



UNIT – I

Embedded System Introduction



UNIT - I: Embedded System Introduction: Contents



1. Embedded systems overview
2. Design challenges
3. Processor technology
4. IC technology
5. Design Technology
6. Trade-offs

:Single purpose processors RT-level

1. Combinational logic
2. Sequential logic (RT level)
3. Custom single purpose processor design(RT – level)
4. Optimizing custom single purpose processors.



Introduction

- 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

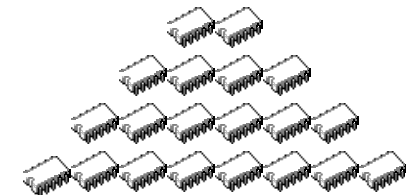
Computers are in here...



and here...



and even here...



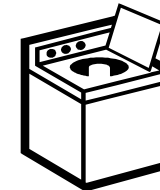
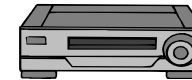
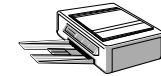
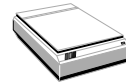
Lots more of these, though they cost a lot less each.



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



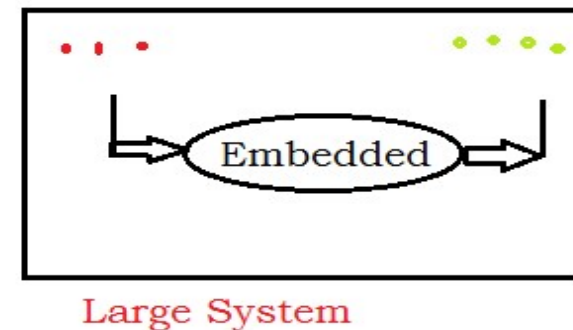
Some common characteristics of embedded systems

- 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

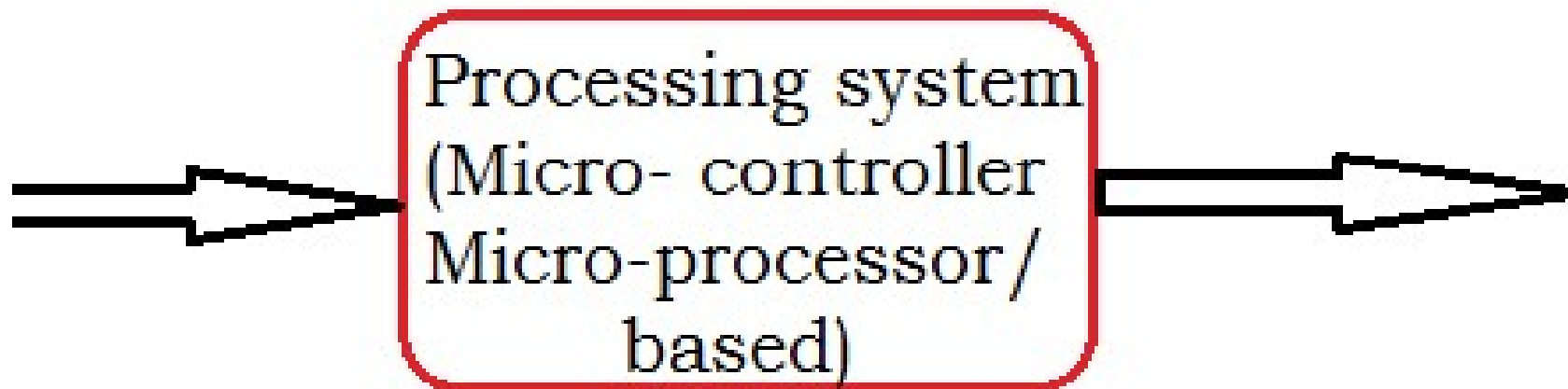


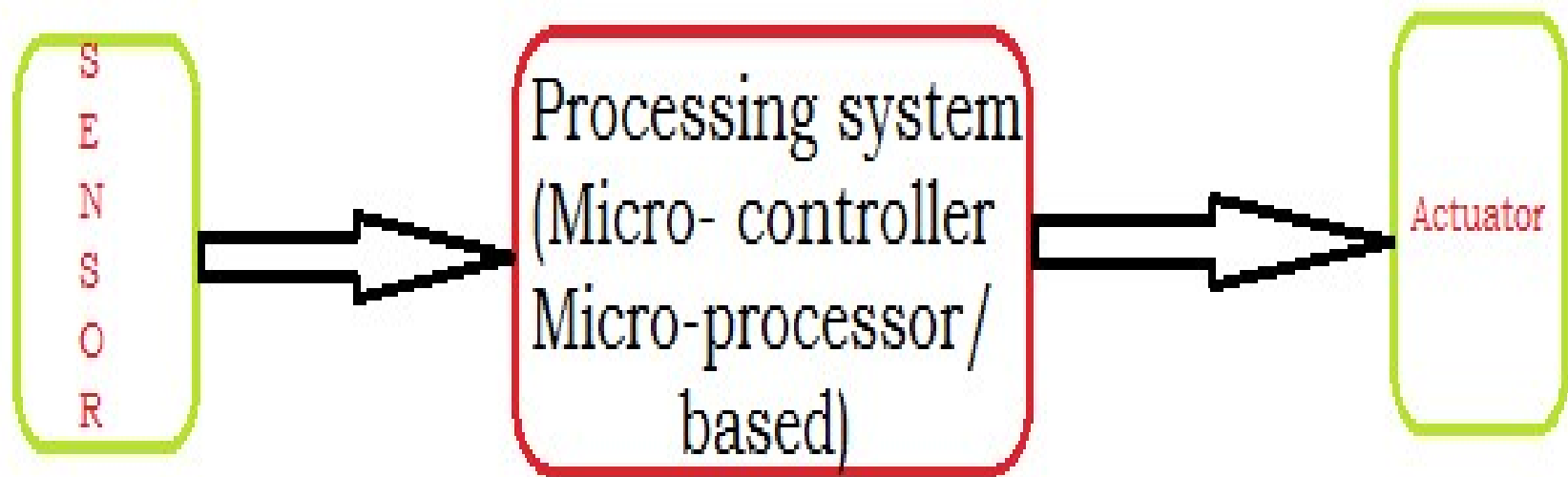
1. Embedded systems overview

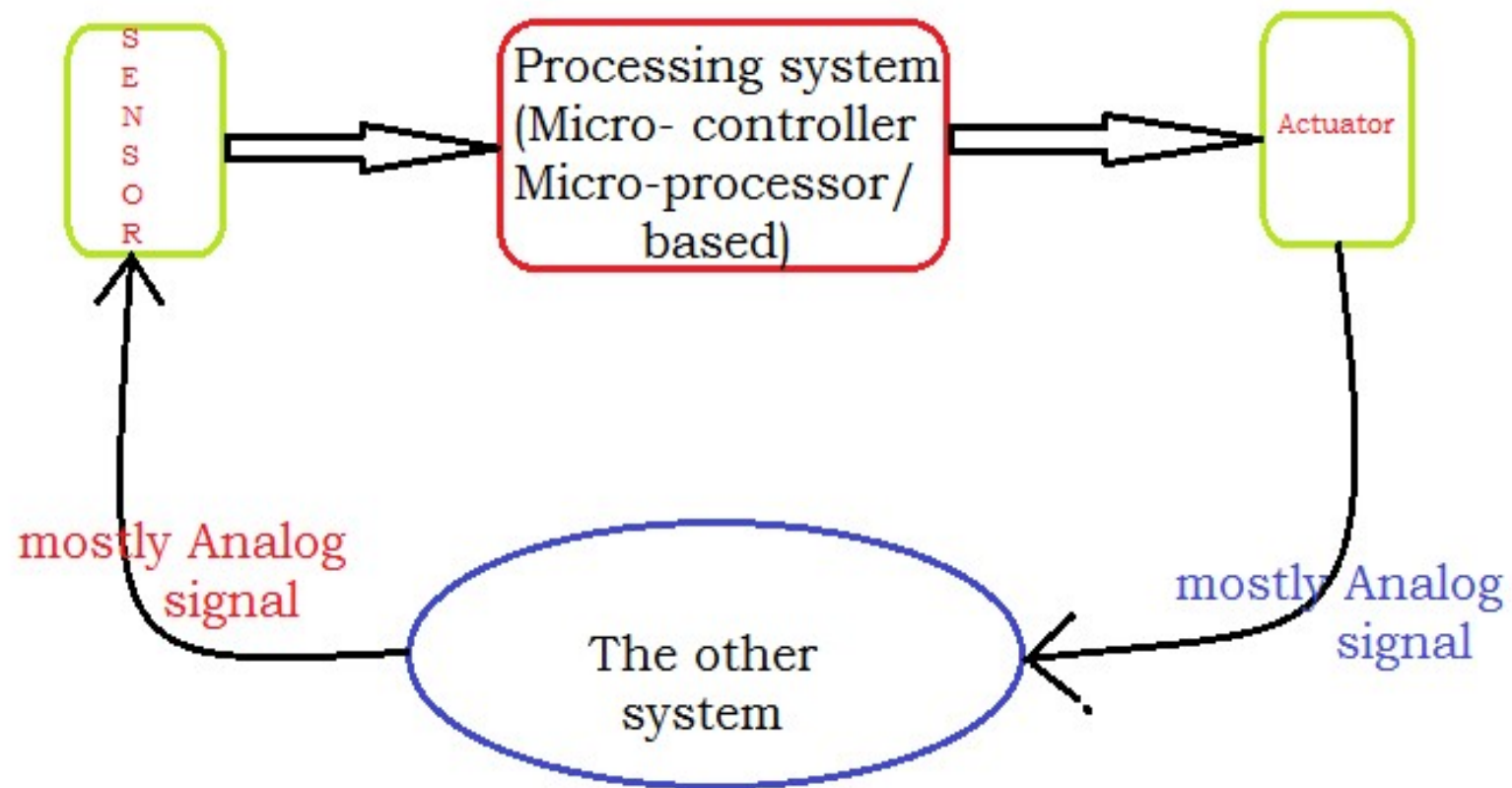
Definition: An **Embedded System (ES)** is a system where **Micro-controller** or **micro-processor based programmable** system is embedded into a large system.

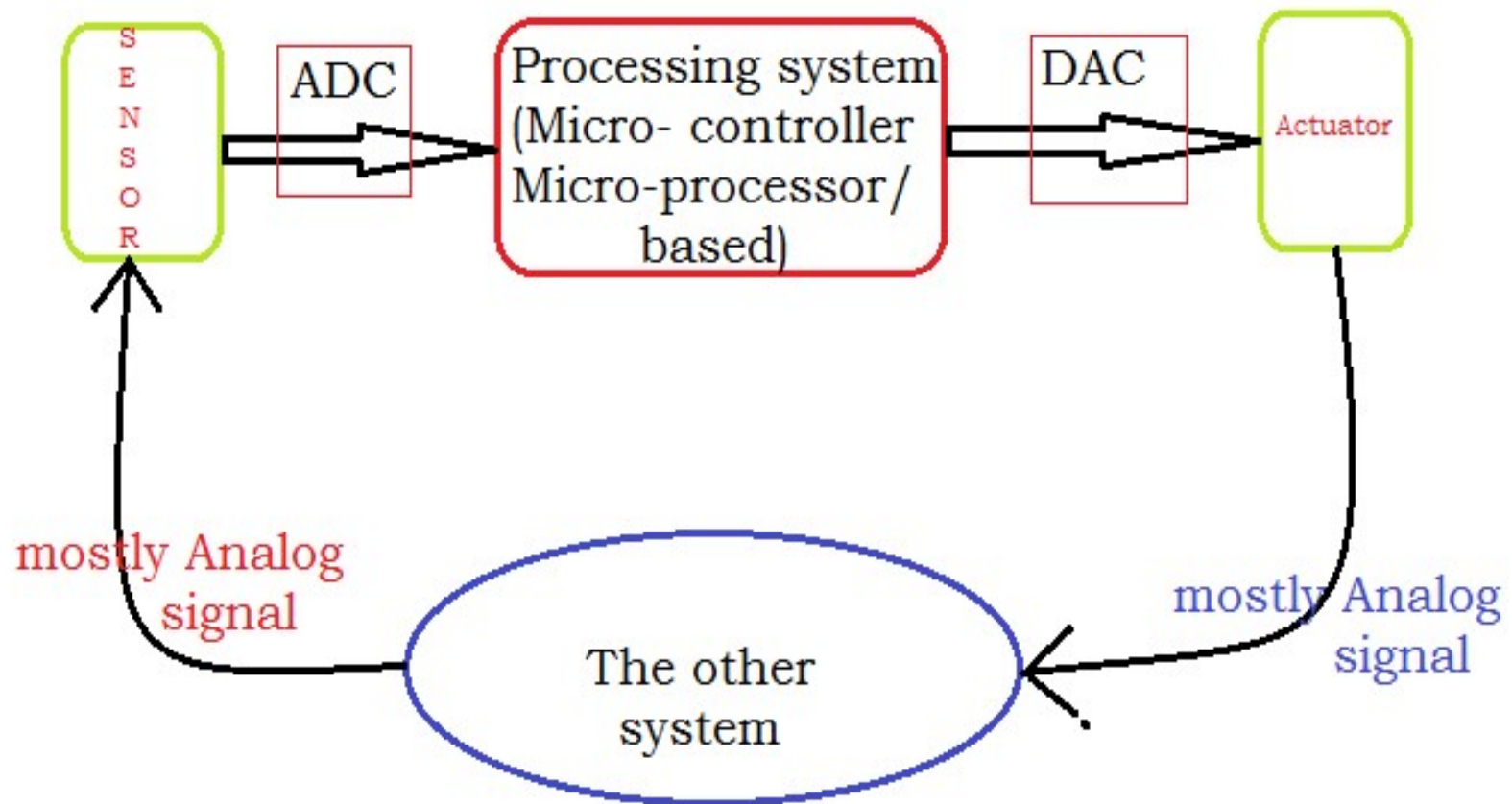


The points which are connected are called sphere of control. (**Red** & **Green** points).











