Review

About how much of the AMD Opteron chip is L2 cache?
What is computer architecture the study of?
The MIPS (a) is a load/store architecture, (b) is a register-register machine, (c) has an ISA, (d) all of the above
What does ISA stand for, and what is it?
In a machine instruction the registers are called (a) operands, (b) noops, (c) opcodes
How many bits are needed to name a MIPS register?
What is the C equivalent to: \texttt{sub $t0, $t1, $t2} \ ?

... and what does RAM stand for?

Memory

Memory sizes are specified much like register files; here is a $2^k \times n$ RAM

<table>
<thead>
<tr>
<th>CS</th>
<th>WR</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>None</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Read selected address</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Write selected address</td>
</tr>
</tbody>
</table>

A chip select input CS enables or ‘disables’ the RAM
ADRS specifies the memory location to access
WR selects between reading from or writing to the memory

- To read from memory, WR should be set to 0. OUT will be the k-bit value stored at ADRS
- To write to memory, we set WR = 1. DATA is the k-bit value to store in memory
MIPS Memory

MIPS memory is byte-addressable, which means that each memory address references an 8-bit quantity. The MIPS architecture can support up to 32 address lines.

- This results in a $2^{32} \times 8$ RAM, which would be 4 GB of memory.
- Not all MIPS machines will actually have that much!

Loading and Storing Bytes

The MIPS instruction set includes dedicated load and store instructions for accessing memory. These differ from other instructions because they use indexed addressing — a base + offset.

- The address operand specifies a register (base) and a signed constant (offset).
- These values are added to generate the effective address.

The MIPS load byte instruction `lb` transfers one byte of data from main memory to a register.

```
lb $t0, 20($a0)  # $t0 = Memory[$a0 + 20]
```

The store byte instruction `sb` transfers the lowest byte of data from a register into main memory.

```
sb $t0, 20($a0)  # Memory[$a0 + 20] = $t0
```
Loading and Storing Words

You can also load or store 32-bit quantities -- a complete word instead of just a byte -- with the `lw` and `sw` instructions.

\[
\begin{align*}
  &\text{lw} \; \$t0, \; 20(\$a0) & \# & \$t0 = \text{Memory}[\$a0 + 20] \\
  &\text{sw} \; \$t0, \; 20(\$a0) & \# & \text{Memory}[\$a0 + 20] = \$t0
\end{align*}
\]

Most programming languages support several 32-bit data types.

- Integers
- Single-precision floating-point numbers
- Memory addresses, or pointers

Unless otherwise stated, we'll assume words are the basic unit of data.

An Array of Words From Memory of Bytes

Use care with memory addresses when accessing words.

For instance, assume an array of words begins at address 2000.

- The first array element is at address 2000
- The second word is at address 2004, not 2001

Example, if \( \$a0 \) contains 2000, then

\[
\begin{align*}
  \text{lw} \; \$t0, \; 0(\$a0) \\
  \text{lw} \; \$t0, \; 8(\$a0)
\end{align*}
\]

accesses the first word of the array, but

\[
\begin{align*}
  \text{lw} \; \$t0, \; 0(\$a0) \\
  \text{lw} \; \$t0, \; 8(\$a0)
\end{align*}
\]

would access the third word of the array, at address 2008.

Memory is byte addressed but usually word referenced.
Memory Alignment

Picture words of data stored in byte-addressable memory as follows

<table>
<thead>
<tr>
<th>Address</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-bit data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Word 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The MIPS architecture requires words to be aligned in memory; 32-bit words must start at an address that is divisible by 4.

- 0, 4, 8 and 12 are valid word addresses
- 1, 2, 3, 5, 6, 7, 9, 10 and 11 are not valid word addresses
- Unaligned memory accesses result in a bus error, which you may have unfortunately seen before

This restriction has relatively little effect on high-level languages and compilers, but it makes things easier and faster for the processor.

Computing on Data in Memory

So, to compute with memory-based data, you must:
1. Load the data from memory to the register file
2. Do the computation, leaving the result in a register
3. Store that value back to memory if needed

Let’s say you want to do some addition on values in memory:

```c
char A[4] = {1, 2, 3, 4};
int result;
```

How can you do the following using MIPS assembly language?

```assembly
```

A two part task:
- Define the data layout
- Define the computation
In MIPS Assembler …

#=======================================
#       Static data allocation and initialization
#=======================================
.data
A: .byte 1, 2, 3, 4  # Create space for A, and give
                    # values in decimal: 1, 2, 3, 4
result: .word 9    # allocate 32 bits,
                    # initialize to 9 for no good reason

In MIPS Assembler

#=======================================
#       Program Text
#=======================================
.text
main:
...
lb  $t0, 0($a0)     #Set up so A's addr is in reg $a0, load A[0]
lb  $t1, 1($a0)     #Get second element A[1]
add $t0, $t1, $t0  #Add in second element
lb  $t1, 2($a0)     #Get third element A[2]
add $t0, $t1, $t0  #Add in third element
lb  $t1, 3($a0)     #Get fourth element A[3]
add $t0, $t1, $t0  #Add in fourth element
sw  $t0, 0($a1)     #Set up so result's addr is in reg $a1, save
Pseudo Instructions

MIPS assemblers support pseudo-instructions giving the illusion of a more expressive instruction set by translating into 1 or more simpler, “real” instructions.

For example, li and move are pseudo-instructions:

- `li   $a0, 2000`  # Load immediate 2000 into $a0
- `move $a1, $t0`  # Copy $t0 into $a1

They are probably clearer than their corresponding MIPS instructions:

- `addi $a0, $0, 2000`  # Initialize $a0 to 2000
- `add $a1, $t0, $0`  # Copy $t0 into $a1

We’ll see more pseudo-instructions this semester.

- A complete list of instructions is given in Appendix A
- Unless otherwise stated, you can always use pseudo-instructions in your assignments and on exams

Control Flow

- Instructions usually execute one after another, but it’s often necessary to alter the normal control flow

  ```c
  // Find the absolute value of a0
  v0 = a0;
  if (v0 < 0)
    v0 = -v0;  // This might not be executed
  v1 = v0 + v0;
  ```

- Conditional statements execute only if some test is true

  ```c
  // Sum the elements of a five-element array a0
  v0 = 0;
  t0 = 0;
  while (t0 < 5) {
    v0 = v0 + a0[t0];  // These statements will
    t0++;             // be executed five times
  }
  ```

- Loops cause some statements to execute many times
MIPS Control Instructions

• MIPS's control-flow instructions
  
  j             # for unconditional jumps  
  bne and beq   # for conditional branches  
  slt and slti  # set if less than (w/o and w/ immediate)

• As in
  
  j   line_label
  bne $4, $7, line_label   #skip to next part
  slt $4, $7, $8           #test $7 less than $8

• For example, compute |$8| ... first test, then branch
  
  slt          $9, $8, $0   #set $9 to 1 if $8 < 0
  beq          $9, notNeg   #branch if $9 not set
  sub          $8, $0, $8   #flip sign

  notNeg: