Power Management as I knew it

Jim Kardach

Agenda

- Philosophy of power management
- PM Timeline
- Era of OS Specific PM (OSSPM)
- Era of OS independent PM (OSIPM)
- Era of OS Assisted PM (APM)
- Era of OS & hardware cooperative PM (ACPI)
- Non-PM (taking advantage of $P=CV^2F$)
- Era of Indirect PM
- Era of behavioral PM

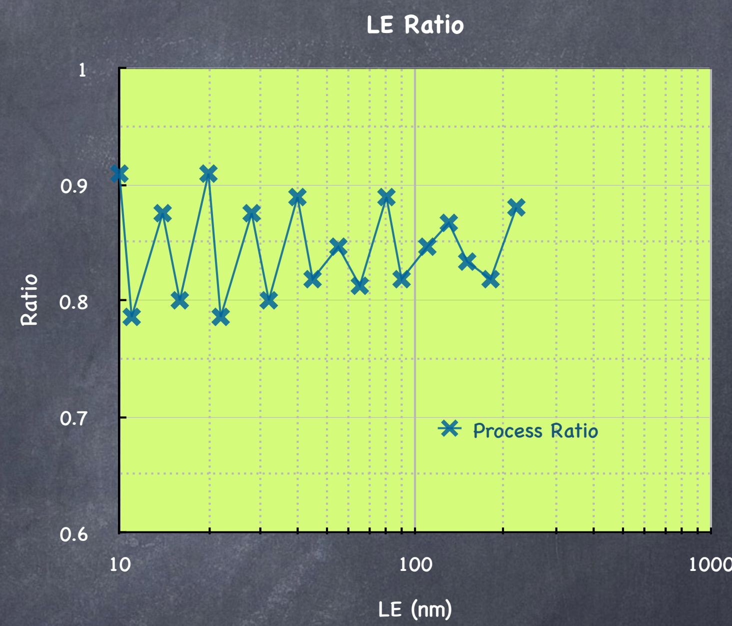
Philosophy of PM

- Design things to work efficiently
- Design things to do nothing efficiently

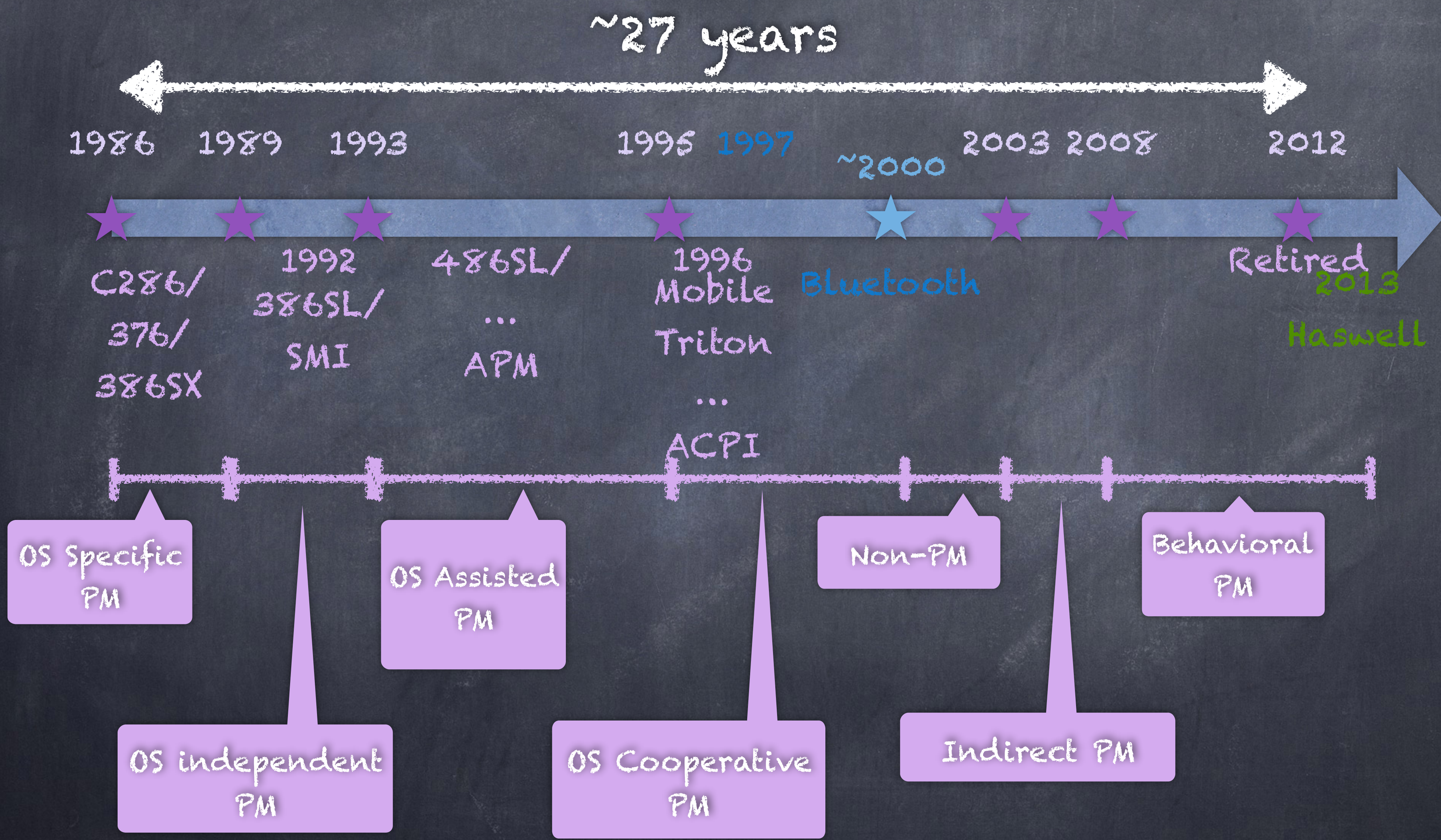
- Intel influences
  - Don't impact performance
  - Don't break anything
  - Products were designed for desktop/server and modified for mobile (until ~2010)

