Introduction to Embedded Systems

Edward A. Lee

UC Berkeley

EECS 149/249A

Fall 2016

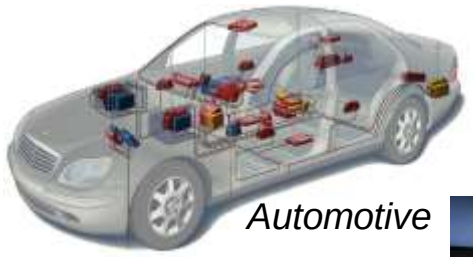
Module 1:

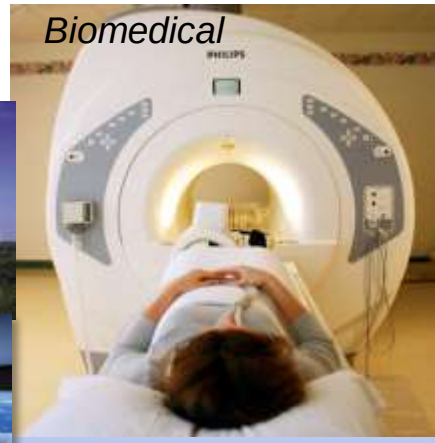
Motivation: Cyber Physical Systems

Focus on Cyber-Physical Systems Full of Contradictory Requirements

It's not just information technology anymore:

- Cyber + Physical
- Computation + Dynamics
- Security + Safety


Automotive




Manufacturing

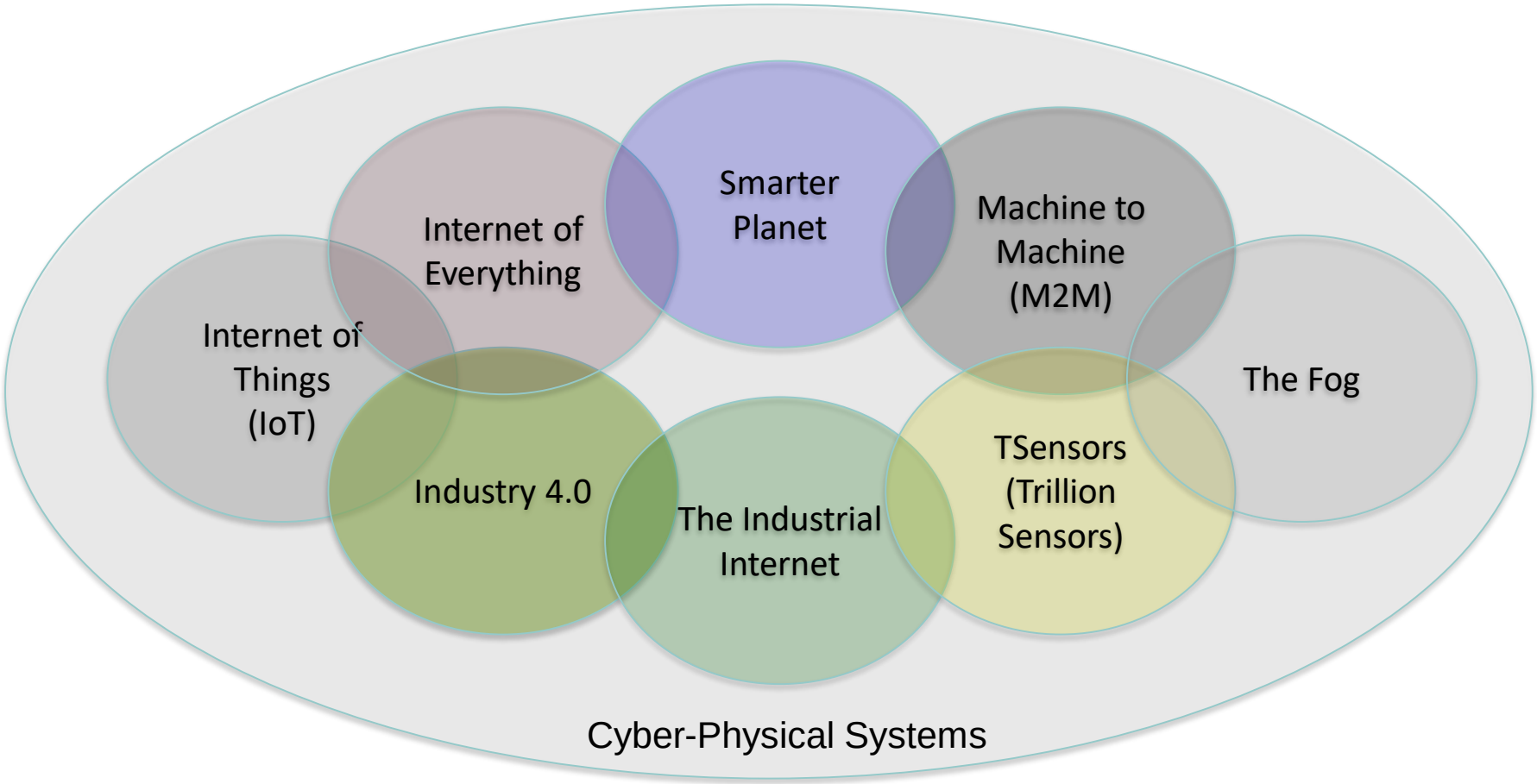
Contradictions:

- Adaptability vs. Repeatability
- High connectivity vs. Security and Privacy
- High performance vs. Low Energy
- Asynchrony vs. Coordination/Cooperation
- Scalability vs. Reliability and Predictability
- Laws and Regulations vs. Technical Possibilities
- Economies of scale (cloud) vs. Locality (fog)
- Open vs. Proprietary
- Algorithms vs. Dynamics

Innovation:

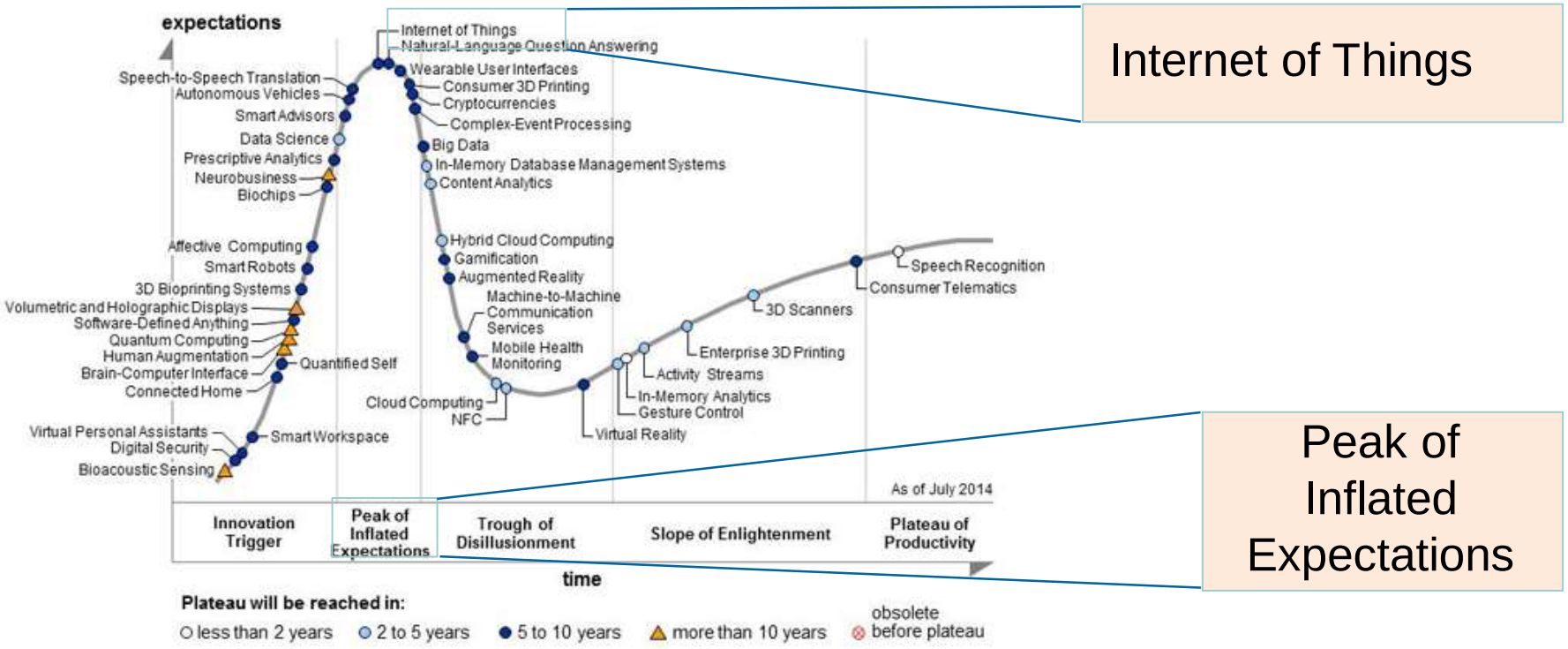
Cyber-physical systems require new engineering methods and models to address these contradictions.

E Pluribus Unum: Out of Many, One



The Hype Around The Internet of Things

Using Internet technology to connect physical devices ("things").



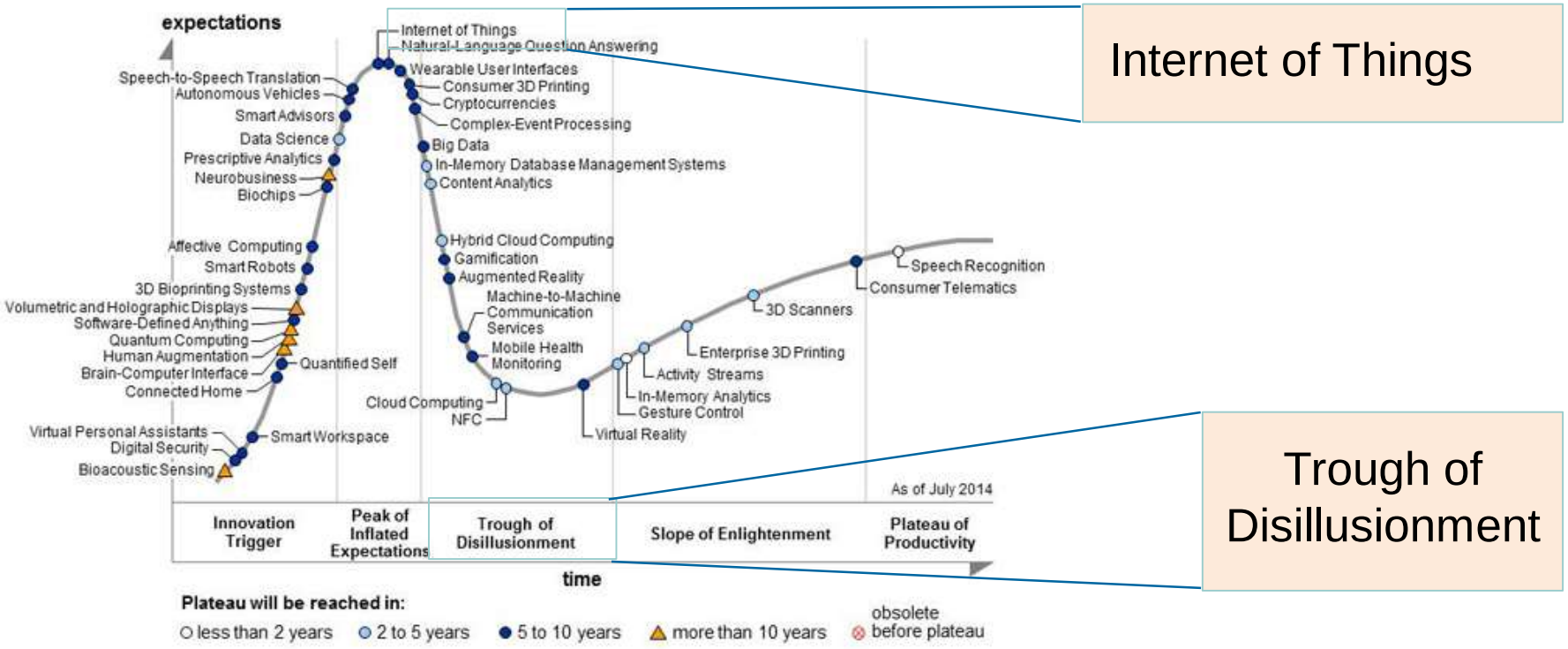
Internet of Things

Peak of Inflated Expectations

http://www.gartner.com/technology/research/hype-cycles/

The Hype Around The Internet of Things

Using Internet technology to connect physical devices ("things").



Internet of Things

Trough of Disillusionment

http://www.gartner.com/technology/research/hype-cycles/

IoT is the use of Internet technology for Cyber-Physical Systems

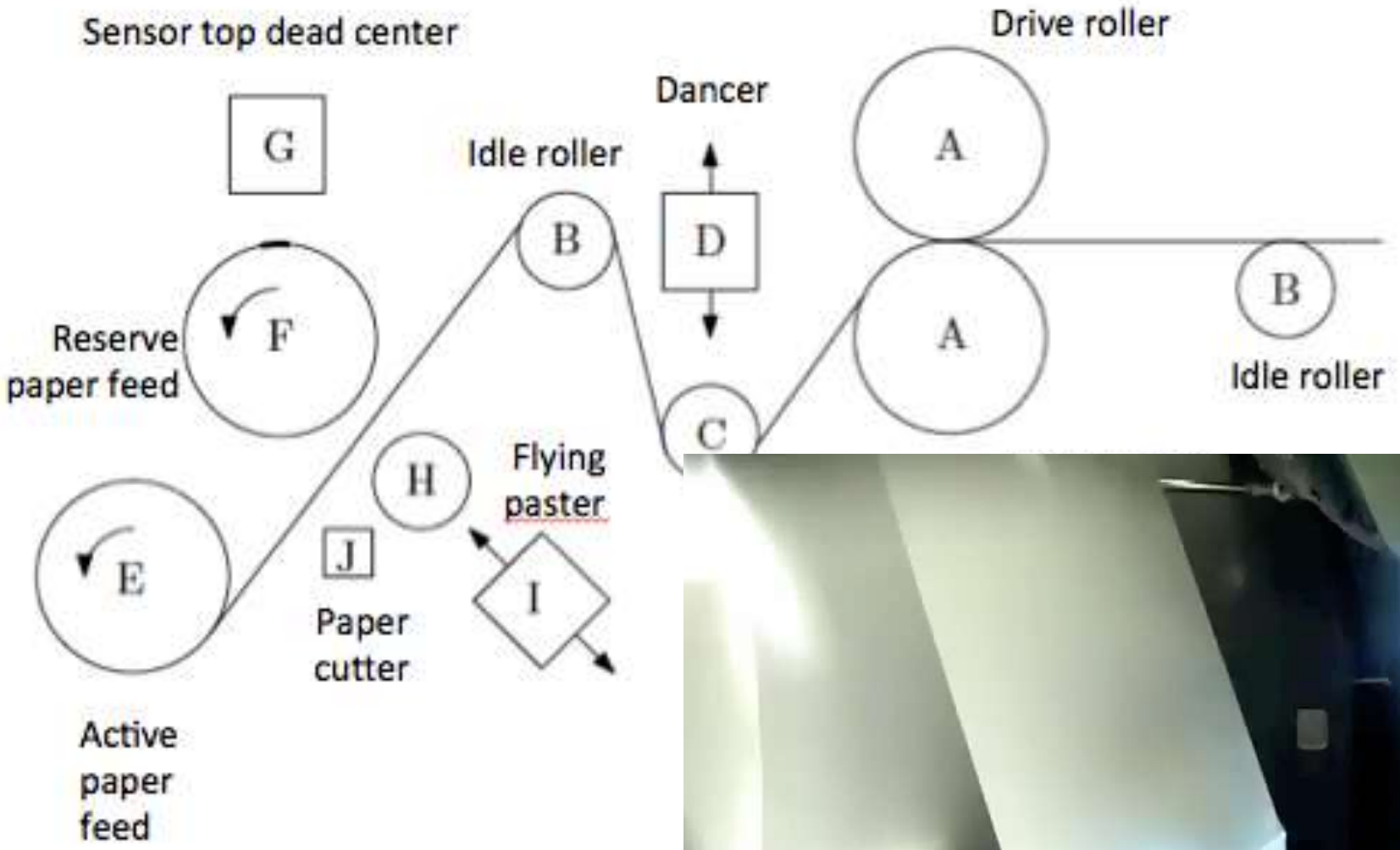
Industrial automation example from 2008: Bosch-Rexroth printing press.

The term "IoT" includes the technical solution "Internet technology" in the problem statement "connect things".

The term CPS does not.

This Bosch Rexroth printing press is a cyber-physical factory using Ethernet and TCP/IP with high-precision clock synchronization (IEEE 1588) on an isolated LAN.

Example – Flying Paster



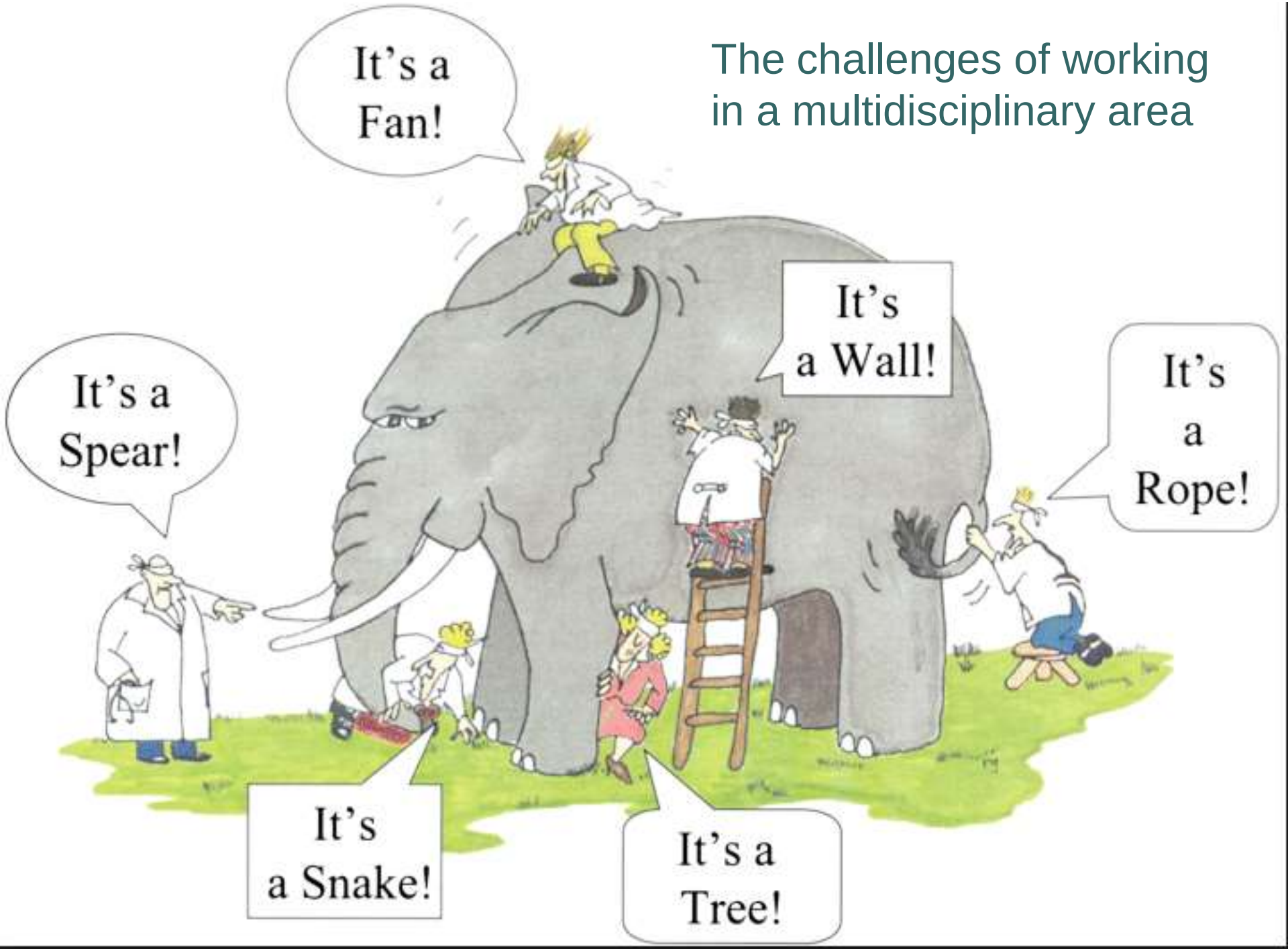
Source: http://offsetpressman.blogspot.com/2011/03/how-flying-paster-works.htr

