CS64 Lecture 7
Logic/Shift + I/O

Prof. Heather Zheng
HW 2 assigned today

• 4 questions
• Q1-3: submit via HW dropbox
• Q4: submit the complete program via turnin, using hw2

• Due: April 25, 4:30pm

• NOTE: Lab 1-3: connect-4 will be assigned later this week.
Review: Logical Operators

• Two basic logical operators:
  – AND: outputs 1 only if both inputs are 1
  – OR: outputs 1 if at least one input is 1

• In MIPS assembly, both of these accept exactly 2 inputs and produce 1 output
  – Again, rigid syntax, simpler hardware
Logical Operators

• Truth Table: standard table listing all possible combinations of inputs and resultant output for each

• Truth Table for AND and OR

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A AND B</th>
<th>A OR B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Logical Operators

• Instruction Names:
  – and, or: Both of these expect the third argument to be a register
  – andi, ori: Both of these expect the third argument to be an immediate

• MIPS Logical Operators are all **bitwise**, meaning that bit n of the output is produced by the respective bit n’s of the inputs, bit 1 by the bit 1’s, etc.
Question 1

What is the bit-wise OR of the following patterns:

0110 1101
0100 1010

- A. 0110 1111
- B. 0111 0110
- C. 0110 1001
- D. 1110 0110
Question 2

• What is the bit-wise AND of the following patterns:
  0110 1101
  0100 1010

• A. 0110 0110
• B. 0100 1000
• C. 0110 1001
• D. 1110 0110
Question 3

Which of the following instructions loads register $5$ with the bit pattern that represents positive decimal 48?

- A. ori $5,$0,0x48
- B. ori $5,$5,0x48
- C. ori $5,$0,48
- D. ori $0,$5,0x48
Question 4

- Which of the following instructions clears all the bits in register $8$ except the low order byte (the right most byte), which is unchanged?

- A. ori $8,$8,0x00FF
- B. ori $8,$0,0x00FF
- C. andi $8,$8,0xFFFF
- D. andi $8,$8,0x00FF
Uses for Logical Operators

• Note that anding a bit with 0 produces a 0 at the output while anding a bit with 1 produces the original bit.
• This can be used to create a mask.
  – Example:
    
    \[
    \begin{array}{cccccccccccc}
    1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
    0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
    1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
    \end{array}
    \]

    \[
    \begin{array}{cccccccccccc}
    1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
    1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
    \end{array}
    \]

  – The result of anding these:
    
    \[
    \begin{array}{cccccccccccc}
    0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
    1 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
    \end{array}
    \]
Uses for Logical Operators

• Similarly, note that oring a bit with 1 produces a 1 at the output while oring a bit with 0 produces the original bit.

• This can be used to force certain bits of a string to 1s.
  — For example, if $t0$ contains 0x12345678, then after this instruction:  
    ori $t0$, $t0$, 0xFFFF
  —... $t0$ contains 0x1234FFFF (e.g. the high-order 16 bits are untouched, while the low-order 16 bits are forced to 1s).
Shift Instructions

• MIPS shift instructions:
  1. \texttt{sll} (shift left logical): shifts left and \underline{fills emptied bits with 0s}
  2. \texttt{srl} (shift right logical): shifts right and \underline{fills emptied bits with 0s}
  3. \texttt{sra} (shift right arithmetic): shifts right and \underline{fills emptied bits by sign extending}
Shift Instructions

• Example: shift right arithmetic by 8 bits
  0001 0010 0011 0100 0101 0110 0111 1000

  0000 0000 0001 0010 0011 0100 0101 0110

• Example: shift right arithmetic by 8 bits
  1001 0010 0011 0100 0101 0110 0111 1000

  1111 1111 1001 0010 0011 0100 0101 0110
Question 5

Complete the program so that register $4$ receives the pattern in register $5$ shifted left logical by three positions.

ori $5$, $0$, 0x92AF  # put a bit pattern into register $5$

sll ____ , ____ , ____  # shift left logical by three, put result in register $4$

