EMBEDDED SYSTEM DESIGN (AEC551)

B.Tech -ECE-VI Sem

IARE-R16

Prepared By

Mr. N Nagaraju, Assistant Professor, ECE

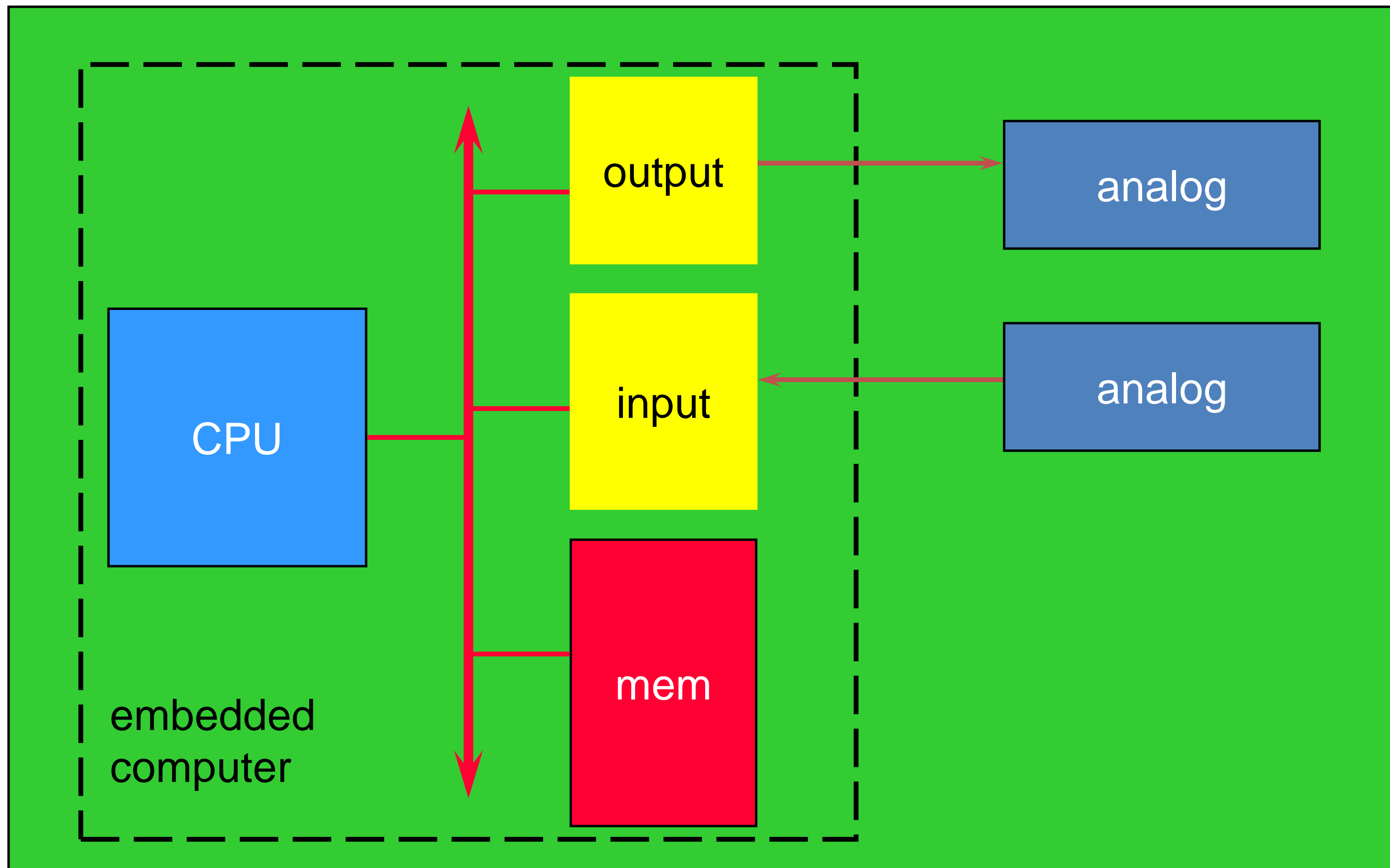
UNIT-I

EMBEDDED COMPUTING

Definition

- Embedded computing system: any device that includes a programmable computer but is not itself a general-purpose computer.
- Take advantage of application characteristics to optimize the design:
  - don't need all the general-purpose bells and whistles.

Embedding a computer



Early history

- Late 1940's: MIT Whirlwind computer was designed for real-time operations.
  - Originally designed to control an aircraft simulator.
- First microprocessor was Intel 4004 in early 1970's.
- HP-35 calculator used several chips to implement a microprocessor in 1972.

Early history, cont'd.

- Automobiles used microprocessor-based engine controllers starting in 1970's.
  - Control fuel/air mixture, engine timing, etc.
  - Multiple modes of operation: warm-up, cruise, hill climbing, etc.
  - Provides lower emissions, better fuel efficiency.

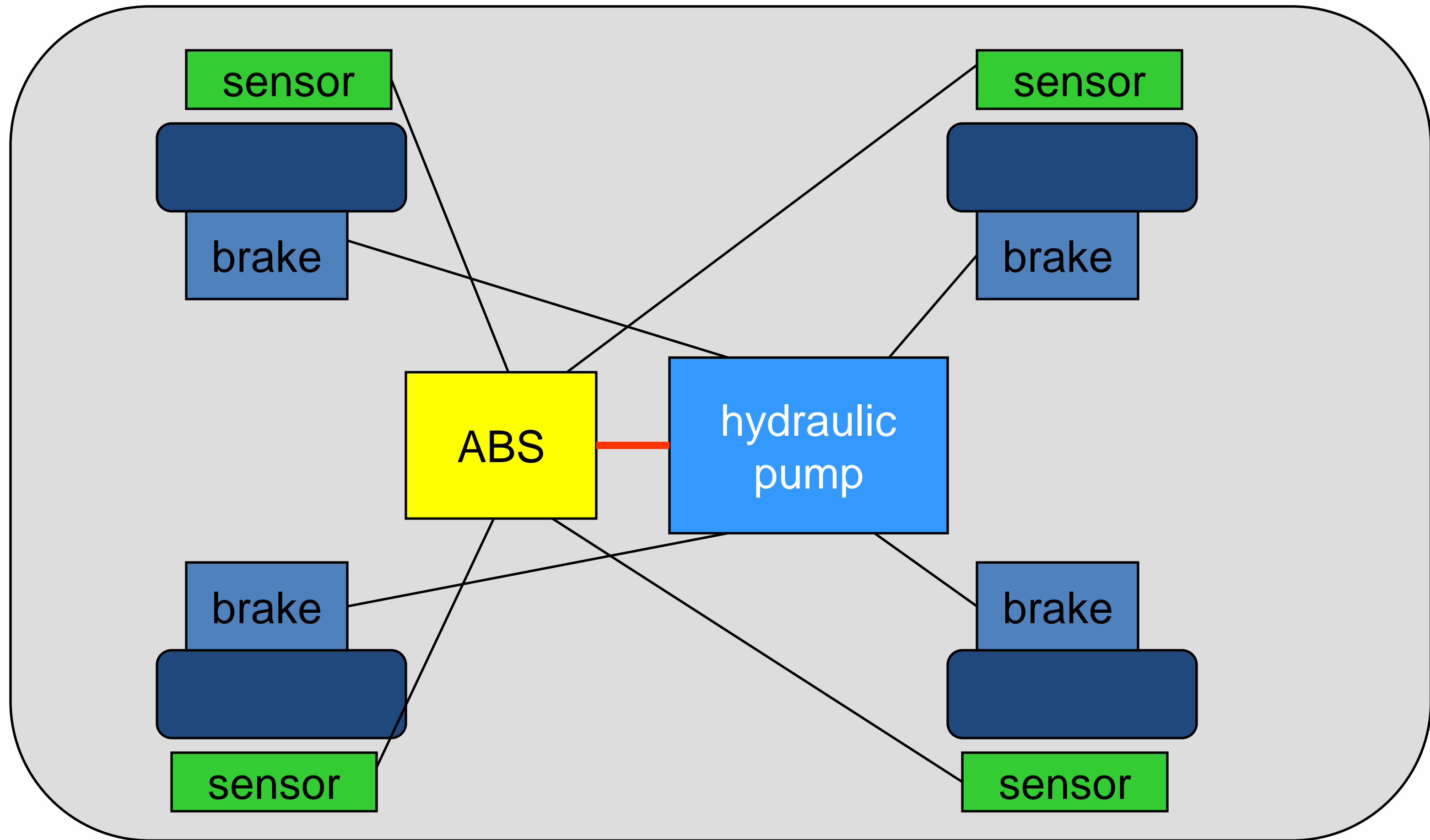
Automotive embedded systems

- Today's high-end automobile may have 100 microprocessors:
  - 4-bit microcontroller checks seat belt;
  - microcontrollers run dashboard devices;
  - 16/32-bit microprocessor controls engine.

BMW 850i brake and stability control system

- Anti-lock brake system (ABS): pumps brakes to reduce skidding.
- Automatic stability control (ASC+T): controls engine to improve stability.
- ABS and ASC+T communicate.
  - ABS was introduced first---needed to interface to existing ABS module.

BMW 850i, cont'd.



Application Areas

- TV
- stereo
- remote control
- phone / mobile phone
- refrigerator
- microwave
- washing machine
- electric tooth brush
- oven / rice or bread cooker
- watch
- alarm clock
- electronic musical instruments
- electronic toys (stuffed animals, handheld toys, pinballs, etc.)
- medical home equipment (e.g. blood
pressure, thermometer)
- …
- [PDAs?? More like standard computer system]

Consumer Products

Application Areas

- **Medical Systems**
  - pace maker, patient monitoring systems, injection systems, intensive care units, …
- **Office Equipment**
  - printer, copier, fax, …
- **Tools**
  - multimeter, oscilloscope, line tester, GPS, …
- **Banking**
  - ATMs, statement printers, …
- **Transportation**
  - (Planes/Trains/[Automobiles] and Boats)
- radar, traffic lights, signalling systems, …

Application Areas

- **Automobiles**
  - engine management, trip computer, cruise control, immobilizer, car alarm,
  - airbag, ABS, ESP, …
- **Building Systems**
  - elevator, heater, air conditioning, lighting, key card entries, locks, alarm systems, …
- **Agriculture**
  - feeding systems, milking systems, …
- **Space**
  - satellite systems, …

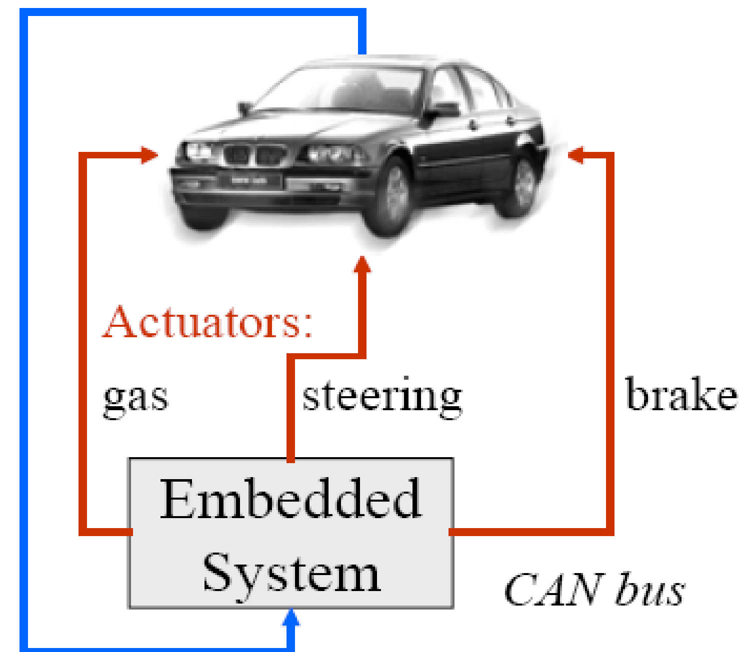
Application Areas

Automobiles

Autonomous cars:

- Electronic gas
- Electronic brake
- Electronic steering

See: *The Daimler Story*



Sensors: Stereo-cameras, speedometer, accelerometers, signalling

Application Areas

Automobiles

2002: Opel Vectra has over 40 sensors (25 types)

Embedded Systems vs General-Purpose Systems

- Embedded System is a **special-purpose computer system** designed to perform one or a few dedicated functions -- Wikipedia
  - In general, it does not provide programmability to users, as opposed to general purpose computer systems like PC
  - Embedded systems are virtually everywhere in your daily life

Embedded Systems (Cont)

- Even though embedded systems cover a wide range of special-purpose systems, there are common characteristics
  - Low cost
    - Should be cheap to be competitive
      - Memory is typically very small compared to a general purpose computer system
      - Lightweight processors are used in embedded systems
  - Low power
    - Should consume low power especially in case of portable devices
    - Low-power processors are used in embedded systems

Embedded Systems (Cont)

- High performance
  - Should meet the computing requirements of applications
    - Users want to watch video on portable devices
      - » Audio should be in sync with video
    - Gaming gadgets like playstation should provide high performance
- Real-time property
  - Job should be done within a time limit
    - Aerospace applications, Car control systems, Medical gadgets are critical in terms of time constraint – Otherwise, it could lead to catastrophe such as loss of life
  - Will talk more about this

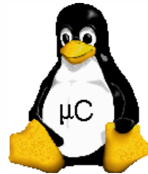
Embedded Systems (Cont)

- It is challenging to satisfy the characteristics
  - You may not be able to achieve high performance while maintaining low power consumption and making use of cheap components
  - So, you got to do your best in a given circumstance to be competitive in the market

HW/SW Stack of Embedded Systems

- Identical to the general-computer systems

Application Software



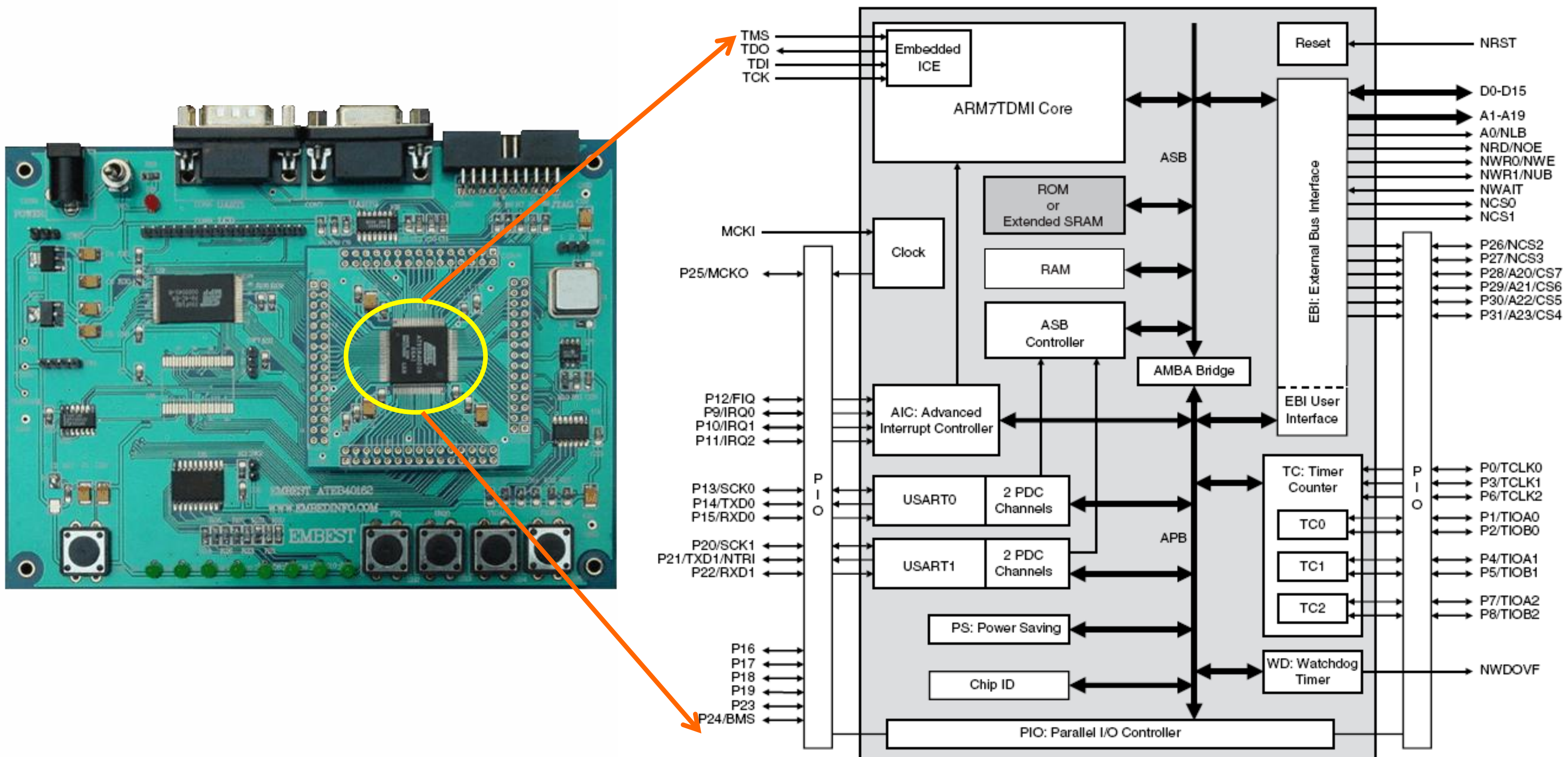
VxWorks

OS / Device Drivers

Hardware

Components of Embedded Systems

- **Hardware**
  - It is mainly composed of processor (1 or more), memory, I/O devices including network devices, timers, sensors etc.



Components of Embedded Systems

- **Software**
  - System software
    - Operating systems
      - Many times, a multitasking (multithreaded) OS is required, as embedded applications become complicated
        - » Networking, GUI, Audio, Video
        - » Processor is context-switched to process multiple jobs
      - Operating system footprint should be small enough to fit into memory of an embedded system
        - » In the past and even now, real-time operating systems (RTOS) such as VxWorks or uC/OS-II have been used because they are light-weighted in terms of memory requirement
        - » Nowadays, little heavy-weighted OSs such as Windows-CE or embedded Linux (uClinux) are used, as embedded processors support computing power and advanced capabilities such as MMU (Memory Management Unit)
    - Device drivers for I/O devices

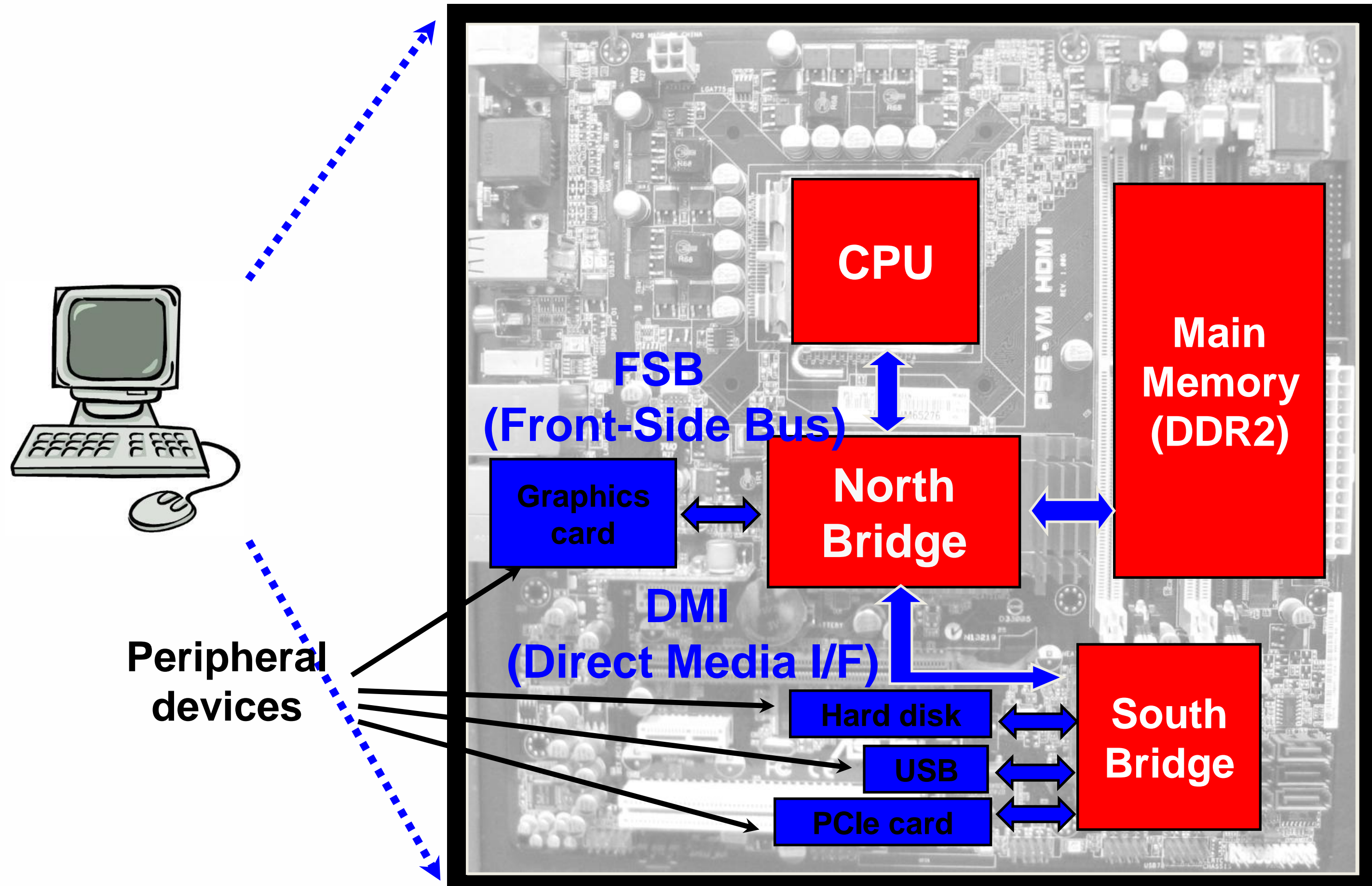
Components of Embedded Systems (Cont)

- **Software (cont.)**
  - Application software
    - Run on top of operating system
    - Execute tasks that users wish to perform
      - Web surfing, Audio, Video playback

Real-Time System

- Real-time operating system (RTOS)
  - Multitasking operating system intended for real-time applications
  - RTOS facilitates the creation of real-time systems
  - RTOS does not necessarily have a high throughput
  - RTOS is valued more for how quickly and/or predictably it can respond to a particular event
    - **Hard real-time systems** are required to complete a critical task within a guaranteed amount of time
    - **Soft real-time systems** are less restrictive
  - Implementing real-time system requires a careful design of scheduler
    - System must have the priority-based scheduling
      - Real-time processes must have the highest priority
      - **Priority inheritance (next slide)**
        - » **Solve the priority inversion problem**
    - Process dispatch latency must be small

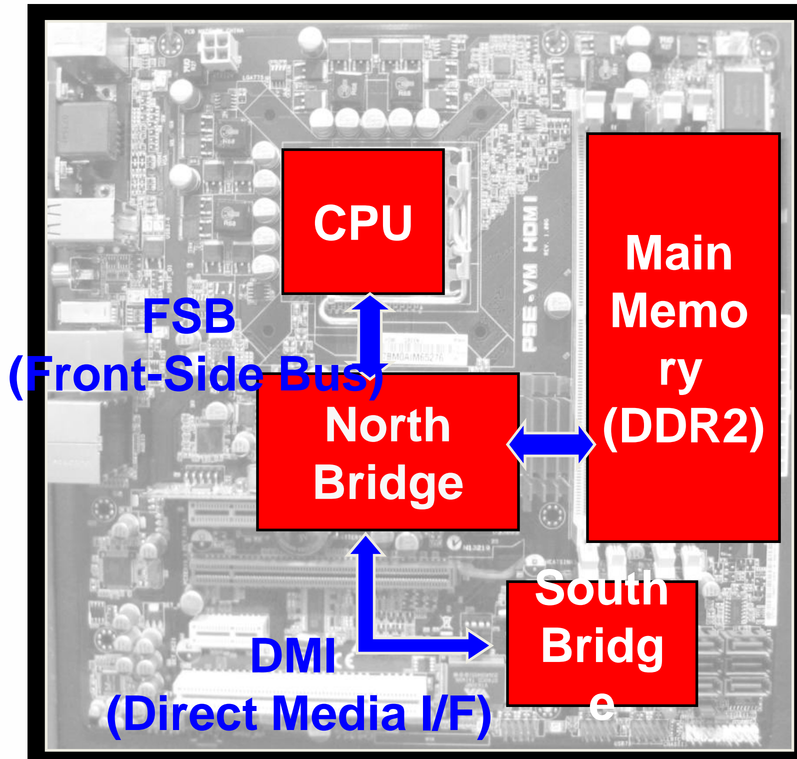
A General-Purpose Computer System



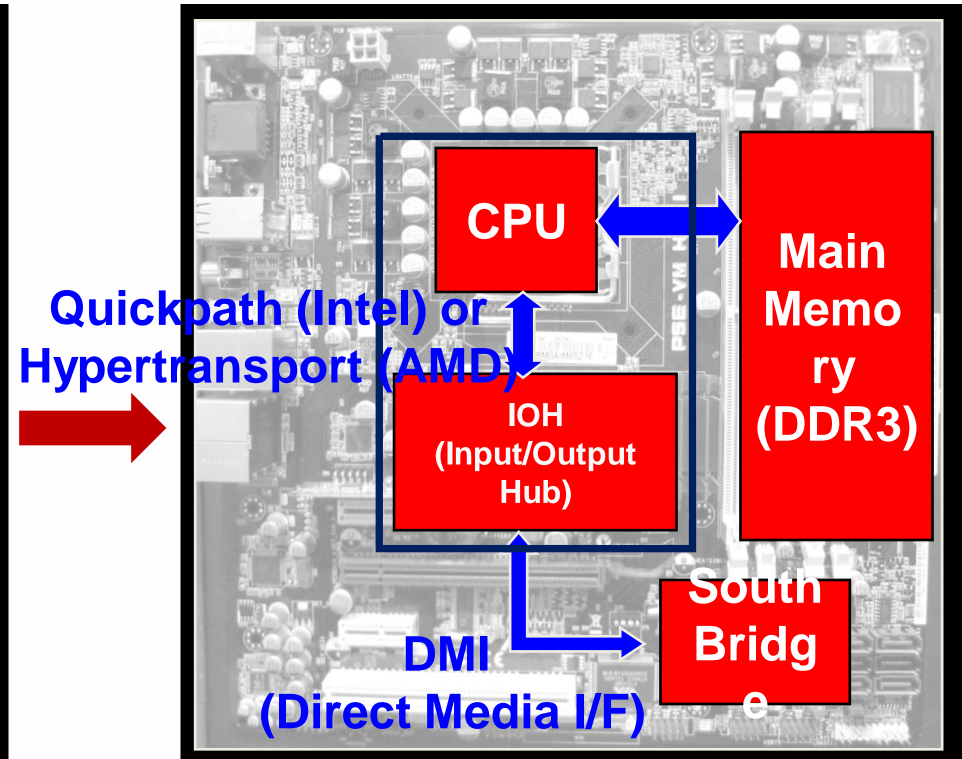
But, don't forget the big picture!

Present, Near Future and More…

- Core 2 Duo – based Systems



- Core i7– based Systems



Keep in mind that CPU and computer systems are evolving at a fast pace

x86 History (as of 2008)

x86 History (Cont.)

1968 Intel Founded

1969 3101 Schottky Bipolar RAM announced.

1971 4004 Microprocessor

1972 8008 Microprocessor

1974 8080 Microprocessor

1979 8088 Microprocessor

1982 286 Microprocessor

1985 INTEL386™ Microprocessor

32-bit (i586)

32-bit (i686)

64-bit (x86_64)

1989 INTEL486™ Microprocessor

1993 Intel® Pentium® Processor

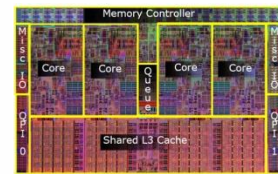
1997 Intel® Pentium® II Processor

1999 Intel® Pentium® III Processor

2000 Intel® Pentium® 4 Processor

2006 Intel® Core 2™ Duo Processor

2008 Intel® Core 2™ Quad Processor



2009

Core i7

x86?

- What is x86?
  - Generic term referring to processors from Intel, AMD and VIA
  - Derived from the model numbers of the first few generations of processors:
    - 80**86**, 802**86**, 803**86**, 804**86** → **x86**
  - Now it generally refers to processors from Intel, AMD, and VIA
    - x86-16: 16-bit processor
    - x86-32 (aka IA32): 32-bit processor   * IA: Intel Architecture
    - x86-64: 64-bit processor

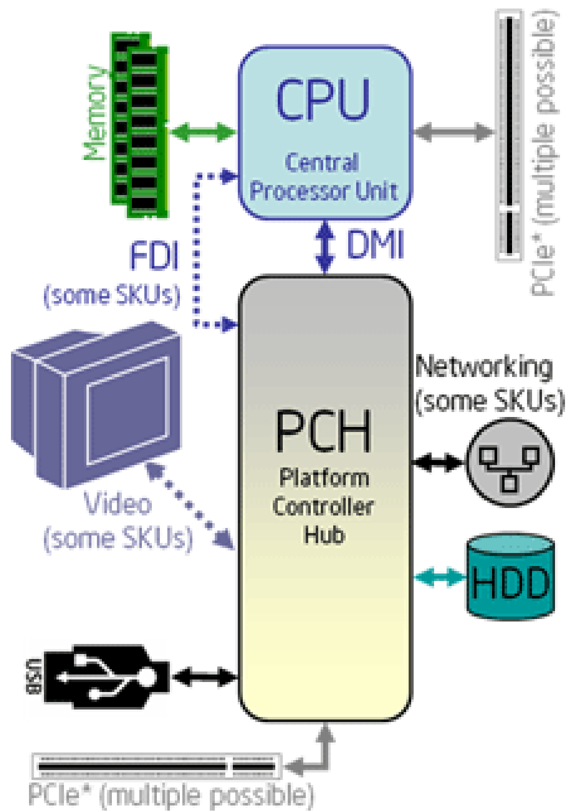
- Intel takes about 80% of the PC market and AMD takes about 20%
  - Apple also have been introducing Intel-based Mac from Nov. 2006

Power under the hood.

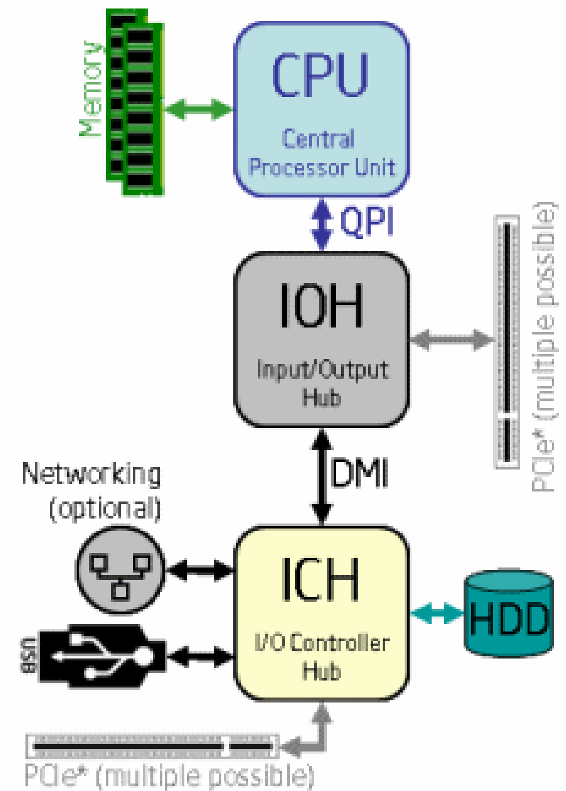
Inside the MacBook is a powerful 2.13GHz Intel Core 2 Duo processor based on advanced Core microarchitecture. MacBook provides a fast 1066MHz frontside bus and 3MB of shared L2 cache, so you'll have more than enough horsepower to get the job done.

Core i7-based Systems

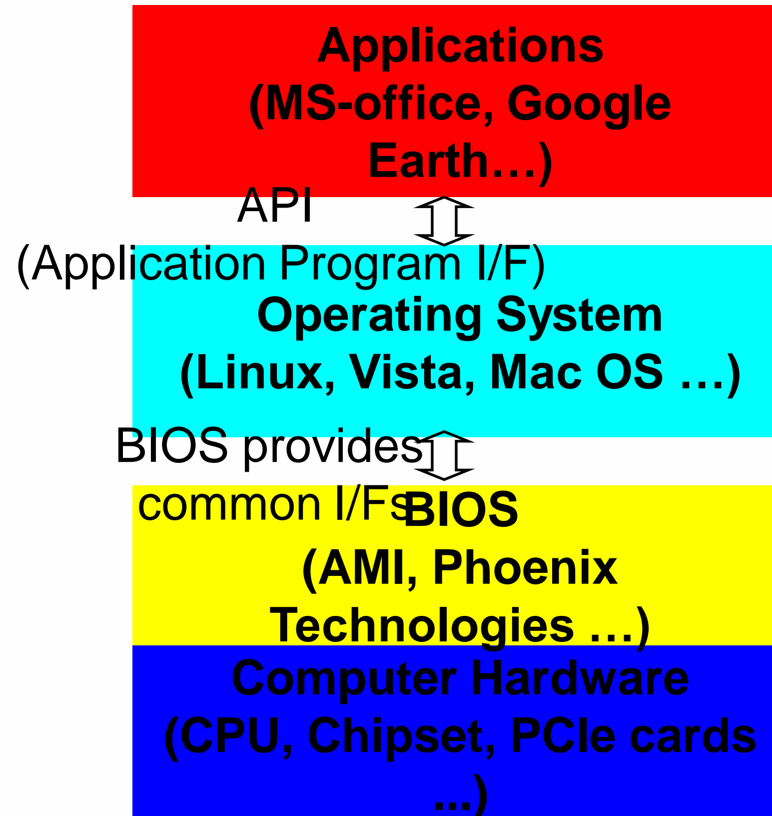
- Core i7 860 (Lynnfield) – based system



- Core i7 920 (Bloomfield) – based system



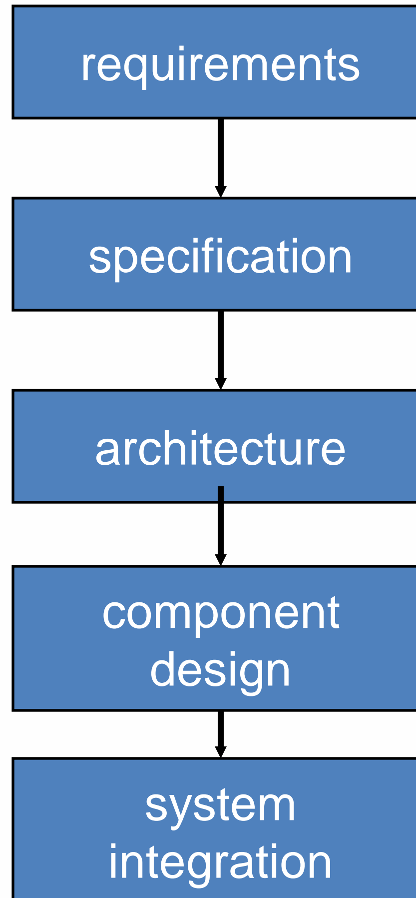
Software Stack



Design goals

- Performance.
  - Overall speed, deadlines.
- Functionality and user interface.
- Manufacturing cost.
- Power consumption.
- Other requirements (physical size, etc.)

Levels of abstraction



Top-down vs. bottom-up

- Top-down design:
  - start from most abstract description;
  - work to most detailed.
- Bottom-up design:
  - work from small components to big system.
- Real design uses both techniques.

Stepwise refinement

- At each level of abstraction, we must:
  - analyze the design to determine characteristics of the current state of the design;
  - refine the design to add detail.

Requirements

- Plain language description of what the user wants and expects to get.
- May be developed in several ways:
  - talking directly to customers;
  - talking to marketing representatives;
  - providing prototypes to users for comment.

Functional vs. non-functional requirements

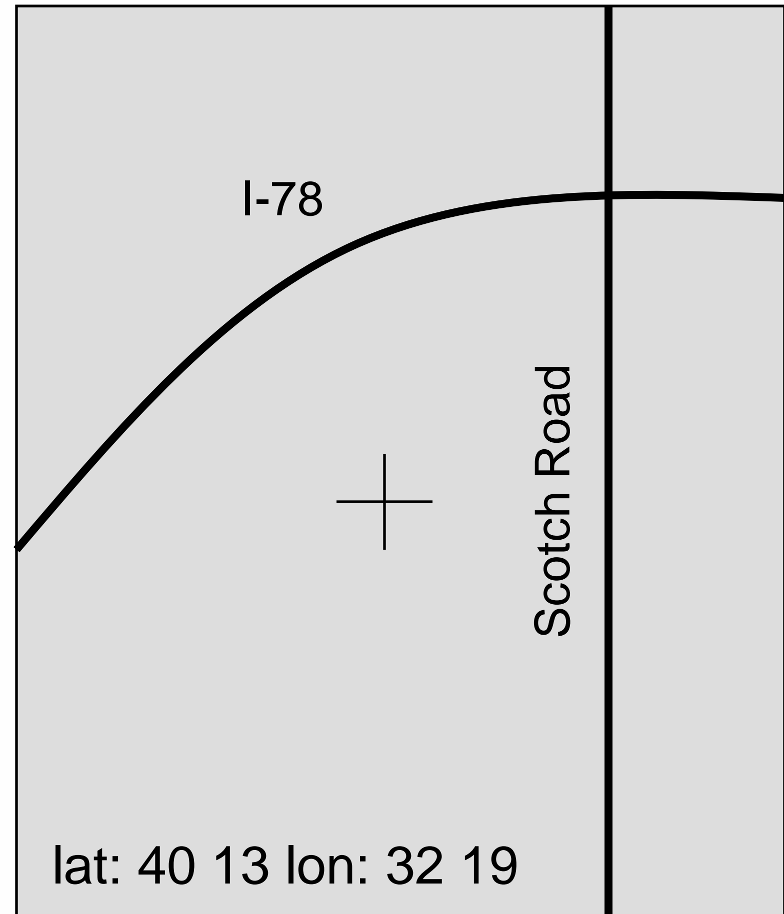
- Functional requirements:
  - output as a function of input.
- Non-functional requirements:
  - time required to compute output;
  - size, weight, etc.;
  - power consumption;
  - reliability;
  - etc.

Our requirements form

name
purpose
inputs
outputs
functions
performance
manufacturing cost
power
physical size/weight

Example: GPS moving map requirements

- Moving map obtains position from GPS, paints map from local database.



GPS moving map needs

- Functionality: For automotive use. Show major roads and landmarks.
- User interface: At least 400 x 600 pixel screen. Three buttons max. Pop-up menu.
- Performance: Map should scroll smoothly. No more than 1 sec power-up. Lock onto GPS within 15 seconds.
- Cost: $120 street price = approx. $30 cost of goods sold.

GPS moving map needs, cont'd.

- Physical size/weight: Should fit in hand.
- Power consumption: Should run for 8 hours on four AA batteries.

GPS moving map requirements form

name	GPS moving map
purpose	consumer-grade moving map for driving
inputs	power button, two control buttons
outputs	back-lit LCD 400 X 600
functions	5-receiver GPS; three resolutions; displays current lat/lon
performance	updates screen within 0.25 sec of movement
manufacturing cost	$100 cost-of-goods-sold
power	100 mW
physical size/weight	no more than 2: X 6:, 12 oz.

Specification

- A more precise description of the system:
  - should not imply a particular architecture;
  - provides input to the architecture design process.
- May include functional and non-functional elements.
- May be executable or may be in mathematical form for proofs.

