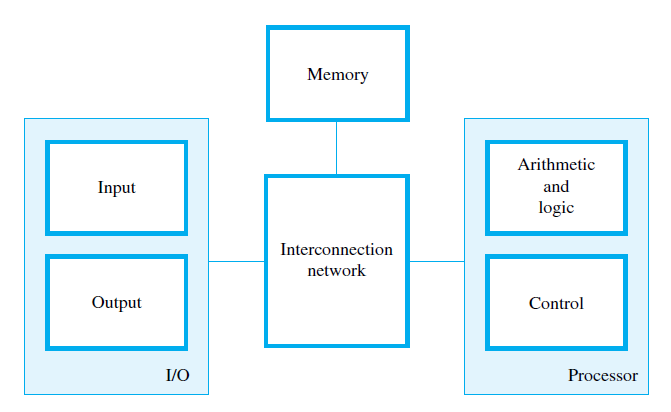
# Chapter 1: Basic Structure of Computers

## Computer Types

* **Embedded Computers**Monitor or control processes/environments in a larger device/system
* **Personal Computers**  
  Computers for individual use in homes, education, businesses
* **Servers/Enterprise Systems**Large computers, shared by users through personal computers accessing public/private networks
  + Used to host large databases, process data, etc.
* **Supercomputers**High cost computers used for demanding computations
* **Grid Computers**Combines many personal computers and disk storage units into a high speed network

## 1.2 Functional Units

Computers have five functionally independent parts



Allows functional units to exchange data and coordinate actions

Stores programs and data

Sends processed results back to outside world

Transfers coded information to computers

Coordinates actions

Processes information

**Machine Instructions**A series of binary bits stored in memory and interpreted by the CPU into commands that specify:

* Transfer of information, within computer and between computer and I/O
* Arithmetic and logic operations

**Program**List of instructions to perform a task

* Stored in memory
* Fetched by processor to be executed
* **Data**: numbers/characters used as operands for instructions

### Memory Unit

Primary Memory  
Fast memory – operates at electronic speeds; volatile memory

* Programs are stored here during execution
* Processor controls reading/writing to memory
* Stored bits are handled in fixed-size groups called **words**
* **Address**: location in memory where a word is stored
* **Random Access Memory (RAM)**: memory where any location can be accessed in a short time, independent of location

Cache Memory   
Small, fast RAM unit that holds sections of program currently executing + its data

* Allows repeated data or instructions to be fetched quickly

Secondary Memory  
Cheaper, non-volatile, permanent storage

* Alternative to primary memory, which is expensive and doesn’t retain memory when power is off

### Arithmetic and Logic Unit (ALU)

* Performs addition, subtraction, multiplication, division, and comparison operations
* Required operands brought into processor and stored in **registers**, high-speed storage elements
* Result retained by processor or stored in memory

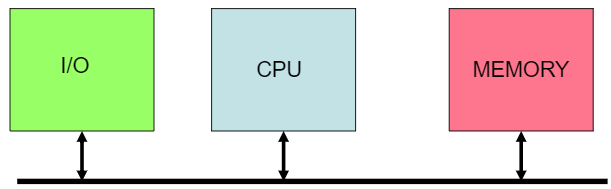
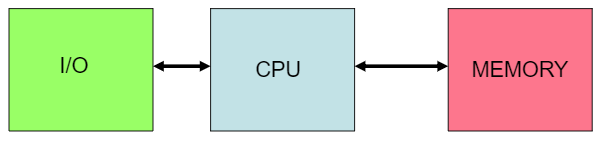
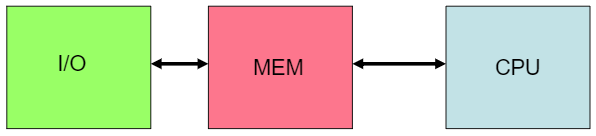
### Control Unit

* Sends control signals to other units and senses their states
* Connects ALU, memory, and I/O
* Generates **timing signals** to determine when an action should take place, govern data transfer
* **Control lines** (wires) carry timing signals and synchronization signals to all units

### Bus Structure

A group of lines that connect and transfer data between components in a computer

Two types of bus organizations:

* **Single Bus**: All components connected by a single set of wires   
  
* **Two Bus**: Components are connected by two independent buses  
   or 

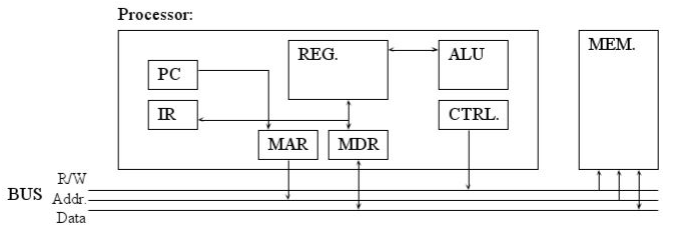
Types of buses:

* Data bus: carries data to and from memory and I/O
* Address bus: carries addresses of data in memory, e.g. source or destination of data
* Control bus: carries control signals between various components; controls use of data and address buses

### Computer Operation

1. Computer accepts programs and data through input unit and stores them in memory
2. Information stored in memory is fetched into ALU for processing
3. Processed information leaves computer through output unit
4. Everything is directed by control unit

## Basic Operational Concepts



Instruction Register (IR)   
Holds the instruction currently being executed

Program Counter (PC)   
Contains memory address of the next instruction to be fetched and executed

Memory Address Register (MAR)   
Stores memory address from which data will be fetched or sent/stored

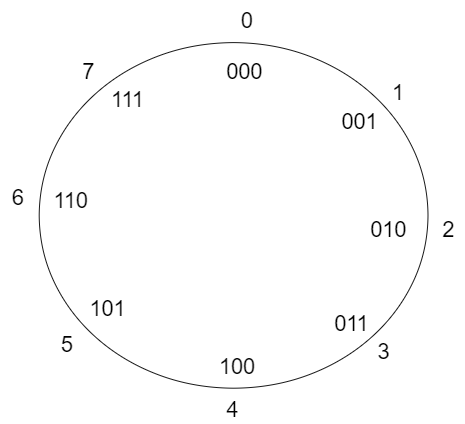
Memory Data Register (MDR)   
Holds data fetched from memory or data to be sent to memory

### Program Execution

1. Program transferred from input unit into memory; PC points at first instruction
2. “Read” control signal and address in PC sent to memory, initiating instruction fetch
3. Instruction transferred to IR, ready for execution; PC increments
4. Operands are read from memory and stored in process registers
5. ALU performs operation and sends result to a process register
6. Result is transferred to memory if necessary, with a “Write” command

## Number Representation and Arithmetic Operations

### Modular Arithmetic



* Addition is CW rotation, subtraction is CCW rotation
* When we cross between 111 and 000, generate a **carry**
* When we cross between 000 and 111, generate a **borrow**

### 1’s Complement

* The 1’s complement of an n-bit number N is

(“Flip the bits”)

* Leading 1 indicates a negative number
* Two representations of 0

### 2’s Complement

* The 2’s complement of an n-bit number N is

(“Flip the bits and add 1”)

* Leading 1 indicates a negative number
* Only one representation of 0

### Addition

* **1’s Complement:** If carry occurs, drop carry and add 1 to the answer (end-around carry)
* **2’s Complement**: If carry occurs, drop it (throw away carry)
* Overflow: two positive numbers produce a negative number
* Underflow: two negative numbers produce a positive number

# Chapter 2: Instruction Set Architecture

## 2.1 Memory Locations and Addresses

Memory consists of storage cells, each storing a bit (0 or 1)

* Generally organized into **words**, groups of n bits (typically 16-64 bits)
  + E.g. a 32-bit word can store a 32-bit signed number or four 8-bit ASCII characters
* Accessing memory to store/retrieve information requires distinct **addresses** for each location

### Byte-Addressable Memory

* Every memory address references successive byte locations in memory (1 byte = 8 bits)
  + Thus, if a machine’s word length is 32 bits (4 bytes), successive words are located at addresses 0, 4, 8
  + Words have aligned addresses if the begin at a byte address that is a multiple of the number of bytes in it

### Big-Endian

Lower byte addresses store the more significant (leftmost) bytes of a word

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Word address** | **Byte address** | | | |
| 0 | LSB  0 | 1 | 2 | MSB 3 |
| 4 | 4 | 5 | 6 | 7 |
| 8 | 8 | 9 | 10 | 11 |

### Little Endian

Lower byte addresses are used for the less significant (rightmost) bytes of a word

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Word address** | **Byte address** | | | |
| 0 | MSB  3 | 2 | 1 | LSB 0 |
| 4 | 7 | 6 | 5 | 4 |
| 8 | 11 | 10 | 9 | 8 |

Basic operations:

* **Load (Read/Fetch):** memory reads data stored at requested address and sends it to processor
* **Store (Write):** transfers data from CPU to specified memory location, overwriting former contents

## 2.3 Instructions and Instruction Sequencing

### Basic Machine Instructions

Instructions are characterized by the number of addresses (of operands) they require

* **3-address instructions (ADD A, B, C)**
  + Operation, Destination, Source1, Source2
  + Instructions are flexible, no overwriting of variables, but programs are very large due to multiple word instructions
* **2-address instructions (ADD A, B)** 
  + Operation, Destination, Source
* **1-address instructions (ADD A)** 
  + One of the operands is in a default location (e.g. a CPU register)
* **1½-address instructions (ADD A, Reg)**
  + One of the operands is in a named CPU register
* **0-address instructions (PUSH or POP)**
  + Operands are in a fixed location (e.g. a stack)f

### Reduced Instruction Set Computers (RISC)

* Each instruction fits in a single word
* Simple addressing modes
* **“Load/Store” architecture**
  + Memory operands only accessed with Load/Store instructions to move to/from registers
  + Operands for arithmetic/logic operations must be stored in registers or explicit values in instruction
* Small number of instructions
* Larger programs

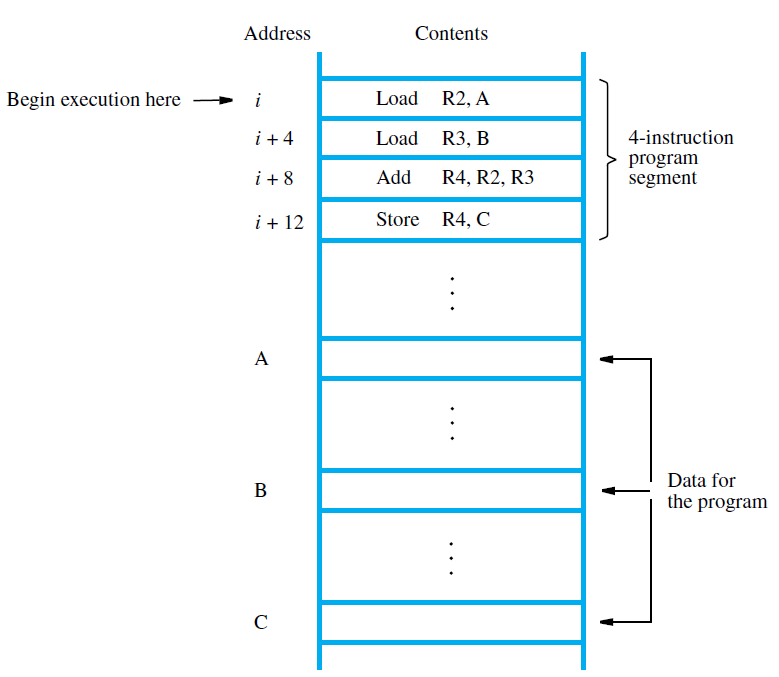
### Complex Instruction Set Computers (CISC)

* Instructions are multiple words
* Many complex addressing modes
* Operands can be in registers or memory
* Smaller programs

### Straight Line Sequencing

Processor control circuits use information in PC to fetch and execute instructions, one at a time, in order of increasing addresses

* Instruction fetch: instruction is fetched from memory address stored in PC, and placed in IR
* Instruction execute: instruction in IR is examined and specified operation is performed; PC increments



### Branching

* **Branch instructions** load a new address, the branch target address, into the PC
* **Conditional branch instruction** tests the contents of a Condition Code Register or Status Code Register, causing branching when a specified condition is met
* If not, PC is incremented normally to next instruction
* Common flags, set upon result of arithmetic/logic operations:
  + N – negative
  + Z – zero
  + V – overflow
  + C – carry

## 2.4 Addressing Modes

### Immediate

Operand value given explicitly in instruction

ADD R6, #200 (R6) 🡨 (R6) + 200

MOVE R1, #7 (R1) 🡨 7

### Register

Operand is contents of a register, whose name is given in instruction

ADD R2, R3 (R2) 🡨 (R2) + (R3)

MOVE R1, R2 (R1) 🡨 (R2)

### Absolute (Direct)

Operand is contents of a memory location; address of location (typically a global variable) given in instruction

MOVE R2, LOC (R2) 🡨 (LOC)

### Indirect

Operand is contents of a memory location, address stored in a register or (another) memory location

ADD R4, (R3) (R4) 🡨 (R4) + ((R3))

ADD R4, (LOC) (R4) 🡨 (R4) + ((LOC))

### Index

Operand is contents of a register/memory location, address stored in a register, plus an offset

ADD R5, 8(R8) (R5) 🡨 ((R8) + 8)

MOVE R3, 6(R0) (R3) 🡨 ((R0) + 6)

### Relative

Operand is contents of memory address in PC, plus an offset

BNE LOOP if Z = 1, (PC) 🡨 (PC) + appropriate #, else (PC) 🡨 (PC) + offset X

### Base with Index (and Offset)

Operand is contents of the sum of two addresses stored in registers, plus offset if given

MOVE R3, (R2, R1) (R3) 🡨 ((R1) + (R2))

MOVE R3, 2(R2, R1) (R3) 🡨 ((R1) + (R2) + 2)

### Auto-Increment (Post Increment)

Operand is address stored in register; address is automatically incremented after operation execution

ADD R3, (R2)+ (R3) 🡨 ((R2))  
 (R2) 🡨 (R2) + increment

### Auto-Decrement (Pre Decrement)

Operand is address stored in register, which is automatically decremented before operation execution

ADD R3, -(R2) (R2) 🡨 (R2) – increment  
 (R3) 🡨 ((R2))

## 2.5 Assembly Language

* Machine instructions are patterns of 1’s and 0’s
  + When writing programs, we use mnemonics to represent operations
  + Assemblers translate assembly language programs into machine language
* **Assembler directives** let us control certain parts of the assembly process
  + MYMEM DCD 0 associates MYMEM with a memory location, sets the location value to 0
  + NINE EQU 9 associates the value 9 with the label NINE

## 2.6 Stacks

* Ordered list of elements
* **Stack pointer (SP):** process register that points at the processor stack and keeps track of the top element (TOS) at all times
  + In ARM, the SP is R13
* **Push**: moves a new operand to the top-of-stack (TOS); the previous element is moved down

STR Rj, [R13, #-4]!

