Embedded Systems Design: A Unified Hardware/Software Introduction

Chapter 1: Introduction

Outline

- Embedded systems overview
  - What are they?
- Design challenge – optimizing design metrics
- Technologies
  - Processor technologies
  - IC technologies
  - Design technologies

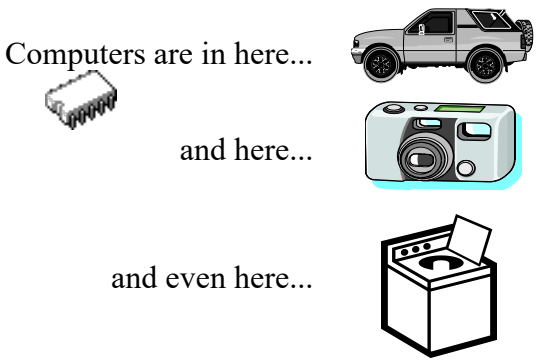
Embedded systems overview

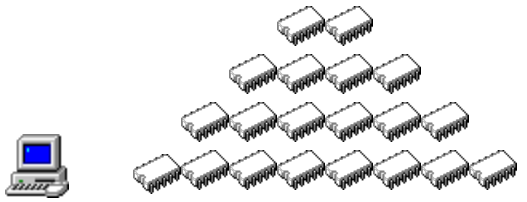
- Computing systems are everywhere
- Most of us think of “desktop” computers
  - PC’s
  - Laptops
  - Mainframes
  - Servers

- But there’s another type of computing system
  - Far more common...

Embedded systems overview

- Embedded computing systems
  - Computing systems embedded within electronic devices
  - Hard to define. Nearly any computing system other than a desktop computer
  - Billions of units produced yearly, versus millions of desktop units
  - Perhaps 50 per household and per automobile





Lots more of these, though they cost a lot less each.

A “short list” of embedded systems

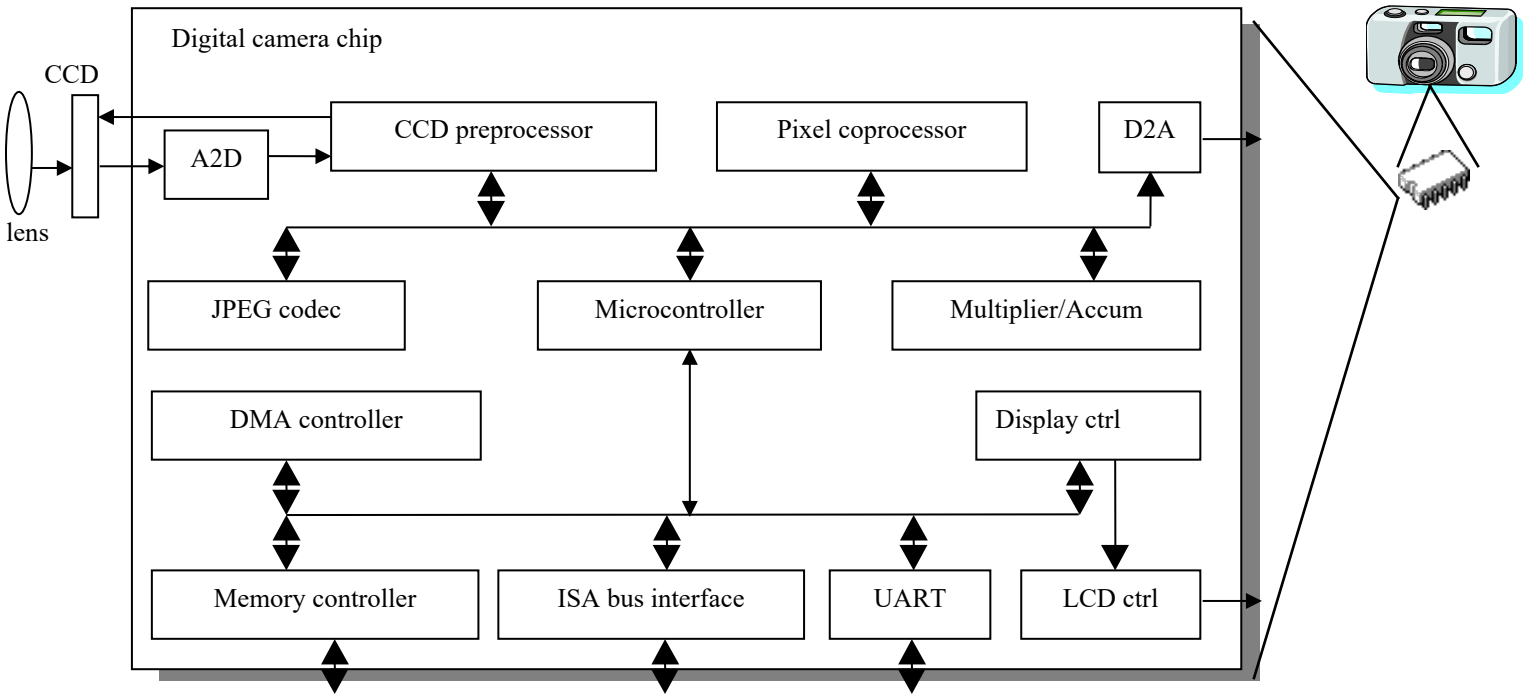
Anti-lock brakes
Auto-focus cameras
Automatic teller machines
Automatic toll systems
Automatic transmission
Avionic systems
Battery chargers
Camcorders
Cell phones
Cell-phone base stations
Cordless phones
Cruise control
Curbside check-in systems
Digital cameras
Disk drives
Electronic card readers
Electronic instruments
Electronic toys/games
Factory control
Fax machines
Fingerprint identifiers
Home security systems
Life-support systems
Medical testing systems
Modems
MPEG decoders
Network cards
Network switches/routers
On-board navigation
Pagers
Photocopiers
Point-of-sale systems
Portable video games
Printers
Satellite phones
Scanners
Smart ovens/dishwashers
Speech recognizers
Stereo systems
Teleconferencing systems
Televisions
Temperature controllers
Theft tracking systems
TV set-top boxes
VCR’s, DVD players
Video game consoles
Video phones
Washers and dryers

And the list goes on and on

Some common characteristics of embedded systems

- Single-functioned
  - Executes a single program, repeatedly
- Tightly-constrained
  - Low cost, low power, small, fast, etc.
- Reactive and real-time
  - Continually reacts to changes in the system's environment
  - Must compute certain results in real-time without delay

An embedded system example -- a digital camera



- Single-functioned -- always a digital camera
- Tightly-constrained -- Low cost, low power, small, fast
- Reactive and real-time -- only to a small extent

Design challenge – optimizing design metrics

- Obvious design goal:
  - Construct an implementation with desired functionality
- Key design challenge:
  - Simultaneously optimize numerous design metrics
- Design metric
  - A measurable feature of a system's implementation
  - Optimizing design metrics is a key challenge

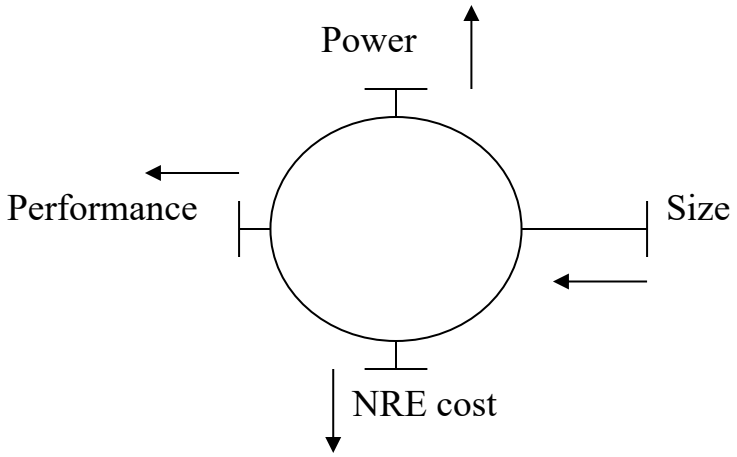
Design challenge – optimizing design metrics

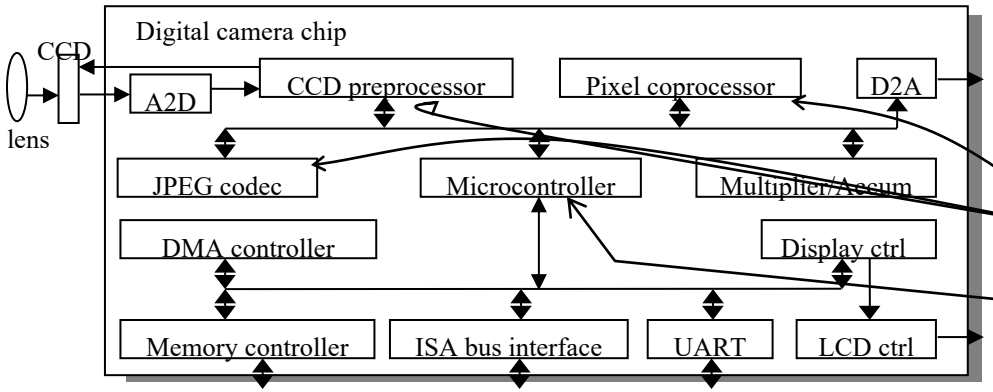
- Common metrics
  - Unit cost: the monetary cost of manufacturing each copy of the system, excluding NRE cost
  - NRE cost (Non-Recurring Engineering cost): The one-time monetary cost of designing the system
  - Size: the physical space required by the system
  - Performance: the execution time or throughput of the system
  - Power: the amount of power consumed by the system
  - Flexibility: the ability to change the functionality of the system without incurring heavy NRE cost

Design challenge – optimizing design metrics

- Common metrics (continued)
  - Time-to-prototype: the time needed to build a working version of the system
  - Time-to-market: the time required to develop a system to the point that it can be released and sold to customers
  - Maintainability: the ability to modify the system after its initial release
  - Correctness, safety, many more

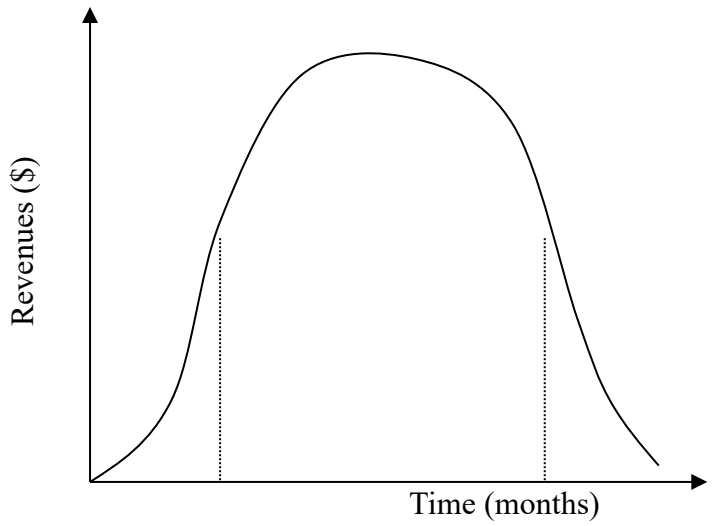
Design metric competition -- improving one may worsen others





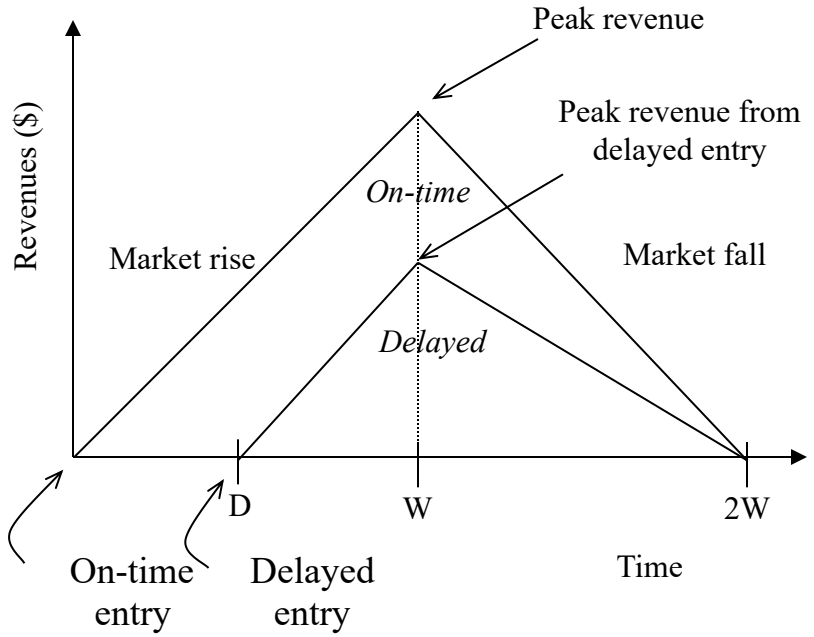
- Expertise with both **software and hardware** is needed to optimize design metrics
  - Not just a hardware or software expert, as is common
  - A designer must be comfortable with various technologies in order to choose the best for a given application and constraints

Time-to-market: a demanding design metric



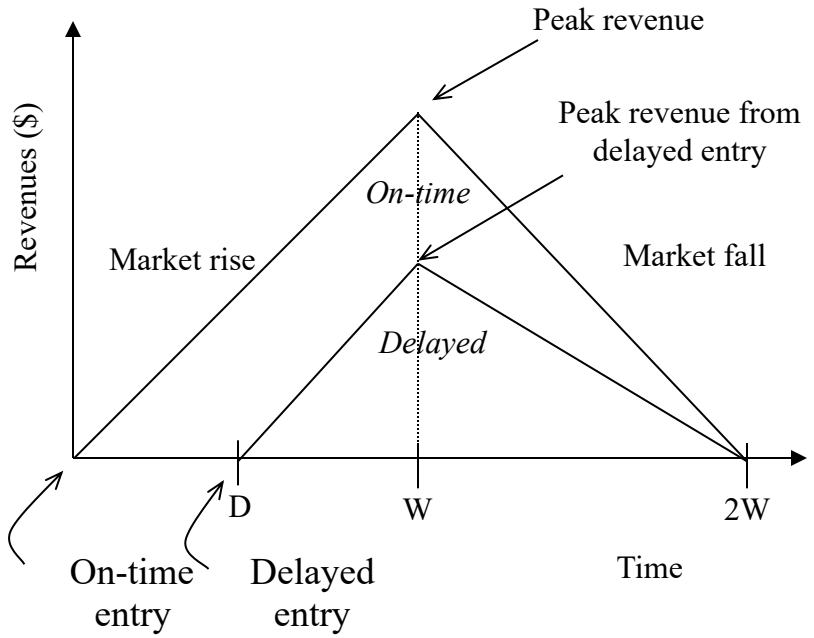
- Time required to develop a product to the point it can be sold to customers
- Market window
  - Period during which the product would have highest sales
- Average time-to-market constraint is about 8 months
- Delays can be costly

Losses due to delayed market entry



- Simplified revenue model
  - Product life = 2W, peak at W
  - Time of market entry defines a triangle, representing market penetration
  - Triangle area equals revenue
- Loss
  - The difference between the on-time and delayed triangle areas

Losses due to delayed market entry (cont.)



- Area = 1/2 * base * height
  - On-time = 1/2 * 2W * W
  - Delayed = 1/2 * (W-D+W)*(W-D)
- Percentage revenue loss = $(D(3W-D)/2W^2)*100\%$
- Try some examples
  - Lifetime 2W=52 wks, delay D=4 wks
  - (4*(3*26 –4)/2*26^2) = 22%
  - Lifetime 2W=52 wks, delay D=10 wks
  - (10*(3*26 –10)/2*26^2) = 50%
  - Delays are costly!

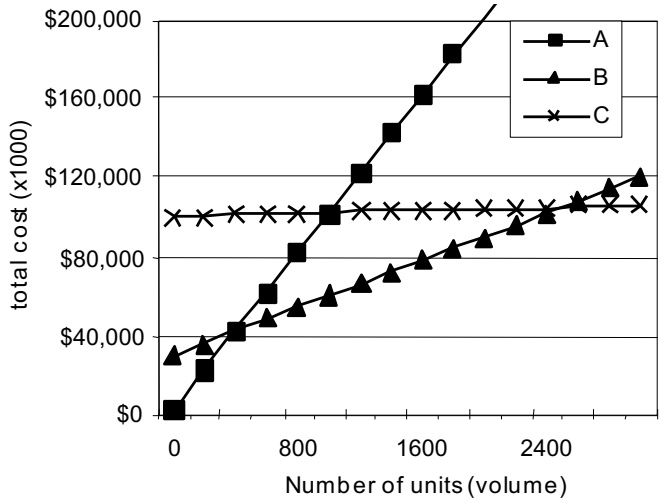
NRE and unit cost metrics

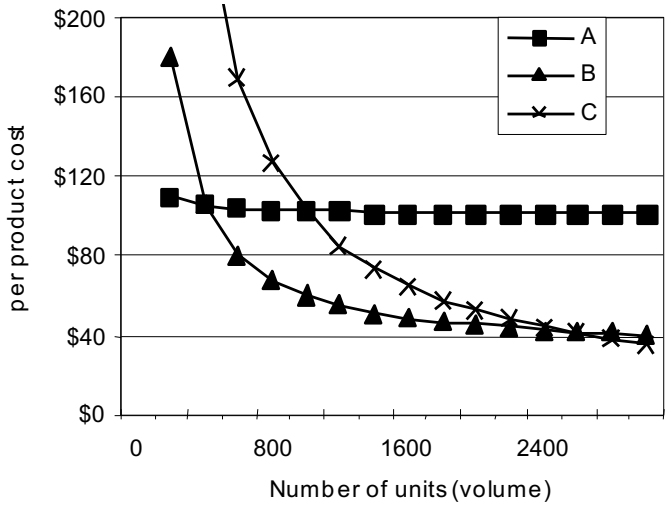
- Costs:
  - Unit cost: the monetary cost of manufacturing each copy of the system, excluding NRE cost
  - NRE cost (Non-Recurring Engineering cost): The one-time monetary cost of designing the system
  - *total cost = NRE cost + unit cost * # of units*
  - *per-product cost = total cost / # of units*
    = (NRE cost / # of units) + unit cost
- Example
  - NRE=$2000, unit=$100
  - For 10 units
    - total cost = $2000 + 10*$100 = $3000
    - per-product cost = $2000/10 + $100 = $300

Amortizing NRE cost over the units results in an additional $200 per unit

NRE and unit cost metrics

- Compare technologies by costs -- best depends on quantity
  - Technology A: NRE=$2,000, unit=$100
  - Technology B: NRE=$30,000, unit=$30
  - Technology C: NRE=$100,000, unit=$2





- But, must also consider time-to-market

The performance design metric

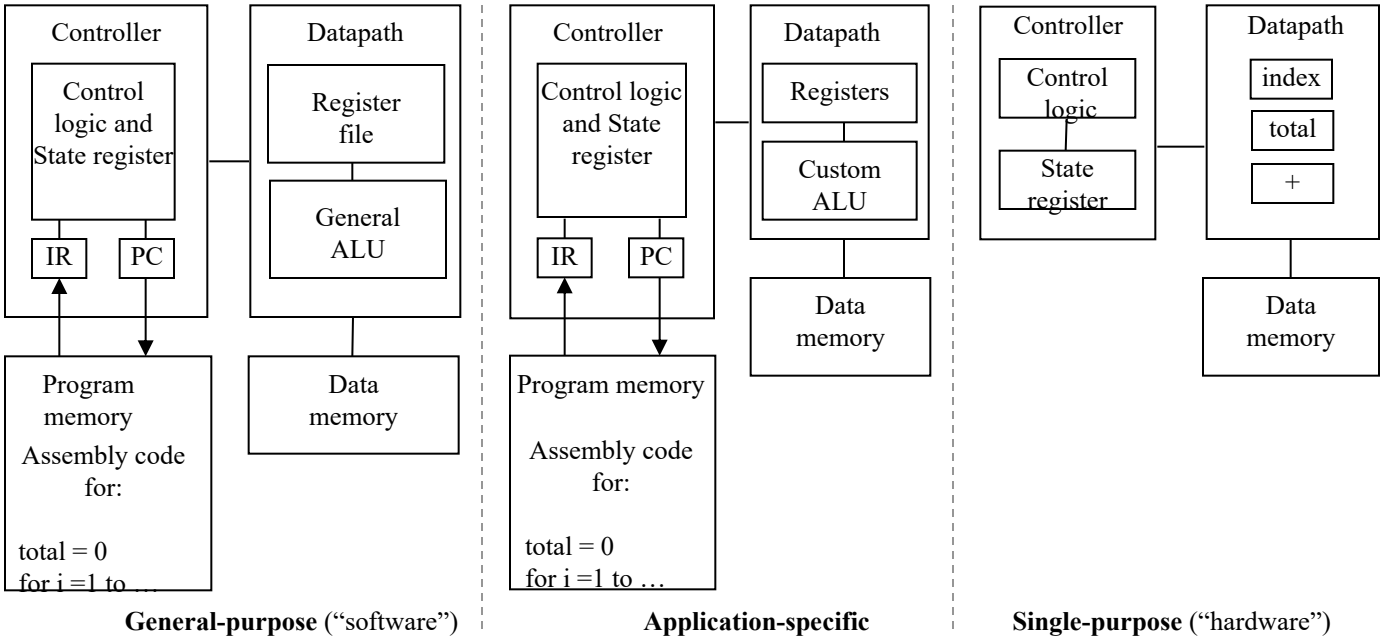
- Widely-used measure of system, widely-abused
  - Clock frequency, instructions per second – not good measures
  - Digital camera example – a user cares about how fast it processes images, not clock speed or instructions per second
- Latency (response time)
  - Time between task start and end
  - e.g., Camera's A and B process images in 0.25 seconds
- Throughput
  - Tasks per second, e.g. Camera A processes 4 images per second
  - Throughput can be more than latency seems to imply due to concurrency, e.g. Camera B may process 8 images per second (by capturing a new image while previous image is being stored).
- *Speedup* of B over S = B's performance / A's performance
  - Throughput speedup = 8/4 = 2

Three key embedded system technologies

- Technology
  - A manner of accomplishing a task, especially using technical processes, methods, or knowledge
- Three key technologies for embedded systems
  - Processor technology
  - IC technology
  - Design technology

Processor technology

- The architecture of the computation engine used to implement a system's desired functionality
- Processor does not have to be programmable
  - "Processor" *not* equal to general-purpose processor



Processor technology

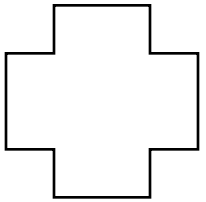
- Processors vary in their customization for the problem at hand

Desired functionality

```
total = 0
for i = 1 to N  loop
   total += M[i]
end loop
```

General-purpose processor

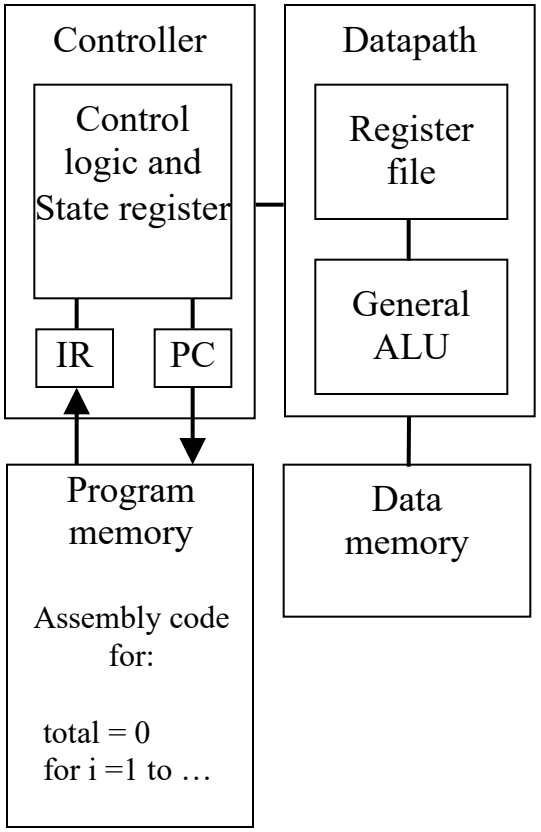
Application-specific processor



Single-purpose processor

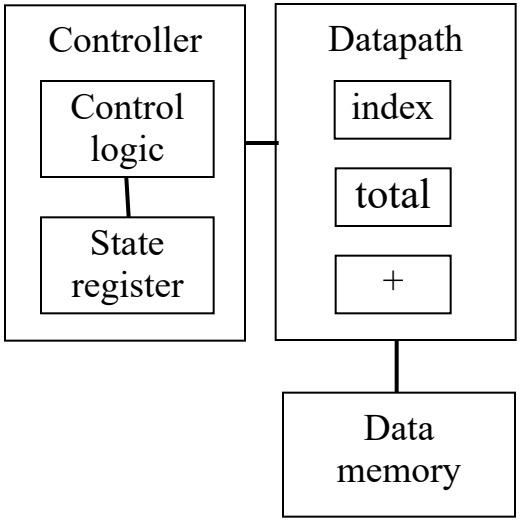
General-purpose processors

- Programmable device used in a variety of applications
  - Also known as “microprocessor”
- Features
  - Program memory
  - General datapath with large register file and general ALU
- User benefits
  - Low time-to-market and NRE costs
  - High flexibility
- “Pentium” the most well-known, but there are hundreds of others



