MIPS Coding Snippets

Prof. James L. Frankel
Harvard University

Version of 11:54 AM 26-Oct-2021

Loading a 32-bit constant into a register

```
# Example loading 12345678 decimal into register $t0
# 12345678 decimal == 0xbc614e
    lui     $t0, 0xbc                   # $t0 <- 0xbc0000
    ori     $t0, $t0, 0x614e            # $t0 <- 0xbc614e


# The above could also be written as
    lui     $t0, 188                    # $t0 <- 0xbc0000
    ori     $t0, $t0, 24910             # $t0 <- 0xbc614e


# The above could also be written as
    lui     $t0, (12345678>>16)         # $t0 <- 0xbc0000
    ori     $t0, $t0, (12345678&0xffff) # $t0 <- 0xbc614e
```

Loading a 32-bit constant (with low half zero) into a register

```
# Example loading 19070976 decimal into register $t0
# 19070976 decimal == 0x1230000
    lui     $t0, 0x123                    # $t0 <- 0x1230000


# The above could also be written as
    lui     $t0, 291                      # $t0 <- 0x1230000


# The above could also be written as
    lui     $t0, (19070976>>16)           # $t0 <- 0x1230000
```

Setting a low bit in a register

```
# Example setting bit 12 of register $t0
    ori     $t0, $t0, 0x1000            # set bit 12 in $t0


# The above could also be written as
    ori     $t0, $t0, (1<<12)           # set bit 12 in $t0
```

Setting a high bit in a register

```
# Example setting bit 30 of register $t0
# Modifies register $t1
    lui     $t1, 0x4000              # $t1 <- just bit 30 set
    or      $t0, $t0, $t1            # set bit 30 in $t0


# The above could also be written as
# Modifies register $t1
    lui     $t1, ((1<<30)>>16)       # $t1 <- just bit 30 set
    or      $t0, $t0, $t1            # set bit 30 in $t0
```

Clearing a low bit in a register

```
# Example clearing bit 12 of register $t0
# Modifies register $t1
    ori     $t1, $0, 0x1000          # $t1 <- just bit 12 set
    nor     $t1, $t1, $0             # $t1 <- ~0x1000 (just bit 12 cleared)
    and     $t0, $t0, $t1            # clear bit 12 in $t0


# The above could also be written as
# Modifies register $t1
    ori     $t1, $0, (1<<12)         # $t1 <- just bit 12 set
    nor     $t1, $t1, $0             # $t1 <- ~0x1000 (just bit 12 cleared)
    and     $t0, $t0, $t1            # clear bit 12 in $t0
```

Clearing a high bit in a register

```
# Example clearing bit 30 of register $t0
# Modifies register $t1
    lui     $t1, 0x4000              # $t1 <- just bit 30 set
    nor     $t1, $t1, $0             # $t1 <- just bit 30 cleared
    and     $t0, $t0, $t1            # clear bit 30 in $t0


# The above could also be written as
# Modifies register $t1
    lui     $t1, ((1<<30)>>16)       # $t1 <- just bit 30 set
    nor     $t1, $t1, $0             # $t1 <- just bit 30 cleared
    and     $t0, $t0, $t1            # clear bit 30 in $t0
```

Testing a low bit in a register

```
# Example determining the state of bit 12 of register $t0
# Modifies register $t1
    andi    $t1, $t0, 0x1000            # $t1 <- just bit 12 of $t0
    beq     $t1, $0, bitIsZero          # branch if bit 12 is zero
    …
bitIsZero:


# The above could also be written as
# Modifies register $t1
    andi    $t1, $t0, (1<<12)           # $t1 <- just bit 12 of $t0
    beq     $t1, $0, bitIsZero          # branch if bit 12 is zero
    …
bitIsZero:
```

Testing a high bit in a register

```
# Example determining the state of bit 30 of register $t0
# Modifies register $t1
      lui       $t1, 0x4000                 # $t1 <- just bit 30 set
      and       $t1, $t0, $t1               # $t1 <- just bit 30 of $t0
      beq       $t1, $0, bitIsZero          # branch if bit 30 is zero
      …
bitIsZero:
```

```
# The above could also be written as
# Modifies register $t1
      lui       $t1, ((1<<30)>>16)          # $t1 <- just bit 30 set
      and       $t1, $t0, $t1               # $t1 <- just bit 30 of $t0
      beq       $t1, $0, bitIsZero          # branch if bit 30 is zero
      …
bitIsZero:
```

