CS 61C: Great Ideas in Computer Architecture

Introduction to Machine Language

Instructor: Justin Hsia
Review of Last Lecture

• C Memory Layout
  — Local variables disappear because Stack changes
  — Global variables don’t disappear because they are in Static Data
  — Dynamic memory available using `malloc` and `free`, but must be used VERY CAREFULLY

• Memory Management
  — K&R: first-fit, last-fit, best-fit for `malloc()`

• Many Common Memory Problems
Great Idea #1: Levels of Representation/Interpretation

Higher-Level Language Program (e.g. C)

Compiler

Assembly Language Program (e.g. MIPS)

Assembler

Machine Language Program (MIPS)

Machine Interpretation

Hardware Architecture Description (e.g. block diagrams)

Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)

temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;

lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)

0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111

Register File

ALU

Logic Circuit Description (Circuit Schematic Diagrams)
Agenda

• Machine Languages
• Registers
• Administrivia
• Instructions and Immediates
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to MIPS Practice
• Bonus: Additional Instructions
Machine Language (1/2)

• “Word” a computer understands: instruction

• Vocabulary of all “words” a computer understands: instruction set architecture (ISA)

• Why might you want the same ISA? Why might you want different ISAs?
  – e.g. iPhone and iPad use the same
  – e.g. iPhone and Macbook use different
Machine Language (2/2)

- **Single ISA**
  - Leverage common compilers, operating systems, etc.
  - BUT fairly easy to retarget these for different ISAs (e.g. Linux, gcc)

- **Multiple ISAs**
  - Specialized instructions for specialized applications
  - Different tradeoffs in resources used (e.g. functionality, memory demands, complexity, power consumption, etc.)
  - Competition and innovation is good, especially in emerging environments (e.g. mobile devices)
Why Study Assembly?

• Understand computers at a deeper level
  – Learn to write more compact and efficient code
  – Can sometimes hand optimize better than a compiler

• More sensible for minimalistic applications
  – e.g. distributed sensing and systems
  – Eliminating OS, compilers, etc. reduce size and power consumption
  – Embedded computers outnumber PCs!
Reduced Instruction Set Computing

• The early trend was to add more and more instructions to do elaborate operations – this became known as Complex Instruction Set Computing (CISC)

• Opposite philosophy later began to dominate: Reduced Instruction Set Computing (RISC)
  – Simpler (and smaller) instruction set makes it easier to build fast hardware
  – Let software do the complicated operations by composing simpler ones
Common RISC Simplifications

• **Fixed instruction length:**
  Simplifies fetching instructions from memory

• **Simplified addressing modes:**
  Simplifies fetching operands from memory

• **Few and simple instructions in the instruction set:**
  Simplifies instruction execution

• **Minimize memory access instructions (load/store):**
  Simplifies necessary hardware for memory access

• **Let compiler do heavy lifting:**
  Breaks complex statements into multiple assembly instructions
Mainstream ISAs

• Intel 80x86
  – Used in Macbooks and PCs
  – Found in Core i3, Core i5, Core i7, etc.

• Advanced RISC Machine (ARM)
  – Smart phone-like devices: iPhone, iPad, iPod, etc.
  – The most popular RISC (20x more common than 80x86)

• MIPS This is the ISA we will learn in CS61C
  – Networking equipment, PS2, PSP
  – Very similar to ARM
Agenda

- Machine Languages
- Registers
- Administrivia
- Instructions and Immediates
- Data Transfer Instructions
- Decision Making Instructions
- Bonus: C to MIPS Practice
- Bonus: Additional Instructions
Five Components of a Computer

• We begin our study of how a computer works!
  – Control
  – Datapath
  – Memory
  – Input
  – Output

• Registers are part of the Datapath
Computer Hardware Operands

• In high-level languages, number of variables limited only by available memory
• ISAs have a fixed, small number of operands called registers
  – Special locations built directly into hardware
  – **Benefit:** Registers are EXTREMELY FAST (faster than 1 billionth of a second)
  – **Drawback:** Operations can only be performed on these predetermined number of registers
MIPS Registers

• MIPS has 32 registers
  – Each register is 32 bits wide and hold a word
• Tradeoff between speed and availability
  – Smaller number means faster hardware but insufficient to hold data for typical C programs
• Registers have no type (C concept); the operation being performed determines how register contents are treated
MIPS Registers

