

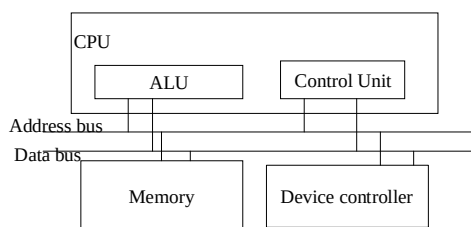
Computer System Organization

S H Srinivasan
shs@cs.ucsd.edu

von Neumann Architecture

- CPU
 - arithmetic logical unit
 - control unit
- Memory
- I/O
- Bus

von Neumann Architecture



Control Unit

- Operation
 - fetch
 - decode
 - execute
- Registers
 - PC (program counter)
 - IR (instruction register)
 - PS (processor status)
 - SP (stack pointer)

Control unit operation

```
PC = <machine start address>;
IR = memory[PC];
haltFlag = FALSE;
while (!haltFlag) {
    execute(IR);
    PC = PC + sizeof(INSTRUCTION);
    IR = memory[PC];
}
```

Memory

- MAR
- MDR
- command: Read/Write

ALU

- General purpose registers
- Functional units: addition, multiplication, etc.
- Status flags: carry, overflow, etc.

Devices

- Control register
 - accepts commands
- Status register: *ready*, *busy*
 - polling
- Data registers
- I/O: character, block

I/O operation

```
while (device.busy || device.done);
device.data[0] = <value to write>
device.command = WRITE;
while(device.busy);
device.done = TRUE;
```

The above illustrates “busy-wait” operation
This can be speed up using
interrupts
DMA

Interrupts

- Asynchronous input to CPU
- Processor state (PC) needs to be saved
 - *where?* Stack
 - one more register: SP
- Priorities: each interrupt has a priority
- Masking
 - IPL: interrupt priority level
 - if (input IPL < current IPL) ignore the interrupt

Interrupt processing

/ assuming the interrupt is not masked */*

- (1) push PC and PS on the stack
- (2) set current IPL = input IPL
- (3) handle interrupt */* jump to service routine */*
- (4) restore the IPL
- (5) restore the PC and PS (from the stack)

DMA

- Device $\longleftrightarrow$ Memory data transfer is usually through the CPU
 - read device, write memory or read memory, write device
- Huge data transfer between memory and device: lot of overhead
- DMA: Direct device $\longleftrightarrow$ memory data transfer without CPU intervention

Bootstrapping

- Power-up: CPU starts executing from a fixed location
- This program reads a fixed number of bytes from a fixed location in disk (*boot sector*)
- The program in the boot sector is called the *bootstrap loader*
- The bootstrap loader loads the OS

Hardware support

- Processor modes
- Special instructions
 - traps
- Atomic operations
 - test and set
- Exceptions

Processor Modes

- Two modes: supervisor and user
- Supervisor
 - all instructions can be executed, e.g., halt
 - all memory can be accessed
- User
 - subset of instructions and memory

Traps

- “trap” is a software instruction with a single parameter
 - example, “trap 40”
- can be executed in the user mode
- when executed, changes the mode to supervisor and calls a subroutine associated with the number (“40”)

Traps (cont’d)

- These subroutines are written by the OS developer and not the user
- *system calls* are basically C language wrappers for traps

Exceptions

- Exceptions signal error conditions
 - illegal memory access
 - divide by zero
- Exceptions are equivalent to traps except that they are invoked by the hardware

- traps, exceptions: *synchronous*
 - caused by the current instruction
- interrupts: *asynchronous*
 - caused by external events like user typing
- traps: invoked by user
- exceptions: invoked by hardware because of error

System parameters

- Data, Address size
- Clock speed
- Bandwidth: memory, device

What does the OS provide?

- Multiple processors
- Unlimited memory
- Easy I/O
- *How?*

What does a program really want?

- Current instruction (*locality*)
- Register values: PC, SP, SR, other registers
- Nothing else!
- We can execute the program one instruction at a time, keeping only one instruction in memory.

What does the kernel do? (Highly simplified)

- Take one program
 - restore the register values
 - execute one instruction
 - save the register values
- Take another program
 - repeat the same operation
- In practice, kernel executes the program for one *time slice* and keeps several *pages* in memory.
- Kernel needs some machinery to do this

Timer

- Timer is absolutely essential for time slice allocation, error detection (*timeout*), etc.
- Most systems also have a realtime clock
- timer vs. cpu clock vs. wall clock

Programming abstraction for processes

```
if ((pid = fork()) == 0) {  
    /* child code */  
    /* begins independent execution */  
}  
else {  
    /* error */  
}  
  
/* rest of parent code */  
  
/* if there is no error, there are two processes here */
```