or

STMFD R13! {Ri, Rj}

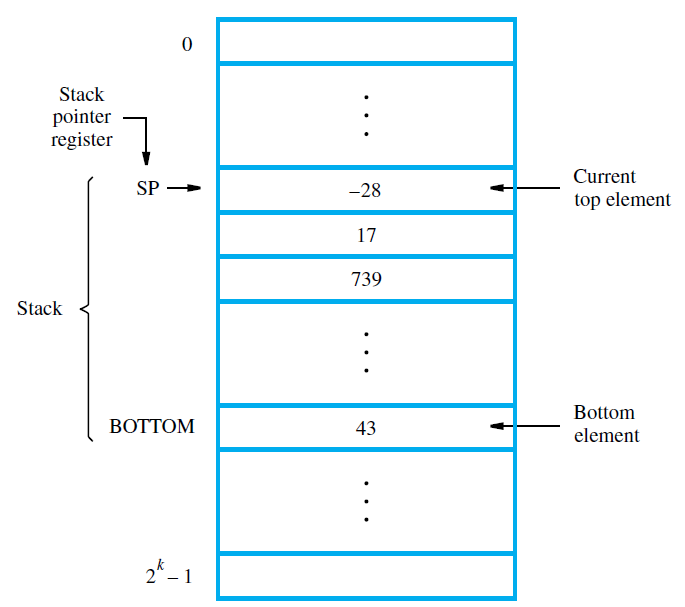
* **Pop**: removes the top element and moves the next lower element to the TOS

LOAD Rj, [R13], #4

or

LDMFD R13!, {Ri, Rj}

* Data can be stored on a stack in successive memory locations
  + In ARM, the stack is usually placed in higher memory and programs in lower memory



## 2.7 Subroutines

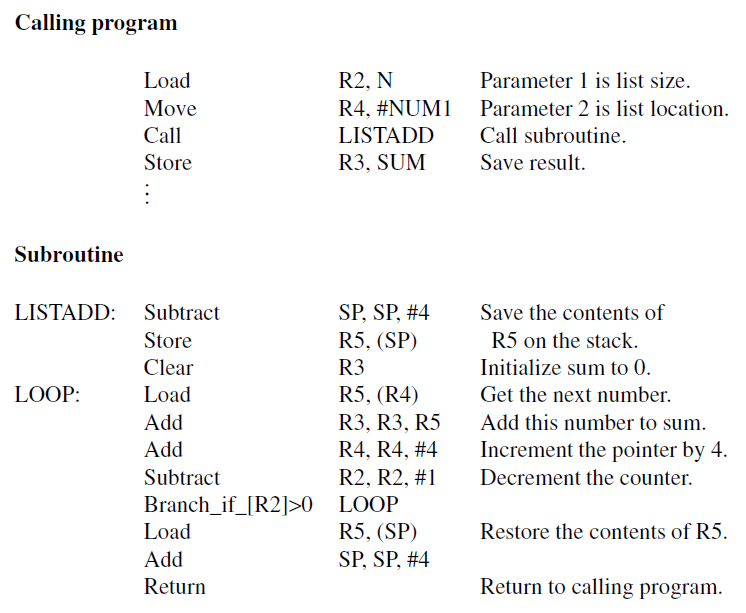
* Performs a particular task many times with different values
  + Avoids duplication, saving space since only one copy of instructions is stored in memory
  + Produces more compact (albeit slower) code
  + Reuses code
  + Enables modular approach to programming
* **Call instruction**: program branches to a subroutine
  + ARM uses Branch and Link (BL) instruction for subroutine calls
* **Return instruction**: return from subroutine to the program that called it

### Subroutine Linkage

* Use fixed memory location to store return address
* Use a link register to store the return address
* **Stack linkage**
  + Call instruction pushes next sequential instruction address (from PC) onto the stack
  + Return instruction pops instruction address back into PC

### Parameter Passing

* Several ways to pass parameters to/from subprograms:
  + Fixed locations – good if you know them ahead of time
  + Registers – good if you have lots of them
  + Push/pop onto the stack
    - Load parameters onto the stack, access them in the subprogram, return results to the calling routine, and clean up stack
* Pass by value: pass actual value of parameter to subroutine
* Pass by reference: pass the address of the parameter to subroutine



### Subroutine Nesting

* When calling a subroutine from another subroutine, the contents of the link register should be stored onto the stack frame for that subroutine, before calling the next one
* When completed, the return address should be restored onto the link register

# Appendix D: ARM Processor

* 32-byte word length / byte-addressable memory addresses / registers
  + Three operand sizes: Byte (8 bits), Half-word (16 bits), Word (32 bits)
* MOVE, STORE commands for accessing memory
  + Auto-decrement, Auto-increment, and PC relative modes also available
  + Multiple registers can be loaded from/stored into a block of consecutive memory words
* Fifteen general-purpose registers (R0 – R14) and a 32-bit program counter (R15)
* 32-bit Current Program Status Register (CPSR) that holds condition flags (N, Z, C, V) amongst other things
* Six operating modes:
  + User
  + Fast Interrupt Mode (FIQ)
  + Normal Interrupt (IRQ)
  + Supervisor
  + Abort (handles memory access violations)
  + Undefined (handles undefined instructions)

## ARM Indexing Modes

* **Immediate** 
  + 12 or 16 bits available for an immediate value
  + MOVW – moves to bottom 16 bits of register
  + MOVT – moves to top 16 bits of register
* **Register**
  + Basic mode for arithmetic/logic operations
* **Absolute**
  + Load/Store operations
* **Basic Indexed Addressing**
  + Pre-indexed mode

LDR R1, [R2, #4]

LDR R1, [R2, R3]

LDR R1, [R2, -R3]

* **Relative**
* **Indexed Address Mode with Writeback**
  + Auto update of addresses (e.g. going through a list)
  + Pre-index with Writeback

LDR R1, [R2, #4]!

(R1) 🡨 ((R2) + 4)

(R2) 🡨 (R2) + 4

* + Post-index with Writeback

LDR R1, [R2], #4

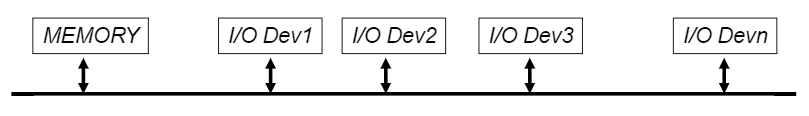
(R1) 🡨 ((R2))

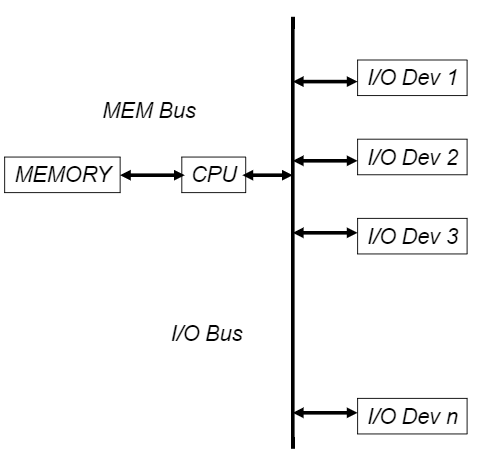
(R2) 🡨 (R2) + 4

# Chapter 3: Basic Input and Output

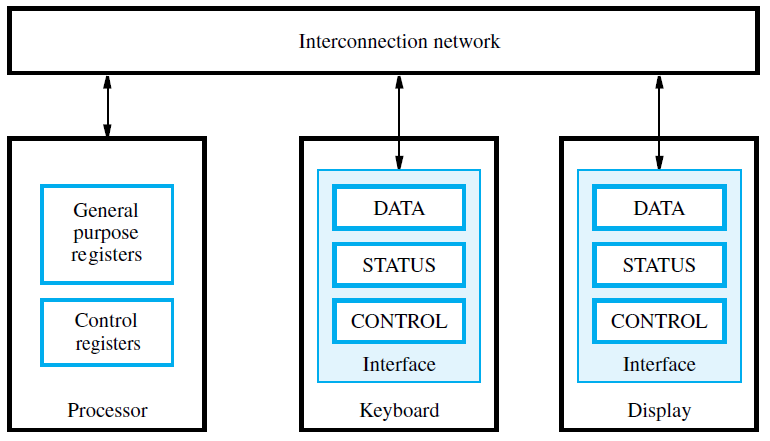
## 3.1 Accessing I/O Devices

**Single Bus** – Memory and I/O share the same bus



**Two Bus** – Memory and I/O are connected to CPU by separate buses

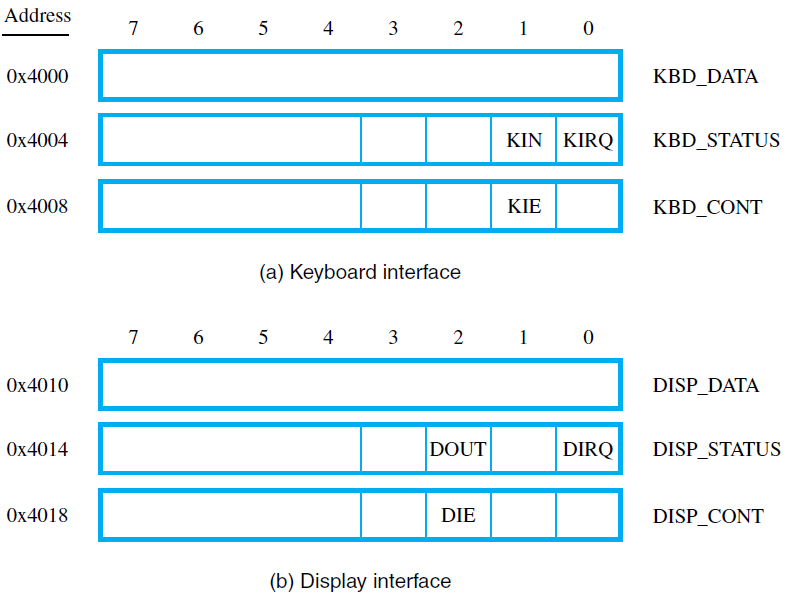
* I/O registers: addresses in the processor that are assigned to I/O locations
* **Memory-mapped I/O**: Each I/O device appears to the processor to contain addressable locations, like memory; I/O and memory share the same address space
  + Any instruction that can be used to access memory (e.g. Load/Store), can be used to transfer data to/from I/O devices



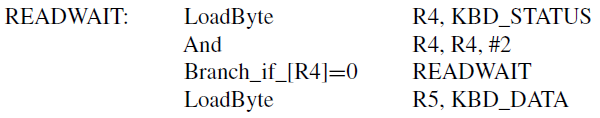
* The **I/O interface** of a device connects it to the bus
  + One side: bus lines for address, data, and control
  + Other side: connections to transfer data between I/O device and interface

### Program-controlled I/O

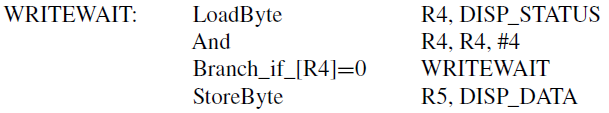
Program controls all aspects of I/O operation



* Processor *polls* device – checks device’s status flag



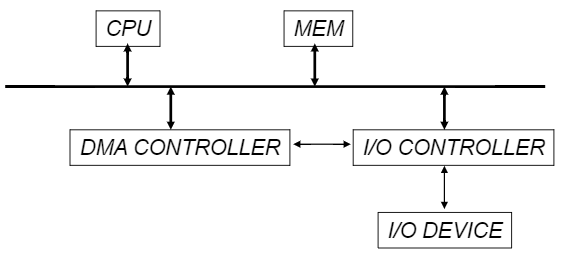
* + Load contents of KBD\_STATUS into process register R4
  + Test the KIN flag, which is bit b1 of the status information in R4
  + Keep executing READWAIT until status bit b1 = 1, then load KBD\_DATA into R5

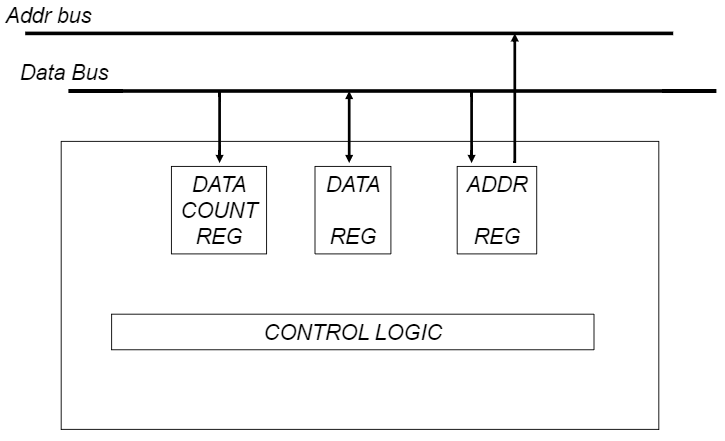


* + Load contents of DSP\_STATUS into process register R4
  + Test the KIN flag, which is bit b1 of the status information in R4
  + Keep executing WRITEWAIT until status bit b1 = 1, then load DISP\_DATA into R5
* Processor transfers data in/out or out and clears the flag
* **Advantages**
  + Simple
  + Cheap
* **Disadvantages**
  + Slow – CPU controls all operations so it is tied up
  + A lot of instruction overhead per byte transferred
  + Higher speed devices can overrun the system

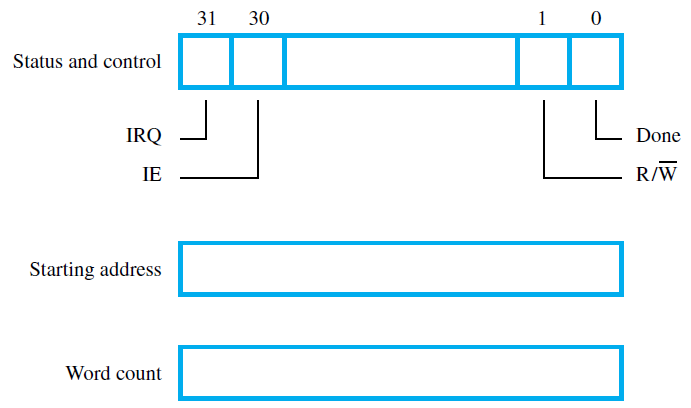
### Direct Memory Address

Controls transfer of blocks of data between Memory and I/O devices without frequent program-controlled intervention by processor





* CPU doesn’t require direct memory address every cycle
* If there is a large volume of data to/from sequential locations in memory, **DMA controller** can take control of memory bus, store data from I/O device directly into memory, release the bus and wait for next byte/word from I/O device
  + DMA controller:
    - Provides memory address and control signals
    - Increments memory address for successive words
    - Keeps track of number of transfers
  + CPU provides:
    - Memory start address
    - Word count
    - Read/Write function
    - Address of I/O device



* + When the DMA controller is done transferring a block of data, it sets the Done flag to 1, along with the Interrupt Enable (IE) flag, which will raise an interrupt; when an interrupt has been requested, it sets the Interrupt Request (IRQ) flag to 1
* **DMA Cycle**

1. CPU: loads DMA controller with parameters, issues GO to DMA controller
2. Device: Data Ready, requests DMA operation
3. DMAC: Bus Request
4. CPU: Bus Grant
5. DMAC: becomes Bus Master, places address and R/W on bus
6. Device: puts data on bus, removes request
7. DMAC: after memory function completed, removes Address
8. Device: removes data
9. DMAC: updates memory address and word count, removes Bus Request so CPU becomes Bus Master again
10. Device: gets next word
11. DMAC: checks if word count = 0; if so, send interrupt to CPU indicating task complete

* **Advantages**
  + Good for data stored in continuous memory locations
* **Disadvantages**
  + Expensive
  + DMA controller must have facilities to generate addresses and control signals, and drive data bus
  + Bus arbitration required to prevent multiple devices from trying to drive bus simultaneously
  + How does DMA get control of the bus?
    - Cycle stealing: take over the bus for each byte of data to be transferred, then return control to the CPU
    - Burst mode: take over bus to transfer a block of data, then return bus control to CPU
    - Special DMA memory

