Embedded Systems

Basics of Embedded Systems

98% of microprocessors are in embedded systems

Microprocessor

- Controls non-computer device
  - Appliance, machine, toy, gadget, ...
- Used instead of many integrated circuits
- Processors not directly accessible to user
- Processor runs fixed code **(firmware)**

General code structure

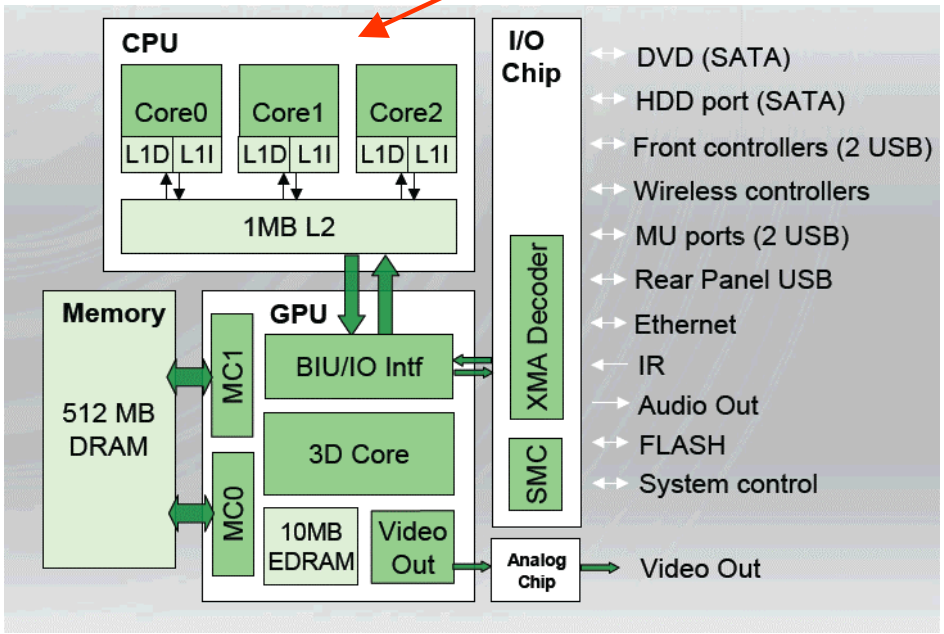
- Initialization routines
- Main loop
  - Polls devices that may require attention
  - Responds to interrupts from devices that require immediate attention
  - Continues at start of loop
- Shutdown routines
- Handler routines for events

Embedded Systems

XBOX 360 System Block Diagram

©2005 Microsoft Corporation

Triple-core 3.2 GHz custom CPU

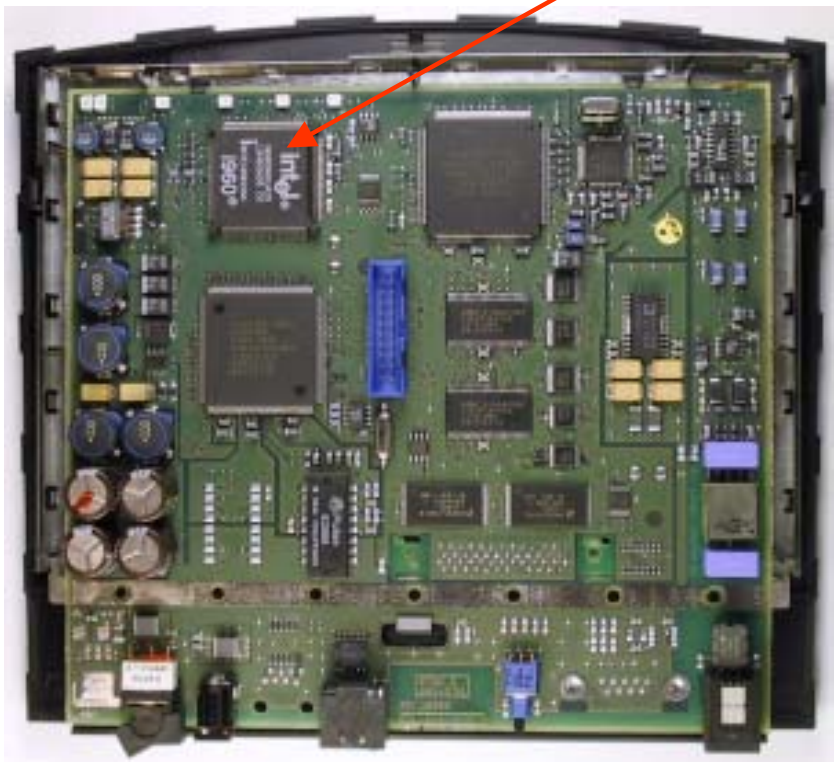


Alcatel Speed Touch™ Home

ADSL-modem

Intel i960®

RISC-based microprocessor



Simple Example

Controller for **wireless mouse**

- Mouse contains optical transceiver
- Transmits to USB adapter on PC

Mouse is one of several devices

- Identifies itself to PC with **device ID**

Mouse powered by battery

- **Sleeps** after 5 seconds idle
- Maintains state
- **Wakes up** on motion or button click

Wireless Mouse Hardware

Switches (for buttons)

Motion sensors (for X-Y position)

Microprocessor

Translates switch and sensor outputs to **motion data**

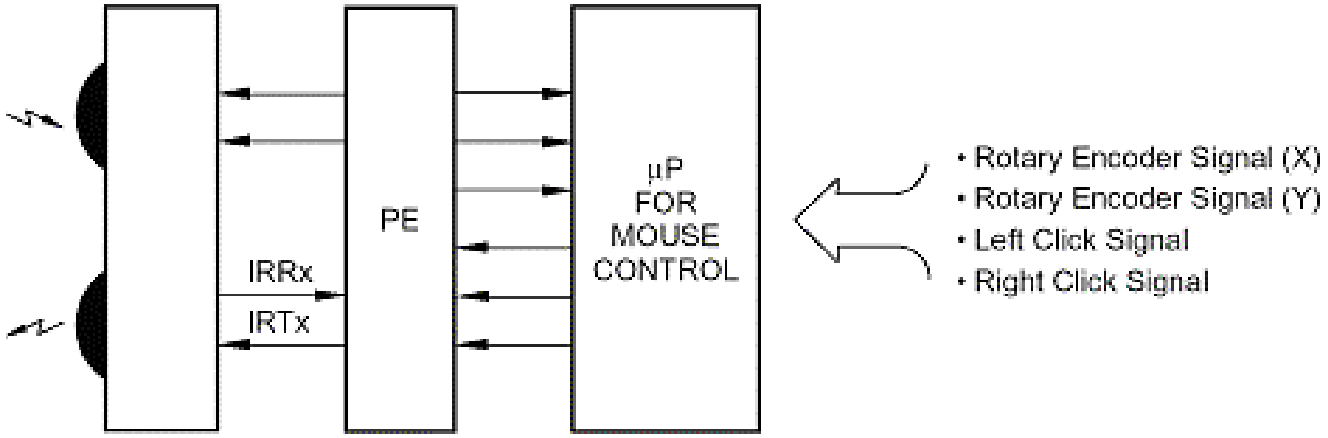
Manages **battery usage** and **mouse status**

Peripheral Engine (PE)

Controls optical link

Stores device information, button status, and X,Y-position

Optical Transceiver (Transmitter / Receiver)



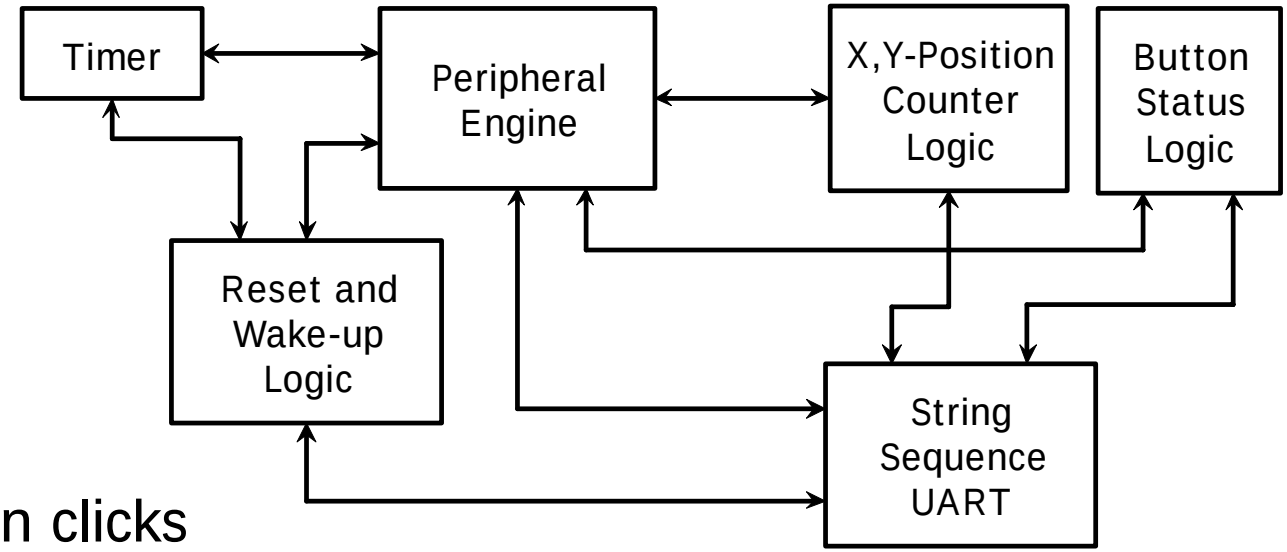
Mouse Routine (1)

```
        ; initialization
init:   request ID from PC                  ; send stored request string
        describe hardware to PC             ; send stored request string
        store X-Y position                  ; in memory buffer
        enable reset interrupt              ; use NMI input
        enable wake-up interrupt            ; use NMI input
        reset 5 second timer                ; PC clock: 18 ticks / second
                                            ; CX ← 1518h = 5400₁₀ = 18×60×5
                                            ; each loop: SUB CX, loop runtime
                                            ; CX == 0 ⇒ 5 second timeout
        ; main loop
L1:     read button status
        cmp button status, stored status    ; Peripheral Engine (PE) encodes
        JE L2                               ; status of 3 buttons and
        CALL button                         ; sends button status to PC
L2:     read motion detectors
        calculate X,Y-position
        CMP position, stored position       ; Peripheral Engine (PE) encodes
        JE L3                               ; X,Y-position and
        CALL motion                         ; sends position to PC
L3:     SUB timer, loop runtime             ; decrease timer
        CMP timer,0
        JE sleep                            ; if 5 seconds passed, sleep
        JMP L1                              ; continue
        ; end main loop
```