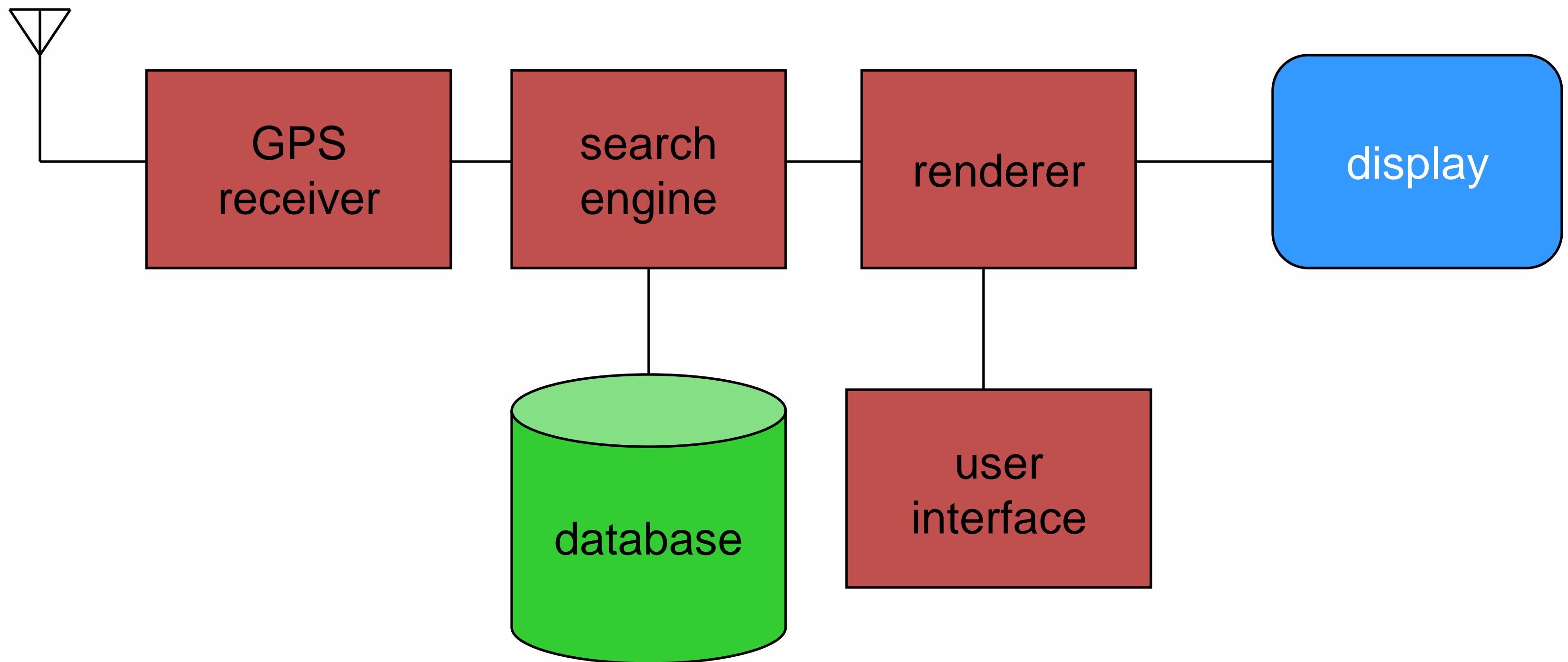
GPS specification

- Should include:
  - What is received from GPS;
  - map data;
  - user interface;
  - operations required to satisfy user requests;
  - background operations needed to keep the system running.

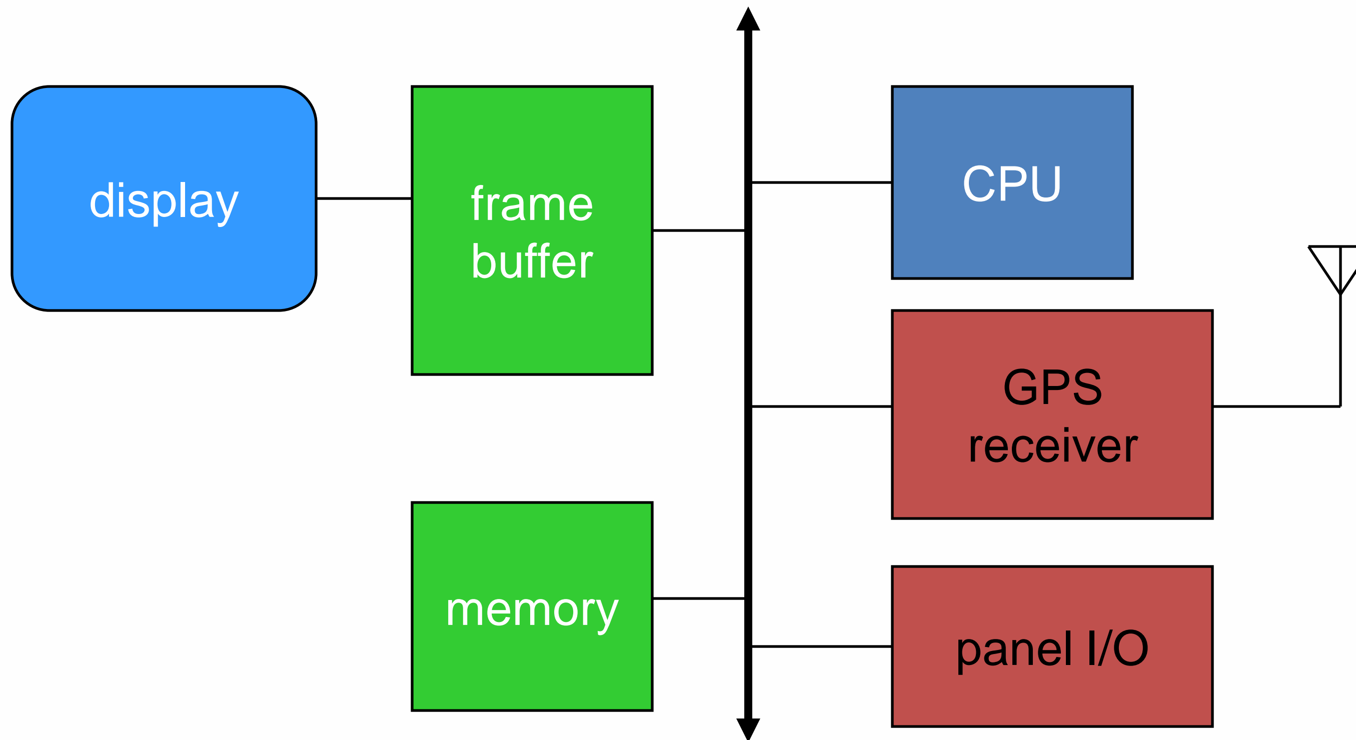
Architecture design

- What major components go satisfying the specification?
- Hardware components:
  – CPUs, peripherals, etc.
- Software components:
  – major programs and their operations.
- Must take into account functional and non-functional specifications.

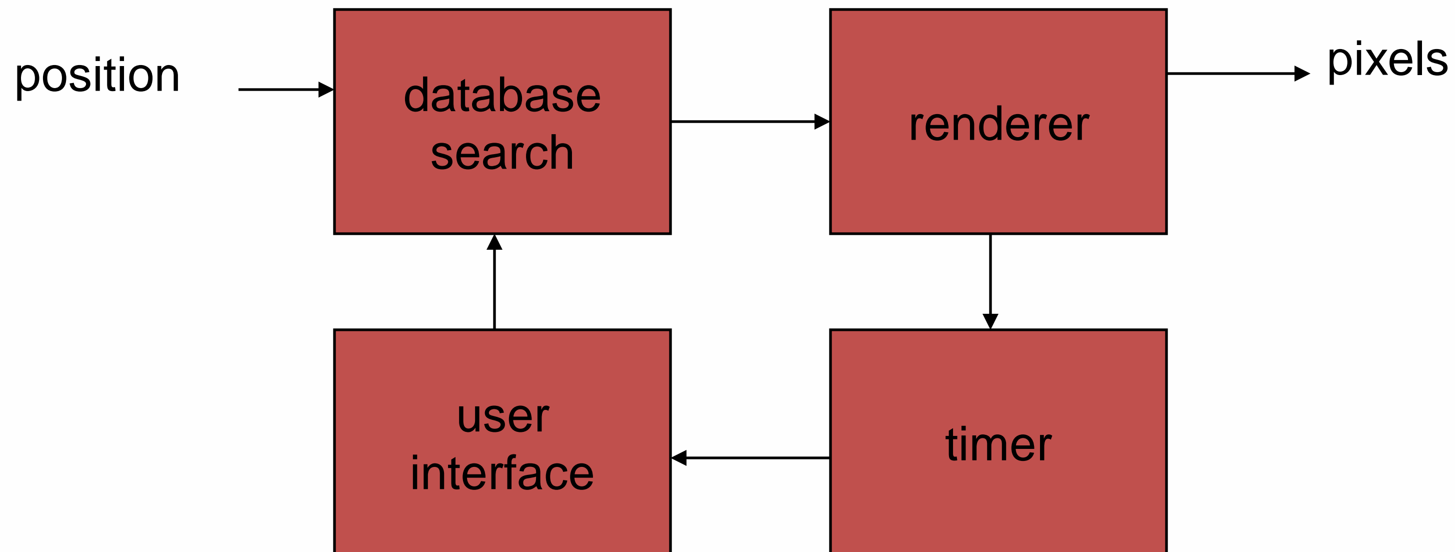
GPS moving map block diagram



GPS moving map hardware architecture



GPS moving map software architecture



Designing hardware and software componen

- Must spend time architecting the system before you start coding.
- Some components are ready-made, some can be modified from existing designs, others must be designed from scratch.

System integration

- Put together the components.
  - Many bugs appear only at this stage.
- Have a plan for integrating components to uncover bugs quickly, test as much functionality as early as possible.

System modeling

- Need languages to describe systems:
  - useful across several levels of abstraction;
  - understandable within and between organizations.
- Block diagrams are a start, but don’t cover everything.

Object-oriented design

- Object-oriented (OO) design: A generalization of object-oriented programming.
- Object = state + methods.
  - State provides each object with its own identity.
  - Methods provide an abstract interface to the object.

Objects and classes

- Class: object type.
- Class defines the object’s state elements but state values may change over time.
- Class defines the methods used to interact with all objects of that type.
  - Each object has its own state.

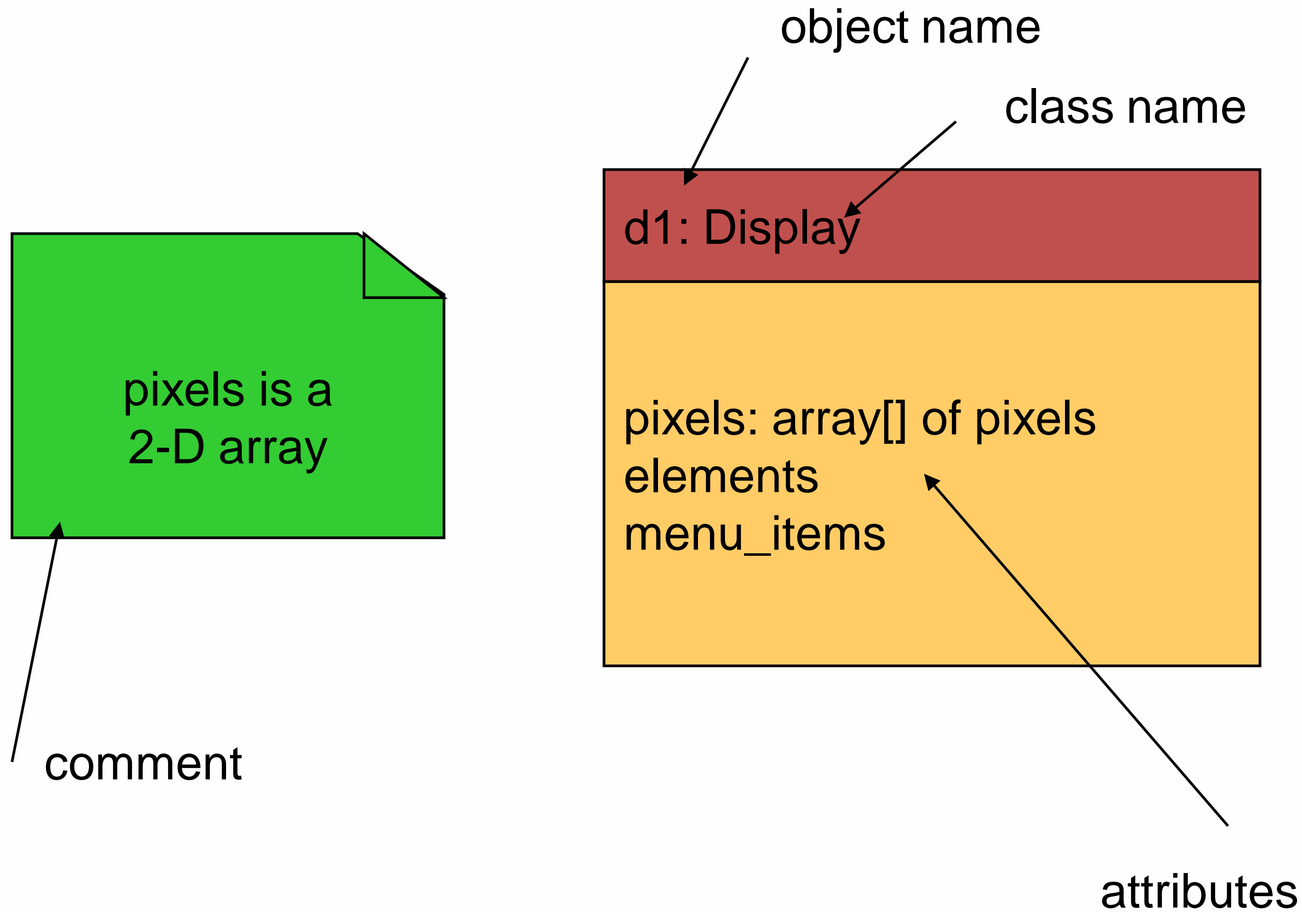
OO design principles

- Some objects will closely correspond to real-world objects.
  - Some objects may be useful only for description or implementation.
- Objects provide interfaces to read/write state, hiding the object's implementation from the rest of the system.

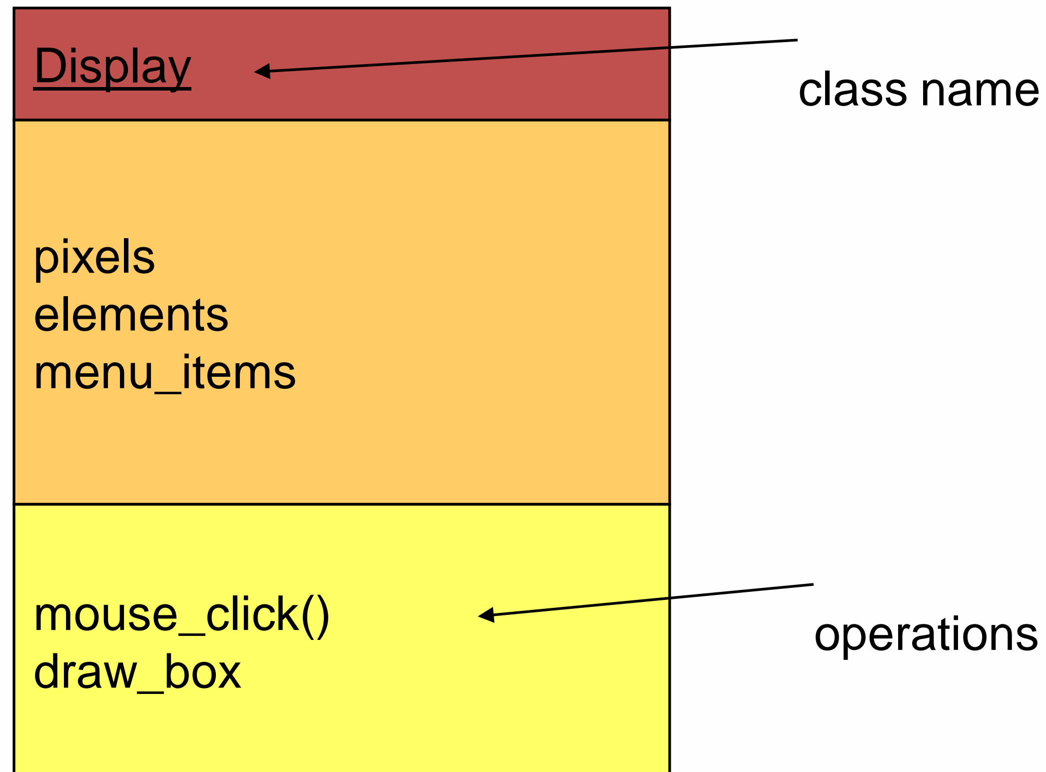
UML

- Developed by Booch et al.
- Goals:
  - object-oriented;
  - visual;
  - useful at many levels of abstraction;
  - usable for all aspects of design.

UML object



UML class



The class interface

- The operations provide the abstract interface between the class's implementation and other classes.
- Operations may have arguments, return values.
- An operation can examine and/or modify the object's state.

Choose your interface properly

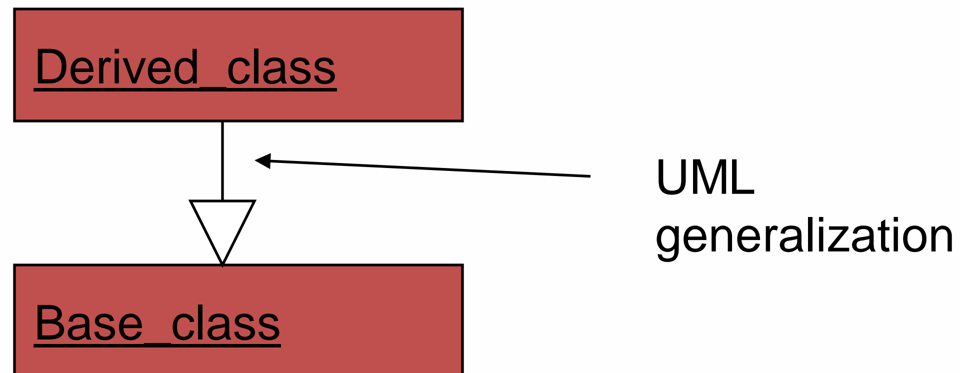
- If the interface is too small/specialized:
  - object is hard to use for even one application;
  - even harder to reuse.
- If the interface is too large:
  - class becomes too cumbersome for designers to understand;
  - implementation may be too slow;
  - spec and implementation are probably buggy.

Relationships between objects and classes

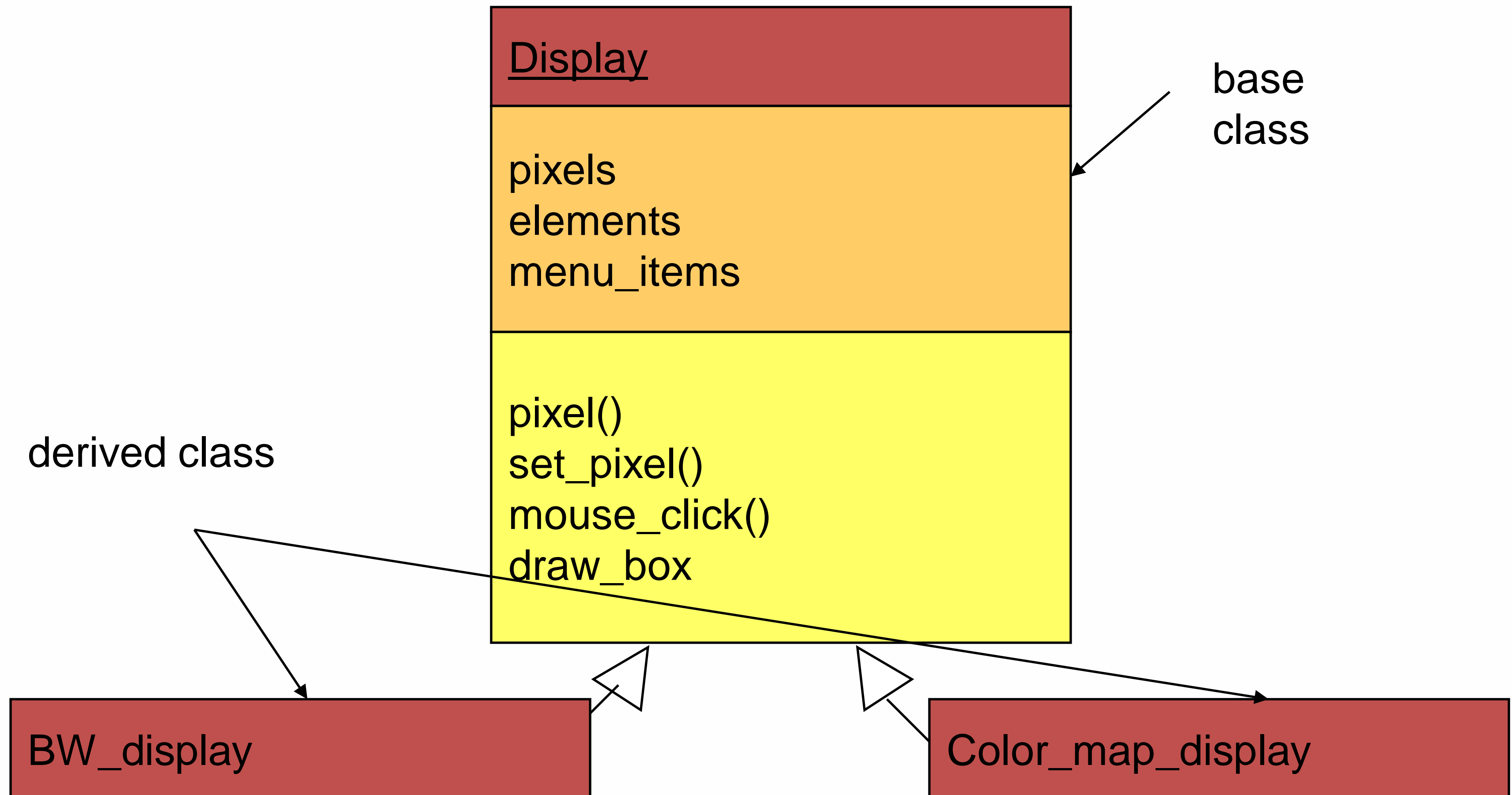
- Association: objects communicate but one does not own the other.
- Aggregation: a complex object is made of several smaller objects.
- Composition: aggregation in which owner does not allow access to its components.
- Generalization: define one class in terms of another.

Class derivation

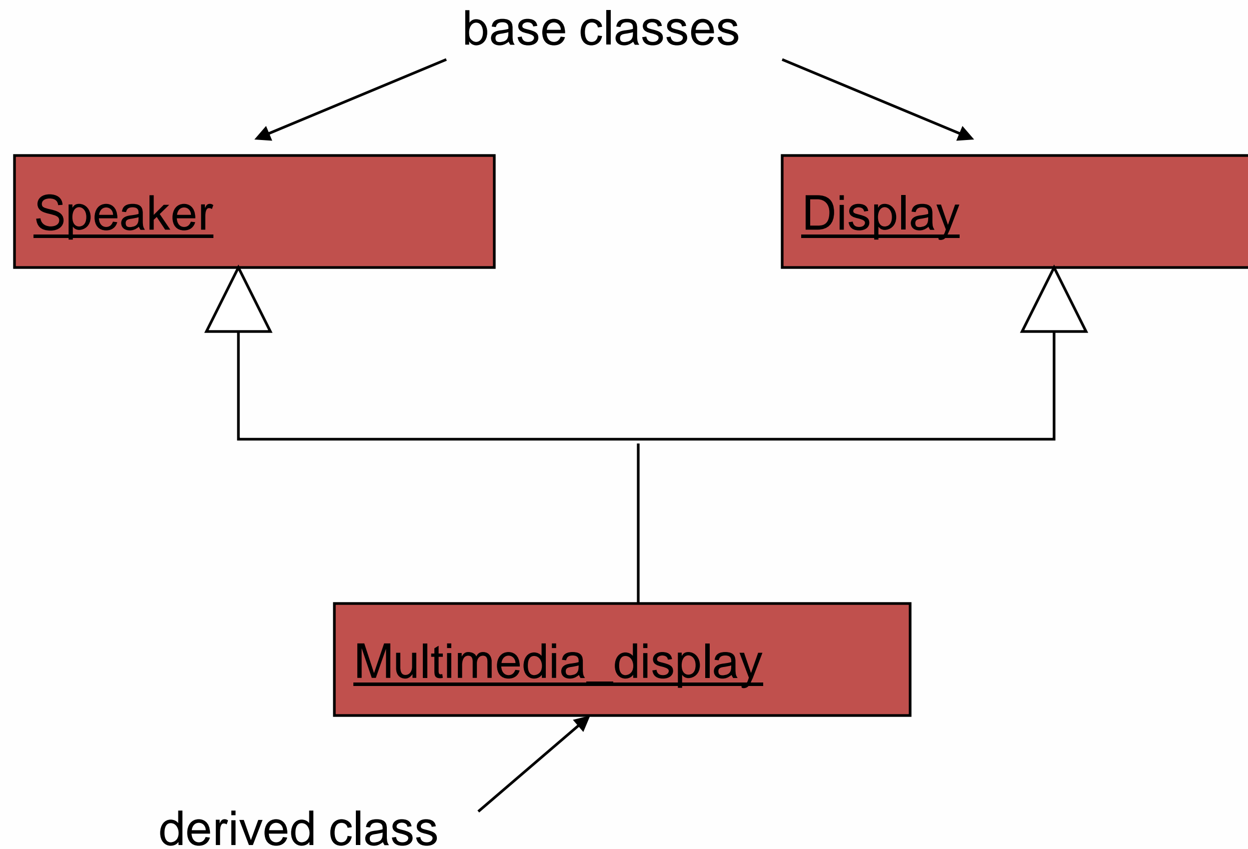
- May want to define one class in terms of another.
  - Derived class inherits attributes, operations of base class.



Class derivation example



Multiple inheritance

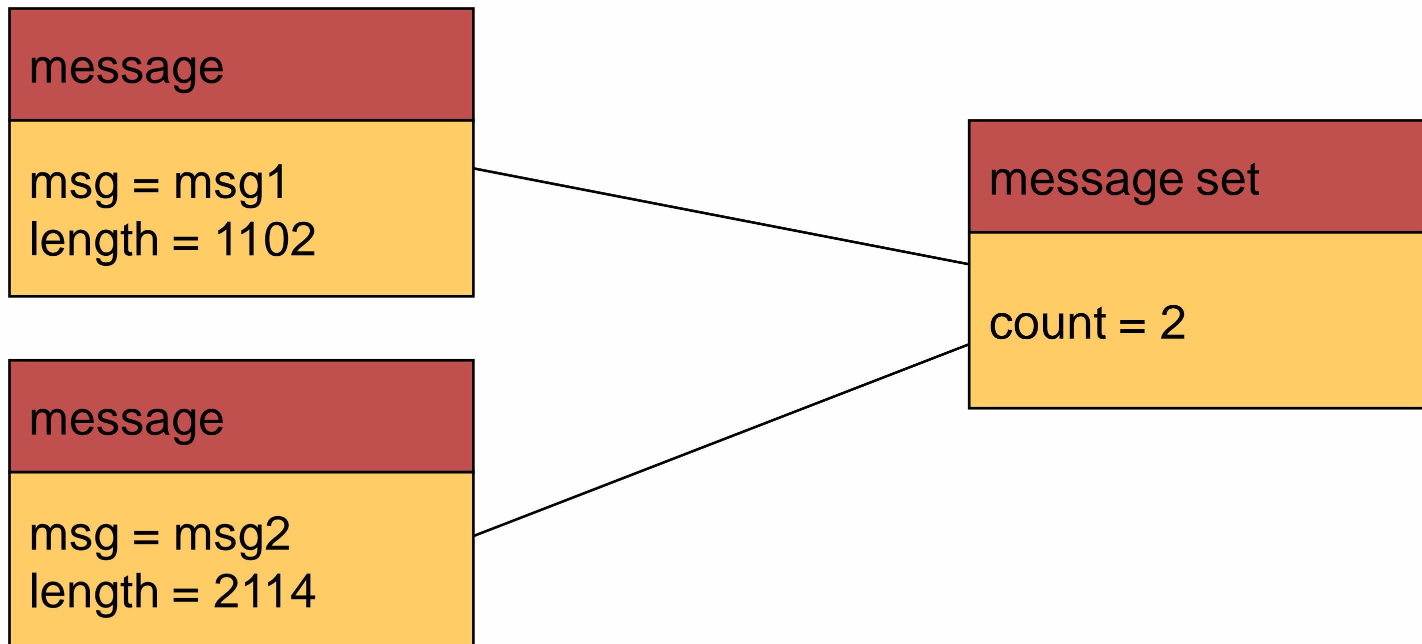


Links and associations

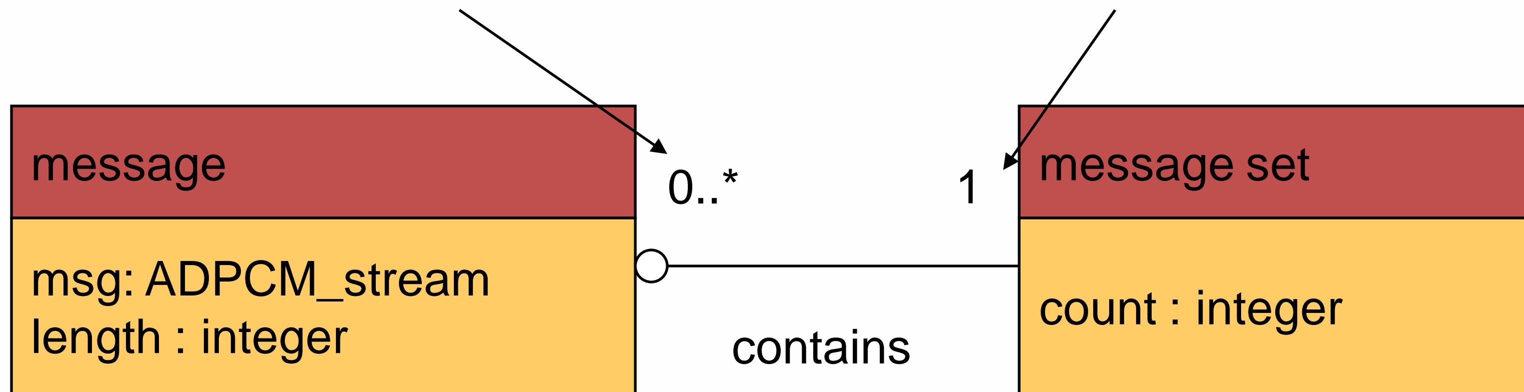
- Link: describes relationships between objects.
- Association: describes relationship between classes.

Link example

- Link defines the contains relationship:



Association example



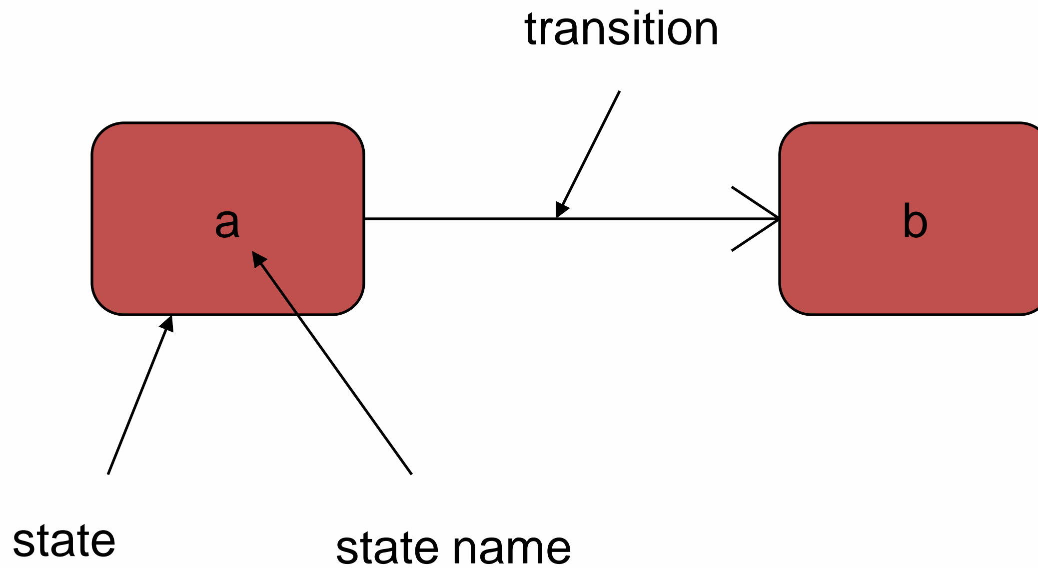
Stereotypes

- Stereotype: recurring combination of elements in an object or class.
- Example:
  - <<foo>>

Behavioral description

- Several ways to describe behavior:
  - internal view;
  - external view.

State machines



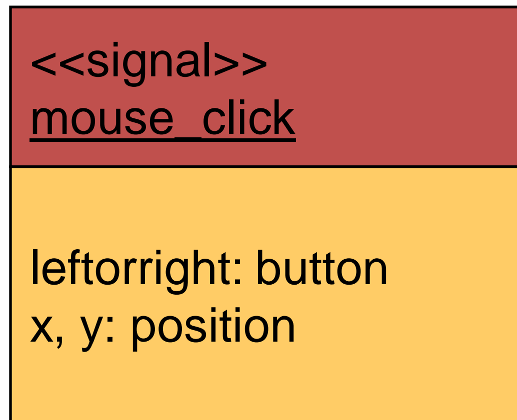
Event-driven state machines

- Behavioral descriptions are written as event-driven state machines.
  - Machine changes state when receiving an input.
- An event may come from inside or outside of the system.

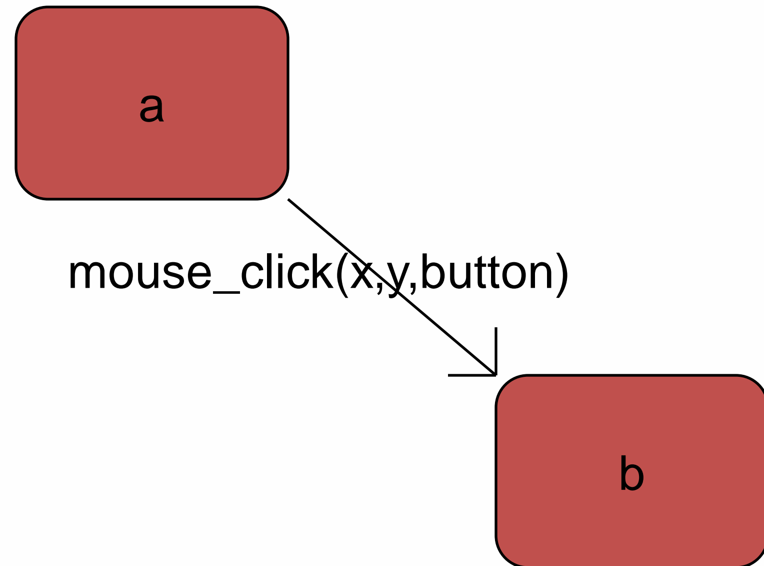
Types of events

- Signal: asynchronous event.
- Call: synchronized communication.
- Timer: activated by time.

Signal event

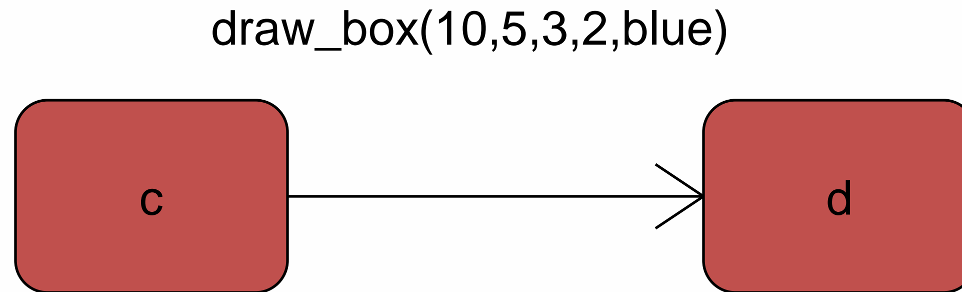


declaration

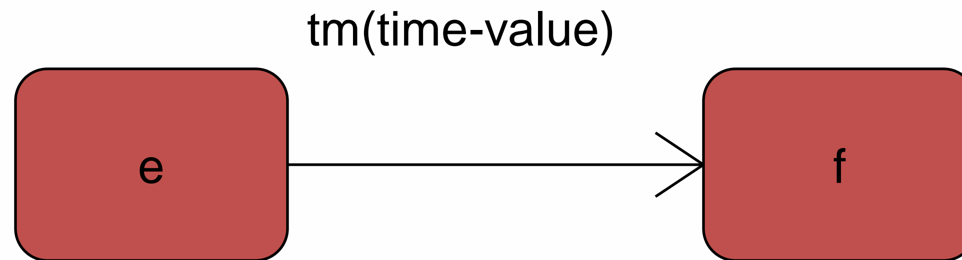


event description

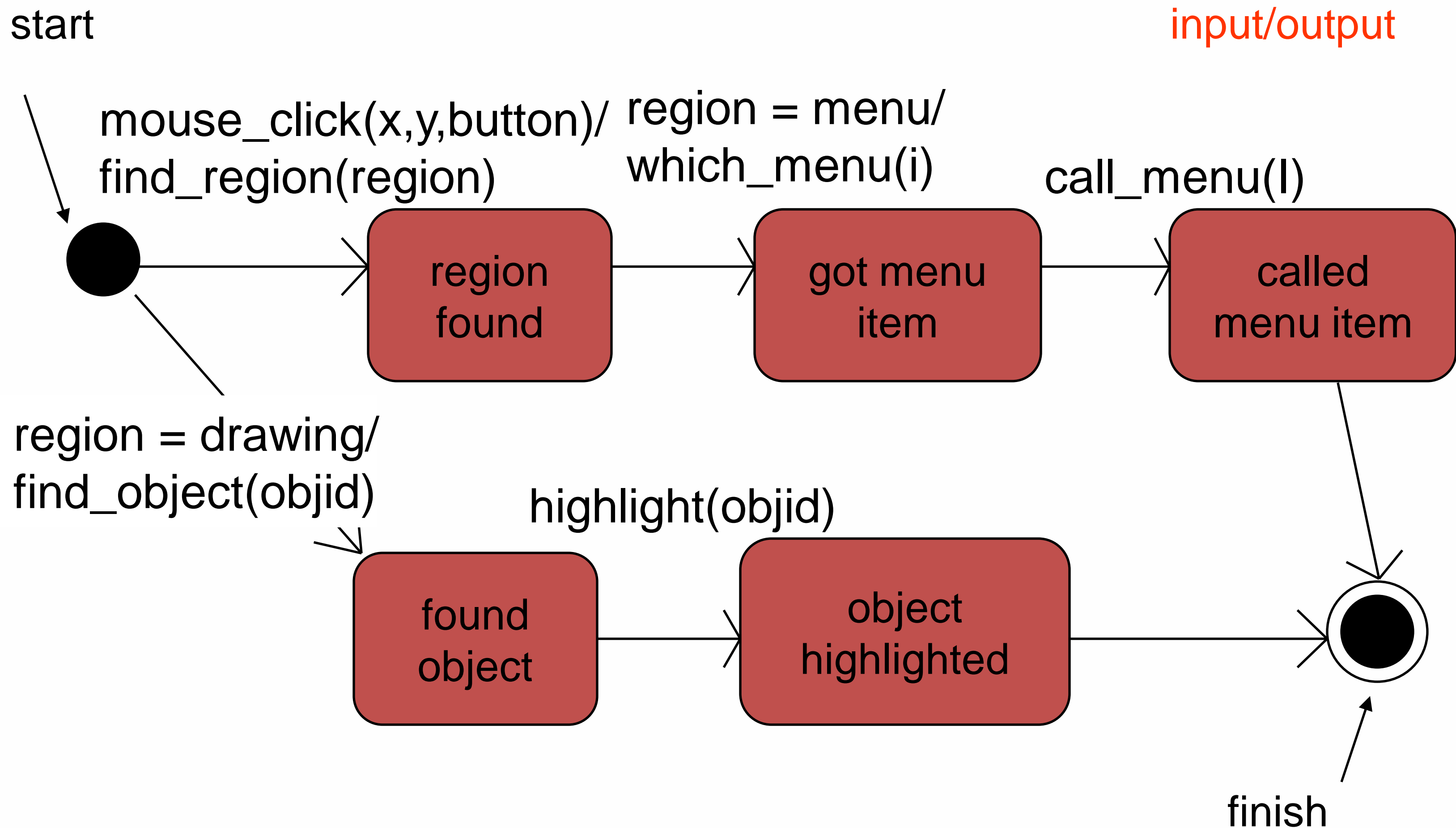
Call event



Timer event



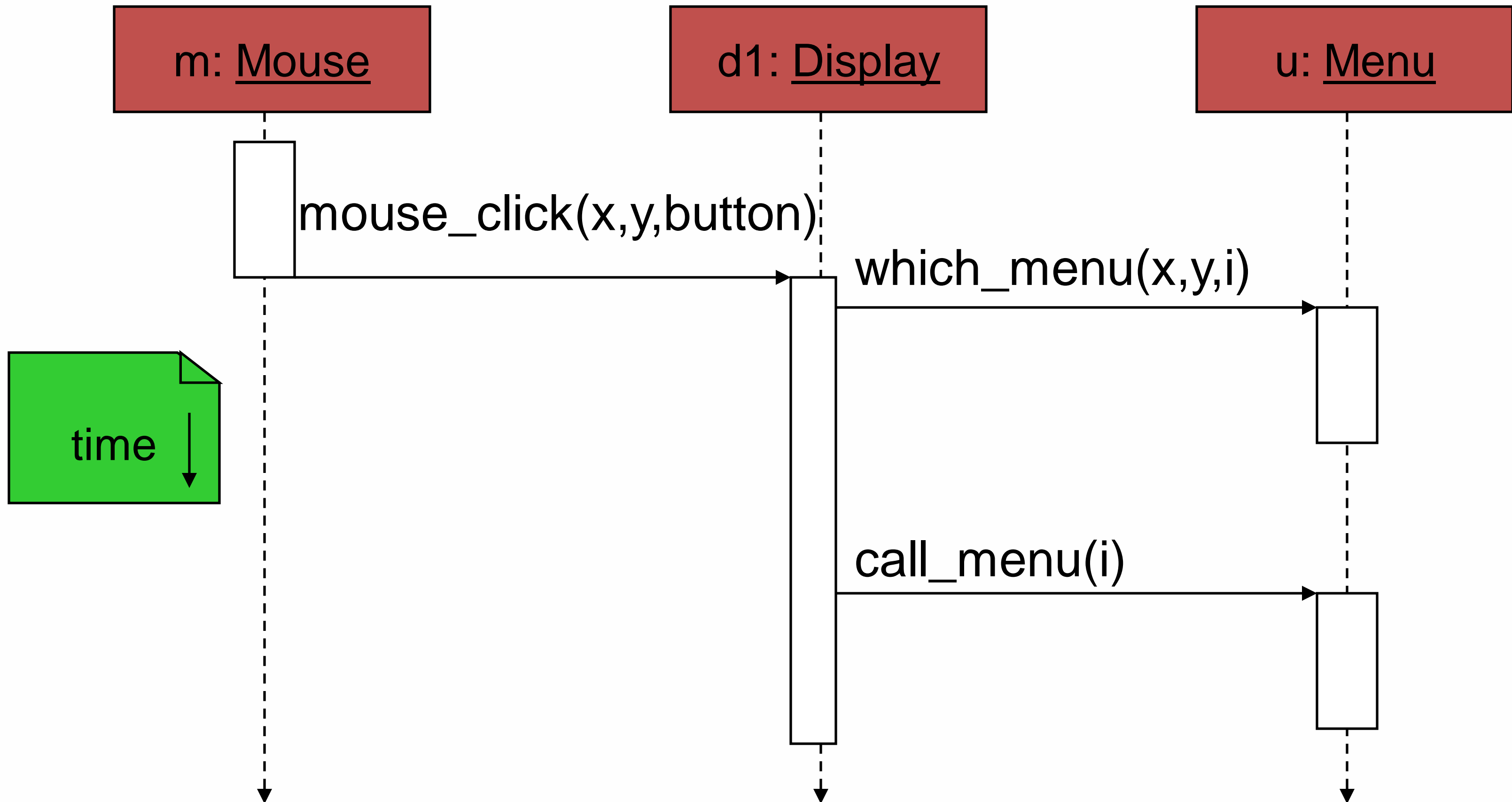
Example state machine



Sequence diagram

- Shows sequence of operations over time.
- Relates behaviors of multiple objects.

