2.1: Simple loops in MIPS assembly language

Topics:

loops in MIPS
stepping through a MIPS program
tracing a MIPS program with breakpoints
adding output statements for debugging

Introduction:

In your CSc 256 lectures, you should have seen by now how we can rewrite loops with if-else statements, and implement them in MIPS. (If you haven't, you should probably wait until this material has been covered in class.) In this exercise, we'll look at a program with a simple for loop, and trace through it using spim.

Steps:

Earlier, we copied some files from ~whsu/csc256/LABS/PROGS/. One of these files is pretty much Example 2.4 from your lecture slides, illustrating a for loop. An excerpt:

```
# sum   $s0
# i     $s1
# limit $s2

        .data
endl:   .asciiz "\n"

        .text
        .globl  loop
        .globl  cont

main:   li      $v0,5           #   cin >> limit;
        syscall
        move    $s2,$v0

        li      $s0,0
        li      $s1,1           #   for (i=1; i<=limit; i++)
        bgt     $s1,$s2,cont
loop:   add     $s0,$s0,$s1     #     sum = sum + i;
        add     $s1,$s1,1
        ble     $s1,$s2,loop
```

```
cont:   move    $a0,$s0         #   cout << sum << endl;
        li      $v0,1
        syscall
```

The main difference is we added these two lines:

```
        .globl  loop
        .globl  cont
```

As we mentioned before, **.text** is an assembler directive that says "the following is the start of the program code and should be read-only". Then comes the assembler directive **.globl loop**, which means "make loop a global label".

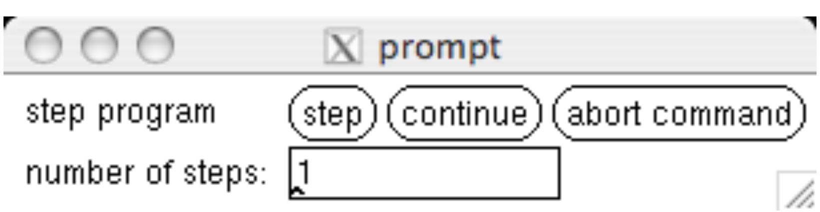
If we do not make **loop** a global label (i.e., if we delete the line .globl loop), the program will still run correctly. However, spim will not be able to "see" the label **loop** when we trace the program. Making **loop** a global label does not change how the programs runs; however, it helps us to trace and debug the program, if necessary.

The simplest way to walk through a program, instruction by instruction, is just to step through it using spim's **step** command. Let's invoke spim, load Example 2.4, and just hit **step** many times (note that hitting **[RETURN]** repeats the last command. Note that we start off in some code that doesn't appear in the source file for Example 2.4! These are instructions from spim's *kernel*; they set up the execution environment for your main program. Don't worry about them for now:

```
(spim) lo "2.4"
(spim) step
[0x00400000]    0x8fa40000  lw $4, 0($29)                  ; 183:
lw $a0 0($sp)               # argc
(spim) step
[0x00400004]    0x27a50004  addiu $5, $29, 4               ; 184:
addiu $a1 $sp 4             # argv
(spim)
[0x00400008]    0x24a60004  addiu $6, $5, 4                ; 185:
addiu $a2 $a1 4             # envp
(spim)
[0x0040000c]    0x00041080  sll $2, $4, 2                  ; 186:
sll $v0 $a0 2
(spim)
[0x00400010]    0x00c23021  addu $6, $6, $2                ; 187:
addu $a2 $a2 $v0
(spim)
[0x00400014]    0x0c100009  jal 0x00400024 [main]          ; 188:
jal main
(spim)
               finally we reach our main program here …
[0x00400024]    0x34020005  ori $2, $0, 5                  ; 32: li
```

```
$v0,5           #   cin >> limit;
(spim)
[0x00400028]    0x0000000c  syscall                          ; 33:
syscall
3          user enters limit…
(spim) step
[0x0040002c]    0x00029021  addu $18, $0, $2                 ; 34:
move  $s2,$v0
(spim) step
[0x00400030]    0x34100000  ori $16, $0, 0                   ; 36: li
$s0,0
(spim) step
[0x00400034]    0x34110001  ori $17, $0, 1                   ; 37: li
$s1,1           #   for (i=1; i<=limit; i++)
(spim)
[0x00400038]    0x0251082a  slt $1, $18, $17                 ; 38:
bgt   $s1,$s2,cont
(spim)
[0x0040003c]    0x14200005  bne $1, $0, 20 [cont-0x0040003c]
(spim)
[0x00400040]    0x02118020  add $16, $16, $17                ; 39:
add   $s0,$s0,$s1     #     sum = sum + i;
(spim)
[0x00400044]    0x22310001  addi $17, $17, 1                 ; 40:
add   $s1,$s1,1
(spim)
[0x00400048]    0x0251082a  slt $1, $18, $17                 ; 41:
ble   $s1,$s2,loop
(spim)
[0x0040004c]    0x1020fffd  beq $1, $0, -12 [loop-0x0040004c]
(spim)
```