Source: http://offsetpressman.blogspot.com/2011/03/how-flying-paster-works.html

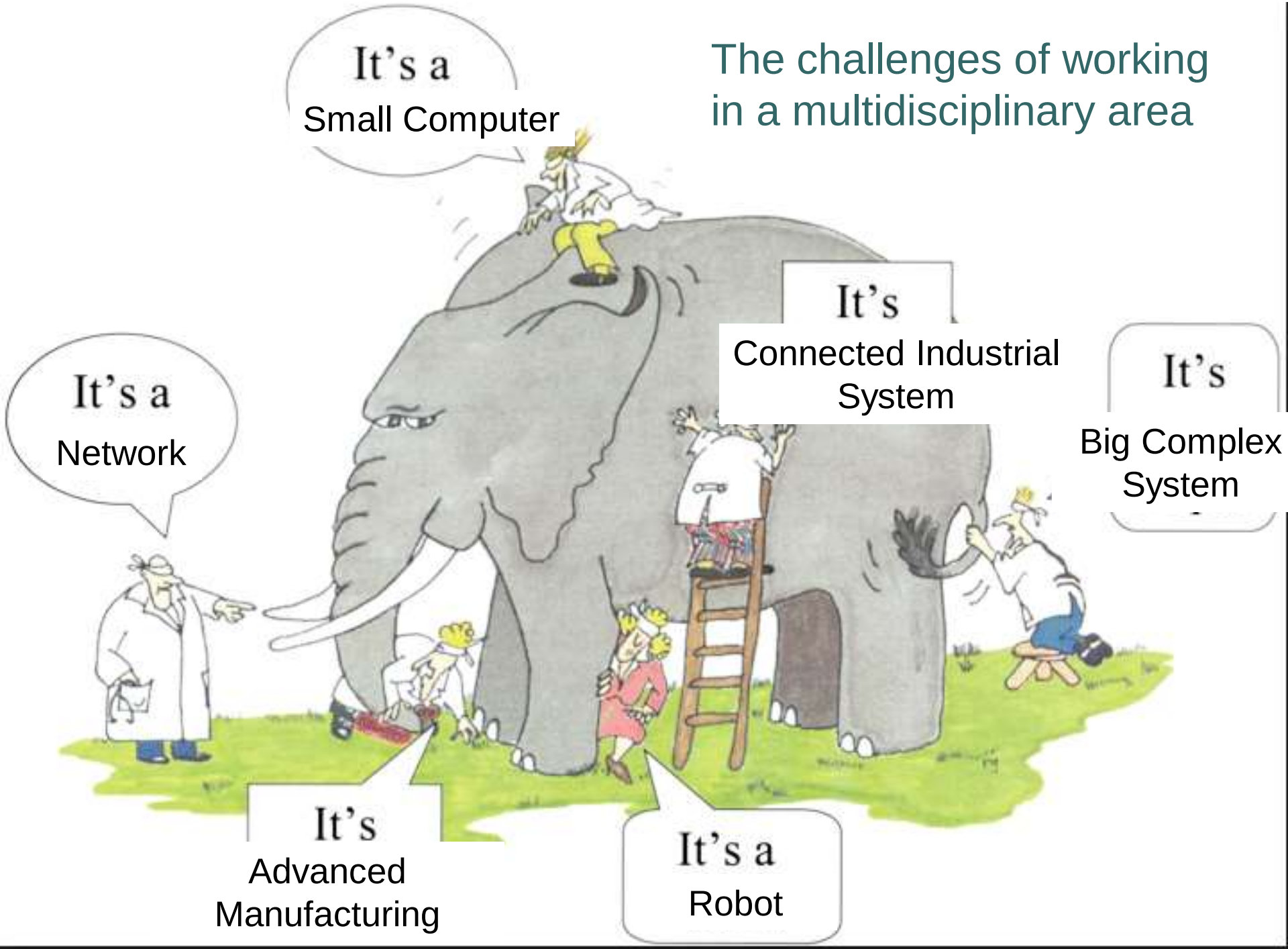
Flying Paster

CPS Challenge Problem: Prevent This

The challenges of working in a multidisciplinary area

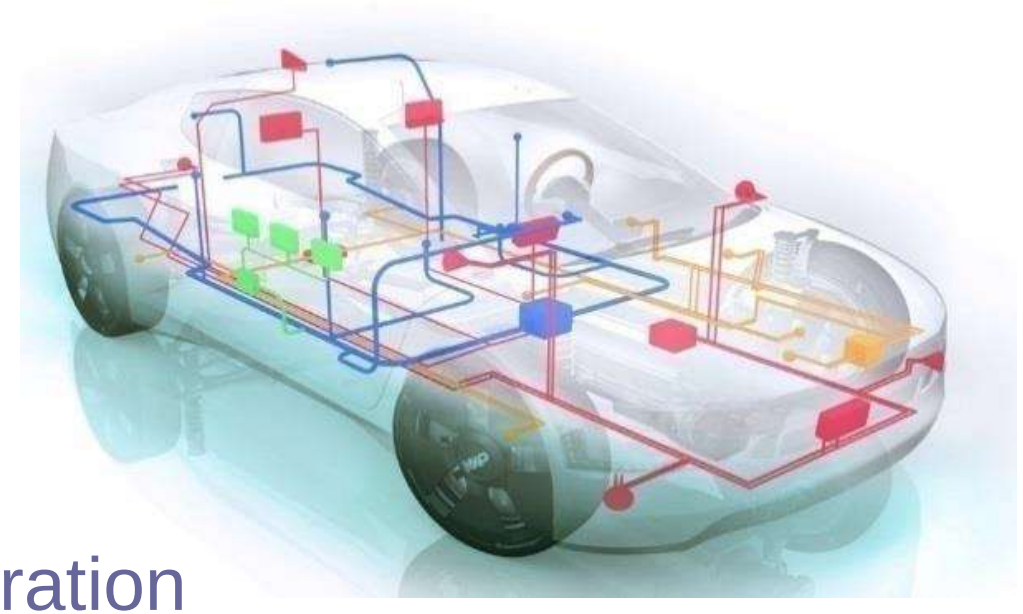


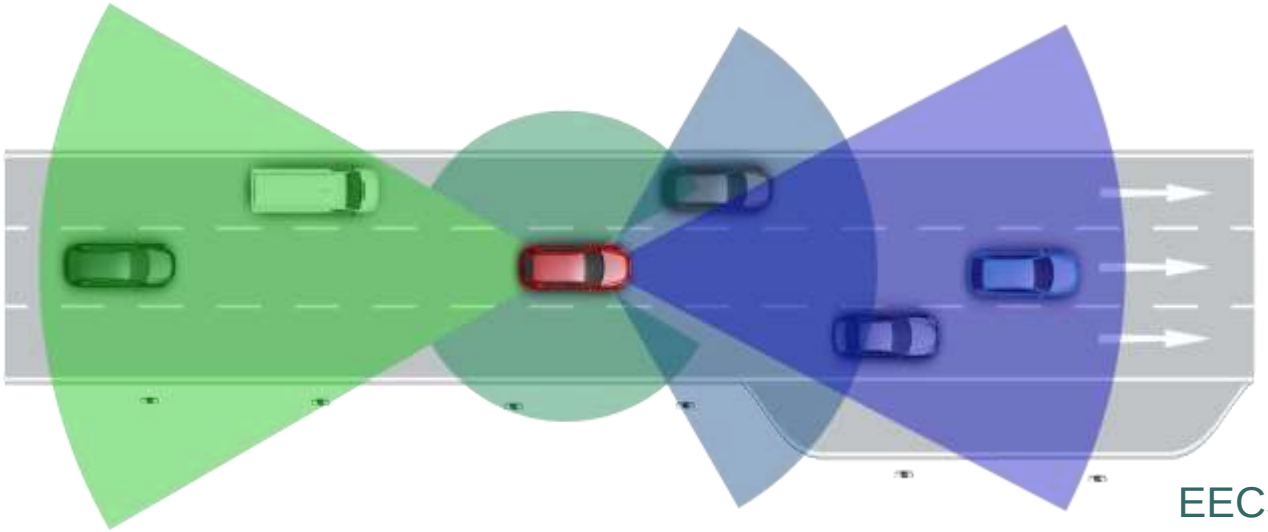
The challenges of working in a multidisciplinary area



Automotive CPS and Societal Challenges

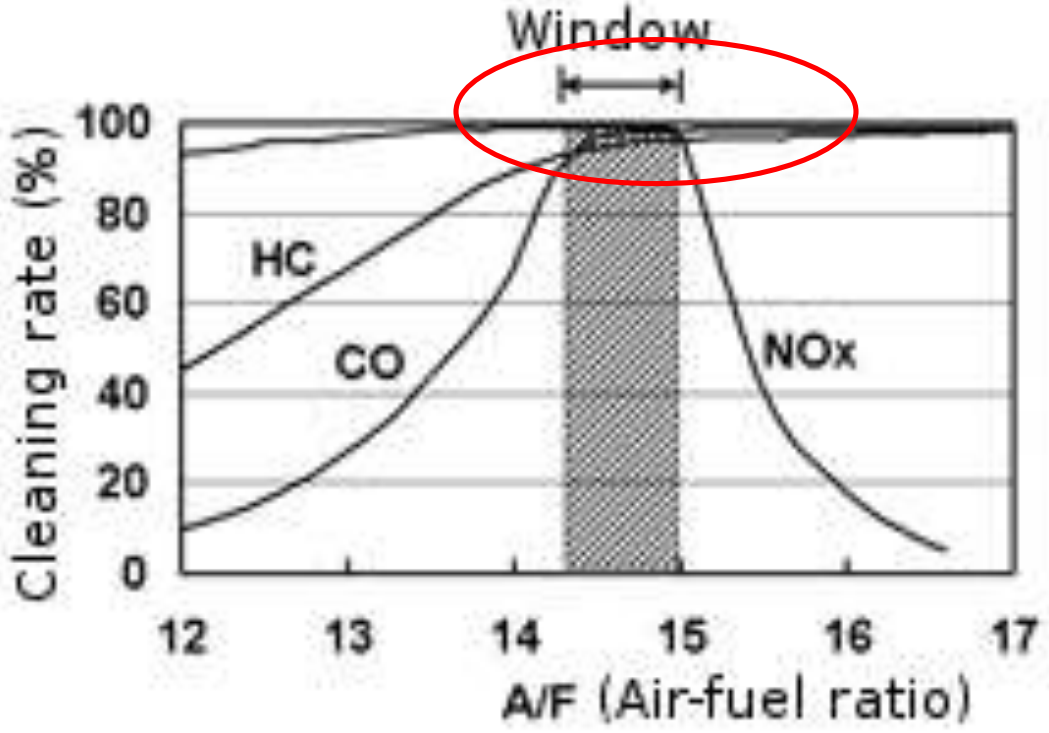
- Safer Transportation
- Reduced Emissions
- Smart Transportation
- Energy Efficiency
- Climate Change
- Human-Robot Collaboration





Example: Air-Fuel ratio control to reduce emissions

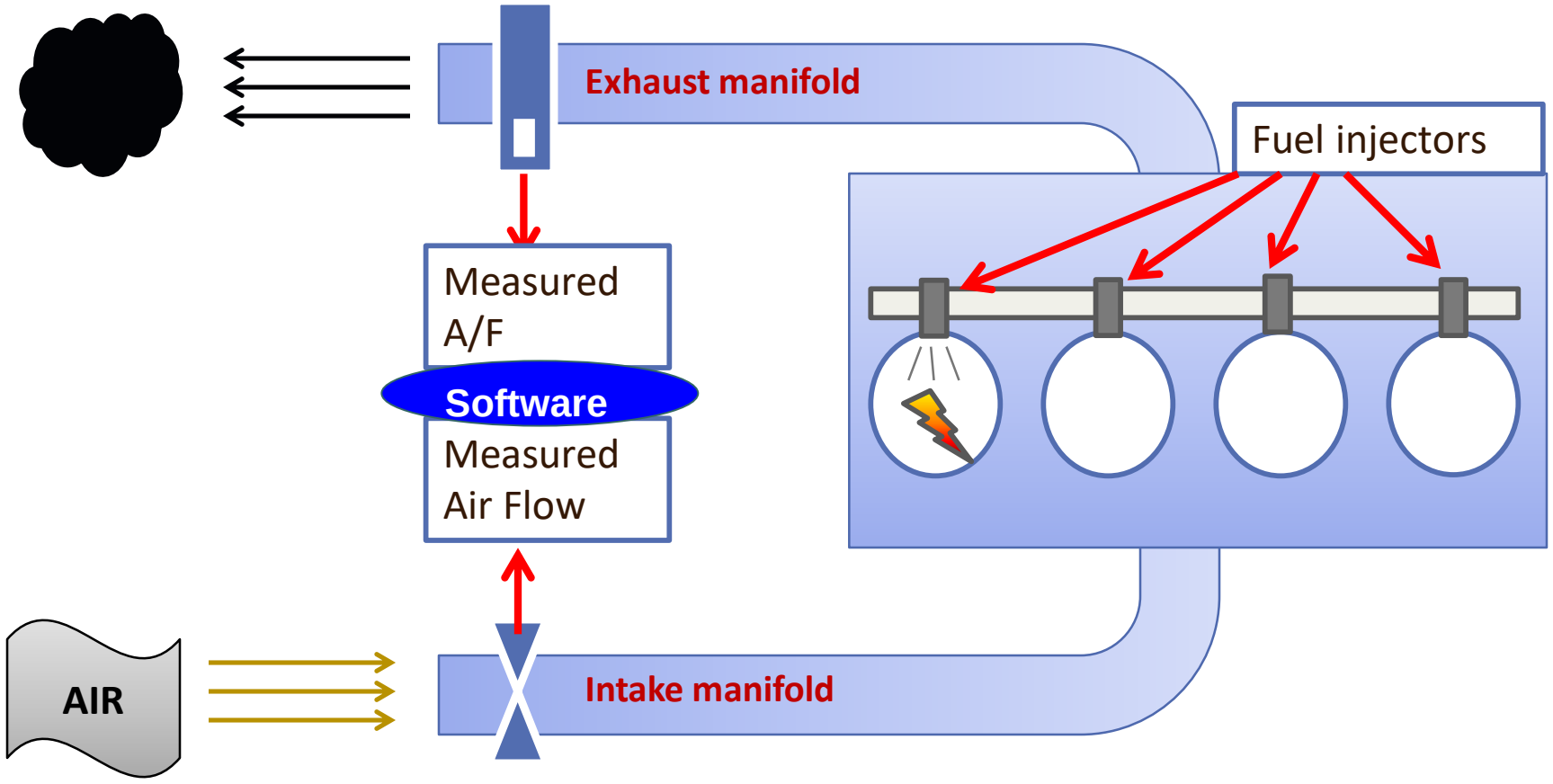
- Catalytic converters reduce CH_4, CO_2, and NO_x emissions
- Conversion efficiency optimal at stoichiometric value



See:
Jin. Kapinski. Deshmukh, Ueda, Butts,
"Powertrain Control Verification Benchmark,"
HSCC 2014

[Slide due to J. Deshmukh, Toyota]

Air-Fuel ratio control: Gasoline Engine setting



[Slide due to J. Deshmukh, Toyota]

Report: McKinsey Global Institute

Disruptive technologies:

Advances that will transform life, business, and the global economy

May 2013

… with major CPS components

Twelve potentially economically disruptive technologies

Mobile Internet	Increasingly inexpensive and capable mobile computing devices and Internet connectivity
Automation of knowledge work	Intelligent software systems that can perform knowledge work tasks involving unstructured commands and subtle judgments
The Internet of Things	Networks of low-cost sensors and actuators for data collection, monitoring, decision making, and process optimization
Cloud technology	Use of computer hardware and software resources delivered over a network or the Internet, often as a service
Advanced robotics	Increasingly capable robots with enhanced senses, dexterity, and intelligence used to automate tasks or augment humans
Autonomous and near-autonomous vehicles	Vehicles that can navigate and operate with reduced or no human intervention
Next-generation genomics	Fast, low-cost gene sequencing, advanced big data analytics, and synthetic biology ("writing" DNA)
Energy storage	Devices or systems that store energy for later use, including batteries
3D printing	Additive manufacturing techniques to create objects by printing layers of material based on digital models
Advanced materials	Materials designed to have superior characteristics (e.g., strength, weight, conductivity) or functionality
Advanced oil and gas exploration and recovery	Exploration and recovery techniques that make extraction of unconventional oil and gas economical
Renewable energy	Generation of electricity from renewable sources with reduced harmful climate impact

Economic Potential

	**The Internet of Things**	**300%** Increase in connected machine-to-machine devices over past 5 years **80–90%** Price decline in MEMS (microelectromechanical systems) sensors in past 5 years	**1 trillion** Things that could be connected to the Internet across industries such as manufacturing, health care, and mining **100 million** Global machine to machine (M2M) device connections across sectors like transportation, security, health care, and utilities	**$36 trillion** Operating costs of key affected industries (manufacturing, health care, and mining)
	**Cloud technology**	**18 months** Time to double server performance per dollar **3x** Monthly cost of owning a server vs. renting in the cloud	**2 billion** Global users of cloud-based email services like Gmail, Yahoo, and Hotmail **80%** North American institutions hosting or planning to host critical applications on the cloud	**$1.7 trillion** GDP related to the Internet **$3 trillion** Enterprise IT spend
	**Advanced robotics**	**75–85%** Lower price for Baxter[3] than a typical industrial robot **170%** Growth in sales of industrial robots, 2009–11	**320 million** Manufacturing workers, 12% of global workforce **250 million** Annual major surgeries	**$6 trillion** Manufacturing worker employment costs, 19% of global employment costs **$2–3 trillion** Cost of major surgeries
	**Autonomous and near-autonomous vehicles**	**7** Miles driven by top-performing driverless car in 2004 DARPA Grand Challenge along a 150-mile route **1,540** Miles cumulatively driven by cars competing in 2005 Grand Challenge **300,000+** Miles driven by Google's autonomous cars with only 1 accident (which was human-caused)	**1 billion** Cars and trucks globally **450,000** Civilian, military, and general aviation aircraft in the world	**$4 trillion** Automobile industry revenue **$155 billion** Revenue from sales of civilian, military, and general aviation aircraft

Google Strategy

CNET › Internet › Google closes $3.2 billion purchase of Nest

Google closes $3.2 billion purchase of Nest

The acquisition brings with it the Learning Thermostat and the Protect smoke and CO detector as Google looks to make its mark in the smart home.

by Lance Whitney @lancewhit / February 12, 2014 5:00 AM PST / Updated: February 12, 2014 5:19 AM PST

theguardian | TheObserver

Search

Google's drive into robotics should concern us all

The company's expansion into robotics was developed in tandem with the US military. Where will its power play stop?

John Naughton
The Observer, Sunday 29 December 2013

Google's robotic cars have about $150,000 in equipment including a $70,000 LIDAR (laser radar) system. The range finder mounted on the top is a Velodyne 64-beam laser. This laser allows the vehicle to generate a detailed 3D map of its environment. The car then takes these generated maps and combines them with high-resolution maps of the world, producing different types of data models that allow it to drive itself.

Google and Facebook

Artist's rendering of Titan's Solara 50, which in theory at least, can stay aloft for years.

Wall Street Journal:

By Alistair Barr and Reed Albergotti
April 14, 2014

Google Inc. on Monday acquired a maker of solar-powered drones—a startup that Facebook Inc. had also considered acquiring—as the technology giants battle to extend their influence and find new users in the far corners of the earth.

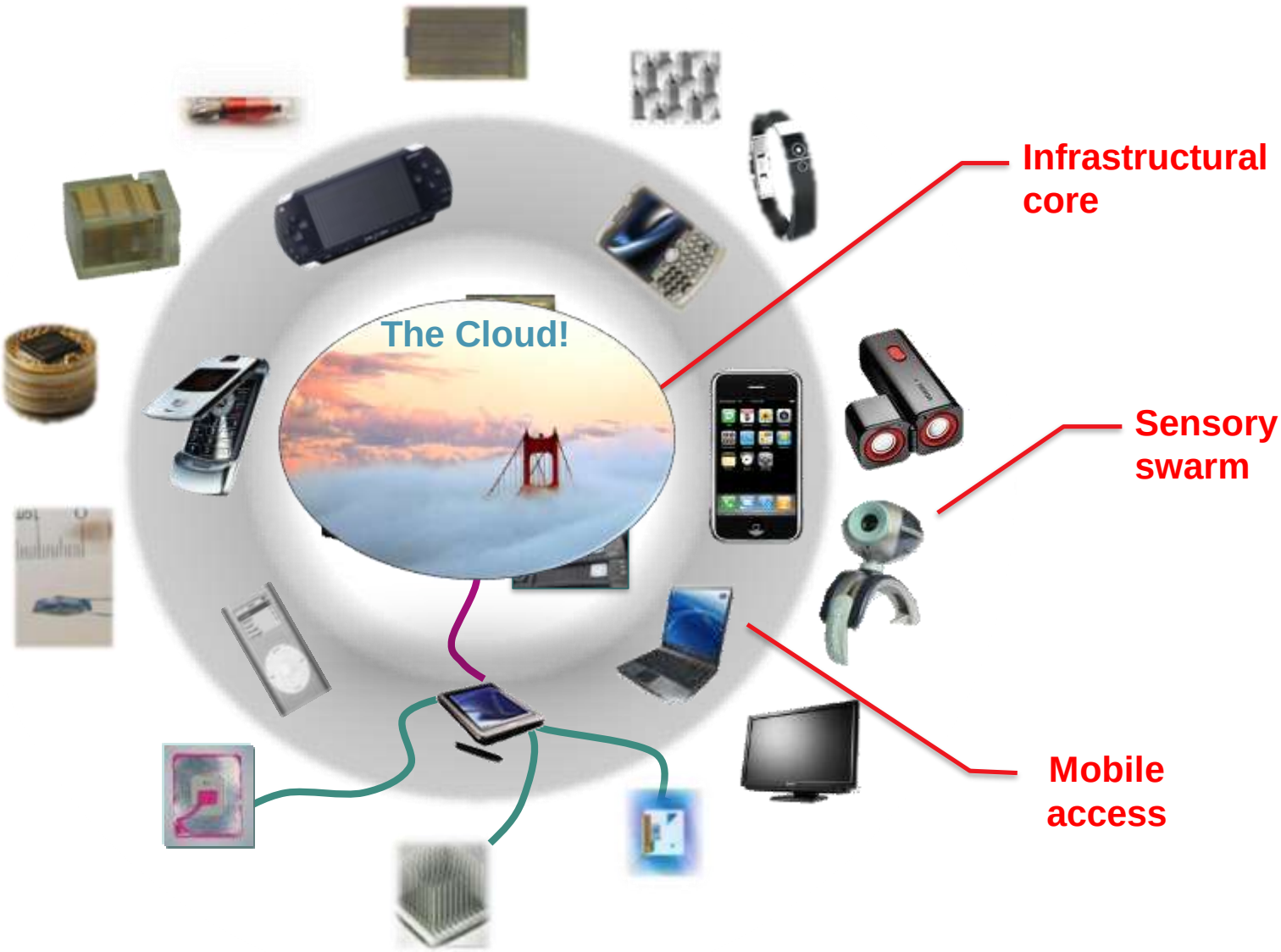
Tesla Gigafactory

Artists conception of battery factory under construction in Nevada.
From: https://www.tesla.com/gigafactory

Apple iCar?