Moving high half of register into low half

```
# Example copying the high half of register $t0 into register $t1 with zero fill
    srl     $t1, $t0, 16
```

Arithmetically moving high half of register into low half

```
# Example copying the high half of register $t0 into register $t1 with sign
#   extension
      sra       $t1, $t0, 16
```

Multiplying a register by a power of two

```
# Example multiplying register $t0 by 128 putting result into register $t1
    sll     $t1, $t0, 7
```

Multiplying a register by another register

```
# Example multiplying register $t0 by register $t1 putting result into register $t2
    mult    $t0, $t1                    # LO <- low word of product;
                                        # HI <- high word of product;
                                        # performed signed multiply
    mflo    $t2                         # $t2 <- low word of product

# Example multiplying register $t0 by register $t1 putting result into register $t2
    multu   $t0, $t1                    # LO <- low word of product;
                                        # HI <- high word of product;
                                        # performed unsigned multiply
    mflo    $t2                         # $t2 <- low word of product
```

Comparing the value in a register to another register (part 1 of 3)

```
# Example determining if register $t0 is equal to register $t1
    beq     $t0, $t1, regsAreEqual          # branch if $t0 == $t1
    …
regsAreEqual:


# Example determining if register $t0 is not equal to register $t1
    bne     $t0, $t1, regsAreNotEqual       # branch if $t0 != $t1
    …
regsAreNotEqual:
```

Comparing the value in a register to another register (part 2 of 3)

```
# Example determining if register $t0 is less than register $t1 (signed arith)
# ($t0 < $t1)  ==  (($t0-$t1) < 0)
# Modifies register $t2
        subu    $t2, $t0, $t1                   # $t2 <- ($t0 - $t1)
        bltz    $t2, regIsLT                    # branch if $t0 < $t1 (signed arith)
        …
regIsLT:


# Example determining if register $t0 is less than or equal to register $t1 (signed arith)
# ($t0 <= $t1)  ==  (($t0-$t1) <= 0)
# Modifies register $t2
        subu    $t2, $t0, $t1                   # $t2 <- ($t0 - $t1)
        blez    $t2, regIsLE                    # branch if $t0 <= $t1 (signed arith)
        …
regIsLE:
```

Comparing the value in a register to another register (part 3 of 3)

```
# Example determining if register $t0 is greater than register $t1 (signed arith)
# ($t0 > $t1)  ==  (($t0-$t1) > 0)
# Modifies register $t2
      subu     $t2, $t0, $t1                    # $t2 <- ($t0 - $t1)
      bgtz     $t2, regIsGT                     # branch if $t0 > $t1 (signed arith)
      …
regIsGT:


# Example determining if register $t0 is greater than or equal to register $t1 (signed
arith)
# ($t0 >= $t1)  ==  (($t0-$t1) >= 0)
# Modifies register $t2
      subu     $t2, $t0, $t1                    # $t2 <- ($t0 - $t1)
      bgez     $t2, regIsGE                     # branch if $t0 >= $t1 (signed arith)
      …
regIsGE:
```

How to branch to a target that is too far away

```
# Example determining if register $t0 is greater than register $t1 (signed arith)
# ($t0 > $t1)  ==  (($t0-$t1) > 0)
# Assumes that regIsGT is near (within -32K to +32K-1 instructions of the
#   instruction following the "bgtz")
# Modifies register $t2
        subu    $t2, $t0, $t1           # $t2 <- ($t0 - $t1)
        bgtz    $t2, regIsGT            # branch if $t0 > $t1 (signed arith)
        …
regIsGT:
```

```
# Example determining if register $t0 is greater than register $t1 (signed arith)
# ($t0 > $t1)  ==  (($t0-$t1) > 0)
# Assumes that regIsGT is far (*not* within -32K to +32K-1 instructions of the
#   instruction following the "bgtz")
# Modifies register $t2
        subu    $t2, $t0, $t1           # $t2 <- ($t0 - $t1)
        blez    $t2, regIsLE            # branch if $t0 <= $t1 (signed arith)
        j       regIsGT
regIsLE:
        …
regIsGT:
```

