Module I

Course Introduction
And Overview
The Big Questions

- Most CS programs require an architecture course, but you might ask:
The Big Questions

- Most CS programs require an architecture course, but you might ask:

  Is knowledge of computer organization and the underlying hardware relevant these days?

  Should we take this course seriously?
The Answers

- Companies (such as Google, IBM, Microsoft, Apple, Cisco,...) look for knowledge of architecture when hiring (i.e., understanding computer architecture can help you land a job)

- The most successful software engineers understand the underlying hardware (i.e., knowing about architecture can help you earn promotions)

- As a practical matter: knowledge of computer architecture is needed for later courses, such as systems programming, compilers, operating systems, and embedded systems
A Word About Future Employment
A Word About Future Employment

- Traditional software engineering jobs are saturated
A Word About Future Employment

- Traditional software engineering jobs are saturated
- The future lies in embedded systems
  - Cell phones
  - Video games
  - MP3 players
  - Set-top boxes
  - Smart sensor systems
A Word About Future Employment

- Traditional software engineering jobs are saturated
- The future lies in embedded systems
  - Cell phones
  - Video games
  - MP3 players
  - Set-top boxes
  - Smart sensor systems
- Understanding architecture is key for programming embedded systems
Some Bad News About Architecture
Some Bad News About Architecture

• Hardware is ugly
  – Lots of low-level details
  – Can be counterintuitive

• Hardware is tricky
  – Timing is important
  – A small addition in functionality can require many pieces of hardware

• The subject is so large that we cannot hope to cover it in one course

• You will need to think in new ways
Some Good News About Architecture
Some Good News About Architecture

• You will learn to think in new ways
• It is possible to understand basics without knowing all low-level technical details
• Programmers only need to learn the essentials
  – Characteristics of major components
  – Role in overall system
  – Consequences for software
The Four Main Topics

• Basics of digital hardware
  – You will build a few simple circuits

• Processors
  – You will program RISC and CISC processors in lab

• Memories
  – You will learn about memory organization and caching

• I/O operates
  – You will explore buffering and learn about interrupts
Organization Of The Course

• Basics
  – A taste of digital logic
  – Data paths and execution
  – Data representations

• Processors
  – Instruction sets and operands
  – Assembly languages and programming

• Memories
  – Physical and virtual memories
  – Addressing and caching
Organization Of The Course
(continued)

- Input/Output
  - Devices and interfaces
  - Buses and bus address spaces
  - Role of device drivers
- Advanced topics
  - Parallelism and data pipelining
  - Power and energy
  - Performance and performance assessment
  - Architectural hierarchies
What We Will Not Cover

• The course emphasizes breadth over depth

• Omissions
  – Most low-level details (e.g., discussion of electrical properties of resistance, voltage, current and semiconductor physics)
  – Quantitative analysis that engineers use to design hardware circuits
  – Design rules that specify how logic gates may be interconnected
  – Circuit design and design tools
  – VLSI chip design and languages such as Verilog
Terminology Used With Digital Systems

• Three key ideas
  – Architecture
  – Design
  – Implementation
Architecture

• Refers to the overall organization of a computer system
• Analogous to a blueprint
• Specifies
  – Functionality of major components
  – Interconnections among components
• Abstracts away details
Design

- Needed before a digital system can be built
- Translates architecture into components
- Fills in details that the architectural specification omits
- Specifies items such as
  - How components are grouped onto boards
  - How power is distributed to boards
- Many designs can satisfy a given architecture
Implementation

- All details necessary to build a system
- Includes
  - Specific part numbers to be used
  - Mechanical specifications of chassis and cases
  - Layout of components on boards
  - Power supplies and power distribution
  - Signal wiring and connectors
Summary

- Architecture is required because understanding computer organization leads to programming excellence
- This course covers the four essential aspects of computer architecture
  - Digital logic
  - Processors
  - Memory
  - I/O
- You will have fun with hardware in the lab
Questions?
Module II

Fundamentals Of Digital Logic
Our Goals

- Understand the basics
  - Concepts
  - How computers work at the lowest level

- Avoid whenever possible
  - Device physics
  - Engineering design rules
  - Implementation details
Electrical Terminology

- **Voltage**
  - Quantifiable property of electricity
  - Measure of potential force
  - Unit of measure: *volt*

- **Current**
  - Quantifiable property of electricity
  - Measure of electron flow along a path
  - Unit of measure: *ampere (amp)*
Analogy

• Voltage is analogous to water pressure
• Current is analogous to flowing water
• Can have
  – High pressure with little flow
  – Large flow with little pressure
Measuring Voltage

- Device used is called *voltmeter*
- Note: can only be measured as *difference* between two points
- We will
  - Assume one point represents zero volts (known as *ground*)
  - Express voltage of second point with respect to ground
In Practice

- In lab, chips will operate on five volts
- Two wires connect each chip to *power supply*
  - Ground (zero volts)
  - Power (five volts)
- Every chip needs power and ground connections
- Notes
  - Logic diagrams do not show power and ground
  - Raspberry Pi operates on 3.3 volts, so conversion is required to connect the Pi to other chips
Transistor

- Basic building block of electronic circuits
- Operates on electrical current
- Traditional transistor
  - Has three external connections
    * Emitter
    * Base (control)
    * Collector
  - Acts like an amplifier — a small current between base and emitter controls large current between collector and emitter
Illustration Of A Traditional Transistor

- **Amplification** means the large output current varies exactly like the small input current.
Field Effect Transistor

- Called a \textit{Metal Oxide Semiconductor FET (MOSFET)} when used on a CMOS chip
- Three external connections
  - Source
  - Gate
  - Drain
- Designed to act as a switch (on or off)
  - When the input reaches a threshold (i.e., becomes logic 1), the transistor turns on and passes full current
  - When the input falls below a threshold (i.e., becomes logic 0), the transistor turns off and passes no current
Illustration Of A Field Effect Transistor
(Used For Switching)

- Input arrives at the gate
- Logic zero (zero volts) means the transistor is off; logic 1 (positive voltage) turns the transistor on
Alternative Field Effect Transistor
(Also Used For Switching)

- Circle on the gate indicates an inversion
- Logic 0 (zero volts) turns the transistor on, and logic 1 (positive voltage) turns the transistor off
Boolean Logic

• Mathematical basis for digital circuits
• Three basic functions: *and*, *or*, and *not*

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A and B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A or B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>not A</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Digital Logic

- Can implement Boolean functions with transistors
- Five volts represents Boolean 1 (*true*)
- Zero volts represents Boolean 0 (*false*)
Transistors Implementing Boolean Not

- When input is zero volts, output is connected to + voltage
- When input is five volts, output is connected to 0 volts
- Hardware engineers use $V_{dd}$ to denote positive voltage
Logic Gate

- Hardware component
- Consists of integrated circuit
- Implements an individual Boolean function
- To reduce complexity, hardware uses inverse of Boolean functions
  - Nand gate implements not and
  - Nor gate implements not or
  - Inverter implements not
### Truth Tables For Nand, Nor, and Xor Gates

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A nand B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A nor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A xor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Example Of Internal Gate Structure (Nand Gate)

- Solid dot indicates electrical connection
Symbols Used In Schematic Diagrams

- Basic gates

![NAND gate](image)

![NOR gate](image)

![Inverter](image)

![AND gate](image)

![OR gate](image)

![XOR gate](image)
Technology For Logic Gates

- Most popular technology known as *Transistor-Transistor Logic (TTL)*
- Allows direct interconnection (a wire can connect output from one gate to input of another)
- Single output can connect to multiple inputs
  - Called *fanout*
  - Limited to a small number
Example Interconnection Of TTL Gates

- Suppose we need a signal to indicate that the power button is depressed and the disk is ready
- Two logic gates are needed to form logical *and*
  - Output from nand gate connected to input of inverter
Consider The Following Circuit

Question: what does the circuit implement?
Two Ways To Describe A Circuit

- Boolean expression
  - Often used when designing circuit
  - Can be transformed to equivalent version that requires fewer gates
- Truth table
  - Enumerates inputs and outputs
  - Often used when debugging a circuit
Describing A Circuit With Boolean Algebra

- Value at point A is: not Y
- Value at point B is: Z nor (not Y)
Describing A Circuit With Boolean Algebra

- Value at point C is: \((X \text{ nand } ((Z \text{ nor } (not\ Y)))\)
- Value at output is: \(X \text{ and } (Z \text{ nor } (not\ Y))\)
Simplifying Boolean Expressions

• Rules are similar to conventional algebra
  – Associative
  – Reflexive
  – Distributive
• See Appendix 2 in the text for details
Describing A Circuit With A Truth Table

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- Table lists all possible inputs and output for each
- Can also state values for intermediate points
Nand/Nor Vs. And/Or

- Mathematically, \textit{nand/nor/not} is equivalent to \textit{and/or/not}
- Practically
  - It is possible to construct \textit{and} and \textit{or} gates
  - Sometimes, humans find \textit{and} and \textit{or} operations easier to understand
- Example circuit or truth table output can be described by Boolean expression:

  \[ X \text{ and } Y \text{ and } (\text{not } Z) \]
Binary Addition

• How does a computer perform addition?
• Analogous to the method used in elementary school
• Each digit is a single bit

<table>
<thead>
<tr>
<th>1 0 1 0 0</th>
<th>+</th>
<th>1 1 1 0 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 0 0 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: first bit never has a carry input
Half-Adder Circuit

- Adds two input bits
- Produces two output bits
  - Sum
  - Carry
- We will use *exclusive or* gate plus *and* gate
Full-Adder Circuit

- Input is two bits plus a carry
- Produces two output bits
  - Sum
  - Carry
In Practice

- A single gate only has a few connections
- A chip has many pins for external connections
- Result: package multiple gates on each chip
- We will see examples shortly
An Example Logic Gate Technology

- 7400 family of chips
- Package is about one-half inch long
- Implement TTL logic
- Powered by five volts
- Each chip contains multiple gates
Example Gates On 7400-Series Chips

- Pins 7 and 14 connect to ground and power
- Power and ground *must* be connected for the chip to operate
Logic Gates And Computers
Logic Gates And Computers

• Question: how can computers be constructed from simple logic gates?
Logic Gates And Computers

• Question: how can computers be constructed from simple logic gates?
• Answer: they cannot
Logic Gates And Computers

• Question: how can computers be constructed from simple logic gates?
• Answer: they cannot
• Logic gates only provide a Boolean combination of inputs (known as \textit{combinatorial circuits})
Logic Gates And Computers

• Question: how can computers be constructed from simple logic gates?
• Answer: they cannot
• Logic gates only provide a Boolean combination of inputs (known as *combinatorial circuits*)
• Additional functionality is needed
  – Circuits that maintain state
  – Circuits that operate on a clock
Circuits That Maintain State

- More sophisticated than combinatorial circuits
- Output depends on history of previous input as well as values on input lines
Basic Circuit That Maintains State

- Known as *latch*
- Has two inputs: *data* and *enable*
- When enable is 1, output is same as data
- When enable goes to 0, output stays locked at current value
Propagation Delay
Propagation Delay

- Key in understanding a latch
Propagation Delay

- Key in understanding a latch
- Consider the circuit

- What does it do?
Propagation Delay

- Key in understanding a latch
- Consider the circuit

\[ d \]

What does it do?

- Mathematically, the circuit is meaningless because an inverter produces the complement of its input, but in this case the output is fed back into the input
- Practically, a propagation delay means the output stays the same for a short time, and then changes
- Result: output varies over time, 0 for time \( t \), 1 for time \( t \), 0 for time \( t \), and so on, where \( t \) is the propagation delay
Register

- Basic building block for a computer
- Acts like a miniature N-bit memory
- Can be built out of latches
A More Complex Circuit That Maintains State

• Basic *flip-flop*

• Can be constructed from a pair of latches

• Analogous to push-button power switch (i.e., push-on push-off)

• Each new 1 received as input causes output to reverse
  – First input pulse causes flip-flop to turn on
  – Second input pulse causes flip-flop to turn off
Output Of A Flip-Flop

Note: output only changes when input makes a transition from zero to one (i.e., *rises*)
Flip-Flop Action Plotted As Transition Diagram

- All changes synchronized with clock (described later)
- Output changes on *rising edge* of input
- Also called *leading edge*
Binary Counter

- Counts input pulses
- Output is binary value
- Includes reset line to restart count at zero
- Example: 4-bit counter available as single integrated circuit
• Part (a) shows the schematic of a counter chip
• Part (b) shows the output as the input changes
Clock

- Permits active circuits
- Electronic circuit that pulses regularly
- Measured in cycles per second (Hz)
- Output of clock is *square wave* (sequence of 1 0 1 0 1 ... )
Decoder/Demultiplexor

- Takes binary number as input
- Uses input to select one output
- Technical distinction
  - *Decoder* simply selects one of its outputs
  - *Demultiplexor* feeds a special input to the selected output
- In practice: engineers often use the term “demux” for either, and blur the distinction
Illustration Of Decoder

- Binary value on input lines determines which output is active

- Technical detail: on some decoder chips, an active output is logic 0 and all others are logic 1
Example: Execute A Sequence Of Steps

- Imagine the power-on sequence for an embedded system
  - Test the battery
  - Power on and test the memory
  - Start the disk
  - Power up the display
  - Read boot sector from disk into memory
  - Start the CPU
- Separate hardware module performs each task
- Need to activate the modules in sequence
Circuit To Execute A Sequence

- Technique: count clock pulses and use decoder to select an output for each possible counter output

- Note: counter will wrap around to zero, so this is an infinite loop
Feedback

- Output of circuit used as an input
- Called *feedback*
- Allows more control
- Example: stop sequence when output $F$ becomes active
- Boolean algebra

$\text{CLOCK and (not F)}$
Illustration Of Feedback For Termination

- Note additional input needed to restart sequence
A Fundamental Difference

- Software
  - Uses iteration
  - Software engineers are taught to avoid replicating code
  - Iteration increases elegance

- Hardware
  - Uses replicated (parallel) hardware units
  - Hardware engineers are taught to avoid iterative circuits
  - Replication increases performance and reliability
Using Spare Gates

- Note: because chip contains multiple gates, some gates may be unused
- May be possible to reduce total chips needed by employing unused gates
- Example: use a spare nand gate as an inverter by connecting one input to five volts:

\[
1 \textit{nand} \ x \ = \ \textit{not} \ x
\]

- Previous circuit can be implemented with a single chip (a quad 2-input \textit{nand} gate)
Practical Engineering Concerns

- Power consumption (wiring must carry sufficient power)
- Heat dissipation (chips must be kept cool)
- Timing (gates take time to settle after input changes)
- Clock synchronization (clock signal must travel to all chips simultaneously)
- Difference in clock signals (clock skew) can cause problems
Illustration Of Clock Skew

- Length of wire determines time required for signal to propagate
Clockless Logic

- Active circuits built without a clock
- Advantages
  - Possible power savings
  - Avoids clock skew
- Uses two wires to transfer a bit

<table>
<thead>
<tr>
<th>Wire 1</th>
<th>Wire 2</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Reset before starting a new bit</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Transfer a 0 bit</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Transfer a 1 bit</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Undefined (not used)</td>
</tr>
</tbody>
</table>
Moore’s Law And Classifications

- Gordon Moore predicted that the number of transistors on a chip would double each year (revised in 1970 to every 18 months)

- Led to the following classifications

<table>
<thead>
<tr>
<th>Name</th>
<th>Example Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Scale Integration (SSI)</td>
<td>The most basic logic such as Boolean gates</td>
</tr>
<tr>
<td>Medium Scale Integration (MSI)</td>
<td>Intermediate logic such as counters</td>
</tr>
<tr>
<td>Large Scale Integration (LSI)</td>
<td>More complex logic such as embedded processors</td>
</tr>
<tr>
<td>Very Large Scale Integration (VLSI)</td>
<td>The most complex processors (i.e., CPUs)</td>
</tr>
</tbody>
</table>
Other Terminology Associated With Chips

• ASIC (Application-Specific Integrated Circuit)
  – Custom design for a specific product
  – Used when higher speed is needed

• SoC (System on Chip)
  – Single IC that contains one or more processors, memories, and I/O device interfaces all interconnected to form a working system
  – Used in many low-end devices
**Levels Of Abstraction**

- Digital systems can be described at various levels of abstraction
- Some examples

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Implemented With</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computer</td>
<td>Circuit board(s)</td>
</tr>
<tr>
<td>Circuit board</td>
<td>Components such as processor and memory</td>
</tr>
<tr>
<td>Processor</td>
<td>VLSI chip</td>
</tr>
<tr>
<td>VLSI chip</td>
<td>Many gates</td>
</tr>
<tr>
<td>Gate</td>
<td>Many transistors</td>
</tr>
<tr>
<td>Transistor</td>
<td>Semiconductor implemented in silicon</td>
</tr>
</tbody>
</table>
Reconfigurable Logic

- Alternative to standard gates
- Allows chip to be configured multiple times
- Can create
  - Various gates
  - Interconnections
- Typical approach: view a gate as an array and inputs as an index
- Most popular form: Field Programmable Gate Array (FPGA)
Summary

- Computer systems are constructed of digital logic circuits
- Fundamental building block is called a gate
- Digital circuit can be described by
  - Boolean algebra (most useful when designing)
  - Truth table (most useful when debugging)
- Clock allows active circuit to perform sequence of operations
- Feedback allows output to control processing
- Practical engineering concerns include
  - Power consumption and heat dissipation
  - Clock skew and synchronization
Questions?
Module III

Data And Program Representation
Digital Logic

• Built on two-valued logic system
• Can be interpreted as
  – Positive voltage and zero volts
  – High and low
  – True and false
  – Asserted and not asserted
• Underneath, it’s all just electrons and wires
Data Representation

• Builds on digital logic
• Applies familiar abstractions
• Interprets sets of Boolean values as
  – Numbers
  – Characters
  – Addresses
• Underneath, it’s all just bits
Bit (Binary Digit)

• Direct representation of digital logic values
• Assigned mathematical interpretation
  – 0 and 1
• Multiple bits used to represent complex data item
• The same underlying hardware can represent bits of an integer or bits of a character
Byte

- Set of multiple bits
- Size depends on computer
- Examples of byte sizes
  - CDC: 6-bit byte
  - BBN: 10-bit byte
  - IBM: 8-bit byte
- On most computers, the byte is the smallest addressable unit of storage
- Note: following modern convention, we will assume an 8-bit byte
Byte Size And Values

- Number of bits per byte determines range of values that can be stored
- Byte of $k$ bits can store $2^k$ values
- Examples
  - Six-bit byte can store 64 possible values
  - Eight-bit byte can store 256 possible values
Binary Representation

• Bits themselves have no intrinsic meaning
• Byte merely stores string of 0’s and 1’s
• Example: all possible combinations of three bits

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

• All meaning is determined by how bits are interpreted
Two Possible Interpretations Of Three Bits

- **Device status**
  - First bit has the value 1 if a disk is connected
  - Second bit has the value 1 if a printer is connected
  - Third bit has the value 1 if a keyboard is connected

- **Integer interpretation**
  - Positional representation uses base 2
  - Values are 0 through 7
  - We must specify order of bits
Binary Weighted Positional Interpretation

\[ 2^5 = 32 \quad 2^4 = 16 \quad 2^3 = 8 \quad 2^2 = 4 \quad 2^1 = 2 \quad 2^0 = 1 \]

- Example
  
  \[ 0 \ 1 \ 0 \ 1 \ 0 \ 1 \]
  
  is interpreted as
  
  \[ 0 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 21 \]

- A set of k bits can represent integers 0 through \( 2^k - 1 \)
# Powers Of Two

<table>
<thead>
<tr>
<th>Power Of 2</th>
<th>Decimal Value</th>
<th>Decimal Digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
<td>4</td>
</tr>
<tr>
<td>11</td>
<td>2048</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>4096</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>16384</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>32768</td>
<td>5</td>
</tr>
<tr>
<td>20</td>
<td>1048576</td>
<td>7</td>
</tr>
<tr>
<td>30</td>
<td>1073741824</td>
<td>10</td>
</tr>
<tr>
<td>32</td>
<td>4294967296</td>
<td>10</td>
</tr>
<tr>
<td>64</td>
<td>18446744073709551616</td>
<td>20</td>
</tr>
</tbody>
</table>
Review: Hexadecimal Notation

• Mathematically, it’s base 16
• Practically, it’s easier to write than binary
• Each hex digit encodes four bits

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0010</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0011</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0100</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0101</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0110</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0111</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Hex</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1000</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>1001</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>1010</td>
<td>10</td>
</tr>
<tr>
<td>B</td>
<td>1011</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>1100</td>
<td>12</td>
</tr>
<tr>
<td>D</td>
<td>1101</td>
<td>13</td>
</tr>
<tr>
<td>E</td>
<td>1110</td>
<td>14</td>
</tr>
<tr>
<td>F</td>
<td>1111</td>
<td>15</td>
</tr>
</tbody>
</table>

• Note: hexadecimal merely represents bits
Hexadecimal Constants

- Supported in some programming languages
- Typical syntax: constant begins with 0x
- Example

\[ 0x\text{DEC90949} \]

\[
\begin{array}{cccccccccccccccc}
D & E & C & 9 & 0 & 9 & 4 & 9 \\
1101 & 1110 & 1001 & 0010 & 0010 & 0100 & 0100 & 0101 \\
\end{array}
\]
Character Sets

- Symbols for upper and lower case letters, digits, and punctuation marks
- Set of symbols defined by computer system
- Each symbol assigned unique bit pattern
- Typically, character set size determined by byte size
- Various character sets have been used in commercial computers
  - EBCDIC
  - ASCII
  - Unicode
EBCDIC

- Extended Binary Coded Decimal Interchange Code
- Defined by IBM
- Popular in 1960s
- Still used on IBM mainframe computers
- Specifies 128 characters
- Example encoding: lower case letter $a$ assigned binary value

10000001
ASCII

- American Standard Code for Information Interchange
- Vendor independent: defined by American National Standards Institute (ANSI)
- Adopted by PC manufacturers
- Specifies 128 characters
- Example encoding: lower case letter *a* assigned binary value

  01100001

- Unprintable characters used for modem control
# Full ASCII Character Set

<table>
<thead>
<tr>
<th>00 nul</th>
<th>01 soh</th>
<th>02 stx</th>
<th>03 etx</th>
<th>04 eot</th>
<th>05 enq</th>
<th>06 ack</th>
<th>07 bel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0B vt</td>
<td>12 dc2</td>
<td>13 dc3</td>
<td>14 dc4</td>
<td>15 nak</td>
<td>16 syn</td>
<td>17 etb</td>
<td></td>
</tr>
<tr>
<td>0C np</td>
<td>0D cr</td>
<td>0E so</td>
<td>0F si</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>09 ht</td>
<td>11 dc1</td>
<td>19 em</td>
<td>21 !</td>
<td>22 &quot;</td>
<td>23 #</td>
<td>24 $</td>
<td>25 %</td>
</tr>
<tr>
<td>10 dle</td>
<td>12 dc2</td>
<td>1A sub</td>
<td>1B esc</td>
<td>1C fs</td>
<td>1D gs</td>
<td>1e rs</td>
<td>1f us</td>
</tr>
<tr>
<td>18 can</td>
<td>19 em</td>
<td>20 sp</td>
<td>21 !</td>
<td>22 &quot;</td>
<td>23 #</td>
<td>24 $</td>
<td>25 %</td>
</tr>
<tr>
<td>28 (</td>
<td>29 )</td>
<td>2A *</td>
<td>2B +</td>
<td>2C ,</td>
<td>2D –</td>
<td>2E .</td>
<td>2F /</td>
</tr>
<tr>
<td>30 O</td>
<td>31 1</td>
<td>32 2</td>
<td>33 3</td>
<td>34 4</td>
<td>35 5</td>
<td>36 6</td>
<td>37 7</td>
</tr>
<tr>
<td>38 8</td>
<td>39 9</td>
<td>3A :</td>
<td>3B ;</td>
<td>3C &lt;</td>
<td>3D =</td>
<td>3E &gt;</td>
<td>3F ?</td>
</tr>
<tr>
<td>40 @</td>
<td>41 A</td>
<td>42 B</td>
<td>43 C</td>
<td>44 D</td>
<td>45 E</td>
<td>46 F</td>
<td>47 G</td>
</tr>
<tr>
<td>48 H</td>
<td>49 I</td>
<td>4A J</td>
<td>4B K</td>
<td>4C L</td>
<td>4D M</td>
<td>4E N</td>
<td>4F O</td>
</tr>
<tr>
<td>50 P</td>
<td>51 Q</td>
<td>52 R</td>
<td>53 S</td>
<td>54 T</td>
<td>55 U</td>
<td>56 V</td>
<td>57 W</td>
</tr>
<tr>
<td>58 X</td>
<td>59 Y</td>
<td>5A Z</td>
<td>5B [</td>
<td>5C \</td>
<td>5D ]</td>
<td>5E ^</td>
<td>5F _</td>
</tr>
<tr>
<td>60 '</td>
<td>61 a</td>
<td>62 b</td>
<td>63 c</td>
<td>64 d</td>
<td>65 e</td>
<td>66 f</td>
<td>67 g</td>
</tr>
<tr>
<td>68 h</td>
<td>69 i</td>
<td>6A j</td>
<td>6B k</td>
<td>6C l</td>
<td>6D m</td>
<td>6E n</td>
<td>6F o</td>
</tr>
<tr>
<td>70 p</td>
<td>71 q</td>
<td>72 r</td>
<td>73 s</td>
<td>74 t</td>
<td>75 u</td>
<td>76 v</td>
<td>77 w</td>
</tr>
<tr>
<td>78 x</td>
<td>79 y</td>
<td>7A z</td>
<td>7B {</td>
<td>7C</td>
<td></td>
<td>7D }</td>
<td>7E ~</td>
</tr>
</tbody>
</table>
Unicode

- Extends ASCII
  - Assigns meaning to values from 128 through 255
  - Character can be 16 bits long
- Advantage: can represent larger set of characters
- Motivation: accommodate languages such as Chinese
Integer Representation In Binary

- Each binary integer represented in $k$ bits
- Computers have used $k = 8, 16, 32, 60,$ and $64$
- Many computers support multiple integer sizes (e.g., $16, 32,$ and $64$ bit integers)
- $2^k$ possible bit combinations exist for $k$ bits
- Positional interpretation produces *unsigned integers*
Unsigned Integers

- Straightforward positional interpretation
- Each successive bit represents next power of 2
- No provision for negative values
- Precision is fixed (size of integers is a constant)
- Arithmetic operations can produce overflow or underflow (result cannot be represented in \( k \) bits)
- Overflow handled with wraparound and carry bit
Illustration Of Overflow

- Values wrap around address space
- Hardware records overflow in separate carry indicator
  - Software must test after arithmetic operation
  - Can be used to raise an exception
Numbering Bits And Bytes

• Need to choose order for
  – Storage in physical memory system
  – Transmission over a data network

• Bit order
  – Handled by hardware
  – Usually hidden from programmer

• Byte order
  – Affects multi-byte data items such as integers
  – Visible and important to programmer
Integer Byte Order

- **Little Endian** places least significant byte of integer in lowest memory location
- **Big Endian** places most significant byte of integer in lowest memory location

Interesting historical variation: Digital Equipment Corporation once used an ordering with 32-bit integers divided into sixteen-bit words in big endian order and bytes within the words in little endian order.
Illustration Of Big And Little Endian Byte Order

(a) Integer 497,171,303 in binary positional representation

<table>
<thead>
<tr>
<th>loc. i</th>
<th>loc. i+1</th>
<th>loc. i+2</th>
<th>loc. i+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>• • •</td>
<td>01100111</td>
<td>00111011</td>
<td>10100010</td>
</tr>
</tbody>
</table>

(b) The integer stored in little endian order

<table>
<thead>
<tr>
<th>loc. i</th>
<th>loc. i+1</th>
<th>loc. i+2</th>
<th>loc. i+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>• • •</td>
<td>00011101</td>
<td>10100010</td>
<td>00111011</td>
</tr>
</tbody>
</table>

