CS61C so far...

C Programs
```
# include < stdlib . h >

int fib ( int n ) {
    return fib ( n - 1 ) + fib ( n - 2 );
}
```

MIPS Assembly
```
 lw $t0, 4 ($r0)
 addi $t1, $t0, 3
 beq $t1, $t2, foo
```

CPU

Caches

Memory

11/13/17

Fall 2017 -- Lecture #22
So How is a Laptop Any Different?
Adding I/O

C Programs
```
#include <stdlib.h>

int fib(int n) {
    return fib(n-1) + fib(n-2);
}
```

HW 1
```
.foo
lw $t0, 4($r0)
addi $t1, $t0, 3
beq $t1, $t2, foo
nop
```

Project 2

Project 1

CPU

Caches

Memory

Screen

Keyboard

Storage

I/O (Input/Output)

What kind of locality are we taking advantage of?

Fall 2017 -- Lecture #22
Raspberry Pi (Less than $40 on Amazon in 2017)
It’s a Real Computer!
But Wait...

• That’s not the same! When we run VENUS, it only executes one program and then stops.
• When I switch on my computer, I get this:

Yes, but that’s just software! The Operating System (OS)
Well, “Just Software”

- The biggest piece of software on your machine?
- How many lines of code? These are guesstimates:

Codebases (in millions of lines of code).
CC BY-NC 3.0 —
David McCandless © 2013
http://www.informationisbeautiful.net/visualizations/million-lines-of-code/

11/13/17
Operating System
What Does the OS do?

• OS is first thing that runs when computer starts
• Finds and controls all devices in the machine in a general way
  – Relying on hardware specific “device drivers”
• Starts services (100+)
  – File system,
  – Network stack (Ethernet, WiFi, Bluetooth, …),
  – TTY (keyboard),
  – …
• Loads, runs and manages programs:
  – Multiple programs at the same time (time-sharing)
  – Isolate programs from each other (isolation)
  – Multiplex resources between applications (e.g., devices)
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
How to Interact with Devices?

• Assume a program running on a CPU. How does it interact with the outside world?

• Need I/O interface for Keyboards, Network, Mouse, Screen, etc.
  – Connect to many types of devices
  – Control these devices, respond to them, and transfer data
  – Present them to user programs so they are useful
Instruction Set Architecture for I/O

• What must the processor do for I/O?
  – Input: read a sequence of bytes
  – Output: write a sequence of bytes

• Interface options
  a) Special input/output instructions & hardware
  b) Memory mapped I/O
    ▪ Portion of address space dedicated to I/O
    ▪ I/O device registers there (no memory)
    ▪ Use normal load/store instructions, e.g. \texttt{lw/sw}
    ▪ Very common, used by RISC-V
Memory Mapped I/O

• Certain addresses are not regular memory
• Instead, they correspond to registers in I/O devices

address
0x00000000

Memory-mapped
0x7FFFFFFF
0x8000000

Program & Data Memory
0xFFFFFFFF
Processor-I/O Speed Mismatch

• 1 GHz microprocessor I/O throughput:
  – 4 Gi-B/s \((1w/sw)\)
  – Typical I/O data rates:
    ▪ 10 B/s (keyboard)
    ▪ 100 Ki-B/s (Bluetooth)
    ▪ 60 Mi-B/s (USB 2)
    ▪ 100 Mi-B/s (Wifi, depends on standard)
    ▪ 125 Mi-B/s (G-bit Ethernet)
    ▪ 550 Mi-B/s (cutting edge SSD)
    ▪ 1.25 Gi-B/s (USB 3.1 Gen 2)
    ▪ 6.4 GiB/s (DDR3 DRAM)
  – These are peak rates – actual throughput is lower

• Common I/O devices neither deliver nor accept data matching processor speed
KEEP CALM
IT'S
BREAK TIME
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
Processor Checks Status before Acting

- Device registers generally serve two functions:
  - **Control Register**, says it’s OK to read/write (I/O ready)
    [think of a flagman on a road]
  - **Data Register**, contains data

- Processor reads from Control Register in loop
  - Waiting for device to set *Ready* bit in Control reg (0 → 1)
  - Indicates “data available” or “ready to accept data”

- Processor then loads from (input) or writes to (output) data register
  - I/O device resets control register bit (1 → 0)

- Procedure called “**Polling**”
I/O Example (Polling)

• Input: Read from keyboard into a0

\[
\text{Waitloop:}
\]
\[
lui \quad t0,0x7ffff \quad \#7ffff000 \quad (\text{io addr})
\]
\[
lw \quad t1,0(t0) \quad \#\text{read control}
\]
\[
andi \quad t1,t1,0x1 \quad \#\text{ready bit}
\]
\[
beq \quad t1,\text{zero},\text{Waitloop}
\]
\[
lw \quad a0,4(t0) \quad \#\text{data}
\]

• Output: Write to display from a1

\[
\text{Waitloop:}
\]
\[
lui \quad t0,0x7ffff \quad \#7ffff0000
\]
\[
lw \quad t1,8($t0) \quad \#\text{write control}
\]
\[
andi \quad t1,t1,0x1 \quad \#\text{ready bit}
\]
\[
beq \quad t1,\text{zero},\text{Waitloop}
\]
\[
sw \quad a1,12(t0) \quad \#\text{data}
\]

“Ready” bit is from processor’s point of view!
Cost of Polling?

