Topic 2: Instructions
Part A: Basic Concepts
These slides are mostly taken verbatim, or with minor changes, from those prepared by
Mary Jane Irwin (www.cse.psu.edu/~mji)
of The Pennsylvania State University
[Adapted from Computer Organization and Design, 4th Edition,
Patterson & Hennessy, © 2008, MK]
Key to the Slides

- The source of each slide is coded in the footer on the right side:
  - **Irwin CSE331 PSU** = slide by Mary Jane Irwin from the course CSE331 (Computer Organization and Design) at Pennsylvania State University.
  - **Irwin CSE431 PSU** = slide by Mary Jane Irwin from the course CSE431 (Computer Architecture) at Pennsylvania State University.
  - **Hegner UU** = slide by Stephen J. Hegner at Umeå University.
This course is all about how computers work

But what do we mean by a computer?

- Different types: embedded, laptop, desktop, server, supercomputer
- Different uses: robotics, graphics, finance, genomics,…
- Different manufacturers: Intel, IBM, AMD, ARM, Freescale, Fujitsu, TI, Sun (Oracle) , MIPS, NEC, …
- Different underlying technologies and different costs!

Best way to learn:

- Focus on a specific instance and learn how it works
- While learning general principles and historical perspectives
Below the Program

- **System software**
  - Operating system – supervising program that interfaces the user’s program with the hardware (e.g., Linux, MacOS, Windows)
    - Handles basic input and output operations
    - Allocates storage and memory
    - Provides for protected sharing among multiple applications
  - Compiler – translate programs written in a high-level language (e.g., C, Java) into instructions that the hardware can execute
Advantages of High-Level Languages?

- Higher-level languages
  - Allow the programmer to think in a more natural language and for their intended use (Fortran for scientific computation, Cobol for business programming, Lisp for symbol manipulation, Java for web programming, …)
  - Improve programmer productivity – more understandable code that is easier to debug and validate
  - Improve program maintainability
  - Allow programs to be independent of the computer on which they are developed (compilers and assemblers can translate high-level language programs to the binary instructions of any machine)
  - Emergence of optimizing compilers that produce very efficient assembly code optimized for the target machine

- As a result, very little programming is done today at the assembler level
Machine Organization

- Capabilities and performance characteristics of the principal Functional Units (FUs)
  - e.g., register file, arithmetic-logic unit (ALU), multiplexors, memories, ...

- The ways those FUs are interconnected
  - e.g., buses

- Logic and means by which information flow between FUs is controlled

- The machine’s Instruction Set Architecture (ISA)
Instruction Set Architecture (ISA)

- ISA, or simply architecture – the abstract interface between the hardware and the lowest level software that encompasses all the information necessary to write a machine language program, including instructions, registers, memory access, I/O, …
  - Enables implementations of varying cost and performance to run identical software

- The combination of the basic (user portion of the) instruction set (the ISA) and the operating system interface is called the application binary interface (ABI)
  - Defines a standard for binary portability across computers.
Two Key Principles of Machine Design

1. Instructions are represented as numbers and, as such, are indistinguishable from data

2. Programs are stored in alterable memory (that can be read or written to) just like data

- Stored-program (von Neumann) concept
  - Programs can be shipped as files of binary numbers – binary compatibility
  - Computers can inherit ready-made software provided they are compatible with an existing ISA – leads industry to align around a small number of ISAs

Memory

<table>
<thead>
<tr>
<th>Accounting prg (machine code)</th>
</tr>
</thead>
<tbody>
<tr>
<td>C compiler (machine code)</td>
</tr>
<tr>
<td>Payroll data</td>
</tr>
<tr>
<td>Source code in C for Acct program</td>
</tr>
</tbody>
</table>
Assembly Language Instructions

- The language of the machine
  - Want an ISA that makes it easy to build the hardware and the compiler (whose job it is to translate programs written in a high level language (like C) to assembly code) while maximizing performance and minimizing cost

- Our target: the MIPS ISA
  - similar to other ISAs developed since the 1980's
  - used by Broadcom, Cisco, NEC, Nintendo, Sony, ...

