# Lab 9: Using the MPLAB simulator to debug MIPS assembly programs

Lina Mi

ID:@01377283

**Purpose:**

1) Get to know the instruction set of MIPS assembly language and be able to programming with assembly language;
2) Be familiar with Newton's Method and using it to compute the square root of a number with both integer part and fractional part;
3) Be familiar with MPLAB simulator;
4) Be able to use MPLAB to edit, compile and run MIPS assembly program, be able to debugging MIPS assembly program by monitor the content of registers.

**Content:**

Using Newton's Method, which is a series of approximation, to compute the square root of a number with both integer part and fractional part. The relevant formula for computing the next approximation, an+1 for $\sqrt{x}$, given an is

$$a_{n+1} = (a_n + x/a_n)/2$$

During programming, we need to figure out whether and how to handle the hexadecimal point, and write a division function(here we choose to place point between 17$^{th}$ bit and 16$^{th}$ bit for 32 bits number, namely the number has same amount of bits in integer part as that of bits in fractional part ). To avoid complexity of division of fixed-point number, we transfer division to multiplication, namely we calculate the reciprocal of denominator first, then times the numerator with the reciprocal of denominator.

We need to write two functions, one is sqrt, which will take the value for x in $\$a_0$, and return the fixed-point square root in $\$v_0$, the other one named frecip, which will take the value for a number in $\$a_0$ and return the reciprocal of the number in $\$v_1$.

