Linked Lists in MIPS

- Let’s see how *singly* linked lists are implemented in MIPS
  - on MP2, we have a special type of *doubly* linked list

Each node consists of 8 bytes:
- the *first* 4 bytes contain the int data
- the *next* 4 bytes contain a pointer to the next node in the list
- the last node’s next pointer is **NULL** (zero)

Suppose $a0$ points to the first node in the list ($a0 = 1000$)

<table>
<thead>
<tr>
<th>C/C++</th>
<th>MIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>t0 = a0-&gt;data</td>
<td>lw $t0, 0($a0)</td>
</tr>
<tr>
<td>t1 = a0-&gt;next</td>
<td>lw $t1, 4($a0)</td>
</tr>
</tbody>
</table>
Setting the next pointers

- Suppose we want the links to point as follows:

```
addi   $t1, $a0, 16
addi   $t2, $a0, 8
sw     $t1, 4($a0)
sw     $t2, 4($t1)
sw     $0,  4($t2)
```

- Again, suppose $a0 points to the first node in the list ($a0 = 1000)
Printing the elements of a linked list

- Translate the following code into MIPS

```c
// head points to the first node in the list
while(head != NULL) {
    print(head->data);
    head = head->next;
}

loop:  # assume $t0 = head   (WARNING: errors!!)
    beq  $t0, $0, done
    lw   $a0, 0($t0)
    jal  print
    lw   $t0, 4($t0)
    j    loop

done:
```
Recursive Functions

- Recall that recursive functions have one or more base-cases, and one or more recursive calls. Example:

```c
int length(node *head) {
    if(head == NULL) return 0;
    return 1 + length(head->next);
}
```

- Useful tip: Translate the base case first (rarely needs the stack)

```assembly
length:    # $a0 = head
    bne  $a0, $0, rec_case
    li   $v0, 0       # return-value = NULL = 0
    jr   $ra
rec_case:  ...
```
The Recursive Case

- Let’s examine the recursive step more carefully:

  ```c
  return 1 + length(head->next);
  ```

- **Useful tip**: Apart from $ra$, what else must be preserved across the function call?
  - Here: nothing

```
rec_case:
  addi   $sp, $sp, -4  # save space for $ra
  sw     $ra, 0($sp)
  lw     $a0, 4($a0)
  jal    length       # recursive call
  addi   $v0, $v0, 1
  lw     $ra, 0($sp)
  addi   $sp, $sp, 4
  jr      $ra
```
Instruction Set Architecture (ISA)

- The ISA is an **abstraction layer** between hardware and software
  - Software doesn’t need to know how the processor is implemented
  - Processors that implement the same ISA appear equivalent

![Diagram showing ISA as an abstraction layer between Software and two processors](image)

- An ISA enables processor innovation without changing software
  - This is how Intel has made billions of dollars

- Before ISAs, software was re-written/re-compiled for each new machine
ISA history: RISC vs. CISC

- 1964: IBM System/360, the first computer family
  - IBM wanted to sell a range of machines that ran the same software

- 1960’s, 1970’s: Complex Instruction Set Computer (CISC) era
  - Much assembly programming, compiler technology immature
  - Hard to optimize, guarantee correctness, teach

- 1980’s: Reduced Instruction Set Computer (RISC) era
  - Most programming in high-level languages, mature compilers
  - Simpler, cleaner ISA’s facilitated pipelining, high clock frequencies

- 1990’s: Post-RISC era
  - ISA compatibility outweighs any RISC advantage in general purpose
  - CISC and RISC chips use same techniques (pipelining, superscalar, ..)
  - Embedded processors prefer RISC for lower power, cost

- 2000’s: Multi-core era
Comparing x86 and MIPS

- x86 is a typical CISC ISA, MIPS is a typical RISC ISA

- Much more is similar than different:
  - Both use registers and have byte-addressable memories
  - Same basic types of instructions (arithmetic, branches, memory)

- A few of the differences:
  - Fewer registers: 8 (vs. 32 for MIPS)
  - 2-register instruction formats (vs. 3-register format for MIPS)
  - Additional, complex addressing modes
  - Variable-length instruction encoding (vs. fixed 32-bit length for MIPS)
x86 Registers

- Few, and special purpose
  - 8 integer registers
  - two used only for stack
  - not all instructions can use all registers

- Little room for temporary values
  - x86 uses “two-address code”
  - \( \text{op } x, \ y \quad \# \ y = y \ \text{op } x \)

- Rarely can the compiler fit everything in registers
  - Stack is used much more heavily, so it is *architected* (not just a convention)
  - The \texttt{esp} register is the stack pointer
  - Explicit push and pop instructions
Memory Operands

- Most instructions can include a memory operand:
  ```
  addl  -8(%ebp), %eax  # equivalent MIPS code:
  #    lw   $t0, -8($ebp)
  #    add  $eax, $eax, $t0
  ```

- MIPS supports just one addressing mode: `offset($reg)`

  refers to `Mem[reg + offset]`

- X86 supports complex addressing modes: `offset(%rb, %ri, scale)`

  refers to `Mem[b + i * scale + offset]`
Variable Length Instructions

08048344 <sum>:

8048344:      55                      push %ebp
8048345:      89 e5                   mov %esp,%ebp
8048347:      8b 4d 08                mov 0x8(%ebp),%ecx
804834a:      ba 01 00 00 00          mov $0x1,%edx

- Instructions range in size from 1 to 17 bytes
  - Commonly used instructions are short (think Huffman Codes)
  - In general, x86 has smaller code than MIPS

- Many different instruction formats, plus prefixes, suffixes
  - Harder to decode for the machine
Why did Intel win?

- x86 won because it was the first 16-bit chip by two years
- IBM put it in PCs because there was no competing choice
- Rest is inertia and “financial feedback”
  - x86 is most difficult ISA to implement for high performance, but
  - Because Intel sells the most processors …
  - It has the most money …
  - Which it uses to hire more and better engineers …
  - Which is uses to maintain competitive performance …
  - And given equal performance, compatibility wins …
  - So Intel sells the most processors!