*Design goals: maximize performance, minimize cost, reduce design time (time-to-market), minimize power consumption, maximize reliability*
RISC - Reduced Instruction Set Computer

- RISC philosophy
  - fixed instruction lengths
  - load-store instruction sets
  - limited number of addressing modes
  - limited number of operations

- MIPS, Sun SPARC, HP PA-RISC, IBM PowerPC …

- Instruction sets are measured by how well compilers can use them as opposed to how well assembly language programmers can use them

- CISC (C for complex), e.g., Intel x86
The Four RISC Design Principles

1. Simplicity favors regularity.
2. Smaller is faster.
3. Make the common case fast.
4. Good design demands good compromises.
MIPS Arithmetic Instruction

- MIPS assembly language arithmetic statement
  
  ```
  add $t0, $s1, $s2
  sub $t0, $s1, $s2
  ```

- Each arithmetic instruction performs only **one** operation
- Each arithmetic instruction specifies exactly **three** operands

  
  destination ← source1 op source2

- Operand order is fixed (the destination is specified first)
- The operands are contained in the datapath’s **register file** ($t0, s1, s2$)
Compiling More Complex Statements

- Assuming variable b is stored in register $s1, c is stored in $s2, and d is stored in $s3 and the result is to be left in $s0, what is the assembler equivalent to the C statement

\[ h = (b - c) + d \]

\[
\text{sub } \$t0, \$s1, \$s2 \\
\text{add } \$s0, \$t0, \$s3
\]
MIPS Register File

- Operands of arithmetic instructions must be from a limited number of special locations contained in the datapath’s register file.
  - Thirty-two 32-bit registers
    - Two read ports
    - One write port

- Registers are
  - Faster than main memory
    - Smaller is faster & Make the common case fast
  - Easy for a compiler to use
    - e.g., \((A*B) - (C*D) - (E*F)\) can do multiplies in any order
  - Improves code density
    - Since register are named with fewer bits than a memory location

- Register addresses are indicated by using $
# MIPS Register Naming Convention

<table>
<thead>
<tr>
<th>Nick Name</th>
<th>Register Number</th>
<th>Usage</th>
<th>Preserve on call?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0 (hardware)</td>
<td>n.a.</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
<td>n.a.</td>
</tr>
<tr>
<td>$v0 - $v1</td>
<td>2-3</td>
<td>returned values</td>
<td>no</td>
</tr>
<tr>
<td>$a0 - $a3</td>
<td>4-7</td>
<td>arguments</td>
<td>yes</td>
</tr>
<tr>
<td>$t0 - $t7</td>
<td>8-15</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$s0 - $s7</td>
<td>16-23</td>
<td>saved values</td>
<td>yes</td>
</tr>
<tr>
<td>$t8 - $t9</td>
<td>24-25</td>
<td>temporaries</td>
<td>no</td>
</tr>
<tr>
<td>$k0 - $k1</td>
<td>26-27</td>
<td>reserved for OS</td>
<td>n.a.</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>global pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
<td>yes</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return addr (hardware)</td>
<td>yes</td>
</tr>
</tbody>
</table>
Registers vs. Memory

- Arithmetic instructions *operands* must be in registers
  - But only thirty-two registers are provided

- The compiler associates variables with registers

What about programs with lots of variables?
- Memory is a large, single-dimensional array.
- A memory address acts as the index into the memory array.
Word Addresses vs Byte Addresses

- Alignment restriction - the memory address of a word *must* be on natural word boundaries (a multiple of 4 in MIPS-32)
  - But since 8-bit bytes are so useful, most architectures also support addressing individual bytes in memory

- Big Endian: most-significant byte is at the word address
  - IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA

- Little Endian: least-significant byte is at the word address
  - Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

- For 0a0b0c0d hex stored in a word beginning at byte address α:

<table>
<thead>
<tr>
<th>Big endian:</th>
<th>0a</th>
<th>0b</th>
<th>0c</th>
<th>0d</th>
</tr>
</thead>
<tbody>
<tr>
<td>Little endian:</td>
<td>0d</td>
<td>0c</td>
<td>0b</td>
<td>0a</td>
</tr>
<tr>
<td>Memory location:</td>
<td>α</td>
<td>α+1</td>
<td>α+2</td>
<td>α+3</td>
</tr>
</tbody>
</table>
Accessing Memory

- MIPS has two basic data transfer instructions for accessing memory (assume $s3 holds $24_{10}$)

```plaintext
lw $t0, 4($s3)  # load word from memory $28_{10}$
sw $t0, 8($s3)  # store word to memory $32_{10}$
```

- The data transfer instruction must specify
  - where in memory to read from (load) or write to (store) – memory address
  - where in the register file to write to (load) or read from (store) – register destination (source)

- The memory address is formed by summing the constant portion of the instruction and the contents of the second register
The memory address is formed by summing the constant portion of the instruction and the contents of the second (base) register.

$s3$ holds 8

```
Memory

<table>
<thead>
<tr>
<th>32 bit Data</th>
<th>Word Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>...0110</td>
<td>24</td>
</tr>
<tr>
<td>...0101</td>
<td>20</td>
</tr>
<tr>
<td>...1100</td>
<td>16</td>
</tr>
<tr>
<td>...0001</td>
<td>12</td>
</tr>
<tr>
<td>...0010</td>
<td>8</td>
</tr>
<tr>
<td>...1000</td>
<td>4</td>
</tr>
<tr>
<td>...0100</td>
<td>0</td>
</tr>
</tbody>
</table>

lw  $t0, 4($s3)  #what? is loaded into $t0

sw  $t0, 8($s3)  #$t0 is stored where?

...0001

in memory location 16
Assuming variable \( b \) is stored in \( \$s2 \) and that the base address of array \( A \) is in \( \$s3 \), what is the three statement MIPS assembly code for the C statement

\[
\]

```
lw $t0, 8($s3)
sub $t0, $t0, $s2
sw $t0, 32($s3)
```
Compiling with a Variable Array Index

- Assuming that the base address of array A is in register $s4, and variables b, c, and i are in $s1, $s2, and $s3, respectively, complete the MIPS assembly code for the C statement

\[ c = A[i] - b \]

```
add  $t1, $s3, $s3  #array index i is in $s3
add  $t1, $t1, $t1  #temp reg $t1 holds 4*i
add  $t1, $t1, $s4  #addr of A[i] now in $t1
lw   $t0, 0($t1)    #addr of A[i] now in $t1
sub  $s2, $t0, $s1  #c = A[i] - b
```
Dealing with Constants

- Small constants are used quite frequently (50% of operands in many common programs)
  
  e.g.,
  
  \[
  A = A + 5; \\
  B = B + 1; \\
  C = C - 18;
  \]

- Solutions? Why not?
  - Create hard-wired registers (like `$zero`) for constants like 1, 2, 4, 10, …
  - Put “typical constants” in memory and load them
  - …

- How do we make this work? How do we Make the common case fast!
Constant (or Immediate) Operands

- Include constants inside arithmetic instructions
  - Much faster than if they have to be loaded from memory (they come in from memory *with* the instruction itself)

- MIPS immediate instructions

```
addi $s3, $s3, 4  #$s3 = $s3 + 4
```

- And to move (copy) the contents of one register to another in one instruction