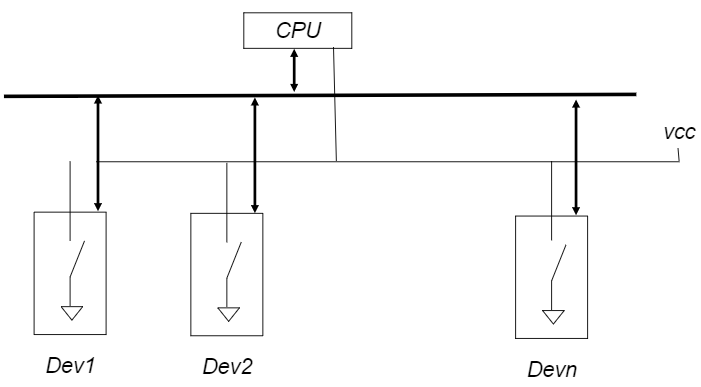
## 3.2 Interrupts

* If device could signal CPU when it is ready, CPU doesn’t have to wait while checking device status
* CPU can do normal processing, then run **Interrupt Service Routine (ISR)** when device signals it is ready through an **interrupt request**
* Process:
  1. Complete current instruction
  2. Save processing environment – PC and **status register** PS on stack
  3. Go to ISR and service interrupt
  4. Return and restore environment

### Single Interrupt Request (IRQ) Line

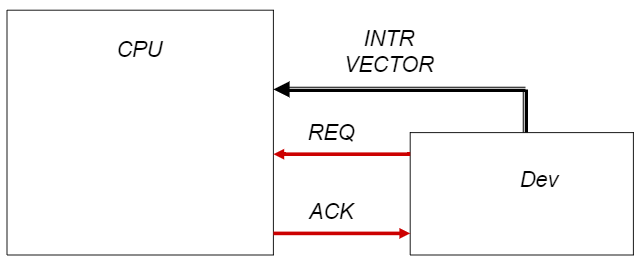
CPU polls status register of each device for IRQ bit; assigns priority by order

* Easy to implement, but waste of time to poll devices that didn’t send requests



### Multiple Interrupt Lines

Have several IRQ lines, each with assigned priority; service higher priority requests first

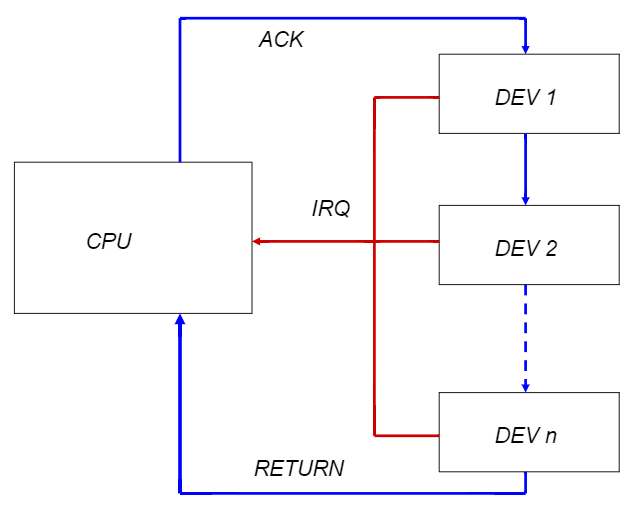


* **Vectored Interrupts**: when devices request an interrupt, they identify themselves to processor and the start address of the interrupt service routine
* **Interrupt Acknowledge**: control signal sent by processor to inform devicethatits IRQ has been received, so that device removes interrupt request signal

### Daisy Chain (Hardware Polling)

Allows for multiple requests, priority assigned by physical order of devices (device closest to CPU gets serviced first)

* Devices share the Interrupt Request line
* Interrupt Acknowledge line connect devices in predetermined order



Many systems combine methods:

* Multiple IRQ lines
* Daisy-chained interrupt acknowledgement
* Vectored interrupts for each device

### Controlling Interrupt Requests

**CPU**

* Enable interrupts
* Priority assignment
* Masking – CPU may only listen for certain priorities

**Device**

* Enable interrupt requests

## D.7 ARM Exceptions

* The ARM has two “normal” interrupt lines, I and F, which can be disabled in the status register
* User mode is not privileged – cannot manipulate these bits
* The five exception modes and System mode (which can only be entered from an exception) are privileged
* The exception modes, in order of priority:
  + **1** **– Supervisor (SVC)**: power-up, reset, or program executes software interrupt instruction for operating system routine
  + **2 – Memory access violation (Abort)**: attempt to fetch instruction/data operand causes memory access violation
  + **3 – Fast interrupt (FIQ)**: fast-interrupt requests for urgent service
    - General registers R8-R12 and R13/R14 are replaced with banked registers (different copy of register at same address) – any changes to these registers will not affect user registers after exception service complete
  + **4 – Ordinary interrupt (IRQ)**: normal interrupt requests
    - Only R13/R14 are replaced – any registers used in the exception service must be saved/restored
  + **5 – Instruction access violation (Abort)**: attempt to access instruction not permitted
  + **6 – Unimplemented instruction (Undefined)**: current program attempts to execute unimplemented instruction

## 7.4.2 Interface Circuits

### Parallel Transfers

* Transfers multiple bits of data simultaneously, over separate wires
* Advantage: high speed
* Disadvantages: expensive, timing problems with long distances

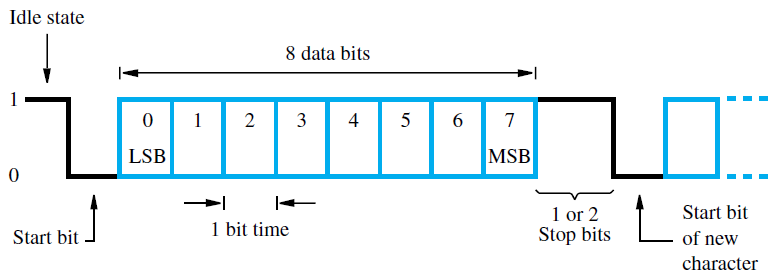
### Serial Transfers

* Transfers data one bit at a time, over a single wire
* Advantage: less costly over long distances
* Disadvantage: slow speed
* Connection are classified by their type and direction
  + Simplex: information can only be sent in one direction
  + Half Duplex: information can be sent one direction or the other, but only one at a time
  + Full Duplex: data can be sent both ways simultaneously

**Synchronous Serial Transmission**

* Transmitter sends data along with clock signal, so receiver knows when it is valid
  + Receiver generates clock that is synchronized to the transmitter’s
* Higher speed, but requires additional lines
* Speed can be varied

**Asynchronous Serial Transmission**



* No clock signal is sent
* Sender and receiver agree on a baud rate (bps), the duration of each bit
* Start/stop transmission
  + 1-to-0 transition at beginning of Start bit (0) alerts receiver that data transmission is about to begin
  + Using own clock, receiver determines positions of next 8 data bits and loads them into input register
  + 0-to-1 transition with Stop bits (1) ensure that the next Start bit will be recognized

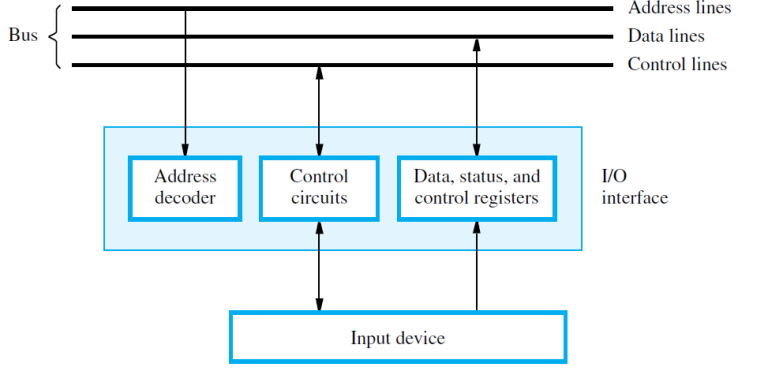
# Chapter 7: Input / Output Organization

## 7.2 Bus Operation

Bus connections make up the interconnecting network between memory, the CPU, and I/O devices

There are three lines that make up the Bus:

* Data
* Address
* Control



**Bus protocol**: a set of rules that govern how a bus is used by devices; implemented by control signals that indicate when actions should be taken

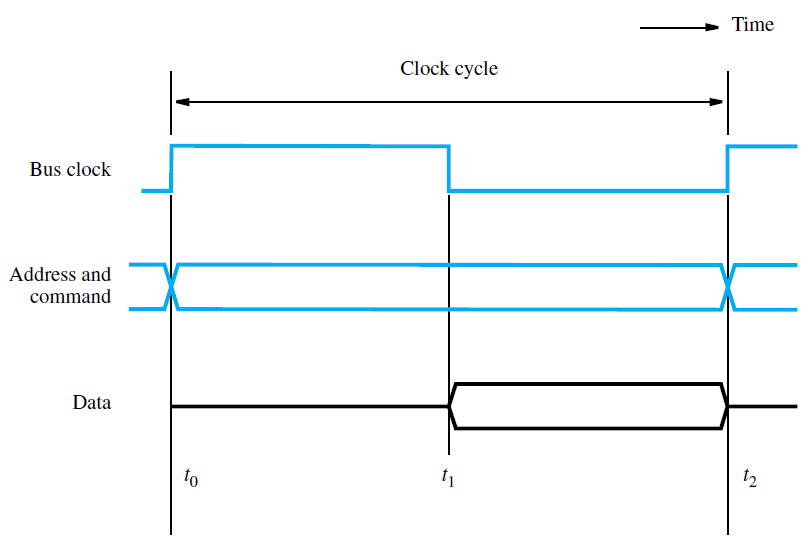
**Bus Master**: the device that controls the bus – only one at a time

## Bus Transfers

The timing of data transfers over a bus can be either Synchronous or Asynchronous

### Synchronous Bus Transfer

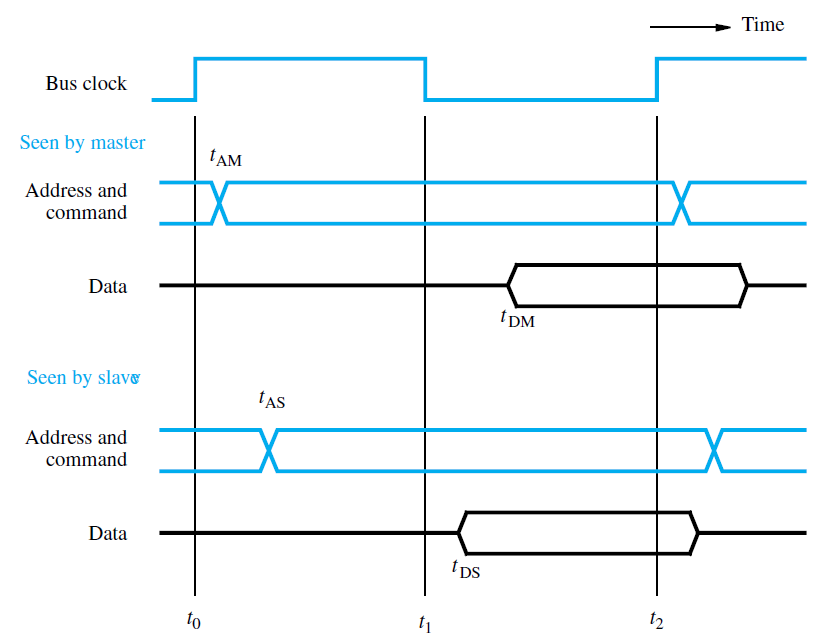
* All devices derive timing information from common bus clock control line
  + Clock cycle: high level signal followed by low level signal
  + Equally spaced clock pulses define equal timing intervals
  + Must be long enough to accommodate slowest device
* One data transfer takes place per bus cycle



2) Addressed slave puts data on bus data lines

3) Master strobes the data into its input buffer for Read operation

1) Master puts address and command on the bus   
(e.g. Read)

* In reality, delays cause different parts of the circuit to see signals at different times

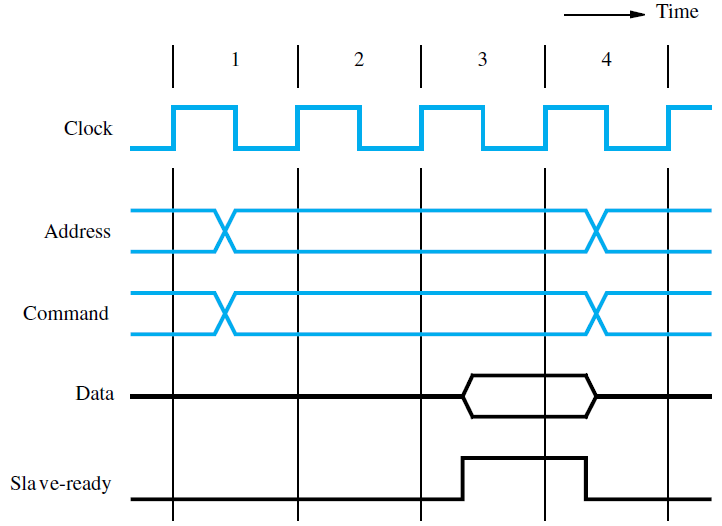
4) Data reaches master

3) Data appears on bus

2) Address and command reach slave

1) Address and command appear on bus

* Memory usually runs much slower than CPU – by about 4-5 clock cycles
  + Previous case is simple but results in slow transfer rate
  + Add signal to allow device to tell master when it is ready



3) Master strobes the data into its input buffer

2) Addressed slave puts data on bus data lines, sends slave-ready signal

1) Master puts address and command on the bus   
(e.g. Read)

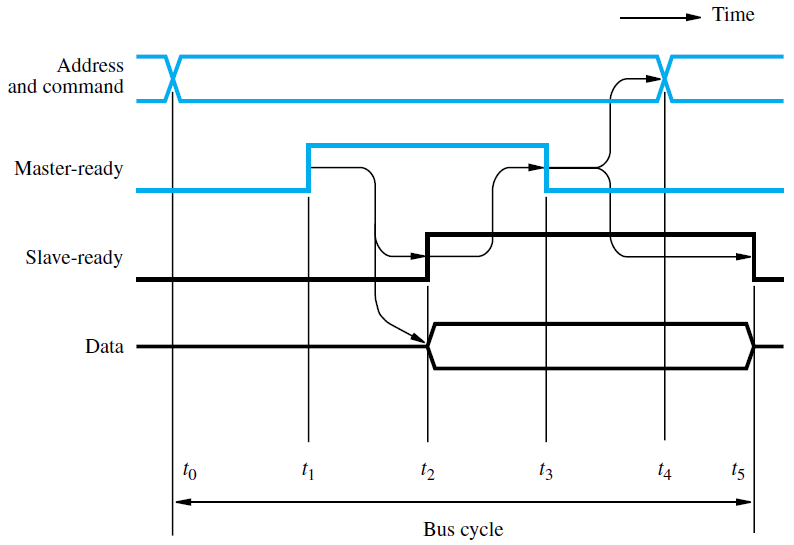
### Asynchronous Bus Transfer

* No central clock – data transfer controlled by handshake between master and slave
* Process:

1. Master places address and command on bus
2. Master sends master-ready signal

* Gap between 1 and 2 to allow for **bus skew** – two signals transmitted simultaneously reach destination at different times
* t1 should be larger than maximum skew