• Register denoted by ‘$’ can be referenced by number ($0-$31) or name:
  – Registers that hold programmer variables:
    \$s0-$s7 $16-$23
  – Registers that hold temporary variables:
    \$t0-$t7 \$8-$15
    \$t8-$t9 \$24-$25
  – You’ll learn about the other 14 registers later

• In general, using register names makes code more readable
Agenda

• Machine Languages
• Registers
• Administrivia
• Instructions and Immediates
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to MIPS Practice
• Bonus: Additional Instructions
Administrivia

• HW2 posted, due this Sunday (7/1)
• Lab 3 posted, Lab 4 posted tomorrow
• Project 1 posted mid-week, due next Sunday (7/8)
  – MIPS Instruction Set Emulator
Agenda

• Machine Languages
• Registers
• Administrivia
• Instructions and Immediates
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to MIPS Practice
• Bonus: Additional Instructions
MIPS Green Sheet

- Contains MIPS instructions and lots of other useful information
  - [http://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf](http://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf)
  - Hard copy in textbook (will be provided on exams)

- Inspired by the IBM 360 “Green Card” from the late 1960’s and 1970’s
MIPS Instructions

• Instruction Syntax is rigid:

  \[ \text{op} \ \text{dst}, \ \text{src1}, \ \text{src2} \]

  – 1 operator, 3 operands
    • \text{op} = \text{operation name ("operator")}
    • \text{dst} = \text{register getting result ("destination")}
    • \text{src1} = \text{first register for operation ("source 1")}
    • \text{src2} = \text{second register for operation ("source 2")}

• Keep hardware simple via regularity
MIPS Instructions

• One operation per instruction, at most one instruction per line

• Assembly instructions are related to C operations ( =, +, −, *, /, &, |, etc.)
  – Must be, since C code decomposes into assembly!

• A single line of C may break up into several lines of MIPS
MIPS Instructions Example

• Your very first instructions!
  (assume here that the variables \( a, b, \) and \( c \) are assigned to
  registers \( $s1, $s2, \) and \( $s3, \) respectively)

• Integer Addition (\texttt{add})
  \begin{itemize}
  \item C: \hspace{1cm} a = b + c
  \item MIPS: \hspace{1cm} \texttt{add} \hspace{0.5cm} $s1, \hspace{0.5cm} $s2, \hspace{0.5cm} $s3
  \end{itemize}

• Integer Subtraction (\texttt{sub})
  \begin{itemize}
  \item C: \hspace{1cm} a = b - c
  \item MIPS: \hspace{1cm} \texttt{sub} \hspace{0.5cm} $s1, \hspace{0.5cm} $s2, \hspace{0.5cm} $s3
  \end{itemize}
MIPS Instructions Example

• Suppose $a \rightarrow s0, b \rightarrow s1, c \rightarrow s2, d \rightarrow s3, \text{ and } e \rightarrow s4$. Convert the following C statement to MIPS:

\[
a = (b + c) - (d + e);
\]

add $t1, s3, s4
add $t2, s1, s2
sub $s0, $t2, $t1

Ordering of instructions matters (must follow order of operations)
Utilize temporary registers
Comments in MIPS

- Comments in MIPS follow hash mark (#) until the end of line
  - Improves readability and helps you keep track of variables/registers!

```
add $t1, $s3, $s4 # $t1=d+e
add $t2, $s1, $s2 # $t2=b+c
sub $s0, $t2, $t1 # a=(b+c)-(d+e)
```
The Zero Register

• Zero appears so often in code and is so useful that it has its own register!

• Register zero (\$0 or \$zero) always has the value 0 and cannot be changed!
  – i.e. any instruction with \$0 as dst has no effect

• Example Uses:
  – add \$s3, \$0, \$0  # c=0
  – add \$s1, \$s2, \$0  # a=b
Immediates

• Numerical constants are called **immediates**
• Separate instruction syntax for immediates:
  
  ```
  opi dst, src, imm
  ```

  – Operation names end with ‘i’, replace 2\textsuperscript{nd} source register with an immediate (check Green Sheet for un/signed)

• Example Uses:
  
  – `addi $s1, $s2, 5  # a=b+5`
  – `addi $s3, $s3, 1  # c++`

• **Why no `subi` instruction?**
Agenda

• Machine Languages
• Registers
• Administrivia
• Instructions and Immediates
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to MIPS Practice
• Bonus: Additional Instructions
Five Components of a Computer

• Data Transfer instructions are between registers (Datapath) and Memory
  – Allow us to fetch and store operands in memory
Data Transfer

• C variables map onto registers; What about large data structures like arrays?
  – Don’t forget memory, our one-dimensional array indexed by addresses starting at 0
• MIPS instructions only operate on registers!
• Specialized data transfer instructions move data between registers and memory
  – Store: register TO memory
  – Load: register FROM memory
Data Transfer

• Instruction syntax for data transfer:

\[ \text{op reg, off(bAddr)} \]

- \( \text{op} \) = operation name ("operator")
- \( \text{reg} \) = register for operation source or destination
- \( \text{bAddr} \) = register with pointer to memory ("base address")
- \( \text{off} \) = address offset (immediate) in bytes ("offset")

• Accesses memory at address \( \text{bAddr} + \text{off} \)

• **Reminder**: A register holds a word of raw data (no type) – make sure to use a register (and offset) that point to a valid memory address
Memory is Byte-Addressed

• What was the smallest data type we saw in C?
  – A char, which was a *byte* (8 bits)
  – Everything in multiples of 8 bits (e.g. 1 word = 4 bytes)

• Memory addresses are indexed by *bytes*, not words

• *Word addresses* are 4 bytes apart
  – Word addr is same as left-most byte
  – Offsets are multiples of 4 to be “word-aligned”

• Pointer arithmetic not done for you in assembly
  – Must take data size into account yourself
Data Transfer Instructions

• **Load Word** \((\text{lw})\)
  
  – Takes data at address \(\text{bAddr}+\text{off}\) FROM memory and places it into \(\text{reg}\)

• **Store Word** \((\text{sw})\)
  
  – Takes data in \(\text{reg}\) and stores it TO memory at address \(\text{bAddr}+\text{off}\)

• **Example Usage:**

  ```
  # addr of int A[] -> $s3, a -> $s0
  lw  $t0,12($s3)  # $t0=A[3]
  add $t0,$s2,$t0  # $t0=A[3]+a
  ```

6/25/2012 Summer 2012 -- Lecture #5
Registers vs. Memory

• What if more variables than registers?
  – Keep most frequently used in registers and move the rest to memory (called *spilling* to memory)

• Why not all variables in memory?
  – Smaller is faster: registers 100-500 times faster
  – Registers more versatile
    • In 1 arithmetic instruction: read 2 operands, perform 1 operation, and 1 write
    • In 1 data transfer instruction: 1 read/write, no operation
Great Idea #3: Principle of Locality/Memory Hierarchy

- Processor
- CPU
- Processor Register
- CPU Cache
  - Level 1 (L1) Cache
  - Level 2 (L2) Cache
  - Level 3 (L3) Cache
- Physical Memory
  - Random Access Memory (RAM)
- Solid State Memory
  - Non-Volatile Flash-Based Memory
- Virtual Memory
  - File-Based Memory
- EDO, SD-RAM, DDR-SDRAM, RD-RAM and More...
- SSD, Flash Drive
- Mechanical Hard Drives

Registers

Memory
**Question:** Which of the following is TRUE?

- add $t0,$t1,4($t2) is valid MIPS
- Can byte address 8GB with a MIPS word
- off must be a multiple of 4 for lw $t0,off($s0) to be valid
- If MIPS halved the number of registers available, it would be twice as fast
Chars and Strings

• **Recall:** A string is just an array of characters and a `char` in C uses 8-bit ASCII

• **Method 1:** Move words in and out of memory using bit-masking and shifting
  
  ```
  lw    $s0,0($s1)
  andi  $s0,$s0,0xFF # lowest byte
  ```

• **Method 2:** Load/store byte instructions
  
  ```
  lb    $s0,0($s1)
  sb    $s0,1($s1)
  ```

  Offsets no longer need to be multiples of 4
Byte Instructions

• `lb/sb` utilize the least significant byte of the register
  – On `sb`, upper 24 bits are ignored
  – On `lb`, upper 24 bits are filled by sign-extension

