EE 4720 Homework 1 Solution Due: 15 February 2002

At the time this was assigned computer accounts and solution templates were not available. If they become available they can be used for the solution, either way a paper submission is acceptable.

Problem 1: The value computed by the program below approaches π. Re-write the program in MIPS assembler. The code should execute quickly. Assume that all integer instructions take one cycle, floating-point divides take ten cycles, floating-point compares take one cycle, and all other floating-point instructions, including conversion, take four cycles. *Note: As originally assigned only the time for divides and adds was given.* Make changes to the code to improve speed (possibly using an integer for `i` or even using both an integer and double). Do not use a different technique for computing π.

```
int
main(int argv, char **argc)
{
  double i;
  double sum = 0;

  for(i=1; i<50000000;)
    {
      sum = sum + 4.0 / i;   i += 2;
      sum = sum - 4.0 / i;   i += 2;
    }

  printf("After %d iterations sum = %.8f\n", (int)(i-1)/2, sum);

  return 0;
}
```

The code appears on the next page. But first, here are some reminders based on submitted solutions:

Double-precision values must be placed in an even-numbered fp register.

When speed is a goal delay slots should be filled!

Immediates are limited to 16 bits.

```
# Solution to problem 1.
        .data
ITERATIONS:
        .double 50000000.0
MSG:
        .asciiz "After %/f0/.0f iterations sum = %/f2/.8f\n";

        .text
        .globl __start
__start:
        addi $t1, $0, 1
        mtc1 $t1, $f0
        cvt.d.w $f0, $f0   # f0 -> i
        add.d $f12, $f0, $f0    # f12 <- 2 (constant)
        add.d $f14, $f12, $f12  # f14 <- 4 (constant)
        la $t2, ITERATIONS
        ldc1 $f16, 0($t2)       # f16 -> number of iterations.
        sub.d $f2, $f0, $f0     # f2  -> sum <- 0 initialize
LOOP:
        div.d $f4, $f14, $f0    # 4.0 / i
        add.d $f0, $f0, $f12    # i+=2
        add.d $f2, $f2, $f4     # sum += 4.0/i
        div.d $f4, $f14, $f0    # 4.0 / i
        add.d $f0, $f0, $f12    # i+=2
        c.lt.d $f0, $f16
        bc1t LOOP
        sub.d $f2, $f2, $f4     # sum -= 4.0/i

        div.d $f16, $f16, $f12

        addi $v0, $0, 11
        la $a0, MSG
        syscall

        addi $v0, $0, 10
        syscall
```

Problem 2: The program below is used to generate a password based on the outcome of several rolls of a twenty-sided die. The program was compiled using the Sun Workshop Compiler 5.0 targeting SPARC V7 (`-xarch=v7`) and SPARC V9 (`-xarch=v8plus`, code which can run on a V9 processor with a 32-bit OS), the output of the compiler is shown for the `for` loop.

Use the V8 architecture manual to look up V7 instructions, available at `http://www.ece.lsu.edu/ee4720/samv8.pdf`; the V9 architecture manual is available at `http://www.ece.lsu.edu/ee4720/samv9.pdf`.

Here are a few useful facts about SPARC:

Register names for SPARC are: `%g0-%g7` (global), `%l0-%l7` (local), `%i0-%i7` (input), `%o0-%o7` (output), and `%f0-%f31` (floating point). Registers `%fp` (frame pointer) and `%sp` are aliases for `%i6` and `%o6`, respectively. Register `%g0` is a zero register.

Local variables (the only kind used in the code fragment shown) are stored in memory at some offset from the stack pointer (in `%sp`). For example, `ldd [%sp+96],%f0` loads a local variable into register `%f0`.

All V7 and V8 integer registers are 32 bits. V9 registers are 64 bits but with the v8plus option only the 32 lower bits are used.

Unlike MIPS and DLX, the last register in an assembly language instruction is the destination. For example, `add %g1, %g2, %g3`, puts the sum of `g1` and `g2` in register `g3`.

Like MIPS, SPARC branches are delayed. Unlike MIPS, some delayed branches are annulled, indicated with a ",a" in the mnemonic. In an annulled branch the instruction in the delay slot is executed if and *only if* the branch is taken.

(*a*) For each compilation, identify which registers are used for which program variables.

See comments in the code on the following pages.

(*b*) For each instruction used in the V9 version of the code but not in the V7 version, explain what it does and how it improves execution over the V7 version.

`udivx`: performs 64-bit unsigned integer division. It's probably faster than the divide routine called in the V7 code.

`mulx`: performs 64-bit multiplications, used for finding the remainder. It's probably faster than the remainder routine called in the V7 code.

Because the V9 uses divx and mulx it makes no procedure calls in the loop and so there is no need to save and restore the floating-point registers.