Mouse Routine (2)

```
            ; button press handler
button:     store button status in PE
            instruct PE to send button status to PC
            reset 5 second timer
            RET
            ; motion handler
motion:     store position in PE
            instruct PE to send X,Y-position to PC
            reset 5 second timer
            RET
            ; sleep state
sleep:      lock PE registers           ; store state
            turn-off battery to PE and transceiver
wait:       JMP wait                    ; low power do-nothing loop
                                        ; stays here until interrupt
            ; wake-up ISR
WAKE:       unlock PE registers         ; wake up
            JMP L1                      ; continue
            ; reset ISR
reset:      JMP init                    ; start
```

The Logic Gate Alternative



X,Y-position logic

Counts mouse motion clicks

Adjusts stored value in PE memory

Button status logic

Encodes button presses to PE

Reset and wake-up logic

Initializes and wakes up sleeping mouse

Puts idle mouse to sleep

String sequence logic

Transmits code strings to PC as bits

Gate Implementations

Individual ICs

Large “footprint”

Costly manufacture

Custom IC

Costly development

Embedded Hardware Environment

Microprocessor or microcontroller

Small RAM

Firmware ROM

Other storage (disk or flash memory)

Timers and interrupt controller

Sensors

- Position, touch, temperature, pressure, light, ...

Actuators

- Displays, motors, solenoids, relays, ...

Microcontroller

Single-chip embedded controller

Usually contains:

- Microprocessor
- RAM
- ROM
- Timers
- Interrupt controller

May also contain:

- Analog to digital converters
- Digital system processor (DSP)
- UART

Examples of I/O: VCR

Sensors in a VCR

- User interface
  - Switches, push buttons, touch-screen, tracking knob, ...
- Motion detectors
  - Tape inserted, tape ejected, beginning of tape, end of tape, ...
- Safety detectors
  - Dew detector, thermostat, over-voltage, no-signal, ...

Actuators in a VCR

- Tape drive motor, insert/eject motor
- Gear control solenoids

Motor turns
Solenoid pushes or pulls

Embedded Software Environment

Strict performance goals

I/O-oriented low level system programming

Hardware-specific code requirements

- Programmer must understand hardware operations
- Few opportunities for portable or reusable code

Event-driven system

- System handles multiple independent external events

Heavy use of control theory

Complex debugging issues

Goals in Embedded Design

Reliability

- Device operation depends on embedded processor
- A bug can be inconvenient, expensive, and possibly life-threatening
- Device must work 24/7/365 without reset

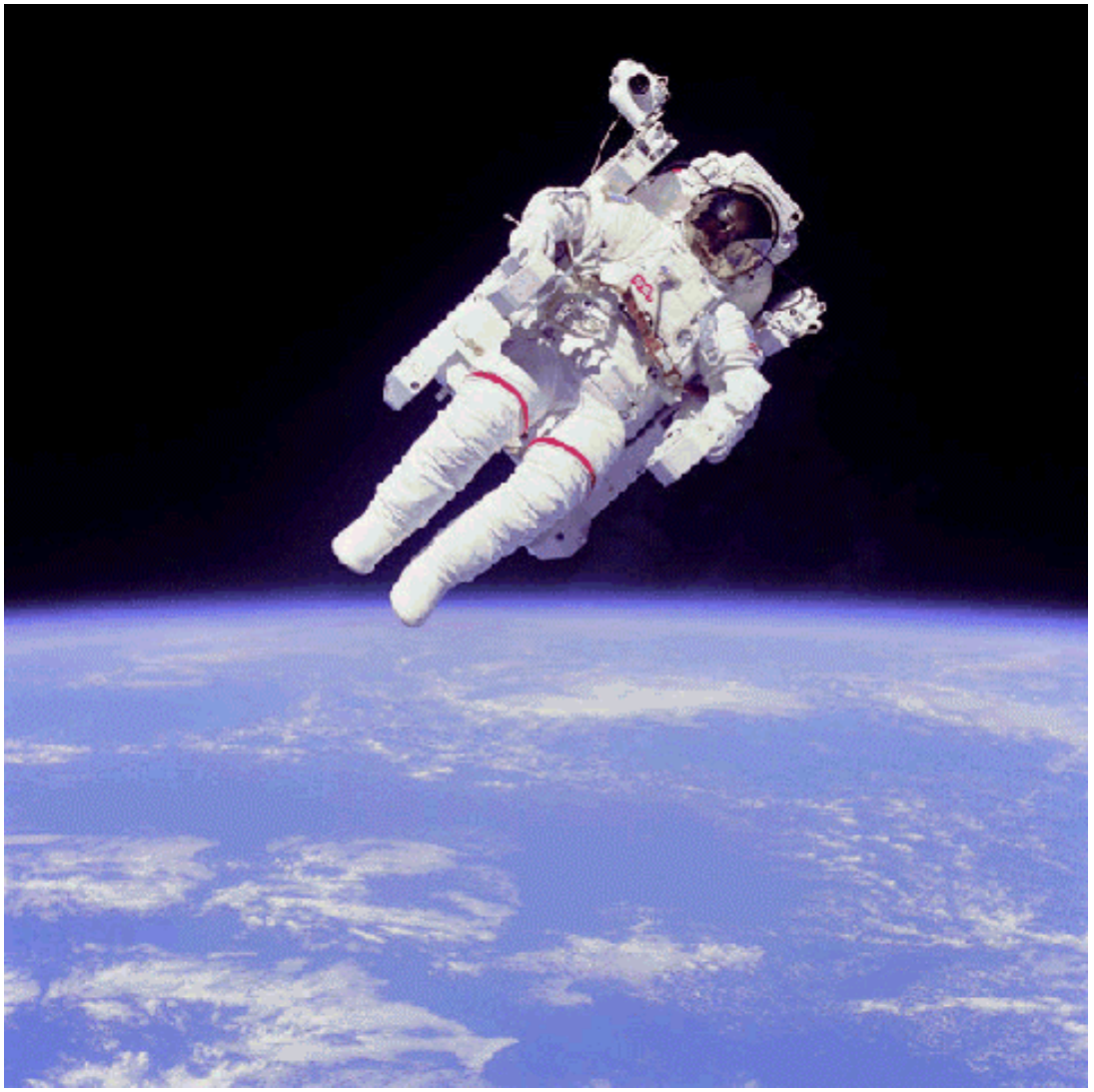
Performance

- Real time system
  - Device must respond within fixed time limits
  - Satisfy real world timing constraints
- Scheduling
- Optimized I/O

Cost

- Minimize manufacturing cost for consumer market
- Fast Time to Market
- No opportunity for future modification

Programmer's Requirements Nightmare



NASA Photo ID: S84-27017 (February 11, 1984)

Multitasking and Concurrency

System must handle **multiple independent events**

- Inputs and external processes
- Outputs and internally generated processes
- Events may occur simultaneously

Multitasking

- Switching between routines that handle different events

Concurrency

- System appears to handle multiple tasks simultaneously

Example of Concurrency

Programmable Thermostat

Three concurrent tasks

```
    /* Monitor Temperature */
do forever {
    measure temp ;
    if (temp < setting)
        start furnace ;
    else if (temp > setting + delta)
        stop furnace ;
}
```

```
    /* Monitor Time of Day */
do forever {
    measure time ;
    if (6:00am)
        setting = 72ºF ;
    else if (11:00pm)
        setting = 60ºF ;
}
```

```
    /* Monitor Keypad */
do forever {
    check keypad ;
    if (raise temp)
        setting++ ;
    else if (lower temp)
        setting-- ;
}
```

Ref: Daniel W. Lewis, **Fundamentals of Embedded Software: Where C and Assembly Meet**

Methods in Embedded Systems

Sequential Programming Model

Main loop handles each system component in turn

Possibility of interrupts for critical components

Finite State Machine

Define all possible states of the system

Define every legal transition from state to state

Concurrent Process Model

Two or more tasks running concurrently (multitasking)

Tasks exchange data among themselves

Real time constraints limit running time for each task

Object Oriented Model

Most embedded systems use a **combination** of models

Debugging

No bugs allowed

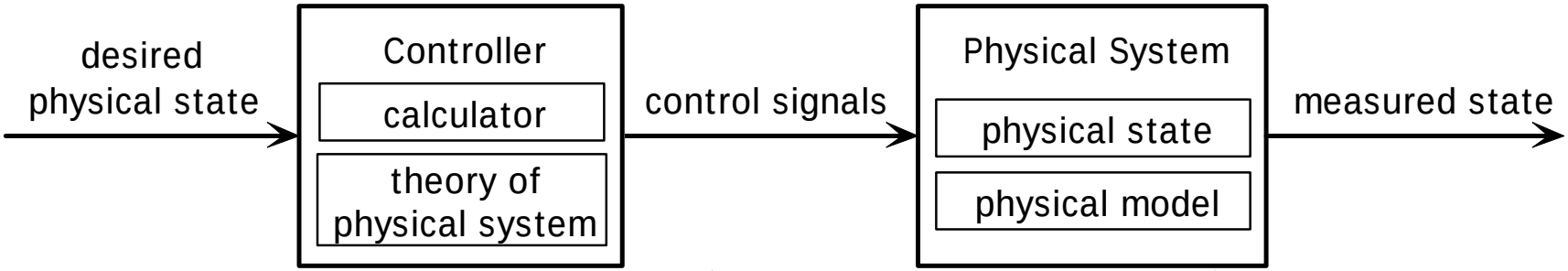
- High performance demands
- No human operator to perform “workaround”
- No extended beta-testing (“service packs”)