General purpose system Vs Embedded system

General purpose system	Embedded system
Multi-purpose applications	Single purpose / single application or predefined application
High cost High power	Tightly constrained. . Low cost .Low power .portable . Some times real time



Real time system(RTS) Vs

Embedded system (ES)

- RTS: 1. Hard RTS. 2. Soft RTS.

1. Hard RTS:

- . it has stringent time constraint.
- . If it violates controlling takes place.

2. Soft RTS:

- . If it violates QoS is reduced.

All RTS are ESs but all ESs are not RTS.

(ES s are RTS are not RTS)



Reactive system vs Transformational system

- . Reactive system : React to an event.

Exp: camera reacts by clicking a button.

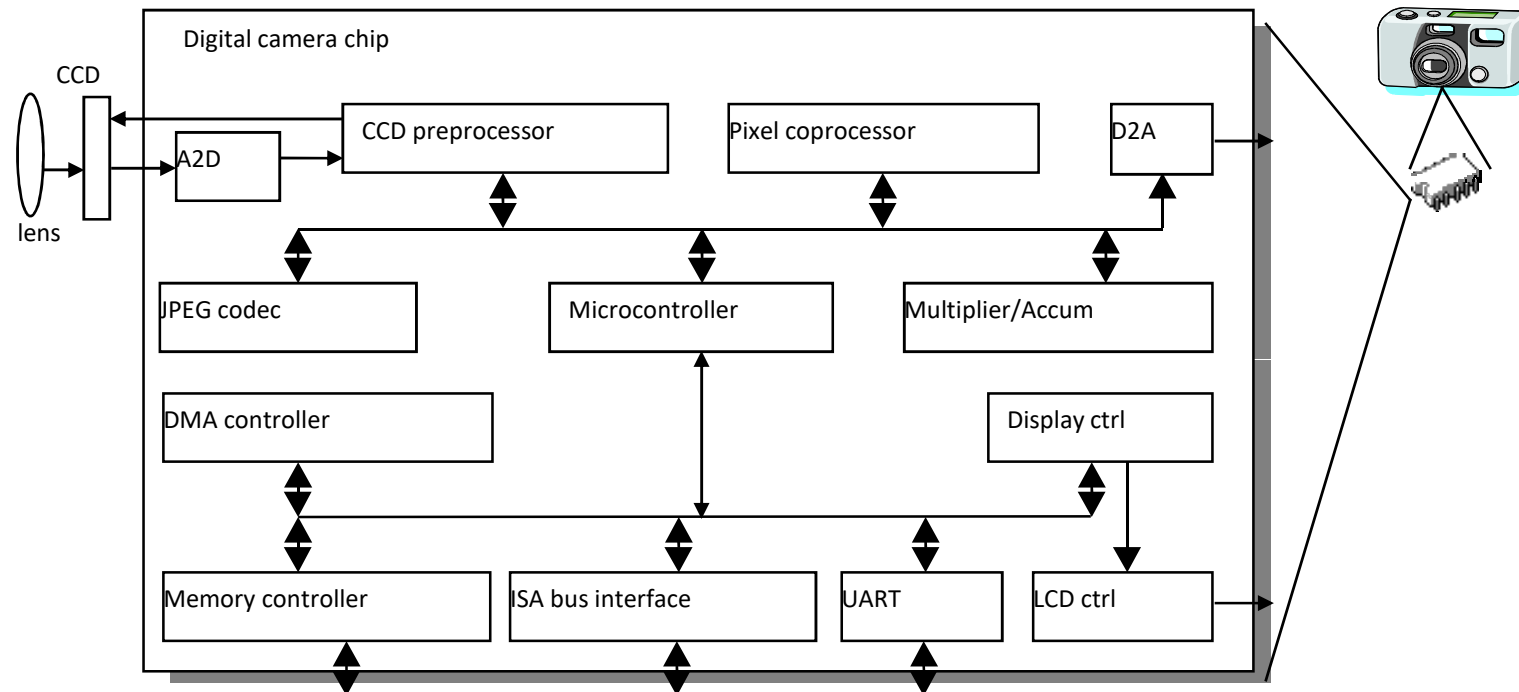
- . Transformational system: sequence of transformations has to be applied.

Exp: Image processing

- . Segmentation .noise removal etc..



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



2. 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



- Performance: Latency & Throughput

Latency: Time to take a task / operation.

Exp:

Let camera A & B both takes **0.25 sec** to take a picture. (It is the **Latency**).

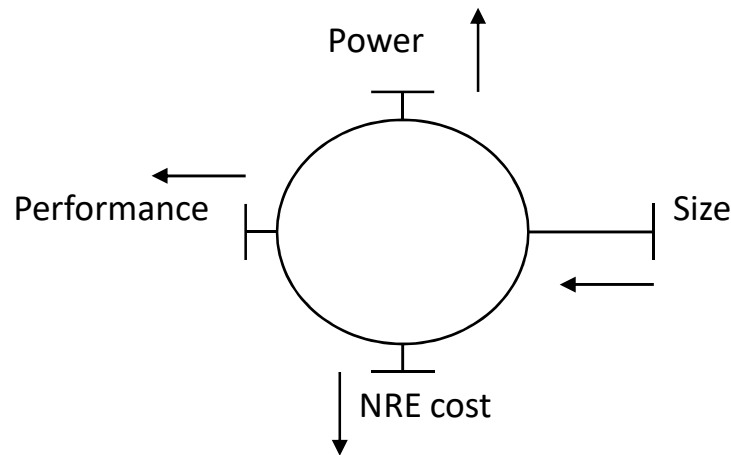
Camera A give 4 pic/sec

Camera B give 8 pic/sec (2 parallel processing units)

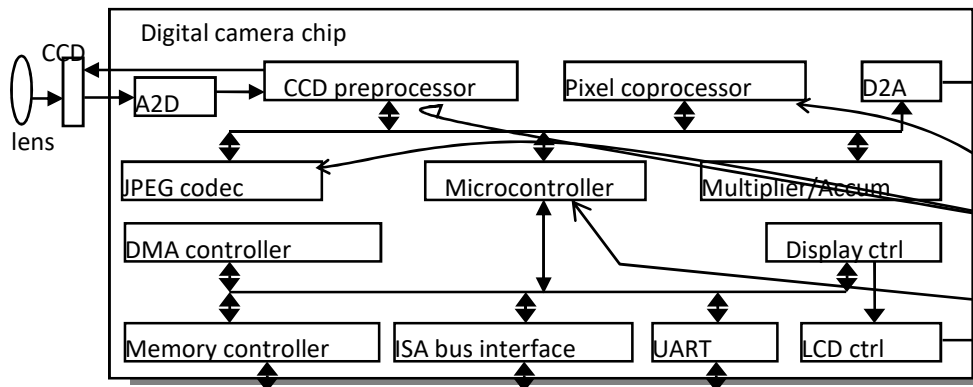
: Camera B has more **Throughput**.



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

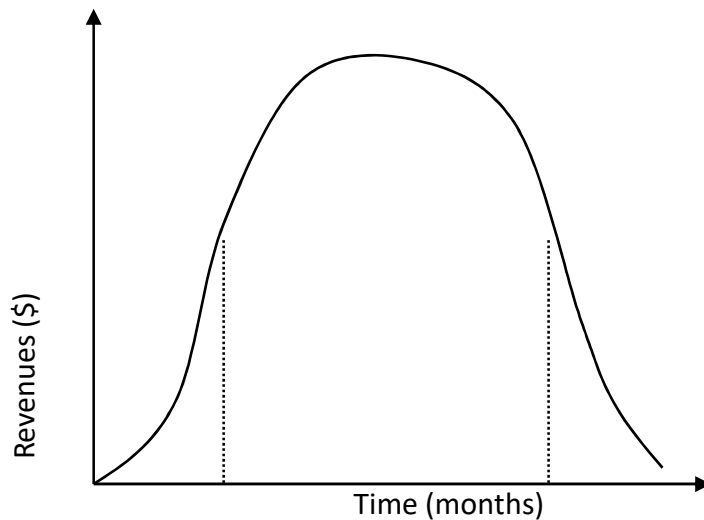


Hardware

Software



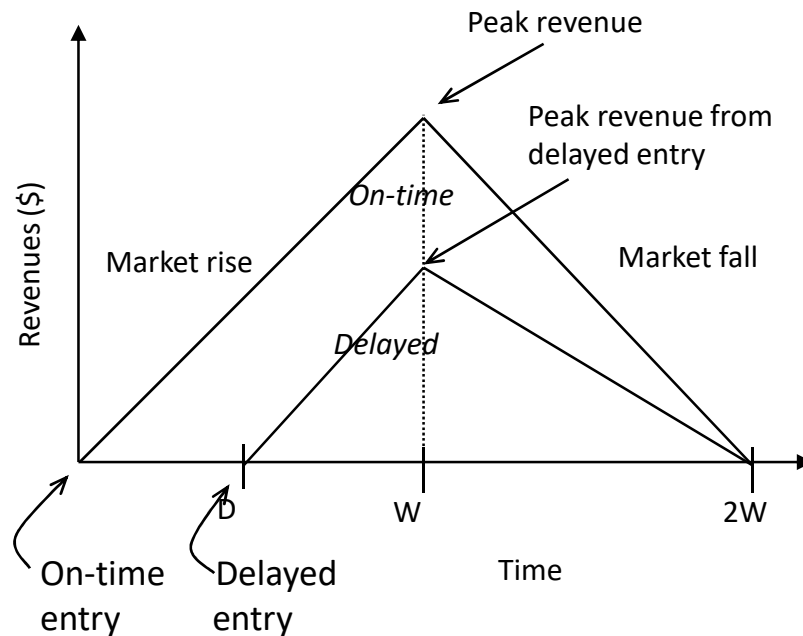
Time-to-market: a demanding design metric



- 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



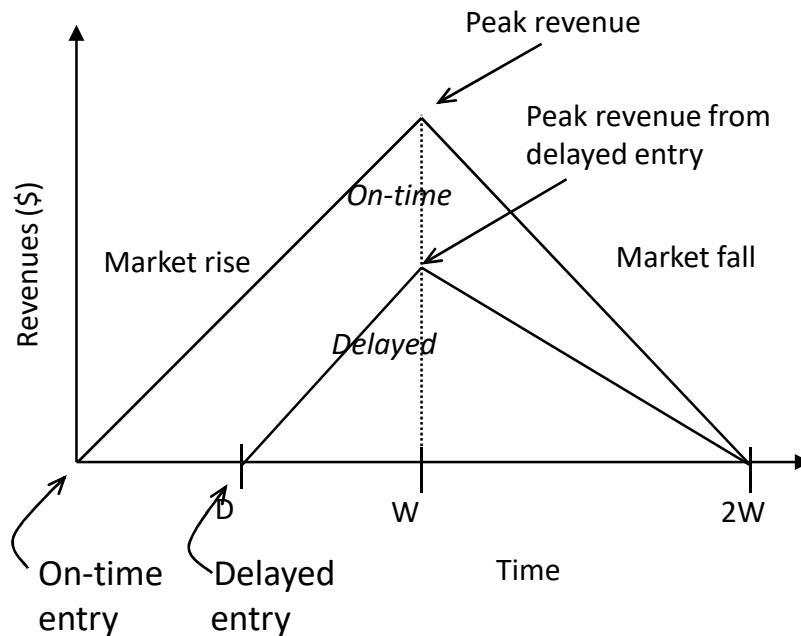
Losses due to delayed market entry



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



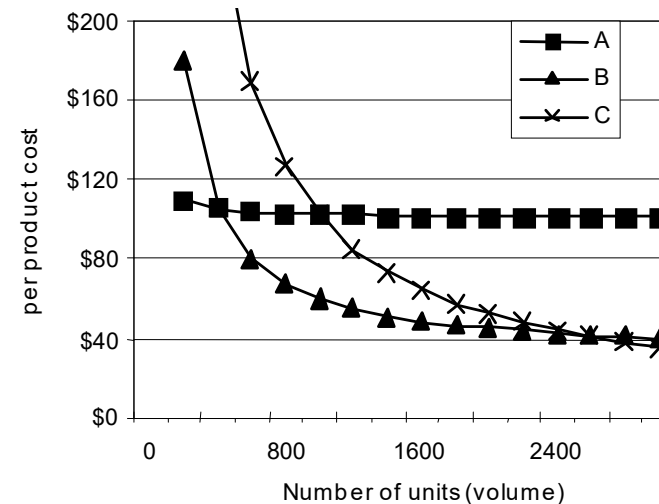
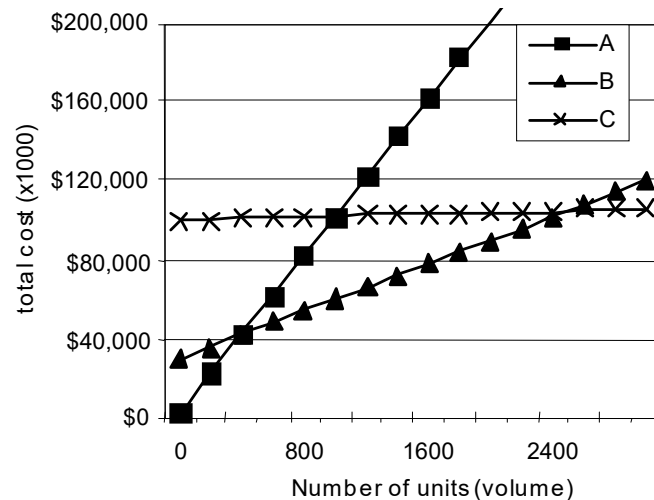
NRE and unit cost metrics

- 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 = \underbrace{\$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



3. PROCESSORS

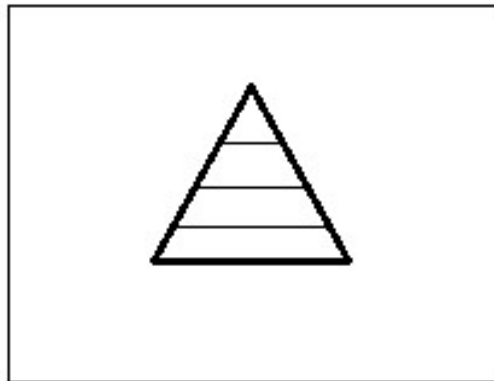
- 1. General purpose
- 2. Application Specific
- 3. Single Purpose







Processor technology

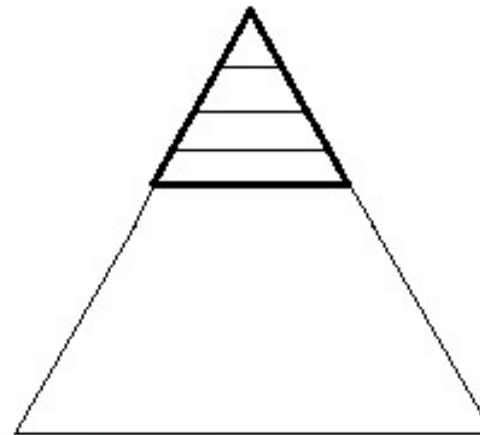


Required functionality




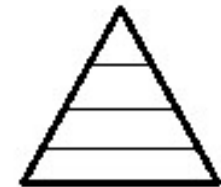
GPP

Map to GPP along
with many systems
like    



ASP

Only  functions

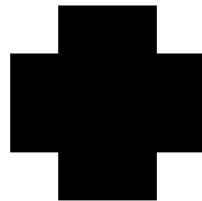


SPP



Processor technology

- Processors vary in their customization for the problem at hand

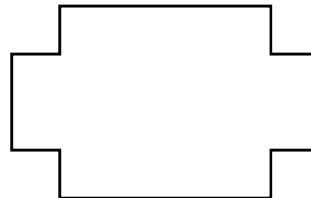


Desired
functionality

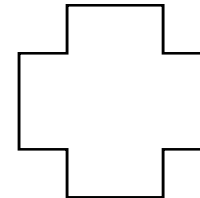
```
total = 0  
for i = 1 to N loop  
    total += M[i]  
end loop
```



General-purpose
processor



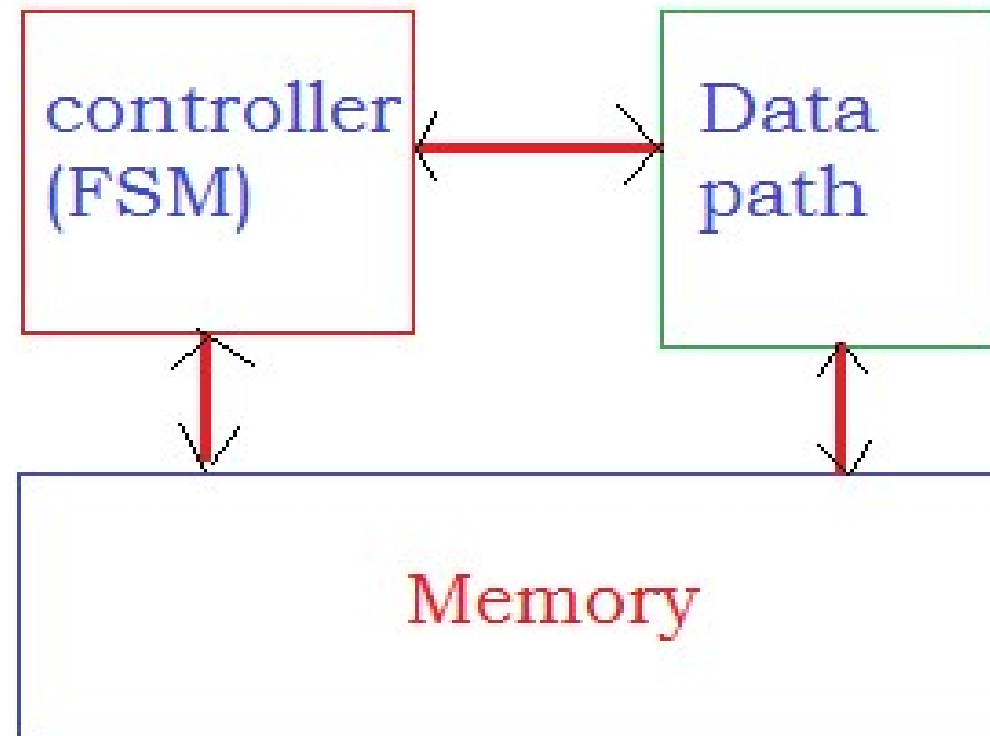
Application-specific
processor



Single-purpose
processor



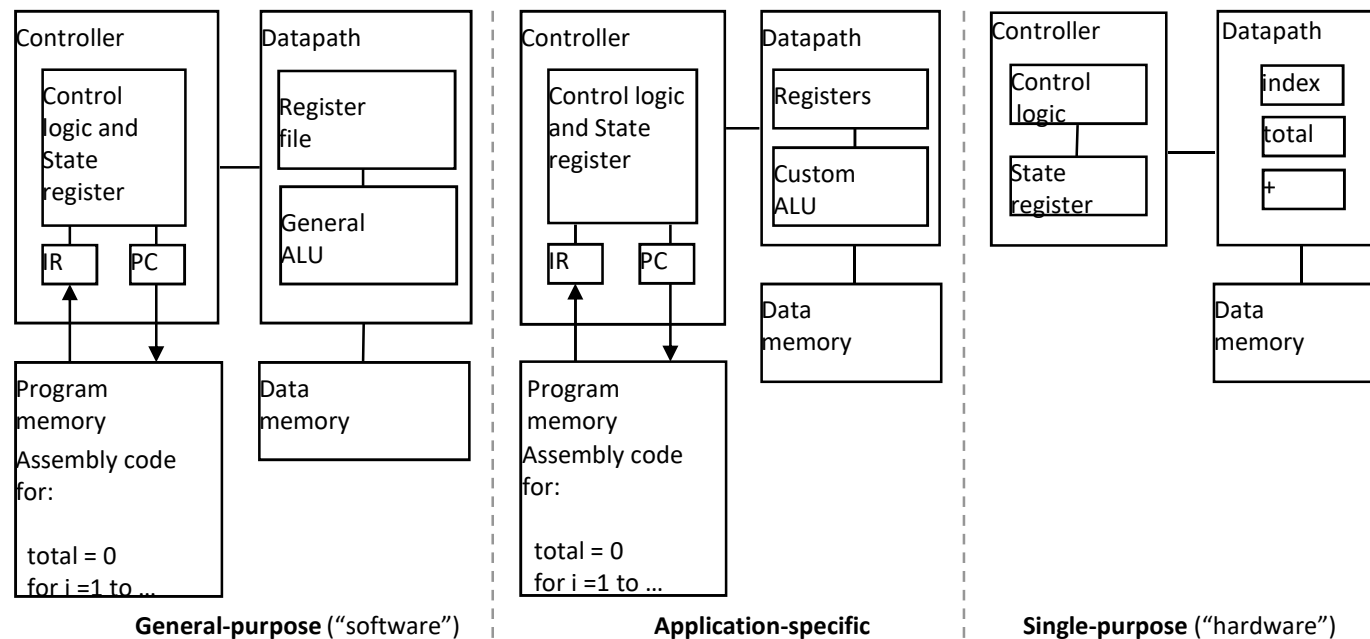
Processor





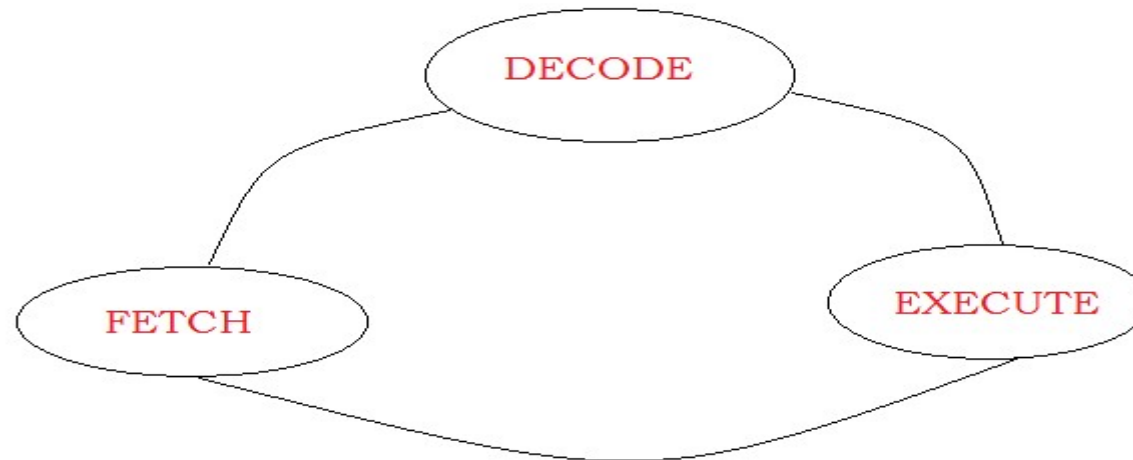
Processor technology

- 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





Instruction execution cycle



- An instruction is **FETCH** from Memory through IR.



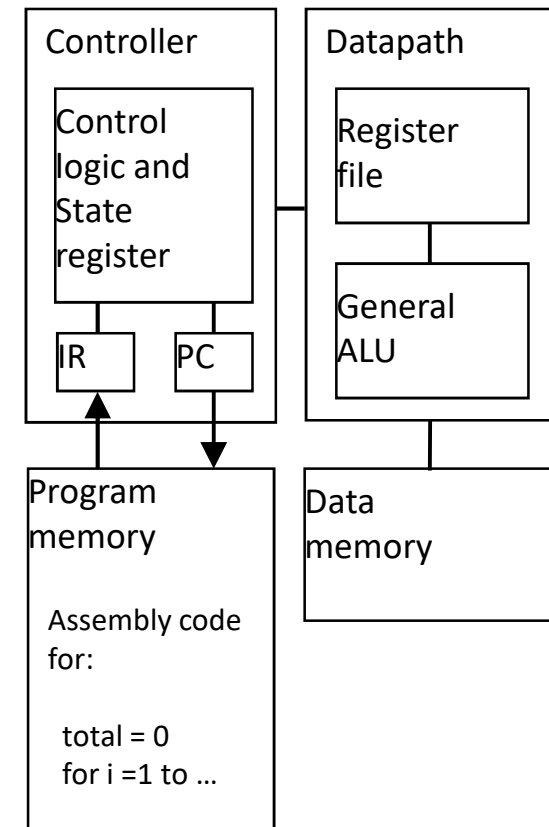
- **Data path:** Registers, Wires, MUX, data elements...
- **Control Path:** **FSM**.
sending the control signals to different elements of the data path.

Controlling is either H/W or S/W.