`fbul,a,pt` and `fbge,a,pt`: branches with prediction hints. Can speed execution if predictions correct and heeded by hardware.

```
!   SOLUTION.
!
!   Register   Variable
!   f0:        bits_per_letter
!   f30:       bits
!   {i1,i2}:   seed (seed is 64 bits so two 32-bit registers used).
!   i0:        pw_ptr
!
!   Note:  fp registers are caller-saved (and caller-restored).
!
!   Also see comments.
!
!   Compiled with -xarch=v7
!
!   32        !  for( ; bits >= bits_per_letter;  bits -= bits_per_letter )

/* 0x010c   32 */ ldd [%sp+96],%f0    ! Load f0 from stack.
                       .L900000118:
/* 0x0110   32 */ fcmped %f30,%f0 ! Compare bits, bits_per_letter
/* 0x0114      */ nop
/* 0x0118      */ fbul .L77000009      ! Branch if less than
/* 0x011c      */ or %g0,0,%o2
                       .L900000116:
!   33        !    {
!   34        !      *pw_ptr++ = 'a' + seed % 26;
/* 0x0120   34 */ or %g0,%i2,%o1     ! Move seed to {o0,o1}
/* 0x0124      */ or %g0,%i1,%o0     ! for procedure call.
/* 0x0128      */ or %g0,26,%o3     ! Move 26 to o3 for call.
/* 0x012c      */ call __urem64 ! params =  %o0 %o1 %o2 %o3 ! Result =  %o0
/* 0x0130      */ std %f30,[%sp+104]  ! Caller save.
/* 0x0134      */ add %o1,97,%g2      ! {seed%26} + 'a'
/* 0x0138      */ stb %g2,[%i0]       ! *pw_ptr={'a'+seed%26}

!   35        !      seed = seed / 26;
/* 0x013c   35 */ or %g0,%i1,%o0     ! Move seed to {o0,o1}
/* 0x0140      */ or %g0,0,%o2       ! and 26 to {o2,o3}
/* 0x0144      */ or %g0,26,%o3      ! for procedure call.
/* 0x0148      */ call __udiv64 ! params =  %o0 %o1 %o2 %o3 ! Result =  %o0
/* 0x014c      */ or %g0,%i2,%o1
/* 0x0150      */ ldd [%sp+96],%f0    ! Caller restore
/* 0x0154   34 */ add %i0,1,%i0       ! pw_ptr++
/* 0x0158   35 */ or %g0,%o0,%i1     ! seed = {seed/26}
/* 0x015c      */ ldd [%sp+104],%f30  ! Caller restore
/* 0x0160      */ fsubd %f30,%f0,%f30  ! bits -= bits_per_letter
/* 0x0164      */ fcmped %f30,%f0
/* 0x0168      */ or %g0,%o1,%i2     ! seed = {seed/26}
/* 0x016c      */ fbge .L900000116
/* 0x0170      */ or %g0,0,%o2
                       .L77000009:

!   36        !    }
```

```
!   SOLUTION.
!
!   Register   Variable
!   f4:        bits_per_letter
!   f8:        bits
!   o0,g2:     seed
!   i0:        pw_ptr
!
!   Note:  fp registers are still caller-saved (and caller-restored)
!          but since there are no calls in the loop there is no need
!          to save and restore fp regs.  (The V7 code used calls for
!          64-bit integer division and remainder, V9 has 64-bit
!          divide and multiply instructions it can use instead.)
!
!   Also see comments.
!

!   Compiled With -xarch=v8plus
!
!   32       !  for( ; bits >= bits_per_letter;  bits -= bits_per_letter )

/* 0x00e8   32 */ fcmped %fcc0,%f8,%f4
                       .L900000117:
/* 0x00ec   32 */ fbul,a,pt %fcc0,.L900000115
/* 0x00f0      */ stb %g0,[%i0]

!   33       !    {
!   34       !      *pw_ptr++ = 'a' + seed % 26;

/* 0x00f4   34 */ udivx %o0,26,%g2   ! %g2 = seed / 26
                       .L900000114:
/* 0x00f8   34 */ mulx %g2,26,%g3   ! g3 = 26 ( seed / 26 )
/* 0x00fc      */ sub %o0,%g3,%g3  ! g3 = seed % 26

!   35       !      seed = seed / 26;

/* 0x0100   35 */ or %g0,%g2,%o0  ! o0 = seed / 26
/* 0x0104      */ fsubd %f8,%f4,%f8  ! bits -= bits_per_letter
/* 0x0108   34 */ add %g3,97,%g3   ! 'a' + seed % 26
/* 0x010c      */ stb %g3,[%i0]    ! *pw_ptr = 'a'
/* 0x0110      */ add %i0,1,%i0    ! pw_ptr++
/* 0x0114   35 */ fcmped %fcc1,%f8,%f4
/* 0x0118      */ fbge,a,pt %fcc1,.L900000114
/* 0x011c      */ udivx %o0,26,%g2   ! g2 = seed / 26
                       .L77000009:

!   36       !    }
```