

Using the MPLAB simulator to debug MIPS assembly programsSubmitted by: Piyusha Jahagirdar

In this lab, we wrote a code in MIPS to compute the square root of a number.

SQURT.S

```

/*
#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 $x_{n+1} = x_n - (1/x_n - D)/(-1/x_n^2)$

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 <p32xxx.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

*/

frecep:

```
# 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
    #  $x_{n1} = x_n * (2 - a_0 * x_n)$ 
    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 BP follows 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
    #  $x_{n1} = (x_n + a_3 * v_0) * 1/2$ 
    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
```

Calculating the square root of 1

The screenshot shows the MPLAB X IDE v3.20 interface. The main window displays the assembly source code for a MIPS program. The code is as follows:

```
185  
186  
187 pop $ra
```

The CPU registers window at the bottom shows the following values:

Register	Address	Name	Hex	Decimal	Binary	Char
r0	a0		0x00010000	65536	00000000 00000000 00000000 00000000
r05	a1		0x00010000	65536	00000000 00000000 00000000 00000000
r06	a2		0x00000000	0	00000000 00000000 00000000 00000000
r07	a3		0x00010000	65536	00000000 00000000 00000000 00000000
r01	at		0x00000000	0	00000000 00000000 00000000 00000000
	BadVAddr		0x00000000	0	00000000 00000000 00000000 00000000
	Cause		0x00800000	8388608	00000000 10000000 00000000 00000000
	Compare		0xFFFFFFFF	4294967295	11111111 11111111 11111111 11111111	yyyy
	Config		0x04210582	275362	00100100 00100001 00000101 10000010	..D..
	Config1		0x00000006	214748	10000000 00000000 00000000 00000110
	Config2		0x00000000	214748	10000000 00000000 00000000 00000000
	Config3		0x00000160	352	00000000 00000000 00000001 01100000
	Count		0x00000000	189	00000000 00000000 00000000 10111101
	Debug		0x02000000	33554432	00000010 00000000 00000000 00000000
	Debug2		0x00000000	0	00000000 00000000 00000000 00000000
	DEPC		0x00000000	0	00000000 00000000 00000000 00000000
	DESAVE		0x00000000	0	00000000 00000000 00000000 00000000
	EBASE		0x9D000000	263402	10011101 00000000 00000000 00000000
	EPC		0x00000000	0	00000000 00000000 00000000 00000000
	ErrorEPC		0x00000000	0	00000000 00000000 00000000 00000000
	fp		0x00000000	0	00000000 00000000 00000000 00000000
r30	gp		0x000081F0	268438	10100000 00000000 10000001 11110000	..D..
r28	h1		0x00000001	1	00000000 00000000 00000000 00000001
	HWREna		0x00000000	0	00000000 00000000 00000000 00000000
	IntCtl		0x00000020	32	00000000 00000000 00000000 00100000
r26	k0		0x00000000	0	00000000 00000000 00000000 00000000
r27	k1		0x00000000	0	00000000 00000000 00000000 00000000
	lo		0x00000000	0	00000000 00000000 00000000 00000000
	PRId		0x00000000	0	00000000 00000000 00000000 00000000
r31	ra		0x9D000904	263402	10011101 00000000 00001001 00000100
r16	a0		0x00000000	0	00000000 00000000 00000000 00000000
r17	a1		0x00000000	0	00000000 00000000 00000000 00000000
r18	a2		0x00000000	0	00000000 00000000 00000000 00000000
r19	a3		0x00000000	0	00000000 00000000 00000000 00000000
r20	a4		0x00000000	0	00000000 00000000 00000000 00000000
r21	a5		0x00000000	0	00000000 00000000 00000000 00000000
r22	a6		0x00000000	0	00000000 00000000 00000000 00000000
r23	a7		0x00000000	0	00000000 00000000 00000000 00000000
r29	sp		0x04000000	67108864	00000100 00000000 00000000 00000000
	SRSctl		0x00000000	0	00000000 00000000 00000000 00000000
	SRSMap		0x00000000	0	00000000 00000000 00000000 00000000
	Status		0x00100000	1048576	00000000 00010000 00000000 00000000
r08	t0		0x00010000	65536	00000000 00000001 00000000 00000000
r09	t1		0x00020000	131072	00000000 00000010 00000000 00000000
r10	t2		0x00000020	32	00000000 00000000 00000000 00100000

The bottom status bar shows the program is running: `sqrt (Build, Load, ...)` and the debugger is halted.