Moore's Law

- In the old days, mobile processors would get a Moore's law kicker
- Initial 386/486/Pentium/... would be a new micro-architecture
    - the mobile version (a year later) would be a modified version on a shrink process
        - Voltage reduces, Frequency increases, capacitance decreases (# number of devices increases, geometry halves)
        - Free power reduction ($P = CV^2F$)
            - $P = 1/2*(0.84*C)(3.3/5*V)^2(1*F) = 1/2*0.55*C*V^2*F$
- We would complement this with architectural changes to reduce platform power
- Over time:
    - Voltage drop would decrease (tough to go below V_t)
    - Capacitance would not drop as much
        - interconnect capacitance goes up
        - number of devices during shrink ("tick") would increase
        - Would start using different size LE to control leakage Vs. speed



Timeline of PM work



OS Specific PM (pre '92)

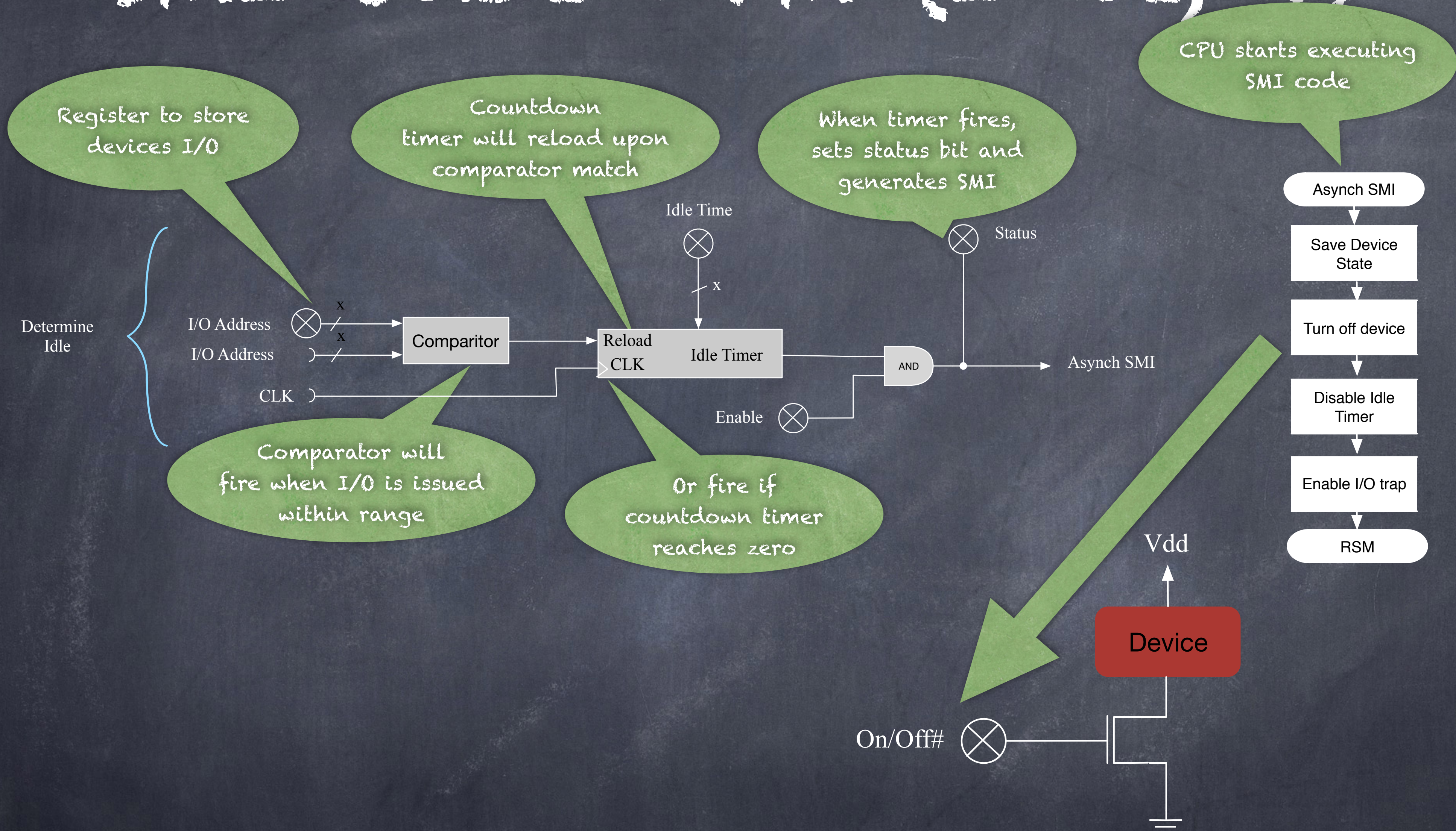
- Put things in a low power mode when idle
- Turn them back on when needed