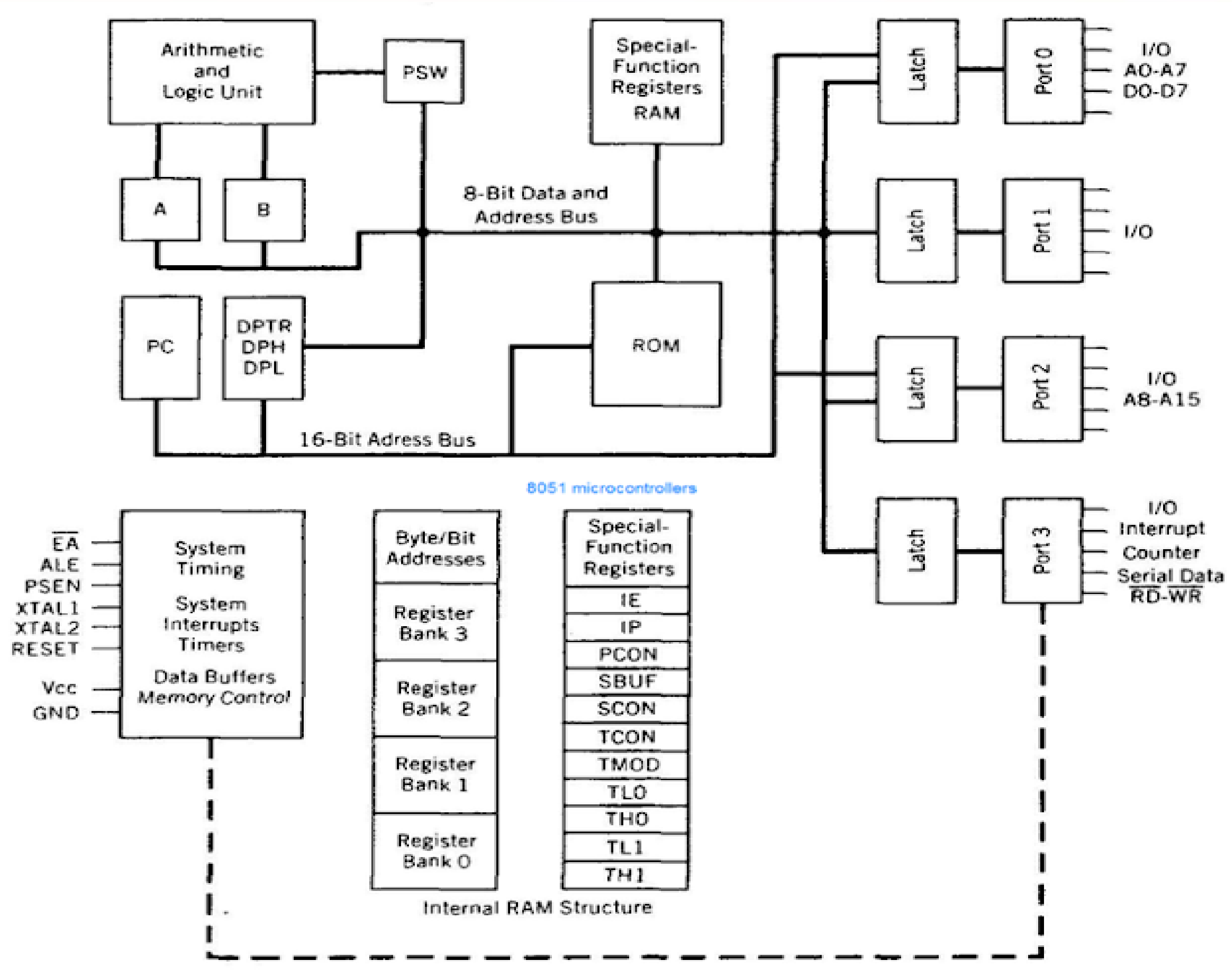
Sequence diagram example



UNIT-II

8051 Microcontroller

Block Diagram



Introduction

- Internal ROM and RAM
- I/O Ports with programmable Pins
- ALU
- Working Registers
- Timers and Counters
- Serial Data Communication.

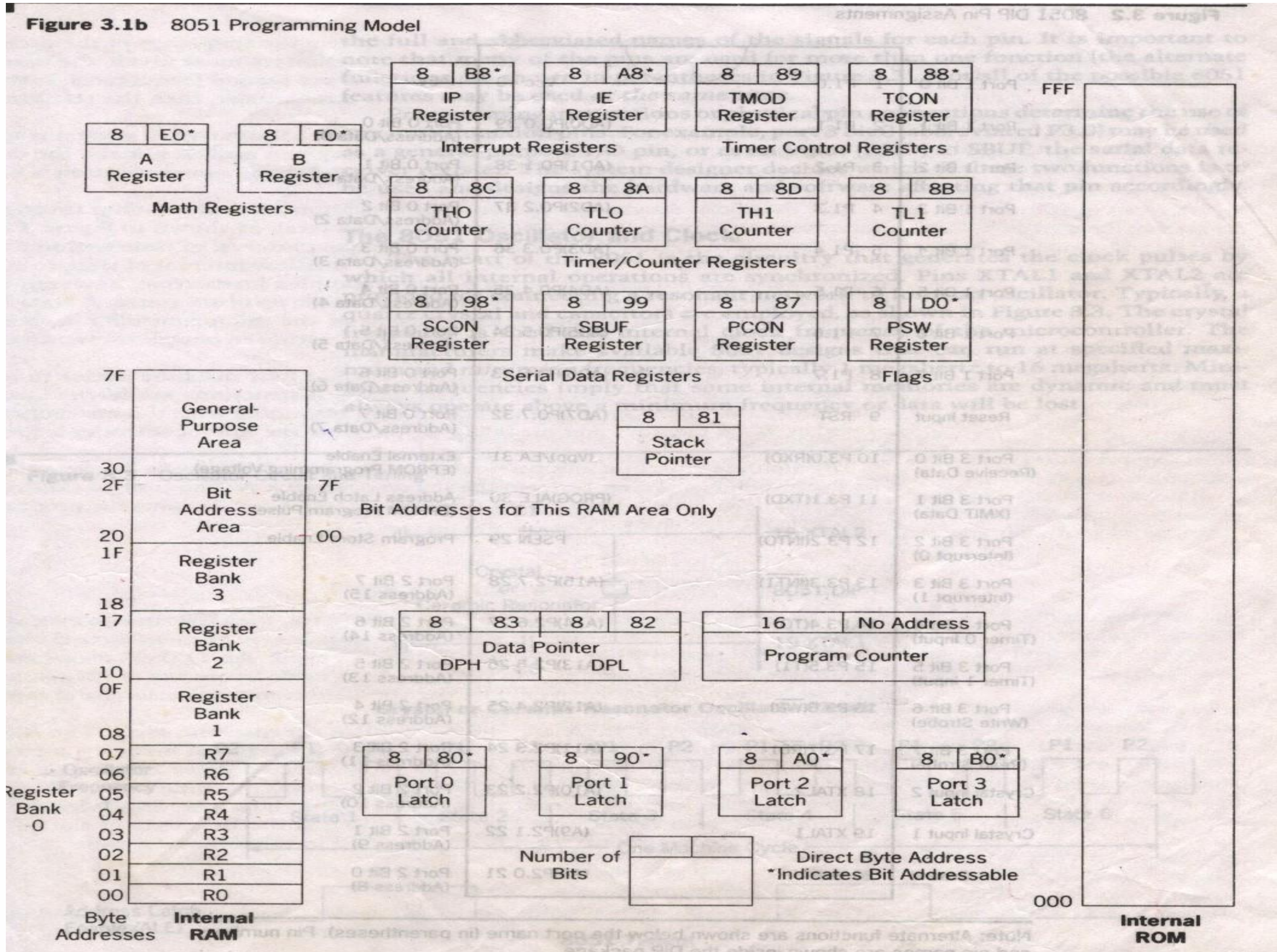
Specific Features

- 8 bit CPU with registers A and B
- 16 bit PC and DPTR(data pointer).
- 8 bit Program Status Word(PSW)
- 8 bit Stack Pointer(SP)
- 4K Internal ROM
- 128bytes Internal RAM
  - →4 register banks each having 8 registers
  - →16 bytes, which may be addressed at the bit level.
  - →80 bytes of general purpose data memory

Specific Features

- 32 i/o pins arranged as four 8 bit ports:P0 to P3
- Two 16 bit timer/counters:T0 and T1
- Full duplex serial data receiver/transmitter: SBUF
- Control registers: TCON, TMOD, SCON, PCON, IP and IE
- Two external and Three internal interrupt sources.
- Oscillator and Clock Circuits.

8051 Programming Model



Program Counter & Data Pointer(DPTR)

- They are both 16 bit registers.
- Each is to hold the address of a byte in memory
- PC contains the address of the next instruction to be executed. (does not have internal address)
- DPTR is made up of two 8 bit register DPH and DPL;
- DPTR contains the address of internal & external code and data that has to be accessed.

A and B CPU registers

- Totally 34 general purpose registers or working registers.
- Two of these A and B hold results of many instructions, particularly math and logical operations of 8051 CPU.
- The other 32 are in four banks, B0 – B3 of eight registers each named as R0 to R7.
- A(accumulator) is used for addition, subtraction, multiplication, division, Boolean bit manipulation and for data transfers (between 8051 and any external memory).
- But B register can only be used for multiplication and division operations.

Flag bits and the PSW register

- Program Status Word Register

CY	AC	F0, GF0, GF1	RS1	RS0	OV	--	P

Carry flag	PSW.7	**CY**
Auxiliary carry flag	PSW.6	**AC**
Available to the user for general purpose	PSW.5	**--**
Register Bank selector bit 1	PSW.4	**RS1**
Register Bank selector bit 0	PSW.3	**RS0**
Overflow flag	PSW.2	**OV**
User define bit	PSW.1	**--**
Parity flag Set/Reset odd(1)/even(0) parity	PSW.0	**P**

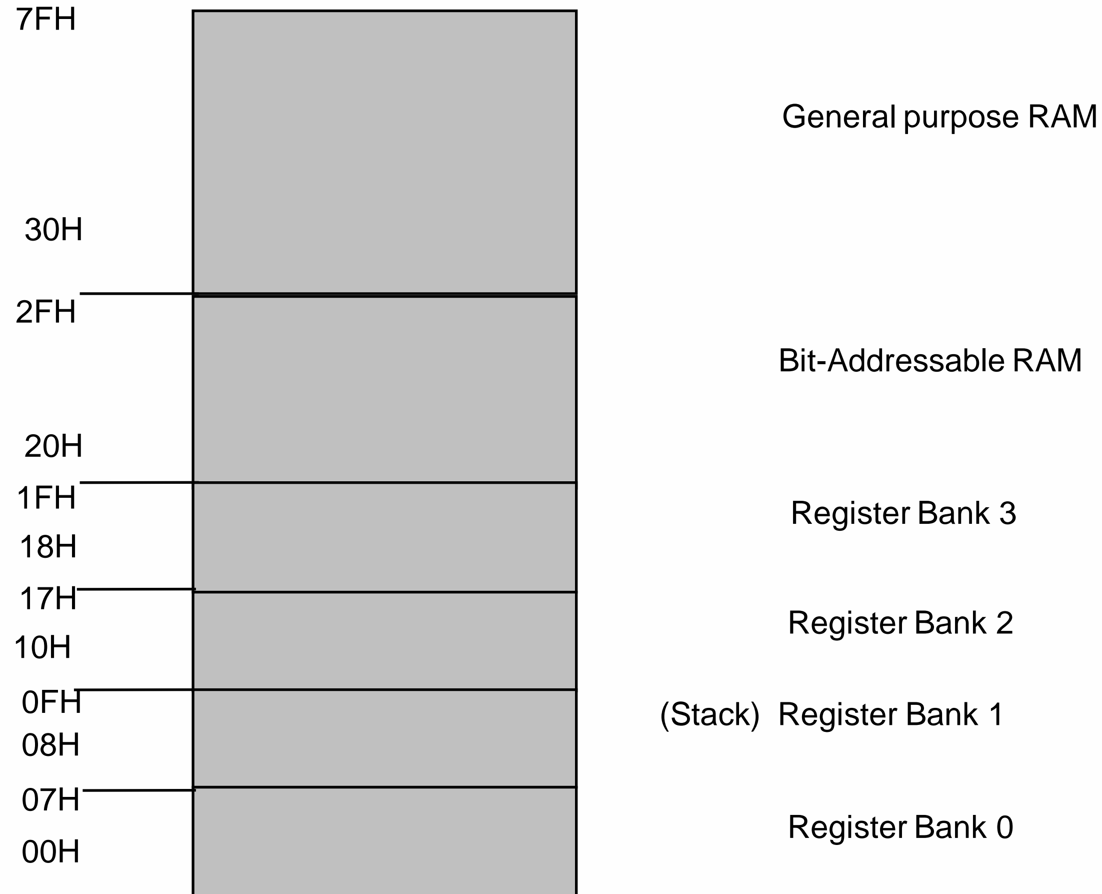
RS1	**RS0**	**Register Bank**	**Address**
0	0	0	00H-07H
0	1	1	08H-0FH
1	0	2	10H-17H
1	1	3	18H-1FH

Flag bits and the PSW register

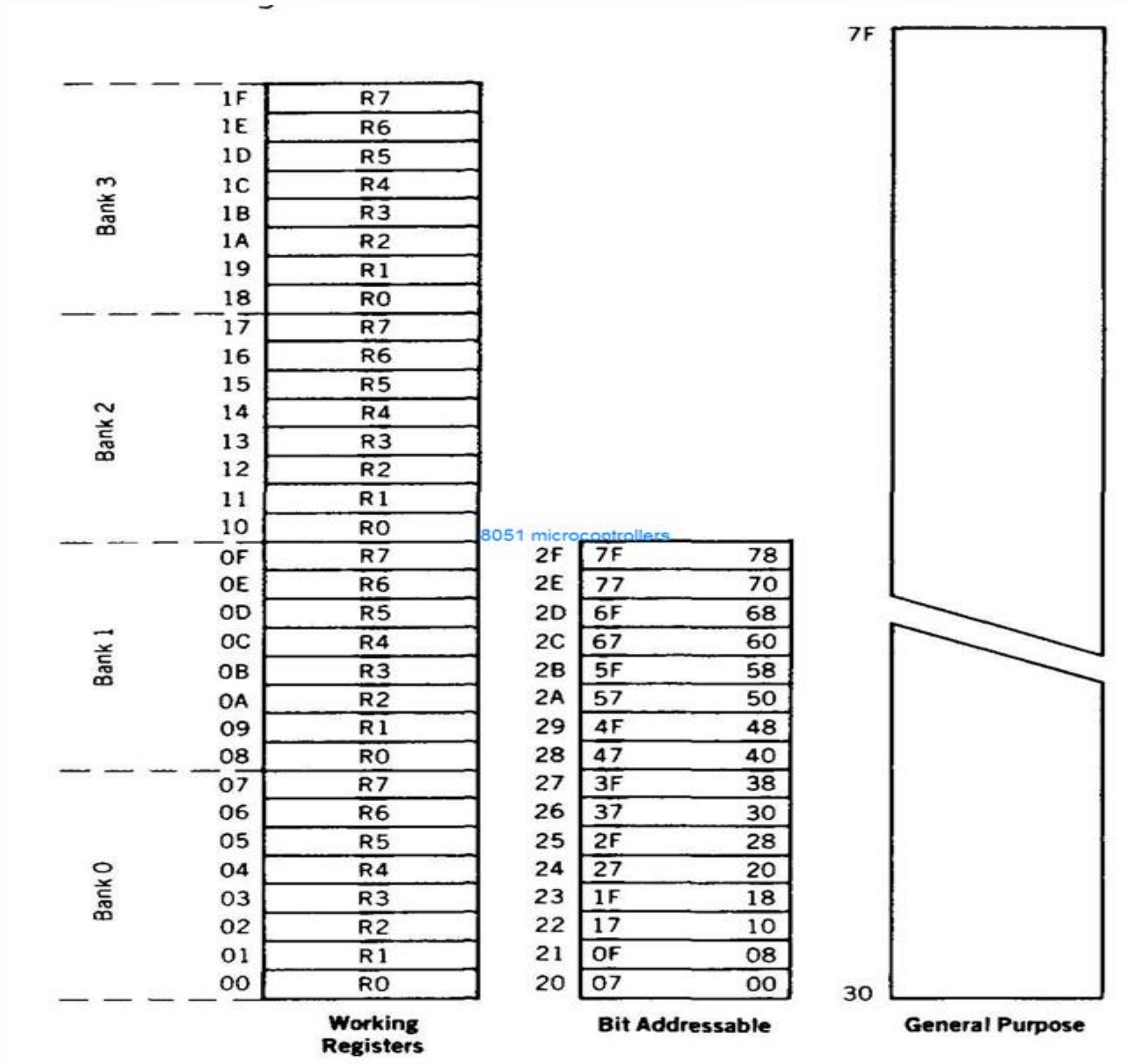
- Two flag bits are stored in PCON(Power control) registers also.
- They are the GF1 (3rd) and GF0(2nd) bits
- They are general purpose user flag bit 1 and 0 respectively
- They can be set or cleared by the program

Memory Organization

- 128 bytes of RAM memory space allocation in the 8051

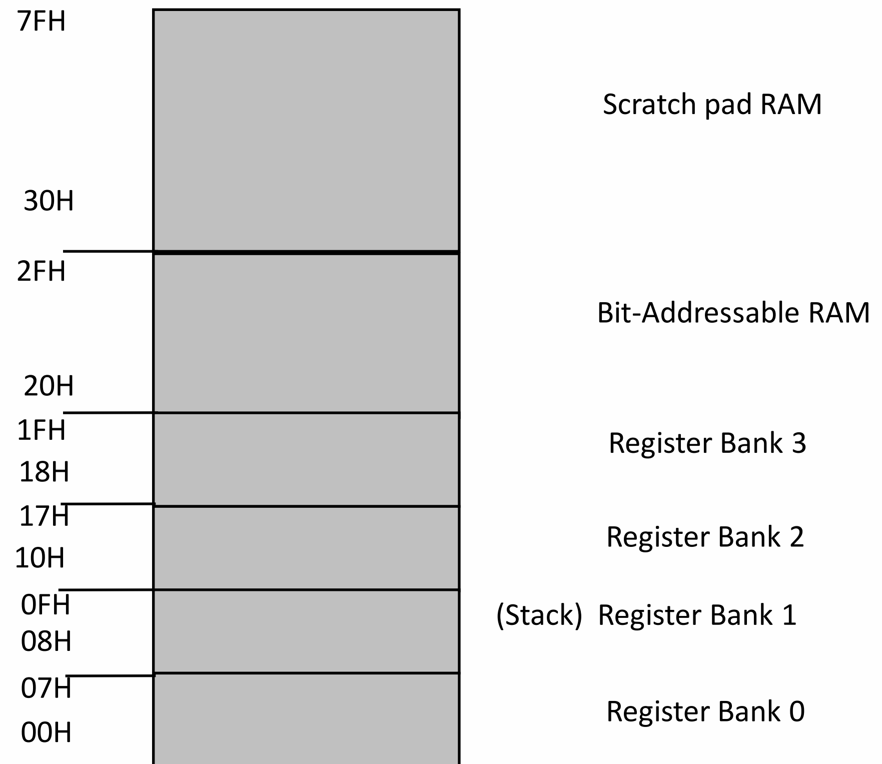


Internal RAM organization

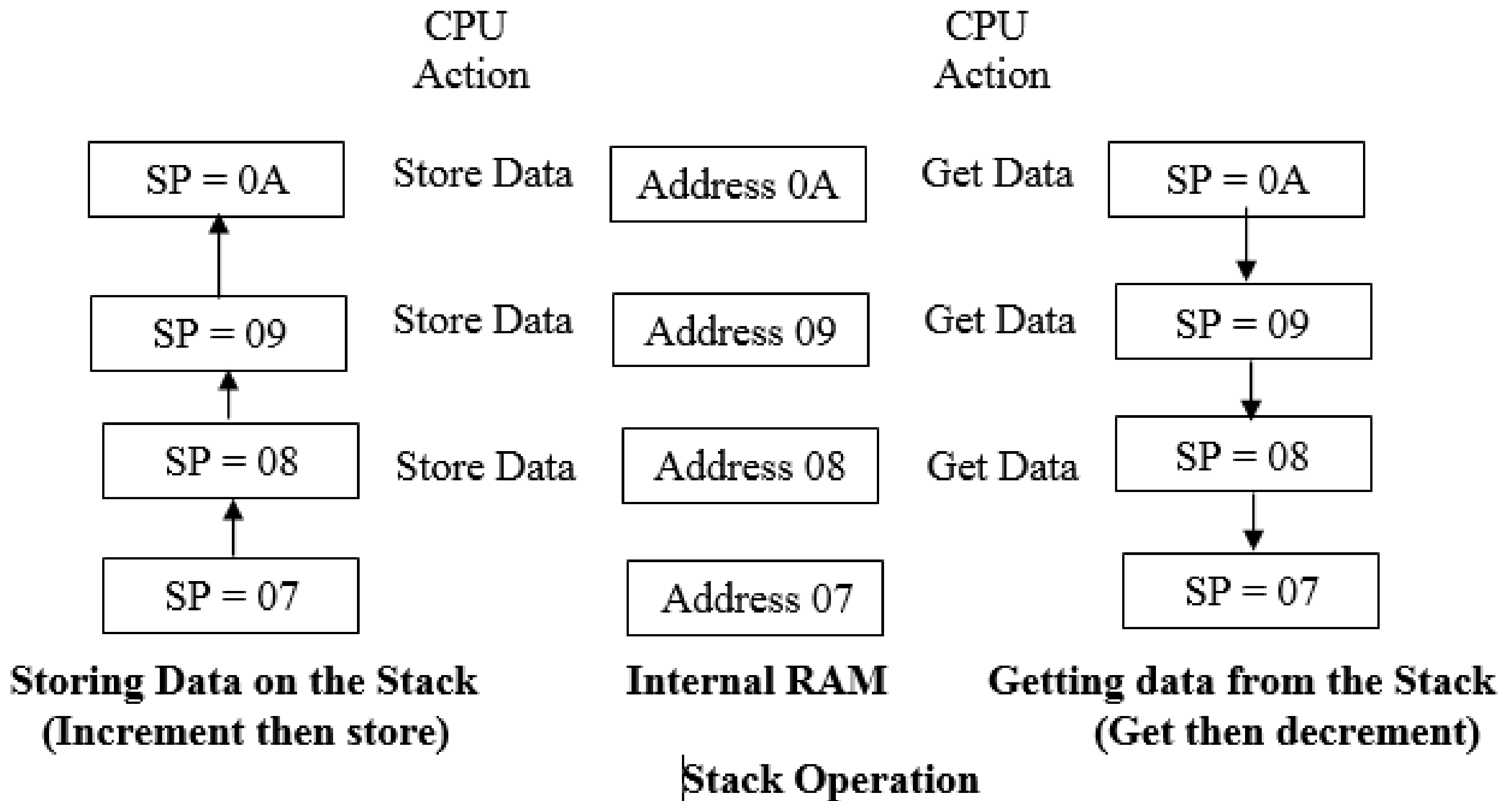


Stack in the 8051

- The register used to access the stack is called **SP** (stack pointer) register.
- The stack pointer in the 8051 is only 8 bits wide, which means that it can take value 00 to FFH. When 8051 powered up, the SP register contains value 07.



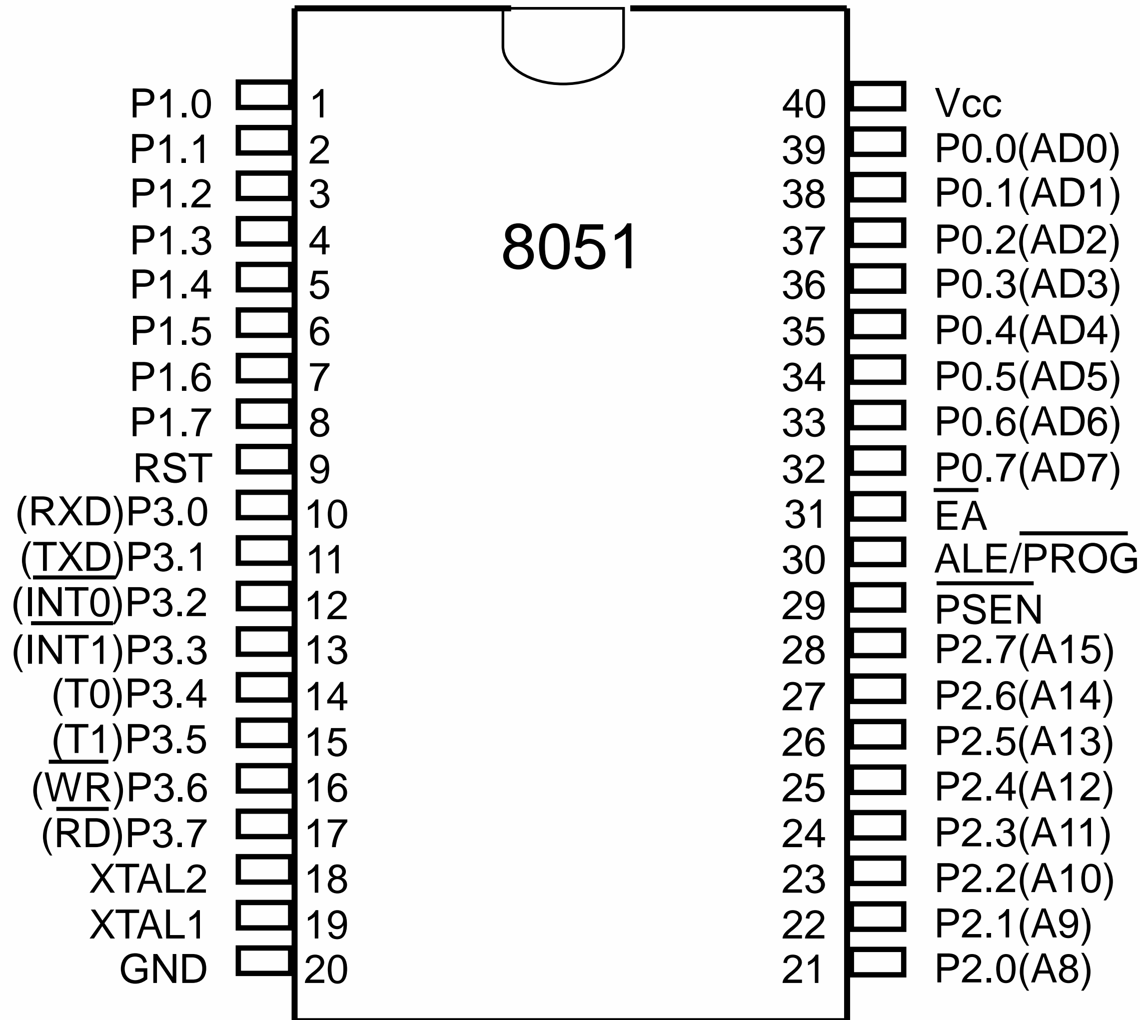
Stack in the 8051



Special Function Registers

Name	Function	Name	Function
A	Accumulator	PSW	Pgm Status Word
B	Arithmetic	SCON	Serial Port Control
DPH	Addressing Ext Memory	SBUF	Serial Port data buffer
DPL	Addressing Ext Memory	SP	Stack Pointer
IE	Interrupt enable	TMOD	Timer/Counter mode control
IP	Interrupt Priority	TCON	Timer/Counter control
P0	I/O Port Latch	TL0	Timer0 lower byte
P1	I/O Port Latch	TH0	Timer0 higher byte
P2	I/O Port Latch	TL1	Timer1 lower byte
P3	I/O Port Latch	TH1	Timer1 higher byte
PCON	Power Control		

Pin Description of the 8051



Pins of 8051（1/4）

- Vcc（pin 40）：
  - Vcc provides supply voltage to the chip.
  - The voltage source is +5V.
- GND（pin 20）：ground
- XTAL1 and XTAL2（pins 19,18）：
  - These 2 pins provide external clock.

Pins of 8051（2/4）

- RST（pin 9）：reset
  - It is an input pin and is active high（normally low）.
    - Upon applying a high pulse to RST, the microcontroller will reset and all values in registers will be lost.

Pins of 8051（3/4）

- /EA（pin 31）：external access
  - The /EA pin is connected to GND to indicate the code is stored externally.
  - For 8051, /EA pin is connected to Vcc. Program fetches the address 0000H through 0FFFH are directed to internal ROM.
  - Program fetches the address 1000H through FFFFH are directed to external ROM
  - “/” means active low.
- /PSEN（pin 29）: program store enable
  - This is an output pin and is connected to the OE pin of the ROM

Pins of 8051（4/4）

- ALE（pin 30）：address latch enable
  - It is an output pin and is active high.
  - 8051 port 0 provides both address and data.
  - When ALE=0, P0 provides data D0-D7.
  - When ALE=1, P0 provides address A0-A7.
  - The ALE pin is used for de-multiplexing the address and data by connecting to the G pin of the 74LS373 latch.
- I/O port pins
  - The four ports P0, P1, P2, and P3.
  - Each port uses 8 pins.
  - All I/O pins are bi-directional.

Pins of I/O Port

- The 8051 has four I/O ports
  - Port 0 （pins 32-39）：P0（P0.0～P0.7）
  - Port 1（pins 1-8） ：P1（P1.0～P1.7）
  - Port 2（pins 21-28）：P2（P2.0～P2.7）
  - Port 3（pins 10-17）：P3（P3.0～P3.7）
  - Each port has 8 pins.
    - Named P0.X （X=0,1,...,7）, P1.X, P2.X, P3.X
    - Ex：P0.0 is the bit 0（LSB）of P0
    - Ex：P0.7 is the bit 7（MSB）of P0
    - These 8 bits form a byte.
- Each port can be used as input or output (bi-direction).

I/O Port Programming

Port 0（pins 32-39）

- When connecting an 8051 to an external memory, the 8051 uses ports to send addresses and read instructions.
  - 16-bit address：P0 provides both address A0-A7,
  - P2 provides address A8-A15.
  - Also, P0 provides data lines D0-D7.
- When P0 is used for address/data multiplexing, it is connected to the 74LS373 to latch the address.

I/O Port Programming

Port 1（pins 1-8）

- Port 1 is denoted by P1.
  - P1.0 ~ P1.7
  - P1 as an output port (i.e., write CPU data to the external pin)
  - P1 as an input port (i.e., read pin data into CPU bus)

I/O Port Programming

Port 3（pins 10-17）

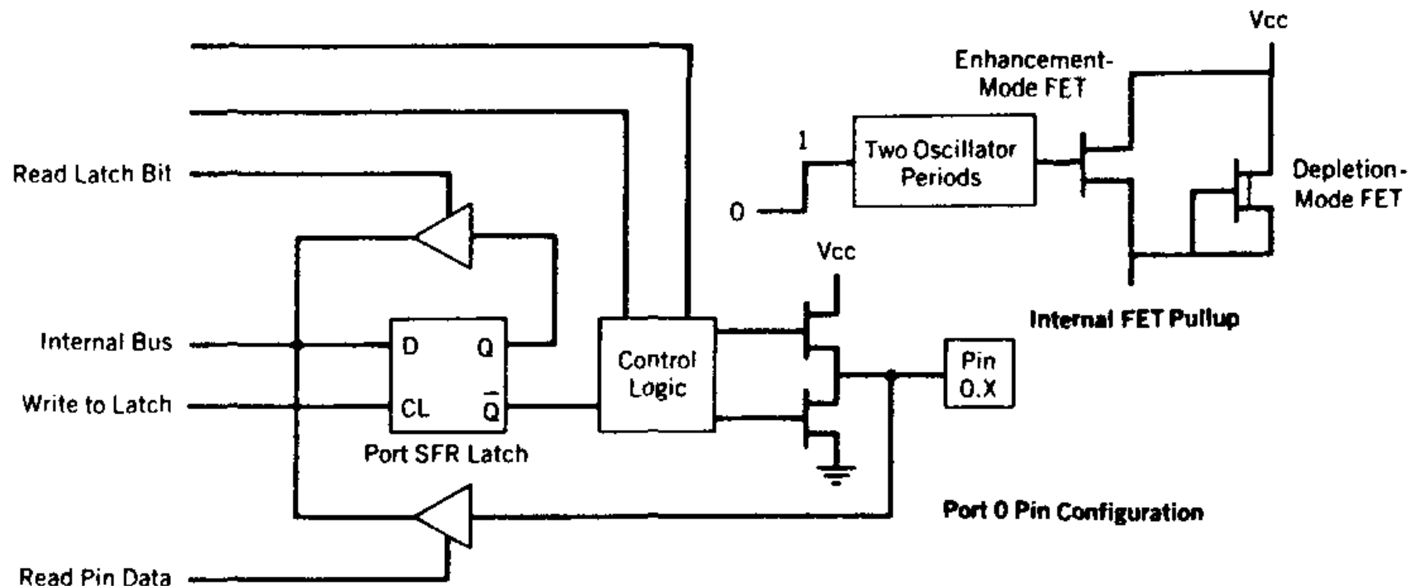
- Although port 3 is configured as an output port upon reset, this is not the way it is most commonly used.
- Port 3 has the additional function of providing signals.
  - Serial communications signal：RxD, TxD
  - External interrupt：/INT0, /INT1
  - Timer/counter：T0, T1
  - External memory accesses ：/WR, /RD

Port 3 Alternate Functions

P3 Bit	Function	Pin
P3.0	RxD	10
P3.1	TxD	11
P3.2	$\overline{\text{INT0}}$	12
P3.3	$\overline{\text{INT1}}$	13
P3.4	T0	14
P3.5	T1	15
P3.6	$\overline{\text{WR}}$	16
P3.7	$\overline{\text{RD}}$	17

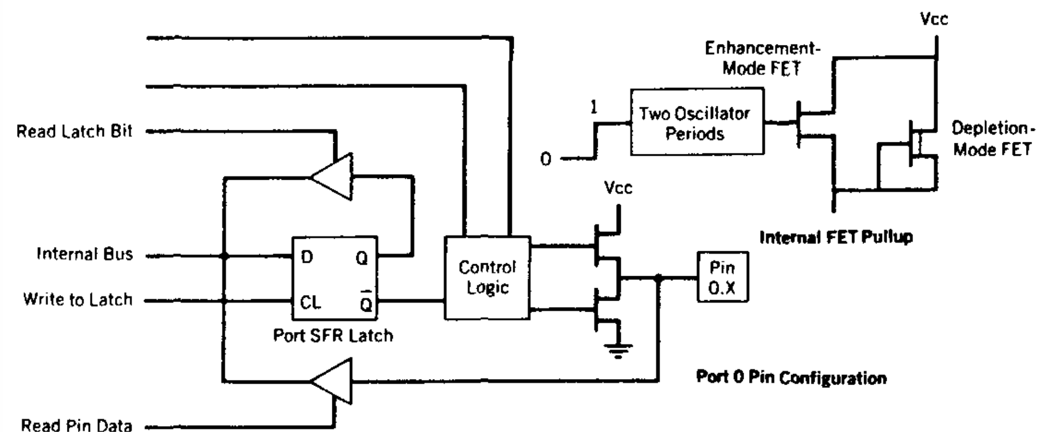
I/O Port Circuits (Port-0)

- Each port has D type o/p latch for each pin
- The SFR for each port is made-up of these eight latches
- Eight latches for port0 are addressed at location 80H
- The data on latches does not have to be the same as that on the pins
- The signal “write to latch” acts as clock i/p for D flip-flop. The data from the internal bus is data-in in response to a “write to latch” signal from CPU

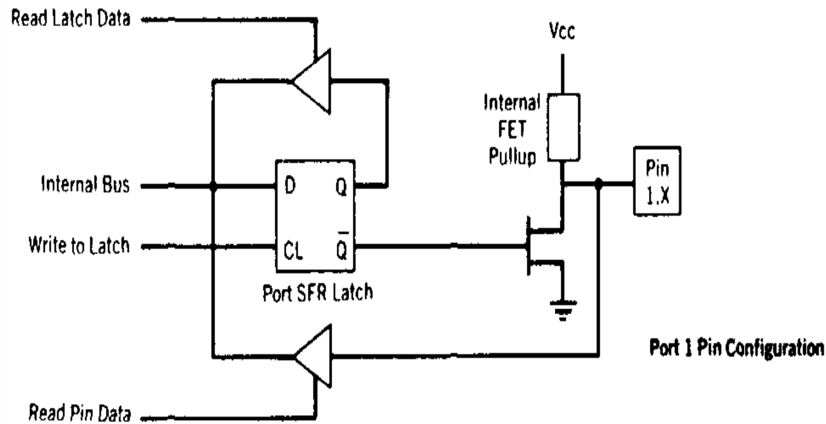


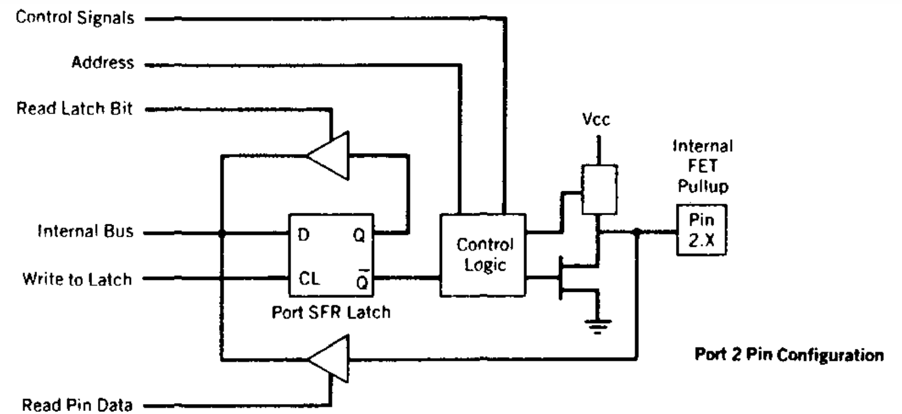
I/O Port Circuits (Port-0)

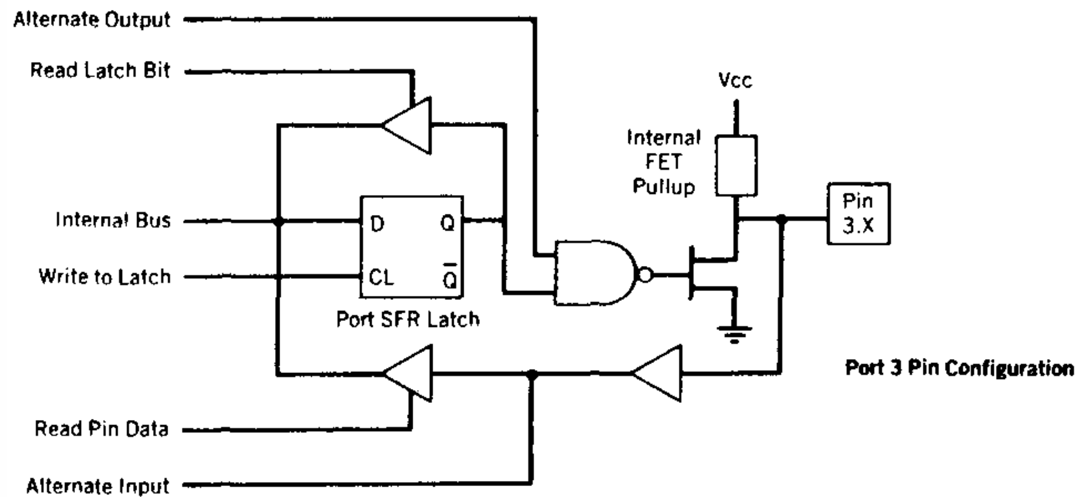
- The two data paths, that read the latch or pin data using two separate buffers.
- Upper buffer is enabled when latch data is read and lower buffer when pin state is read.
- The Qbar or Q o/p after inversion from D FF is connected at the gate i/p of driver FET. The ON and OFF state of the driver FET due to the data available at the o/p of latch decides the status of the o/p pin.
- It is possible to read Q o/p of the latch by activating “read latch” signal from the CPU. The actual port status can be read by activating “read pin” signal.