Macworld, Aug. 10, 2016:
Reports suggest that Apple is developing an electric iCar to rival Tesla. With reports that Apple is negotiating with BMW, and poaching Samsung employees (especially battery specialists) and reassigning large numbers of staff for its Project Titan, is Apple manufacturing an iCar, and when will the iCar be launched?

The Emerging IT Scene



Courtesy: J. Rabaey

What this course is about

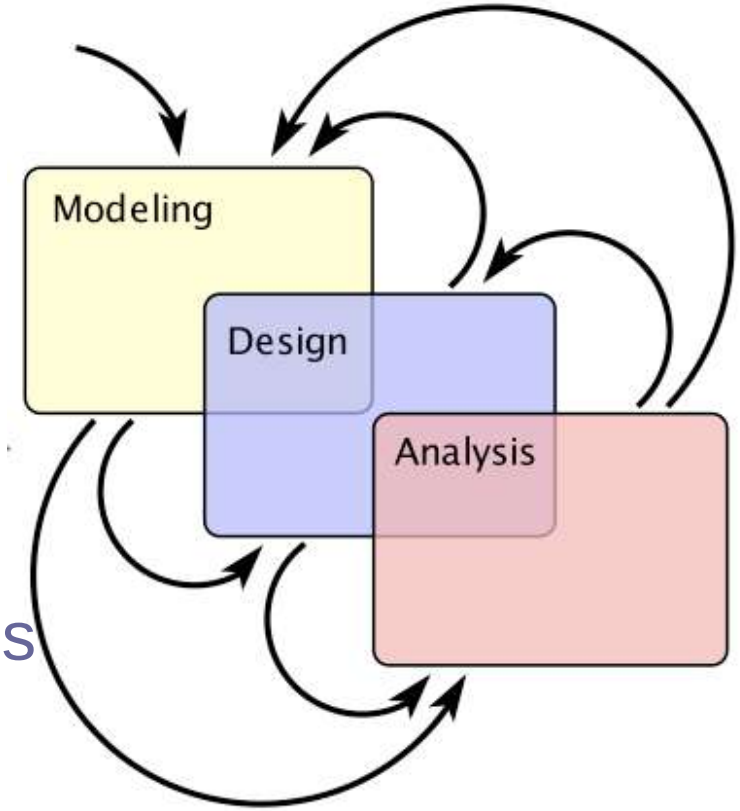
A principled, scientific approach to designing and implementing embedded systems

Not just hacking!!

Hacking can be fun, but it can also be very painful when things go wrong…

Focus on *model-based system design*, and on *embedded software*

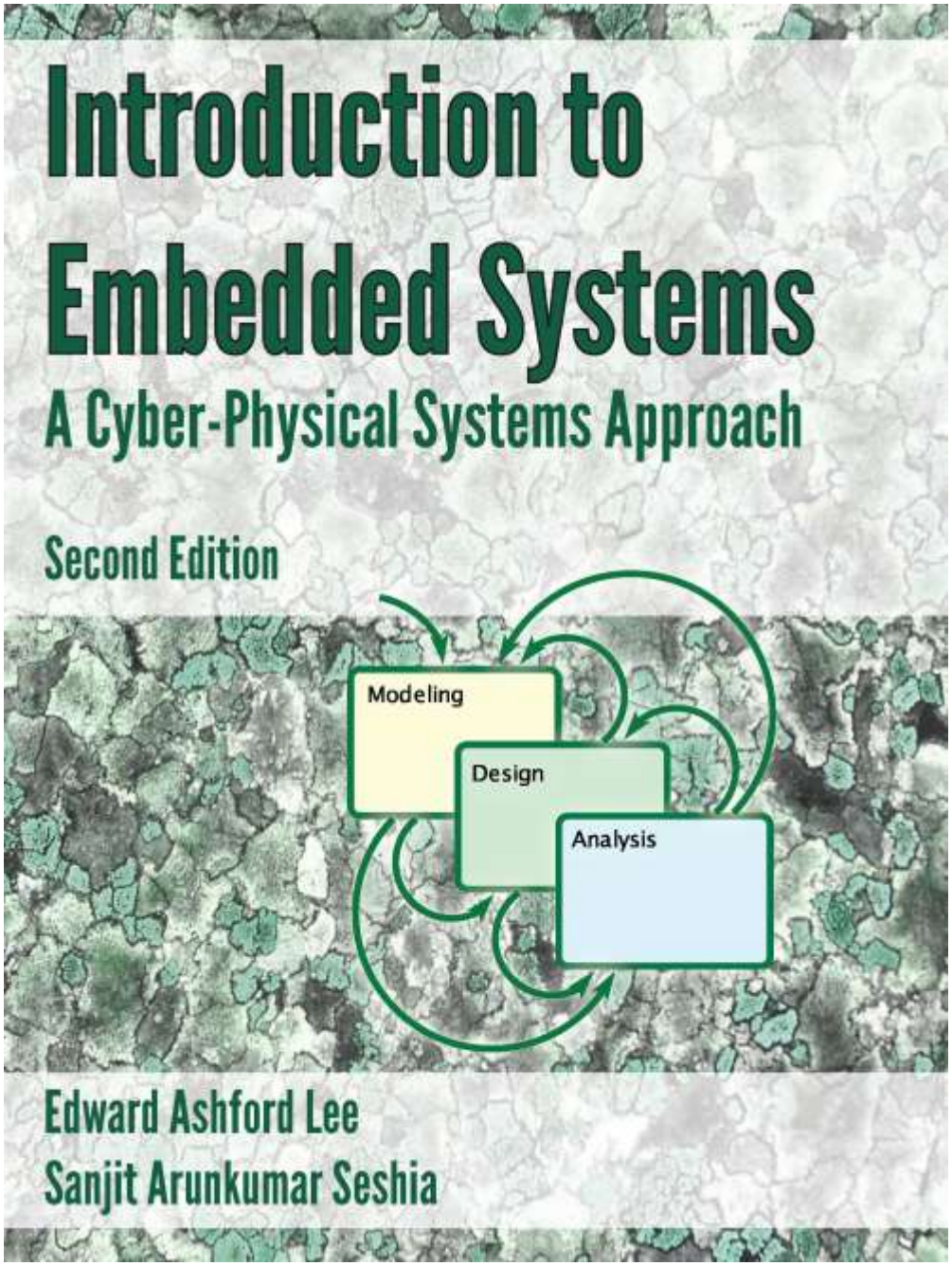
Modeling, Design, Analysis



Modeling is the process of gaining a deeper understanding of a system through imitation. Models express **what** a system does or should do.

Design is the structured creation of artifacts. It specifies **how** a system does what it does.

Analysis is the process of gaining a deeper understanding of a system through dissection. It specifies **why** a system does what it does (or fails to do what a model says it should do).



Your textbook, written for this course, strives to identify and introduce the *durable intellectual ideas* of embedded systems as a technology and as a subject of study. The emphasis is on modeling, design, and analysis of cyber-physical systems, which integrate computing, networking, and physical processes.

http://LeeSeshia.org

Motivating Example of a Cyber-Physical System
(see Chapter 1 in book)

STARMAC quadrotor aircraft (Tomlin, et al.)

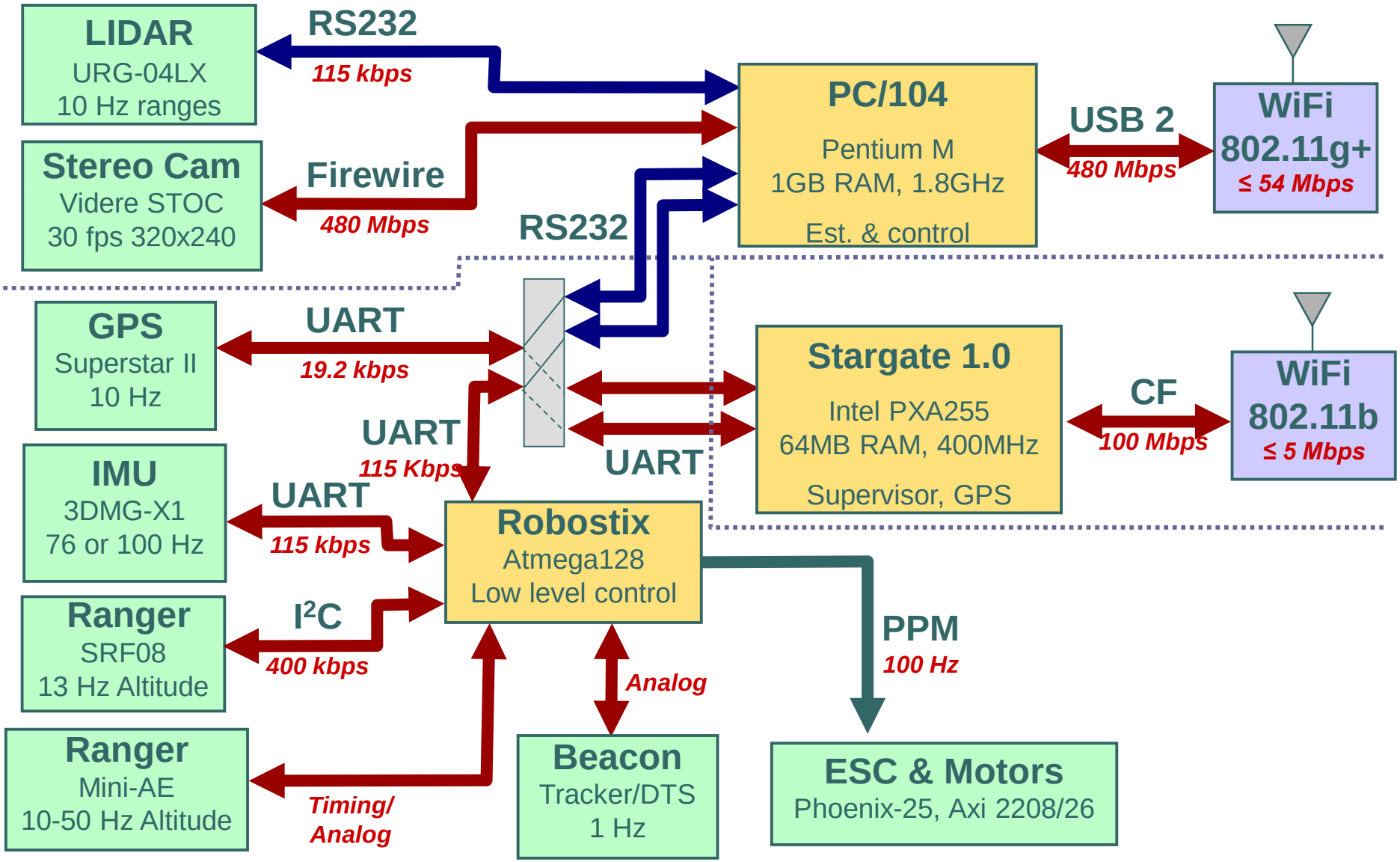
- Introductory Video:
http://www.youtube.com/watch?v=rJ9r2orcaYo
- Back-Flip Manuever:
http://www.youtube.com/watch?v=iD3QgGpzzIM

Modeling:
- Flight dynamics (ch2)
- Modes of operation (ch3)
- Transitions between modes (ch4)
- Composition of behaviors (ch5)
- Multi-vehicle interaction (ch6)

Design:
- Sensors and Actuators (ch7)
- Processors (ch8)
- Memory system (ch9)
- Sensor interfacing (ch10)
- Concurrent software (ch11)
- Real-time scheduling (ch12)

Analysis
- Specifying safe behavior (ch13)
- Achieving safe behavior (ch14)
- Verifying safe behavior (ch15)
- Guaranteeing timeliness (ch16)
- Security and privacy (ch17)

STARMAC Design Block Diagram



A Theme in This Course: Think Critically

Any course that purports to teach you how to design embedded systems is misleading you.

The technology will change!

Our goal is to teach you how things are done today, and why that is not good enough. So you will not be surprised by the changes that *are* coming.

Introduction to Embedded Systems

Edward A. Lee

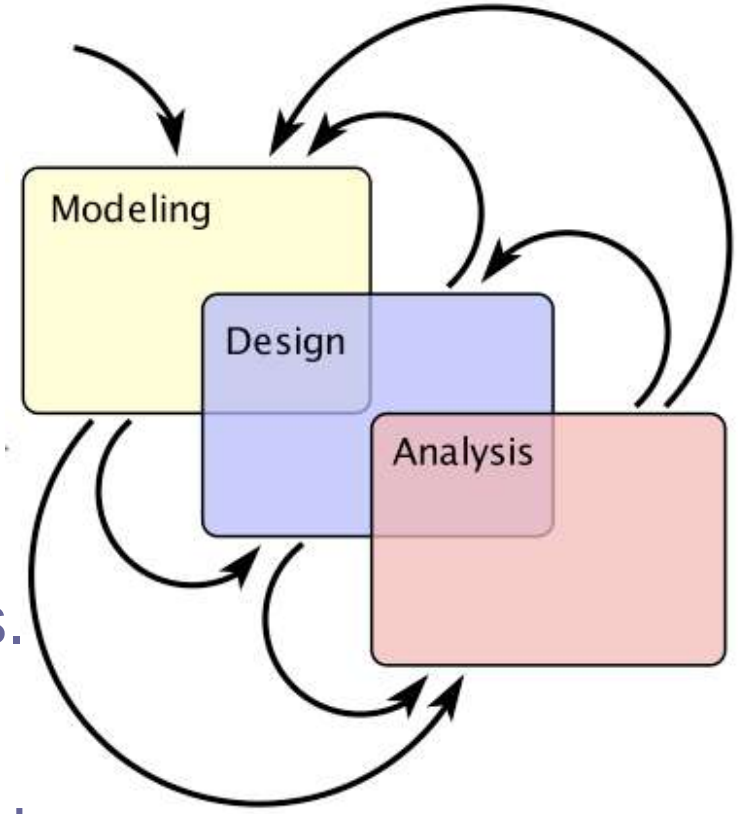
UC Berkeley

EECS 149/249A

Fall 2016

Module *2*: Model Based Design

Modeling, Design, Analysis: An Iterative Process

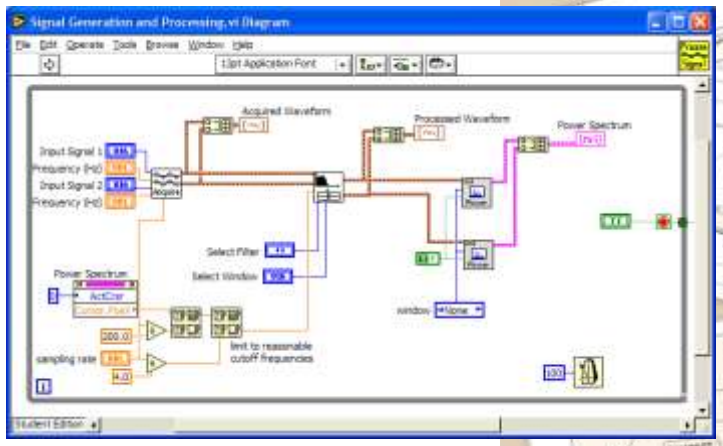


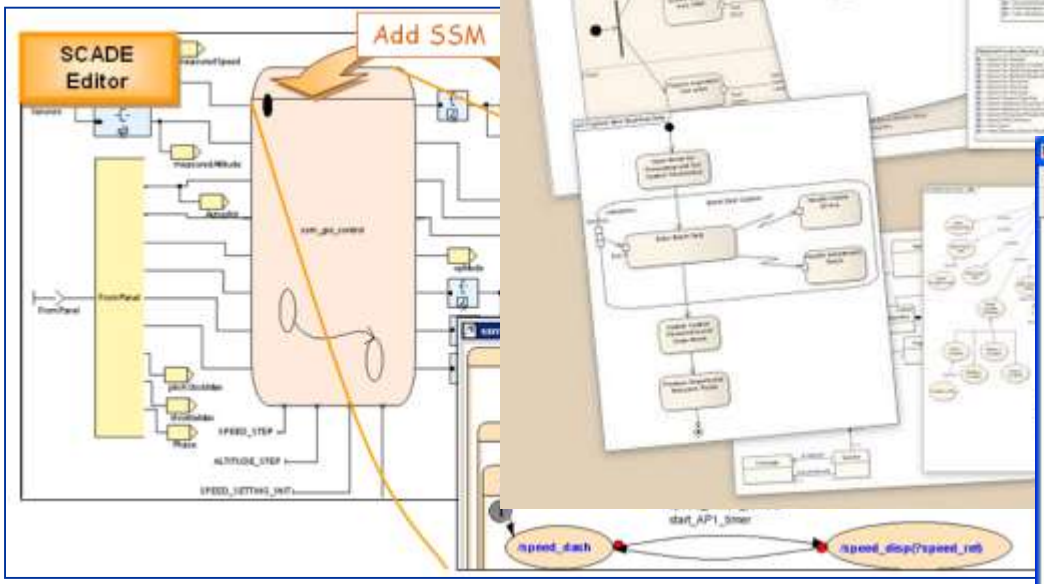
Modeling is the process of gaining a deeper understanding of a system through imitation. Models specify **what** a system does.

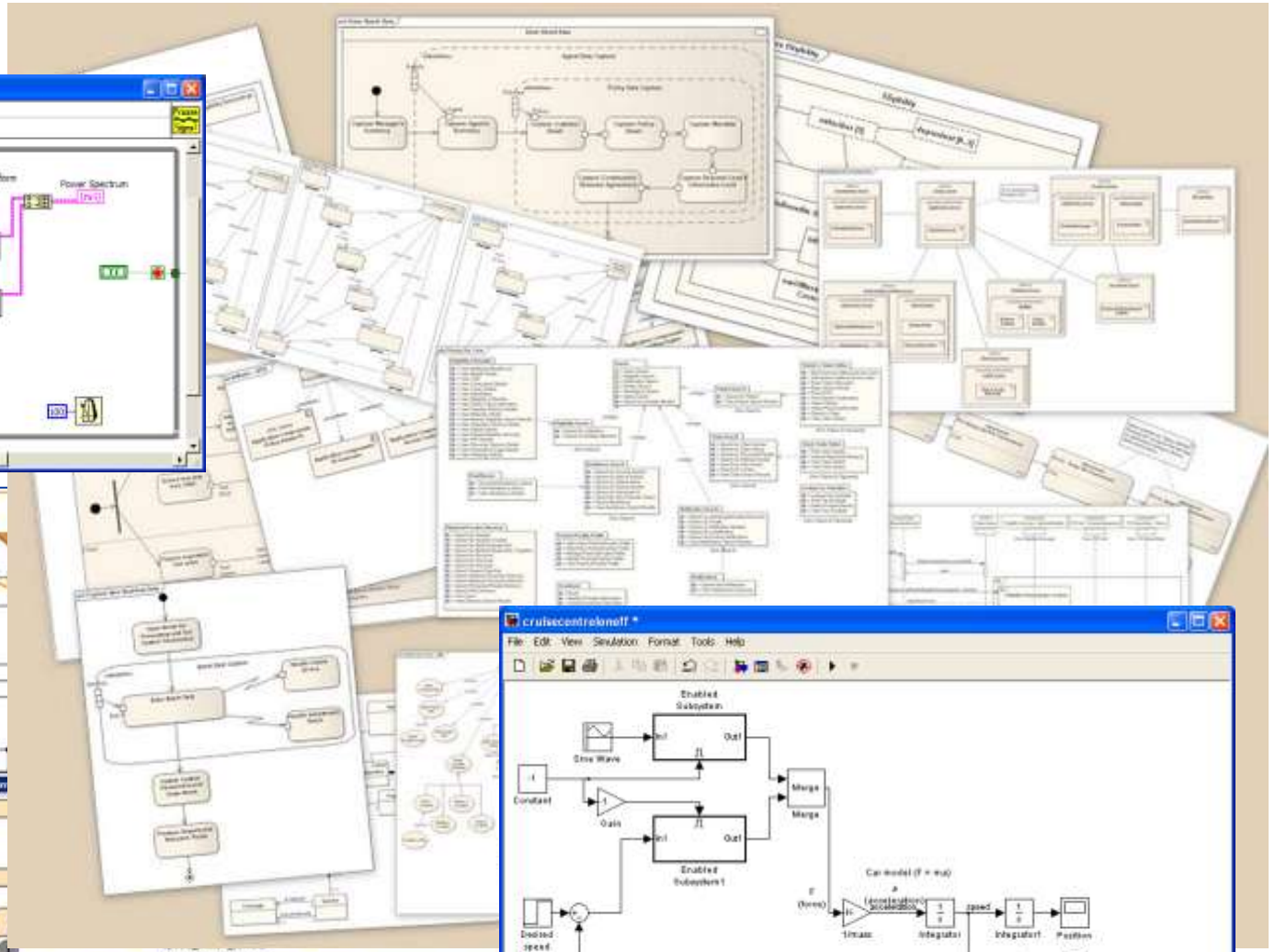
Design is the structured creation of artifacts. It specifies **how** a system does what it does. This includes optimization.

