CS-740 Homework #1 Sample Answers

1. Warm Up

1A:

Given a, b and K such that $0 \le a, b < K$, and $s = (a + b) \bmod K$.

Case 1 : $0 \le a + b < K$, therefore, $s = a + b$, but $a \ge 0$ and $b \ge 0$. Therefore, $s \ge b$ and $s \ge a$.

Case 2 : $K \le a + b < 2K$, therefore, $s = a + b - K < K$, but $a - K < 0$ and $b - K < 0$. Therefore, $s < a$ and $s < b$.

1B:

If you macro is:

#define ADD(Carry, Sum, a, b) { Sum=a+b; Carry=Sum<a; },

I will subtract 2 points. The reason is that you didn't consider the argument aliasing, when Sum and a are the same. For example, if you call this macro by ADD(Carry,a,a,b), then Carry will always be set to 0. The correct macro definition should be:

#define ADD(Carry, Sum, a, b) { unsigned a1 = a; Sum=a+b; Carry=Sum<a1; }.

The corresponding machine instructions on MIPS are : (move $2, $4), (addu $4,$4,$5) and (sltu $2,$4,$2).

2. Exact Multiplication

2A: Shift_Add Multiplier

Here is a sample program for shift_add version:

```
/* initialize the double word <hi,lo> to 0, at each loop add x to hi if
   the lsb of y is 1. Then shift the double word <hi,lo> one bit to the
   right with the carry generated from the addition.
*/
void mult_step(unsigned x, unsigned y, unsigned *hip, unsigned *lop)
{
  unsigned lo = 0, hi = 0,carry;
  int i;
  for ( i=0; i < WSIZE; i++, y>>=1) {
     if ( y & 0x1) {
        ADD(carry,hi,hi,x);
        lo = (lo >> 1) + (hi << (WSIZE -1));
        hi = (hi >> 1) + (carry << (WSIZE -1));
     } else {
        lo = (lo >> 1) + (hi << (WSIZE -1));
        hi = hi >> 1;
     }
  }
  *lop = lo;
  *hip = hi;
}
```

2B: Half_Word Multiplier

This version of the multiplication routine is based upon the fact that:
$(a2^n + b) \times (c2^n + d) = ac2^{2n} + (ad + bc)\,2^n + bd$, where n is the half-word size.

```
#define HALF_SIZE (WSIZE/2)
#define DOUBLE_SIZE (WSIZE << 1)
#define LOW_MASK ((1 << HALF_SIZE) -1)
#define HIGH_MASK (~LOW_MASK)
#define HIGH_BITS(X)  ((X & HIGH_MASK) >> HALF_SIZE)
#define LOW_BITS(X) (X & LOW_MASK)

void mult_half(unsigned x, unsigned y, unsigned *hip, unsigned *lop)
{ unsigned  x_lo = LOW_BITS(x);
  unsigned  x_hi = HIGH_BITS(x);
  unsigned  y_lo = LOW_BITS(y);
  unsigned  y_hi = HIGH_BITS(y);
  unsigned  lolo = x_lo * y_lo;
  unsigned  hilo = x_hi * y_lo;
  unsigned  lohi = x_lo * y_hi;
  unsigned  hihi = x_hi * y_hi;
  unsigned  mid_hi, mid_lo, mid_carry, sum, carry, lo;
  ADD(mid_carry, sum, hilo, lohi);
  mid_lo = LOW_BITS(sum) << HALF_SIZE;
  mid_hi = HIGH_BITS(sum);
  ADD(carry, lo, lolo, mid_lo);
  *lop = lo;
  *hip = hihi + mid_hi + carry + (mid_carry << HALF_SIZE);
}
```

2C: Assembly_Code Multiplier

This version of multiplication routine uses the MIPS instructions are the multu, mfhi and mflo instructions:

```
mult_asm:
       .frame $sp,0,$31     # vars= 0, regs= 0/0, args = 0, extra= 0
       .mask  0x00000000,0
       .fmask 0x00000000,0
       multu  $4,$5         #unsigned multiplication and store the product in <hi,lo>.
       mflo   $4             # move the low part of result to $4
       mfhi   $5             # move the high part of result to $5
       #nop
       #nop
       sw     $4,0($7)
       .set   noreorder
       .set   nomacro
       j      $31
       sw     $5,0($6)
       .set   macro
       .set   reorder
       .end   mult_asm
```

3. Performance Evaluation

The following performance is obtained from 12MHZ DECstation 2100. The assembly version is

TABLE 1. Performance of three different multiplication routines.

Routine	Microseconds	Clock cycles
mult_step	34.72	417
mult_half	4.92	59
mult_asm	1.09	14

the fastest one. The shift_add version is the slowest one, because it has to performs a lot of addition and shift instructions. The mult_half is about 4 times slower than the assembly version, because mult_half must perform 4 multiplications.

4. Setjmp and longjmp

The following table outlines the structure of the jump_buf started from the address of $a0:

TABLE 2. The mapping of the jump_buf

Offset	Contents
8	$ra
12	magic word
76--104	$s0, $s1, ..., $s7
128	$sp
132	$s8
232--276	$f20,$f21,....,$f31
280	FPU control

General speaking, setjmp saves it's own return address, the stack pointer and all callee save registers (including general purpose, FPU and FPU control registers). It calculates a magic number and stores it in the jump_buf and then returns 0.

First, longjmp checks the correctness of the magic number. If it is not correct, then it calls error handle routine. Otherwise, it restores all the registers(including the stack pointer and return address) saved by setjmp. It then puts its own second argument into the return value register and returns to the setjmp's return address saved setjmp. Any stack that was allocated between the call to setjmp and the call to longjmp is ignored and will be overwritten in future code..

Here is the commented, disassembled code for setjmp:

```
0x4005d0 <_setjmp>:sw $ra,8($a0)          #store return address in offset 8.
0x4005d4 <_setjmp+4>:sw $sp,128($a0)      #store stack pointer in offset 128
0x4005d8 <_setjmp+8>:sw $s0,76($a0)       #store callee save integer registers
0x4005dc <_setjmp+12>:sw $s1,80($a0)
0x4005e0 <_setjmp+16>:sw $s2,84($a0)
0x4005e4 <_setjmp+20>:sw $s3,88($a0)
0x4005e8 <_setjmp+24>:sw $s4,92($a0)
0x4005ec <_setjmp+28>:sw $s5,96($a0)
0x4005f0 <_setjmp+32>:sw $s6,100($a0)
0x4005f4 <_setjmp+36>:sw $s7,104($a0)
0x4005f8 <_setjmp+40>:sw $s8,132($a0)
0x4005fc <_setjmp+44>:swc1 $f20,232($a0)  #store callee save FPU registers.
0x400600 <_setjmp+48>:swc1 $f21,236($a0)
0x400604 <_setjmp+52>:swc1 $f22,240($a0)
0x400608 <_setjmp+56>:swc1 $f23,244($a0)
0x40060c <_setjmp+60>:swc1 $f24,248($a0)
0x400610 <_setjmp+64>:swc1 $f25,252($a0)
0x400614 <_setjmp+68>:swc1 $f26,256($a0)
0x400618 <_setjmp+72>:cfc1 $v0,$f31       #Move FPU control register to $v0
0x40061c <_setjmp+76>:cfc1 $v0,$f31       #Do it again.
0x400620 <_setjmp+80>:swc1 $f27,260($a0)  #store more FPU registers.
0x400624 <_setjmp+84>:swc1 $f28,264($a0)
0x400628 <_setjmp+88>:sw $v0,280($a0)     #store FPU control register in offset 280
```

```
0x40062c <_setjmp+92>:lui $v0,44269                  #calculate the magic word.
0x400630 <_setjmp+96>:ori $v0,$v0,47838
0x400634 <_setjmp+100>:sw $v0,12($a0)               #store the magic word in offset 12.
0x400638 <_setjmp+104>:swc1 $f29,268($a0)           #store more FPU registers.
0x40063c <_setjmp+108>:swc1 $f30,272($a0)
0x400640 <_setjmp+112>:swc1 $f31,276($a0)
0x400644 <_setjmp+116>:jr $ra
0x400648 <_setjmp+120>:move $v0,$zero               #return to the return address with 0.
```

