Review

• MIPS Assembly Language and the SPIM Environment
  - Write your code in MIPS assembly lang in the SPIM editor window
    (it is one level removed from the tedious machine language)
  - ASSEMBLE: translate to machine language using the SPIM assembler
  - RUN the program under SPIM simulator
  - DEBUG using SPIM debugging window

Anatomy: 5 components of any Computer

Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.

Data Transfer: Memory to Reg (1/4)

• To transfer a word of data, we need to specify two things:
  • Register: specify this by # ($0 - $31) or symbolic name ($s0, ..., $t0, ...)
  • Memory address: more difficult
    - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
    - Other times, we want to be able to offset from this pointer.

• Remember: “Load FROM memory”

Data Transfer: Memory to Reg (2/4)

• To specify a memory address to copy from, specify two things:
  • A register containing a pointer to memory
  • A numerical offset (in bytes)

• The desired memory address is the sum of these two values.

• Example: \( 8 \ ($t0) \)
  • specifies the memory address pointed to by the value in $t0, plus 8 bytes
Data Transfer: Memory to Reg (3/4)

- **Load Instruction Syntax:**
  - \( a \ b, c(d) \)
  - where
    - a) operation name
    - b) register that will receive value
    - c) numerical offset in bytes
    - d) register containing pointer to memory

- **MIPS Instruction Name:**
  - `lw` (meaning Load Word, so 32 bits or one word are loaded at a time)

Example:
\[ \text{lw} \ W1, 12(W0) \]
This instruction will take the pointer in \( W0 \), add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register \( W1 \)

- **Notes:**
  - \$s0 is called the **base register**
  - 12 is called the **offset**
  - offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure

Data Transfer: Memory to Reg (4/4)

- **Example:**
  - \[ \text{lw} \ W0, 12(W0) \]

Data Transfer: Reg to Memory

- Also want to store from register into memory
  - Store instruction syntax is identical to Load's
- **MIPS Instruction Name:**
  - `sw` (meaning Store Word, so 32 bits or one word are loaded at a time)

Example:
\[ \text{sw} \ W0, 12(W0) \]
This instruction will take the pointer in \( W0 \), add 12 bytes to it, and then store the value from register \( W0 \) into that memory address

- **Remember:** “Store INTO memory”

Pointers v. Values

- **Key Concept:** A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory address), and so on

  - If you write \( \text{add} \ $t2,$t1,$t0 \) then \( W0 \) and \( W1 \) better contain values
  - If you write \( \text{lw} \ $t2,0(W0) \) then \( W0 \) better contain a pointer
  - Don’t mix these up!

Addressing: Byte vs. word

- Every word in memory has an **address**, similar to an index in an array
- Early computers numbered words like C numbers elements of an array:
  - \([0],[2],[4],...\) (Called TRS-80 “address” of a word)
- Computers needed to access 8-bit **bytes** as well as words (4 bytes/word)
- Today machines address memory as bytes, i.e., “Byte Addressed” hence 32-bit (4 byte) word addresses differ by 4
  - \([0],[4],[8],...\)

Compilation with Memory

  - \( 4 \times 5 = 20 \) to select \( A[5] \): byte v. word
- Compile by hand using registers:
  - \( g = h + A[5] \)
  - \( g: s1, h: s2, s3: \) base address of A
- 1st transfer from memory to register:
  - \[ \text{lw} \ W0, 20(W3) \] # \( W0 \) gets \( A[5] \)
  - Add \( 20 \) to \( s3 \) to select \( A[5] \), put into \( W0 \)
- Next add it to \( h \) and place in \( g \)
  - \( \text{add} \ s1, s2, W0 \) # \( s1 = h + A[5] \)
Notes about Memory

• Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  - Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
  - So remember that for both `lw` and `sw`, the sum of the base address and the offset must be a multiple of 4 (to be word aligned)

