# Question

- Consider the following MIPS `interrupt_handler`

```
.ktext 0x80000080
interrupt_handler:
  addi   $sp, $sp, -8
  sw     $t0, 0($sp)            # save $t0
  sw     $t1, 4($sp)            # save $t1

  ...

  lw     $t0, 0($sp)            # restore $t0
  lw     $t1, 4($sp)            # restore $t1
  addi   $sp, $sp, 8

  ...
```

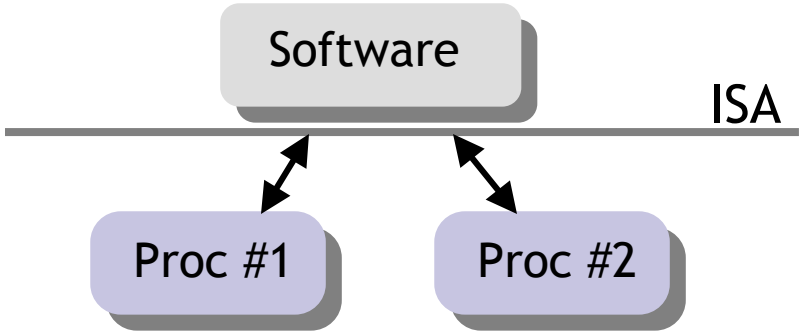
Why don't we use the stack to save and restore registers?

Answer: Because the stack may be *corrupted*, e.g. `$sp` may not be correctly set

# Instruction Set Architecture (ISA)

- The ISA is the interface between hardware and software.

- The ISA serves as an abstraction layer between the HW and SW
  - Software doesn't need to know how the processor is implemented
  - Any processor that implements the ISA appears equivalent




- 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.

# A little ISA history

- 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
  - Simple machine implementations
  - Complex instructions simplified programming, little impact on design
- 1980's: Reduced Instruction Set Computer (RISC) era
  - Most programming in high-level languages, mature compilers
  - Aggressive machine implementations
  - Simpler, cleaner ISA's facilitated pipelining, high clock frequencies
- 1990's: Post-RISC era
  - ISA complexity largely relegated to non-issue
  - CISC and RISC chips use same techniques (pipelining, superscalar, ..)
  - ISA compatibility outweighs any RISC advantage in general purpose
  - Embedded processors prefer RISC for lower power, cost
- 2000's: ??? EPIC? Dynamic Translation? Just more x86?

# RISC vs. CISC

- MIPS was one of the first RISC architectures. It was started about 20 years ago by John Hennessy, one of the authors of our textbook.
- The architecture is similar to that of other RISC architectures, including Sun's SPARC, IBM and Motorola's PowerPC, and ARM-based processors.
- Older processors used complex instruction sets, or CISC architectures.
  - Many powerful instructions were supported, making the assembly language programmer's job much easier.
  - But this meant that the processor was more complex, which made the hardware designer's life harder.
- Many new processors use reduced instruction sets, or RISC architectures.
  - Only relatively simple instructions are available. But with high-level languages and compilers, the impact on programmers is minimal.
  - On the other hand, the hardware is much easier to design, optimize, and teach in classes.
- Even most current CISC processors, such as Intel 8086-based chips, are now implemented using a lot of RISC techniques.