General-purpose processors

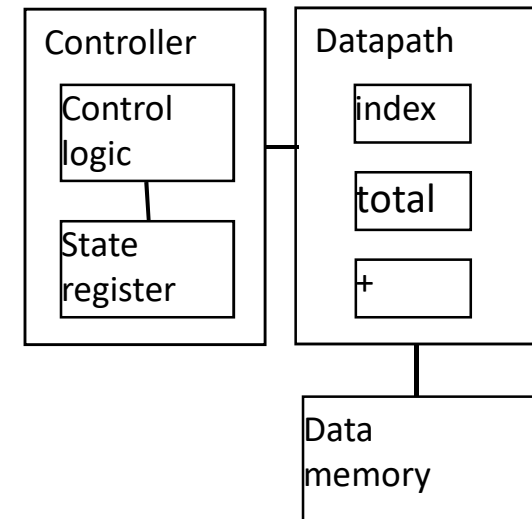
- Programmable device used in a variety of applications
 - Also known as “microprocessor”
- Features
 - Program memory
 - General datapath with large register file and general ALU
- User benefits
 - Low time-to-market and NRE costs
 - High flexibility
- “Pentium” the most well-known, but there are hundreds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





Single-purpose processors

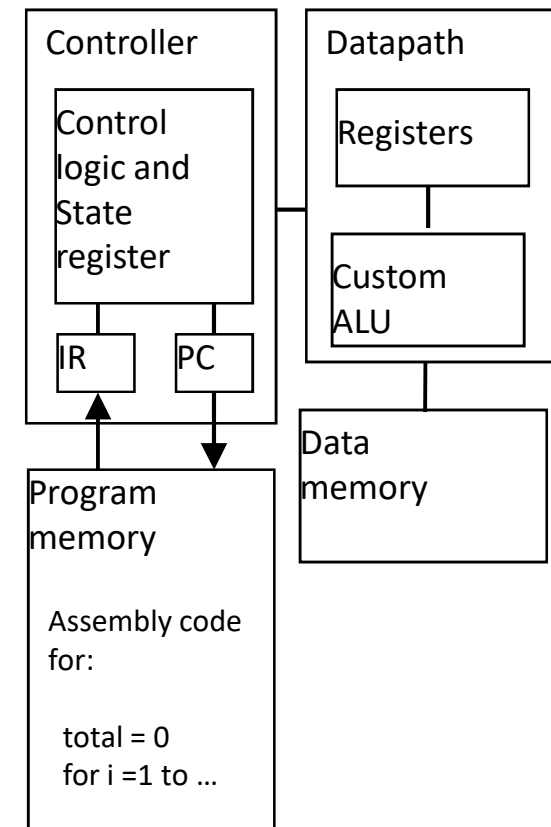
- Digital circuit designed to execute exactly one program
 - a.k.a. coprocessor, accelerator or peripheral
- Features
 - Contains only the components needed to execute a single program
 - No program memory
- Benefits
 - Fast
 - Low power
 - Small size





Application-specific processors

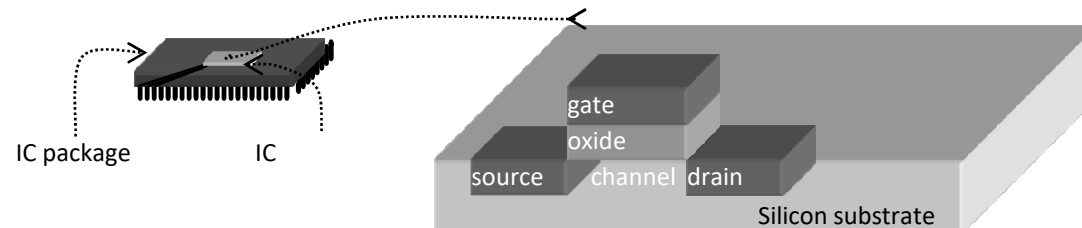
- Programmable processor optimized for a particular class of applications having common characteristics
 - Compromise between general-purpose and single-purpose processors
- Features
 - Program memory
 - Optimized datapath
 - Special functional units
- Benefits
 - Some flexibility, good performance,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





5. 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



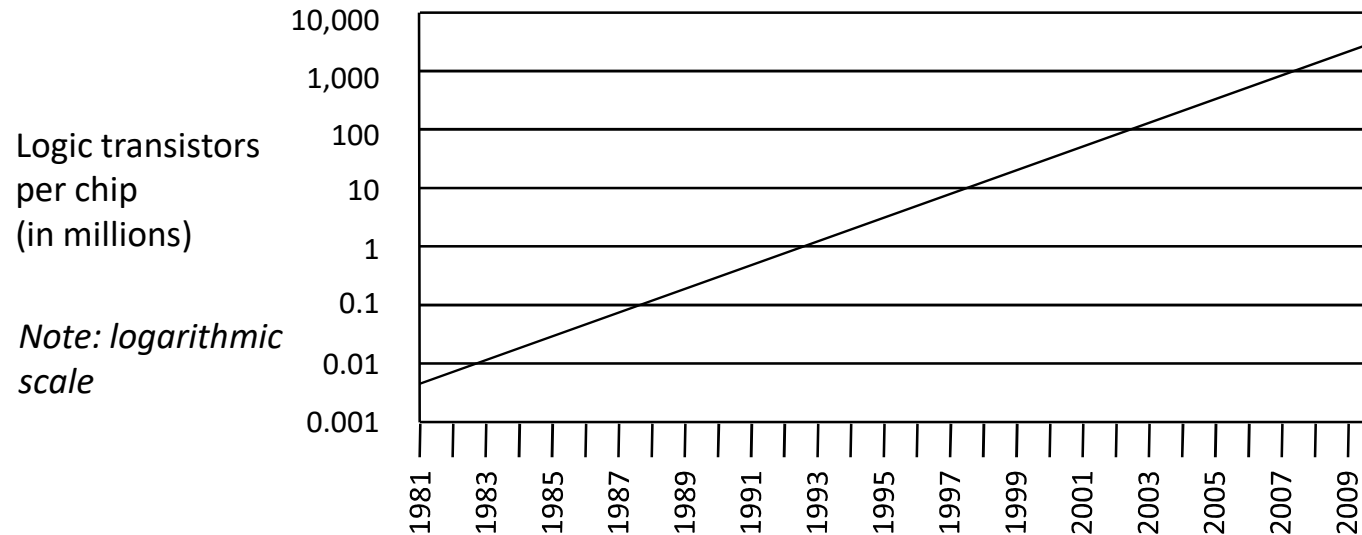
PLD (Programmable Logic Device)

- 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**



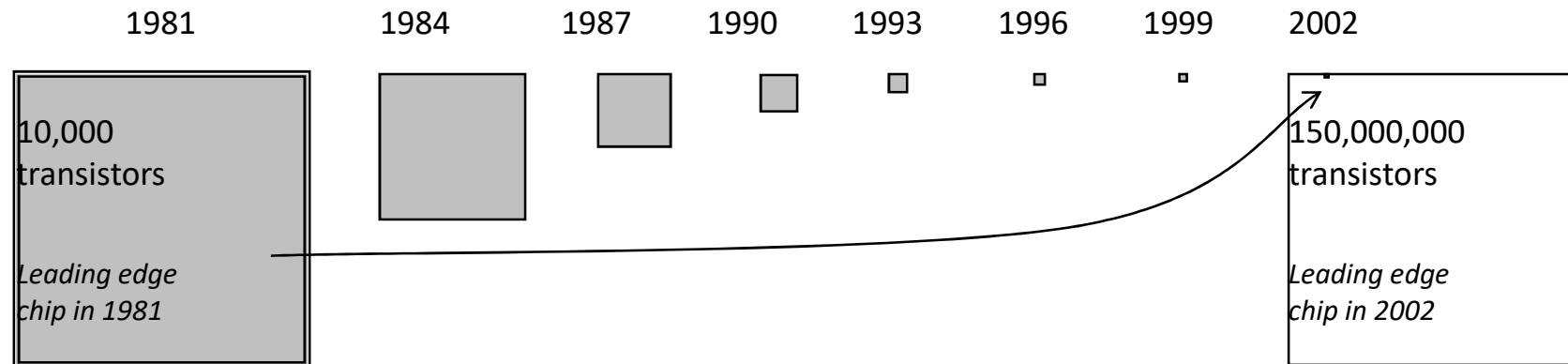


Moore's law

- 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!)*



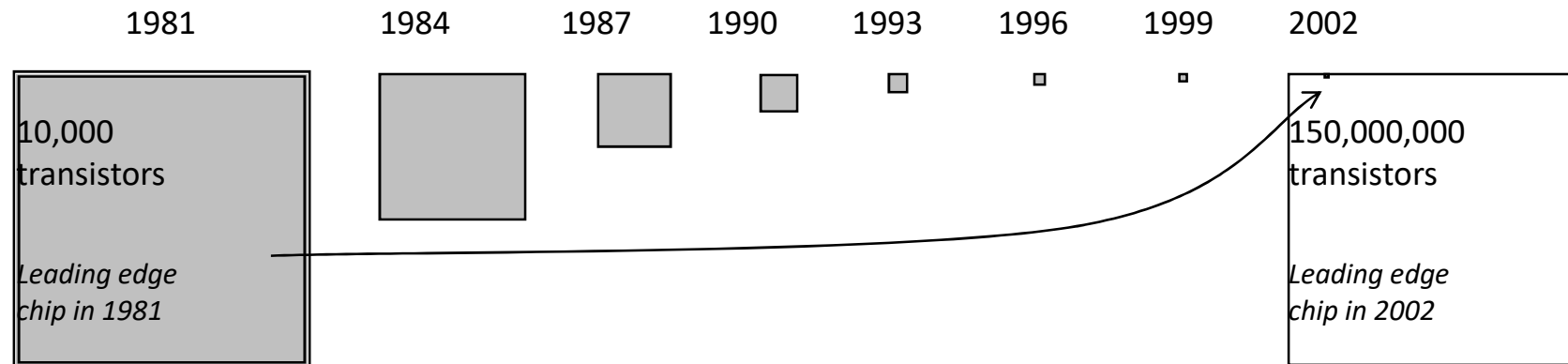
Graphical illustration of Moore's law



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



Graphical illustration of Moore's law

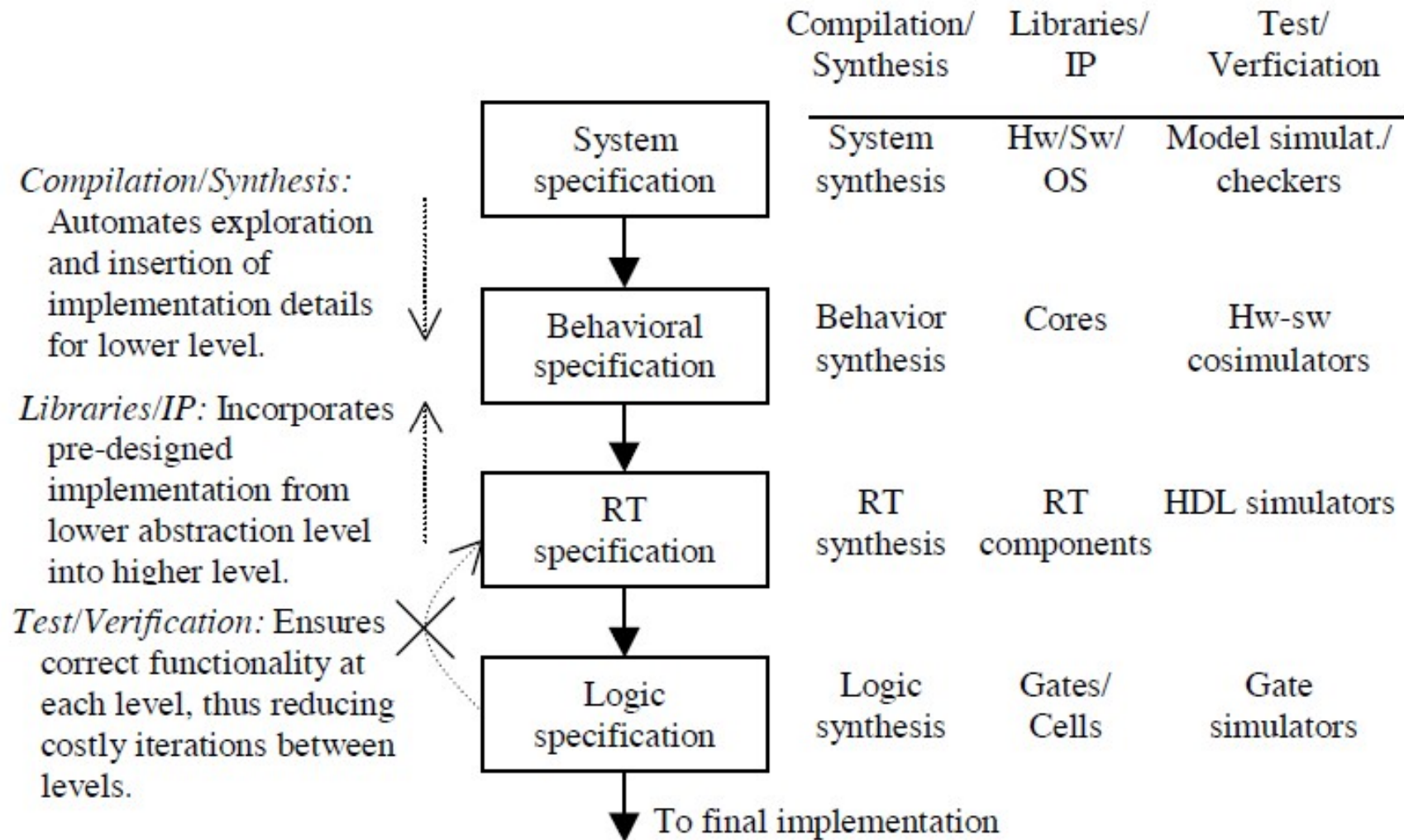


- Something that doubles frequently grows more quickly than most people realize!
 - A 2002 chip can hold about 15,000 1981 chips inside itself

Design Technology

- Design technology involves the manner in which we convert our concept of desired system functionality into an implementation.
- We must not only design the implementation to optimize design metrics, but we must do so quickly.
- To understand how to improve the design process, we must first understand the design process.

Figure 1.9: Ideal top-down design process, and productivity improvers.



Compilation/Synthesis

- designer specify desired functionality in an abstract manner, and automatically generates lower-level implementation details.
- A logic synthesis tool converts Boolean expressions into a connection of logic gates(called a netlist).
- A register-transfer (RT) synthesis tool converts finite-state machines and register-transfers into a datapath of RT components and a controller of Boolean equations.

- A behavioral synthesis tool converts a sequential program into finite-state machines and register transfers.
- a software compiler converts a sequential program to assembly code, which is essentially register-transfer code.
- a system synthesis tool converts an abstract system specification into a set of sequential programs on general and single-purpose processors.
- 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

Libraries/IP

- Libraries involve re-use of pre-existing implementations.
- Using libraries of existing implementations can improve productivity if the time it takes to find, acquire, integrate and test a library item is less than that of designing the item oneself.
- A logic-level library may consist of layouts for gates and cells. An RT-level library may consist of layouts for RT components, like registers, multiplexors, decoders, and functional units.