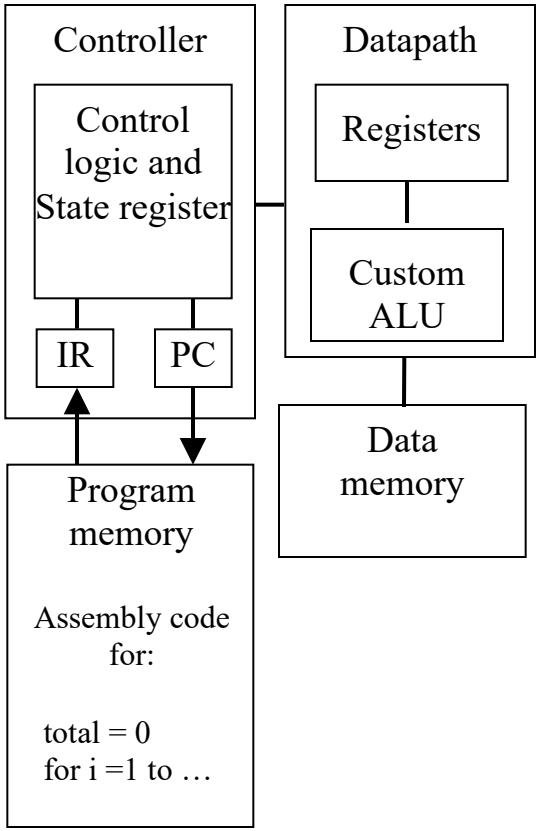
Single-purpose processors

- Digital circuit designed to execute exactly one program
  - a.k.a. coprocessor, accelerator or peripheral
- Features
  - Contains only the components needed to execute a single program
  - No program memory
- Benefits
  - Fast
  - Low power
  - Small size



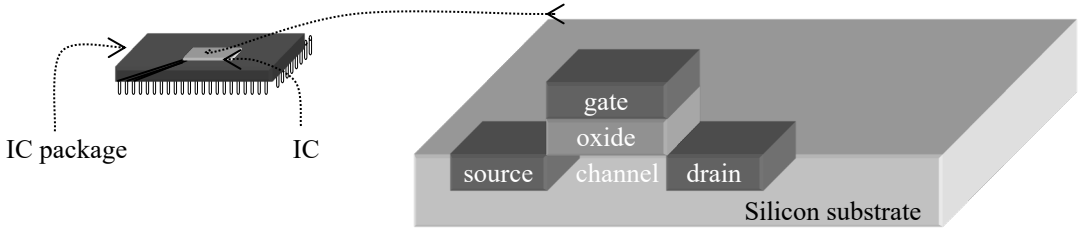
Application-specific processors

- Programmable processor optimized for a particular class of applications having common characteristics
  - Compromise between general-purpose and single-purpose processors
- Features
  - Program memory
  - Optimized datapath
  - Special functional units
- Benefits
  - Some flexibility, good performance, size and power



IC technology

- The manner in which a digital (gate-level) implementation is mapped onto an IC
  - IC: Integrated circuit, or "chip"
  - IC technologies differ in their customization to a design
  - IC's consist of numerous layers (perhaps 10 or more)
    - IC technologies differ with respect to who builds each layer and when



IC technology

- Three types of IC technologies
  - Full-custom/VLSI
  - Semi-custom ASIC (gate array and standard cell)
  - PLD (Programmable Logic Device)

Full-custom/VLSI

- All layers are optimized for an embedded system's particular digital implementation
  - Placing transistors
  - Sizing transistors
  - Routing wires
- Benefits
  - Excellent performance, small size, low power
- Drawbacks
  - High NRE cost (e.g., $300k), long time-to-market

Semi-custom

- Lower layers are fully or partially built
  - Designers are left with routing of wires and maybe placing some blocks
- Benefits
  - Good performance, good size, less NRE cost than a full-custom implementation (perhaps $10k to $100k)
- Drawbacks
  - Still require weeks to months to develop

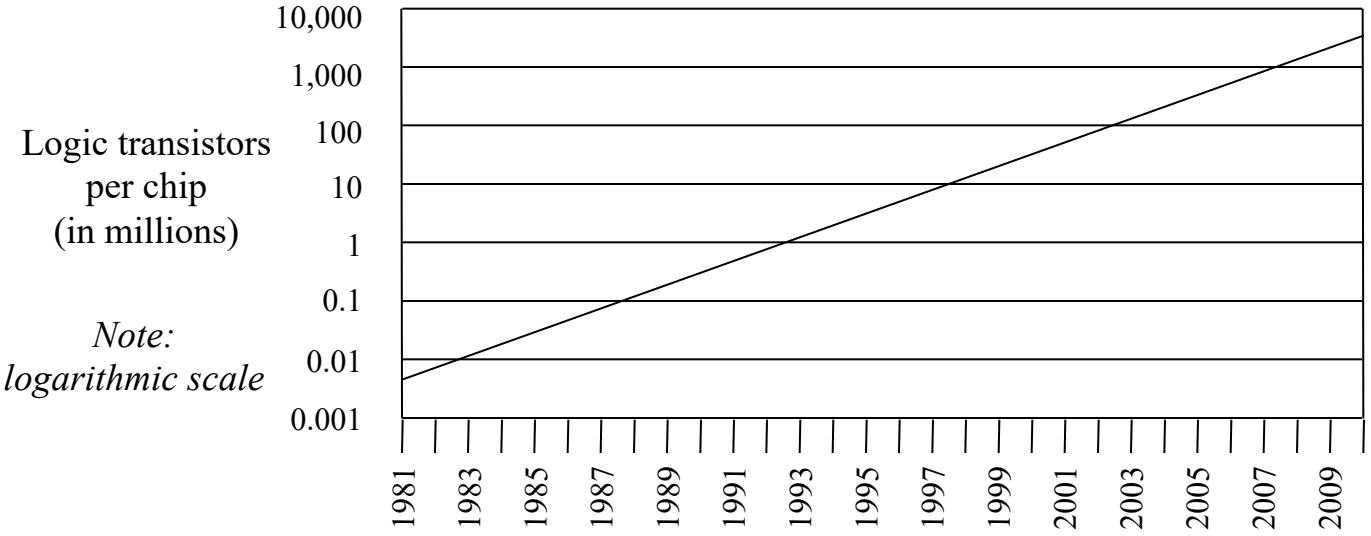
PLD (Programmable Logic Device)

- All layers already exist
  - Designers can purchase an IC
  - Connections on the IC are either created or destroyed to implement desired functionality
  - Field-Programmable Gate Array (FPGA) very popular
- Benefits
  - Low NRE costs, almost instant IC availability
- Drawbacks
  - Bigger, expensive (perhaps $30 per unit), power hungry, slower

Moore's law

- The most important trend in embedded systems
  - Predicted in 1965 by Intel co-founder Gordon Moore

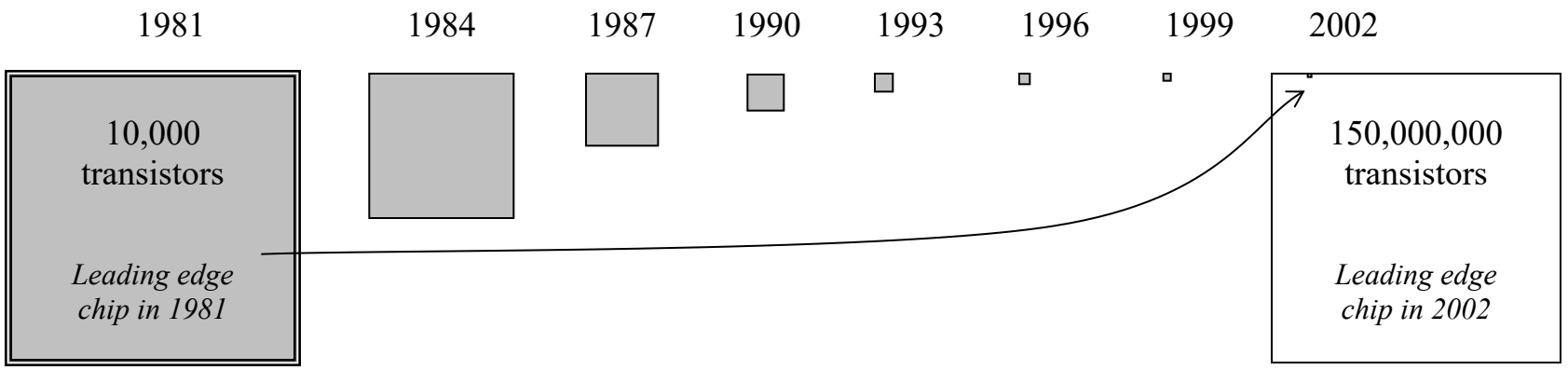
IC transistor capacity has doubled roughly every 18 months for the past several decades



Moore's law

- Wow
  - This growth rate is hard to imagine, most people underestimate
  - How many ancestors do you have from 20 generations ago
    - i.e., roughly how many people alive in the 1500's did it take to make you?
    - 2^{20} = more than *1 million people*
  - *(This underestimation is the key to pyramid schemes!)*

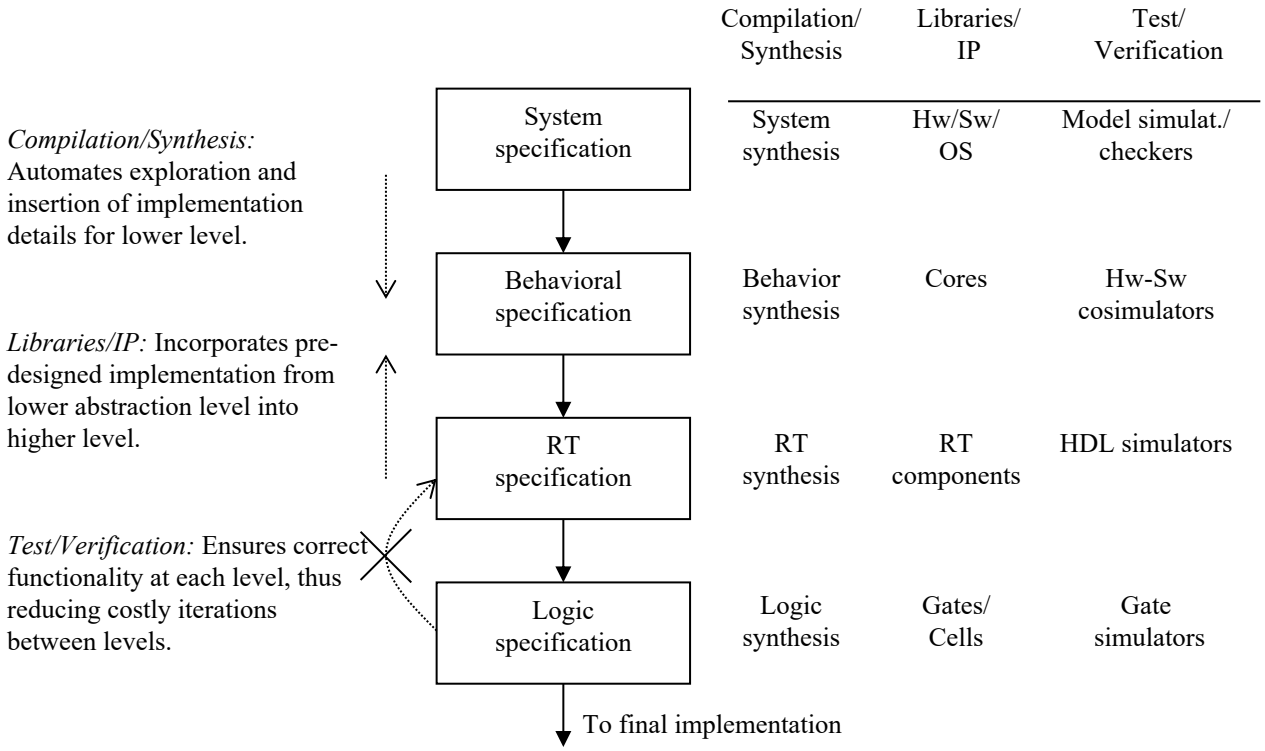
Graphical illustration of Moore's law



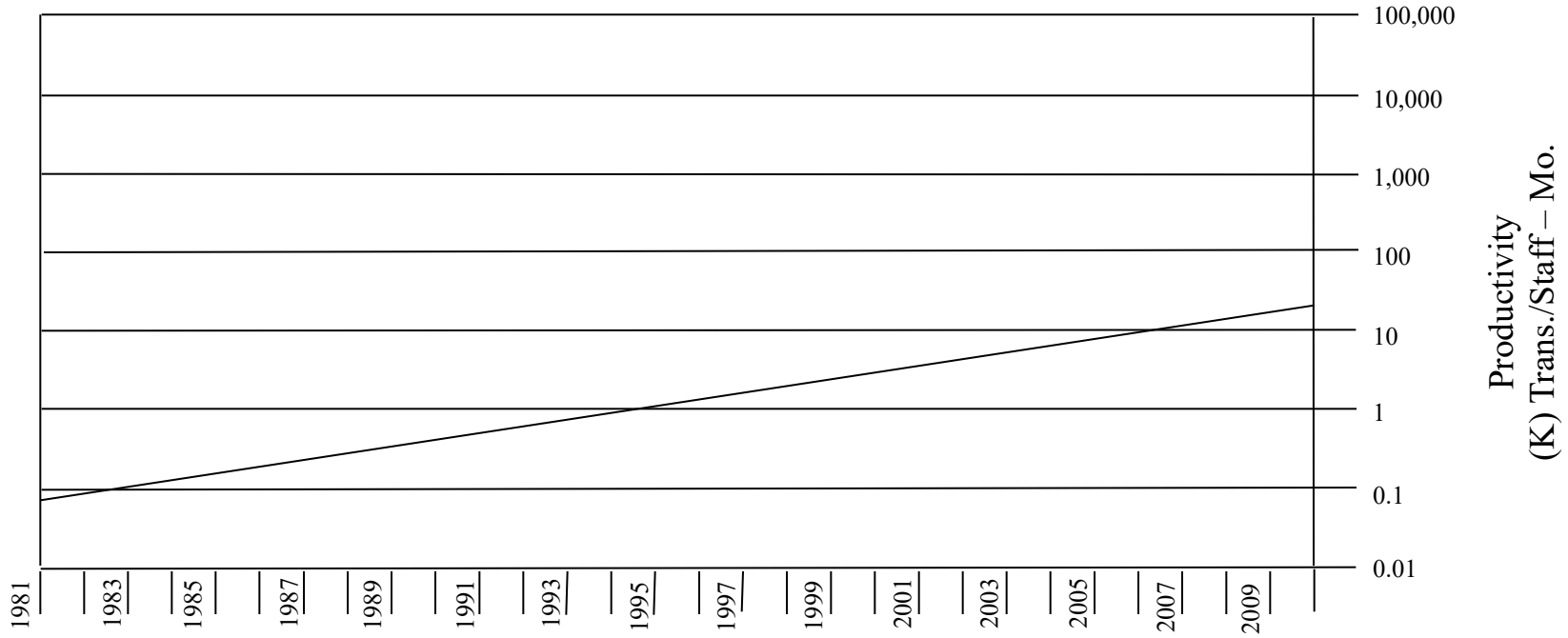
- Something that doubles frequently grows more quickly than most people realize!
  - A 2002 chip can hold about 15,000 1981 chips inside itself

Design Technology

- The manner in which we convert our concept of desired system functionality into an implementation



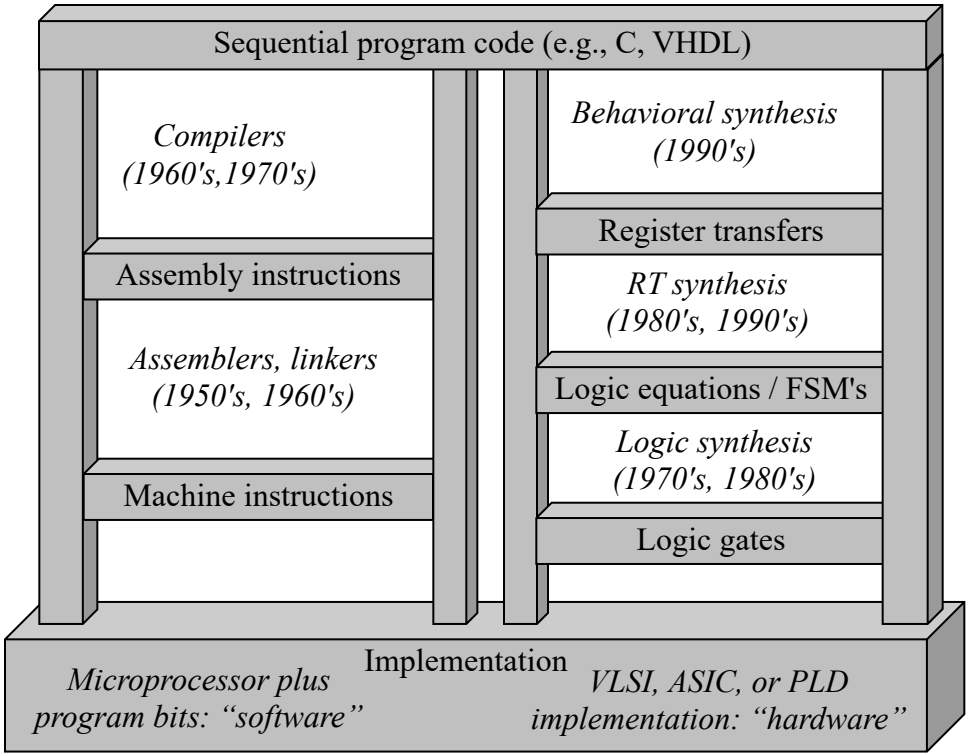
Design productivity exponential increase



- Exponential increase over the past few decades

The co-design ladder

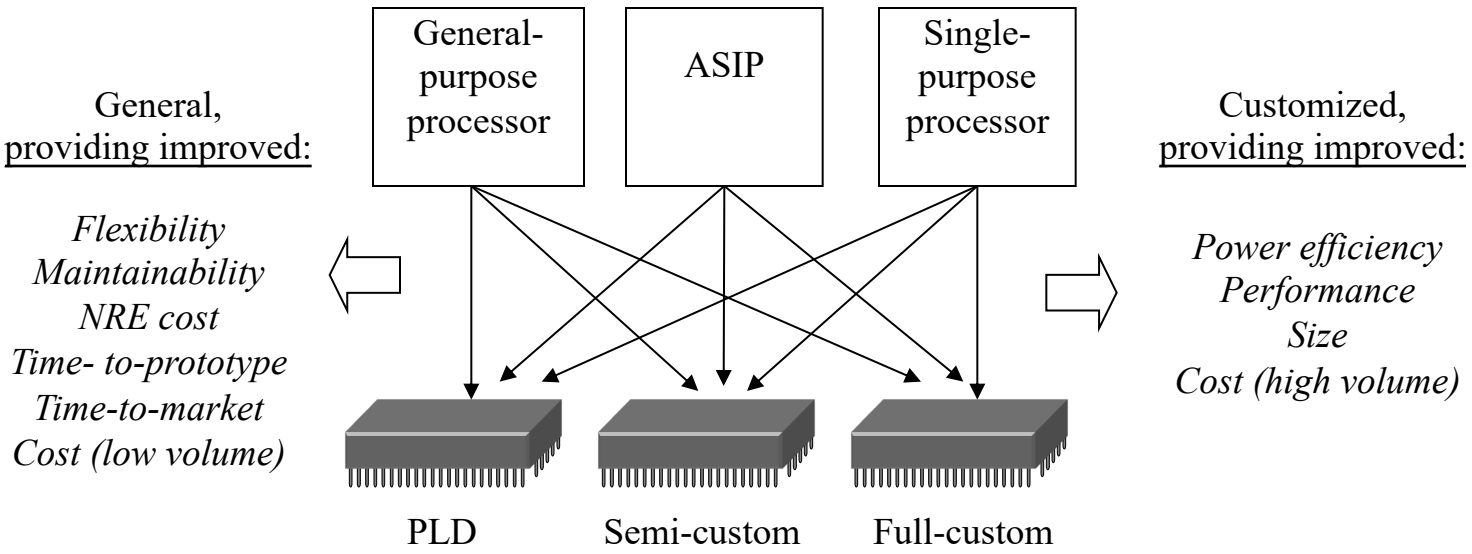
- In the past:
  - Hardware and software design technologies were very different
  - Recent maturation of synthesis enables a unified view of hardware and software
- Hardware/software "codesign"



The choice of hardware versus software for a particular function is simply a tradeoff among various design metrics, like performance, power, size, NRE cost, and especially flexibility; there is no fundamental difference between what hardware or software can implement.

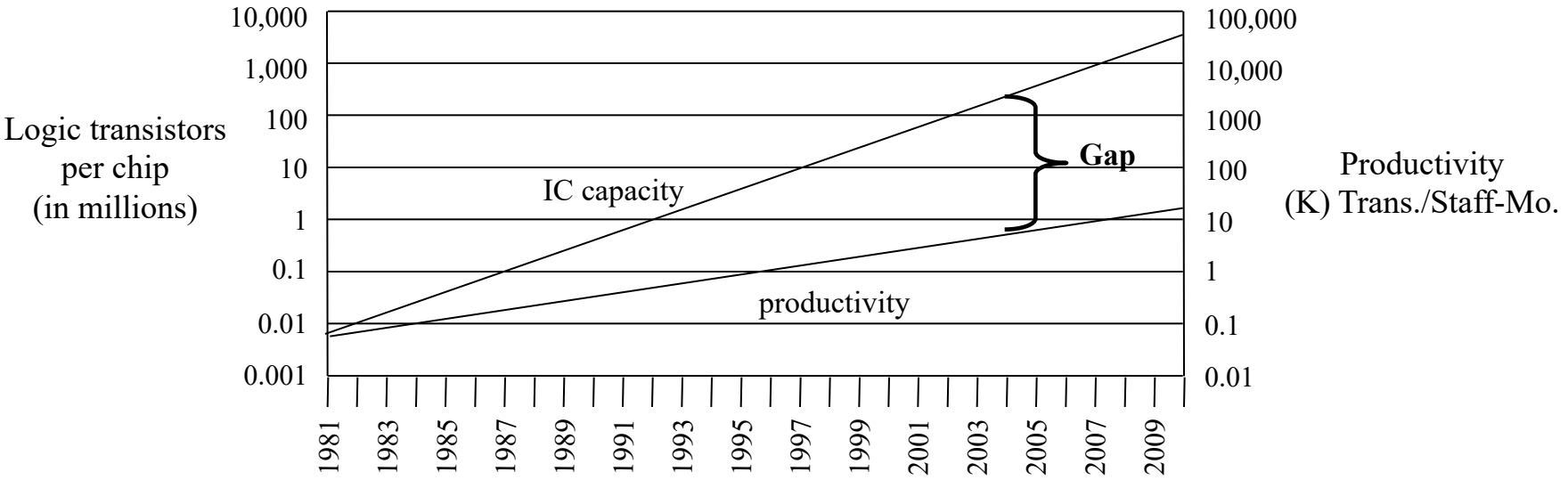
Independence of processor and IC technologies

- Basic tradeoff
  - General vs. custom
  - With respect to processor technology or IC technology
  - The two technologies are independent



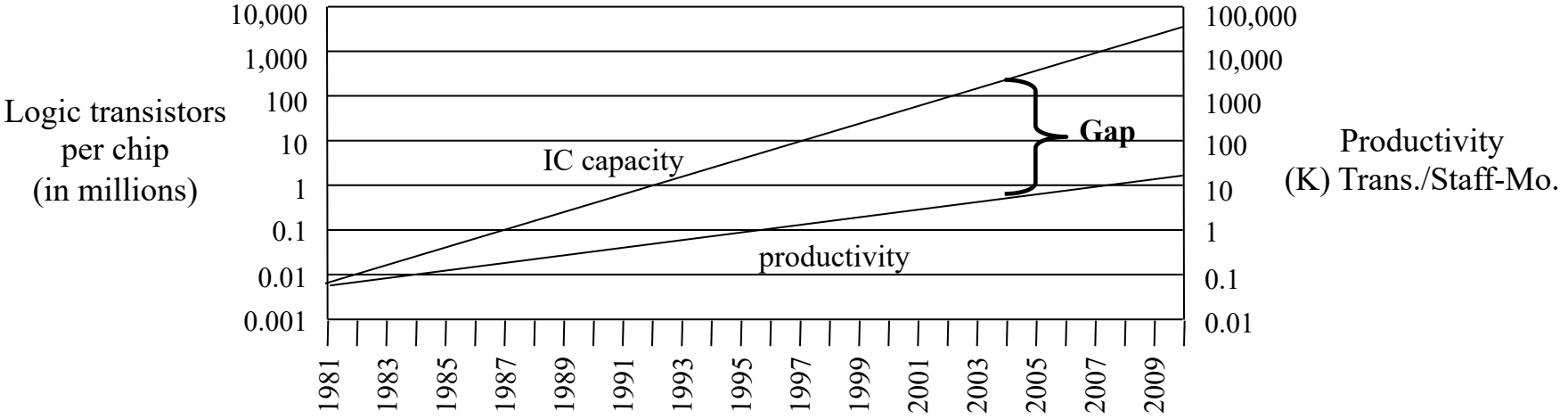
Design productivity gap

- While designer productivity has grown at an impressive rate over the past decades, the rate of improvement has not kept pace with chip capacity



Design productivity gap

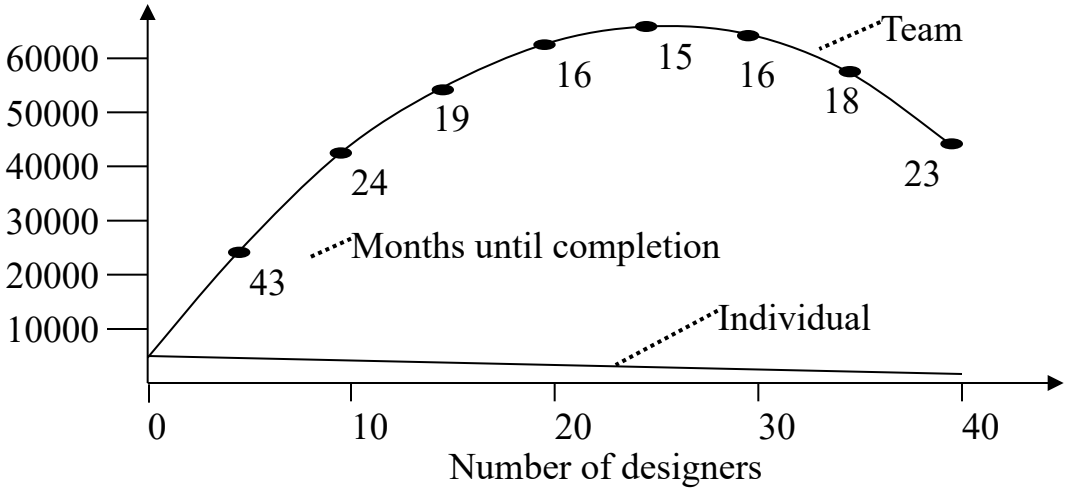
- 1981 leading edge chip required 100 designer months
  - 10,000 transistors / 100 transistors/month
- 2002 leading edge chip requires 30,000 designer months
  - 150,000,000 / 5000 transistors/month
- Designer cost increase from $1M to $300M



