Deeper Assembly:
Addressing, Conditions, Branching, and Loops

Slides revised 2/13/2014 by Patrick Kelley

Overview

- **Data Transfer Instructions**
- Addition and Subtraction
- Indirect Addressing
- JMP and BRANCH Instructions

Data Transfer Instructions

- Operand Types
- Direct Memory Operands
- move/load/store Instructions
- Zero & Sign Extension

Operand Types

- Imm (immediate) – a constant value
- Label – address specified by a label
- R(t,s,d) – Contents of a register
- (R) – Contents of memory pointed to by a register.
- offset(R) – Contents pointed to by R + offset

Examples:

```
li   $v0, 0x4f        # Imm into register $v0
la   $a0, mynum       # label into register $a0
lw   $ra, 12($sp)     # data at $sp + 12 into $ra
```

Direct Memory Operands

- MIPS, unlike some architectures, does not support direct memory addressing
- Instead you must set up your memory accesses.
  - Point to the data by loading the address into a register: la $a0, myVar
  - Now you can access myVar with an offset to the address: lw $a1, 0($a0) # load myVar into a1
- Note that it was Ok that the offset was 0, but we could also use a non-zero offset if we needed, for instance to access elements of an array.

Move, Load, and Store

- Load and Store are the memory access operations
  - There are several variations of each based on data size and type. (lb, lbu, lwl, sh, sw, etc.)
  - There are also load variations that don't affect data directly (la, li)
- Move is used to transport data between registers only and never affects memory
- Variations on Move are use to access special registers.

Zero- and Sign- extension

- Extension is used when a small piece of data is stored into a larger space. (ex. byte into word.)
- The space to the left is filled with either zero or the sign bit.
- Address operations sign extend the offset before ORing it to the base.
- Arithmetic operations select which to use by type:
  - Unsigned operations zero-extend
  - Signed operations sign-extend

What's Next

- Data Transfer Instructions
- **Addition and Subtraction**
- Indirect Addressing
- JMP and BRANCH Instructions

ADD and SUB Instructions

- add or addu Rd, Rs, Rt
  - Logic: *destination* ← *source* Rs + source Rt
- sub or subu Rd, Rs, Rt
  - Logic: *destination* ← *source* Rs – source Rt
- addi or addiu Rd, Rs, Imm
  - Logic: *destination* ← *source* Rs + immediate
- Note that memory is not affected by these operations

Negate

- neg or negu Rd, Rs
  - Logic: *destination* $\leftarrow$ 0 - *source* Rs
  - negu does not give exception if you try to negate maximum negative
- Subtract operations are really a convenience, not needed since you can negate and add a number to get the same effect.
- However, negate in MIPS is implemented by doing a subtraction, so it actually would add a step.

What's Next

- Data Transfer Instructions
- Addition and Subtraction
- **Indirect Addressing**
- JMP and BRANCH Instructions

Indirect Addressing

- Recall that some operations take an offset and base as an operand
- We can store the base address of a data structure, like an array, in a register.
- Then we can refer to individual elements by adding an offset to the base address.
  - This works well when offsets are known and the elements are few, as in stack frames.
  - But since the offset must be a constant, it is cumbersome to use for long lists.
- An example is on the following slide:

Example: Using indirect addressing

```
# TITLE Add and Subtract              (AddSub2.s)

# This program adds and subtracts 32-bit integers.
# It uses offsets to get to the variables

          .data
# variables
Nums:     .word     0x1000
          .word     0x5000
          .word     0x3000
Sum:      .word     0

          .text
          .globl    main

main:               # start of the main procedure
          la        $a0, Nums # Put the base address in $a0
          lw        $t0, 0($a0)        # Put first number into $t0
          lw        $t1, 4($a0)        # Put second number into $t1
          lw        $t2, 8($a0)        # Put third number into $t2
          add       $t4, $t0, $t1      # Add first two numbers, put in $t4
          sub       $t4, $t4, $t2      # Subtract third number from result
          sw        $t4, Sum  # Put result in sum

          jr        $ra       # return to caller (exit program)
```

Indirect Addressing (cont.)

- A better way is to iterate through a list by adding to the base address.
- If the list contains complex data, we can create a list of base addresses (pointers) and iterate through those.
- Using base addresses as simple pointers (without an offset) also works well for structures like linked lists and queues.

What's Next

- Data Transfer Instructions
- Addition and Subtraction
- Indirect Addressing
- **JMP and BRANCH Instructions**

Program Flow

- A processor does operations sequentially
  - The Fetch – Decode – Execute cycle gets instructions in order
  - If we wish to change the program flow, we need a special instruction
- Jump instructions are unconditional
  - Equivalent to a 'goto' statement.
  - Used to call subroutines or functions
  - May jump to a label or address in register
- Branch instructions are tied to a condition
  - Similar in effect to an 'if – then – else' construct
  - Always branch to a label

Jump Instruction

- J is an unconditional jump to a label that is usually within the same procedure.
- Syntax: j *Label*
- Logic: IR $\leftarrow$ *Label*
- Example:

```
top:
     .
     .
     j top
```

Conditional Branching

- In some processors, each instruction sets status flags in the processor
  - Usually in a status register with a bit for each flag
  - Typical flags are sign, carry, overflow, zero, and parity
- There is a certain efficiency in that we don't have to test a result if the flags are already set
- Ex:

```
mov AX, 5
sub AX, 5
jz goHereIfZero
…                    # else continue on
```

Conditional Branching (cont.)