Hardware/software debugging

- Hardware configuration is time consuming
- Programmer must understand
  - Hardware at depth that enables debugging
  - Hardware/software interactions
  - All possible external situations
  - Interaction of C code and assembly code
  - Compiler choices
- Debugging usually involves emulation of target system

Basics of Control Systems

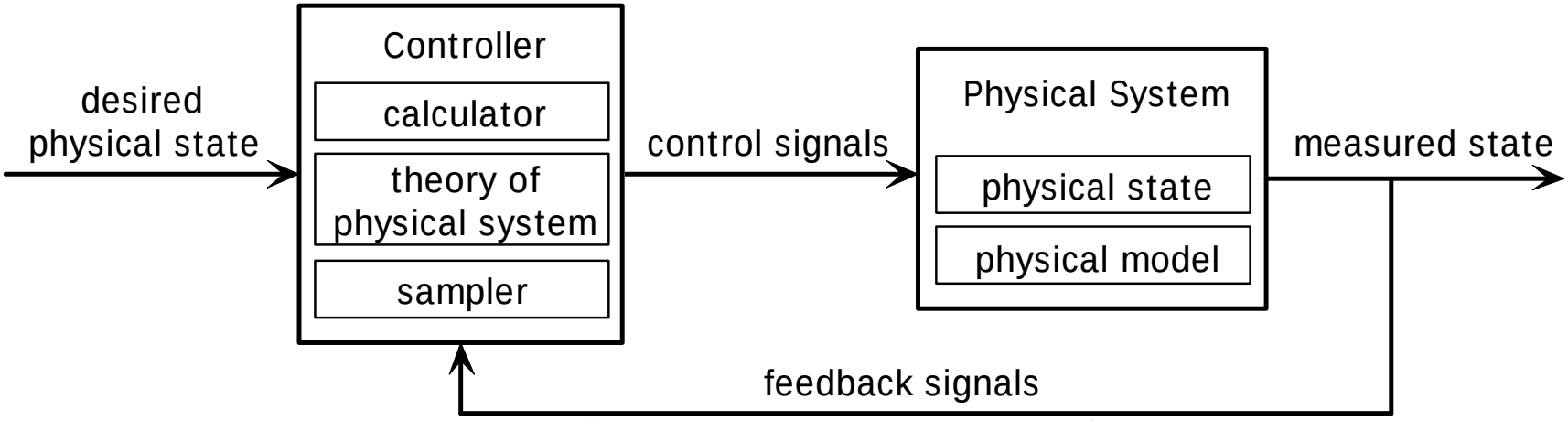
Open Loop Control



$$state(t_{n+1}) = F(state(t_n), control(t_n))$$

$$control(t_{n+1}) = G(desired\ state(t_n))$$

Closed Loop Control

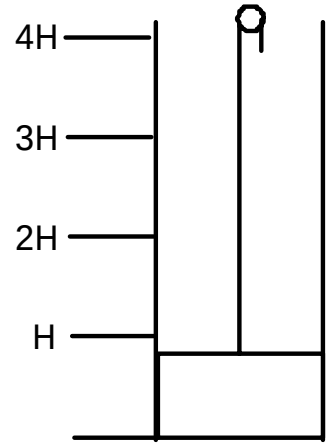


$$state(t_{n+1}) = F(state(t_n), control(t_n))$$

$$control(t_{n+1}) = G(desired\ state(t_n), measured\ state(t_n))$$

Elevator: Open Loop vs. Closed Loop

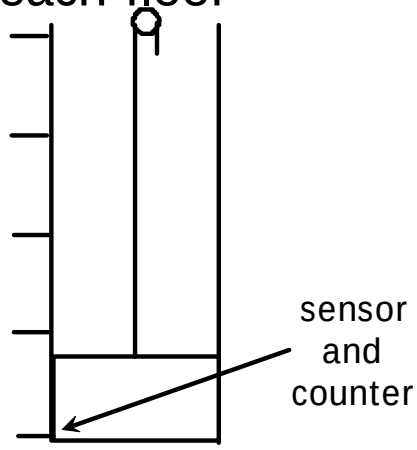
Open loop elevator control



- Elevator model
  - **N** floors in building
  - **H** meters per floor
- Controller model
  - Initialize at ground floor (**`height = 0`**)
  - Rise **H** × **N** meters to reach floor **N**

Closed loop elevator control

- Elevator model
  - **N** floors in building
  - Sensor activates when elevator positioned correctly at each floor
- Controller model
  - Initialize at ground floor (**`floor_counter = 0`**)
  - Count sensor signals at each floor
  - Count **N** sensor signals to reach floor **N**
  - Slow elevator when **`floor_counter = N – 1`**
  - Stop elevator precisely when **`floor_counter = N`**



Embedded Elevator Controller

Requirements

- Goes up or down
  - When called from a floor (external request)
  - When passenger requests (internal request)
- Remembers all floor requests
  - Distinguishes up requests and down requests
- Processes floor requests in order
  - Stops for internal requests in current direction
  - Stops at external up requests when going up
  - Stops at external down requests when going down
- Opens door at each floor
- Closes door after delay if passage is clear
- Announces floor (in case you fall asleep during ride)

State Machine

Elevator States

- Up — elevator is going up
- Down — elevator is going down
- Door — door is open
- Idle — no motion and no pending requests

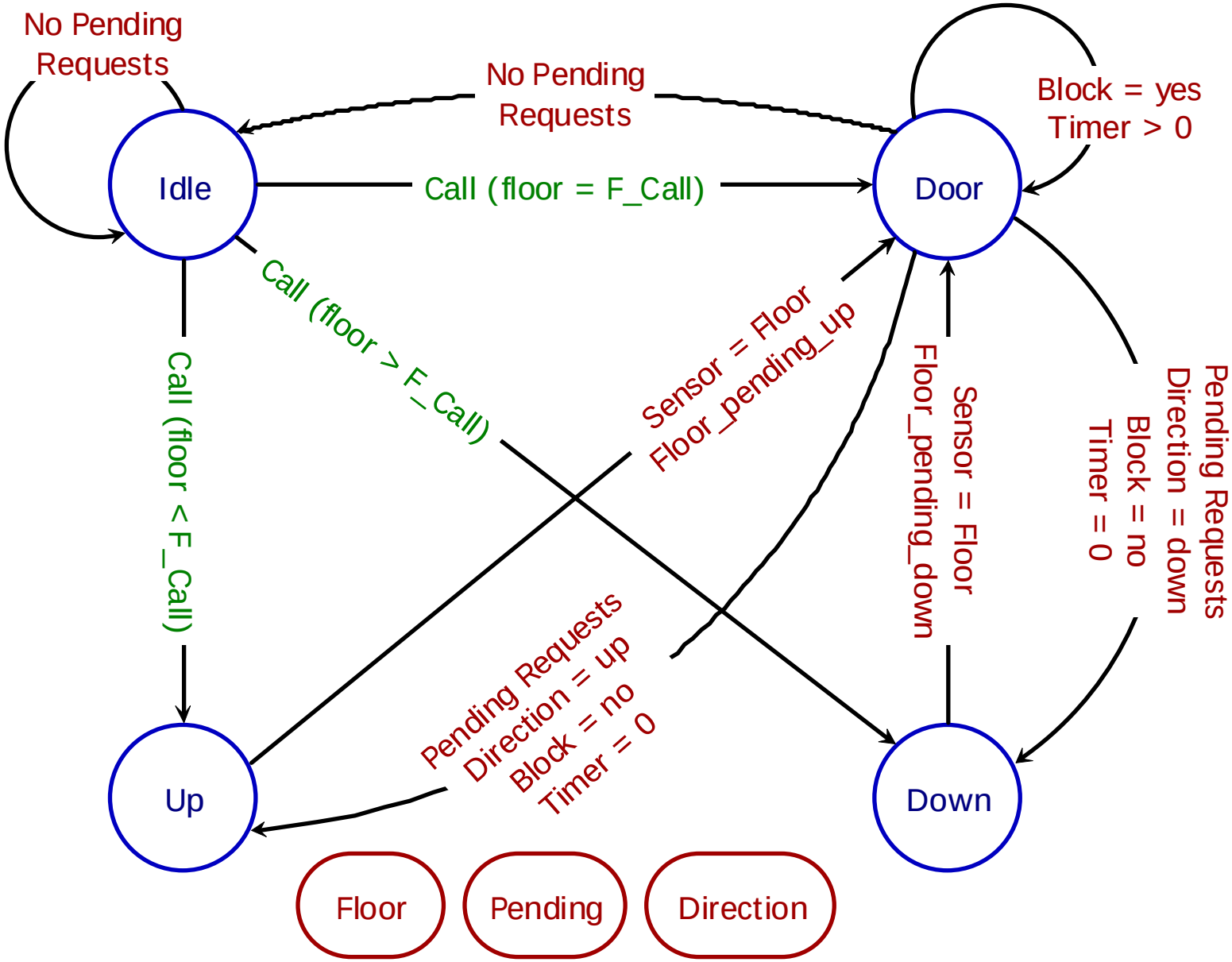
I/O Events

- Pass — (internal request) passenger requests floor F_Pass
- Call — (external request) elevator called
  - From floor F_Call
  - For direction (up/down) D_Call
- Block — passenger in elevator doorway