• For example, let `*(\$s0) = 0x00000180:`
  
  ```
  lb  \$s1,1(\$s0)  \#  \$s1=0x00000001
  lb  \$s2,0(\$s0)  \#  \$s2=0xFFFFFFFFF80
  sb  \$s2,2(\$s0)  \#  *(\$s0)=0x00800180
  ```

• Normally you don’t want to sign-extend chars
  – Use `lbu` (load byte unsigned)
Endianness

- **Big Endian:** Most-significant byte at least address of word
  - word address = address of most significant byte
- **Little Endian:** Least-significant byte at least address of word
  - word address = address of least significant byte

![Binary representation of big and little endian](image)

- MIPS is bi-endian (can go either way)
  - Using MARS simulator in lab, which is little endian

- Why is this confusing?
  - Data stored in reverse order than you write it out!
  - Data \(0x01020304\) stored as \(04\ 03\ 02\ 01\) in memory

6/25/2012 Summer 2012 -- Lecture #5
Get to Know Your Staff

• Category: Games
Agenda

• Machine Languages
• Registers
• Administrivia
• Instructions and Immediates
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to MIPS Practice
• Bonus: Additional Instructions
Computer Decision Making

• In C, we had control flow
  – Outcomes of comparative/logical statements determined which blocks of code to execute

• In MIPS, we can’t define blocks of code; all we have are labels
  – Defined by text followed by a colon (e.g. main:) and refers to the instruction that follows
  – Generate flow control by jumping labels
  – C has these too, but they are considered bad style
Decision Making Instructions

- **Branch If Equal (beq)**
  - `beq reg1, reg2, label`
  - If value in `reg1` = value in `reg2`, go to `label`

- **Branch If Not Equal (bne)**
  - `bne reg1, reg2, label`
  - If value in `reg1` ≠ value in `reg2`, go to `label`

- **Jump (j)**
  - `j label`
  - Unconditional jump to `label`
Breaking Down the If Else

C Code:
if (i==j) {
    a = b /* then */
} else {
    a = -b /* else */
}

In English:
• If TRUE, execute the THEN block
• If FALSE, execute the ELSE block

MIPS (beq):
# i→$s0, j→$s1
# a→$s2, b→$s3
beq $s0,$s1,???
???

sub $s2, $0, $s3
j  end
then:
add $s2, $s3, $0
end:

This label unnecessary
Breaking Down the If Else

C Code:

```c
if(i==j) {
    a = b  /* then */
} else {
    a = -b /* else */
}
```

In English:

• If TRUE, execute the THEN block
• If FALSE, execute the ELSE block

MIPS (bne):

```mips
# i→$s0, j→$s1
# a→$s2, b→$s3
bne $s0,$s1,???
???
add $s2, $s3, $0
j   end
else:
sub $s2, $0, $s3
end:
```
Loops in MIPS

• There are three types of loops in C:
  – while, do...while, and for
  – Each can be rewritten as either of the other two, so the same concepts of decision-making apply

• You will examine how to write these in MIPS in discussion

• **Key Concept:** Though there are multiple ways to write a loop in MIPS, the key to decision-making is the conditional branch
Question: Which of the following is FALSE? (and if TRUE, try writing it out)

- We can make an unconditional branch from a conditional branch instruction
- We can make a loop with just j (no beq or bne)
- We can make a for loop without using j
- Every control flow segment written with beq can be written in the same number of lines with bne
Summary

• Computers understand the instructions of their ISA

• RISC Design Principles
  – Smaller is faster, keep it simple

• MIPS Registers: $s0–$s7, $t0–$t9, $0

• MIPS Instructions
  – Arithmetic: add, sub, addi
  – Data Transfer: lw, sw, lb, sb, lbu
  – Branching: beq, bne, j

• Memory is byte-addressed
You are responsible for the material contained on the following slides, though we may not have enough time to get to them in lecture. They have been prepared in a way that should be easily readable and the material will be touched upon in the following lecture.
Agenda

• Machine Languages
• Registers
• Administrivia
• Instructions and Immediates
• Data Transfer Instructions
• Decision Making Instructions