(c) The integer stored in big endian order

- Note: difference is especially important when transferring data over the Internet between computers for which the byte ordering differs
Signed Binary Integers

- Signed arithmetic is needed by most programs
- Several representations are possible
- Each has been used in at least one computer
- Some bit patterns are used for negative values (typically half)
- Tradeoff: unsigned representation cannot store negative values, but can store integers that are twice as large as a signed representation
Signed Integer Representations

- Three signed representations have been used
  - Sign magnitude
  - One’s complement
  - Two’s complement
- Each has interesting quirks
Sign Magnitude Representation

- Familiar to humans
- First bit represents sign
- Successive bits represent absolute value of integer
- Interesting quirk: can create negative zero
One’s Complement Representation

- Positive number uses positional representation
- Negative number formed by inverting all bits of positive value
- Example of 4-bit one’s complement
  - 0010 represents 2
  - 1101 represents −2
- Interesting quirk: two representations for zero (all 0’s and all 1’s)
- Note: Internet checksum uses one’s complement
Two’s Complement Representation

- Positive number uses positional representation
- Negative number formed by subtracting 1 from positive value and inverting all bits of result
- Example of 4-bit two’s complement
  - 0 0 1 0 represents 2
  - 1 1 1 0 represents −2
  - High-order bit is set if number is negative
- Interesting quirk: one more negative value than positive values
Implementation Of Unsigned And Two’s Complement

- We consider unsigned and two’s complement together because
  - A single piece of hardware can handle both unsigned and two’s complement integer arithmetic
  - Software can choose an interpretation for each integer
- Example using 4 bits
  - Adding 1 to binary 1001 produces 1010
  - Unsigned interpretation goes from 9 to 10
  - Two’s complement interpretation goes from −7 to −6
### Example Of Signed Representation (4 bit integers)

<table>
<thead>
<tr>
<th>Binary String</th>
<th>Unsigned (positional) Interpretation</th>
<th>Sign Magnitude Interpretation</th>
<th>One’s Complement Interpretation</th>
<th>Two’s Complement Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0011</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>0101</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>0110</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>0111</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>8</td>
<td>−0</td>
<td>−7</td>
<td>−8</td>
</tr>
<tr>
<td>1001</td>
<td>9</td>
<td>−1</td>
<td>−6</td>
<td>−7</td>
</tr>
<tr>
<td>1010</td>
<td>10</td>
<td>−2</td>
<td>−5</td>
<td>−6</td>
</tr>
<tr>
<td>1011</td>
<td>11</td>
<td>−3</td>
<td>−4</td>
<td>−5</td>
</tr>
<tr>
<td>1100</td>
<td>12</td>
<td>−4</td>
<td>−3</td>
<td>−4</td>
</tr>
<tr>
<td>1101</td>
<td>13</td>
<td>−5</td>
<td>−2</td>
<td>−3</td>
</tr>
<tr>
<td>1110</td>
<td>14</td>
<td>−6</td>
<td>−1</td>
<td>−2</td>
</tr>
<tr>
<td>1111</td>
<td>15</td>
<td>−7</td>
<td>−0</td>
<td>−1</td>
</tr>
</tbody>
</table>
Sign Extension

- Needed for unsigned and two’s complement representations
- Used to accommodate multiple sizes of integers
- Extends high-order bit (known as \textit{sign bit})
Explanation Of Sign Extension

- Assume computer
  - Supports 32-bit and 64-bit integers
  - Uses two’s complement representation
- When 32-bit integer assigned to 64-bit integer, correct numeric value requires upper 32 bits to be filled with
  - Zeroes for a positive number
  - Ones for a negative number
- In essence, high-order (sign) bit from the 32-bit integer must be replicated to fill high-order bits of larger integer
Example Of Sign Extension During Assignment

- The 8-bit version of integer \(-3\) is
  
  \[
  \begin{array}{cccccccc}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \ 0 & 1 \\
  \end{array}
  \]

- The 16-bit version of integer \(-3\) is
  
  \[
  \begin{array}{cccccccccccccccc}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \\
  \end{array}
  \]

- During assignment to a larger integer, hardware copies all bits of smaller integer and then replicates the high-order (sign) bit in remaining bits
Summary Of Sign Extension

*Sign extension:* in two’s complement arithmetic, when an integer \( Q \) composed of \( K \) bits is copied to an integer of more than \( K \) bits, the additional high-order bits are set equal to the top bit of \( Q \). Extending the sign bit means the numeric value remains the same.
Sign Extension During Shift

- Right shift of a negative value should produce a negative value
- Example
  - Shifting −4 one bit should produce −2 (divide by 2)
  - Using sixteen-bit representation, −4 is:
    
    \[
    \begin{array}{cccccccccccccccc}
    1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0
    \end{array}
    \]

- After right shift of one bit, value is −2:
  
  \[
  \begin{array}{cccccccccccccccc}
  1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0
  \end{array}
  \]

- Solution: replicate high-order bit during right shift
A Consequence For Programmers

- Most computers use two’s complement hardware, which performs sign extension.
- Same hardware is used for unsigned arithmetic, which means that assigning an unsigned integer to a larger unsigned integer can change the value.
- To prevent errors from occurring, a programmer or a compiler must add code to mask off the extended sign bits.
- Example code:

```c
unsigned int x;
char y;
y = 0xf0;
x = y; /* should be x = y & 0xff; */
```
Binary Coded Decimal

- Pioneered by IBM
- Represents integer as a string of digits
  - Unpacked: one digit per 8-bit byte
  - Packed: one digit per 4-bit nibble
- Uses sign-magnitude representation
- Example of unpacked BCD
  - Integer 123456 is stored as
    
    \[
    \begin{array}{ccccccc}
    0x01 & 0x02 & 0x03 & 0x04 & 0x05 & 0x06
    \end{array}
    \]
  - Integer –123456 is stored as:
    
    \[
    \begin{array}{ccccccccc}
    0x01 & 0x02 & 0x03 & 0x04 & 0x05 & 0x06 & 0x0D
    \end{array}
    \]
Assessment Of Binary Coded Decimal

- Disadvantages:
  - Take more space
  - Hardware is slower than integer or floating point

- Advantages:
  - Gives results humans expect (compare to Excel)
  - Avoids repeating binary value for .01

- Preferred by banks
Floating Point

- Fundamental idea: follow standard scientific representation that specifies a few significant digits and an order of magnitude
- Example: Avogadro’s number

\[ 6.022 \times 10^{23} \]

- Hardware
  - Uses base 2 instead of base 10
  - Allocates fixed-size bit strings for
    * Exponent
    * Mantissa
Optimizing Floating Point

• Mantissa
  – *Normalized* to eliminate leading zeroes
  – No need to store most significant bit because it is always 1
  – Zero is a special case

• Exponent
  – Allows negative as well as positive values
  – Biased to permit rapid magnitude comparison
Example Floating Point Representation: IEEE Standard 754

- Specifies single-precision and double-precision representations
- Widely adopted by computer architects
Special Values In IEEE Floating Point

- Zero
- Positive infinity
- Negative infinity
- Note: infinity values handle cases such as the result of dividing by zero
Range Of Values In IEEE Floating Point

• The single precision range is

\[2^{-126} \text{ to } 2^{127}\]

• The decimal equivalent is approximately

\[10^{-38} \text{ to } 10^{38}\]
The double precision range is enormously larger than single precision

$$2^{-1022} \text{ to } 2^{1023}$$

The decimal equivalent is approximately

$$10^{-308} \text{ to } 10^{308}$$
An Example Floating Point Value

- Consider the decimal value 6.5
- In binary, 6 is 110 and .5 is .1, giving 110.1
- Normalizing gives $1.101 \times 2^2$
- In IEEE floating point
  - The sign bit is zero (for a positive number)
  - The exponent is biased by adding 127, giving 129 (10000001 in binary)
  - The leading 1 of the mantissa is not stored, giving (10100000...0 in binary)
- The resulting binary value is

<table>
<thead>
<tr>
<th>S</th>
<th>exponent (23–30)</th>
<th>mantissa (bits 0–22)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0 0 0 0 0 0 1</td>
<td>1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
Data Aggregates

- Typically arranged in contiguous memory
- Example: struct with three integers

![Diagram of struct with three integers]

- More details later in the course
Summary

- Fundamental value in digital logic is a bit
- Bits grouped into sets to represent
  - Integers
  - Characters
  - Floating point values
- Integers can be represented as
  - Sign magnitude
  - One’s complement
  - Two’s complement
Summary

- One piece of hardware can be used for both
  - Two’s complement arithmetic
  - Unsigned arithmetic
- Bytes of integer can be numbered in
  - Big-endian order
  - Little-endian order
- Organizations such as ANSI and IEEE define standards for data representation
Questions?
Terminology

- The terms *processor* and *computational engine* refer broadly to any mechanism that drives computation.
- Wide variety of sizes and complexity.
- Processor is key element in all computational systems.
Von Neumann Architecture

- Characteristic of most modern processors
- Reference to mathematician John Von Neumann, a pioneer in computer architecture
- Unlike Harvard architecture, there is one memory
- Fundamental concept is a stored program (i.e., a program in the same memory as the data)
- Three basic components interact to form a computational system
  - Processor
  - Memory
  - I/O facilities
Illustration Of Von Neumann Architecture
Processor

- Digital device
- Performs computation involving multiple steps
- Wide variety of capabilities
- Mechanisms available
  - Fixed logic
  - Selectable logic
  - Parameterized logic
  - Programmable logic
Hierarchical Structure And Processors

- Most computer architecture follows a hierarchical approach
- Subparts of a large, central processor are sophisticated enough to meet our definition of processor
- Some engineers use term *computational engine* for subpiece that is less powerful than main processor
Illustration Of Processor Hierarchy
Major Components Of A Conventional Processor

- Controller to coordinate operation (often omitted from architecture diagrams)
- Arithmetic Logic Unit (ALU)
- Local data storage
- Internal interconnections
- External interfaces (I/O buses)
Illustration Of A Conventional Processor
Parts Of A Conventional Processor

- **Controller**
  - Overall responsibility for execution
  - Moves through sequence of steps
  - Coordinates other units
  - Timing-based operation: knows how long each unit requires and schedules steps accordingly

- **Arithmetic Logic Unit**
  - Operates as directed by controller
  - Provides arithmetic and Boolean operations
  - Performs one operation at a time as directed
Parts Of A Conventional Processor (continued)

- Internal interconnections
  - Allow transfer of values among units of the processor
  - Also called *data paths*

- External interface
  - Handles communication between processor and rest of computer system
  - Provides interaction with external memory as well as external I/O devices
Parts Of A Conventional Processor
(continued)

- Local data storage
  - Holds data values for operations
  - Values must be inserted (e.g., loaded from memory) before the operation can be performed
  - Typically implemented with *registers*
Arithmetic Logic Unit

- Main computational engine in conventional processor
- Complex unit that can perform variety of tasks
- Typical ALU operations
  - Arithmetic (integer add, subtract, multiply, divide)
  - Shift (left, right, circular)
  - Boolean (\textit{and, or, not, exclusive or})
Processor Categories And Roles

- Many possible roles for individual processors in
  - Coprocessors
  - Microcontrollers
  - Embedded system processors
  - General-purpose processors
Coprocessor

• Operates in conjunction with and under the control of another processor
• Usually
  – Special-purpose processor
  – Performs a single task
  – Operates at high speed
• Example: floating point accelerator
Microcontroller

- Programmable device
- Dedicated to control of a physical system
- Example: control an automobile engine or grocery store door
- Negative: extremely limited (slow processor and tiny memory)
- Positive: very low power consumption
Example Steps A Microcontroller Performs
(Automatic Door)

do forever {
    wait for the sensor to be tripped;
    turn on power to the door motor;
    wait for a signal that indicates the
door is open;
    wait for the sensor to reset;
debug ten seconds;
    turn off power to the door motor;
}
Embedded System Processor

- Runs sophisticated electronic device
- May be more powerful than microcontroller
- Generally low power consumption
- Example: control DVD player, including commands received from a remote control as well as from the front panel
General-Purpose Processor

- Most powerful type of processor
- Completely programmable
- Full functionality
- Power consumption is secondary consideration
- Example: CPU in a personal computer
Processor Implementation

- Originally: discrete logic
- Later: single circuit board
- Even later: single chip
- Now: usually part of a single chip
Definition Of Programmable Device

• To a software engineer programming means
  – Writing, compiling, and loading code into memory
  – Executing the resulting memory image

• To a hardware engineer a *programmable* device
  – Has a processor separate from the program it runs
  – May have the program burned onto a chip
Fetch-Execute Cycle

- Basis for programmable processors
- Allows processor to move through program steps automatically
- Implemented by processor hardware
- At some level, every programmable processor implements a fetch-execute cycle
Fetch-Execute Algorithm

Repeat forever {

    Fetch: access the next step of the program from the location in which the program has been stored.

    Execute: Perform the step of the program.

}

• Note: we will discuss in more detail later
Program Translation

- Processors require a program to be
  - In memory
  - Represented in binary

- Programmers prefer a program to be
  - Readable by humans
  - In a *High Level Language*

- Solution: allow programmers to write code in a readable high-level language and translate to binary

- Use computer software to perform the translation
Illustration Of Program Translation

source code \[\rightarrow\] preprocessor \[\rightarrow\] preprocessed source code \[\rightarrow\] compiler \[\rightarrow\] assembly code

source code \[\rightarrow\] assembler \[\rightarrow\] relocatable object code \[\rightarrow\] linker \[\rightarrow\] binary object code

object code (functions) in libraries
Clock Rate And Instruction Rate

- Clock rate
  - Rate at which gates are clocked
  - Provides a measure of the underlying hardware speed

- Instruction rate
  - Measures the number of instructions a processor can *execute* per unit time

- On some processors, a given instruction may take more clock cycles than other instructions

- Example: multiplication may take longer than addition
Stopping A Processor

- Processor runs fetch-execute indefinitely
- Software must plan next step
- Two possibilities when last step of computation finishes
  - Smallest embedded systems: code enters a loop testing for a change in input
  - Larger systems: operating system runs and executes an infinite loop
- Note: to reduce power consumption, hardware may provide a way to put processor to sleep until I/O activity occurs (covered later in the course)
Starting A Processor

- Processor hardware includes a reset line that stops the fetch-execute cycle
- For power-down: reset line is asserted
- During power-up, logic holds the reset until the processor and memory are initialized
- Power-up steps known as bootstrap
Summary

- Processor performs a computation involving multiple steps
- Many types of processors
  - Coprocessor
  - Microcontroller
  - Embedded system processor
  - General-purpose processor
- Arithmetic Logic Unit (ALU) performs basic arithmetic and Boolean operations
Summary
(continued)

- Hardware in programmable processor runs fetch-execute cycle
- Until a processor is powered down, fetch-execute must continue
Questions?
Module V

Processor Types
And
Instruction Sets
What Instructions Should A Processor Offer?
What Instructions Should A Processor Offer?

- Minimum set is sufficient, but inconvenient
What Instructions Should A Processor Offer?

- Minimum set is sufficient, but inconvenient
- Extremely large set is convenient, but inefficient
What Instructions Should A Processor Offer?

- Minimum set is sufficient, but inconvenient
- Extremely large set is convenient, but inefficient
- Architect must consider additional factors
  - Physical size of processor chip
  - Expected use
  - Power consumption
What Instructions Should A Processor Offer?

- Minimum set is sufficient, but inconvenient
- Extremely large set is convenient, but inefficient
- Architect must consider additional factors
  - Physical size of processor chip
  - Expected use
  - Power consumption
What Instructions Should A Processor Offer?

- Minimum set is sufficient, but inconvenient
- Extremely large set is convenient, but inefficient
- Architect must consider additional factors
  - Physical size of processor chip
  - Expected use
  - Power consumption
- Tradeoffs mean a variety of designs exist
Instruction Set Architecture

- Idea pioneered by IBM
- Allows multiple, compatible models
- Define
  - Set of instructions
  - Operands and meaning
- Do not define
  - Implementation details
  - Processor speed
A Few Choices

- Functionality: what the instructions provide
  - Arithmetic (integer or floating point)
  - Logic (bit manipulation and testing)
  - Control (branching, function call)
  - Other (graphics, data conversion)

- Format: representation for each instruction

- Semantics: effect when instruction is executed

- An Instruction Set Architecture includes all of the above
Parts Of An Instruction

- *Opcode* specifies operation to be performed
- *Operands* specify data values on which to operate
- *Result location* specifies where result is to be placed
Instruction Format

- Instruction represented as sequence of bits in memory (usually multiples of bytes)
- Typically
  - Opcode at beginning of instruction
  - Operands follow opcode
Instruction Length

- Fixed-length
  - Every instruction is same size
  - Hardware is less complex
  - Hardware can run faster
  - Wasted space: some instructions do not use all the bits

- Variable-length
  - Some instructions shorter than others
  - Allows instructions with no operands, a few operands, or many operands
  - Efficient use of memory (no wasted space)
General-Purpose Registers

- High-speed storage mechanism
- Part of the processor (on chip)
- Each register holds an integer or a pointer
- Numbered from 0 through N–1
- Basic uses
  - Temporary storage during computation
  - Operand for arithmetic operation
- Note: some processors require all operands for an arithmetic operation to come from general-purpose registers
Floating Point Registers

- Usually separate from general-purpose registers
- Each holds one floating-point value
- Floating point registers are operands for floating point arithmetic
Example Of Programming With Registers

- Task
  - Start with variables X and Y in memory
  - Add X and Y and place the result in variable Z (also in memory)
Example Of Programming With Registers

- **Task**
  - Start with variables X and Y in memory
  - Add X and Y and place the result in variable Z (also in memory)

- **Example steps**
  - Load a copy of X into register 1
  - Load a copy of Y into register 2
  - Add the value in register 1 to the value in register 2, and put the result in register 3
  - Store a copy of the value in register 3 in Z

- Note: the above assumes registers 1, 2, and 3 are available
## Terminology

- **Register spilling**
  - Occurs when a register is needed for a computation and all registers contain values
  - General idea
    * Save current contents of register(s) in memory
    * Reload registers(s) from memory when values are needed

- **Register allocation**
  - Refers to choosing which values to keep in registers at a given time
  - Performed by programmer or compiler
Double Precision

- Refers to value that is twice as large as a standard integer
- Most processors do not have dedicated registers for double precision computation
- Approach taken: programmer must use a contiguous pair of registers to hold a double precision value
- Example: multiplication of two 32-bit integers
  - Result can require 64 bits
  - Programmer specifies that result goes into a pair of registers (e.g., 4 and 5)
Register Banks

- Registers partitioned into disjoint sets called *banks*
- Additional hardware detail
- Optimizes performance
- Complicates programming
Register Banks

- Registers partitioned into disjoint sets called *banks*
- Additional hardware detail
- Optimizes performance
- Complicates programming
Typical Register Bank Scheme

- Registers divided into two banks
- ALU instruction that takes two operands must have one operand from each bank
- Programmer must ensure operands are in separate banks
- Note: having two operands from the same bank will cause a run-time error
Why Register Banks Are Used

- Parallel hardware facilities allow simultaneous access of both banks

- Access takes half as long as using a single bank
Consequence For Programmers

• Even trivial programs cause problems
• Example

\[
R \leftarrow X + Y \\
S \leftarrow Z - X \\
T \leftarrow Y + Z
\]

• Operands must be assigned to banks
• No feasible choice for the above
Register Conflicts

• Occur when operands specify same register bank
• May be reported by compiler/assembler
• Programmer must rewrite code or insert extra instruction to copy an operand value to the opposite register bank
• In the previous example
  – Start with Y and Z in the same bank
  – Before adding Y and Z, copy one to another bank
Two Types Of Instruction Sets

- CISC: Complex Instruction Set Computer
- RISC: Reduced Instruction Set Computer
CISC Instruction Set

- Many instructions (often hundreds)
- Given instruction can require arbitrary time to compute
- Example: Intel/AMD (x86/x64) or IBM instruction set
- Typical complex instructions
  - Move graphical item on bitmapped display
  - Copy or clear a region of memory
  - Perform a floating point computation
RISC Instruction Set

- Few instructions (typically 32 or 64)
- Each instruction executes in one clock cycle
- Example: MIPS or ARM instruction set
- Omits complex instructions
  - No floating-point instructions
  - No graphics instructions
- Sequence of instructions needed to perform complex action
Instruction Pipeline

- A major idea in processor design
- Also called *execution pipeline*
- Optimizes performance
- Permits processor to complete more instructions per unit time
- Typically used with RISC instruction set
Basic Steps In A Fetch-Execute Cycle

- Fetch the next instruction
- Decode the instruction and fetch operands from registers
- Perform the arithmetic operation specified by the opcode
- Perform memory read or write, if needed
- Store result back to the registers
Instruction Pipeline Approach

- Build separate hardware block for each step of the fetch-execute cycle
- Arrange hardware to pass an instruction through the sequence of hardware blocks
- Allows step K of one instruction to execute while step K−1 of next instruction executes
- Result is an *execution pipeline*
Illustration Of An Execution Pipeline

- Example pipeline has five stages
- All stages operate at the same time
- Instruction passes through like a factory assembly line
### Illustration Of Instructions In A Pipeline

<table>
<thead>
<tr>
<th>clock</th>
<th>stage 1</th>
<th>stage 2</th>
<th>stage 3</th>
<th>stage 4</th>
<th>stage 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>inst. 1</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>inst. 2</td>
<td>inst. 1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>inst. 3</td>
<td>inst. 2</td>
<td>inst. 1</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>inst. 4</td>
<td>inst. 3</td>
<td>inst. 2</td>
<td>inst. 1</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>inst. 5</td>
<td>inst. 4</td>
<td>inst. 3</td>
<td>inst. 2</td>
<td>inst. 1</td>
</tr>
<tr>
<td>6</td>
<td>inst. 6</td>
<td>inst. 5</td>
<td>inst. 4</td>
<td>inst. 3</td>
<td>inst. 2</td>
</tr>
<tr>
<td>7</td>
<td>inst. 7</td>
<td>inst. 6</td>
<td>inst. 5</td>
<td>inst. 4</td>
<td>inst. 3</td>
</tr>
<tr>
<td>8</td>
<td>inst. 8</td>
<td>inst. 7</td>
<td>inst. 6</td>
<td>inst. 5</td>
<td>inst. 4</td>
</tr>
</tbody>
</table>
Pipeline Speed

- All stages operate in parallel
- Given stage can start to process a new instruction as soon as current instruction finishes
- Effect: N-stage pipeline can operate on N instructions simultaneously, producing speedup
- Result
  - One instruction completes every time pipeline moves
  - For RISC processor, one instruction completes on every clock cycle
- Comparison: without a pipeline, each instruction would take five clock cycles
Significance Of A Pipeline To A Programmer

- Pipeline is *transparent* to programmers (i.e., is automatic)
- Execution speed
  - Is never worse than a processor without a pipeline
  - May be $K$ times faster than processor without a pipeline
- Pipeline *stalls* (i.e., pauses) if item is not available when a stage needs the item
- Programmer who does not understand pipeline can produce code that stalls frequently
Example Of Instructions That Cause A Stall

- Consider code that
  - Performs addition and subtraction operations
  - Uses registers $A$ through $E$ for operands and results
- Example instruction sequence
  
  Instruction K: $C \leftarrow \text{add } A \ B$
  Instruction K+1: $D \leftarrow \text{subtract } E \ C$

- Instruction K+1 must wait for operand $C$ to be computed
- Result is a *stall*
### Effect Of Stall On Pipeline

<table>
<thead>
<tr>
<th>clock</th>
<th>stage 1 fetch instruction</th>
<th>stage 2 fetch operands</th>
<th>stage 3 ALU operation</th>
<th>stage 4 access memory</th>
<th>stage 5 write results</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>(inst. K+2)</td>
<td>inst. K+1</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>7</td>
<td>inst. K+3</td>
<td>inst. K+2</td>
<td>inst. K+1</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

- We say a *bubble* passes through pipeline


Actions That Cause A Pipeline Stall

- Access external storage (i.e., memory reference)
- Invoke a coprocessor (i.e., I/O)
- Branch to a new location
- Call a subroutine
Achieving Maximum Speed

- Program must be written to accommodate instruction pipeline
- To minimize stalls
  - Avoid introducing unnecessary branches
  - Delay references to result register(s)
- A contradiction
  - Good software engineering practice divides a large program into smaller functions
  - A function call stalls the pipelining
Example Of Avoiding Stalls

C ← add A B C ← add A B
D ← subtract E C F ← add G H
F ← add G H M ← add K L
J ← subtract I F D ← subtract E C
M ← add K L J ← subtract I F
P ← subtract M N P ← subtract M N

(a) (b)

• Stalls eliminated by rearranging (a) to (b)
• Compilers for RISC processors usually optimize code to avoid stalls
A Note About Pipelines

- We can think of pipelining as an automatic optimization
  - Hardware speeds up processing if possible
  - If speedup is not possible, hardware is still correct

- Consequence: code that is not optimized will work correctly, but may run slower than necessary
Forwarding

- Hardware optimization to avoid a stall
- Allows ALU to reference result in next instruction
- Example

  Instruction K: \[ C \leftarrow \text{add} \ A \ B \]
  Instruction K+1: \[ D \leftarrow \text{subtract} \ E \ C \]

- Forwarding hardware
  - Passes result of \textit{add} operation directly to ALU without waiting to store it in a register
  - Ensures the value arrives by the time subtract instruction reaches the pipeline stage for execution
No-Op Instruction

- Often included in RISC instruction sets
- May seem unnecessary
- Has no effect on
  - Registers
  - Memory
  - Program counter
  - Computation
- Purpose: can be inserted to avoid instruction stalls
Use Of No-Op

• Example

Instruction K: \( C \leftarrow \text{add} \ A \ B \)

Instruction K+1: no-op

Instruction K+2: \( D \leftarrow \text{subtract} \ E \ C \)

• If forwarding is available, no-op allows time for result from register \( C \) to be fetched for \textit{subtract} operation

• Compilers insert no-op instructions to optimize performance
Types Of Opcodes

- Operations usually classified into groups
- An example categorization
  - Arithmetic instructions (integer arithmetic)
  - Logical instructions (also called Boolean)
  - Data access and transfer instructions
  - Conditional and unconditional branch instructions
  - Floating point instructions
  - Processor control instructions
  - Graphics instructions
Program Counter

- Hardware register
- Used during fetch-execute cycle
- Gives address of next instruction to execute
- Also known as *instruction pointer* or *instruction counter*
Fetch-Execute Algorithm Details

Assign the program counter an initial program address.

Repeat forever {

Fetch: access the next step of the program from the location given by the program counter.

Set an internal address register, A, to the address beyond the instruction that was just fetched.

Execute: Perform the step of the program.

Copy the contents of address register A to the program counter.

}
Branches And Fetch Execute

- Absolute branch
  - Typically named \(jump\)
  - Operand is an address
  - Assigns operand value to internal register \(A\)

- Relative branch
  - Typically named \(br\)
  - Operand is a signed value
  - Adds operand to internal register \(A\)
Subroutine Call

- Jump to subroutine (*jsr* instruction)
  - Similar to a *jump*
  - Saves value of internal register A
  - Replaces A with operand address

- Return from subroutine (*ret* instruction)
  - Retrieves value saved during *jsr*
  - Replaces A with saved value
Passing Arguments

- Multiple methods are used
- Choice depends on language/compiler as well as hardware
- Examples
  - Store arguments in memory
  - Store arguments in special-purpose hardware registers
  - Store arguments in general-purpose registers
- Many techniques also used to return result from function
Register Window

- Hardware optimization for argument passing
- Processor contains many general-purpose registers
- Only a small subset of registers visible at any time
- Caller places arguments in reserved registers
- During procedure call, register window moves to hide old registers and expose new registers
Illustration Of Register Window

- (a) registers before calling a subroutine
- (b) registers when the subroutine runs
An Example Instruction Set

- Known as *MIPS* instruction set
- Early RISC design
- Minimalistic
- Only 32 instructions
# MIPS Instruction Set (Part 1)

## Instruction Set

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></td>
<td></td>
</tr>
<tr>
<td>add</td>
<td>integer addition</td>
</tr>
<tr>
<td>subtract</td>
<td>integer subtraction</td>
</tr>
<tr>
<td>add immediate</td>
<td>integer addition (register + constant)</td>
</tr>
<tr>
<td>add unsigned</td>
<td>unsigned integer addition</td>
</tr>
<tr>
<td>subtract unsigned</td>
<td>unsigned integer subtraction</td>
</tr>
<tr>
<td>add immediate unsigned</td>
<td>unsigned addition with a constant</td>
</tr>
<tr>
<td>move from coprocessor</td>
<td>access coprocessor register</td>
</tr>
<tr>
<td>multiply</td>
<td>integer multiplication</td>
</tr>
<tr>
<td>multiply unsigned</td>
<td>unsigned integer multiplication</td>
</tr>
<tr>
<td>divide</td>
<td>integer division</td>
</tr>
<tr>
<td>divide unsigned</td>
<td>unsigned integer division</td>
</tr>
<tr>
<td>move from Hi</td>
<td>access high-order register</td>
</tr>
<tr>
<td>move from Lo</td>
<td>access low-order register</td>
</tr>
<tr>
<td><strong>Logical (Boolean)</strong></td>
<td></td>
</tr>
<tr>
<td>and</td>
<td>logical and (two registers)</td>
</tr>
<tr>
<td>or</td>
<td>logical or (two registers)</td>
</tr>
<tr>
<td>and immediate</td>
<td>and of register and constant</td>
</tr>
<tr>
<td>or immediate</td>
<td>or of register and constant</td>
</tr>
<tr>
<td>shift left logical</td>
<td>Shift register left N bits</td>
</tr>
<tr>
<td>shift right logical</td>
<td>Shift register right N bits</td>
</tr>
</tbody>
</table>
# MIPS Instruction Set (Part 2)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Transfer</strong></td>
<td></td>
</tr>
<tr>
<td>load word</td>
<td>load register from memory</td>
</tr>
<tr>
<td>store word</td>
<td>store register into memory</td>
</tr>
<tr>
<td>load upper immediate</td>
<td>place constant in upper sixteen bits of register</td>
</tr>
<tr>
<td>move from coproc. register</td>
<td>obtain a value from a coprocessor</td>
</tr>
<tr>
<td><strong>Conditional Branch</strong></td>
<td></td>
</tr>
<tr>
<td>branch equal</td>
<td>branch if two registers equal</td>
</tr>
<tr>
<td>branch not equal</td>
<td>branch if two registers unequal</td>
</tr>
<tr>
<td>set on less than</td>
<td>compare two registers</td>
</tr>
<tr>
<td>set less than immediate</td>
<td>compare register and constant</td>
</tr>
<tr>
<td>set less than unsigned</td>
<td>compare unsigned registers</td>
</tr>
<tr>
<td>set less than immediate</td>
<td>compare unsigned register and constant</td>
</tr>
<tr>
<td><strong>Unconditional Branch</strong></td>
<td></td>
</tr>
<tr>
<td>jump</td>
<td>go to target address</td>
</tr>
<tr>
<td>jump register</td>
<td>go to address in register</td>
</tr>
<tr>
<td>jump and link</td>
<td>procedure call</td>
</tr>
</tbody>
</table>
## MIPS Floating Point Instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arithmetic</strong></td>
<td></td>
</tr>
<tr>
<td>FP add</td>
<td>floating point addition</td>
</tr>
<tr>
<td>FP subtract</td>
<td>floating point subtraction</td>
</tr>
<tr>
<td>FP multiply</td>
<td>floating point multiplication</td>
</tr>
<tr>
<td>FP divide</td>
<td>floating point division</td>
</tr>
<tr>
<td>FP add double</td>
<td>double-precision addition</td>
</tr>
<tr>
<td>FP subtract double</td>
<td>double-precision subtraction</td>
</tr>
<tr>
<td>FP multiply double</td>
<td>double-precision multiplication</td>
</tr>
<tr>
<td>FP divide double</td>
<td>double-precision division</td>
</tr>
<tr>
<td><strong>Data Transfer</strong></td>
<td></td>
</tr>
<tr>
<td>load word coprocessor</td>
<td>load value into FP register</td>
</tr>
<tr>
<td>store word coprocessor</td>
<td>store FP register to memory</td>
</tr>
<tr>
<td><strong>Conditional Branch</strong></td>
<td></td>
</tr>
<tr>
<td>branch FP true</td>
<td>branch if FP condition is true</td>
</tr>
<tr>
<td>branch FP false</td>
<td>branch if FP condition is false</td>
</tr>
<tr>
<td>FP compare single</td>
<td>compare two FP registers</td>
</tr>
<tr>
<td>FP compare double</td>
<td>compare two double precision values</td>
</tr>
</tbody>
</table>
Aesthetic Aspects Of Instruction Sets

- Elegance
  - Balanced
  - No frivolous or useless instructions
- Orthogonality
  - No unnecessary duplication
  - No overlap among instructions
- Ease of programming
  - Instructions match programmer’s intuition
  - Instructions are free from arbitrary restrictions
Principle Of Orthogonality

- Specifies that each instruction should perform a unique task
- No instruction duplicates or overlaps another
Condition Codes

- Extra hardware bits (not part of general-purpose registers)
- Set by ALU each time an instruction produces a result
- Used to indicate
  - Overflow
  - Underflow
  - Whether result is positive, negative, or zero
  - Other exceptions
- Tested in *conditional branch* instruction
Example Of Condition Code

cmp r4, r5  # compare regs. 4 & 5, and set condition code
be   lab1  # branch to lab1 if cond. code specifies equal
mov r3, 0  # place a zero in register 3

lab1:...program continues at this point

- Above code places a zero in register 3 if register 4 is not equal to register 5
Questions?
Module VI

DATA PATHS

Interconnection Of Processor Components
And Instruction Execution
Review Of Digital Hardware

- We are proceeding from basics to more complexity

- Covered so far
  - Interconnecting transistors to form gates
  - Interconnecting gates to form combinatorial circuits
  - Adding a clock to execute a sequence of steps
  - Using feedback to control processing
The Next Step

- Build a *programmable* processor
- We will assume a program already resides in memory
- The processor must repeatedly
  - Fetch the next instruction from memory
  - Perform the instruction
Questions We Will Consider
Questions We Will Consider

• What are the major building blocks needed to create a processor?
Questions We Will Consider

- What are the major building blocks needed to create a processor?
- How are the building blocks arranged?
Questions We Will Consider

- What are the major building blocks needed to create a processor?
- How are the building blocks arranged?
- What happens when an instruction is executed?
Let’s Build A Computer!
Let’s Build A Computer!

- Of course, we’ll build a *very* simplified computer
Let’s Build A Computer!

- Of course, we’ll build a very simplified computer
- Thirty-two bit processor
- Sixteen registers used for arithmetic
- Harvard architecture: separate memories for
  - Instruction store
  - Data store
- Memories are byte-addressable (realistic)
- Instruction memory is preloaded with a program
- Consider the hardware needed to execute four basic instructions: load, store, add, jump
Instructions

- Load: copies a value from memory to a register
- Store: copies a value from a register to memory
- Add: adds the values in two registers and places the result in a register
- Jump: forces the processor to a new location in the program instead of the next sequential location
Instructions In Assembly Language

• A programmer writes instructions with an operation followed by operands
• Commas separate operands
• Example
  
  \texttt{load \ operand1, operand2}

• The program must be translated to binary before being loaded into our computer
Operands For Our Example Instructions

- Illustrate a couple of basic types
  - Register access
  - Memory access
- Other operand types will be covered later
Operand Examples
Operand Examples

• Example 1: add the contents of register 4 to the contents of register 11, and place the result in register 9

```
add       reg9, reg11, reg4
```
Operand Examples

- Example 1: add the contents of register 4 to the contents of register 11, and place the result in register 9

  \[ \text{add reg9, reg11, reg4} \]

- Example 2: add an offset of 20 to the contents of register 12, use the result as a memory address, and load register 1 with the value from memory

  \[ \text{load reg1, 20(reg12)} \]
Operand Examples

- Example 1: add the contents of register 4 to the contents of register 11, and place the result in register 9
  
  \[
  \text{add} \quad \text{reg9, reg11, reg4}
  \]

- Example 2: add an offset of 20 to the contents of register 12, use the result as a memory address, and load register 1 with the value from memory
  
  \[
  \text{load} \quad \text{reg1, 20(reg12)}
  \]

- Example 3: add an offset of 64 to the contents of register 7, treat the result as the address of code in memory, and branch to the address
  
  \[
  \text{jump} \quad 64(\text{reg7})
  \]

- Note: many processors allow an operand to specify an offset plus the contents of a register
Binary format chosen to simplify hardware

- Field *reg A* is a register used in a memory address
- Field *reg B* holds a value to be added
- Field *dst reg* specifies a register to receive the result
Notes About Instructions

- Only the `add` instruction uses all three register fields.
- If an instruction has an operand of the form `offset(register)` the register will always be in field `reg A`.
- The offset is limited to 15 bits.
An Example Instruction In Memory

- Suppose $rX$ denotes register $X$, and consider an *add* instruction

  \[
  \text{add } r4, r2, r3
  \]

  (a)

<table>
<thead>
<tr>
<th>operation</th>
<th>reg A</th>
<th>reg B</th>
<th>dst reg</th>
<th>offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 1</td>
<td>0 0 1 0</td>
<td>0 0 1 1</td>
<td>0 1 0 0</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

  (b)

- (a) shows the instruction in assembly language
- (b) shows the instruction in binary as it is stored in memory
Data And Instruction Memories

- Instruction memory (read only)
  - Input: 32-bit byte \textit{address}
  - Output: 32-bit data value (the four bytes starting at the specified address)

- Data memory (RAM — can be read or written)
  - Inputs
    * 32-bit byte \textit{address}
    * 32-bit \textit{data} (only used during write)
    * 1-bit \textit{fetch/store signal}
  - Output 32-bit data value (if the signal is \textit{fetch})
Illustration Of The Two Memories

- Block diagram hides multiple gates
- Note: we assume instruction memory is preloaded with a program (i.e., it is read-only memory)
Moving To The Next Instruction

• Facts
  – Our instruction memory is byte-addressable
  – Each instruction is 32-bits long (4 bytes)
  – The program counter must be incremented by 4 to move to the next instruction

• Hardware needed
  – Gates to store a program counter
  – Adder to compute the increment
  – Clock to control when updates occur
Arrows indicate *data path* of multiple, parallel wires

In our example, each data path is thirty-two bits wide
Fetching An Instruction

• Recall
  – Instructions in separate *instruction memory*
  – Instruction memory takes a 32-bit address as input and produces a 32-bit output value equal to the contents of the specified address
The memory output changes whenever the input changes (i.e., whenever a new address is supplied)
Decoding An Instruction

- Must break out fields
- Instruction format chosen to make decoding efficient
- Decoder hardware separates fields of an instruction
- Each field sent along separate data path
- Our example design is trivial: the decoder merely consists of a 32-bit register with output wires grouped into smaller data paths
Illustration Of An Instruction Decoder

- Note: data paths emerging from the instruction decoder are not thirty-two bits wide
Registers

- The registers are implemented as a single hardware unit
- Think of each register as holding a 32-bit value
- The register unit has four inputs and two outputs
- Input → output
  - First register number → contents of register
  - Second register number → contents of register
  - Third register number plus data → data is stored in the specified register
• Note: there are two inputs and two outputs because we assume the register unit has hardware that can perform two lookups simultaneously
Control And Coordination

• A clock is used to synchronize all units

• Additional *controller* hardware coordinates overall data movement
  – Connects to each hardware unit
  – Specifies when to transfer data

• *Control connections* between controller and individual units are not shown because diagram illustrates *data paths*

• Example: control lines (not shown) signal the register unit when to perform a *fetch* operation or a *store* operation
Arithmetic Operations And Multiplexing

- Although example only has one arithmetic operation, *add*, additional arithmetic instructions can be added easily (e.g., *shift* and *subtract*)

- Use an Arithmetic Logic Unit (ALU)

- Problem: inputs to ALU can be
  - Two registers
  - Register and offset

- Solution: use a multiplexor to choose
Multiplexor

- Small hardware unit
- Fits into data path (i.e., handles parallel data)
- Take two inputs and has one output
- Each input or output is 32-bits wide
- At any time
  - Multiplexor forwards 32 bits from one input path to the output
  - Selection is determined by a controller (not shown)
On some instructions, ALU adds register and offset; on *add* instruction, ALU adds two registers.
Instructions That Access Data Memory

- Additional hardware unit implements data memory
- Two basic operations: fetch and store
- Fetch
  - Place an address on the address input
  - Arrange for controller to signal fetch
  - Read a value from the data output
- Store
  - Place a value on the address input
  - Place a data value on data input
  - Arrange for controller to signal store
A controller (not shown) uses the operation to set the multiplexers
Individual Instruction Execution

- Previous diagram shows all physical data paths
- When an instruction is executed, controller selects which data paths are used
  - Memory and register units honor *fetch* or *store*
  - Each multiplexor selects one input
  - Other data paths are ignored
- Examples follow
Data Paths Used During A Load Instruction
Data Paths Used During A Store Instruction

- 32-bit pgm. ctr.
- 32-bit adder
- Instruction memory
- 4
- Data in
- instr. decoder
- Reg A
- Reg B
- Data out
- Offset
- Operation
- ALU
- Data in
- Addr. in
- Data memory
- Addr. out
- Data out
Data Paths Used During An Add Instruction
Data Paths Used During A Jump Instruction

- 32-bit program counter
- 32-bit adder
- Instruction memory
- Data in
- Instruction decoder
- Register unit
- ALU
- Data memory
- Adder
- Data out
- Offset
- Operation
- Data in
- Address in
- Data out
Summary

- The term *data path* describes interconnections among pieces of a processor.
- Each data path contains \( N \) parallel wires.
- Building blocks of a processor include:
  - Program counter
  - Decoder
  - Register unit
  - Instruction and data memories
  - ALU
Summary (continued)

- A multiplexor passes one of its input data paths to the output data path
- Control signals determine which input a multiplexor selects at a given time
- By controlling multiplexors, processor hardware chooses which data paths are active for a given instruction
Questions?
Module VII

Operands, Operand Addressing
And
Instruction Representation
How Many Operands On Each Instruction?

- Given architecture usually has the same number for most instructions
- Four basic architectural types
  - 0-address
  - 1-address
  - 2-address
  - 3-address
0-Address Architecture

- Stack-based architecture
- No explicit operands in the instruction

Program
  - Pushes operands onto stack in memory
  - Executes instruction

Instruction execution
  - Removes top $N$ items from stack
  - Leaves result on top of stack
Illustration Of 0-Address Instructions

- Example: increment variable \( X \) in memory by 7

```plaintext
push X
push 7
add
pop X
```

- *Push* instruction places a copy of variable \( X \) on the stack
- *Add* instruction removes two arguments from stack and leaves result on stack
- *Pop* instruction removes item on the top of the stack, and places the item in variable \( X \)
1-Address Architecture

- Analogous to a calculator
- One explicit operand per instruction
- Processor has special register known as an *accumulator*
  - Holds second argument for each instruction
  - Used to store result of instruction
Illustration Of 1-Address Instructions

• Example: increment variable $X$ in memory by 7

  load $X$
  add 7
  store $X$

• *Load* places copy of variable $X$ in the accumulator
• *Add* increases value in accumulator
• *Store* copies accumulator value into variable $X$ in memory
2-Address Architecture

- Two explicit operands per instruction
- Result overwrites one of the operands
- Operands known as source and destination
- Works well for instructions such as memory copy
Illustration Of 2-Address Instructions

- Example: increment variable $X$ in memory by 7

\[ \text{add 7, } X \]

- Computes $X + 7$ and places the result in variable $X$
3-Address Architecture

- Three explicit operands per instruction
- Operands specify two values and a location for the result
- Operands are often called
  - Source
  - Destination (for instructions that only need two operands)
  - Result (if all three operands are needed)
Illustration Of 3-Address Instructions

- Example: add variable $Y$ to variable $X$ and place result in variable $Z$

    add $X$, $Y$, $Z$
Source And Destination Operands

• Source operand can specify
  – A signed constant
  – An unsigned constant
  – The contents of a register
  – A value in memory

• Destination operand can specify
  – A single register
  – A pair of contiguous registers
  – A memory location
Operand Types

- Question: how does a processor know whether an operand specifies a constant, a register or a memory address?
- Answer: each operand has a *type* that tells the processor how to interpret the operand
Immediate Values And Memory References

- An operand that gives a signed or unsigned constant is known as an immediate operand.
- Of course, constants could be placed in memory.
- Question: why have immediate operands?
- Answer: memory references are expensive compared to accessing an immediate value.
Von Neumann Bottleneck

• General engineering principle
• Refers to the cost of memory references
• Often stated as follows

*On a computer that follows the Von Neumann architecture, the time spent performing memory accesses can limit the overall performance*

• Motivates using immediate operands or placing operands in registers
Two Styles Of Operand Encoding

- Implicit type encoding
  - Opcode specifies the type of each operand
  - Many opcodes needed
  - Example opcode is `add_signed_immediate_to_register`

- Explicit type encoding
  - Each operand has extra bits that specify a type
  - Fewer opcodes required
  - Example: opcode is `add`, and the two operands specify the types `signed_immediate` and `register`
## Examples Of Implicit Encoding

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add register</td>
<td>R1 R2</td>
<td>R1 ← R1 + R2</td>
</tr>
<tr>
<td>Add immediate signed</td>
<td>R1 I</td>
<td>R1 ← R1 + I</td>
</tr>
<tr>
<td>Add immediate unsigned</td>
<td>R1 UI</td>
<td>R1 ← R1 + UI</td>
</tr>
<tr>
<td>Add memory</td>
<td>R1 M</td>
<td>R1 ← R1 + memory[M]</td>
</tr>
</tbody>
</table>
Examples Of Explicit Encoding

- *Add* operation with registers 1 and 2 as operands

<table>
<thead>
<tr>
<th>opcode</th>
<th>operand 1</th>
<th>operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>register 1</td>
<td>register 2</td>
</tr>
</tbody>
</table>

- *Add* operation with register 1 and signed immediate value of −93 as operands

<table>
<thead>
<tr>
<th>opcode</th>
<th>operand 1</th>
<th>operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>register 1</td>
<td>signed integer −93</td>
</tr>
</tbody>
</table>
Operands That Combine Multiple Types

- Operand contains multiple items
- Processor computes operand value from individual items
- Typical computation: sum
- Example
  - A register-offset operand specifies a register and an immediate value
  - Processor adds immediate value to contents of register and uses result as operand
# Illustration Of Register-Offset

<table>
<thead>
<tr>
<th>opcode</th>
<th>operand 1</th>
<th>operand 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>register-offset</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>register-offset</td>
<td>4</td>
</tr>
</tbody>
</table>

- First operand consists of value in register 2 minus 17
- Second operand consists of value in register 4 plus 76
Operand Tradeoffs

- No single style of operand optimal for all purposes
- Tradeoffs among
  - Ease of programming
  - Fewer instructions
  - Smaller instructions
  - Larger range of immediate values
  - Faster operand fetch and decode
  - Decreased hardware size
Operands In Memory And Indirect Reference

- Operand can specify
  - Value in memory (*memory reference*)
  - Location in memory that contains the address of the operand (*indirect reference*)

- Note: accessing memory is relatively expensive
Types Of Indirection

- Indirection through a register
  - Operand specifies register number, $R$
  - Obtain $A$, the current value from register $R$
  - Interpret $A$ as a memory address, and fetch the operand from memory location $A$

- Indirection through a memory location
  - Operand specifies memory address, $A$
  - Obtain $M$, the value in memory location $A$
  - Interpret $M$ as a memory address, and fetch the operand from memory location $M$
Illustration Of Operand Addressing Modes

1 Immediate value (in the instruction)
2 Direct register reference
3 Direct memory reference
4 Indirect through a register
5 Indirect memory reference
Summary

- Architect chooses the number and types of operands for each instruction
- Possibilities include
  - Immediate (constant value)
  - Contents of register
  - Value in memory
  - Indirect reference to memory
Summary (continued)

- Type of operand can be encoded
  - Implicitly (opcode determines types of operands)
  - Explicitly (extra bits in each operand specify the type)
- Many variations exist; each represents a tradeoff
Questions?
Module VIII

CPUs:
Microcode, Protection, And Processor Modes
Evolution Of Computers

• Early systems
  – Single *Central Processing Unit (CPU)* controlled entire computer
  – Responsible for all I/O as well as computation

• Modern computer
  – Decentralized architecture
  – CPU chip may contain multiple cores
  – Each I/O device (e.g., a disk) contains processor
  – CPU performs computation and coordinates other processors
CPU Complexity

- CPU designed for wide variety of control and processing tasks
- The most complex CPUs have many special-purpose hardware subunits
- Example: Intel makes a multicore chip that contains 2.5 billion transistors
CPU Characteristics

- Completely general
- Can perform control functions as well as basic computation
- Offers multiple levels of protection and privilege
- Provides mechanism for hardware priorities
- Handles large volumes of data
- Uses parallelism to achieve high speed
Modes Of Execution

- CPU hardware has several possible modes
- At any time, CPU operates in one mode
- Mode dictates
  - Instructions that are valid
  - Regions of memory that can be accessed
  - Amount of privilege
  - Backward compatibility with earlier models
- CPU behavior can vary widely among modes
How To Think About Modes

- Imagine multiple hardware units inside the CPU
- Mode selects which hardware is used at a given current time
- Two modes may have different
  - Word sizes
  - Numbers of registers
  - Instruction sets
How Can Mode Change?

- **Automatic**
  - Initiated by hardware (e.g., when device needs service)
  - Prior to change, software (OS) must specify which code to run when the change occurs

- **Manual**
  - Application makes explicit request
  - Typically occurs when application calls an operating system function
Privilege And Protection
Privilege Level

• Determines which resources a program can use
• Usually coupled to mode
• Basic scheme: two levels
  – *User mode* for applications
  – *Kernel mode* for operating system
• Advanced scheme: multiple levels
• In almost any architecture, the OS can execute additional instructions that an application cannot
Illustration Of Two-Level Privilege Scheme

- Applications run with low privilege
- OS runs with high privilege
Microcode
Microcoded Instructions

- Hardware technique used with CISC processors
- Employs two levels of processor hardware
  - Microprocessor (microcontroller) provides basic operations
  - Macro instruction set built on micro instructions
  - Macro instructions and micro instructions may differ completely
- Key concept: it is easier to construct complex processors by writing programs than by building hardware from scratch
A CISC CPU Using Microcoded Instructions

- **macro instruction set** (implemented with microcode)
- **micro instruction set** (implemented with digital logic)

**CPU**

visible to programmer

hidden (internal)

Microcontroller
Integer And Register Sizes

- Size used by micro instructions can differ from size used by macro instructions
- Example
  - Micro instructions only offer 16-bit arithmetic
  - Macro instructions provide 32-bit arithmetic
Microcoded Arithmetic

• Assumptions for the example
  – Macro registers
    * Each 32 bits wide
    * Named R0, R1, ...
  – Micro registers
    * Each 16 bits wide
    * Named r0, r1, ...
• Devise microcode to add values from R5 and R6
Example Microcode

add32: /* Compute R5 + R6 */
move low-order 16 bits from R5 into r2
move low-order 16 bits from R6 into r3
add r2 and r3, placing result in r1
save value of the carry indicator
move high-order 16 bits from R5 into r2
move high-order 16 bits from R6 into r3
add r2 and r3, placing result in r0
copy the value in r0 to r2
add r2 and the carry bit, placing the result in r0
check for overflow and set the condition code
move the thirty-two bit result from r0 and r1
to the desired destination
Microcode Variations

• Restricted or full scope
  – Special-purpose instructions only (e.g., complex instructions or extensions to normal instruction set)
  – All instructions

• Partial or complete use
  – Entire fetch-execute cycle
  – Instruction fetch and decode
  – Opcode processing
  – Operand decode and fetch
Why Use Microcode Instead Of Circuits?

- Higher level of abstraction
- Easier to build and less error prone than building with logic gates
- Easier to change
  - Easy upgrade to next version of chip
  - Can allow field upgrade
Disadvantages Of Microcode

- More overhead
- Macro instruction performance depends on micro instruction set
- Microprocessor hardware must run at extremely high clock rate to accommodate multiple micro instructions per macro instruction
Visibility To Programmers

• Fixed (immutable) microcode
  – Approach used by most CPUs
  – Microcode only visible to CPU designer

• Alterable microcode
  – Microcode loaded dynamically
  – May be restricted to extensions (creating new macro instructions)
  – User software written to use new instructions
  – Known as a reconfigurable CPU

• If you could change microcode, what would you change?
In Practice

- Writing microcode is tedious and time-consuming compared to application programming
- Results are difficult to test
- Performance of microcode can be much worse than performance of dedicated hardware
- Result: reconfigurable CPUs have not enjoyed much success
- More recent technology for reconfigurable processors: FPGA
Two Fundamental Types Of Microcode

- What programming paradigm is used for microcode?
- Two fundamental types
  - Vertical
  - Horizontal
Vertical Microcode

- Vertical microcode similar to conventional assembly language
- Microprocessor uses fetch-execute and executes one instruction at a time
- Micro instructions can access
  - An ALU
  - The macro general-purpose registers
  - Memory
  - I/O buses
Example Of Vertical Microcode

- Macro instruction set is CISC
- Microprocessor is fast RISC processor
- Programmer writes microcode for each macro instruction
- Hardware decodes macro instruction and invokes correct microcode routine
Advantages And Disadvantages
Of Vertical Microcode

- Easy to read
- Programmers are comfortable using it
- Unattractive to hardware designers because higher clock rates needed
- Generally has low performance (many micro instructions needed for each macro instruction)
Horizontal Microcode

- Alternative to vertical microcode
- Exploits parallelism in underlying hardware
- Controls functional units and data movement
- Extremely difficult to program
- Paradigm
  - Each micro instruction controls a set of hardware units
  - An instruction specifies which hardware units to operate and how data is transferred among them
Horizontal Microcode Example

- Consider the internal structure of a CPU
- Data can only move along specific paths between functional units
- Example
# Example Hardware Control Commands

<table>
<thead>
<tr>
<th>Unit</th>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALU</td>
<td>0 0 0</td>
<td>No operation</td>
</tr>
<tr>
<td></td>
<td>0 0 1</td>
<td>Add</td>
</tr>
<tr>
<td></td>
<td>0 1 0</td>
<td>Subtract</td>
</tr>
<tr>
<td></td>
<td>0 1 1</td>
<td>Multiply</td>
</tr>
<tr>
<td></td>
<td>1 0 0</td>
<td>Divide</td>
</tr>
<tr>
<td></td>
<td>1 0 1</td>
<td>Left shift</td>
</tr>
<tr>
<td></td>
<td>1 1 0</td>
<td>Right shift</td>
</tr>
<tr>
<td></td>
<td>1 1 1</td>
<td>Continue previous operation</td>
</tr>
<tr>
<td>operand</td>
<td>0</td>
<td>No operation</td>
</tr>
<tr>
<td>1 or 2</td>
<td>1</td>
<td>Load value from data transfer mechanism</td>
</tr>
<tr>
<td>result</td>
<td>0</td>
<td>No operation</td>
</tr>
<tr>
<td>1 or 2</td>
<td>1</td>
<td>Send value to data transfer mechanism</td>
</tr>
<tr>
<td>register</td>
<td>0 0 x x x x</td>
<td>No operation</td>
</tr>
<tr>
<td>interface</td>
<td>0 1 x x x x</td>
<td>Move register xxxx to data transfer mechanism</td>
</tr>
<tr>
<td></td>
<td>1 0 x x x x</td>
<td>Move data transfer mechanism to register xxxx</td>
</tr>
<tr>
<td></td>
<td>1 1 x x x x</td>
<td>No operation</td>
</tr>
</tbody>
</table>
Microcode Instructions For Our Example

- Diagram shows how groups of bits in an instruction are interpreted
- Each set of bits controls one hardware unit
Example Horizontal Microcode Steps

- Move the value from register 4 to the hardware unit for operand 1
- Move the value from register 13 to the hardware unit for operand 2
- Arrange for the ALU to perform addition
- Move the value from the hardware unit for result2 (the low-order bits of the result) to register 4
Example Horizontal Microcode
(In Binary)

<table>
<thead>
<tr>
<th>Instr.</th>
<th>ALU</th>
<th>OP₁</th>
<th>OP₂</th>
<th>RES₁</th>
<th>RES₂</th>
<th>REG. INTERFACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0 1 0 1 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0 0 1 1 1 0 1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1 1 0 0 1 0 0</td>
</tr>
</tbody>
</table>

- Observe that the code does not resemble a conventional program
Horizontal Microcode And Timing

- Each microcode instruction takes one micro cycle
- Given functional unit may require more than one cycle to complete an operation
- Programmer must accommodate hardware timing or errors can result
- To wait for functional unit, insert microcode instructions that continue the operation
- Similar to no-op
**Example Of Continuing An Operation**

<table>
<thead>
<tr>
<th>ALU</th>
<th>OP₁</th>
<th>OP₂</th>
<th>RES₁</th>
<th>RES₂</th>
<th>REG. INTERFACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Assume ALU operation \(111\) acts as a delay to continue the previous operation
- None of the other hardware units are active
Example Of Parallel Execution

- A single microcode instruction can continue the ALU operation and also load the value from register 7 into operand unit 1

- By using horizontal microcode, a programmer can specify simultaneous, parallel operation of multiple hardware units
Intelligent Microprocessor

- Schedules instructions by assigning work to functional units
- Handles operations in parallel
- Performs branch optimization by beginning to execute both paths of a branch
- Constrains results so instructions have sequential semantics
  - Keeps results separate
  - Decides which path to use when branch direction finally known
Taming Parallel Execution Units

- Parallel hardware can
  - Compute values out-of-order
  - Follow two possible branches
- CPU must preserve sequential macro execution semantics as expected by programmer
- Mechanisms used
  - Scoreboard
  - Re-Order Buffer (ROB)
- Note: when results computed from two paths, CPU eventually discards results that are not needed
Branch Prediction

- Alternative to parallel execution
- Handles conditional execution
- Hardware assumes branch *will* be taken, and unrolls computation if it is not
- Note: studies show branch is taken approximately 60% of the time
Summary

- CPU offers modes of execution that determine protection and privilege
- Complex CPU usually implemented with microcode
- Vertical microcode uses conventional instruction set
- Horizontal microcode uses unconventional instructions
- Each horizontal microcode instruction controls underlying hardware units
- Horizontal microcode offers parallelism
Summary
(continued)

- Most complex CPUs have mechanism to schedule instructions on parallel execution units
- Scoreboard and Re-Order Buffer used to maintain sequential semantics
Questions?
Module IX

Assembly Languages
And
Programming Paradigm
Characteristics Of High-Level Language

- One-to-many translation (statement translates to multiple machine instructions)
- Hardware independence
- Application orientation
- General-purpose
- Powerful abstractions
Characteristics Of Low-Level Language

- One-to-one translation (each statement translates to one machine instruction)
- Hardware dependence
- Systems programming orientation
- Special-purpose
- Few abstractions
Perlis’ Comment On Language Level

- Computer scientist Alan Perlis once quipped that a programming language is low-level if programming requires attention to irrelevant details.
- Perlis’ point: because most applications do not need direct control of hardware, a low-level language increases programming complexity without providing benefits.
- In most cases, programmers do not need assembly language, only compilers do.
Terminology

- **Assembly language**
  - Term used for a special type of low-level language
  - Each assembly language is specific to a processor

- **Assembler**
  - Term used for a program that translates assembly language into binary code
  - Analogous to compiler
An Important Concept

• Bad news
  – Many assembly languages exist
  – Each has instructions for one particular processor architecture

• Good news
  – Assembly languages all have the same general structure
  – A programmer who understands one assembly language can learn another quickly
Our Approach

- We will discuss general concepts in class
- You will learn two specific assembly languages in lab
Assembly Language Statements

• General format

  label:  opcode  operand\(_1\),  operand\(_2\),  ...

• Most assembly languages use whitespace to separate items in a statement

• Label is optional and is only needed for branching

• Opcode and operands are processor specific
Opcode Names

- Specific to each assembly language
- Most assembly languages use short mnemonics
- Examples
  - `ld` instead of `load_value_into_register`
  - `jsr` instead of `jump_to_subroutine`
Comment Syntax

- Typically
  - A character reserved to start a comment
  - Comment extends to end of line

- Examples of comment characters
  - Pound sign (#)
  - Semicolon (;)
Commenting Conventions

- Similar to high-level languages: block comments are used to explain the overall purpose of each large section of code
- Unlike high-level languages: each line of assembly code usually contains a comment explaining purpose of the instruction
Block Comment Example

# Search linked list of free memory blocks to find a block of size N bytes or greater. Pointer to list must be in register 3 and N must be in register 4. The code also destroys the contents of register 5, which is used to walk the list.

#############################################################
# #
# Search linked list of free memory blocks to find a block #
# of size N bytes or greater. Pointer to list must be in #
# register 3 and N must be in register 4. The code also #
# destroys the contents of register 5, which is used to #
# walk the list. #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
# #
Per-Line Comment Example

```
ld  r5, r3    # load the address of list into r5
loop_1: cmp  r5, r0    # test to see if at end of list
             bz notfnd  # if reached end of list go to notfnd
```

- Note: it is typical to find a comment on every line of an assembly language program
Operand Order

• Annoying fact: assembly languages differ on operand order

• Example
  – Consider an instruction to move (i.e., copy) register 5 to register 3
  – There are two possible operand orders

    \[
    \text{mov} \ r5, r3 \quad \# \text{left-to-right order (source on left)}
    \]

    \[
    \text{mov} \ r3, r5 \quad \# \text{right-to-left order (source on right)}
    \]

• Note: in one historic case, DEC and AT&T each built an assembly language for the same processor, and they used opposite orders for operands!
Remembering Operand Order

- When programming an assembly language that uses
  
  \((\text{source, destination})\)

  remember that \textit{we read left-to-right}

- When programming an assembly language that uses
  
  \((\text{destination, source})\),

  remember that the operands are in the same order as an assignment statement
Names For General-Purpose Registers

- Registers are used heavily
- Most assembly languages use short names for registers
- Typical format is letter $r$ followed by a number, such as r1
- However... various assembly languages have used variants (e.g., reg1, R1, $1$)
- And some assembly languages assign registers names instead of numbers (e.g., ax, bx, cx, sp)
Symbolic Definitions

- Some assemblers permit a programmer to define abbreviations
- Analogous to `#define` in C
- Example definitions

```
# Define register names used in the program
#
r1    register 1    # define name r1 to be register 1
r2    register 2    # and so on for r2, r3, and r4
r3    register 3
r4    register 4
```
Using Meaningful Names

- Symbolic definition allows meaningful names
- Can make code easier to understand
- Example: registers used for a linked list

```c
#
# Define register names for a linked list program
#
listhd register 6     # holds starting address of list
listptr register 7    # moves along the list
```
Specifying The Operand Type

- Assembly language provides a way to specify the type of each operand (e.g., immediate, register, memory reference, indirect memory reference)
- Typically, compact syntax is used
- Example using right-to-left order

```plaintext
mov r3, r4    # copy contents of reg. 4 into reg. 3
mov r2, (r1)  # treat r1 as a pointer to memory and
              # copy from the mem. location to reg. 2
```
Assembly Language Idioms

- Assembly language has no way to declare programming abstractions
  - No data aggregates (arrays or structs)
  - No control structures (while loops, if-then-else, case)
  - No function declarations or arguments
- Programmer can only write a sequence of instructions
- To make code readable, programmer must follow conventions that others expect
- Term *idiom* is used to describe conventional code structure
- Next slides show example idioms
Assembly Language For Conditional Execution

\[
if (\text{condition}) \{
\begin{align*}
\text{body} \\
\end{align*}
\}
\text{next statement;}
\]

- code to test the condition and set the condition code
- branch to label if condition false
- code to perform body
- label: code for next statement
Assembly Language For If-Then Else

```plaintext
if (condition) {
    then_part
} else {
    else_part
}
next statement;
```

code to test the condition and set the condition code
branch to label1 if condition false
code to perform then_part
branch to label2
label1: code for else_part
label2: code for next statement
Assembly Language For Definite Iteration

\[\text{for } (i=0; i<10; i++) \{ \text{body} \}\]

set r4 to zero

\text{label1:compare r4 to 10}

branch to label2 if \( \geq \)

code to perform body

\text{increment r4}

\text{branch to label1}

\text{label2:code for next statement}
Assembly Language For Indefinite Iteration

\[ \text{while (condition)} \{ \]
\[ \quad \text{body} \]
\[ \} \]
\[ \text{next statement;} \]

\[ \text{label1: code to compute condition} \]
\[ \text{branch to label2 if false} \]
\[ \text{code to perform body} \]
\[ \text{branch to label1} \]
\[ \text{label2: code for next statement} \]
Assembly Language For Procedure Call

x () {
    body of function x
}

x ();
other statement;
x ();
next statement;

x: code for body of x
ret

jsr x
code for other statement
jsr x
code for next statement
Argument Passing

- Hardware possibilities
  - Stack in memory used for arguments
  - Register windows used to pass arguments
  - Special-purpose argument registers used

- Consequence: assembly language for passing arguments depends on hardware

- See Appendix 3 and Appendix 4 in the text for x86 and MIPS calling sequence
Example Argument Passing
Using Registers 1 and 2

\[ x \left( a, b \right) \{ \]
\[ \quad \text{body of function } x \]
\[ \} \]

\[ x \left( -4, 17 \right); \]
\[ \quad \text{load } -4 \text{ into register 1} \]
\[ \quad \text{load } 17 \text{ into register 2} \]
\[ \quad \text{jsr } x \]
\[ \text{code for body of } x \text{ that assumes} \]
\[ \quad \text{register 1 contains parameter } a \]
\[ \quad \text{and register 2 contains } b \]
\[ \text{ret} \]

\[ \text{other statement; } \]
\[ x \left( 71, 27 \right); \]
\[ \quad \text{load } 71 \text{ into register 1} \]
\[ \quad \text{load } 27 \text{ into register 2} \]
\[ \quad \text{jsr } x \]
\[ \text{code for other statement} \]

\[ \text{next statement} \]
\[ \text{code for next statement} \]
Function Invocation

- Like procedure invocation except also returns a result
- Computers have been built that return a value
  - On a stack in memory
  - In a special-purpose register
  - In a general-purpose register
- Choice may depend on compiler
When Will You Need Assembly Language?

- When debugging really tough problems
- When a high-level language does not produce code that is fast enough
- When a high-level language does not have facilities to use special-purpose instructions
- General rule: assembly language is *only* used for functions where a high-level language has insufficient functionality or results in poor performance
Interaction With High-Level Language

- Assembly language program can call function written in high-level language (e.g., to avoid writing complex functions in assembly language)
- High-level language program can call function written in assembly language
  - When higher speed is needed
  - When access to special-purpose hardware is required
- Interactions must follow calling conventions of the high-level language
Declaration Of Variables In Assembly Language

- Most assembly languages have no variable declarations or variable types
- However, a programmer can reserve a block of storage for a variable, and use a label to allow the block to be referenced in instructions
- Typical directives to reserve storage
  - .word
  - .byte or .char
  - .long
Examples Of Equivalent Declarations

- int x, y, z;  x: .long
- y: .long
- z: .long
- short w, q;
- w: .word
- q: .word

statement(s) code for statement(s)

- Warning: code and variable storage can be intermixed
- Good news: many assemblers allow a programmer to place code and data in separate memory segments
Specifying Initial Values

- Usually allowed as arguments to directives
- Example to declare 16-bit storage with initial value 949

\[ x: \text{.word } 949 \]
Assembler

- Software component
- Accepts assembly language program as input
- Produces binary form of program as output
- Uses two-pass algorithm
  - Pass 1: computes instruction offset for each label
  - Pass 2: generates code
What An Assembler Provides

- Each statement in source program is translated to one machine instruction
- Assembler
  - Computes relative location for each label
  - Fills in branch offsets automatically
  - Allows a programmer to use mnemonic labels instead of byte offsets
Example Of Code Offsets And Labels

<table>
<thead>
<tr>
<th>locations</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00 – 0x03</td>
<td>x: .long</td>
</tr>
<tr>
<td>0x04 – 0x07</td>
<td>label1: cmp r1, r2</td>
</tr>
<tr>
<td>0x08 – 0x0B</td>
<td>bne label2</td>
</tr>
<tr>
<td>0x0C – 0x0F</td>
<td>jsr label3</td>
</tr>
<tr>
<td>0x10 – 0x13</td>
<td>label2: load r3, 0</td>
</tr>
<tr>
<td>0x14 – 0x17</td>
<td>br label4</td>
</tr>
<tr>
<td>0x18 – 0x1B</td>
<td>label3: add r5, 1</td>
</tr>
<tr>
<td>0x1C – 0x1F</td>
<td>ret</td>
</tr>
<tr>
<td>0x20 – 0x23</td>
<td>label4: ld r1, 1</td>
</tr>
<tr>
<td>0x24 – 0x27</td>
<td>ret</td>
</tr>
</tbody>
</table>

- In `bne` instruction, assembler uses 0x10 in place of `label2`
Assembly Language Macros

- Syntactic substitution
- Parameterized for flexibility
- Programmer supplies *macro definitions*
- Code contains *macro invocations*
- Assembler handles *macro expansion* in extra pass
- Known as *macro assembly language*
- Note: assembly macros predate #define
Macro Syntax

- Varies among assembly languages
- Typical definition bracketed by keywords
- Example keywords
  - *macro*
  - *endmacro*
- Invocation
  - Uses macro name
  - Allows arguments
- Note: Unix assemblers often use cpp as a macro processor
Example Of Macro Definition

- **Definition of macro addmem**

  ```
  macro  addmem(a, b, c)
  load r1, a  # load 1st arg into register 1
  load r2, b  # load 2nd arg into register 2
  add  r1, r2 # add register 2 to register 1
  store r3, c # store the result in 3rd arg
  endmacro
  ```

- **Code produced by addmem(xxx, YY, zqz)**

  ```
  load r1, xxx  # load 1st arg into register 1
  load r2, YY   # load 2nd arg into register 2
  add  r1, r2  # add register 2 to register 1
  store r3, zqz # store the result in 3rd arg
  ```
Programming With Macros

- Macros only provide syntactic substitution
  - Parameters are treated as a string of characters
  - Arbitrary text permitted
  - No error checking performed

- Consequences for programmers
  - An extra blank can change the meaning of the instruction
  - Macro invocation can generate invalid code
  - May be difficult to debug
Example Of Illegal Code That Can Result From A Macro Expansion

- Calling `addmem(1+, %*J, +)` results in

```plaintext
load r1, 1+  # load 1st arg into register 1
load r2, %*J # load 2nd arg into register 2
add r1, r2   # add register 2 to register 1
store r3, +   # store the result in 3rd arg
```

- Assembler substitutes macro arguments literally
- Error messages refer to expanded code, not macro definition
- It may be hard to trace errors back to macro invocations
Summary

- Assembly language is low-level and incorporates details of a specific processor
- Many assembly languages exist, one per processor
- Each assembly language statement corresponds to one machine instruction
- Same basic programming paradigm used in most assembly languages
- Programmers must code assembly language equivalents of abstractions such as
  - Conditional execution
  - Definite and indefinite iteration
  - Function call
Summary (continued)

- Assembler translates assembly language program into binary code
- Assembler uses two-pass processing
  - First pass assigns locations to labels
  - Second pass generates code
- Macro assemblers have additional pass to expand macros
Questions?
Module X

Memory And Storage
Two Key Aspects Of Memory

- **Technology**
  - The type of the underlying hardware
  - Choice determines cost, persistence, performance
  - Many variants are available

- **Organization**
  - How underlying hardware is used to build memory system (i.e., bytes, words, etc.)
  - Directly visible to programmer
Memory Characteristics

- Volatile or nonvolatile
- Random or sequential access
- Read-write or read-only
- Primary or secondary
Memory Volatility

• **Volatile memory**
  – Contents disappear when power is removed
  – Fastest access times
  – Least expensive

• **Nonvolatile memory**
  – Contents remain without power
  – More expensive than volatile memory
  – May have slower access times
  – Some embedded systems “cheat” by using a battery to maintain memory contents
Memory Access Paradigm

- Random access
  - Typical for most applications

- Sequential access
  - Known as a \textit{FIFO (First-In-First-Out)}
  - Typically associated with \textit{streaming} applications
  - Requires special purpose hardware
Permanence Of Nonvolatile Memory

- **ROM (Read Only Memory)**
  - Values can be read, but not changed
  - Useful for firmware

- **PROM (Programmable Read Only Memory)**
  - Contents can be altered, but doing so is time-consuming
  - Change may involve removal from a circuit, exposure to ultraviolet light

- **Flash**
  - Contents can be altered easily
  - Used in solid state disks and digital cameras
Primary And Secondary Memory

• **Primary memory**
  – Highest speed
  – Most expensive, and therefore the smallest
  – Typically solid state technology

• **Secondary memory**
  – Lower speed
  – Less expensive, and therefore can be larger
  – Traditionally used magnetic media and electromechanical drive mechanisms
  – Moving to solid state (flash)
In Practice

- Distinction between primary and secondary
  - Used to be absolutely clear
  - Is now blurring

- Secondary memory is now using solid state technology instead of electromechanical technology

- Examples
  - Flash cards used in smart phones
  - Solid-state disks (SSDs) used in laptop computers
Memory Hierarchy

- Key concept to memory design
- Extend the primary/secondary tradeoff to multiple levels
- Basic idea
  - Highest performance memory costs the most
  - Can obtain better performance at lower cost by using a set of memories
- The key is choosing the memory sizes and speeds carefully
High Performance At Low Cost

- Select a set of memories
- A small memory has highest performance
- A slightly larger amount of memory has somewhat lower performance
- The largest memory has the lowest performance
- Example hierarchy
  - Dozens of general-purpose registers
  - A dozen gigabytes of main memory
  - Several terabytes of solid state disk
Review: Two Paradigms For Main Memory

- Harvard architecture
  - Two separate memories known as
    * Instruction store
    * Data store
  - One memory holds programs and the other holds data
  - Used on early computers and some embedded systems
- Von Neumann architecture
  - A single memory holds both programs and data
  - Used on most general-purpose computers
Consequence Of A Von Neumann Architecture
Consequence Of A Von Neumann Architecture

- Instructions and data occupy the same memory
Consequence Of A Von Neumann Architecture

- Instructions and data occupy the same memory
- Consider the following C code

```c
short main[] = {
-25117, -16480, 16384, 28, -28656, 8296, 16384, 26, -28656, 8293, 16384, 24, -28656, 8300, 16384, 22, -28656, 8300, 16384, 20, -28656, 8303, 16384, 18, -28656, 8224, 16384, 16, -28656, 8311, 16384, 14, -28656, 8303, 16384, 12, -28656, 8306, 16384, \n', -28656, 8300, 16384, \b', -28656, 8292, 16384, 6, -28656, 8238, 16384, 4, -28656, 8202, -32313, -8184, -32280, 0, -25117, -16480, 4352, 5858, -18430, 8600, -4057, -24508, -17904, 8192, -17913, 24577, -32601, 16412, 9919, -1, -17913, 24577, -27632, 8193, -28656, 8193, 16384, 4, -28153, -24505, -32313, -8184, -32280, 0, -32240, 8196, -28208, 8192, 6784, 4, 6912, \b', -26093, 24800, -32317, 16384, 256, 0, -32317, -8184, 256, 0, 0, 0, -32240, 8193, -28208, 8192, 768, \b', -12256, 24816, -32317, -8184, -28656, 16383
};
```
Consequence Of A Von Neumann Architecture

- Instructions and data occupy the same memory

- Consider the following C code

```c
short main[] = {
    -25117, -16480, 16384, 28, -28656, 8296, 16384, 26, -28656, 8293, 16384,
    24, -28656, 8300, 16384, 22, -28656, 8300, 16384, 20, -28656, 8303,
    16384, 18, -28656, 8224, 16384, 16, -28656, 8311, 16384, 14, -28656,
    8303, 16384, 12, -28656, 8306, 16384, '\n', -28656, 8300, 16384, '\b',
    -28656, 8292, 16384, 6, -28656, 8238, 16384, 4, -28656, 8202, -32313,
    -8184, -32280, 0, -25117, -16480, 4352, 5858, -18430, 8600, -4057,
    -24508, -17904, 8192, -17913, 24577, -32601, 16412, 9919, -1, -17913,
    24577, -27632, 8193, -28656, 8193, 16384, 4, -28153, -24505, -32313,
    -8184, -32280, 0, -32240, 8192, -28208, 8192, 6784, 4, 6912, '\b', -26093,
    24800, -32317, 16384, 256, 0, -32317, -8184, 256, 0, 0, 0, -32240, 8193,
    -28208, 8192, 768, '\b', -12256, 24816, -32317, -8184, -28656, 16383
};
```

- Does the code specify instructions or data?
Consequence Of A Von Neumann Architecture

- Instructions and data occupy the same memory
- Consider the following C code

```c
short main[] = {
-25117, -16480, 16384, 28, -28656, 8296, 16384, 26, -28656, 8293, 16384,
24, -28656, 8300, 16384, 22, -28656, 8300, 16384, 20, -28656, 8303,
16384, 18, -28656, 8224, 16384, 16, -28656, 8311, 16384, 14, -28656,
8303, 16384, 12, -28656, 8306, 16384, \'\n\', -28656, 8300, 16384, \'\b\',
-28656, 8292, 16384, 6, -28656, 8238, 16384, 4, -28656, 8202, -32313,
-8184, -32280, 0, -25117, -16480, 4352, 5858, -18430, 8600, -4057,
-24508, -17904, 8192, -17913, 24577, -32601, 16412, 9919, -1, -17913,
24577, -27632, 8193, -28656, 8193, 16384, 4, -28153, -24505, -32313,
-8184, -32280, 0, -32240, 8196, -28208, 8192, 6784, 4, 6912, \'\b\', -26093,
24800, -32317, 16384, 256, 0, -32317, -8184, 256, 0, 0, 0, -32240, 8193,
-28208, 8192, 768, \'\b\', -12256, 24816, -32317, -8184, -28656, 16383
};
```

- Does the code specify instructions or data?
- Answer: on a Sparc, it compiles and prints *hello world*
Tradeoffs For Separate Memories

• Advantages
  – Allows separate caches (described later)
  – Permits memory technology to be optimized for access patterns
    * Instructions: sequential access
    * Data: random access

• Disadvantage
  – Must choose a size for each when computer is designed
The Fetch-Store Paradigm

- Access paradigm used by memory
- Hardware only supports two operations
  - *Fetch* a value from a specified location
  - *Store* a value into a specified location
- Programmers often use the terms *read* and *write*
- We will discuss the implementation and consequences of fetch/store later
Summary

- The two key aspects of memory are
  - Technology
  - Organization
- Memory can be characterized as
  - Volatile or nonvolatile
  - Random or sequential access
  - Permanent or nonpermanent
  - Primary or secondary
Separating instruction and data memories has potential advantages but a big disadvantage.

Memory systems use fetch-store paradigm.

Only two operations available:
- Fetch (read)
- Store (write)
Questions?
Module XI

Physical Memory

And

Physical Addressing
Computer Memory

- Main memory
  - Designed to permit arbitrary pattern of references
  - Known by the term RAM (*Random Access Memory*)
- Usually volatile
- Two basic technologies available
  - Static RAM
  - Dynamic RAM
Static RAM (SRAM)

- Easiest to understand
- Basic elements built from a latch

When *enable* is asserted (i.e., logical 1), output is same as input
- Once enable line goes to logical 0, output is the last input value
Advantages And Disadvantages Of SRAM

• Advantages
  – High speed
  – Access circuitry is straightforward

• Disadvantages
  – Higher power consumption
  – Heat generation
  – High cost
Dynamic RAM (DRAM)

- Alternative to SRAM
- Consumes less power
- Analogous to a capacitor (i.e., stores an electrical charge)
The Facts Of Electronic Life
The Facts Of Electronic Life

- Entropy increases
The Facts Of Electronic Life

- Entropy increases
- Any electronic storage device gradually loses charge
The Facts Of Electronic Life

- Entropy increases
- Any electronic storage device gradually loses charge
- When left for a long time, a bit in DRAM changes from logical 1 to logical 0
The Facts Of Electronic Life

- Entropy increases
- Any electronic storage device gradually loses charge
- When left for a long time, a bit in DRAM changes from logical 1 to logical 0
- Discharge time can be less than a second
The Facts Of Electronic Life

- Entropy increases
- Any electronic storage device gradually loses charge
- When left for a long time, a bit in DRAM changes from logical 1 to logical 0
- Discharge time can be less than a second
- Conclusion: although it is inexpensive, DRAM is a horrible memory device!
Making DRAM Work

- Cannot leave bits too long or they change
- Additional hardware known as a refresh circuit is used
- Trick: refresh circuitry repeatedly
  - Steps through each location $i$ of DRAM
  - Reads the value from location $i$
  - Writes same value back into location $i$ (i.e., recharges the memory)
- Note: refresh hardware runs in the background at all times
Illustration Of A DRAM Refresh Circuit

- Much more complex than the figure implies
- Refresh must not interfere with normal read and write operations
  - Correctness must be guaranteed
  - Performance must not suffer
Measures Of Memory

• Density
  – Refers to memory cells per square area of silicon
  – Usually stated as number of bits on standard size chip
  – Example: 1 gig chip holds 1 gigabit of memory
  – Note: higher density chip generates more heat

• Latency
  – Time that elapses between the start of an operation and the completion of the operation
  – May depend on previous operations (see below)
Separation Of Read And Write Latency

- In many memory technologies
  - The time required to store exceeds the time required to fetch
  - Difference can be dramatic
- Consequence: any measure of memory performance must give two values
  - Performance of read
  - Performance of write
Memory Organization

- Hardware unit called a *memory controller* connects a processor to a physical memory

- Main point: because all memory requests go through the controller, the interface a processor “sees” can differ from the underlying hardware organization
Steps Taken To Honor A Memory Request

- **Processor**
  - Presents request to controller
  - Waits for response

- **Controller**
  - Translates request into signals for physical memory chips
  - Returns answer to processor as quickly as possible
  - Sends signals to *reset* physical memory for next request
Consequence Of Memory Reset

- Means next memory operation may be delayed
- Conclusion
  - Latency of a single operation is an insufficient measure of performance
  - Must measure the time required for successive operations
Memory Cycle Time

- Time that elapses between two successive memory operations
- More accurate measure than latency
- Two separate measures
  - Read cycle time (tRC)
  - Write cycle time (tWC)
Synchronous Memory Technologies

- Both memory and processor use a clock
- Synchronized memory systems ensure two clocks coincide
- Allows higher memory speeds
- Technologies
  - *Synchronous Static Random Access Memory (SSRAM)*
  - *Synchronous Dynamic Random Access Memory (SDRAM)*
- Note: the RAM in most computers is SDRAM
Multiple Data Rate Memory Technologies

• Goals
  – Improve memory performance
  – Avoid mismatch between CPU speed and memory speed

• Technique: memory hardware runs at a multiple of the CPU clock rate

• Available for both SRAM and DRAM

• Examples
  – Double Data Rate SDRAM (DDR-SDRAM)
  – Quad Data Rate SRAM (QDR-SRAM)
A Sample Of Memory Technologies

- Many memory technologies exist
- Examples include

<table>
<thead>
<tr>
<th>Technology</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDR-DRAM</td>
<td>Double Data Rate Dynamic RAM</td>
</tr>
<tr>
<td>DDR-SDRAM</td>
<td>Double Data Rate Synchronous Dynamic RAM</td>
</tr>
<tr>
<td>FCRAM</td>
<td>Fast Cycle RAM</td>
</tr>
<tr>
<td>FPM-DRAM</td>
<td>Fast Page Mode Dynamic RAM</td>
</tr>
<tr>
<td>QDR-DRAM</td>
<td>Quad Data Rate Dynamic RAM</td>
</tr>
<tr>
<td>QDR-SRAM</td>
<td>Quad Data Rate Static RAM</td>
</tr>
<tr>
<td>SDRAM</td>
<td>Synchronous Dynamic RAM</td>
</tr>
<tr>
<td>SSRAM</td>
<td>Synchronous Static RAM</td>
</tr>
<tr>
<td>ZBT-SRAM</td>
<td>Zero Bus Turnaround Static RAM</td>
</tr>
<tr>
<td>RDRAM</td>
<td>Rambus Dynamic RAM</td>
</tr>
<tr>
<td>RLDRAM</td>
<td>Reduced Latency Dynamic RAM</td>
</tr>
</tbody>
</table>
Memory Organization

- Parallel interface used between computer and memory
- Called a *bus* (more later in the course)
Memory Transfer Size

• Amount of memory that can be transferred to computer simultaneously
• Determined by bus between computer and controller
• Example memory transfer sizes
  – 16 bits
  – 32 bits
  – 64 bits
• Important to programmers
Physical Memory And Word Size

• Bits of physical memory are divided into blocks of \( N \) bits each
• \( N \) is determined by bus width
• Terminology
  – Group of \( N \) bits is called a \textit{word}
  – \( N \) is known as the \textit{width} of a word or the \textit{word size}
• Computer is often characterized by its word size (e.g., one might speak of a 64-bit computer)
Physical Memory Addresses

- Each word of memory is assigned a unique number known as a *physical memory address*.
- Physical memory is organized as an array of words.

- Underlying hardware applies *read* or *write* to entire word.
Choosing A Physical Word Size

- Word size represents a fundamental tradeoff
- Larger word size
  - Results in higher performance
  - Requires more parallel wires and circuitry
  - Has higher cost and more power consumption
- Note: architect usually designs all data paths in a computer to use one size for
  - Word in physical memory
  - Integers and general-purpose registers
  - Floating point numbers and floating-point registers
Byte Addressing And Translation

- Byte addressing
  - View of memory presented to processor
  - Each byte of memory assigned an address
  - Convenient for programmers
  - However... the underlying memory uses word addressing

- Memory controller
  - Provides translation
  - Allows programmers to use byte addresses (convenient)
  - Allows physical memory to use word addresses (efficient)
Example Of Address Translation

- Assume physical memory is organized into 32-bit words
- Programmer views memory as an array of bytes
- We think of each byte has having an address 0 through \( N-1 \)
- Each physical word corresponds to 4 byte addresses

![Physical Address Table](image)

- A byte address assigned to each byte of each word

- 32 bits
Given A Byte Address, B, Find The Byte

• Let $N$ be the number of bytes per word

• The physical address of the word containing the byte is

\[ W = \left\lfloor \frac{B}{N} \right\rfloor \]

• And the byte offset within the word is

\[ O = B \mod N \]

• Example
  
  – Find byte $B = 11$ when $N = 4$
  
  – $B$ can be found in word 2 at offset 3
Efficient Translation

- Think binary and choose word size $N$ to be a power of 2
- Avoids arithmetic calculations, especially division and remainder
- Word address computed by extracting high-order bits
- Offset computed by extracting low-order bits
- Example: byte 11 with $N$ equal to 4 bytes per word

![Diagram showing byte address, word address, and offset](image-url)
Byte Alignment

- Refers to storing multibyte values (e.g., integers) in memory
- Two designs have been used
  - Access \textit{must} correspond to word boundary in underlying physical memory
  - Access can be unaligned, memory controller handles details, but \textit{fetch} and \textit{store} operations are slower
- Unaligned version is common
- Consequences for programmers
  - Performance may be improved by aligning integers
  - Some I/O devices require buffers to be aligned
Memory Size And Address Space

- Size of address limits maximum memory
- Example: 32-bit address can represent
  \[2^{32} = 4,294,967,296\]
  unique addresses
- Known as *address space*
- Note: word addressing allows larger memory than byte addressing, but is seldom used because it is difficult to program
Measures Of Memory Size

- Memory sizes expressed as powers of two, not powers of ten
- *Kilobyte* defined to be $2^{10}$ bytes
- *Megabyte* defined to be $2^{20}$ bytes
- *Gigabyte* defined to be $2^{30}$ bytes
- *Terabyte* defined to be $2^{40}$ bytes
Measure Of Network Speed

- Speeds of data networks and other I/O devices are usually expressed in powers of ten
  - Example: a *Gigabit Ethernet* operates at $10^9$ bits per second
- Programmer must accommodate differences between measures for storage and transmission
C Programming And Memory Addressability

- C has a heritage of both byte and word addressing
- Example of byte pointer declaration
  ```
  char    *iptr;
  ```
- Example of word pointer declaration
  ```
  int     *iptr;
  ```
- If integer size is four bytes, iptr ++ increments by four
Memory Dump

- Debugging tool
- Gives hex representation of bytes in memory
- Each line of output specifies memory address and bytes starting at that address
Example Memory Dump: Linked List In Memory

- *Head* consists of pointer to the list
- Each *node* has the following structure

```c
struct node {
    int value;
    struct node *next;
}
```

- Example list has structure

```
head

node 1
  192

node 2
  200

node 3
  100
```
Memory Dump Output

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents Of Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001bde0</td>
<td>00000000 0001bdf8</td>
</tr>
<tr>
<td>0001bdf0</td>
<td>6d657200 0001be18</td>
</tr>
<tr>
<td>0001be00</td>
<td>00000064 00000000</td>
</tr>
<tr>
<td>0001be10</td>
<td>00000000 000000c8</td>
</tr>
</tbody>
</table>
### Memory Dump Output

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents Of Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001bde0</td>
<td>00000000 0001bdf8 deadbeef 4420436f</td>
</tr>
<tr>
<td>0001bdf0</td>
<td>6d657200 0001be18 000000c0 0001be14</td>
</tr>
<tr>
<td>0001be00</td>
<td>00000064 00000000 00000000 00000002</td>
</tr>
<tr>
<td>0001be10</td>
<td>00000000 000000c8 0001be00 00000006</td>
</tr>
</tbody>
</table>

- Assume head is located at address 0x0001bde4
Memory Dump Output

Assume head is located at address 0x0001bde4

First node at 0x0001bdf8 contains value 192 (0xc0)
Memory Dump Output

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents Of Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001bde0</td>
<td>00000000 0001bdf8 1deadbeef 4420436f</td>
</tr>
<tr>
<td>0001bdf0</td>
<td>6d657200 0001be18 000000c0 0001be14</td>
</tr>
<tr>
<td>0001be00</td>
<td>00000064 00000000 00000000 00000002</td>
</tr>
<tr>
<td>0001be10</td>
<td>00000000 000000c8 0001be00 00000006</td>
</tr>
</tbody>
</table>

- Assume head is located at address 0x0001bde4
- First node at 0x0001bdf8 contains value 192 (0xc0)
- Second node at 0x0001be14 contains value 200 (0xc8)
Assume head is located at address 0x0001bde4

First node at 0x0001bdf8 contains value 192 (0xc0)

Second node at 0x0001be14 contains value 200 (0xc8)

Last node at 0x0001be00 contains value 100 (0x64)
Increasing Physical Memory Performance

- Two major techniques
  - Memory banks
  - Interleaving
- Both employ parallel hardware
Memory Banks

- Modular approach to constructing large memory
- Basic memory module is replicated multiple times
- Selection circuitry chooses which bank
- Basic idea
  - Use high-order bits of address to select a bank
  - Use low-order bits to select a word within a bank
- Key ideas
  - Hardware for each bank is identical
  - Parallel access — one bank can reset while another is being used
Address Bits Passed To Memory Banks

Address

$\text{SE}$

$k$ low-order bits passed to all memory banks

high-order bits used to select a bank

four identical memory modules that each handle addresses $0$ to $2^{k-1}$

Bank 0

Bank 1

Bank 2

Bank 3
Programming With Memory Banks

- Two approaches have been used
  - Transparent
    - Programmer is not concerned with banks
    - Hardware automatically finds and exploits parallelism
  - Opaque
    - Programmer informed about banks
    - To optimize performance, programmer must place items that will be accessed sequentially in separate banks
Interleaving

- Related to memory banks
- Transparent to programmer
- Hardware places consecutive words (or consecutive bytes) in separate physical memories
- Technique: use low-order bits of address to choose module
- Known as $N$-way interleaving, where $N$ is number of physical memories
Illustration Of 4-Way Interleaving

- Consecutive words stored in separate physical memories
Content Addressable Memory (CAM)

- Blends two key ideas
  - Memory technology
  - Memory organization
- Includes parallel hardware for high-speed search
CAM

- Think of CAM as a two-dimensional array of special-purpose hardware cells
- A row in the array is called a *slot*
- The hardware cells
  - Can answer the question: “Is X stored in any row of the CAM?”
  - Operate in parallel to make search fast
- Query is known as a *key*
Illustration Of CAM

Key

one slot

CAM Storage
Lookup In A CAM

- CAM presented with key for lookup
- Hardware cells test whether key is present
  - Search operation performed in parallel on all slots simultaneously
  - Result is index of slot where value found
- Note: parallel search hardware makes CAM expensive
Ternary CAM (TCAM)

- Variation of CAM that adds *partial match searching*
- Each bit in slot can have one of three possible values
  - Zero
  - One
  - Don’t care
- TCAM ignores “don’t care” bits and reports match
- TCAM can either report
  - First match
  - All matches (bit vector)
Summary

• Physical memory
  – Organized into fixed-size words
  – Accessed through a controller
• Controller can use
  – Byte addressing when communicating with a processor
  – Word addressing when communicating with a physical memory
• To avoid arithmetic, use powers of two for
  – Address space size
  – Bytes per word
Summary (continued)

- Many memory technologies exist
- A memory dump that shows contents of memory in a printable form can be an invaluable tool
- Two techniques used to optimize memory access
  - Separate memory banks
  - Interleaving
- Content Addressable Memory (CAM) permits parallel search; variation of CAM known as Ternary Content Addressable Memory (TCAM) allows partial match retrieval
Questions?
Module XII

Caches And Caching
Caching

- Key concept in computing
- Used in hardware and software
- Memory cache is essential to reduce the Von Neumann bottleneck
Cache

- Acts as an intermediary
- Located between source of requests and source of replies

Cache contains temporary local storage
- Very high-speed
- Limited size

Copy of selected items kept in local storage
Cache answers requests from local copy when possible
Cache Characteristics

- Small (usually much smaller than storage needed for entire set of items)
- Active (makes decisions about which items to save)
- Transparent (invisible to both requester and data store)
- Automatic (uses sequence of requests; does not receive extra instructions)
Range Of Possibilities

- Implemented in hardware, software, or a combination
- Small or large data items (a byte of memory or a complete file)
- Textual or binary data
- For an individual processor or shared among processors
- Retrieval-only or store-and-retrieve
- One of the most important optimization techniques available
Cache Terminology

- *Cache hit*: request can be satisfied from cache
- *Cache miss*: request cannot be satisfied from cache
- *Locality of reference*: refers to whether requests are repeated
  - High locality means many repetitions
  - Low locality means few repetitions
- Note: cache works well with high locality of reference
Cache Performance

- Cost measured with respect to requester

\[ C_h \] is the cost of an item found in the cache (hit)
\[ C_m \] is the cost of an item not found in the cache (miss)
Analysis Of Cache Performance

- Worst case for sequence of $N$ requests

$$C_{\text{worst}} = N \cdot C_m$$
Analysis Of Cache Performance

- Worst case for sequence of $N$ requests
  
  \[ C_{\text{worst}} = N \cdot C_m \]

- Best case for sequence of $N$ requests
  
  \[ C_{\text{best}} = C_m + (N - 1) \cdot C_h \]
Analysis Of Cache Performance

- Worst case for sequence of $N$ requests

$$C_{worst} = N \cdot C_m$$

- Best case for sequence of $N$ requests

$$C_{best} = C_m + (N - 1) \cdot C_h$$

- For best case, the average cost per request is:

$$\frac{C_m + (N - 1) \cdot C_h}{N} = \frac{C_m}{N} - \frac{C_h}{N} + C_h$$

- Key idea: as $N \to \infty$, average cost approaches $C_h$
The Reason Caching Works Well

- If we ignore overhead
  - In the worst case, the performance of caching is no worse than if the cache were not present
  - In the best case, the cost per request is approximately equal to the cost of accessing the cache
- Note: for memory caches, parallel hardware means almost no overhead
Definition Of Hit and Miss Ratios

• **Hit ratio**
  – Percentage of requests satisfied from cache
  – Given as value between 0 and 1

• **Miss ratio**
  – Percentage of requests not satisfied from cache
  – Equal to 1 minus the hit ratio

• Allows us to assess expected cost
Expected Performance Of A Cache

- Access cost depends on hit ratio

\[ \text{Cost} = r \ C_h + (1 - r) \ C_m \]

where \( r \) is the hit ratio

- Notes
  - The cost of a miss is often much larger than the cost of a hit
  - The performance improves if hit ratio increases or cost of access from cache decreases
Cache Replacement Policy

- Recall: a cache is smaller than data store
- Once cache is full, existing item must be ejected before another can be inserted
- Replacement policy chooses item to eject
- Most popular replacement policy known as *Least Recently Used (LRU)*
  - Easy to implement
  - Tends to retain items that will be requested again
  - Works well in practice
Multilevel Cache Hierarchy

- Can use multiple caches to improve performance
- Arranged in hierarchy by speed (i.e., by cost)
- Example: insert an extra, faster cache in previous diagram
Analysis Of Two-Level Cache

- Cost is:

\[
Cost = r_1 C_{h1} + r_2 C_{h2} + (1 - r_1 - r_2)C_m
\]

- \( r_1 \) is fraction of hits for the new cache
- \( r_2 \) is fraction of hits for the original cache
- \( C_{h1} \) is cost of accessing the new cache
- \( C_{h2} \) is cost of accessing the original cache
Preloading Caches

- Optimization technique
- Stores items in cache before requests arrive
- Works well if data accessed in related groups
- Examples
  - When web page is fetched, web cache can preload images that appear on the page
  - When byte of memory is fetched, memory cache can preload succeeding bytes
Memory Caches

- Several memory mechanisms operate as a cache
- Examples
  - Physical memory caches
  - TLB used in a virtual memory system (covered later)
  - Pages in a demand paging system (covered later)
Physical Memory Caches

- Located between processor and physical memory
- Smaller than physical memory
- Use parallel hardware to achieve high performance
- Perform two operations in parallel
  - Search local cache
  - Send request to underlying physical memory
- If answer found in cache, cancel request to memory
The Two Basic Types Of Memory Caches

- Differ in how the caches handle a write operation
- **Write-through**
  - Place a copy of item in cache
  - Also send (write) a copy to physical memory
- **Write-back**
  - Much faster
  - Place a copy of item in cache
  - Only write the copy to physical memory when necessary
  - Works well for frequent updates (e.g., a loop increments a value)
Each processor (or core) has its own cache
Each cache can retain copy of item
*Cache coherence* needed to ensure correctness when one core changes an item and others hold a copy
Multilevel Memory Caches

- Traditional memory cache was separate from both the memory and the processor.
- To access traditional memory cache, a processor used pins that connect the processor chip to the rest of the computer.
- Using pins to access external hardware takes much longer than accessing functional units that are internal to the processor chip.
- Advances in technology have made it possible to increase the number of transistors per chip, which means a processor chip can contain a cache.
Multilevel Memory Caches

- **Level 1 cache (L1 cache)**
  - Per core
- **Level 2 cache (L2 cache)**
  - May be per core
- **Level 3 cache (L3 cache)**
  - Shared among all cores

- Historical note: definitions used to specify L1 as on-chip, L2 as off-chip, and L3 as part of memory
## Example Cache Sizes

<table>
<thead>
<tr>
<th>Cache</th>
<th>Size</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>32KB to 64KB</td>
<td>Per core</td>
</tr>
<tr>
<td>L2</td>
<td>256KB to 512KB</td>
<td>May be per core</td>
</tr>
<tr>
<td>L3</td>
<td>8MB to 20MB</td>
<td>Shared among all cores</td>
</tr>
</tbody>
</table>
Instruction And Data Caches

- Instruction references are typically sequential
  - High locality of reference
  - Amenable to prefetching
- Data references typically exhibit more randomness
  - Lower locality of reference
  - Prefetching does not work well
- Question: does performance improve with separate caches for data and instructions?
Instruction And Data Caches
(continued)

- Cache tends to work well with sequential references
- Adding many random references tends to lower cache performance
- Therefore, separating instruction and data caches can improve performance
- However: if cache is “large enough”, separation doesn’t help
- Current thinking: instead of separate caches, simply use a single larger cache
Two Memory Cache Technologies

- Direct mapped memory cache
- Set associative memory cache
Direct Mapped Memory Cache

- Divides memory into blocks of size $B$
- Blocks are numbered modulo $C$, where $C$ is slots in cache
- Example: block size of $B = 8$ bytes and cache size $C = 4$

- Also called direct mapping cache
Direct Mapped Memory Cache Operation

- When byte is referenced, always place entire block in the cache
- If block number is \( n \), place the block in cache slot \( n \)
- Use a *tag* to specify which actual addresses are currently in slot \( n \)
- Tag is the relative number of the block in memory
Illustration Of Tags

- General idea: using tags allows a smaller cache
Efficient Memory Cache

- Think binary: if all values are powers of two, bits of an address can be used to specify a tag, block, and offset

- For the example above (an unrealistically small cache)
  - Block size $B$ is 8, so use 3 bits of offset
  - Cache size $C$ is 4, so use 2 bits of block number
  - Tag is remainder of address ($32 - 5$ bits)
Algorithm For Direct Mapped Cache Lookup

Given:
    A memory address

Find:
    The data byte at that address

Method:
    Extract the tag number, t, block number, b, and offset, o, from the address.
    Examine the tag in slot b of the cache. If the tag matches t, extract the value from slot b of the cache.
    If the tag in slot b of the cache does not match t, use the memory address to extract the block from memory, place a copy in slot b of the cache, replace the tag with t, and use o to select the appropriate byte from the value.
Parallel Hardware in A Cache

incoming address

index bits

decoder selects only one slot

tag bits from address

only the selected slot passes values down

comparator

= ?

logical and

“valid” output

value output
Set Associative Memory Cache

- Alternative to direct mapped memory cache
- Uses parallel hardware
- Maintains two, independent caches

- Allows two items with same block number to be cached simultaneously
Advantage Of Set Associative Cache

• Assume two memory addresses $A_1$ and $A_2$
  – Both have block number zero
  – Have different tags

• In direct mapped cache
  – $A_1$ and $A_2$ contend for single slot
  – Only one can be cached at a given time

• In set associative cache
  – $A_1$ and $A_2$ can be placed in separate caches
  – Both can be cached at a given time
Fully Associative Cache

- Generalization of set associative cache
- Many parallel caches
- Each cache has exactly one slot
- Slot can hold arbitrary item
Conceptual Continuum Of Caches

- No parallelism corresponds to direct mapped cache
- Some parallelism corresponds to set associative cache
- More parallelism corresponds to fully associative cache
- Arbitrary parallelism corresponds to Content Addressable Memory
Consequences For Programmers

- In many programs, caching works well without extra work
- To optimize cache performance
  - Group related data items into same cache line (e.g., related bytes into a word)
  - Perform all operations on one data item before moving to another data item
How Important Is A Memory Cache?

• One day, on an operating systems project
  – Someone rewrote the processor startup code
  – They inadvertently turned off the L1 cache
How Important Is A Memory Cache?

- One day, on an operating systems project
  - Someone rewrote the processor startup code
  - They inadvertently turned off the L1 cache
- The performance of the system and application processes was slowed
How Important Is A Memory Cache?

• One day, on an operating systems project
  – Someone rewrote the processor startup code
  – They inadvertently turned off the L1 cache
• The performance of the system and application processes was slowed
• Guess how much faster the system ran with the L1 cache enabled
How Important Is A Memory Cache?

- One day, on an operating systems project
  - Someone rewrote the processor startup code
  - They inadvertently turned off the L1 cache
- The performance of the system and application processes was slowed
- Guess how much faster the system ran with the L1 cache enabled

With the L1 cache enabled, performance was 15 times faster!
Summary

- Caching is fundamental optimization technique
- Cache intercepts requests, automatically stores values, and answers requests quickly, whenever possible
- Caching can be used with both physical and virtual memory addresses
- Memory cache uses hierarchy
  - L1 onboard processor
  - L2 between processor and memory
  - L3 built into memory
Summary (continued)

- Two basic technologies used for memory cache
  - Direct mapped
  - Set associative
- Fully associative cache generalizes set associative approach
Questions?
Module XIII

Virtual Memory Technologies
And
Virtual Addressing
What Is Virtual Memory?

• Broad concept with lots of variants

• General idea
  – Hide the details of the underlying physical memory
  – Provide a view of memory that is more convenient to a programmer

• Goal is to allow physical memory and addressing to be structured in a way that is optimal for hardware while providing an interface that is optimal for software
A Trivial Example: Byte Addressing

- Architecture uses byte addresses
- Underlying physical memory uses word addresses
- Memory controller translates automatically
- Fits our definition of *virtual memory*
Virtual Memory Terminology

• **Memory Management Unit (MMU)**
  – Hardware unit
  – Provides translation between virtual and physical memory schemes

• **Virtual address**
  – Generated by processor (either instruction fetch or data fetch)
  – Translated into corresponding physical address by MMU

• **Physical address**
  – Used by underlying hardware
  – May be completely hidden from programmer
Virtual Memory Terminology (continued)

- **Virtual address space**
  - Set of all possible virtual addresses
  - Can be larger or smaller than physical memory
  - Each process may have its own virtual space
- **Virtual memory system**
  - All of the above
A Basic Example: Multiple Physical Memories

- Most computers have more than one physical memory module
- Each physical memory module
  - Offers addresses zero through \( N-1 \) for some \( N \)
  - May use an arbitrary memory technology (e.g., SRAM or DRAM)
- Virtual memory system can provide uniform address space for all physical memories
A Note About Banks And Modules

- Concepts are similar
- **Bank**
  - Generally refers to physical memory
  - Used when identical memory modules are replicated
- **Module**
  - More generic term often used with virtual memory systems
  - Preferred when heterogeneous memory units are combined
Illustration Of Hardware For
Two Dissimilar Memory Modules
Virtual Addressing For Multiple Modules

- Typical scheme: processor has a single virtual address space
- Address space covers all memory modules
- MMU translates from virtual space to underlying physical memories
- Example
  - Two physical memories with 1GB each (0x40000000) bytes
  - Virtual addresses 0 through 0x3fffffff correspond to memory 1
  - Virtual addresses 0x40000000 through 0x7fffffff correspond to memory 2
Illustration Of Virtual Addressing

- Notes
  - $0x40000000$ is 1 gigabyte or 1073741824 bytes
  - For identical modules, these are called memory banks
Address Translation

- Performed by MMU
- Also called *address mapping*
- For our example
  - To determine which physical memory, test if address is 0x40000000 or above
  - Both memory modules use addresses 0 through 0x3fffffff
  - Subtract 0x40000000 from address when forwarding a request to memory 2
Algorithm To Perform The Example Address Translation

Receive a virtual memory request from processor;
Let V be the address in the request;
if ( V >= 0 through 0x40000000 ) {
    V2 = V - 0x40000000;
    Pass the modified request (address V2) to memory 2;
} else {
    Pass the unmodified request (address V) to memory 1;
}
Avoiding Arithmetic Calculation

• Subtraction is relatively expensive

• To optimize, think binary
  – Always divide the virtual address space along boundaries that correspond to powers of two

• Virtual address can be divided into groups of bits that
  – Choose among underlying physical memories
  – Specify an address in the physical memory

• Note: selecting bits in hardware merely requires running wires (no gates and no computation)
Addresses above 0x3fffffff are the same as the previous set except for high-order bit.

Hardware uses the high-order bit to select a physical memory module.
Address Space Continuity

- **Contiguous address space**
  - All locations correspond to physical memory
  - Inflexible: requires all memory sockets to be populated

- **Discontiguous address space**
  - One or more blocks of address space do not correspond to physical memory
  - Called *hole*
  - *Fetch* or *store* to any address in a hole causes an error
  - Flexible: allows owner to decide how much memory to install
Illustration Of Discontiguous Address Space

Address

N
memory 2

N/2
memory 1

N/2−1

0

Hole (not present)

Hole (not present)
Programming And Discontinuities

- Consider a program running in an address space that has holes
- If the program attempts to store or fetch an address that corresponds to a hole, an error results
- For most systems, holes are only relevant to operating systems programmers
- For an embedded system, application programmer may need to avoid holes
Some Motivations For Virtual Memory

- Hardware perspective
  - Allow multiple memory modules
  - Provide homogeneous integration
- Software prospective
  - Programmer convenience
  - Support for multiprogramming and protection
Multiple Virtual Spaces And Multiprogramming

- Operating system allows multiple application programs to run concurrently
- To prevent one application from interfering with another
  - Each application runs as a separate process
  - Each process has its own virtual address space
- Operating system arranges for MMU to translate a given process’s addresses into the correct physical memory address
One Way To Map Four Virtual Spaces

![Diagram showing mapping of four virtual spaces to physical memory.]