I/O Port Circuits







I/O Port Circuits

- Port 0 and port 2 drivers are switchable to internal address/data and address bus respectively, by internal “control logic”. The switching is required to access external memory.
- Port3 has multi function pins, each pin of port3 can be programmed to use as i/o or as one of the alternate function. This is achieved by the another control i/p “alternate o/p function”.

I/O Port Circuits

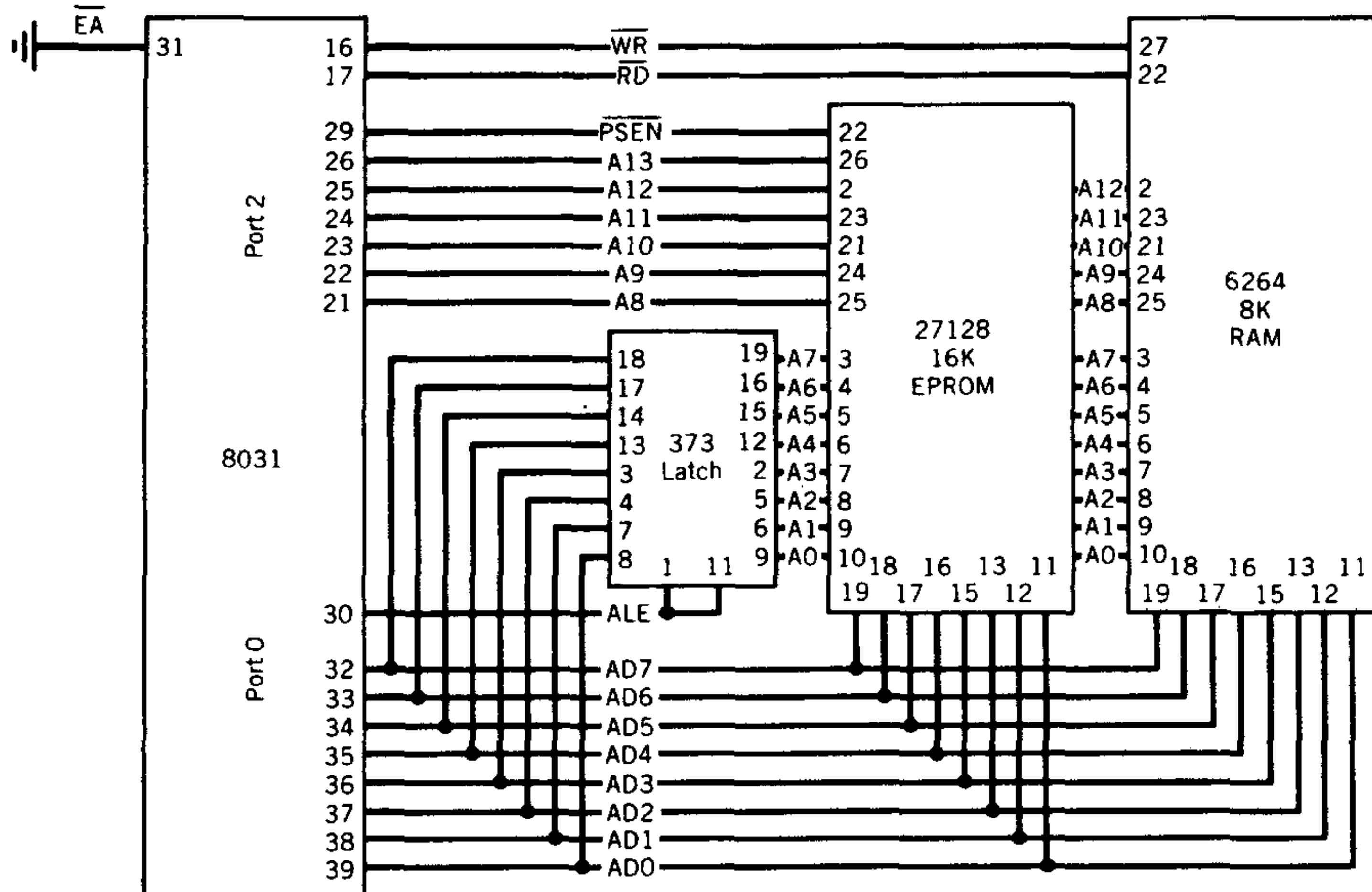
- The port pin can be configured as an i/p by writing ‘1’ in the latch bit of the corresponding pin, it turn OFF the o/p driver FET. Then for ports 1,2,3 the pin is pulled high by the internal pull up, but can be pulled down/low by an external source.

- There is no pull up for port0, therefore its o/p pin floats when ‘1’ is written in the latch bit, so port 0 is said to be “time bi-directional”, because when configured as an i/p it floats.

- On the other hand the o/p of ports 1,2,3 are pulled high with pull registers, when configured as an i/p, thus they are sometimes called “quasi bi-directional” ports.

External Memory

- The designer is not limited by the amount of internal RAM and ROM available on chip.



External Memory

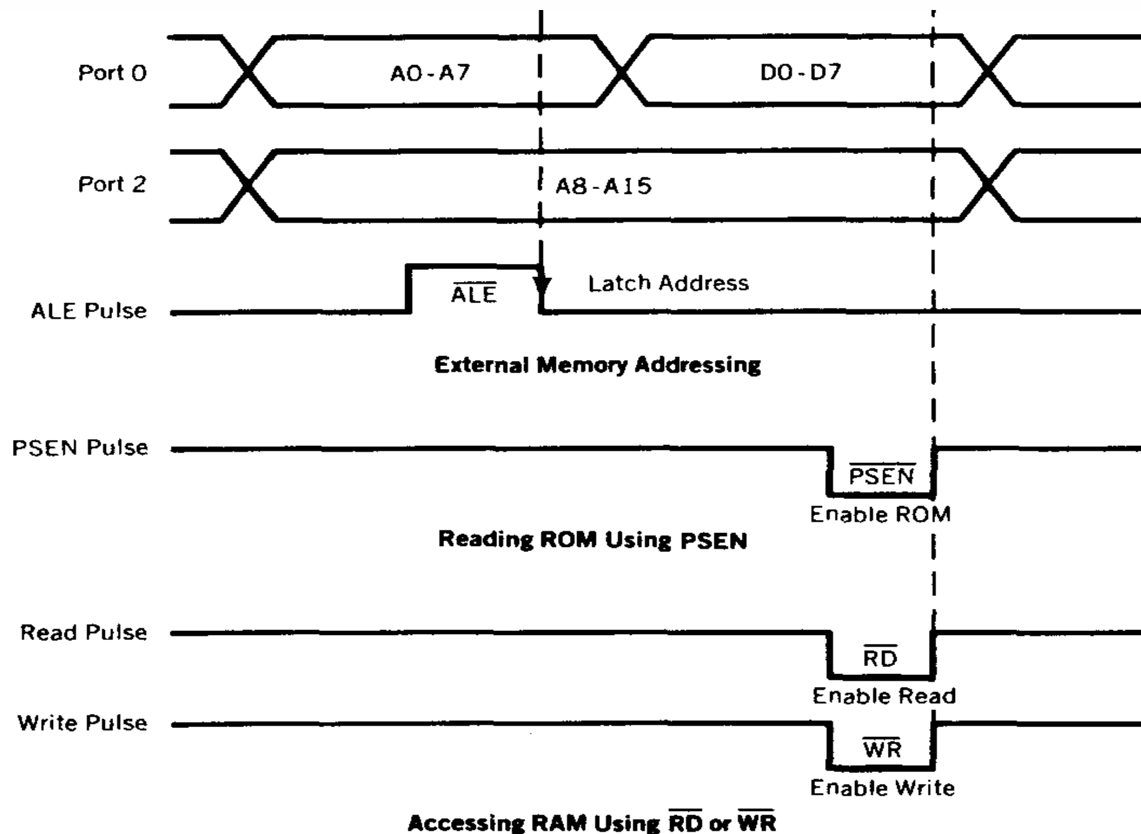
- The designer is not limited by the amount of internal RAM and ROM available on chip.
- Two separate external memory spaces are available by the 16 bit PC and DPTR and by different control pins for enabling external ROM and RAM Chips.
- The reason for adding external memory, particularly program memory is when project is in the prototype stage, the expense in-time and money of having a masked internal ROM made for each program “try” is prohibitive.

External Memory

- 4K on chip EPROM for erased and reprogram.
- To program EPROM 8751 specialized programmers are required.
- If the program size exceeds more than 4K external memory is required.
- If /EA pin is grounded when using 8051, and program code is present in external EPROM of 64K size.
- External RAM, which is accessed by the DPTR may need when 128 bytes of internal storage is not sufficient. External RAM up to 64K size may also be added to 8051.

External Memory

- External ROM is accessed whenever the /EA pin is connected to GND or when the PC contains an address higher than the last address in the internal 4K ROM (0FFFH)



External Memory

- Port 0 is multiplexed, it first provides the lower byte of the 16-bit memory address, then acts as a bidirectional data bus to write or read a byte of memory data.

- Port 2 provides the high byte of the memory address during the entire memory read/write cycle.

- The lower address byte from port 0 must be latched into an external register to save the byte. Address byte save is accomplished by the ALE clock pulse that provides the correct timing, Port 0pins then free to serve as a data bus.

- If the memory access is for a byte of program code in the ROM, the /PSEN pin will go low to enable the ROM to place a byte of program code on the data bus.

External Memory

- If the access is for a RAM byte, the /WR and /RD pins will go low enabling data to flow b/w RAM and data bus.
- ROM may be expanded to 64K by using a 27512 type EPROM.
- The use of external memory consumes many of the port pins, leaving only port 1 and parts of port 3 for general I/O.

8051 Addressing Modes

- The CPU can access data in various ways, which are called addressing modes

1. Immediate
2. Register
3. Direct
4. Register indirect
5. External Direct

Immediate Addressing Mode

- The source operand is a **constant.**
- The immediate data must be preceded by the pound sign, **“#”**
- Can load information into **any registers**, including 16-bit DPTR register
  - DPTR can also be accessed as two 8-bit registers, the high byte DPH and low byte DPL

```
MOV A,#25H        ;load 25H into A
MOV R4,#62        ;load 62 into R4
MOV B,#40H        ;load 40H into B
MOV DPTR,#4521H   ;DPTR=4512H
MOV DPL,#21H      ;This is the same
MOV DPH,#45H      ;as above

;illegal!! Value > 65535 (FFFFH)
MOV DPTR,#68975
```

Register Addressing Mode

- Use registers to hold the data to be manipulated.

```
MOV A,R0      ;copy contents of R0 into A
MOV R2,A      ;copy contents of A into R2
ADD A,R5      ;add contents of R5 to A
ADD A,R7      ;add contents of R7 to A
MOV R6,A      ;save accumulator in R6
```

- The source and destination registers must match in size.
  MOV DPTR,A will give an error

```
MOV DPTR,#25F5H
MOV R7,DPL
MOV R6,DPH
```

- The movement of data between Rn registers is not allowed
  MOV R4,R7 is invalid

Direct Addressing Mode

- It is most often used the direct addressing mode to access RAM locations 30 – 7FH.
- The entire 128 bytes of RAM can be accessed.
- Contrast this with immediate addressing mode, there is no "#" sign in the operand.

```
MOV R0,40H   ;save content of 40H in R0
MOV 56H,A    ;save content of A in 56H
```

SFR Registers & their Addresses

```
MOV   0E0H,#55H    ;is the same as
MOV   A,#55H       ;which means load 55H into A (A=55H)

MOV   0F0H,#25H    ;is the same as
MOV    B,#25H      ;which means load 25H into B (B=25H)

MOV   0E0H,R2      ;is the same as
MOV   A,R2         ;which means copy R2 into A

MOV   0F0H,R0      ;is the same as
MOV   B,R0         ;which means copy R0 into B
```

Stack and Direct Addressing Mode

Show the code to push R5 and A onto the stack and then pop them back them into R2 and B, where B = A and R2 = R5

Solution:

```
PUSH 05         ;push R5 onto stack
PUSH 0E0H       ;push register A onto stack
POP  0F0H       ;pop top of stack into B
                ;now register B = register A
POP  02         ;pop top of stack into R2
                ;now R2=R6
```

Register Indirect Addressing Mode

- A **register** is used as a pointer to the data.
- Only register **R0** and **R1** are used for this purpose.
- **R2 – R7** cannot be used to hold the address of an operand located in RAM.
- When **R0** and **R1** hold the addresses of RAM locations, they must be preceded by the **"@"** sign.

```
MOV A,@R0     ;move contents of RAM whose
              ;address is held by R0 into A
MOV @R1,B     ;move contents of B into RAM
              ;whose address is held by R1
```

Register Indirect Addressing Mode

- Write a program to copy the value 55H into RAM memory locations 40H to 41H using **(a) direct addressing mode**, **(b) register indirect addressing mode without a loop**, and **(c) with a loop.**

Solution:

```
(a)
      MOV  A,#55H   ;load A with value 55H
      MOV  40H,A    ;copy A to RAM location 40H
      MOV  41H.A    ;copy A to RAM location 41H
(b)
      MOV  A,#55H   ;load A with value 55H
      MOV  R0,#40H  ;load the pointer. R0=40H
      MOV  @R0,A    ;copy A to RAM R0 points to
      INC  R0       ;increment pointer. Now R0=41h
      MOV  @R0,A    ;copy A to RAM R0 points to
(c)
         MOV  A,#55H      ;A=55H
         MOV  R0,#40H     ;load pointer.R0=40H,
         MOV  R2,#02      ;load counter, R2=3
AGAIN:   MOV  @R0,A       ;copy 55 to RAM R0 points to
         INC  R0          ;increment R0 pointer
         DJNZ R2,AGAIN    ;loop until counter = zero
```

Register Indirect Addressing Mode

- The advantage is that it makes accessing data dynamic rather than static as in **direct addressing mode**.
- **Looping is not possible in direct addressing mode.**
- Write a program to clear 16 RAM locations starting at RAM address 60H.

```
        CLR A           ;A=0
        MOV R1,#60H     ;load pointer. R1=60H
        MOV R7,#16      ;load counter, R7=16
AGAIN:  MOV @R1,A       ;clear RAM R1 points to
        INC R1          ;increment R1 pointer
        DJNZ R7,AGAIN   ;loop until counter=zero
```

External Direct

- External Memory is accessed.

- There are only two commands that use External Direct addressing mode:
  - **MOVX A, @DPTR**
    MOVX @DPTR, A

- DPTR must first be loaded with the address of external memory.

8051 Instruction Set

- **8051 instructions have 8-bit opcode**
- **There are 256 possible instructions of which 255 are**
- **implemented**

MOV Instruction

- **MOV destination, source** ; copy source to destination.

- MOV A,#55H ;load value 55H into reg. A
  MOV R0,A ;copy contents of A into R0
  ;(now A=R0=55H)
  MOV R1,A ;copy contents of A into R1
  ;(now A=R0=R1=55H)
  MOV R2,A ;copy contents of A into R2
  ;(now A=R0=R1=R2=55H)
  MOV R3,#95H ;load value 95H into R3
  ;(now R3=95H)
  MOV A,R3 ;copy contents of R3 into A
  ;now A=R3=95H)

INSTITUTE OF AERONAUTICAL ENGINEERING

ADD Instruction

- **ADD A, source** ;ADD the source operand to the accumulator

- MOV A, #25H ;load 25H into A

  MOV R2,#34H ;load 34H into R2

  ADD A,R2 ;add R2 to accumulator

  ;(A = A + R2)

INSTITUTE OF AERONAUTICAL ENGINEERING

Structure of Assembly Language

```
        ORG   0H          ;start (origin) at location 0
        MOV  R5,#25H      ;load 25H into R5
        MOV  R7,#34H      ;load 34H into R7
        MOV  A,#0         ;load 0 into A
        ADD  A,R5         ;add contents of R5 to A
                          ;now A = A + R5
        ADD  A,R7         ;add contents of R7 to A
                          ;now A = A + R7
        ADD  A,#12H       ;add to A value 12H
                          ;now A = A + 12H
HERE: SJMP HERE           ;stay in this loop
        END               ;end of asm source file
```

Data Types & Directives

```
        ORG   500H
DATA1:  DB    28
DATA2:  DB    00110101B
DATA3:  DB    39H
        ORG   510H
DATA4:  DB    "2591"
        ORG   518H
DATA6:  DB    "My name is Joe"
```

Multiplication of Unsigned Numbers

MUL AB ; A × B, place 16-bit result in B and A

```
MOV   A,#25H        ;load 25H to reg. A
MOV   B,#65H        ;load 65H in reg. B
MUL   AB            ;25H * 65H = E99 where B = 0EH and A = 99H
```

Table 6-1:Unsigned Multiplication Summary (MUL AB)

Multiplication	Operand 1	Operand 2	Result
byte byte	A	B	A=low byte, B=high byte

Division of Unsigned Numbers

DIV AB ; divide A by B

- MOV A,#95H ;load 95 into A
- MOV B,#10H ;load 10 into B
- DIV AB ;now A = 09 (quotient) and B = 05 (remainder)

Table 6-2:Unsigned Division Summary (DIV AB)

Division	Numerator	Denominator	Quotient	Remainder
byte / byte	A	B	A	B

Unconditional Jump Instructions

- All conditional jumps are short jumps
  - Target address within -128 to +127 of PC
- **LJMP** (long jump): 3-byte instruction
  - 2-byte target address: 0000 to FFFFH
  - Original 8051 has only 4KB on-chip ROM
- **SJMP** (short jump): 2-byte instruction
  - 1-byte relative address: -128 to +127

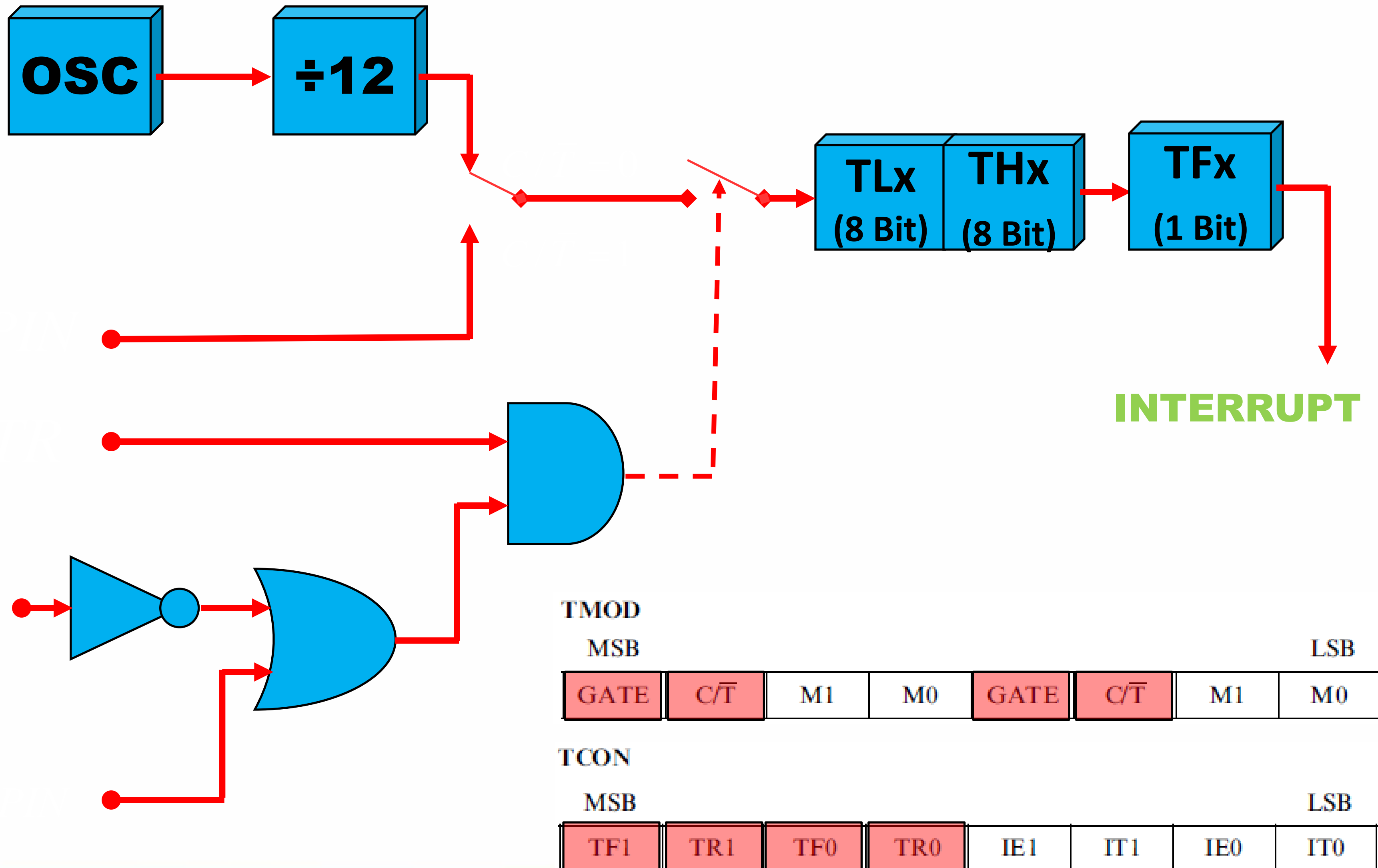
Call Instructions

- LCALL (long call): 3-byte instruction
  - 2-byte address
  - Target address within 64K-byte range
- ACALL (absolute call): 2-byte instruction
  - 11-bit address
  - Target address within 2K-byte range

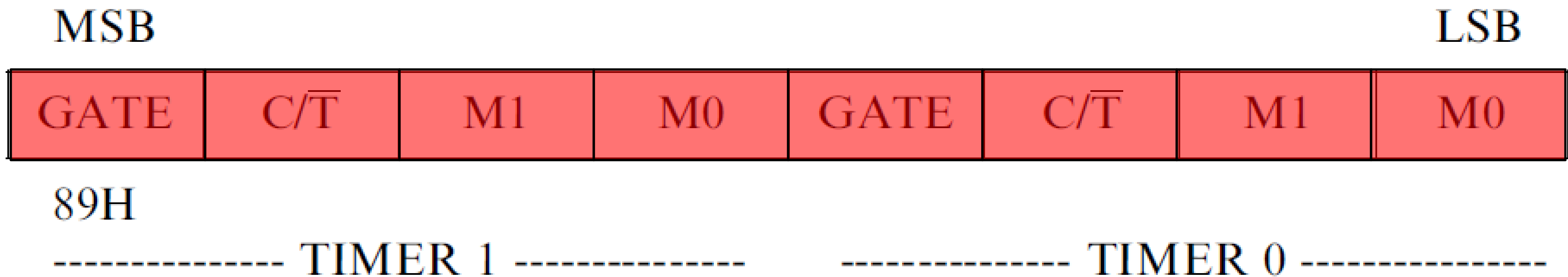
8051 Peripheral Overview

1. Timers
2. Serial Port
3. Interrupts

8051 Timer/Counter



TMOD Register



GATE:

When set, timer/counter x is enabled, **if** INTx pin is high and TRx is set.

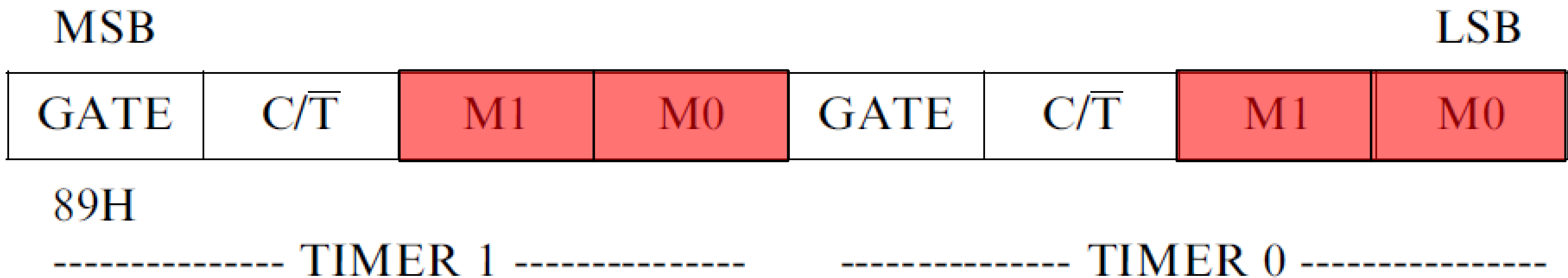
When cleared, timer/counter x is enabled, **if** TRx bit set.

C/T*:

When set, counter operation (input from Tx input pin).

When cleared, timer operation (input from internal clock).

TMOD Register



The TMOD byte is not bit addressable.

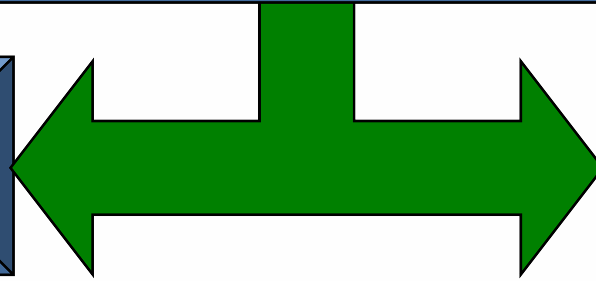
M1	M0	Operation
0	0	8048 8-bit timer TLx serves as 5-bit prescaler
0	1	16-bit timer/counter. THx and TLx are cascaded. No prescaler
1	0	8-bit autoreload timer/counter. THx contents loaded into TLx when it overflows
1	1	TL0 is 8-bit counter controlled by timer 0 control bits. TH0 is 8-bit timer controlled by timer 1 control bits
1	1	Timer 1 off

TCON Register

MSB							LSB
TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
8FH	8EH	8DH	8CH	8BH	8AH	89H	88H

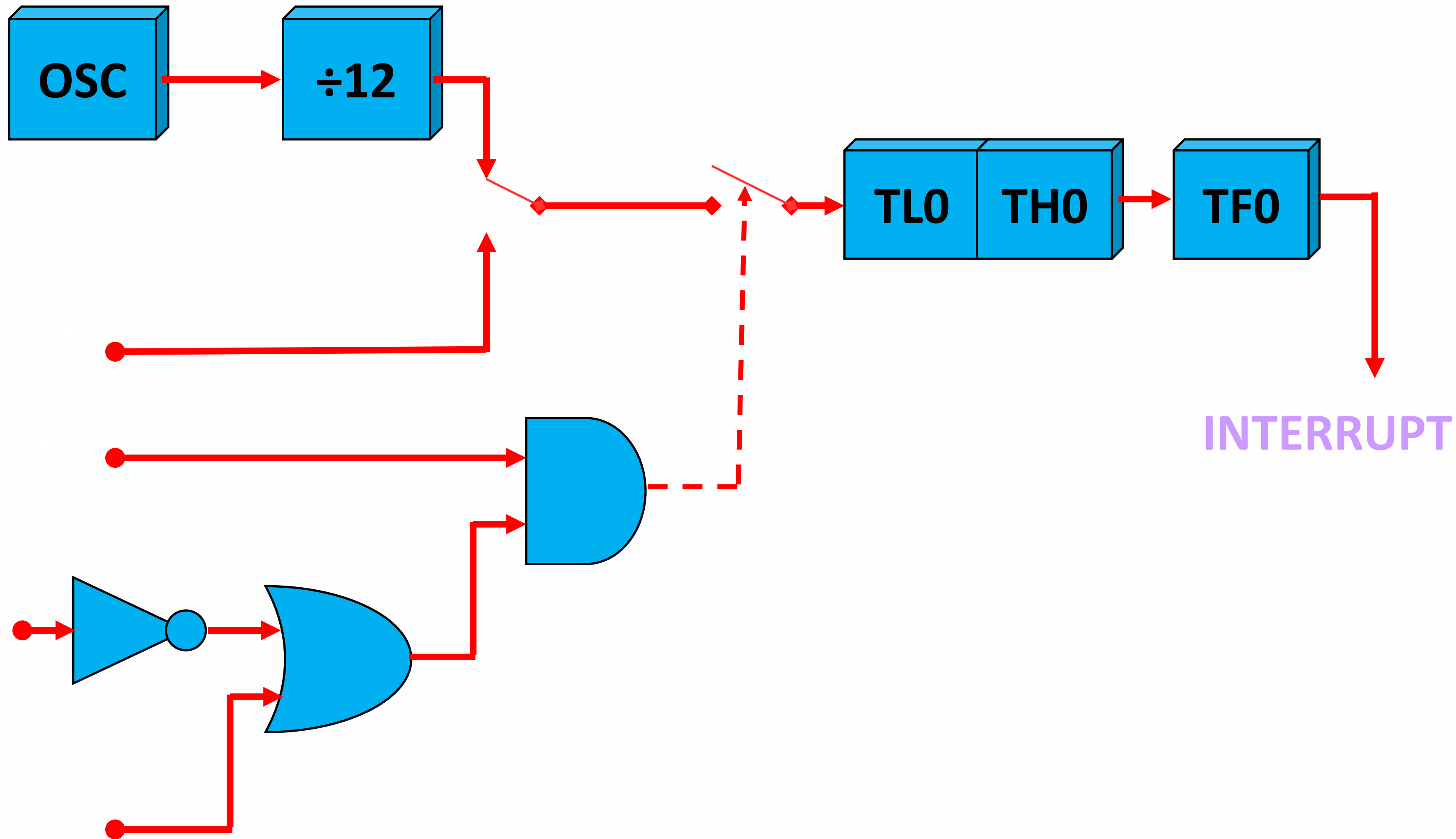
Bit	Function
TF1/0	Timer 1/0 overflow flag. Set by hardware on timer/counter overflow. Cleared when CPU vectors to interrupt routine
TR1/0	Timer 1/0 run control bit. Set/cleared by software to turn timer/counter on/off
IE1/0	Interrupt 1/0 edge flag. Set by hardware when external interrupt edge detected. Cleared when interrupt processed
IT1/0	Interrupt 1/0 control bit. Set/cleared by software to specify falling edge/ low level triggered external interrupts

8051 TIMERS



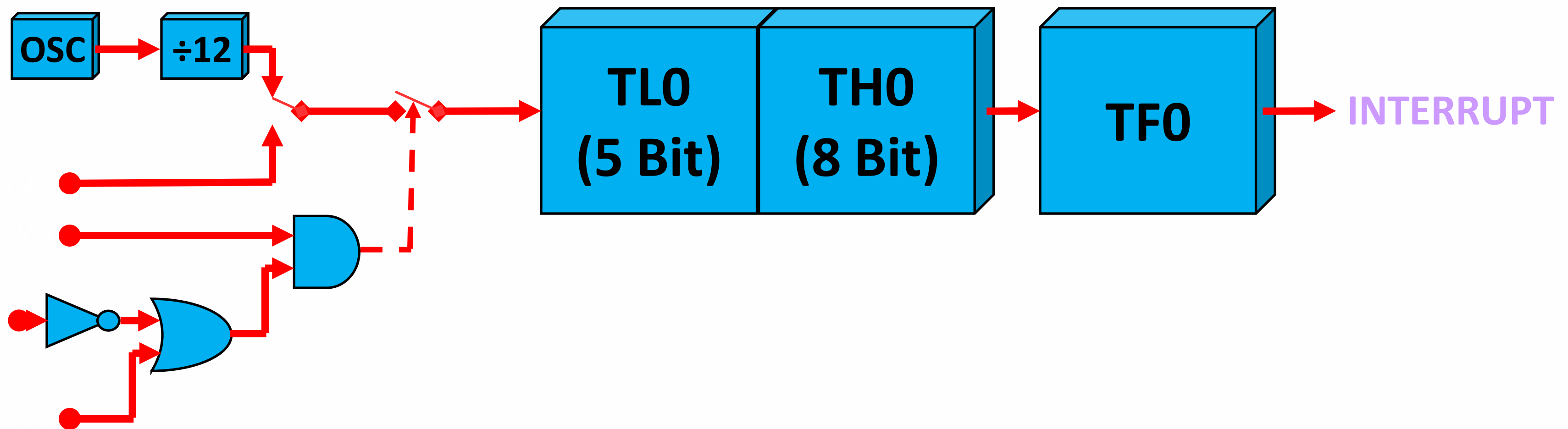
Timer 0	Timer 1
Mode 0	Mode 0
Mode 1	Mode 1
Mode 2	Mode 2
Mode 3	

TIMER 0



TIMER 0 – Mode 0

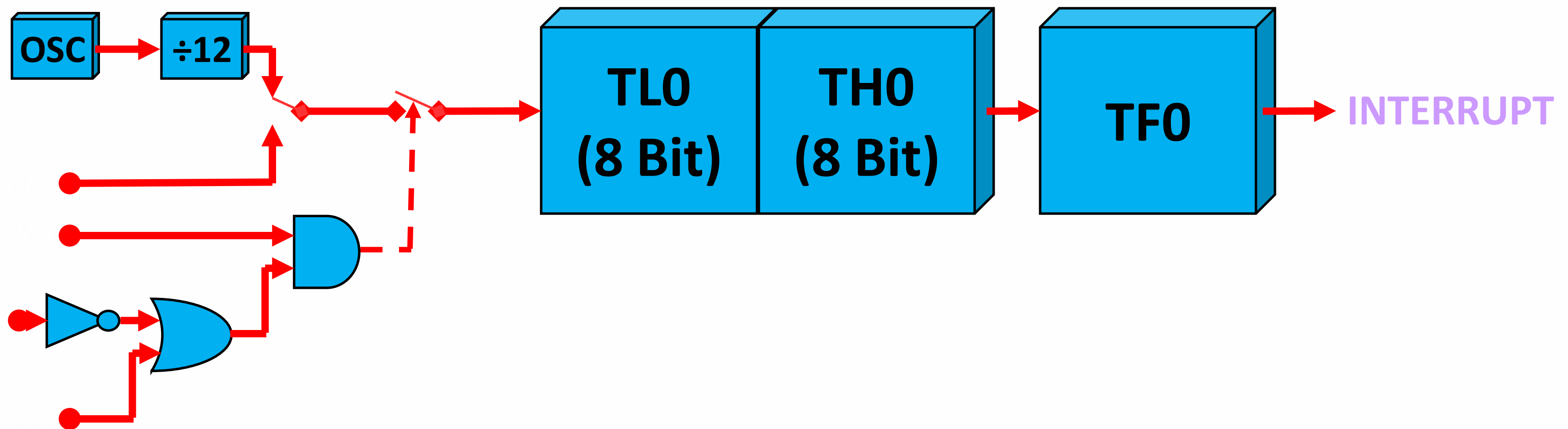
13 Bit Timer / Counter



Maximum Count = 1FFFh (0001111111111111)

TIMER 0 – Mode 1

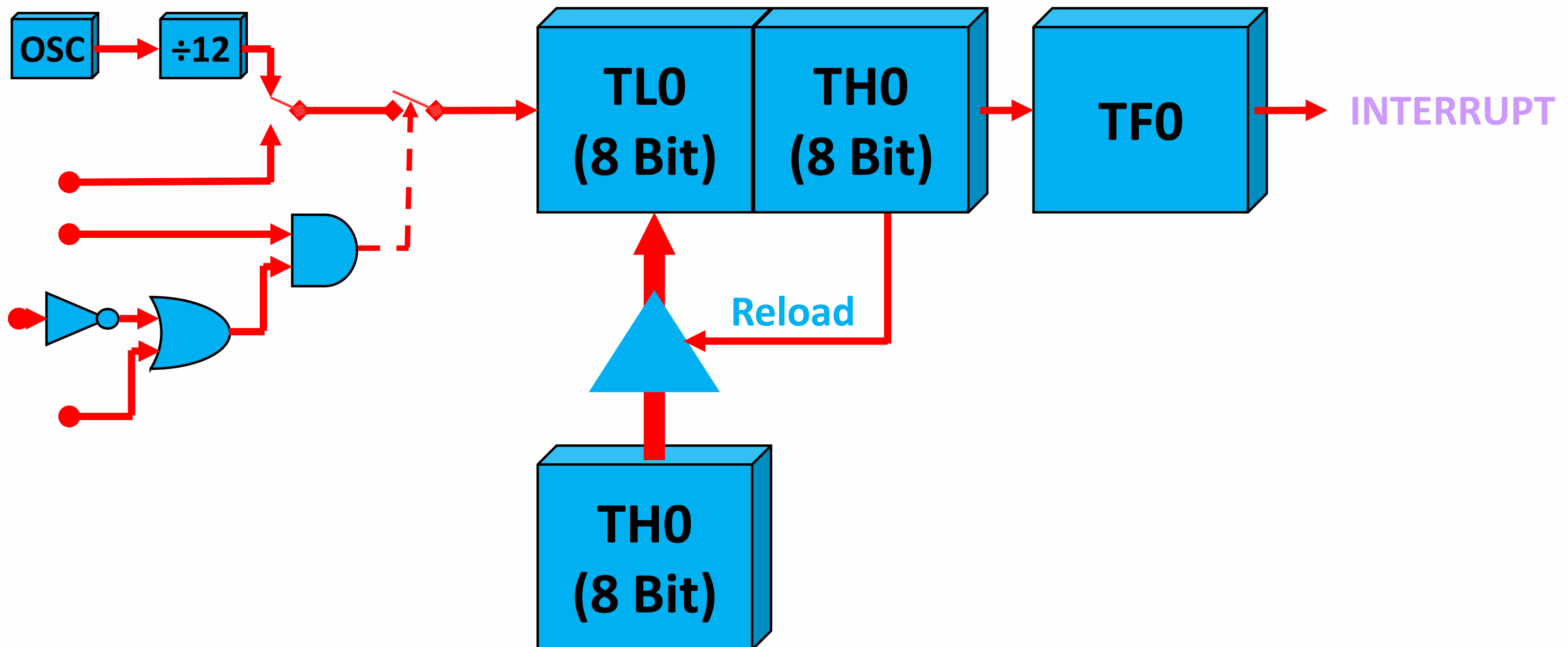
16 Bit Timer / Counter



Maximum Count = FFFFh (1111111111111111)