1. All devices on the bus decode the address
2. Addressed slave performs required operation, sends slave-ready signal when finished
3. Master removes all signals from bus once slave-ready signal is asserted, strobes slave data into input buffer if necessary



t5: Slave receives transition of master-ready signal; it removes data and slave-ready signal

t0: Master puts address and command on the bus

t4: Master removes address and command; t4-t3 allow for bus skew

t3: Master receives slave-ready signal, removes master-ready signal

t2: Slave decodes address, places data on bus and asserts slave-ready signal

t1: Master asserts master-ready signal

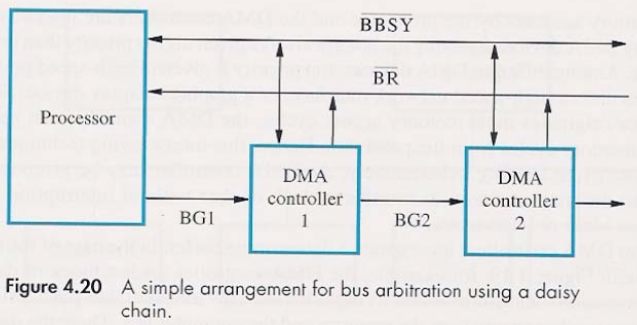
## 7.3 Bus Arbitration

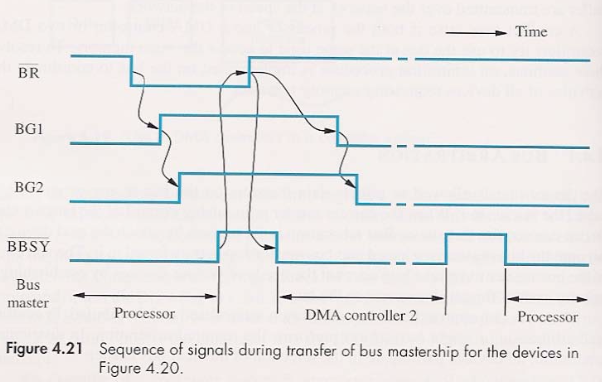
When multiple devices can control the bus, arbitration is required to decide who will control it when

* Each device sends a request for a slave
* Arbiter circuit determines priority for each request; if two requests are received at the same time; use of the slave is granted to device with higher priority

### Centralized Arbitration

* CPU (or arbiter) supervises control of the bus
* Example: multiple DMA controllers
  + DMA controller requests bus control by sending BUS REQUEST (BR)
  + CPU activates BUS GRANT (BG1)
  + Bus is indicated in-use by BUS BUSY (BBSY) signal





1) DMA controller 2 asserts Bus Request (BR)

3) BG1 propagates to DMA Controller 2 via BG2

2) Processor activates Bus Grant (BG1)

4) Processor releases control of Bus by setting Bus Busy (BBSY)

### Distributed Arbitration

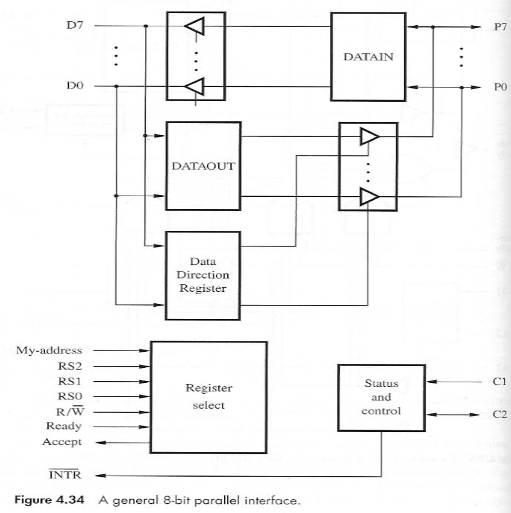
* If devices are peers, arbitration can be done without a central controller
* All devices are connected by 5 lines – 4 arbitration lines for 4-bit IDs, 1 line for Start Arbitration
* “Competition” begins when one or more devices activate the Start Arbitration signal
* Each device “bids” for control of the bus by placing bits of its ID on an arbitration bus
* All competing devices look for their address on the bus
  + The logical OR of all the IDs placed on the arbitration lines
  + Each device compares the pattern to its ID; if it detects a difference, it transmits 0s on the arbitration lines for that and all lower bit positions
  + The highest address wins control, and then drops out of competition until all other devices have had a chance to control the bus
  + Example:
    - Device A has ID 5; requests bus by placing 0101 on arbitration lines
    - Device B has ID 6; requests bus by placing 0110 on arbitration lines
    - 0101 OR 0110 = 0111 appears on arbitration lines
    - Device A compares its ID (0101) to 0111 – detects a difference at bit 1, so transmits 0100
    - 0100 OR 0110 = 0110 appears on arbitration lines
    - Device B’s ID is same as arbitration line, so it wins

## 7.4 Interface Circuits

* Inputs to registers can always listen to the bus, are only clocked when they are addressed
* Outputs from the register are only transferred to the bus when required (tri-state drivers)
* Each device requires address decoding and control signal generation
* I/O interface consists of:
  + Side that connects to the computer, which contains bus signals
  + Side that connects to the I/O device, which has a port that controls data transfer between interface and device

### Parallel Ports

* Transfer data in a number of bits, typically 8 or 16
* Multiple pin connector between device and processor + cable with as many wires as number of simultaneously transferred bits
* Each input/output can be programmed



My-address is connected to the output of an address decoder

Ready/Accept lines are the handshake control lines on the processor side – connected to Master-ready and Slave-ready signals

C1 and C2 control interaction between interface circuit and I/O devices

Processor places 8-bit pattern in Data Direction Register (DDR), who determines which data lines are used for input and output: 1 = output, 0 = input

Data lines to I/O device are bidirectional – data lines P7 to P0 can be used for both input and output

### M68230 Parallel Interface/Timer

* Three 8-bit bidirectional I/O ports
* 24-bit timer
* Port General Control Register (PGCR) controls all the ports
* Direction control registers

## 7.5 Interconnection Standards

* Allow devices from different manufacturers to interact
* Allow different types of devices to operate on the same system

### RS 232C

* Standard for serial communications
* Synchronous and asynchronous modes
* Specifies electrical, physical, mechanical, and procedural aspects of communications

### Nubus

* Designed for high speed, low cost
* Up to 16 devices on one backplane (parallel electrical connectors)
* 10MHz clock
* 32-bit shared address space
* Single master – all other devices are slaves

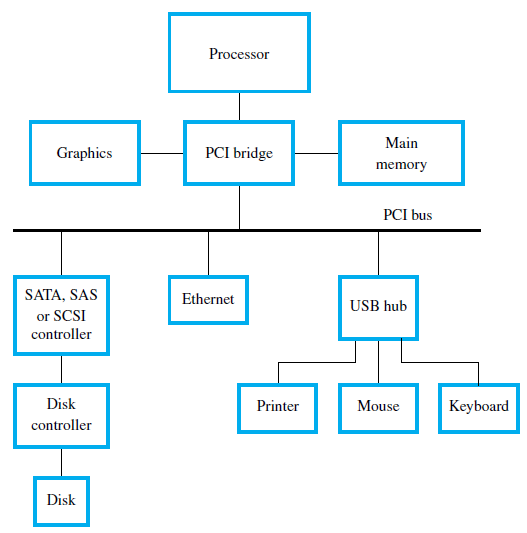
### Multibus

* Computer bus standard developed by Intel
* 8, 16, 32 bit asynchronous transfers
* IEEE 796 (16-bit)
* IEEE 1296 (32-bit)

### IEEE 488

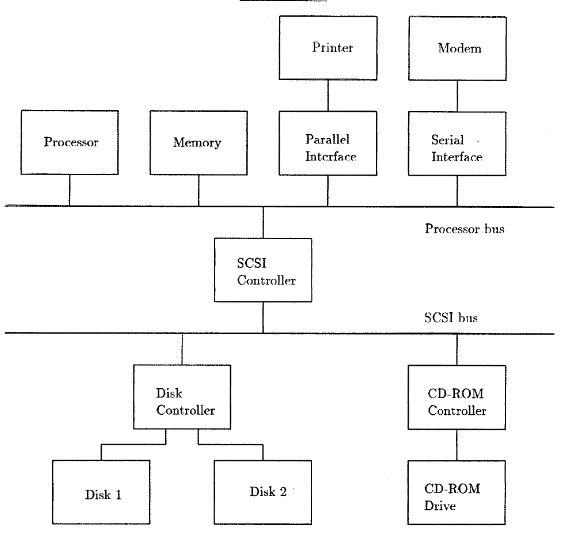
* Standard for lab instrumentation
* 8-bit parallel (250 KB/sec, 20 meters)
* Modes:
  + Listener
  + Talker
  + Controller

### Peripheral Component Interconnect (PCI) Bus

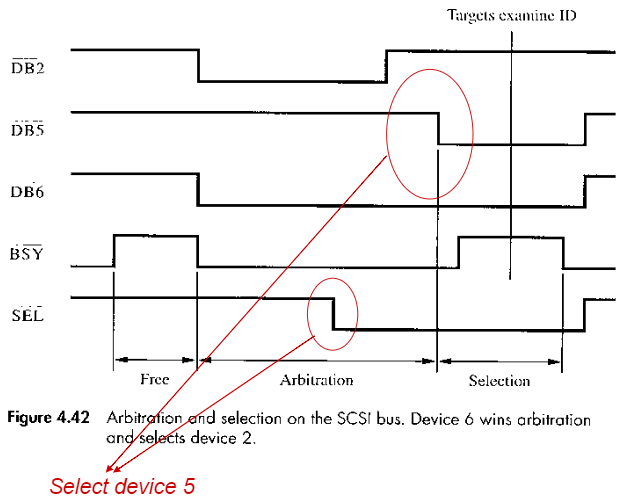


* Connected devices appear as if they are connected directly to processor bus
* First “plug-and-play”: connect device to bus, software takes care of the rest
* CPU has one or more layers of cache
* Transfers from main memory take place in “burst mode”, transfers of multiple successive words; single word = burst of length 1
* PCI bus connected to processor via PCI bridge controller, which has a special port that connects to the computer’s main memory
  + Bridge relays commands, responses, and data from one bus to another
* PCI supports 3 independent address spaces – 4-bit command accompanies data transfer to indicate which space to use:
  + Memory
  + I/O
  + Configuration – intended for plug-and-play capability
* Signalling – master/slave relationship
  + Bus Master is **initiator** that initiates data transfer with Read/Write commands
  + The addressed slave is a **target**
  + **Transaction**: a transfer operation with an address and burst of data

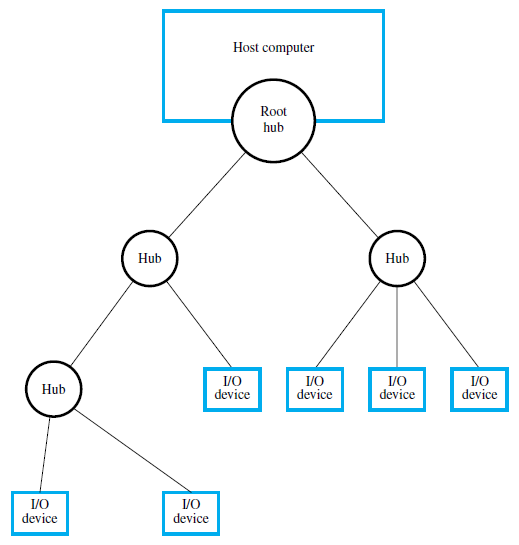
### Small Computer System Interface (SCSI)



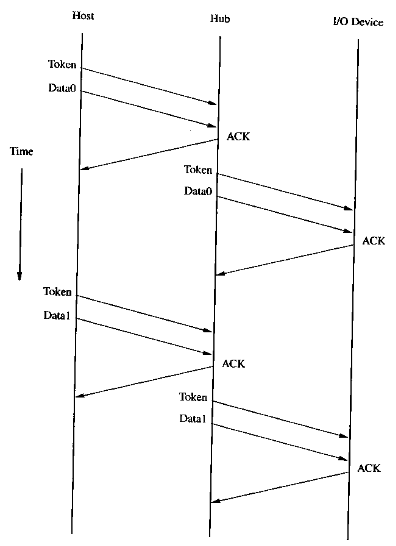
* 50, 68 or 80 pins wire connection up to 25 meters
* 8 or 16 bit wide transfers
* Up to 620 MB/sec – depends on number and length of cables (fewer/shorter = higher rate)
* Address space not part of processor address space
* Up to 8 or 16 devices can be connected
* Acts as a DMA device to/from memory
  + Device is either an initiator or target
  + Initiator has ability to address a specific target
  + No address lines – data lines used to identify devices
* Bus control decided by arbitration – devices bid for control of bus by placing its ID on the 8/16 data lines
* Master/slave relation to transfer data
  + BSY signal released when data transfer finished



### Universal Serial Bus (USB)



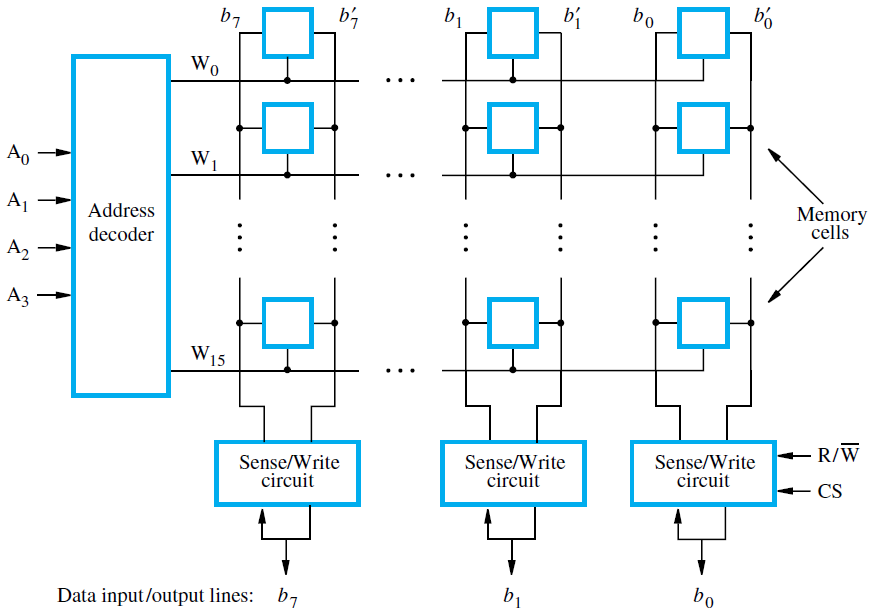
* Designed to support wide range of I/O devices
  + Plug-and-play capabilities
* Initially two speeds: 1.5 Mbps (low speed) and 12 Mbps (high speed)
* USB 2.0: 480 Mbps, USB 3.0: 4.8 Gbps
* Data transferred in serial form – clock and data combined to prevent skew problems
* USB uses a tree architecture:
  + User devices connect to a **hub**; **root hub** connects entire tree to computer
  + I/O devices are **functions**
* Data flow: hub copies any message it receives on its upstream connection to all of its downstream connections
  + E.g. message from host computer is broadcast to all I/O devices, but messages from I/O devices are only sent upstream
* Arbitration: Hub is master, all I/O devices are slaves that can only respond when polled (no real arbitration required) – allows for simple, inexpensive hubs
* Addressing: each device assigned 7-bit address local to USB tree
  + When new device detected by hub, it records this in its status information
  + When host polls hub and detects new device, handshake takes place to determine device characteristics
* Two “packets” exchanged:
  + **Control**: address, acknowledge, errors, etc.
  + **Data**: actual data
  + Each packet has PID that identifies its type
* Host initiates transmission with a token, then data packets
* Hub checks for errors, then forwards transmissions downstream to devices



