwr = x4 * xpwr;
        result0 += a[i] * xpwr;
        result1 += a[i+1] * xpwr;
        result2 += a[i+2] * xpwr;
        result3 += a[i+3] * xpwr;
        xpwr = nxpwr;
    }
    result0 += x*(result1 + x*(result2 + x*result3));
    for (; i <= degree; i++) {
        result0 += a[i] * xpwr;
        xpwr = x * xpwr;
    }
    return result0;
}
```

The final method is a variant on Horner’s method with reassociation. This function gave a CPE of 2.01.

```
/* Apply Horner’s method */
double poly4h(double a[], double x, long degree)
{
    long i;
    double x2 = x*x;
    double x3 = x2*x;
    double x4 = x2*x2;
    double result = a[degree];
    for (i = degree-1; i >= 3; i-=4)
        result = ((a[i-3] + x*a[i-2]) + (x2*a[i-1] + x3*a[i])) + x4*result;
```

```
    for (; i >= 0; i--)
        result = a[i] + x*result;
    return result;
}
```

Problem 5.19 Solution:

Prefix sum allows for interesting optimizations. Making use of reassociation requires some cleverness. It's difficult to balance the competing goals of maximizing parallelism and avoiding the throughput lower bound of the floating-point adder. Increasing parallelism generally requires introducing more addition operations.

Our best results were obtained for a version that does a small amount of unrolling and breaks the sequential dependency between successive iterations by reassociating the summation. This involves increasing the number of additions beyond the minimum.

Here is the version that unrolls by 2:

```
/* Compute prefix sum of vector a */
/* Use 2-way loop unrolling + lookahead */
void psum2a(float a[], float p[], long cnt)
{
    long i;
    /* last_val holds p[i-1] */
    float last_val;
    last_val = p[0] = a[0];
    for (i = 1; i < cnt-1; i+=2) {
        float a01 = a[i] + a[i+1];
        p[i]   = last_val + a[i];
        p[i+1] = last_val = last_val + a01;
    }
    /* Finish remaining elements */
    for (; i < cnt; i++) {
        p[i] = last_val  = last_val + a[i];
    }
}
```

It achieves a CPE of 2.00, even though it seems like it should be 1.50, due to a both a latency bound of $3.00/2$, and a throughput bound caused by performing three additions to compute two elements.

Here is the version that unrolls by 3:

```
/* Compute prefix sum of vector a */
/* Use 3-way loop unrolling + lookahead */
void psum3a(float a[], float p[], long cnt)
{
    long i;
    /* last_val holds p[i-1] */
    float last_val;
    last_val = p[0] = a[0];
    for (i = 1; i < cnt-2; i+=3) {
```

```
        p[i]   = last_val + a[i];
        float a01   = a[i] + a[i+1];
        p[i+1] = last_val + a01;
        float a012  = a01  + a[i+2];
        p[i+2] = last_val = last_val + a012;
    }
    /* Finish remaining elements */
    for (; i < cnt; i++) {
        p[i] = last_val = last_val + a[i];
    }
}
```

This code yields a CPE of 1.67, due to the throughput bound caused by performing five additions to compute three elements.

Unrolling by larger factors does not help, because it only introduces more additions.

Chapter 6: The Memory Hierarchy

Problem 6.22 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 6.23 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 6.24 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 optimal 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 6.25 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 6.26 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 between 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 6.27 Solution:

This is an inverse cache indexing problem (akin to Problem 6.29) 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 6.28 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 6.29 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 6.30 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 6.31 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 6.32 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 6.33 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 6.34 Solution:

This problem is tougher than it looks. The approach is similar to the solution to Problem 6.17. 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 6.35 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 6.36 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 6.37 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 6.38 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 6.39 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 6.40 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 6.41 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 6.42 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 6.43 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 6.43 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:

Solution approach: Use the `mountain` program to generate a graph similar to Figure 6.42, 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 6.45 Solution:

No solution provided.

Problem 6.46 Solution:

No solution provided.

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	`m.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`	`.bss`

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:

```
static double x;

void f() {
      x = -0.0;
}
```

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 `foo6.o` overrides the weak symbol associated with the variable `main` in `bar6.o`. Thus, the reference to `main` in `p2` resolves to the value of strong symbol `main`, which in this case is the address of the first byte of the compiled function `main`, which happens to be `0x48`.

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:

The solution outline is identical to Problem 7.5:

A. 0x4004f8 − 4 − 0x4004ea = 0xa

B. 0x400500 − 4 − 0x4004da = 0x22

Problem 7.13 Solution:

A. On our system, `libc.a` has 1492 members and `libm.a` has 444 members:

```
linux>  ar -t /usr/lib/x86_64-linux-gnu/libc.a | wc -l
1496
linux> ar -t /usr/lib/x86_64-linux-gnu/libm.a | wc -l
444
```

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`), the dynamic linker (`ld-linux-x86-64.s` and the vDSO (`linux-vdso.so.1`). The vDSO (virtual dynamic shared object) is linked into every ELF executable and provides some hooks to speed up frequently-used system calls.

```
linux> ldd /usr/bin/gcc
    linux-vdso.so.1 =>  (0x00007fff5abff000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f57507af000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f5750b8b000)
```

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 8.17. 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 a similar process graph as the program in Figure 8.17. 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 outcome (each column is a possible outcome):

```
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
```

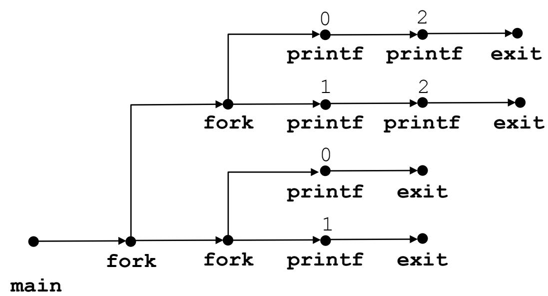
Problem 8.17 Solution:

Inspection of the process graph in the solution to Problem 8.4 shows that 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 key is to draw the process graph:



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

```
#include "csapp.h"

int main(int argc, char **argv) {
  Execve("/bin/ls", argv, environ);
  exit(0);
}
```

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

```
#include "csapp.h"

int mysystem(char *command)
{
    pid_t pid;
    int status;

    if (command == NULL)
        return -1;

    if ((pid = fork()) == -1)
        return -1;

    if (pid == 0) { /* child */
        char *argv[4];
        argv[0] = "sh";
        argv[1] = "-c";
        argv[2] = command;
        argv[3] = NULL;
        execve("/bin/sh", argv, environ);
        exit(-1); /* control should never reach here */
    }

    /* parent */
    while (1) {
        if (waitpid(pid, &status, 0) == -1) {
            if (errno != EINTR) /* restart waitpid if interrupted */
                return -1;
        }
        else {
            if (WIFEXITED(status))
                return WEXITSTATUS(status);
            else
                return status;
        }
    }
}
```

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

```
#include "csapp.h"

#define NCHILDREN 2

int main()
{
    int status, i;
    pid_t pid;
    char buf[MAXLINE];

    for (i = 0; i < NCHILDREN; i++) {
        pid = Fork();
        if (pid == 0)  /* child */
            /* child attempts to modify first byte of main */
            *(char *)main = 1;
    }

    /* parent waits for all children to terminate */
    while ((pid = wait(&status)) > 0) {
        if (WIFEXITED(status))
            printf("child %d terminated normally with exit status=%d\n",
                   pid, WEXITSTATUS(status));
        else
            if (WIFSIGNALED(status)) {
                sprintf(buf, "child %d terminated by signal %d",
                        pid, WTERMSIG(status));
                psignal(WTERMSIG(status), buf);
            }
    }
    if (errno != ECHILD)
        unix_error("wait error");

    return 0;
}
```

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

```
#include "csapp.h"

static sigjmp_buf env;

static void handler(int sig)
{
    Alarm(0);
    siglongjmp(env, 1);
}

char *tfgets(char *s, int size, FILE *stream)
{
    Signal(SIGALRM, handler);

    Alarm(5);
    if (sigsetjmp(env, 1) == 0)
        return(Fgets(s, size, stream)); /* return user input */
    else
        return NULL;  /* return NULL if fgets times out */
}

int main()
{
    char buf[MAXLINE];

    while (1) {
        bzero(buf, MAXLINE);
        if (tfgets(buf, sizeof(buf), stdin) != NULL)
            printf("read: %s", buf);
        else
            printf("timed out\n");
    }
    exit(0);
}
```

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 for instructors.

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