TIMER 0 – Mode 2

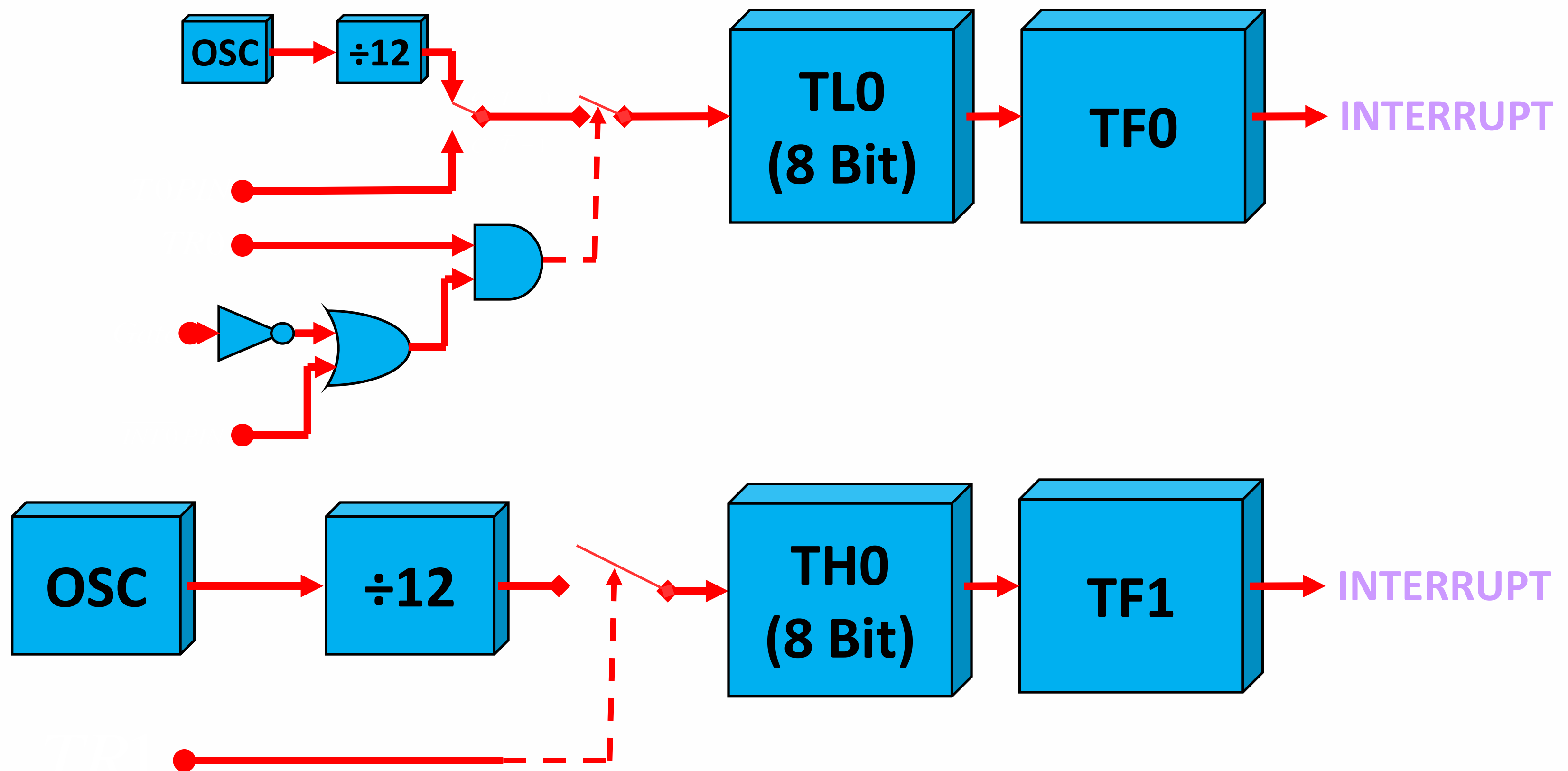
8 Bit Timer / Counter with AUTORELOAD



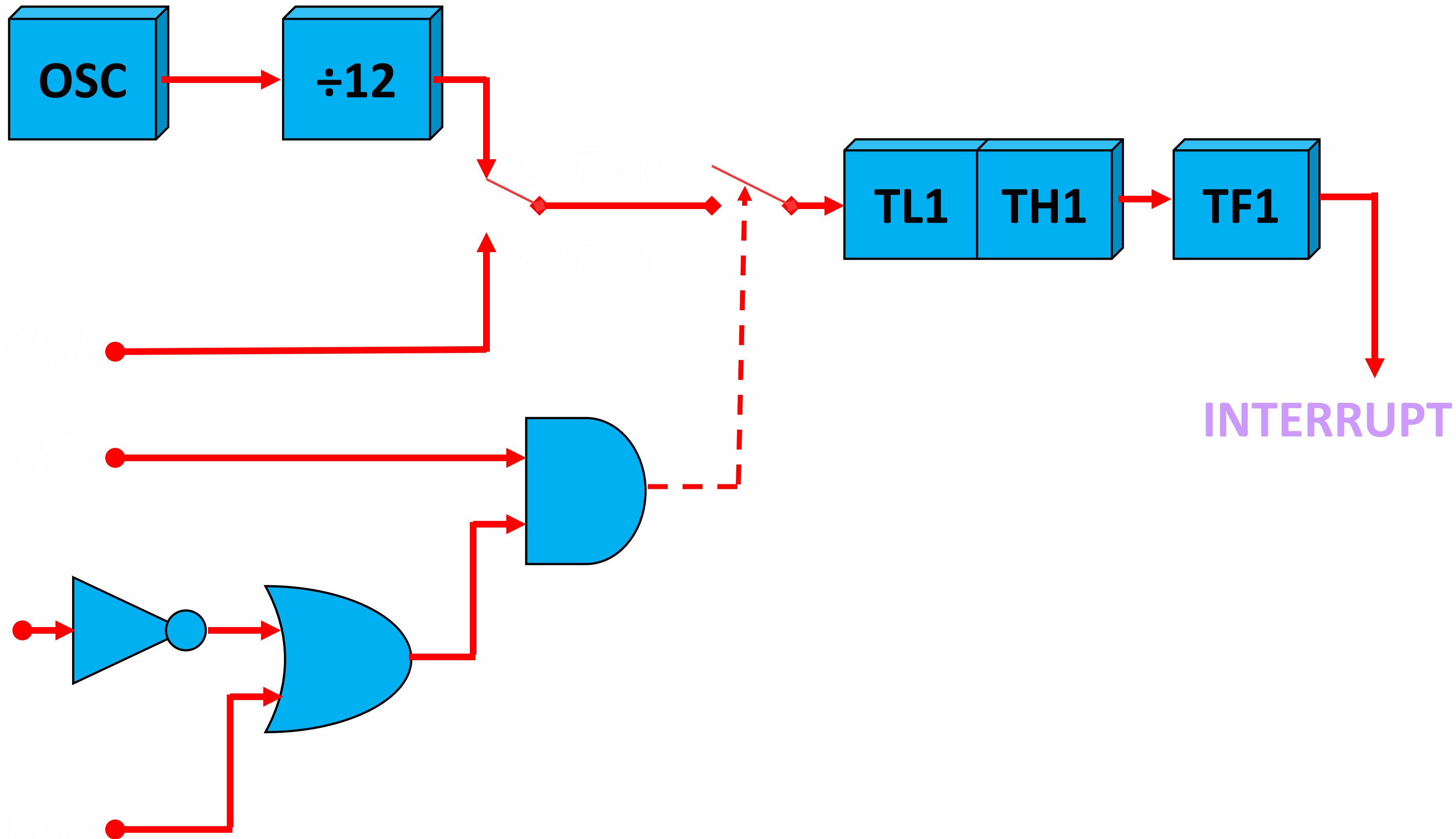
Maximum Count = FFh (11111111)

TIMER 0 – Mode 3

Two - 8 Bit Timer / Counter

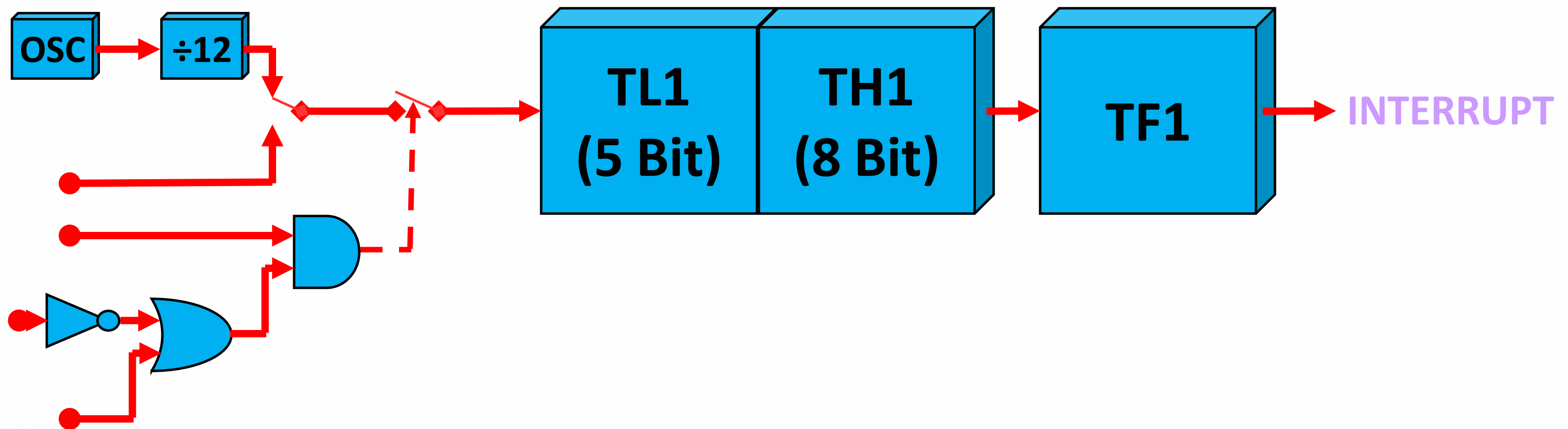


TIMER 1



TIMER 1 – Mode 0

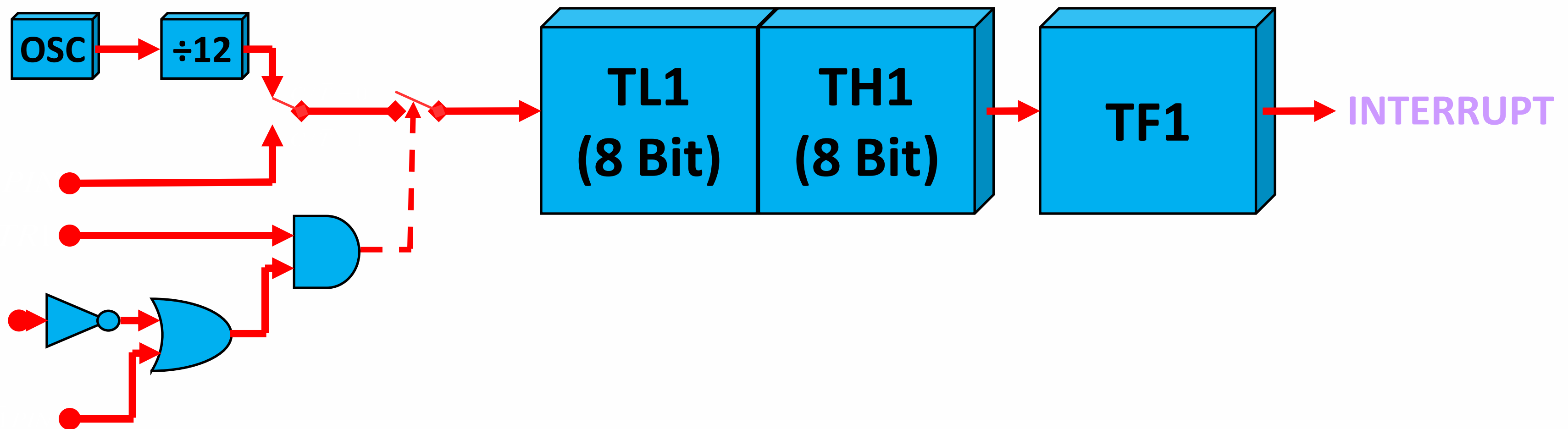
13 Bit Timer / Counter



Maximum Count = 1FFFh (0001111111111111)

TIMER 1 – Mode 1

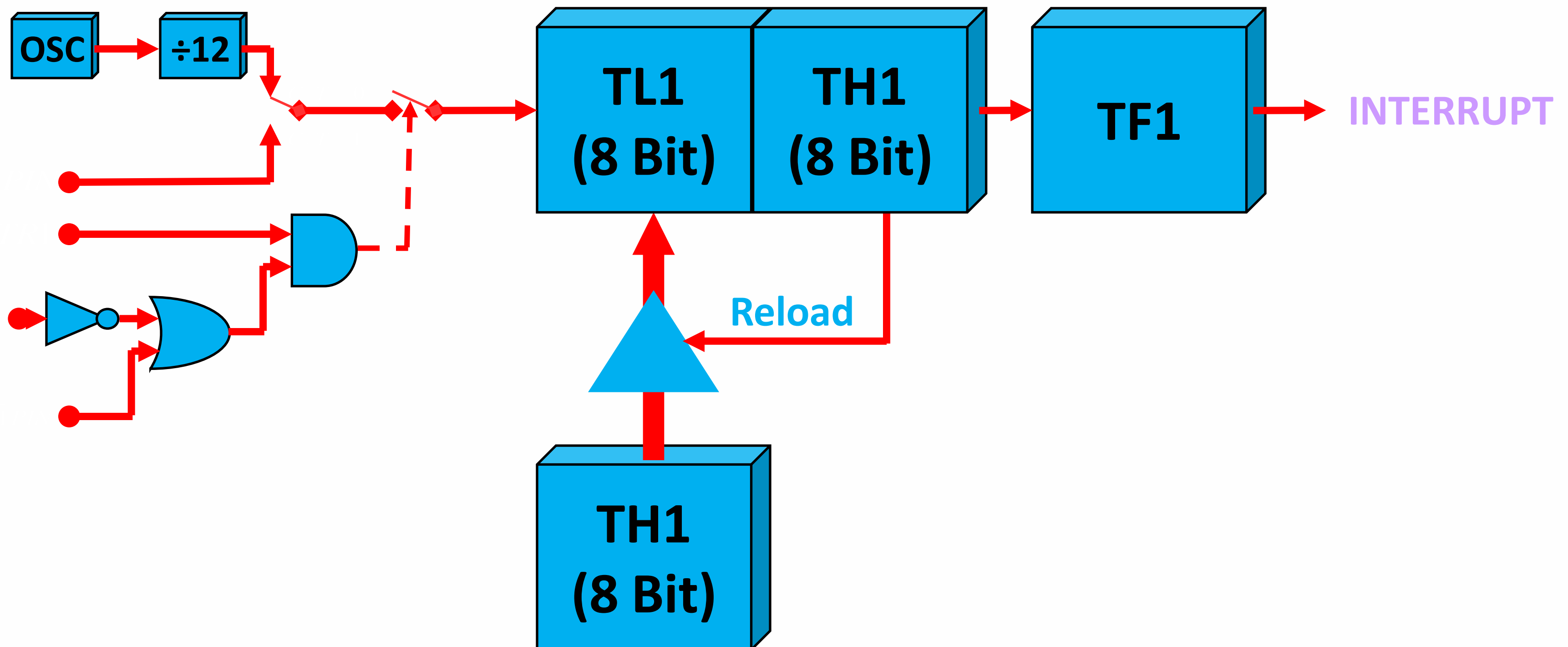
16 Bit Timer / Counter



Maximum Count = FFFFh (1111111111111111)

TIMER 1 – Mode 2

8 Bit Timer / Counter with AUTORELOAD



Maximum Count = FFh (11111111)

Programming Timers

- **Example:** Indicate which mode and which timer are selected for each of the following.

  (a) MOV TMOD, #01H (b) MOV TMOD, #20H (c) MOV TMOD, #12H

- **Solution:** We convert the value from hex to binary.

  (a) TMOD = 00000001, mode 1 of timer 0 is selected.

  (b) TMOD = 00100000, mode 2 of timer 1 is selected.

  (c) TMOD = 00010010, mode 2 of timer 0, and mode 1 of timer 1

  are selected.

Programming Timers

- Find the timer's clock frequency and its period for various 8051-based system, with the crystal frequency 11.0592 MHz when C/T bit of TMOD is 0.

- **Solution:**

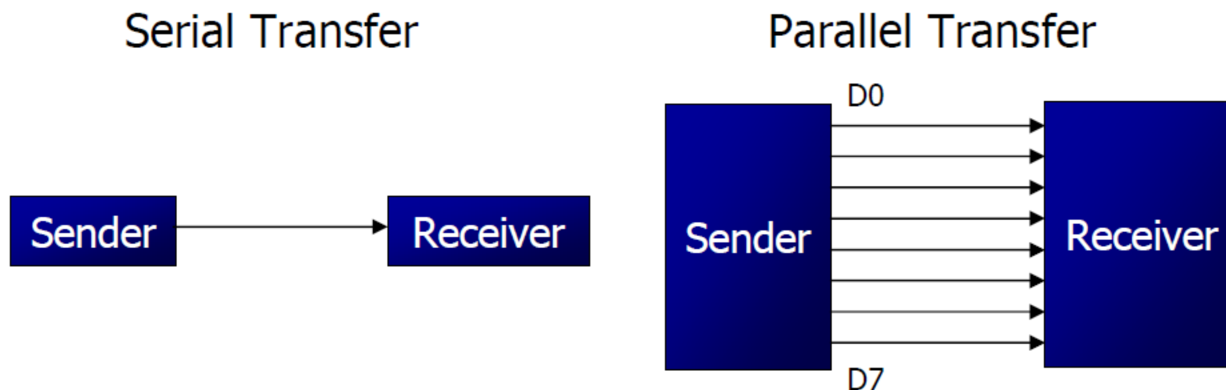
1/12 × 11.0529 MHz = 921.6 MHz;

T = 1/921.6 kHz = 1.085 us

8051 Serial Port

Basics of Serial Communication

- Computers transfer data in **two** ways:
  - **Parallel:** Often 8 or more lines (wire conductors) are used to transfer data to a device that is only a few feet away.
  - **Serial:** To transfer to a device located many meters away, the serial method is used. The data is sent one bit at a time.



Basics of Serial Communication

- Serial data communication uses **two** methods
  - **Synchronous** method transfers a block of data at a time
  - **Asynchronous** method transfers a single byte at a time

- There are **special IC's** made by many manufacturers for serial communications.
  - **UART** (universal asynchronous Receiver transmitter)
  - **USART** (universal synchronous-asynchronous Receiver-transmitter)

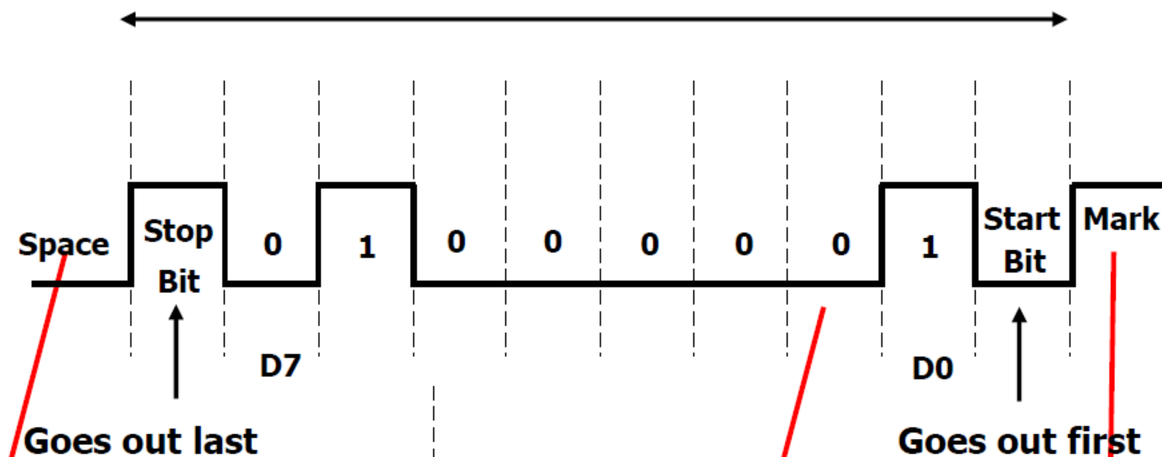
Asynchronous – Start & Stop Bit

- Asynchronous serial data communication is widely used for **character-oriented** transmissions
  - Each character is placed in between **start and stop bits**, this is called **framing.**
  - **Block-oriented** data transfers use the synchronous method.

- **The start bit is always one bit, but the stop bit can be one or two bits**

- **The start bit is always a 0 (low) and the stop bit(s) is 1 (high)**

Asynchronous – Start & Stop Bit



The 0 (low) is referred to as *space*

The transmission begins with a start bit followed by D0, the LSB, then the rest of the bits until MSB (D7), and finally, the one stop bit indicating the end of the character

When there is no transfer, the signal is 1 (high), which is referred to as *mark*

Data Transfer Rate

- The rate of data transfer in serial data communication is stated in **bps** (**bits per second**).

- Another widely used terminology for bps is **baud rate.**
  - It is modem terminology and is defined as **the number of signal changes per second**
  - In modems, there are occasions when a single change of signal transfers several bits of data

- As far as the **conductor wire** is concerned, **the baud rate and bps are the same.**

8051 Serial Port

- Synchronous and Asynchronous
- SCON Register is used to Control
- Data Transfer through TXd & RXd pins
- Some time - Clock through TXd Pin
- Four Modes of Operation:

Mode 0	**:Synchronous Serial Communication**
Mode 1	**:8-Bit UART with Timer Data Rate**
Mode 2	**:9-Bit UART with Set Data Rate**
Mode 3	**:9-Bit UART with Timer Data Rate**

Registers related to Serial Communication

1. SBUF Register
2. SCON Register
3. PCON Register

SBUF Register

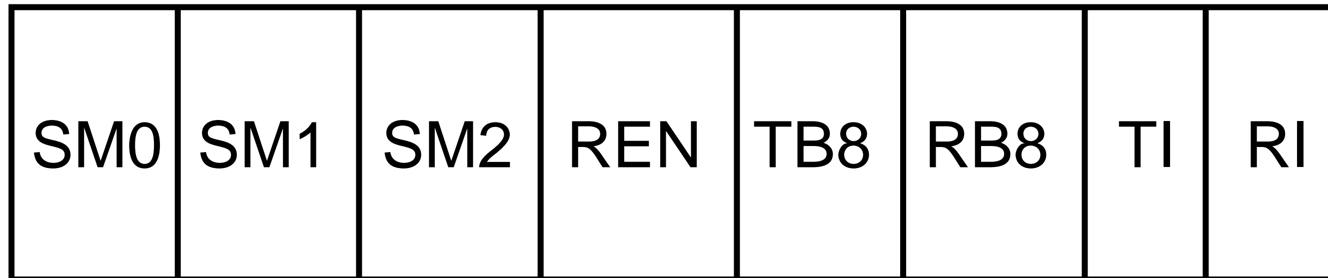
- **SBUF** is an **8-bit register** used solely for serial communication.
- For a byte data to be transferred via the **TxD line**, it must be placed in the **SBUF register**.
- The moment a byte is written into SBUF, it is framed with the start and stop bits and transferred serially via the TxD line.
- SBUF holds the byte of data when it is received by 8051 **RxD** line.
- When the bits are received serially via RxD, the **8051 deframes** it by eliminating the stop and start bits, making a byte out of the data received, and then placing it in SBUF.

SBUF Register

- **Sample Program:**

```
MOV SBUF,#'D'    ;load SBUF=44h, ASCII for 'D'
MOV SBUF,A       ;copy accumulator into SBUF
MOV A,SBUF       ;copy SBUF into accumulator
```

SCON Register



Serial Mode	Explanation
0	8-bit Shift Register
1	8-bit UART
2	9-bit UART
3	9-bit UART

Set when a Character received

Set to Enable Serial Data reception

Set when Stop bit Txed

Enable Multiprocessor Communication Mode

9th Data Bit Sent in Mode 2,3

9th Data Bit Received in Mode 2,3

8051 Serial Port – Mode 0

The Serial Port in Mode-0 has the following features:

1. Serial data **enters and exits through RXD**

2. **TXD** outputs the **clock**

3. 8 bits are transmitted / received

4. The baud rate is fixed at (1/12) of the oscillator frequency

8051 Serial Port – Mode 1

The Serial Port in Mode-1 has the following features:

1. Serial data **enters through RXD**
2. Serial data **exits through TXD**
3. On receive, the stop bit goes into RB8 in SCON
4. **10 bits** are transmitted / received
   1. *Start bit (0)*
   2. *Data bits (8)*
   3. *Stop Bit (1)*
5. Baud rate is determined by the Timer 1 over flow rate.

8051 Serial Port – Mode 2

The Serial Port in Mode-2 has the following features:

1. Serial data **enters through RXD**
2. Serial data **exits through TXD**
3. 9th data bit (**TB8**) can be assign value 0 or 1
4. On receive, the 9th data bit goes into **RB8** in SCON
5. **11 bits** are transmitted / received
    1. Start bit (0)
    2. Data bits (9)
    3. Stop Bit (1)
6. **Baud rate** is programmable

8051 Serial Port – Mode 3

The Serial Port in Mode-3 has the following features:

1. Serial data **enters through RXD**
2. Serial data **exits through TXD**
3. 9th data bit (**TB8**) can be assign value 0 or 1
4. On receive, the 9th data bit goes into **RB8** in SCON
5. **11 bits** are transmitted / received
    1. Start bit (0)
    2. Data bits (9)
    3. Stop Bit (1)
6. **Baud rate** is determined by Timer 1 overflow rate.

Programming Serial Data Transmission

1. **TMOD register** is loaded with the value **20H**, indicating the use of timer 1 in mode 2 (8-bit auto-reload) **to set baud rate**.
2. The **TH1** is loaded with one of the values to set baud rate for serial data transfer.
3. The **SCON register** is loaded with the value **50H**, indicating serial mode 1, where an 8- bit data is framed with start and stop bits.
4. **TR1** is set to 1 to start timer 1
5. **TI** is cleared by **CLR TI** instruction
6. The character byte to be transferred serially is written into **SBUF register.**
7. The **TI flag bit** is monitored with the use of instruction **JNB TI, xx** to see if the character has been transferred completely.
8. To transfer the next byte, **go to step 5**

Programming Serial Data Reception

1. **TMOD register** is loaded with the value **20H**, indicating the use of timer 1 in mode 2 (8-bit auto-reload) **to set baud rate.**
2. **TH1** is loaded to set baud rate
3. The **SCON register** is loaded with the value **50H**, indicating serial mode 1, where an 8- bit data is framed with start and stop bits.
4. **TR1** is set to 1 to start timer 1
5. **RI** is cleared by **CLR RI** instruction
6. The **RI flag bit** is monitored with the use of instruction **JNB RI, xx** to see if an entire character has been received yet
7. **When RI is raised**, **SBUF** has the byte, its contents are moved into a safe place.
8. To receive the next character, **go to step 5.**

Doubling Baud Rate

- There are two ways to increase the baud rate of data transfer
  1. **By using a higher frequency crystal**
  2. **By changing a bit in the PCON register**

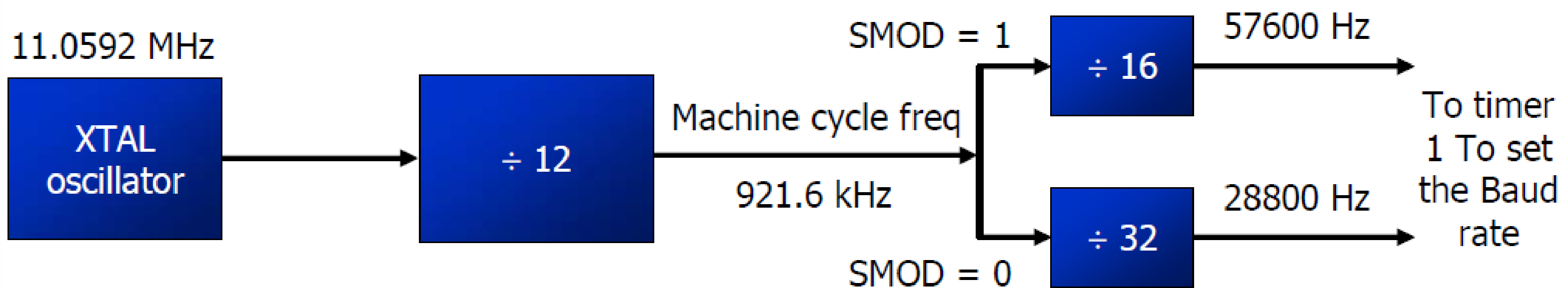
- **PCON register** is an 8-bit register.

SMOD	--	--	--	GF1	GF0	PD	IDL

- When 8051 is powered up, **SMOD** is zero
- **We can set it to high** by software and thereby **double** the baud rate.

Doubling Baud Rate (cont...)

```
MOV   A,PCON       ;place a copy of PCON in ACC
SETB  ACC.7        ;make D7=1
MOV   PCON,A       ;changing any other bits
```



Baud Rate comparison for SMOD=0 and SMOD=1

TH1 (Decimal)	(Hex)	SMOD=0	SMOD=1
-3	FD	9600	19200
-6	FA	4800	9600
-12	F4	2400	4800
-24	E8	1200	2400

8051 Interrupts

INTERRUPTS

- An interrupt is an external or internal event that interrupts the microcontroller to inform it that a device needs its service
- A single microcontroller can serve several devices by two ways:

  1. **Interrupt**
  2. **Polling**

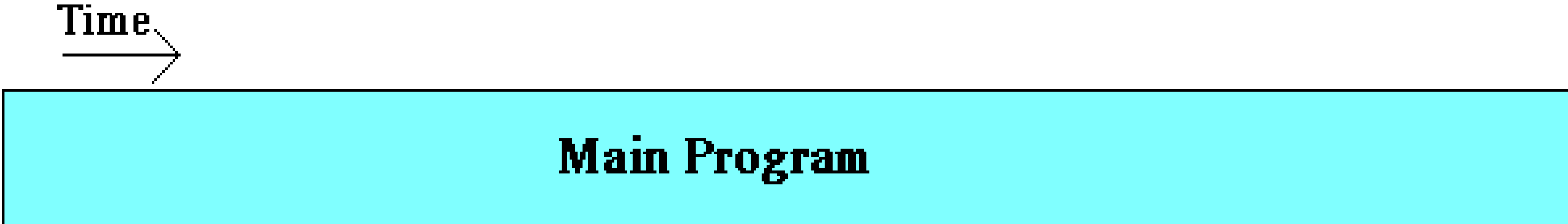
Interrupt Vs Polling

1. **Interrupts**
   - Whenever any device needs its service, the device notifies the microcontroller by sending it an **interrupt signal**.
   - Upon receiving an interrupt signal, the **microcontroller interrupts** whatever it is doing and serves the device.
   - The program which is associated with the interrupt is called the **interrupt service routine (ISR)** or interrupt handler.
2. **Polling**
   - The microcontroller **continuously monitors** the status of a given device.
   - When the **conditions** met, it performs the service.
   - After that, it moves on to monitor the **next device** until every one is serviced.

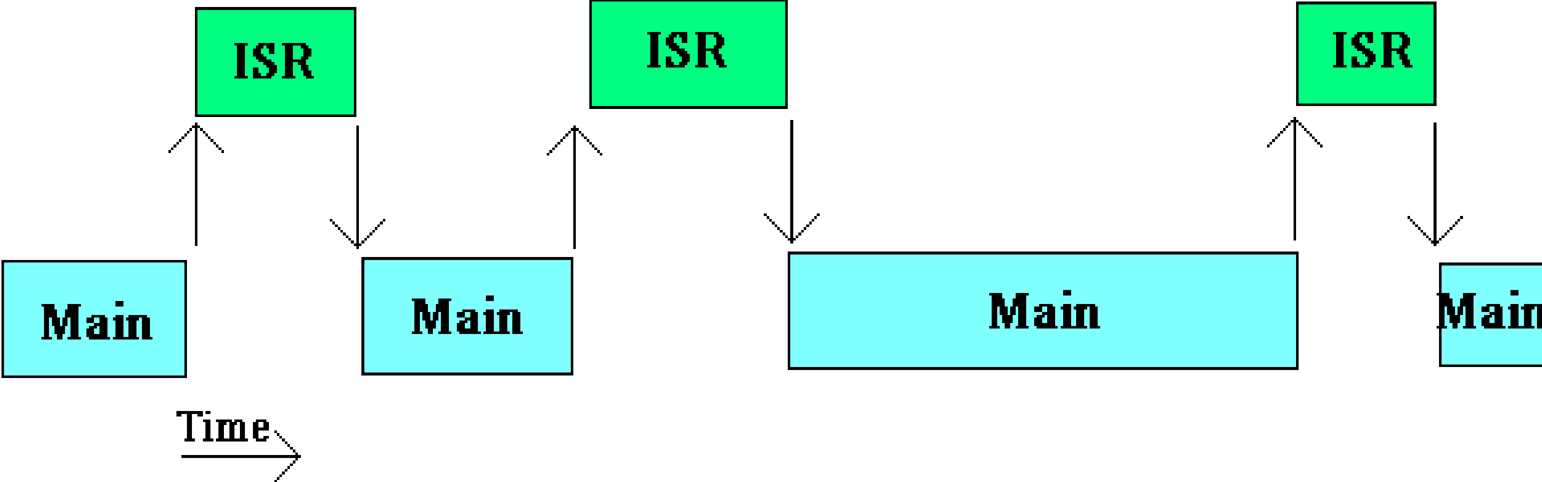
Interrupt Vs Polling

- The **polling method is not efficient**, since it wastes much of the microcontroller's time by polling devices that do not need service.
- The **advantage of interrupts** is that the microcontroller can serve many devices (not all at the same time).
- Each devices can get the attention of the microcontroller based on the **assigned priority**.
- For the polling method, it is **not possible** to assign priority since it checks all devices in a round-robin fashion.
- The microcontroller can also **ignore (mask)** a device request for service in Interrupt.

Program execution without intrrupts :



Program execution with intrrupts :



Six Interrupts in 8051

Six interrupts are allocated as follows:

1. **Reset – power-up reset.**

2. **Two interrupts are set aside for the timers.**
   – one for timer 0 and one for timer 1

3. **Two interrupts are set aside for hardware external interrupts.**
   – P3.2 and P3.3 are for the external hardware interrupts INT0 (or EX1), and INT1 (or EX2)

4. **Serial communication has a single interrupt that belongs to both receive and transfer.**

What events can trigger Interrupts?

- We can configure the 8051 so that any of the following events will cause an interrupt:
  - Timer 0 Overflow.
  - Timer 1 Overflow.
  - Reception/Transmission of Serial Character.
  - External Event 0.
  - External Event 1.
- We can configure the 8051 so that when Timer 0 Overflows or when a character is sent/received, the appropriate interrupt handler routines are called.

8051 Interrupt Vectors

INTERRUPT VECTORS

When the original 8051 and 8031 were introduced, only 5 interrupts were provided.

Interrupt Number	Interrupt Vector Address	Description
0	0003h	EXTERNAL 0
1	000Bh	TIMER/COUNTER 0
2	0013h	EXTERNAL 1
3	001Bh	TIMER/COUNTER 1
4	0023h	SERIAL PORT

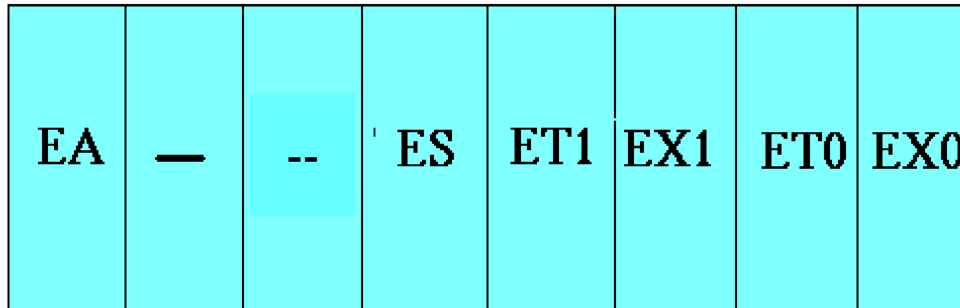
8051 Interrupt related Registers

- **The various registers associated with the use of interrupts are:**
  - **TCON - Edge and Type bits for External Interrupts 0/1**
  - **SCON - RI and TI interrupt flags for RS232**
  - **IE - Enable interrupt sources**
  - **IP - Specify priority of interrupts**

Enabling and Disabling an Interrupt

- Upon **reset**, all interrupts are **disabled (masked),** meaning that none will be responded to by the microcontroller if they are activated.
- The interrupts must be **enabled** by software in order for the microcontroller to respond to them.
- There is a register called **IE (interrupt enable)** that is responsible for enabling (unmasking) and disabling (masking) the interrupts.

Interrupt Enable (IE) Register



- EA : Global enable/disable.
- --- : Reserved for additional interrupt hardware.
- ES : Enable Serial port interrupt.
- ET1 : Enable Timer 1 control bit.
- EX1 : Enable External 1 interrupt.
- ET0 : Enable Timer 0 control bit.
- EX0 : Enable External 0 interrupt.

MOV IE,#08h
or
SETB ET1

Enabling and Disabling an Interrupt

- **Example:** Show the instructions to (a) enable the serial interrupt, timer 0 interrupt, and external hardware interrupt 1 and (b) disable (mask) the timer 0 interrupt, then (c) show how to disable all the interrupts with a single instruction.
- **Solution:**
  - (a) MOV IE,#10010110B ;enable serial, timer 0, EX1
    - Another way to perform the same manipulation is:
      - SETB IE.7 ;EA=1, global enable
      - SETB IE.4 ;enable serial interrupt
      - SETB IE.1 ;enable Timer 0 interrupt
      - SETB IE.2 ;enable EX1
  - (b) CLR IE.1 ;mask (disable) timer 0 interrupt only
  - (c) CLR IE.7 ;disable all interrupts

Interrupt Priority

- When the 8051 is powered up, the priorities are assigned according to the following.

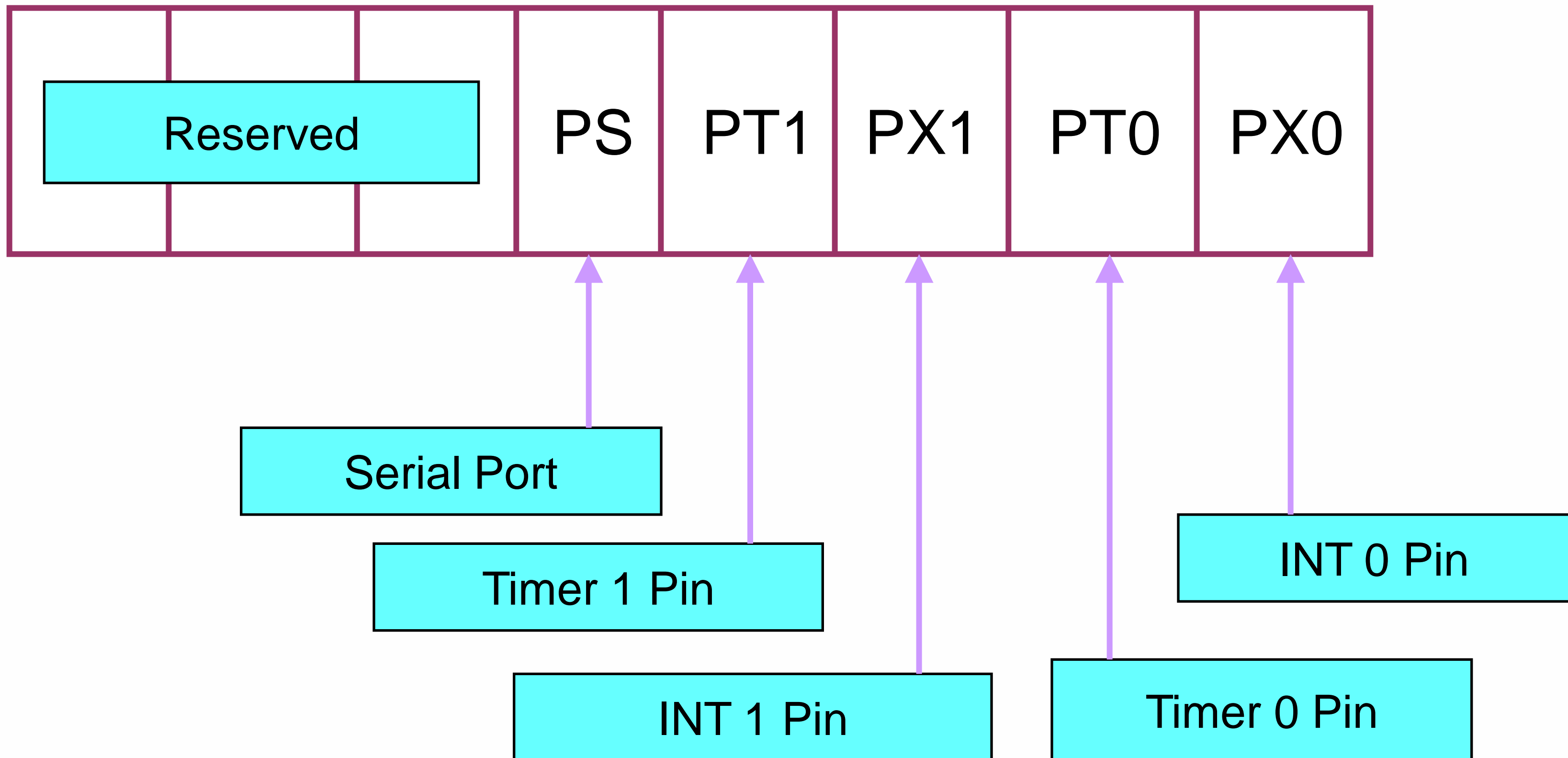
- In reality, the priority scheme is nothing but an internal polling sequence in which the 8051 polls the interrupts in the sequence listed and responds accordingly.

