EECS 322 Computer Architecture
Language of the Machine
Load, Store and Dense Arrays

Instructor: Francis G. Wolff
wolff@eecs.cwru.edu
Case Western Reserve University
This presentation uses powerpoint animation: please viewshow
Review: Design Abstractions

An abstraction omits unneeded detail, helps us cope with complexity

High Level Language Program (e.g., C)
Compiler
Assembly Language Program (e.g. MIPS)
Assembler
Machine Language Program (MIPS)
Machine Interpretation
Control Signal Specification

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

ALUOP[0:3] <= InstReg[9:11] & MASK
Review: Registers

- Unlike C++, assembly instructions cannot directly use variables. **Why not?** Keep Hardware Simple

- Instruction operands are **registers**: limited number of special locations; 32 registers in MIPS ($r0 - r31)

  **Why 32?** Performance issues: Smaller is faster

- Each MIPS register is 32 bits wide
  Groups of 32 bits called a **word** in MIPS

- A **word** is the natural size of the host machine.
Register Organization

- Viewed as a tiny single-dimension array (32 words), with an register address.

- A register address ($r0-$r31) is an index into the array

<table>
<thead>
<tr>
<th>$r0</th>
<th>0</th>
<th>32 bits of data</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r1</td>
<td>1</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>$r2</td>
<td>2</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>$r3</td>
<td>3</td>
<td>32 bits of data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>. . .</td>
</tr>
<tr>
<td>$r28</td>
<td>28</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>$r29</td>
<td>29</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>$r30</td>
<td>30</td>
<td>32 bits of data</td>
</tr>
<tr>
<td>$r31</td>
<td>31</td>
<td>32 bits of data</td>
</tr>
</tbody>
</table>
ANSI C integers (section A4.2 Basic Types)

- Examples: `short x; int y; long z; unsigned int f;`

- Plain `int` objects have the **natural size** suggested by the **host machine architecture**;

- the other sizes are provided to meet special needs

- Longer integers provide **at least as much as** shorter ones,

- but the implementation may make plain integers **equivalent** to either `short` integers, or `long` integers.

- The `int` types all represent **signed** values unless specified otherwise.
Review: Compilation using Registers

• Compile by hand using registers:

```c
int f, g, h, i, j;
f = (g + h) - (i + j);
```

• Assign MIPS registers:

```c
# $s0=int f, $s1=int g, $s2=int h,
# $s3=int i, $s4=int j
```

• MIPS Instructions:

```c
add $s0,$s1,$s2  # $s0 = g+h
add $t1,$s3,$s4  # $t1 = i+j
sub $s0,$s0,$t1  # f=(g+h)-(i+j)
```

Note: whereas C declares its operands, Assembly operands (registers) are fixed and not declared.
Objects declared *register* are automatic, and *(if possible)* stored in fast registers of the machine.

Previous example:

```c
register int f, g, h, i, j;
f = (g + h) - (i + j);
```

The *register* keyword tells the compiler your intent.

This allows the programmer to guide the compiler for better results. *(i.e. faster graphics algorithm)*

This is one reason that the C language is successful because it caters to the hardware architecture!
Assembly Operands: Memory

- C variables map onto registers
- What about data structures like arrays?
- But MIPS arithmetic instructions only operate on registers?

- **Data transfer instructions** transfer data between registers and memory

Think of memory as a large single dimensioned array, starting at 0
Memory Organization: bytes

- Viewed as a large, single-dimension array, with an address.

- A memory address is an index into the array

- "Byte addressing" means that the index points to a byte of memory.

- C Language:
  - bytes multiple of word
  - Not guaranteed though
    - char f;
    - unsigned char g;
    - signed char h;
Memory Organization: words

- **Bytes** are nice, but most data items use larger "words"

- For MIPS, a **word** is 32 bits or 4 **bytes**.

  \[ \begin{array}{c|c}
  \hline
  0 & 32 \text{ bits of data} \\
  4 & 32 \text{ bits of data} \\
  8 & 32 \text{ bits of data} \\
  12 & 32 \text{ bits of data} \\
  \hline
  \end{array} \]

  Note: Registers hold 32 bits of data = word size (not by accident)

- \(2^{32}\) **bytes** with **byte addresses** from 0 to \(2^{32}-1\)

- \(2^{30}\) **words** with **byte addresses** 0, 4, 8, ... \(2^{32}-4\)
Memory Organization: alignment

- MIPS requires that all words start at addresses that are multiples of 4

- Called alignment: objects must fall on address that is multiple of their size.

- (Later we’ll see how alignment helps performance)
Memory Organization: Endian

- Words are aligned (i.e. 0, 4, 8, 12, 16, … not 1, 5, 9, 13, …)
  i.e., what are the least 2 significant bits of a word address? Selects the which byte within the word

- How?

  **Little Endian byte 0**

  big endian byte 0

- Little Endian address of least significant byte:
  Intel 80x86, DEC Alpha

- Big Endian address of most significant byte:
  HP PA, IBM/Motorola PowerPC, SGI, Sparc
Data Transfer Instruction: Load Memory to Reg (lw)

- **Load**: moves a word from memory to register

- MIPS syntax, lw for load word:
  - operation name
  - register to be loaded
  - constant and register to access memory

**example**: \( \text{lw } \$t0, 8(\$s3) \)

Called "offset"  Called "base register"

- MIPS lw semantics: \( \text{reg}[$t0] = \text{Memory}[8 + \text{reg}[$s3]] \)
**lw example**

**Suppose:**

Array A address = 3000  
reg[$s3$]=Array A  
reg[$t0$]=12;  
mem[3008]=42;  

Then  

\[ \text{lw } t0, 8(s3) \]

Adds offset “8”  
to $s3$ to select A[8],  
to put “42” into $t0$

\[ \text{reg}[$t0$]=\text{mem}[8+\text{reg}[$s3$]] \]

\[ =_0\text{mem}[8+3000]=\text{mem}[3008] \]

\[ =42 \]  
  =Hitchhiker's Guide to the Galaxy
Data Transfer Instruction: Store Reg to Memory (sw)

- **Store Word (sw):** moves a word from register to memory

- **MIPS syntax:** \( sw \) \$rt, offset($rindex) \\
  **MIPS semantics:** \( \text{mem}[\text{offset} + \text{reg}[\$rindex]] = \text{reg}[\$rt] \)

- **MIPS syntax:** \( lw \) \$rt, offset($rindex) \\
  **MIPS semantics:** \( \text{reg}[\$rt] = \text{mem}[\text{offset} + \text{reg}[\$rindex]] \)

- **MIPS syntax:** add \$rd, \$rs, \$rt \\
  **MIPS semantics:** \( \text{reg}[\$rd] = \text{reg}[\$rs] + \text{reg}[\$rt] \)

- **MIPS syntax:** sub \$rd, \$rs, \$rt \\
  **MIPS semantics:** \( \text{reg}[\$rd] = \text{reg}[\$rs] - \text{reg}[\$rt] \)
Compile Array Example

C code fragment:

```c
register int g, h, i;
int A[66]; /* 66 total elements: A[0..65] */
g = h + A[i]; /* note: i=5 means 6rd element */
```

Compiled MIPS assembly instructions:

```assembly
add $t1,$s4,$s4               # $t1 = 2*i
add $t1,$t1,$t1               # $t1 = 4*i
add $t1,$t1,$s3              #$t1=addr A[i]
lw $t0,0($t1)               # $t0 = A[i]
add $s1,$s2,$t0               # g = h + A[i]
```
Execution Array Example: \( g = h + A[i] \);

<table>
<thead>
<tr>
<th>C variables</th>
<th>Instruction</th>
<th>$s1$</th>
<th>$s2$</th>
<th>$s3$</th>
<th>$s4$</th>
<th>$t0$</th>
<th>$t1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>$s1$</td>
<td>?</td>
<td>4</td>
<td>3000</td>
<td>5</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>h</td>
<td>$s2$</td>
<td>3000</td>
<td>5</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>$s3$</td>
<td>3000</td>
<td>5</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
<tr>
<td>i</td>
<td>$s4$</td>
<td>3000</td>
<td>5</td>
<td>?</td>
<td>?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**suppose** \((\text{mem}[3020]=38)\)

- `lw $t0,0($t1)`
- `add $t1,$t1,$s3`
- `add $t1,$t1,$t1`
- `add $t1,$s4,$s4`

\(\text{suppose} \) \((\text{mem}[3020]=38)\)

- `add $t1,$s4,$s4`
- `add $t1,$t1,$t1`
- `add $t1,$t1,$s3`
- `lw $t0,0($t1)`
- `add $s1,$s2,$t0`

\(\text{suppose} \) \((\text{mem}[3020]=38)\)
Immediate Constants

C expressions can have constants:

\[ i = i + 10; \]

MIPS assembly code:

\[
\begin{align*}
# & \text{Constants kept in memory with program} \\
\text{lw} & \quad \$t0, 0(\$s0) \quad # \text{load 10 from memory} \\
\text{add} & \quad \$s3, \$s3, \$t0 \quad # \text{i = i + 10}
\end{align*}
\]

MIPS using constants: (addi: add immediate)

So common operations, have instruction to add constants (called “immediate instructions”)

\[
\text{addi} \quad \$s3, \$s3, 10 \quad # \text{i = i + 10}
\]
Constants: Why?

Why include immediate instructions?

Design principle: Make the common case fast

Why faster?

a) Don’t need to access memory
b) 2 instructions v. 1 instruction
Zero Constant

Also, perhaps most popular constant is zero. MIPS designers reserved 1 of the 32 register to always have the value 0; called $r0, $0, or “$zero”

Useful in making additional operations from existing instructions;

copy registers: $s2 = $s1;
   add $s2, $s1, $zero  # $s2 = $s1 + 0

2’s complement: $s2 = –$s1;
   sub $s2, $zero, $s1  # $s2 = – $s1

Load a constant: $s2 = number;
   addi $s2, $zero, 42  # $s2 = 42
C Constants

C code fragment

```c
int i;
const int limit = 10;

i = i + limit;
```

Is the same as

```c
i = i + limit; /* but more readable */
```

And the compiler will protect you from doing this

```c
limit=5;
```
Class Homework: Due next class

C code fragment:

```
register int g, h, i, k;
int    A[5], B[5];
B[k] = h + A[i+1];
```

1. Translate the C code fragment into MIPS

2. Execute the C code fragment using:

```
A=address 1000, B=address 5000, i=3, h=10, k=2,
/* i.e. A[0]=24; A[1]=33; ... */.
```