The mythical man-month

- The situation is even worse than the productivity gap indicates
- In theory, adding designers to team reduces project completion time
- In reality, productivity per designer decreases due to complexities of team management and communication
- In the software community, known as “the mythical man-month” (Brooks 1975)
- At some point, can actually lengthen project completion time! (“Too many cooks”)

- 1M transistors, 1 designer=5000 trans/month
- Each additional designer reduces for 100 trans/month
- So 2 designers produce 4900 trans/month each



Summary

- Embedded systems are everywhere
- Key challenge: optimization of design metrics
  - Design metrics compete with one another
- A unified view of hardware and software is necessary to improve productivity
- Three key technologies
  - Processor: general-purpose, application-specific, single-purpose
  - IC: Full-custom, semi-custom, PLD
  - Design: Compilation/synthesis, libraries/IP, test/verification

Embedded Systems Design: A Unified Hardware/Software Introduction

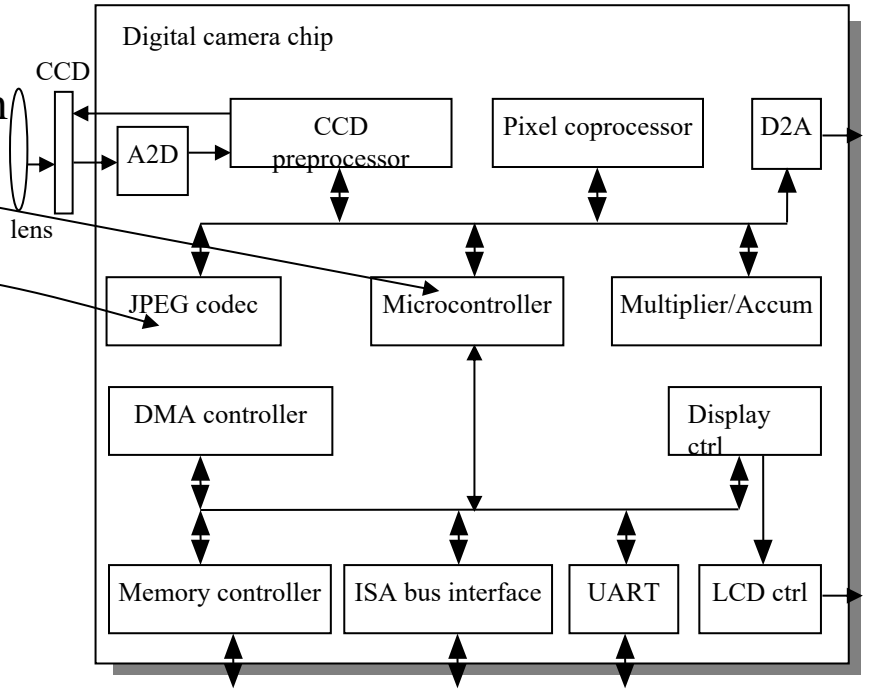
Chapter 2: Custom single-purpose processors

Outline

- Introduction
- Combinational logic
- Sequential logic
- Custom single-purpose processor design
- RT-level custom single-purpose processor design

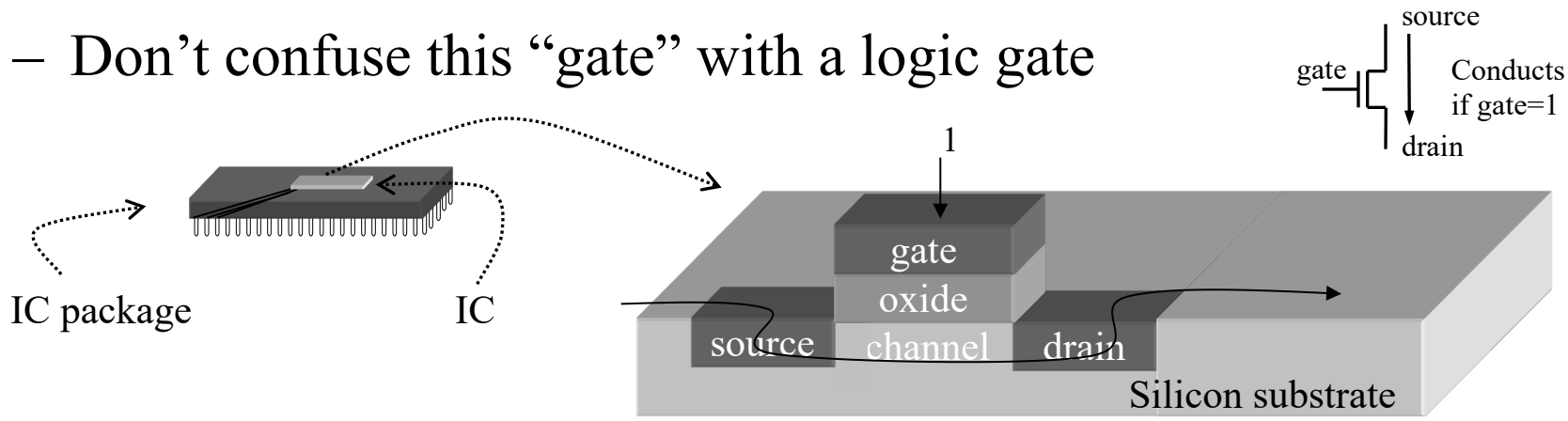
Introduction

- Processor
  - Digital circuit that performs a computation tasks
  - Controller and datapath
  - General-purpose: variety of computation tasks
  - Single-purpose: one particular computation task
  - Custom single-purpose: non-standard task
- A custom single-purpose processor may be
  - Fast, small, low power
  - But, high NRE, longer time-to-market, less flexible



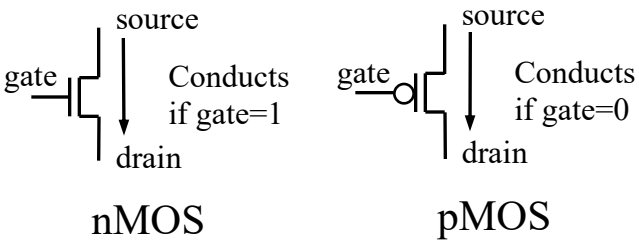
CMOS transistor on silicon

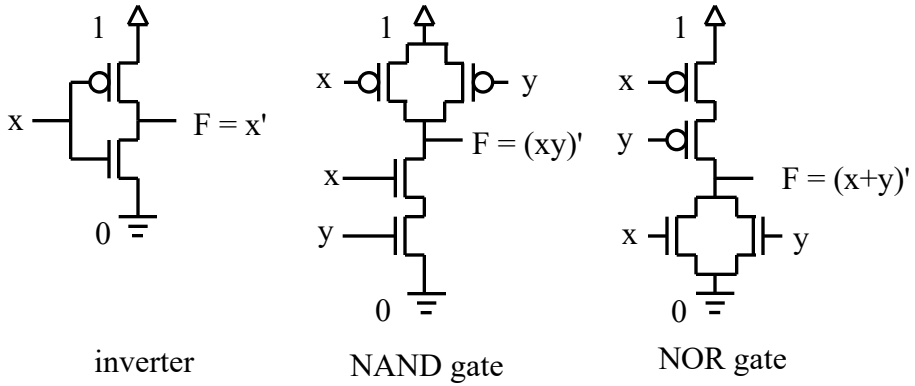
- Transistor
  - The basic electrical component in digital systems
  - Acts as an on/off switch
  - Voltage at "gate" controls whether current flows from source to drain
  - Don't confuse this "gate" with a logic gate



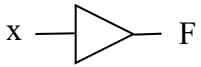
CMOS transistor implementations

- Complementary Metal Oxide Semiconductor
- We refer to logic levels
  - Typically 0 is 0V, 1 is 5V
- Two basic CMOS types
  - nMOS conducts if gate=1
  - pMOS conducts if gate=0
  - Hence “complementary”
- Basic gates
  - Inverter, NAND, NOR



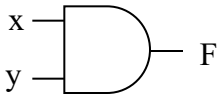


Basic logic gates



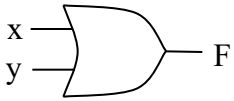
x	F
0	0
1	1

F = x
Driver



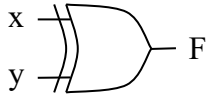
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

F = x y
AND



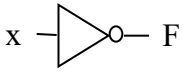
x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

F = x + y
OR



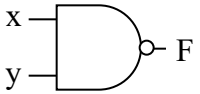
x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

F = x ⊕ y
XOR



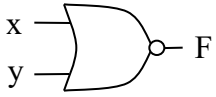
x	F
0	1
1	0

F = x'
Inverter



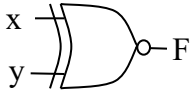
x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

F = (x y)'
NAND



x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

F = (x+y)'
NOR



x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

F = x ⊙ y
XNOR

Combinational logic design

A) Problem description

y is 1 if a is to 1, or b and c are 1. z is 1 if b or c is to 1, but not both, or if all are 1.

B) Truth table

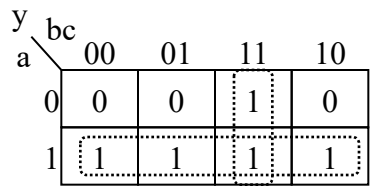
Inputs			Outputs	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

C) Output equations

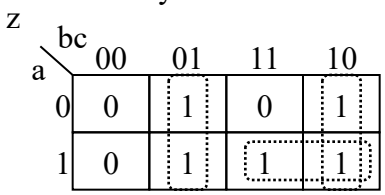
y = a'bc + ab'c' + ab'c + abc' + abc

z = a'b'c + a'bc' + ab'c + abc' + abc

D) Minimized output equations

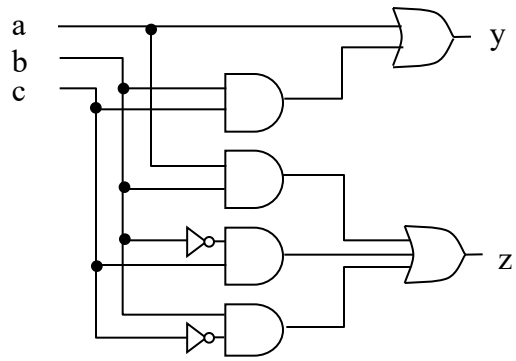


y = a + bc

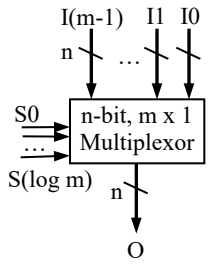
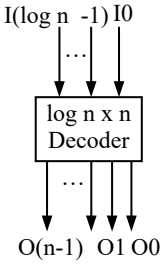
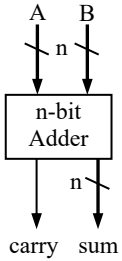
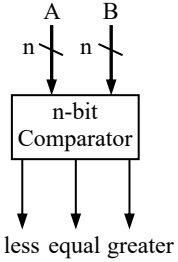
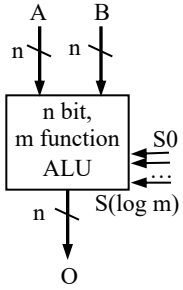


z = ab + b'c + bc'

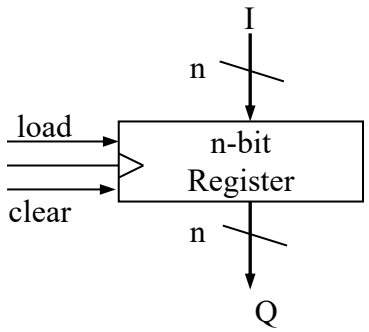
E) Logic Gates



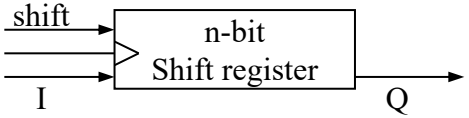
Combinational components

				
O = I0 if S=0..00 I1 if S=0..01 … I(m-1) if S=1..11	O0 =1 if I=0..00 O1 =1 if I=0..01 … O(n-1) =1 if I=1..11	sum = A+B (first n bits) carry = (n+1)'th bit of A+B	less = 1 if A<B equal =1 if A=B greater=1 if A>B	O = A *op* B *op* determined by S.
	With enable input e → all O's are 0 if e=0	With carry-in input Ci→ sum = A + B + Ci		May have status outputs carry, zero, etc.

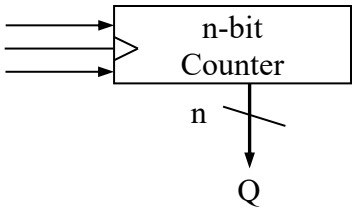
Sequential components



Q =
0 if clear=1,
I if load=1 and clock=1,
Q(previous) otherwise.



Q = lsb
- Content shifted
- I stored in msb



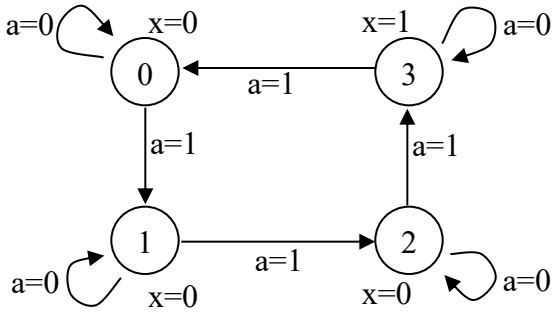
Q =
0 if clear=1,
Q(prev)+1 if count=1 and clock=1.

Sequential logic design

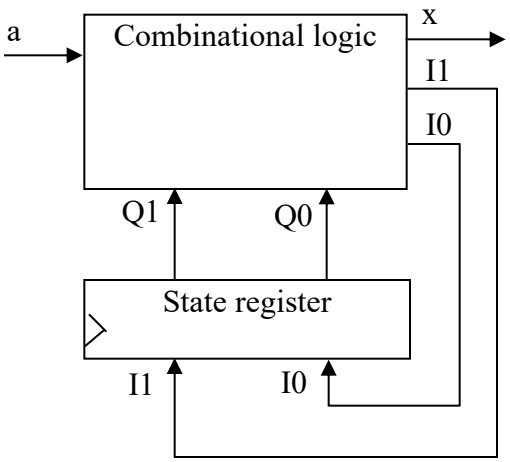
A) Problem Description

You want to construct a clock divider. Slow down your pre-existing clock so that you output a 1 for every four clock cycles

B) State Diagram



C) Implementation Model



D) State Table (Moore-type)

Inputs			Outputs		
Q1	Q0	a	I1	I0	x
0	0	0	0	0	0
0	0	1	0	1	
0	1	0	0	1	0
0	1	1	1	0	
1	0	0	1	0	0
1	0	1	1	1	
1	1	0	1	1	1
1	1	1	0	0	

- Given this implementation model
  - Sequential logic design quickly reduces to combinational logic design

Sequential logic design (cont.)

E) Minimized Output Equations

I1 Q1Q0

a \ Q1Q0	00	01	11	10
0	0	0	1	1
1	0	1	0	1

I1 = Q1'Q0a + Q1a' + Q1Q0'

I0 Q1Q0

a \ Q1Q0	00	01	11	10
0	0	1	1	0
1	1	0	0	1

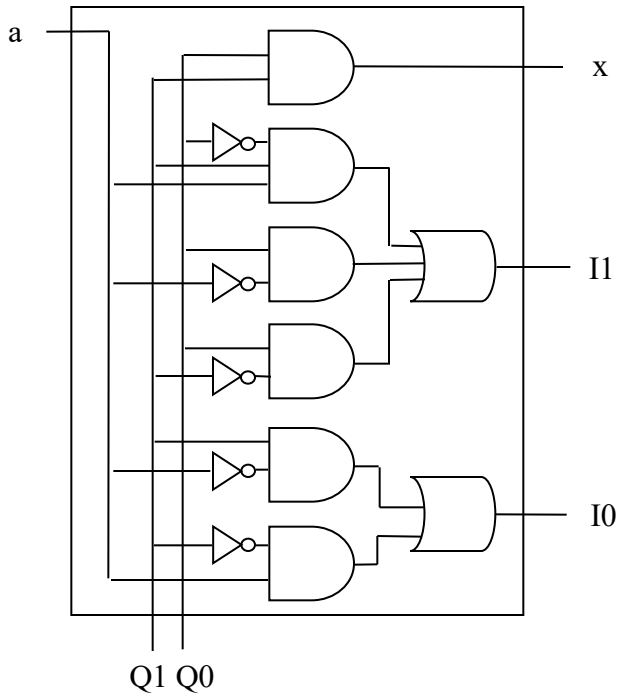
I0 = Q0a' + Q0'a

x Q1Q0

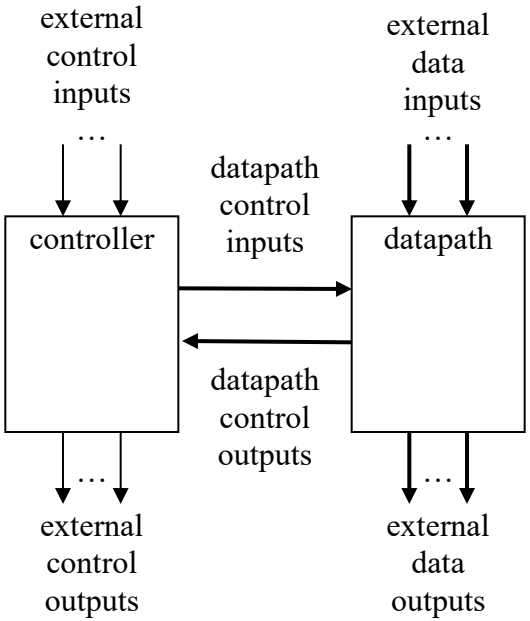
a \ Q1Q0	00	01	11	10
0	0	0	1	0
1	0	0	1	0

x = Q1Q0

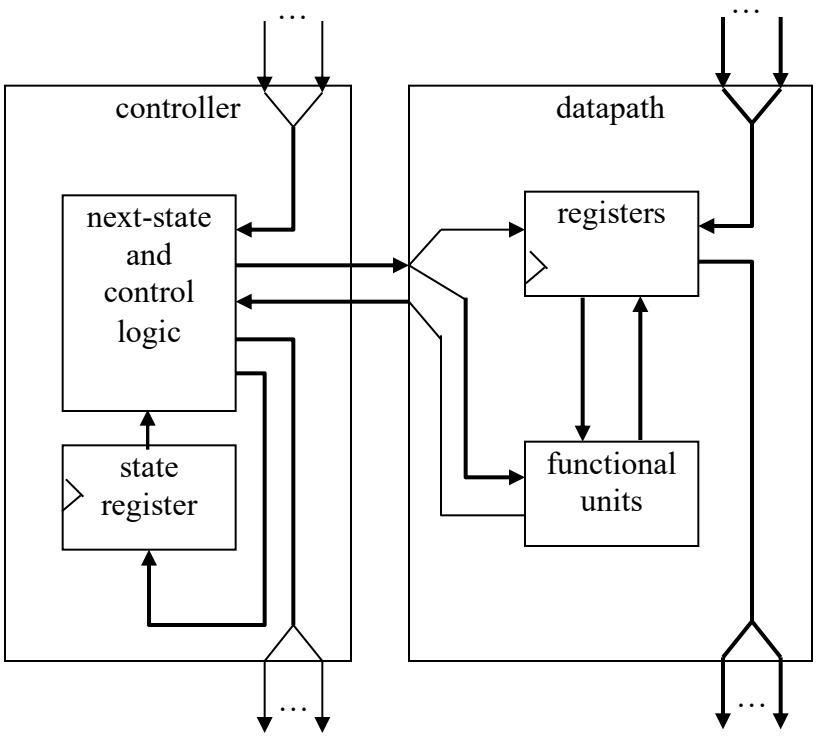
F) Combinational Logic



Custom single-purpose processor basic model



controller and datapath

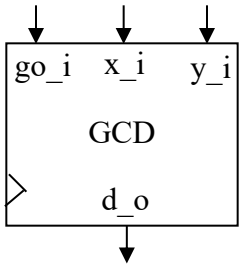


a view inside the controller and datapath

Example: greatest common divisor

- First create algorithm
- Convert algorithm to "complex" state machine
  - Known as FSMD: finite-state machine with datapath
  - Can use templates to perform such conversion

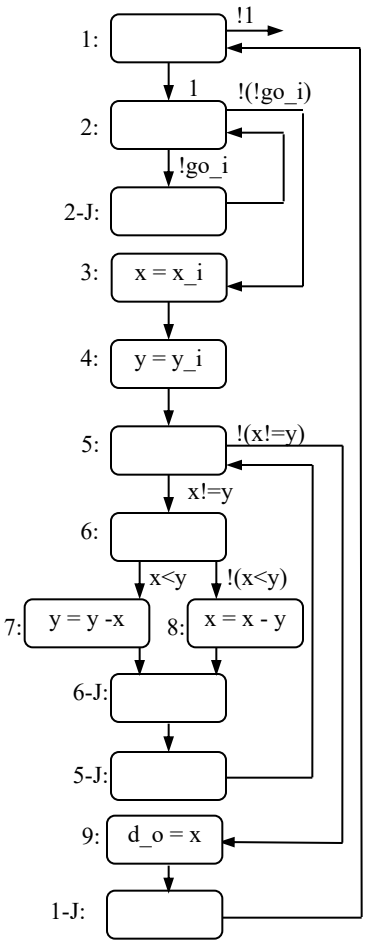
(a) black-box view



(b) desired functionality

```
0: int  x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while  (x != y)  {
6:       if  (x < y)
7:           y = y - x;
         else
8:           x = x - y;
     }
9:   d_o = x;
   }
```

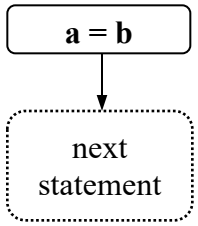
(c) state diagram



State diagram templates

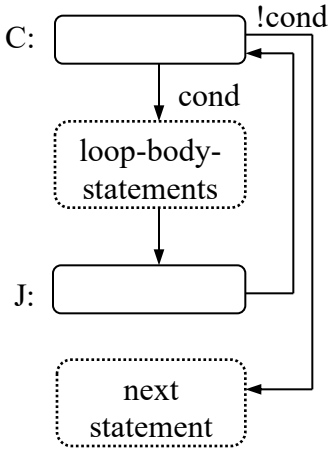
Assignment statement

a = b
next statement



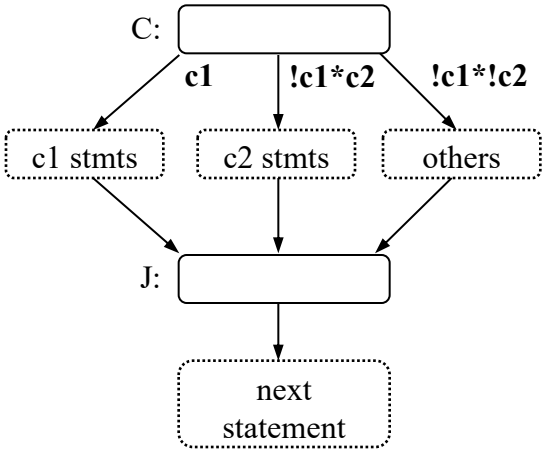
Loop statement

while (cond) {
loop-body-statements
}
next statement



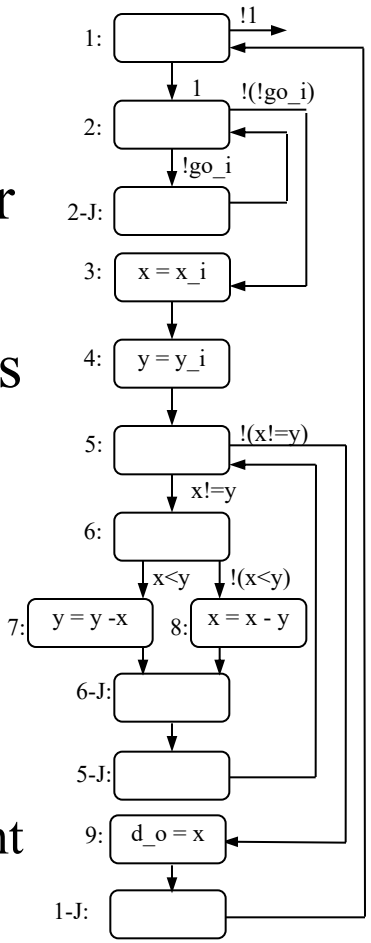
Branch statement

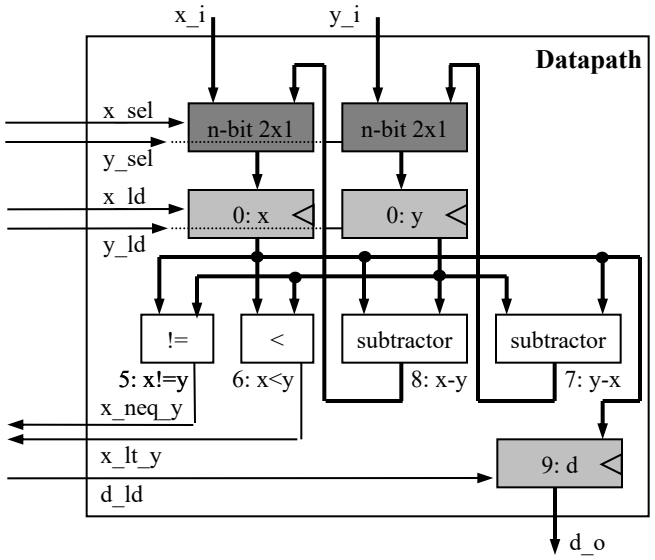
if (c1)
c1 stmts
else if c2
c2 stmts
else
other stmts
next statement