• Assume for a processor with
  – 1 GHz clock rate
  – Taking 400 clock cycles for a polling operation
    ▪ Call polling routine
    ▪ Check device (e.g., keyboard or wifi input available)
    ▪ Return
  – What’s the percentage of processor time spent polling?

• Example:
  – Mouse
  – Poll 30 times per second
    ▪ Set by requirement not to miss any mouse motion
      (which would lead to choppy motion of the cursor on the screen)
Peer Instruction

Hard disk: transfers data in 16-Byte chunks and can transfer at 16 MB/second. No transfer can be missed. What percentage of processor time is spent in polling (assume 1 GHz clock)?

- 2%
- 4%
- 20%
- 40%
What is the Alternative to Polling?

• Polling wastes processor resources
• Akin to waiting at the door for guests to show up
  – What about a bell?
• Computer lingo for bell:
  – Interrupt
  – Occurs when I/O is ready or needs attention
    ▪ Interrupt current program
    ▪ Transfer control to special code “interrupt handler”
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
Traps/Interrupts/Exceptions: altering the normal flow of control

An *external or internal event* that needs to be processed - by another program - the OS. The event is often unexpected from original program’s point of view.
**Interrupt-Driven I/O**

1. Incoming interrupt suspends instruction stream
2. Looks up the vector (function address) of a handler in an interrupt vector table stored within the CPU
3. Perform a jal to the handler (*save PC in special MEPC* register)
4. Handler run on current stack and returns on finish (thread doesn’t notice that a handler was run)

---

**Handler Execution**

**Stack Frame**

**Stack Frame**

**Stack Frame**

**Label:**

- `sll t1,s3,2`
- `add t1,t1,s5`
- `lw t1,0(t1)`
- `or s1,s1,t1`
- `add s3,s3,s4`
- `bne s3,s2,Label`

---

**CPU Vector Interrupt Table**

<table>
<thead>
<tr>
<th>Interrupt</th>
<th>SPI0</th>
<th>handler</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*MEPC: Machine Exception Program Counter*
Terminology

In CS61C (other definitions in use elsewhere):

- **Interrupt** – caused by an event *external* to current running program
  - E.g., key press, disk I/O
  - Asynchronous to current program
    - Can handle interrupt on any convenient instruction
    - “Whenever it’s convenient, just don’t wait too long”

- **Exception** – caused by some event *during* execution of one instruction of current running program
  - E.g., divide by zero, bus error, illegal instruction
  - Synchronous
    - Must handle exception *precisely* on instruction that causes exception
    - “Drop whatever you are doing and act now”

- **Trap** – action of servicing interrupt or exception by hardware jump to “interrupt or trap handler” code
Precise Traps

• Trap handler’s view of machine state is that every instruction prior to the trapped one (e.g., overflow) has completed, and no instruction after the trap has executed.

• Implies that handler can return from an interrupt by restoring user registers and jumping back to interrupted instruction
  – Interrupt handler software doesn’t need to understand the pipeline of the machine, or what program was doing!
  – More complex to handle trap caused by an exception than interrupt

• Providing precise traps is tricky in a pipelined superscalar out-of-order processor!
  – But a requirement, e.g., for
    ▪ Virtual memory to function properly (see next lecture)
Exceptions are handled *like pipeline hazards*

1) Complete execution of instructions before exception occurred
2) Flush instructions currently in pipeline (i.e., convert to *nops* or “bubbles”)
3) Optionally store exception cause in status register
   - Indicate type of exception
   - Note: several exceptions can occur in a single clock cycle!
4) Transfer execution to trap handler
Trap Pipeline Diagram

\[
\begin{array}{cccccccc}
\text{time} & t0 & t1 & t2 & t3 & t4 & t5 & t6 & t7 \\
(\text{I}_1) 096: \text{DIV} & \text{IF}_1 & \text{ID}_1 & \text{EX}_1 & \text{MA}_1 & - & - & - & \ldots \\
(\text{I}_2) 100: \text{XOR} & \text{IF}_2 & \text{ID}_2 & \text{EX}_2 & - & - & - & - & - \\
(\text{I}_3) 104: \text{SUB} & \text{IF}_3 & \text{ID}_3 & - & - & - & - & - & - \\
(\text{I}_4) 108: \text{ADD} & \text{IF}_4 & - & - & - & - & - & - & - \\
(\text{I}_5) \text{Trap Handler code} & \text{IF}_5 & \text{ID}_5 & \text{EX}_5 & \text{MA}_5 & \text{WB}_5 & - & - & - \\
\end{array}
\]

*MEPC = 100 (instruction following offending ADD)
KEEP CALM
IT'S BREAK TIME
Administrivia

• Homework 5 is due tomorrow!
• Project 3: Performance Programming
  – Due Monday, Nov 20 (right before Thanksgiving break)
  – Project Party tomorrow 7-9pm in Cory 293

• Guerrilla Session tonight in 7-9pm in Cory 293
  – Topics covered: Parallelism & MapReduce (Floating Point if time permits)
  – Second to last guerrilla session—come and get valuable exam practice!

