Topic 2: Introduction to MIPS

- Introduction
- Registers vs. Memory
- Basic Instructions
- Load/Store Instructions
- Branches
- Pseudoinstructions
- Syscalls and Strings
Basic Computer Architecture

Basic pieces of a computer

This isn't 100% true, but it's close!
Basic Computer Architecture

- All operations are driven by the CPU

- CPU is actually very limited:
  - Read/write memory
  - Do simple math (add, sub, logic)
  - Do simple comparisons (eq, less than)
  - Choose to go left or right

- Only one operation at a time (per CPU)
What is MIPS?

- The MIPS-32 architecture defines:
  - Available instructions and what they mean
  - How registers are used

- MIPS designed for performance, simplicity
- Used in real hardware
What is MIPS?

- MIPS is RISC (Reduced Instruction Set Computing)
  - Simpler, easier to understand
  - Faster hardware
  - More steps to do basic operations

- x86 (Intel/AMD) is similar, but CISC (Complex)
  - Fewer instructions
  - More complex, slower hardware
Memory

- Memory is a 1D array of bytes
- Memory addresses are indices into the array
- Bytes used for addresses, even when accessing words
- MIPS limited to $2^{32}$ bytes

What is $2^{10}$?  
What is $2^{20}$?  
What is $2^{30}$?  
What is $2^{32}$?
Memory

Byte addressing, but we usually access words

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff_fff8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xffff_ffff0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Registers vs. Memory

- Registers
- Cache (later in this class)
- Main Memory

Faster

Cheaper
Registers vs. Memory

- In MIPS, there are exactly 32 registers (no more!)
- Each is exactly 32 bits wide
# Registers

<table>
<thead>
<tr>
<th>Name (assembly)</th>
<th>Number (machine language)</th>
<th>Purpose</th>
<th>Must Preserve</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>Always zero; no write</td>
<td>n/a</td>
</tr>
<tr>
<td>$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>Return values</td>
<td>no</td>
</tr>
<tr>
<td>$a0-$a3</td>
<td>4-7</td>
<td>Parameters</td>
<td>no*</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 address</td>
<td>yes</td>
</tr>
</tbody>
</table>

A few others:
- PC
- Floating-point
- Hi/Lo

- $k0-$k1: Reserved for OS
- $gp, $sp, $fp, $ra: Yes
Registers You Can Use

- \$sX general purpose
  (later: will be “saved” registers)
- \$tX general purpose
  (later: will be “temporary” registers)
- \$zero constant
- \$a0/\$v0 syscall parameters
  (later: arguments, return values)
Basic Instructions

\[
\text{add } $s0,$t1,$t2 \quad \# \ s0 = t1 + t2
\]

- Basic arithmetic instructions read from two registers, write to a third
  - Directly accessing memory impossible
    - (without load/store)

\[
\text{add } $s0,$s0,$s0 \quad \# \ s0 *= 2
\]

- OK to use the same register 2 or 3 times
Available Operations

- **Basic 3-operand instructions:**
  - `add` 32-bit only
  - `sub` 32-bit only
  - `and` bitwise AND
  - `or` bitwise inclusive OR
  - `nor` bitwise OR, negated
  - `xor` bitwise exclusive OR

---

How would you perform 8-bit or 16-bit addition?

Why aren't there any logical AND/OR operations?

Where is the NOT operator?
Building Complex Operations

- Build complex operations using a sequence of simpler steps
  
  \[
  \text{add}\ \$t0,\$s1,\$s2\quad \# \ t0=s1+s2 \\
  \text{add}\ \$s0,\$t0,\$s3\quad \# \ s0=s1+s2+s3
  \]

- Use registers to hold temporary values
  - Be careful not to overwrite important values!
Group Exercise:

Assume that the following variables are in registers:

- alpha   $s0
- bravo   $s1
- charlie $s2

Write a sequence of instructions which will calculate the following values. You may use any $tX$ registers, but do not modify any $sX$ registers (except the one you're required to modify):

- $s3 = alpha + bravo + charlie$
- $s4 = -bravo - charlie$
- $s5 = charlie - (alpha + bravo)$
- $s6 = bravo*2 + charlie*3$
Building Complex Operations

alpha \quad $s0
bravo \quad $s1
charlie \quad $s2

\$s3 = \alpha + \bravo + \charlie \\
\$s4 = - \bravo - \charlie \\
\$s5 = \charlie - (\alpha + \bravo) \\
\$s6 = \bravo \times 2 + \charlie \times 3

add \quad $t0,$s0,$s1 \quad \# \ t0 = \alpha+\bravo \\
add \quad $s3,$t0,$s2 \quad \# \ s3 = \alpha+\bravo+\charlie \\
sub \quad $t0,$zero,$s1 \quad \# \ t0 = 0-\bravo \\
sub \quad $s4,$t0,$s2 \quad \# \ s4 = -\bravo-\charlie 
alpha  $s0
bravo  $s1
charlie $s2