1) MIPS assembly code

The assembly code used to calculate the square root of fixed-point hexadecimal number is shown in the following file:

```
/*
  #include <xc.h>
```

```
#define IOPORT_BIT_7 (1 << 7)
    .global sqrt    /* define all global symbols here */
   /* .text
    /* define which section (for example "text")
     * does this portion of code resides in. Typically,
     * all your code will reside in .text section as
     * shown below.
     */
   /* .set noreorder
    /* This is important for an assembly programmer. This
     * directive tells the assembler not to optimize
     * the order of the instructions as well as not to insert
     * 'nop' instructions after jumps and branches.
     */
/*********************************************************************
     * main()
     * This is where the PIC32 start-up code will jump to after initial
     * set-up.
     *********************************************************************/
   /*.ent main /* directive that marks symbol 'main' as function in the ELF
            * output
            */
/*sqrt:
   /* Call function to clear bit relevant to pin 7 of port A.
    * The 'jal' instruction places the return address in the $ra
    * register.
    */
   /*li $a0, 0x00000004
   add $a1,$zero,$zero
```

```
  add $a2, $zero, $zero
  li $s0,0x00000000
  add $v0, $zero, $zero
newton:
   add $a1, $zero, $a0
   add $a2, $zero, $a0

  jal  divide:



  ori     a0, $0, IOPORT_BIT_7
  jal     mPORTAClearBits
  nop
  /* endless                                                    loop */
/*endless:
  j endless
  nop
  .end main /* directive that marks end of 'main' function and its
            * size in the ELF output
            */
/*********************************************************************
 * mPORTAClearBits(int bits)
 * This function clears the specified bits of IOPORT A.
 *
 * pre-condition: $ra contains return address
 * Input: Bit mask in $a0
 * Output: none
 * Side effect: clears bits in IOPORT A
```

```
**********************************************************************/
  /*.ent mPORTAClearBits
mPORTAClearBits:
  /* function prologue - save registers used in this function
   * on stack and adjust stack-pointer
   */
  /*addiu   sp, sp, -4
  sw      s0, 0(sp)
  la      s0, LATACLR
  sw      a0, 0(s0)      /* clear specified bits */
  /* function epilogue - restore registers used in this function
   * from stack and adjust stack-pointer
   */
  /*lw      s0, 0(sp)
  addiu   sp, sp, 4
  /* return to caller */
  /*jr      ra
  /*nop
  .end mPORTAClearBits*/

/*   n implementation of a fixed point computation of
 reciprocal using 32bit Q16.16 fixed bit numbers.

 The code finds x such that 1/x - D = 0 using Newton's method:

  x_{n+1} = x_n (2 - x_n D)

  which is a simplification of x_{n+1} = x_n - (1/x_n -D)/(-1/x^2)
```

```
  The first approximation x_0 is computed  by using the position of the leading one  bit relative to the
binary point.  The approximate reciprocal is  a one bit which is reflected around bit 16 (which is the
position of  the value one in Q16.16 fixed point) from the position of the highest one.


 Stephen Taylor

 November 4, 2016
*/


    equ  BPfollows, 0x10      # position of binary point; only tested for value 16


   #include <p32xxxx.h>


     .global main    /* define all global symbols here */

     .text

     /* define which section (for example "text")

      * does this portion of code resides in. Typically,

      * all your code will reside in .text section as

      * shown below.

      */

     .set noreorder

     /* This is important for an assembly programmer. This

      * directive tells the assembler not to optimize

      * the order of the instructions as well as not to insert

      * 'nop' instructions after jumps and branches.

      */

   /*******************************************************************

    * main()

    * This is where the PIC32 start-up code will jump to after initial

    * set-up.
```

```
    ************************************************************************/
/* all macro arguments for times should be register names */
/* multiply source1 by source2 and store result in dest */
/* where all are in Q16.16 format, with the integer part in the high
   sixteen bits and the binary fraction in the low sixteen bits.
   The macro actually assumes Q(32-BPfollows).(BPfollows)
   but only the value 16 is tested.
*/

 .macro times dest source1 source2
    .set noat #disable assembler use of $at so I can use it here
        mult \source1,\source2
        mfhi \dest
        sll \dest,\dest,BPfollows
        mflo $at
        srl $at,(0x20-BPfollows)
        and $at,(1<<BPfollows)-1
        or \dest,$at
    .set at # reenable assembler use  of $at
        .endm

#push and pop macros
.macro push reg
    addi $sp,$sp,-4
    sw \reg,0($sp)
.endm

.macro pop reg
    lw \reg,0($sp)
```

```
    addi $sp,$sp,4
.endm

    .ent main /* directive that marks symbol 'main' as function in the ELF
            * output
            */
    main:
    push $ra

#       first a little test scaffold
main1:
    jal recipTest
     nop
    jal sqrtTest    # test sqrt
     nop
    # test quadform
    li $a0,0x10000
    li $a1,0x30000
    li $a2,0x20000
    /*jal quadform*/
     nop

    b main1
     nop
# this return is just extra baggage...
    pop $ra
    jr $ra
      nop
```

```
recipTest:
    push $ra
    li $a0,0x1ffff
    jal frecip
     nop

    pop $ra
    jr $ra
     nop

sqrtTest:
    push $ra
         nop

        li $a0,0x10000   # sqrt should be 1.0, 0x10000
        jal sqrt
         nop

        li $a0,0x20000   # 2.  sqrt should be 17xxx?
        jal sqrt
         nop
        li $a0,0x190000 # 2.  sqrt should be 17xxx?
        jal sqrt
         nop
        li $a0,0x90000   # 2.  sqrt should be 17xxx?
        jal sqrt
         nop

        li $a0,0x30000   #3.   sqrt should be 1Bxxx?
```

```
        jal sqrt
         nop

    pop $ra
        jr $ra
         nop

/* compute reciprocal in Q16.16 format.
   description of algorithm above
   argument in $a0
   reciprocal returned in $v0 -- used as xn in the algorithm
   $v1 is used as xn1
   $t0, $t1, $t8 used as temporary variables, clobbered
*/

frecip:
        # check for negative:
        lui $t0,0x8000   # look for the leftmost 1 bit in $a0
        and $t1,$a0,$t0
        beqz $t1,fr1      #don't worry, it's positive
         nop
        move $t8,$ra              # save return address, since jal changes it
        jal fr1                   # get reciprocal in $v0
         sub $a0,$zero,$a0        # negate the negative number (this is delay slot
        jr $t8              # return using saved return address
         sub $v0,$zero,$v0        # negate the returned value (in delay slot)


fr1:
```

```
        # check for zero -- which has no reciprocal
        bnez $a0,fr2
     lui $v0,0x7FFF    # harmless if we branch; executed anyway. I broke li into
                    #lui, ori to load $v0 with one fewer instructions
    # here if $a0 == 0.  Return 0x7FFFFFFF,
    ori $v0,0xFFFF        # an approximation of positive infinity
        jr $ra
         nop

        # find first approximation
fr2:
        lui $t0,0x4000              #first non-negative bit
        li $v0,4              # corresponding reciprocal
        sub $v1,$zero,$zero       # set up xn1 for fr4 loop now.

fr3:
        and $t1,$a0,$t0# found it yet?
        bnez $t1,fr4      # found it
     nop          # don't put sll into delay slot ...
        sll $v0,$v0,1
        b fr3
         srl $t0,$t0,1      # adjust (in delay slot)

fr4:
        # this is the newton-raphson loop
        # xn1 = xn * (2 - $a0 * xn)
        times $t0,$v0,$a0
        lui $t1,0x2         # this will be 2.0 in Q16.16 format.  Would
        sub $t0,$t1,$t0 #  have to be fixed if BPfollows changed.
```

```
        times $v0,$t0,$v0


        bne $v0,$v1,fr4 #loop until $v0 == $v1
        add $v1, $zero, $v0        #use add instead of move to fit delay slot
        jr $ra     # done, answer in $v0
       nop
sqrt:
        move $t6,$ra
        move $a3,$a0
    # this is the newton-raphson loop
        # xn1 = (xn + $a3 * $v0) * 1/2
        lui $a1,0x1
s1:
        move $a0,$a1
        jal frecip
        nop
        times $t7,$v0,$a3
        add $t7,$t7,$a1
        li $t8,0x8000
        times $t7,$t7,$t8
        bne $t7,$a1,s1 #loop until $v0 == $v1
        add $a1, $zero, $t7        #use add instead of move to fit delay slot
        jr $t6     # done, answer in $v0
       nop
      end main
```