Highest To Lowest Priority	
External Interrupt 0	(INT0)
Timer Interrupt 0	(TF0)
External Interrupt 1	(INT1)
Timer Interrupt 1	(TF1)
Serial Communication	(RI + TI)

Interrupt Priority

- **We can alter** the sequence of interrupt priority by assigning a higher priority to any one of the interrupts by programming a register called **IP (interrupt priority).**
- To give a higher priority to any of the interrupts, we make the **corresponding bit in the IP register high.**

Interrupt Priority (IP) Register



Priority bit=1 assigns high priority
Priority bit=0 assigns low priority

UNIT-III

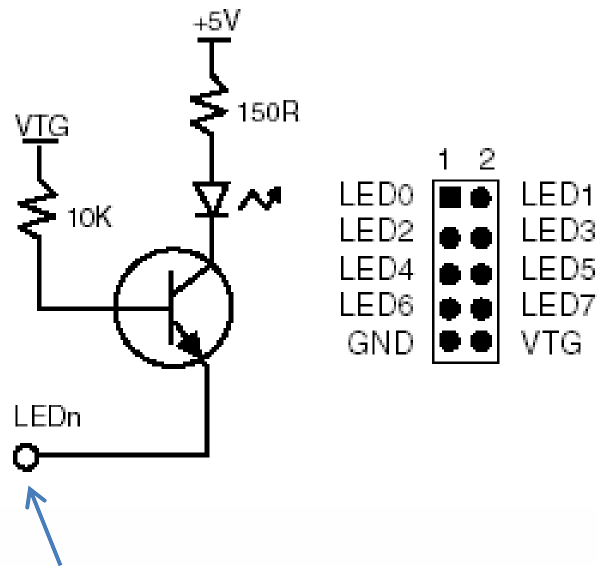
INTRODUCTION TO EMBEDDED C AND APPLICATIONS

An Embedded C Program the simplest form

```
void main()
{
            while (1)  //do forever
            ;
}
```

```
void main()
{
            printf(“begin measuring speed”);
            while(1)   //do forever
            ;
}
```

Turn on/off LED



Output a 0 to turn on LED
Output a 1 to turn off LED

Variable Types and Sizes

Type	Size(Bits)	Range
bit	1	0,1
char	8	-128 to 127
unsigned char	8	0 to 255
int	16	-32768 to 32767
short int	16	-32768 to 32767
unsigned int	16	0 to 65535
signed int	16	-32768 to 32767
long int	32	
unsigned long int	32	
signed long int	32	
float	32	+-1.175e-38 to +-3.4e38
double	32	+-1.175e-38 to +-3.4e38

Constants

Numerical Constants

decimal	1234
binary	0b10101011
hexadecimal	0xff
octal	0777

Character Constants

character	representation	Equivalent Hex Value
TAB	'\t'	'\x09'
LF (new line)	'\n'	'\x0a'
CR	'\r'	'\x0d'
Backspace	'\b'	'\x08'
--		
--		

example

```
printf("c = %d\n", c) //
printf("c = %d\n\r", c) //
```

Operators

Arithmetic Operators

Multiply	*
Divide	/
Modulo	%
Addition	+
Subtraction	-
Negation	-

Bitwise Operators

Ones complement	~
Left Shift	<<
Right Shift	>>
AND	&
Exclusive OR	^
OR	|

Beware division:

- If second argument is integer, the result will be integer (rounded):
  5 / 10 → 0 *whereas* 5 / 10.0 → 0.5
- Division by 0 will cause overflow

Bitwise Operations

Given an unsigned char y = 0xC9

operation	result
x = ~y	x = 0x36
x = y <<3	x = 0x48
x = y>>4	x = 0x0C
x = y&0x3F	x = 0x09
x = y^1	x = 0xC8
x = y	0x10	x = 0xD9

other examples:

```
unsigned  char z
z = PINA & 0x06;
PORTB = PORTB | 0x60;
PORTB = PORTB & 0xfe;
```

Logical Operators

Logical operator

AND	&&
OR	||

x =5 and y =2

(x && y)	is true, because both are non-zero
(x & y)	is false, because 00000101 bitwise AND 00000010 equal to zero
(x || y)	is true, because either value is non-zero
(x | y)	is true, b101 bitwise OR b010 is b111 (non-zero)

I/O Operations

```
unsigned char z;

void main (void)
{
        DDRB = 0xff;   // set port B as output port
        DDRA = 0x00; // set port A as input port

        while (1)
        {
                z = PINA;           // read port A
                PORTB = z + 1;      // write to port B
        }
}
// DDRx register is used to set which bits are to be used for output/input
//   DDRB = 0xc3;   11000011--, upper two bits and lower two bits for
//   output
```

I/O operations

```
unsigned char i; // temporary variable

DDRA = 0x00; // set PORTA for input
DDRB = 0xFF; // set PORTB for output
PORTB = 0x00; // turn ON all LEDs initially

while(1){
// Read input from PORTA.
// This port will be connected to the 8 switches
i = PINA;
// Send output to PORTB.
// This port will be connected to the 8 LEDs
PORTB = i;
}
```

I/O operations

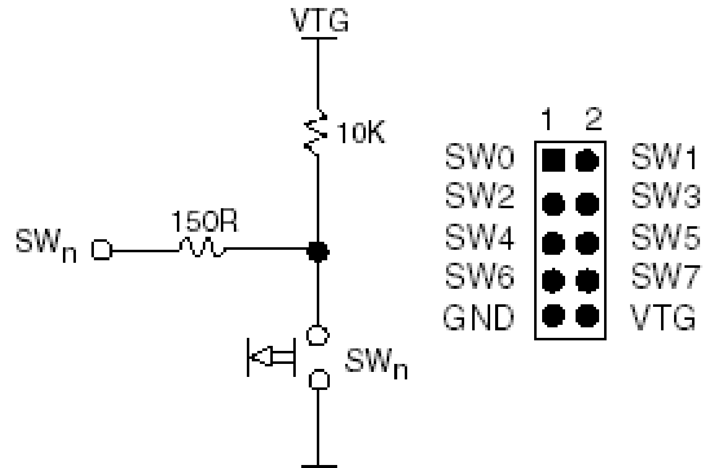
Turn on an LED connected to PB3

```
PORTB |= 0xF7;   // b11110111; PORTB=0x00 initially;
Must do the whole port
```

Turn on an LED connected to PB3

```
PORTB.3 = 0 ;  // access the bit 3 of port B, turn on the LED

for (delay = 0; delay < 10000; delay++); // declare delay as int somewhere

PORTB.3 = 1; // turn off the LED
```

I/O operation



Check if user pushed the button connected to PA5

```
swInput = PINA;
swInput = ~PINA;
if(swInput & 0x20) …
```

Division

Beware division:

- If second argument is integer, the result will be integer (rounded):
  5 / 10 → 0 *whereas* 5 / 10.0 → 0.5
- Division by 0 will cause a problem

Relational Operators

Relational Operators

Is Equal to	==
Is Not equal to	!=
Less Than	<
Less Than or Equal to	<=
Greater than	>
Greater Than or equal to	>=

x = 3 and y =5

(x == y)	FALSE
(x != y)	TRUE
(x < y)	TRUE
(x<=y)	TRUE
(x>y)	FALSE
(x >= y)	FALSE

Data format

Conversion specifier	Description
%d	display as a signed decimal integer
%6d	at least 6 characters wide
%u	display as an unsigned decimal integer
%x	display as an unsigned hexadecimal integer
%e	display a floating point value in exponential notation, such as 9.4567e2
%f	display a floating point value in fixed point notation, such as 945.67
%6f	at least 6 characters wide
%.2f	2 characters after decimal point
%6.2f	at least 6 characters wide and 2 after decimal point

Assignment Operators

```
x = y    assign y to x
x++      post-increment x
++x      pre-increment x
x--      post-decrement x
--x      pre-decrement x
```

```
x += y   assign (x+y) to x
x -= y   assign (x-y) to x
x *= y   assign (x*y) to x
x /= y   assign (x/y) to x
x %= y   assign (x%y) to x
```

```
int x=5;
int y;
y = ++x;
/* x == 6, y == 6 */
```

```
int x=5;
int y;
y = x++;
/* x == 6, y == 5 */
```

Do While loop

```
do                                  // mix up the numbers
      {                             // while waiting for button release.
         first ^= seed>>1; // Exclusive ORing in the moving seed
         second ^= seed>>2;
         third ^= seed>>3;
          seed++;          // keep rolling over the seed pattern
      }

while(PINA.0 == 0); // while the button is pressed
```

For Loop

```
for(count = 0; count < 5; count++)  // flash light while moving..
    {
        for(delay = 0; delay < 10000; delay++)
            ;                    // just count up and wait
        PORTB.1 = 0;         // turn the LED on..
        for(delay = 0; delay < 10000; delay++)
            ;
        PORTB.1 = 1;         // turn the LED off..
    }
```

If Then Else

```
if((first == 3) && (second == 3) && (third == 3))
        printf("Paid out: JACKPOT!!\n");  // Three "Cherries"
    else if((first == 3) || (second == 3) || (third == 3))
        printf("Paid out: One Dime\n");     // One "Cherry"
    else if((first == second) && (second == third))
        printf("Paid out: One Nickle\n");   // Three of a kind
    else
        printf("Paid out: ZERO\n");                   // Loser..
```

UNIT-IV
INTRODUCTION TO REAL – TIME OPERATING SYSTEMS

Tasks and Task States

- A **task** is the basic building block of software in an RTOS and is usually a subroutine.
- The RTOS starts a task by specifying its corresponding subroutine, priority, stack etc.
- The task can have 3 states :-

1. **Running –** The task code is currently being executed by the microprocessor. Except in multi-processor systems, only one task is in running state.
2. **Ready –** The task is waiting to execute but another task is currently running.
3. **Blocked –** The task cannot run even if the microprocessor is free. It might be waiting for an external event or a response. e.g. a push-button task stays blocked until the button is pushed.

- Most RTOSs have other states like **suspended, waiting, dormant** etc. but these are just sub-divisions of one of the three states above.

Tasks and Task States (contd.)

The Scheduler

- A **scheduler** in the RTOS decides which task to run.
- Unlike Unix or Windows the scheduler is simple – the task in the ready state that has the highest priority will run.
- It is the user's responsibility to ensure that the highest priority task doesn't hog the processor.
- Fig. 6.1 from Simon shows the task states. Also the following definitions will be assumed:
  - **Block** – move into blocked state
  - **Run** – move into the running state
  - **Switch** – change which task runs
- A few results from the fig. are as follows:
  - A task can be blocked only by its own decision and not by the scheduler or another task. Hence a task can only enter the blocked state from the running state.
  - A task will remain blocked until the event it is waiting for occurs.
  - Only a scheduler can move a task to the running state from the ready state.

Tasks and Task States (contd.)

- Some common questions about the scheduler and task states are

  How does a scheduler know that a task is blocked or unblocked?

  The task calls RTOS functions to indicate which functions it is waiting for and if they have happened.

  What if all tasks are blocked?

  It is the user's responsibility to ensure this doesn't happen. Unless an interrupt or event unblocks a task, tasks will stay in this state (deadlock).

  What if two tasks with same priority are ready?

  Depends on the RTOS. Some require distinct priorities, some time-slice between the 2 tasks.

  If a higher priority task unblocks, is the current running task moved to the ready state right away?

  This will happen only if the RTOS is **preemptive**.

Tasks and Task States (contd.)

Example

The following pseudo-code e.g. (fig. 6.2 Simon) illustrates tasks in RTOS.

```
// "Button Task"
void vButtonTask ()   // High priority
{
   while (TRUE)
   {
         !! Block until user pushes a button
         !! Quick: respond to the user
   }
}

// "Levels Task"
void vLevelsTask ()   // Low priority
{
   while (TRUE)
   {
         !! Read float levels in tank
         !! Calculate average float level
```

Tasks and Task States (contd.)

```
        !! Do a lot of calculations
        !! Select next tank
    }
}
```

- The code is from the underground tank monitoring system.
- Task `vLevelsTask` uses as much computing time possible to determine the gasoline level and is hence kept at a lower priority.
- `vButtonTask` will pre-empt the lower priority task whenever it is ready, finish servicing the pushed button, and then block.
- Tasks can be independent of one another in an RTOS.
- To insert the tasks in the RTOS, it must be initialized (`InitRTOS()`) tasks must be started and their priorities specified (`StartTask()`) and the OS needs to be started (`StartRTOS()`).
- Once the OS is started, function never returns.

Tasks and Data

- Each of the tasks have a set of register, a program counter and a stack.
- Tasks can also communicate using shared (global) variables.
- Fig. 6.6 illustrates this. It is basically the previous underground tank example code with some additional functions.
- The two tasks share the `tankData` array

Tasks and Data (contd.)

Shared-Data Problem

- The above code will have bugs because they are sharing the same variable and the lower priority task might be pre-empted in the middle of a data write operation.
- A similar problem occurs when 2 tasks call the same function.

```
void Task1 (void)
{
        .
        .
        vCountErrors(9);
        .
        .
}

void Task1 (void)
{
        .
        .
        vCountErrors(11);
        .
        .
}
```

Tasks and Data (contd.)

```
static int cErrors;

void vCountErrors(int cNewErrors)
{
          cErrors +=cNewErrors;
}
```

- As both tasks call `vCountErrors` they hence share the variable `cErrors` causing potential bugs.

Reentrancy

- A function that can be called by multiple tasks and still work correctly is called **reentrant**.
- The 3 rules that determine if a function is reentrant are :-
  1. The function must not use variables nonatomically unless they are stored on the stack of the calling task or are the local variables of that task.
  2. The function must not call functions that are not reentrant
  3. It must not use hardware nonatomically.

Tasks and Data (contd.)

Review of C Variable Storage

- The following code (fig. 6.9)shows which variables are stored in memory instead of stack and can hence cause problems.

```
static int static_int;
int public_int;
int initialized =4;
char *string = "Where am I stored?";
void *vPointer;

void function (int parm, int *parm_ptr)
{
        static int static_local;
        int local;
        .
        .
}
```

Tasks and Data (contd.)

- `static_int` – stored in memory and hence a shared variable
- `public_int` – Same as above. However in addition, functions in other C files can also access this variable.
- `intitialized`– Ditto.
- `string` – Same
- `vPointer` – Same
- `parm` – stored on stack so will not cause a problem
- `parm_ptr` – stack. Will not cause problem as long as every task passes a different value for it.
- `static_local` – stored in memory. Only difference between it and `static_int` is that the other can be accessed by other functions in the C file while this variable can only be accessed by `function`.
- `local` – stack.

Tasks and Data (contd.)

Gray Areas of Reentrancy

- The following code falls in the gray area between reentrant and nonreentrant functions.

```
static int errors;
void vCountErrors()
{
  ++errors;
}
```

- Where it falls depends on the processor e.g. an 8051 might translate `++errors` as 8-9 assembly instructions in which case it is nonatomic while an 8086 microprocessor might just give

```
INC (errors)
RET
```

In which case `++errors` is atomic making the function reentrant.

Semaphores and Shared Data

- **Semaphores** are one way of protecting shared variables.
- Name is derived from the old railroad days when they were used to share a segment of rail between more than one train.

RTOS Semaphores

- Consider two functions for dealing with the RTOS binary semaphores – `TakeSemaphore` and `ReleaseSemaphore`.
- If a task has called `TakeSemaphore` to take the semaphore, any other task calling it will block until the semaphore is released (`ReleaseSemaphore`).
- Fig. 6.12 solves the shared-data problem of fig. 6.6 using a semaphore.

Semaphores and Shared Data (contd.)

- With this new setup consider the scenario where the 'levels task' (`vCalculateTankLevels`) has just taken the semaphore and is pre-empted by the higher priority 'push button' task.
  - The RTOS will move the higher priority task to the running state.
  - When this task tries to lock the semaphore, it will block.
  - The OS will then run the task which has the semaphore (levels task) until it releases it.
  - As soon as this happens, the RTOS will switch back to the higher priority task which is now unblocked.

Semaphores and Shared Data (contd.)

Reentrancy and Semaphores

- The following code shows how a shared function shown before can be made reentrant by using semaphores.

```
void Task1(void)
{  .
   .
   vCountErrors(5);
   .
}
void  Task2(void)
{  .
   .
   vCountErrors(10);
   .
}

static int cErrors;
static NU_SEMAPHORE semErrors;
```

Semaphores and Shared Data (contd.)

```
void vCountErrors (int cNewErrors)
{
  NU_Obtain_Semaphore (&semErrors, NU_SUSPEND);
  cErrors +=cNewErrors;
  NU_Release_Semaphore (&semErrors);
}
```

- Functions and data structures beginning with “NU” are those used in an RTOS called **Nucleus**.

Multiple Semaphores

- RTOSs normally allow the users to have multiple semaphores that are distinctly identified by a parameter (`semErrors` in above code) and are independent of each other.
- This speeds task responses – a high priority task does not have to be blocked by a lower priority one as long as it is using a different semaphore.
- It is the users responsibility to remember which semaphore protects which shared data – the OS will not do that.

Semaphores and Shared Data (contd.)

Semaphores as Signaling Devices

- Semaphores can be used to communicate between a task and another task or an interrupt routine.
- Fig. 6.16 – Simon shows a printer example.
- A task stores formatted reports, to be printed, into memory.
- The printer interrupts after each line, on which the ISR feeds it the next line.
- This is done by having the task wait on a semaphore after it has formatted a report. The ISR will release the semaphore once the report has been printed and the task can start on the next report.
- Note – *the semaphore has been initialized as already taken*. Hence the task can only take the semaphore after the first report. This is acts as initializing the process for the task.

Semaphores and Shared Data (contd.)

Semaphores Problems

- ***Forgetting to take it*** – Semaphores only work if tasks actually remember to use them while accessing shared data.
- ***Forgetting to release the semaphore*** – This will cause all tasks using the semaphore to be eventually blocked forever.
- ***Holding it for too long*** – Higher priority tasks might loose their deadlines if some lower priority task holds the semaphore for too long.

  A problem that can happen is if a low priority task C has a semaphore and a higher priority task B that does not use the semaphore pre-empts it in between.

  Now suppose the highest priority task A comes along and is blocked on the semaphore. As B has a higher priority than C it will run instead and might block A for long enough that it misses its deadline.

  This is called **priority inversion**. Some RTOSs temporarily give task A's priority to C (and hence prevent B from pre-empting C).

Semaphores and Shared Data (contd.)

- ***Causing a deadly embrace (deadlock)*** – The following code (fig. 6.18) illustrates this problem.

```
int a,b;
AMXID hSemaphoreA;
AMXID hSemaphoreB;
void Task1 ()
{
  ajsmrsv (hSemaphoreA, 0, 0);
  ajsmrsv (hSemaphoreB, 0, 0);
  a = b;
  ajsmrls (hSemaphoreA);
  ajsmrls (hSemaphoreB);
}

void Task2 ()
{
  ajsmrsv (hSemaphoreB, 0, 0);
  ajsmrsv (hSemaphoreA, 0, 0);
```

Semaphores and Shared Data (contd.)

```
        b = a;
        ajsmrls (hSemaphoreB);
        ajsmrls (hSemaphoreA);
    }
```

- Functions `ajsmrsv` (reserve semaphore) and `ajsmrls` (release semaphore) are from an RTOS **AMX**.
- The additional parameters in `ajsmrsv` are the time-out and priority.
- Now suppose `Task1` has just reserved `hSemaphoreA` and is pre-empted by `Task2`. `Task2` reserves `hSemaphoreB` but when it tries to reserve `hSemaphoreA` it is blocked.
- The RTOS switches to `Task1` which tries to reserve `hSemaphoreB` and is blocked by `Task2`. Hence the 2 tasks block each other and are caught in a deadlock.
- Hence use of semaphores should be avoided where possible.

Semaphores and Shared Data (contd.)

Semaphores Variants

- Some systems allow semaphores that can be taken multiple time. Taking them decrements their count and releasing increments it. They are hence called **counting semaphores**
- Semaphores that can only be released by the task that took them are **resource semaphores**. Though they prevent shared-data bugs, they cannot be used for task inter-communication.
- A semaphore that deals with priority inversion is commonly called a **mutex semaphore** or **mutex** (mutually exclusive).

Methods to Protect Shared Data

- The 2 basic methods are disabling interrupts and using semaphores.
- A third method is disabling task switches, but this has no effect on interrupt routines.
- Note – ***interrupts are not allowed to take semaphores*** so they cannot be used if the data is shared between the task code and the ISR.

Message Queues, Mailboxes, and Pipes

- Besides shared variables and semaphores, tasks can communicate with each other using queues, mailboxes and pipes.
- E.g. if there are 2 tasks `Task1` and `Task2` that have to report each error they encounter, they can do so by reporting it through a queue to a separate `ErrorsTask` and going back to their own functions.
- `ErrorsTask` can be lower priority and hence the two main tasks are not delayed.

Some Ugly Details

- Most RTOSs require queues to be initialized before being used. Some OSs also require the user to allocate memory for them
- An additional parameter is required to distinguish among queues.
- If a task tries to write to a queue that is full, the RTOS generally returns an error, or in some cases it might block the task until another task reads the queue. The code should deal with these errors.
- RTOSs include functions that read from queues if they have data and return an error if they don't.
- Many RTOSs only allow a fixed amount of data that can be written in one call. A common method is to allow the user to write the number of bytes taken up by a *void pointer* in one function call.

Message Queues, Mailboxes, and Pipes (contd.)

Pointers and Queues

- Fig. 7.2 (Simon) shows how a void pointer is used to write to the queue.
- Users who want to send only a small amount of data can typecast the data as a void pointer.
- Another method involves allocating a data buffer (using `malloc`) and passing a pointer to that buffer to the queue.

Mailboxes

- Mailboxes are similar to queues. They have variations in different RTOSs as follows :-
  - Most RTOSs allow only one message in a mailbox though some might allow more.
  - Some OSs have unlimited message length mailboxes. The total messages in all the mailboxes however is limited and this is distributed among individual mailboxes.
  - Some RTOSs allow priorities in mailboxes – higher priority mailboxes will be read first

Message Queues, Mailboxes, and Pipes (contd.)

- In the RTOS *Multitask!* mailboxes are creating during the system configuration. This allows the following mailbox functions to be used

```
int sndmsg (unsigned int uMbId, void *p_vMsg, unsigned int
     uPriority);
void *rcvmsg (unsigned int uMbId, unsigned int uTimeout);
void *chkmsg (unsigned int uMbId);
```

- `uMbid` identifies the mailbox. `sndmsg` adds `p_vMsg` into the mailbox with the priority indicated by `uPriotity`.
- `rcvmsg` returns the highest priority message from the mailbox. If it is empty, the calling task blocks. `uTimeout` limits the time the task will wait.
- `chkmsg` returns the first message in the mailbox. If it is empty, it immediately returns `NULL`

Message Queues, Mailboxes, and Pipes (contd.)

Pipes

- Pipes are also similar to queues. Like mailboxes they have variations in different RTOSs.
  - Some RTOSs allow variable message lengths to be written to pipes.
  - In some RTOSs pipes are byte-oriented e.g. if Tasks A and B wrote 5 and 10 bytes respectively to the pipe and if Task C reads 7 bytes, it will get 5 bytes of A and 2 of B. The remaining bytes of B will remain in the pipe
  - Some RTOSs use C library functions `fread` and `fwrite` to read and write to pipes

Which to Use?

- Since queues, mailboxes and pipes vary in each RTOS, the user should read the corresponding documentation and determine which would best suit his communication requirement.

Pitfalls

- Although the above 3 mechanisms simplify sharing data they can also introduce several bugs.
  - User should ensure that tasks write to the correct mailboxes otherwise there might be errors.

Message Queues, Mailboxes, and Pipes (contd.)

- If a task writes an integer to a queue and the second task is expecting a character this will cause errors. Sometimes the compiler might not find the bug.
- Running out of space when a task needs to write data is another problem. The common solution is to make the pipe or queue large enough.
- Passing pointers from through a queue, mailbox or pipe creates shared data unintentially. Consider fig. 7.4. It does not use a data buffer.

  If the RTOS switches from to `vMainTask` to `vReadTemperaturesTask` while it was comparing `iTemperatures[0]` to `iTemperatures[1]` this translates into the shared-data problem discussed in Ch-4 and 6.

Timer Functions

- Embedded systems generally require to track time.
- A cell phone preserves battery by turning its display off after a few seconds. Network connections re-transmit data if an acknowledgement is not received within a certain period.
- Most RTOSs have a delay function that delays or blocks a task for a certain time period.
- e.g. in the U.S. each of the tones representing a digit in a phone call must sound for 1/10th of a second followed by the same period of silence between tones.

  This can be done by utilizing the function `taskDelay(100)` that delays the task for 100ms

Questions

- *How do I know that* `taskDelay` *takes milliseconds as its parameter?*

  You don't. In *VxWorks* for e.g. `taskDelay` takes system ticks as the parameter
- *How accurate are these delays?*

  They are accurate to the nearest tick. The RTOS sets up a hardware timer often called a **heartbeat timer** to periodically interrupt and bases its timings on the interrupt.

  Note – the task will *unblock* after it receives all the interrupts specified by the delay function, however it will run if no other higher priority task is ready.

Timer Functions (contd.)

- *How does the RTOS know how to setup the timer hardware ?*

  RTOSs are microprocessor-dependent and hence the engineers that wrote the RTOS know which processor it will run on and hence can program the corresponding timer. If the timer hardware is non-standard, the user is required to write his own timer setup and interrupt routines that will be called by the RTOS.

  Many vendors provide **board support packages** (**BSP**s) that help users to write driver software for any special hardware they are using

- *What is the "normal length" for a system tick?*

  There isn't one. Short system times provide accurate timings with the added disadvantage of occupying the processor more and reducing throughput. The designer must make a trade-off between the two.

- *What if the system requires extremely accurate timing?*

  The user can either use short system ticks or use a separate dedicated hardware timer for functions requiring accurate times and the RTOS for all other timings. The advantage of using the OS is that one timer handles many operations simultaneously.

Timer Functions (contd.)

Other Timing Services

- RTOSs offer many timing services based on the system tick like deciding the time a task should wait on a semaphore or mailbox etc.
- However user should exercise caution e.g. a higher priority task might timeout waiting on a semaphore and not get the shared data. The user code should handle such potential problems. A better design would be to allow a lower priority task get the shared data so the high priority task can continue its work.
- Fig. 7.7 shows a use of the timing services, where a function is called after waiting a given no. of system ticks.
- The code is written in *VxWorks* and basically handles hardware for a radio.
- Turning off the radio only requires turning the power off, however turning it on requires :–
  - First the power must be turned on.
  - After 12 ms the radio frequency must be set
  - 3 ms later the transmitter or receiver can be turned on and the radio is ready.

Events

- An **event** is basically a Boolean flag that tasks can set or reset for other tasks to wait on. Listed below are some of its features.
  - When an event occurs the RTOS unblocks all tasks waiting on it.
  - RTOSs normally form groups of events and a task can wait for a subset of events of that group.
  - Different RTOSs deal differently when resetting events. Some do this automatically while some require the user code to do this.

Comparison of Methods for Intertask Communication

- Here is a comparison between using queues, pipes, mailboxes or events for intertask communication
  - Semaphores are usually the fastest and simplest.
  - Events are slightly more complicated than semaphores and use a little more processor time. However a task can wait for any one of several events simultaneously but only for one semaphore.
  - Queues allow a lot of *data* as opposed to just events to be sent between tasks which makes them more flexible than events. However (1) Handling these messages is more microprocessor-intensive and (2) They can also cause more bugs.

    Mailboxes and pipes share all these charecteristics.

Memory Management

- RTOSs prefer allocating and freeing fixed-size buffers rather than using C functions like `malloc` and `free`, as they are faster and more predictable.
- The *MultiTask!* RTOS consists of **pools** that have a certain number of same-size memory buffers.
- `reqbuf` and `getbuf` allocate a buffer from the pool. The difference between the two is that the former returns a NULL pointer if the pool is empty while the latter blocks the task

```
void *getbuf (unsigned int PoolId, unsigned int Timeout);
void *reqbuf (unsigned int PoolId);
void relbuf (unsigned int PoolId, void *p_vBuffer);
```

- The buffer size returned depends on the pool. `relbuf` frees a buffer
- *MultiTask!* like typical RTOSs needs to be told where the memory is (`init_mem_pool`).

```
int init_mem_pool (
        unsigned int PoolId,
        void *p_vMemory,
        unsigned int BufSize,
        unsigned int BufCount,
        unsigned int PoolType);
```

Memory Management (contd.)

- `PoolId` – identifier
- `p_vMemory` – points to the pool memory block
- `BufSize` and `BufCount` – indicate size and count of buffer
- `PoolType` – indicates if buffers will be used by tasks or by interrupt routines.

Interrupt Routines in an RTOS Environment

- Most RTOSs require interrupt routines to follow 2 rules :-
  - *An ISR must not call any RTOS function that might block the caller*. Hence ISRs must not get semaphores, read from queues or empty mailboxes etc.
  - *An ISR must not call any RTOS function that might cause the OS to switch tasks unless the RTOS knows that an ISR, and not a task is running*. Hence ISRs must not release semaphores, write to queues or mailboxes, set events etc. – unless the RTOS knows that this is being done by the RTOS. If the ISR breaks this rule, the RTOS might switch control away from it thinking it is a task and the ISR might not get completed.

Rule 1: No Blocking

- Fig. 7.12 shows a nuclear reactor. The ISR and task code share the temperature through a semaphore.
- The code violates Rule 1 and will not work.
- If `vTaskTestTemperatures` is interrupted when it had the semaphore, the ISR would try to get the semaphore (`GetSemaphore`) and block.

Interrupt Routines in an RTOS Environment (contd.)

- Hence the ISR and task `vTaskTestTemperatures` would be caught in a deadlock.
- Some functions never block and can hence be called by the ISR e.g. a function that returns the status of a semaphore

Rule 2: No RTOS Calls without Fair Warning

- Suppose a low priority task is interrupted. If the ISR now writes to a mailbox (*breaks rule 2!*) what should ideally happen is that the RTOS should unblock tasks waiting on the mailbox and return to the ISR, eventually completing it.
- However what actually happens is that as soon as the tasks are unblocked, the RTOS thinks that the current task (ISR) is not the highest priority and instead of returning to the ISR it starts executing the highest priority task in the ready queue.
- Fig. 7.16 shows a solution. The RTOS intercepts all interrupts and calls the ISR. It thus knows it is entering an ISR and that it must return to it instead of switching to a high priority task.
- Fig. 7.17 shows another method. The ISR calls an RTOS function that tells it that it is running and so the RTOS will not switch to another task when it writes to a mailbox.
- After the ISR completes, it calls another RTOS function and the scheduler runs the highest priority task
- A third method is to provide separate special functions for ISRs. Hence `OSSemPost` for the ISR will be `OSSemPost`.
- In case of nested interrupts all the ISRs should inform the RTOS that they are executing, otherwise after completing a high priority ISR, the OS might switch over to a task and forget to execute the remaining lower priority ISRs.

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

- Introduction
  - Application programs are typically developed, compiled, and run on host system
  - Embedded programs are targeted to a target processor (different from the development/host processor and operating environment) that drives a device or controls
  - What tools are needed to develop, test, and locate embedded software into the target processor and its operating environment?

  - Distinction
  - Host: Where the embedded software is developed, compiled, tested, debugged, optimized, and prior to its translation into target device. (Because the host has keyboards, editors, monitors, printers, more memory, etc. for development, while the target may have not of these capabilities for developing the software.)
  - Target: After development, the code is cross-compiled, translated – cross-assembled, linked (into target processor instruction set) and ***located*** into the target

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

- Introduction – 1

- Cross-Compilers –
  - Native tools are good for host, but to port/locate embedded code to target, the host must have a tool-chain that includes a cross-compiler, one which runs on the host but produces code for the target processor
  - Cross-compiling doesn't guarantee correct target code due to (e.g., differences in word sizes, instruction sizes, variable declarations, library functions)

- Cross-Assemblers and Tool Chain
  - Host uses cross-assembler to assemble code in target's instruction syntax for the target
  - Tool chain is a collection of compatible, translation tools, which are 'pipelined' to produce a complete binary/machine code that can be linked and located into the target processor
  - (See Fig 9.1)

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

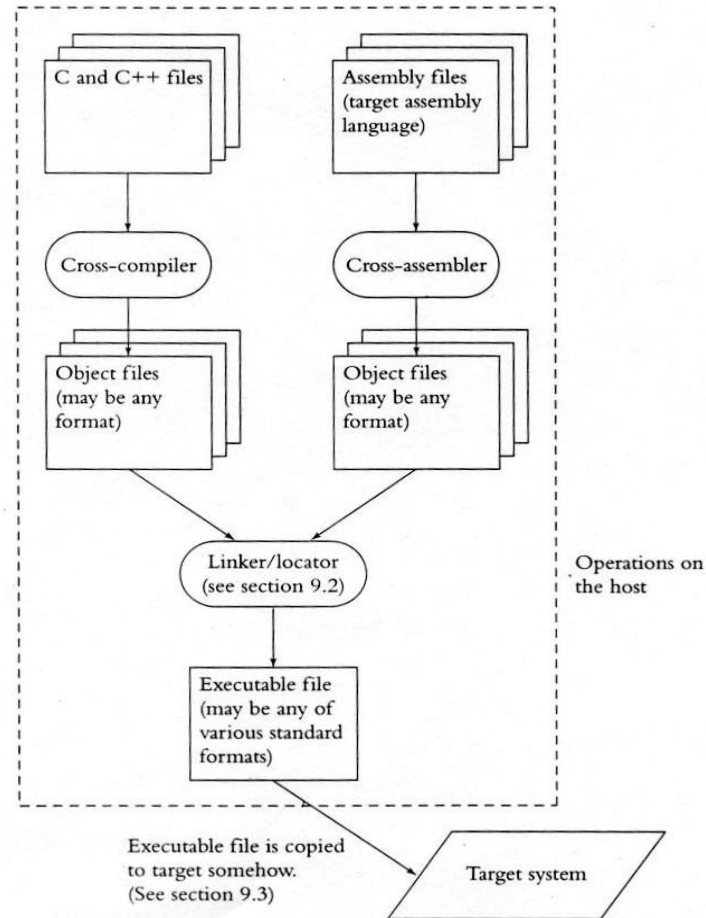


Figure 9.1 Tool Chain for Building Embedded Software

- **Linker/Locators for Embedded Software**
- Native linkers are different from cross-linkers (or locators) that perform additional tasks to ***locate*** embedded binary code into target processors

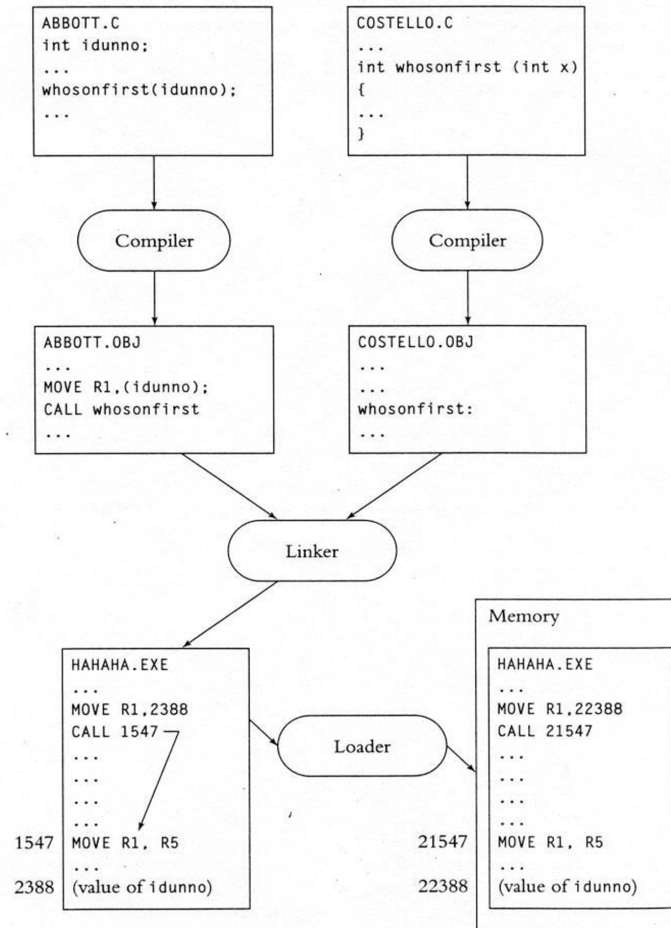
- Address Resolution –
  - Native Linker: produces host machine code on the hard-drive (in a named file), which the loader loads into RAM, and then schedules (under the OS control) the program to go to the CPU.

  - In RAM, the application program/code's logical addresses for, e.g., variable/operands and function calls, are ordered or organized by the linker. The loader then maps the logical addresses into physical addresses – a process called **address resolution**. The loader then loads the code accordingly into RAM (see Fig 9.2). In the process the loader also resolves the addresses for calls to the native OS routines

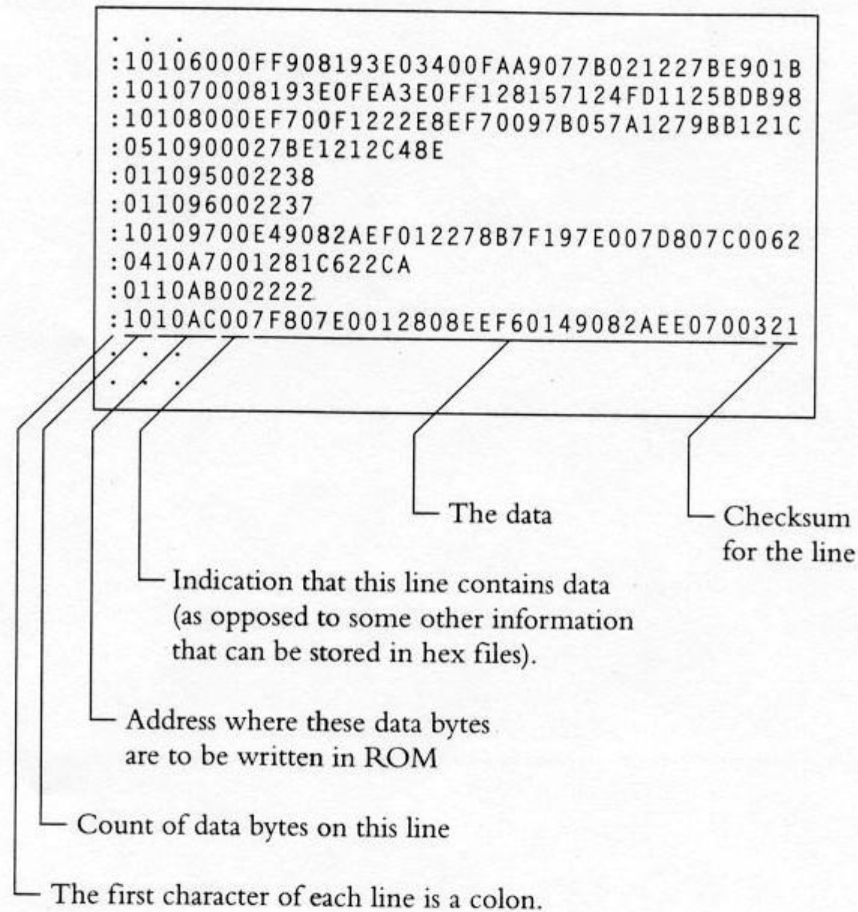
  - Locator: produces target machine code (which the locator glues into the RTOS) and the combined code (called **map**) gets copied into the target ROM. The locator doesn't stay in the target environment, hence all addresses are resolved, guided by locating-tools and directives, prior to running the code (See Fig 9.3 and Fig 9.4)

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

Figure 9.2 Native Tool Chain

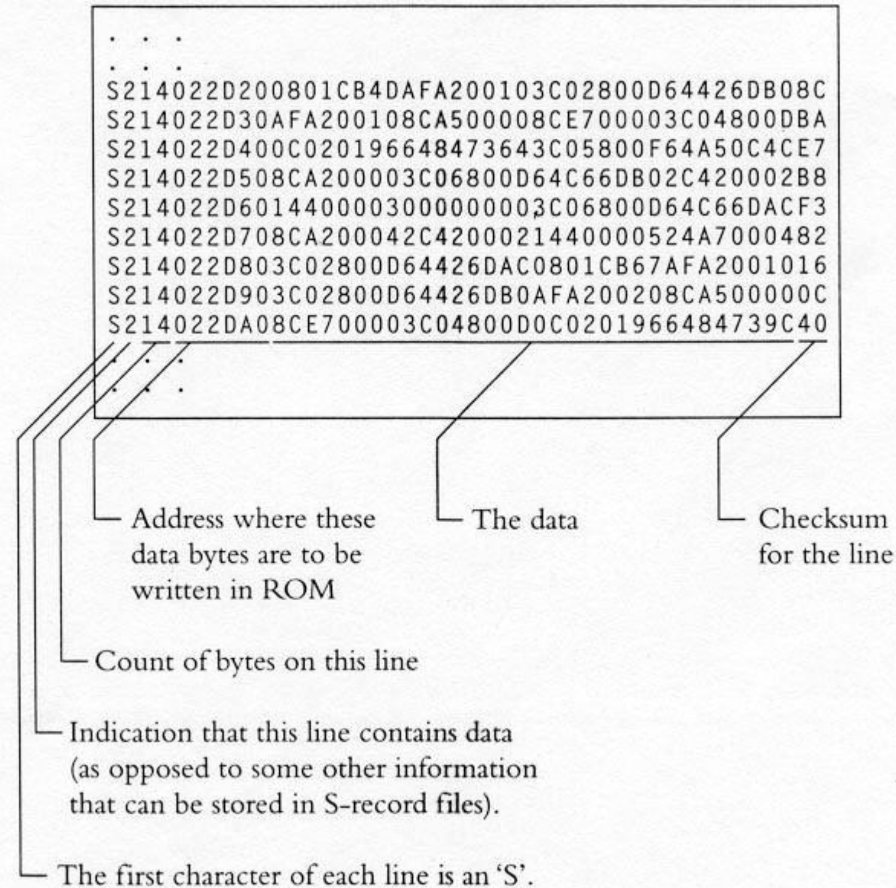


EMBEDDED SOFTWARE DEVELOPMENT TOOLS



EMBEDDED SOFTWARE DEVELOPMENT TOOLS

Figure 9.4 Motorola S-Record Format



EMBEDDED SOFTWARE DEVELOPMENT TOOLS

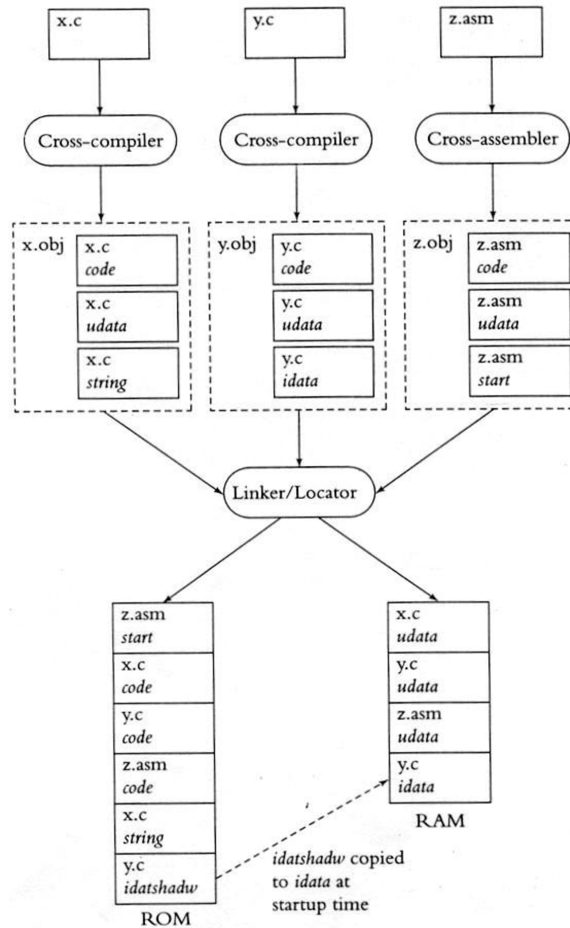
- Locating Program Components – Segments