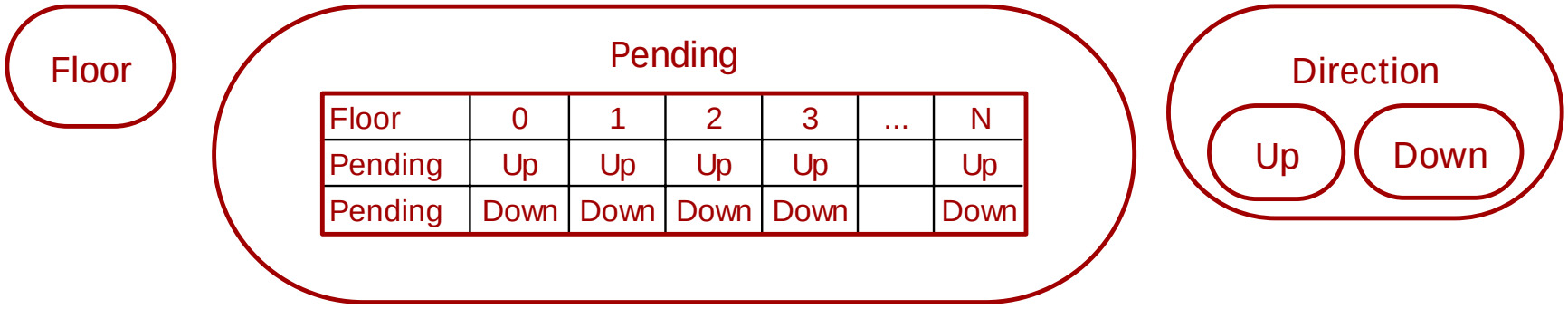
Internal state (memory)

- Floor — current elevator location
- Direction — elevator on up cycle or down cycle
- Pending — table of stored floor and direction requests

Elevator State Diagram



Sequential Model for Internal State



Floor	0	1	2	3	...	N
Pending	Up	Up	Up	Up		Up
Pending	Down	Down	Down	Down		Down

Elevator **not** idle

Pass (internal request)

Passenger requests floor F_Pass

Floor_pending_up (F_Pass) ← 1 if (F_Pass > Floor)

Floor_pending_down (F_Pass) ← 1 if (F_Pass < Floor)

Call (external request)

Elevator called from floor F_Call for direction D_Call

Floor_pending_up (F_ Call) ← 1 if (D_Call = Up)

Floor_pending_down (F_ Call) ← 1 if (D_Call = Down)

Controller I/O (1)

Elevator position (sensors)

Ground floor marked by limit switch (fl_0)

Upper floors marked by simple switch (fl_1)

Elevator car uses switches to determine stopping points

Controller reads ports **`fl_0`** and **`fl_1`**

Door closed (sensor)

Controller reads door closed at ports **`dr_open`**

Door blocked (sensor)

Controller reads door blocked at ports **`dr_blk`**

Controller I/O (2)

Call buttons (sensors)

Ground floor has Up call button

Top floor has Down call button

Middle floors have Up and Down call buttons

Controller reads call buttons as

ports **`up_0 ... up_N`** (sequential port numbers)

ports **`down_0 ... down_N`** (sequential port numbers)

Floor buttons (sensors)

Elevator car has floor buttons for passenger choice

Controller reads floor buttons as

ports **`pass_0 ... pass_N`** (sequential port numbers)

Elevator motors (actuators)

Car up / down — **`mot_up, mot_dn`**

Door open / close — **`mot_dr`**

Code Outline

```
defines and macros
initialize variables
start elevator on ground floor
main loop
{
    handle floor requests
    control up and down
    open and close door
}
functions
```

Outline of Main Loop

```
main loop{
    read call and passenger buttons
    if (idle) respond to pending calls
    if (door open) close door
    if (going down){
       if (reaching floor) {update floor, stop, open
          door, close door
          if (pending down) continue down
       }
    }
    if (going up){
       if (reaching floor) {update floor, stop, open
          door, close door
          if (pending up) continue up
       }
}
```

Controller Code (1)

```
; macros and defines
     fl_0      EQU   ground floor sensor I/O port
     fl_1      EQU   higher floor sensor I/O port
     up_1      EQU   up call button I/O sensor on floor 0
     .
     .
     .
     up_N      EQU   up call button I/O sensor on floor N
     dn_1      EQU   down call button I/O sensor on floor 0
     .
     .
     .
     dn_N      EQU   down call button I/O sensor on floor N
     pass_0    EQU   passenger floor 0 button I/O sensor
     .
     .
     .
     pass_N    EQU   passenger floor N button I/O sensor
     dr_open   EQN   door open sensor I/O port
     dr_blk    EQN   door blocked sensor I/O port
     mot_up    EQN   up motor I/O port
     mot_dn    EQN   down motor I/O port
     mot_dr    EQN   door motor I/O port
     delta     EQN   timer interval for door
```

Controller Code (2)

```
section data
        pending_up      dw   0          ; bit map for pending tables
        pending_dn      dw   0          ; assume 15 floors
        floor           db   0
        state           db   0          ; bit map for Idle (4), Door (3),
                                        ; Down (2), Up (1), Direction (0)
        timer           dw   540        ; 30 second counter
section text
; initialization
        in al,dr_open                   ; read door state
        cmp al,0                        ; is door open
        jne L1                          ; if door closed, continue
        call close_door
L1:     in al,fl_0                      ; AL <-- ground floor sensor
        cmp al,1
        je L2                           ; if on ground floor, continue
        call find_ground                ; else call routine
L2:     mov [state],00010000b           ; state <-- idle
        mov [floor],0                   ; set ground floor
        mov [pending_up],0              ; clear
        mov [pending_dn],0              ; clear
```

Controller Code (3)

```
; MAIN LOOP
M1:   call update_pending
; check for idle state
      cmp [state],00010000b   ; is idle bit set
      jne M2                  ; if not, check door
      mov cx,[floor]          ; CX <-- this floor
      mov bp,00000001b        ; mask for floor 0
      shl bp,cl               ; mask for this floor
I1:   cmp [pending_up],0      ; check pending up calls
      je I2                   ; no up calls, check down
      test [pending_up],bp    ; check pending up call for this floor
      jne I11                 ; not this floor, go up
      call open_door          ; if call from this floor
      jmp M1                  ; start again
I11:  call going_up
      jmp M1                  ; start again
I2:   cmp [pending_dn],0      ; check pending down calls
      je M1                   ; no down calls, start over
      test [pending_dn],bp    ; check pending down call for this floor
      jne I21                 ; not this floor, go down
      call open_door          ; if call from this floor
      jmp M1                  ; start again
I21:  call going_dn
      jmp M1                  ; start again
```

Controller Code (4)

```
; check for door open state
M2:    cmp [state],00001000b          ; check door open
       jne M3                         ; if not, check GoingDown
       call close_door                ; closes door, subject to timer
       jmp M1
```

Controller Code (5)

```
; check for going down state
M3:   cmp [state],00000100b     ; GoingDown
      jne M4                    ; if not, check GoingUp
      in al,fl_1                ; check if reaching a floor
      jne M1                    ; not reaching floor, start over
      dec [floor]               ; register new floor
      mov cx,[floor]            ; CX <-- this floor
      mov bp,00000001b          ; mask for floor 0
      shl bp,cl                 ; mask for this floor
      test [pending_dn],bp      ; check pending up call for this floor
      jne M1                    ; no stop on this floor, start over
      call stop_dn              ; stop elevator
      call open_door            ; open door
      call close_door           ; start timed close door cycle
      not bp                    ; invert floor mask
      and [pending_dn],bp       ; clear pending down call for this floor
      cmp [pending_dn],0        ; check pending down calls
      jne M31                   ; if more pending down calls, start down
      mov [state],00010000b     ; state <-- idle
      jmp M1                    ; start over
M31:  call going down           ; start down
      jmp M1                    ; start over
```

Controller Code (6)

```
; check for going up state
M4:   cmp [state],00000010b     ; GoingUp
      jne M1                    ; if not, loop again
      in al,fl_1                ; check if reaching a floor
      jne M1                    ; not reaching floor, start over
      inc [floor]               ; register new floor
      mov cx,[floor]            ; CX <-- this floor
      mov bp,00000001b          ; mask for floor 0
      shl bp,cl                 ; mask for this floor
      test [pending_dn],bp      ; check pending up call for this floor
      jne M1                    ; no stop on this floor, start over
      call stop_up              ; stop elevator
      call open_door            ; open door
      call close_door           ; start timed close door cycle
      not bp                    ; invert floor mask
      and [pending_up],bp       ; clear pending up call for this floor
      cmp [pending_up],0        ; check pending up calls
      jne M41                   ; if more pending up calls, start down
      mov [state],00010000b     ; state <-- idle
      jmp M1                    ; start over
M41:  call going_up             ; start up
      jmp M1                    ; start over
```

Controller Code (7)

```
find_ground:
        out mot_dn,1            ; going down
FL0:    in al,fl_0              ; ground floor sensor
        cmp al,1                ; check for ground floor
        jne FL0                 ; keep trying
        out mot_dn,0            ; stop
        ret

open_door:
        mov [timer],540         ; set timer
OP1:    out mot_dr,1            ; open door
        in al,dr_open           ; check that door actually opened
        cmp al,0                ; is door open
        je OP1                  ; try again
        and [state],00001000b   ; set open state
        ret
```