More Notes about Memory: Alignment

• MIPS requires that all words start at byte addresses that are multiples of 4 bytes

<table>
<thead>
<tr>
<th>Last hex digit of address is:</th>
<th>0, 4, 8, or ( C_{\text{hex}} )</th>
<th>1, 5, 9, or ( D_{\text{hex}} )</th>
<th>2, 6, A, or ( E_{\text{hex}} )</th>
<th>3, 7, B, or ( F_{\text{hex}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aligned</td>
<td>![Aligned]</td>
<td>![Aligned]</td>
<td>![Aligned]</td>
<td>![Aligned]</td>
</tr>
<tr>
<td>Not Aligned</td>
<td>![Not Aligned]</td>
<td>![Not Aligned]</td>
<td>![Not Aligned]</td>
<td>![Not Aligned]</td>
</tr>
</tbody>
</table>

• Called Alignment: objects must fall on address that is multiple of their size.

Role of Registers vs. Memory

• What if more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Less common in memory: spilling

• Why not keep all variables in memory?
  - Smaller is faster: registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation

C Decisions: if Statements

• 2 kinds of `if` statements in C
  - `if (condition) clause`
  - `if (condition) clause1 else clause2`

• Rearrange 2nd `if` into following:
  ```
  if (condition) goto L1;
  goto L2;
  L1: clause1;
  L2: 
  ```

• Not as elegant as `if-else`, but same meaning

MIPS Decision Instructions

• Decision instruction in MIPS:
  ```
  beq register1, register2, L1
  ```

• `beq` is “Branch if (registers are) equal”
  Same meaning as (using C):
  ```
  if (register1==register2) goto L1
  ```

• Complementary MIPS decision instruction
  ```
  bne register1, register2, L1
  ```

• `bne` is “Branch if (registers are) not equal”
  Same meaning as (using C):
  ```
  if (register1!=register2) goto L1
  ```

• Called conditional branches

MIPS Goto Instruction

• In addition to conditional branches, MIPS has an unconditional branch:
  ```
  j label
  ```

• Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition

• Same meaning as (using C):
  ```
  goto label
  ```

• Technically, it’s the same as:
  ```
  beq $0,$0,label
  ```
  since it always satisfies the condition.
**Compiling C if into MIPS (1/2)**

- Compile by hand
  
  \[
  \begin{align*}
  \text{if (i == j)} & \quad f = g + h; \\
  \text{else} & \quad f = g - h;
  \end{align*}
  \]

- Use this mapping:
  
  \[
  \begin{align*}
  f &: s0 \\
  g &: s1 \\
  h &: s2 \\
  i &: s3 \\
  j &: s4
  \end{align*}
  \]

**Compiling C if into MIPS (2/2)**

- Compile by hand
  
  \[
  \begin{align*}
  \text{if (i == j)} & \quad f = g + h; \\
  \text{else} & \quad f = g - h;
  \end{align*}
  \]

- Final compiled MIPS code:
  
  ```
  beq $s3, $s4, True  # branch i==j
  sub $s0, $s1, $s2  # f=g-h(false)
  j Fin  # goto Fin
  True: add $s0, $s1, $s2  # f=g+h (true)
  Fin:
  ```

  Note: Compiler automatically creates labels to handle decisions (branches). Generally not found in HLL code.

**“And in Conclusion...”**

- Memory is byte-addressable, but \texttt{lw} and \texttt{sw} access one word at a time.
- A pointer (used by \texttt{lw} and \texttt{sw}) is just a memory address, so we can add to it or subtract from it (using offset).
- A Decision allows us to decide what to execute at run-time rather than compile-time.
- C Decisions are made using conditional statements within \texttt{if}, \texttt{while}, \texttt{do while}, \texttt{for}.
- MIPS Decision making instructions are the conditional branches: \texttt{beq} and \texttt{bne}.
- New Instructions:
  
  \[
  \texttt{lw, sw, beq, bne, j}
  \]