Analysis is the process of gaining a deeper understanding of a system through dissection. It specifies **why** a system does what it does (or fails to do what a model says it should do).

Focus on Models





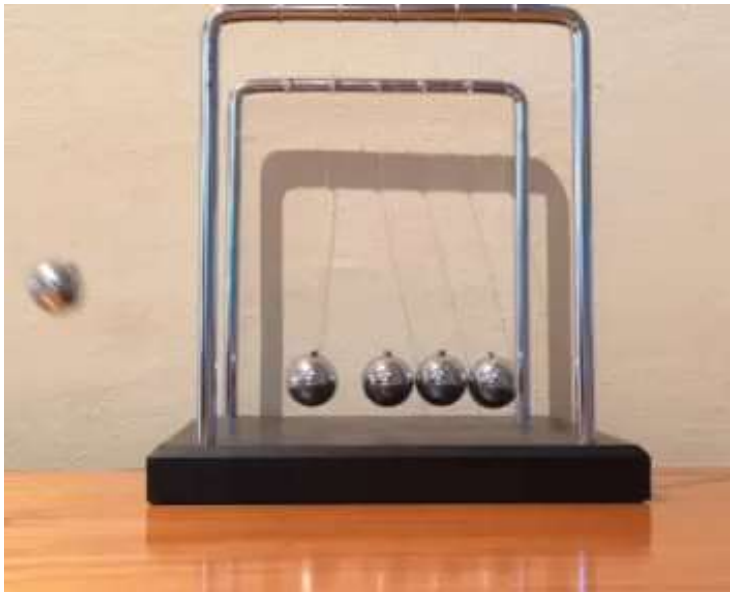


Models vs. Reality

$$x(t) = x(0) + \int_0^t v(\tau)d\tau$$

$$v(t) = v(0) + \frac{1}{m}\int_0^t F(\tau)d\tau$$

The model



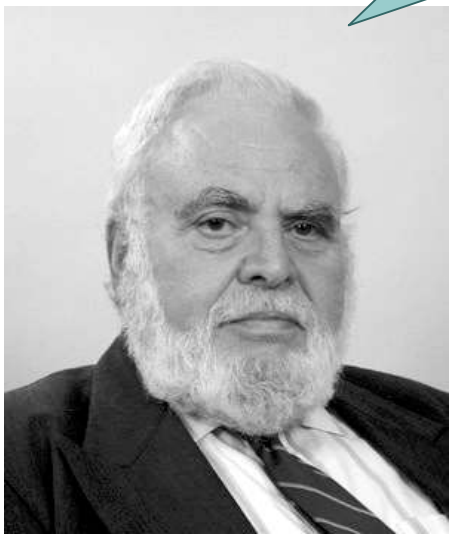
The target (the thing being modeled).

In this example, the *modeling framework* is calculus and Newton's laws.

Fidelity is how well the model and its target match

Engineers often confuse the model with its target

You will never strike oil by drilling through the map!



Solomon Wolf Golomb

But this does not in any way diminish the value of a map!

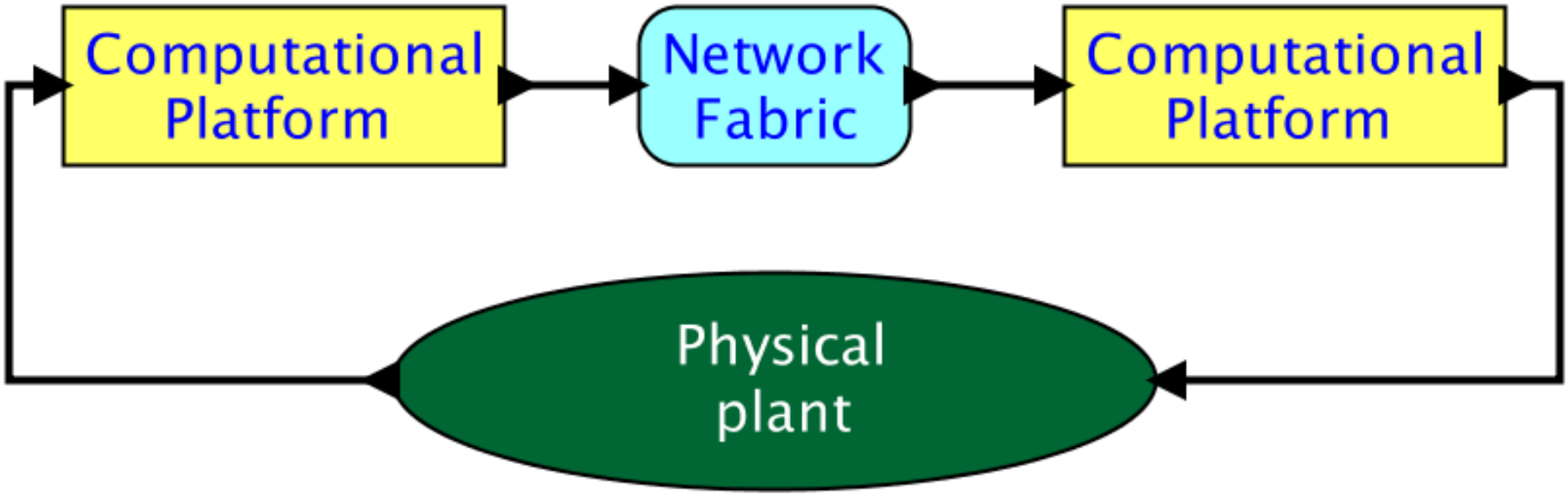
Determinacy

Some of the most valuable models are *deterministic*.

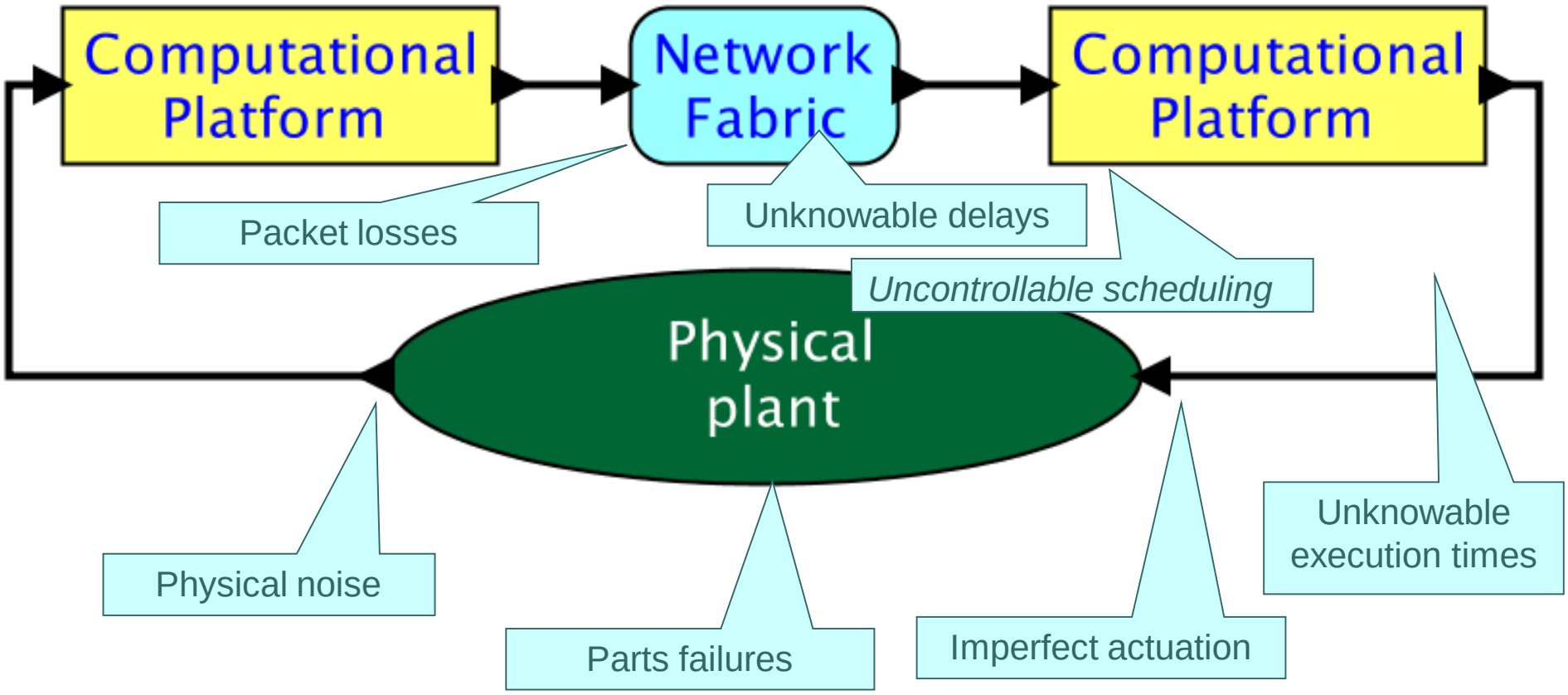
A model is *deterministic* if, given the initial state and the inputs, the model defines exactly one behavior.

Deterministic models have proven extremely valuable in the past.

Schematic of a simple CPS



Do deterministic models make sense for Cyber-physical systems?

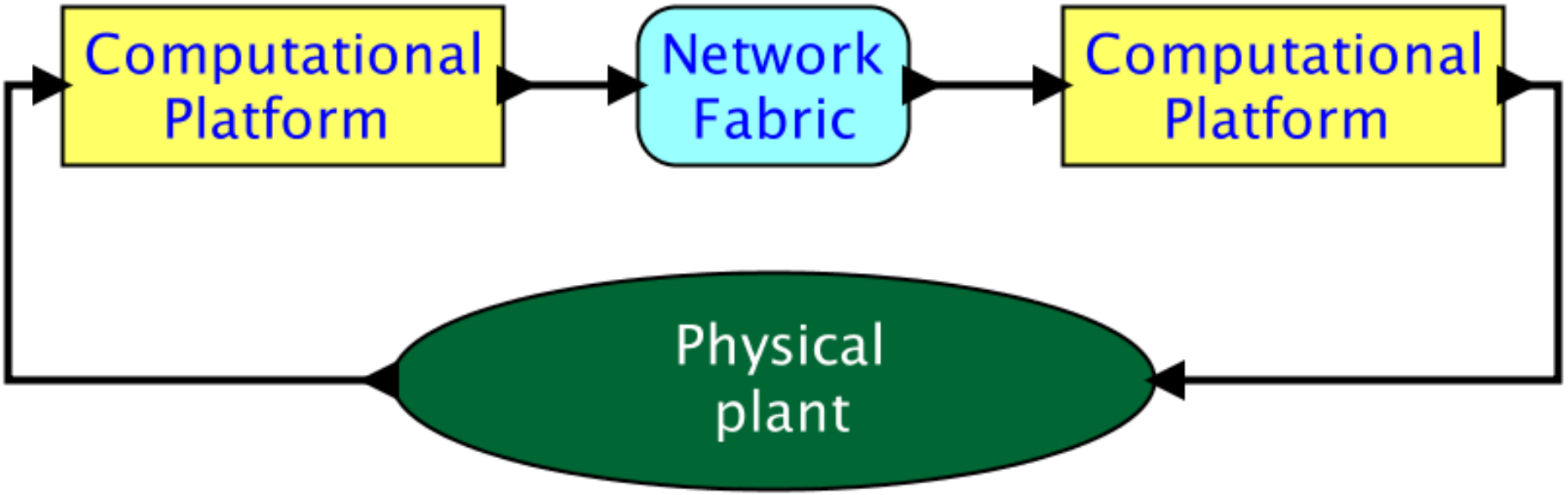


A Model Need not be *True* to be *Useful*

"Essentially, all models are wrong, but some are useful."

Box, G. E. P. and N. R. Draper, 1987: *Empirical Model-Building and Response Surfaces*. Wiley Series in Probability and Statistics, Wiley.

What kinds of models should we use?



Let's look at the most successful kinds of models from the cyber and the physical worlds.

Software is a Model

Physical System

Model

```
/** Reset the output receivers, which are the inside receivers of
 *  the output ports of the container.
 *  @exception IllegalActionException If getting the receivers fails.
 */
private void _resetOutputReceivers() throws IllegalActionException {
    List<IOPort> outputs = ((Actor) getContainer()).outputPortList();
    for (IOPort output : outputs) {
        if (_debugging) {
            _debug("Resetting inside receivers of output port: "
                    + output.getName());
        }
        Receiver[][] receivers = output.getInsideReceivers();
        if (receivers != null) {
            for (int i = 0; i < receivers.length; i++) {
                if (receivers[i] != null) {
                    for (int j = 0; j < receivers[i].length; j++) {
                        if (receivers[i][j] instanceof FSMReceiver) {
                            receivers[i][j].reset();
                        }
                    }
                }
            }
        }
    }
}
```

Single-threaded imperative programs are deterministic models

Consider single-threaded imperative programs

```
1  void foo(int32_t x) {
2       if (x > 1000) {
3            x = 1000;
4       }
5       if (x > 0) {
6            x = x + 1000;
7            if (x < 0) {
8                 panic();
9            }
10      }
11 }
```

This program defines exactly one behavior, given the input x.

Note that the modeling framework (the C language, in this case) defines "behavior" and "input."

The target of the model is nondeterministic (electrons sloshing around in silicon).

Software relies on another deterministic model that abstracts the hardware

Physical System

Model

Image: Wikimedia Commons

Integer Register-Register Operations

RISC-V defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct* field selects the type of operation.

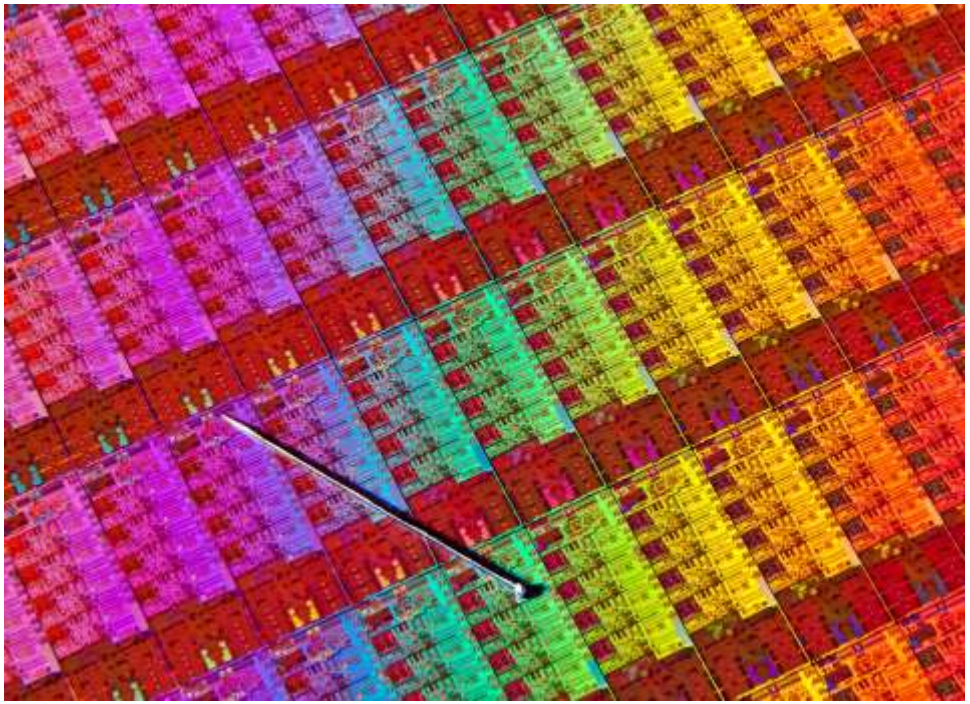
31 27	26 22	21 17	16 7	6 0
rd	rs1	rs2	funct10	opcode
5	5	5	10	7
dest	src1	src2	ADD/SUB/SLT/SLTU	OP
dest	src1	src2	AND/OR/XOR	OP
dest	src1	src2	SLL/SRL/SRA	OP
dest	src1	src2	ADDW/SUBW	OP-32
dest	src1	src2	SLLW/SRLW/SRAW	OP-32

Waterman, et al., The RISC-V Instruction Set Manual, UCB/EECS-2011-62, 2011

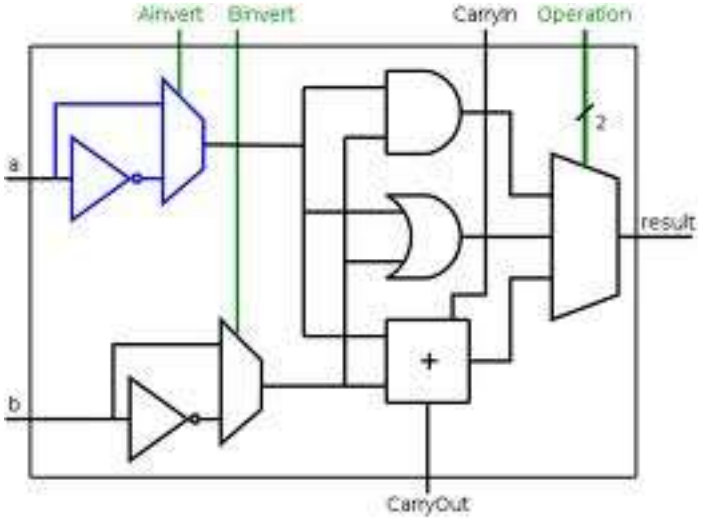
Instruction Set Architectures (ISAs) are deterministic models

… which relies on yet another deterministic model

Physical System



Model



Synchronous digital logic is a deterministic model

Deterministic Models for the Physical Side of CPS

Physical System

Image: Wikimedia Commons

Model

$$\dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M}\int_0^t \mathbf{F}(\tau)d\tau$$

Differential Equations are deterministic models

A Major Problem for CPS: Combinations of Deterministic Models are Nondeterministic

```
void initTimer(void) {
    SysTickPeriodSet(SysCtlClockGet() / 1000);
    SysTickEnable();
    SysTickIntEnable();
}
volatile uint timer_count = 0;
void ISR(void) {
    if(timer_count != 0) {
        timer_count--;
    }
}
int main(void) {
    SysTickIntRegister(&ISR);
    .. // other init
    timer_count = 2000;
    initTimer();
    while(timer_count != 0) {
        ... code to run for 2 seconds
    }
    ... // other code
}
```