- Issues
  - Power Management software was dependent on the OS & HW specific drivers
  - Things were very un-reliable

OS independent PM (1) '92-'93 ish

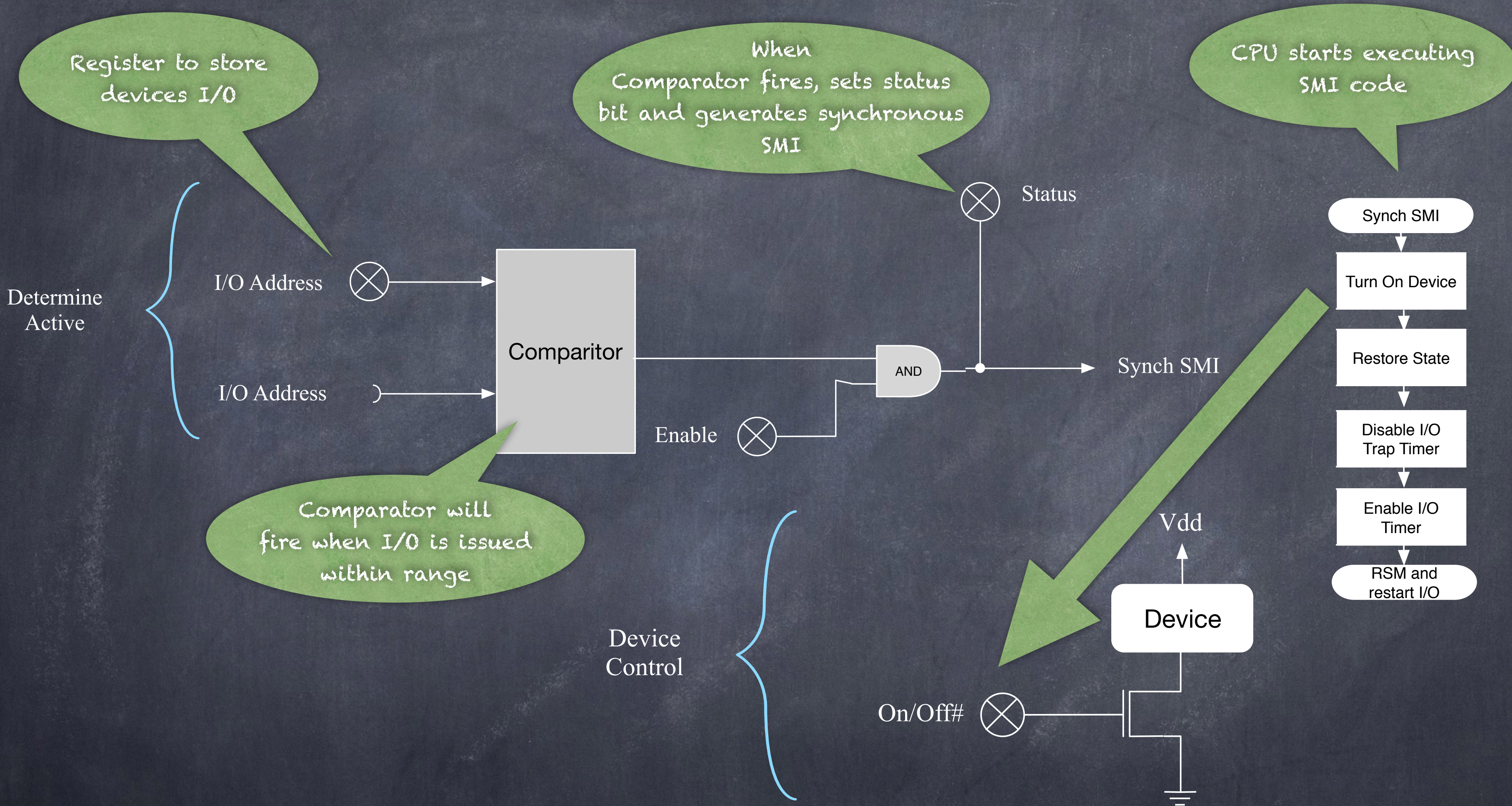
- Goal:
  - "Hardware like" Power management that ships with the notebook and works on any OS
  - Enable Suspend/Resume, long battery life
- A software based architecture was enabled through new platform/CPU feature
  - System Management Mode (SMM)
    - A System Management Interrupt enabled execution of OEM firmware within a new operating mode (regardless of what the system was doing previously)
    - A new RSM instruction that would resume the CPU back to what it was previously doing
    - OEMs could write firmware to respond to "power management events" that would then turn devices on or off
  - The OEM could deliver the feature as part of the notebook firmware, and the code would work regardless of the OS running
  - Enabled turning devices on/off, and suspending/resuming the entire platform