- MIPS tests conditions as part of the Branch instruction
- No need for the overhead of maintaining status flags
- Also fewer instructions to needed in the architecture
- But branch instructions take a little longer because they have to do the test first
- Ex:

```
li $S1, -5
add $S1, $S1, 5
beq $zero, $S1, goHereIfZero
...                 # else continue on
```

Applications (1 of 5)

- Task: Jump to a label if unsigned $V0 is greater than $V1
- Solution: Use bgtu

```
bgtu $V0, $V1, Larger
```

- Task: Jump to a label if signed $V0 is greater than $V1
- Solution: Use bgt

```
bgt  $V0, $V1, Larger
```

Applications (2 of 5)

- Jump to label L1 if unsigned $V0 is less than or equal to Val1

```
lw   $V1, Val1
bleu $V0, $V1, L1    # below or equal
```

- Jump to label L1 if signed $V0 is less than or equal to Val1

```
lw   $V1, Val1
ble  $V0, $V1, L1    # below or equal
```

Applications (3 of 5)

- Compare unsigned $V0 to $V1, and copy the larger of the two into a variable named Large

```
        sw   $V1, Large
        bleu $V0, $V1, Next
        sw   $V0, Large
Next:
```

- Compare signed $V0 to $V1, and copy the smaller of the two into a variable named Small

```
        sw   $V0, Small
        bge  $V0, $V1, Next
        sw   $V1, Small
Next:
```

Applications (4 of 5)

- Jump to label L1 if the memory word pointed to by $A0 equals Zero

```
lw  $V0, ($a0)
beq $V0, $0, L1
    **OR**
beqz $V0, L1
```

- Jump to label L2 if the word in memory pointed to by $A0 is even

```
lw    $V0, ($a0)
andi  $V0, $V0, 1
beqz  $V0, L2
```

Applications (5 of 5)

- Task: Jump to label L1 if bits 0, 1, and 3 in $S0 are all set.
- Solution: Clear all bits except bits 0, 1,and 3. Then compare the result with 0000000B in hex.

```
li $S1, 0x0000000B
and $S0, $S0, $S1        ; clear unwanted bits
beq $S0, $S1, L1         ; check remaining bits
```

Encrypting a String (do until)

The following loop uses the XOR instruction to transform every character in a string into a new value.

```
# The encryption key is 232, can be any byte value
# We'll use an arbitrary string size of 128
  .data
Buffer:  .space 129
bufSize: .word 128

  .text
    lw   $a0, bufSize            # loop counter
    la   $a1, Buffer             # point to buffer
L1:
    lbu  $a3, ($a1)              # get a byte
    xori $a3, $a3, 232           # translate a byte
    sb   $a3, ($a1)              # store translated byte
    addi $a0, $a0, -1            # decrement loop counter
    beqz $a0, Next               # if done, leave loop
    addi $a1, $a1, 1             # point to next byte
    j    L1                      # loop
Next:
```

Encrypting a String (while, for)

The following loop uses the XOR instruction to transform every character in a string into a new value.

```
# The encryption key is 232, can be any byte value
# We’ll use an arbitrary string size of 128
  .data
Buffer:  .space 129
bufSize: .word 128

  .text
    lw   $a0, bufSize            # loop counter
    la   $a1, Buffer             # point to buffer
L1:
    blez $a0, Next               # if done, leave loop
    lbu  $a3, ($a1)              # get a byte
    xori $a3, $a3, 232           # translate a byte
    sb   $a3, ($a1)              # store translated byte
    addi $a0, $a0, -1            # decrement loop counter
    addi $a1, $a1, 1             # point to next byte
    j    L1                      # loop
Next:
```

Block-Structured IF Statements

Assembly language programmers can easily translate logical statements written in C++/Java into assembly language. For example:

```
if( op1 == op2 )
  X = 1;
else
  X = 2;
```

```
    lw   $a0, op1
    lw   $a1, op2
    bne $a0, $a1, L1
    li   $a3, 1
    sw   $a3, X
    j    L2
L1: li   $a3, 2
    sw   $a3, X
L2:
```

Compound Expression with AND
(1 of 3)

- When implementing the logical AND operator, consider that HLLs use short-circuit evaluation
- In the following example, if the first expression is false, the second expression is skipped:

```
if (al > bl) AND (bl > cl)
  X = 1;
```

Compound Expression with AND

(2 of 3)

```
if (al > bl) AND (bl > cl)
   X = 1;
```

This is one possible implementation . . .

```
# al, bl, and cl are in $a0, $a1, and $a2
# repsectively; $a3 holds 1
    bgt $a0, $a1, L1          # first expression...
    j   next
L1:
    bgt $a1, $a2, L2          # second expression...
    j   next
L2:                           # both are true
    sw  $a3, X                # set X to 1
next:
```

*Note that the branches could have been bgtu.

Compound Expression with AND

(3 of 3)

```
if (al > bl) AND (bl > cl)
  X = 1;
```

But the following implementation uses 40% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:

```
# al, bl, and cl are in $a0, $a1, and $a2
# repsectively; $a3 holds 1
   ble $a0, $a1, next       # first expression...
                            # quit if false
   ble $a1, $a2, next       # second expression...
                            # quit if false
   sw  $a3, X               # both are true
next:
```

Compound Expression with OR
(1 of 2)

- When implementing the logical OR operator, consider that HLLs use short-circuit evaluation
- In the following example, if the first expression is true, the second expression is skipped:

```
if (al > bl) OR (bl > cl)
  X = 1;
```

Compound Expression with OR

(2 of 2)

```
if (al > bl) OR (bl > cl)
  X = 1;
```

We can use "fall-through" logic to keep the code as short as possible:

```
# al, bl, and cl are in $a0, $a1, and $a2
# repsectively; $a3 holds 1
    bgt $a0, $a1, L1        # is al > bl?
    bgt $a1, $a2, next      # no: is BL > CL?
    jbe next                # no: skip next statement
L1: sw  $a3, X              # set X to 1
next:
```