- Virtual space 1 maps to physical memory from 0 to 3N/4.
- Virtual space 2 maps to physical memory from 3N/4 to N/2.
- Virtual space 3 maps to physical memory from N/2 to N/4.
- Virtual space 4 maps to physical memory from N/4 to 0.
Dynamic Address Space Creation

- Note: MMU translates each virtual address to a physical address
- The MMU configuration can be changed at any time
- Typically
  - Access to MMU restricted to operating system
  - When operating system runs, no mapping is performed
  - Processor only changes to virtual memory mode when running an application
Example Technologies Used For Address Space Creation

- Base-bound registers
- Segmentation
- Demand paging
Base-Bound Registers

- Requires two special hardware registers (part of the MMU)
- \textit{Base} register specifies starting address
- \textit{Bound} register specifies size of address space
- Values changed by operating system
  - Set before application runs
  - Changed by operating system when switching to another application
- Was once popular, but no longer used
Illustration Of Base-Bound Registers

- Each process’s address space is mapped to a region of memory
Protection Using Base-Bound Technology

- Key for systems that run multiple applications concurrently
- Each application is allocated a separate area of physical memory
- Operating system sets base-bound registers before application runs
- MMU hardware checks each memory reference
- Reference to any address outside the valid range results in an error
- Prevents an application from snooping or changing another application’s memory
Segmentation

- Alternative to base-bound
- Provides fine-granularity mapping
  - Divides program into segments (typical segment corresponds to one procedure)
  - Maps each segment to physical memory
- Key idea
  - Segment is only placed in physical memory when needed
  - When segment is no longer needed, OS moves it to disk
Problems With Segmentation

- Need hardware support to make moving segments efficient
- Two choices
  - Variable-size segments cause memory \textit{fragmentation}
  - Fixed-size segments may be too small or too large
- Neither choice works well
- Consequence: segmentation is seldom used
Demand Paging

- Alternative to segmentation and base-bound
- Currently, the most popular virtual memory technology
- Divides program into fixed-size pieces called *pages*
- No attempt is made to align page boundaries with functions, objects, or large data structures
- Typical page size 4K bytes
- Only some pages of a given application are in memory at any time; others are kept on disk and fetched when needed
- Allows the physical memory allocated to a process to change over time
Demand Paging Support

- Hardware is needed to handle address mapping and detect missing pages
- Software is needed to move pages between external store and physical memory
Paging Hardware

• Part of MMU
• Intercepts each memory reference
• If referenced page is present in memory, translate address and perform the operation
• If referenced page not present in memory, generate a page fault (i.e., an error condition)
• Record the details and allow operating system to handle the fault
Demand Paging Software

- Part of the operating system
- Works closely with hardware
- Responsible for overall memory management
- Determines which pages of each application to keep in memory and which to keep on disk
- Records location of all pages
- Fetches pages on demand (when an application references an address that is not in memory)
- Configures the MMU
Page Replacement

- When a computer starts
  - Applications run and reference pages
  - Each referenced page is placed in physical memory
- Eventually
  - Memory is completely full
  - An existing page must be written to disk before memory can be used for new page
- Choosing a page to expel is known as *page replacement*
- Optimization: replace a page that will not be needed soon
Paging Terminology

- *Page*: fixed-size piece of program’s address space
- *Frame*: slot in memory exactly the size of one page
- *Resident*: a page that is currently in memory
- *Resident set*: pages from a given application that are present in memory
Paging Data Structure

- Known as a *page table*
- One page table per process
- Created and managed by the operating system
- Used by the MMU when translating an address
- Think of a page table as a one-dimensional array
  - Indexed by page number
  - Entry stores a pointer to the location of the page in memory (or a bit that indicates the page is currently on disk)
- Each page table entry points to a frame in memory or null
Address Translation With A Page Table

- Given virtual address $V$, find underlying memory address $P$
- Three conceptual steps
  - Determine the number of the page on which address $V$ lies
  - Use the page number as an index into the process’s page table to find the starting address of a frame in memory that contains the specified byte
  - Determine how far into the page address $V$ lies, and convert to a position in the frame in memory
Mathematical View Of Address Translation

- Page number computed by dividing the virtual address by the number of bytes per page, $K$

$$N = \left\lfloor \frac{V}{K} \right\rfloor$$

- Offset within the page, $O$, can be computed as the remainder

$$O = V \mod K$$
Mathematical View Of Address Translation
(continued)

- Use $N$ and $O$ to translate virtual address $V$ to real memory address $A$

$$A = \text{pagetable}[N] + O$$
Using Powers Of Two

- Cannot afford division or remainder operation for each memory reference
- Think binary, and use powers of two to eliminate arithmetic
- Let number of bytes per page be $2^k$
  - Offset $O$ is given by low-order $k$ bits
  - Page number is given by remaining (high-order) bits
- Computation is:

$$P = \text{pagetable}[\text{high_order_bits}(V)] \text{ or } \text{low_order_bits}(V)$$
Illustration Of Translation With MMU Hardware

- Typical paging system uses 12 bits of offset (4 Kbytes per page)
Presence, Use, And Modified Bits

- Found in most paging hardware
- One set for each page table entry
- Shared by hardware and software
- Purpose of the bits

<table>
<thead>
<tr>
<th>Control Bit</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presence bit</td>
<td>Tested by hardware to determine whether page is currently present in memory</td>
</tr>
<tr>
<td>Use bit</td>
<td>Set by hardware whenever page is referenced</td>
</tr>
<tr>
<td>Modified bit</td>
<td>Set by hardware whenever page is changed</td>
</tr>
</tbody>
</table>
Page Table Storage

- In some systems, the MMU holds page tables
- Most systems place the page tables in memory
- Interesting idea
  - Page table entry only needs to store the address of a frame
  - Each frame is a power of two bytes, so the starting address will have zero in the $k$ low-order bits
  - Instead of storing zeros, store the presence, use, and modify bits
  - Allows page table entry to remain aligned on word boundary
Where Are Page Tables In Memory?

• Typical position: above the operating system

![Diagram showing memory, operating system, page tables, and frame storage]

• Consequence: only part of memory is divided into frames that hold applications
The Importance Of Efficiency

• When paging is used, an address translation must occur
  – For each instruction fetch
  – For each data reference

• Translation can become a bottleneck, and it must be optimized

• Note: early virtual memory systems that did not have special hardware for address translation were unusable
Translation Lookaside Buffer (TLB)

- Hardware mechanism used to optimize address translation
- Employs a form of Content Addressable Memory (CAM)
- Hardware unit stores pairs of
  
  (virtual address, physical address)

- If pair is in TLB
  - Virtual address can be translated without a page table reference
  - MMU returns the translation much faster than a page table lookup
In Practice

- A virtual memory system without TLB is unacceptable
- The TLB approach works well because application programs tend to reference a given page many times
- Principle known as *locality of reference*
Consequence For Programmers

- Programmer can optimize program performance by accommodating the paging system
- Examples
  - Group related data items on same page
  - Reference arrays in an order that accesses contiguous memory locations
Array Reference

- Consider an array stored in *row-major order*

  row 0  row 1  row 2  row 3  row 4  row 5  ...  row N

- Location of A[i, j] given by

  \[ \text{location}(A) + i \times Q + j \]

  where \( Q \) is number of bytes per row

- Accessing items by row makes repeated accesses to the same page before moving on
Programming To Optimize Array Access

- Optimal

```plaintext
for i = 1 to N {
    for j = 1 to M {
        A [ i, j ] = 0;
    }
}
```

- Nonoptimal

```plaintext
for j = 1 to M {
    for i = 1 to N {
        A [ i, j ] = 0;
    }
}
```
Virtual Memory Caching

- Can build a system that caches
  - Physical memory address and contents
  - Virtual memory address and contents

- Notes
  - If MMU is off-chip, L1 cache must use virtual addresses
  - Key point: multiple processes have separate address spaces, but each uses the same set of virtual addresses
Handling Overlapping Virtual Addresses

• Each application process uses virtual addresses 0 through \( N \)
• System must ensure that an application does not receive data from another application’s memory
• Two possible approaches
  – OS performs cache *flush* operation when changing applications
  – Cache includes disambiguating tag with each entry (i.e., a process ID)
Illustration Of ID Register

- Assign each running application a unique ID (e.g., use a process ID)
- Operating system places ID in a special hardware register when an application runs
- Memory system attaches ID to each address in the cache
Summary

- Virtual memory systems present illusion to processor and programs
- Many virtual memory architectures are possible
- Examples include
  - Hiding details of word addressing
  - Create uniform address space that spans multiple memories
  - Incorporate heterogeneous memory technologies into single address space
Summary
(continued)

• Virtual memory offers
  – Convenience for programmer
  – Support for multiprogramming
  – Protection

• Three technologies have been used for virtual memory
  – Base-bound registers
  – Segmentation
  – Demand paging (currently popular)
• Demand paging
  – The chief technology used in most systems
  – Combination of hardware and software
  – Uses page tables to map virtual addresses to physical addresses
  – High-speed lookup mechanism known as TLB makes demand paging practical

• Caching virtual addresses requires either
  – Flushing the cache during context switch
  – Using an ID to disambiguate
Questions?
Module XIV

Input / Output
Concepts And Terminology
I/O Devices

- Third major component of computer system
- Wide range of types
  - Keyboards and mice
  - Monitors and displays
  - Hard disks
  - Solid state disks
  - Printers
  - Cameras
  - Audio speakers
  - Sensors and actuators
Conceptual Properties Of An I/O Device

- Operates independent of processor
- May have separate power supply
- Digital signals used for control
- Trivial example: panel lights
Illustration Of Modern Interface Controller

- Controller placed at each end of physical connection
- Allows arbitrary voltage and signals to be used
Two Types Of Interfaces

• Serial interface
  – Single signal wire (also need ground); one bit at a time
  – Less complex hardware with lower cost

• Parallel interface
  – Many wires; each wire carries one bit at any time
  – Width is number of wires
  – Complex hardware with higher cost
  – Theoretically faster than serial
  – Practical limitation: at high data rates, close parallel wires have potential for interference
Clock Rates And Coordination

- Logic on each side of a connection has its own clock
  - Processor
  - I/O device
- Communication must be designed so they can coordinate
- We say signals are *self-clocking* if the receiver can determine the boundary of bits without knowing about the sender’s clock
Duplex Terminology

• Full-duplex
  – Simultaneous, bidirectional transfer
  – Example: disk drive supports simultaneous read and write operations

• Half-duplex
  – Transfer in one direction at a time
  – Interfaces must negotiate access before transmitting
  – Example: processor can read or write to a disk, but can only perform one operation at a time
Measures Of I/O Performance

• **Latency**
  – Measure of the time required to perform a transfer
  – Latencies of input and output may differ

• **Throughput**
  – Measure of the amount of data that can be transferred per unit time
  – Informally called *speed*
Data Multiplexing

- Fundamental idea
- Arises from hardware limits on parallelism (pins or wires)
- Allows sharing

Multiplexor
  - Accepts input from many sources
  - Sends each item along with an ID

Demultiplexor
  - Receives ID along with transmission
  - Uses ID to reassemble items correctly
Illustration Of Multiplexing

- Example: 64 bits of data multiplexed over 16-bit path

- Hardware iterates, transferring one chunk at a time
Multiple Devices Per External Interface

- Cannot afford to have a separate physical interconnect per device
  - Too many physical wires
  - Not enough pins on a processor chip
  - Interface hardware adds economic cost
- Solution is sharing
  - Allow multiple devices to use a given interconnection
  - Known as a bus
  - Discussed in the next section
Questions?
Module XV

Buses
And
Bus Architecture
Definition Of A Bus

- Digital interconnection mechanism
- Allows two or more functional units to transfer data
- Typical use: connect processor to
  - Memory
  - I/O devices
- Design can be
  - Proprietary (owned by one company)
  - Open standard (available to many companies)
Illustration Of A Bus

- Double-headed arrow often used to denote a bus
- Each component connects to the bus
- Example

- Bus may have many parallel wires (e.g., 64)
Sharing

- Most buses shared by multiple devices
- Need an access protocol
  - Determines which device can use the bus at any time
  - All attached devices follow the protocol
- Note: it is possible to have multiple buses in one computer
Characteristics Of A Bus

- May support parallel data transfer
  - Hardware can transfer multiple bits at the same time
  - Typical width is 32 or 64 bits
- Essentially passive
  - Bus does not contain many electronic components
  - Attached devices handle communication
- Conceptual view: think of a bus as a set of wires
- Bus may have *arbiter* that manages sharing
Implementation Of A Bus

• Several possibilities
• Can consist of
  – A cable with multiple wires
  – Traces on a circuit board
• Usually, a bus has sockets into which devices plug
Illustration Of Bus On A PC Motherboard

Area on mother board for the processor, memory, and other units.

Sockets placed near the edge of the board.

Bus formed from parallel wires.
Side View Of Circuit Board And Corresponding Sockets

- Each I/O device on a circuit board
- I/O devices plug into sockets on the mother board
Bus Interface

- Access protocol is nontrivial
- Controller circuitry is required
- Circuitry part of each I/O device
- Good news: you don’t have to understand access circuits
Conceptual Bus Functions

- Each device attached to a bus is assigned an address (in practice, there may be a small set of addresses)
- Bus allows processor to specify
  - Address for the device
  - Data to transfer
  - Control (e.g., to specify input or output)
- We can think of a bus as having a separate group of wires (lines) for each of the above functions
• Early bus designs did indeed use separate wires

• To lower cost, many bus designs now arrange to multiplex address and data information over the same wires (in a request, use the wires to send an address; in a response, use the same wires to send data)

• *Serial bus* multiplexes all communication over one wire
Bus Operations

- Bus hardware only supports two operations
  - *Fetch* (also called *read*)
  - *Store* (also called *write*)
- Access paradigm is known as the *fetch-store paradigm*
- Obvious for memory access
- Surprise: all device interaction, including communication with video cameras, speakers, and microphones, must be performed using the fetch-store paradigm
Fetch-Store Over A Bus

- **Fetch**
  - Place an address on the address lines
  - Use control line to signal *fetch* operation
  - Wait for control line to indicate *operation complete*
  - Extract data item from the data lines

- **Store**
  - Place an address on the address lines and a data item on the data lines
  - Use control line to signal *store* operation
  - Wait for control line to indicate *operation complete*
Width Of A Bus

- *Width* refers to the number of parallel data lines
- Larger width
  - Advantage: higher performance
  - Disadvantages: higher cost and more pins
- Smaller width
  - Advantages: lower cost and fewer pins
  - Disadvantage: lower performance
- Typical designs use multiplexing to lower cost
- Extreme case: serial bus has a width of one
Memory Bus

- Bus provides path between processor and memory
- Memory hardware includes bus controller

- Each memory module responds to a set of addresses
Steps A Memory Module Takes

Let R be the range of addresses assigned to this memory module

Repeat forever {
    Monitor the bus until a request appears;
    if (the request specifies an address in R) {
        respond to the request
    } else {
        ignore the request
    }
}
Potential Errors On A Bus

- Address conflict
  - Two devices attempt to respond to a given address
- Unassigned address
  - No device responds to a given address
- Bus hardware detects the problems and raises an error condition (sometimes called a bus error)
- Unix reports bus error to an application that attempts to dereference an invalid pointer
Address Configuration And Sockets

- Three options for address configuration
  - Configure each device before attaching it to a bus
  - Arrange sockets so that wiring limits each socket to a range of addresses
  - Design bus hardware that configures addresses when system boots (or when a device attaches)
- Socket wiring is typically used for memory (user can plug in additional modules without configuring the hardware)
- Automatic configuration is usually used for I/O devices
Example Of Using Fetch-Store

- Imagine we are designing a device with LEDs used as status indicators
- Assume the hardware
  - Provides sixteen separate LEDs
  - Connects to 32-bit bus
- Desired functions are
  - Turn the display unit on
  - Turn the display unit off
  - Set the brightness for the display unit
  - Turn the \( i^{th} \) LED on or off
**Example Of Meaning Assigned To Addresses**

- Device designer chooses semantics for *fetch* and *store*
- Example assignment

<table>
<thead>
<tr>
<th>Address</th>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000 – 10003</td>
<td>store</td>
<td>nonzero data value turns the display on, and a zero data value turns the display off</td>
</tr>
<tr>
<td>10000 – 10003</td>
<td>fetch</td>
<td>returns zero if display is currently off, and nonzero if display is currently on</td>
</tr>
<tr>
<td>10004 – 10007</td>
<td>store</td>
<td>Change brightness. Low-order four bits of the data value specify brightness value from zero (dim) through fifteen (bright)</td>
</tr>
<tr>
<td>10008 – 10011</td>
<td>store</td>
<td>The low order sixteen bits each control a status light; a zero bit sets the corresponding light off and a one bit sets the light on</td>
</tr>
</tbody>
</table>
if (address == 10000) {
    if (op == store) {
        if (data != 0) {
            turn_on_display;
        } else {
            turn_off_display;
        }
    } else { /* handle fetch */
        if (device is on) {
            send value 1 as data;
        } else {
            send value 0 as data;
        }
    }
}
Asymmetry

- *Fetch* and *store* operations on a bus
  - Mean “fetch data” and “store data” for a memory
  - May have other meanings for devices
  - Are often asymmetric for devices

- Consequences
  - For a device, *fetch* from location $N$ may not be related to *store* into location $N$
  - A device may define *fetch*, *store*, both, or neither for a given location
Unification Of Memory And Devices

- Single bus can attach
  - Multiple memories
  - Multiple devices
- Bus address space includes all units
Illustration Of Single Bus

- Bus connects processor to
  - Multiple physical memory units
  - Multiple I/O devices
- Single address space includes all devices and memories
Example Address Assignment

- Example includes
  - Two memories of 1 megabyte each
  - Two devices that use 12 bytes of address space

<table>
<thead>
<tr>
<th>Device</th>
<th>Address Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory 1</td>
<td>0x000000 through 0x0ffffff</td>
</tr>
<tr>
<td>Memory 2</td>
<td>0x100000 through 0x1ffffff</td>
</tr>
<tr>
<td>Device 1</td>
<td>0x200000 through 0x20000b</td>
</tr>
<tr>
<td>Device 2</td>
<td>0x20000c through 0x200017</td>
</tr>
</tbody>
</table>

- Note: memories occupy *many* addresses; devices occupy *few* addresses
• We use the term *address map* to describe the set of assignments
An Address Map Example That Shows Holes

The diagram illustrates an address map with a series of addresses available for devices and memory. There are holes (not available) in the address space:

- From 0xdfff to 0xbfff
- From 0x7fff to 0x3fff
- At 0x0000

These holes indicate areas of memory that are not accessible or available for allocation.
Address Maps

- In a typical system
  - A device only requires a few bytes of address space
  - Designers leave room for many devices
- Consequence: address space available for devices is sparsely populated
Example Code To Manipulate A Bus

- Software such as an OS that has access to the bus address space can fetch or store to a device
- Example code

```c
int *p; /* declare p to be a pointer to an integer */
p = (int *)10000; /* set pointer to address 10000 */
*p = 1; /* store 1 in addresses 10000 – 10003 */
```
**Bridge**

- Hardware mechanism
- Used to connect two buses
- Maps range of addresses from one bus to the other
- Forwards operations and replies from one bus to the other
- Especially useful for adding an auxiliary bus
Illustration Of Bridge Mapping Addresses

bridge supplies the mapping

not mapped
Switching Fabric

- Alternative to bus
- Connects multiple devices
- Sender supplies data and destination device
- Fabric delivers data to specified destination
• Solid dot indicates a connection
Summary

- Bus is fundamental mechanism that interconnects
  - Processor
  - Memory
  - I/O devices
- Bus uses fetch-store paradigm for all communication
- Each unit assigned set of addresses in bus address space
- Bus address space can contain holes
- Bridge maps subset of addresses on one bus to another bus
Summary
(continued)

- Programmer uses conventional memory address mechanism to communicate over a bus
- Switching fabric is alternative to bus that allows parallelism
Questions?
Module XVI

Programmed And Interrupt-driven I/O
Two Basic Approaches To I/O

- **Programmed I/O**
  - A terrible name
  - Also called *polled I/O*

- **Interrupt-driven I/O**
  - Another poor naming choice
  - Software actually drives I/O
Programmed I/O

- Used in early computers and in the smallest embedded systems
- Device has no intelligence (called *dumb*)
- CPU does all the work
- Processor
  - Is much faster than device
  - Starts operation on device
  - Waits for device to complete
Waiting For A Device To Complete

- Basic technique used with programmed I/O is *polling*
- To wait for an operation to complete, a processor
  - Executes a loop that repeatedly requests status from device
  - Allows the loop to continue until device indicates “ready”
- Also called *busy waiting*
Example Of Polling (Imaginary Printer)

• Typical sequence of steps
  – Test to see if the printer is powered on
  – Cause the printer to load a blank sheet of paper
  – Poll to determine when the paper has been loaded
  – Specify data in memory that tells what to print
  – Poll to wait for the printer to load the data
  – Cause the printer to start spraying a band of ink
  – Poll to determine when the ink mechanism finishes
  – Cause the printer to advance the paper to the next band
  – Poll to determine when the paper has advanced
  – Repeat the above six steps for each band to be printed
  – Cause the printer to eject the page
  – Poll to determine when the page has been ejected
Example Specification Of Addresses Used For Device Polling

- Each device defines a set of addresses and meanings for *fetch* and *store* operations
- An interface for our imaginary printer

<table>
<thead>
<tr>
<th>Addresses</th>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 – 3</td>
<td>fetch</td>
<td>Nonzero if the printer is powered on</td>
</tr>
<tr>
<td>4 – 7</td>
<td>store</td>
<td>Nonzero starts loading a sheet of paper</td>
</tr>
<tr>
<td>8 – 11</td>
<td>store</td>
<td>Memory address of data to print</td>
</tr>
<tr>
<td>12 – 15</td>
<td>store</td>
<td>Nonzero causes printer to pick up address</td>
</tr>
<tr>
<td>16 – 19</td>
<td>store</td>
<td>Start the inkjet spraying current band</td>
</tr>
<tr>
<td>20 – 23</td>
<td>store</td>
<td>Nonzero advances paper to the next band</td>
</tr>
<tr>
<td>24 – 27</td>
<td>fetch</td>
<td>Busy: nonzero when device is busy</td>
</tr>
<tr>
<td>28 – 31</td>
<td>fetch</td>
<td>CMYK ink levels in four octets</td>
</tr>
</tbody>
</table>

- Addresses shown are *relative*
- We will imagine that the interface starts at address 0x110000
Example C Code For Device Polling

```c
int *p; /* Pointer to the device address area */
p = (int *)0x110000; /* Initialize pointer to device address */
if (*p == 0) /* Test if printer is powered on */
    error("printer not on");
*(p+1) = 1; /* Start loading paper */
while (*(p+6) != 0) /* Poll to wait for the load to complete */);
*(p+2) = &mydata; /* Specify the location of data in memory */
*(p+3) = 1; /* Cause printer to pick up data */
while (*(p+6) != 0) /* Poll to wait for printer to complete loading data */;
*(p+4) = 1; /* Start inkjet spraying */
while (*(p+6) != 0) /* Poll to wait for the inkjet to finish */;
*(p+5) = 1; /* Advance the paper to the next band */
while (*(p+6) != 0) /* Poll to wait for the paper advance to complete*/;
```

• Note: code does not contain any infinite loops!
Terminology

- Set of addresses a device defines are known as its *Control and Status Registers (CSRs)*
- CSRs are used to transfer data and control the device
- The hardware designer chooses whether a given CSR responds to
  - A *fetch* operation
  - A *store* operation
  - Both
- In many cases, individual CSR bits are assigned meanings
- In C, a *struct* can be used to define CSRs
Polling Code Rewritten To Use A Struct (Part 1)

```c
struct csr {
    int csr_power;   /* Is printer powered on? */
    int csr_load;    /* Load a sheet of paper */
    int csr_addr;    /* Specify address of data to print */
    int csr_getdata; /* Upload data from memory */
    int csr_spray;   /* Start inkjet spraying */
    int csr_advance; /* Advance paper to next band */
    int csr_dev_busy; /* Nonzero => device busy */
    int csr_levels;  /* CMYK Ink levels in 4 bytes */
};
struct csr *p;   /* Pointer to the device address area */
p = (struct csr *)0x110000; /* Set p to device address */
if (p->csr_power == 0) {
    error("printer not on");
}
p->csr_load = 1; /* Start loading paper */
while (p->csr_dev_busy) {
    /* Poll to wait for the load to complete */
}
```

Copyright © 2016 by Douglas Comer. All rights reserved
Polling Code Rewritten To Use A Struct (Part 2)

```c
p->csr_addr = &mydata /* Specify the location of data in memory */
p->csr_getdata = 1; /* Cause printer to pick up data */
while (p->csr_dev_busy) /* Poll to wait for printer to complete loading data */
    ;
p->csr_spray = 1; /* Start the inkjet spraying */
while (p->csr_dev_busy) /* Poll to wait for the inkjet to finish */
    ;
p->csr_ = 1; /* Advance the paper to the next band */
while (p->csr_dev_busy) /* Poll to wait for the paper advance to complete*/
    ;
```
Interrupt-Driven I/O

- Motivation: increase performance by eliminating polling loops
- Technique
  - Add special hardware to processor and devices
  - Allow processor to start operation on a device
  - Arrange for device to interrupt the processor when the operation completes
Interrupt Mechanism

- Processor hardware
  - Saves current instruction pointer
  - Jumps to code for the interrupt
  - Resumes executing the application when the code executes a *return from interrupt*
Programming Paradigms

- Polling uses a *synchronous* paradigm
  - Code is sequential
  - Programmer includes device polling for each I/O operation
- Interrupts use an *asynchronous* paradigm
  - Device temporarily interrupts processor
  - Processor services device and returns to computation in progress
  - Programmer creates separate piece of software to handle each type of interrupt
Fetch-Execute Cycle With Interrupts

Repeat forever {

Test: if any device has requested interrupt, handle the interrupt and then continue with the next iteration of the loop.

Fetch: access the next step of the program from the location in which the program has been stored.

Execute: Perform the step of the program.

} 

• Note: interrupt appears to occur between two instructions
Saving And Restoring State

- Entire state of computation must be saved when interrupt occurs
  - Values in registers
  - Program counter
  - Condition code

- Hardware usually saves and restores a few items; interrupt code must save and restore the rest
Vectored Interrupts

- Technique used to optimize interrupt handling
- OS maintains, $V$, an array of pointers to interrupt code
  - Called an *interrupt vector*
  - Informs bus hardware of the location of $V$
- Each device is assigned a number from 0 through $K-1$
- Device specifies its number, $i$, when interrupting
- Hardware (or in some architectures, the OS) branches to interrupt code at address $V[i]$
Illustration Of Interrupt Vectors

Interrupt vectors in memory

0
1
2
3

Device 0
Device 1
Device 2
Device 3

Handler for Device 0
Handler for Device 1
Handler for Device 2
Handler for Device 3
Interrupt Vector Initialization

- Processor boots with interrupts *disabled*
- OS
  - Keeps interrupts disabled during initialization
  - Fills in interrupt vector with pointers to interrupt code for each device
- Once all interrupt table entries have been initialized, OS enables interrupts, which allows I/O to proceed
Preventing Interrupt Code From Being Interrupted

- Fact: multiple devices can request an interrupt simultaneously
- To prevent confusion, an OS should handle one device before another interrupts
- Typical technique: hardware disables further interrupts while an interrupt is being handled
Multiple Interrupt Levels

- Simplest processors: only one interrupt at a time
- Advanced processors: devices assigned a priority, and higher priority devices can interrupt lower level interrupt code
- Typically a few priority levels (e.g., 7)
- Rule: at any given time, at most one device can be interrupting at each priority level
- Note: the lowest priority (usually zero) means no interrupt is occurring (i.e., an application program is executing)
Interrupt Vector Assignments

- Each device must be assigned an interrupt vector ID
- The OS must know which device has been assigned which interrupt ID
- Assignments can be
  - Manual (only used on small embedded systems)
  - Automated (more flexible; used on most systems)
Dynamic Bus Connections And Pluggable Devices

- Some bus technologies allow devices to be connected or disconnected at run-time
- Example: Universal Serial Bus (USB)
- Computer contains a USB hub device that has a fixed interrupt vector
- When a new device is attached, the hub generates an interrupt, and the interrupt code loads additional software for the device into the OS
Optimizations Used With Interrupt-Driven I/O

- Provide higher data transfer rates
- Offload CPU
- Three basic types
  - Direct Memory Access (DMA)
  - Buffer Chaining
  - Operation Chaining
Direct Memory Access (DMA)

- Widely used
- Works well for high-speed I/O and streaming
- Requires smart device that can move data across the bus to/from memory without using processor
- Example: Wi-Fi network interface can read an entire packet and place the packet in a specified buffer in memory
- Basic idea
  - CPU tells device location of buffer
  - Device fills buffer and then interrupts
Buffer Chaining

- Extends DMA to handle multiple transfers on one command
- Device given linked list of buffers
- Device hardware uses next buffer on list automatically
Scatter Read And Gather Write

- Special cases of buffer chaining
- Large data transfer formed from separate blocks in memory
- Example: to write a network packet, combine packet header from buffer 1, encryption header from buffer 2, and packet data from buffer 3
- Eliminates application program from copying data into single large buffer
Operation Chaining

- Further extension of DMA
- Allows sequence of *read*, *write*, and *control* operations
- Processor passes a list of commands to the device
- Device carries out successive commands automatically
- Illustration of disk reads and writes with operation chaining
Summary

- Devices can use
  - Programmed I/O
  - Interrupt-driven I/O

- Interrupts
  - Allow processor to continue running while waiting for I/O
  - Use vector (usually in memory)
  - Occur “between” instructions in fetch-execute cycle
Summary
(continued)

- Multi-level interrupts handle high-speed and low-speed devices on same bus
- Smart device has some processing power built into the device
- Optimizations for high-speed devices include
  - Direct Memory Access (DMA)
  - Buffer chaining
  - Operation chaining
Questions?
Module XVII

A Programmer’s View Of I/O And Buffering
Device Driver

- Piece of software
- Responsible for communicating with specific device
- Usually part of operating system
- Performs basic functions
  - Initializes the device
  - Manipulates device’s CSRs to start operations when I/O is needed
  - Handles interrupts from device
Why A Device Driver?