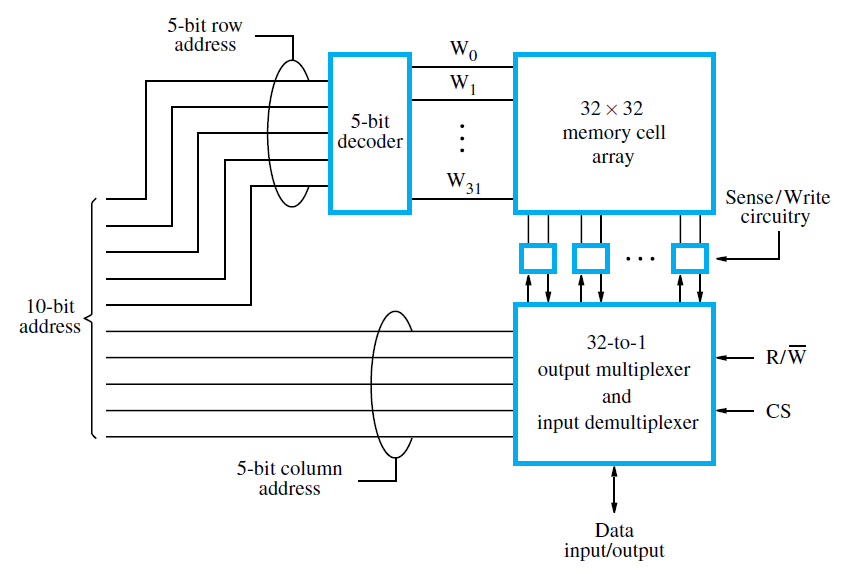
* **Isochronous data transfer** (occurs at regular intervals):
  + USB creates 1ms frames, initiated with Start of Frame (SOF) packet
  + Devices then send data on a regular basis

# Chapter 8: Memory Systems

* Memory is built from a collection of memory cells (1 cell = 1 bit)
* Cells are grouped into row of words
  + All cells in a **row** are connected to a common word line
  + All cells in a **column** are connected to a Sense/Write circuit by two bit lines
* A memory address returns a group of memory cells

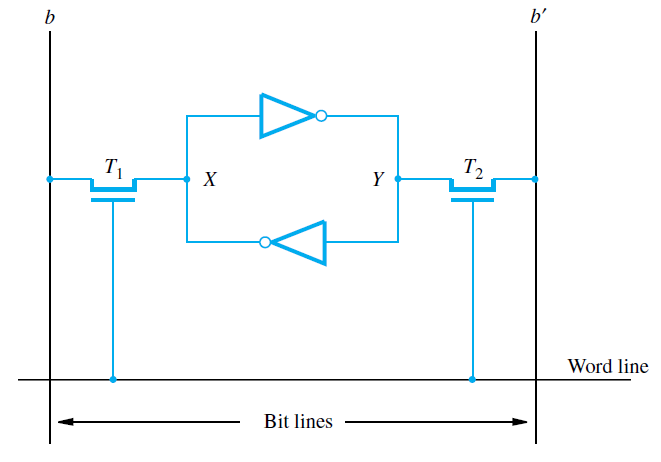


## 8.2 Semiconductor RAM Memories



* **Random Access Memory (RAM)**: memory unit where any location can be accessed in a fixed amount of time, regardless of address
* Transistors used to form memory cells – VLSI memory allowed many to be placed in one chip
* Individual cells placed in a 2D array – each address accesses one bit in the array
  + As size of memory gets larger, address decoding becomes more complex

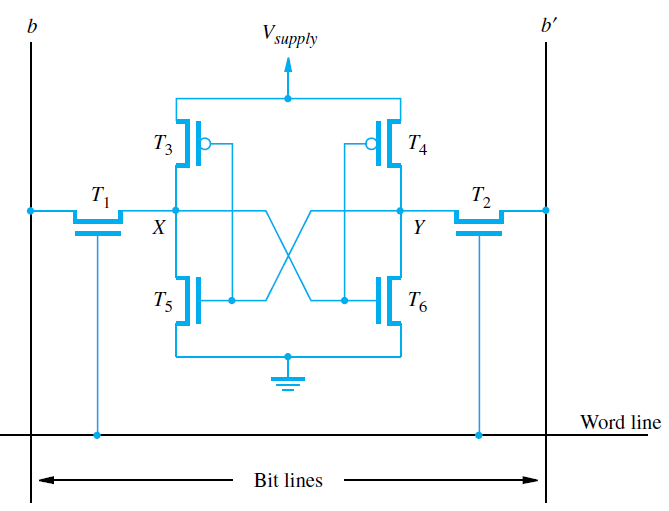
### Static RAMs (SRAMs)



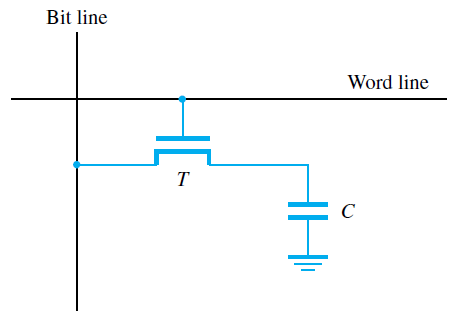
* Volatile memory that maintains contents as long as power is supplied
* Two back-to-back inverters form a latch
* Two other transistors connect/disconnect cell from complementary bit lines
  + **Read operation**: word line activates to close T1 and T2
    - b and b’ always complement each other, e.g. state 1: b signal = high, b’ signal = low
    - Sense/Write circuit at the end of the two bit lines monitor their state and set corresponding output accordingly
  + **Write operation**: Sense/Write circuit drive b and b’ (instead of sensing their states) by placing appropriate values them, then activates the word line
* Low power consumption – no current flow when cell inactive
* Very fast access time (nanoseconds)
* High cost

### CMOS Cell

* (T3, T5) and (T4, T6) pairs form the inverter latch
* State 1: voltage at X is high, T3, T6 on while T4, T5 are off – if T1 and T2 are turned on, b signal = high and b’ signal = low
* Low power consumption – current only flows in cell when being accessed; otherwise, T1 and T2 are off, so no electrical path between Vsupply and ground

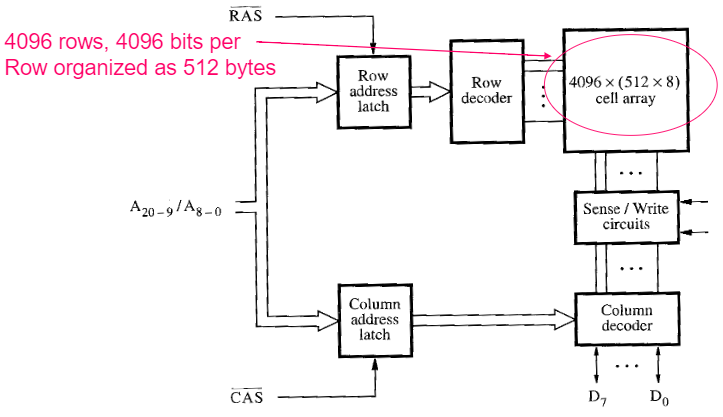


### Dynamic RAMs (DRAMs)



* Do not retain state indefinitely unless they are accessed often for Read/Write operations
* Memory stored as charge in capacitor; cell formed with isolation transistor and a capacitor
* Contents must be periodically refreshed by restoring capacitor charge to full, when accessing for read/write operations
* **Store operation**: T is turned on and appropriate voltage applied to bit line, causing known amount of charge to be stored in capacitor
  + After transistor turned off, capacitor begins to discharge, so information stored in cell can only be retrieved if correctly if read before charge drops below threshold
* **Read operation**: T is turned on; sense amplifier detects whether charge in capacitor is above or below threshold value
  + **I**f above, amplifier drives the bit line to full voltage, charging the capacitor full
  + If below, amplifier pulls bit line to ground to fully discharge capacitor

### Asynchronous DRAMs



* Address usually broken up into row/column components
* Identify which row/column to be latched with **Column Address Strobe (CAS)** and **Row Address Strobe (RAS)** controlsignals
* Asynchronous timing – must allow for delays in circuitry
* All cells in the row are selected and read, but only 8 bits put out on the data lines – the column address
* **Fast page mode**: sequential bytes in selected row are transferred in sequential order, through applying a consecutive sequence of column addresses under the control of successive CAS signals (without reselecting the row)
  + Allows block of data to be transferred a lot faster than random accesses

### Synchronous DRAMs

* Memory access is directly synchronized to clock signal
* Data and addresses buffered in latches
* Burst mode: all contents of a row are loaded into latch and output sequentially
* Built-in refresh circuitry: refresh counter keeps track of which rows need to be refreshed (every 64 ms)

**Latency**: the amount of time it takes to read/write a byte/word of data to/from memory

* Worst case: single word
* Burst mode is better, but depends on overhead to start transfer (large) and time to transfer successive words (short)

**Bandwidth**: the amount of data that can be transferred per unit time

* Depends on speed of access to data and number of bits that can be transferred in parallel

### Double-Data-Rate SDRAM

* Two interleaved memory banks
* Transfers data on both edges of the clock
* Same latency as SDRAM – double bandwidth at best
* Most effective for large block transfers (no advantage for individual word transfers)

### Content Addressable Memory (CAM)

* Supply data word
* Memory is searched – if present, returns list of locations where it can be found
* **Ternary CAM**: addition of ability to include “Don’t Care” for some bits in search

### Multi-Port Memories

Some memory systems (cache applications) allow multiple ports to access memory

* Very expensive

### RAMBUS

* Uses differential signalling +/- 0.3V around a reference voltage
* Bus width: 8 or 16 bits
* Uses packets for transfer – no separate address lines

## Read Only Memories (ROM)

* Memory can only be read – data is written into ROM when manufactured

### Programmable ROM (PROM)

* Data can be loaded by user
* Process of inserting data is irreversible

### Erasable PROM (EPROM)

* Similar to ROM, but cells can be programmed on or off
* Contents can be erased through UV light exposure and reprogrammed

### Electronically Erasable PROM (EEPROM)

* Apply a special voltage to erase program contents – doesn’t have to be removed from circuit
* Flash is an extension of EEPROM, but cells are written in blocks

### Memory Systems

* Generally, memory chips aren’t large enough to form the entire memory, so use a series of chips
* **Memory access time**: the time between access request and completion
* **Memory cycle**: the time between successive operations
* Speed up access:
  + Consecutive memory locations – must wait for successive accesses to complete
  + If we store consecutive words in different modules, then we can access them simultaneously
    - Advantage: much faster
    - Disadvantage: have to populate entire address space

### Cache Memory

* Small amounts of fast memory, placed physically close to processor
* Based on locality of reference in programs
* Where to place contents of main memory in cache?
  + **Direct Mapping**: always store in the same location
    - Simple, but blocks may be replaced even when some are empty, slowing system down
  + **Associative Mapping:** blocksof main memory placed in any block of cache
    - More flexible and efficient
    - Takes time to check for a word in the cache
  + **Block-Set Associative Mapping:** only select blocks in cache where main memory block can be placed
    - To search for word, only search the limited number of blocks, taking less time
* Which blocks do we replace in cache?
  + Least Recently Used (LRU)
    - Counter keeps track of block “age”
    - Best performance
  + First In First Out (FIFO)
    - Worst performance
  + Random
    - Decent performance
* Cache hit: requested word is present in cache
* Cache miss: requested word not present in cache
* Average memory access time: Tavg = hC + (1 –h)M
  + h = hit rate
  + C = cache access time
  + M = main memory access time
  + To decrease Tavg:
    - Increase hit rate
      * Larger block size
      * Higher associativity
      * Prefetching
    - Decrease C
      * Small and simple cache
      * Decrease distance to processor
      * Faster tag checking
    - Decrease M
      * Place sub-blocks optimally
      * Multiple cache levels
      * Lock-up free caches
        + Two-level (L2) cache: slower but larger
        + Tavg = h1C1 + (1 –h1)h2C2 + (1 –h1)( (1 –h2)M
    - **Write buffer**: temporary storage of write requests
      * Processed when memory has no read requests
      * Read addresses must be compared to write addresses to prevent read of outdated data

### Virtual Memory

* Often, programs won’t fit into physical memory
* Parts of programs not currently executing must be kept in secondary storage
  + Blocks of program have to be moved in/out of main memory
  + CPU generates virtual addresses or logical addresses relative to program – must be translated into physical addresses with **Memory Management Unit (MMU)**
    - Assume program is broken up into fixed-size pages:
      * Upper bits: virtual page number
      * Lower bits: location offset within page
  + MMU:
    - Used in every read/write operation
    - Translation Lookaside Buffer (small cache) kept in MMU for most recently accessed pages
    - Performance depends on speed of MMU

## Secondary Storage

### Magnetic Hard Disks

* Bulk storage for programs/data
* Much cheaper than main memory (per bit)
* Rotating disk with thin magnetic film + series of read/write heads mounted on flexible arms
* Disk spinning forces read heads to float above disk at a fixed distance
  + Each disk divided into concentric tracks, each track into sectors
  + Data is stored serially on tracks in sectors that begin with a sector header that contains information about the sector and tracks
* **Error correcting codes** ensure accuracy – add bits to data to correct errors that may occur
* Access time = seek time + latency time:
  + Seek time: move heads to proper track
  + Latency time: time required after head is over correct track, before data moves under head

### Disk Controllers

* Provide interface between disk drives and system bus
* Usually uses DMA process
  + CPU loads main memory address, disk address, and word count into DMA
* Controller operations:
  + Seek (move head)
  + Read
  + Write
  + Error check

### RAID Disk Arrays (Redundant Array of Inexpensive Disks)

* Data striping: single large file broken up into pieces and stored on several disks
  + During Read operation, disks can be read simultaneously – fast
* Error recovery: add parity based error recovery schemes, stored over multiple disks
* **RAID 0**
  + Stripe data over 2+ disks
  + No redundancy, improves performance
* **RAID 1**
  + Mirror data on a second disk
  + Data lost if both disks fail
* **RAID 2**
  + Stripe data at bit level
  + Error correcting code used to spread data over several disks
  + If error occurs, it can be corrected
* **RAID 3**
  + Stripe data at byte level
  + Dedicated parity (error correction) disk
* **RAID 4**
  + Stripe data at block level
  + Dedicated parity disk
* **RAID 5**
  + Stripe data at block level
  + Parity (error correction) distributed all over disks

### Optical Storage

Large capacity – cheap per bit

* E.g. CD-ROMs, DVD-ROMs, etc.