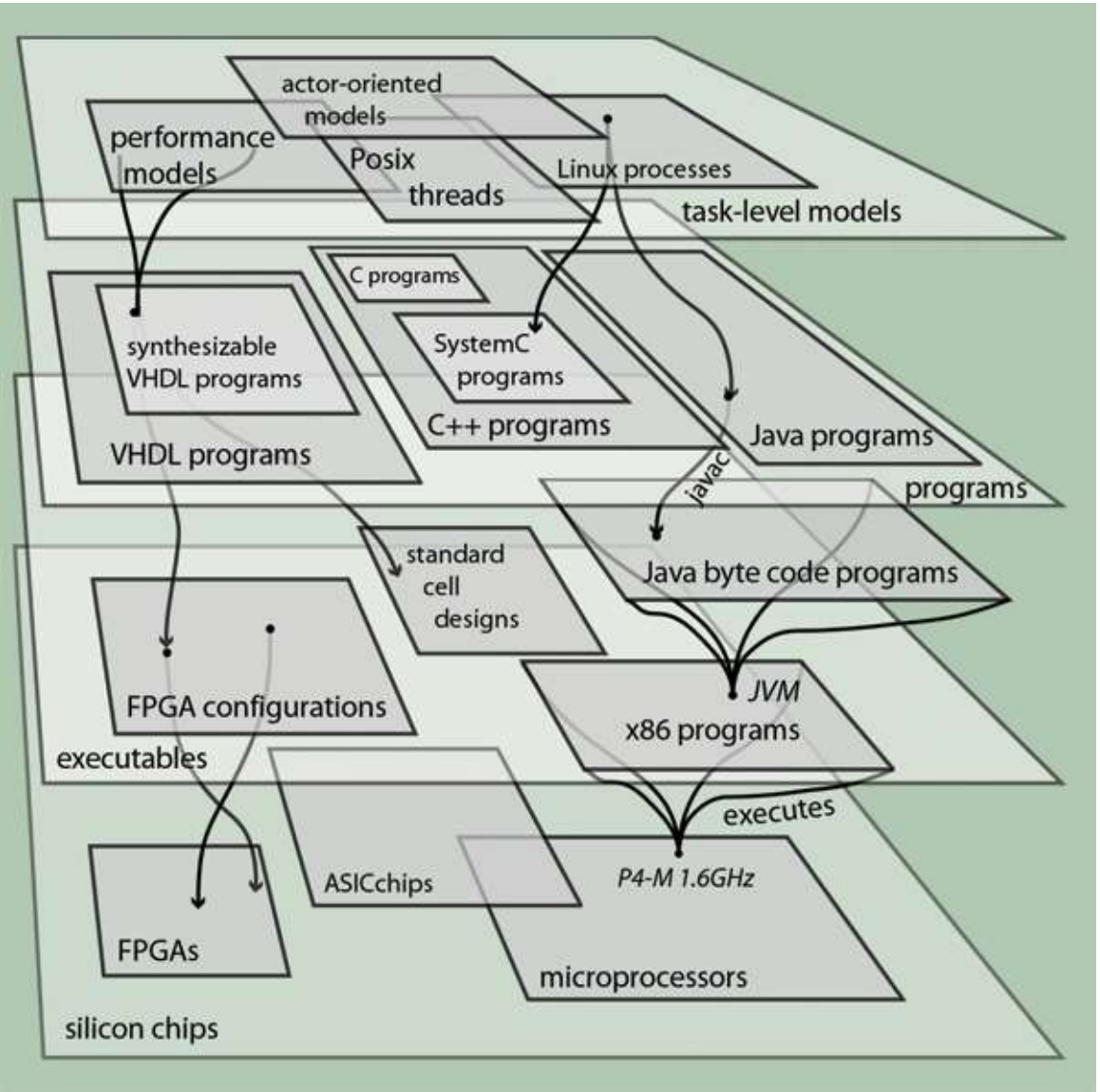
$$\dot{\mathbf{x}}(t) = \dot{\mathbf{x}}(0) + \frac{1}{M}\int_0^t \mathbf{F}(\tau)d\tau$$

Image: Wikimedia Commons

A Story

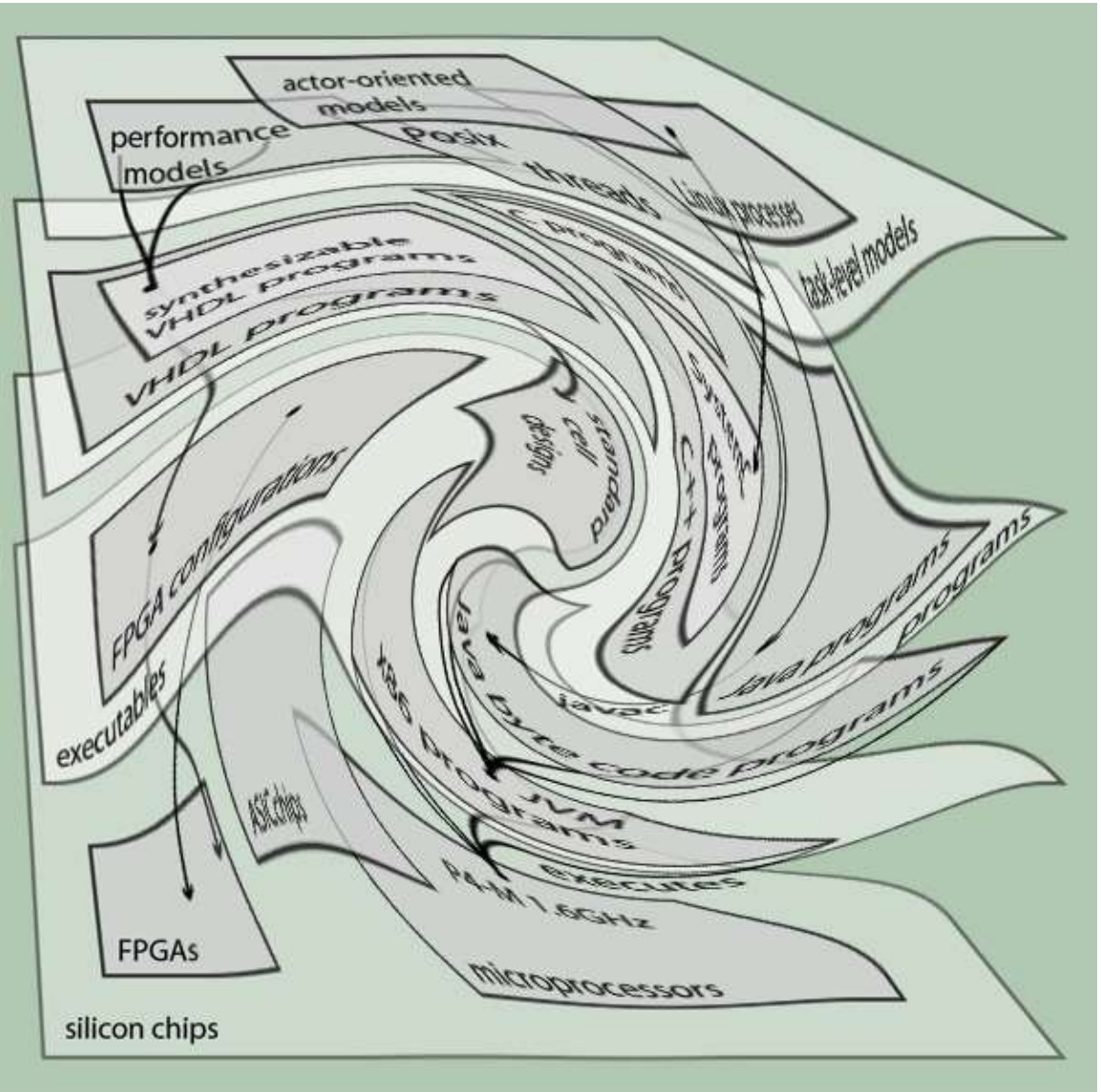
In “fly by wire” aircraft, computers control the plane, mediating pilot commands.

Abstraction Layers



The purpose of an abstraction is to hide details of the implementation below and provide a platform for design from above.

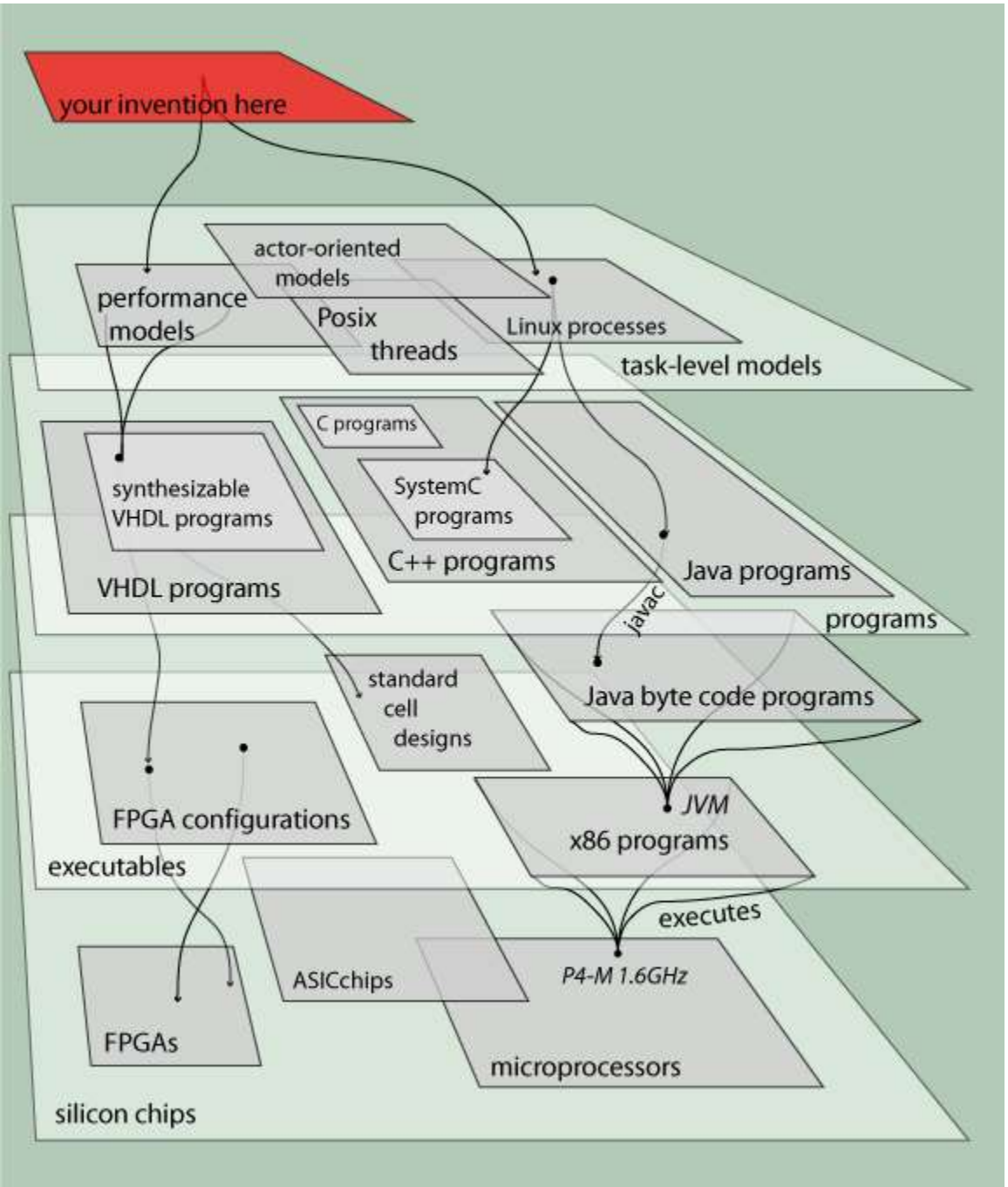
Abstraction Layers



Every abstraction layer has failed for the aircraft designer.

The design *is* the implementation.

Abstraction Layers



How about raising the level of abstraction to solve these problems?

Higher abstractions rely on an increasingly problematic fiction: WCET

A war story:

Ferdinand et al. [2001] determine the WCET of astonishingly simple avionics code from Airbus running on a Motorola ColdFire 5307, a pipelined CPU with a unified code and data cache. Despite the software consisting of a fixed set of non-interacting tasks containing only simple control structures, their solution required detailed modeling of the seven-stage pipeline and its precise interaction with the cache, generating a large integer linear programming problem.

Fundamentally, the ISA of the processor has failed to provide an adequate abstraction. And the problem has gotten worse since 2001!

Timing is not Part of Software and Network Semantics

Correct execution *of a program in all widely used programming languages, and* ***correct delivery*** *of a network message in all general-purpose networks has nothing to do with how long it takes to do anything.*

Programmers have to step outside the programming abstractions to specify timing behavior.

Embedded software designers have no map!

Determinism? Really?

CPS applications operate in an intrinsically nondeterministic world.

Does it really make sense to insist on deterministic models?

The Value of Models

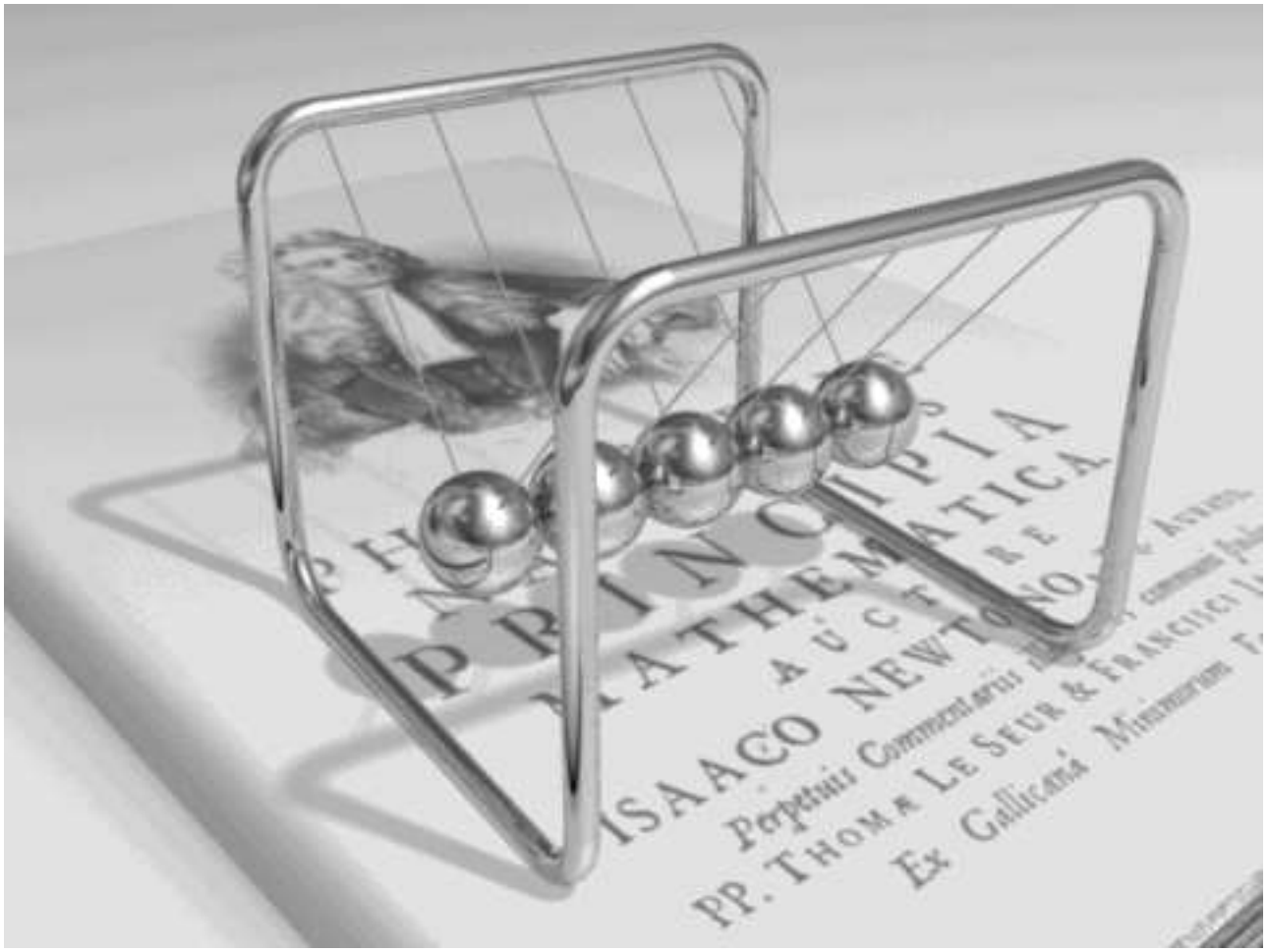
In *science*, the value of a *model* lies in how well its behavior matches that of the physical system.

In *engineering*, the value of the *physical system* lies in how well its behavior matches that of the model.

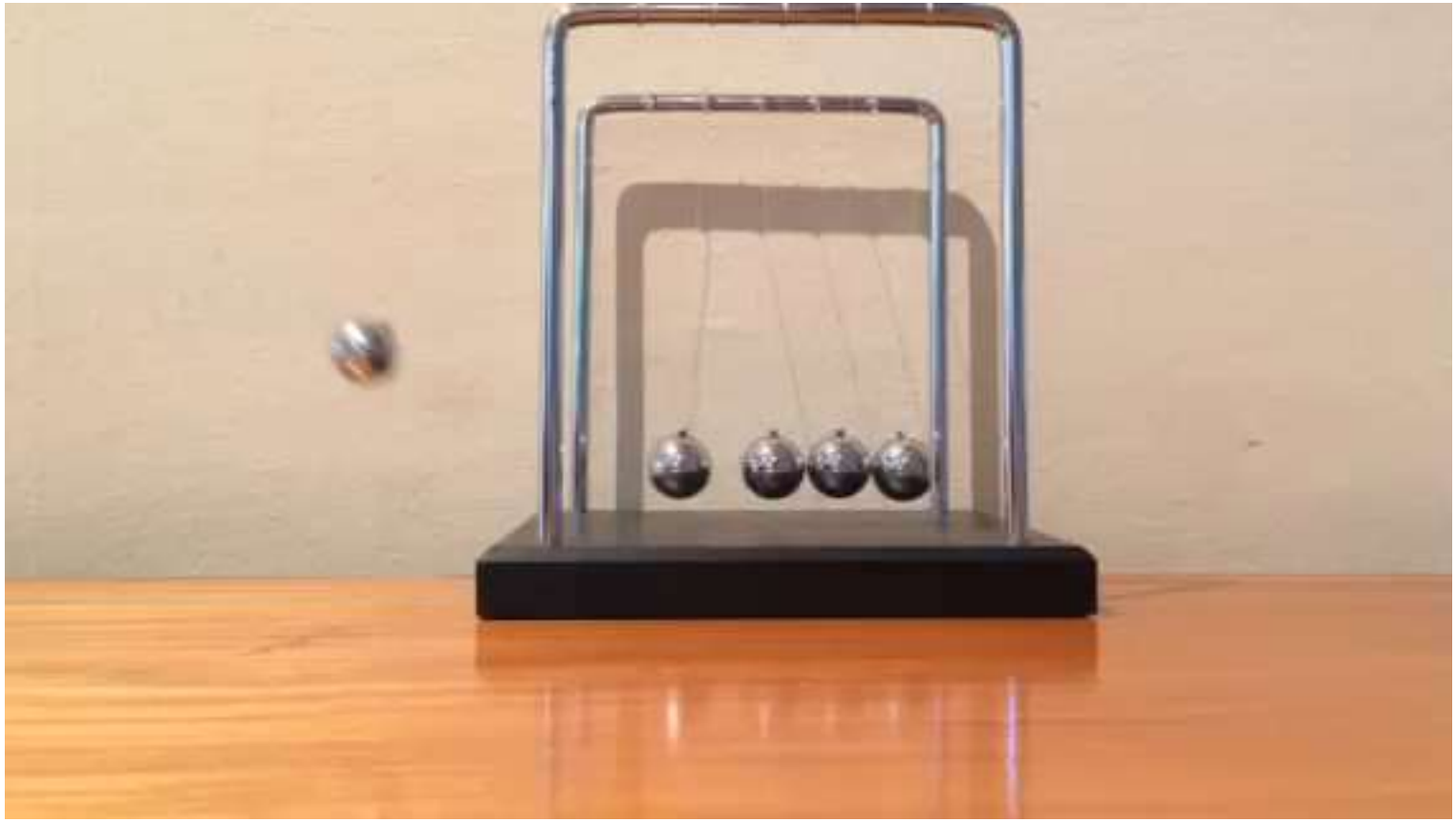
In engineering, model fidelity is a two-way street!

For a model to be useful, it is necessary (but not sufficient) to be able to be able to construct a faithful physical realization.

A Model


PRINCIPIA
MATHEMATICA
ISAACO NEWTONO
Perpetuis Commentariis illustrata
PP. THOMAS LE SEUR & FRANCISCI JACQUIER
Ex Gallicana Minimorum

A Physical Realization



Model Fidelity

To a *scientist*, the model is flawed.

To an *engineer*, the realization is flawed.

I'm an engineer…

For CPS, we need to Change the Question

The question is *not* whether deterministic models can describe the behavior of cyber-physical systems (with high fidelity).

The question is whether we can build cyber-physical systems whose behavior matches that of a deterministic model (with high probability).

Determinism?

What about Resilience? Adaptability?

Deterministic models do not eliminate the need for robust, fault-tolerant designs.

In fact, they *enable* such designs, because they make it much clearer what it means to have a fault!

We have to fix the models!

But how?

