# CS 15-213, Fall 1998

# Practice Exam 1

**Instructions:**

- This exam is OPEN BOOK. You may use any books or notes you like.

## Problem 1. (4 points):

*Sometimes a bit is just a bit.*

Consider representing numbers with a 3-bit word size. We will consider four different encodings:

**Unsigned** Only nonnegative numbers are encoded.

**Two's Complement** Signed integers are encoded in two's complement form.

**Biased** Signed integers are encoded with a bias of 4.

**Sign Magnitude** The most significant bit indicates the sign, while the remaining bits indicate the absolute value.

Fill in the following table giving the numeric value for all eight possible bit patterns for each of the four encodings.

| Bits | Unsigned | Two's Comp. | Biased | Sign Magnitude |
|---|---|---|---|---|
| 000 | | | | |
| 001 | | | | |
| 010 | | | | |
| 011 | | | | |
| 100 | | | | |
| 101 | | | | |
| 110 | | | | |
| 111 | | | | |

## Problem 2. (16 points):

*Q: What's purple and commutes? A: An abelian grape.*

In this problem, all numbers are 32-bit. Signed numbers are represented in 2's complement format. Recall from lecture that the promotion rules tell us signed integers will be promoted to unsigned when necessary. The variables used in this problem are declared and initialized as follows:

```
int x = foo();
int y = bar();
unsigned ux = (unsigned) x;
unsigned uy = (unsigned) y;
```

For each of the expressions below, circle T if it is always true, or F if it can be false. If you circle F, give a concise counterexample.

A. T F `(x < 0) --> ((x * 2) < 0)`

B. T F `ux >= 0`

C. T F `(x & 7 == 7) --> (x << 30) < 0`

D. T F `(x < 0) --> ((x << 1) != 0)`

E. T F `ux > -1`

F. T F `(x > y) --> (-x < -y)`

G. T F `(x * x) >= 0`

H. T F `(x > 0) && (y > 0) --> (x + y) >= 0`

## Problem 3. (8 points):

This problem tests your understanding of Alpha assembly language. The following C function returns the largest element in an array. It assumes the array has length $\geq 1$.

```
long get_largest (long* array, long length)
{
  long i;
  long largest = array[0];
  for (i = 1; i < length; i++)
    if (array[i] > largest)
      largest = array[i];
  return largest;
}
```

The corresponding machine code (output from dis -h) is as follows:

```
  0x0:  42203da3 cmple r17, 0x1, r3
  0x4:  a4100000 ldq r0, 0(r16)
  0x8:  f460000a bne r3, 0x34
  0xc:  47e03402 bis r31, 0x1, r2
  0x10: 20300008 lda r1, 8(r16)
  0x14: 5fff041f cpys $f31,$f31,$f31  # this is a nop
  0x18: a6010000 ldq r16, 0(r1)
  0x1c: 40403402 addq r2, 0x1, r2
  0x20: 20210008 lda r1, 8(r1)
  0x24: 401009a4 cmplt r0, r16, r4
  0x28: 449004c0 cmovne r4, r16, r0
  0x2c: 405109a4 cmplt r2, r17, r4
  0x30: f49ffff9 bne r4, 0x18
  0x34: 6bfa8001 ret r31, (r26), 1
```

A. Identify the inner loop for this function (i.e. the only loop). What are the addresses of the first and last instructions in this loop?

B. Give the valid C language expression that best describes the contents of the following registers. A valid C language expression is one that can be parsed by a C compiler, e.g., `i+1`, `v[0]`, etc.

(a) r0:

(b) r1:

(c) r2:

(d) r16 (initial value):

(e) r16 (value during execution of the inner loop):

(f) r17:

(g) r31:

This next problem will test your understanding of stack frames. It is based on the following C function which recursively (and inefficiently!) calculates the value of the nth Fibonacci number:

```
static int fib (long n)
{
  if (n == 0) {
    return (0);
  } else if (n == 1) {
    return (1);
  } else {
    return (fib(n-1) + fib(n-2));
  }
}
```