Controller Code (8)

```
close_door:
CL1:  cmp [timer],0              ; time-out
      je CL2
      sub [timer],delta          ; try again
      call update_pending        ; read buttons while waiting
      jmp CL1
CL2:  in al,dr_blk               ; read door state
CL3:  cmp al,1                   ; is door blocked
      je CL3                     ; if door blocked, try again
CL4:  out mot_dr,0               ; close door
      in al,dr_open              ; check that door actually closed
      cmp al,0                   ; is door open
      jne CL4                    ; try again
      and [state],11110111b      ; clear open state
      ret
```

Controller Code (9)

```
update_pending:
        mov dl,up_0             ; DL <-- port number of first up button
        call read_16            ; get 16 up button bits
        or [pending_up],si      ; add new calls to pending table
        mov dl,dn_0             ; DL <-- port number of first down button
        call read_16            ; get 16 up button bits
        or [pending_dn],si      ; add new calls to pending table
        mov dl,pass_0           ; DL <-- port number of first down button
        call read_16            ; get 16 up button bits
        call calc_up_dn         ; assign passenger calls to up and down
        or [pending_up],si      ; add passenger up calls to pending table
        or [pending_dn],di      ; add passenger down calls to pending table
        ret
read_16:
        xor si,si
        mov cx,16               ; loop counter
LU1:    in al,dl                ; read up button on floor 0
        and al,1                ; AL is 0 or 1
        or si,al                ; set condition bit by AL
        ror si,1                ; rotate right one bit
        inc dl                  ; point at next floor I/O port
        loop LU1                ; read next
        ret
```

Controller Code (10)

```
calc_up_dn:
        xor cx,cx           ; loop counter
        mov bp,1            ; BP <-- 0000000000000001  (bit mask)
LC1:    test si,bp          ; flags <-- SI AND BP
        jne LC2             ; no button push
        cmp cx,floor        ; higher or lower floor
        jg LC2              ; higher floor (leave SI unchanged)
        or di,bp            ; lower floor => set down register
LC2:    inc cx              ; next floor
        rol bp,1            ; shift bit mask left by 1
        cmp cx,16
        jl LC1              ; if cx < 16 continue
        xor si,di           ; clear up register bit if down bit is set
        ret
```

Controller Code (11)

```
going_up:
      or [state],00000010b    ; set state to going up
      out mot_up,1            ; start going up
      ret

going_dn:
      or [state],00000100b    ; set state to going down
      out mot_dn,1            ; start going down
      ret

stop_up:
      and [state],11111101b   ; clear going up state
      out mot_up,0            ; stop
      ret

stop_dn:
      and [state],11111011b   ; clear going down state
      out mot_dn,0            ; stop
      ret
```

Real Time Considerations

Real time requirements

Run time of critical code sections **< maximum response time**

Response time determined by physical system under control

Example in **elevator controller**

Call request update cycle < average passenger button press

Call requests updated at least every ½ second

Estimating 8086 run time

Intel manual specifies clock cycles required for each instruction

Rough estimate

Total clock cycles for all relevant code × seconds / cycle

Must count every pass in a loop as more instructions

Complication

Intel manual provides clock cycles after loading

Instructions are loaded in parallel to execution of loaded instructions

No instruction can be loaded during load and store operations

Checking Elevator Code

8086 runs at 5 MHz

clock cycle time = 1/5 MHz = 0.2 microseconds

Maximum cycle time = ½ second = 500 milliseconds

500 milliseconds / 0.2 microseconds = $(500 \times 10^{-3} / 0.2 \times 10^{-6})$

$= 2{,}500 \times 10^{3}$

$= 2.5 \times 10^{6}$

2.5 million instructions can run between checking buttons

Not a very strict real-time system

More Exact Estimate

```
update_pending:
        mov dl,up_0                 ;  2 cc
        call read_16                ; 15 cc + 55 cc (function run time)
        or [pending_up],si          ; 10 cc
        or dl,dn_0                  ;  3 cc
        call read_16                ; 15 cc + 55 cc (function run time)
        mov [pending_dn],si         ; 12 cc
        mov dl,pass_0               ;  2 cc
        call read_16                ; 15 cc + 55 cc (function run time)
        call calc_up_dn             ; 15 cc + 53 cc (function run time)
        or [pending_up],si          ; 10 cc
        or [pending_dn],di          ; 10 cc
        ret                         ; 16 cc
read_16:
        xor si,si                   ;  3 cc
        mov cx,16                   ;  2 cc
LU1:    in al,dl                    ; 14 cc
        and al,1                    ;  3 cc
        or si,al                    ;  3 cc
        ror si,1                    ;  2 cc
        inc si                      ;  3 cc
        cmp si,15                   ;  3 cc
        loop LU1                    ;  6 cc
        ret                         ; 16 cc
```

(on next slide)

55 cc = 11 microseconds

Does not include possible delay in loading next instructions during memory reads

With Conditional Branches

```
calc_up_dn:
        xor cx,cx           ;  3 cc
        mov bp,1            ;  2 cc
LC1:    test si,bp          ;  3 cc
        jne LC2             ;  4 cc not taken / 13 cc taken
        cmp cx,floor        ;  3 cc
        jg LC               ;  4 cc not taken / 13 cc taken
        or di,bp            ;  3 cc
LC2:    inc cx              ;  3 cc
        rol bp,1            ;  2 cc
        cmp cx,16           ;  3 cc
        jl LC1              ;  4 cc not taken / 13 cc taken
        xor si,di           ;  3 cc
        ret                 ; 16 cc
```

Cannot know if conditional branch will be **taken** or **not taken**

Depends on **external conditions**

Conservative estimate $\Rightarrow$ assume **worst case** (**13 cc**)

Total = **53 cc**

Total Time to Update Pending Table

```
  2 cc
 15 cc + 55 cc (function run time)
 10 cc
  3 cc
 15 cc + 55 cc (function run time)
 12 cc
  2 cc
 15 cc + 55 cc (function run time)
 15 cc + 53 cc (function run time)
 10 cc
 10 cc
 16 cc
343 cc
```

5 MHz 8086

343 cc × (0.2 microseconds / cc) ≈ 70 microseconds

Debugging Embedded Systems

Principal techniques

Run with breakpoints

- Checks control flow
- Example

```
setup
CMP src1, src2
JNE target
fall-through
-g=setup target fall-through
-t=setup 1
```

Trace

- Run one instruction at a time
- Check registers and other changes between instructions

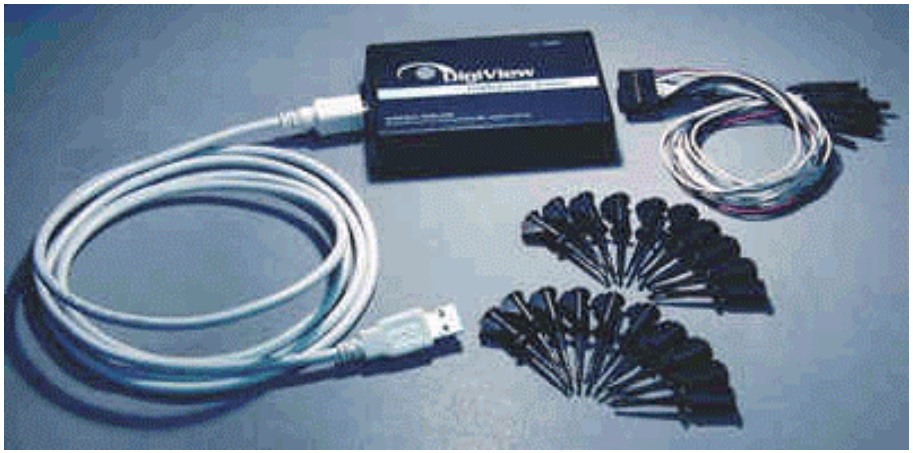
Logic analyzer

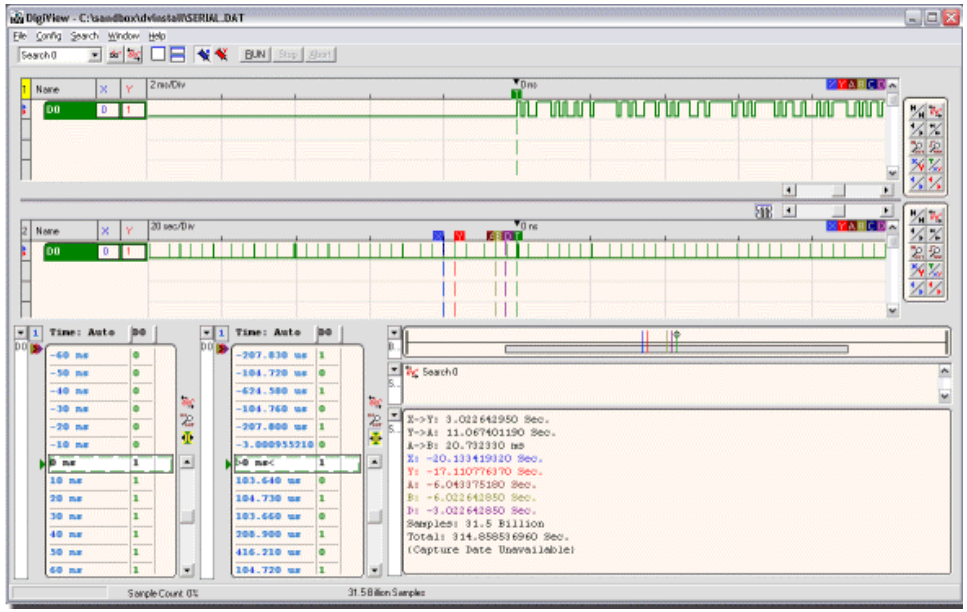
- Multiple electronic probes measure digital values in circuit
- Triggers signal capture on specified conditions
- Displays bit values of measured circuit point as function of time

In-Circuit Emulator (ICE)

- Replace processor with connector cable attached to workstation
- Emulator program controls processor outputs / measures inputs

