Homework

• Reading
  – None (Finish all previous reading assignments)

• Machine Projects
  – Continue with MP5

• Labs
  – Finish lab reports by deadline posted in lab
Pentium Reset / Boot

• Reset
  – Held asserted until power supply voltages stabilize
  – Starts processor in “real mode” for 1 Meg address space
  – Forces %cs = 0xf000 and %eip = 0xffff0
  – First instruction is fetched from address 0xfffff0
    • %cs (offset) 0x f 0 0 0 -
    • %eip 0x - f f f f 0
    • Address: 0x f f f f f 0
  – Address decoding logic will enable ROM BIOS device when processor fetches this address and the control bus lines indicate “instruction fetch”
Pentium Reset / Boot

• Boot
  – BIOS loads an OS (or a debug monitor like Tutor) into RAM from ROM or other non-volatile media such as a hard drive
  – Tutor/OS changes the addressing mode from “real mode” to “protected mode” which supports a flat 32 bit address space
  – Tutor/OS starts executing, interacts with the user, and controls running of user programs
Embedded System Reset / Boot

• An embedded system resets on power up
• Embedded systems may operate unattended by an operator, so a system failure could go unnoticed until some catastrophe occurs
• Other possible causes for system resets:
  – Hardware diagnostics (detects a hardware fault)
  – Software integrity checks (detects corrupt data)
  – A watchdog timer (detects an infinite loop)
  – Remote monitoring system (detects no response)
Operating System Support

• We have been running our embedded system projects under Tutor
  – Tutor is only a single user debug monitor
  – A real operating system such as Linux can support multiple users simultaneously

• Some key processor features are required to support multiple simultaneous users and prevent interference between them:
  – Kernel / User Modes of Operation
  – Memory Protection
Processor Modes

- Most CPU’s can execute code in two modes:
  - Kernel Mode (also called supervisor mode)
  - User Mode (also called application mode)

- In kernel mode, all privileged instructions are allowed including ones such as `sti`, `cli`, `lidt`, `cpuid`, `in`, `out`, etc.

- In user mode, those instructions are prohibited or may only be partially available based on the OS configuration
Processor Modes with Tutor

• The processor boots in kernel mode
• Tutor initializes itself in kernel mode and never switches the processor to user mode
• When we start a program with `go 100100`, it is running in kernel mode
• Hence, our code can execute all instructions and can make normal calls to functions such as our C library `inpt()`/`outpt()` or our callback functions in MP3 and MP5
Processor Modes with an OS

• The processor boots in kernel mode
• The operating system initializes itself and later provides its services in kernel mode
• Only “trusted” code executes in kernel mode
• When the operating system starts “untrusted” code (i.e. user programs), it changes the processor mode from kernel to user mode
• Execution of a prohibited instruction causes an exception to a kernel mode OS service
Processor Modes with an OS

• In user mode, there are only two ways to resume kernel mode operation
  – A hardware interrupt or exception occurs
  – Code makes a “system call” to an OS service using an instruction such as `int $n`

• Hence, ISR/Exception handling code and OS service functions run in kernel mode and must be trusted
Processor Modes with an OS

• Compiled C code makes normal calls to and expects normal returns from library functions which do not change the processor mode
• Hence, many C library functions take the parameters passed to them and reformat them into an OS system call, e.g. `int $n`
• That switches the processor to kernel mode
• The system service returns via `iret` and the library code is running in user mode again
Memory Protection

- Some “hacks” attempt to run user code in kernel mode to violate system security
- If code running in user mode can overwrite trusted kernel mode code, the system is not secure
- Processor memory protection features are one way that an OS can prevent corruption of its trusted code that runs in kernel mode
Memory Protection with Tutor

- Tutor does not utilize memory protection
- With Tutor, we could overwrite our own code or the Tutor code itself in memory
- That allowed us to run experiments that would not have been possible with an OS
Memory Protection with an OS

• With an OS, the critical memory areas for the OS are set up with memory protection
• These memory areas can be accessed only in kernel mode - not in user mode
• A user code attempt to access a prohibited location causes an exception to a kernel mode OS service
The “Downside” of an OS

- OS processor mode and memory protection sound great! Let’s always use them. Hmm.
- So what’s the possible downside?
- Performance!
- Using these features causes the OS to have a long context switching time between tasks.
- This may make it impossible to meet the real-time constraints of an embedded system.
The “Downside” of an OS

• There are versions of “embedded Linux”, Android, and commercial products such as Vxworks or Windows CE, that are intended for use on embedded systems.
• They are used in high-end embedded systems such as cell phones or gaming consoles.
• These devices are expensive enough to absorb the high costs of processor, memory, etc.
The “Downside” of an OS

• Some embedded system software may need to run “raw” on low-end and low-cost hardware without any OS or with only a minimal OS
  – Processor limitations
  – Memory size limitations
  – Hard real-time constraints
  – Costs for licenses or vendor support for an OS

• Think about the diving computer in lecture one, Arduino boards, appliances, control systems, alarm systems, smart thermostats, etc.
CISC / RISC Architectures

- Complex Instruction Set Computers
  - This is the traditional processor architecture
  - Complex instructions:
    - Can be of varying length (1 – 8 or more bytes)
    - Need to be decoded before they can be executed
    - Execution may include many steps
  - We have been studying the i386 processor which is based on a CISC architecture
CISC / RISC Architectures

• Reduced Instruction Set Computers
  – A more recently introduced architecture (1980’s)
  – RISC processors have simpler instruction sets:
    • Instructions are all the same length (typically 32 bits)
    • An instruction word doesn’t need to be decoded
    • Instructions do only one simple thing very fast
    • More instructions are needed to perform any given task
  – The AtMega328P used on our Arduino boards and the Advanced RISC Machines ARM family of processors are good examples
VLIW Architectures

• Very Long Instruction Word Computers
  – Another recently introduced architecture (1980’s)
  – Processors have a Very Long Instruction Word:
    • Instructions are all the same length (up to 1024 bits)
    • Have a different instruction field for each functional unit
  – The processor executes multiple instructions in parallel per clock cycle without elaborate HW to keep track of dependencies – so faster
  – The compiler must keep track of dependencies when it generates the code
  – The HP/Intel Itanium processor that we cover next time is a good example of this architecture
VLIW Architectures

• Example layout of a VLIW processor:
  – If a functional unit can’t be used, “nop” is coded

Very Long Instruction Word (Six Parallel Instructions)

Field A  Field B  Field C  Field D  Field E  Field F

Functional Unit A  Functional Unit B  Functional Unit C  Functional Unit D  Functional Unit E  Functional Unit F

Registers, Memory, and I/O Ports