This yields the following machine code (using dis -h):

```
  0x0:  23deffe0        lda     sp, -32(sp)
  0x4:  47ff0400        bis     r31, r31, r0
  0x8:  b53e0008        stq     r9, 8(sp)
  0xc:  47f00409        bis     r31, r16, r9
  0x10: b75e0000        stq     r26, 0(sp)
  0x14: f6000001        bne     r16, 0x1c
  0x18: c3e0000c        br      r31, 0x4c
  0x1c: 45203810        xor     r9, 0x1, r16
  0x20: f6000003        bne     r16, 0x30
  0x24: 47e03400        bis     r31, 0x1, r0
  0x28: c3e00008        br      r31, 0x4c
  0x2c: 47ff041f        bis     r31, r31, r31
  0x30: 41203530        subq    r9, 0x1, r16
  0x34: d35ffff2        bsr     r26, 0(r31)
  0x38: b01e0010        stl     r0, 16(sp)
  0x3c: 41205530        subq    r9, 0x2, r16
  0x40: d35fffef        bsr     r26, 0(r31)
  0x44: a13e0010        ldl     r9, 16(sp)
  0x48: 41200000        addl    r9, r0, r0
  0x4c: a75e0000        ldq     r26, 0(sp)
  0x50: a53e0008        ldq     r9, 8(sp)
  0x54: 23de0020        lda     sp, 32(sp)
  0x58: 6bfa8001        ret     r31, (r26), 1
```

## Problem 4. (7 points):

A. How many bytes are allocated for fib's stack frame each time it is called?

B. At what stack frame byte offset is the result of fib(n-1) stored?

C. Why is it necessary to store this value on the stack frame?

D. At what stack frame offset is the contents of r9 stored?

E. Why is it necessary to store this value on the stack frame?

F. What (if anything) is stored at offset 0 of the stack frame? A register number is insufficient here, please describe what is contained in the register.

G. What (if anything) is stored at offset 24 of the stack frame? A register number is insufficient here, please describe what is contained in the register.

In the following problem, suppose you were implementing the portion of a compiler that generates Alpha code sequences for expressions of the form $K \cdot x$ where $K$ is a constant and $x$ is a long integer. You want to make use of the (possibly scaled) add and subtract instructions, since these instructions execute much faster than the multiply instruction.

Here are the rules:

- The original value $x$ is in register `$1`
- The result value $K \cdot x$ should be placed in register `$2`.
- You may not alter any registers other than `$2`, including `$1`.
- You may only use the following instructions: `addq`, `subq`, `s4addq`, `s4subq`, `s8addq`, `s8subq`.
- You can make use of the fact that `$31` is always 0.

## Problem 5. (8 points):

Show the code sequences you would use for the the values of $K$ given below. You should use a minimum number of instructions for each case. None requires more than two instructions.

A. $K = 3$:

B. $K = 18$:

C. $K = -7$

D. $K = 55$

## Problem 6. (6 points):

Consider the following data type definitions:

```
typedef struct {
  short k;
  char c;
  short l;
  int i;
} struct1;
```

```
typedef union {
  struct1 s;
  double d;
} union1;
```

A. Using the template below (allowing a maximum of 24 bytes), indicate the allocation of data for a structure of type `struct1`. Mark off and label the areas for each individual element (there are 4 of them). Cross hatch the parts that are allocated, but not used (to satisfy alignment). Clearly indicate the right hand boundary of the data structure.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

B. How many bytes are allocated for a variable of type `struct1`?

C. What alignment is required for a variable of type `struct1`?

D. A union provides the possibility to use one data type in two (or more) different ways. Below we have provided two templates, one for each way in which the data type `union1` can be used. Label the templates in the same way as you did above.

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                                                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

E. How many bytes are allocated for a variable of type `union1`?

F. What alignment is required for a variable of type `union1`?

## Problem 7. (6 points):

This problem tests your understanding of how `for` loops in C relate to Alpha assembly code.
Consider the following Alpha assembly code for a procedure `foo()`, which was generated using `dis -h`:

```
        foo:
  0x0:  47ff0400        bis     r31, r31, r0
  0x4:  46100402        bis     r16, r16, r2
  0x8:  40510521        subq    r2, r17, r1
  0xc:  40410da1        cmple   r2, r1, r1
  0x10: f4200007        bne     r1, 0x30
  0x14: 42110530        subq    r16, r17, r16
  0x18: 47ff041f        bis     r31, r31, r31
  0x1c: 47ff041f        bis     r31, r31, r31
  0x20: 40003400        addq    r0, 0x1, r0
  0x24: 40405522        subq    r2, 0x2, r2
  0x28: 40500da1        cmple   r2, r16, r1
  0x2c: e43ffffc        beq     r1, 0x20
  0x30: 6bfa8001        ret     r31, (r26), 1
```