```
#include "csapp.h"

/*
 * mmapwrite - uses mmap to modify a disk file
 */
void mmapwrite(int fd, int len)
{
    char *bufp;

    bufp = Mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    bufp[0] = 'J';
}
```

```

/* mmapwrite driver */
int main(int argc, char **argv)
{
    int fd;
    struct stat stat;

    /* check for required command line argument */
    if (argc != 2) {
        printf("usage: %s <filename>\n", argv[0]);
        exit(0);
    }

    /* open the input file and get its size */
    fd = Open(argv[1], O_RDWR, 0);
    fstat(fd, &stat);
    mmapwrite(fd, stat.st_size);
    exit(0);
}
```

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	16
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

```
static void *find_fit(size_t asize)
{
    char *oldrover;

    oldrover = rover;

    /* search from the rover to the end of list */
    for ( ; GET_SIZE(HDRP(rover)) > 0; rover = NEXT_BLKP(rover))
        if (!GET_ALLOC(HDRP(rover)) && (asize <= GET_SIZE(HDRP(rover))))
            return rover;

    /* search from start of list to old rover */
    for (rover = heap_listp; rover < oldrover; rover = NEXT_BLKP(rover))
        if (!GET_ALLOC(HDRP(rover)) && (asize <= GET_SIZE(HDRP(rover))))
            return rover;

    return NULL;  /* no fit found */
}
```

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

```
static void *coalesce(void *bp)
{
    int prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    int next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && next_alloc) {            /* Case 1 */
        return bp;
    }

    else if (prev_alloc && !next_alloc) {      /* Case 2 */
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size,0));
    }

    else if (!prev_alloc && next_alloc) {      /* Case 3 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
```

```
    }

    else {                                          /* Case 4 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
            GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        bp = PREV_BLKP(bp);
    }

    /* Make sure the rover isn't pointing into the free block */
    /* that we just coalesced */
    if ((rover > (char *)bp) && (rover < NEXT_BLKP(bp)))
        rover = bp;

    return bp;
}
```

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.

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

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    char buf[MAXBUF];

    while((n = Rio_readn(STDIN_FILENO, buf, MAXBUF)) != 0)
        Rio_writen(STDOUT_FILENO, buf, n);
    exit(0);
}
```

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

```
#include "csapp.h"

int main (int argc, char **argv)
{
    struct stat stat;
    char *type, *readok;
    int size;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <fd>\n", argv[0]);
        exit(0);
    }
    Fstat(atoi(argv[1]), &stat);
    if (S_ISREG(stat.st_mode))     /* Determine file type */
        type = "regular";
    else if (S_ISDIR(stat.st_mode))
        type = "directory";
    else if (S_ISCHR(stat.st_mode))
```

```
            type = "character device";
        else
            type = "other";

        if ((stat.st_mode & S_IRUSR)) /* Check read access */
            readok = "yes";
        else
            readok = "no";

        size = stat.st_size; /* check size */

        printf("type: %s, read: %s, size=%d\n",
               type, readok, size);

        exit(0);
}
```

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

```
#include "csapp.h"

int main(int argc, char **argv)
{
    int n;
    rio_t rio;
    char buf[MAXLINE];

    if ((argc != 1) && (argc != 2) ) {
```

```
        fprintf(stderr, "usage: %s <infile>\n", argv[0]);
        exit(1);
    }

    if (argc == 2) {
        int fd;
        if ((fd = Open(argv[1], O_RDONLY, 0)) < 0) {
            fprintf(stderr, "Couldn't read %s\n", argv[1]);
            exit(1);
        }
        Dup2(fd, STDIN_FILENO);
        Close(fd);
    }

    Rio_readinitb(&rio, STDIN_FILENO);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0)
        Rio_writen(STDOUT_FILENO, buf, n);
    exit(0);
}
```

code/io/cpfile2-ans.c

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 11.34.

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.