SMI Based PM (Idle, 2)



- Each device would have a set of "shadow registers" with a timer. The notebook would enable an idle time, and when this expired the device would be turned off

SMI Based PM (Trap, 3)



- For activity its the reverse, a match to the I/O address would fire an synch SMI which would turn on the device, restore its context, and then re-start the I/O access after the RSM instruction.

OS independent PM (4) '92-'93 ish

- OS independent Suspend/Resume
  - Used SMI to suspend system
    - STR - kept DRAM powered
    - STD - stored DRAM/context to HDD
  - Used RSM to restore a resumed system

OS independent PM (5) '92-'93 ish

- Pros
  - Power Management just worked, and reliably (versus the previous stuff)
    - regardless of OS (DOS, Windows, Unix, ...)
    - allowed OEMs to ship PM with the box, and to write PM code once
  - Enabled a robust suspend/resume feature
  - SMI scaled beyond power management (bug fixes, new features, ...)
- Cons
  - Policy was based on what the HW knows, which is very low level (I/O, memory accesses and interrupts)
    - The hardware doesn't understand what activity is important or not
  - CPU was poorly power managed
    - Could only divide the clock
  - There were artifacts
    - Suspend/Resume also suspended time

OS independent PM (4) '92–'93 ish

- Why go below the OS?
  - When we started Microsoft was too busy fixing DOS and creating Windows to be bothered with PM
  - For a group focused on portable platforms, having a power management solution was our top priority
  - Decision was to move forward without Microsoft and build something that would work regardless of the OS