This will work pretty much the same way in xspim as well. To step through one single instruction, just click on the **step** button. You'll see this pop-up window:



Click on the **step** button in the window. The line of code being executed will be highlighted in the Text Segment window; watch how it advances one instruction at a time.

```
                                Text Segments
[0x00400000]    0x8fa40000  lw $4, 0($29)                   ; 183: lw $a0 0($sp)# arg
[0x00400004]    0x27a50004  addiu $5, $29, 4                ; 184: addiu $a1 $sp 4# a
[0x00400008]    0x24a60004  addiu $6, $5, 4                 ; 185: addiu $a2 $a1 4# e
[0x0040000c]    0x00041080  sll $2, $4, 2                   ; 186: sll $v0 $a0 2
[0x00400010]    0x00c23021  addu $6, $6, $2                 ; 187: addu $a2 $a2 $v0
[0x00400014]    0x0c100009  jal 0x00400024 [main]           ; 188: jal main
[0x00400018]    0x00000000  nop                             ; 189: nop
[0x0040001c]    0x3402000a  ori $2, $0, 10                  ; 191: li $v0 10
```

```
                                Text Segments
[0x00400000]    0x8fa40000  lw $4, 0($29)                   ; 183: lw $a0 0($sp)# arg
[0x00400004]    0x27a50004  addiu $5, $29, 4                ; 184: addiu $a1 $sp 4# a
[0x00400008]    0x24a60004  addiu $6, $5, 4                 ; 185: addiu $a2 $a1 4# e
[0x0040000c]    0x00041080  sll $2, $4, 2                   ; 186: sll $v0 $a0 2
[0x00400010]    0x00c23021  addu $6, $6, $2                 ; 187: addu $a2 $a2 $v0
[0x00400014]    0x0c100009  jal 0x00400024 [main]           ; 188: jal main
[0x00400018]    0x00000000  nop                             ; 189: nop
[0x0040001c]    0x3402000a  ori $2, $0, 10                  ; 191: li $v0 10
```

```
                                Text Segments
[0x00400000]    0x8fa40000  lw $4, 0($29)                   ; 183: lw $a0 0($sp)# arg
[0x00400004]    0x27a50004  addiu $5, $29, 4                ; 184: addiu $a1 $sp 4# a
[0x00400008]    0x24a60004  addiu $6, $5, 4                 ; 185: addiu $a2 $a1 4# e
[0x0040000c]    0x00041080  sll $2, $4, 2                   ; 186: sll $v0 $a0 2
[0x00400010]    0x00c23021  addu $6, $6, $2                 ; 187: addu $a2 $a2 $v0
[0x00400014]    0x0c100009  jal 0x00400024 [main]           ; 188: jal main
[0x00400018]    0x00000000  nop                             ; 189: nop
[0x0040001c]    0x3402000a  ori $2, $0, 10                  ; 191: li $v0 10
```

And so on…

When stepping through Example 2.4 earlier, you may have noticed something weird. Example 2.4 has an instruction **`bgt  $s1, $s2, cont.`** But when we step through it, we see two instructions instead of one:

```
(spim)
[0x00400038]    0x0251082a  slt $1, $18, $17                ; 38:
bgt   $s1,$s2,cont
(spim)
[0x0040003c]    0x14200005  bne $1, $0, 20 [cont-0x0040003c]
(spim)
```

What's going on here? Remember that MIPS assembly language instructions still have to be translated into low-level *machine language*, before they are executed by the MIPS hardware. Some of these assembly language instructions are actually *pseudo-instructions*, which may translate into more than one machine language instruction. The bgt instruction actually splits into the **slt** (set-less-than) instruction, and the **bne** (branch-not-equals) instruction. Don't worry about the details for now; we'll see what is actually going on in Chapter 6 on machine language.