PC-Based Logic Analyzer





Logic probe

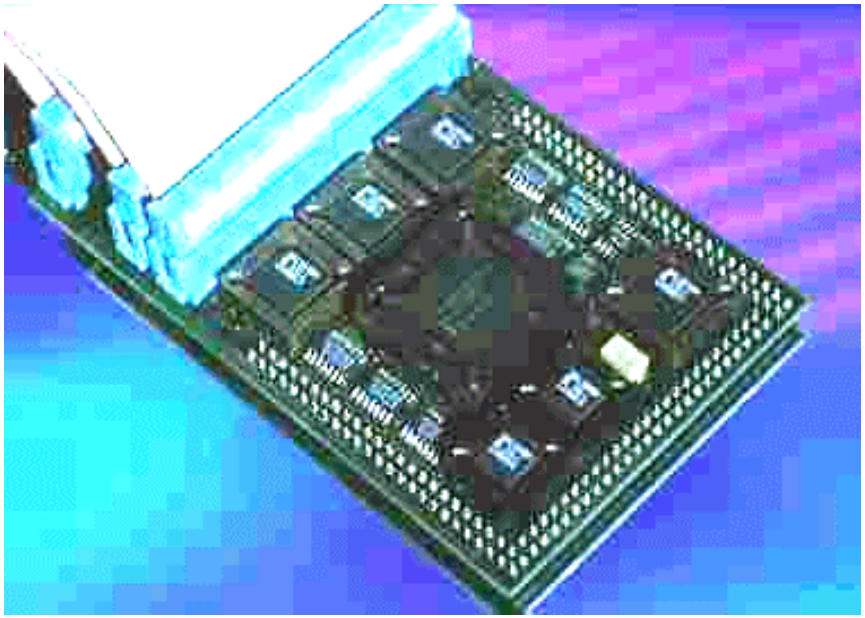
- 18 Channels
- 100 MHz measurement rate
- Trigger on transition of any channel
- USB interface

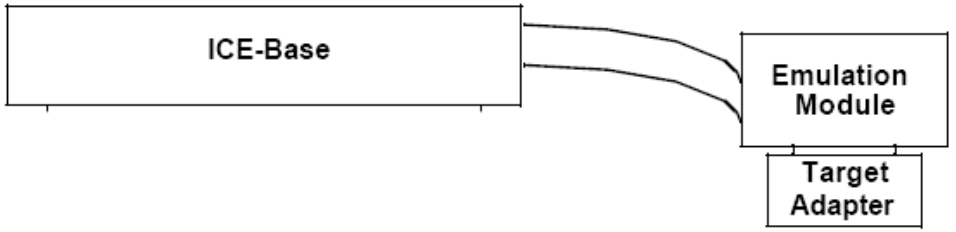
TechTools, Garland, Texas

PC software

- Configure trigger
- Graphical waveform display
- Zoom and Scroll
- Data Tables

In Circuit Emulator





Replace microprocessor chip with emulation adapter

- Adapter connects to target system
- Adapter can hold physical CPU

Can run program on physical CPU and measure input/output signals

Can run program on emulator

- Program runs in ICE-Base
- ICE controls I/O signals to target system (emulating CPU operation)

Alternatives to an Embedded Microprocessor

Choices for the Designer

Discrete digital systems

- Combinational logic gates
  - Boolean functions
  - AND, OR, NOT, XOR, ALUs, encoders, decoders
- Sequential logic gates
  - Memory functions
  - Flip-flops, registers, RAM, ROM, shift registers
- Suited to problems
  - With heavy I/O requirements
  - Inherently parallel operations

Microcontrollers and microprocessors

- Combinations of combinational and sequential logic
- Suited to problems
  - With reasonable I/O
  - Inherently sequential operations

Logic Gates in the 21st Century

Elevator is better suited to discrete logic than processor

- Large I/O domain
  - Floor and passenger buttons, up/down and door motors, floor sensors
- Operations are inherently parallel
  - Buttons feed registers
  - Operations are a basic state machine
  - Can be implemented with flip-flops and combinational logic

Why are elevator controllers built around **microprocessors**?

- More good programmers than good electrical engineers
- Discrete logic devices take up a lot of space
- Fewer parts $\Rightarrow$ smaller, easier manufacture, easier repair

The alternative

- **Gate arrays** and application specific integrated circuits (**ASIC**)
- Integrated circuits with complexity similar to microprocessor
- Contains large number of **programmable Boolean logic blocks**

Field Programmable Gate Arrays (FPGA)

Very Large Scale Integrated Circuit (VLSI) chip

- Similar in size and complexity to microprocessor
- Blocks of **Uncommitted Logic Arrays (ULA)**
- Block performs complex Boolean function based on programming

Program FPGA from workstation via graphic software

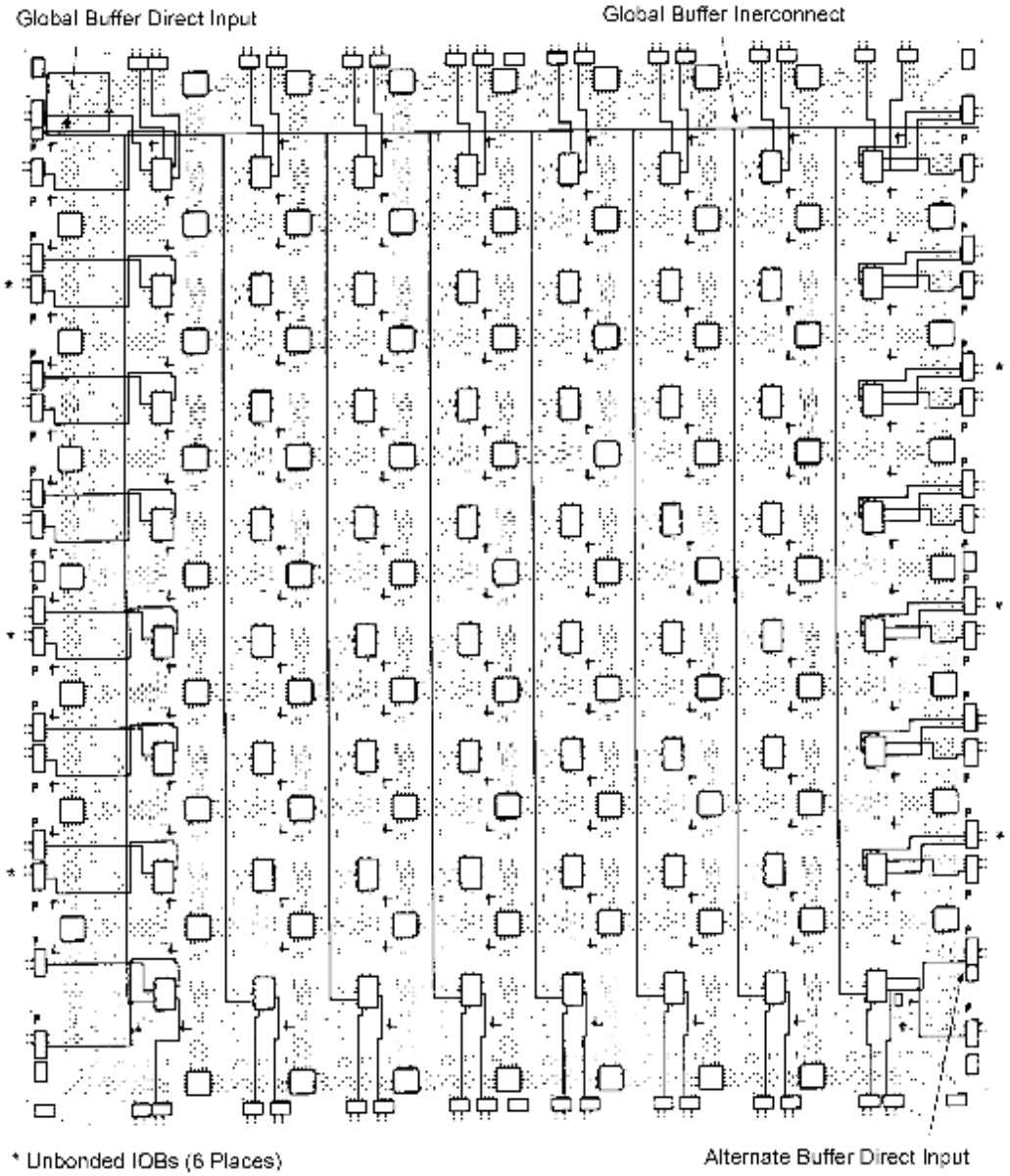
- Each ULA programmed to implement
  - Boolean function
  - Karnaugh map
  - High level hardware description language (**HDL**)
- Interconnections among ULAs programmed signal I/O