- A behavioral-level library may consist of commonly used components, such as compression components, bus interfaces, display controllers, and even general-purpose processors.
- a system-level library might consist of complete systems solving particular problems, such as an interconnection of processors with accompanying operating systems and programs to implement an interface to the Internet over an Ethernet network.

Test/Verification

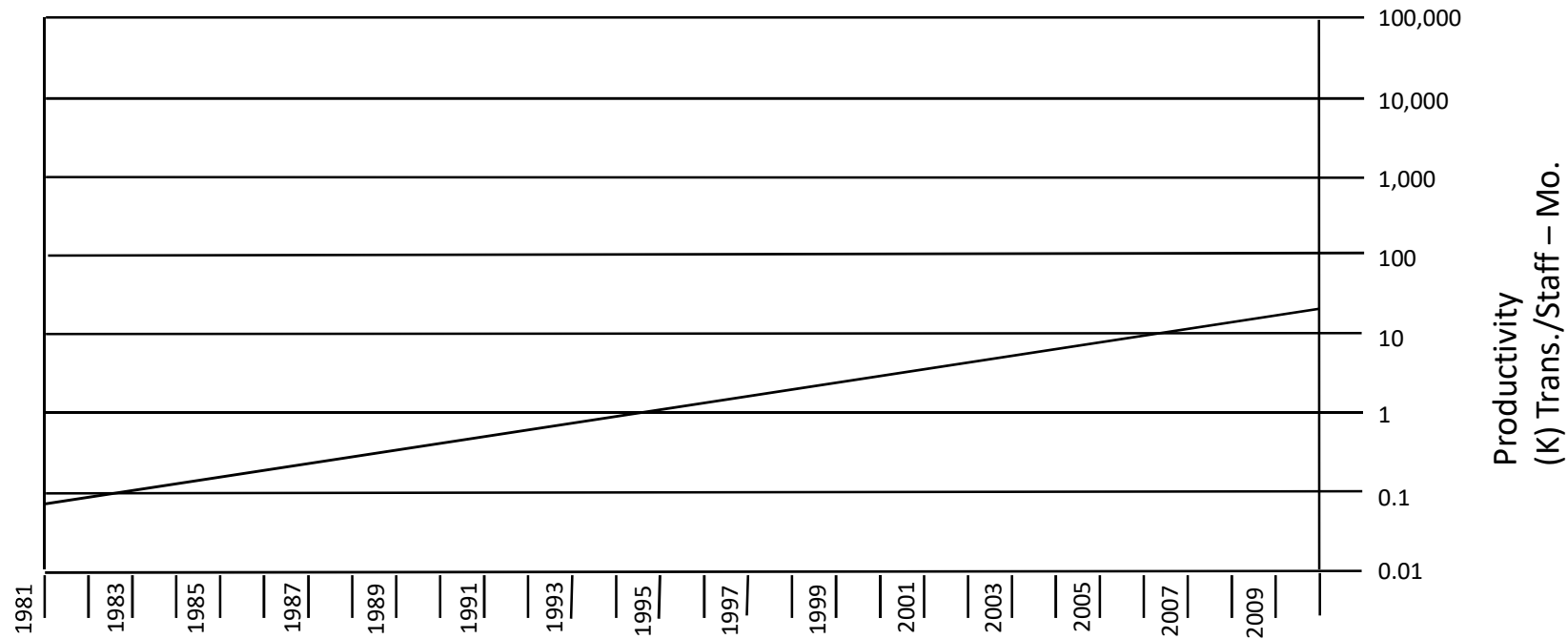
- *Test/Verification involves ensuring that functionality is correct.*
- Such assurance can prevent time-consuming debugging at low abstraction levels and iterating back to high abstraction levels.
- Simulation is the most common method of testing for correct functionality, although more formal verification techniques are growing in popularity.

- At the logic level, gate level simulators provide output signal timing waveforms given input signal waveforms.
- At the RT-level, hardware description language (HDL) simulators execute RT-level descriptions and provide output waveforms given input waveforms.
- At the behavioral level, HDL simulators simulate sequential programs, and co-simulators connect HDL and general-purpose processor simulators to enable hardware/software co-verification.

- At the system level, a model simulator simulates the initial system specification using an abstract computation model, independent of any processor technology, to verify correctness and completeness of the specification.
- Model checkers can also verify certain properties of the specification, such as ensuring that certain simultaneous conditions never occur, or that the system does not deadlock.



Design productivity exponential increase

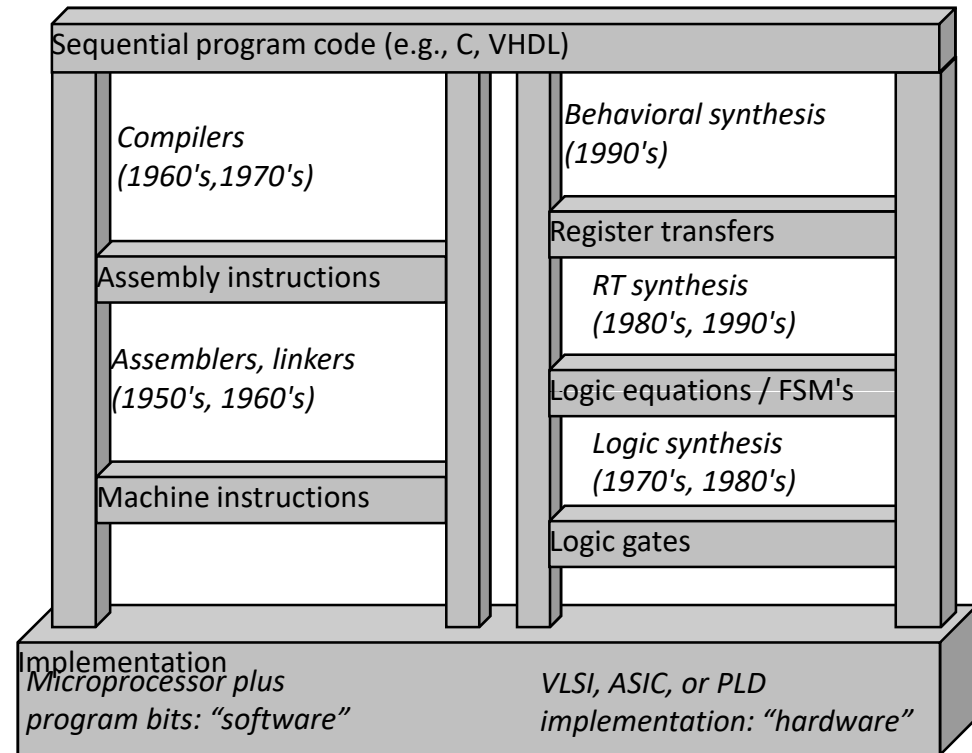


- Exponential increase over the past few decades



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”

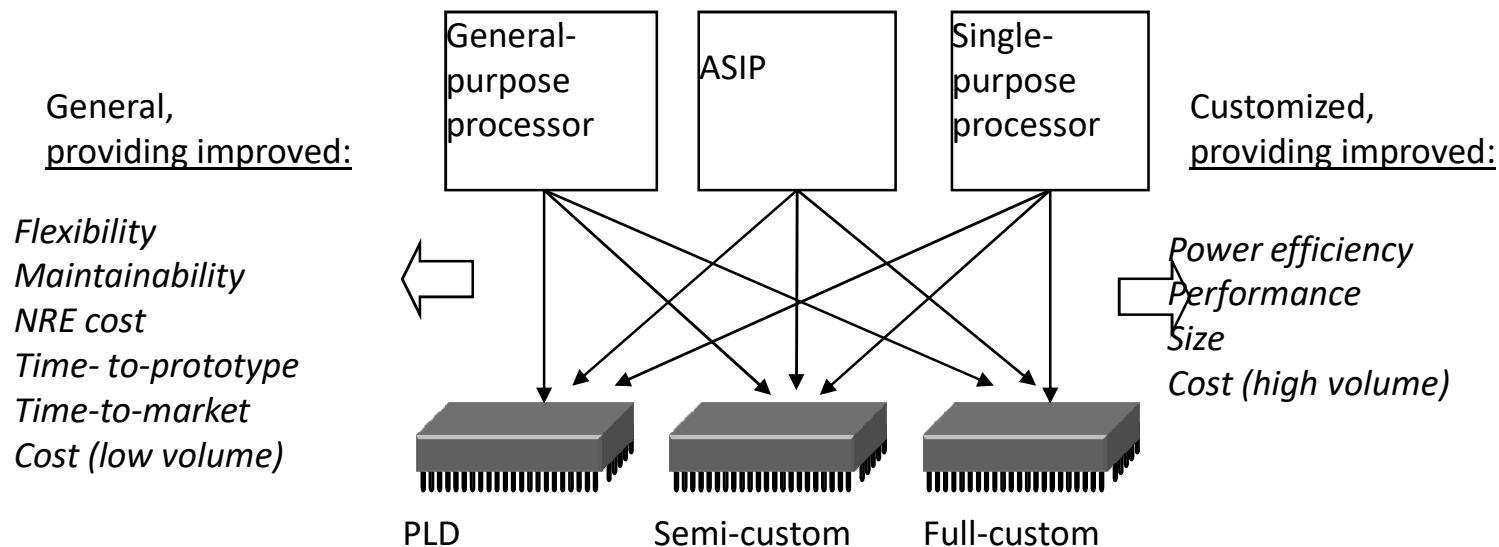


The choice of hardware versus software for a particular function is simply a tradeoff among various design metrics, like performance, power, size, NRE cost, and especially flexibility; there is no fundamental difference between what hardware or software can implement.



Independence of processor and IC technologies

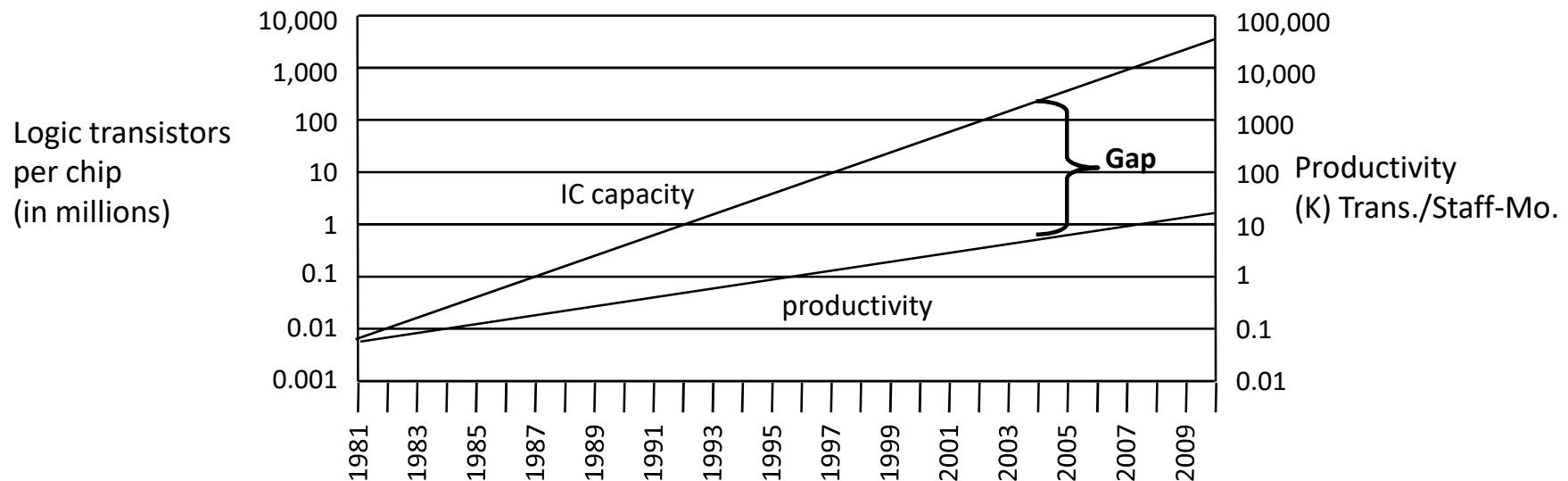
- Basic tradeoff
 - General vs. custom
 - With respect to processor technology or IC technology
 - The two technologies are independent





Design productivity gap

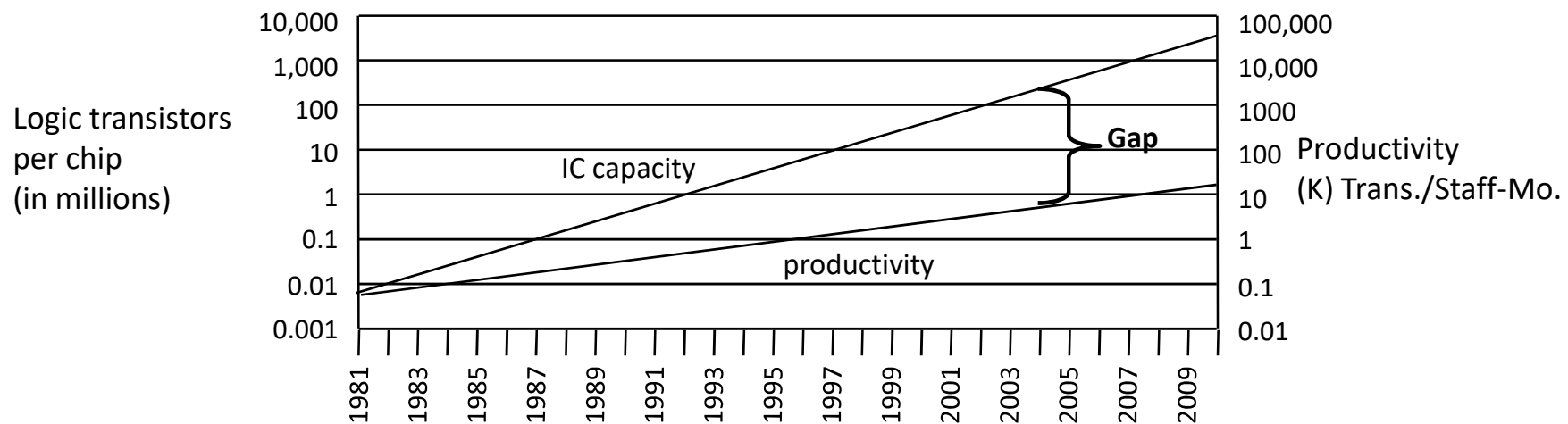
- While designer productivity has grown at an impressive rate over the past decades, the rate of improvement has not kept pace with chip capacity