• Project 4 will be released by next Monday at the latest
  – You’ll get at least 2 weeks to complete it
  – Designed to be straightforward!
Agenda

- Devices and I/O
- Polling
- Interrupts
- **OS Boot Sequence**
- Multiprogramming/time-sharing
What Happens at Boot?

- When the computer switches on, it does the same as VENUS: the CPU executes instructions from some start address (stored in Flash ROM)

PC = 0x2000 (some default value)
What Happens at Boot?

1. **BIOS**: Find a storage device and load first sector (block of data)

2. **Bootloader** (stored on, e.g., disk): Load the OS kernel from disk into a location in memory and jump into it

3. **OS Boot**: Initialize services, drivers, etc.

4. **Init**: Launch an application that waits for input in loop (e.g., Terminal/Desktop/...)

*BIOS: Basic Input Output System*
Launching Applications

• Applications are called “processes” in most OSs
  – Thread: shared memory
  – Process: separate memory
  – Both threads and processes run (pseudo) simultaneously

• Apps are started by another process (e.g., shell) calling an OS routine
  (using a “syscall”)
  – Depends on OS, but Linux uses fork to create a new process, and execve (execute file
    command) to load application

• Loads executable file from disk (using the file system service) and puts
  instructions & data into memory (.text, .data sections), prepares stack and
  heap

• Set argc and argv, jump to start of main

• Shell waits for main to return (join)
Supervisor Mode

• If something goes wrong in an application, it could crash the entire machine. And what about malware, etc.?
• The OS enforces resource constraints to applications (e.g., access to memory, devices)
• To help protect the OS from the application, CPUs have a supervisor mode (e.g., set by a status bit in a special register)
  – A process can only access a subset of instructions and (physical) memory when not in supervisor mode (user mode)
  – Process can change out of supervisor mode using a special instruction, but not into it directly – only using an interrupt
  – Supervisory mode is a bit like “superuser”
    ▪ But used much more sparingly (most of OS code does not run in supervisory mode)
    ▪ Errors in supervisory mode often catastrophic (blue “screen of death”, or “I just corrupted your disk”)
Syscalls

• What if we want to call an OS routine? E.g.,
  – to read a file,
  – launch a new process,
  – ask for more memory (malloc),
  – send data, etc.

• Need to perform a syscall:
  – Set up function arguments in registers,
  – Raise software interrupt (with special assembly instruction)

• OS will perform the operation and return to user mode

• This way, the OS can mediate access to all resources, and devices
Agenda

• Devices and I/O
• Polling
• Interrupts
• OS Boot Sequence
• Multiprogramming/time-sharing
Multiprogramming

• The OS runs multiple applications at the same time
• But not really (unless you have a core per process)
• Switches between processes very quickly (on human time scale) – this is called a “context switch”
• When jumping into process, set timer interrupt
  – When it expires, store PC, registers, etc. (process state)
  – Pick a different process to run and load its state
  – Set timer, change to user mode, jump to the new PC
• Deciding what process to run is called scheduling
Protection, Translation, Paging

• Supervisor mode alone is not sufficient to fully isolate applications from each other or from the OS
  – Application could overwrite another application’s memory.
  – Typically programs start at some fixed address, e.g. 0x8FFFFFFF
    ▪ How can 100’s of programs share memory at location 0x8FFFFFFF?
  – Also, may want to address more memory than we actually have (e.g., for sparse data structures)

• Solution: Virtual Memory
  – Gives each process the illusion of a full memory address space that it has completely for itself
And, in Conclusion, ...

- Basic machine (datapath, memory, IO devices) are application agnostic
- Same concepts / processor architecture apply to large variety of applications. E.g.,
  - OS with command line and graphical interface (Linux, ...)
  - Embedded processor in network switch, car engine control, ...
- Input / output (I/O)
  - Memory mapped: appears like “special kind of memory”
  - Access with usual load/store instructions (e.g., lw, sw)
- Exceptions
  - Notify processor of special events, e.g. divide by 0, page fault (next lecture)
  - “Precise” handling: immediately at offending instruction
- Interrupts
  - Notification of external events, e.g., keyboard input, disk or Ethernet traffic
- Multiprogramming and supervisory mode
  - Enables and isolates multiple programs
- Take CS162 to learn more about operating systems