Here is the commented, disassembled code for longjmp:

```
0x40064c <_longjmp>:lw $v0,12($a0)                  #load the magic word from jump_buf in offset 12.
0x400650 <_longjmp+4>:lui $at,44269                 #caculate the magic word.
0x400654 <_longjmp+8>:ori $at,$at,47838
0x400658 <_longjmp+12>:bne $v0,$at,0x4006cc <_longjmp+128> #compare them. if not equal, got error routine.
0x40065c <_longjmp+16>:nop
0x400660 <_longjmp+20>:lw $v0,280($a0)              #load save FPU control register to $v0.
0x400664 <_longjmp+24>:lw $ra,8($a0)                #retore the saved registers.
0x400668 <_longjmp+28>:lw $sp,128($a0)
0x40066c <_longjmp+32>:lw $s0,76($a0)
0x400670 <_longjmp+36>:lw $s1,80($a0)
0x400674 <_longjmp+40>:lw $s2,84($a0)
0x400678 <_longjmp+44>:lw $s3,88($a0)
0x40067c <_longjmp+48>:lw $s4,92($a0)
0x400680 <_longjmp+52>:lw $s5,96($a0)
0x400684 <_longjmp+56>:lw $s6,100($a0)
0x400688 <_longjmp+60>:lw $s7,104($a0)
0x40068c <_longjmp+64>:lw $s8,132($a0)
0x400690 <_longjmp+68>:lwc1 $f20,232($a0)
0x400694 <_longjmp+72>:lwc1 $f21,236($a0)
0x400698 <_longjmp+76>:lwc1 $f22,240($a0)
0x40069c <_longjmp+80>:lwc1 $f23,244($a0)
0x4006a0 <_longjmp+84>:lwc1 $f24,248($a0)
0x4006a4 <_longjmp+88>:lwc1 $f25,252($a0)
0x4006a8 <_longjmp+92>:lwc1 $f26,256($a0)
0x4006ac <_longjmp+96>:lwc1 $f27,260($a0)
0x4006b0 <_longjmp+100>:lwc1 $f28,264($a0)
0x4006b4 <_longjmp+104>:lwc1 $f29,268($a0)
0x4006b8 <_longjmp+108>:lwc1 $f30,272($a0)
0x4006bc <_longjmp+112>:lwc1 $f31,276($a0)
0x4006c0 <_longjmp+116>:ctc1 $v0,$f31               #restore FPU control register.
0x4006c4 <_longjmp+120>:jr $ra                      # return to the return address saved by setjmp.
0x4006c8 <_longjmp+124>:move $v0,$a1                #set the return values to the second argument
0x4006cc <_longjmp+128>:jal 0x401010 <longjmperror> # go to the error handler.
0x4006d0 <_longjmp+132>:nop
0x4006d4 <_longjmp+136>:jal 0x401040 <abort>
0x4006d8 <_longjmp+140>:nop
0x4006dc <_longjmp+144>:nop
```

In these two codes, there are a double reads(cfc1) from the FPU control register in setjmp, and a single write(ctc1) to the FPU control register in longjmp. The double reads is to make sure that FPU clears its pipeline and sets the control register correctly. While executing the ctc1 instruction, the FPU pipeline is already clear. Therefore, this single write make sure that FPU control register value is set to the desired value. I will subtract 1 points, if you didn't explain this.