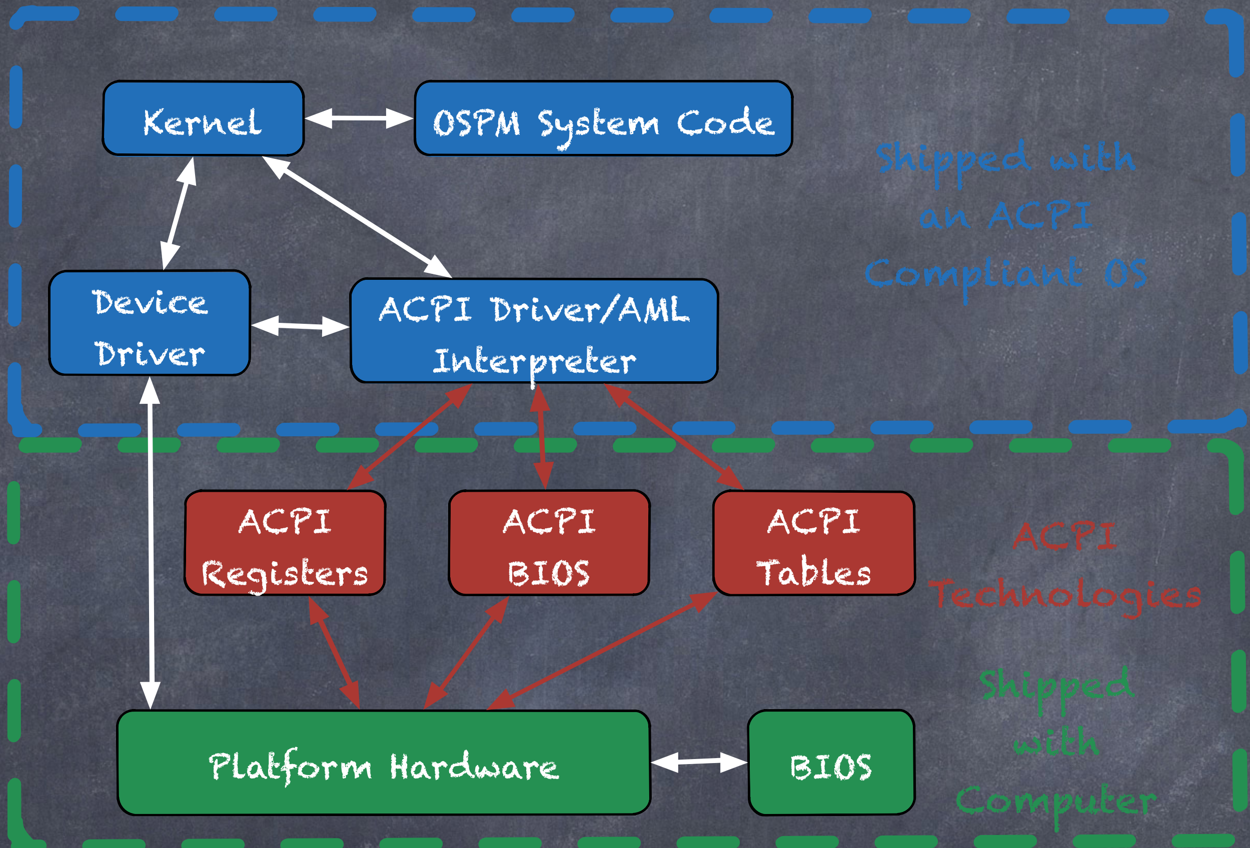
OS Assisted PM
~93 ish

- With the first samples of the 386SL platform, to fix the artifacts we needed an interface to communicate between the OS and hardware.
    - Things like
        - I've just resumed, you might want to
            - check the time (RTC) and update if necessary
        - Indicate the level of activity of the OS
            - If the OS is really idle, the hardware can do very aggressive PM
            - If the OS is really busy, the hardware can turn off PM ...
- Resulted in the creation of Advanced Power Management (APM)
    - Intel, Phoenix (BIOS) and Microsoft worked on an API that allowed communication between the OS and hardware (the SMI layer)
- Solved most of the major artifacts
    - OS notification of power states, transitions, pending transitions (battery about to die, ...)
    - Update time
    - OS policy (wake on events via OS controls)

OS & hardware cooperative PM '95 ish

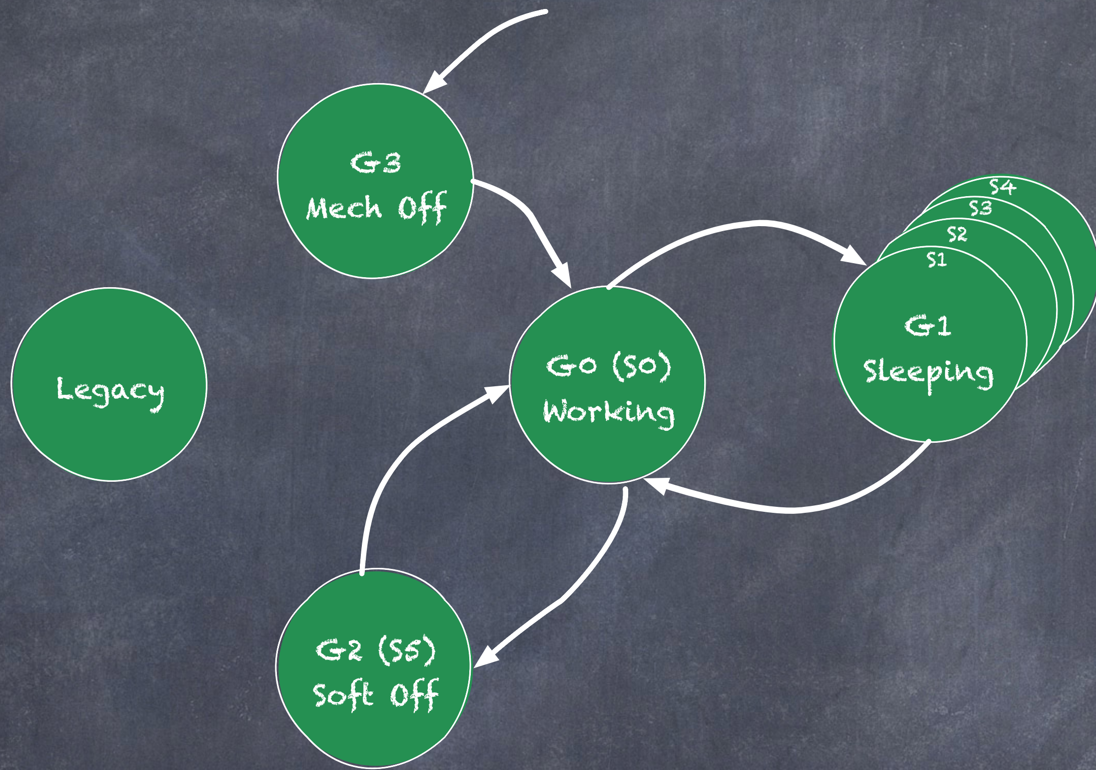
- Goal:
  - Develop an architecture that would work with any OS and was extensible
  - Make platform PM more robust
  - PM the CPU much more aggressively
  - If the OS does not support OS PM, then enable a fallback to the SMI based PM
- The Advanced Configuration and Power Interface (ACPI) specification was created

