Computer Architecture

Andreas Klappenecker *Texas A&M University*

Lecture notes of a course on Computer Architecture, Fall 2003.
Preliminary draft.

Chapter 1

Assembly Language

The purpose of this chapter is to give an introduction to the basics of the MIPS assembly language. If you want to get fluent in a programming language, then the best approach is always *learning by doing*. You should have the SPIM simulator ready so that you can immediately check the behavior of the programs, and—more importantly—create and implement some variations. Read the simulator manuals and solve all the exercises.

This chapter supplements Chapter 3 and Appendix A in Patterson and Hennessy, *Computer Organization and Design*, Morgan Kaufmann Publishers, 1997.

§1 Getting Started

The MIPS architecture is an example of a *Reduced Instruction Set Computer* architecture, which is characterized by a relatively small set of instructions and a comparatively large number of registers. It is a so-called load-store architecture, which means that the main memory is accessed only through load and store operations; all other instructions are between registers. Among other advantages, this helps to simplify the design of pipelining mechanisms. In case you are curious, MIPS is an acronym for *Microprocessor without Interlocked Pipeline Stages*.

The MIPS architecture and its variations was used in SGI workstations, CISCO routers, Nintendo 64 video games, early Linux PDAs, and many embedded systems. We will run our programs on the SPIM simulator of the MIPS architecture.

Let us write our first assembler program. We want to add two integers a and b and store the result in a different memory location c. In a high level

language, you could formulate this by the statement $c = a + b$.

If a, b, and c denote registers `$t1`, `$t2`, `$t3`, then the MIPS assembler statement `add $t3, $t1, $t2` solves the problem just as easily.

If a, b, and c are integers stored in the main memory, then we need to load a and b into registers `$t1` and `$t2`, perform the addition as before, and then store the result of register `$t3` into c. The MIPS assembler has the command `lw` to load a word into a register, and the command `sw` to store a word that is contained in a register into the memory. The four commands to load, add, and store the results are show in the following MIPS assembler program:

```
# Addition of two values, c=a+b
        .text                   # code section
        .globl main
main:
        lw $t1, a_addr          # load a into $t1
        lw $t2, b_addr          # load b into $t2
        add $t3, $t1, $t2       # add $t1 and $t2 and store in $t3
        sw $t3, c_addr          # store $t3 into c
.data                           # data section
a_addr: .word 10                # value a=10
b_addr: .word 20                # value b=20
c_addr: .word 0                 # intialize c arbitrarily
```

In general, a MIPS assembler program has a code section and a data section. The assembler directive `.text` tells the assembler that assembler instructions follow. Similarly, the assembler directive `.data` has the effect that the following data items are stored in the data segment. A label can have local file scope, or might be globally visible. For instance, the label `main` is declared to be globally visible by the directive `.globl`.

In the data section of the above program, the variables a and b are initialized to 10 and 20, respectively. The local labels `a_addr` and `b_addr` allow the assembler to determine the address of these memory locations. The assembler directive `.word` means that the following data is interpreted as a 32 bit word.

Exercise 1.1 *Use the MIPS simulator* `spim` *(which might be called* `xspim` *or* `pcspim` *on your system) to trace step by step through the above program. Make yourself familiar with the functionality of this simulator.*

Exercise 1.2 *Write a MIPS assembler program to calculate $d = (a - b) + c$. Consult Appendix A in [2] to learn about assembler instructions (for instance, to perform a subtraction).*

Exercise 1.3 *In Chapter 3, the textbook repeatedly uses the sequence*

```
add $t0, $t0, $t0
add $t0, $t0, $t0
```

to multiply the register `$t0` *by four. Find a single MIPS assembler command that has the same effect but is not a multiplication.*

§2 Hello World!

The `spim` simulator offers a minimal set of system calls. The available procedures allow reading and writing integers, floating point numbers, and strings. We illustrate a system call with the notorious `Hello World!` program.

