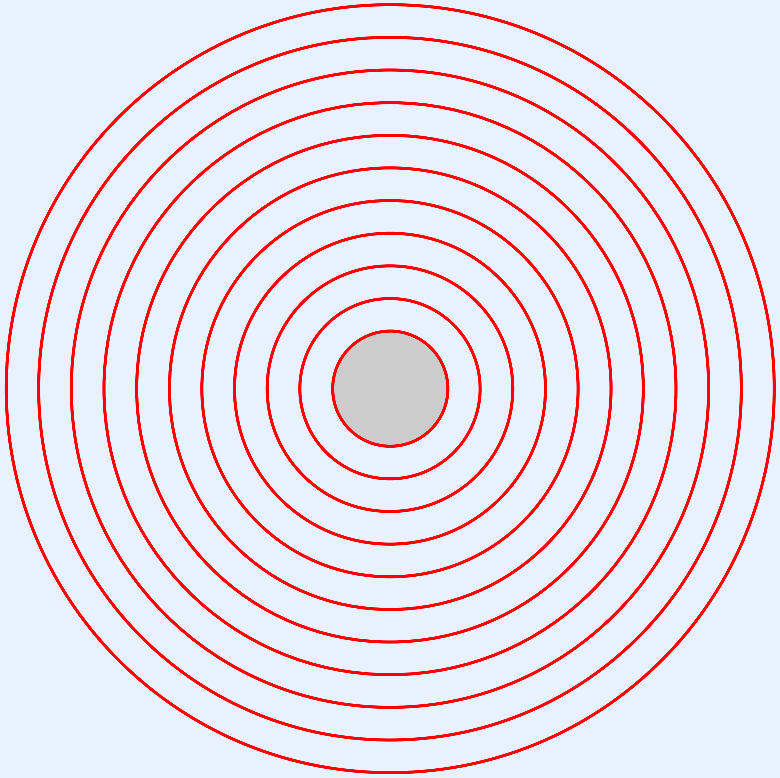
PART 3

Review Of Third Generation Architecture And Runtime Systems

Location Of Hardware In The Hierarchy



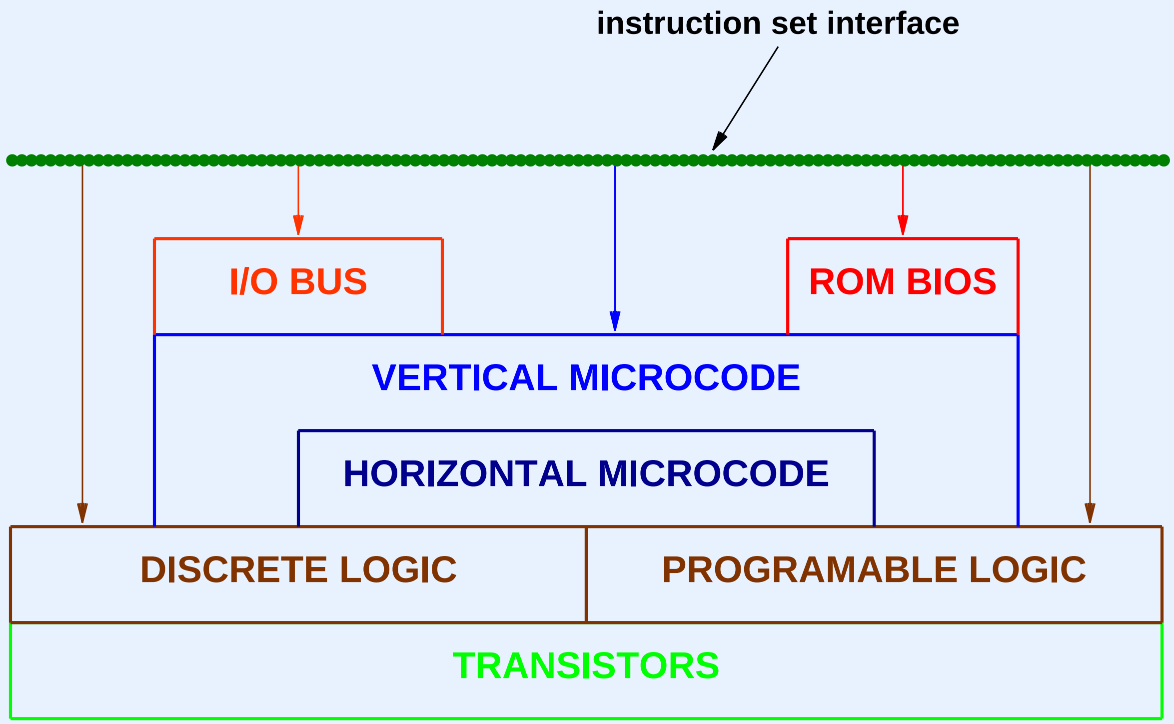
Features Of A Third Generation Machine

- Processor
- Memory system
- I / O Devices
- Interrupts

Processor

- Instruction set
- General-purpose and special-purpose registers
- Addressing modes
- Protection states
- ROM code

What Interface Does An Operating System See?



