Communication

- connection
  - between processors
  - between processors and memory
- alternatives
  - symmetric shared memory systems - bus
  - distributed shared memory systems – interconnection networks

Bus Communication

- economic
- simple design
- lower performance
- does not scale well with the number of processors
- very important - cache memories
  - coherence problems

Interconnection Networks (1)

- between
  - processors
  - processors and memory
- goals
  - flexibility
  - performance
    - more accesses in parallel

Interconnection Networks (2)

- types of connection
  - total - each one to all others
  - partial - some pairs of components are not directly connected
- processor-memory connection
  - more memory circuits
  - can be accessed in parallel by different processors

Memory Coherence

- all processors must use the last value written into a shared variable
- the problem - cache memories
- goal - every shared variable has the same value
  - in all caches
  - in the main memory

Coherence - Write-back Cache

`x` - shared variable

Processor	Action	Cache A value	Cache B value	Memory value
				9
A	`i=x;`	9		9
B	`j=x;`	9	9	9
A	`x=5;`	5	9	9
B	`k=x;`	5	9	9

Coherence - Write-through Cache

`x` - shared variable

Processor	Action	Cache A value	Cache B value	Memory value
				9
A	`i=x;`	9		9
B	`j=x;`	9	9	9
A	`x=5;`	5	9	5
B	`k=x;`	5	9	5

What Is Coherence? (1)

1. execution order
   a) processor P writes to variable X
   b) then processor P reads X
   – there are no writes to X between a) and b)

$\rightarrow$ read b) returns the value written by a)

What Is Coherence? (2)

2. coherent vision of the memory
   a) processor P writes to variable X
   b) then processor Q ($Q \neq P$) reads X
   – there are no writes to X between a) and b)
   – enough time passed between a) and b)

$\rightarrow$ read b) returns the value written by a)

What Is Coherence? (3)

3. write serialization
   a) processor P writes to variable X
   b) processor Q (Q=P or Q≠P) writes to variable X

→ all processors see the two writes in the same order

– not necessarily a) before b)

Maintaining Cache Coherence

- coherence maintenance protocols
- based on the information about the cache lines
  - *invalid* - data is not valid
  - *dirty* - only the current cache has the correct (updated) value
  - *shared* - the current cache has the correct (updated) value, so do main memory and possibly other caches

Types of Protocols

- *directory based*
  - information about each cache line - kept in a single place
- *snooping*
  - each cache has a copy of the shared line
  - no centralized information
  - caches "listen" to the bus
    - detect changes brought to cache lines

Cache Update

- each cache announces the changes it makes
- the other caches react
- only write operations matter
- variants
  - write invalidate
  - write update

Write Invalidate (1)

- a processor changes the value of a location
- the change is made in its own cache
  - all other caches are notified
- every other cache
  - has no copy of that location - no action
  - has a copy of that location - invalidates its corresponding line
  - the correct value will be requested when needed

Write Invalidate (2)

`x` - shared variable

Proc.	Action	Cache reaction	Cache A	Cache B	Memory
					9
A	`i=x;`	read miss	9		9
B	`j=x;`	read miss	9	9	9
A	`x=5;`	invalidation	5	inv.	9
B	`k=x;`	read miss	5	5	5

Write Update (1)

- a processor changes the value of a location
- the change is made in its own cache
  – all other caches are notified
  – the new value is broadcasted
- every other cache
  – has no copy of that location - no action
  – has a copy of that location - gets the new value

Write Update (2)

`x` - shared variable

Proc.	Action	Cache reaction	Cache A	Cache B	Memory
					9
A	`i=x;`	read miss	9		9
B	`j=x;`	read miss	9	9	9
A	`x=5;`	invalidation	5	5	5
B	`k=x;`	read hit	5	5	5

Write Invalidate vs. Write Update (1)

- multiple successive writes to the same location
  - write invalidation - only one invalidation (first time)
  - write update - an update for each write
  - more favorable - write invalidation

Write Invalidate vs. Write Update (2)

- multiple writes to the same cache line
  - changing a location requires invalidating/updating the whole line
  - write invalidation - only one invalidation (first time)
  - write update - write update - an update for each write
  - more favorable - write invalidation

Write Invalidate vs. Write Update (3)

- "response time"
  - the time between writing a value by a processor and reading that value by another processor
  - write invalidation - first invalidation, then read (when necessary)
  - write update - immediate update
  - more favorable - write update

Write Invalidate vs. Write Update (4)

- both variants have advantages and drawbacks
- write invalidate - (much) lower usage of memory and buses
- write update - higher cache hit rate
- more often used - write invalidate

III. Peripheral Devices

Peripheral Device

- provides a certain kind of communication
  - between the processor and the "outside world"
- to manage communication with the processor, it includes an I/O controller
  - which usually contains a series of registers that store information necessary for communication
    - data
    - state information
    - commands (from the processor)

Input-output (I/O)

- how the system sees communication
  - memory-mapped I/O
    - read/write operations are seen as though they are performed on memory locations
    - I/O addresses - within the memory address space
    - the same control signals as for memory
  - isolated I/O
    - I/O addresses - separate from memory addresses
    - control signals - different from those of memory

Communication Modes (1)

- programmed I/O
  - the program waits in a loop until the peripheral device initiates a transfer
  - efficient if the moment when the device will request communication is known in advance
  - useless consumption of processor time

Communication Modes (2)

- Direct Memory Access (DMA)
  - a specialized controller (DMA controller) manages the transfer
  - very fast
  - it takes control of the buses and transfers data directly between the device and memory
    - the processor is bypassed
  - useful for transferring large amounts of data