A system call is performed as follows: Load the system call code into register `$v0`, and the arguments into the registers `$a0`, ..., `$a3`. Execute the system call by `syscall`. If the system call returns a value, then it can be found in register `$v0`.

Printing a string is easy. The code of the print string routine is 4. The load immediate pseudoinstruction `li $v0, 4` prepares the MIPS CPU for the execution of the print string routine. The address of the string must be contained in the argument register `$a0`. The load address instruction `la $a0, str` calculates the address of the label `str` and stores the result in `$a0`. The complete MIPS assembler program to print `Hello World!` is given below.

```
# Hello world program
        .text                   # code section
        .globl main
main:
        li $v0, 4               # system call for print_str
        la $a0, str             # address of string to print
        syscall                 # print the string

        li $v0, 10              # system call for exit
        syscall                 # exit

.data                           # data section
str:    .asciiz "Hello world!\n" # NUL terminated string
```

The assembler directive `.asciiz` means that the following data is a string terminated by `\0`. The second system call is simply to exit the program.

Exercise 1.4 *Write a MIPS assembler program that prompts the user to enter two integers, calculates the sum of these integers, and prints the result. [Hints: The system call code of the read integer routine is 5, and the result of this routine is contained in register* `$v0`*. The system call code to write an*

integer is 1; the argument should be contained in register `$a0`*. The content of register* `a` *can be assigned to register* `b` *by* `move b, a`*.]*

§3 Branching

The control flow of a MIPS assembler program can be influenced by branch instructions. For instance, the branch on equal instruction `beq a, b, label` compares the registers `a` and `b`, and jumps to the label `label` if the register contents are equal. Similarly, `ble a, b, label` jumps to `label` if register `a` is less than or equal to register `b`; this instruction corresponds to the high level command `if a<=b then goto label`.

The instructions `beq` and `ble` are examples of conditional branches. An unconditional branch or jump is given by `j label`, which jumps to `label`. These conditional branch and unconditional jump instructions are useful to implement if-then-else constructs. Suppose that you want to express the statement `if a==b then /* blockA */ else /* blockB */ fi`. We assume that register `$t0` represent the variable a, and register `$t1` the variable b. A direct translation of this statement can be accomplished by the following assembly language template:

```
        beq $t0, $t1, blockA    # if $t0 = $t1 goto blockA
        j blockB                # goto blockB
blockA: ...                     # omitted stmts of blockA (then part)
        j endif                 # goto endif
blockB: ...                     # omitted stmts of blockB (else part)
endif:  ...                     # subsequent statements
```

The resulting sequence of statements, however, is not optimal. The following two exercises show how to improve upon this code:

Exercise 1.5 *Read and memorize all branch and jump commands given in Appendix A of [2].*

Exercise 1.6 *Find a MIPS assembler template for*

```
if a==b then /* blockA */ else /* blockB */ fi
```

that uses just one jump instruction `j` *instead of two.*

§4 Loops

The branch instructions can be used to form loops. We illustrate this with a little assembler program that prints the integers from 1 to 10. The registers

`$s0` and `$s1` contain the lower and the upper loop bounds. The program increases the register `$s0` by one in each iteration of the loop.

```
# Loop printing the integers from 1 to 10
        .text
        .globl main
main:
        li $s0, 1               # $s0 = loop counter = lower bound
        li $s1, 10              # $s1 = upper bound of loop
loop:   move $a0, $s0           # print loop counter
        li $v0, 1               #   "
        syscall                 #   "

        li $v0, 4               # print a linebreak
        la $a0, linebrk         #   "
        syscall                 #   "

        addi $s0, $s0, 1        # increase loop counter by 1
        ble $s0, $s1, loop      # if $s0 <= $s1 goto loop

        li $v0, 10              # system call for exit
        syscall                 # exit

.data
linebrk: .asciiz "\n"
```

The two system calls print the counter `$s0` and a newline, respectively. The instruction `addi a, b, const` adds the constant `const` to the content of register `b` and stores the result in register `a`. This instruction is used to increase the counter by one in each iteration. The branching instruction `ble $s0, $s1, loop` jumps back to `loop` unless `$s0` exceeds 10.