ACPI Architecture



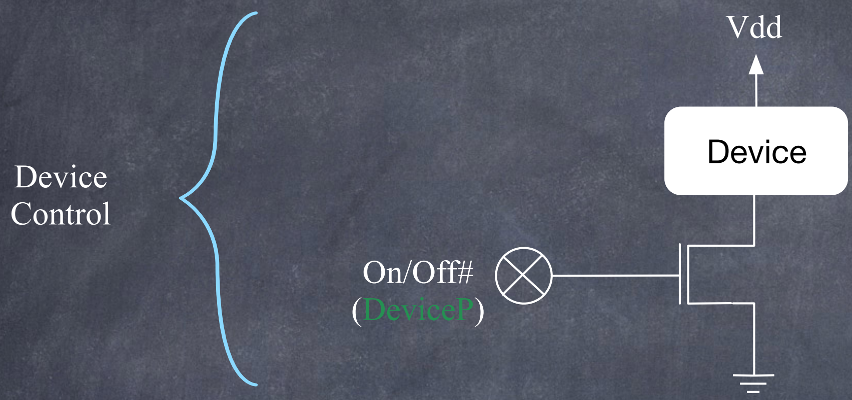
- ACPI is an interface specification (deals with the red arrows and creates the red blocks)
  - ACPI Registers perform defined functions that the OS ACPI driver own
  - ACPI BIOS provides a means for the OS to communicate to the PM hardware
  - ACPI Tables allow the OEM to write PM code in a multi-threaded language in a safe environment (the OS AML interpreter)

ACPI Architecture



- ACPI formalized the terminology and system states

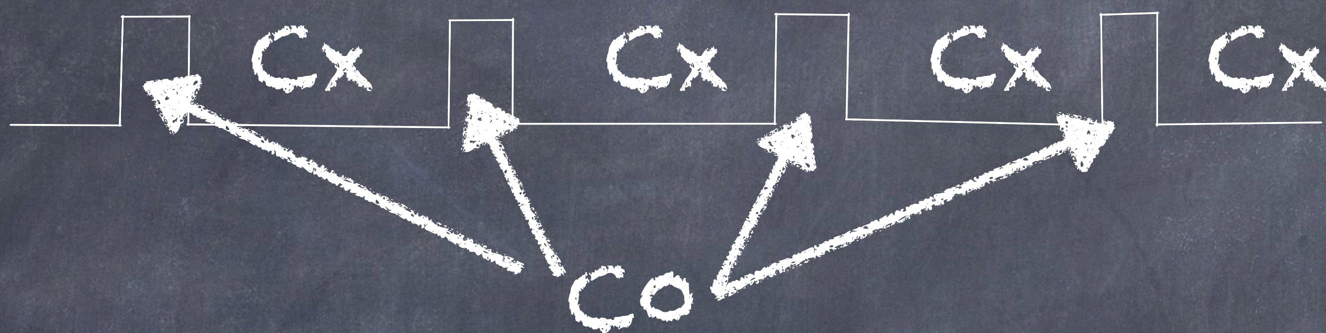
ACPI Architecture




```
PowerResource(Device, 0, 0) {
    Method(_STA) {
        Return (GIO.DeviceP)        // return bit 1
    }
    Method(_ON) {
        Store (One, GIO.DeviceP)  // turn on power
        Sleep (10)                 // wait 10 ms
    }
    Method(_OFF) {
        Store (Zero, GIO.DeviceP) // turn on power
    }
}
```

- All of the unique power management electronics were enumerated in tables
  - Each defined object an have a power resource associated with it.
  - The OS just grabs the device object and if it wants to
    - Turn the device ON, execute the _ON method
    - Turn the device OFF, execute the _OFF method
    - Get status of the device, execute the _STA method
- Note that the existing SMI hardware could be used to control device power
  - Additional support of ACPI required creating the ACPI tables with the appropriate methods