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

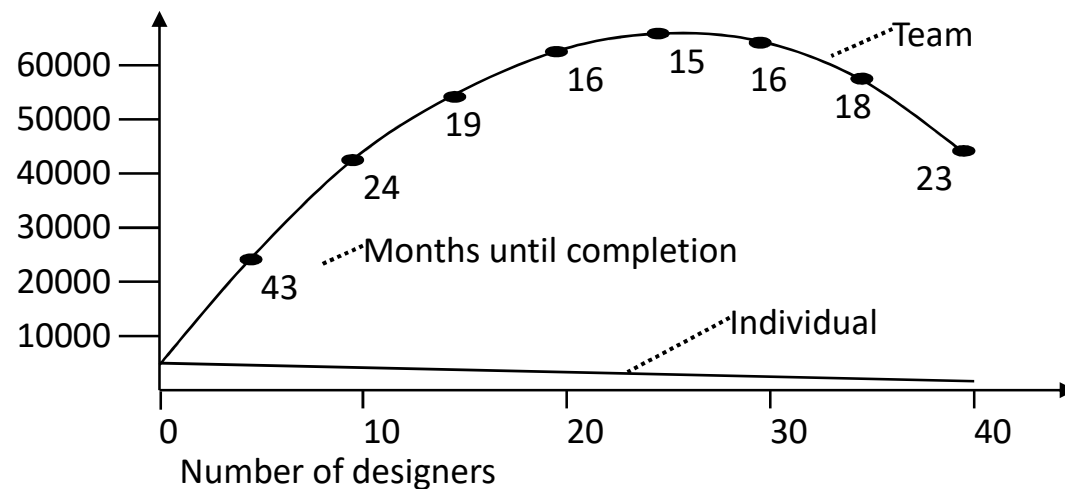




The mythical man-month

- 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



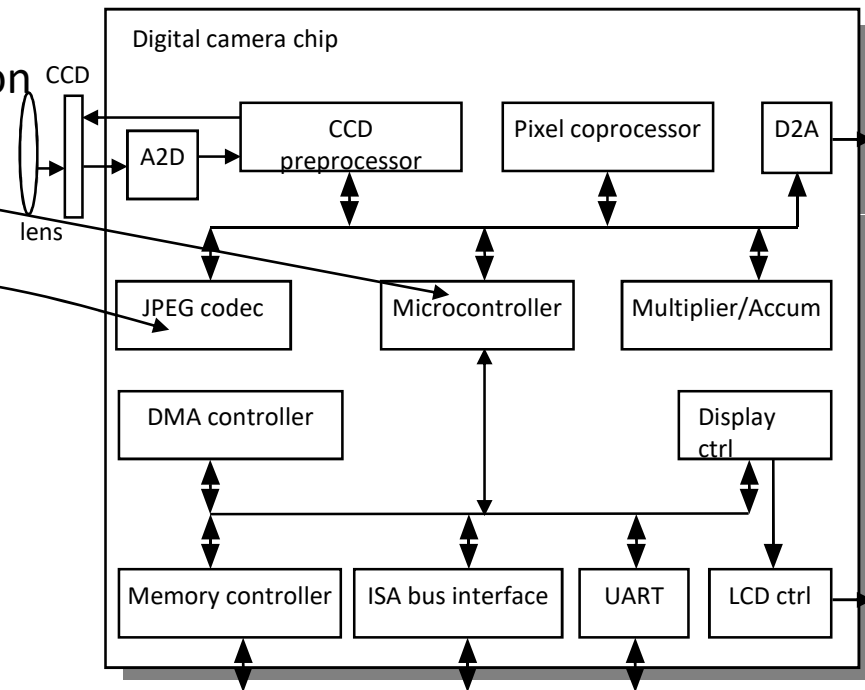
single-purpose processors: Outline

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



Introduction

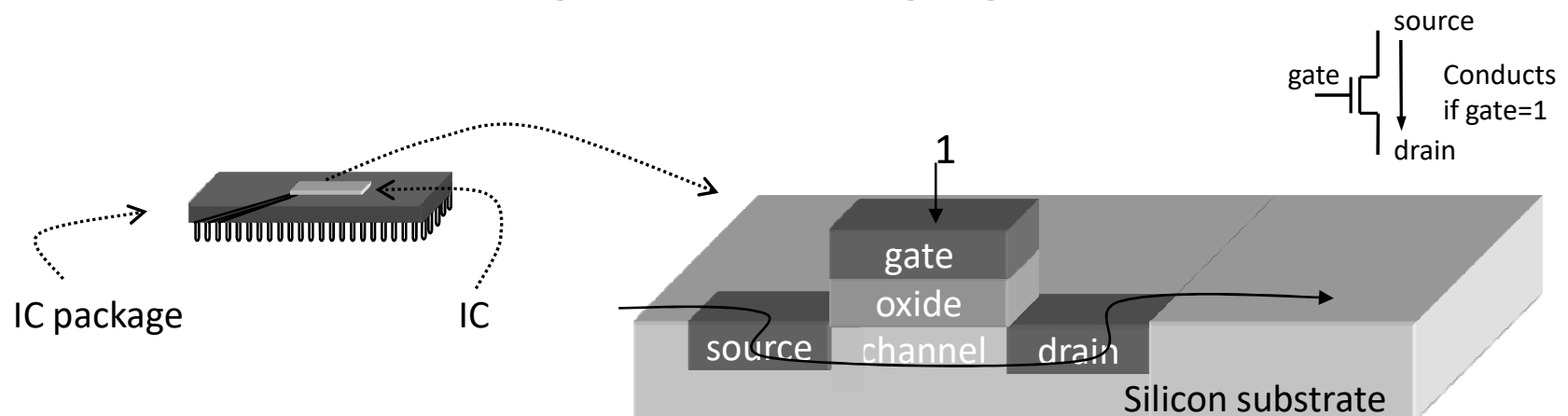
- Processor
 - Digital circuit that performs a computation tasks
 - Controller and datapath
 - General-purpose: variety of computation tasks
 - Single-purpose: one particular computation task
 - Custom single-purpose: non-standard task
- A custom single-purpose processor may be
 - Fast, small, low power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





CMOS transistor on silicon

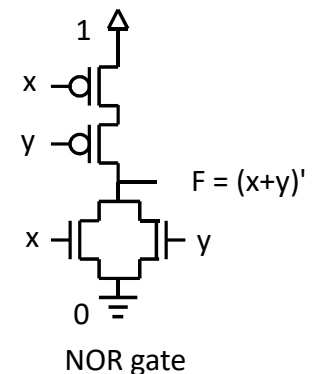
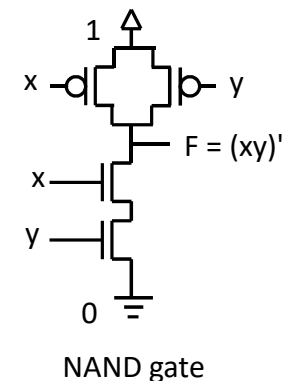
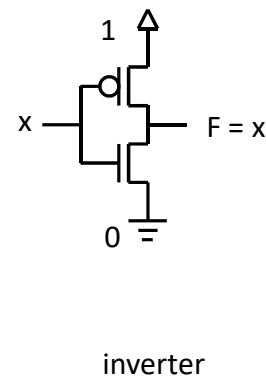
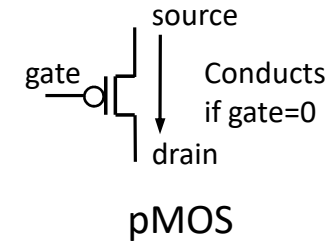
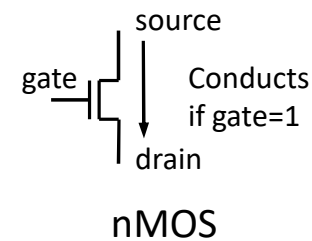
- Transistor
 - The basic electrical component in digital systems
 - Acts as an on/off switch
 - Voltage at “gate” controls whether current flows from source to drain
 - Don’t confuse this “gate” with a logic gate





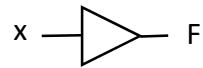
CMOS transistor implementations

- 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



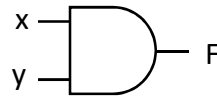


Basic logic gates



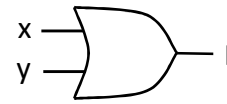
x	F
0	0
1	1

$F = x$
Driver



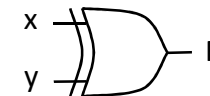
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

$F = x y$
AND



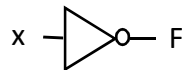
x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

$F = x + y$
OR



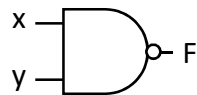
x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

$F = x \oplus y$
XOR



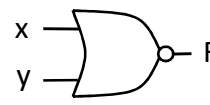
x	F
0	1
1	0

$F = x'$
Inverter



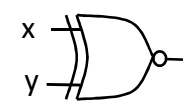
x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

$F = (x y)'$
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

$F = x \odot y$
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

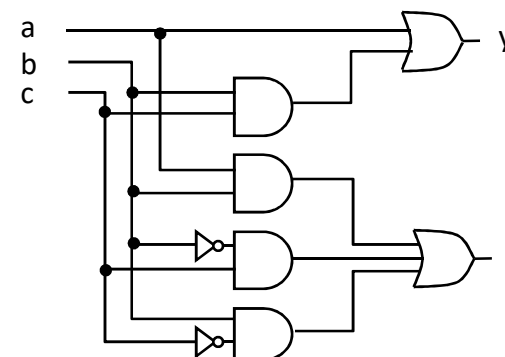
y	bc	00	01	11	10
		0	0	1	0
	a	0	0	1	0
		1	1	1	1

$$y = a + bc$$

z	bc	00	01	11	10
		0	1	0	1
	a	0	1	0	1
		1	0	1	1

$$z = ab + b'c + bc'$$

E) Logic Gates





Combinational components

<p> $O =$ I_0 if $S=0..00$ I_1 if $S=0..01$ \dots $I_{(m-1)}$ if $S=1..11$ </p>	<p> $O_0 = 1$ if $I=0..00$ $O_1 = 1$ if $I=0..01$ \dots $O_{(n-1)} = 1$ if $I=1..11$ </p>	<p> $sum = A+B$ (first n bits) $carry = (n+1)'th$ bit of $A+B$ </p>	<p> $less = 1$ if $A < B$ $equal = 1$ if $A = B$ $greater = 1$ if $A > B$ </p>	<p> $O = A \text{ op } B$ op determined by S. </p>
	<p>With enable input $e \rightarrow$ all O's are 0 if $e=0$</p>	<p>With carry-in input $C_i \rightarrow$ $sum = A + B + C_i$</p>		<p>May have status outputs carry, zero, etc.</p>



Sequential components

<p>Q = 0 if clear=1, I if load=1 and clock=1, Q(previous) otherwise.</p>	<p>Q = lsb - Content shifted - I stored in msb</p>	<p>Q = 0 if clear=1, Q(prev)+1 if count=1 and clock=1.</p>

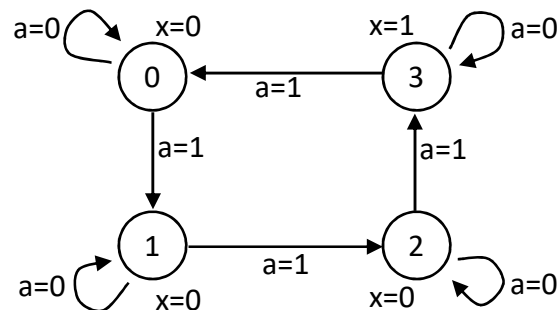


Sequential logic design

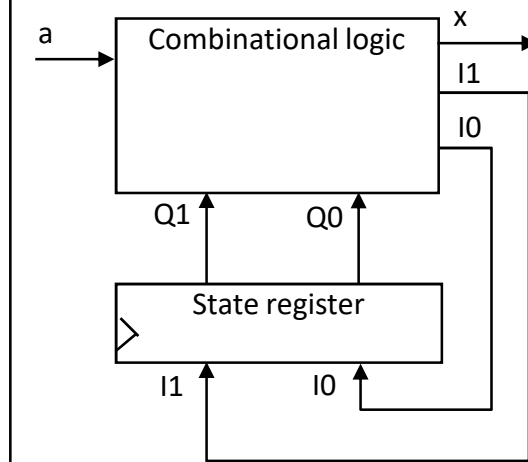
A) Problem Description

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
0	1	0	0	1	0
0	1	1	1	0	0
1	0	0	1	0	0
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	1

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



Sequential logic design (cont.)