- A. sll $5$, $3$, $4$
- B. sll $3$, $5$, $4$
- C. sll $5$, $3$, $4$
- D. sll $4$, $5$, $3$
Question 6

Let’s say $t0$ holds the following 32-bit
0001 0010 0011 0100 0101 0110 0111 1000
aka: 0x12345678

After running the following 2 instructions, what happens to $t0$?

sll $t0,$t0,16
srl $t0,$t0,24
sll $t0,$t0,16
srl $t0,$t0,24

0001 0010 0011 0100 0101 0110 0111 1000
aka: 0x12345678

0001 0010 0011 0100 0101 0110 0111 1000
aka: 0x12345678

0101 0110 0111 1000
aka: 0x00000056

0000 0000 0000 0000
0101 0110
aka: 0x00000056

0000 0000 0000 0000 0000 0000
aka: 0x56780000
INPUT/OUTPUT
# System Calls Format

<table>
<thead>
<tr>
<th>service</th>
<th>call code</th>
<th>arguments</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>print integer</td>
<td>1</td>
<td>int in $a0</td>
<td></td>
</tr>
<tr>
<td>print float</td>
<td>2</td>
<td>float in $a0</td>
<td></td>
</tr>
<tr>
<td>print double</td>
<td>3</td>
<td>double in $a0</td>
<td></td>
</tr>
<tr>
<td>print string</td>
<td>4</td>
<td>addr of string in $a0</td>
<td></td>
</tr>
<tr>
<td><strong>read integer</strong></td>
<td><strong>5</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>read float</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read double</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>read string</td>
<td>8</td>
<td>$a0=buffer, $a1=length</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0 = amount</td>
<td></td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# add2.asm-- A program that computes and prints the sum of two numbers specified at runtime by the user.

# Registers used:
# $t0 - used to hold the first number.
# $t1 - used to hold the second number
# $t2 - used to hold the sum of the $t1 and $t2.
# $v0 - syscall parameter and return value.
# $a0 - syscall parameter.

main:

    ## Get first number from user, put into $t0.
    li  $v0, 5   # load syscall read_int into $v0.
    syscall      # make the syscall.
    move $t0, $v0  # move the number read into $t0.

    ## Get second number from user, put into $t1.
    li  $v0, 5   # load syscall read_int into $v0.
    syscall      # make the syscall.
    add $t1, $v0, $0  # move the number read into $t1.
    add $t2, $t0, $t1  # compute the sum.
# System Calls Format

<table>
<thead>
<tr>
<th>service</th>
<th>call code</th>
<th>arguments</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>print integer</td>
<td>1</td>
<td>int in $a0</td>
<td></td>
</tr>
<tr>
<td>print float</td>
<td>2</td>
<td>float in $a0</td>
<td></td>
</tr>
<tr>
<td>print double</td>
<td>3</td>
<td>double in $a0</td>
<td></td>
</tr>
<tr>
<td>print string</td>
<td>4</td>
<td>addr of string in $a0</td>
<td></td>
</tr>
<tr>
<td>read integer</td>
<td>5</td>
<td></td>
<td>int in $v0</td>
</tr>
<tr>
<td>read float</td>
<td>6</td>
<td></td>
<td>float in $v0</td>
</tr>
<tr>
<td>read double</td>
<td>7</td>
<td></td>
<td>double in $v0</td>
</tr>
<tr>
<td>read string</td>
<td>8</td>
<td>$a0=buffer, $a1=length</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0 = amount</td>
<td>addr in $v0</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Print out $t2.

add $a0, $t2, $0  # move the number to print into $a0.
li $v0, 1     # load syscall print_int into $v0.
syscall   # make the syscall.
li $v0, 10   # syscall code 10 is for exit.
syscall   # make the syscall.