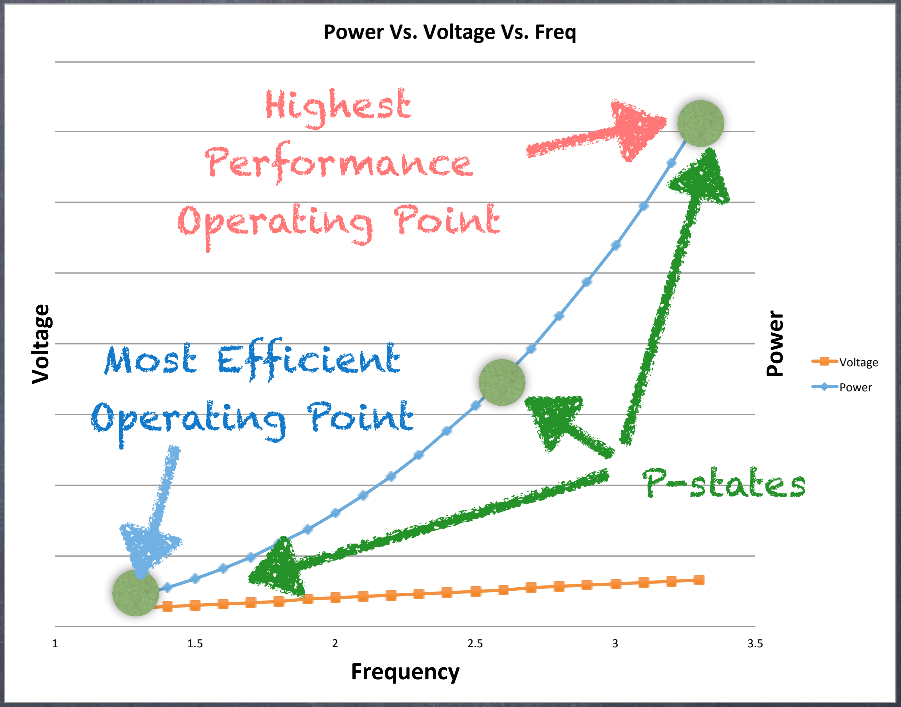
ACPI Architecture



- Preempt Interrupt (Used in preemptive Operating Systems)
  - A regular interrupt that is used by the OS to schedule work for a given CPU/thread
  - Upon interrupt, kernel schedules work
  - When work is done it executes the HLT instruction
- In ACPI, the OS looks to see the time till the next preempt interrupt, and chooses a low power state to go into (C1, C2 or C3).
  - higher number is lower power and longer exit latency
- Prior to ACPI you could slow the CPU to 50%, with C-states a typical CPU at idle will be in a low power state more than 99% of the time.
  - Active CPU/thread might have 50% C0 state, ...

Speed Step, Non-Power Management

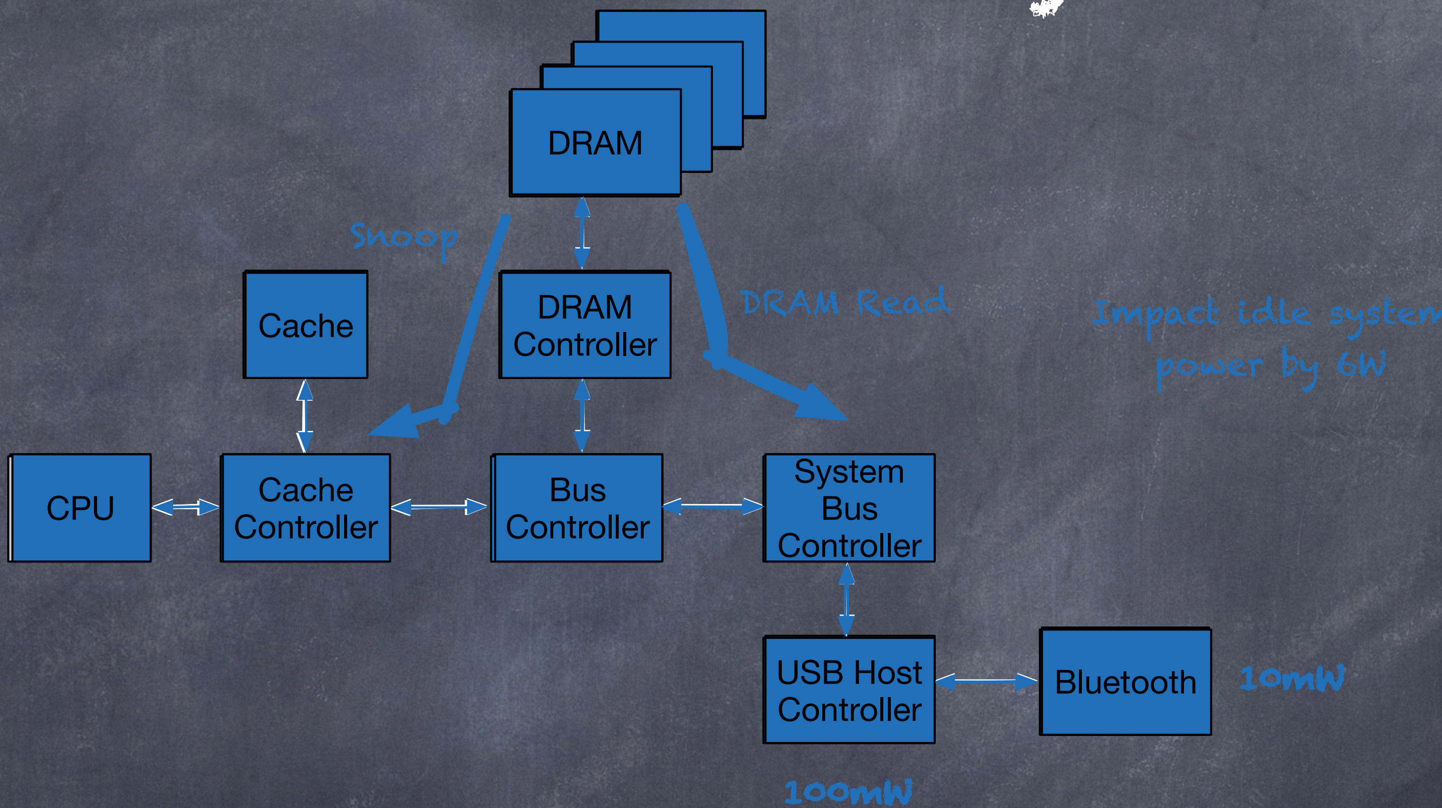
- Power = $C*V^2F$
- Frequency is somewhat linear to voltage
  - As your raise the voltage, the maximum frequency goes up
- Most Efficient operating point is the maximum frequency at minimum voltage
  - Other than non-linear events
- Performance States (P-states) were added to allow the OS to dynamically modify the operating voltage and frequency of the CPU