Introduction to Embedded Systems

Edward A. Lee

UC Berkeley

EECS 149/249A

Fall 2016

Module *2a*: Modeling Physical Dynamics

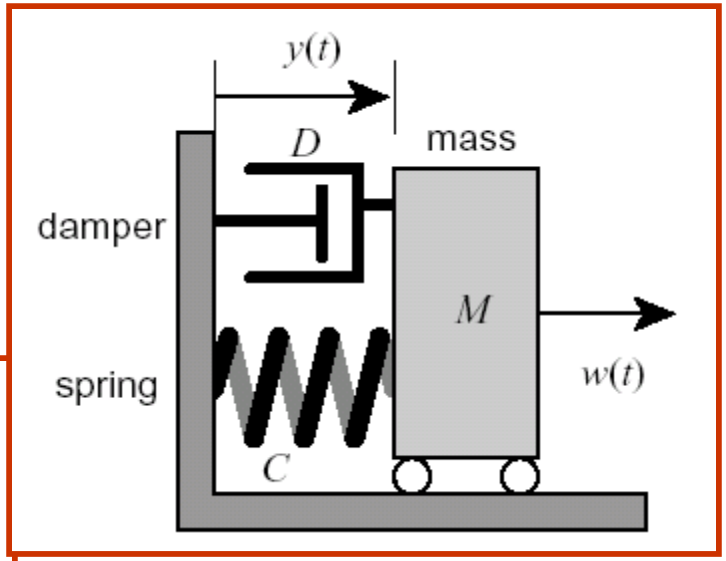
Modeling Techniques in this Course

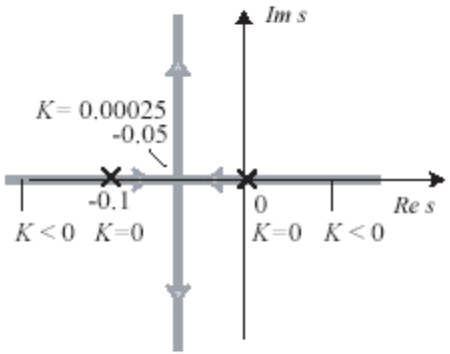
Models that are abstractions of **system dynamics**
(how system behavior changes over time)

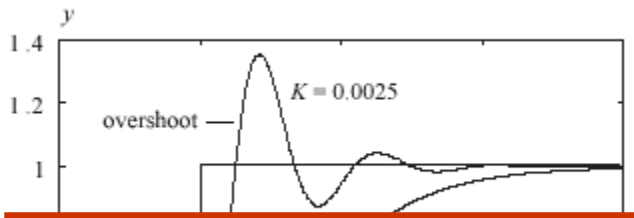
- Modeling physical phenomena – differential equations
- Feedback control systems – time-domain modeling
- Modeling modal behavior – FSMs, hybrid automata, …
- Modeling sensors and actuators –calibration, noise, …
- Hardware and software – concurrency, timing, power, …
- Networks – latencies, error rates, packet losses, …

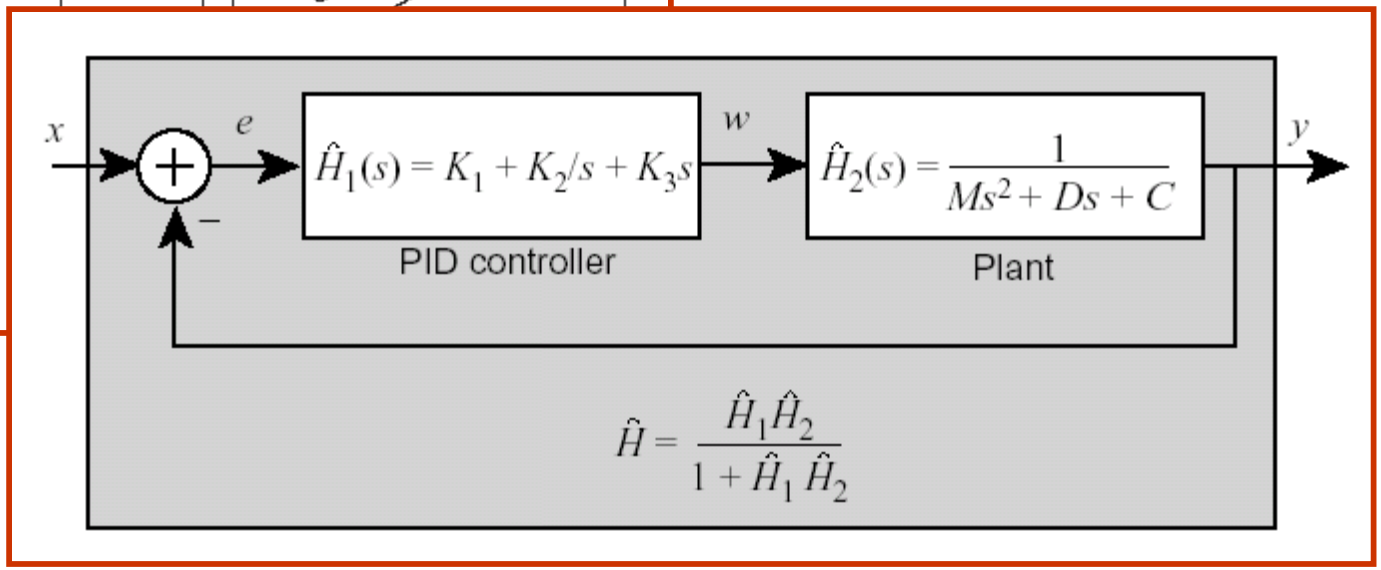
Today's Lecture: Modeling of Continuous Dynamics

Ordinary differential equations, Laplace transforms, feedback control models, …

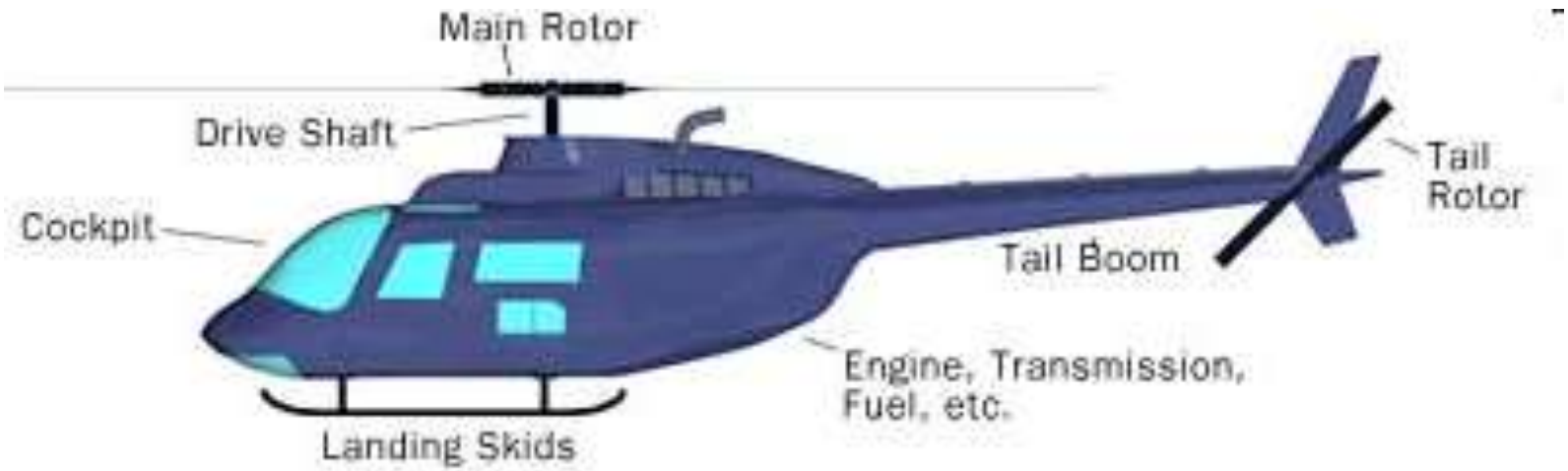








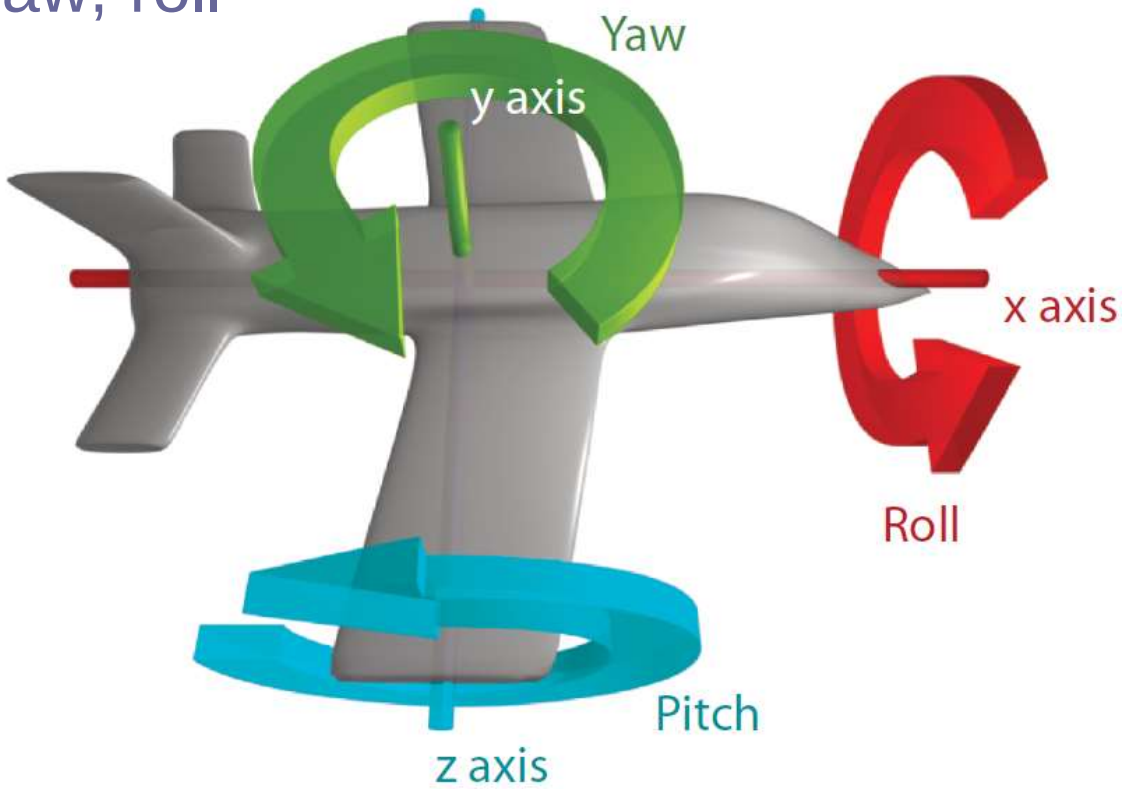
An Example: Helicopter Dynamics



Modeling Physical Motion

Six degrees of freedom:

- Position: x, y, z
- Orientation: pitch, yaw, roll



Notation

Position is given by three functions:

$$x : \mathbb{R} \to \mathbb{R}$$
$$y : \mathbb{R} \to \mathbb{R}$$
$$z : \mathbb{R} \to \mathbb{R}$$

where the domain $\mathbb{R}$ represents time and the co-domain (range) $\mathbb{R}$ represents position along the axis. Collecting into a vector:

$$\mathbf{x} : \mathbb{R} \to \mathbb{R}^3$$

Position at time $t \in \mathbb{R}$ is $\mathbf{x}(t) \in \mathbb{R}^3$.

Notation

Velocity

$$\dot{\mathbf{x}} \colon \mathbb{R} \to \mathbb{R}^3$$

is the derivative, $\forall\, t \in \mathbb{R}$,

$$\dot{\mathbf{x}}(t) = \frac{d}{dt}\mathbf{x}(t)$$

Acceleration $\ddot{\mathbf{x}} \colon \mathbb{R} \to \mathbb{R}^3$ is the second derivative,

$$\ddot{\mathbf{x}} = \frac{d^2}{dt^2}\mathbf{x}$$

Force on an object is $\mathbf{F} \colon \mathbb{R} \to \mathbb{R}^3$.

Newton's Second Law

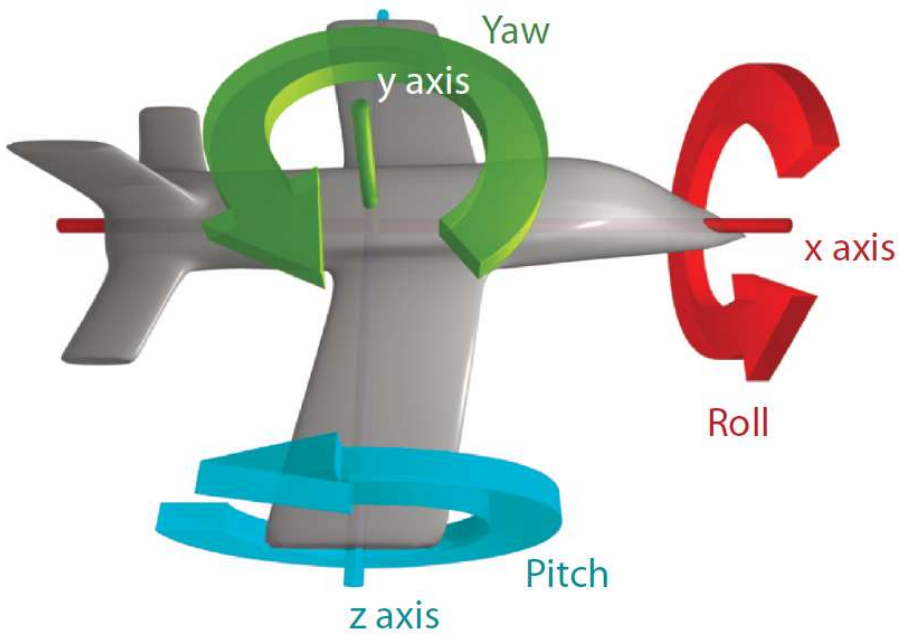
Newton's second law states $\forall\, t \in \mathbb{R}$,

$$\mathbf{F}(t) = M\ddot{\mathbf{x}}(t)$$

where M is the mass. To account for initial position and velocity, convert this to an integral equation

$$\begin{aligned}\mathbf{x}(t) &= \mathbf{x}(0) + \int_0^t \dot{\mathbf{x}}(\tau)d\tau \\ &= \mathbf{x}(0) + t\dot{\mathbf{x}}(0) + \frac{1}{M}\int_0^t \int_0^\tau \mathbf{F}(\alpha)d\alpha d\tau,\end{aligned}$$

Orientation

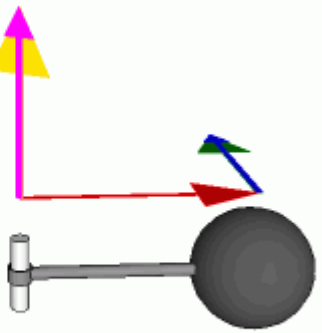


- Orientation: $\theta : \mathbb{R} \rightarrow \mathbb{R}^3$
- Angular velocity: $\dot{\theta} : \mathbb{R} \rightarrow \mathbb{R}^3$
- Angular acceleration: $\ddot{\theta} : \mathbb{R} \rightarrow \mathbb{R}^3$
- Torque: $\mathbf{T} : \mathbb{R} \rightarrow \mathbb{R}^3$

$$\theta(t) = \begin{bmatrix} \dot{\theta}_x(t) \\ \dot{\theta}_y(t) \\ \dot{\theta}_z(t) \end{bmatrix} = \begin{bmatrix} \text{roll} \\ \text{yaw} \\ \text{pitch} \end{bmatrix}$$

Angular version of force is torque.
For a point mass rotating around a fixed axis:

- radius of the arm: $r \in \mathbb{R}$
- force orthogonal to arm: $f \in \mathbb{R}$
- mass of the object: $m \in \mathbb{R}$



$$Ty(t) = r\, f(t)$$

angular momentum, momentum

Just as force is a push or a pull, a torque is a twist.

Units: newton-meters/radian, Joules/radian

Note that radians are meters/meter (2π meters of circumference per 1 meter of radius), so as units, are optional.

Rotational Version of Newton's Second Law

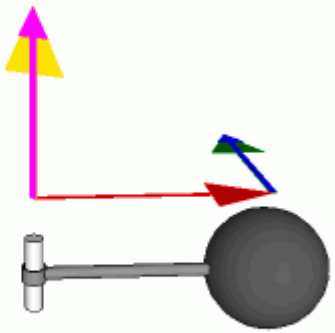
$$\mathbf{T}(t) = \frac{d}{dt}\left(I(t)\dot{\theta}(t)\right),$$

where $I(t)$ is a 3×3 matrix called the moment of inertia tensor.

$$\begin{bmatrix} T_x(t) \\ T_y(t) \\ T_z(t) \end{bmatrix} = \frac{d}{dt}\left(\begin{bmatrix} I_{xx}(t) & I_{xy}(t) & I_{xz}(t) \\ I_{yx}(t) & I_{yy}(t) & I_{yz}(t) \\ I_{zx}(t) & I_{zy}(t) & I_{zz}(t) \end{bmatrix}\begin{bmatrix} \dot{\theta}_x(t) \\ \dot{\theta}_y(t) \\ \dot{\theta}_z(t) \end{bmatrix}\right)$$

Here, for example, $T_y(t)$ is the net torque around the y axis (which would cause changes in yaw), $I_{yx}(t)$ is the inertia that determines how acceleration around the x axis is related to torque around the y axis.

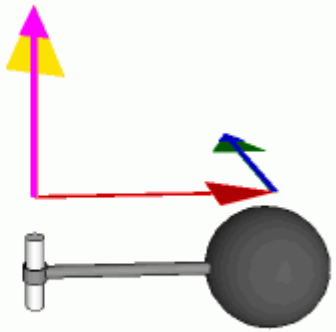
Feedback Control Problem



A helicopter without a tail rotor, like the one below, will spin uncontrollably due to the torque induced by friction in the rotor shaft.

Control system problem: Apply torque using the tail rotor to counterbalance the torque of the top rotor.

Simplified Model



Yaw dynamics:

$$T_y(t) = I_{yy}\ddot{\theta}_y(t)$$

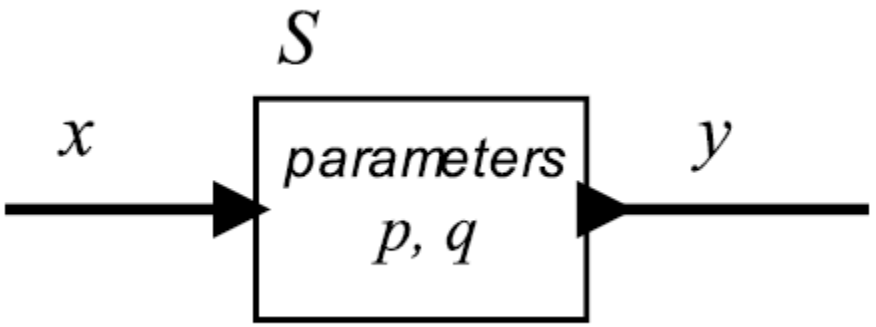
To account for initial angular velocity, write as

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau.$$

“Plant” and Controller

Actor Model of Systems

A *system* is a function that accepts an input *signal* and yields an output signal.



The domain and range of the system function are sets of signals, which themselves are functions.

$$x: \mathbb{R} \rightarrow \mathbb{R}, \quad y: \mathbb{R} \rightarrow \mathbb{R}$$

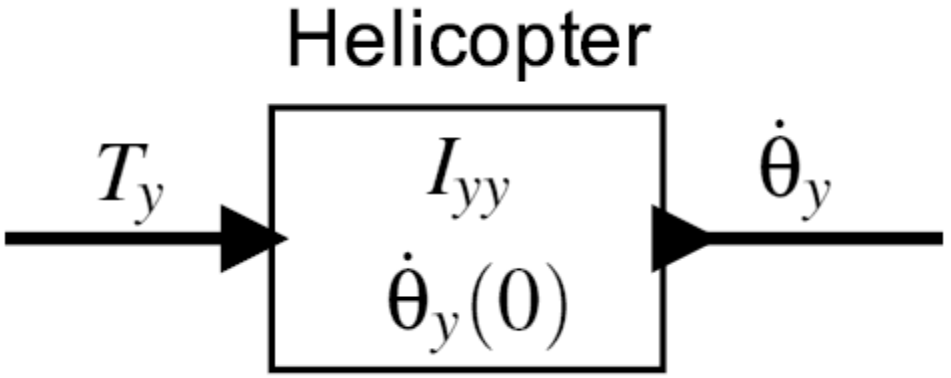
$$S: X \rightarrow Y$$

$$X = Y = (\mathbb{R} \rightarrow \mathbb{R})$$

Parameters may affect the definition of the function *S*.

Actor Model of the Helicopter

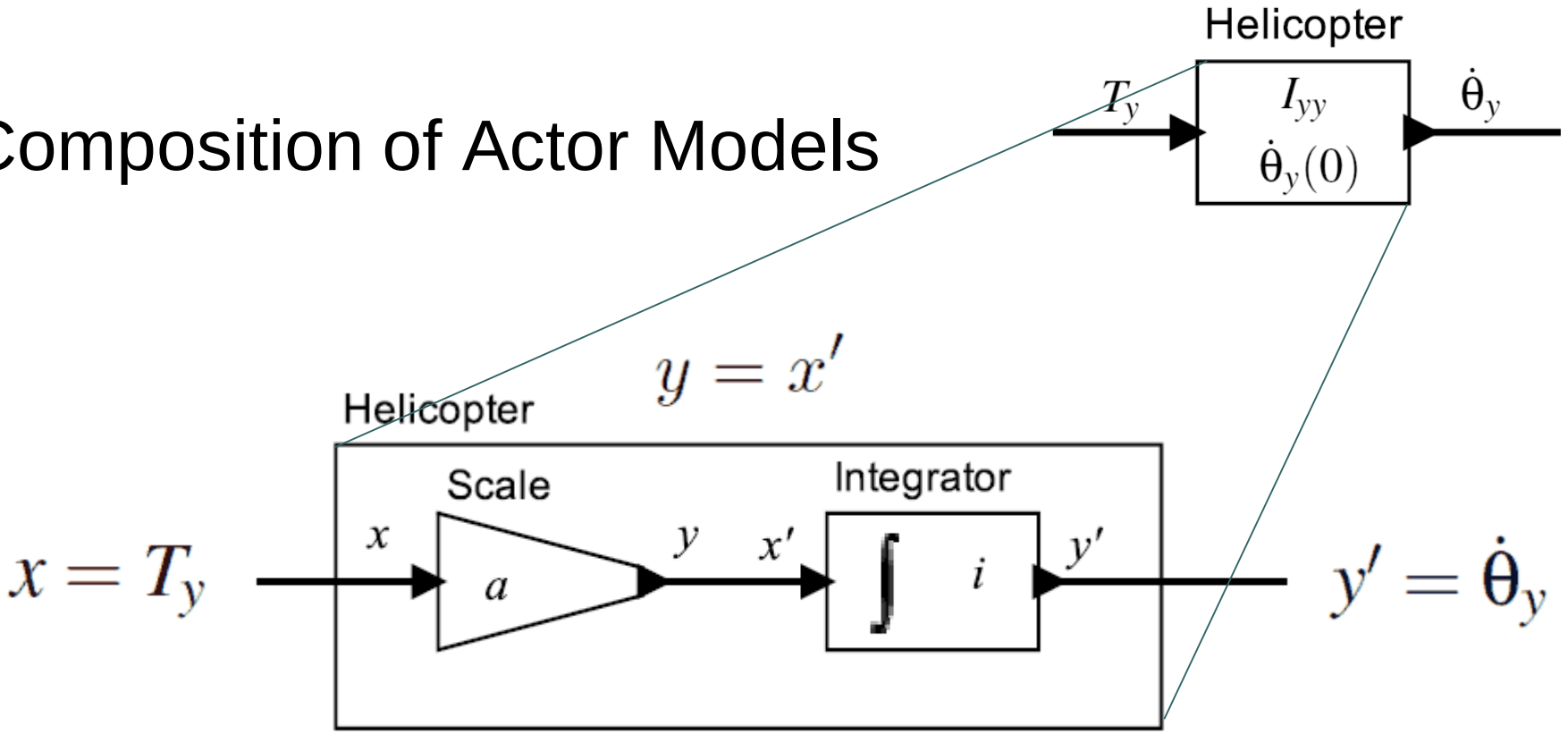
Input is the net torque of the tail rotor and the top rotor. Output is the angular velocity around the *y* axis.