# end of add2.asm.
## System Calls Format

<table>
<thead>
<tr>
<th>service</th>
<th>call code</th>
<th>arguments</th>
<th>results</th>
</tr>
</thead>
<tbody>
<tr>
<td>print integer</td>
<td>1</td>
<td>int in $a0</td>
<td></td>
</tr>
<tr>
<td>print float</td>
<td>2</td>
<td>float in $a0</td>
<td></td>
</tr>
<tr>
<td>print double</td>
<td>3</td>
<td>double in $a0</td>
<td></td>
</tr>
<tr>
<td><strong>print string</strong></td>
<td>4</td>
<td>addr of string in $a0</td>
<td></td>
</tr>
<tr>
<td>read integer</td>
<td>5</td>
<td></td>
<td>int in $v0</td>
</tr>
<tr>
<td>read float</td>
<td>6</td>
<td></td>
<td>float in $v0</td>
</tr>
<tr>
<td>read double</td>
<td>7</td>
<td></td>
<td>double in $v0</td>
</tr>
<tr>
<td><strong>read string</strong></td>
<td>8</td>
<td>$a0=buffer, $a1=length</td>
<td></td>
</tr>
<tr>
<td>sbrk</td>
<td>9</td>
<td>$a0 = amount</td>
<td>addr in $v0</td>
</tr>
<tr>
<td>exit</td>
<td>10</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assembly Program of the Day

.data
prompt: .asciiz "Please input your name (20 char max): ":
statestr: .space 21  # an assembly directive which reserves
#memory (indicated in bytes)

.text
main: li $v0, 4   # code for print_string goes in $v0
    la $a0, prompt  # $a0= address of string to be printed
    syscall  # print the prompt

li $v0, 8       # code for read_string goes in $v0
la $a0, statestr # address of the input buffer in memory goes
#in register $a0
li $a1, 20      # size of buffer goes in register $a1
syscall  # read the input
PROCEDURE
C functions

main() {
    int a,b,c;
    ...
    c = sum(a,b); /* a,b,c: $s0,$s1,$s2 */
    ...
}

/* really dumb sum function */
int sum(int x, int y) {
    return x+y;
}

What information must the compiler/programmer keep track of?

What instructions can accomplish this?
Procedure calls

Main Program

Function

loc:
Function Call Bookkeeping

• Registers play a major role in keeping track of information for function calls

• Register conventions:
  – Return address $ra
  – Arguments $a0, $a1, $a2, $a3
  – Return value $v0, $v1
  – Local variables $s0, $s1, …, $s7

• The stack is also used; more later
In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

C
}

C
int sum(int x, int y) {
    return x+y;
}

In MIPS, all instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.
Procedure calls

Main Program

... 
... 
... 
... 
... 
... 
... 
... 
... 
... 
... 

Function

loc:

... 

Sum(a,b)
Instruction Support for Functions

```c
... sum(a,b);... /* a,b:$s0,$s1 */
}
C int sum(int x, int y) {
    return x+y;
}
```

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>add $a0,$s0,$zero</td>
<td>x = a</td>
</tr>
<tr>
<td>1004</td>
<td>add $a1,$s1,$zero</td>
<td>y = b</td>
</tr>
<tr>
<td>1008</td>
<td>addi $ra,$zero,1016</td>
<td></td>
</tr>
<tr>
<td>$ra=1016</td>
<td>???</td>
<td></td>
</tr>
<tr>
<td>1012</td>
<td>j sum</td>
<td># jump to sum</td>
</tr>
<tr>
<td>1016</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>sum: add $v0,$a0,$a1</td>
<td></td>
</tr>
<tr>
<td>2004</td>
<td>jr $ra</td>
<td># new instruction</td>
</tr>
</tbody>
</table>
Instruction Support for Functions

... sum(a,b);... /* a,b:$s0,$s1 */
}

C int sum(int x, int y) {
    return x+y;
}

> Question: Why use jr here? Why not simply use j?

> Answer: sum might be called by many functions, so we can’t return to a fixed place. The calling proc to sum must be able to say “return here” somehow.