```
add  $s3, $s2, $zero
```

*There is no subi instruction, can you guess why not?*
# MIPS Instructions, so far

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>add</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td></td>
<td>add immediate</td>
<td>addi $s1, $s2, 4</td>
<td>$s1 = $s2 + 4</td>
</tr>
<tr>
<td>Data transfer</td>
<td>load word</td>
<td>lw $s1, 32($s2)</td>
<td>$s1 = Memory($s2+32)</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>sw $s1, 32($s2)</td>
<td>Memory($s2+32) = $s1</td>
</tr>
</tbody>
</table>
MIPS Organization, so far

- Arithmetic instructions – to/from the register file
- Load/store instructions – to/from memory

### Processor

- **Register File**
  - 32 registers ($\text{zero} - \text{ra}$)
  - src1 addr
  - src2 addr
  - dst addr
  - write data

- **ALU**

- **Memory**

  - Word address (binary)
  - Byte address (bigEndian)

- **Word Address**
  - $0 \ldots 0000$
  - $0 \ldots 0100$
  - $0 \ldots 1000$
  - $0 \ldots 1100$

- **Register File Address**
  - 5 bits

- **Memory Address**
  - 30 bits

- **Data**
  - 32 bits

- **Arithmetic instructions**
  - to/from the register file

- **Load/store instructions**
  - to/from memory
### Review: Unsigned Binary Representation

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00000000</td>
<td>0…0000</td>
<td>0</td>
</tr>
<tr>
<td>0x00000001</td>
<td>0…0001</td>
<td>1</td>
</tr>
<tr>
<td>0x00000002</td>
<td>0…0010</td>
<td>2</td>
</tr>
<tr>
<td>0x00000003</td>
<td>0…0011</td>
<td>3</td>
</tr>
<tr>
<td>0x00000004</td>
<td>0…0100</td>
<td>4</td>
</tr>
<tr>
<td>0x00000005</td>
<td>0…0101</td>
<td>5</td>
</tr>
<tr>
<td>0x00000006</td>
<td>0…0110</td>
<td>6</td>
</tr>
<tr>
<td>0x00000007</td>
<td>0…0111</td>
<td>7</td>
</tr>
<tr>
<td>0x00000008</td>
<td>0…1000</td>
<td>8</td>
</tr>
<tr>
<td>0x00000009</td>
<td>0…1001</td>
<td>9</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>0xFFFFFFFFC</td>
<td>1…1100</td>
<td>$2^{32} - 4$</td>
</tr>
<tr>
<td>0xFFFFFFFFD</td>
<td>1…1101</td>
<td>$2^{32} - 3$</td>
</tr>
<tr>
<td>0xFFFFFFFFE</td>
<td>1…1110</td>
<td>$2^{32} - 2$</td>
</tr>
<tr>
<td>0xFFFFFFFFF</td>
<td>1…1111</td>
<td>$2^{32} - 1$</td>
</tr>
</tbody>
</table>

The bit weight represents the value of each bit position in the binary representation. The bit position indicates the position of each bit starting from the least significant bit (LSB) at position 0.

$2^{31} 2^{30} 2^{29} \ldots 2^{3} 2^{2} 2^{1} 2^{0}$ bit weight

$31 \ 30 \ 29 \ \ldots \ 3 \ 2 \ 1 \ 0$ bit position

1111 \ldots 1111 bit

$1 0 0 0 0 \ldots 0 0 0 0 0 - 1$

$2^{32} - 1$
Instructions, like registers and words of data, are also 32 bits long

- Example:

```
add $t0, $s1, $s2
```

Registers have numbers:

- $t0 = 8$
- $s1 = 17$
- $s2 = 18$

**Instruction Format:**

```
<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>10001</td>
<td>10010</td>
<td>01000</td>
<td>00000</td>
<td>100000</td>
</tr>
<tr>
<td>0x00</td>
<td>17</td>
<td>18</td>
<td>8</td>
<td>0</td>
<td>0x20</td>
</tr>
</tbody>
</table>
```

Can you guess what the field names stand for?
## MIPS Instruction Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>op</strong></td>
<td>OpCode indicating operation to be performed</td>
</tr>
<tr>
<td><strong>rs</strong></td>
<td>Register file address of the first source operand</td>
</tr>
<tr>
<td><strong>rt</strong></td>
<td>Register file address of the second source operand</td>
</tr>
<tr>
<td><strong>rd</strong></td>
<td>Register file address of the result's destination</td>
</tr>
<tr>
<td><strong>shamt</strong></td>
<td>Shift amount (for shift instructions)</td>
</tr>
<tr>
<td><strong>funct</strong></td>
<td>Function code that selects the specific variant of the operation specified in the opcode field</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>op</strong></td>
<td>6</td>
</tr>
<tr>
<td><strong>rs</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>rt</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>rd</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>shamt</strong></td>
<td>5</td>
</tr>
<tr>
<td><strong>funct</strong></td>
<td>6</td>
</tr>
</tbody>
</table>

= 32 bits
Consider the load-word and store-word instr’s

- What would the regularity principle have us do?
  - But . . . Good design demands compromise

Introduce a new type of instruction format

- I-type for data transfer instructions (previous format was R-type for register)

Example: \( \text{l}_{\text{w}} \; \$t0, \; 24 \; (\$s2) \)

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x23</td>
<td>18</td>
<td>8</td>
<td>24 (_{10})</td>
</tr>
<tr>
<td>100011</td>
<td>10010</td>
<td>01000</td>
<td>00000000000011000</td>
</tr>
</tbody>
</table>

Where's the compromise?
Memory Address Location

Example: \texttt{lw \$t0, 24(\$s2)}

\[24_{10} + \$s2 = \$t0\]

\[\ldots 0001\ 1000\]
\[+\ldots 1001\ 0100\]
\[\ldots 1010\ 1100 = 0x120040ac\]

Note that the offset can be positive or negative.
**Review: Signed Binary Representation**

<table>
<thead>
<tr>
<th>2’sc binary</th>
<th>decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>-8</td>
</tr>
<tr>
<td>1001</td>
<td>-7</td>
</tr>
<tr>
<td>1010</td>
<td>-6</td>
</tr>
<tr>
<td>1011</td>
<td>-5</td>
</tr>
<tr>
<td>1100</td>
<td>-4</td>
</tr>
<tr>
<td>1101</td>
<td>-3</td>
</tr>
<tr>
<td>1110</td>
<td>-2</td>
</tr>
<tr>
<td>1111</td>
<td>-1</td>
</tr>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
</tr>
</tbody>
</table>

2³ = -8
-2³ = -8
-(2³ - 1) = -8

- 1 = -1
- 2 = -2
- 3 = -3
- 4 = -4
- 5 = -5
- 6 = -6
- 7 = -7

0101
1011
Complement all the bits and add a 1

0110
1010
Complement all the bits

2³ - 1 = 7
Machine Language - Store Instruction

- Example: \texttt{sw $t0, 24($s2)}

\[
\begin{array}{c|c|c|c}
\text{op} & \text{rs} & \text{rt} & \text{16 bit number} \\
\hline
0x2b & 18 & 8 & \texttt{24_{10}} \\
\hline
101011 & 10010 & 01000 & \texttt{0000000000011000} \\
\end{array}
\]

- A 16-bit offset means access is limited to memory locations within a range of \(+2^{13}-1\) to \(-2^{13}\) (~8,192) \textit{words} (\(+2^{15}-1\) to \(-2^{15}\) (~32,768) \textit{bytes}) of the address in the base register \texttt{$s2$}.
  - 2’s complement (1 sign bit + 15 magnitude bits)
What instruction format is used for the `addi`?  
`addi $s3, $s3, 4`  
#$s3 = $s3 + 4$

Machine format:

- Machine Language – Immediate Instructions

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>16 bit immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>19</td>
<td>19</td>
<td>4</td>
</tr>
</tbody>
</table>

- The constant is kept inside the instruction itself!
  - So must use the I format – Immediate format
  - Limits immediate values to the range $+2^{15}-1$ to $-2^{15}$