# Chapter 9: Arithmetic

## Finite Fields

* Addition and multiplication mod p (a prime number) forms a Finite Field/Galois Field
* The generator of the field is a number whose powers produce all the non-zero elements of the field
  + E.g. 3 is a generator of GF(7); 7 is a generator of GF(23)

31 mod 7 = 3

32 mod 7 = 2

33 mod 7 = 6

34 mod 7 = 4

35 mod 7 = 5

36 mod 7 = 1

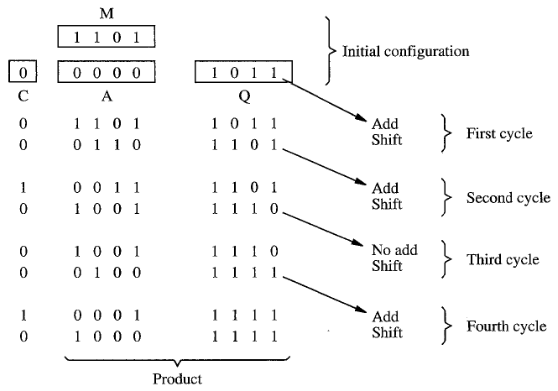
37 mod 7 = 3

## Diffie-Hellman Key Exchange

* Method for two computers to generate a shared private key
  + Users U1 and U2 agree on two prime numbers, x and p
  + U1 chooses a and U2 chooses b, random numbers for their private keys
  + U1 computes A = xa mod p to send to U2, and U2 computes B = xb mod p to send to U1
  + U1 and U2 compute their shared key:
    - U1 computes Ba mod p = (xb)a mod p
    - U2 computes Ab mod p = (xa)b mod p
* Given y = xa mod p, it is very difficult to find a for a very large prime p

## Multiplication of Positive Integers

* (2n – 1) x (2n – 1) = (22n – 2n+1 – 1) 🡨 2n-bit result
* We could design hardware to implement this method in an array
  + For large n arrays, it would be too big to implement
* We could also build a sequential circuit to do multiplication



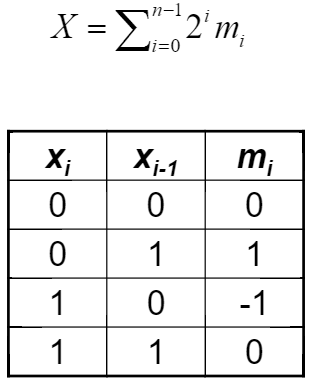
* + Check LSB – if it’s a 1, add M to Q; if it’s a 0, don’t add M
  + Always shift right
  + Repeat n times

## Signed Multiplication

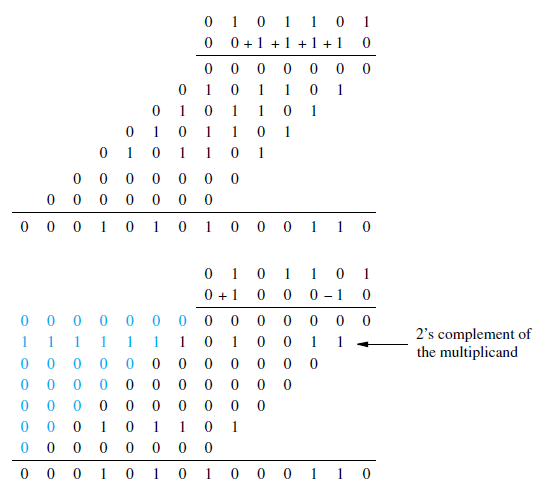
* We could treat the numbers as positive magnitudes and adjust the sign of the result
* Or we can use 2’s complement numbers

## Booth Algorithm

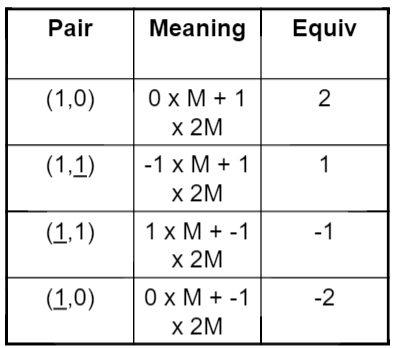
* Convert the multiplier to its Booth recoding



* Multiply – if multiplying by -1, take 2’s complement of multiplicand



* Improvement: consider pairs of Booth digits



* This will ensure at most n/2 operations, independent of patterns in the number

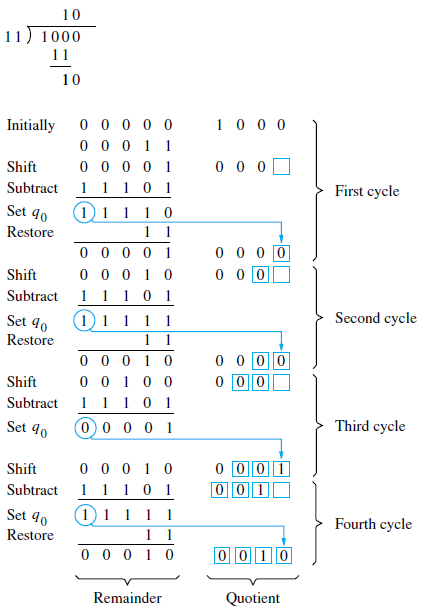
## Integer Division

### Restoring Division

* Consider N = qD + R
* Algorithm (repeats n times):

1. Shift R and N left by one bit
2. Subtract N - D
3. If sign of R is 1 (negative), R = R + D, q0 = 0; else, q0 = 1 🡨 q0 is the LSB of Q

* If remainder is positive, shift and subtract D (i.e. 2N – D)
* If remainder is negative, add D, shift, then subtract D (i.e. 2(N + D) – D = 2N + D)
* Example:



### Non-Restoring Division

* Improves restoring division by removing the need for restoring R after unsuccessful subtraction (i.e. when result is negative)
  + If remainder is positive, shift left and subtract D (i.e. 2N – D)
  + If remainder is negative, shift left and add D (i.e. 2N + D)
* Algorithm (repeats n times):

1. If sign of R is 0 (positive), shift R and N left and subtract N – D   
   Otherwise, shift R and N left and add N + D
2. If the sign of R is 0 (positive), set q0 = 1; otherwise, set q0 = 0
3. Finally, after repeating above steps, if sign of R is 1, add R + D to ensure R is positive

## Floating Point Numbers

* As magnitude increases, precision decreases
* Normalized notation:
* **IEEE standard 32-bit floating point**

|  |  |  |
| --- | --- | --- |
| S | E’ | M |

Represents +/-1.M x 2E’ – 127

* + S = sign (1-bit)
  + E’ = exponent in excess-127 (8-bit)
  + M = mantissa fraction (23-bit)
* Example:

|  |  |  |
| --- | --- | --- |
| 0 | 0 0 1 0 1 0 0 0 | 1 0 1 0 1 0 …. 0 |

Represents 1.10101000 x 2-87

* **Range**: 10+/-38
* **Double precision**: 64-bit numbers with 11-bit excess-1023 exponents and 53-bit fractional part
* **Special numbers**:
  + Zero: E’ = 0 and M = 0
  + Infinity: E’ = 255 and M = 0
  + Denormalized numbers: E’ = 0 and M != 0
    - 0.M x 2-126
  + Not a number (NaN): E’ = 255 and M != 0

### Arithmetic Operations

* **Addition/Subtraction**
  + Increase exponent: pick number with smaller exponent and shift mantissa right until exponents match
  + Perform addition/subtraction
  + Normalize result
* **Multiplication**
  + For N1 = f1 x rE1 = f1 x rE1’ – bias and f2 x rE2 = f2 x rE2’ – bias
  + N3 = f1 x f2
  + E3 = E1 + E2 = E1’ + E2’ – 2bias
  + E3’ = (E1 + E2) + bias = E1’ + E2’ – bias
* **Division**
  + Subtract exponents and add bias, similar to previous case
  + Divide mantissa and determine sign
  + Normalize result

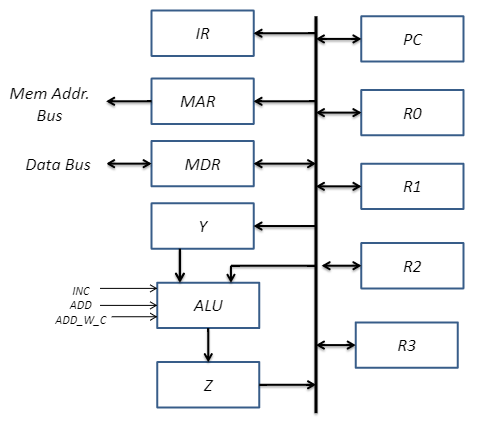
### Truncation Errors

* Results are limited to a fixed number of bits (e.g. 24 for IEEE single precision numbers)
* If we truncated results and then normalized, we could introduce large errors – must keep some extra guard bits during arithmetic operations
* **Von Neumann rounding**:
  + If all chopped bits are 0, retained bits do not change
  + If any chopped bits are 1, lowest retained bit is set to 1
  + Truncation error becomes +/- 1 in LSB of retained bits – larger range of error, but symmetric around zero

# Chapter 7: CPU

## Instruction Execution

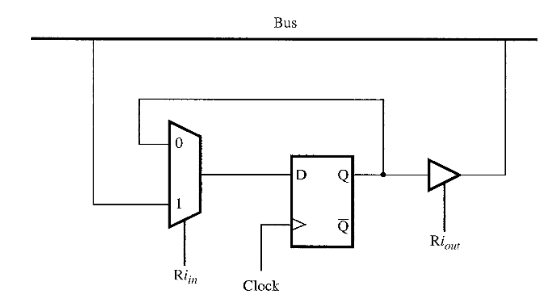
* PC points to start of instruction in memory
* Instruction fetch:Transfer contents of PC to IR
* Instruction execution: circuitry transports instruction code into series of sequential operations



* Y and Z: intermediate registers (Y stores one operand, Z stores result)
* ALU:
  + INC – increment
  + ADD – add
  + ADD\_W\_C – add with carry

## Basic CPU Design

* XXin – reads from (latches data) from the bus
* XXout – drives the bus with data from a register



1 – Drive bus  
0 – off

1 – Store data from bus  
0 – Off

D flip flop

## 

Tri-state buffer

## Instruction Fetch

|  |  |
| --- | --- |
| **Clock cycle** | **Control signal** |
| 1 | **PCout** – place PC instruction address on data bus  **MARin** – memory address bus loads PC address  **READ** – read instruction from memory  **INC** – ALU increments address  **Zin** – puts result in output register |
| 2 | **Zout** – put incremented address on data bus  **PCin** – PC takes in next address  **WMFC** – wait for memory function complete |
| 3 | **MDRout** – put instruction read from memory onto data bus  **IRin** – store instruction in IR |

### Register to Register Transfer

Example: MOVE R1, R2

1. PCout, MARin, READ, INC, Zin

} Instruction fetch

1. Zout, PCin, WMFC
2. MDRout, IRin
3. R2out, R1in
4. End -> when an instruction is finished, reset state of machine to the beginning to start on the next instruction

### Immediate Addressing

Example: ADD R1, #8

Instruction fetch (1-3) as before

1. PCout, MARin, READ, INC, Zin
2. Zout, PCin, WMFC
3. R1out, Yin, WMFC
4. MDRout, ~~SELY~~, ADD, Zin
5. Zout, R1in
6. End

### Absolute Addressing

Example: ADD $1000, R1 (two-word instruction, $1000 is a memory address)

* Three trips to memory, so slower

Instruction fetch (1-3) as before

1. MDRout, MARin, READ
2. R1out, Yin, WMFC - put R1 into temp register before waiting for memory complete
3. MDRout, ~~SELY~~, ADD, Zin
4. Zout, R1in
5. End

### Indirect Addressing

Example: ADD (R1), R2

Instruction fetch (1-3) as before

1. R1out, MARin, READ
2. R2out, Yin, WMFC - put R2 into temp register before waiting for memory complete
3. MDRout, ~~SELY~~, ADD, Zin
4. Zout, R2in
5. End

### Indirect with Offset

Example: ADD $1000(R1), R2

* Three memory accesses

Instruction fetch (1-3) as before

1. PCout, MARin, READ, INC, Zin
2. Zout, PCin
3. R1out, Yin, WMFC
4. MDRout, ~~SELY~~, ADD, Zin - add offset
5. Zout, MARin, READ
6. R2out, Yin, WMFC
7. MDRout, ~~SELY~~, ADD, Zin
8. Zout, R2in
9. End

### Branching (BNE)

* In branching, need to check the status of the condition code register (CCR) and adjust the PC accordingly

Instruction fetch (1-3) as before

1. PCout, Yin, if Z = 1 – End
2. REL-ADDR-FLD-IRout, ~~SELY~~, ADD, Zin
3. Zout, PCin
4. End

## Appearing like a Larger Machine

* Wide buses take up a lot of space on the chip – may not be practical for some devices
* CPU with smaller bus structure can “appear” like a larger bus structure – e.g. a 32-bit architecture implemented with a 16-bit internal bus structure
  + Each register has two parts – high (h) and low (l)
  + Consider a 32-bit instruction words and 32-bit operation
  + Example: ADD R1, R2

1. PClout, MARlin, INC, Zin
2. Zount, PClin
3. PChout, MARhin, INC\_WITH\_C, Zin
4. Zount, PChin, READ, WMFC
5. MDRlout, IRlin
6. MDRhout, IRhin
7. R1lout, Yin
8. R2lout, ADD, Zin
9. Zout, R2lin
10. R1hout, Yin
11. R2hout, ADD\_WITH\_C, Zin
12. Zount, R2hin
13. End
    * System is slow because only one register can drive the bus at a time
    * With more internal buses, e.g. a 2 bus structure, we can speed up the CPU
    * Example: ADD R1, R2
14. PCout, X, MARin, Yin, READ
15. INC, ALUout, PCin, WMFC
16. MDRout, X, IRin
17. R1out, X, Yin
18. R2out, ADD, ALUout, R2in
19. End
    * 2 bus structure speeds up process, since we can use both buses simultaneously
    * Example: ADD R1, R2
20. PCout, MARin, ALU\_Bin, INC, ALUout, PCin, READ, WMFC
21. MDRout, X, IRin
22. R1\_Aout, R2\_Bout, ADD, Yin
23. Yout, R2in
24. End
    * Can extend this idea to more buses – try to have as any active as possible at one time
    * 3-operand instructions would also simplify the instructions
    * Example: ADD R1, R2, R3
25. PCout, R = B, MARin, INC\_PC, READ, WMFC
26. MDR\_Bout, R = B, IRin
27. R1\_Aout, R2\_Bout, ADD, R3in
28. End

## Control Units

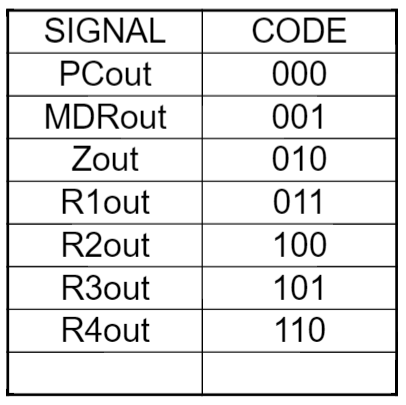
* Processor must have a means to generate control signals at the appropriate times
  + Hardwired controller – synchronous sequential machine
    - Requires:
      * **Step counter**
      * **Instruction register** contents
      * **Condition codes**
      * **External inputs** (interrupts, MFC, etc.)
    - Generates control signals as a combination state (T1, T2, etc.) and appropriate inputs (e.g. branch instructions)
    - Generate logic equations for all control signals
      * Advantages: fast – combinational logic
      * Disadvantages: difficult to change, and signals can become complex

### Microprogrammed controller

* + - Each instruction is a sequence of microcode that generates control signals
      * **Control word**: area of memory where individual control signals are represented
      * **Micro-routine**: sequence of control words
      * **Control store**: memory where microprograms are stored
      * **Microprogram counter**: keeps track of micro-routine execution
        + After an instruction is fetched, micro-PC is set to start location of routine (decoded from IR)
    - Advantages: errors can be fixed in a relatively shorter time
    - Disadvantages: Much slower than hardwired controllers

## Vertical vs. Horizontal Organization

* **Horizontal organization**: if we have a separate bit for each control signal, control store could get very wide
  + No restrictions on type of operations
  + Fast, since control signals are directly available, but expensive
* **Vertical organization**: we could restrict some operations and group control signals together



* + Smaller control store
  + Slower execution, since bits must be decoded

## Enhanced Internal Structure

* Micro-PCs require sequential microcode and lots of branching to many segments – potentially large control store
* If we replace the micro-PC with a **next address field** in the instruction, we can have a much smaller control store

### Nanocoded Controllers

* Some machines use a two-level control store to try and improve performance between horizontal/vertical organization
* Microinstructions are stored in “narrow” control store, but run nanoroutines in a wider nano control store
* Nano processing engine contains a nano-PC

### Emulation

* In some computers, areas of microcode can be edited, allowing designers to emulate instruction sets of a new design on an obsolete device