2) Debugging

The following pictures are captured during debugging the program.

Square root of 3

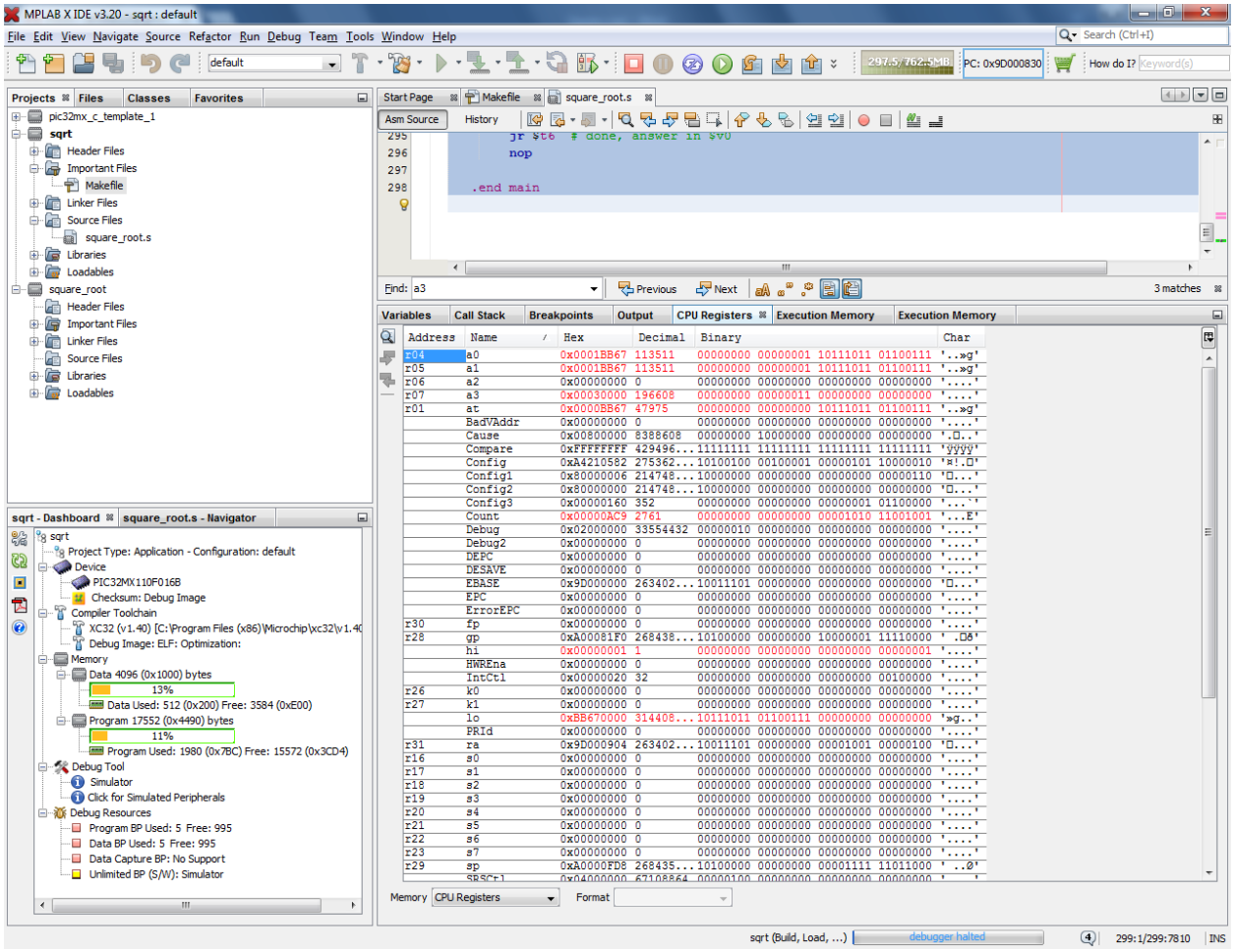


Figure 1 Program output when number is 3(values are expressed in hexadecimal)

The output of program for number 3 is 0x0001BB67, since the point is between 4$^{th}$ and 5$^{th}$ digits, the integer part is 1(decimal), the fractional part equals $11*16^{-1}+11*16^{-2}+6*16^{-3}+7*16^{-4}$=0.732040(decimal), we know that the square root of 3 is approximate to 1.732, so the program result is valid and correct.