Instruction Format Encoding, so far

- Can reduce the complexity with multiple formats by keeping them as similar as possible
  - First three fields are the same in R-type and I-type
- Each format has a distinct set of values in the op field

<table>
<thead>
<tr>
<th>Instr</th>
<th>Frmt</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>32&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>NA</td>
</tr>
<tr>
<td>sub</td>
<td>R</td>
<td>0</td>
<td>reg</td>
<td>reg</td>
<td>reg</td>
<td>0</td>
<td>34&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>NA</td>
</tr>
<tr>
<td>addi</td>
<td>I</td>
<td>8&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>reg</td>
<td>reg</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>constant</td>
</tr>
<tr>
<td>lw</td>
<td>I</td>
<td>35&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>reg</td>
<td>reg</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>address</td>
</tr>
<tr>
<td>sw</td>
<td>I</td>
<td>43&lt;sub&gt;ten&lt;/sub&gt;</td>
<td>reg</td>
<td>reg</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>address</td>
</tr>
</tbody>
</table>
Assembling Code

- Remember the assembler code we compiled last lecture for the C statement


- \texttt{lw} $t0, 8($s3) \quad \# \text{load } A[2] \text{ into } t0$

- \texttt{sub} $t0, t0, s2 \quad \# \text{subtract } b \text{ from } A[2]$

- \texttt{sw} $t0, 32($s3) \quad \# \text{store result in } A[8]$

- Assemble the MIPS object code for these three instructions (in decimal is fine)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Address 1</th>
<th>Address 2</th>
<th>Immediate 1</th>
<th>Immediate 2</th>
<th>Immediate 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>35</td>
<td>19</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>8</td>
<td>18</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>43</td>
<td>19</td>
<td>8</td>
<td></td>
<td>32</td>
</tr>
</tbody>
</table>
## Review: MIPS Instructions, so far

<table>
<thead>
<tr>
<th>Category</th>
<th>Instr</th>
<th>Op Code</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic (R format)</td>
<td>add</td>
<td>0 &amp; 20\text{\textsubscript{hex}}</td>
<td>add $s1, $s2, $s3</td>
<td>$s1 = $s2 + $s3</td>
</tr>
<tr>
<td></td>
<td>subtract</td>
<td>0 &amp; 22\text{\textsubscript{hex}}</td>
<td>sub $s1, $s2, $s3</td>
<td>$s1 = $s2 - $s3</td>
</tr>
<tr>
<td>Arithmetic (I format)</td>
<td>add immediate</td>
<td>8\text{\textsubscript{hex}}</td>
<td>addi $s1, $s2, 4</td>
<td>$s1 = $s2 + 4</td>
</tr>
<tr>
<td>Data transfer (I format)</td>
<td>load word</td>
<td>23\text{\textsubscript{hex}}</td>
<td>lw $s1, 100($s2)</td>
<td>$s1 = \text{Memory}($s2+100)</td>
</tr>
<tr>
<td></td>
<td>store word</td>
<td>2b\text{\textsubscript{hex}}</td>
<td>sw $s1, 100($s2)</td>
<td>\text{Memory}($s2+100) = $s1</td>
</tr>
</tbody>
</table>
Review: Addressing Modes, so far

1. Register addressing

\[
\begin{array}{cccccc}
\text{op} & \text{rs} & \text{rt} & \text{rd} & \text{shamt} & \text{funct} \\
\end{array}
\]

2. Base (displacement) addressing

\[
\begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{offset} \\
\end{array}
\]

3. Immediate addressing

\[
\begin{array}{cccc}
\text{op} & \text{rs} & \text{rt} & \text{operand} \\
\end{array}
\]
MIPS32 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special

3 Instruction Formats: all 32 bits wide

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>sa</th>
<th>funct</th>
<th>R format</th>
</tr>
</thead>
<tbody>
<tr>
<td>op</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td></td>
<td></td>
<td>I format</td>
</tr>
<tr>
<td>op</td>
<td></td>
<td>rt</td>
<td></td>
<td>immediate</td>
<td>J format</td>
<td></td>
</tr>
<tr>
<td>op</td>
<td></td>
<td></td>
<td></td>
<td>jump target</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Registers
- R0 - R31
- PC
- HI
- LO