Older ULA block can form an output based on

- Any Boolean function of 4 – 8 logic inputs
- Clock signal
- D-flip/flop

Newer FPGA logic blocks may contain complex logic

Xilinx XC3000 Chip



Introduced in late 1980s

CLB

Configurable Logic Block

IOB

I/O Block

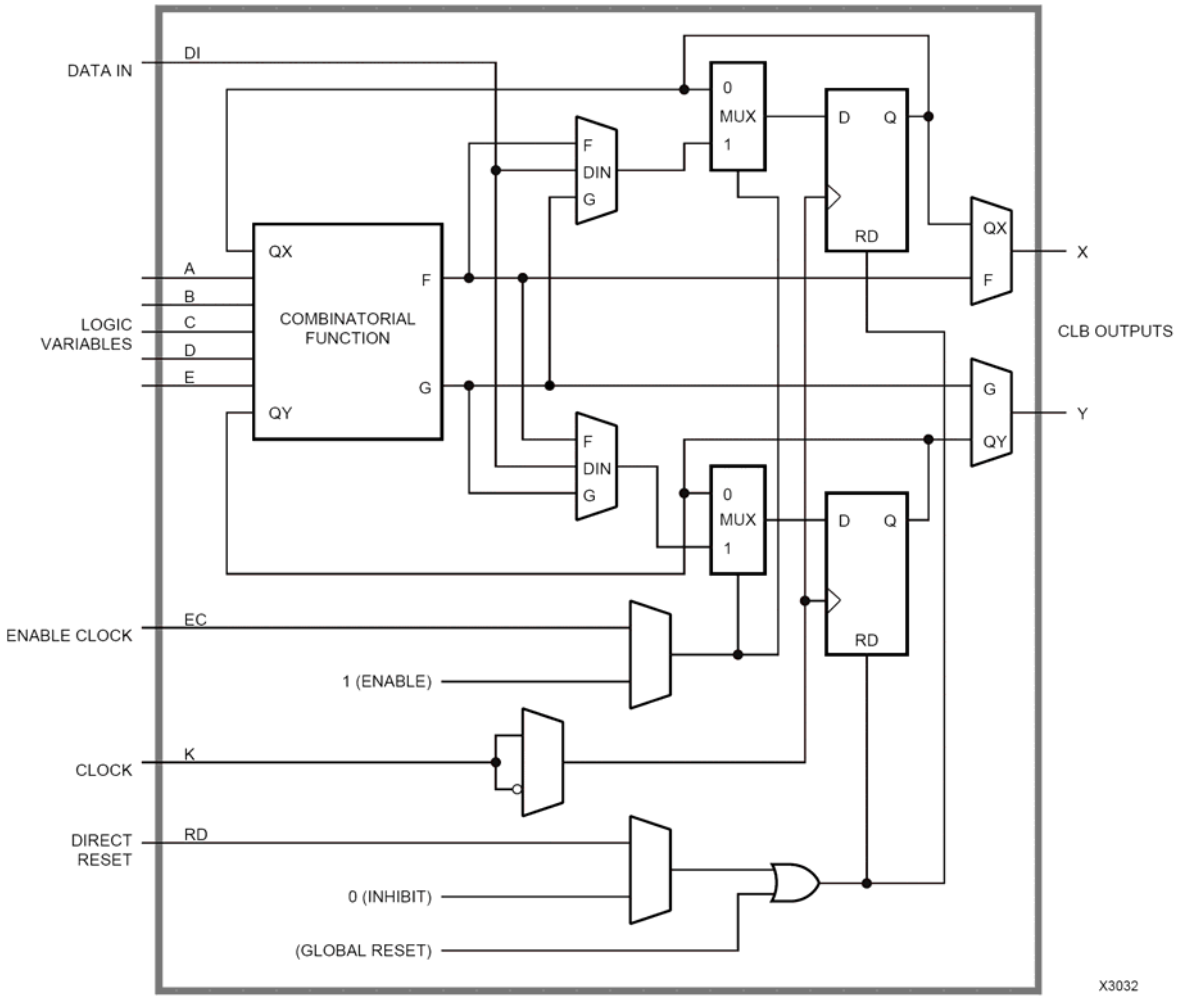
Buffer

Current amplifier

PIP

Programmable Interconnection Point

Xilinx Configurable Logic Block (ULA)



Configurable Logic Block (CLB)

Xilinx XC3000 FPGA

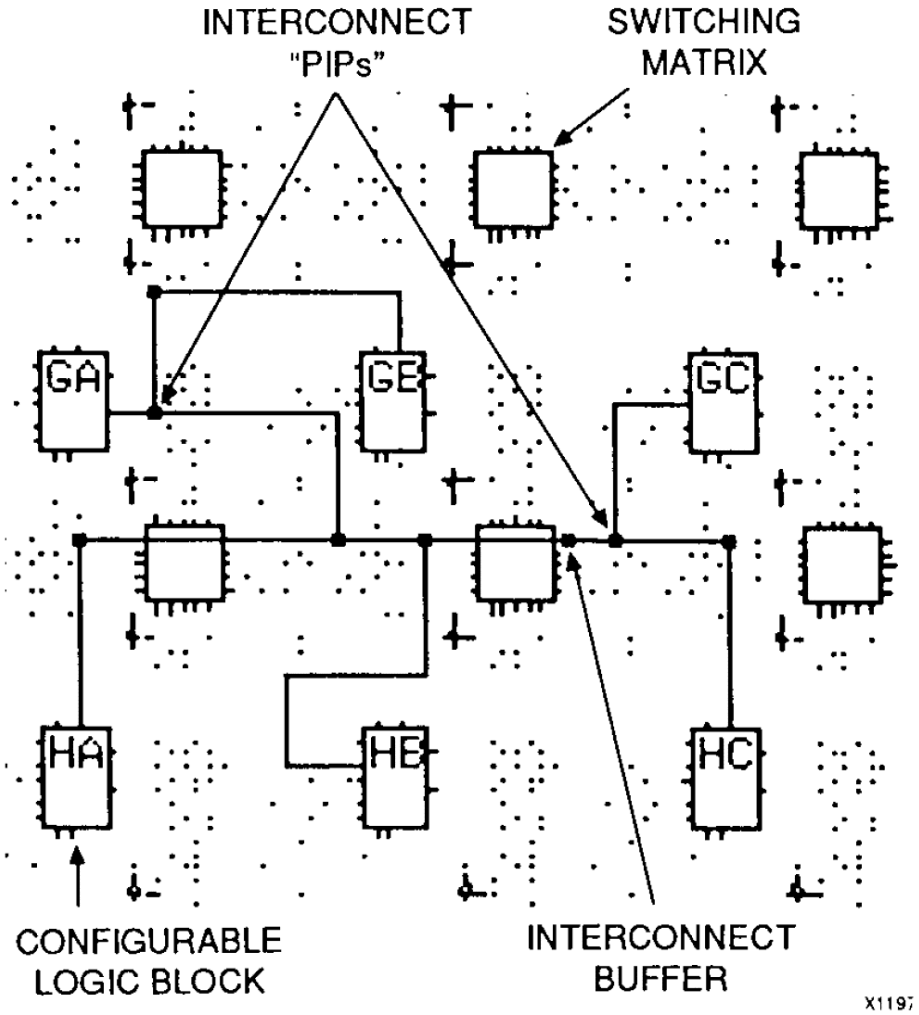
Combinatorial logic section

- 5 inputs
- 2 output functions

Two flip-flops

Programmable multiplexer

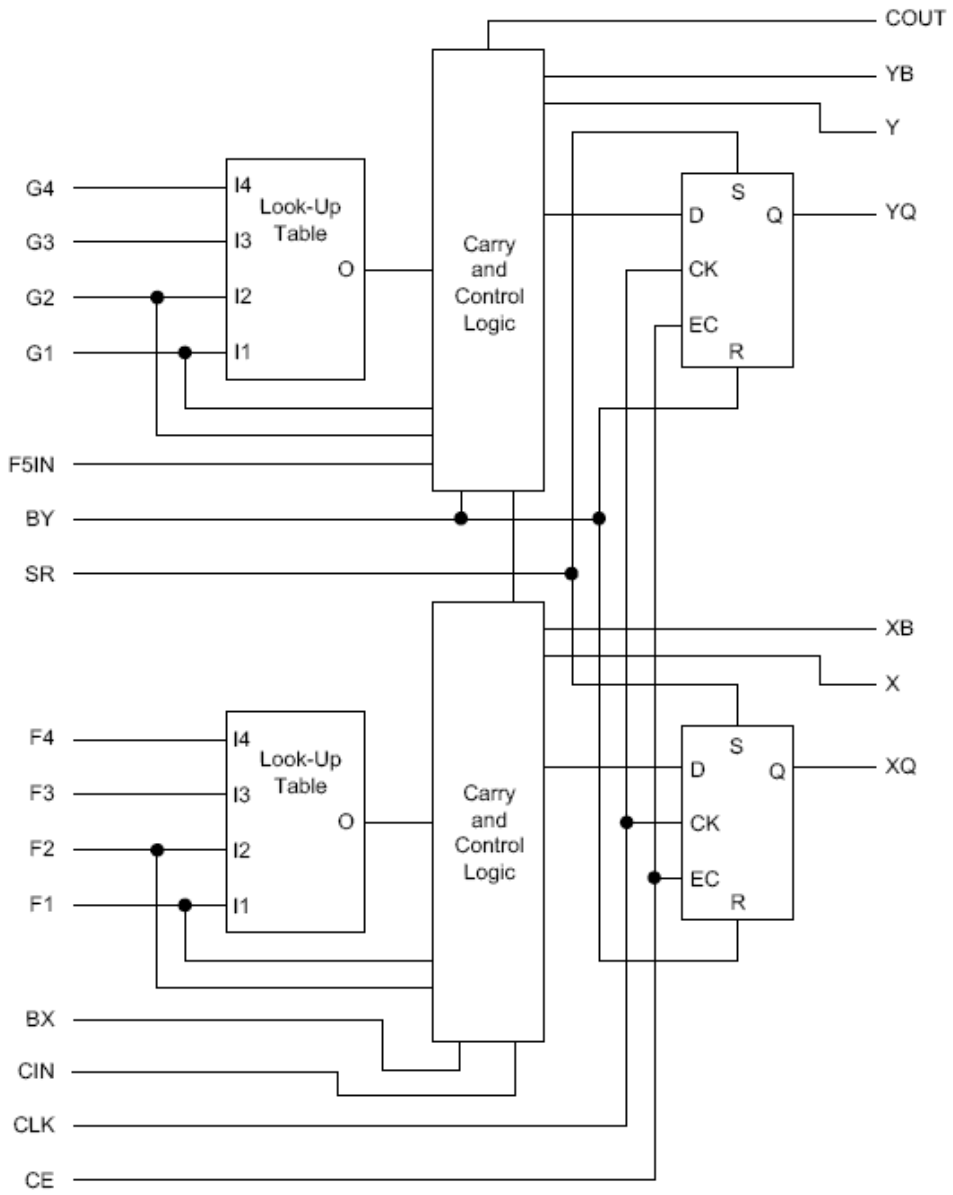
Xilinx XC3000 Interconnection Routing



CLB inputs

- IOB outputs
- CLB outputs
- Global interconnects

Xilinx Spartan-IIE

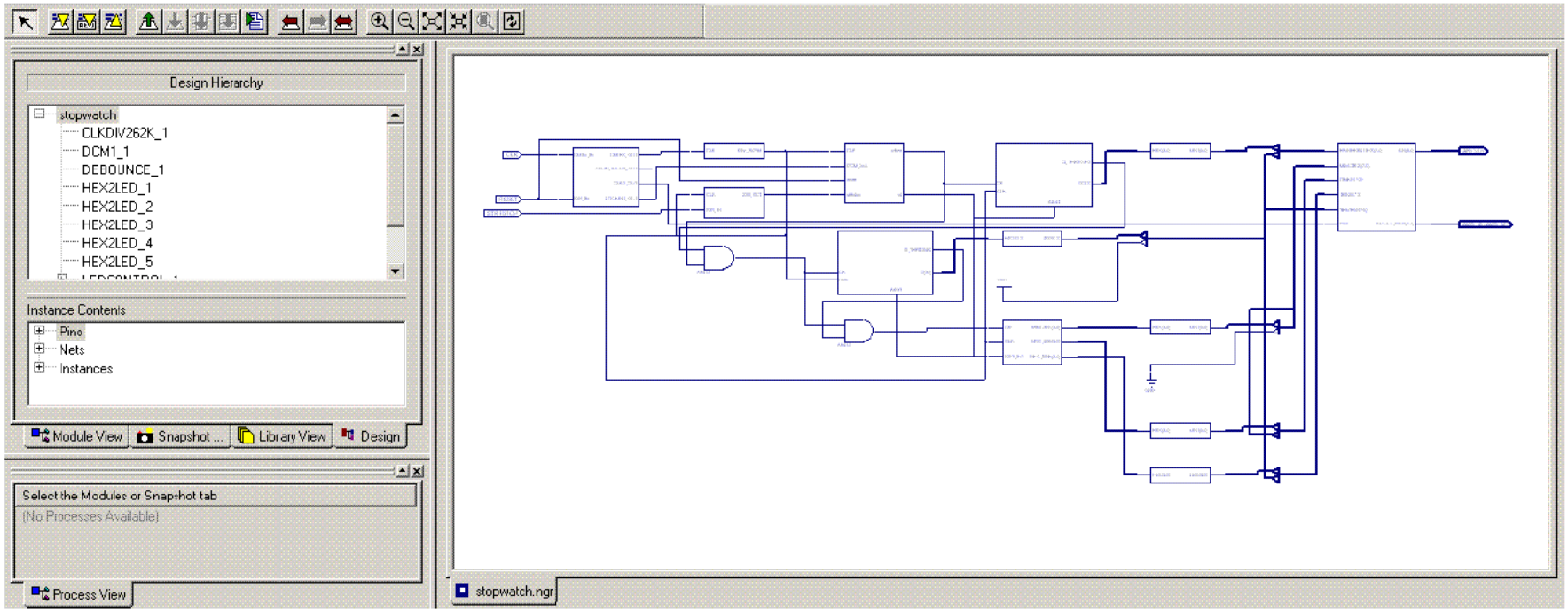


Introduced 2003

CLB contains

- 4 Look-Up Tables
  - 4 inputs each
  - Boolean truth table
  - Generate logic functions
- 4 flip/flops

Programming the FPGA



Typical graphics-oriented integrated development environment (IDE)

Enter circuit characteristics in various forms

- Boolean formula, truth table, Karnaugh map
- Hardware Description Language (**HDL**)
  - VHDL — VHSIC (Very High Speed IC) Hardware Description Language
  - Verilog — HDL based on C

Basic VHDL Code

Entity

Declaration describes I/O ports

Architecture

Declaration describes output as function of inputs

Standard libraries

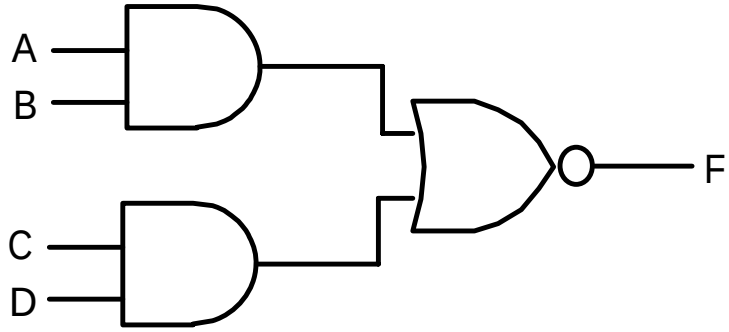
Included prior to entity declaration

```
library ieee;
use ieee.std_logic_1164.all;
entity AND_ent is
port(   x: in std_logic;
        y: in std_logic;
        F: out std_logic
);
end AND_ent;
architecture behav of AND_ent is
begin
    process(x, y)
    begin
     if ((x='1') and (y='1')) then
        F <= '1';
     else
        F <= '0';
     end if;
    end process;
end behav;
```

Example HDL

VHDL Version

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
entity AOI is
port (
  A, B, C, D: in STD_LOGIC;
  F : out STD_LOGIC
);
end AOI;
architecture V1 of AOI is
begin
  F <= not ((A and B) or (C and D));
end V1;
```



Verilog Version

```
module AOI (input A, B, C, D, output F);
  assign F = ~((A & B) | (C & D));
endmodule
```

ASIC

FPGA advantages

- Flexible design of complex logic circuits
- Programmable from GUI for rapid implementation
- Rapid updates for debugging and updates
- Standard libraries and reusable code
- Automated and emulated debugging tools

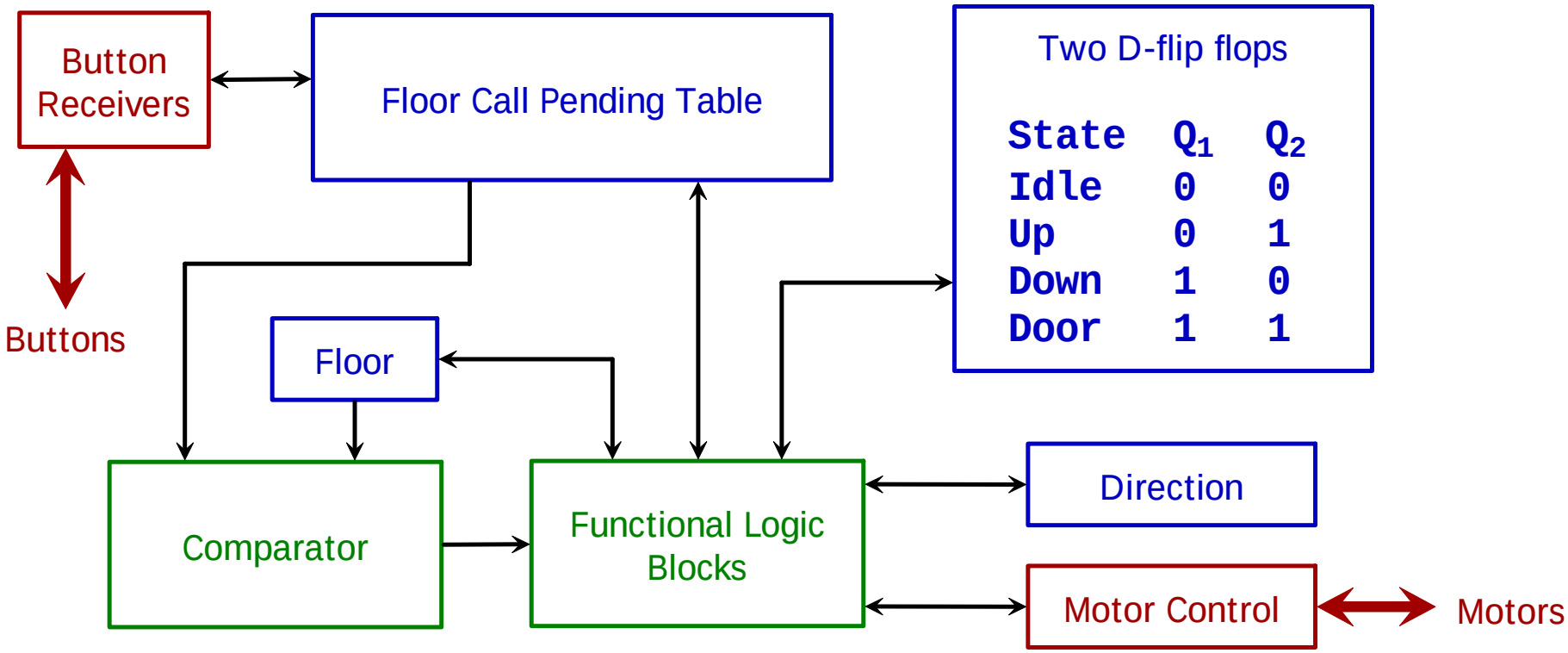
FPGA disadvantages

- Slower than standard VLSI
- Use more power than standard VLSI

Application Specific Integrated Circuit (ASIC)

- Specially made IC for use in a specific system
- ASCI design can be adapted directly from FPGA
- Software converts FPGA program to ASIC design

FPGA/ASIC Approach Elevator Controller



Sequential Logic: Call Pending, Floor, Direction, State (D-flip flops)

Combinational Logic: Comparator, Boolean logic functions

Sensors / Actuators: Buttons and Controllers, Motors and Controllers