- Encapsulation and hiding: details of device hidden from application software
- Device independent applications: application code does not contain the details for any specific device(s)
Three Conceptual Parts Of A Device Driver

- **Lower half**
  - Handler code that is invoked when the device interrupts
  - Communicates directly with device (e.g., to reset hardware)

- **Upper half**
  - Set of functions that are invoked by applications
  - Allows application to request I/O operations

- **Shared variables**
  - Used by both halves to coordinate
  - Contains input and output buffers
Illustration Of Device Driver Organization

- **applications programs**
  - upper half
    - invoked by applications
  - shared variables
  - lower half
    - invoked by interrupts
- **device hardware**
Types Of Devices

• Character-oriented
  – Transfer one byte at a time
  – Examples
    * Keyboard
    * Mouse

• Block-oriented
  – Transfer block of data at a time
  – Examples
    * Disk
    * Network interface
Example Flow In A Network Device Driver

Steps Taken

1. The application sends data over the Internet
2. Protocol software passes a packet to the driver
3. The driver stores the outgoing packet in the shared variables
4. The upper half specifies the packet location and starts the device
5. The upper half returns to the protocol module
6. The protocol software returns to the application
7. The device interrupts and the lower half of the driver executes
8. The lower half removes the copy of the packet from the variables
Queued Output Operations

- Used by most device drivers
- Shared variable area contains queue of requests
- Upper half places request on queue
- Lower half moves to next request on queue when an operation completes
- If device supports operation chaining, upper half can add new items to the queue while the device is processing (coordination required)
Illustration Of An Output Request Queue

- Queue is shared among both halves
- Driver software is designed so that each half ensures the other half will not examine or change the queue at the same time
Managing An Output Queue

• At startup, initialize the queue to empty
• When application performs *write*, upper half
  – Deposits data item in queue
  – Forces the device to interrupt
  – Returns to application
• When interrupt occurs, lower half
  – Extracts the next item from the queue and starts output, if queue is not empty
  – Allows the device to remain idle, if the queue is empty
  – Returns from interrupt
Managing An Input Queue

• At startup, initialize the queue to empty and start the device
• When application performs *read*, upper half
  – Extracts and returns the next item, if queue is nonempty
  – Blocks application if input queue is empty
• When an interrupt occurs, lower half
  – Starts another input operation, if the queue is not full
  – Allows the application to run, if an application is blocked waiting for input
  – Returns from interrupt
Mutual Exclusion

- Needed because interrupts occur asynchronously and multiple applications can attempt I/O on a given device at the same time.
- Guarantees only one operation will be performed at any time.
- Device drivers handle mutual exclusion.
I/O Interface For Applications

- Few programmers write device drivers
- Instead of dealing directly with devices, most programmers use high-level abstractions
  - Files instead of disks
  - Windows instead of display screens
- Typical application invokes *run-time library* functions to perform I/O
- Chief advantage: I/O hardware and/or device drivers can be changed without changing applications
Programming Interfaces For An I/O Library

- Interfaces can differ dramatically
Example Of Two Interfaces

- **UNIX library functions**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf</td>
<td>Generate formatted output for a set of variables</td>
</tr>
<tr>
<td>fprintf</td>
<td>Generate formatted output for a specific file</td>
</tr>
<tr>
<td>scanf</td>
<td>Read formatted data into a set of variables</td>
</tr>
</tbody>
</table>

- **UNIX system calls**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>open</td>
<td>Prepare a device for use (e.g., power up)</td>
</tr>
<tr>
<td>read</td>
<td>Transfer data from the device to the application</td>
</tr>
<tr>
<td>write</td>
<td>Transfer data from the application to the device</td>
</tr>
<tr>
<td>close</td>
<td>Terminate use of the device</td>
</tr>
<tr>
<td>seek</td>
<td>Move to a new location of data on the device</td>
</tr>
<tr>
<td>ioctl</td>
<td>Misc. control functions (e.g., change volume)</td>
</tr>
</tbody>
</table>
Reducing The Cost Of I/O Operations

- Two principles
  - Cost of making a system call is much more expensive than the cost of making a conventional function call
  - The approach used to reduce system calls consists of transferring more data per call
Buffering

- Important optimization
- Widely used
- Usually automated and invisible to programmer
- Key idea: make large I/O transfers to driver
  - Accumulate large block of outgoing data before transfer
  - Transfer large block of incoming data and then extract individual items
Hiding Buffering From Programmers

- Typically performed with *library functions*
- Application
  - Uses functions in the library for all I/O
  - Transfers data in arbitrarily small amounts
- Library functions
  - Buffer data from applications
  - Transfer data to underlying system in large blocks
Example Functionality Used For Buffering

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup</td>
<td>Initialize input and/or output buffers</td>
</tr>
<tr>
<td>input</td>
<td>Perform an input operation</td>
</tr>
<tr>
<td>output</td>
<td>Perform an output operation</td>
</tr>
<tr>
<td>terminate</td>
<td>Discontinue use of the buffers</td>
</tr>
<tr>
<td>flush</td>
<td>Force contents of output buffer to be written</td>
</tr>
</tbody>
</table>

- Device driver in the operating system may also perform buffering to reduce number of transfers between the processor and the device
Using A Buffering Library For Output

• Setup function
  – Called to initialize buffer
  – May allocate buffer
  – Typical buffer sizes 8K to 128K bytes

• Output function
  – Called when application needs to emit data
  – Places data item in buffer
  – Only writes to I/O device when buffer is full

• Terminate function
  – Called when all data has been emitted
  – Forces remaining data in buffer to be written
Implementation Of Output Buffer Functions

Setup(N)

1. Allocate a buffer of N bytes.

2. Create a global pointer, p, and initialize p to the address of the first byte of the buffer.

Output(D)

1. Place data byte D in the buffer at the position given by pointer p, and move p to the next byte.

2. If the buffer is full, make a system call to write the contents of the entire buffer, and reset pointer p to the start of the buffer.
Implementation Of Output Buffer Functions (continued)

Terminate

1. If the buffer is not empty, make a system call to write the contents of the buffer prior to pointer p.

2. If the buffer was dynamically allocated, deallocate it.
Flush An Output Buffer

- Allows a programmer to force data in a buffer to be written

Motivation
- For batch programs: force data to disk
- For interactive programs: force data to be sent over a network (e.g., a single keystroke when using ssh)

When `flush` is called
- If buffer contains data, write data and reset buffer to empty
- If buffer is empty, `flush` has no effect
Implementation Using A Flush Function

Flush

1. If the buffer is currently empty, return to the caller without taking any action.
2. If the buffer is not currently empty, make a system call to write the contents of the buffer and set the global pointer p to the address of the first byte of the buffer.

Terminate

1. Call flush to ensure that any remaining data is written.
2. Deallocate the buffer.
**Buffering On Input**

**Setup(N)**
1. Allocate a buffer of N bytes.
2. Create a global pointer, p, and initialize p to indicate that the buffer is empty.

**Input(N)**
1. If the buffer is empty, make a system call to fill the entire buffer, and set pointer p to the start of the buffer.
2. Extract a byte, D, from the position in the buffer given by pointer p, move p to the next byte, and return D to the caller.

**Terminate**
1. If the buffer was dynamically allocated, deallocate it.
Analysis Of Buffering

• Implementation
  – Both input and output buffering are straightforward
  – Only a trivial amount of code is needed

• Effectiveness
  – Buffer of size $N$ reduces number of system calls by a factor of $N$
  – Example: when buffering character (byte) output, a buffer of only 8K bytes reduces system calls by a factor of 8192
Relation Between Buffering And Caching

- Concepts are closely related
- Chief difference
  - Caching is designed for random access
  - Buffering is designed for sequential access
Example: Unix I/O Functions That Buffer

- Standard I/O library in UNIX contains many functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>fopen</td>
<td>Set up a buffer</td>
</tr>
<tr>
<td>fgetc</td>
<td>Buffered input of one byte</td>
</tr>
<tr>
<td>fread</td>
<td>Buffered input of multiple bytes</td>
</tr>
<tr>
<td>fwrite</td>
<td>Buffered output of multiple bytes</td>
</tr>
<tr>
<td>fprintf</td>
<td>Buffered output of formatted data</td>
</tr>
<tr>
<td>fflush</td>
<td>Flush operation for buffered output</td>
</tr>
<tr>
<td>fclose</td>
<td>Terminate use of a buffer</td>
</tr>
</tbody>
</table>

- Each function uses buffers extensively
- Dramatically improves I/O performance
Summary

- Two aspects of I/O pertinent to programmers
  - Device interface important to systems programmers who write device drivers
  - Relative costs of I/O important to application programmers
- Device driver divided into three parts
  - Upper-half called by application
  - Lower-half handles device interrupts
  - Shared data area accessed by both halves
Summary
(continued)

- Buffering
  - Fundamental technique used to enhance performance
  - Useful with both input and output
- Buffer of size $N$ reduces system calls by a factor of $N$
Questions?
Module XVIII

Parallelism
Techniques Used To Increase Performance
Techniques Used To Increase Performance

- Software designers have many techniques available
  - Caching and buffering
  - Hashing and randomization
  - Better algorithms
  - Data placement and reordering data items during search
    . . . many more . . .
Techniques Used To Increase Performance

- Software designers have many techniques available
  - Caching and buffering
  - Hashing and randomization
  - Better algorithms
  - Data placement and reordering data items during search
    
    . . . *many more* . . .

- Hardware designers have two basic techniques
  - Parallelism
  - Pipelining
Parallelism

- Employs multiple copies of a hardware unit
- All copies can operate simultaneously
- General idea
  - Distribute data items among parallel hardware units
  - Gather (and possibly combine) results
- Occurs at many levels of architecture
- Term *parallel computer* applied when parallelism dominates the entire architecture
Characterizations Of Parallelism

- Microscopic vs. macroscopic
- Symmetric vs. asymmetric
- Fine-grain vs. coarse-grain
- Explicit vs. implicit
Microscopic Vs. Macroscopic Parallelism

- Virtually all computers have some parallelism
- *Microscopic parallelism* refers to parallel facilities in a single, small hardware unit
- *Macroscopic parallelism* refers to parallel facilities across major pieces of hardware
Examples Of Parallelism Scope

- **Microscopic**
  - Parallel hardware in an ALU
  - Parallel data transfer to/from physical memory or an I/O bus

- **Macroscopic**
  - Multiple identical processors, such as a multicore CPU (known as *symmetric*)
  - Multiple dissimilar processors, such as a CPU and GPU (known as *asymmetric*)
Level Of Parallelism

• Fine-grain parallelism
  – Parallelism among individual instructions (e.g., two addition operations occur at the same time)

• Coarse-grain parallelism
  – Parallel execution of programs on multiple cores
Explicit Vs. Implicit Parallelism

- **Explicit parallelism**
  - Visible to programmer
  - Requires programmer to initiate and control parallel activities

- **Implicit parallelism**
  - Hidden from programmer
  - Hardware runs multiple copies of application code or instructions automatically
Parallel Computer

• Design in which a computer has reasonably large number of processors
• Motivation: scaling computation
• Example: computer with thirty-two cores
• Counterexamples (not generally classified as a parallel computer):
  – Dual-core processor
  – Computer with one processor and lots of I/O devices (e.g., multiple disks)
Types Of Parallel Architectures

- Three types named according to the *Flynn classification*

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SISD</td>
<td>Single Instruction stream Single Data stream</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction stream Multiple Data streams</td>
</tr>
<tr>
<td>MISD</td>
<td>Multiple Instruction streams Single Data stream</td>
</tr>
<tr>
<td>MIMD</td>
<td>Multiple Instruction streams Multiple Data streams</td>
</tr>
</tbody>
</table>

- Terminology well-known and widely used
- Flynn taxonomy only provides broad, intuitive definitions
- MISD is unusual
SISD: A Conventional (Nonparallel) Architecture

- Processor executes one instruction at a time
- Each operation applies to one set of data items (operands)
- Synonyms include
  - Sequential architecture
  - Uniprocessor
SIMD: Single Instruction Multiple Data

- Each instruction specifies a single operation
- Hardware applies operation to multiple data items
- Typical implementation
  - \textit{Add} operation performs pairwise addition on two one-dimensional arrays
  - \textit{Store} operation can be used to clear a large block of memory
- Special case of SIMD: \textit{vector processor}
  - Usual focus is on floating point operations
  - Applies a given operation to a 1-dimensional array of values (e.g., normalize values)
Normalization Example

• On a conventional computer

\[
\text{for } i \text{ from 1 to } N \{ \\
\quad V[i] \leftarrow V[i] \times Q; \\
\}
\]

• On a vector processor

\[
V \leftarrow V \times Q;
\]

• Vector code is trivial (no iteration)
• Compiler generates a single \textit{vector instruction}
• Computer has \(K\) copies of the multiplication hardware; vectors longer than \(K\) require multiple steps
Graphics Processor Units (GPUs)

- Special-purpose graphics processors
- Follow SIMD design
- Typically, many GPUs on a single graphics interface card
- Technique: divide image (or video frame) into many parts and have each GPU work on one part
- Modern GPU also has conventional operations (called *scalar*)
MIMD: Multiple Instructions Multiple Data

- Parallel architecture with multiple physical processors
- Each processor
  - Can run an independent program
  - May have dedicated I/O devices (e.g., its own disk)
- Parallelism is visible to programmer
- Works best for applications where computation can be divided into separate, independent pieces
Two Popular Categories Of Multiprocessors

- Symmetric
- Asymmetric
Symmetric Multiprocessor (SMP)

- Most well-known MIMD architecture
- Set of $N$ identical processors
- Historic examples of SMP computers
  - Carnegie Mellon University (C.mmp)
  - Sequent Corporation (Balance 8000 and 21000)
  - Encore Corporation (Multimax)
- Current example: multicore CPU
Illustration Of A Symmetric Multiprocessor

- Major problem with SMP architecture: *contention* for memory and I/O devices
- To improve performance: provide each processor with its own copy of a device
Asymmetric Multiprocessor (AMP)

- Set of processors of various types
- Can have processors optimized for specific tasks
- Special-purpose processors are invoked by main processor as needed
- Examples
  - Graphics coprocessor (e.g., GPU)
  - Math coprocessor handles floating point operations
  - I/O coprocessor optimized for handling devices and interrupts
Programmable I/O Processors

• Old idea

• Pioneered in mainframe computers of 1960s

• Examples
  – Channel (IBM mainframe)
  – Peripheral Processor (CDC mainframe)

• Making a comeback — now used in large systems
Multiprocessor Overhead

- Having many processors is not always a clear win
- Overhead arises from
  - Communication
  - Coordination
  - Contention
Communication In A Multiprocessor

• Needed
  – Among processors
  – Between processors and I/O devices
  – Across networks

• As number of processors increases, communication becomes a bottleneck
Coordination In A Multiprocessor

- Needed when processors work together
- May require one processor to wait for another to compute a result
- One possibility: designate a processor to perform coordination tasks
Contestation In A Multiprocessor

- Processors contend for resources
  - Memory and caches
  - I/O devices
- Speed of resources can limit overall performance
  - Example: bus hardware makes $N-1$ processors wait while one processor accesses memory
Performance Of Multiprocessors

• Has been disappointing

• Bottlenecks include
  – Contention for operating system (only one copy of OS can run)
  – Contention for memory and I/O

• Another problem: caching
  – One centralized cache means contention problems
  – Coordinating multiple caches means complex interaction

• Many applications are I/O bound
According To John Harper

“Building multiprocessor systems that scale while correctly synchronising the use of shared resources is very tricky, whence the principle: with careful design and attention to detail, an N-processor system can be made to perform nearly as well as a single-processor system. (Not nearly N times better, nearly as good in total performance as you were getting from a single processor). You have to be very good — and have the right problem with the right decomposability — to do better than this.”

http://www.john-a-harper.com/principles.htm
Assessing Parallelism And Speedup

• Speedup is defined relative to performance of a single processor
• Measure is execution time, which is lower if performance is higher

\[
\text{Speedup} = \frac{\tau_1}{\tau_N}
\]

• Where
  – \( \tau_N \) denotes the execution time on a multiprocessor
  – \( \tau_1 \) denotes the execution time on a single processor
• Ideal: speedup that is linear in number of processors
Typical Speedup For A Few Processors

![Graph showing speedup vs. number of processors]

- Speedup
- Number of processors (N)
- ideal
- typical
At some point, performance begins to decrease!
Consequences For Programmers

- Writing code for multiprocessors is difficult
  - Need to handle mutual exclusion for shared items
  - Typical mechanism: locks
- Performance may be worse than a single processor
- Beware of
  - Vendors selling multicore systems
  - Projects where software engineers must exploit multicore to achieve high performance
The Need For Locking

• Consider a trivial assignment statement

\[ x = x + 1; \]

• Typical code

  load    x, R5  
  incr    R5   
  store   R5, x

• On a uniprocessor, no problems arise

• Consider a multiprocessor
The Need For Locking
(continued)

- Suppose two processors (cores) attempt to increment item $x$
- The following sequence can result
  - Processor 1 loads $x$ into its register 5
  - Processor 1 increments its register 5
  - Processor 2 loads $x$ into its register 5
  - Processor 1 stores its register 5 into $x$
  - Processor 2 increments its register 5
  - Processor 2 stores its register 5 into $x$
Hardware Locks

- Prevent simultaneous access to data
- A separate lock assigned to each item
- Each lock is assigned an ID
- If lock 17 is used, code becomes

  ```
  lock 17
  load x, R5
  incr R5
  store R5, x
  release 17
  ```

- Hardware allows one processor (core) to hold a given lock at a given time, and blocks others
Programming Parallel Computers

- Implicit parallelism
  - Programmer writes sequential code
  - Hardware runs many copies automatically

- Explicit parallelism
  - Programmer writes code for parallel architecture
  - Code must use locks to prevent interference

- Conclusion: explicit parallelism makes computers extremely difficult to program
Programming Symmetric And Asymmetric Multiprocessors

- Both types can be difficult to program
- Symmetric has two advantages
  - Only one instruction set to learn
  - Programmer does not need to choose processor type for each task
- Asymmetric has an advantage
  - Programmer can use processor that is best-suited to a given task
  - Example: using a GPU may be easier than implementing graphics operations on a standard processor
### Redundant Parallel Architectures

- Used to increase reliability rather than performance
- Multiple copies of hardware perform *same* function
- Watchdog circuitry detects whether all units computed the same result
- Can be used to
  - Test whether hardware is performing correctly
  - Serve as backup in case of hardware failure
Terminology For Degree Of Coupling

• Tightly coupled multiprocessor
  – Multiple processors in single computer
  – Buses or switching fabrics used to interconnect processors, memory, and I/O
  – Usually one operating system

• Loosely coupled multiprocessor
  – Multiple, independent computer systems
  – Computer networks used to interconnect systems
  – Each computer runs its own operating system
  – Known as distributed computing
Cluster Computer

- Special case of distributed computer system
- All computers work on a single problem
- Works best if problem can be partitioned into pieces
- Currently popular in large data centers
- Modern supercomputer is a cluster
- Example supercomputer
  - Tianhe-2 supercomputer in China
  - 16,000 Intel multicore nodes
  - Total of 3,120,000 cores
Grid Computing

- Form of loosely-coupled distributed computing
- Uses computers on the Internet
- Popular for large, scientific computations
- One application: Search for Extra-Terrestrial Intelligence (SETI)
Summary

• Parallelism is fundamental

• Flynn scheme classifies computers as
  – SISD (e.g., conventional uniprocessor)
  – SIMD (e.g., vector computer)
  – MIMD (e.g., multiprocessor)

• Multiprocessors can be
  – Symmetric or asymmetric
  – Explicitly or implicitly parallel

• Multiprocessor speedup usually less than linear
Summary
(continued)

• Programming multiprocessors is usually difficult
  – Programmer must divide tasks onto multiple processors
  – Locks needed for shared items

• Parallel systems can be
  – Tightly-coupled (single computer)
  – Loosely-coupled (computers connected by a network)
Questions?
Module XIX

Data Pipelining
Concept Of Pipelining

- One of the two major hardware optimization techniques
- Information flows through a series of stages (processing components)
- Each stage can perform arbitrary operations on the data
  - Inspect
  - Interpret
  - Modify
Illustration Of Pipelining
Data Pipeline Possibilities

- Hardware or software implementation
- Large or small scale
- Synchronous or asynchronous flow
- Buffered or unbuffered flow
- Finite chunks or continuous bit streams
- Automatic data feed or manual data feed
- Serial or parallel data path between stages
- Homogeneous or heterogeneous stages
Software Implementation Of Data Pipelining

- Popularized by Unix command interpreter (shell)
- User can specify pipeline as a command
- Example

```
cat x | sed ’s/friend/partner/g’ | sed ’/W/d’ | more
```

- Substitutes “partner” for “friend”
- Deletes lines that contain “W”
- Passes result to more for display

- Note: example can be optimized by swapping the order of the two sed commands
Implementing A Software Pipeline

- **Uniprocessor**
  - Each stage is a *process* or *thread*

- **Multiprocessor**
  - Each stage executes on separate processor or core
  - Hardware assist can speed interstage data transfer
Hardware Implementation Of Data Pipelining

- Two basic types
- Instruction pipeline
  - Covered earlier in the course
  - Optimizes performance
  - Heavily used with RISC architecture
  - Each instruction processed in stages
  - Exact details and number of stages depend on instruction set and operand types
- Data pipeline
  - New idea
Hardware Data Pipeline

- Sequence of data items pass through the pipeline
- Each stage performs computation on the data item and passes item to next stage
- Requires designer to divide computation into stages
- Among the most interesting uses of pipelining
Data Pipelining And Performance

- A data pipeline implemented with hardware can dramatically increase performance (throughput)

- To see why, consider an example
  - Internet router handles packets
  - Assume that a router
    * Processes one packet at a time
    * Performs six functions on each packet
Example Of Internet Router Processing

1. Receive a packet (i.e., transfer the packet into memory)
2. Verify packet integrity (i.e., verify that no changes occurred between transmission and reception)
3. Check for forwarding loops (i.e., decrement a value in the header, and reform the header with the new value)
4. Select path (i.e., use the destination address field to select one of the possible output networks and a destination on that network)
5. Prepare for transmission (i.e., compute information that will be used to verify packet integrity)
6. Transmit the packet (i.e., transfer the packet to the output device)
Illustration Of A Processor In A Router And The Algorithm Used

- (a) illustration of an Internet router with multiple outgoing network connections
- (b) the computational steps the router must take for each packet

```plaintext
do forever {
    Wait to receive packet
    Verify integrity
    Check for loops
    Select a path
    Prepare for transmission
    Enqueue packet for output
}
```
Example Data Pipeline Implementation

• Consider a router that uses a data pipeline
Example Data Pipeline Implementation

- Consider a router that uses a data pipeline
Example Data Pipeline Implementation

- Consider a router that uses a data pipeline

- Imagine a packet passing through the pipeline
Example Data Pipeline Implementation

- Consider a router that uses a data pipeline

- Imagine a packet passing through the pipeline

- For now, assume zero delay between stages
Example Data Pipeline Implementation

• Consider a router that uses a data pipeline

• Imagine a packet passing through the pipeline
• For now, assume zero delay between stages
• Question: how long will the pipeline take to process the packet?
Example Data Pipeline Implementation

- Consider a router that uses a data pipeline

Imagine a packet passing through the pipeline
- For now, assume zero delay between stages
- Question: how long will the pipeline take to process the packet?
- Answer: the same amount of time as a conventional router!
The News About Pipelines

- Bad news: if it uses processors of the same speed as a nonpipeline architecture, a data pipeline will not improve the overall time needed to process a given data item.
- Good news: by overlapping computation on multiple items, a pipeline increases throughput.
Data Pipelining Only Improves Throughput If

- It is possible to partition processing into independent stages
- Overhead required to move data from one stage to another is insignificant
- The slowest stage of the pipeline is faster than a single processor
Understanding Pipeline Speed

• Assume
  – The task is packet processing
  – Processing a packet requires exactly 500 instructions
  – A processor executes 10 instructions per \( \mu \text{sec} \)

• Total time required for one packet

\[
\text{time} = \frac{500 \text{ instructions}}{10 \text{ instr. per } \mu \text{sec}} = 50 \mu \text{sec}
\]

• Throughput for a non-pipelined system

\[
T_{np} = \frac{1 \text{ packet}}{50 \mu \text{sec}} = \frac{1 \text{ packet} \times 10^6}{50 \text{ sec}} = 20,000 \text{ packets per second}
\]
Understanding Pipeline Speed
(continued)

- Suppose the problem can be divided into four stages and that the stages require
  - 50 instructions
  - 100 instructions
  - 200 instructions
  - 150 instructions

- The slowest stage takes 200 instructions
- The time required for the slowest stage is:

\[
\text{total time} = \frac{200 \text{ inst}}{10 \text{ inst/\mu sec}} = 20 \text{ \mu sec}
\]
Understanding Pipeline Speed
(continued)

- Important principle: the throughput of a data pipeline is limited by the slowest stage
Understanding Pipeline Speed
(continued)

- Important principle: the throughput of a data pipeline is limited by the slowest stage
- Overall throughput

\[ T_p = \frac{1 \text{ packet}}{20 \mu \text{sec}} = \frac{1 \text{ packet} \times 10^6}{20 \text{ sec}} = 50,000 \text{ packets per second} \]
Understanding Pipeline Speed
(continued)

• Important principle: the throughput of a data pipeline is limited by the slowest stage

• Overall throughput

\[ T_p = \frac{1 \text{ packet}}{20 \ \mu\text{sec}} = \frac{1 \text{ packet} \times 10^6}{20 \ \text{sec}} = 50,000 \text{ packets per second} \]

• Note: throughput of pipelined version is 250% of throughput of the non-pipelined version!
Pipeline Architectures

- Term refers to computer systems in which the primary focus is data pipelining
- Most often used for special-purpose systems
- Data pipeline usually organized around functions
- Less relevant to general-purpose computers
Functional Organization Of A Data Pipeline

- Build one pipeline stage per function
- Illustration

(a) shows a single processor handling three functions
(b) shows processing divided into a 3-stage pipeline with each stage handling one function
Pipeline Terminology

- **Setup time**
  - Refers to time required to start the pipeline initially

- **Stall time**
  - Refers to time required to restart the pipeline after a stage blocks to wait for a previous stage

- **Flush time**
  - Refers to time that elapses between the cessation of input and the final data item emerging from the pipeline (i.e., the time required to shut down the pipeline)
Superpipelining

- Most often used with instruction pipelining
- Subdivides a stage into smaller stages
- Example: subdivide operand processing into
  - Operand decode
  - Fetch immediate value or value from register
  - Fetch value from memory
  - Fetch indirect operand
- Technique: subdivide the slowest pipeline stage
Summary

- Pipelining
  - Broad, fundamental concept
  - Can be used with hardware or software
  - Applies to instructions or data
  - Can be synchronous or asynchronous
  - Can be buffered or unbuffered
Summary  
(continued)

- Pipeline performance
  - Unless faster processors are used, data pipelining does not decrease the overall time required to process a single data item
  - Using a pipeline does increase the overall throughput (items processed per second)
  - The stage of a pipeline that requires the most time to process an item limits the throughput of the pipeline
Questions?
Power and energy constraints are now the driving force in all devices from servers to smartphones.

– Kathryn McKinley
Microsoft, 2013
Power

- Rate at which energy is consumed
- Measured in watts, milliwatts, kilowatts, or megawatts (one watt is one Joule per second)
- Instantaneous value
- The power at time $t$ is given by

$$P(t) = V(t) \times I(t)$$

where $V$ is voltage and $I$ is current
Energy

- A fundamental property of the universe
- Measured in joules, but reported in watts multiplied by time: milliwatt hours (mWh), kilowatt hours (kWh), or megawatt hours (MWh)
- For constant power utilization, energy used from time $t_0$ to $t_1$ is
  \[ E = P \times (t_1 - t_0) \]
- If power consumption is not constant, energy is an integral of power
  \[ E = \int_{t=t_0}^{t_1} P(t) \, dt \]
When Power And Energy Are Important

- **Power**
  - Associated with data centers
  - Question: can supplier deliver the megawatts (or gigawatts) required?