Indirect PM

- Sometimes its not the power your burning, but the power you are causing others to burn on your behalf
  - Crying babies

Power of an idle Bluetooth radio (ie connected to a keyboard)

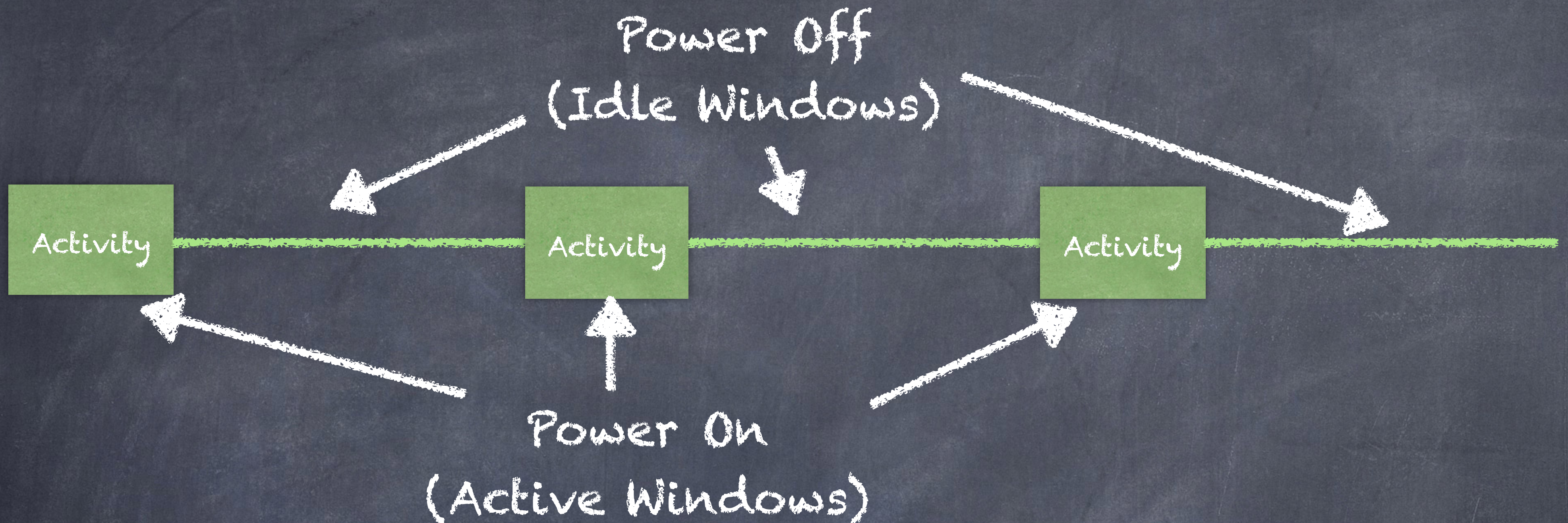


- Bluetooth radios are very low power, but use USB as the host interface
- USB host controller is very low power
  - Spec says at full load it consumes less than 100mW!
- But USB is a polled architecture, the bluetooth radio can't tell it when it has an event, the interface polls it
  - The USB host controller has a task list it must read to poll the Bluetooth radio, to see if it has any work
    - Must access memory

Indirect PM

- USB, PCIe, ...
  - Updating existing standards to have nice idle behavior
    - No activity unless there is real work

Behavioral PM



- Modifying system behavior when idle in order maximize PM opportunity
- Goal was to turn-off the power to the entire system similar to how we power management the CPU's C-states

- At idle, not much activity (random interrupts and DMA)
  - If we could re-arrange this activity so it happens together, then we can shut everything off

Behavioral PM

- Main Issues
  - Interrupts
    - Periodic interrupts (align them)
    - event based interrupts
  - DMA
    - typically caused by a FIFO being full

Behavioral PM

- Solution spaces
  - created new attributes for interrupts allowing non-critical to be deferred by a certain time
  - Any activity indicates to all resources to make activity if needed
    - Kick off pending interrupts
    - kick off pending DMAs
    - This self synchronizes resources

Summary

- Over 27 years, notebooks have improved immensely.
  - IBM Convertible April 3 1986
    - 13 lbs, $1995
    - sub 1 MIP, 4.77MHz 80C88, 256Kbytes RAM, small screen, no HDD
    - 8 hour battery life with 23 Whr battery
      - 2.875 W Average power @ idle
  - My Apple Macbook Air (2014 Haswell)
    - 3 lb, $1,749
    - 7000+ MIPS, Haswell CPU, 8 Gbytes of DRAM, 13" screen, 512 GByte SSD
    - 12 hours battery life with 54 Whr battery
      - 4.5W Average power on battery life test
      - ~3W idle with backlight
      - ~125mW idle backlight off
- The same philosophy applies
  - Design things to work efficiently
  - Design things to do nothing efficiently