How to call a subroutine

```
# Example calling subroutine to compute $a0 ^ $a1
        ori     $a0, $0, 15                     # $a0 <- 15
        ori     $a1, $0, 3                      # $a1 <- 3
        jal     exp                             # $v0 <- 15 ^ 3

# Subroutine: exp
# Description: computes $a0 raised to the $a1 power by simple looping
# Parameters: $a0 is the base
#               $a1 is the exponent
# Results:      $v0 will be $a0 ^ $a1
# Side effects: $a1, HI, LO, and $ra will be overwritten
exp: ori        $v0, $0, 1                      # initial result is 1
        beq     $a1, $0, expZero                # loop is over, exponent is now zero
expLoop: mult $v0, $a0                          # (HI concat LO) <- running product * base
        mflo    $v0                             # update the running product
        addi    $a1, $a1, -1                    # decrement the exponent
        bne     $a1, $0, expLoop
expZero: jr     $ra
```

Use of Registers

- Register $zero always has the value 0. Storing a value into $zero has no effect.
- Register $at is reserved for the assembler (for pseudo instructions)
- Registers $v0 & $v1 are results of subroutines (and used to eval exprs)
- Registers $a0-$a3 are parameters to subroutines
- Registers $t0-$t9 are temporary registers (not saved by subroutines)
- Registers $s0-$s7 are saved registers (subroutines must preserve these)
- Registers $k0 & $k1 are reserved for the OS kernel
- Register $gp is used to point to a global data area
- Register $sp is the stack pointer
- Register $fp is the frame pointer
- Register $ra is the return address

Hardware Use of Registers

- Only two registers are special in the MIPS hardware architecture
  - Register $zero ($0) is special because… when read, it always has the value zero and when written, the writes have no effect
  - Register $ra (the return address register, $31) is special because… the JAL instruction always stores the return address into $ra
- The defined uses of all the other registers are just conventions for the assembly language programmer

Reading from Memory (word)

```
# Example loading a word from memory at address 12 past 0x10000000
#   into $t0
# Modifies register $t1
    lui     $t1, 0x1000                 # $t1 <- 0x10000000
    lw      $t0, 12($t1)                # $t0 <- loadWord($t1+12)
```

Reading from Memory (halfword)

```
# Example loading a halfword from memory at address 12 past 0x10000000
#   into $t0, zero extended
# Modifies register $t1
    lui     $t1, 0x1000         # $t1 <- 0x10000000
    lhu     $t0, 12($t1)        # $t0 <- loadHalf($t1+12) (zero extended)
```

```
# Example loading a halfword from memory at address 12 past 0x10000000
#   into $t0, sign extended
# Modifies register $t1
    lui     $t1, 0x1000         # $t1 <- 0x10000000
    lh      $t0, 12($t1)        # $t0 <- loadHalf($t1+12) (sign extended)
```

Reading from Memory (byte)

```
# Example loading a byte from memory at address 12 past 0x10000000
#   into $t0, zero extended
# Modifies register $t1
    lui     $t1, 0x1000              # $t1 <- 0x10000000
    lbu     $t0, 12($t1)             # $t0 <- loadByte($t1+12) (zero extended)
```

```
# Example loading a byte from memory at address 12 past 0x10000000
#   into $t0, sign extended
# Modifies register $t1
    lui     $t1, 0x1000              # $t1 <- 0x10000000
    lb      $t0, 12($t1)             # $t0 <- loadByte($t1+12) (sign extended)
```

Writing to Memory (word)

```
# Example storing a word in $t0 into memory at address 12 past 0x10000000
# Modifies register $t1
    lui     $t1, 0x1000              # $t1 <- 0x10000000
    sw      $t0, 12($t1)             # storeWord($t0, $t1+12)
```

Writing to Memory (halfword)

```
# Example storing a halfword in the low half of $t0 into memory at
#   address 12 past 0x10000000
# Modifies register $t1
    lui     $t1, 0x1000                 # $t1 <- 0x10000000
    sh      $t0, 12($t1)                # storeHalf($t0, $t1+12)
```