- Multiple levels of hardware
- Each level contributes
- Result is *instruction set*

Effective Instruction Set Composition

- Discrete & programmable logic
- Microcode
  - Low-level (*horizontal*)
  - High-level (*vertical*)
- ROM routines
  - Example: BIOS functions on a PC
- Note: OS can use all

Registers

- Local storage
- Store active values during computation (e.g., used to compute an expression)
- Saved and restored during subprogram invocation
- May control processor mode and address space visibility

Memory System

- Defines size of a *byte*, the smallest addressable unit
- Important property: endianness
- Provides address space, typically
  - *Monolithic*
  - *Linear*
- Includes caching

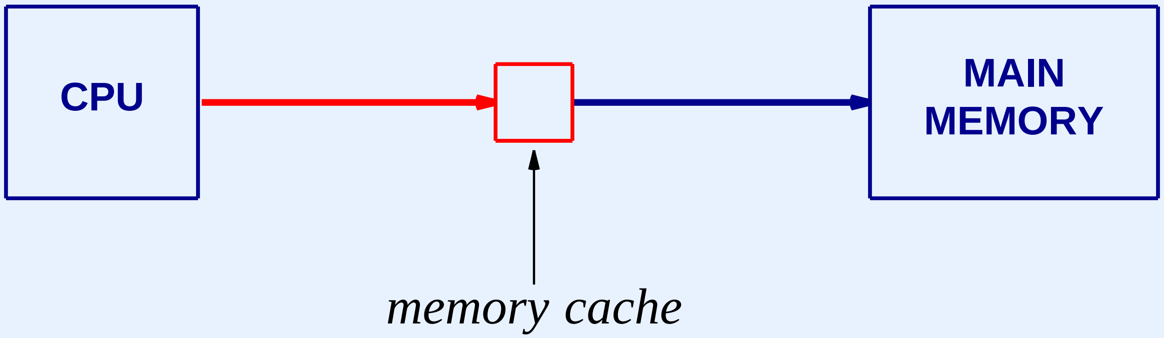
Byte Order

- Order of bytes in integer
- Least-significant byte at lowest address
  - Called *Little Endian*
- Most-significant byte at lowest address
  - Called *Big Endian*

Memory Caches

- Special-purpose hardware units
- Speed memory access
- Less expensive than high-speed memory
- Placed “between” CPU and memory

Conceptual Placement Of Memory Cache



- All references (including instruction fetch) go through cache
- Multi-level cache possible
- Key question: are virtual or physical addresses cached?

I / O Devices

- Wide variety of peripheral devices available
  - Keyboard / mouse
  - Disk
  - Wired or wireless network interface
  - Printer
  - Scanner
  - Camera
  - Sensors
- Multiple transfer paradigms (character, block, packet, stream)

Communication Between Device And CPU

- I/O through *bus*
  - Parallel wires
  - One or more per computer
- CPU uses bus to
  - Interrogate device
  - Control (start or stop) device
- Device uses bus to
  - Transfer data
  - Inform CPU of status

Bus Fundamentals

- Wires on bus divided into
  - Address lines
  - Data lines
  - Control lines
- Only two basic bus operations
  - Fetch
  - Store

Bus Operations

- Fetch
  - CPU places address on bus
  - CPU uses control line to signal *fetch request*
  - Device senses its address
  - Device puts specified data on bus
  - Device uses control line to signal *response*

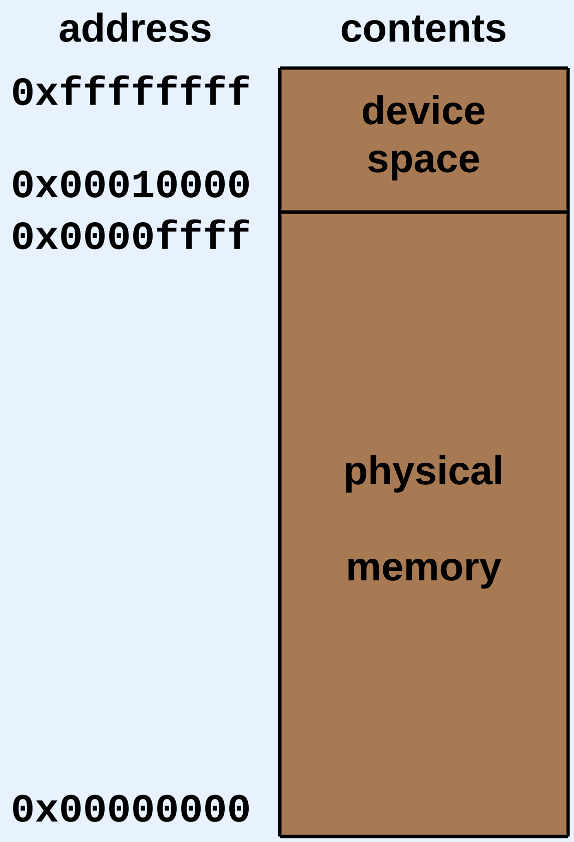
Bus Operations
(continued)