Parameters of the model are shown in the box. The input and output relation is given by the equation to the right.

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau$$

Composition of Actor Models



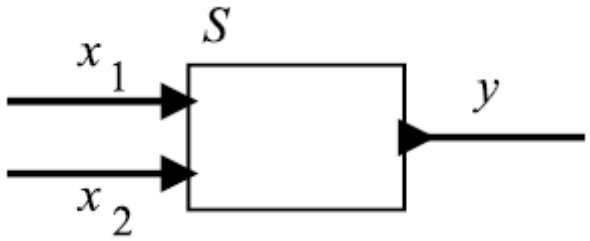
$$\forall\, t \in \mathbb{R}, \quad y(t) = ax(t) \qquad y'(t) = i + \int_0^t x'(\tau)\, d\tau$$

$$y = ax$$

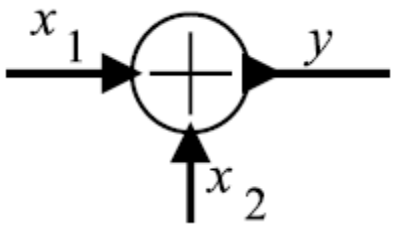
$$i = \dot{\theta}_y(0)$$

$$a = 1/I_{yy}$$

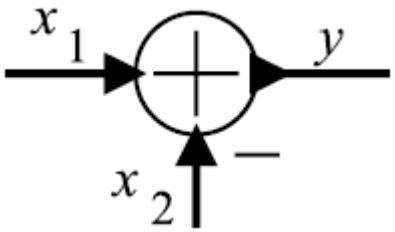
Actor Models with Multiple Inputs



$$S: (\mathbb{R} \to \mathbb{R})^2 \to (\mathbb{R} \to \mathbb{R})$$

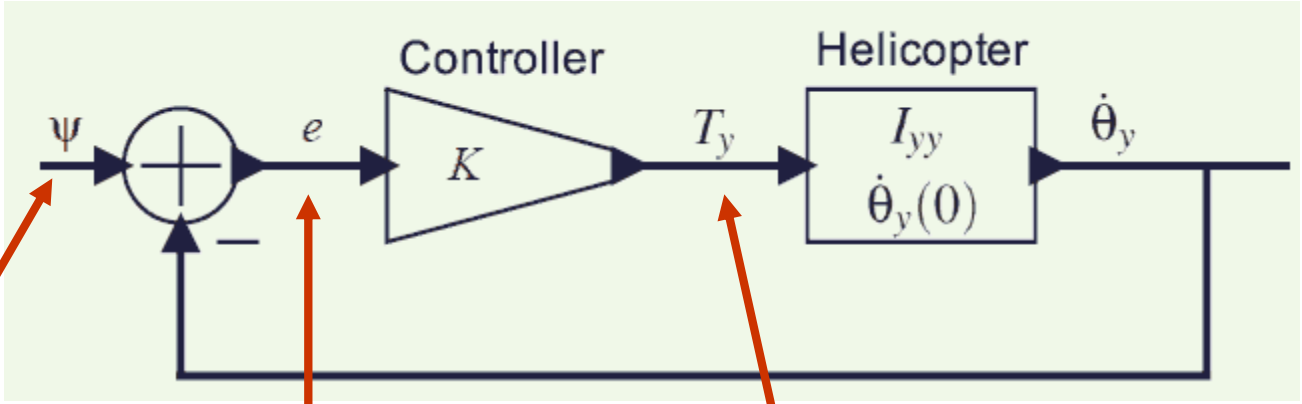


$$\forall\, t \in \mathbb{R}, \quad y(t) = x_1(t) + x_2(t)$$



$$(S(x_1, x_2))(t) = y(t) = x_1(t) - x_2(t)$$

Proportional controller



desired angular velocity

error signal

net torque

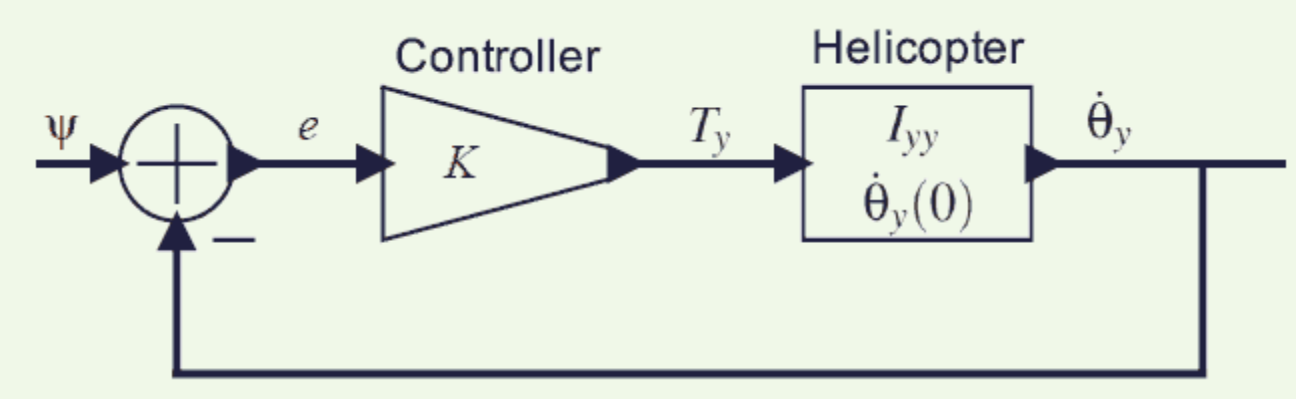
$$e(t) = \psi(t) - \dot{\theta}_y(t) \qquad T_y(t) = Ke(t)$$

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{1}{I_{yy}} \int_0^t T_y(\tau) d\tau$$

$$= \dot{\theta}_y(0) + \frac{K}{I_{yy}} \int_0^t (\psi(\tau) - \dot{\theta}_y(\tau)) d\tau$$

Note that the angular velocity appears on both sides, so this equation is not trivial to solve.

Behavior of the controller



$$\dot{\theta}_y(t) = \dot{\theta}_y(0) + \frac{K}{I_{yy}} \int_0^t (\psi(\tau) - \dot{\theta}_y(\tau)) d\tau$$

Desired angular velocity: $\psi(t) = 0$

Simplifies differential equation to:

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) - \frac{K}{I_{yy}} \int_0^t \dot{\theta}_y(\tau) d\tau$$

Which can be solved as follows (see textbook):

$$\dot{\theta}_y(t) = \dot{\theta}_y(0) e^{-Kt/I_{yy}} u(t)$$

Questions

- Can the behavior of this controller change when it is implemented in software?
- How do we measure the angular velocity in practice? How do we incorporate noise into this model?
- What happens when you have failures (sensors, actuators, software, computers, or networks) https://www.youtube.com/watch?v=MhEXXgiIVuY

Introduction to Embedded Systems

Edward A. Lee

UC Berkeley

EECS 149/249A

Fall 2016

Chapter 3: Discrete Dynamics, State Machines

Discrete Systems

Discrete = “individually separate / distinct”

A **discrete system** is one that operates in a sequence of discrete *steps* or has signals taking discrete *values*.

It is said to have **discrete dynamics**.

Concepts covered in Today's Lecture

Models = Programs

Actor Models of Discrete Systems: Types and Interfaces

States, Transitions, Guards

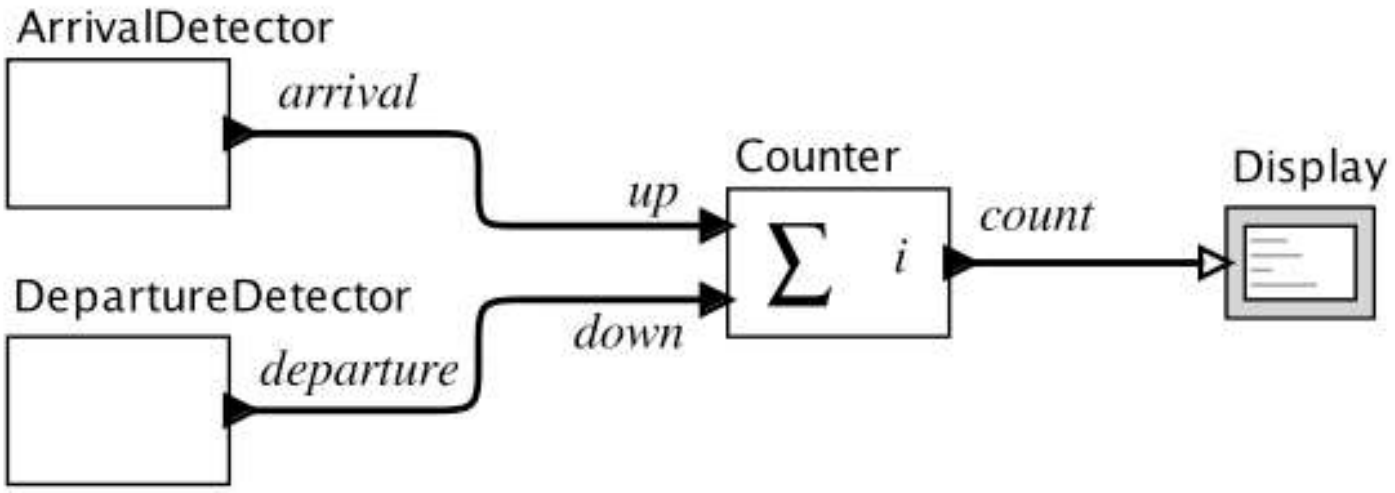
Determinism and Receptiveness

Discrete Systems: Example Design Problem

Count the number of cars that are present in a parking garage by sensing cars enter and leave the garage. Show this count on a display.

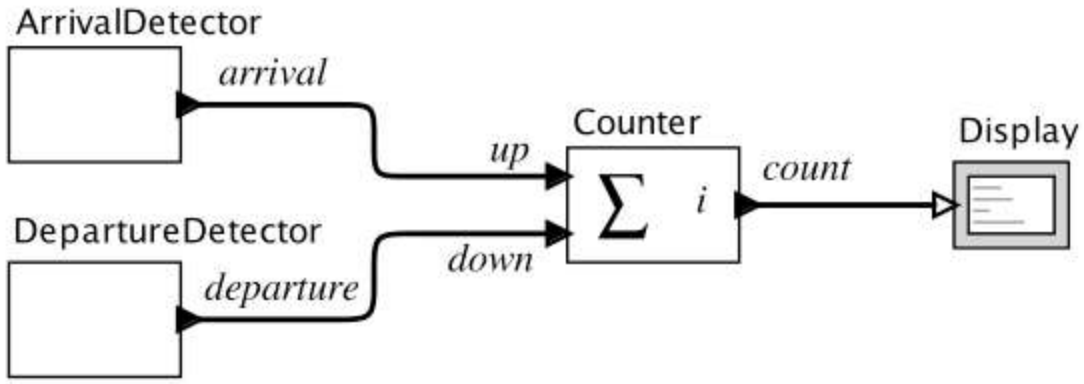
Discrete Systems

Example: count the number of cars in a parking garage by sensing those that enter and leave:



Discrete Systems

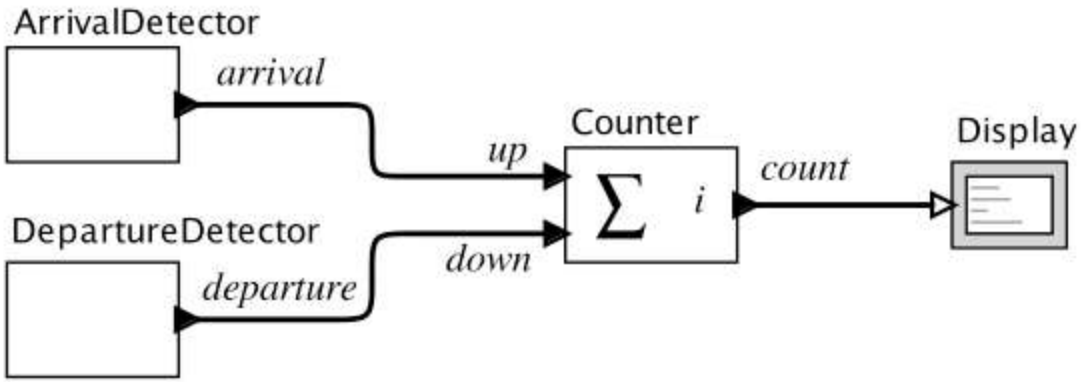
Example: count the number of cars that enter and leave a parking garage:



Pure signal: $up : \mathbb{R} \rightarrow \{absent, present\}$

Discrete Systems

Example: count the number of cars that enter and leave a parking garage:



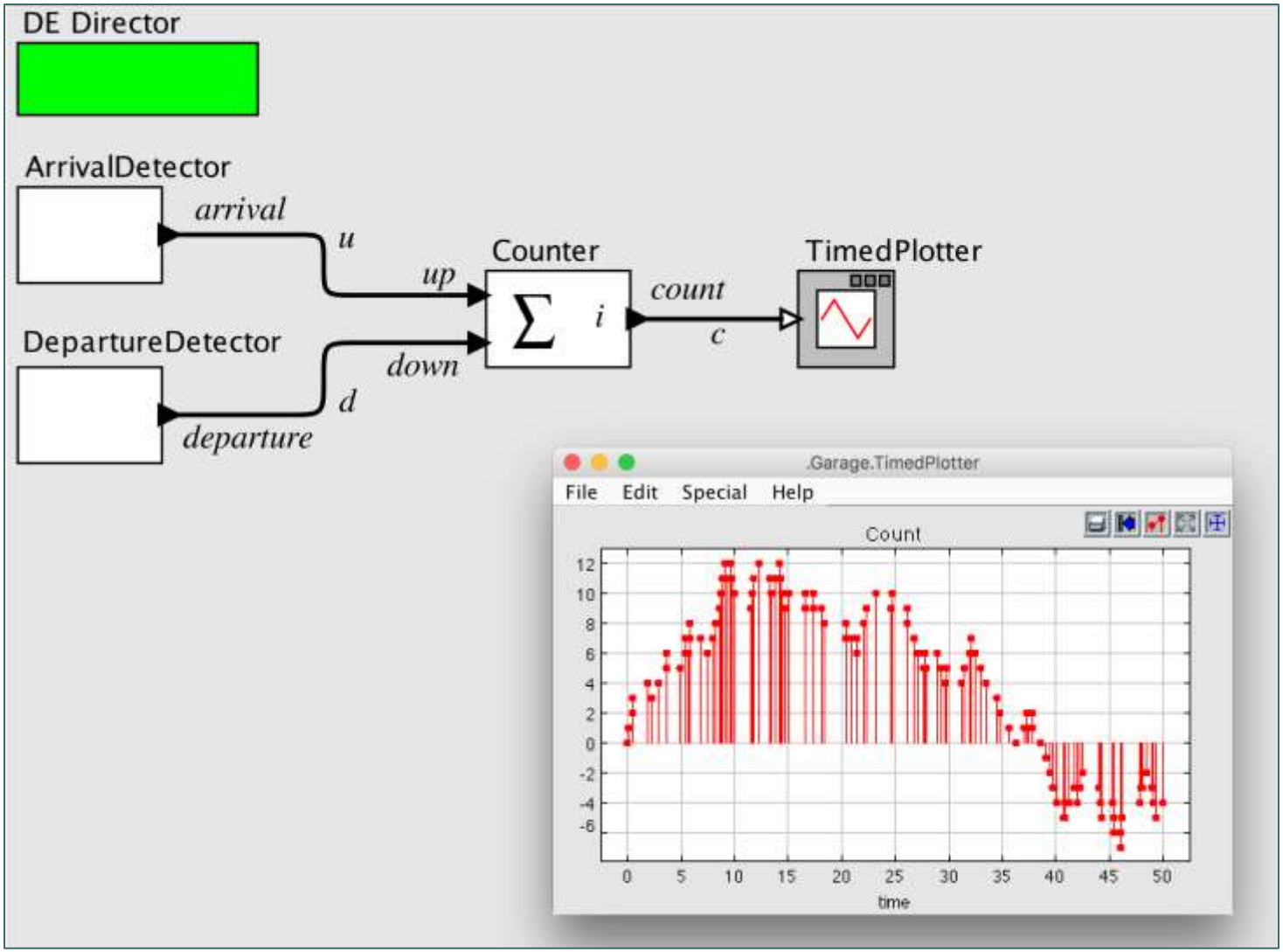
Pure signal: $up : \mathbb{R} \rightarrow \{absent, present\}$

Discrete actor:

$$Counter : (\mathbb{R} \rightarrow \{absent, present\})^P \rightarrow (\mathbb{R} \rightarrow \{absent\} \cup \mathbb{N})$$

$$P = \{up, down\}$$

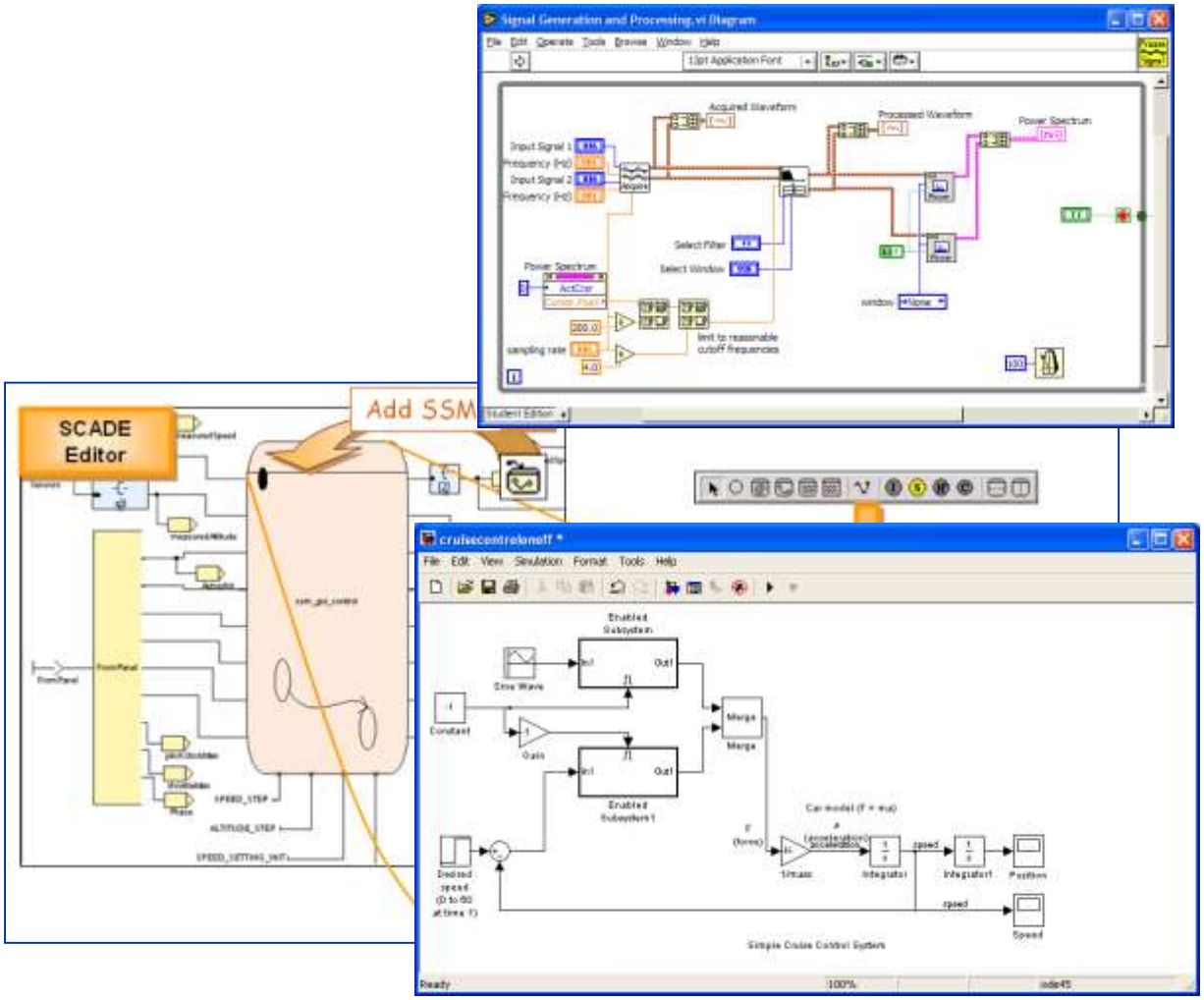
Demonstration of Ptolemy II Model (“Program”)



Actor Modeling Languages / Frameworks

- LabVIEW
- Simulink
- Scade
- …

- Reactors
- StreamIT
- …

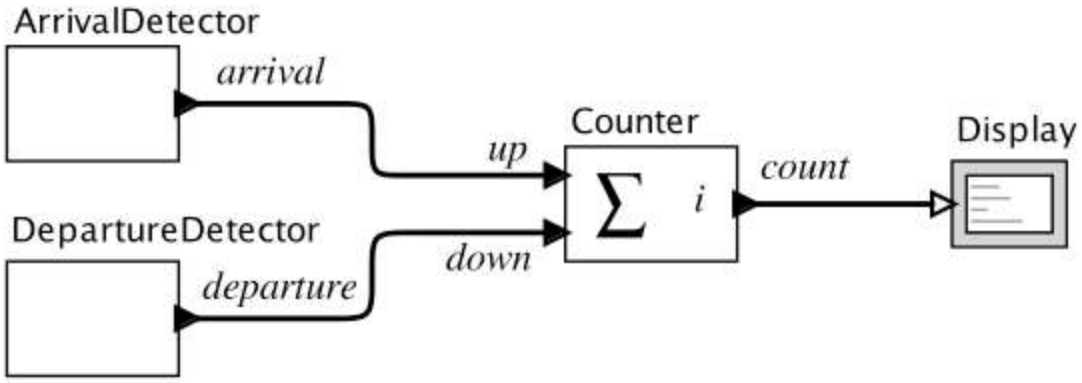


Reaction / Transition

For any $t \in \mathbb{R}$ where $up(t) \neq absent$ or $down(t) \neq absent$ the Counter **reacts**. It produces an output value in $\mathbb{N}$ and changes its internal **state**.