$s3 = alpha + bravo + charlie
$s4 = - bravo - charlie
$s5 = charlie - (alpha + bravo)
$s6 = bravo*2 + charlie*3

add  $t0,$s0,$s1  # $t0 = alpha+bravo
sub  $s5,$s2,$t0  # $s5 = charlie-(alpha+bravo)

add  $t0,$s1,$s1  # $t0 = 2*bravo
add  $t1,$s2,$s2  # $t1 = 2*charlie
add  $t1,$t1,$s2  # $t1 = 3*charlie
add  $s6,$t0,$t1  # $s6 = 2*bravo+3*charlie
Building Complex Operations

- When writing a complex expression, it's OK to use the destination register as a temporary:

  add $s0,$s1,$s2  # $s0 = s1+s2
  add $s0,$s0,$s3  # $s0 = s1+s2+s3

- **WARNING:** Never overwrite any register before it is used the last time. How would you write this?

  $s0 *= 3;
Immediate Operations

• Most (not all) 3-register instructions have an “immediate” alternative
  – Replace 2\textsuperscript{nd} input with a 16-bit constant
  – Hint: $\text{zero}+\text{constant}$ is often useful

\begin{align*}
\text{addi} & \quad \text{t0, t0, 1} \\
\text{andi} & \quad \text{s0, s3, 0x3} \\
\text{ori} & \quad \text{t7, s0, 0xff} \\
\text{addi} & \quad \text{s5, zero, 11}
\end{align*}

Group Exercise:

Convert these operations to pseudocode. Use the register names like variables.
Immediate Operations

- Most (not all) 3-register instructions have an “immediate” alternative
  - Replace 2\textsuperscript{nd} input with a 16-bit constant
  - \textbf{Hint:} $\text{zero+constant}$ is often useful

\begin{verbatim}
addi $t0,$t0,1    # t0++
andi $s0,$s3,0x3  # s0 = s3 & 0x3
ori  $t7,$s0,0xff # t7 = s0 | 0xff
addi $s5,$zero,11 # s5 = 11
\end{verbatim}
Immediate Operations

- **No subi**
  - addi's immediate is **signed** (sign extend to 32 bits)

```
addi $t0,$t0,-1    # t0--
```
Load / Store

- No instructions can access memory directly except for load / store
  - Need to load a variable before you use it
  - Need to store a variable after you change it
    - OK to keep values in registers when possible

- All load/store operations use base+offset
  - A register gives the base address
  - Offset is 16-bit immediate (signed), often zero.
Load / Store

\texttt{lw \$s0, 0($s1)}

- Load Word (32 bits)
- Read memory at address $0+$s1
- Write to $s0
Load / Store

lw  $s0, 0($s1)

- Load / Store operations must be aligned
  - Word operations on a multiple of 4
  - Half-word operations on a multiple of 2
  - Byte operations anywhere

- Otherwise, MIPS will throw exception (crash)
Loads

lw $s0, 0($s1)
lh $s0, 0($s1)
lb $s0, 0($s1)

• Load Word, Half-Word, or Byte
  – Sign-extend all loaded values to 32 bits (overwrites entire register)

• Read bytes **starting** at the given address.
  – Example Word at 0x001c:
    bytes 0x001c, 0x001d, 0x001e, 0x001f
Stores

\[\text{sw} \quad \$s0, \ 0(\$s1)\]
\[\text{sh} \quad \$s0, \ 0(\$s1)\]
\[\text{sb} \quad \$s0, \ 0(\$s1)\]

- Store Word, Half-Word, or Byte
  - **Crop** value down (lose high bits from register)

- **WARNING:** Reg on left is value to **write**
  - Violates the “destination on left” rule
Naming Your Variables

• You **cannot** name your registers (sorry)
  – They change what they represent too often

• You **can (and must!)** name your global variables (memory)
  – Important because their locations might change if you change testcases, or change your code
Naming Your Variables

my_var: .word 123

- Use a label to define names
  - Also use labels for branches (coming up next!)
- Use a storage directive to allocate memory

- “Declare a WORD named my_var and set its initial value to 123.”
Naming Your Variables

\begin{verbatim}
my_var:
  .word 123
\end{verbatim}

• Separate lines are OK, this means the same thing as before.
• Labels give a name to the next thing in the code – whatever it is.
Naming Your Variables

• Available directives:
  
  .word   <val>
  .half   <val>
  .byte   <val>

This is only a partial list. But it's good enough for this class.
Naming Your Variables

- Labels known only to the assembler
  - Machine code uses raw addresses, **never** names

- Use the `la` instruction to turn a name into an address:
  ```
  la   $s0, my_var
  ```

- `la` is basically like setting a pointer:
  ```
  WORD  *s0 = &my_var;
  ```
Two Steps

- Loads and stores are **two steps**:
  - Use `la` to get the address of the variable
  - Use load / store to access the actual memory

```assembly
la  $s0, my_var  # s0 = &my_var
lw  $s1, 0($s0)  # s1 =  my_var
```

- OK to use the destination register as (temporary) address register:

```assembly
la  $t3, my_var  # t3 = &my_var
lw  $t3, 0($t3)  # t3 =  my_var
```

Mixing Code and Data?

- You must separate code from data with assembler directives.

```assembly
.data
count:    .half  0

.text
main:
    # your code here!
```
Branches

• How to implement `if/else` in assembly?

```c
if (x == y)
    doThis();
else
    doThat();
nextThing();
```
Branches

• We have to “flatten” the flowchart:

```
\[
x == y \text{ ?}
\]

\[
\text{doThis}
\]

\[
\text{doThat}
\]

\[
\text{nextThing}
\]
```
if \( x \neq y \)  
goto ELSE;

doThis();
goto DONE;

ELSE:
doThat();

DONE:
nextThing();
Branches

\textbf{bne} \ $s0,\$s1,ELSE

jal doThis
j DONE

ELSE:
jal doThat

DONE:
jal nextThing

\textbf{x == y} ?

T

doThis

F

doThat

nextThing
Conditional Branches

\[
\begin{align*}
\text{beq} & \quad \$s0, \$t0, \text{LABEL} \\
\text{bne} & \quad \$s1, \$zero, \text{OTHER\_PLACE}
\end{align*}
\]

- Conditional branches jump to the label \textbf{if the condition is true}

- MIPS only \textbf{compares} registers
  - Comparing to $\text{\$zero}$ is often useful
Unconditional Branches

\[ j \rightarrow \text{LABEL} \]

- Jumps to the label, no matter what
  - Often useful at the end of an `if/else` (to skip the else block)
  - Also common in loops
Putting It All Together

Group Exercise:

Convert the following C code to MIPS:
- Load necessary variables (all are words)
- Implement comparisons
- Store values as necessary

```c
if (x == y)
    x = 1;
else
    y = 2;
z = 3;
```
Pseudoinstructions

• A **pseudoinstruction** is something that the assembler accepts, which is not really an instruction
  
  \[
  \begin{align*}
  \text{addi} & \quad $s0, $zero, 0xdeadbeef \\
  \text{lw} & \quad $t3, \text{MY\_VAR}
  \end{align*}
  \]

• Pseudoinstructions are **banned**!
  
  – Only use instructions that we have shown you
  
  – I plan to have an automatic scanner as part of the Project 2 grading script
The One Exception

• A single exception:

\[
\text{la } \$s0, \text{ MY\_VAR } \用途{0xdeadbeef}
\]

• The assembler turns this into:

\[
\text{lui } \$at, \$zero, 0xdead
\text{addi } \$s0, \$at, 0xbeef
\]
What is `syscall`?

`syscall` is a MIPS instruction

- Takes no parameters
  - Parameters are passed in registers
  - Set values before the `syscall`
- Calls the “operating system”
  - Implemented as other MIPS instructions, real world
  - Simulated by the simulator
syscall in Action

addi    $v0, $zero, 1    # print_int
addi    $a0, $zero, 123 # print_int(123)
syscall                  # do it!

• Three steps:
  - Set $v0 (selects the operation)
  - Set $a0 (parameter)
  - syscall
## syscall Types to Know

<table>
<thead>
<tr>
<th>$v0</th>
<th>Use</th>
<th>Type of $a0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>print int</td>
<td>int</td>
</tr>
<tr>
<td>4</td>
<td>print string</td>
<td>char*</td>
</tr>
<tr>
<td>11</td>
<td>print char</td>
<td>char</td>
</tr>
<tr>
<td>10</td>
<td>exit()</td>
<td>n/a</td>
</tr>
</tbody>
</table>
Strings

• How to declare a string?

.data
MSG: .asciiz "This is a string.\n"
FOO: .asciiz "Hello world!"
BAR: .asciiz "asdf"
Strings

• `.asciiz` creates **null terminated** strings
  - Works basically like C strings
  - Extra `'\0'` character at the end
  - **Necessary** for syscall 4 !!!

• Rare, but useful: `.ascii` (no z)
  - No null terminator
  - For long strings, across multiple lines
  - Use `.asciiz` for the last one!
Printing a String

.data
MSG: "Hello world!\n"

.text
addi $v0, $zero,4    # print_str
la   $a0, MSG
syscall
There is no `printf()` in assembly!

Must break it down into smaller pieces
- Syscall 1 (int)
- Syscall 4 (string)

Group Exercise:

Convert the following `printf()` call to `syscall`s.
Assume that the variable `val` is already loaded into `$s4`.

```c
printf("val: %d\n", val);
```