• Bonus: C to MIPS Practice
• Bonus: Additional Instructions
C to MIPS Practice

• Let’s put our all of our new MIPS knowledge to use in an example: “Fast String Copy”

• C code is as follows:

/* Copy string from p to q */
char *p, *q;
while((*q++ = *p++) != '\0') ;

• What do we know about its structure?
  – Single while loop
  – Exit condition is an equality test
C to MIPS Practice

• Start with code skeleton:

```c
# copy String p to q
# p→$s0, q→$s1 (pointers)
Loop:
    # $t0 = *p
    # *q = $t0
    # p = p + 1
    # q = q + 1
    # if *p==0, go to Exit
    j Loop
Exit:
```

- N chars in p => N*6 instructions
C to MIPS Practice

• Fill in lines:

# copy String p to q  
# p→$s0, q→$s1 (pointers)

Loop: lb $t0,0($s0)  # $t0 = *p
    sb $t0,0($s1)  # *q = $t0
    addi $s0,$s0,1  # p = p + 1
    addi $s1,$s1,1  # q = q + 1
    beq $t0,$0,Exit  # if *p==0, go to Exit
    j Loop  # go to Loop

Exit:
C to MIPS Practice

• Finished code:

```assembly
# copy String p to q
# p→$s0, q→$s1 (pointers)
Loop: lb $t0,0($s0)   # $t0 = *p
     sb $t0,0($s1)    # *q = $t0
addi $s0,$s0,1      # p = p + 1
addi $s1,$s1,1      # q = q + 1
beq $t0,$0,Exit     # if *p==0, go to Exit
j Loop              # go to Loop
Exit:               # N chars in p => N*6 instructions
```
C to MIPS Practice

• Alternate code using bne:

```
# copy String p to q
# p→$s0, q→$s1 (pointers)
Loop:  lb   $t0,0($s0)  # $t0 = *p
       sb   $t0,0($s1)  # *q = $t0
       addi $s0,$s0,1  # p = p + 1
       addi $s1,$s1,1  # q = q + 1
       bne  $t0,$0,Loop # if *p!=0, go to Loop
```

# N chars in p => N*5 instructions
Agenda

• Machine Languages
• Registers
• Administrivia
• Instructions and Immediates
• Data Transfer Instructions
• Decision Making Instructions
• Bonus: C to MIPS Practice
• **Bonus: Additional Instructions**
MIPS Arithmetic Instructions

- The following commands place results in the special registers \( HI \) and \( LO \)
  - Access these values with “move from HI” \( (mfhi \ dst) \) and “move from LO” \( (mflo \ dst) \)

- **Multiplication** \( (\text{mult}) \)
  - \( \text{mult} \ src1,src2 \)
  - \( src1*src2: \) lower 32-bits in \( LO \), upper 32-bits in \( HI \)

- **Division** \( (\text{div}) \)
  - \( \text{div} \ src1,src2 \)
  - \( src1/src2: \) puts quotient in \( LO \), remainder in \( HI \)
MIPS Arithmetic Instructions

• Example:

# mod using div: $s2 = $s0 mod $s1
mod:
div $s0,$s1 # LO = $s0/$s1
mfhi $s2 # HI = $s0 mod $s1
Arithmetic Overflow

• **Recall:** Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers
  – i.e. not enough bits to represent answer
• MIPS detects overflow *(throws error)*
  – Arithmetic “unsigned” instructions ignore overflow

<table>
<thead>
<tr>
<th>Overflow Detection</th>
<th>No Overflow Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>add    dst,src1,src2</td>
<td>addu    dst,src1,src2</td>
</tr>
<tr>
<td>addi   dst,src1,src2</td>
<td>addiu   dst,src1,src2</td>
</tr>
<tr>
<td>sub    dst,src1,src2</td>
<td>subu    dst,src1,src2</td>
</tr>
</tbody>
</table>
Arithmetic Overflow

• Example:

# $s0=0x80000000, $s1=0x1
add   $t0,$s0,$s0 # overflow (error)
addu  $t1,$s0,$s0  # $t1=0
addi  $t2,$s0,-1   # overflow (error)
addiu $t2,$s0,-1   # $t3=0x7FFFFFFF
sub   $t4,$s0,$s1  # overflow (error)
subu  $t5,$s0,$s1  # $t5=0x7FFFFFFF