E) Minimized Output Equations

I1 Q_1Q_0

a	00	01	11	10
0	0	0	1	1
1	0	1	0	1

$I1 = Q_1'Q_0a + Q_1a' + Q_1Q_0'$

I0 Q_1Q_0

a	00	01	11	10
0	0	1	1	0
1	1	0	0	1

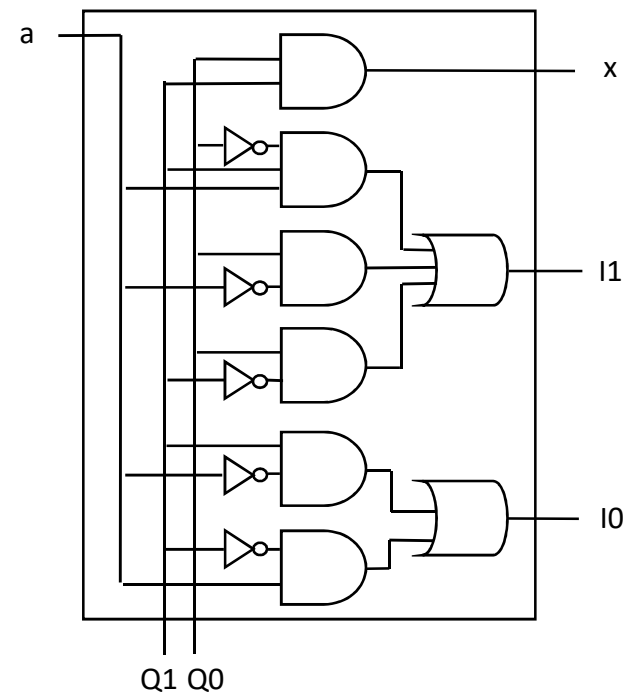
$I0 = Q_0a' + Q_0'a$

x Q_1Q_0

a	00	01	11	10
0	0	0	1	0
1	0	0	1	0

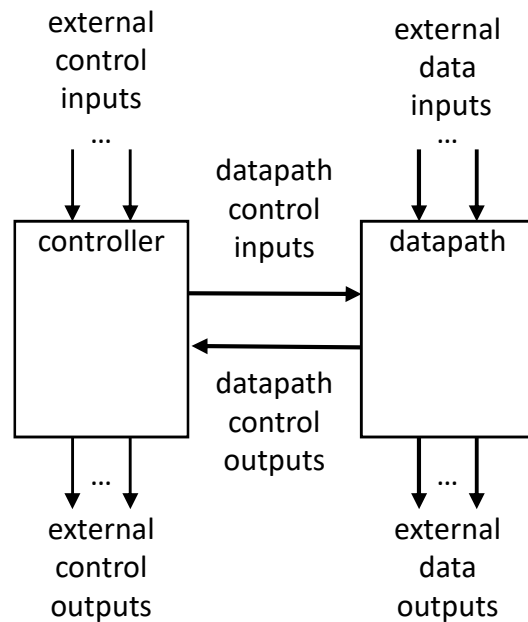
$x = Q_1Q_0$

F) Combinational Logic

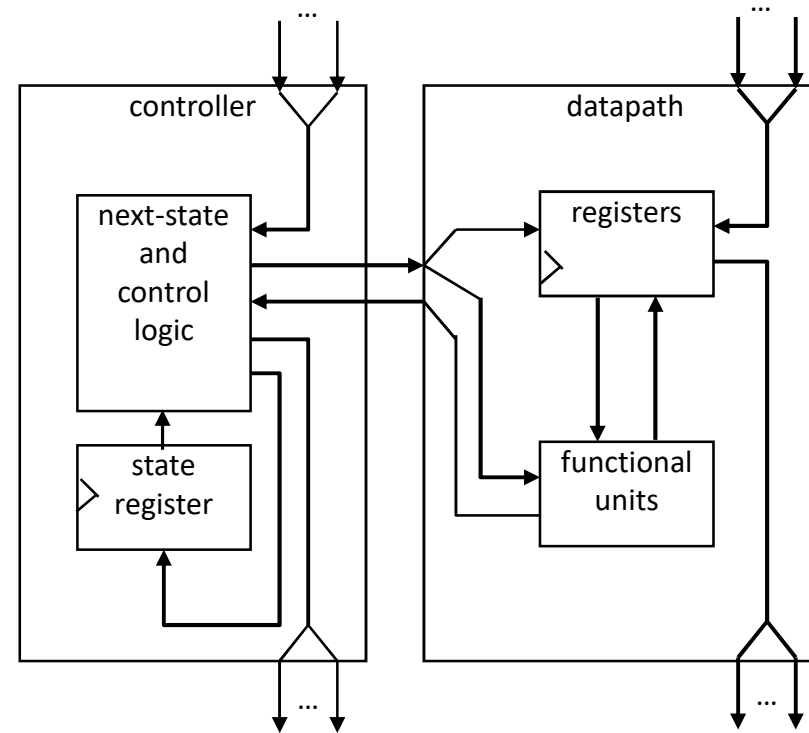




Custom single-purpose processor basic model



controller and datapath



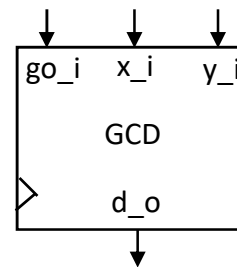
a view inside the controller and datapath



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



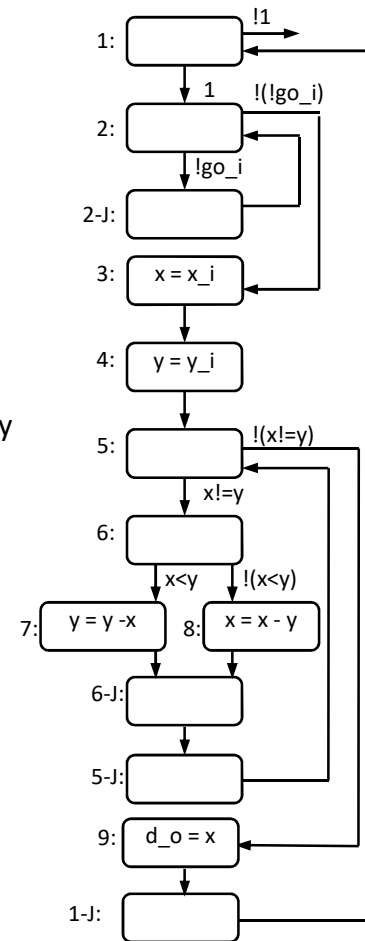
(b) desired functionality

```

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;
8:     else
9:       x = x - y;
9:   d_o = x;
}

```

(c) state diagram

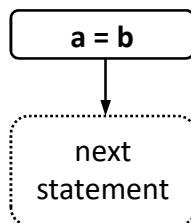




State diagram templates

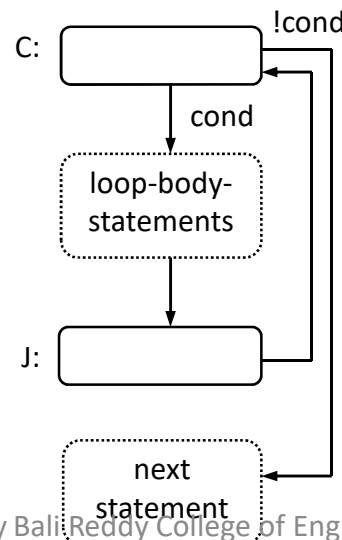
Assignment statement

a = b
next
statement



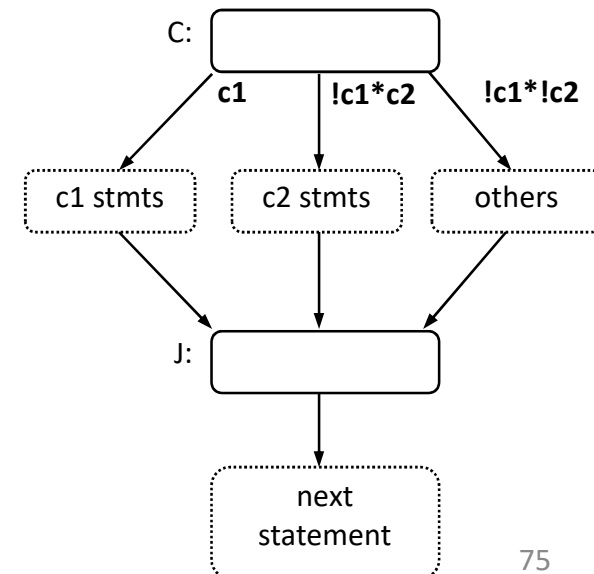
Loop statement

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



Branch statement

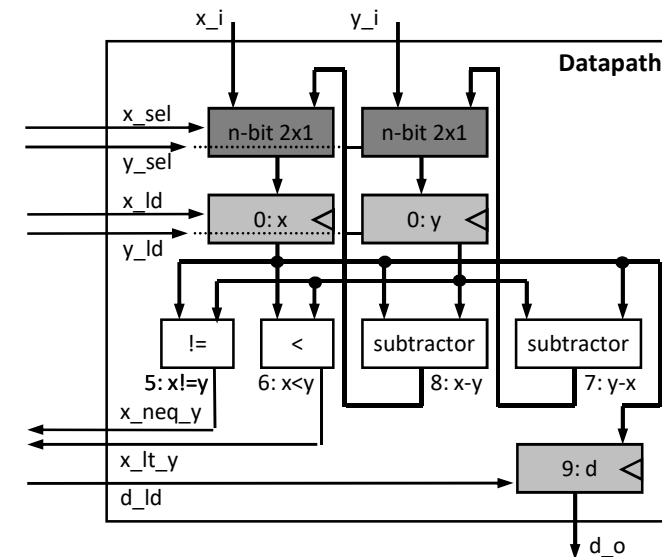
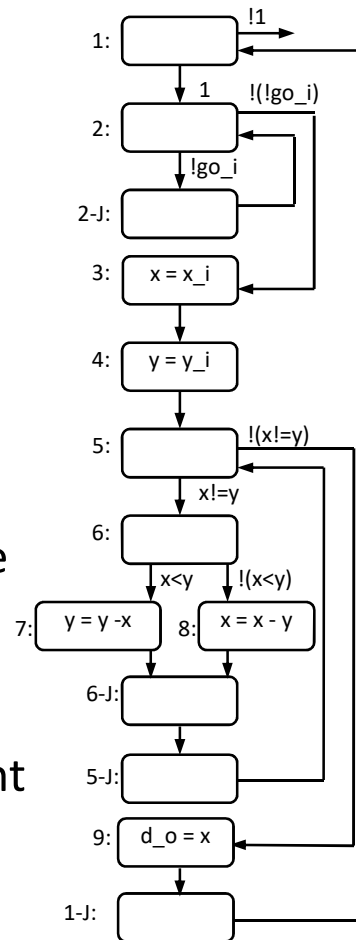
if (c1)
c1 stmts
else if c2
c2 stmts
else
other stmts
next statement





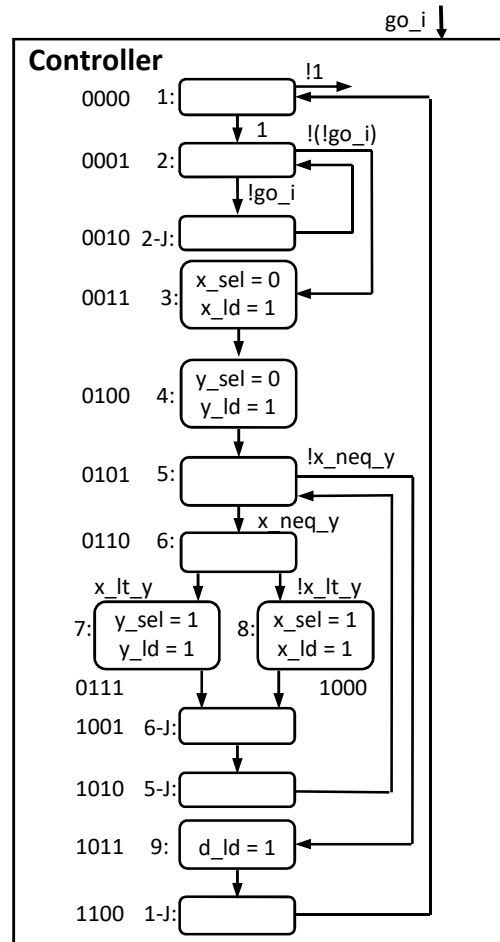
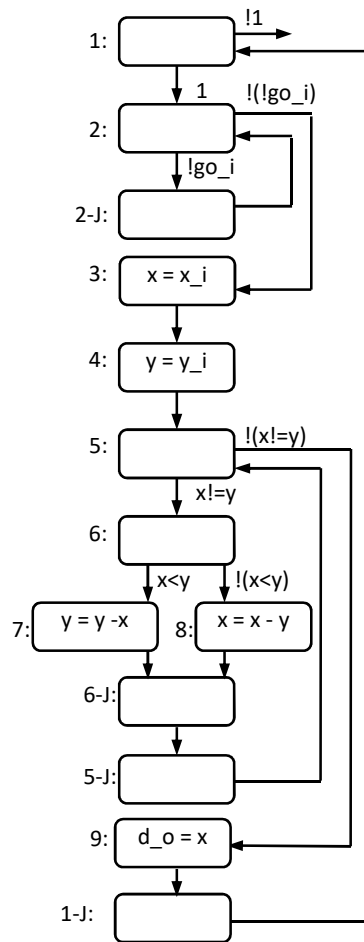
Creating the datapath