- **Energy**
  - Associated with portable systems
  - Question: how long will the battery last?
Two Primary Forms Of Power Consumption In A Digital Circuit

- **Switching or dynamic power** (denoted $P_s$ or $P_d$)
  - Switching is a change of a logic gate output when an input changes
  - Some power is required to cause such a change
- **Leakage power** (denoted $P_{\text{leak}}$)
  - Caused because transistors are imperfect
  - A few electrons penetrate a semiconductor boundary even when the transistor is off
  - Important observation: 40 to 60 percent of power usage is leakage
- **Minor amount** of “short circuit” power lost during switching
Energy Consumed By A CMOS Circuit

• Energy for a single gate change

\[ E_d = \frac{1}{2} C V_{dd}^2 \]

• \( C \) is a value of capacitance that depends on the underlying CMOS technology

• \( V_{dd} \) is the voltage at which the circuit operates
Energy Consumption And Clocks

- Observe
  - Energy is consumed every time a gate changes
  - Many parts of circuit run on a clock
  - When clock pulses, the inputs to some gates change

- Consequences
  - Energy is consumed when a clock runs, even if the circuit is not otherwise active
  - The rate of the clock determines the rate at which a gate uses energy
Clock Rates And Switching Power

- Clock changes state twice per cycle, so the power used in one period is
  \[ P_{avg} = \frac{C \cdot V_{dd}^2}{T_{clock}} \]

- And the frequency of the clock is
  \[ F_{clock} = \frac{1}{T_{clock}} \]

- Which makes the power used
  \[ P_{avg} = C \cdot V_{dd}^2 \cdot F_{clock} \]
Partial Use

• Some systems have the ability to shut down part of a circuit (e.g., shut down some of the cores in a multicore processor)

• If we let $\alpha$ denote the fraction of the circuit in use, $0 \leq \alpha \leq 1$, the average power is

$$P_{avg} = \alpha C V_{dd}^2 F_{clock}$$

• Three factors that control power consumption
  - The fraction of the circuit that is active, $\alpha$
  - The clock frequency, $F_{clock}$
  - The voltage in the circuit, $V_{dd}$
Cooling And The Power Wall

- Amount of heat produced is proportional to the power used
- *Power density* refers to concentration of power
- For chips, power density increases as the industry decreases transistor size according to Moore’s Law
- Cooling technologies determine how much heat can be removed
- With current technologies, the limit is known as a *power wall*, and is given by

\[
\text{PowerWall} = 100 \frac{\text{watts}}{\text{cm}^2}
\]
Power Management

- Decreasing the clock rate
  - Reduces the switching power
  - Does not help with leakage
  - May mean the device runs longer (more leakage)

- Decreasing voltage has biggest potential savings (longest battery life)
  - Underlying technology must be redesigned
  - Cell phones already have lower voltage (3.8 or 2.6 volts)
  - Problem: lower voltage increases gate delay, which means the clock rate must also be lowered
Slower Clocks And Multicore Processors

- Reducing power consumption is the driving force
Slower Clocks And Multicore Processors

- Reducing power consumption is the driving force
- Consider a dual-core chip where each core runs half as fast as a single-core version
- Slower clock rate means voltage can be lowered, reducing power consumption dramatically
Slower Clocks And Multicore Processors

- Reducing power consumption is the driving force
- Consider a dual-core chip where each core runs half as fast as a single-core version
- Slower clock rate means voltage can be lowered, reducing power consumption dramatically
- One example
  - Slowing a clock to one-half the original speed permits voltage to be lowered and cuts the power consumed by a core to approximately 15% of the original value
  - Two cores running at half the clock rate consume about 30% as much power as the original chip and yet have approximately the same computational capability
Clock Rates And Cores

- Can we extend the idea to many cores?
Clock Rates And Cores

- Can we extend the idea to many cores?
- In theory, yes, because using multiple slow cores can save more energy than a single high-speed core
Clock Rates And Cores

- Can we extend the idea to many cores?
- In theory, yes, because using multiple slow cores can save more energy than a single high-speed core
- In practice, however
Can we extend the idea to many cores?

In theory, yes, because using multiple slow cores can save more energy than a single high-speed core.

In practice, however:
- Programmers must find a way to divide computation among all the cores.
- Coordination and communication can mean that N cores cannot perform as well as one core.
- An arbitrarily slow clock rate may not work for some applications (e.g., video).
Software Control Of Power

• *Power gating*
  – Refers to cutting power to some parts of a circuit
  – Achieved with special, low-leakage power transistors

• *Clock gating*
  – Refers to stopping the clock (setting the frequency to zero)
  – Requires software to save state and restore it when restarting the system
Digital Circuit Sleep Modes

- Common for embedded processors
- Series of low-power modes
- Software decides when to sleep and awaken
- Wakeup
  - Typically performed “on demand”
  - Example: user presses a key
Choosing When To Sleep

- Usually employs a timeout mechanism: if circuit has been idle for time $T$, enter a sleep mode
- For user-visible actions, allow the user to specify the timeout
- For other actions, compute a break even point
Entering Sleep Mode

- Goal is typically energy savings
- Enter sleep mode only if doing so will save energy
- Let $T_{\text{shutdown}}$ and $T_{\text{wakeup}}$ denote the time required to shutdown and wake up, respectively
- We will use a simplified model to analyze sleep modes
Energy Used During Transitions

- Shutting down or restarting requires energy

\[
E_{\text{shutdown}} = E_s = P_{\text{shutdown}} \times T_{\text{shutdown}} \\
E_{\text{wakeup}} = E_w = P_{\text{wakeup}} \times T_{\text{wakeup}}
\]

- The energy used while running for time \( t \) or sleeping for time \( t \) is

\[
E_{\text{run}} = P_{\text{run}} \times t \\
E_{\text{sleep}} = E_s + E_w + P_{\text{off}} \left( t - T_{\text{shutdown}} - T_{\text{wakeup}} \right)
\]

- Shutting down the system will be beneficial at breakpoint

\[
E_{\text{sleep}} < E_{\text{run}}
\]
Notes On Our Analysis

- Our model is simplistic
- Breakpoint inequality can be expressed as a function of $t$ and constants, which means we can find a minimum value of $t$ for which sleeping is beneficial
- If processor has five sleep modes, model and analysis must be extended for each of the modes
Summary

- Power is an instantaneous measure of the rate at which energy is used
- Energy is the total amount of power used over a given time
- Two primary power uses in a digital circuit are switching power and leakage power
- Leakage power can account for 40 to 60 percent of all power used
- Reducing voltage reduces the power required and introduces gate delays, which requires reducing the clock speed
- Options for software management of power include clock gating and power gating
Many processors have low-power modes (sleep modes)

Because energy is required to move into and out of a sleep mode, a break even point can be calculated at which sleep mode saves energy.
Questions?
Module XXI

Assessing Performance
Measuring Computational Power

- Difficult to assess computer performance
- Chief problems
  - Flexibility: computer can be used for wide variety of computational tasks
  - Architecture that is optimal for some tasks is suboptimal for others
  - Memory and I/O costs can dominate processing
  - Performance often depends on the specific input data, not just the size of the data
Consequences

- Many groups try to assess computer performance
- A variety of performance measures exist
- No single measure suffices for all situations
Measures Of Computational Power

- Two primary measures
- Integer computation speed
  - Pertinent to most applications
  - Example measure is millions of instructions per second (MIPS)
- Floating point computation speed
  - Used for scientific calculations
  - Typically involve matrices
  - Example measure is floating point operations per second (FLOPS)
Average Execution Speed And Variance

- Can we ignore the data and focus on measuring the performance of various groups of instructions?

- One possible measure is the average (i.e., mean) execution time of all the instructions available on a computer

- Problems
  - Even two closely-related instructions do not take exactly the same time
  - A given program may use some instructions more than others
Example: Average Floating Point Performance

• Assume
  – Addition or subtraction takes $Q$ nanoseconds
  – Multiplication or division takes $2Q$ nanoseconds

• The average cost of a floating point instruction is

\[ T_{\text{avg}} = \frac{Q + Q + 2Q + 2Q}{4} = 1.5Q \text{ ns per instr.} \]

• Note that addition or subtraction takes 33% less than the average, and multiplication or division takes 33% more

• A typical program will not have equal numbers of add, subtract, multiply and divide operations
Application Specific Instruction Counting

- Idea is to find a more accurate assessment of performance for a specific application
- Examine application to determine how many times each instruction occurs
- Example: multiplication of two $N \times N$ matrices
  - $N^3$ floating point multiplications
  - $N^3 - N^2$ floating point additions
  - Using $Q$ and $2Q$ for costs gives:

$$T_{total} = 2 \times Q \times N^3 + Q \times (N^3 - N^2)$$
Weighted Average

- Alternative to precise count of operations
- Typically obtained by instrumentation
- Program is run on many input data sets and each instruction counted
- Counts averaged over all runs
- Example

<table>
<thead>
<tr>
<th>Instruction Type</th>
<th>Count</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>8513508</td>
<td>72</td>
</tr>
<tr>
<td>Subtract</td>
<td>1537162</td>
<td>13</td>
</tr>
<tr>
<td>Multiply</td>
<td>1064188</td>
<td>9</td>
</tr>
<tr>
<td>Divide</td>
<td>709458</td>
<td>6</td>
</tr>
</tbody>
</table>
Computing A Weighted Average

- Uses instruction counts and cost of each instruction
- Example

\[ T_{avg}' = 0.72 \times Q + 0.13 \times Q + 0.09 \times 2Q + 0.06 \times 2Q \]

- Or

\[ T_{avg}' = 1.16 \hspace{0.2cm} Q \hspace{0.2cm} ns \hspace{0.2cm} per \hspace{0.2cm} instruction \]

- Note: the weighted average given here is 23% less than the uniform average obtained above
Instruction Mix

- An attempt to generalize weighted average to a class of applications
- Measure a large set of programs
- Obtain relative weights for each type of instruction
- Use relative weights to assess the performance of a given architecture on the example set
- Try to choose set of programs that represent a typical workload
- Computer architect can use an instruction mix to assess how a proposed architecture will perform.
Standardized Benchmarks

- Provides workload used to measure computer performance
- Represent typical applications
- Independent corporation formed in 1980s to create benchmarks
  - Named *Standard Performance Evaluation Corporation (SPEC)*
  - Not-for-profit
  - Avoids having each vendor choose benchmark that is tailored to their architecture
Examples Of Benchmarks Developed By SPEC

- SPEC cint2006
  - Used to measure integer performance

- SPEC cfp2006
  - Used to measure floating point performance

- Result of measuring performance on a specific architecture is known as the computer’s SPECmark
I/O And Memory Bottlenecks

• CPU performance is only one aspect of system performance
• Other parts of system to be measured
  – Memory
  – I/O
• Bottleneck in a given architecture can be any of the above
• Consequence: benchmarks have also been created to focus on memory and I/O performance rather than computational speed
Increasing Overall Performance

- How can we build a faster computing system?
- Hardware is faster than software (just eliminating the fetch-execute cycle speeds up processing)
- Resulting general principle: to optimize performance, move operations that account for the most CPU time from software into hardware
Which Items Should Be Optimized?

- Adding additional hardware increases cost
- Consequence: we cannot afford to use high-speed hardware for all operations
- Computer architect Gene Amdahl observed that it is a waste of resources to optimize functions that are seldom used
- Amdahl’s Law:

The performance improvement that can be realized from faster hardware technology is limited to the fraction of time the faster technology can be used.
Quantitative Version Of Amdahl’s Law

\[
\text{Speedup}_{\text{overall}} = \frac{1}{1 - \text{Fraction}_{\text{enhanced}} + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}
\]

- **Notes**
  - \( \text{Speedup}_{\text{overall}} \) is the overall speedup achieved
  - \( \text{Fraction}_{\text{enhanced}} \) is the fraction of time the enhanced hardware runs
  - \( \text{Speedup}_{\text{enhanced}} \) is the speedup the enhanced hardware gives
Amdahl’s Law And Parallel Systems

• Consider a parallel architecture
• Increasing parallelism adds more hardware
• Amdahl’s law explains why adding processors does not always increase performance
Summary

- A variety of performance measures exist
- Simplistic measures include MIPS and FLOPS
- More sophisticated measures use a weighted average derived by counting the instructions in a program or set of programs
- A set of weights from multiple applications corresponds to an instruction mix
- Benchmark refers to a standardized program or set of programs used to measure performance
- Best-known benchmarks, known as SPECmarks, are produced by the SPEC Corporation
- Amdahl’s Law helps architects select functions to be optimized (moved from software to hardware)
Questions?
Module XXII

Architecture Examples
And Hierarchy
General Idea

- Recall that architecture can be presented at multiple levels of abstraction
- We use the term *architectural hierarchy*
- Broad classifications
  - Macroscopic (e.g., entire computer system)
  - Microscopic (e.g., single integrated circuit)
# Possible Architectural Levels

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>A complete computer with processor(s), memory, and I/O devices. A typical system architecture describes the interconnection of components with buses.</td>
</tr>
<tr>
<td>Board</td>
<td>An individual circuit board that forms part of a computer system. A typical board architecture describes the interconnection of chips and the interface to a bus.</td>
</tr>
<tr>
<td>Chip</td>
<td>An individual integrated circuit that is used on a circuit board. A typical chip architecture describes the interconnection of functional units and gates.</td>
</tr>
</tbody>
</table>
Example System-Level Architecture
(A Personal Computer)

- Functional units
  - Processor
  - Memory
  - I/O interfaces

- Interconnections
  - High-speed buses for high-speed devices and functional units
  - Low-speed buses for lower-speed devices
Bus Interconnection And Bridging

• Recall: *bridge* technology used to interconnect buses

• Allows
  – Multiple buses in a computer system
  – Processor only connects to one bus

• Bridge maps between bus address spaces

• Permits *backward compatibility* (e.g., old I/O device can connect to old bus and still be used with newer processor and newer bus)
Example Of Bridging

• Consider a PC
• Assume
  – Processor uses *Peripheral Component Interconnect* bus (PCI)
  – Some I/O devices use older *Industry Standard Architecture* (ISA)
• The two buses are incompatible (cannot be directly connected)
• Solution: use two buses connected by a bridge
Logical PC Architecture Using A Bridge

- Interconnection can be *transparent*
Physical Architecture

- Implementation of bridge is more complex than our conceptual diagram implies
- Usually uses special-purpose controller chips
- Separates high-speed and low-speed units onto separate chips
- Provides the illusion of a bus over a direct connection (bus does not need sockets for devices)
Typical PC Architecture

- Two controller chips used
- *Northbridge* chip connects higher-speed units
  - Processor
  - Memory
  - Advanced Graphics Port (AGP) interface
- *Southbridge* chip connects lower-speed units
  - Local Area Network (LAN) interface
  - PCI bus
  - Keyboard, mouse, or printer ports
Illustration Of Physical PC Architecture

CISC CPU (x86)

Northbridge

controller

AGP port

Stream Comm.

controller

proprietary hub connection

Southbridge

dual-ported memory

DDR SDRAM

DDR SDRAM

6-chan. audio

LAN interface

ISA bus

USB

PCI

controller
Example Bridge Products

- Northbridge: Intel 82865PE
- Southbridge: Intel ICH5
Example Connection Speeds

- Rates increase over time, so look at relative speeds, not absolute numbers in the following examples

<table>
<thead>
<tr>
<th>Connection</th>
<th>Clock Rate</th>
<th>Width</th>
<th>Throughput†</th>
</tr>
</thead>
<tbody>
<tr>
<td>USB 1.0</td>
<td>33 MHz</td>
<td>32 bits</td>
<td>1.5 MB/s</td>
</tr>
<tr>
<td>FCC broadband</td>
<td>–</td>
<td>–</td>
<td>3.1 MB/s</td>
</tr>
<tr>
<td>AGP</td>
<td>100–200 MHz</td>
<td>64–128 bits</td>
<td>2.0 GB/s</td>
</tr>
<tr>
<td>USB 3.0</td>
<td>up to 500 MHz</td>
<td>32 bits</td>
<td>5.0 GB/s</td>
</tr>
<tr>
<td>Memory</td>
<td>200–800 MHz</td>
<td>64–128 bits</td>
<td>6.4 GB/s</td>
</tr>
<tr>
<td>PCI 3.0</td>
<td>33 MHz</td>
<td>32 bits</td>
<td>126.0 GB/s</td>
</tr>
<tr>
<td>Registers</td>
<td>1000–2000 MHz</td>
<td>64–128 bits</td>
<td>672.0 GB/s</td>
</tr>
</tbody>
</table>

- The FCC’s definition of broadband network speed has been included as a point of comparison
Bridging Functionality And Virtual Buses

- Controller chips can virtualize hardware
- Example: controller can present the illusion of multiple buses to the processor
- One possible form: controller presents three virtual buses
  - Bus 1 contains the host and memory
  - Bus 2 contains a high-speed graphics device
  - Bus 3 corresponds to the external PCI slots for I/O devices
Example Board-Level Architecture

- Consider an Ethernet interface
  - Connects computer to Local Area Network
  - Transfers data between computer and network
  - Physically consists of separate circuit board
  - Usually contains an embedded processor and buffer memory
Example Board-Level Architecture: LAN Interface
Memory On A LAN Interface

- **SRAM**
  - Highest speed
  - Typically used for instructions
  - May be used to hold packet headers

- **DRAM**
  - Lower speed
  - Typically used to hold packets

- Designer decides which data items to place in each memory
Chip-Level Architecture

- Describes structure of single integrated circuit
- Components are functional units
- Can include on-board processors, memory, or buses
Example Chip-Level Architecture
(Netronome Network Processor)
Structure Of Functional Units On A Chip
(SRAM Access Unit)

- Each item further composed of logic gates
Summary

- Architecture of a digital system can be viewed at several levels of abstraction
- System architecture shows entire computer system
- Board architecture shows individual circuit board
- Chip architecture shows individual IC
- Functional unit architecture shows individual unit on an IC
Summary (continued)

- We examined an example hierarchy
  - Entire PC
  - Physical interconnections of a PC
  - LAN interface in a PC
  - Network processor chip on a LAN interface
  - SRAM access unit on a network processor chip
Questions?
Module XXIII

Examples Of Chip-Level Architecture
(Network Processors)
**Definition**

A *network processor* is a special-purpose programmable hardware device that combines the low cost and flexibility of a RISC processor with the speed and scalability of custom silicon (i.e., ASIC chips), and is designed to provide computational power for packet processing systems such as Internet routers.
Commercial Network Processors
Commercial Network Processors

- First emerged in late 1990s
- Used in products 2000–
- By 2003, more than thirty vendors existed
- Large variety of architectures
- Optimizations: parallelism and pipelining
Commercial Network Processors

- First emerged in late 1990s
- Used in products 2000–
- By 2003, more than thirty vendors existed
- Large variety of architectures
- Optimizations: parallelism and pipelining
- Currently, only a handful of vendors remain viable
Augmented RISC (Alchemy)
Pipeline Of Homogeneous Processors (Cisco)
Pipeline Of Parallel Heterogeneous Processors (EZchip)
Extensive And Diverse Processors (Hifn)

```
<table>
<thead>
<tr>
<th>Ingress Data Store</th>
<th>Ingress Switch Interface</th>
<th>Internal SRAM</th>
<th>Egress Switch Interface</th>
<th>Egress Data Store</th>
</tr>
</thead>
</table>

SRAM for Ingress Data

Embedded Processor Complex (EPC)

Ingress Physical MAC Multiplexor

Egress Physical MAC Multiplexor

Traffic Management and Scheduling

Packets from Physical Devices

Packets to Physical Devices

Egress Data Store

To Switching Fabric

PCI Bus

External DRAM and SRAM

From Switching Fabric
```
Short Pipeline Of Unconventional Processors (Agere)

- Classifier uses programmable pattern matching engine
- Traffic manager includes 256,000 queues
Extremely Long Pipeline (Xelerated)

- Each processor executes four instructions per packet
- External coprocessor calls used to pass state
Parallel Packet Processors (Netronome²)

optional host connection

SRAM buses

SRAM

coprocessor

FLASH

DRAM

SRAM access

PCI access

scratch memory

Slowport access

DRAM access

MSF access

multiple, independent internal buses

Embedded RISC processor (Xscale)

Microengine 1

Microengine 2

Microengine 3

... 

Microengine N

PCI bus

serial line

receive bus

transmit bus

High-speed I/O buses

Slowport

IXP2xxx chip

DRAM bus
Example Of Complexity (PCI Access Unit)
Questions?
Module XXIV

HARDWARE MODULARITY
BOARDS AND REPLICATION
Modularity

- For software
  - Easy
  - Just build parameterized functions
- For hardware
  - Difficult
  - Must replicate hardware units
Hardware Design

- Desiderata
  - Build series of products
  - Include a range of sizes
  - Avoid designing each from scratch

- Solution
  - Design a basic building block
  - Replicate the block as needed
  - Arrange to activate pieces as needed
Example: Rebooter For The Xinu Lab

- Lab
  - Large set of *backend* computers
  - Students create and download an operating system
  - Student OS runs and interacts over a console line
- However
  - Student OS can wedge the backend computer
  - Must power-cycle backend to regain control
Rebooter System

- Specialized, homemade hardware mechanism
- Provides power to each backend
- Receives commands from lab control software
- Can power-cycle specified backend
Rebooter Concept

- All back-end computers are numbered 0, 1, 2,…
- Lab control software issues command to reboot machine $X$
- Command converted to binary value and sent to rebooter
- Rebooter power-cycles specified backend
The Question Of Size

• How big should a rebooter be?
The Question Of Size

• How big should a rebooster be?
• The lab started with 8 machines, but now has over 100
• Building a rebooster that is too small is insufficient
• Building a rebooster that is too large is wasteful
The Question Of Size

• How big should a rebooster be?
• The lab started with 8 machines, but now has over 100
• Building a rebooster that is too small is insufficient
• Building a rebooster that is too large is wasteful
• Size depends on student enrollment
• We did not know in advance how large the lab would grow
The Question Of Size

- How big should a rebooster be?
- The lab started with 8 machines, but now has over 100
- Building a rebooster that is too small is insufficient
- Building a rebooster that is too large is wasteful
- Size depends on student enrollment
- We did not know in advance how large the lab would grow
- Note: hardware engineers designing products face the same dilemma
Achieving Hardware Modularity
Achieving Hardware Modularity

- Design a basic rebooter hardware module
Achieving Hardware Modularity

- Design a basic rebooter hardware module
- Replicate the module as needed
Achieving Hardware Modularity

• Design a basic rebooster hardware module
• Replicate the module as needed
• One possible design: arrange a basic module that controls sixteen devices
A Modular Design

- Think binary
  - Assume an 8-bit binary input (up to 256 backends)
  - Low-order 4 bits of binary input used to select one of 16 devices
  - High-order 4 bits of binary input used to select a module
- Each module given a unique ID between 0 and 15
- A given module only responds if high-order bits of input match its ID
- Design allows the same binary input to be passed to all modules in parallel
Illustration Of Using Modules
Illustration Of Using Modules

- Four modules allows 64 backends

```
module responds to ID 0
module responds to ID 1
module responds to ID 2
module responds to ID 3
```

8-bit binary input value

power connections for 64 backend computers

other modules can be added
Illustration Of Using Modules

- Four modules allows 64 backends

- System can be expanded by adding more modules
Illustration Of Using Modules

- Four modules allows 64 backends
- System can be expanded by adding more modules
- Hardware designers use this modular approach to build a series of products with various sizes
Assigning An ID To A Module

- One technique: DIP switches
  - Small physical device about as large as a 7400-series IC
  - Each device contains 8 individual switches that can be set (e.g., with the end of a paper clip)
- Switches on a module are set to specify ID before module is installed
- Comparator circuit compares ID in switches to high-order bits of input
- Potential advantage: if a module fails, it can be replaced
- Of course, care must be taken to ensure each module has a unique ID (i.e., only one module responds to a given input)
Interpretation Of The Input

- The same input bits are sent to all modules
- All modules operate in parallel to check the *module identification* bits
- Only one module will match the identification (assuming the hardware is configured correctly)
Summary

- A hardware design is expensive and usually unique
- The technique used for modularization is replication of a basic building block
- Data is sent to all modules in parallel
- Each module is configured to respond to a specific set of inputs
- Typical scheme: use high-order bits of the input to select a module and low-order bits to specify a function on that module
Questions?
Module XXV

SEMESTER WRAP-UP
What You Learned

• The four basic aspects of computer architecture
  – Digital logic
  – Processors
  – Memory
  – I/O

• The vocabulary of hardware

• General ways a hardware designer approaches problems

• How to think in binary

• A potpourri of additional items
Key Ideas From Our Study Of Digital Logic

- Logic gates are building blocks that can be interconnected
- A clock allows a circuit to execute multiple steps in sequence
- Arithmetic operations, such as addition and subtraction, can be performed without iteration
- Underneath, it’s all bits; semantic value depends on how the bits are interpreted
Key Ideas From Our Study Of Processors

• Many types of processors exist
• An *instruction set* defines the operations a processor can perform
  – RISC processors: a small set of basic instructions
  – CISC processors: many instructions that can be complex
• Most processors use one or more general-purpose registers
• An instruction pipeline can increase performance
Key Ideas From Our Study Of Memory

• The chief characteristics of memory systems are
  – Technology (e.g., SRAM and DRAM)
  – Organization (e.g., word addressing)
• Many memory technologies exist (e.g., DDR-DRAM)
• Physical memory organization includes banks and interleaving
• Virtual memory systems provide protection among applications and allow a programmer to use more addresses than the physical memory supports
• Caching can improve memory performance dramatically
• Content Addressable Memory (CAM) provides parallel search
Key Ideas From Our Study Of I/O

- I/O devices attach to a bus, and all I/O is performed using *fetch* and *store* operations on the bus
- A device can be polled or can use interrupts
- Device driver software (in the OS) is divided into
  - Upper-half functions that applications call when they *read* or *write* data
  - Lower-half functions that are invoked when an interrupt occurs
- Sophisticated devices use DMA to transfer data between the device and memory without requiring the CPU to take action
- Buffering can improve I/O performance dramatically
**Miscellaneous Important Ideas**

- Architecture can be viewed at multiple levels of abstraction, including a complete system, a board, or a chip.
- To debug or optimize at one level, need to understand the next lower level.
- Because processors are complex, performance depends on the software that invokes instructions (instruction mix).
- Hardware designers use two principal optimizations:
  - Parallelism
  - Pipelining
- Pipelining increases throughput, but does not reduce latency.
Miscellaneous Important Ideas
(continued)

- To achieve modularity, a hardware designer creates a basic building block and then replicates the block; each copy is configured to respond to a subset of the inputs

- Parallel architectures (e.g., multicore processors, clusters)
  - Are difficult to program (e.g., the programmer may need to use *locks*)
  - Often have contention for shared memory and devices
  - Have not delivered on the promise of performance
What You Take With You From This Course

- Experience connecting chips to form a digital circuit
- Insight into basic structure of a computer and the data paths used to fetch and execute instructions
- Enhanced programming background
- An understanding that hardware designers think in terms of parallel units
- An appreciation of the startling difference between the high-level abstractions software provides and the low-level facilities the hardware provides
- Knowing how to think in binary!
What You Take With You From This Course (continued)

- The insight that dividing computation into a data pipeline can improve throughput, even if each stage of a pipeline runs at the same speed as the original processor.
- An understanding that two cores running at lower voltage and half the clock rate can consume substantially less power than a single core.
- Familiarity with assembly language.
  Note: you may not enjoy programming in assembly language, but it should not be a mystery and you will be able to use it when necessary.
- A sense that you understand what’s going on underneath the software.
Enjoy Your Career!
Questions?