CS 61C: Great Ideas in Computer Architecture (Machine Structures)

Operating Systems, Interrupts

Instructors:
Nicholas Weaver & Vladimir Stojanovic
http://inst.eecs.berkeley.edu/~cs61c/
CS61C so far...

CPU

Caches

Memory

C Programs

MIPS Assembly

Project 1

Project 2

Labs
So how is this any different?
Adding I/O

C Programs

#include <stdlib.h>
int fib(int n) {
    return fib(n-1) +
    fib(n-2);
}

MIPS Assembly

.f.b
lw $t0, 4($r0)
addi $t1, $t0, 3
beq $t1, $t2, foo

Project 1

Project 2

CPU

Caches

Memory

Screen

Keyboard

Storage

I/O (Input/Output)

• Four##words/block,#cache#size#=#1K#words#

# Index:
# Data:
# Index:
# Tag:
# Valid:
# 0:
# 1:
# 2:
# .:
# .:
# .:
# 253:
# 254:
# 255:
# 31:
# 30:

32
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0

Byte offset

Instructions

Decoding

Register Read

Instruction Fetch

Control Unit
Raspberry Pi 3 ($35 on Amazon)
It’s a real computer!
But wait...

• That’s not the same! When we run MARS, 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/
What does the OS do?

• One of the first things that runs when your computer starts (right after firmware/bootloader)
• 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)
• Services: File System, Network stack, etc.
• Finds and controls all the devices in the machine in a general way (using “device drivers”)
Administrivia

• Project 4 delayed due date to tomorrow
  – But extra credit for turning it in today
• Project 5 will be out ASAP
  – I'm worried that we made it too easy, but eh...
Agenda

• Devices and I/O
• OS Boot Sequence and Operation
• Multiprogramming/time-sharing
• Introduction to Virtual Memory
Agenda

• Devices and I/O
  • OS Boot Sequence and Operation
  • Multiprogramming/time-sharing
  • Introduction to Virtual Memory
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: reads a sequence of bytes
  – Output: writes a sequence of bytes

• Some processors have special input and output instructions

• Alternative model (used by MIPS):
  – Use loads for input, stores for output (in small pieces)
  – Called Memory Mapped Input/Output
  – A portion of the address space dedicated to communication paths to Input or Output devices (no memory there)
Memory Mapped I/O

- Certain addresses are not regular memory
- Instead, they correspond to registers in I/O devices
Processor-I/O Speed Mismatch

• 1GHz microprocessor can execute 1B load or store instructions per second, or 4,000,000 KB/s data rate
  • I/O data rates range from 0.01 KB/s to 1,250,000 KB/s
• Input: device may not be ready to send data as fast as the processor loads it
  • Also, might be waiting for human to act
• Output: device not be ready to accept data as fast as processor stores it
• What to do?
Processor Checks Status before Acting

• Path to a device generally has 2 registers:
  • **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) to say it’s OK
• Processor then loads from (input) or writes to (output) data register
  • Load from or Store into Data Register resets Ready bit (1 ≈ 0) of Control Register
• This is called “Polling”
I/O Example (polling)

- **Input:** Read from keyboard into \$v0
  
  \begin{align*}
  \text{lui} & \quad \$t0, 0xffff \quad \text{ffff0000} \\
  \text{Waitloop:} & \quad \text{lw} \quad \$t1, 0(\$t0) \quad \#\text{control} \\
  \text{} & \quad \text{andi} \quad \$t1,\$t1,0x1 \\
  \text{} & \quad \text{beq} \quad \$t1,\$zero, \text{Waitloop} \\
  \text{} & \quad \text{lw} \quad \$v0, 4(\$t0) \quad \#\text{data}
  \end{align*}

- **Output:** Write to display from \$a0
  
  \begin{align*}
  \text{lui} & \quad \$t0, 0xffff \quad \text{ffff0000} \\
  \text{Waitloop:} & \quad \text{lw} \quad \$t1, 8(\$t0) \quad \#\text{control} \\
  \text{} & \quad \text{andi} \quad \$t1,\$t1,0x1 \\
  \text{} & \quad \text{beq} \quad \$t1,\$zero, \text{Waitloop} \\
  \text{} & \quad \underline{\text{sw}} \quad \underline{\$a0, 12}(\$t0) \quad \#\text{data}
  \end{align*}