Obviously, stepping through a long program is very tedious! We want to execute many lines of a program, and only stop at a few important instructions. Breakpoints let us do this.

We saw earlier how we make certain labels global; this lets us mark them as **breakpoints**. Breakpoints are simply special marked places in the program code. When we run our program and come to an instruction that is marked as a breakpoint, the program stops. This is very useful for debugging.

Now let's try to run Example 2.4, and make it stop at one of our labels, before the program is actually finished! First we invoke spim (or xspim), and load 2.4. To make a global label a breakpoint, use the **bre** command:

```
(spim) lo "2.4"
(spim) bre loop            we've just made loop a breakpoint!
(spim)
```

To look at the list of breakpoints, use the **list** command:

```
(spim) list
Breakpoint at 0x00400040
(spim)
```

(Note that the actual numeric address may be different, if you run spim on a different system.) The message says "Breakpoint at 0x00400040" because the address of loop is 0x00400040. We can check by printing what's at address 0x400040, with the **print** command (abbreviated **pr**):

```
(spim) pr 0x400040
*[0x00400040]   0x02118020  add $16, $16, $17       ; 39: add
$s0,$s0,$s1     #     sum = sum + i;
```

It's the same line at the label **loop**. We've made **loop** a breakpoint, and the program will stop when we get to **loop**. Note the asterisk in the left margin; that also indicates that we are stopped at a breakpoint. Let's also set a breakpoint at **cont**, to stop the program when we leave the for loop. Then we start running the program:

```
(spim) bre cont
(spim) list
       Breakpoint at 0x00400040     breakpoint at loop
       Breakpoint at 0x00400050     breakpoint at cont
(spim) run
```

The program waits for the user to enter limit; let's enter 10:

```
3
Breakpoint encountered at 0x00400040
(spim)
```

As expected, we've stopped at **loop**. Let's check our important registers, $s0 (sum), $s1 (i), $s2 (limit):

```
(spim) pr $s0
Reg 16 = 0x00000000 (0)     sum = 0
(spim) pr $s1
Reg 17 = 0x00000001 (1)     i = 1
(spim) pr $s2
Reg 18 = 0x00000003 (3)   limit = 3
(spim)
```

They contain what we expect, of course! To continue running the program, and stop at the next breakpoint, we use the **continue** command (abbreviated **cont**), and check $s0, $s1, $s2 on the next iteration:

```
(spim) cont
(spim) pr $s0
Reg 16 = 0x00000001 (1)     sum = 1
(spim) pr $s1
Reg 17 = 0x00000002 (2)     i = 2
(spim) pr $s2
Reg 18 = 0x00000003 (3)   limit = 3
(spim)
```

Then one more iteration, stopping at **loop** again:

```
(spim) cont
(spim) pr $s0
Reg 16 = 0x00000003 (3)     sum = 3
(spim) pr $s1
Reg 17 = 0x00000003 (3)     i = 3
(spim) pr $s2
Reg 18 = 0x00000003 (3)   limit = 3
(spim)
```

Then we go through the last iteration, and stop at the final breakpoint, which is **cont (0x400050)**. Again we display $s0, $s1 and $s2:

```
(spim) cont
Breakpoint encountered at 0x00400050
(spim) pr $s0
Reg 16 = 0x00000006 (6)     sum = 6
```

```
(spim) pr $s1
Reg 17 = 0x00000004 (4)    i = 4
(spim) pr $s2
Reg 18 = 0x00000003 (3)    limit = 3
(spim)
```

Setting breakpoints in xspim is very similar. Start xspim, load Example 2.4, then clock on the **breakpoints** button. You'll see this pop-up window:

In the textbox, you can type in the names of the two global labels we used as breakpoints earlier (**loop** and **cont**; one by one, please!), then click on the **add** button.