- Unchanging embedded program (binary code) and constants must be kept in ROM to be remembered even on power-off
- Changing program segments (e.g., variables) must be kept in RAM
- Chain tools separate program parts using **segments** concept
- Chain tools (for embedded systems) also require a ‘start-up’ code to be in a separate segment and ‘located’ at a microprocessor-defined location where the program starts execution
- Some cross-compilers have default or allow programmer to specify segments for program parts, but cross-assemblers have no default behavior and programmer must specify segments for program parts

- (See Fig 9.5 - locating of object-code segments in ROM and RAM)

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

Figure 9.5 How the Tool Chain Uses Segments

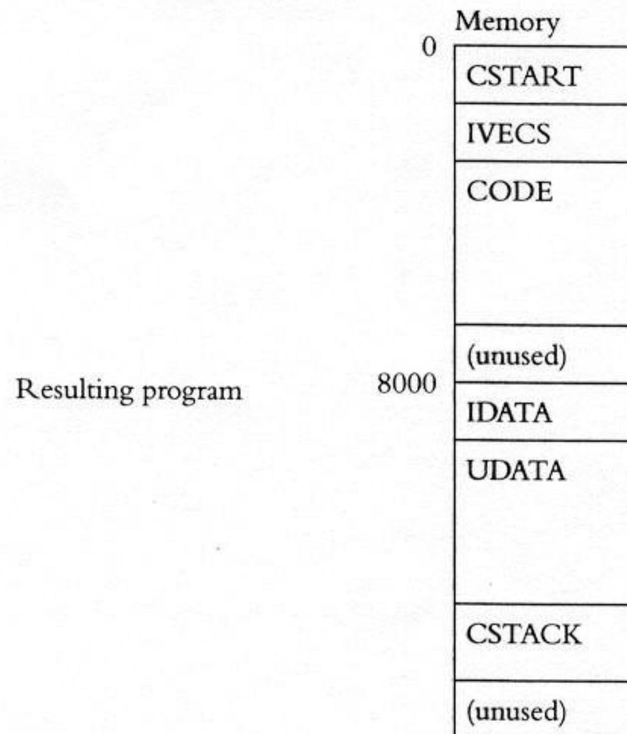


- **Locating Program Components – Segments – 1**
- Telling/directing the locator where (which segments) to place parts
- E.g., Fig 9.6
  - The –Z tells which segments (list of segments) to use and the start-address of the first segment
  - The first line tells which segments to use for the code parts, starting at address 0; and the second line tells which segments to use for the data parts, starting at x8000
  - The proper names and address info for the directing the locator are usually in the cross-compiler documentation
  - Other directives: range of RAM and ROM addresses, end of stack address (segment is placed below this address for stack to grow towards the end)
  - Segments/parts can also be ***grouped***, and the group is located as a unit

Figure 9.6 Locator Places Segments in Memory

Instructions to the locator:

```
-CZSTART,IVECS,CODE=0
-ZIDATA,UDATA,CSTACK=8000
```



EMBEDDED SOFTWARE DEVELOPMENT TOOLS

- Initialized Data and Constant Strings

- Segments with initialized values in ROM are shadowed (or copied into RAM) for correct reset of initialized variables, in RAM, each time the system comes up (esp. for initial values that are take #define constants, and which can be changed)
- In C programs, a host compiler may set all uninitialized variable to zero or null, but this is not generally the case for embedded software cross-compilers (unless the startup code in ROM does so
- If part(s) of a constant string is(are) expected to be changed during run-time, the cross-compiler must generate a code to allow 'shadowing' of the string from ROM

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

- Locator Maps and Executing Out of RAM

- Output file of locators are Maps – list addresses of all segments
- Maps are useful for debugging

- An ‘advanced’ locator is capable of running (albeit slowly) a startup code in ROM, which (could decompress and) load the embedded code from ROM into RAM to execute quickly since RAM is faster, especially for RISC microprocessors

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

Figure 9.7 Locator Map

```
LINK MAP OF MODULE:  XYZ

   TYPE     BASE      LENGTH    RELOCATION   SEGMENT NAME

-------------------------------------------------------

   * * * * * * *  X D A T A   M E M O R Y  * * * * * * *

            0000H     8100H                  *** GAP ***
   XDATA    8100H     0001H     UNIT         ?XD?PROGFLSH
   XDATA    8101H     000CH     UNIT         ?XD?VPROG?PROGFLSH
   XDATA    810DH     0006H     UNIT         ?XD?CHKSM?PROGFLSH
   XDATA    8113H     0080H     UNIT         ?C_LIB_XDATA
   XDATA    8193H     0002H     UNIT         ?XD?MAIN?PAD
   XDATA    8195H     0002H     UNIT         ?XD?RXCALLBACK?PAD
   .
   .
   .

   * * * * * * *  C O D E   M E M O R Y  * * * * * * *

            0000H     0017H                  *** GAP ***
   CODE     0080H     000FH     UNIT         PROGFLSTSTA
   CODE     008FH     0055H     UNIT         PROGFLSA
   CODE     00E4H     01ADH     UNIT         ?PR?VPROG?PROGFLSH
   CODE     0291H     0073H     UNIT         ?PR?SEND?PROGFLSH
   CODE     0304H     001DH     UNIT         ?PR?RX?PROGFLSH
   CODE     0321H     0072H     UNIT         ?PR?CHKSM?PROGFLSH
   CODE     0393H     007EH     INBLOCK      SCC_INIT
   CODE     0411H     082EH     UNIT         ?C_LIB_CODE
   .
   .
   .
```

Figure 9.7 *(continued)*

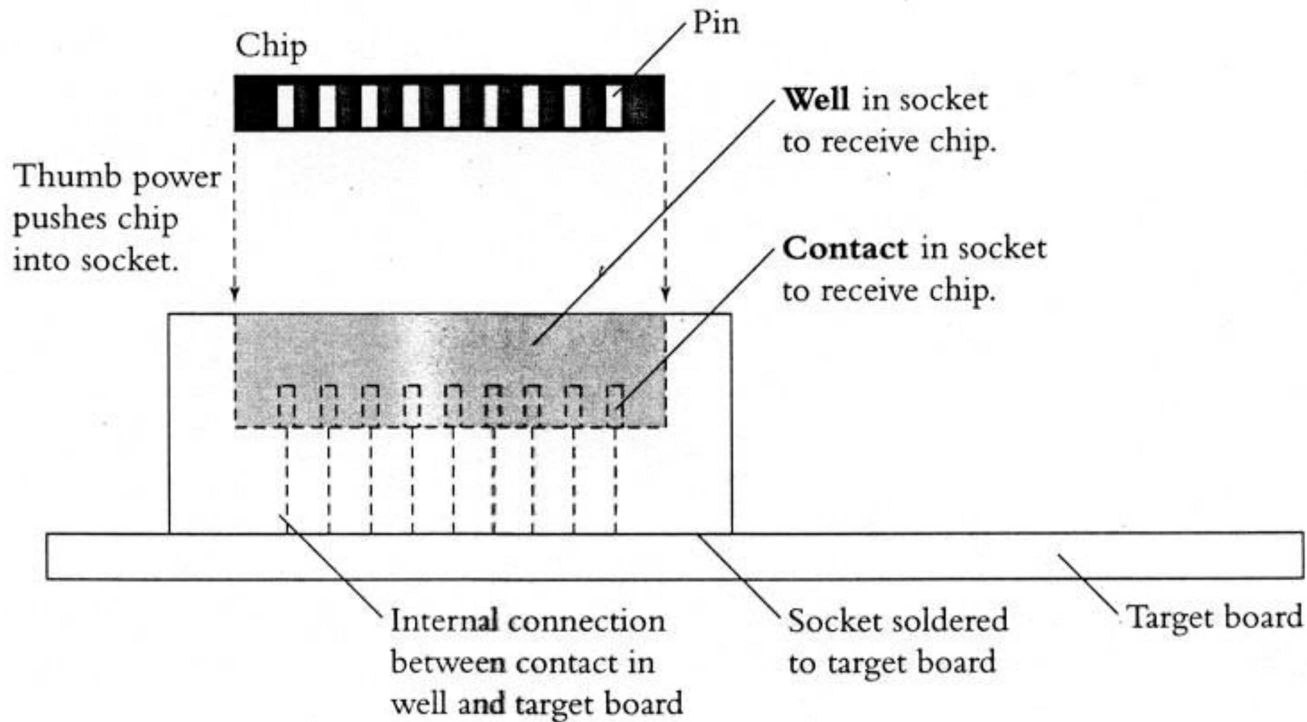
```
SYMBOL TABLE OF MODULE:  XYZ .

   VALUE          TYPE          NAME

-----------------------------------------

   -------        PROC          _FDECIMALASCIITOBYTE
   X:8301H        SYMBOL        p_b
   X:8304H        SYMBOL        p_byAscii
   X:8307H        SYMBOL        sizeofAByAscii
   D:0007H        SYMBOL        fReturn
   D:0006H        SYMBOL        bTemp
   -------        PROC          _FDECIMALASCIITOWORD
   X:8308H        SYMBOL        p_w
   X:830BH        SYMBOL        p_byAscii
   X:830EH        SYMBOL        sizeofAByAscii
   .
```

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

- **Getting Embedded Software into Target System**

- Moving maps into ROM or PROM, is to create a ROM using hardware tools or a PROM programmer (for small and changeable software, during debugging)
- If PROM programmer is used (for changing or debugging software), place PROM in a **socket** (which makes it erasable – for EPROM, or removable/replaceable) rather than 'burnt' into circuitry
- PROM's can be pushed into sockets by hand, and pulled using a chip puller
- The PROM programmer must be compatible with the format (syntax/semantics) of the Map

Figure 9.8 Schematic Edge View of a Socket

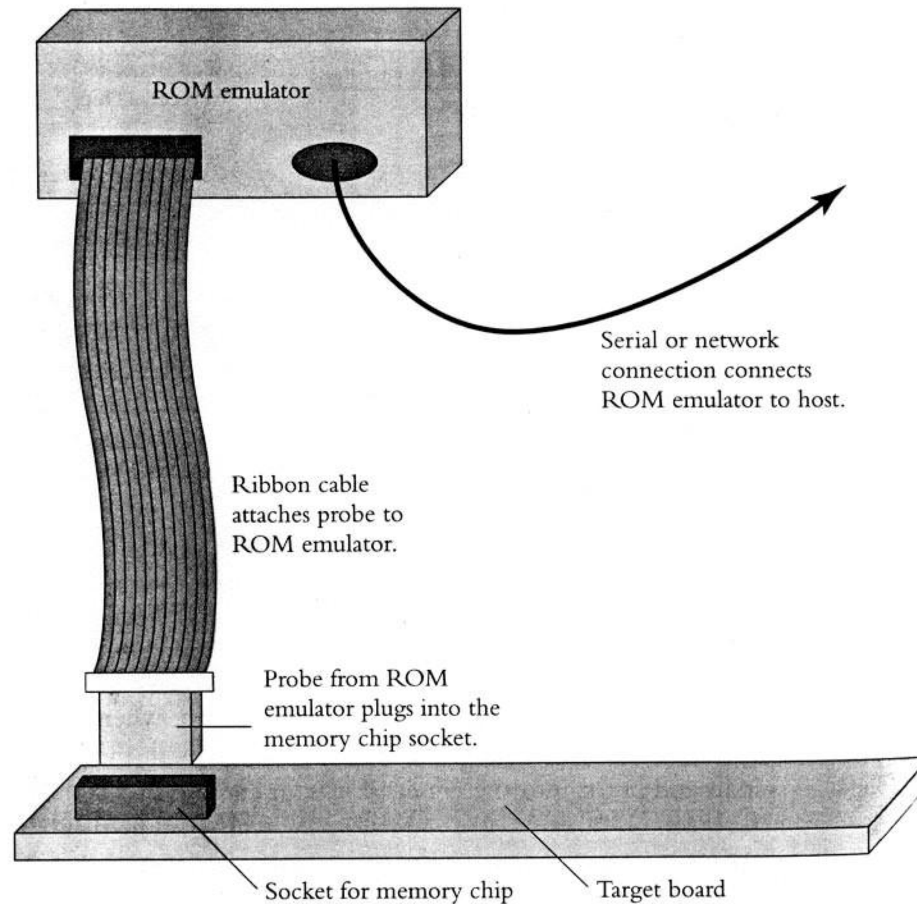


EMBEDDED SOFTWARE DEVELOPMENT TOOLS

- Getting Embedded Software into Target System – 1
- ROM Emulators – Another approach is using a ROM emulator (hardware) which emulates the target system, has all the ROM circuitry, and a serial or network interface to the host system. The locator loads the Map into the emulator, especially, for debugging purposes.
- Software on the host that loads the Map file into the emulator must understand (be compatible with) the Map's syntax/semantics

EMBEDDED SOFTWARE DEVELOPMENT TOOLS

Figure 9.9 ROM Emulator



EMBEDDED SOFTWARE DEVELOPMENT TOOLS

- Getting Embedded Software into Target System – 2

- Using Flash Memory
  - For debugging, a flash memory can be loaded with target Map code using a software on the host over a serial port or network connection (just like using an EPROM)
  - Advantages:
    - No need to pull the flash (unlike PROM) for debugging different embedded code
    - Transferring code into flash (over a network) is faster and hassle-free
    - New versions of embedded software (supplied by vendor) can be loaded into flash memory by customers over a network - Requires a) protecting the flash programmer, saving it in RAM and executing from there, and reloading into flash after new version is written and b) the ability to complete loading new version even if there are crashes and protecting the startup code as in (a)
    - Modifying and/or debugging the flash programming software requires moving it into RAM, modify/debug, and reloading it into target flash memory using above methods

UNIT-V
INTRODUCTION TO ADVANCED ARCHITECTURES

ARM instruction set

- ARM versions.
- ARM assembly language.
- ARM programming model.
- ARM memory organization.
- ARM data operations.
- ARM flow of control.

ARM versions

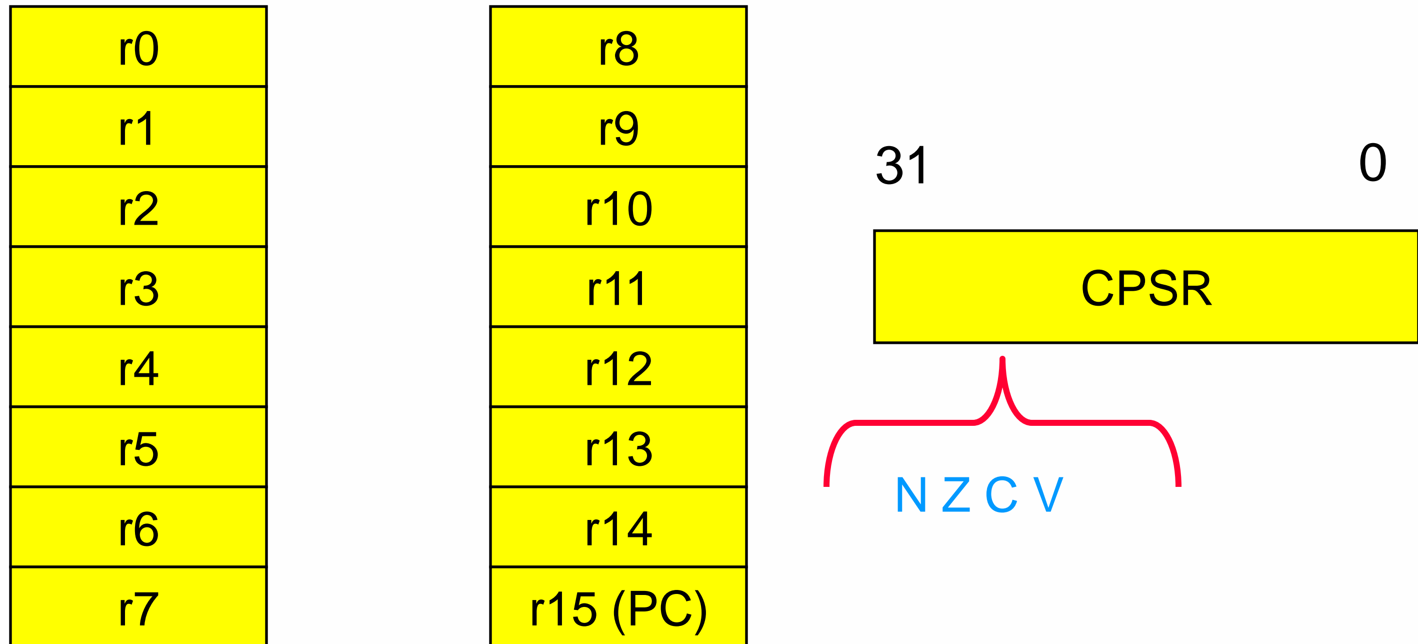
- ARM architecture has been extended over several versions.
- We will concentrate on ARM7.

ARM assembly language

- Fairly standard assembly language:

```
        LDR r0,[r8] ; a comment
label   ADD r4,r0,r1
```

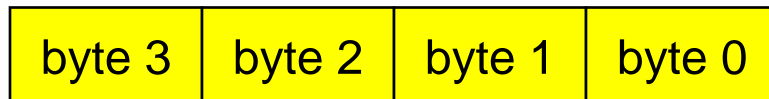
ARM programming model



Endianness

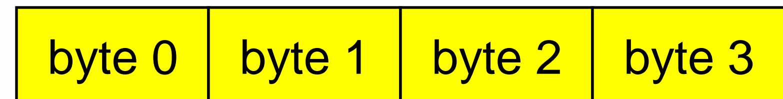
- Relationship between bit and byte/word ordering defines endianness:

bit 31 bit 0



little-endian

bit 0 bit 31



big-endian

ARM data types

- Word is 32 bits long.
- Word can be divided into four 8-bit bytes.
- ARM addresses can be 32 bits long.
- Address refers to byte.
  - Address 4 starts at byte 4.
- Can be configured at power-up as either little- or bit-endian mode.

ARM status bits

- Every arithmetic, logical, or shifting operation sets CPSR bits:
  - N (negative), Z (zero), C (carry), V (overflow).
- Examples:
  - -1 + 1 = 0: NZCV = 0110.
  - 2^{31}-1+1 = -2^{31}: NZCV = 1001.

ARM data instructions

- Basic format:

```
ADD r0,r1,r2
```

  – Computes r1+r2, stores in r0.

- Immediate operand:

```
ADD r0,r1,#2
```

  – Computes r1+2, stores in r0.

ARM data instructions

- ADD, ADC : add (w. carry)
- SUB, SBC : subtract (w. carry)
- RSB, RSC : reverse subtract (w. carry)
- MUL, MLA : multiply (and accumulate)
- AND, ORR, EOR
- BIC : bit clear
- LSL, LSR : logical shift left/right
- ASL, ASR : arithmetic shift left/right
- ROR : rotate right
- RRX : rotate right extended with C

Data operation varieties

- Logical shift:
  - fills with zeroes.
- Arithmetic shift:
  - fills with ones.
- RRX performs 33-bit rotate, including C bit from CPSR above sign bit.

ARM comparison instructions

- CMP : compare
- CMN : negated compare
- TST : bit-wise AND
- TEQ : bit-wise XOR
- These instructions set only the NZCV bits of CPSR.

ARM move instructions

- MOV, MVN : move (negated)

```
MOV r0, r1 ; sets r0 to r1
```

ARM load/store instructions

- LDR, LDRH, LDRB : load (half-word, byte)
- STR, STRH, STRB : store (half-word, byte)
- Addressing modes:
  - register indirect : `LDR r0,[r1]`
  - with second register : `LDR r0,[r1,-r2]`
  - with constant : `LDR r0,[r1,#4]`

ARM ADR pseudo-op

- Cannot refer to an address directly in an instruction.
- Generate value by performing arithmetic on PC.
- ADR pseudo-op generates instruction required to calculate address:

```
ADR r1,FOO
```

Example: C assignments

- C:

```
x = (a + b) - c;
```

- Assembler:

```
ADR r4,a          ; get address for a
LDR r0,[r4]       ; get value of a
ADR r4,b          ; get address for b, reusing r4
LDR r1,[r4]       ; get value of b
ADD r3,r0,r1      ; compute a+b
ADR r4,c          ; get address for c
LDR r2,[r4]       ; get value of c
```

C assignment, cont'd.

```
SUB r3,r3,r2      ; complete computation of x
ADR r4,x          ; get address for x
STR r3,[r4]       ; store value of x
```

Example: C assignment

- C:

```
y = a*(b+c);
```

- Assembler:

```
ADR r4,b ; get address for b
LDR r0,[r4] ; get value of b
ADR r4,c ; get address for c
LDR r1,[r4] ; get value of c
ADD r2,r0,r1 ; compute partial result
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
```

C assignment, cont'd.

```
MUL r2,r2,r0 ; compute final value for y
ADR r4,y ; get address for y
STR r2,[r4] ; store y
```

Example: C assignment

- C:

```
z = (a << 2) |  (b & 15);
```

- Assembler:

```
ADR r4,a ; get address for a
LDR r0,[r4] ; get value of a
MOV r0,r0,LSL 2 ; perform shift
ADR r4,b ; get address for b
LDR r1,[r4] ; get value of b
AND r1,r1,#15 ; perform AND
ORR r1,r0,r1 ; perform OR
```

C assignment, cont'd.

```
ADR r4,z ; get address for z
STR r1,[r4] ; store value for z
```

Additional addressing modes

- Base-plus-offset addressing:

```
LDR r0,[r1,#16]
```

  – Loads from location r1+16

- Auto-indexing increments base register:

```
LDR r0,[r1,#16]!
```

- Post-indexing fetches, then does offset:

```
LDR r0,[r1],#16
```

  – Loads r0 from r1, then adds 16 to r1.

ARM flow of control

- All operations can be performed conditionally, testing CPSR:
  - `EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE`
- Branch operation:

```
B #100
```

  - Can be performed conditionally.

Example: if statement

- C:

```
if (a > b) { x = 5; y = c + d; } else x = c - d;
```

- Assembler:

```
; compute and test condition
  ADR r4,a ; get address for a
  LDR r0,[r4] ; get value of a
  ADR r4,b ; get address for b
  LDR r1,[r4] ; get value for b
  CMP r0,r1 ; compare a < b
  BLE fblock ; if a ><= b, branch to false block
```

If statement, cont'd.

```
; true block
  MOV r0,#5 ; generate value for x
  ADR r4,x ; get address for x
  STR r0,[r4] ; store x
  ADR r4,c ; get address for c
  LDR r0,[r4] ; get value of c
  ADR r4,d ; get address for d
  LDR r1,[r4] ; get value of d
  ADD r0,r0,r1 ; compute y
  ADR r4,y ; get address for y
  STR r0,[r4] ; store y
  B after ; branch around false block
```

If statement, cont'd.

```
; false block
fblock ADR r4,c ; get address for c
   LDR r0,[r4] ; get value of c
   ADR r4,d ; get address for d
   LDR r1,[r4] ; get value for d
   SUB r0,r0,r1 ; compute a-b
   ADR r4,x ; get address for x
   STR r0,[r4] ; store value of x
after ...
```

Example: switch statement

- C:

```
switch (test) { case 0: … break; case 1: … }
```

- Assembler:

```
  ADR r2,test ; get address for test
  LDR r0,[r2] ; load value for test
  ADR r1,switchtab ; load address for switch table
  LDR r1,[r1,r0,LSL #2] ; index switch table
switchtab DCD case0
  DCD case1
...
```

Example: FIR filter

- C:

```
for (i=0, f=0; i<N; i++)
  f = f + c[i]*x[i];
```

- Assembler

```
; loop initiation code
  MOV r0,#0 ; use r0 for I
  MOV r8,#0 ; use separate index for arrays
  ADR r2,N ; get address for N
  LDR r1,[r2] ; get value of N
  MOV r2,#0 ; use r2 for f
```

FIR filter, cont'.d

```
   ADR r3,c ; load r3 with base of c
   ADR r5,x ; load r5 with base of x
; loop body
loop LDR r4,[r3,r8] ; get c[i]
   LDR r6,[r5,r8] ; get x[i]
   MUL r4,r4,r6 ; compute c[i]*x[i]
   ADD r2,r2,r4 ; add into running sum
   ADD r8,r8,#4 ; add one word offset to array index
   ADD r0,r0,#1 ; add 1 to i
   CMP r0,r1 ; exit?
   BLT loop ; if i < N, continue
```

ARM subroutine linkage

- Branch and link instruction:

```
BL foo
```

  – Copies current PC to r14.
- To return from subroutine:

```
MOV r15,r14
```

Nested subroutine calls

- Nesting/recursion requires coding convention:

```
f1      LDR r0,[r13] ; load arg into r0 from stack
        ; call f2()
        STR r14,[r13]! ; store f1's return adrs
        STR r0,[r13]! ; store arg to f2 on stack
        BL f2 ; branch and link to f2
        ; return from f1()
        SUB r13,#4 ; pop f2's arg off stack
        LDR r13!,r15 ; restore register and return
```

SHARC instruction set

- SHARC programming model.
- SHARC assembly language.
- SHARC memory organization.
- SHARC data operations.
- SHARC flow of control.

SHARC programming model

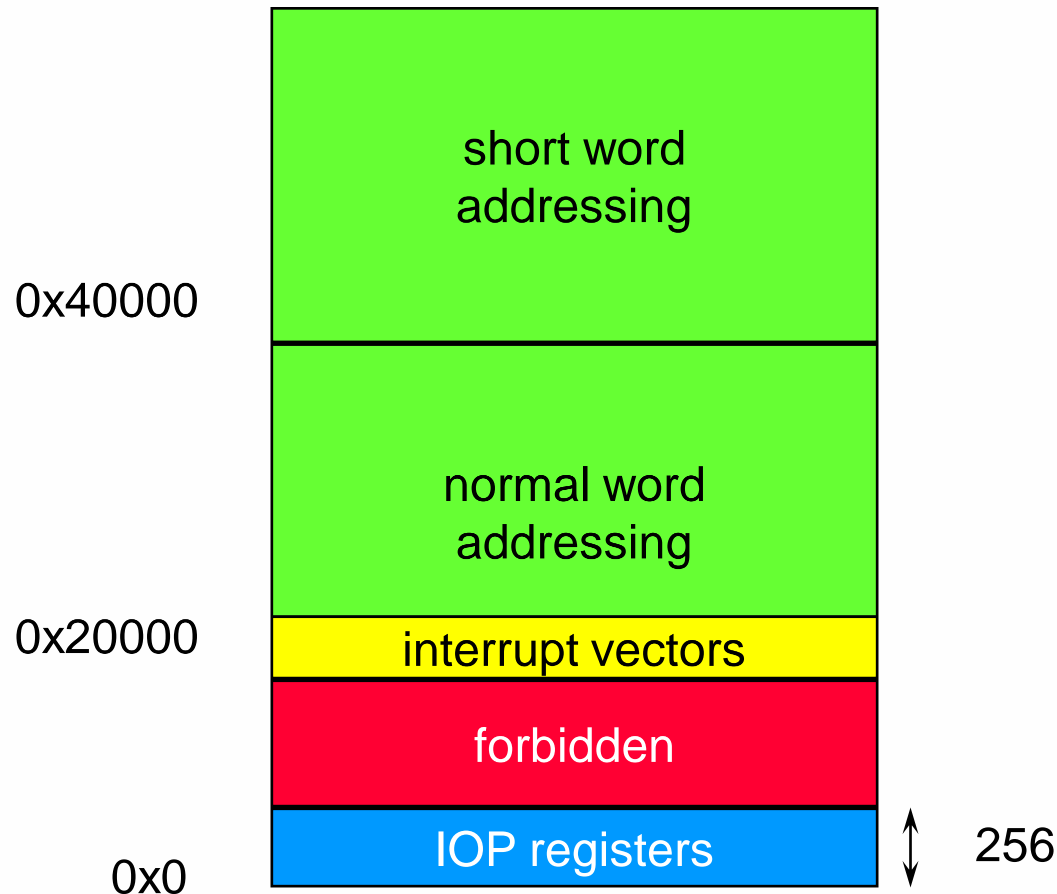
- Register files:
  - R0-R15 (aliased as F0-F15 for floating point)
- Status registers.
- Loop registers.
- Data address generator registers.
- Interrupt registers.

SHARC assembly language

- Algebraic notation terminated by semicolon:

```
R1=DM(M0,I0), R2=PM(M8,I8); ! comment
label: R3=R1+R2;
```

SHARC memory space



SHARC data types

- 32-bit IEEE single-precision floating-point.
- 40-bit IEEE extended-precision floating-point.
- 32-bit integers.
- Memory organized internally as 32-bit words.

SHARC micro architecture

- Modified Harvard architecture.
  - Program memory can be used to store some data.
- Register file connects to:
  - multiplier
  - shifter;
  - ALU.

SHARC mode registers

- Most important:
  - ASTAT: arithmetic status.
  - STKY: sticky.
  - MODE 1: mode 1.

Rounding and saturation

- Floating-point can be:
  - rounded toward zero;
  - rounded toward nearest.
- ALU supports saturation arithmetic (ALUSAT bit in MODE1).
  - Overflow results in max value, not rollover.

Multiplier

Fixed-point operations can accumulate into local MR registers or be written to register file. Fixed-point result is 80 bits.

Floating-point results always go to register file.

Status bits: negative, under/overflow, invalid, fixed-point undeflow, floating-point unerflow, floating-point invalid.

ALU/shifter status flags

ALU:

– zero, overflow, negative, fixed-point carry, inputsign, floating-point invalid, last op was floating-point, compare accumulation registers, floating-point under/oveflow, fixed-point overflow, floating-point invalid

Shifter:

– zero, overflow, sign

Flag operations

- All ALU operations set AZ (zero), AN (negative), AV (overflow), AC (fixed-point carry), AI (floating-point invalid) bits in ASTAT.
- STKY is sticky version of some ASTAT bits.

Example: data operations

- Fixed-point -1 + 1 = 0:
  - AZ = 1, AU = 0, AN = 0, AV = 0, AC = 1, AI = 0.
  - STKY bit AOS (fixed point underflow) not set.
- Fixed-point -2*3:
  - MN = 1, MV = 0, MU = 1, MI = 0.
  - Four STKY bits, none of them set.
- LSHIFT 0x7fffffff BY 3: SZ=0,SV=1,SS=0.

Multifunction computations

Can issue some computations in parallel:

- dual add-subtract;
- fixed-point multiply/accumulate and add,subtract,average
- floating-point multiply and ALU operation
- multiplication and dual add/subtract

Multiplier operand from R0-R7, ALU operand from R8-R15.

SHARC load/store

- Load/store architecture: no memory-direct operations.
- Two data address generators (DAGs):
  - program memory;
  - data memory.
- Must set up DAG registers to control loads/stores.

DAG1 registers

I	M	L	B
I0	M0	L0	B0
I1	M1	L1	B1
I2	M2	L2	B2
I3	M3	L3	B3
I4	M4	L4	B4
I5	M5	L5	B5
I6	M6	L6	B6
I7	M7	L7	B7

Data address generators

Provide indexed, modulo, bit-reverse indexing.

MODE1 bits determine whether primary or alternate registers are active.

BASIC addressing

- Immediate value:

```
R0 = DM(0x20000000);
```

- Direct load:

```
R0 = DM(_a); ! Loads contents of _a
```

- Direct store:

```
DM(_a)= R0; ! Stores R0 at _a
```

Post-modify with update

- I register holds base address.
- M register/immediate holds modifier value.

```
R0 = DM(I3,M3) ! Load
DM(I2,1) = R1 ! Store
```

- Circular buffer: L register is buffer start index, B is buffer base address.

Data in program memory

- Can put data in program memory to read two values per cycle:

```
F0 = DM(M0,I0), F1 = PM(M8,I9);
```

- Compiler allows programmer to control which memory values are stored in.

Example: C assignments

- C:

```
x = (a + b) - c;
```

- Assembler:

```
R0 = DM(_a) ! Load a
R1 = DM(_b); ! Load b
R3 = R0 + R1;
R2 = DM(_c); ! Load c
R3 = R3-R2;
DM(_x) = R3; ! Store result in x
```

Example, cont'd.

- C:

```
y = a*(b+c);
```

- Assembler:

```
R1 = DM(_b) ! Load b
R2 = DM(_c); ! Load c
R2 = R1 + R2;
R0 = DM(_a); ! Load a
R2 = R2*R0;
DM(_y) = R23; ! Store result in y
```

Example, cont'd.

- Shorter version using pointers:

```
! Load b, c
  R2=DM(I1,M5), R1=PM(I8,M13);
  R0 = R2+R1, R12=DM(I0,M5);
  R6 = R12*R0(SSI);
  DM(I0,M5)=R8; ! Store in y
```

Example, cont'd.

- C:

```
z = (a << 2) |  (b & 15);
```

- Assembler:

```
R0=DM(_a); ! Load a
R0=LSHIFT R0 by #2; ! Left shift
R1=DM(_b); R3=#15; ! Load immediate
R1=R1 AND R3;
R0 = R1 OR R0;
DM(_z) = R0;
```

SHARC program sequencer

Features:

- instruction cache;
- PC stack;
- status registers;
- loop logic;
- data address generator;

Conditional instructions

Instructions may be executed conditionally.

Conditions come from:

- arithmetic status (ASTAT);
- mode control 1 (MODE1);
- flag inputs;
- loop counter.

SHARC jump

- Unconditional flow of control change:

```
JUMP foo
```

- Three addressing modes:
  - direct;
  - indirect;
  - PC-relative.

Branches

Types: CALL, JUMP, RTS, RTI.

Can be conditional.

Address can be direct, indirect, PC-relative.

Can be delayed or non-delayed.

JUMP causes automatic loop abort.

Example: C if statement

- C:

```
if (a > b) { x = 5; y = c + d; }
  else x = c - d;
```

- Assembler:

```
! Test
  R0 = DM(_a); R1 = DM(_b);
  COMP(R0,R1); ! Compare
  IF GE JUMP fblock;
```

C if statement, cont'd.

```
! True block
tblock: R0 = 5; ! Get value for x
  DM(_x) = R0;
  R0 = DM(_c); R1 = DM(_d);
  R1 = R0+R1;
  DM(_y)=R1;
  JUMP other; ! Skip false block
```

C if statement, cont'd.

```
! False block
fblock: R0 = DM(_c);
  R1 = DM(_d);
  R1 = R0-R1;
  DM(_x) = R1;
other:  ! Code after if
```

Fancy if implementation

- C:

```
if (a>b) y = c-d; else y = c+d;
```

- Use parallelism to speed it up---compute both cases, then choose which one to store.

Fancy if implementation, cont'd.

```
! Load values
R1=DM(_a); R2=DM(_b);
R3=DM(_c); R4=DM(_d);
! Compute both sum and difference
R12 = r2+r4, r0 = r2-r4;
! Choose which one to save
comp(r8,r1);
if ge r0=r12;
dm(_y) = r0 ! Write to y
```

DO UNTIL loops

DO UNTIL instruction provides efficient looping:

```
        LCNTR=30, DO label UNTIL LCE;
        R0=DM(I0,M0), F2=PM(I8,M8);
        R1=R0-R15;
label:  F4=F2+F3;
```

Loop length

Last instruction in loop

Termination condition

Example: FIR filter

- C:

```
for (i=0, f=0; i<N; i++)
  f = f + c[i]*x[i];
```

FIR filter assembler

```
! setup
  I0=_a; I8=_b; ! a[0] (DAG0), b[0] (DAG1)
  M0=1; M8=1 ! Set up increments
! Loop body
  LCNTR=N, DO loopend UNTIL LCE;
  ! Use postincrement mode
  R1=DM(I0,M0), R2=PM(I8,M8);
  R8=R1*R2;
loopend: R12=R12+R8;
```

Optimized FIR filter code

```
  I4=_a; I12=_b;
  R4 = R4 xor R4, R1=DM(I4,M6),
  R2=PM(I12,M14);
  MR0F = R4, MODIFY(I7,M7);
! Start loop
  LCNTR=20, DO(PC,loop) UNTIL LCE;
loop: MR0F=MR0F+42*R1 (SSI), R1=DM(I4,M6),
  R2=PM(I12,M14);
! Loop cleanup
  R0=MR0F;
```

SHARC subroutine calls

- Use CALL instruction:

```
CALL foo;
```

- Can use absolute, indirect, PC-relative addressing modes.
- Return using RTS instruction.

PC stack

PC stack: 30 locations X 24 instructions.

Return addresses for subroutines, interrupt service routines, loops held in PC stack.

Example: C function

- C:

```
void f1(int a) { f2(a); }
```

- Assembler:

```
f1:  R0=DM(I1,-1); ! Load arg into R0
  DM(I1,M1)=R0; ! Push f2's arg
  CALL f2;
  MODIFY(I1,-1); ! Pop element
  RTS;
```

Important programming reminders

- Non-delayed branches (JUMP, CALL, RTS, RTI) do not execute 2 following instructions. Delayed branches are available.
- Cache miss costs at least one cycle to allow program memory bus to complete.

Important programming reminders, cont'd

- Extra cache misses in loops:
  - misses on first and last loop iteration if data memory is accessed in last 2 instructions of loop;
  - 3 misses if loop has only one instruction which requires a program memory bus access.
- 1-instr. loops should be executed 3 times, 2-instr. loops 2 times to avoid NOPs.

Important programming reminders, cont'd

- NOPs added for DAG register write followed by DAG data addressing in same register bank.
- Can program fixed wait states or ACK.
- Interrupt does not occur until 2 instructions after delayed branch.
- Initialize circular buffer by setting L to positive value, loading B to base.

Important programming reminders, cont'd

- Some DAG register transfers are disallowed.
- When given 2 writes to same register file in same cycle, only one actually occurs.
- Fixed- to floating-point conversion always rounds to 40 bits.
- Only DM bus can access all memory spaces.

Important programming reminders, cont'd

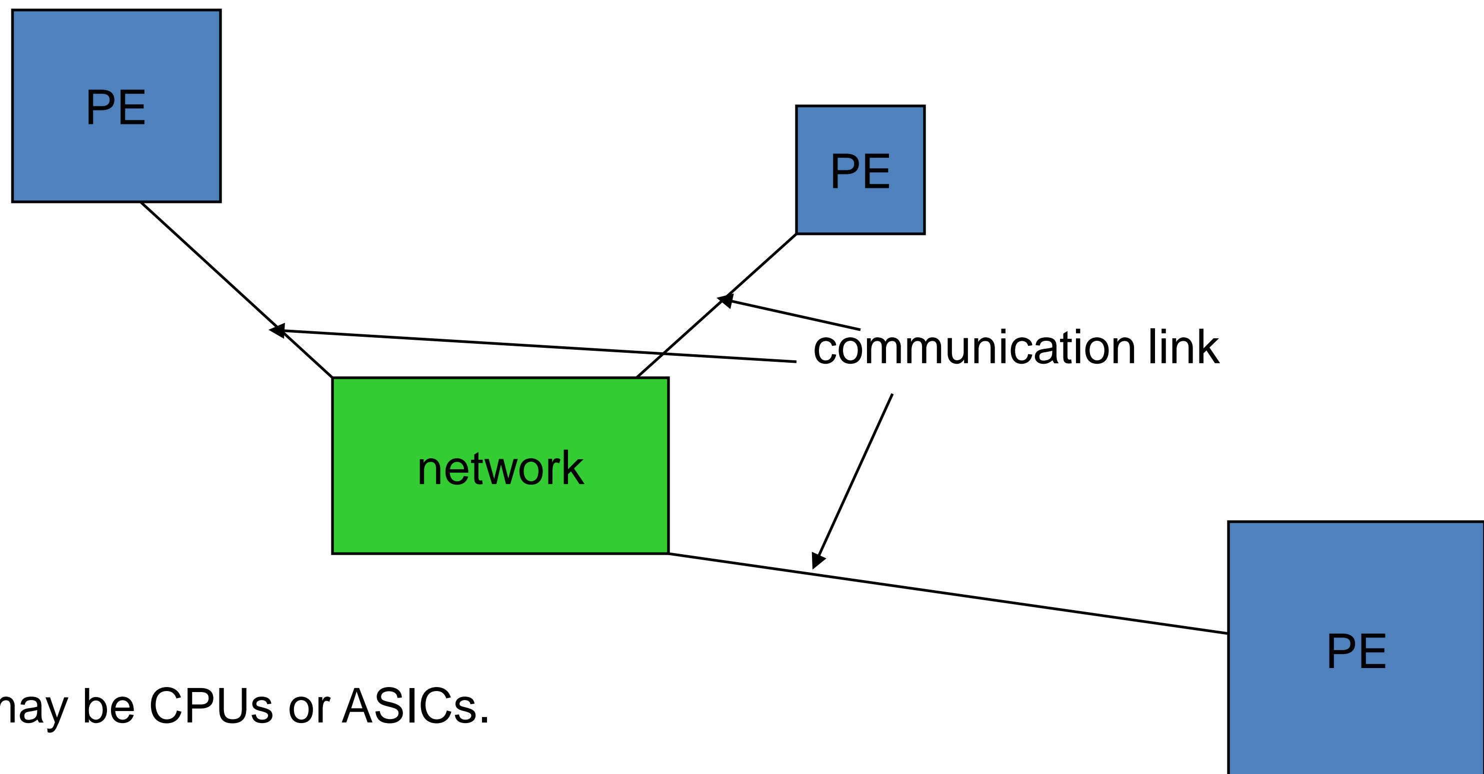
- When mixing 32-bit and 48-bit words in a block, all instructions must be below data.
- 16-bit short words are extended to 32 bits.
- For dual data access, use DM for data-only access, PM for mixed data/instruction block. Instruction comes from cache.
- A variety of conditions cause stalls.