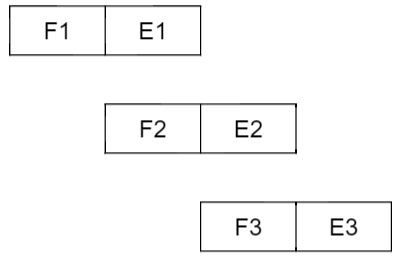
# Chapter 8: Pipelining

## RISC vs. CISC

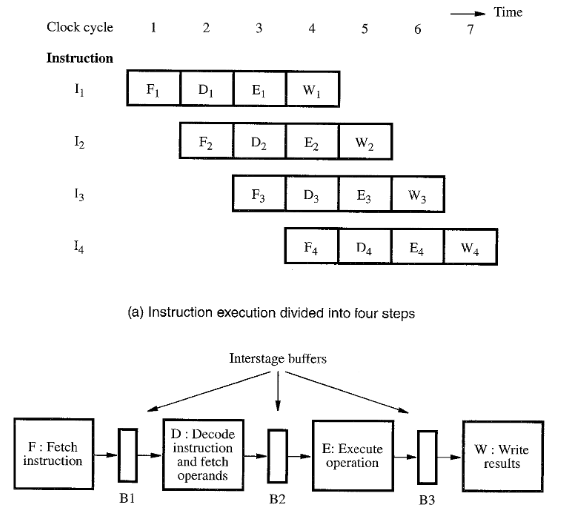
* **CISC**: build a computer with a large number of complex instructions
  + Minimize the number of instructions required
  + Provide instructions that support typical constructs of higher-level languages
* **RISC**: determine which instructions are easily generated by a compiler, reducing number of instructions and addressing modes
  + Larger amount of code, but if instructions can execute at the same time, we can pipeline them
  + IBM 801 = first commercially available RISC processor
  + Execution time of a task: ET = N \* S \* T
    - N = number of instructions
    - S = number of steps per instruction
    - T = time per step
  + CISC reduces N, RISC reduces S and T

## RISC

* T is determined by clock period, number of logic levels for decoding instructions, and control signal generation – keep simple
* If we can form a pipeline where one instruction is completed per clock cycle, performance is much better
* Simplest design: **2 stage pipeline**
  + Fetch next instruction while executing current instruction

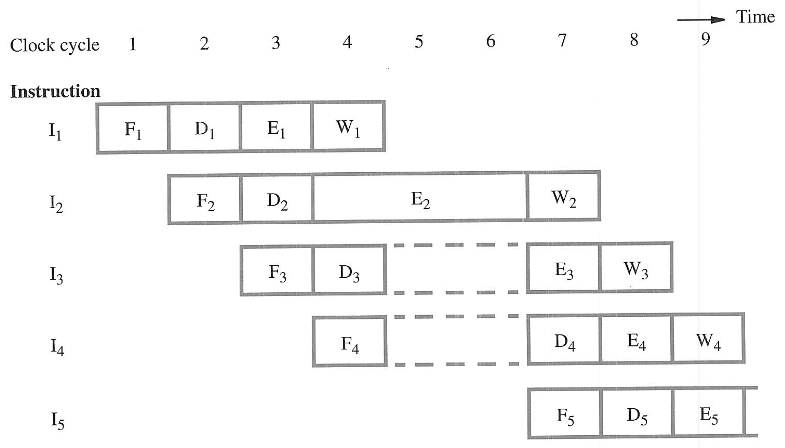


* Objective: increase depth of pipeline, so each part of pipeline requires less time
* **4-stage pipeline**: breaks operation down to four parts
  + F – fetch instruction from memory
  + D – decode instruction
  + E – execute instruction
  + W – write results to register



E.g. clock cycle 4:

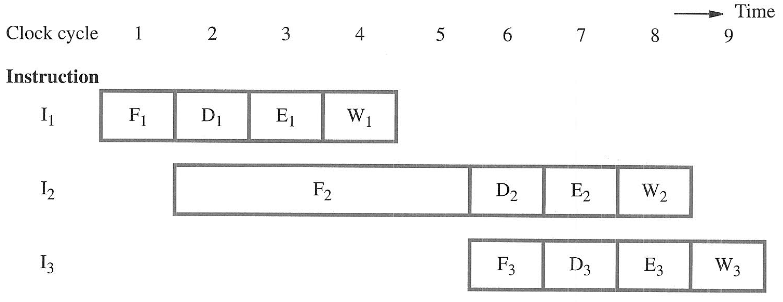
* B1 holds I3
* B2 holds I2
* B3 holds I1
* Pipelines perform best when each stage requires about the same time to complete
  + Since instruction fetches are frequent, use cache to ensure fetch process is fast
  + Cache access generally takes about same time as other processor operations, while memory access can take up to 10 times longer
* Sometimes, a pipeline stage may not be able to complete execution in one clock cycle



* + In cycles 5 and 6, W1 must be told to not do anything since there’s nothing to write
  + D4 and F5 must be postponed, since buffers are full
  + Pipeline operation is said to have **stalled** for two cycles
  + **Hazard**: any condition that causes a pipeline to stall
* Pipelining doesn’t make individual instructions run faster; it makes the rate of instruction execution completion faster
* Whenever one stage in the pipeline can’t complete in one clock cycle, the pipeline stalls so performance decreases

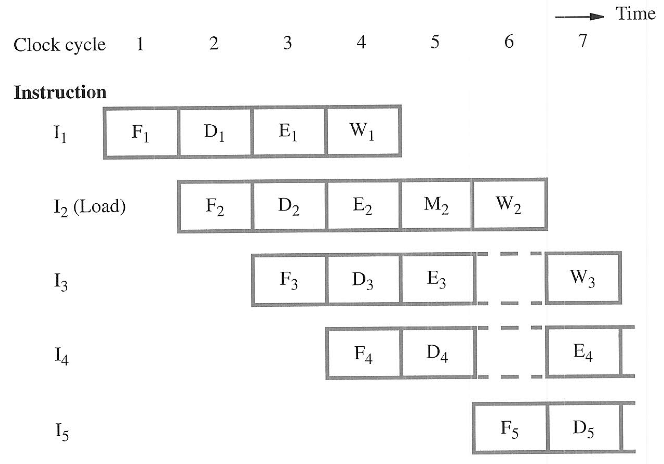
## Control/Instruction Hazard

* Instruction availability is delayed
  + E.g. cache miss results in instruction needing to be fetched from main memory



## Structural Hazard

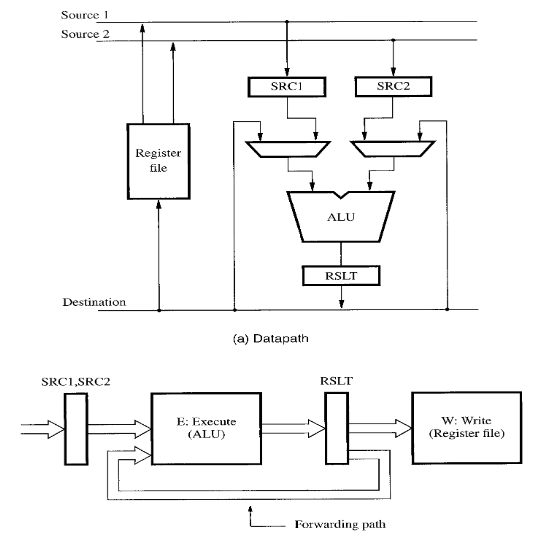
* Two instructions require use of the same hardware at the same time
  + E.g. one instruction requires memory access during Execute or Write stage, while other is in Fetch stage – if memory lies in same cache unit, only one can proceed while other is delayed



* + - In this case, E2 is followed by memory access M2, so W2 and W3 become simultaneous, so W3 is delayed one cycle
    - This causes the pipeline to stall for one cycle

## Data Hazard

* Source or destination operands for an instruction aren’t available at the expected time in the pipeline
  + E.g. data is changed in one instruction and needed in the next one
  + If data hasn’t been written back to the register before it is fetched, a stall occurs
  + **Data forwarding**: since data is available at output of ALU before being written to registers, it could be forwarded to the pipeline from the ALU
    - In 3-bus structure, add three registers (inter-stage buffers) to the ALU plus feedback paths for the result to the ALU inputs



* + - * SRC1, SRC2 are part of B2 from before, and RSLT is part of B3

### Data Hazards in Software

* Data hazards can be detected in software – compiler inserts NOP (no operation) instructions
  + Example:

ADD R1, R2, R3

NOP

NOP

ADD R3, R5, R6

ADD R1, R8, R9

ADD R2, R4, R10

* This simplifies hardware, but compiler must know about a lot about pipeline structure and clock cycles wasted on NOP’s
* Better solution: compiler reorders code
  + Example:

ADD R1, R2, R3

ADD R3, R5, R6

ADD R1, R8, R9

ADD R2, R4, R10

Reorder:

ADD R1, R2, R3

ADD R1, R8, R9

ADD R2, R4, R10

ADD R3, R5, R6

* **Side effects**: instructions that change registers other than those explicitly named (e.g. auto increment/decrement, CCR, push, pop)
  + Either have complex hardware to detect possible side effects, or have the compiler do it
  + Example: Depending on the pipeline, a stall may be required to account for the carry

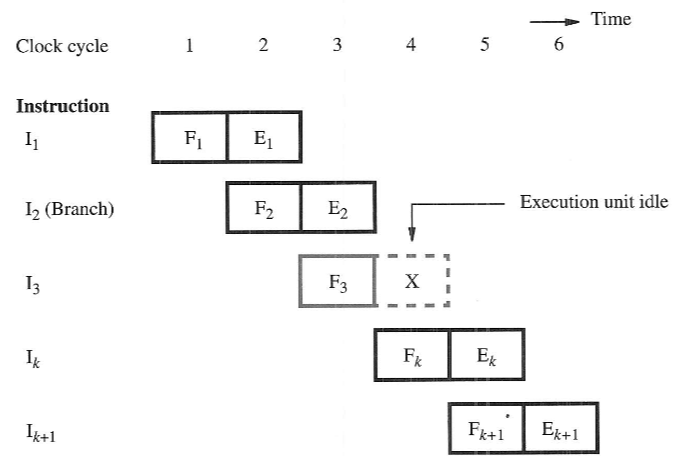
ADD R1, R3

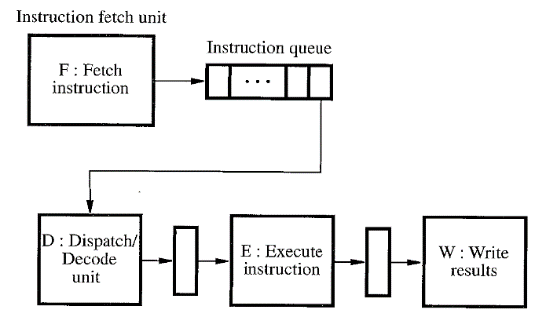
ADD\_W\_C R2, R4

* **Branch penalties**: the number of cycles that the pipeline is stalled until execution resumes at target location

## Unconditional Branches

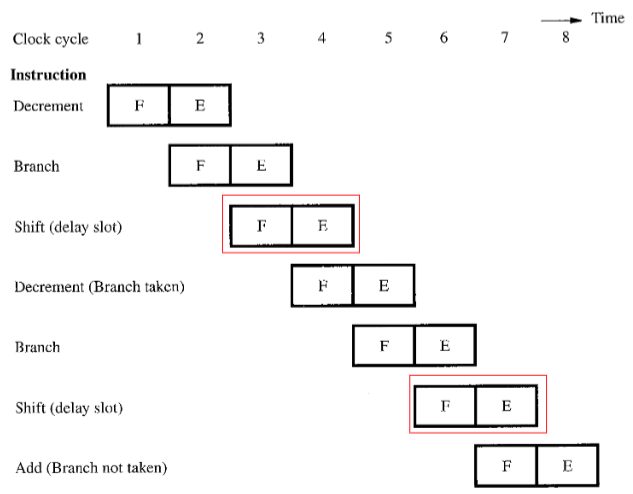
* Example:



* + Fetch instruction 3 (F3) occurs at the same time as the branch instruction is being decoded
  + In clock cycle 4, I3 must be discarded and branch target instruction Ik is fetched in clock cycle 5 – stall of one clock cycle
* The longer the pipeline, the longer the branch penalty
* To reduce branch penalties, compute branch target addresses earlier in the pipeline, like at the decode stage
* If we add hardware to pre-fetch instructions and queue them, then on a cache miss the instructions can be fetched before they are needed
  + **Instruction queues** typically store a few instructions
  + **Dispatch unit** moves instructions from front of queue to execution unit
    - Tries to keep pipeline filled at all times
    - To avoid branch penalties, dispatch unit must have sufficient processing power and speed to perform/predict branch instructions in the D phase
    - **Branch folding**: when a branch can be executed concurrently with other instructions

## Conditional Branches

* Problems arise when conditional branches occur – two possible execution paths
* **Branch delay slot**: the instruction after BCC
  + Example:



* + - Branch delay slot is always executed, regardless of whether or not branch is taken
      * First case: pipeline would stall if branch taken
      * Second case: shift execution always executed, and branch can be calculated

### Branch Prediction

* If branch result could be predicted ahead of time, branch penalty could be avoided
  + **Speculative execution**: assume no branch and carry on with execution
    - Must make sure any changes to registers aren’t made until branch is confirmed
    - If wrong decision made, must purge execution unit and go back to correct path
  + In many cases, speculative execution will be wrong
    - E.g. loops – when at end of a loop, better to assume branch will be taken; when at beginning of a loop, better to assume branch will not be taken
* **Static branch prediction**: some processors include a branch prediction bit that can be set by the compiler to determine whether a branch is likely to be taken or not
* **Dynamic branch prediction**: maintain history information of states, in order to predict
  + Four states:
    - ST: strongly likely
    - LT: likely
    - LNT: not likely
    - SNT: strongly not likely
  + Initially, state is set to LNT
  + If state is ST or LT, instructions are fetched from target address
  + First pass through a loop would be incorrect – this can be fixed by including static prediction to set the initial state

## Instruction Sets and Pipelines

* Complex addressing modes and instructions cause problems in pipelines
* Example: LOAD (X(R1)), R2 🡨 takes 5 cycles to complete

Could replace with:

ADD #X, R1, R2

LOAD (R2), R2

LOAD (R2), R2

## Condition Codes

* Branches are determined based on the results of execution operations, which can cause pipeline stall while waiting for the CCR to be set
* Example:

ADD R1, R2

CMP R3, R4

BEQ LOOP

Would be better as:

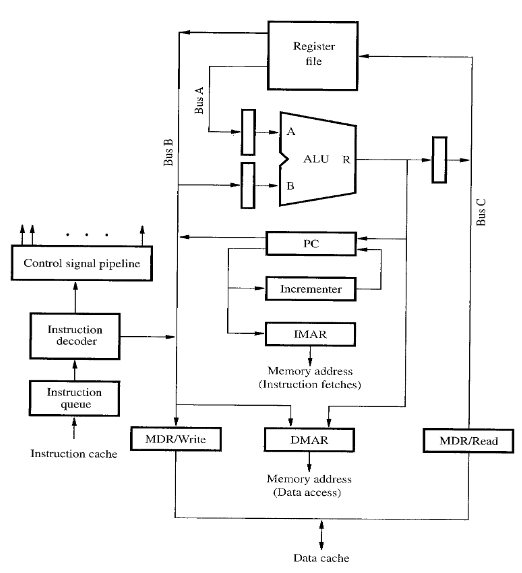
CMP R3, R4

ADD R1, R2

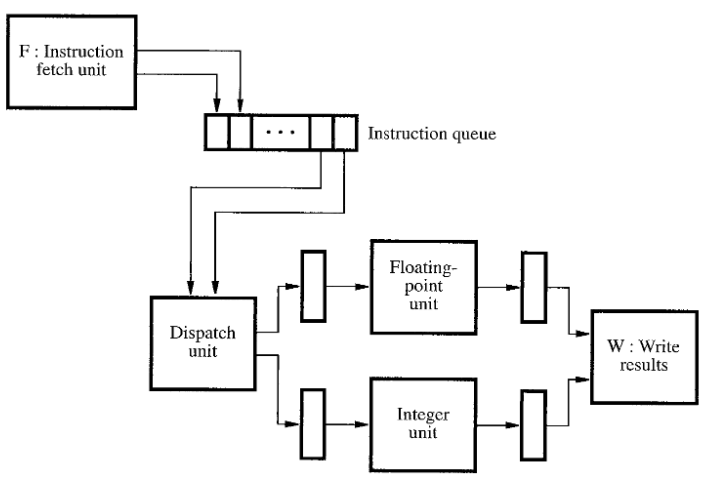
BEQ LOOP

## Datapath

* To make a pipeline efficient, buffers and multiple data paths should be used in CPU design
* Data and instructions should be stored in separate caches, to prevent lockup
* 4-stage pipeline architecture:

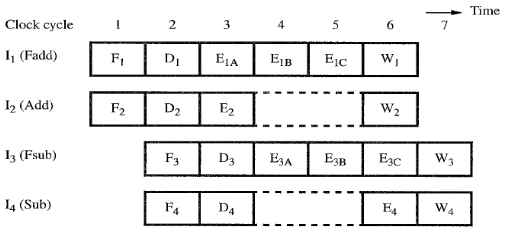


## Superscalar Architecture

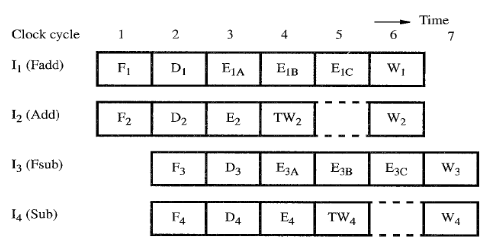
* Pipelines are designed to achieve one instruction completion per clock cycle
* Performance can be improved if multiple processing units are present in the CPU – if they work in parallel, more instructions can be completed per clock cycle
* **Multiple issue/superscalar**: when more than one instruction can start execution in one clock cycle
  + Usually separate execution units for integer and floating point operations
  + Execution can take different times, but this will cause the effects of hazards to be more pronounced

### Out of Order Execution

* If there is data dependency between instructions, and instruction execution is started in the same order as in the program, results may not always be ready when needed
* This can lead to exceptions out of execution
  + **Imprecise exceptions**: when an exception occurs, PC must point to the instruction where exception occurred, but the processor state may not be correct because other instructions might have completed during this time
  + Must ensure results are written in the same order that they appear in the program
    - Delayed write can be used to maintain order



* If processing units have different clock cycle times, it can be advantageous to start out of order instruction execution while instructions with longer cycle times are executing
* Introduce temporary register to hold results of faster unit until slower unit can catch up – ensures correct completion order



# Chapter 12: Case Studies

## M68020

* First Motorola 32-bit microprocessor that aimed had high performance, pipelined architecture
* Elements:
  + Bus controller + instruction cache unit
  + Instruction pre-fetch + decode unit
  + Sequencer + control unit
  + Execution unit
* Concurrent operation up to three words at a time
  + Could all be part of the same instruction
  + Required MMU as separate chip

## M68030

* Added 256-byte instruction cache and data cache
* MMU on board
* Execution unit generates virtual addresses to cache – translated to physical address in parallel with cache access; if cache miss, physical address is immediately available

## M68040

* Improved MMU: independent translators for instruction cache and data cache – could simultaneously translate instruction and data addresses
* Added floating point unit
* Larger data and instruction caches (4KB)

## M68060

* Pipelined superscalar architecture – up to 3 instructions started per clock cycle
* 4 basic stages + 2 additional stages for memory writeback
* 8KB data and instruction caches, 4-way set associative (16-byte blocks with 4 cache locations per set)
* Two 64-entry, 4-way set associative translation lookaside buffers for virtual to physical address translation
* Dynamic branch prediction

## M88000

* RISC chips
* Two execution units – integer and floating point
* 32 registers
* Separate data and instruction buses
* Up to 5 parallel executions:
  + Instruction fetch from instruction cache
  + Data fetch from data cache
  + Three internal data manipulation operations

## MM88100

* 51 instructions – 12 floating point
* 3 load/store operations:
  + LOAD
  + STORE
  + Exchange memory (read followed by write)
* Three-operand instruction format
* Constant register R0 always = 0 – can provide clear operation without a separate instruction set
* 3 basic instruction formats:
  + Op\_code Rd, Rs1, Rs2
  + Op\_code Rd, Rs1, Immed\_16
  + Op\_code Rd, Rs1, Immed\_10
* 3 data memory addressing modes:
  + Register indirect with unsigned offset
  + Register indirect with index
  + Register indirect with scaled index

## Intel Pentium IV

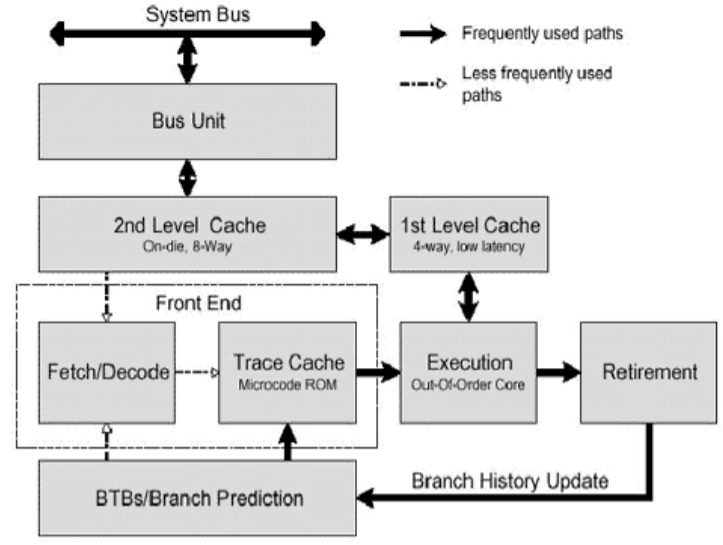
* Integer operations
  + 2 integer double ALU’s running at 2 times the CPU clock
  + 2 floating point execution units
    - Dedicated floating point unit + dedicated floating point move/store unit
* Registers:
  + Eight 32-bit general purpose
  + Eight 80-bit floating point
  + Eight 64-bit MMX registers
  + Eight 128-bit XMM SIMD (single instruction multiple data) registers
* Cache:
  + 8KB L1 data (4-way associative)
  + L1 execution cache (buffers up to 12000 micro-operations)
  + 256KB L2 cache (8-way associative)

## P4 Pipeline

* 20-stage pipeline with as many as 126 instructions in various stages of execution
* Fewer instructions per clock cycle than P3
* 128 “renaming” registers – handles data dependencies
* 4KB branch prediction table

## Trace Cache

* For frequently reused operations, trace cache locally recalls operations that have already been used and sends them to the instruction cache – bypasses fetch/decode unit
* Has extremely high speed buffers – must operate at CPU clock speed (buffer up to 12000 micro-operations)



## Intel Pentium D

* Dual cores, for simultaneous execution
* Each core has 1MB L2 cache

## AMD Athlon 64 X2 Dual-Core

* 64KB L1 instruction cache and L1 data cache per core
* Up to 1MB L2 cache per core

## Intel Core

* Replaces Pentium 4
* Based on Pentium M -> based on Pentium III
* Lower power than Pentium IV
* SSE3 (Streaming SIMD Extensions 3)
* Core Duo has 151 million transistors in 65 nm technology

## Itanium IA-64

* 64-bit Explicitly Parallel Instruction-set Computer (EPIC)
  + Parallel execution responsibilities shifted to compiler (from CPU control circuitry)
  + Improved on VLIW (Very Long Instruction Word, successor to RISC)
  + Instructions issued in bundles
  + 256+ registers
* Intended successor to Pentium IV (IA-32) but not backwards compatible with IA-32
* Developed by Intel and HP, used in servers
* Itanium2 Dual-Core: 1.7 billion transistors, 24MB cache

## Core

* Made by Sony, Toshiba, and IBM for gaming, television, supercomputing
* 1 power processor “supervisor”
* 8 SIMD co-processors connected in a ring, controlled by supervisor – downloads instructions through DMA
* Uses Rambus memory

## G4 Architecture

* Up to 16 instructions executing at a time
* 8-way associative caches:
  + 32KB L1 instruction and data caches
  + 256KB L2 cache

## G5 Processor

* 58 million transistors, 130 nm process
* 2.5GHz max clock speed
* Up to 8 instructions can enter fetch/decode stage per clock cycle
* 2 double-precision floating point units
* Each execution unit has an instruction queue
* SIMD processing – vector processing unit
* Three component branch prediction
* Caches:
  + 64KB direct-mapped L1 instruction cache
  + 32KB 2-way associative L1 data cache
  + 512Kb 8-way associative L2 cache

## IBM Cell Processor

* Each Synergistic Processing Unit (SPU) has 256KB local storage SIMD structure
  + SPU = Synergistic Processing Unit
  + SPE = Synergistic Processing Element
  + SXU = Synergistic Execution Unit
  + LS = Load/Store
  + SMF = Synergistic Memory Flow Controller
  + EIB = Element Interconnect Bus
* >4GHz clock speed
* 234 million transistors
* Performance:
  + 256 gigaflops single precision
  + 26 gigaflops double precision

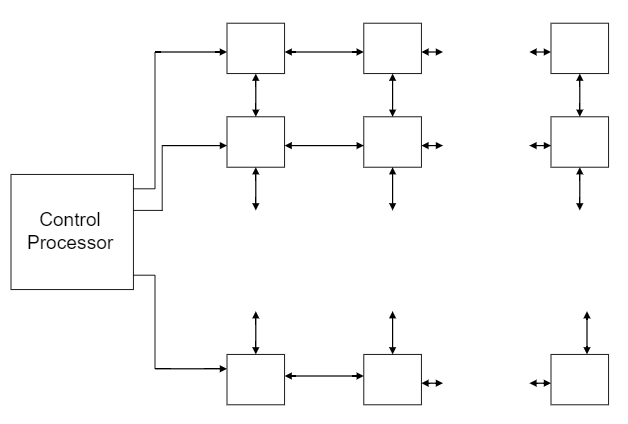
## NVIDIA Telsa

* Graphic Processing Unit (GPU)
* Up to 192 cores running in parallel per block; 15 blocks – 2880 cores
* 7.1 billion transistors
* > 1 teraflop peak processing
* Runs CUDA (parallel computing platform)

## Multiprocessor Structures

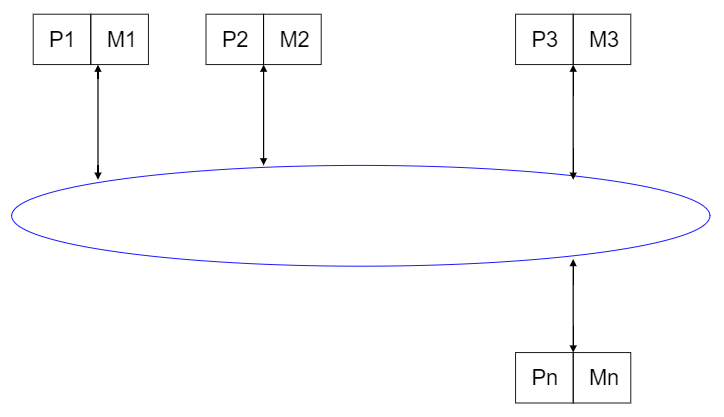
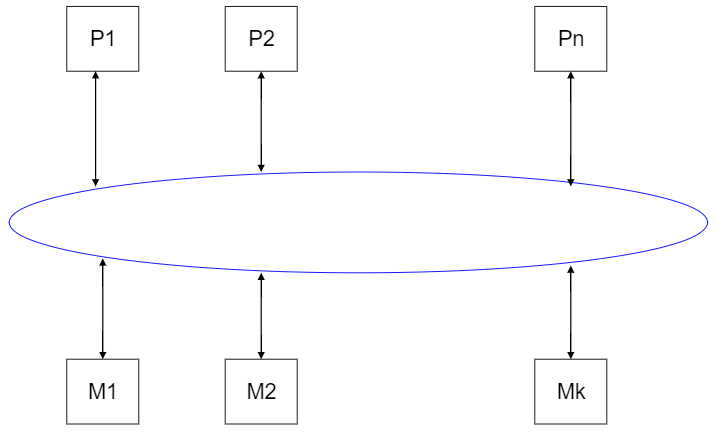
* Sometimes it’s cheaper to have multiple processors work in parallel to increase performance
* Classifications:
  + **Single Instruction Stream, Single Data Stream (SISD)** – conventional single processor
  + **Single Instruction Stream, Multiple Data Stream (SIMD)** – each processor operates on its own data
  + **Multiple Instruction Stream, Multiple Data Stream (MIMD)** – each processor is assigned a task and data set

## ARRAY Processor (SIMD)



* 2D grid of processors
* Central control processor broadcasts instruction stream to processors for execution
* All processors execute simultaneously
* Each element has 4 closest neighbours for data exchange
* Good for computations requiring successive iterations, e.g. weather models, fluid flow, temperature distribution in a solid

## Multiprocessors (MIMD)

* Collection of processors able to execute independent tasks in parallel
  + n processors
  + k memory modules
  + interconnection network
* Each processor could have local memory, or memory is network element with very fast connection network
* Address space:
  + Simplest way is to have all processors access same global address space, with local caches – but what about variable integrity with multiple caches?
  + Could also divide address space into private/global
  + Use multi-computers in a system where processors communicate through “messages”

## Parallel Processing

* Difficult to break up most problems into parallel subtasks
* Some high-level languages include constructs to specify parallel execution

PARBEGIN

…

PAREND

* Difficult to run multiple processors at the same time, while ensuring shared variables are correct

## Shared Variables

* Generally, processes can LOCK a variable in a read\_modify\_write cycle
* If a process needs a shared variable, it first checks whether it is locked, before access
* M68000 has a Test\_and\_Set instruction to lock variables

## Cache Coherency

* What happens when shared variables appear in multiple caches?
* Can do a global write-through of cache whenever a process updates a variable locally, but this is hard to implement
* **Snoopy Cache Coherency**
  + When processor writes cache variable the first time, it is marked dirty and information is broadcast to network
  + Global memory updates its copy
  + All other caches with that code segment mark the variable as invalid; original processor is considered owner of that variable
  + Owner can do further writes to that variable without notifying system
  + Other processes needing that variable obtain it from the owner
  + When a copy is broadcast, all copies are updated and owner marks variable as clean, so other processes can become owner

# Embedded Systems

## Automotive Industry

* 1978 Cadillac Seville – first microprocessor in a car
* Today, some vehicles have 100-150 microprocessors running over 100 million lines of code
* On-board Diagnostics OBD-II software connection on every vehicle
  + Can be hacked – lock car, change temperature controls, control radio, and disable brakes
* Caches aren’t used in automotive microprocessors as there can be unpredictable delays (i.e. cache misses), especially dangerous in valve timing, braking, etc.
* 2W maximum power dissipation, as processors must be sealed against water (so minimum air flow to cool them down)
* Embedded controllers have to be cheap