MIPS Examples

Thomas Finley, April 2000

Contents and Introduction

This document is not intended as a beginner's guide to MIPS. It is intended for people that have coded some with MIPS and feel somewhat comfortable with its use. If this is not you you will not get much out of this document.

This document provides examples that are supposed to give greater insight into what MIPS does, and how to use MIPS for (more or less) useful applications. I cover how to read in strings in MIPS and what happens to memory when you read in strings. I also cover using arrays in MIPS.

String from the Console

I will provide a very simple example to give a feel for syscall functionality for reading in strings. It will help if you open up your book to A-49 in the "Computer Organization & Design" book by Patterson and Hennessy, because I will make reference to the table at the top of that page in my example. I provide a copy of the table here.

Service	System call code	Arguments	Result
print_int	1	$a0=integer	
print_float	2	$f12=float	
print_double	3	$f12=double	
print_string	4	$a0=string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0=buffer, $a1=length	
sbrk	9	$a0=amount	
exit	10		

The "Arguments" column explains what should be in a specific argument register (or registers) before a specific syscall. The "Result" column tells what the contents of registers will hold after the syscall. For example, if you want to output 23, you must put 23 into register $a0, and then do a syscall 1. If you want to read an int, you simply do a syscall 5. The register $v0 holds the result of the read.

How *NOT* to do Strings in MIPS

About strings, one prevalent problem I noticed with people's code that I reviewed was that people would try to output a string by putting ASCII values into $a0. This is not correct. Just as in C, you output a string by passing the MEMORY ADDRESS of the beginning of a sequence of characters (bytes).

Similarly, if you do a syscall 8 (read_string), the contents of the string read in are not in $a0. How could, say, a 256 byte string fit into a 4 byte quantity? That doesn't make sense.

The Example

Now suppose you have a file with this very simple MIPS code in it:

```
    .data
theString:
    .space 64
    .text
main:
    li      $v0, 8
    la      $a0, theString
    li      $a1, 64
    syscall
    jr      $ra
```

I'll go through it line by line.

The first line ".data" tells SPIM that what follows will be data.

".space 64" then sets aside 64 bytes for use of whatever purpose we want, the first byte of which may be referenced by the label "theString:", which appears on the line before.

".text" then tells the computer that what follows will be actual code.

"main:" is our requisite main label that symbolizes the start of the program.

The next three lines of "la" and "li" statements set registers to appropriate values before we say "syscall". This is where you should take look at the table at the top of A-49. Find the "read_string" line, and then read the rest of this. I provide a line of the code, and then some background.

```
    li      $v0, 8
```

When you call "syscall" in your code, a value called the "system call code" will determine what function syscall performs. The "system call code" is stored in the register $v0. The system call code for reading a string is 8, so I stored the number 8 into register $v0 using "li".

```
    la      $a0, theString
```

If you look at the "arguments" column in this table, it says "$a0 = buffer, $a1 = length". What this means is that the $a0 register must be set to the location in memory to which the computer will record the input. This is accomplished by loading the address (la) of theString into $a0.

```
    li      $a1, 64
```

The second part of the arguments entry in the table says "$a1 = length", which you set to the maximum number of characters that should be read in. I chose 64 characters. You can have it be 50, or 200, or 37, or whatever you like, but you shouldn't go above 64 (in this example) because in the first part of this program you only set aside 64 bytes using the ".space" directive. Note that this space set aside includes the terminating null '\0' character, so it will actually read only up to 63 characters from the buffer when the syscall executes.

```
    syscall
```

At long last, having set your argument ($a0, $a1) registers and your call code register ($v0), you call syscall. Upon receiving the syscall command, the system says, "what do I need to do?" and sees that $v0 == 8. It now knows to read in a line from the SPIM console, and to write the input to the memory location referenced by $a0 (which was set to theString), for a string of maximum length of $a1 (which we set to 64).

It reads input until it encounters a '\n' character or reaches the maximum number of characters it can reach (which we stored in $a1 as 64), and stores that input (including the '\n' character) into a string null-terminated with a '\0'. Notice that the maximum length of the string includes the '\0' terminating null character.

```
    jr      $ra
```

This jump-returns to the return address in the $ra register.