- 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

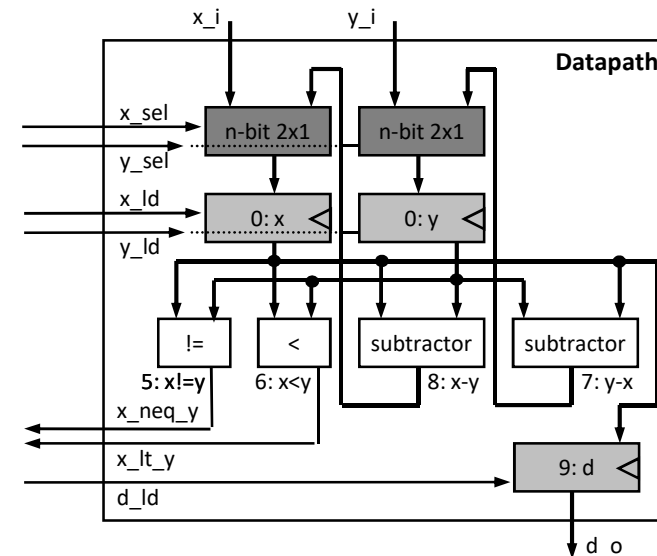




Creating the controller's FSM

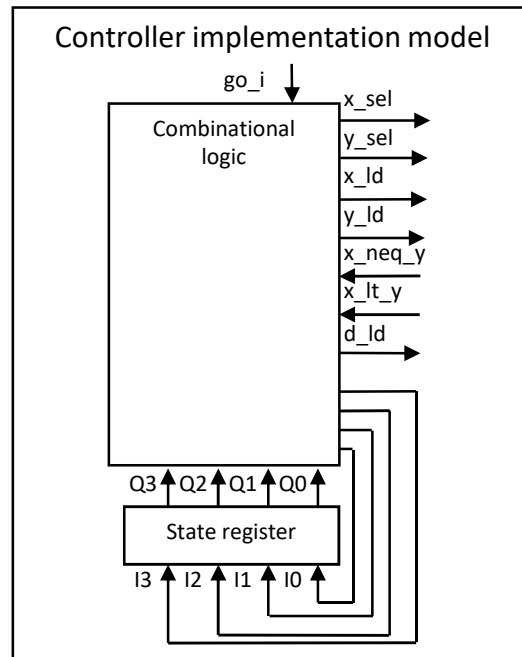


- Same structure as FSMD
- Replace complex actions/conditions with datapath configurations

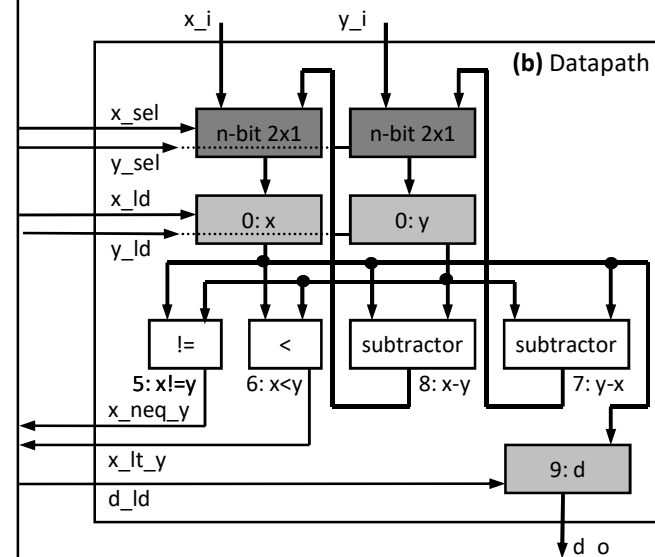
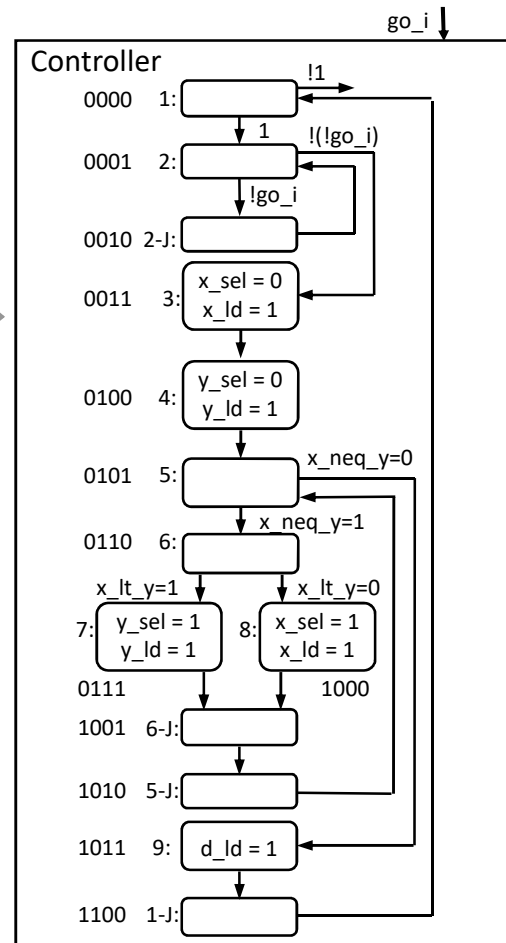




Splitting into a controller and datapath



→





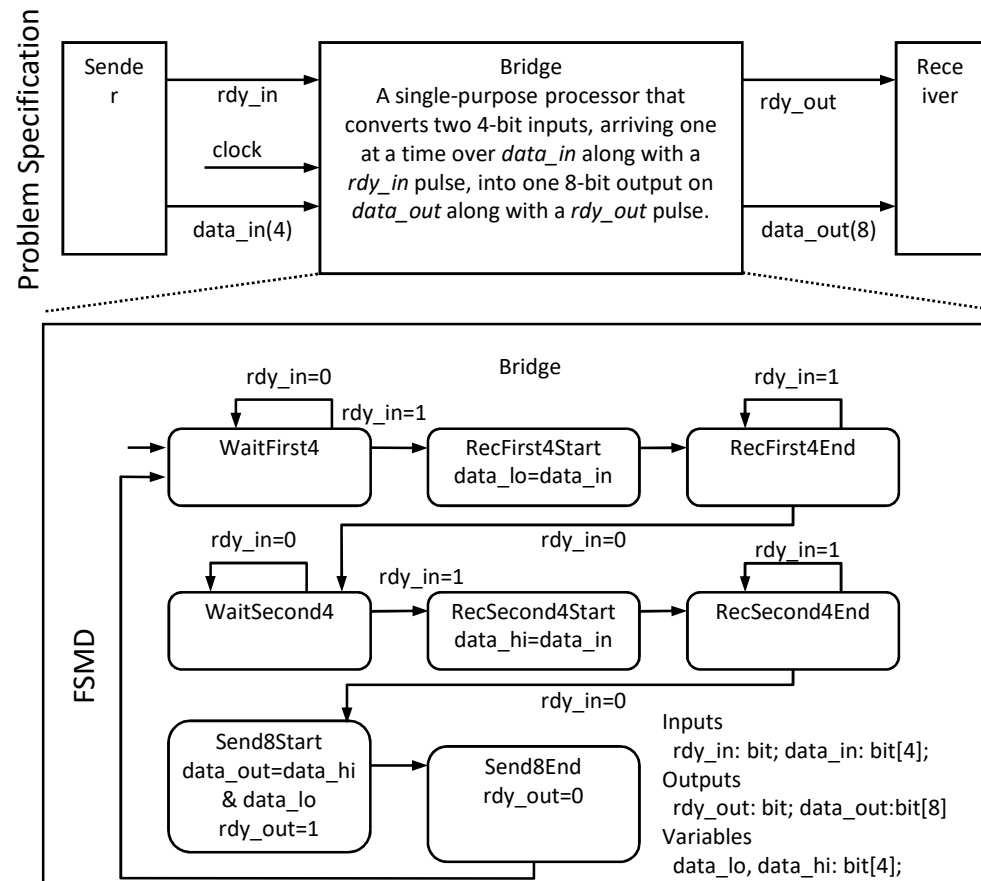
Controller state table for the GCD example

Inputs							Outputs								
Q3	Q2	Q1	Q0	x_ne q_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



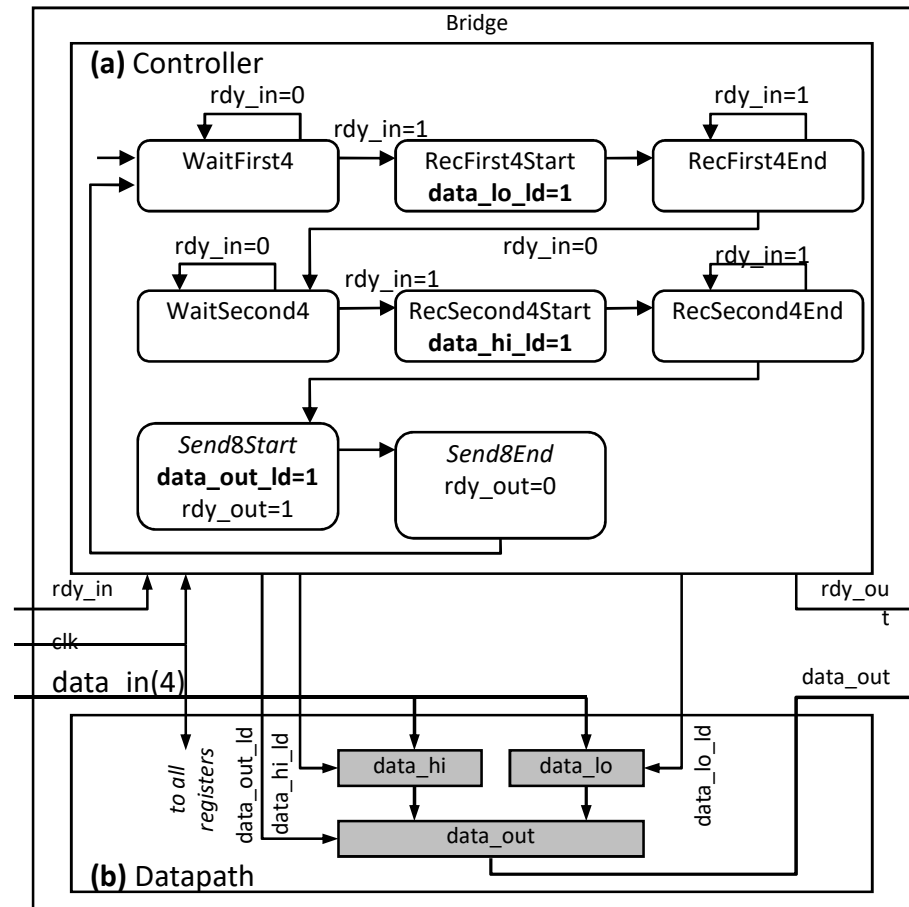
RT-level custom single-purpose processor design

- 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;  
8:     else  
9:       x = x - y;  
10:  }  
11:  d_o = x;  
12: }
```

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);  
3:   // x must be the larger number  
4:   if (x_i >= y_i) {  
5:     x=x_i;  
6:     y=y_i;  
7:   }  
8:   else {  
9:     x=y_i;  
10:    y=x_i;  
11:  }  
12:  while (y != 0) {  
13:    r = x % y;  
14:    x = y;  
15:    y = r;  
16:  }  
17:  d_o = x;  
18: }
```

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

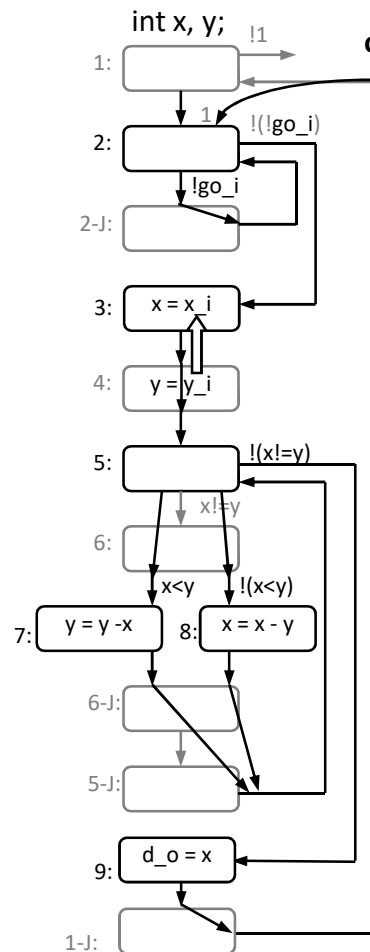


Optimizing the FSMD

- 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

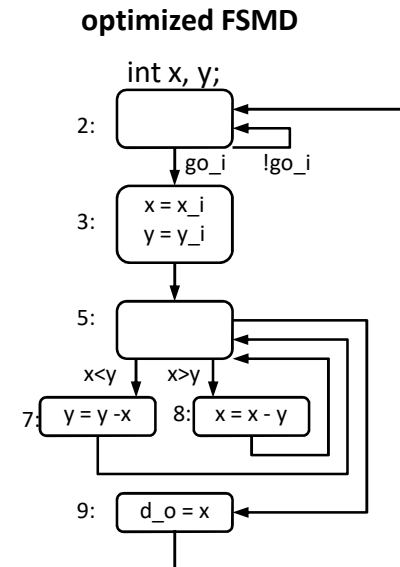
merge state 2 and state 2J – no loop operation in between them

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



Thank you...