Writing to Memory (byte)

```
# Example storing a byte in the low byte of $t0 into memory at
#   address 12 past 0x10000000
# Modifies register $t1
    lui     $t1, 0x1000             # $t1 <- 0x10000000
    sb      $t0, 12($t1)            # storeByte($t0, $t1+12)
```

Assembler Features

- The assembly language programmer can rely on many features of the assembler
  - There is no need to use numerical memory addresses, the assembler will compute these
  - There is no need to determine the numerical offset for branch instructions, the assembler will compute these
  - The programmer may use expressions as operands if the assembler can compute the value of these at assembly-time
    - The operators in these expressions are C Programming Language operators
  - Labels may be used both for instruction addresses and for data addresses
  - Integers may be expressed in decimal, octal, or hexadecimal
  - Pound sign introduces comments
  - The assembler includes pseudo instructions for ease of programming (these are labeled with a dagger in the SPIM documentation)

Assembler Directives

- Directives to the assembler begin with a period
  - .text indicates that following lines are in the program/instruction section
  - .text <addr> indicates that following lines are in the program/instruction section beginning at address <addr>
  - .data indicates that following lines are in the data section
  - .data <addr> indicates that following lines are in the data section beginning at address <addr>
  - .globl <name> indicates that <name> is known outside this module (*i.e.,* has global linkage). <name> must be a label defined in the module. Each program must have a single label named "main" that has global linkage.
  - .byte <b1>, <b2>, … initializes successive bytes to <b1>, <b2>, …
  - .half <h1>, <h2>, … initializes successive halfwords to <h1>, <h2>, …
  - .word <w1>, <w2>, … initializes successive words to <w1>, <w2>, …
  - .space <n> reserves <n> bytes of memory (uninitialized)
  - .ascii <string> initializes successive bytes to the ASCII values of the characters in the string <string>
  - .asciiz <string> initializes successive bytes to the ASCII values of the characters in the string <string> followed by a null byte (*i.e.,* a byte with the value 0)
- Strings may include backslash escape notation for special characters

The Assembler Pseudo Instruction: la

- The assembler pseudo instruction “la” stands for load address
- It puts the value of the address of a label into a specified register (*i.e.,* it makes the specified register “point to” the labeled code or data)
- In essence it computes the high and low halfwords of the address and uses “lui” and “ori” instructions to load that address into a specified register
- Its form is

  la <destinationRegister>, <label/address>

Complete program example with a subroutine and system calls

```
            .text
            .globl          main
# Example calling subroutine to compute $a0 ^ $a1
main:       ori             $a0, $0, 15             # $a0 <- 15
            ori             $a1, $0, 3              # $a1 <- 3
            jal             exp                     # $v0 <- 15 ^ 3 (exp result)
            or              $s0, $v0, $0            # $s0 <- exp result
            ori             $v0, $0, 4              # $v0 <- print_string system call code
            la              $a0, outStr             # $a0 -> outStr
            syscall                                 # print the output string
            ori             $v0, $0, 1              # $v0 <- print_int system call code
            or              $a0, $s0, $0            # $a0 <- exp result
            syscall                                 # print the integer exp result
            ori             $v0, $0, 4              # $v0 <- print_string system call code
            la              $a0, newlineStr         # $a0 -> newline string
            syscall                                 # print a newline
            ori             $v0, $0, 10             # $v0 <- exit system call code
            syscall                                 # exit

# Subroutine: exp
# Description: computes $a0 raised to the $a1 power by simple looping
# Parameters: $a0 is the base
#                           $a1 is the exponent
# Results:                  $v0 will be $a0 ^ $a1
# Side effects: $a1, HI, LO, and $ra will be overwritten
exp:        ori             $v0, $0, 1              # initial result is 1
            beq             $a1, $0, expZero        # loop is over, exponent is now zero
expLoop: mult $v0, $a0      # (HI concat LO) <- running product * base
            mflo            $v0                     # update the running product
            addi            $a1, $a1, -1            # decrement the exponent
            bne             $a1, $0, expLoop
expZero: jr                 $ra

            .data
outStr:     .asciiz         “The product is “
newlineStr: .asciiz         “\n“
```