Communication Modes (3)

- interrupt-driven I/O
  - when a peripheral device wishes to communicate, it notifies the processor
    - through an interrupt request
  - during the rest of the time, the processor may carry out other tasks
  - the most flexible method

The Buses (1)

- communication paths for the information
- a bus is a unique connection between more than 2 components
- bus description
  - electrical signals
  - communication rules - must be respected by all parts involved
  - connecting mode

The Buses (2)

Bus access

- more entities may request access simultaneously
- an arbitration procedure is required
  - decides who will be granted access
  - the other entities must wait until the bus becomes free again

Bus Arbitration

Types of bus arbitration

- centralized
  - the decision is made by a dedicated circuit (arbiter)
- de-centralized
  - components negotiate to each other
  - based on the rules that define the way the bus works

Connecting to the Bus

- electrical issues
- more circuits connected together
  - input and output
- cannot connect more outputs
  - different voltage levels would destroy the circuits
- one solution - multiplexing
  - all outputs are connected to a multiplexer

Tri-state Circuits

- output has 3 possible states
  - 0
  - 1
  - high impedance (*High*-*Z*)
- first two states correspond to usual values
- third state means decoupling from the bus
  - just as the output of the circuit would not be connected to the bus

Open-collector Circuits

- in some cases are called *open-drain*
  - depends on technology used
- it is possible to connect more outputs together
- result value - Boolean AND between the outputs that are connected

Buses - General View

Advantages

- bus activity - easy to control
- economic - relatively simple structure

Drawback

- lower performance
  - only 2 components can communicate at a certain moment

IV. The Interrupt System

What Is an Interrupt?

- the processor can suspend the execution of the current program
- goal - handling unexpected situations
- after that, the interrupted program is resumed
- initial purpose - communicate to peripherals
- the processor does not "wait" for peripherals
  - they notify the processor when necessary

Hardware Interrupts

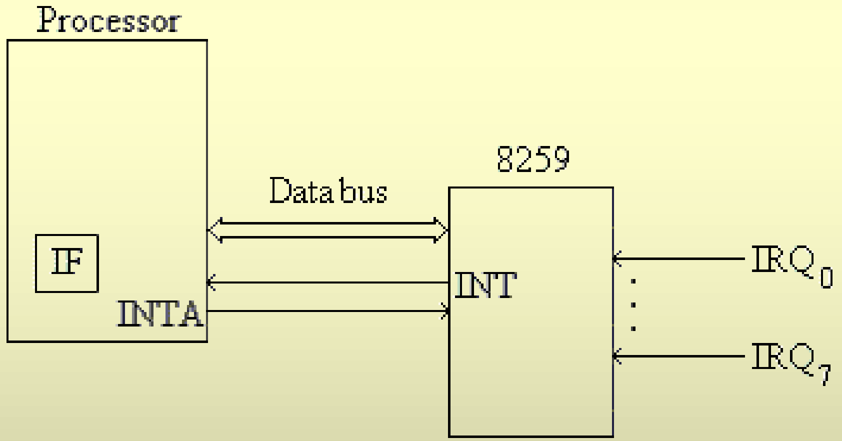
- maskable
  - the processor may refuse to service them
  - depends on the value of the IF (*Interrupt Flag*) flip-flop: 1 - accept, 0 - deny
  - IF can be modified by software
- non-maskable
  - the processor always services them

The Interrupt Controller (1)

- specialized circuit
- collects interrupt requests from the peripherals
- sends them to the processor
- arbitrates the conflicts (more requests coming simultaneously)
  - each peripheral is assigned a certain priority

The Interrupt Controller (2)

- initially - Intel 8259
  - multiple controllers can be used (cascading)



- today - integrated into the chipset

Interrupt Handling - Phases (1)

- the peripheral generates an interrupt request on its IRQ_i line
- the controller activates signal INT
- the processor checks the value of the IF flip-flop
  - only for maskable interrupts
  - if 0 - refuses the request; stop
  - if 1 - responds by activating signal INTA

Interrupt Handling - Phases (2)

- execution of the current program is suspended
- the registers (including the PC) are saved on the stack
- the IF flip-flop is reset
  - blocks the execution of another interrupt while the current interrupt is serviced
  - can be set again by software

Interrupt Handling - Phases (3)

- identify the peripheral (the source of the request)
  - the controller places a *type* byte on the data bus
  - it indicates the peripheral that made the request
  - at most 2^8=256 interrupt sources
  - each source has its own service routine
  - different peripherals - different services

Interrupt Handling - Phases (4)

- determine the address of the service routine
  - physical address 0 - interrupt vector table
  - contains the addresses of all service routines
  - size: 256 addresses $\times$ 4 bytes = 1 KB
  - type byte = n $\rightarrow$ the address of the service routine = n $\times$ 4

Interrupt Handling - Phases (5)

- jump to the address of the service routine
- execute the service routine
- go back to the interrupted program
  - restore the value of the IF flip-flop
  - restore the values of the registers (from stack)
  - resume the execution of the program from the point it was interrupted

Extension

- this system proved powerful and flexible
- can be extended - broader usage
- programs must be interrupted in other situations as well
  - not only for communication with peripherals
- especially useful for the operating system

Types of Interrupts

- *hardware* - generated by peripherals
- *traps* - generated by the processor itself
  - indicate an abnormal situation
  - example: division by 0
- *software* - generated by programs
  - used to request certain services to the operating system