Based on the assembly code above, fill in the blanks below in its corresponding C source code. (Note: you may only use symbolic variables from the source code in your expressions below—do *not* use register names.)

```
long int foo(long int x, long int y)
{
  long int i;
  long int result = 0;

  for (i = _________ ; i ______________ ; i = ___________ ) {
    result++;
  }
  return result;
}
```

You are charged with maintaining a large C program, and you come across the following code:

```
typedef struct {
  int left;
  a_struct a[CNT];
  int right;
} b_struct;

void test(long int i, b_struct *bp)
{
  int n = bp->left + bp->right;
  a_struct *ap = &bp->a[i];
  ap->x[3] = n;
}
```

Unfortunately, the '.h' file defining the compile-time constant CNT and the structure a_struct are in files for which you do not have access privileges. Fortunately, you have access to a '.o' version of code, which you are able to disassemble with the dis program. Here is the disassembly of the procedure test:

```
0x0:  a0510000        ldl     r2, 0(r17)
0x4:  a03100e0        ldl     r1, 224(r17)
0x8:  40410002        addl    r2, r1, r2
0xc:  42100570        s4subq  r16, r16, r16
0x10: 42011650        s8addq  r16, 0x8, r16
0x14: 42300411        addq    r17, r16, r17
0x18: b0510014        stl     r2, 20(r17)
0x1c: 6bfa8001 X       ret     r31, (r26), 1
```

**Problem 8. (8 points):**

Using your reverse engineering skills, deduce the following:

A. The size of the structure a_struct.

B. The alignment requirements for a_struct.

C. The value of CNT.

D. A diagram of the complete layout for a_struct. [You may not know the names of all of the fields for this structure, but you can determine much of the information about their types.] Mark off the different fields and indicate as much information as you can. Clearly indicate the end of the structure by crossing off any unused bytes. The diagram allows a maximum of 36 bytes.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                                         |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

**C's Structures, Unions and Pointers**

The following problems test your understanding of C's structures, unions and pointers. You have just joined the FooBar software company. For your first task, you need to reconstruct a lost C source file—an ugly job. Luckily, you have the corresponding `.h` file and the object code. From these clues reconstruct the rest of the source file below.

## Problem 9. (6 points):

As a starting point for your project, fill out the field offsets and lengths for each of the following structures and the union.

```
/* hack.h */
struct s1 {
  int t;
  int y;
  int l;
  struct s1 *h;
};
```

| field | offset | length |
|---|---|---|
| t | | |
| y | | |
| l | | |
| h | | |

```
struct s2 {
  union u1 *k;
  union u1 *m;
  struct s3 *r;
  struct s2 *c;
};
```

| field | offset | length |
|---|---|---|
| k | | |
| m | | |
| r | | |
| c | | |

```
struct s3 {
  struct s1 *a;
  struct s2 *o;
};
```

| field | offset | length |
|---|---|---|
| a | | |
| o | | |

```
union u1 {
  int e;
  struct s3 *d;
};
```

| field | offset | length |
|---|---|---|
| e | | |
| d | | |

## Problem 10. (12 points):

Given each block of assembly code, fill in the corresponding C code to recreate `hack.c`. There should be no casts in your answers.

```
p1:
ldq $1, 16($16)
ldq $2, 0($1)
ldl $0, 4($2)
ret $31, ($26), 1

int p1(struct s2 *c) {
  return c->
}

p2:
ldq $1, 8($16)
ldq $2, 24($1)
ldq $3, 0($2)
ldl $0, 0($3)
ret $31, ($26), 1

int p2(struct s3 *c) {
  return c->
}

p3:
ldq $1, 8($16)
ldq $2, 0($1)
ldq $3, 0($2)
ldq $4, 16($3)
ldl $0, 8($4)
ret $31, ($26), 1

int p3(struct s2 *a) {
  return a->
}
```