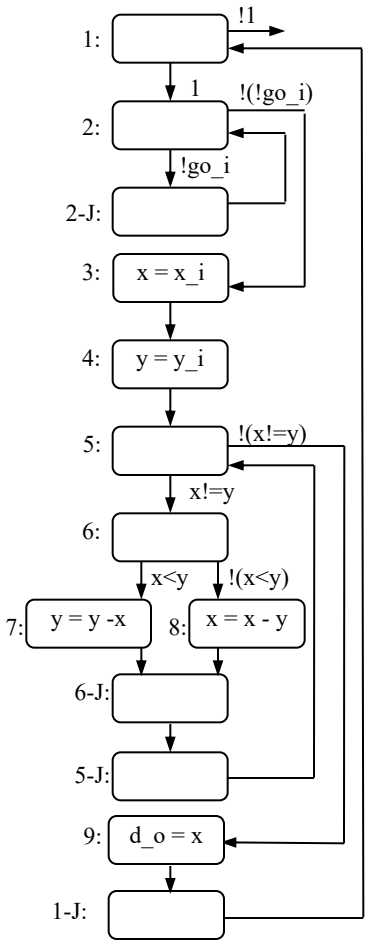
Creating the datapath

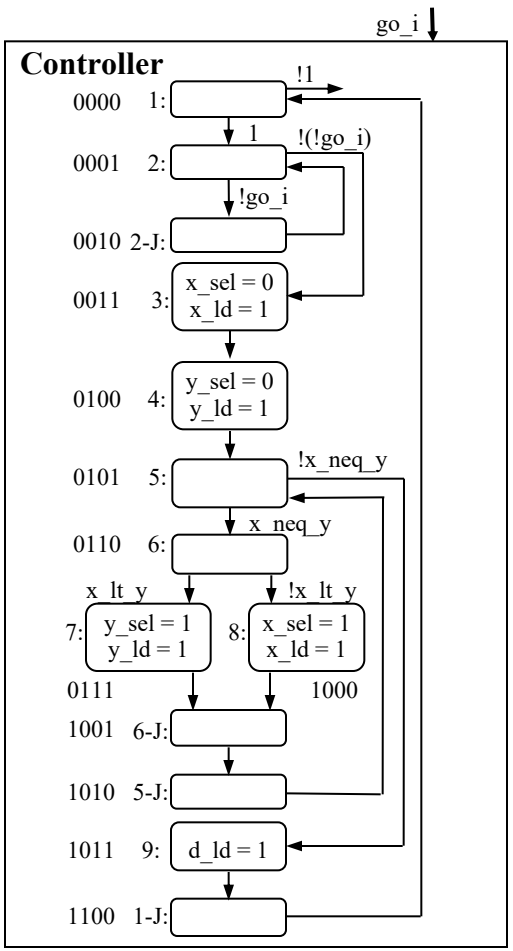
- Create a register for any declared variable
- Create a functional unit for each arithmetic operation
- Connect the ports, registers and functional units
  - Based on reads and writes
  - Use multiplexors for multiple sources
- Create unique identifier
  - for each datapath component control input and output



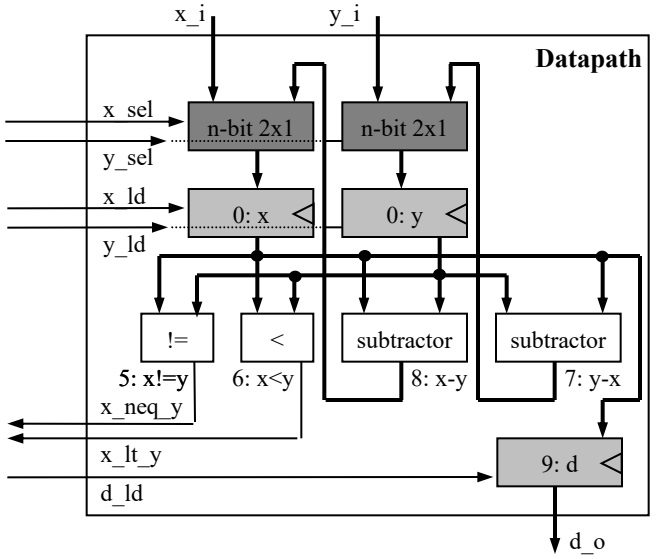


Creating the controller's FSM

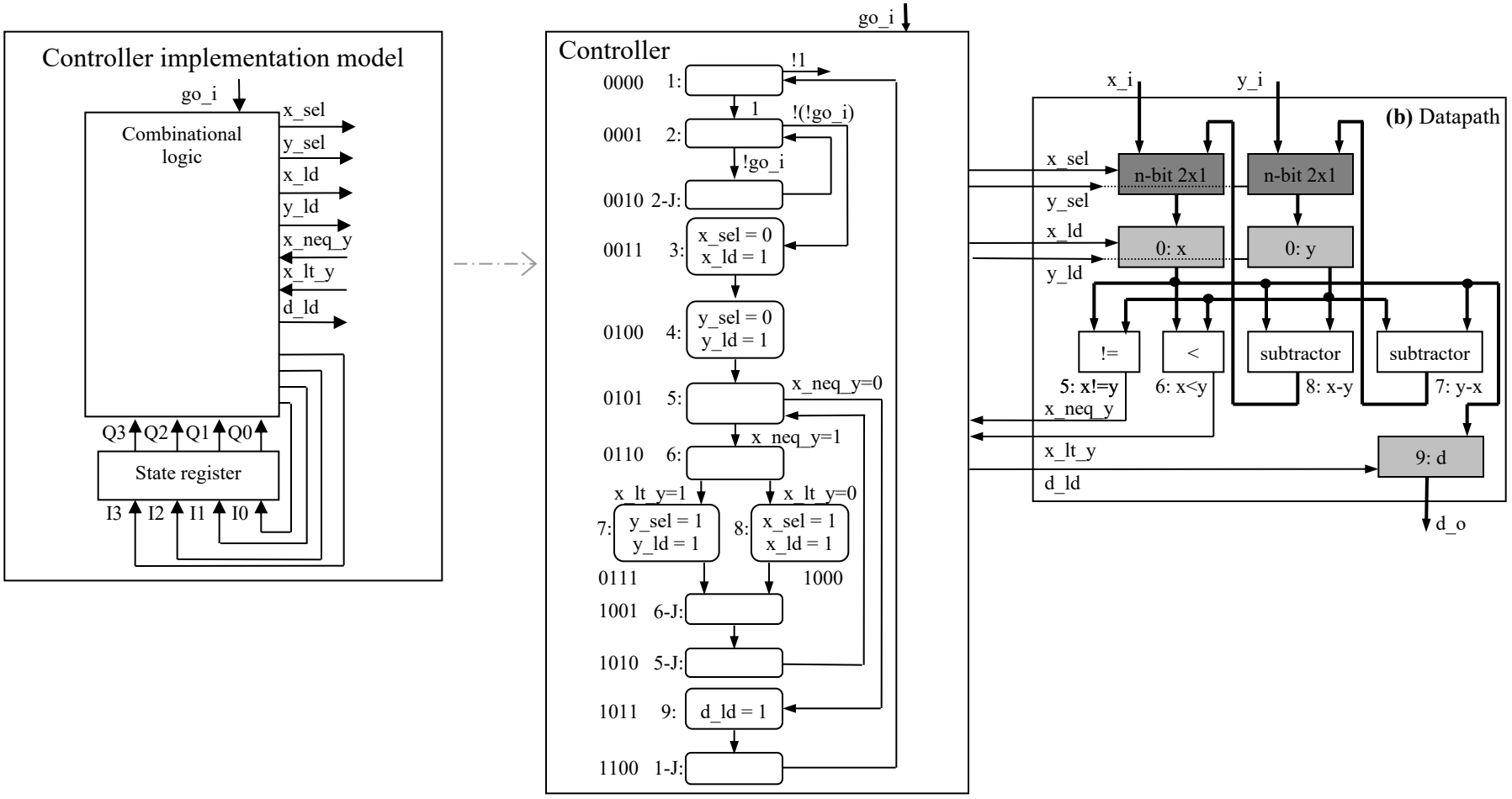




- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations



Splitting into a controller and datapath

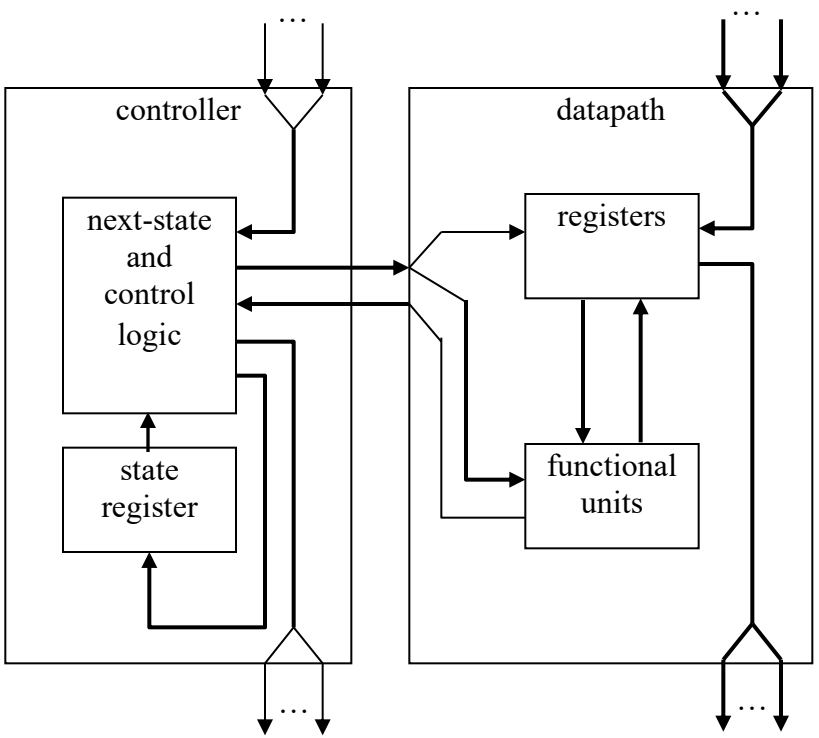


Controller state table for the GCD example

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_neq_y	x_lt_y	go_i	I3	I2	I1	I0	x_sel	y_sel	x_ld	y_ld	d_ld
0	0	0	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	0	1	*	*	0	0	0	1	0	X	X	0	0	0
0	0	0	1	*	*	1	0	0	1	1	X	X	0	0	0
0	0	1	0	*	*	*	0	0	0	1	X	X	0	0	0
0	0	1	1	*	*	*	0	1	0	0	0	X	1	0	0
0	1	0	0	*	*	*	0	1	0	1	X	0	0	1	0
0	1	0	1	0	*	*	1	0	1	1	X	X	0	0	0
0	1	0	1	1	*	*	0	1	1	0	X	X	0	0	0
0	1	1	0	*	0	*	1	0	0	0	X	X	0	0	0
0	1	1	0	*	1	*	0	1	1	1	X	X	0	0	0
0	1	1	1	*	*	*	1	0	0	1	X	1	0	1	0
1	0	0	0	*	*	*	1	0	0	1	1	X	1	0	0
1	0	0	1	*	*	*	1	0	1	0	X	X	0	0	0
1	0	1	0	*	*	*	0	1	0	1	X	X	0	0	0
1	0	1	1	*	*	*	1	1	0	0	X	X	0	0	1
1	1	0	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	0	1	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	0	*	*	*	0	0	0	0	X	X	0	0	0
1	1	1	1	*	*	*	0	0	0	0	X	X	0	0	0

Completing the GCD custom single-purpose processor design

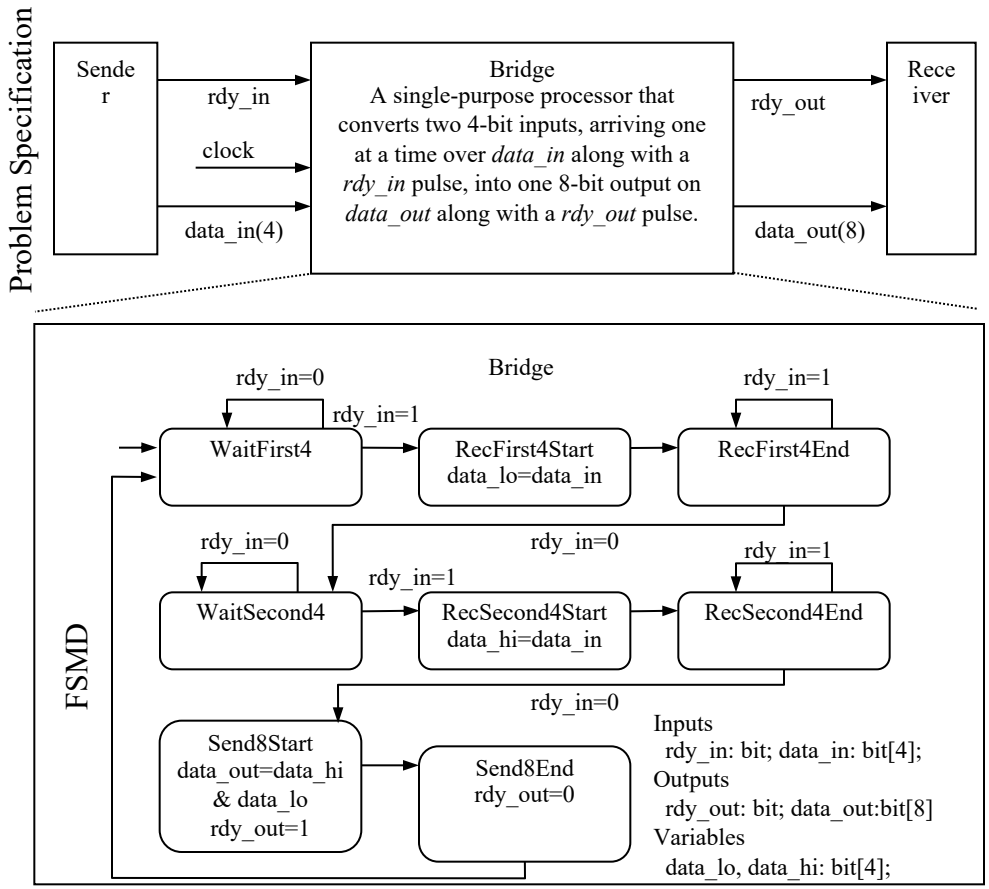
- We finished the datapath
- We have a state table for the next state and control logic
  - All that's left is combinational logic design
- This is *not* an optimized design, but we see the basic steps



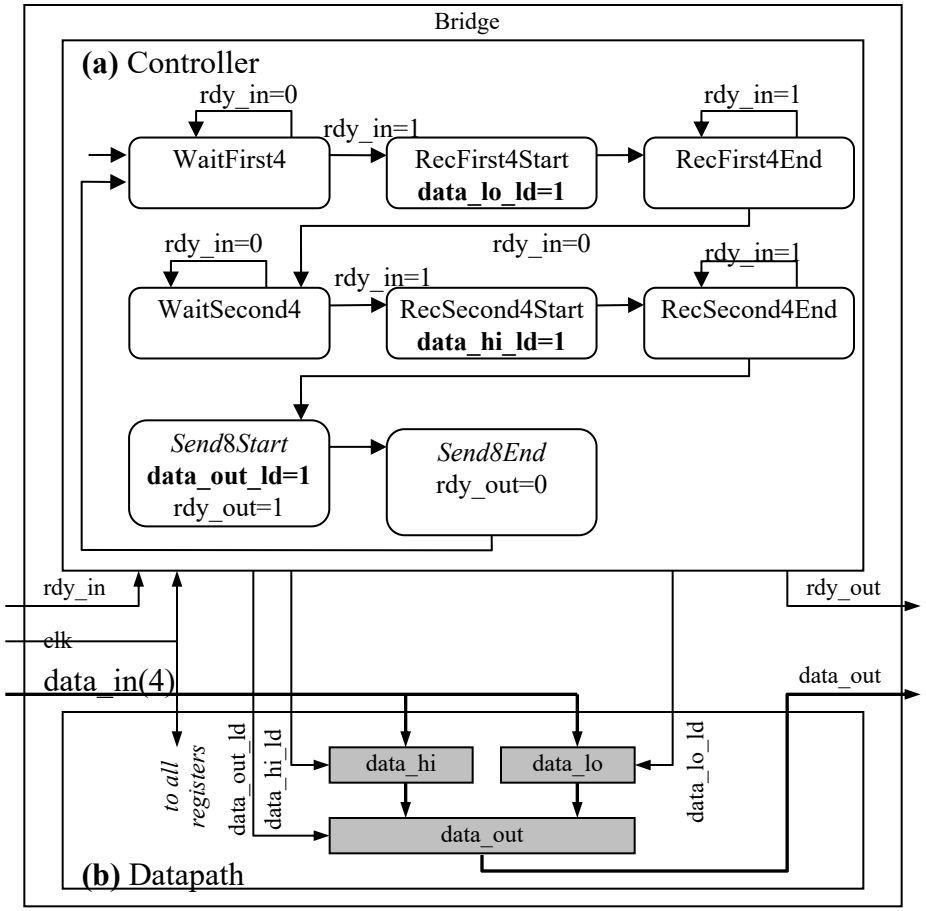
a view inside the controller and datapath

RT-level custom single-purpose processor design

- We often start with a state machine
  - Rather than algorithm
  - Cycle timing often too central to functionality
- Example
  - Bus bridge that converts 4-bit bus to 8-bit bus
  - Start with FSMD
  - Known as register-transfer (RT) level
  - Exercise: complete the design



RT-level custom single-purpose processor design (cont')



Optimizing single-purpose processors

- Optimization is the task of making design metric values the best possible
- Optimization opportunities
  - original program
  - FSMD
  - datapath
  - FSM

Optimizing the original program

- Analyze program attributes and look for areas of possible improvement
  - number of computations
  - size of variable
  - time and space complexity
  - operations used
    - multiplication and division very expensive

Optimizing the original program (cont')

original program

```
 0: int  x, y;
 1: while (1) {
 2:   while (!go_i);
 3:   x = x_i;
 4:   y = y_i;
 5:   while  (x != y)  {
 6:      if  (x < y)
 7:         y = y - x;
        else
 8:         x = x - y;
      }
 9:   d_o = x;
   }
```

replace the subtraction operation(s) with modulo operation in order to speed up program

optimized program

```
 0: int  x, y, r;
 1: while (1) {
 2:   while (!go_i);
      // x must be the larger number
 3:   if (x_i >= y_i)  {
 4:      x=x_i;
 5:      y=y_i;
      }
 6:   else {
 7:      x=y_i;
 8:      y=x_i;
      }
 9:   while  (y != 0)  {
10:      r = x % y;
11:      x = y;
12:      y = r;
      }
13:   d_o = x;
    }
```

GCD(42, 8) - 9 iterations to complete the loop

x and y values evaluated as follows : (42, 8), (43, 8), (26,8), (18,8), (10, 8), (2,8), (2,6), (2,4), (2,2).

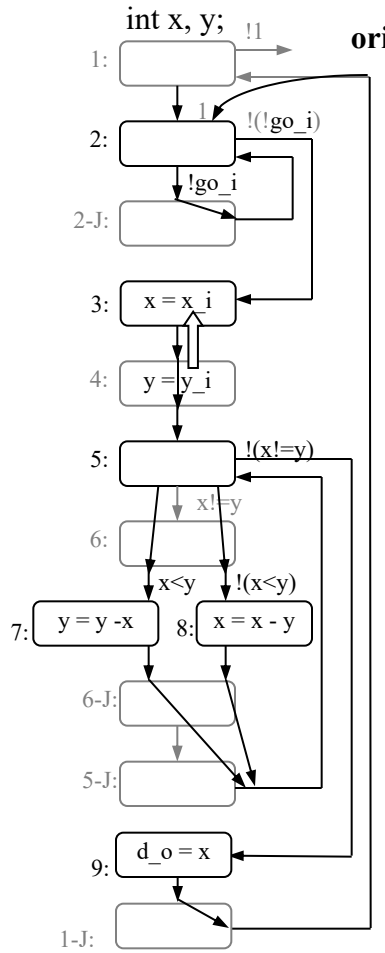
GCD(42,8) - 3 iterations to complete the loop

x and y values evaluated as follows: (42, 8), (8,2), (2,0)

Optimizing the FSMD

- Areas of possible improvements
  - merge states
    - states with constants on transitions can be eliminated, transition taken is already known
    - states with independent operations can be merged
  - separate states
    - states which require complex operations (a*b*c*d) can be broken into smaller states to reduce hardware size
  - scheduling

Optimizing the FSMD (cont.)



original FSMD

eliminate state 1 – transitions have constant values

merge state 2 and state 2J – no loop operation in between them

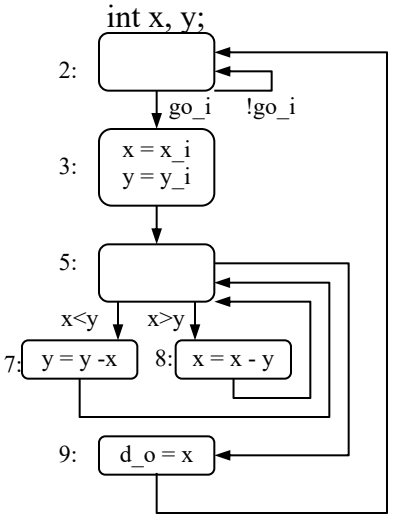
merge state 3 and state 4 – assignment operations are independent of one another

merge state 5 and state 6 – transitions from state 6 can be done in state 5

eliminate state 5J and 6J – transitions from each state can be done from state 7 and state 8, respectively

eliminate state 1-J – transition from state 1-J can be done directly from state 9

optimized FSMD



Optimizing the datapath

- Sharing of functional units
  - one-to-one mapping, as done previously, is not necessary
  - if same operation occurs in different states, they can share a single functional unit
- Multi-functional units
  - ALUs support a variety of operations, it can be shared among operations occurring in different states

Optimizing the FSM

- State encoding
  - task of assigning a unique bit pattern to each state in an FSM
  - size of state register and combinational logic vary
  - can be treated as an ordering problem
- State minimization
  - task of merging equivalent states into a single state
    - state equivalent if for all possible input combinations the two states generate the same outputs and transitions to the next same state

Summary

- Custom single-purpose processors
  - Straightforward design techniques
  - Can be built to execute algorithms
  - Typically start with FSMD
  - CAD tools can be of great assistance

Embedded Systems Design: A Unified Hardware/Software Introduction

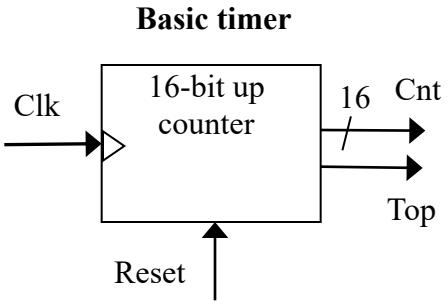
Chapter 4 Standard Single Purpose Processors: Peripherals

Introduction

- Single-purpose processors
  - Performs specific computation task
  - Custom single-purpose processors
    - Designed by us for a unique task
  - *Standard* single-purpose processors
    - "Off-the-shelf" -- pre-designed for a common task
    - a.k.a., peripherals
    - serial transmission
    - analog/digital conversions

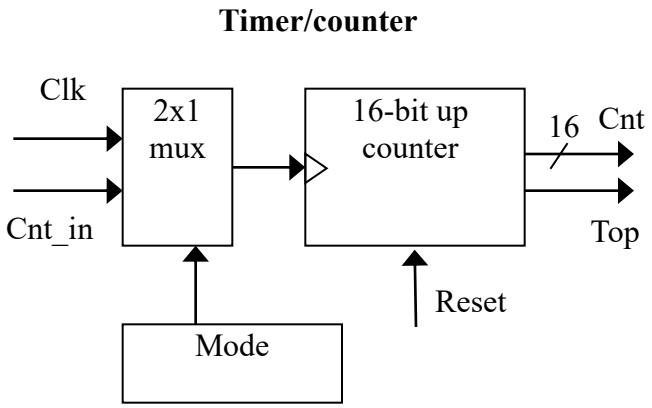
Timers, counters, watchdog timers

- Timer: measures time intervals
  - To generate timed output events
    - e.g., hold traffic light green for 10 s
  - To measure input events
    - e.g., measure a car's speed
- Based on counting clock pulses
    - E.g., let Clk period be 10 ns
    - And we count 20,000 Clk pulses
    - Then 200 microseconds have passed
    - 16-bit counter would count up to 65,535*10 ns = 655.35 microsec., resolution = 10 ns
    - Top: indicates top count reached, wrap-around