Recall: this is the most negative number
# MIPS Bitwise Instructions

**Note:** $a \rightarrow s1$, $b \rightarrow s2$, $c \rightarrow s3$

<table>
<thead>
<tr>
<th>Instruction</th>
<th>C</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>And</td>
<td>$a = b \ &amp; \ c;$</td>
<td>and $s1, s2, s3$</td>
</tr>
<tr>
<td>And Immediate</td>
<td>$a = b \ &amp; \ 0x1;$</td>
<td>andi $s1, s2, 0x1$</td>
</tr>
<tr>
<td>Or</td>
<td>$a = b \</td>
<td>\ c;$</td>
</tr>
<tr>
<td>Or Immediate</td>
<td>$a = b \</td>
<td>\ 0x5;$</td>
</tr>
<tr>
<td>Not Or</td>
<td>$a = \neg (b \</td>
<td>\ c);$</td>
</tr>
<tr>
<td>Exclusive Or</td>
<td>$a = b \ ^\ c;$</td>
<td>xor $s1, s2, s3$</td>
</tr>
<tr>
<td>Exclusive Or Immediate</td>
<td>$a = b \ ^\ 0xF;$</td>
<td>xori $s1, s2, 0xF$</td>
</tr>
</tbody>
</table>
Shifting Instructions

• In binary, shifting an unsigned number left is the same as multiplying by the corresponding power of 2
  – Shifting operations are faster
  – Does not work with shifting right/division
• **Logical shift**: Add zeros as you shift
• **Arithmetic shift**: Sign-extend as you shift
  – Only applies when you shift right (preserves sign)
• Can shift by immediate or value in a register
# Shifting Instructions

<table>
<thead>
<tr>
<th>Instruction Name</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift Left Logical</td>
<td><code>sll $s1,$s2,1</code></td>
</tr>
<tr>
<td>Shift Left Logical Variable</td>
<td><code>sllv $s1,$s2,$s3</code></td>
</tr>
<tr>
<td>Shift Right Logical</td>
<td><code>srl $s1,$s2,2</code></td>
</tr>
<tr>
<td>Shift Right Logical Variable</td>
<td><code>srlv $s1,$s2,$s3</code></td>
</tr>
<tr>
<td>Shift Right Arithmetic</td>
<td><code>sra $s1,$s2,3</code></td>
</tr>
<tr>
<td>Shift Right Arithmetic Variable</td>
<td><code>srav $s1,$s2,$s3</code></td>
</tr>
</tbody>
</table>

- When using immediate, only values 0-31 are accepted
- When using variable, only lowest 5 bits are used (read as unsigned)
Shifting Instructions

# sample calls to shift instructions
addi $t0,$0 ,-256  # $t0=0xFFFFFFFF00
sll  $s0,$t0,3     # $s0=0xFFFFFFFF800
srl  $s1,$t0,8     # $s1=0x00FFFFFFFF
sra  $s2,$t0,8     # $s2=0xFFFFFFFF

addi $t1,$0 ,-22   # $t1=0xFFFFFFFFFEA
                  # low 5: 0b0101010
sllv $s3,$t0,$t1   # $s3=0xFFFC0000
                  # same as sll $s3,$t0,10
Shifting Instructions

• Example 1:

```
# lb using lw:  lb $s1,1($s0)
lw       $s1,0($s0)  # get word
andi      $s1,$s1,0xFF00 # get 2nd byte
srl       $s1,$s1,8    # shift into lowest
```
Shifting Instructions

• Example 2:

```assembly
# sb using sw: sb $s1,3($s0)
lw   $t0,0($s0)  # get current word
andi $t0,$t0,0xFFFFFFFF # zero top byte
sll  $t1,$s1,24  # shift into highest
or   $t0,$t0,$t1 # combine
sw   $t0,0($s0)  # store back
```
Shifting Instructions

• Extra for Experts:
  – Rewrite the two preceding examples to be more general
  – Assume that the byte offset (e.g. 1 and 3 in the examples, respectively) is contained in $s_2$

• Hint:
  – The variable shift instructions will come in handy
  – Remember, the offset can be negative