- Store
  - CPU places data and address on bus
  - CPU uses control line to signal *store request*
  - Device senses its address
  - Device extracts data from the bus
  - Device uses control line to signal *data extracted*

Bus Access By CPU

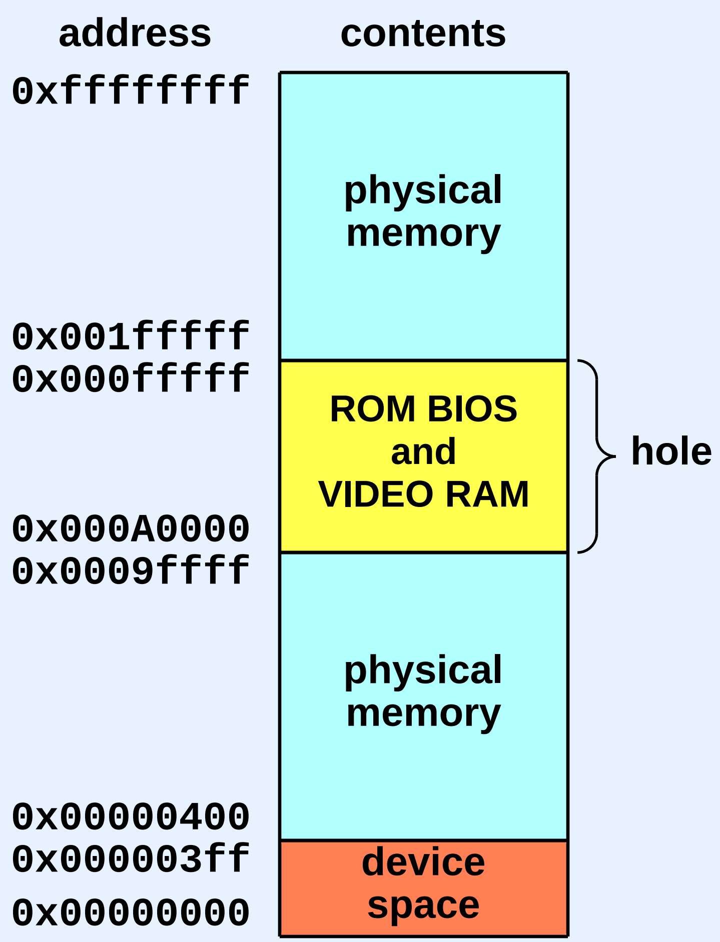
- Two basic approaches
  - Special instruction(s) used to access bus
  - Bus mapped into same address space as memory
    - * Devices placed beyond physical memory
    - * CPU uses normal fetch / store memory instructions
    - * Known as *memory-mapped I / O*

Illustration Of Address Space On A Typical 32-Bit Embedded System



- Processor uses conventional memory operations to communicate with devices

Address Space In An Intel PC



- Nonlinear for backward compatibility
- “Hole” in physical memory from 640KB to 1MB

Interrupt Mechanism

- Fundamental role in modern system
- Permits I/O concurrent with execution
- Allows device priority
- Informs CPU when I/O finished
- *Software interrupt* also possible

Interrupt-Driven I / O

- CPU starts device
- Device operates concurrently
- Device interrupts the CPU when finished with the assigned task
- Interrupt timing
  - Asynchronous wrt instructions
  - Synchronous wrt an individual instruction (occurs between instructions)

Interrupt Details

- Device and CPU communicate over bus
- Device posts an interrupt
- CPU polls bus during fetch / execute cycle
- CPU requests interrupt vector
- Device sends interrupt number to CPU
- CPU saves program state (e.g., by pushing onto the stack)
- CPU uses interrupt number to fetch new program state from the interrupt vector in memory
- CPU continues the fetch / execute cycle

Interrupt Mask

- Bit mask kept in CPU status register
- Set by hardware when interrupt occurs; can be reset by OS
- Determines which interrupts are permitted
- Priorities
  - Each device assigned priority level (binary number)
  - When servicing level *K* interrupt, mask set to disable interrupts at level *K* and lower