An Actual Run

If you run this program and type this in:

```
    Hello!
```

...and hit return, the memory in the computer at the point referenced by theString will look like the following. Each block represents a byte in data. The blocks are adjacent, and so are the bytes in memory. The byte holds the ASCII value for the character I display. The first byte is the byte referenced by "theString", and the string is termined by a null character.

H	e	l	l	o	!	\n	\0

After syscall is finished, the byte referenced by "theString" would contain the ascii value for 'H', the next byte would contain 'e', etc, etc.

Vectors

For an explanation of "vectors" in SPIM, I will construct a SPIM program that calculates the first 40 terms of my favorite sequence, the Fibonacci sequence, and stores it in an array like structure. For those that do not know, the Fibonacci sequence is the sequence of numbers {1, 1, 2, 3, 5, 8, 13, etc} such that $a_{n+2} = a_{n+1} + a_n$. The "classic" Fibonacci sequence, if there can be said to be such a thing, is the sequence where $a_0 = 1$ and $a_1 = 1$.

First we see an effort in C. The intention for providing this code is to create a natural flow from C to MIPS, and to demonstrate how arrays in C and arrays in MIPS need not be thought of as radically different entities.

I assume familiarity with C, and some basic familiarity with how to read data to and from memory in MIPS (specifically with lw and sw).

Incidentally, my work that follows is often purposefully inefficient for the purpose of greater clarity, though sometimes being clear one way leads to being unclear in some other way. I wrote this all very late at night while afflicted with insomnia.

```
#include <stdio.h>

int theArray[40];

int main() {
  int n = 2;
  theArray[0] = 1;
  theArray[1] = 1;
  do {
    theArray[n] = theArray[n-1] + theArray[n-2];
    n++;
  } while (n < 40);

  return 0;
}
```

Now I'll rewrite the program to make it even MORE inefficient. It will still be in C, except it will be built to aid our transition to SPIM when we attempt to accomplish the same feat with MIPS.

```
#include <stdio.h>

int theArray[40];

int main() {
  int t0, t1, t2, t3, t4, t5, t6, t7;  /* Our "registers" */
  t6 = 1;
  t7 = 1;
  theArray[0] = t6;  /* Storing the first two terms of the  */
  theArray[t7] = t6; /* sequence into our array             */
  t0 = 2;
LLoop:
  t3 = t0 - 2;
  t4 = t0 - 1;
  t1 = theArray[t3];
  t2 = theArray[t4];
  t5 = t1 + t2;
  theArray[t0] = t5;
  t0 = t0 + 1;
  if (t0 < 40) goto LLoop;
  return 0;
}
```

Presumably we're all familiar with C, so we can see how this program works. Just look over it until it begins to make sense, because (aside from the ambiguous variable names) it's not that tough. We then "translate" this into MIPS assembler language. When we do so, there are several differences that must be kept in mind.

The thing with "arrays" in MIPS, if we're to call them that, is that the "indices" are always incremented in terms of bytes. The address "theArray($t0)" will address theArray, but offset by $t0 bytes, that is the address referenced by the label "theArray" plus the contents of register $t0. Integers take up a word; that is, they are four bytes long. If you have a segment of memory that you intend to use as an array of integers, to move up (or down) one "element" you must increment (or decrement) your addresses not by one, but by four!

This actually isn't that different. The only difference is, C does this for you. It knows that you have an array of integers, and you're referencing "theArray[i]" and then reference "theArray[i+1]", it will react as you'd expect. With SPIM, you must make allowances yourself. Here is the SPIM code.

```
  .data
theArray:
  .space 160
  .text
main:
  li    $t6, 1              # Sets t6 to 1
  li    $t7, 4              # Sets t7 to 4
  sw    $t6, theArray($0)   # Sets the first term to 1
  sw    $t6, theArray($t7)  # Sets the second term to 1
  li    $t0, 8              # Sets t0 to 8
loop:
  addi  $t3, $t0, -8
  addi  $t4, $t0, -4
  lw    $t1, theArray($t3)  # Gets the last
  lw    $t2, theArray($t4)  #   two elements
  add   $t5, $t1, $t2       # Adds them together...
  sw    $t5, theArray($t0)  # ...and stores the result
  addi  $t0, $t0, 4         # Moves to next "element" of theArray
  blt   $t0, 160, loop      # If not past the end of theArray, repeat
  jr    $ra
```

Before some punk points out how inefficient this MIPS code is, the point of this program is to illustrate how to read from and write to locations in memory in a manner reminiscent of arrays. It is not a paradigm of efficiency.