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

```
#include "csapp.h"

void *thread(void *vargp);

int main(int argc, char **argv)
{
    pthread_t *tid;
    int i, n;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <nthreads>\n", argv[0]);
        exit(0);
    }
    n = atoi(argv[1]);
    tid = Malloc(n * sizeof(pthread_t));

    for (i = 0; i < n; i++)
        Pthread_create(&tid[i], NULL, thread, NULL);
    for (i = 0; i < n; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp)
{
    printf("Hello, world!\n");
    return NULL;
```

```
}
```

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

```
#include "csapp.h"

/* Global variables */
int readcount;      /* Initially = 0 */
sem_t mutex, w, wg; /* All initially = 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcount++;
        if (readcount == 1)
            P(&w);
        V(&mutex);
```

```

        /* Critical section: */
        /* Reading happens   */

        P(&mutex);
        readcount--;
        if (readcount == 0)
            V(&w);
        V(&mutex);
    }
}

void writer(void)
{
    while (1) {
        P(&wg);
        P(&w);

        /* Critical section: */
        /* Writing happens   */

        V(&w);
        V(&wg);
    }
}
```

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

```
#include "csapp.h"

#define N 10

/* Global variables */
sem_t sem;    /* Initially = N */
sem_t wmutex; /* Initially = 1 */

void reader(void)
{
    while (1) {
        P(&sem);

```

```
        /* Critical section: */
        /* Reading happens   */

        V(&sem);
    }
}

void writer(void)
{
    int i;

    while (1) {
        P(&wmutex);
        for (i=0; i<N; i++)
            P(&sem);
        V(&wmutex);

        /* Critical section: */
        /* Writing happens   */

        for (i=0; i<N; i++)
            V(&sem);
    }
}
```

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

```
#include "csapp.h"
#define TIMEOUT 5

static sigjmp_buf env; /* buffer for non-local jump */
```

```
static char *str;      /* global to keep gcc -Wall happy */

/* SIGCHLD signal handler */
static void handler(int sig)
{
    Wait(NULL);
    siglongjmp(env, 1);
}

char *tfgets(char *s, int size, FILE *stream)
{
    pid_t pid;

    str = NULL;

    Signal(SIGCHLD, handler);

    if ((pid = Fork()) ==  0) { /* child */
        Sleep(TIMEOUT);
        exit(0);
    }
    else {  /* parent */
        if (sigsetjmp(env, 1) == 0) {
            str = fgets(s, size, stream);
            Kill(pid, SIGKILL);
            pause();
        }
        return str;
    }
}
```

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

```
#include "csapp.h"

#define TIMEOUT 5

char *tfgets(char *s, int size, FILE *stream)
{
    struct timeval tv;
    fd_set rfds;
    int retval;

    FD_ZERO(&rfds);
    FD_SET(0, &rfds);
```

```

    /* Wait for 5 seconds for stdin to be ready */
    tv.tv_sec = 5;
    tv.tv_usec = 0;
    retval = select(1, &rfds, NULL, NULL, &tv);
    if (retval)
        return fgets(s, size, stream);
    else
        return NULL;
}
```

code/conc/tfgets-select-ans.c

Problem 12.33 Solution:

A version of `tfgets` based on threads:

code/conc/tfgets-thread-ans.c

```
#include "csapp.h"
#define TIMEOUT 5

void *fgets_thread(void *vargp);
void *sleep_thread(void *vargp);

char *returnval;  /* fgets output string */
typedef struct {  /* fgets input arguments */
    char *s;
    int size;
    FILE *stream;
} args_t;

char *tfgets(char *str, int size, FILE *stream)
{
    pthread_t fgets_tid, sleep_tid;
    args_t args;

    args.s = str;
    args.size = size;
    args.stream = stdin;
    returnval = NULL;
    Pthread_create(&fgets_tid, NULL, fgets_thread, &args);
    Pthread_create(&sleep_tid, NULL, sleep_thread, &fgets_tid);
    Pthread_join(fgets_tid, NULL);
    return returnval;
}

void *fgets_thread(void *vargp)
{
    args_t *argp = (args_t *)vargp;
    returnval = fgets(argp->s, argp->size, stdin);
```

```
    return NULL;
}

void *sleep_thread(void *vargp)
{
    pthread_t fgets_tid = *(pthread_t *)vargp;
    Pthread_detach(Pthread_self());
    Sleep(TIMEOUT);
    pthread_cancel(fgets_tid);
    return NULL;
}
```

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.