You might have recognized that this implementation resembles a repeat-until loop of high-level programming languages. It is possible to find implementations for while loops and for loops as well.

§5 Addressing

The memory operations can only store or load data from addresses aligned to suit the data type. This means that a word (which is four bytes long) can be loaded only from addresses that are divisible by four. Load and store operations are available for bytes, halfwords, words, and double words.

So far, we have accessed memory locations by explicitly referencing a label. Essentially, the assembler calculated the address, a convenient but inflexible

solution. Alternately, the load and store operations can select a memory location by the value of a register to which a 16-bit signed offset is added. For instance, the instruction `lw $t0, 16($t1)` loads the register `$t0` with the content of the memory at address (value of `$t1`)+16.

We illustrate this addressing mode by a program that adds the constant 5 to each element of an "array" represented by a sequence of memory cells. We initialize this sequence by the directive `.word 0,2,1,4,5` in the data section. The first element is accessible via the label `Aaddr`; this address is loaded into register `$t0`. We have a loop that steps through each of the array elements by increasing the address by four bytes in each iteration. Readers familiar with C or C++ can easily formulate an equivalent high-level program using pointer arithmetic.

```
# Increases all elements of array by 5
        .text
        .globl main

main:
        la  $t0, Aaddr          # $t0 = pointer to array A
        lw  $t1, len            # $t1 = length (of array A)
        sll $t1, $t1, 2         # $t1 = 4*length
        add $t1, $t1, $t0       # $t1 = address(A)+4*length
                                #     = just beyond the last element of A

loop:   lw  $t2, 0($t0)         # $t2 = A[i]
        add $t2, $t2, 5         # $t2 = $t2 + 5
        sw  $t2, 0($t0)         # A[i] = $t2
        add $t0, $t0, 4         # i = i+1
        bne $t0, $t1, loop      # if $t0<$t1 goto loop

        li $v0, 10              # exit
        syscall                 #

        .data
Aaddr:  .word 0,2,1,4,5         # array with 5 elements
len:    .word 5
```

Exercise 1.7 *The above program assumes that the "array" contains at least one element. Modify the program such that empty "arrays" of length zero are correctly treated.*

§6 Procedure Calls

An assembler program can be structured into subroutines or functions, similar to high-level languages. For longer assembler programs, the design will usually consist of smaller subroutines that are easier to test.

In a high-level language, calling a subroutine usually means remembering the place from which the call originated, and saving all variables that will be overshadowed by local variables of the called procedure. At the end of the procedure, the execution will continue with the next command after the subroutine call, after restoring the saved variables.

In assembly language, the subroutine call behaves similarly, but only *by convention* rather than by rules enforced and checked by the assembler. This means that you need to be extra careful when coding procedures, so that nobody using your routines—including yourself—will be annoyed.

Register Conventions. The use of the 32 registers in the MIPS architecture is governed by conventions. The registers `$t0`–`$t9` are temporary and can be used by the called subroutine without saving. This means that the value of these registers is possibly modified after a call to a subroutine, so the calling procedure might have to save and restore them, if that is necessary.

A subroutine using any of the registers `$s0`–`$s7` has to save the value of the register before modifying it, and has to restore its value before it exits. It is instrumental that you follow this rule because everybody will rely on this behavior. The register `$sp` contains the stack pointer, and the stack is used, for instance, to store and retrieve register values.

The arguments for the subroutine are stored in the registers `$a0`–`$a3`. The return values are stored in register `$v0` and, possibly, `$v1`.

Simple Procedures. We will ignore for the moment that variables might have to be stored and restored, and focus on the procedure call itself.