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

• Assume for a processor with a 1GHz clock it takes 400 clock cycles for a polling operation (call polling routine, accessing the device, and returning). Determine % of processor time for polling
  – Mouse: polled 30 times/sec so as not to miss user movement
  – Hard disk: assume transfers data in 16-Byte chunks and can transfer at 16 MB/second. Again, no transfer can be missed. (we’ll come up with a better way to do this)
% Processor time to poll

- Mouse Polling [clocks/sec]
  \[= 30 \text{ polls/s} \times 400 \text{ clocks/poll} = 12K \text{ clocks/s}\]

- % Processor for polling:
  \[
  \frac{12 \times 10^3 \text{ clocks/s}}{1 \times 10^9 \text{ clocks/s}} = 0.0012\%
  \]
  ✍️ Polling mouse little impact on processor
Clicker Time

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 1GHz clock)?

- A: 2%
- B: 4%
- C: 20%
- D: 40%
- E: 80%
% Processor time to poll hard disk

- Frequency of Polling Disk
  \[= \frac{16 \text{ [MB/s]}}{16 \text{ [B/poll]}} = 1 \text{M [polls/s]}\]

- Disk Polling, Clocks/sec
  \[= 1 \text{M [polls/s]} \times 400 \text{ [clocks/poll]}\]
  \[= 400 \text{M [clocks/s]}\]

- % Processor for polling:
  \[\frac{400 \times 10^6 \text{ [clocks/s]}}{1 \times 10^9 \text{ [clocks/s]}} = 40\%\]

\[\Rightarrow\] Unacceptable

(Polling is only part of the problem – main problem is that accessing in small chunks is inefficient)
What is the alternative to polling?

• Wasteful to have processor spend most of its time “spin-waiting” for I/O to be ready
• Would like an unplanned procedure call that would be invoked only when I/O device is ready
• Solution: use exception mechanism to help I/O. **Interrupt** program when I/O ready, return when done with data transfer
• Allow to register (post) **interrupt handlers**: functions that are called when an interrupt is triggered
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 (needs to store any state)
4. Handler run on current stack and returns on finish (thread doesn’t notice that a handler was run)

```assembly
handler:
  lui  $t0, 0xffff
  lw   $t1, 0($t0)
  andi $t1,$t1,0x1
  lw   $v0, 4($t0)
  sw   $t1, 8($t0)
  ret
```

Stack Frame

```
Label: sll  $t1,$s3,2
       addu $t1,$t1,$s5
  lw   $t1,0($t1)
       add  $s1,$s1,$t1
       addu $s3,$s3,$s4
       bne  $s3,$s2,Label
```

CPU Interrupt Table

<table>
<thead>
<tr>
<th>Interrupt(SPI0)</th>
<th>SPI0</th>
<th>handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Direct Memory Access

• Complements interrupts:
  – The device itself can directly read or write to a specified block of memory
• Used to buffer transfers
  – DMA write the data
  – Then trigger an interrupt
• Can even go to great extremes
  – You can buy an FPGA-based network card which will directly write into process buffers
• We will go into this in more detail later on
Agenda

• Devices and I/O
• OS Boot Sequence and Operation
• Multiprogramming/time-sharing
• Introduction to Virtual Memory
What happens at boot?

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

PC = 0x2000 (some default value)
What happens at boot?

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

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/...
Validated Boot...

- The old-school BIOS (Basic Input/Output System) just started running whatever was in the boot sector
  - Allowed all sorts of shenanigans
- Modern firmware (UEFI (Universal Extensible Firmware Interface), iPhone, etc) performs validated boot
  - Cryptographically verifies that the boot code is signed by a valid cryptographic signature
- Essential to maintain a chain of trust
  - Trust the hardware and EFI to validate the boot...
- If you run Windows only, *turn this on!*
Launching Applications

• Applications are called “processes” in most OSs.
• Created by another process calling into an OS routine (using a “syscall”, more details later).
  – Depends on OS, but Linux uses `fork` to create a new process, and `execve` to load application.
• Loads executable file from disk (using the file system service: often just 'mapping' the file into memory to be loaded on demand, which we will get to when talking about virtual memory) and puts instructions & data into memory (.text, .data sections), prepare stack and heap.
• Set argc and argv, jump into the main function.
Supervisor Mode

• If something goes wrong in an application, it could crash the entire machine. And what about malware, etc.?