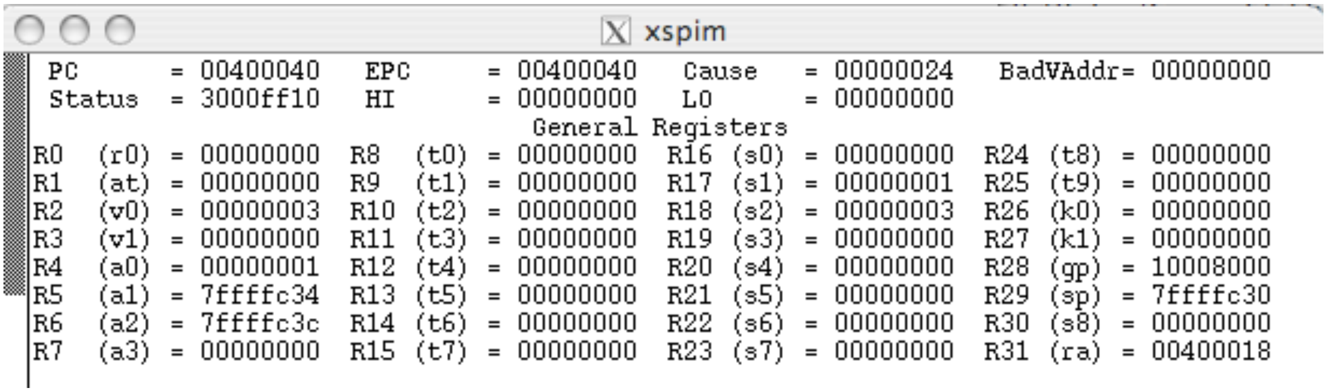
After entering the two breakpoints, click on the **list** button. If you look in the bottom-most window in xspim, you'll see the list of breakpoints (at addresses 0x400040 for **loop**, and 0x400050 for **cont**):

```
SPIM Version 8.0 of January 8, 2010
Copyright 1990-2010, James R. Larus.
All Rights Reserved.
See the file README for a full copyright notice.
Loaded: /afs/sfsu.edu/f1/whsu/spim-8.0/lib/exceptions.s
Breakpoint at 0x00400050
Breakpoint at 0x00400040
```

Then we can run the program by clicking on **run**; we stop when the program waits for us to enter **limit**. Then we'll stop the first time we reach the **loop** label, indicated (again) by the asterisk in the Text Segment window:

```
                              Text Segments
 [0x00400028]   0x0000000c  syscall                        ; 33: syscall
 [0x0040002c]   0x00029021  addu $18, $0, $2               ; 34: move  $s2,$v0
 [0x00400030]   0x34100000  ori $16, $0, 0                 ; 36: li    $s0,0
 [0x00400034]   0x34110001  ori $17, $0, 1                 ; 37: li    $s1,1   #   f
 [0x00400038]   0x0251082a  slt $1, $18, $17               ; 38: bgt   $s1,$s2,cont
 [0x0040003c]   0x14200005  bne $1, $0, 20 [cont-0x0040003c]
*[0x00400040]   0x02118020  add $16, $16, $17              ; 39: add   $s0,$s0,$s1#
 [0x00400044]   0x22310001  addi $17, $17, 1               ; 40: add   $s1,$s1,1
```

We can track the contents of registers by looking at the top window, where all the registers are displayed ($s0 = 0, $s1 = 1, $s2 = 3 at the moment):



Then click on **continue** a few more times to run through the program, stopping at a breakpoint each time. Again, this is very similar to what happened in spim.

Here's a summary of what we learned about breakpoints:

1. Breakpoints are places in the code where program execution stops.
2. To make a label in a MIPS program a breakpoint, we first have to make the label a global label. This is usually done right after the .text directive:

```
.text
.globl  loop
```

3. Just because a label is globalized doesn't mean it's a breakpoint! To make it a breakpoint, we have to go into spim, and use the **bre** command. (**bre label** makes a breakpoint at "label".)
4. To list all breakpoints, use the command **list.**
5. When we're stopped at a breakpoint, spim will indicate to us the address of the breakpoint that it stopped at (since there might be several breakpoints).
6. To continue running a program until the next breakpoint (or until the end of the program), type **continue.**
7. To look at register contents, use the **print** command.