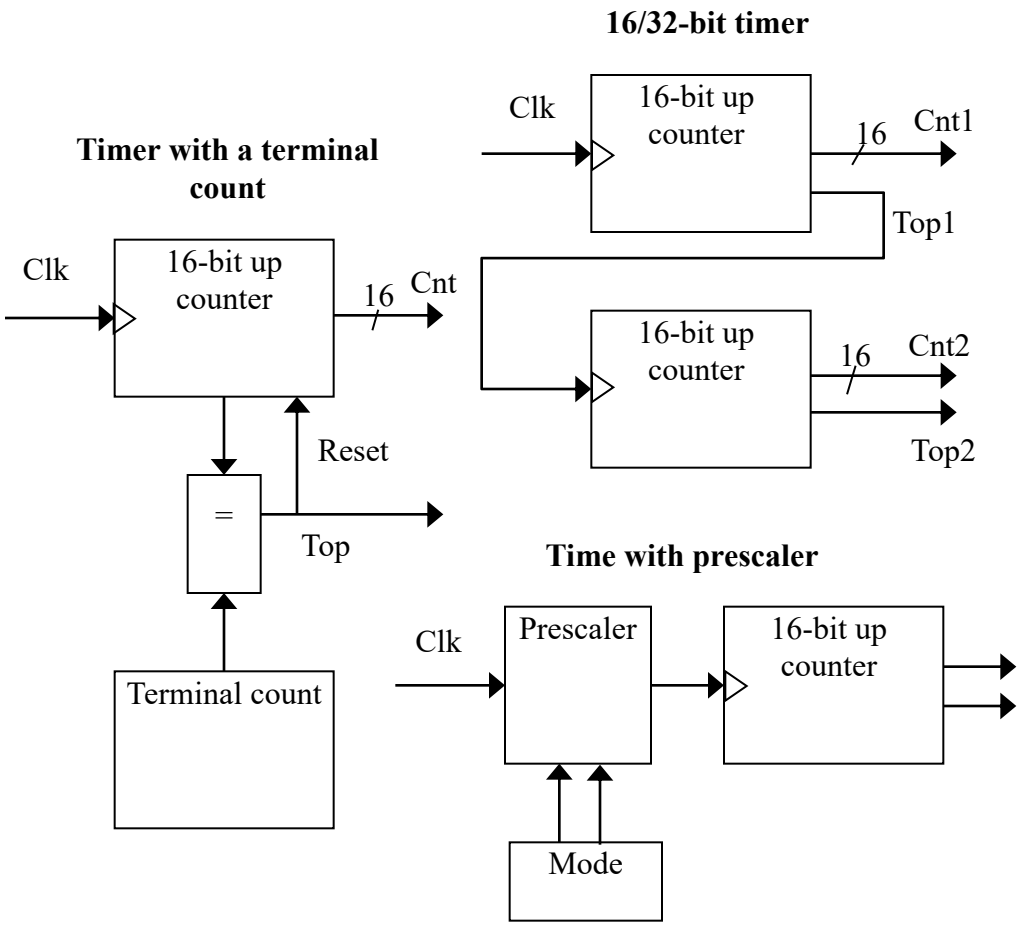
Counters

- Counter: like a timer, but counts pulses on a general input signal rather than clock
  - e.g., count cars passing over a sensor
  - Can often configure device as either a timer or counter

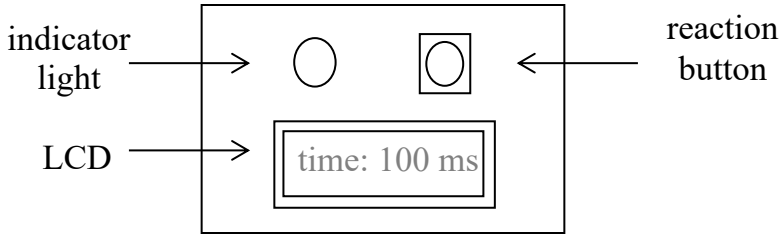


Other timer structures

- Interval timer
  - Indicates when desired time interval has passed
  - We set terminal count to desired interval
    - *Number of clock cycles = Desired time interval / Clock period*
- Cascaded counters
- Prescaler
  - Divides clock
  - Increases range, decreases resolution



Example: Reaction Timer

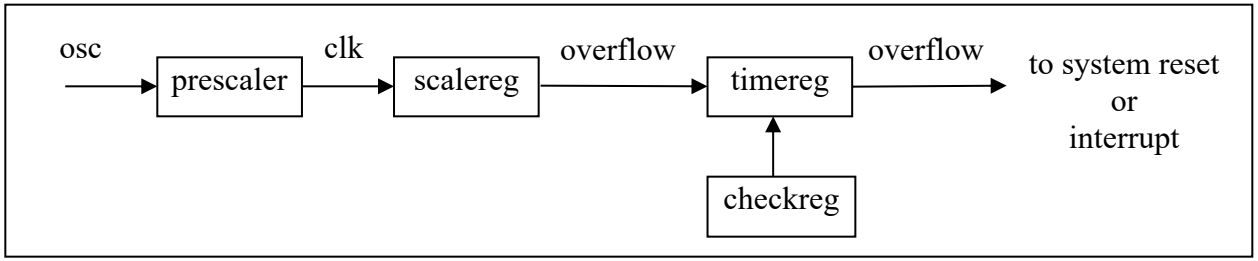


- Measure time between turning light on and user pushing button
  - 16-bit timer, clk period is 83.33 ns, counter increments every 6 cycles
  - Resolution = 6*83.33=0.5 microsec.
  - Range = 65535*0.5 microseconds = 32.77 milliseconds
  - Want program to count millisec., so initialize counter to 65535 – 1000/0.5 = 63535

```
/* main.c */

#define MS_INIT       63535
void main(void){
  int count_milliseconds = 0;

  configure timer mode
  set Cnt to MS_INIT

  wait a random amount of time
  turn on indicator light
  start timer

while (user has not pushed reaction button){
  if(Top) {
    stop timer
    set Cnt to MS_INIT
    start timer
    reset Top
    count_milliseconds++;
  }
}
turn light off
printf("time:  %i ms", count_milliseconds);
}
```

Watchdog timer

- Must reset timer every X time unit, else timer generates a signal
- Common use: detect failure, self-reset
- Another use: timeouts
  - e.g., ATM machine
  - 16-bit timer, 2 microsec. resolution
  - *timereg* value = $2*(2^{16}-1)–X = 131070–X$
  - For 2 min., X = 120,000 microsec.

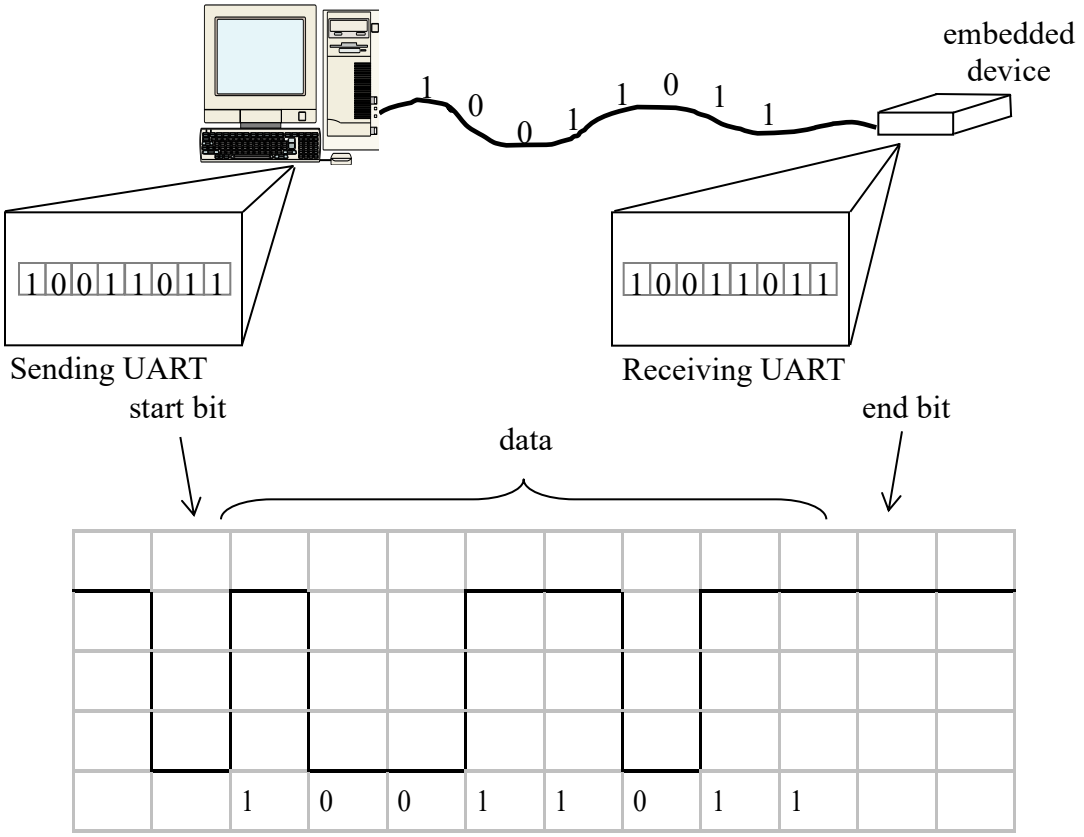



```
/* main.c */

main(){
  wait until card inserted
  call watchdog_reset_routine

  while(transaction in progress){
    if(button pressed){
      perform corresponding action
      call watchdog_reset_routine
    }

/* if watchdog_reset_routine not called every
< 2 minutes, interrupt_service_routine is
called */
}
```

```
watchdog_reset_routine(){
/* checkreg is set so we can load value into
timereg.  Zero is loaded into scalereg and
11070 is loaded into timereg */

  checkreg = 1
  scalereg = 0
  timereg  = 11070
}

void interrupt_service_routine(){
  eject card
  reset screen
}
```

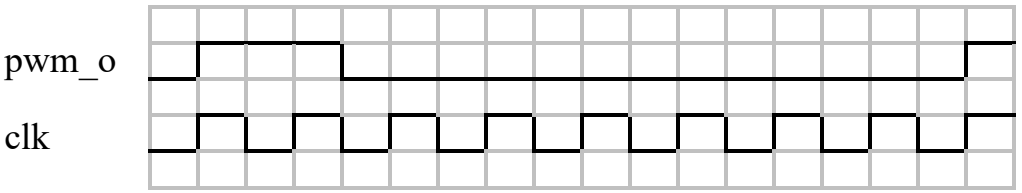
Serial Transmission Using UARTs

- UART: Universal Asynchronous Receiver Transmitter
  - Takes parallel data and transmits serially
  - Receives serial data and converts to parallel
- Parity: extra bit for simple error checking
- Start bit, stop bit
- Baud rate
  - signal changes per second
  - bit rate usually higher

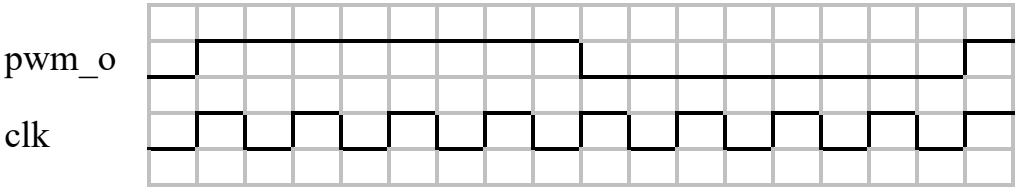


Pulse width modulator

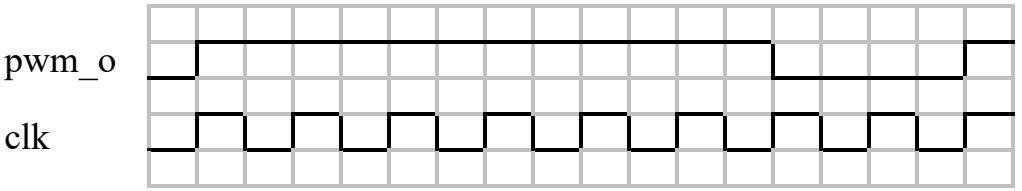
- Generates pulses with specific high/low times
- Duty cycle: % time high
  - Square wave: 50% duty cycle
- Common use: control average voltage to electric device
  - Simpler than DC-DC converter or digital-analog converter
  - DC motor speed, dimmer lights
- Another use: encode commands, receiver uses timer to decode



25% duty cycle – average pwm_o is 1.25V

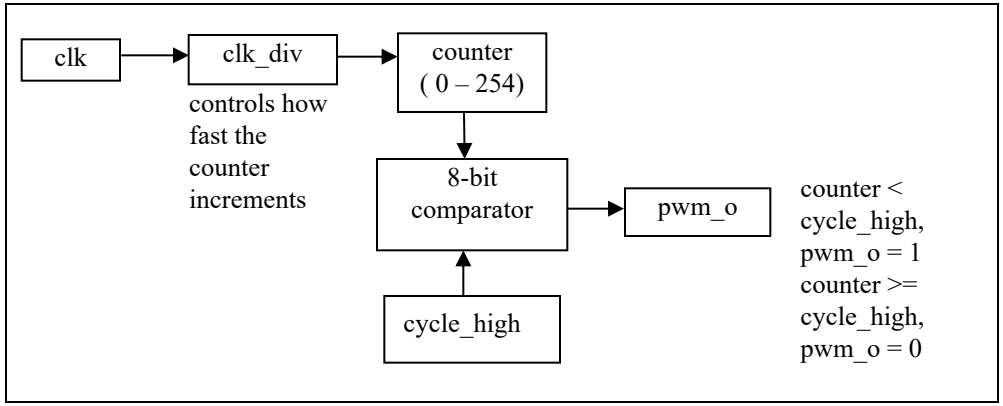


50% duty cycle – average pwm_o is 2.5V.



75% duty cycle – average pwm_o is 3.75V.

Controlling a DC motor with a PWM



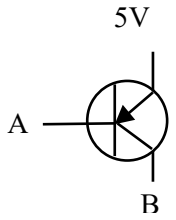
Internal Structure of PWM

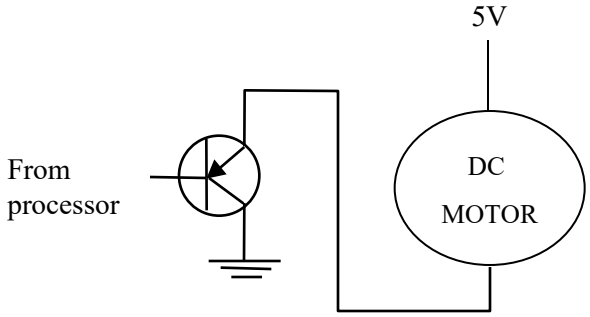
Input Voltage	% of Maximum Voltage Applied	RPM of DC Motor
0	0	0
2.5	50	1840
3.75	75	6900
5.0	100	9200

Relationship between applied voltage and speed of the DC Motor

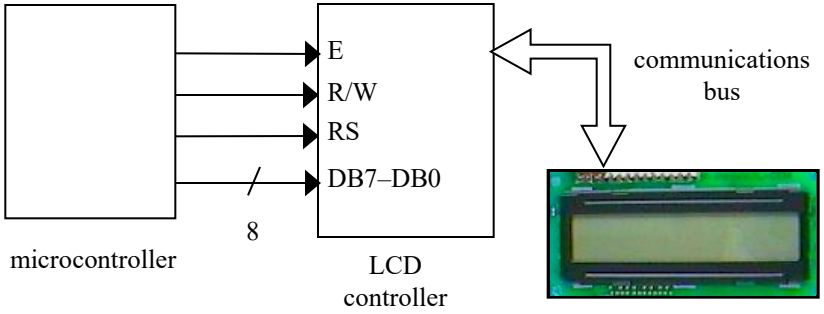
```
void main(void){

   /* controls period */
   PWMP = 0xff;
   /* controls duty cycle */
   PWM1 = 0x7f;

   while(1){};
 }
```

The PWM alone cannot drive the DC motor, a possible way to implement a driver is shown below using an MJE3055T NPN transistor.





LCD controller

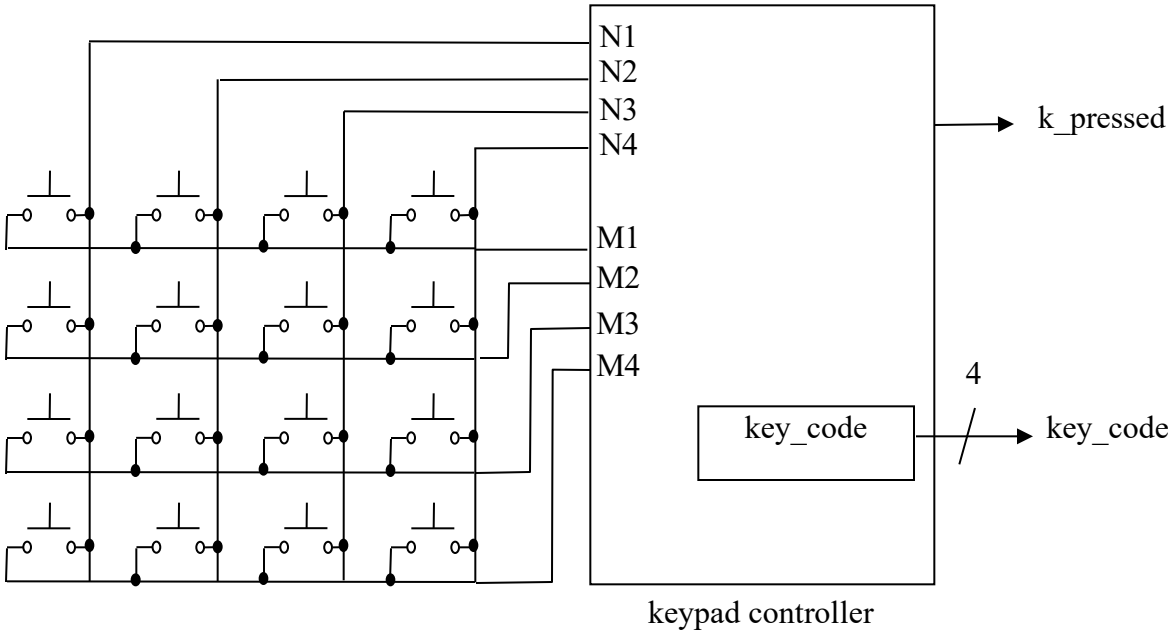



```
void WriteChar(char c){

  RS = 1;                  /* indicate data being sent */
  DATA_BUS = c;            /* send data to LCD */
  EnableLCD(45);           /* toggle the LCD with appropriate delay */
}
```

CODES	
I/D = 1 cursor moves left	DL = 1 8-bit
I/D = 0 cursor moves right	DL = 0 4-bit
S = 1 with display shift	N = 1 2 rows
S/C =1 display shift	N = 0 1 row
S/C = 0 cursor movement	F = 1 5x10 dots
R/L = 1 shift to right	F = 0 5x7 dots
R/L = 0 shift to left	

RS	R/W	DB_7	DB_6	DB_5	DB_4	DB_3	DB_2	DB_1	DB_0	Description
0	0	0	0	0	0	0	0	0	1	Clears all display, return cursor home
0	0	0	0	0	0	0	0	1	*	Returns cursor home
0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and/or specifies not to shift display
0	0	0	0	0	0	1	D	C	B	ON/OFF of all display(D), cursor ON/OFF (C), and blink position (B)
0	0	0	0	0	1	S/C	R/L	*	*	Move cursor and shifts display
0	0	0	0	1	DL	N	F	*	*	Sets interface data length, number of display lines, and character font
1	0	WRITE DATA								Writes Data

Keypad controller

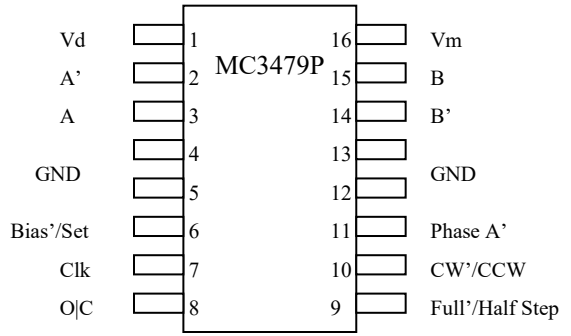


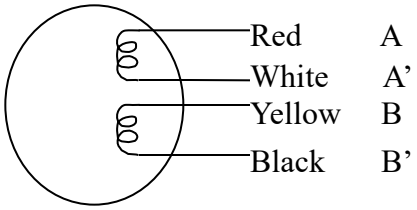
N=4, M=4

Stepper motor controller

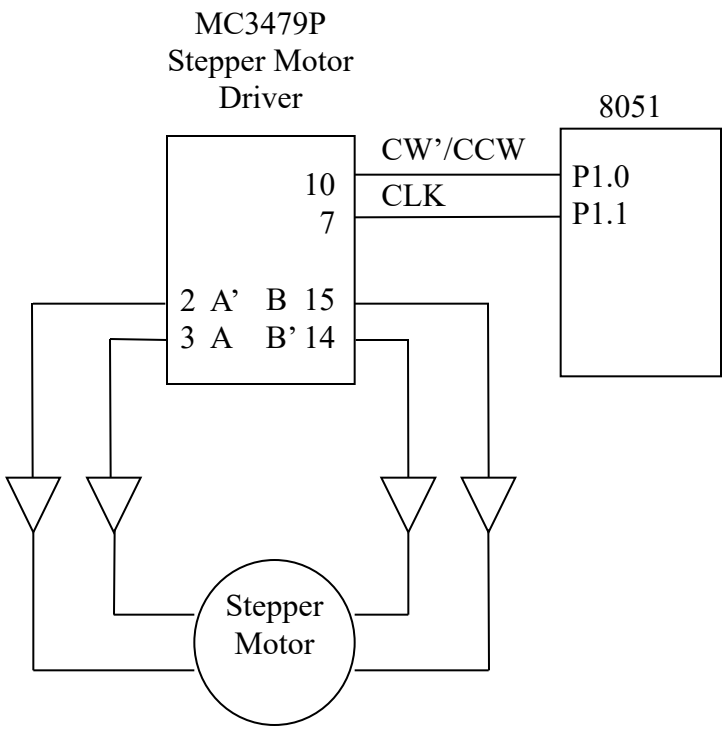
- Stepper motor: rotates fixed number of degrees when given a "step" signal
  - In contrast, DC motor just rotates when power applied, coasts to stop
- Rotation achieved by applying specific voltage sequence to coils
- Controller greatly simplifies this

Sequence	A	B	A'	B'
1	+	+	-	-
2	-	+	+	-
3	-	-	+	+
4	+	-	-	+
5	+	+	-	-





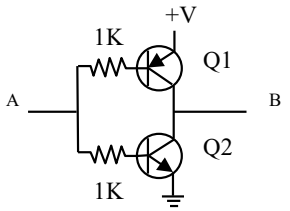
Stepper motor with controller (driver)



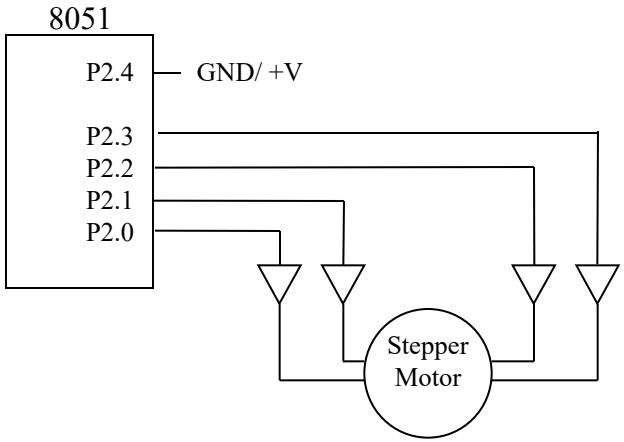

```
/* main.c */

sbit clk=P1^1;
sbit cw=P1^0;

void delay(void){
  int i, j;
  for (i=0; i<1000; i++)
    for ( j=0; j<50; j++)
      i = i + 0;
}
```

```
void main(void){

  */turn the motor forward */
  cw=0;            /* set direction */
  clk=0;           /* pulse clock */
  delay();
  clk=1;

  /*turn the motor backwards */
  cw=1;            /* set direction */
  clk=0;           /* pulse clock */
  delay();
  clk=1;

}
```

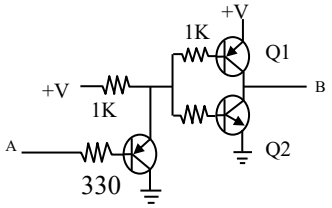
The output pins on the stepper motor driver do not provide enough current to drive the stepper motor. To amplify the current, a buffer is needed. One possible implementation of the buffers is pictured to the left. Q1 is an MJE3055T NPN transistor and Q2 is an MJE2955T PNP transistor. A is connected to the 8051 microcontroller and B is connected to the stepper motor.



Stepper motor without controller (driver)



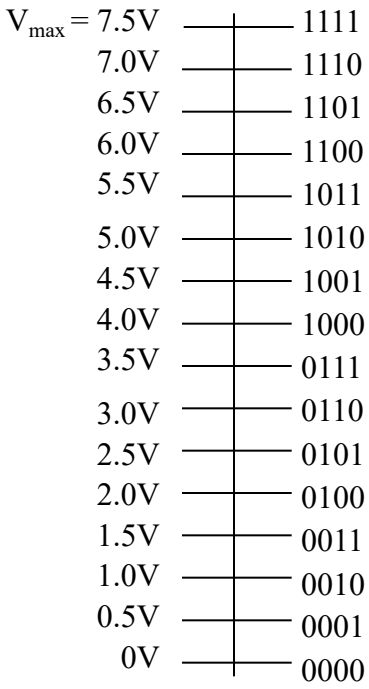
A possible way to implement the buffers is located below. The 8051 alone cannot drive the stepper motor, so several transistors were added to increase the current going to the stepper motor. Q1 are MJE3055T NPN transistors and Q3 is an MJE2955T PNP transistor. A is connected to the 8051 microcontroller and B is connected to the stepper motor.



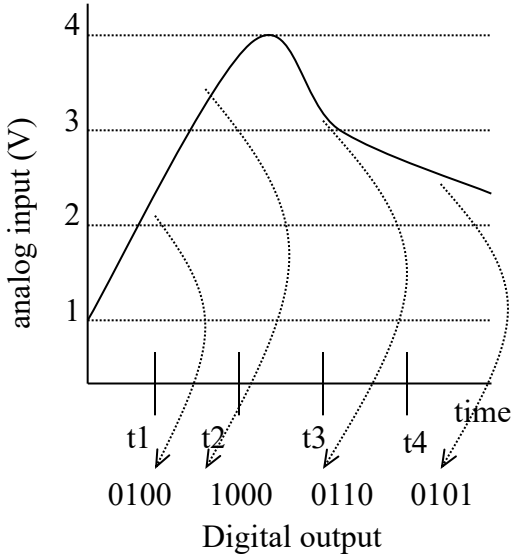

```
/*main.c*/
sbit notA=P2^0;
sbit isA=P2^1;
sbit notB=P2^2;
sbit isB=P2^3;
sbit dir=P2^4;

void delay(){
  int a, b;
  for(a=0; a<5000; a++)
    for(b=0; b<10000; b++)
      a=a+0;
}

void move(int dir, int steps) {
int y, z;
  /* clockwise movement */
  if(dir == 1){
    for(y=0; y<=steps; y++){
      for(z=0; z<=19; z+4){
        isA=lookup[z];
        isB=lookup[z+1];
        notA=lookup[z+2];
        notB=lookup[z+3];
        delay();
      }
    }
  }
```

```
/* counter clockwise movement */
 if(dir==0){
    for(y=0;  y<=step; y++){
      for(z=19; z>=0; z - 4){
        isA=lookup[z];
        isB=lookup[z-1];
        notA=lookup[z -2];
        notB=lookup[z-3];
       delay( );
      }
    }
  }
}
void main( ){
  int z;
  int lookup[20] = {
    1,  1,  0,  0,
    0,  1,  1,  0,
    0,  0,  1,  1,
    1,  0,  0,  1,
    1,  1,  0,  0  };
  while(1){
     /*move forward, 15 degrees (2 steps) */
    move(1, 2);
    /* move backwards,  7.5 degrees (1step)*/
    move(0, 1);
  }
}
```

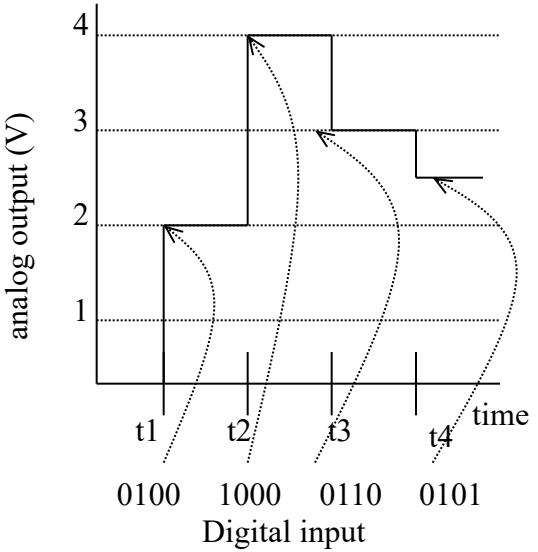
Analog-to-digital converters



proportionality



analog to digital



digital to analog

Digital-to-analog conversion using successive approximation

Given an analog input signal whose voltage should range from 0 to 15 volts, and an 8-bit digital encoding, calculate the correct encoding for 5 volts. Then trace the successive-approximation approach to find the correct encoding.

5/15 = d/(28-1)
d= 85

Encoding: 01010101

Successive-approximation method

½(V_{max} – V_{min}) = 7.5 volts
V_{max} = 7.5 volts.

0	0	0	0	0	0	0	0

½(7.5 + 0) = 3.75 volts
V_{min} = 3.75 volts.

0	1	0	0	0	0	0	0

½(7.5 + 3.75) = 5.63 volts
V_{max} = 5.63 volts

0	1	0	0	0	0	0	0

½(5.63 + 3.75) = 4.69 volts
V_{min} = 4.69 volts.

0	1	0	1	0	0	0	0

½(5.63 + 4.69) = 5.16 volts
V_{max} = 5.16 volts.

0	1	0	1	0	0	0	0

½(5.16 + 4.69) = 4.93 volts
V_{min} = 4.93 volts.

0	1	0	1	0	1	0	0

½(5.16 + 4.93) = 5.05 volts
V_{max} = 5.05 volts.

0	1	0	1	0	1	0	0

½(5.05 + 4.93) = 4.99 volts

0	1	0	1	0	1	0	1

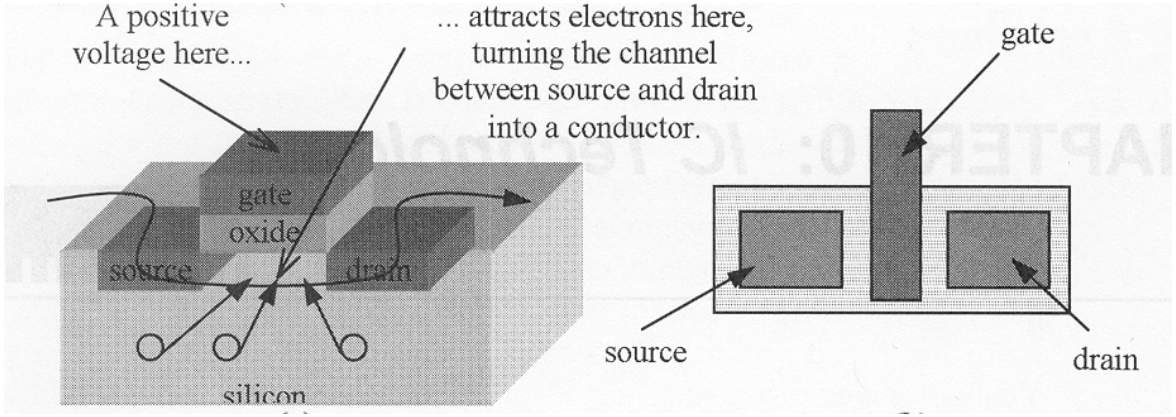
Chapter 10: IC Technology

Outline

- Anatomy of integrated circuits
- Full-Custom (VLSI) IC Technology
- Semi-Custom (ASIC) IC Technology
- Programmable Logic Device (PLD) IC Technology

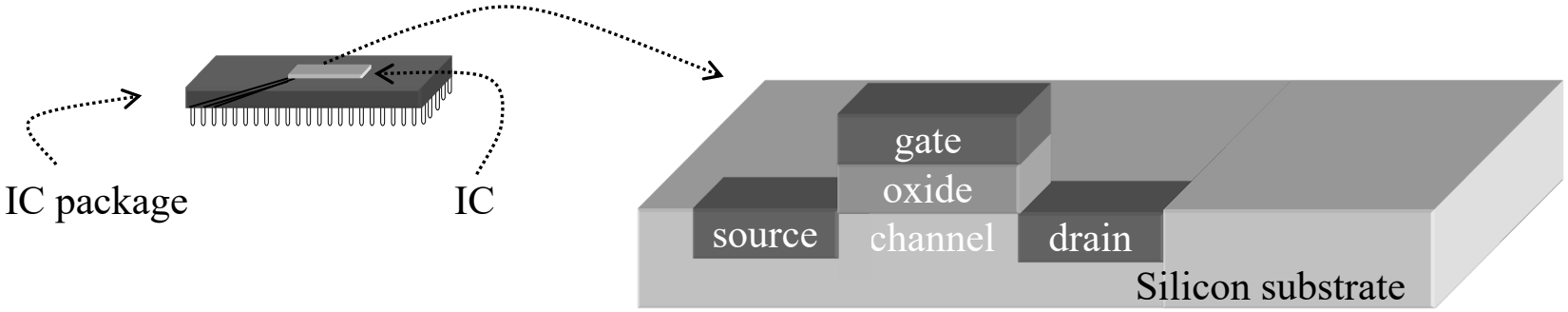
CMOS transistor

- Source, Drain
  - Diffusion area where electrons can flow
  - Can be connected to metal contacts (via's)
- Gate
  - Polysilicon area where control voltage is applied
- Oxide
  - Si O_2 Insulator so the gate voltage can't leak



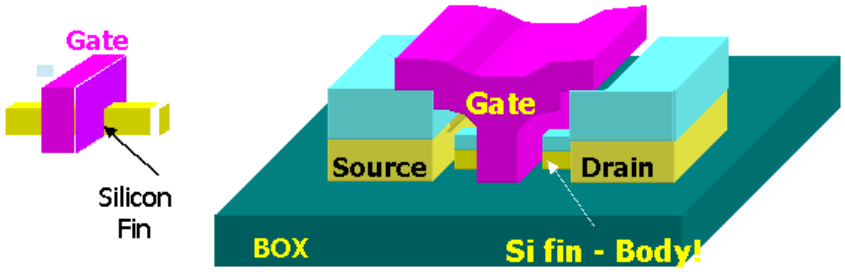
End of the Moore's Law?

- Every dimension of the MOSFET has to scale
  - (PMOS) Gate oxide has to scale down to
    - Increase gate capacitance
    - Reduce leakage current from S to D
    - Pinch off current from source to drain
  - Current gate oxide thickness is about 2.5-3nm
- That's about 25 atoms!!!



Proposed Structures: FinFET

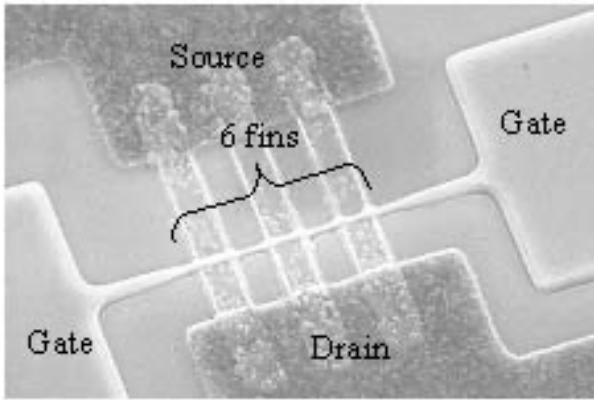
Body is a Thin Silicon Film
Double Gate Structure + Raised Source Drain



X. Huang, et al, 1999 IEDM, p.67-70

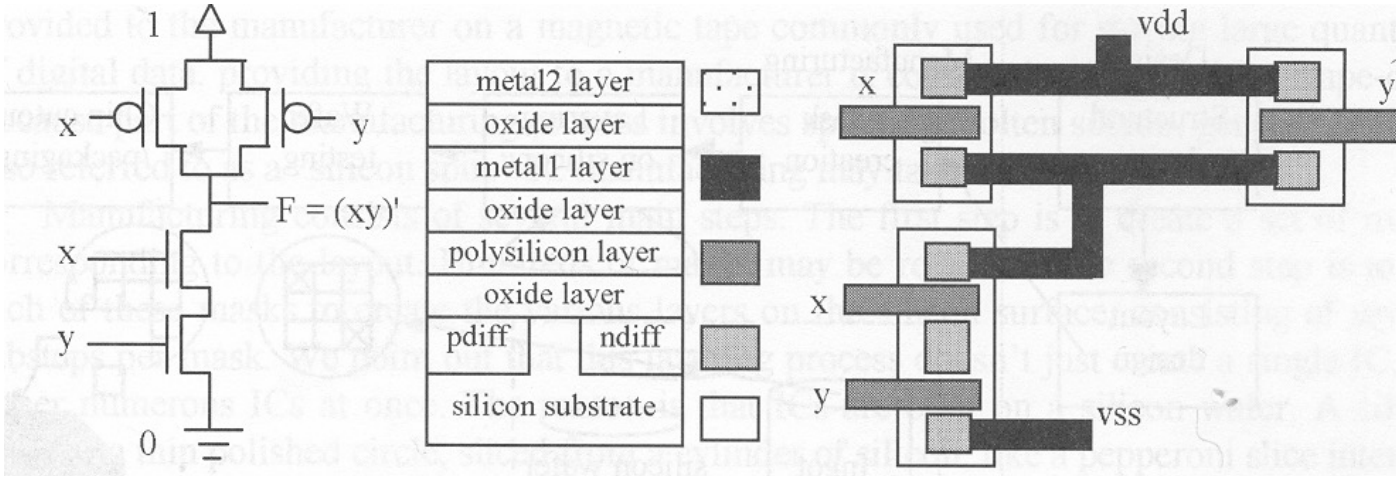
20Ghz +

- FinFET has been manufactured to 18nm
  - Still acts as a very good transistor
- Simulation shown that it can be scaled to 10nm
  - Quantum effect start to kick in
    - Reduce mobility by ~10%
  - Ballistic transport become significant
    - Increase current by about ~20%



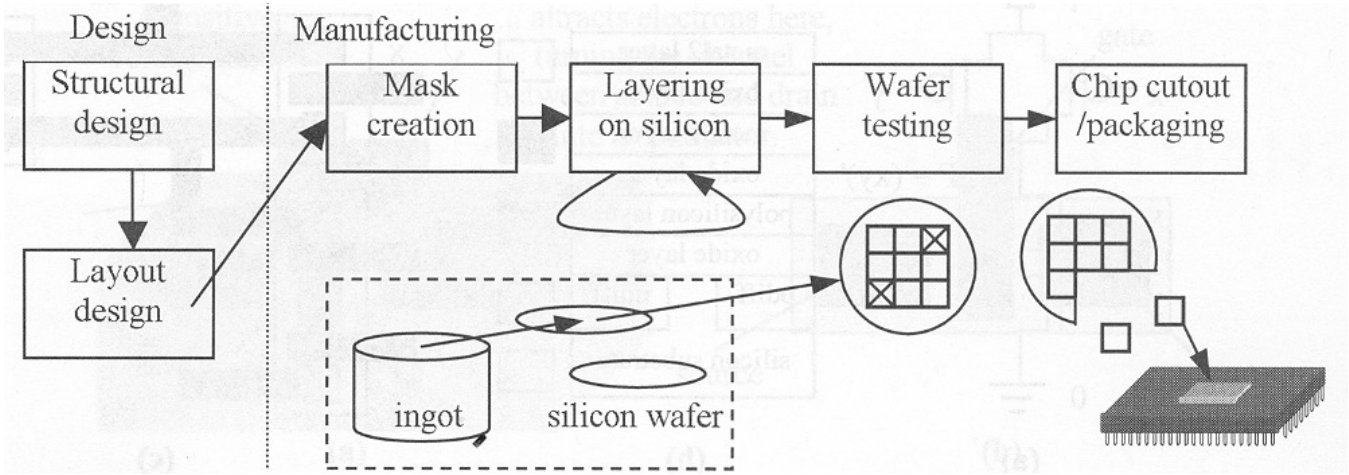
NAND

- Metal layers for routing (~10)
- PMOS don't like 0
- NMOS don't like 1
- A stick diagram form the basis for mask sets



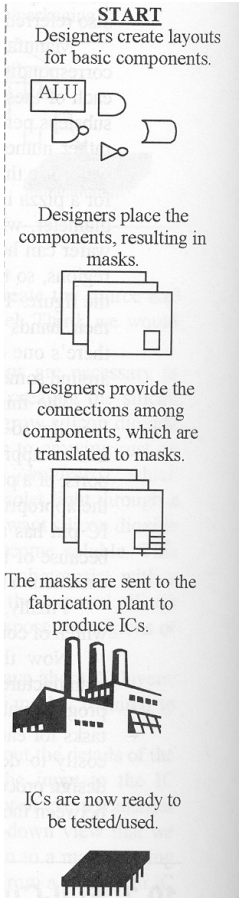
Silicon manufacturing steps

- Tape out
  - Send design to manufacturing
- Spin
  - One time through the manufacturing process
- Photolithography
  - Drawing patterns by using photoresist to form barriers for deposition



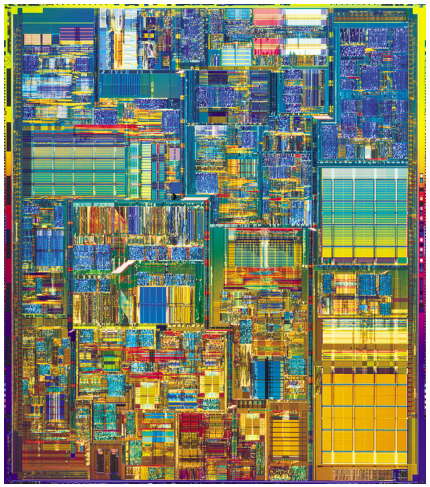
Full Custom

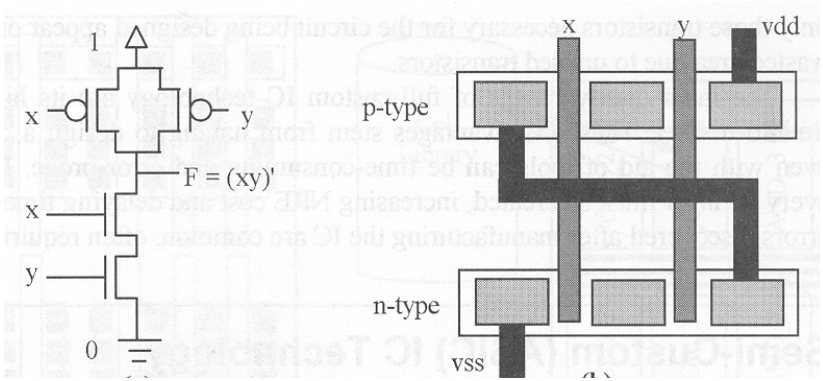
- Very Large Scale Integration (VLSI)
- Placement
  - Place and orient transistors
- Routing
  - Connect transistors
- Sizing
  - Make fat, fast wires or thin, slow wires
  - May also need to size buffer
- Design Rules
  - "simple" rules for correct circuit function
    - Metal/metal spacing, min poly width…

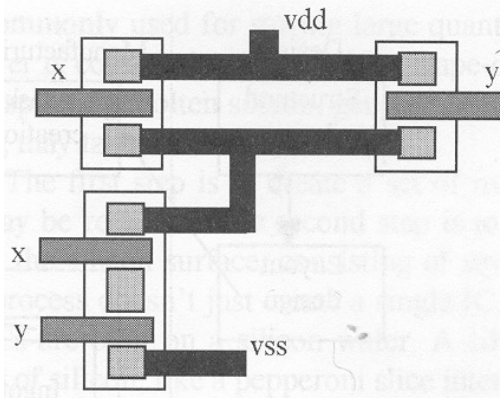


Full Custom

- Best size, power, performance
- Hand design
  - Horrible time-to-market/flexibility/NRE cost…
  - Reserve for the most important units in a processor
    - ALU, Instruction fetch…
- Physical design tools
  - Less optimal, but faster…

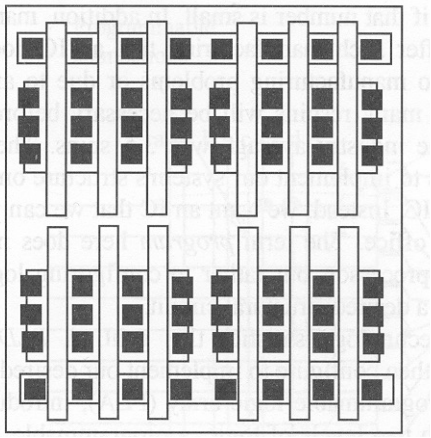




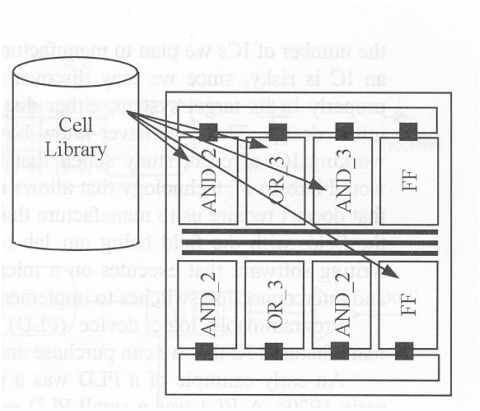


Semi-Custom

- Gate Array
  - Array of prefabricated gates
  - “place” and route
  - Higher density, faster time-to-market
  - Does not integrate as well with full-custom

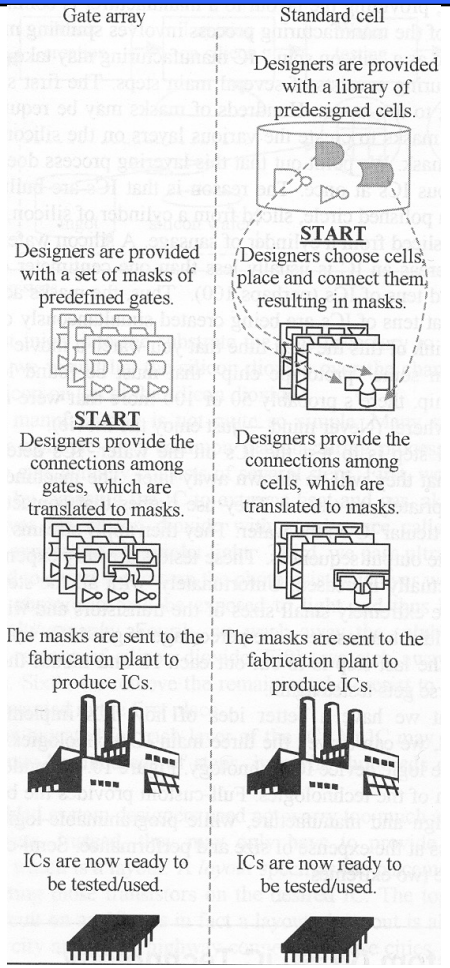


- Standard Cell
  - A library of pre-designed cell
  - Place and route
  - Lower density, higher complexity
  - Integrate great with full-custom

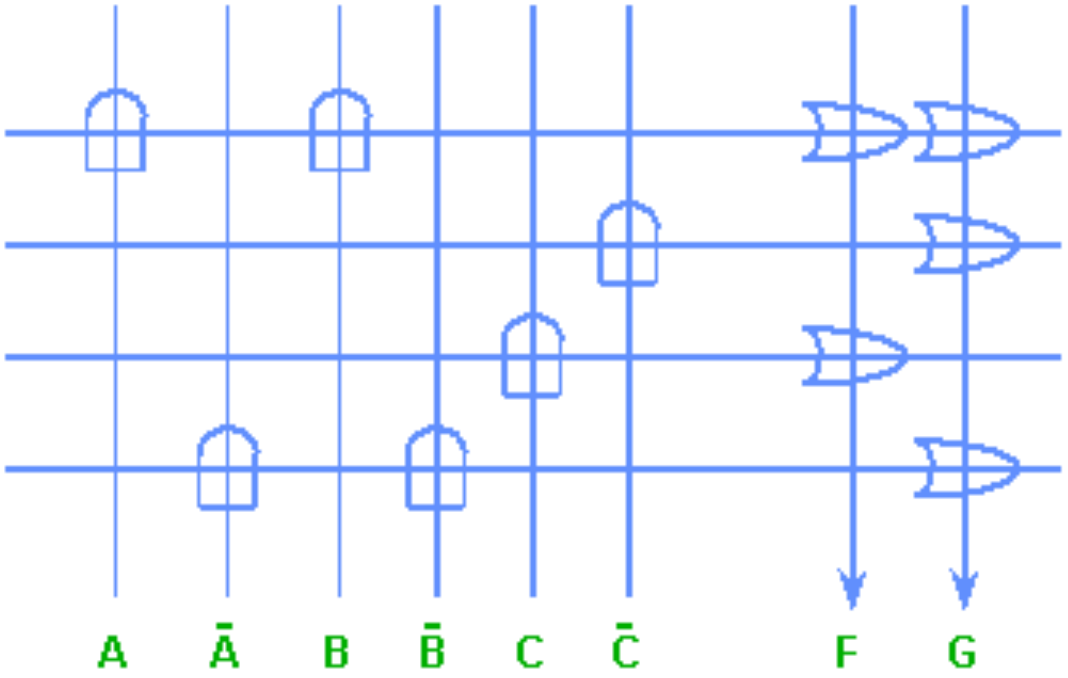


Semi-Custom

- Most popular design style
- Jack of all trade
  - Good
    - Power, time-to-market, performance, NRE cost, per-unit cost, area…
- Master of none
  - Integrate with full custom for critical regions of design



Programmable Logic Array (PLA)



PLA exploits structure of expression.

$F = AB + C$

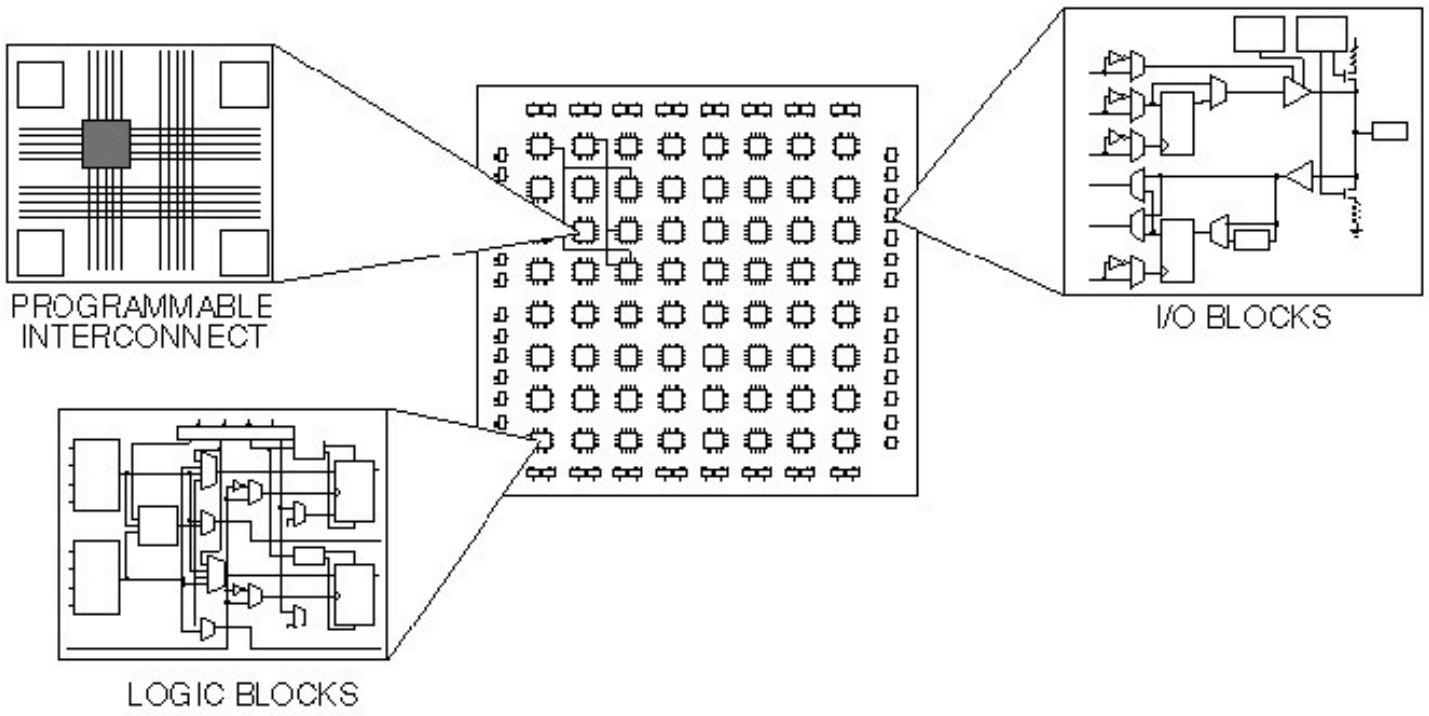
$G = AB + \bar{C} + \bar{A}\bar{B}$

Programmable Logic Device

- Programmable Logic Device
  - Programmable Logic Array, Programmable Array Logic, Field Programmable Gate Array
- All layers already exist
  - Designers can purchase an IC
  - To implement desired functionality
    - Connections on the IC are either created or destroyed to implement
- Benefits
  - Very low NRE costs
  - Great time to market
- Drawback
  - High unit cost, bad for large volume
  - Power
    - Except special PLA
  - slower

1600 usable gate, 7.5 ns
$7 list price

Xilinx FPGA



Configurable Logic Block (CLB)

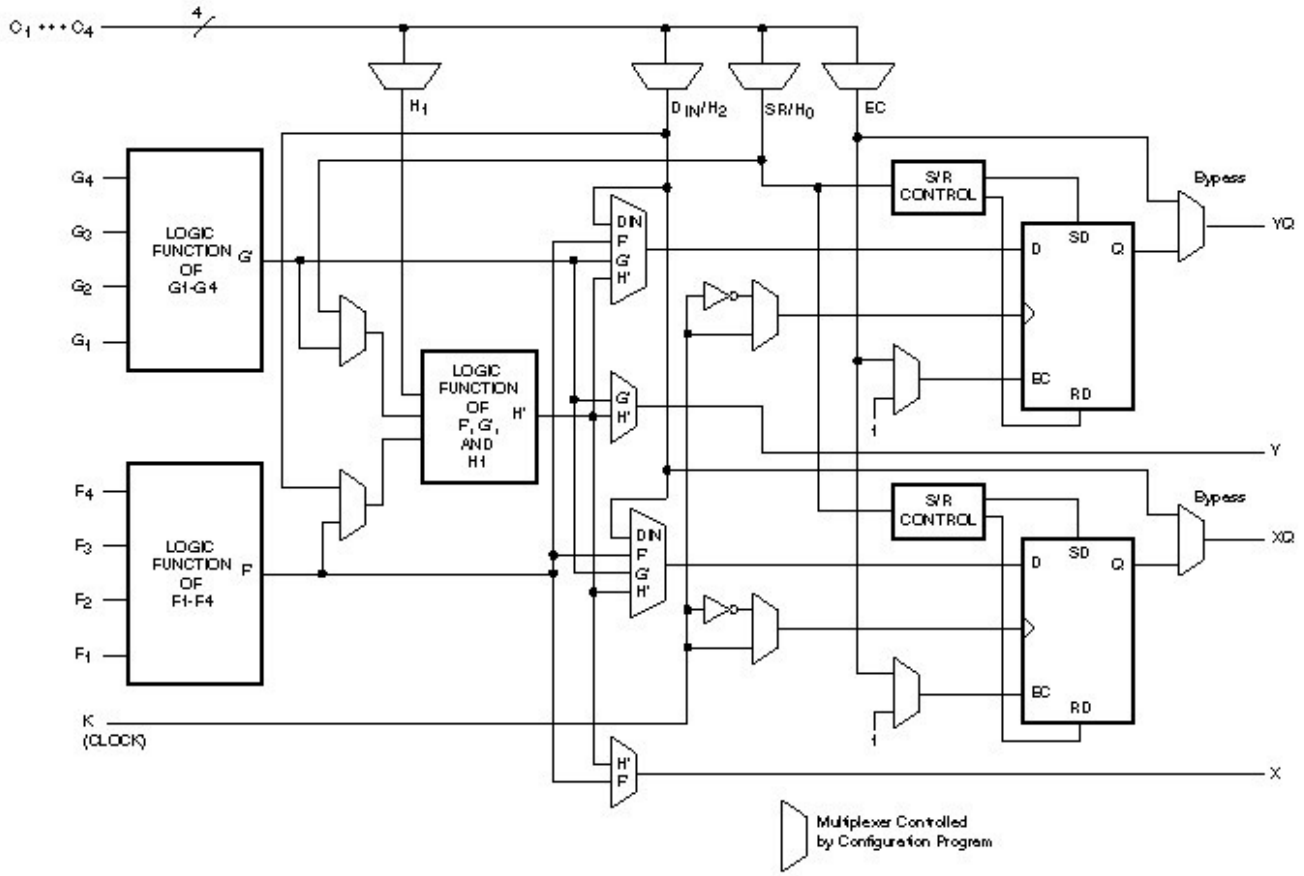
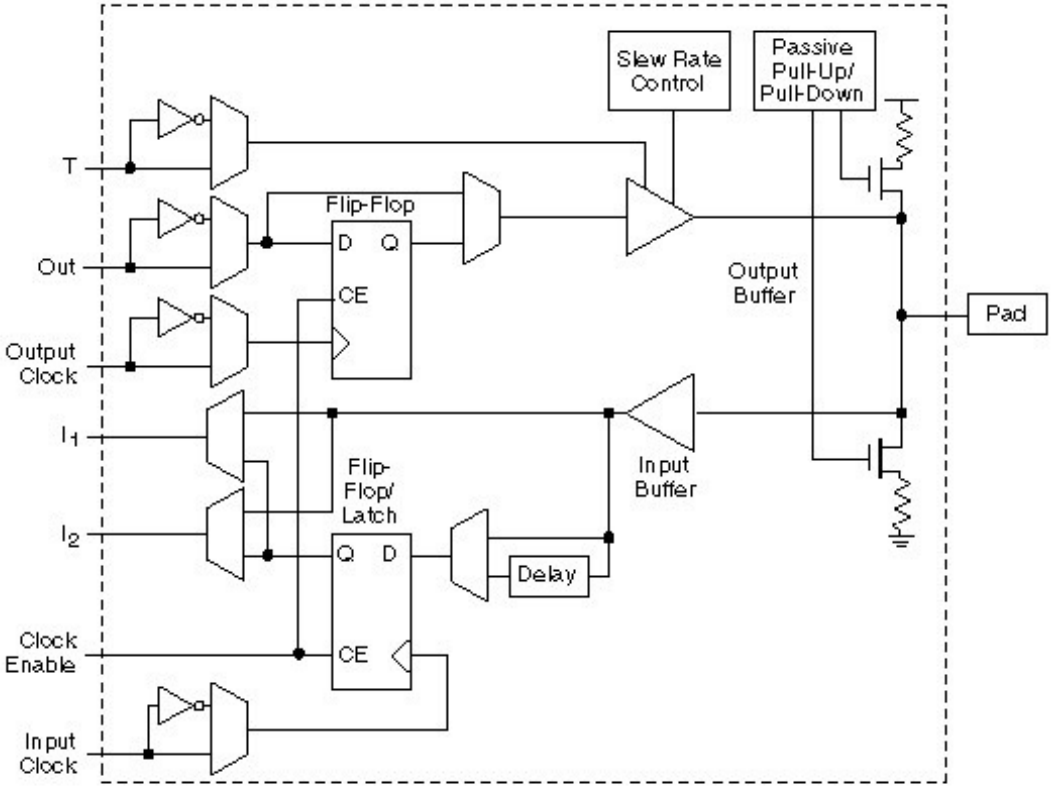


Figure 1: Simplified Block Diagram of XC4000-Series CLB (RAM and Carry Logic functions not shown)

I/O Block



Simplified Block Diagram of XC4000E IOB

Embedded Systems Design: A Unified Hardware/Software Introduction

Chapter 11: Design Technology

Outline

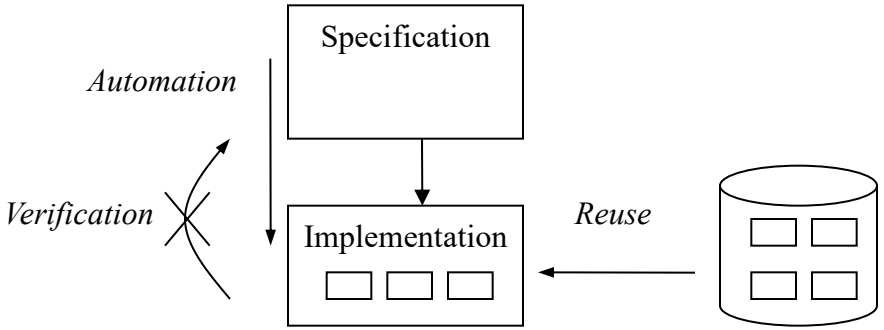
- Automation: synthesis
- Verification: hardware/software co-simulation
- Reuse: intellectual property cores
- Design process models

Introduction

- Design task
  - Define system functionality
  - Convert functionality to physical implementation while
    - Satisfying constrained metrics
    - Optimizing other design metrics
- Designing embedded systems is hard
  - Complex functionality
    - Millions of possible environment scenarios
    - Competing, tightly constrained metrics
  - Productivity gap
    - As low as 10 lines of code or 100 transistors produced per day

Improving productivity

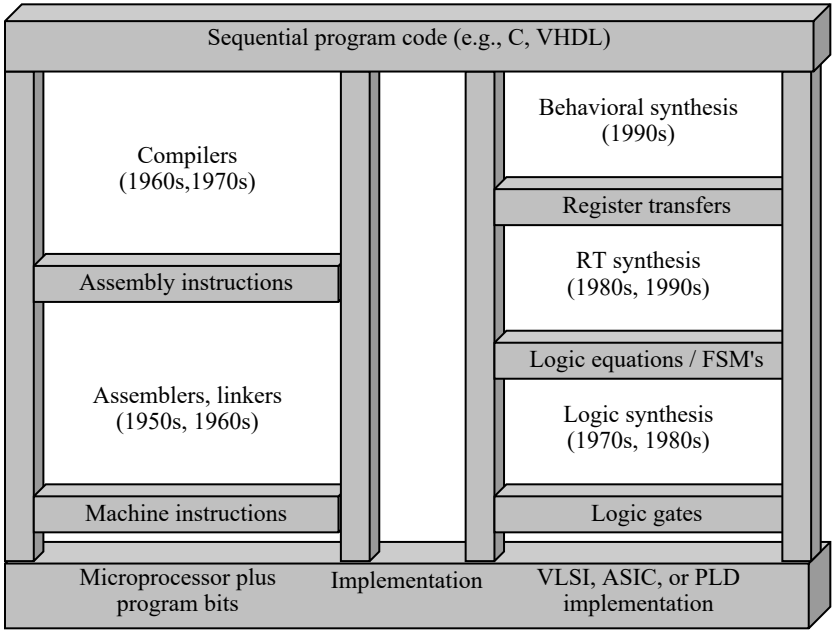
- Design technologies developed to improve productivity
- We focus on technologies advancing hardware/software unified view
  - Automation
    - Program replaces manual design
    - Synthesis
  - Reuse
    - Predesigned components
    - Cores
    - General-purpose and single-purpose processors on single IC
  - Verification
    - Ensuring correctness/completeness of each design step
    - Hardware/software co-simulation



Automation: synthesis

- Early design mostly hardware
- Software complexity increased with advent of general-purpose processor
- Different techniques for software design and hardware design
  - Caused division of the two fields
- Design tools evolve for higher levels of abstraction
  - Different rate in each field
- Hardware/software design fields rejoining
  - Both can start from behavioral description in sequential program model
  - 30 years longer for hardware design to reach this step in the ladder
    - Many more design dimensions
    - Optimization critical

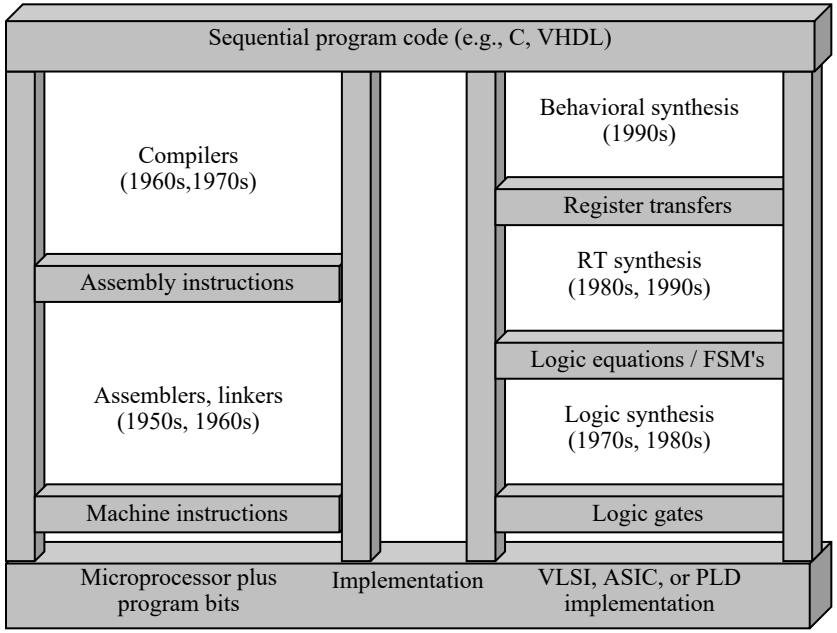
The codesign ladder



Hardware/software parallel evolution

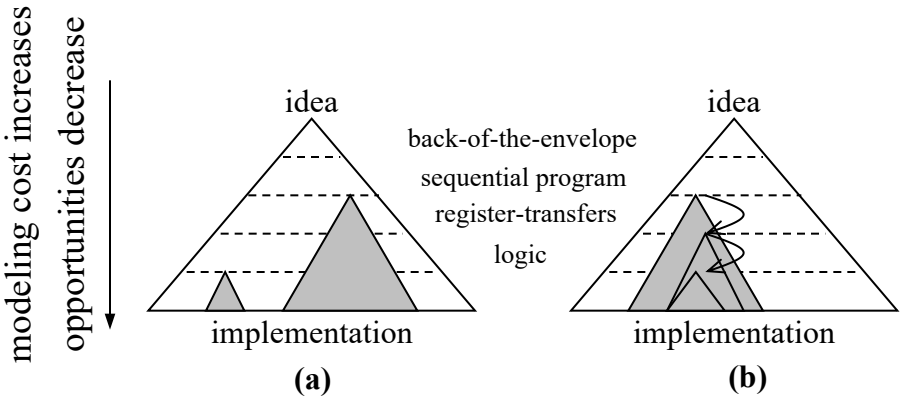
- Software design evolution
  - Machine instructions
  - Assemblers
    - convert assembly programs into machine instructions
  - Compilers
    - translate sequential programs into assembly
- Hardware design evolution
  - Interconnected logic gates
  - Logic synthesis
    - converts logic equations or FSMs into gates
  - Register-transfer (RT) synthesis
    - converts FSMDs into FSMs, logic equations, predesigned RT components (registers, adders, etc.)
  - Behavioral synthesis
    - converts sequential programs into FSMDs

The codesign ladder



Increasing abstraction level

- Higher abstraction level focus of hardware/software design evolution
  - Description smaller/easier to capture
    - E.g., Line of sequential program code can translate to 1000 gates
  - Many more possible implementations available
    - (a) Like flashlight, the higher above the ground, the more ground illuminated
      - Sequential program designs may differ in performance/transistor count by orders of magnitude
      - Logic-level designs may differ by only power of 2
    - (b) Design process proceeds to lower abstraction level, narrowing in on single implementation

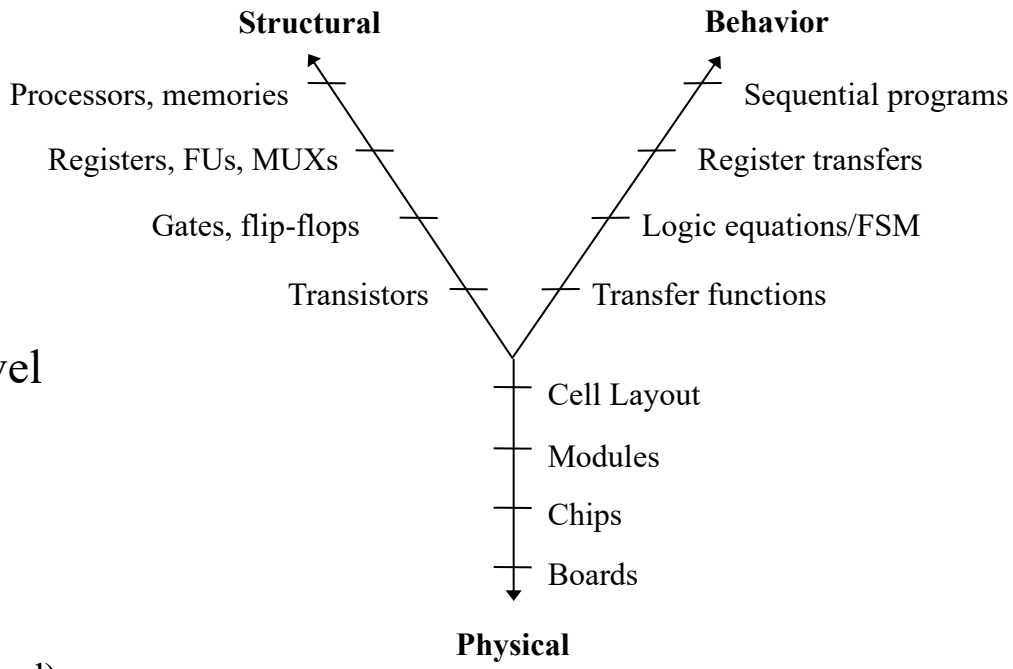


Synthesis

- Automatically converting system's behavioral description to a structural implementation
  - Complex whole formed by parts
  - Structural implementation must optimize design metrics
- More expensive, complex than compilers
  - Cost = $100s to $10,000s
  - User controls 100s of synthesis options
  - Optimization critical
    - Otherwise could use software
  - Optimizations different for each user
  - Run time = hours, days

Gajski's Y-chart

- Each axis represents type of description
  - Behavioral
    - Defines outputs as function of inputs
    - Algorithms but no implementation
  - Structural
    - Implements behavior by connecting components with known behavior
  - Physical
    - Gives size/locations of components and wires on chip/board
- Synthesis converts behavior at given level to structure at same level or lower
  - E.g.,
    - FSM → gates, flip-flops (same level)
    - FSM → transistors (lower level)
    - FSM X registers, FUs (higher level)
    - FSM X processors, memories (higher level)



Logic synthesis

- Logic-level behavior to structural implementation
  - Logic equations and/or FSM to connected gates
- Combinational logic synthesis
  - Two-level minimization (Sum of products/product of sums)
    - Best possible performance
      - Longest path = 2 gates (AND gate + OR gate/OR gate + AND gate)
    - Minimize size
      - Minimum cover
      - Minimum cover that is prime
      - Heuristics
  - Multilevel minimization
    - Trade performance for size
    - Pareto-optimal solution
      - Heuristics
- FSM synthesis
  - State minimization
  - State encoding

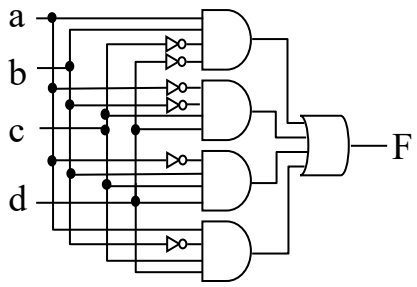
Two-level minimization

- Represent logic function as sum of products (or product of sums)
  - AND gate for each product
  - OR gate for each sum
- Gives best possible performance
  - At most 2 gate delay
- Goal: minimize size
  - Minimum cover
    - Minimum # of AND gates (sum of products)
  - Minimum cover that is prime
    - Minimum # of inputs to each AND gate (sum of products)

Sum of products

F = abc'd' + a'b'cd + a'bcd + ab'cd

Direct implementation



4 4-input AND gates and
1 4-input OR gate

→ 40 transistors

Minimum cover

- Minimum # of AND gates (sum of products)
- **Literal**: variable or its complement
  - a or a', b or b', etc.
- **Minterm**: product of literals
  - Each literal appears exactly once
    - abc'd', ab'cd, a'bcd, etc.
- **Implicant**: product of literals
  - Each literal appears no more than once
    - abc'd', a'cd, etc.
  - Covers 1 or more minterms
    - a'cd covers a'bcd and a'b'cd
- **Cover**: set of implicants that covers all minterms of function
- **Minimum cover**: cover with minimum # of implicants

Minimum cover: K-map approach

- Karnaugh map (K-map)
  - 1 represents minterm
  - Circle represents implicant
- Minimum cover
  - Covering all 1's with min # of circles
  - Example: direct vs. min cover
    - Less gates
      - 4 vs. 5
    - Less transistors
      - 28 vs. 40

K-map: sum of products

ab \ cd	00	01	11	10
00	0	0	1	0
01	0	0	1	0
11	1	0	0	0
10	0	0	1	0

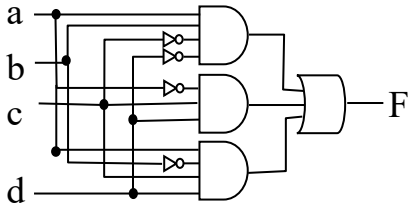
K-map: minimum cover

ab \ cd	00	01	11	10
00	0	0	(1)	0
01	0	0	(1)	0
11	(1)	0	0	0
10	0	0	(1)	0

Minimum cover

F=abc'd' + a'cd + ab'cd

Minimum cover implementation

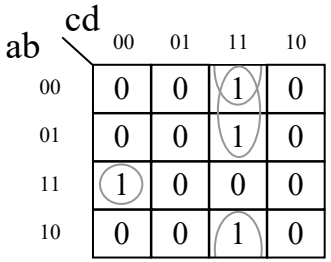


2 4-input AND gate
1 3-input AND gates
1 4 input OR gate
→ 28 transistors

Minimum cover that is prime

- Minimum # of inputs to AND gates
- Prime implicant
  - Implicant not covered by any other implicant
  - Max-sized circle in K-map
- Minimum cover that is prime
  - Covering with min # of prime implicants
  - Min # of max-sized circles
  - Example: prime cover vs. min cover
    - Same # of gates
      - 4 vs. 4
    - Less transistors
      - 26 vs. 28

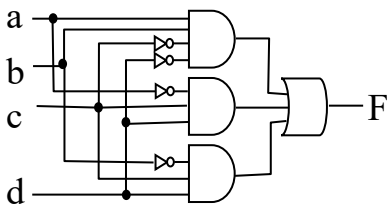
K-map: minimum cover that is prime



Minimum cover that is prime

F=abc'd' + a'cd + b'cd

Implementation



1 4-input AND gate
2 3-input AND gates
1 4 input OR gate

→ 26 transistors

Minimum cover: heuristics

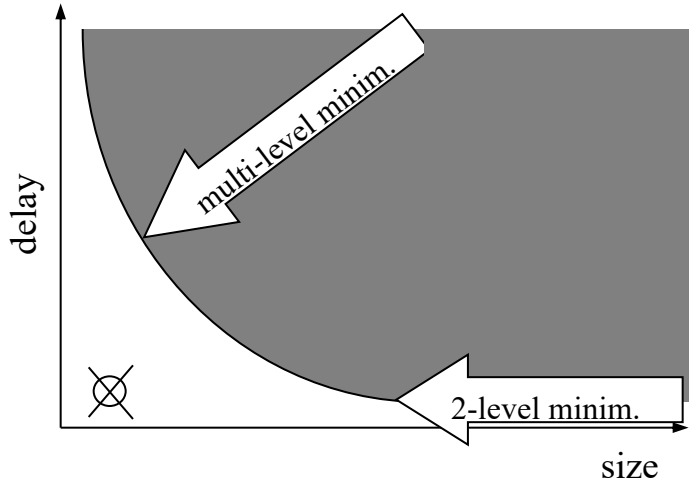
- K-maps give optimal solution every time
  - Functions with > 6 inputs too complicated
  - Use computer-based tabular method
    - Finds all prime implicants
    - Finds min cover that is prime
    - Also optimal solution every time
    - Problem: 2^n minterms for n inputs
      - 32 inputs = 4 billion minterms
      - Exponential complexity
- Heuristic
  - Solution technique where optimal solution not guaranteed
  - Hopefully comes close

Heuristics: iterative improvement

- Start with initial solution
  - i.e., original logic equation
- Repeatedly make modifications toward better solution
- Common modifications
  - Expand
    - Replace each nonprime implicant with a prime implicant covering it
    - Delete all implicants covered by new prime implicant
  - Reduce
    - Opposite of expand
  - Reshape
    - Expands one implicant while reducing another
    - Maintains total # of implicants
  - Irredundant
    - Selects min # of implicants that cover from existing implicants
- Synthesis tools differ in modifications used and the order they are used

Multilevel logic minimization

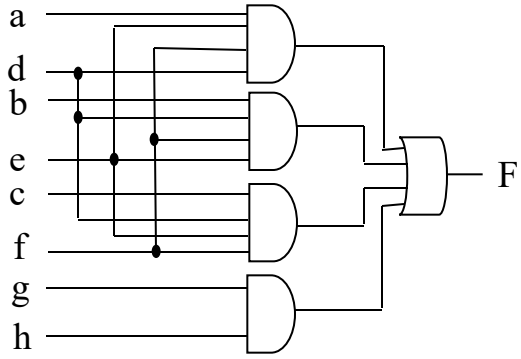
- Trade performance for size
  - Increase delay for lower # of gates
  - Gray area represents all possible solutions
  - Circle with X represents ideal solution
    - Generally not possible
  - 2-level gives best performance
    - max delay = 2 gates
    - Solve for smallest size
  - Multilevel gives pareto-optimal solution
    - Minimum delay for a given size
    - Minimum size for a given delay



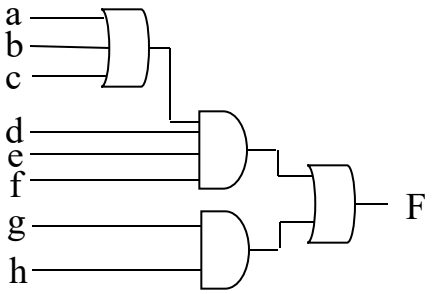
Example

- Minimized 2-level logic function:
  - F = adef + bdef + cdef + gh
  - Requires 5 gates with 18 total gate inputs
    - 4 ANDS and 1 OR
- After algebraic manipulation:
  - F = (a + b + c)def + gh
  - Requires only 4 gates with 11 total gate inputs
    - 2 ANDS and 2 ORs
  - Less inputs per gate
  - Assume gate inputs = 2 transistors
    - Reduced by 14 transistors
      - 36 (18 * 2) down to 22 (11 * 2)
  - Sacrifices performance for size
    - Inputs a, b, and c now have 3-gate delay
- Iterative improvement heuristic commonly used

2-level minimized



multilevel minimized



FSM synthesis

- FSM to gates
- State minimization
  - Reduce # of states
    - Identify and merge equivalent states
      - Outputs, next states same for all possible inputs
      - Tabular method gives exact solution
        - Table of all possible state pairs
        - If n states, n^2 table entries
        - Thus, heuristics used with large # of states
- State encoding
  - Unique bit sequence for each state
  - If n states, $\log_2(n)$ bits
  - n! possible encodings
  - Thus, heuristics common

Technology mapping

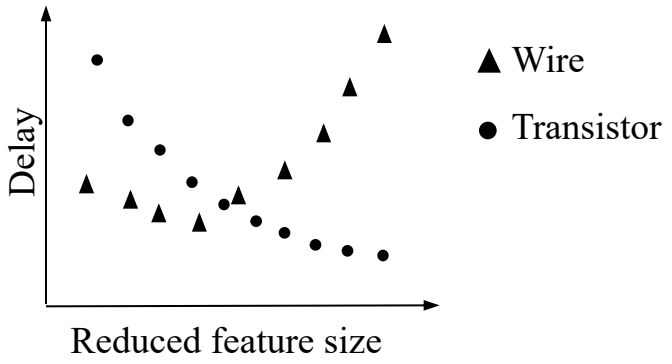
- Library of gates available for implementation
  - Simple
    - only 2-input AND,OR gates
  - Complex
    - various-input AND,OR,NAND,NOR,etc. gates
    - Efficiently implemented meta-gates (i.e., AND-OR-INVERT,MUX)
- Final structure consists of specified library's components only
- If technology mapping integrated with logic synthesis
  - More efficient circuit
  - More complex problem
  - Heuristics required

Complexity impact on user

- As complexity grows, heuristics used
- Heuristics differ tremendously among synthesis tools
  - Computationally expensive
    - Higher quality results
    - Variable optimization effort settings
    - Long run times (hours, days)
    - Requires huge amounts of memory
    - Typically needs to run on servers, workstations
  - Fast heuristics
    - Lower quality results
    - Shorter run times (minutes, hours)
    - Smaller amount of memory required
    - Could run on PC
- Super-linear-time (i.e. n^3) heuristics usually used
  - User can partition large systems to reduce run times/size
  - $100^3 > 50^3 + 50^3$ (1,000,000 > 250,000)

Integrating logic design and physical design

- Past
  - Gate delay much greater than wire delay
  - Thus, performance evaluated as # of levels of gates only
- Today
  - Gate delay shrinking as feature size shrinking
  - Wire delay increasing
    - Performance evaluation needs wire length
  - Transistor placement (needed for wire length) domain of physical design
  - Thus, simultaneous logic synthesis and physical design required for efficient circuits



Register-transfer synthesis

- Converts FSMD to custom single-purpose processor
  - Datapath
    - Register units to store variables
      - Complex data types
    - Functional units
      - Arithmetic operations
    - Connection units
      - Buses, MUXs
  - FSM controller
    - Controls datapath
  - Key sub problems:
    - Allocation
      - Instantiate storage, functional, connection units
    - Binding
      - Mapping FSMD operations to specific units

Behavioral synthesis

- High-level synthesis
- Converts single sequential program to single-purpose processor
  - Does not require the program to schedule states
- Key sub problems
  - Allocation
  - Binding
  - Scheduling
    - Assign sequential program's operations to states
    - Conversion template given in Ch. 2
- Optimizations important
  - Compiler
    - Constant propagation, dead-code elimination, loop unrolling
  - Advanced techniques for allocation, binding, scheduling

System synthesis

- Convert 1 or more processes into 1 or more processors (system)
  - For complex embedded systems
    - Multiple processes may provide better performance/power
    - May be better described using concurrent sequential programs
- Tasks
  - Transformation
    - Can merge 2 exclusive processes into 1 process
    - Can break 1 large process into separate processes
    - Procedure inlining
    - Loop unrolling
  - Allocation
    - Essentially design of system architecture
      - Select processors to implement processes
      - Also select memories and busses

System synthesis

- Tasks (cont.)
  - Partitioning
    - Mapping 1 or more processes to 1 or more processors
    - Variables among memories
    - Communications among buses
  - Scheduling
    - Multiple processes on a single processor
    - Memory accesses
    - Bus communications
  - Tasks performed in variety of orders
  - Iteration among tasks common

System synthesis

- Synthesis driven by constraints
  - E.g.,
    - Meet performance requirements at minimum cost
      - Allocate as much behavior as possible to general-purpose processor
        - Low-cost/flexible implementation
      - Minimum # of SPPs used to meet performance
- System synthesis for GPP only (software)
  - Common for decades
    - Multiprocessing
    - Parallel processing
    - Real-time scheduling
- Hardware/software codesign
  - Simultaneous consideration of GPPs/SPPs during synthesis
  - Made possible by maturation of behavioral synthesis in 1990's

Temporal vs. spatial thinking

- Design thought process changed by evolution of synthesis
- Before synthesis
  - Designers worked primarily in structural domain
    - Connecting simpler components to build more complex systems
      - Connecting logic gates to build controller
      - Connecting registers, MUXs, ALUs to build datapath
  - "capture and simulate" era
    - Capture using CAD tools
    - Simulate to verify correctness before fabricating
  - Spatial thinking
    - Structural diagrams
    - Data sheets

Temporal vs. spatial thinking

- After synthesis
  - "describe-and-synthesize" era
  - Designers work primarily in behavioral domain
  - "describe and synthesize" era
    - Describe FSMDs or sequential programs
    - Synthesize into structure
  - Temporal thinking
    - States or sequential statements have relationship over time
- Strong understanding of hardware structure still important
  - Behavioral description must synthesize to efficient structural implementation

Verification

- Ensuring design is correct and complete
  - Correct
    - Implements specification accurately
  - Complete
    - Describes appropriate output to all relevant input
- Formal verification
  - Hard
  - For small designs or verifying certain key properties only
- Simulation
  - Most common verification method

Formal verification

- Analyze design to prove or disprove certain properties
- Correctness example
  - Prove ALU structural implementation equivalent to behavioral description
    - Derive Boolean equations for outputs
    - Create truth table for equations
    - Compare to truth table from original behavior
- Completeness example
  - Formally prove elevator door can never open while elevator is moving
    - Derive conditions for door being open
    - Show conditions conflict with conditions for elevator moving

Simulation

- Create computer model of design
  - Provide sample input
  - Check for acceptable output
- Correctness example
  - ALU
    - Provide all possible input combinations
    - Check outputs for correct results
- Completeness example
  - Elevator door closed when moving
    - Provide all possible input sequences
    - Check door always closed when elevator moving

Increases confidence

- Simulating all possible input sequences impossible for most systems
  - E.g., 32-bit ALU
    - $2^{32} * 2^{32} = 2^{64}$ possible input combinations
    - At 1 million combinations/sec
    - ½ million years to simulate
    - Sequential circuits even worse
- Can only simulate tiny subset of possible inputs
  - Typical values
  - Known boundary conditions
    - E.g., 32-bit ALU
      - Both operands all 0's
      - Both operands all 1's
- Increases confidence of correctness/completeness
- Does not prove

Advantages over physical implementation

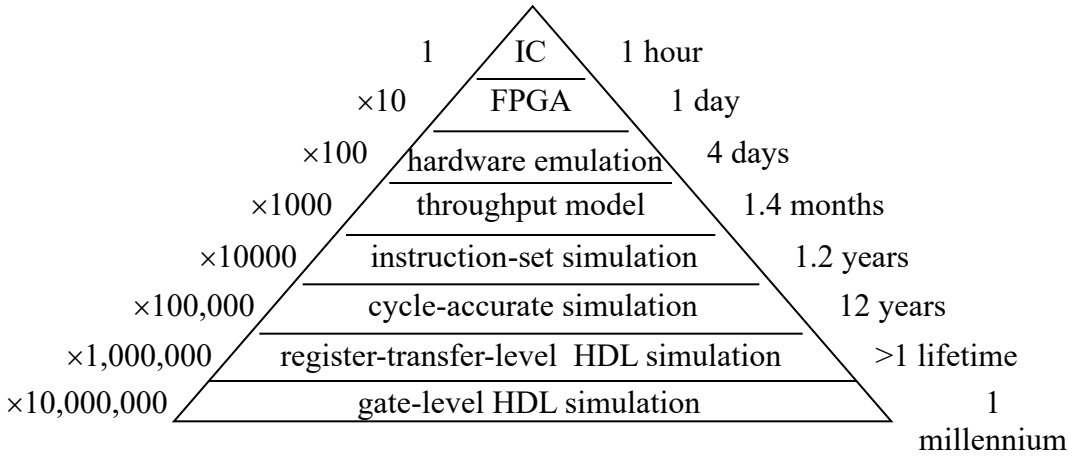
- Controllability
  - Control time
    - Stop/start simulation at any time
  - Control data values
    - Inputs or internal values
- Observability
  - Examine system/environment values at any time
- Debugging
  - Can stop simulation at any point and:
    - Observe internal values
    - Modify system/environment values before restarting
  - Can step through small intervals (i.e., 500 nanoseconds)

Disadvantages

- Simulation setup time
  - Often has complex external environments
  - Could spend more time modeling environment than system
- Models likely incomplete
  - Some environment behavior undocumented if complex environment
  - May not model behavior correctly
- Simulation speed much slower than actual execution
  - Sequentializing parallel design
    - IC: gates operate in parallel
    - Simulation: analyze inputs, generate outputs for each gate 1 at time
  - Several programs added between simulated system and real hardware
    - 1 simulated operation:
      - = 10 to 100 simulator operations
      - = 100 to 10,000 operating system operations
      - = 1,000 to 100,000 hardware operations

Simulation speed

- Relative speeds of different types of simulation/emulation
  - 1 hour actual execution of SOC
    - = 1.2 years instruction-set simulation
    - = 10,000,000 hours gate-level simulation



Overcoming long simulation time

- Reduce amount of real time simulated
  - 1 msec execution instead of 1 hour
    - 0.001sec * 10,000,000 = 10,000 sec = 3 hours
  - Reduced confidence
    - 1 msec of cruise controller operation tells us little
- Faster simulator
  - Emulators
    - Special hardware for simulations
  - Less precise/accurate simulators
    - Exchange speed for observability/controllability

Reducing precision/accuracy

- Don't need gate-level analysis for all simulations
  - E.g., cruise control
    - Don't care what happens at every input/output of each logic gate
  - Simulating RT components ~10x faster
  - Cycle-based simulation ~100x faster
    - Accurate at clock boundaries only
    - No information on signal changes between boundaries
- Faster simulator often combined with reduction in real time
  - If willing to simulate for 10 hours
    - Use instruction-set simulator
    - Real execution time simulated
      - 10 hours * 1 / 10,000
      - = 0.001 hour
      - = 3.6 seconds

Hardware/software co-simulation

- Variety of simulation approaches exist
  - From very detailed
    - E.g., gate-level model
  - To very abstract
    - E.g., instruction-level model
- Simulation tools evolved separately for hardware/software
  - Recall separate design evolution
  - Software (GPP)
    - Typically with instruction-set simulator (ISS)
  - Hardware (SPP)
    - Typically with models in HDL environment
- Integration of GPP/SPP on single IC creating need for merging simulation tools

Integrating GPP/SPP simulations

- Simple/naïve way
  - HDL model of microprocessor
    - Runs system software
    - Much slower than ISS
    - Less observable/controllable than ISS
  - HDL models of SPPs
  - Integrate all models
- Hardware-software co-simulator
  - ISS for microprocessor
  - HDL model for SPPs
  - Create communication between simulators
  - Simulators run separately except when transferring data
  - Faster
  - Though, frequent communication between ISS and HDL model slows it down

Minimizing communication

- Memory shared between GPP and SPPs
  - Where should memory go?
  - In ISS
    - HDL simulator must stall for memory access
  - In HDL?
    - ISS must stall when fetching each instruction
- Model memory in both ISS and HDL
  - Most accesses by each model unrelated to other's accesses
    - No need to communicate these between models
  - Co-simulator ensures consistency of shared data
  - Huge speedups (100x or more) reported with this technique

Emulators

- General physical device system mapped to
  - Microprocessor emulator
    - Microprocessor IC with some monitoring, control circuitry
  - SPP emulator
    - FPGAs (10s to 100s)
  - Usually supports debugging tasks
- Created to help solve simulation disadvantages
  - Mapped relatively quickly
    - Hours, days
  - Can be placed in real environment
    - No environment setup time
    - No incomplete environment
  - Typically faster than simulation
    - Hardware implementation

Disadvantages

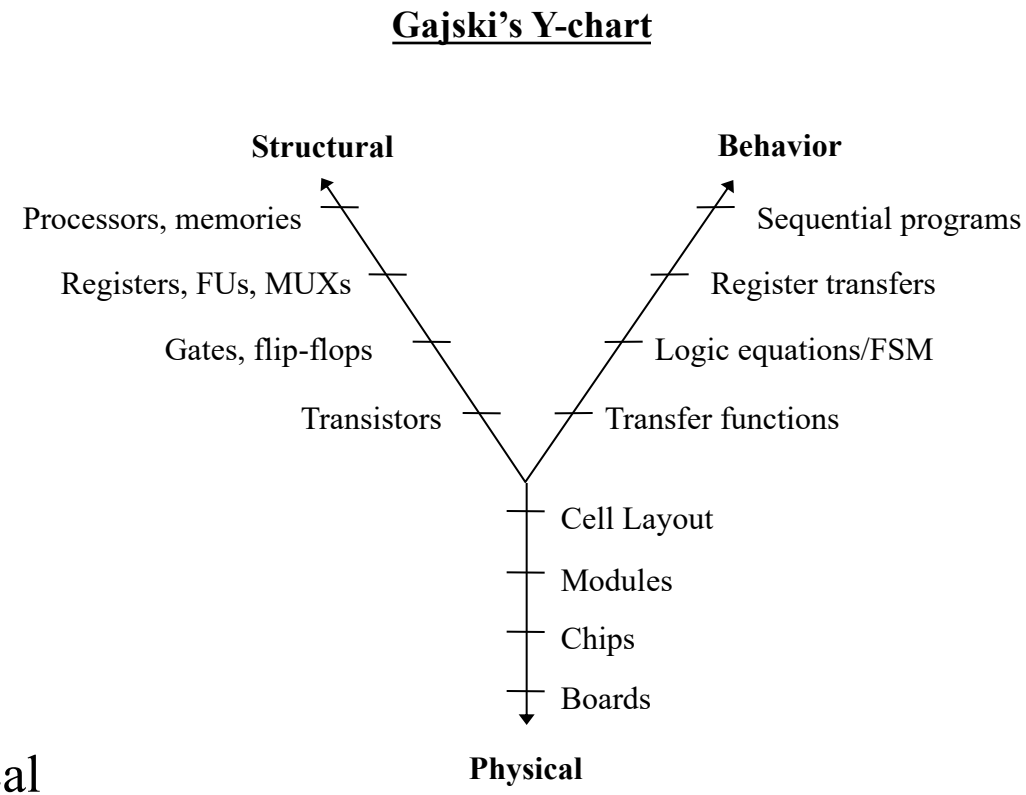
- Still not as fast as real implementations
  - E.g., emulated cruise-control may not respond fast enough to keep control of car
- Mapping still time consuming
  - E.g., mapping complex SOC to 10 FPGAs
    - Just partitioning into 10 parts could take weeks
- Can be very expensive
  - Top-of-the-line FPGA-based emulator: $100,000 to $1mill
  - Leads to resource bottleneck
    - Can maybe only afford 1 emulator
    - Groups wait days, weeks for other group to finish using

Reuse: intellectual property cores

- Commercial off-the-shelf (COTS) components
  - Predesigned, prepackaged ICs
  - Implements GPP or SPP
  - Reduces design/debug time
  - Have always been available
- System-on-a-chip (SOC)
  - All components of system implemented on single chip
  - Made possible by increasing IC capacities
  - Changing the way COTS components sold
    - As intellectual property (IP) rather than actual IC
      - Behavioral, structural, or physical descriptions
      - Processor-level components known as cores
    - SOC built by integrating multiple descriptions

Cores

- Soft core
  - Synthesizable behavioral description
  - Typically written in HDL (VHDL/Verilog)
- Firm core
  - Structural description
  - Typically provided in HDL
- Hard core
  - Physical description
  - Provided in variety of physical layout file formats



Advantages/disadvantages of hard core

- Ease of use
  - Developer already designed and tested core
    - Can use right away
    - Can expect to work correctly
- Predictability
  - Size, power, performance predicted accurately
- Not easily mapped (retargeted) to different process
  - E.g., core available for vendor X's 0.25 micrometer CMOS process
    - Can't use with vendor X's 0.18 micrometer process
    - Can't use with vendor Y

Advantages/disadvantages of soft/firm cores

- Soft cores
  - Can be synthesized to nearly any technology
  - Can optimize for particular use
    - E.g., delete unused portion of core
      - Lower power, smaller designs
  - Requires more design effort
  - May not work in technology not tested for
  - Not as optimized as hard core for same processor
- Firm cores
  - Compromise between hard and soft cores
    - Some retargetability
    - Limited optimization
    - Better predictability/ease of use

New challenges to processor providers

- Cores have dramatically changed business model
  - Pricing models
    - Past
      - Vendors sold product as IC to designers
      - Designers must buy any additional copies
        - Could not (economically) copy from original
    - Today
      - Vendors can sell as IP
      - Designers can make as many copies as needed
    - Vendor can use different pricing models
      - Royalty-based model
        - Similar to old IC model
        - Designer pays for each additional model
      - Fixed price model
        - One price for IP and as many copies as needed
      - Many other models used

IP protection

- Past
  - Illegally copying IC very difficult
    - Reverse engineering required tremendous, deliberate effort
    - “Accidental” copying not possible
- Today
  - Cores sold in electronic format
    - Deliberate/accidental unauthorized copying easier
    - Safeguards greatly increased
    - Contracts to ensure no copying/distributing
    - Encryption techniques
      - limit actual exposure to IP
    - Watermarking
      - determines if particular instance of processor was copied
      - whether copy authorized

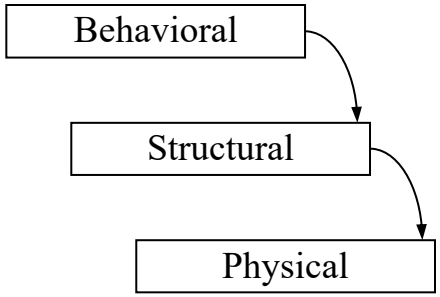
New challenges to processor users

- Licensing arrangements
  - Not as easy as purchasing IC
  - More contracts enforcing pricing model and IP protection
    - Possibly requiring legal assistance
- Extra design effort
  - Especially for soft cores
    - Must still be synthesized and tested
    - Minor differences in synthesis tools can cause problems
- Verification requirements more difficult
  - Extensive testing for synthesized soft cores and soft/firm cores mapped to particular technology
    - Ensure correct synthesis
    - Timing and power vary between implementations
  - Early verification critical
    - Cores buried within IC
    - Cannot simply replace bad core

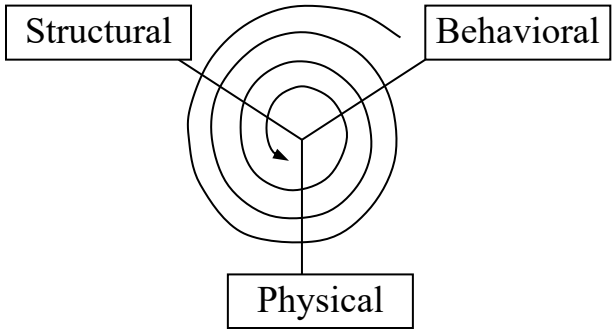
Design process model

- Describes order that design steps are processed
  - Behavior description step
  - Behavior to structure conversion step
  - Mapping structure to physical implementation step
- Waterfall model
  - Proceed to next step only after current step completed
- Spiral model
  - Proceed through 3 steps in order but with less detail
  - Repeat 3 steps gradually increasing detail
  - Keep repeating until desired system obtained
  - Becoming extremely popular (hardware & software development)

Waterfall design model



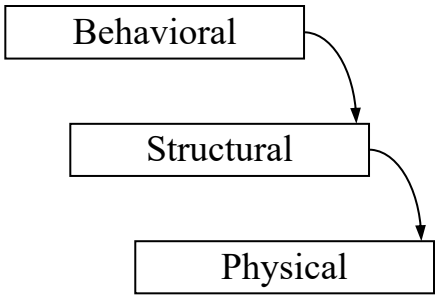
Spiral design model



Waterfall method

- Not very realistic
  - Bugs often found in later steps that must be fixed in earlier step
    - E.g., forgot to handle certain input condition
  - Prototype often needed to know complete desired behavior
    - E.g, customer adds features after product demo
  - System specifications commonly change
    - E.g., to remain competitive by reducing power, size
      - Certain features dropped
- Unexpected iterations back through 3 steps cause missed deadlines
  - Lost revenues
  - May never make it to market

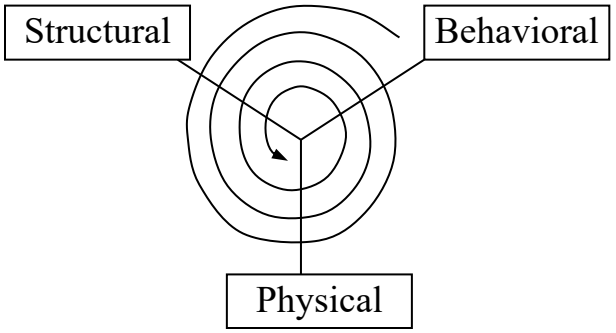
Waterfall design model



Spiral method

- First iteration of 3 steps incomplete
- Much faster, though
  - End up with prototype
    - Use to test basic functions
    - Get idea of functions to add/remove
  - Original iteration experience helps in following iterations of 3 steps
- Must come up with ways to obtain structure and physical implementations quickly
  - E.g., FPGAs for prototype
    - silicon for final product
  - May have to use more tools
    - Extra effort/cost
- Could require more time than waterfall method
  - If correct implementation first time with waterfall

Spiral design model

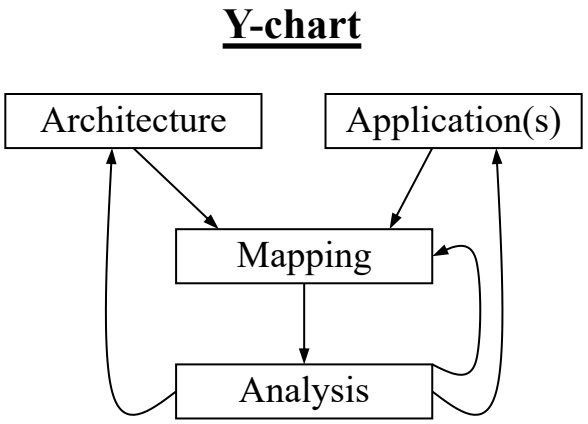


General-purpose processor design models

- Previous slides focused on SPPs
- Can apply equally to GPPs
  - Waterfall model
    - Structure developed by particular company
    - Acquired by embedded system designer
    - Designer develops software (behavior)
    - Designer maps application to architecture
      - Compilation
      - Manual design
  - Spiral-like model
    - Beginning to be applied by embedded system designers

Spiral-like model

- Designer develops or acquires architecture
- Develops application(s)
- Maps application to architecture
- Analyzes design metrics
- Now makes choice
  - Modify mapping
  - Modify application(s) to better suit architecture
  - Modify architecture to better suit application(s)
    - Not as difficult now
      - Maturation of synthesis/compilers
      - IPs can be tuned
- Continue refining to lower abstraction level until particular implementation chosen



Summary

- Design technology seeks to reduce gap between IC capacity growth and designer productivity growth
- Synthesis has changed digital design
- Increased IC capacity means sw/hw components coexist on one chip
- Design paradigm shift to core-based design
- Simulation essential but hard
- Spiral design process is popular

Book Summary

- Embedded systems are common and growing
  - Such systems are very different from in the past due to increased IC capacities and automation tools
  - Indicator: National Science Foundation just created a separate program on Embedded Systems (2002).
- New view:
  - Embedded computing systems are built from a collection of *processors*, some general-purpose (sw), some single-purpose (hw)
  - Hw/sw differ in design metrics, not in some fundamental way
  - Memory and interfaces necessary to complete system
  - Days of embedded system design as assembly-level programming of one microprocessor are fading away
- Need to focus on higher-level issues
  - State machines, concurrent processes, control systems
  - IC technologies, design technologies
- There's a growing, challenging and exciting world of embedded systems design out there. There's also much more to learn. Enjoy!