Most of it, you see, is a very clear translation, but there are differences that are important to notice. The first has to do with the portion of memory referenced by "theArray:".

```
theArray:
  .space 160
```

The ".space" directive reserves a section of free space in a size given by bytes. Since an int takes up 4 bytes and we want to store 40 integers, 4*40 is 160, so we reserve 160 bytes.

```
  t7 = 1;
  theArray[0] = t6;  /* Storing the first two terms of the  */
  theArray[t7] = t6; /* sequence into our array.            */
```

We see that t7 is used to store 1, which is the index of the second element. In SPIM, the second element would not be referenced by an offset of one from the address of the beginning of the array, but instead by an offset of four bytes.

```
  li    $t7, 4              # Sets t7 to 4.
  sw    $t6, theArray($0)   # Sets the first term to 1.
  sw    $t6, theArray($t7)  # Sets the second term to 1.
```

Moving on.

```
  t0 = 2;
```

I set have the idea that theArray[t0] would get theArray[t0-1] + theArray[t0-2]. I set t0 to 2 so that it starts at the THIRD ELEMENT of theArray. To accomplish the same effect with MIPS, I must start at an offset of 8.

```
  li    $t0, 8              # Sets t0 to 8.
```

Moving on.

```
  t3 = t0 - 2;
  t4 = t0 - 1;
```

In the C code, we wanted to reference the two elements before the element of index t0. I store the indices of these two elements in t3 and t4.

```
  addi  $t3, $t0, -8
  addi  $t4, $t0, -4
```

Similar to before, if theArray($t0) points to the element of this array that we're currently calculating, by the definition of the Fibonacci sequence we want to reference the previous two terms. I've gone over offsets by factors of eights several times already, so I'm going to assume that you can figure out that what I'm doing here is equivalent to what I do in my C code. Moving on.

```
  t0 = t0 + 1;
```

This offsets the index by 1, which in SPIM would be accomplished by increasing the offset by 4 bytes. Remember, when we increment in the C code, that is REALLY going forward the length of an int in memory, or four bytes. MIPS does not do this for us, so we must add four.

```
  addi  $t0, $t0, 4
```

Moving on.

```
  if (t0 < 40) goto LLoop;
```

If the index is now 40 after we've incremented it, then we're done. The effect for the MIPS branch is similar, except we take into account that we're dealing with indices of bytes, not words.

```
  blt   $t0, 160, loop
```

The 40 elements are referenced by the addresses (theArray + 0), (theArray + 4), (theArray + 8), etc etc, all the way up to (theArray + 156). If our offset has reached 160, then we shouldn't do any more, and the program ends.

```
  jr    $ra
```

This jump-returns to the return address in the $ra register.

I ran this program, and look what memory contained after execution.

```
[0x10010000]                  0x00000001  0x00000001  0x00000002  0x00000003
[0x10010010]                  0x00000005  0x00000008  0x0000000d  0x00000015
[0x10010020]                  0x00000022  0x00000037  0x00000059  0x00000090
[0x10010030]                  0x000000e9  0x00000179  0x00000262  0x000003db
[0x10010040]                  0x0000063d  0x00000a18  0x00001055  0x00001a6d
[0x10010050]                  0x00002ac2  0x0000452f  0x00006ff1  0x0000b520
[0x10010060]                  0x00012511  0x0001da31  0x0002ff42  0x0004d973
[0x10010070]                  0x0007d8b5  0x000cb228  0x00148add  0x00213d05
[0x10010080]                  0x0035c7e2  0x005704e7  0x008cccc9  0x00e3d1b0
[0x10010090]                  0x01709e79  0x02547029  0x03c50ea2  0x06197ecb
```

Isn't that pretty?

I'd recommend trying this program out in xspim and seeing what the contents of memory end up as, and perhaps you can fool around with it if you doubt your mastery.

Thomas Finley 2000