> 2000 sum: add $v0,$a0,$a1
> 2004 jr $ra    # new instruction
Instruction Support for Functions

• Single instruction to jump and save return address:
  
  \texttt{jump and link (jal)}

• Before:
  
  \begin{verbatim}
  1008  addi $ra,$zero,1016  #$ra=1016?
  1012  j  sum         #go to sum
  \end{verbatim}

• After:
  
  \begin{verbatim}
  1008  jal  sum  # $ra=1012?, go to sum
  \end{verbatim}

• Why have a \texttt{jal}? Make the common case fast:
  
  function calls are very common. Also, you don’t have to know where the code is loaded into memory with \texttt{jal}.
Instruction Support for Functions

• Syntax for `jal` (jump and link) is same as for `j` (jump):
  
  ```
  jal  label
  ```

• `jal` should really be called `laj` for “link and jump”:
  
  – Step 1 (link): Save address of next instruction into $ra (Why next instruction? Why not current one?)
  
  – Step 2 (jump): Jump to the given label
Instruction Support for Functions

• Syntax for \texttt{jr} (jump register):
  \begin{center}
  \texttt{jr register}
  \end{center}

• Instead of providing a label to jump to, the \texttt{jr} instruction provides a register which contains an address to jump to

• Only useful if we know exact address to jump

• Very useful for function calls:
  \begin{itemize}
  \item \texttt{jal} stores return address in register (\$ra)
  \item \texttt{jr \$ra} jumps back to that address
  \end{itemize}
Instruction Support for Functions

```c
... sum(a,b);... /* a,b:$s0,$s1 */
}

int sum(int x, int y) {
    return x+y;
}
```

address

```
1000  add $a0,$s0,$zero    # x = a
1004  add $a1,$s1,$zero    # y = b
1012  jal sum           # jump to sum
1016  ...
2000  sum:   add $v0,$a0,$a1
2004  jr   $ra            # new instruction
```
Steps for Making a Procedure Call

1) Save necessary values onto stack
2) Assign argument(s), if any
3) jal call
4) Restore values from stack
Procedure calls

caller

jal loc

 callee

jr $ra
Example

```c
main() {
    int i,j,k,m; /* i-m:$s0-$s3 */
    ...
    i = mult(j,k); ...
    m = mult(i,i); ... }

int mult (int mcand, int mlier){
    int product;
    product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1; }
    return product;
}
```
Example

```c
main() {
    int i,j,k,m; /* i-m:$s0-$s3 */
    ...
    i = mult(j,k); ...
    m = mult(i,i); ...  }

__start:
    add $a0,$s1,$0 # arg0 = j
    add $a1,$s2,$0 # arg1 = k
    jal mult # call mult
    add $s0,$v0,$0 # i = mult()
    ...

    add $a0,$s0,$0 # arg0 = i
    add $a1,$s0,$0 # arg1 = i
    jal mult # call mult
    add $s3,$v0,$0 # m = mult()
    ...

done
```
Example

int mult (int mcand, int mlier) {
    int product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;
}

int mult (int mcand, int mlier) {
    int product = 0;
    while (mlier > 0) {
        product += mcand;
        mlier -= 1;
    }
    return product;
}

mult: add $t0,$0,$0  # prod=0
Loop: slt $t1,$0,$a1  # mlr > 0?
      beq $t1,$0,Fin  # no=>$Fin
      add $t0,$t0,$a0  # prod+=mc
      addi $a1,$a1,-1  # mlr-=1
      j   Loop          # goto Loop

Fin: add $v0,$t0,$0  # $v0=prod
      jr   $ra         # return
Example

• Notes:
  – main function ends with done, not jr $ra, so there’s no need to save $ra onto stack
  – All variables used in main function are saved registers, so there’s no need to save these onto stack
Rules for Procedures

• Called with a `jal` instruction, returns with a `jr $ra`
• Accepts up to 4 arguments in `$a0, $a1, $a2 and $a3`
• Return value is always in `$v0` (and if necessary in `$v1`)
• Must follow register conventions (even in functions that only you will call)! So what are they?