Operating System Responsibility

- Operating system must
  - Store correct information in interrupt vector for each device
  - Arrange for interrupt code to save registers used during the interrupt
  - Arrange for interrupt code to restore registers before returning from interrupt
  - Distinguish among devices, including multiple physical copies of a given device type

Returning From An Interrupt

- Special hardware instruction used
- Atomically restores
  - Old program state
  - Interrupt mask
  - Program counter
- After a return from interrupt, the interrupted code continues and registers are unchanged

Transfer Size And Interrupts

- Interrupt occurs after I / O operation completes
- Transfer size depends on device
  - Serial port transfers one character
  - Disk controller transfers one block (e.g., 512 bytes)
  - Network interface transfers one packet
- Large transfers use *Direct Memory Access* (*DMA*)

Direct Memory Access (DMA)

- Hardware mechanism
- I / O device transfers data to / from memory
  - Occurs over bus
  - Does not involve CPU
- Example use
  - Transfer incoming network packet to memory

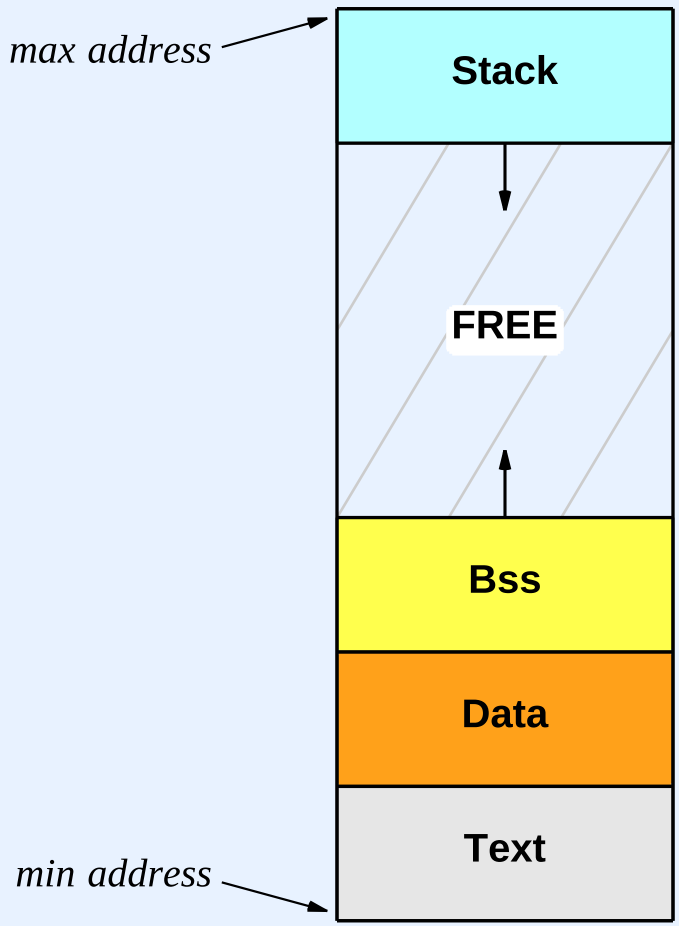
Direct Memory Access
(continued)

- Motivation
  - Free CPU from I / O
- Interface hardware that uses DMA
  - More expensive
  - May contain RAM, ROM and microprocessor
  - More complex to design

Memory Segments In C Programs

- C Program has four primary data areas called *segments*
- Text segment
  - Contains program code
  - Usually unwritable
- Data segment
  - Contains initialized data values (globals)
- Bss segment
  - Contains uninitialized data values
- Stack segment
  - Used for procedure calls

Typical C Storage Layout



- Stack grows downward
- Heap grows upward

Symbols For Segment Addresses

- C compiler and / or linker adds three reserved names to symbol table
- _*etext* lies beyond text segment
- _*edata* lies beyond data segment
- _*end* lies beyond bss segment
- Only the addresses are significant; values are irrelevant
- Program can use the addresses of the reserved symbol to determine the size of segments
- Note: names are declared to be *extern* without the underscore:

```
extern   int      end;
```

Runtime Storage For C Function Running As A Process

- Text can be shared
- Data areas *may* be shared
- Stack cannot be shared
- Exact details depend on address space model OS offers

Example Runtime Storage Model: Xinu

- Single, shared copy of
  - Text segment
  - Data segment
  - Bss segment
- One stack segment per process

Summary

- Components of third generation computer
  - Processor
  - Main memory
  - I / O Devices
    * Accessed over bus
    * Operate concurrently with CPU
    * Usually memory mapped
    * Can use DMA
    * Use interrupts

Summary
(continued)

- Interrupt mechanism
  - Informs CPU when I/O completes
  - Permits asynchronous device operation
- C uses four memory areas: text, data, bss, and stack segments
- Multiple concurrent computations
  - Can share text, data, and bss
  - Cannot share stack