A procedure call is initiated by executing the command `jal procname`, where `procname` is a label denoting the first address of the procedure. This instruction jumps to the address `procname` and it stores the return address, that is, the address of the following instruction, in the return address register `$ra`. Thus, at the end of the procedure, the instruction `jr $ra` returns the control to the instruction just behind the procedure call.

Let us have a look at a simple example. We rewrite our program to print integers from 1 to 10, with a line break after each integer. This time, we use a procedure `print_int` to print an integer, and `print_eol` to print the line break. The structure of the loop remains the same, but the purpose of the

system calls is more transparent by packaging them into procedures with more meaningful names. A minor disadvantage is that we get some small overhead through the procedure calls.

```
# Loop printing the integers from 1 to 10
.text                   # code section
.globl main

print_int:                      # prints the integer contained in $a0
        li $v0, 1               #
        syscall                 #
        jr $ra                  # return();

print_eol:                      # prints "\n"
        li $v0, 4               #
        la $a0, linebrk         #
        syscall                 #
        jr $ra                  # return();

main:
        li $s0, 1               # $s0 = loop counter = lower loop bound
        li $s1, 10              # $s1 = upper bound of loop
loop:   move $a0, $s0           # print loop counter
        jal print_int           #   store return address and jump to print_int
        jal print_eol           # print "\n"
        addi $s0, $s0, 1        # increase loop counter by 1
        ble $s0, $s1, loop      # if $s0 <= $s1 goto loop

        li $v0, 10              # exit program
        syscall                 #

        .data
linebrk: .asciiz "\n"
```

It was not necessary to save any registers in the above example. We explain the basic principles of such housekeeping principles in the next paragraph.

The Stack. It is convenient to store and retrieve register values from the stack during procedure calls, particularly when the procedure is recursive. Usually, an abstract stack is equipped with functions `push` and `pop`. In the MIPS architecture, the treatment is a little bit more spartanic.

The register `$sp` holds the address of the top element of the stack. By convention, the stack grows from higher addresses to lower addresses. We can push the content of a register, say `$s0`, onto the stack by

```
sub $sp, $sp, 4
sw  $s0, 0($sp)
```

Similarly, we can pop the value by

```
lw $s0, 0($sp)
add $sp, $sp, 4
```

The only thing to keep in mind is that a sequence of pushes must be undone by the corresponding sequence of pops in reverse order, when saving and restoring register variables.

A subtle point occurs when a procedure `procA` calls another procedure `procB`. In this case, the second call `jal procB` overwrites the return address of procedure A. This means that we need to store the return address register `$ra` before calling `procB` and restore the value of `$ra` afterwards.

An Example. The basic principles of procedure calls are best illustrated with the help of a simple example. We implement a recursive procedure that calculates the nth Fibonacci number $\text{fib}(n)$. Recall that

$$\text{fib}(0) = 0,\ \text{fib}(1) = 1,\ \text{fib}(2) = 1,\ \text{fib}(3) = 2,\ \text{fib}(4) = 3,\ \text{fib}(5) = 5,\ \ldots$$

Further elements of this sequence can be calculated by $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$.