• The OS may need to enforce resource constraints to applications (e.g., access to devices).

• To help protect the OS from the application, CPUs have a supervisor mode bit.
  – 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.
Syscalls

• What if we want to call into an OS routine? (e.g., to read a file, launch a new process, send data, etc.)
  – Need to perform a syscall: set up function arguments in registers, and then raise software interrupt
  – OS will perform the operation and return to user mode

• Also, OS uses interrupts for scheduling process execution:
  – OS sets scheduler timer interrupt then drops to user mode and start executing a user task, when interrupts triggers, switch into supervisor mode, select next task to execute (& set timer) and drop back to user mode.

• This way, the OS can mediate access to all resources, including devices and the CPU itself.
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.
In CS61C (you’ll see other definitions in use elsewhere):

- **Interrupt** – caused by an event *external* to current running program (e.g. key press, mouse activity)
  - Asynchronous to current program, can handle interrupt on any convenient instruction

- **Exception** – caused by some event during execution of one instruction of current running program
  - Examples include integer overflow (add), lw/sw to invalid memory, not a valid opcode, etc...
  - Or deliberate syscall operation
  - Synchronous, must handle exception on instruction that causes exception

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

• *Trap handler’s view of machine state is that every instruction prior to the trapped one 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 (EPC register will hold the instruction address)
  – 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 handling imprecise interrupts in software is even worse.
Trap Handling in 5-Stage Pipeline

- How to handle multiple simultaneous exceptions in different pipeline stages?
- How and where to handle external asynchronous interrupts?
Save Exceptions Until Commit

- PC
- PC address Exception
- Inst. Mem
- D
- Decode
- Illegal Opcode
- E
- +
- Overflow
- M
- Data Mem
- Commit Point
- Kill D
- Kill E
- Kill F
- Asynchronous Interrupts
- Data address Exceptions
- Select Handler
- PC
- Writeback
- EPC Cause
Handling Traps in In-Order Pipeline

• Hold exception flags in pipeline until commit point (M stage)
• Exceptions in earlier instructions override exceptions in later instructions
• Exceptions in earlier pipe stages override later exceptions *for a given instruction*
• Inject external interrupts at commit point (override others)
• If exception/interrupt at commit: update Cause and EPC registers, kill all stages, inject handler PC into fetch stage
Trap Pipeline Diagram

\[
\begin{align*}
\text{time} & \quad \text{t0} \quad \text{t1} \quad \text{t2} \quad \text{t3} \quad \text{t4} \quad \text{t5} \quad \text{t6} \quad \text{t7} \quad \ldots \ldots \\
(I_1) \quad \text{096: ADD} & \quad \text{IF}_1 \quad \text{ID}_1 \quad \text{EX}_1 \quad \text{MA}_1 \quad \text{overflow!} \\
(I_2) \quad \text{100: XOR} & \quad \text{IF}_2 \quad \text{ID}_2 \quad \text{EX}_2 \quad - \quad - \\
(I_3) \quad \text{104: SUB} & \quad \text{IF}_3 \quad \text{ID}_3 \quad - \quad - \quad - \\
(I_4) \quad \text{108: ADD} & \quad \text{IF}_4 \quad - \quad - \quad - \quad - \\
(I_5) \quad \text{Trap Handler code} & \quad \text{IF}_5 \quad \text{ID}_5 \quad \text{EX}_5 \quad \text{MA}_5 \quad \text{WB}_5
\end{align*}
\]
In Conclusion

- Once we have a basic machine, it’s mostly up to the OS to use it and define application interfaces.
- Hardware helps by providing the right abstractions and features (e.g., Virtual Memory, I/O).
- If you want to learn more about operating systems, you should take CS162!
- What’s next in CS61C?
  - More details on I/O
  - More about Virtual Memory