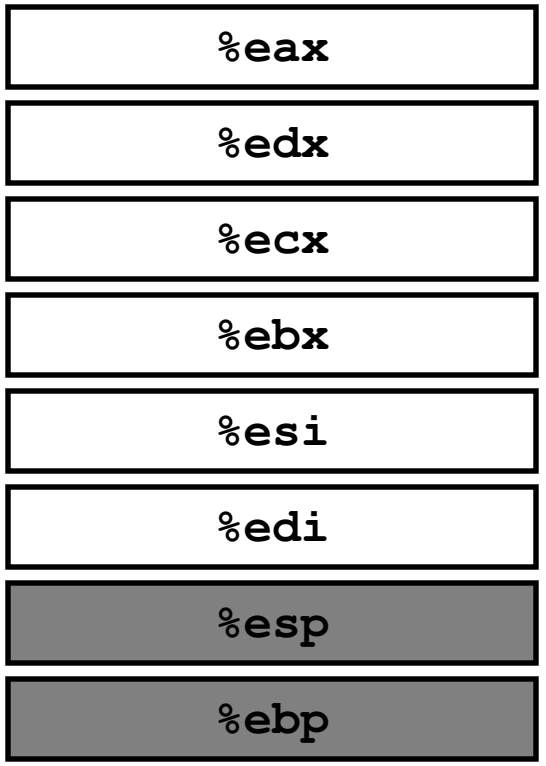
# Comparing x86 and MIPS

- 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)
  - Greater reliance on the stack, which is part of the architecture
  - 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”
  - **op x, y** # y = y 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 `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[%rb + %ri*scale + offset]`

# Address Computation Examples

| | |
|---|---|
| **%edx** | **0xf000** |
| **%ecx** | **0x100** |

| Expression | Computation | Address |
|---|---|---|
| 0x8(%edx) | 0xf000 + 0x8 | 0xf008 |
| (%edx,%ecx) | 0xf000 + 0x100 | 0xf100 |
| (%edx,%ecx,4) | 0xf000 + 4*0x100 | 0xf400 |
| 0x80(,%edx,2) | 2*0xf000 + 0x80 | 0x1e080 |

# 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 compression)
  - In general, x86 has smaller code than MIPS

- Many different instruction formats, plus pre-fixes, post-fixes
  - Harder to decode for the machine

- Typical exam question:

  How does _____ depend on the complexity of the ISA?

# Why did Intel win?

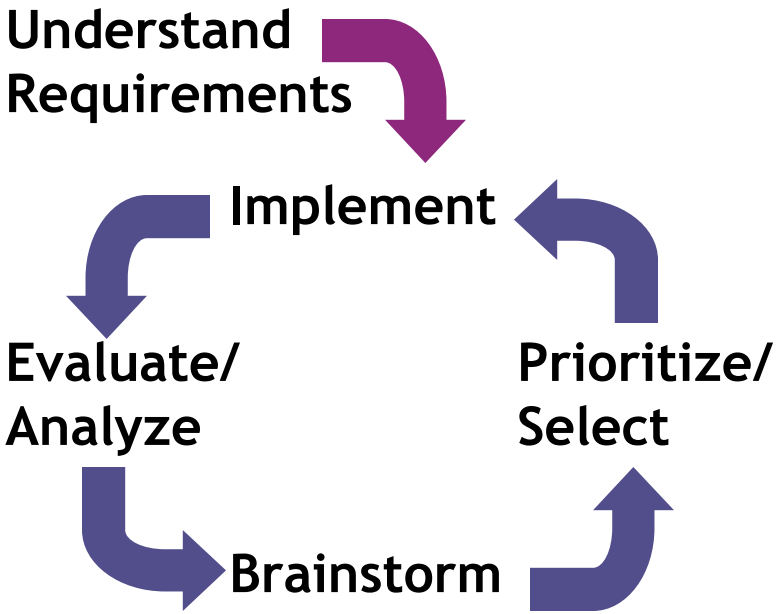
x86 won because it was the first 16-bit chip by two years.

- 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.

# The SPIMbot Design Process

**Key Idea:** Iterative Refinement

1. Build simplest possible implementation
2. Does it meet criteria? If so, stop.
   Else, what can be improved?
3. Generate ideas on how to improve it
4. Select best ideas, based on benefit/cost
5. Modify implementation based on best ideas
6. Goto step 2.




It is very tempting to go straight to an "optimized" solution. Pitfalls:

1. You never get anything working
2. Incomplete problem knowledge leads to selection of wrong optimizations

With iterative refinement, you can stop at any time!
Result is optimal for time invested.

# The basic bot: MP 3

- Playing field divided into 400 “sectors”, tokens placed at random

[Part 2(b)] Issue a scan request to sector (*p*, *q*) and set `scanning`
  - While `scanning` is set, loop
  - When `scanning` gets cleared, scan next sector

[Part 2(c)] When a scan-interrupt occurs, process scan result
  - Processing as in MP 2
  - If tokens found, set a timer-interrupt, otherwise clear `scanning`

[Part 2(d)] When a timer-interrupt occurs, drive to pick up tokens
  - When finished, clear `scanning`