The main procedure prompts the user to input n. It calls the recursive procedure $\text{fib}(n)$. As a recursive procedure, fib has to store the return address on the stack. It also stores the registers `$s0` and `$a0`. The purpose of these registers is to store intermediate results; `$s0` will contain $\text{fib}(n-1)$, and `$a0` keeps track of the arguments. The program is adapted from Waldron [3], after a few modifications.

```
# Calculating Fibonacci numbers
# fib(0) = 0
# fib(1) = 1
# fib(n) = fib(n-1)+fib(n-2)
        .text
        .globl main
print_str:
        li $v0, 4 # print string at ($a0)
        syscall #
        jr $ra # return;

print_eol:
```

```
        la $a0, eol             # print "\n"
        li $v0, 4               #
        syscall                 #
        jr $ra                  # return;

print_int:
        li $v0, 1               # print integer ($a0)
        syscall                 #
        jr $ra                  # return;


# fib(n) - recursive function to compute nth Fibonacci number
#
fib:    sub $sp,$sp,12          # save registers on stack
        sw $a0, 0($sp)          # save $a0 = n
        sw $s0, 4($sp)          # save $s0
        sw $ra, 8($sp)          # save $ra to allow recursive calls

        bgt $a0,1, gen          # if n>1 then goto generic case
        move $v0,$a0            # output = input if n=0 or n=1
        j rreg                  # goto restore registers

gen:    sub $a0,$a0,1           # param = n-1
        jal fib                 # compute fib(n-1)
        move $s0,$v0            # save fib(n-1)

        sub $a0,$a0,1           # set param to n-2
        jal fib                 # and make recursive call
        add $v0, $v0, $s0       # $v0 = fib(n-2)+fib(n-1)

rreg:   lw  $a0, 0($sp)         # restore registers from stack
        lw  $s0, 4($sp)         #
        lw  $ra, 8($sp)         #
        add $sp, $sp, 12        # decrease the stack size
        jr $ra


main:
        la $a0, en              # print "n = "
        jal print_str

        li $v0, 5               # read integer
        syscall                 #

        move $a0, $v0           # $a0 := $v0
```

```
        jal fib                 # call fib(n)
        move $s0, $v0           # store result in $s0

        la $a0, fibstr          # print "fib(n) = "
        jal print_str           #

        move $a0,$s0            # print fib(n)
        jal print_int           #
        jal print_eol           # print "\n"
        li $v0,10               # exit
        syscall                 #

        .data
eol:    .asciiz "\n"
en:     .asciiz "n = "
fibstr: .asciiz "fib(n) = "
```

The overall structure is simple, except that at the beginning of the procedure fib, some registers are saved that are restored at the end of the procedure.

Exercise 1.8 *Why is it necessary to save and restore register* `$a0`*?*

Exercise 1.9 *Write a recursive procedure that calculates* $n! = n(n-1)\cdots 1$.

§7 Elements of Style

An assembler program is much harder to read than a high-level language program. The changing roles of register contents can be confusing, and the large amount of detail makes it hard to interpret a program. A proper style of commenting your program is absolutely essential. If I would not have included any comments in the previous examples, they would be much harder to understand, even tough they are tiny.

As a rule of thumb, you should comment nearly every line of your program. An exception are system calls, where the purpose—such as print "abc"—can be clarified by a single line of comment. In this case, a comment that `syscall` is a system call is hardly illuminating.

Try to stick to similar indentations for the label, instruction, and comment columns of your program. In general, you should try to make the presentation appealing. Nobody wants to read some messy code, particularly assembly code.

You can help the reader to better understand a procedure by explaining the purpose of the registers. Especially, it is good practice to indicate which

variables are used. Especially, if your subroutine is affecting the content of registers, such as `$t0`, then you should mention that.

If your program is long, then it is helpful indicate the purpose of some assembly code by stating equivalent high-level code in comments. This allows a reader to navigate more quickly through a large assembly language program.

§8 Further Reading

Appendix A by James Larus in [2] is essential reading material. This appendix contains all the instructions and pseudo-instructions that you will need for your programming assignments. There are some helpful links on the class homepage that contain numerous examples and good explanations.

If you are really curious, then you can read the books by Britton [1] or Waldron [3] on MIPS assembly language programming.

Bibliography

[1] R.L. Britton. *MIPS Assembly Language Programming*. Pearson Education, Upper Saddle River, NJ, 2004.

[2] D. Patterson and J. Hennessy. *Computer Organization: The Hardware-Software Interface*. Morgan Kaufman Publishers, 1997.

[3] J. Waldron. *Introduction to RISC assembly language*. Pearson Addison Wesley, 1998.