Square root of 1

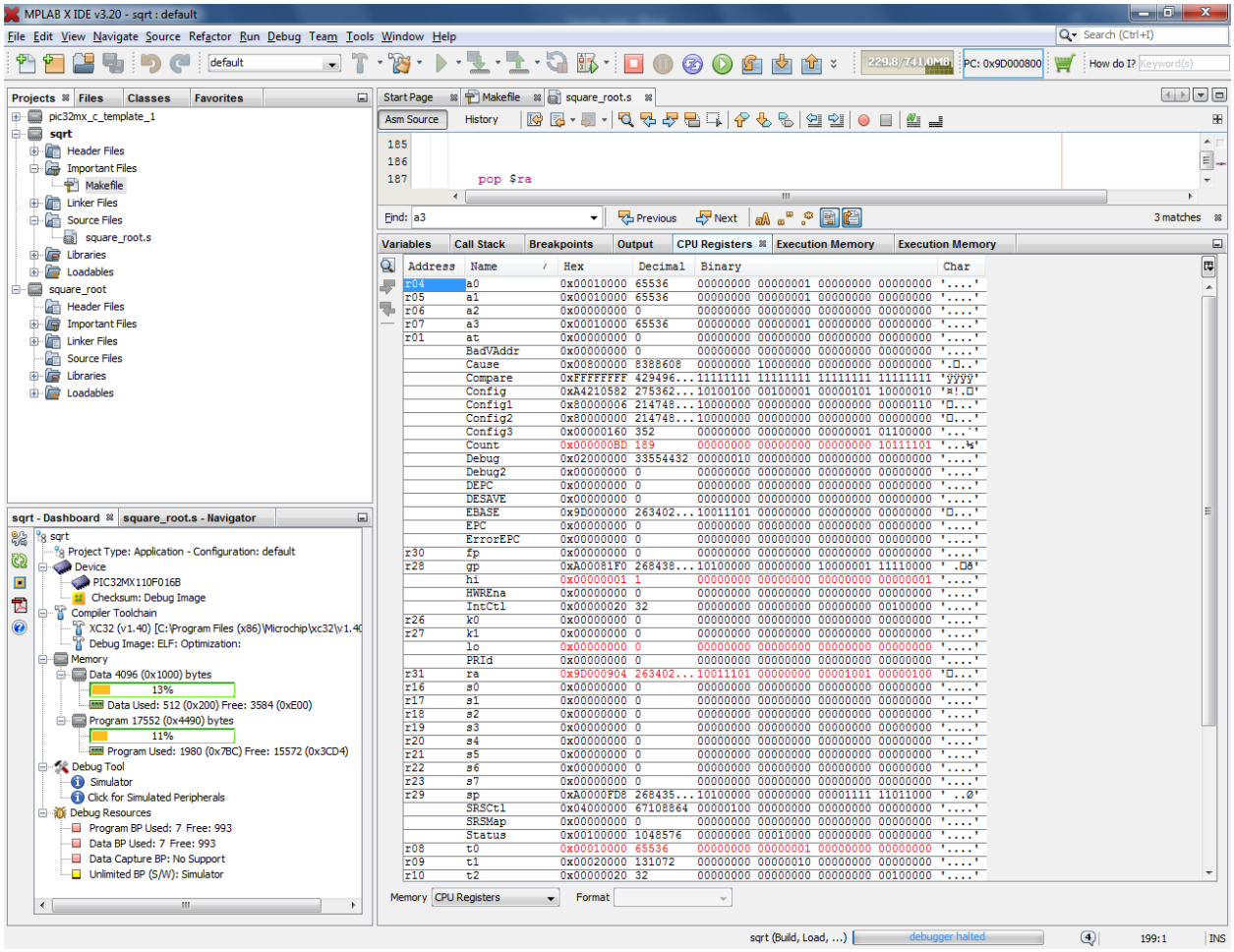


Figure 2 Program output when number is 1(values are expressed in hexadecimal)

The output of program is 1 for number 1, we know that the square root of 1 is 1, so the result is correct.

From the debugging results, we know that the previous program is valid and correct.