Networking for Embedded Systems

- Why we use networks.
- Network abstractions.
- Example networks.

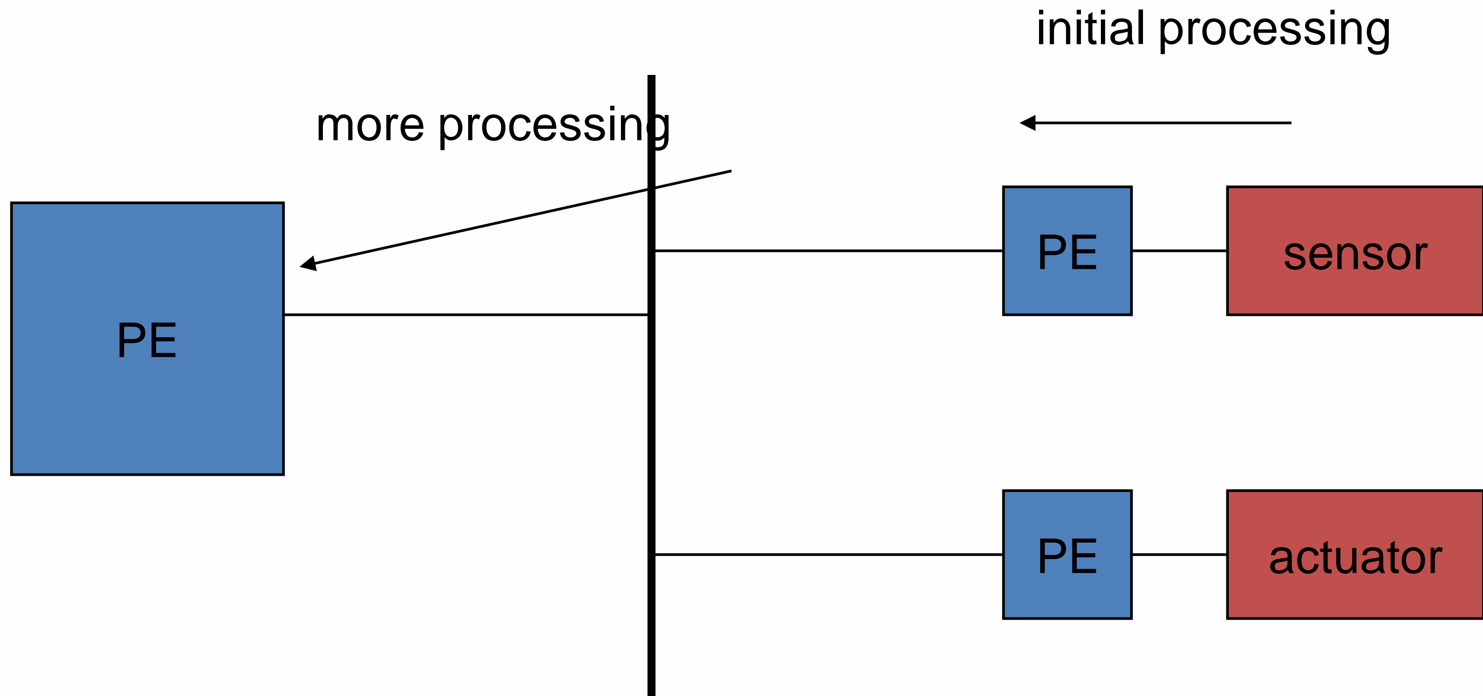
Network elements

distributed computing platform:



PEs may be CPUs or ASICs.

Networks in embedded systems



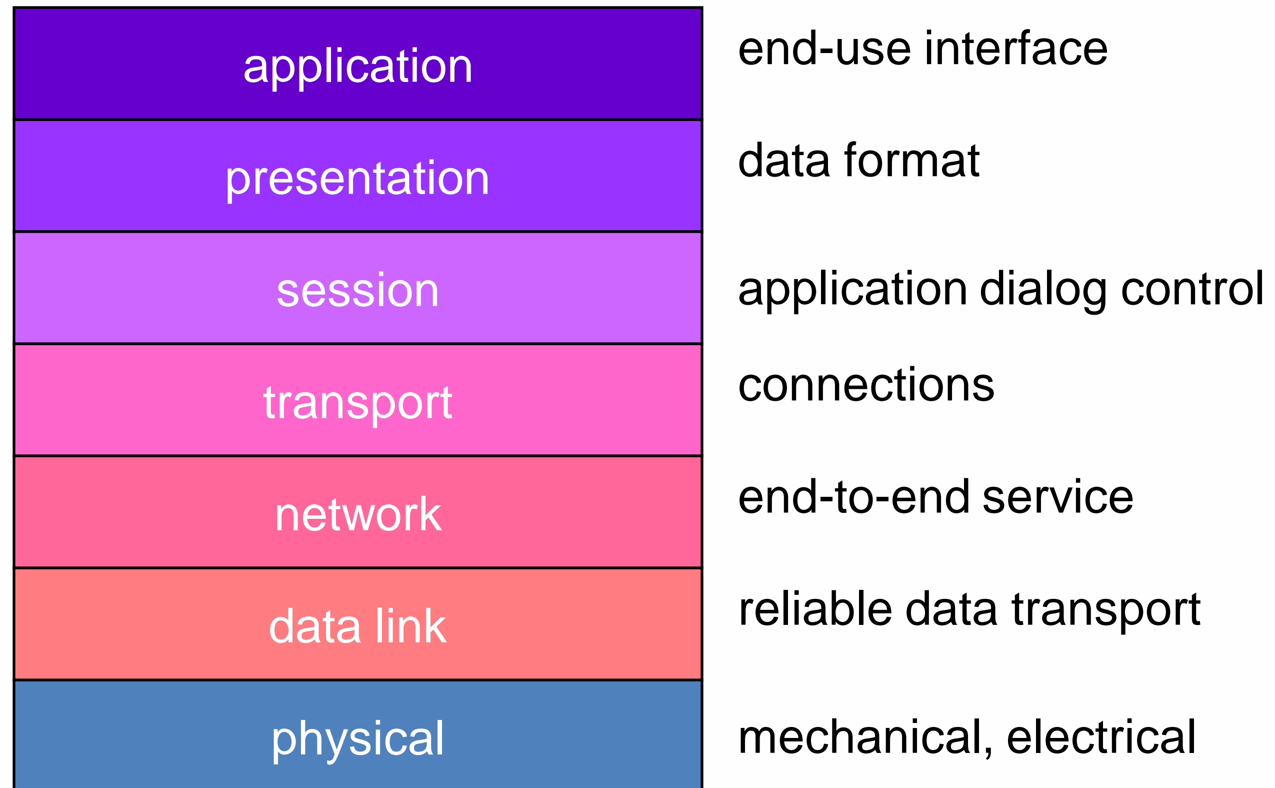
Why distributed?

- Higher performance at lower cost.
- Physically distributed activities---time constants may not allow transmission to central site.
- Improved debugging---use one CPU in network to debug others.
- May buy subsystems that have embedded processors.

Network abstractions

- International Standards Organization (ISO) developed the Open Systems Interconnection (OSI) model to describe networks:
  - 7-layer model.
- Provides a standard way to classify network components and operations.

OSI model



OSI layers

- Physical: connectors, bit formats, etc.
- Data link: error detection and control across a single link (single hop).
- Network: end-to-end multi-hop data communication.
- Transport: provides connections; may optimize network resources.

OSI layers, cont'd.

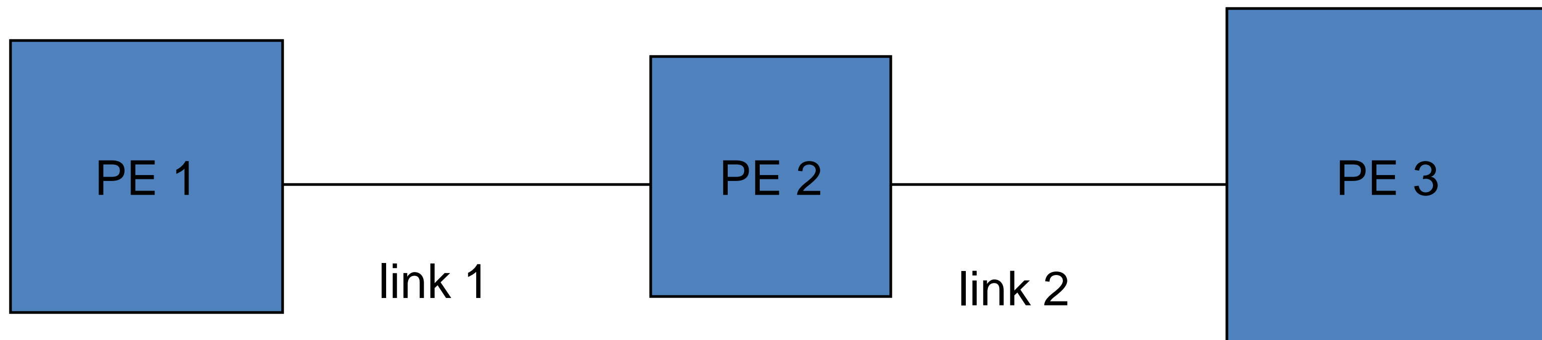
- Session: services for end-user applications: data grouping, checkpointing, etc.
- Presentation: data formats, transformation services.
- Application: interface between network and end-user programs.

Hardware architectures

- Many different types of networks:
  - topology;
  - scheduling of communication;
  - routing.

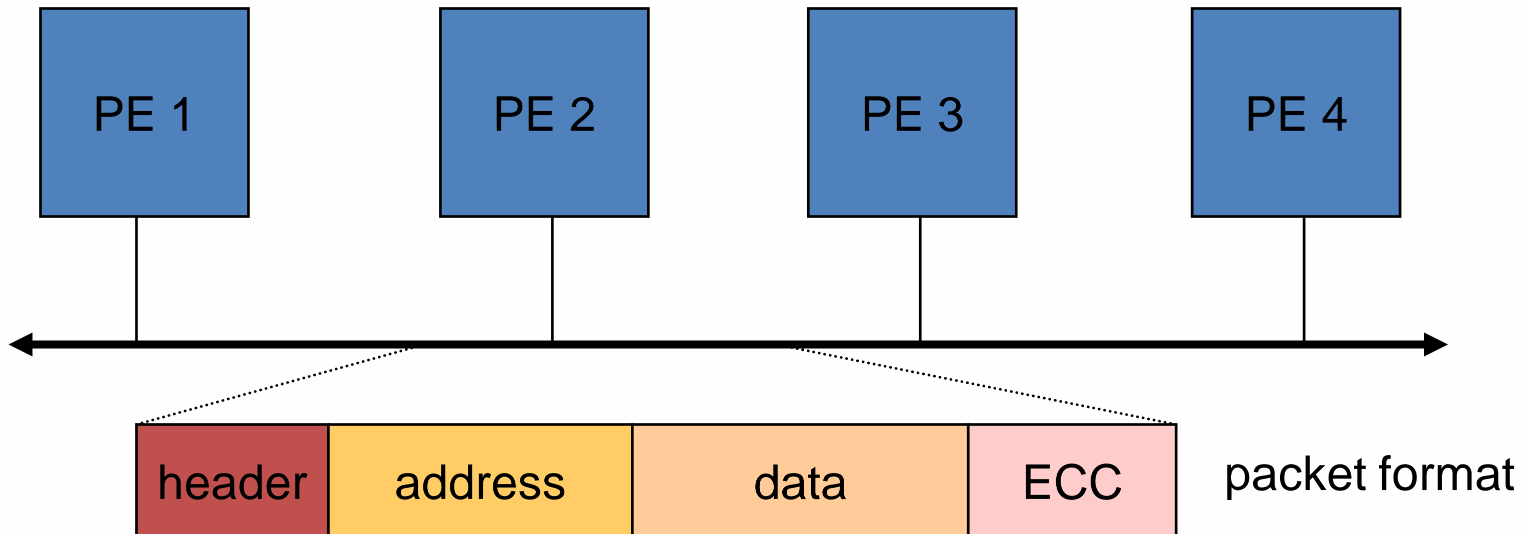
Point-to-point networks

- One source, one or more destinations, no data switching (serial port):



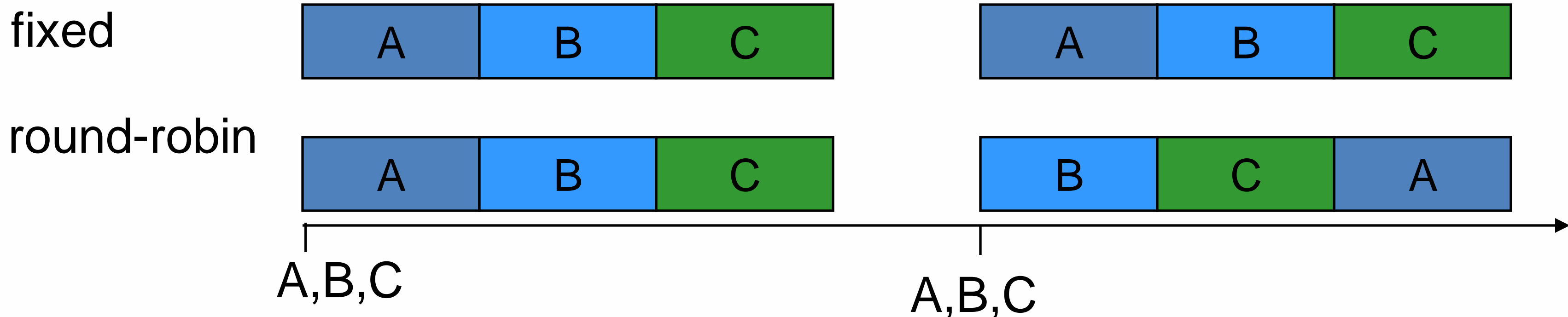
Bus networks

- Common physical connection:

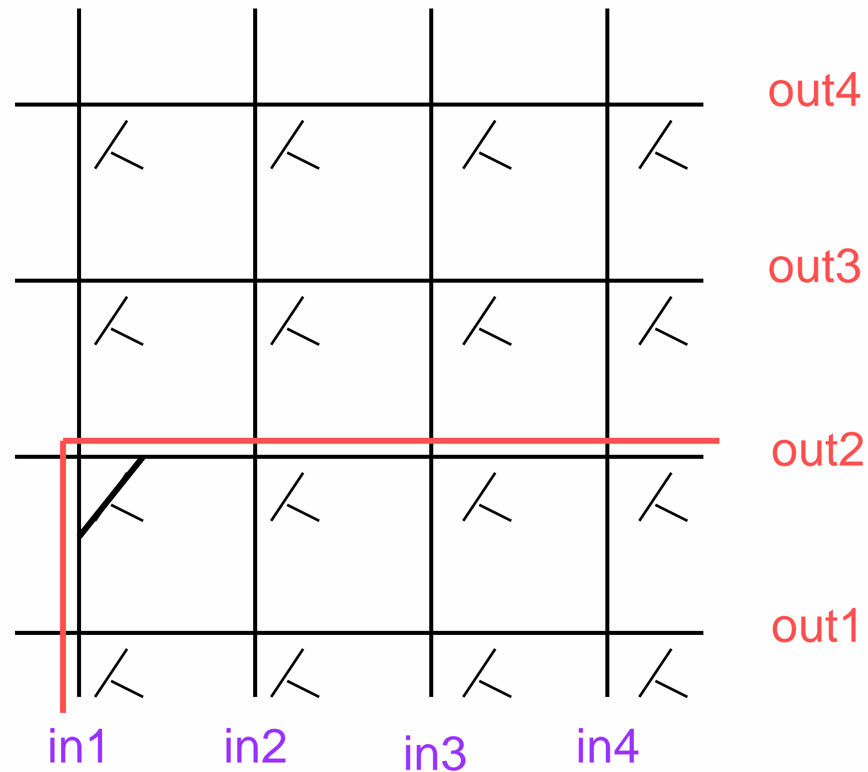


Bus arbitration

- Fixed: Same order of resolution every time.
- Fair: every PE has same access over long periods.
  - round-robin: rotate top priority among Pes.



Crossbar

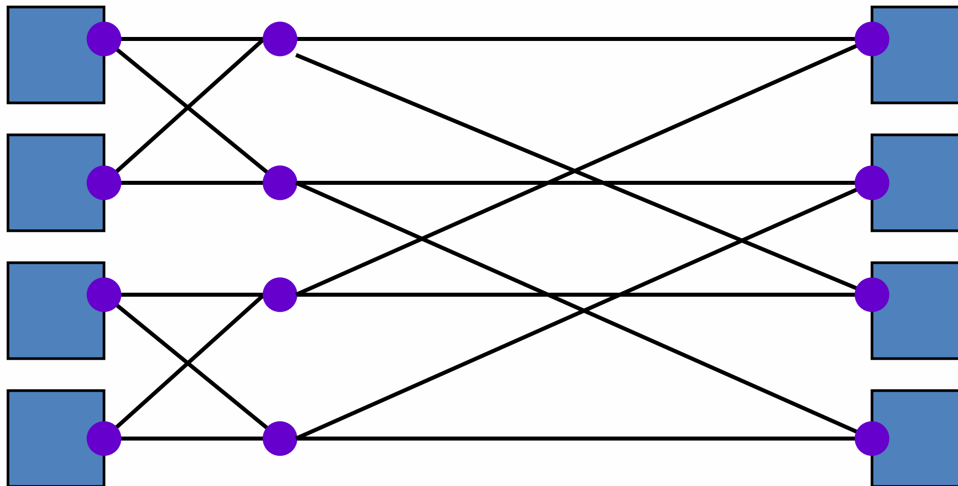


Crossbar characteristics

- Non-blocking.
- Can handle arbitrary multi-cast combinations.
- Size proportional to n^2.

Multi-stage networks

- Use several stages of switching elements.
- Often blocking.
- Often smaller than crossbar.



Message-based programming

- Transport layer provides message-based programming interface:

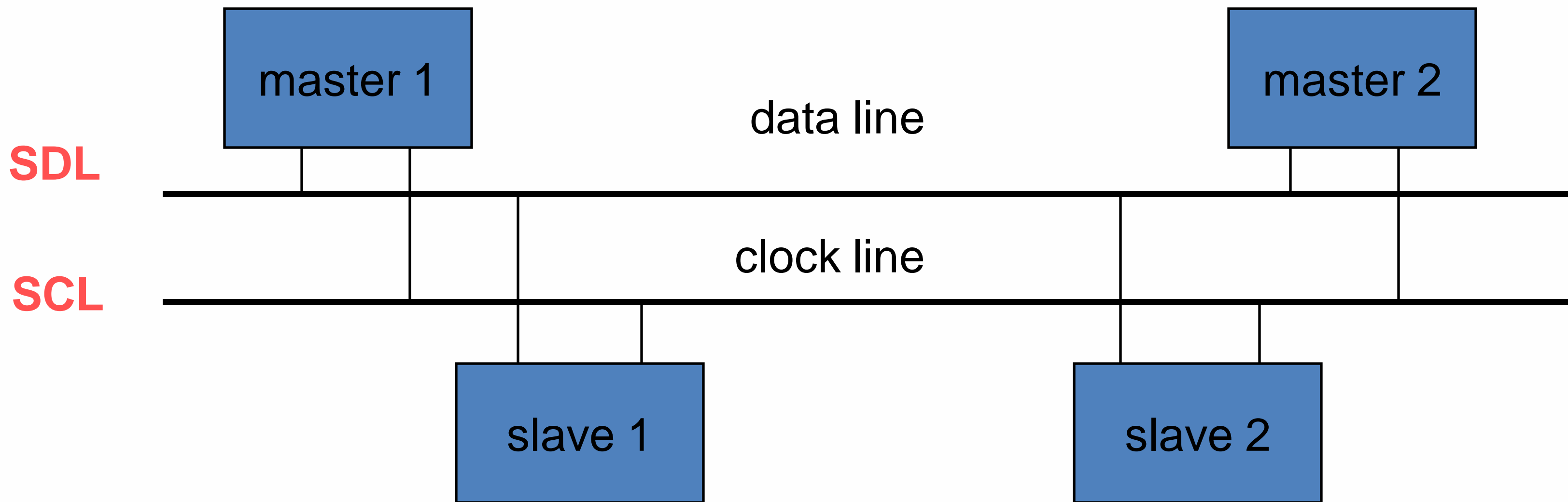
```
send_msg(adrs,data1);
```

- Data must be broken into packets at source, reassembled at destination.
- Data-push programming: make things happen in network based on data transfers.

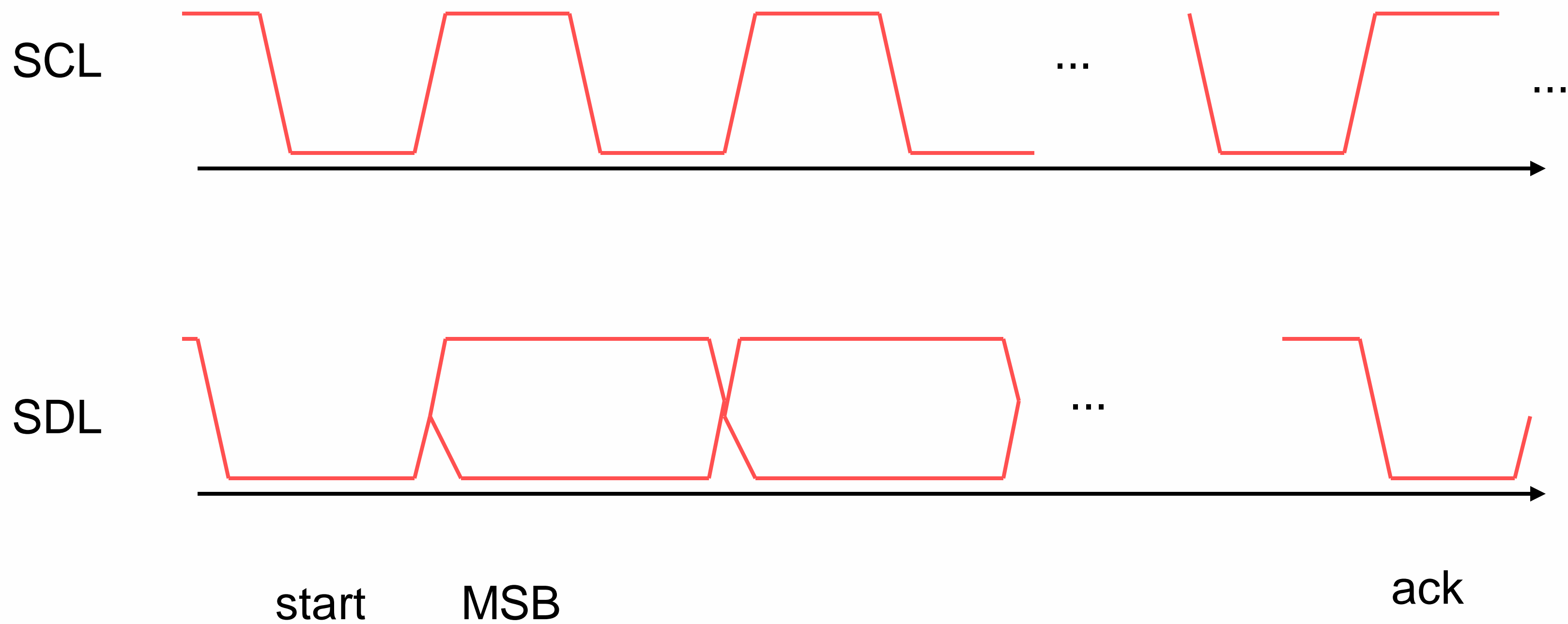
I^2C bus

- Designed for low-cost, medium data rate applications.
- Characteristics:
  - serial;
  - multiple-master;
  - fixed-priority arbitration.
- Several microcontrollers come with built-in I^2C controllers.

I^{2}C physical layer

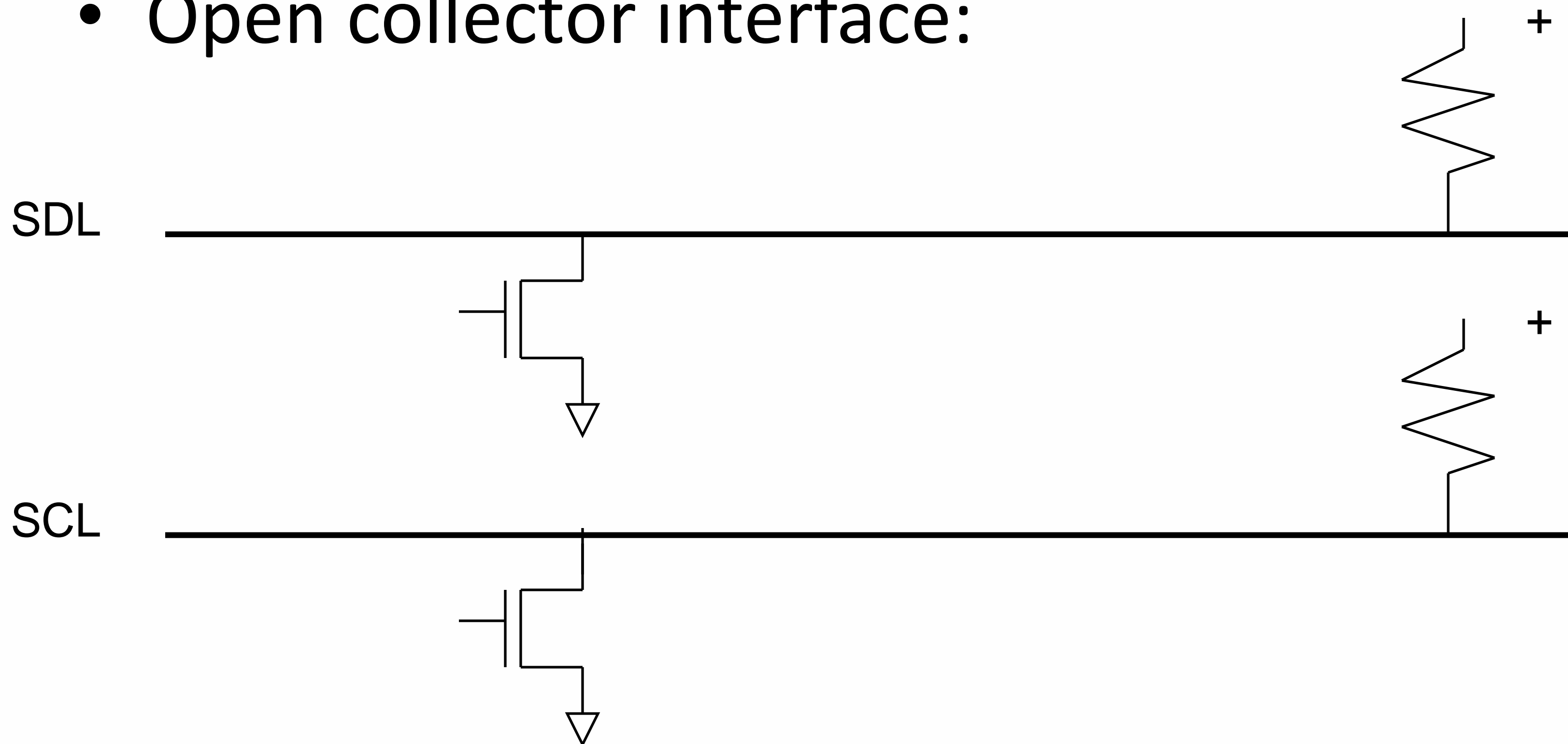


I^2C data format



I^2C electrical interface

- Open collector interface:



I^2C signaling

- Sender pulls down bus for 0.
- Sender listens to bus---if it tried to send a 1 and heard a 0, someone else is simultaneously transmitting.
- Transmissions occur in 8-bit bytes.

I^2C data link layer

- Every device has an address (7 bits in standard, 10 bits in extension).
  - Bit 8 of address signals read or write.
- General call address allows broadcast.

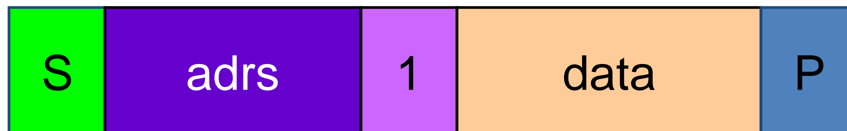
I^2C bus arbitration

- Sender listens while sending address.
- When sender hears a conflict, if its address is higher, it stops signaling.
- Low-priority senders relinquish control early enough in clock cycle to allow bit to be transmitted reliably.

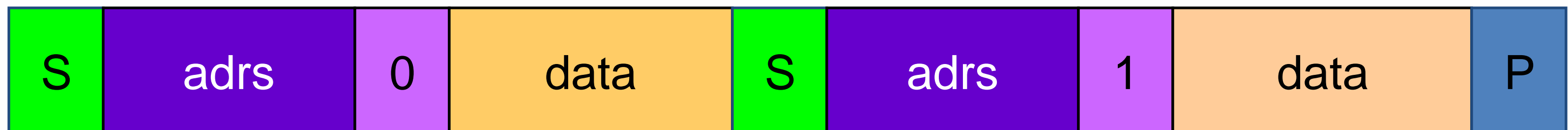
I^2C transmissions

multi-byte write

read from slave

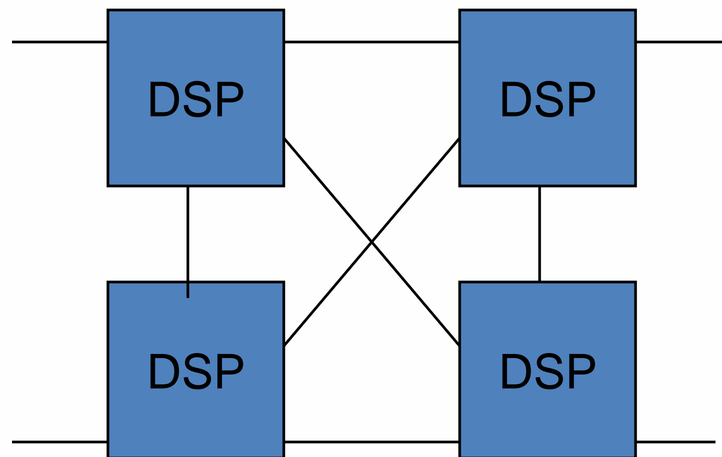


write, then read



Multiprocessor networks

- Multiple DSPs are often connected by high-speed networks for signal processing:



SHARC link ports

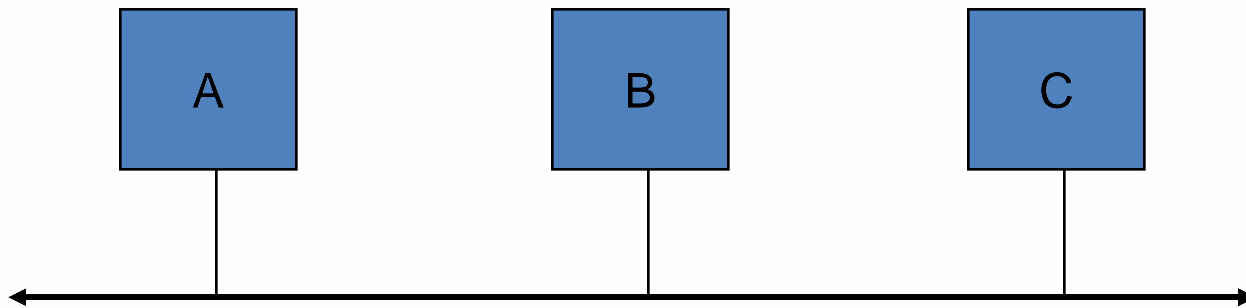
- Six per CPU.
- Four bits per link port.
- Packets have 32- or 48-bit payload.
- Can be controlled by DMA.
- Are half-duplex---must be turned around by program.

Ethernet

- Dominant non-telephone LAN.
- Versions: 10 Mb/s, 100 Mb/s, 1 Gb/s 10 Gb/s.
- Goal: reliable communication over an unreliable medium.

Ethernet topology

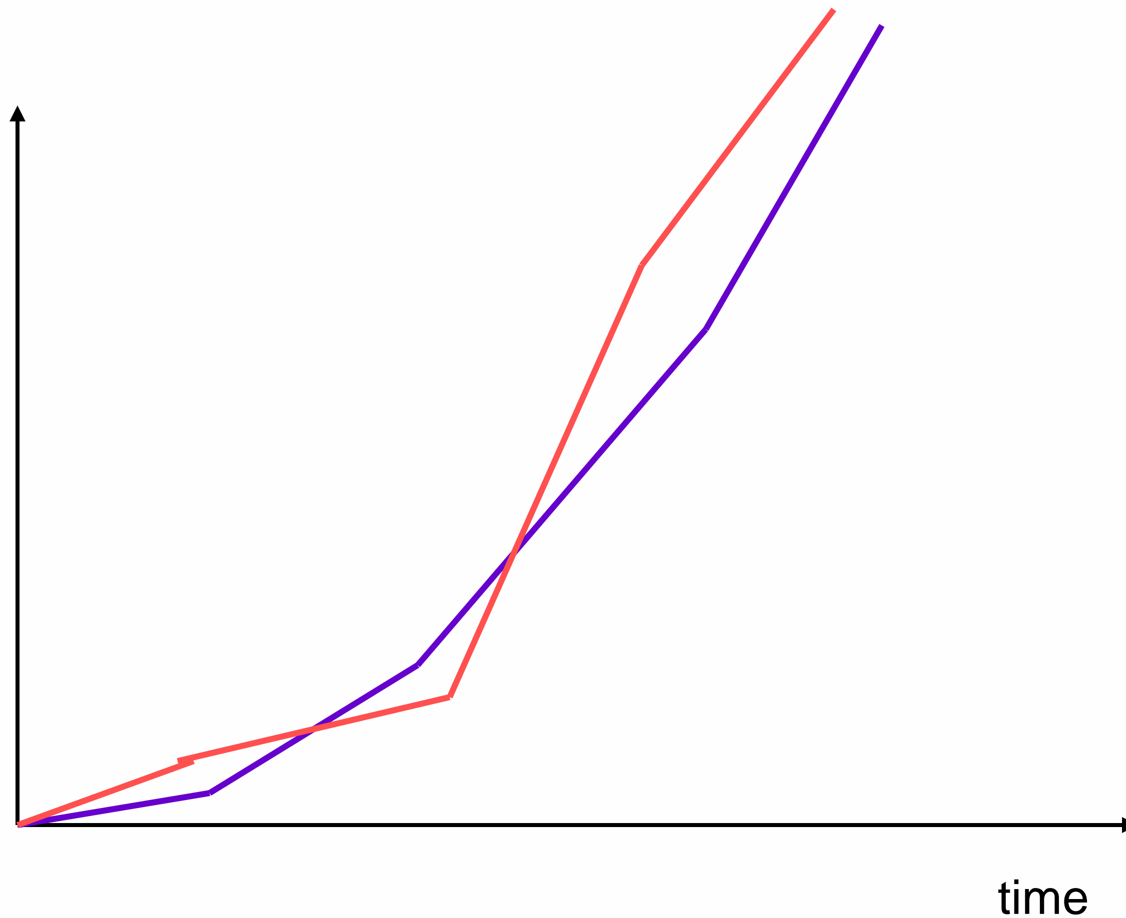
- Bus-based system, several possible physical layers:



CSMA/CD

- Carrier sense multiple access with collision detection:
  - sense collisions;
  - exponentially back off in time;
  - retransmit.

Exponential back-off times



Ethernet packet format

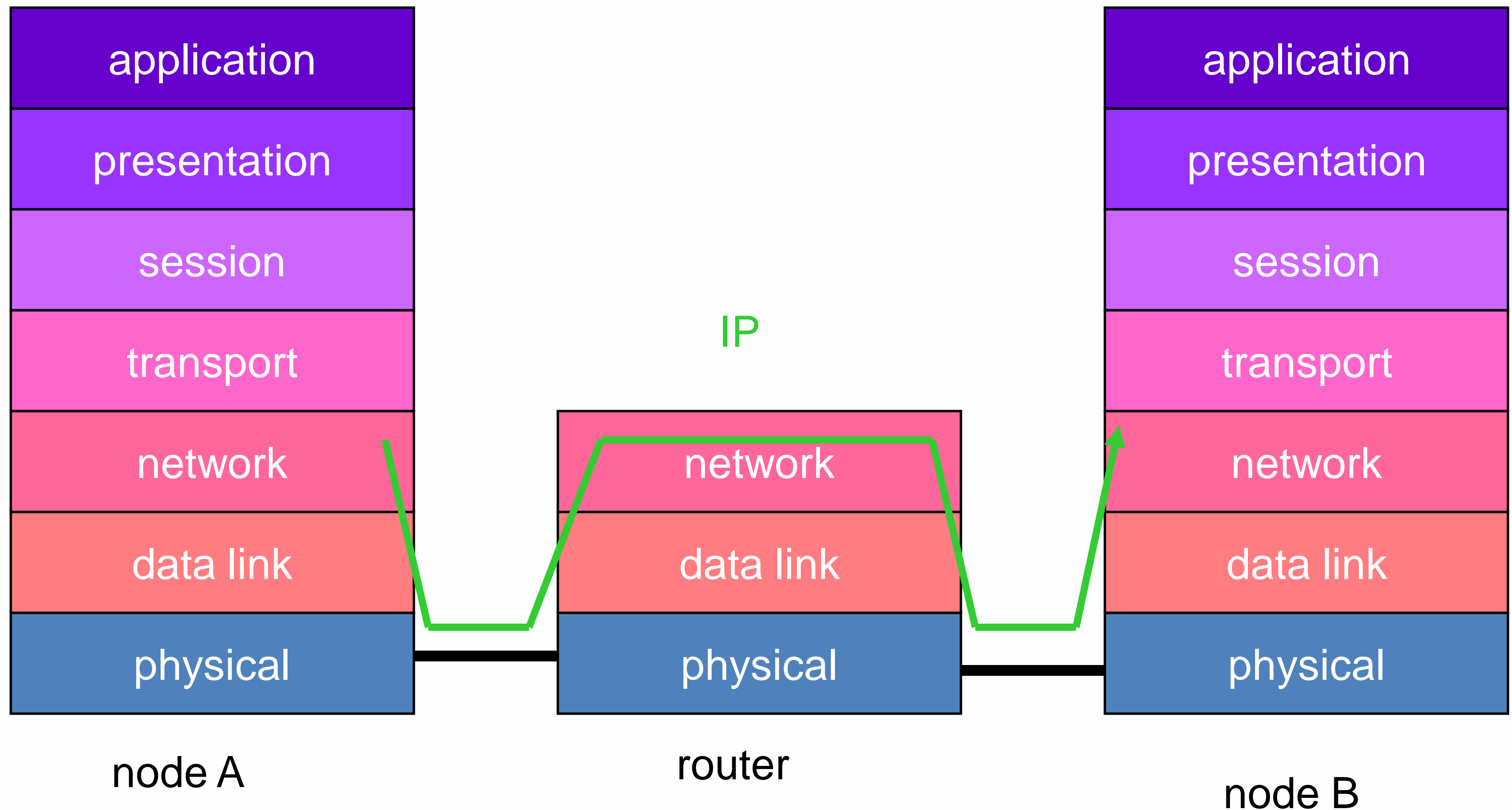
Ethernet performance

- Quality-of-service tends to non-linearly decrease at high load levels.
- Can’t guarantee real-time deadlines. However, may provide very good service at proper load levels.

Internet Protocol

- Internet Protocol (IP) is basis for Internet.
- Provides an internetworking standard: between two Ethernets, Ethernet and token ring, etc.
- Higher-level services are built on top of IP.

IP in communication



IP packet

- Includes:
  - version, service type, length
  - time to live, protocol
  - source and destination address
  - data payload
- Maximum data payload is 65,535 bytes.

IP addresses

- 32 bits in early IP, 128 bits in IPv6.
- Typically written in form `xxx.xx.xx.xx`.
- Names (foo.baz.com) translated to IP address by domain name server (DNS).

Internet routing

- Best effort routing:
  - doesn't guarantee data delivery at IP layer.
- Routing can vary:
  - session to session;
  - packet to packet.

Higher-level Internet services

- Transmission Control Protocol (TCP) provides connection-oriented service.
- Quality-of-service (QoS) guaranteed services are under development.

The Internet service stack