State: condition of the system at a particular point in time

- Encodes everything about the past that influences the system's reaction to current input



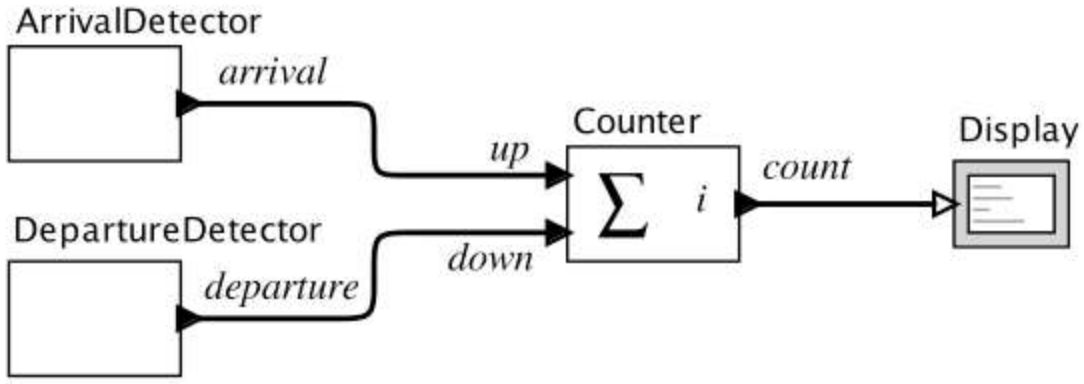
Inputs and Outputs at a Reaction

For $t \in \mathbb{R}$ the inputs are in a set

$$Inputs = (\{up, down\} \rightarrow \{ absent, present\})$$

and the outputs are in a set

$$Outputs = (\{count\} \rightarrow \{ absent\} \cup \mathbb{N}) \ ,$$



Question

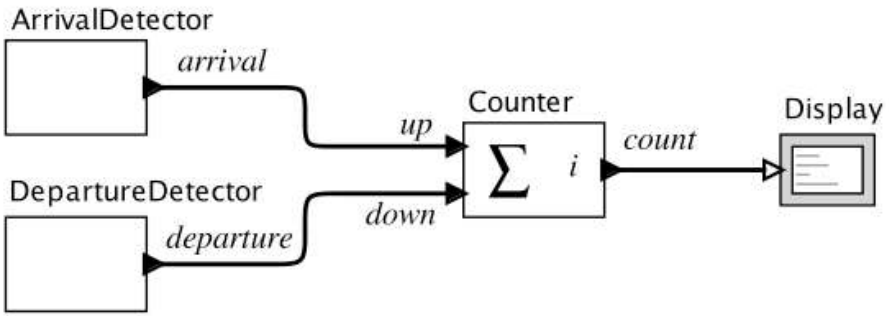
What are some scenarios that the given parking garage (interface) design does not handle well?

For $t \in \mathbb{R}$ the inputs are in a set

$$Inputs = (\{up, down\} \rightarrow \{absent, present\})$$

and the outputs are in a set

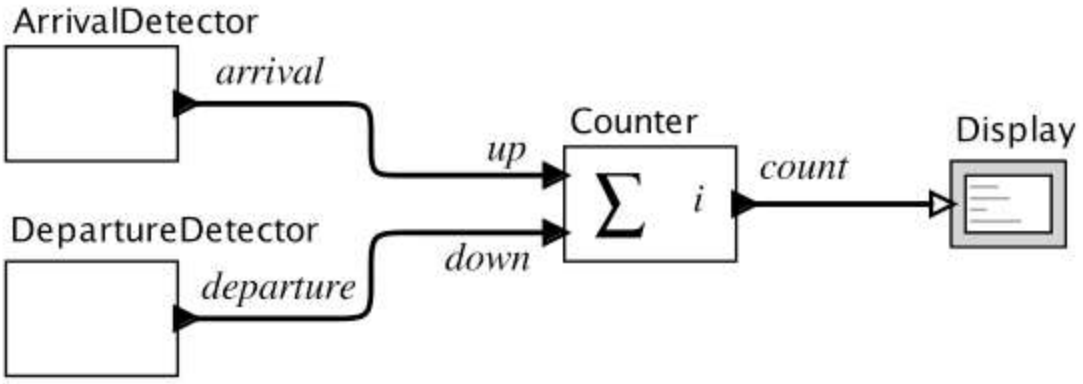
$$Outputs = (\{count\} \rightarrow \{absent\} \cup \mathbb{N}) \ ,$$



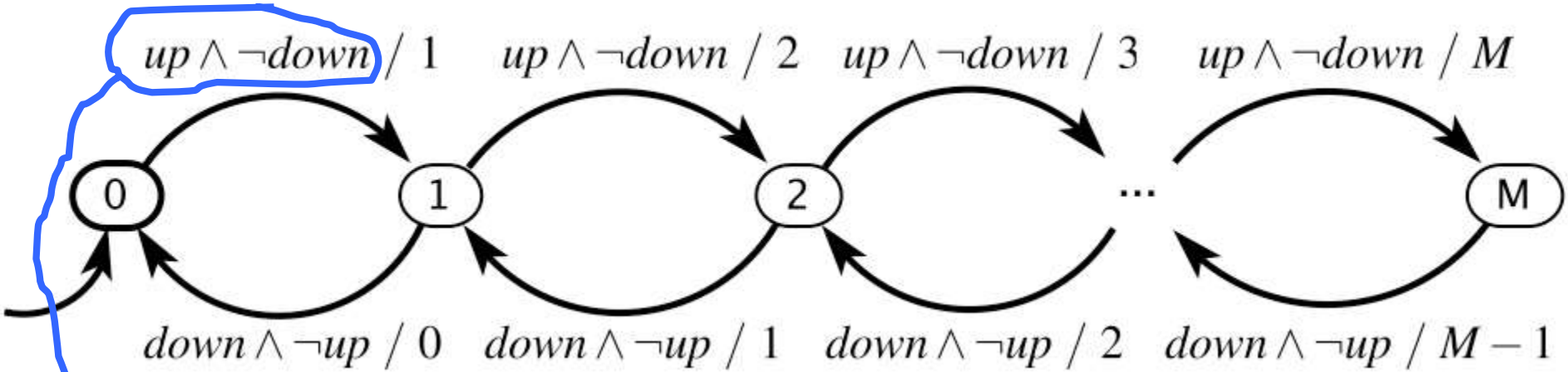
State Space

A practical parking garage has a finite number M of spaces, so the state space for the counter is

$$States = \{0, 1, 2, \cdots, M\} \ .$$



Garage Counter Finite State Machine (FSM) in Pictures



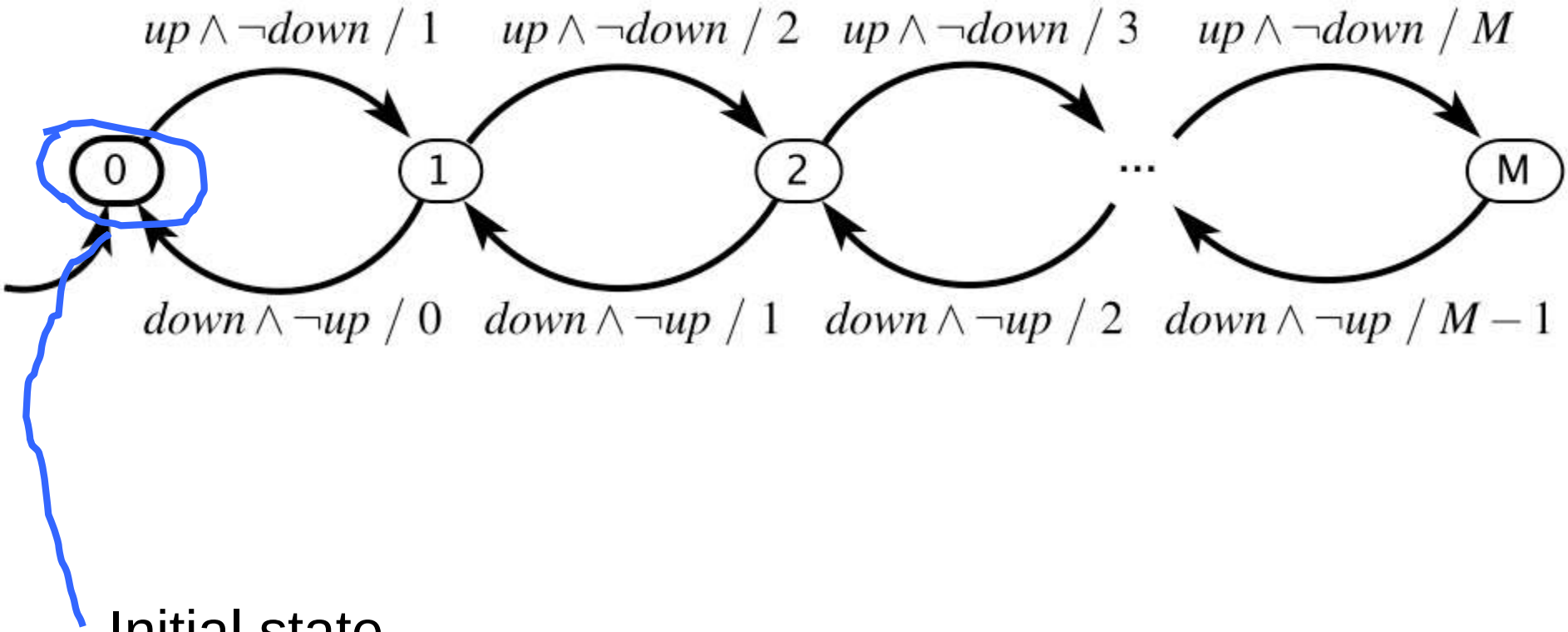
Guard $g \subseteq Inputs$ is specified using the shorthand

$$up \wedge \neg down$$

which means

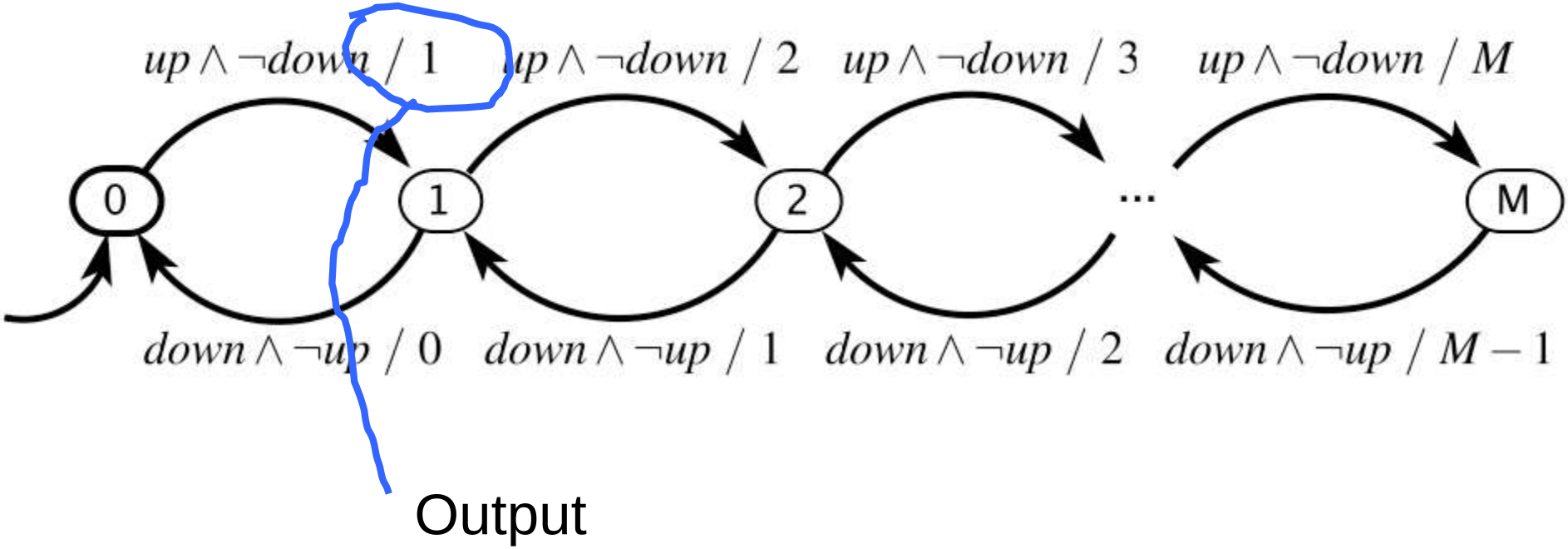
$$g = \{\{up\}\} .$$

Garage Counter Finite State Machine (FSM) in Pictures



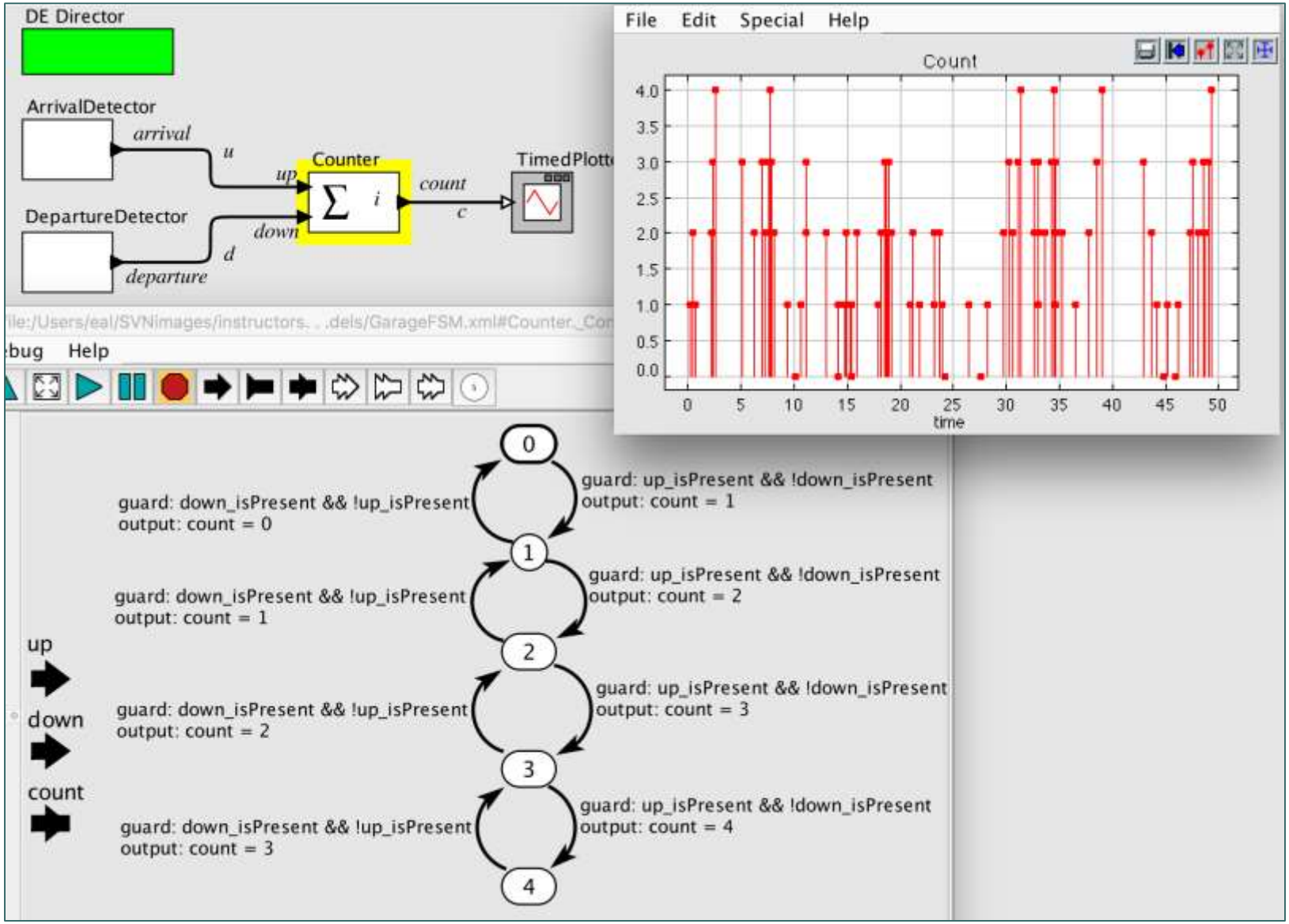
Initial state

Garage Counter Finite State Machine (FSM) in Pictures



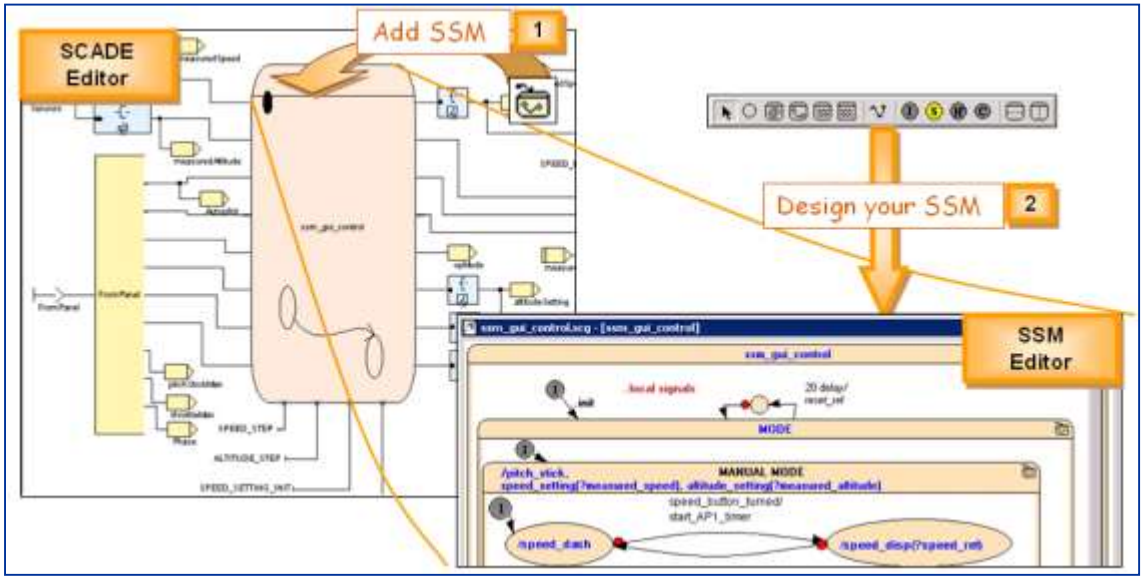
Output

Ptolemy II Model

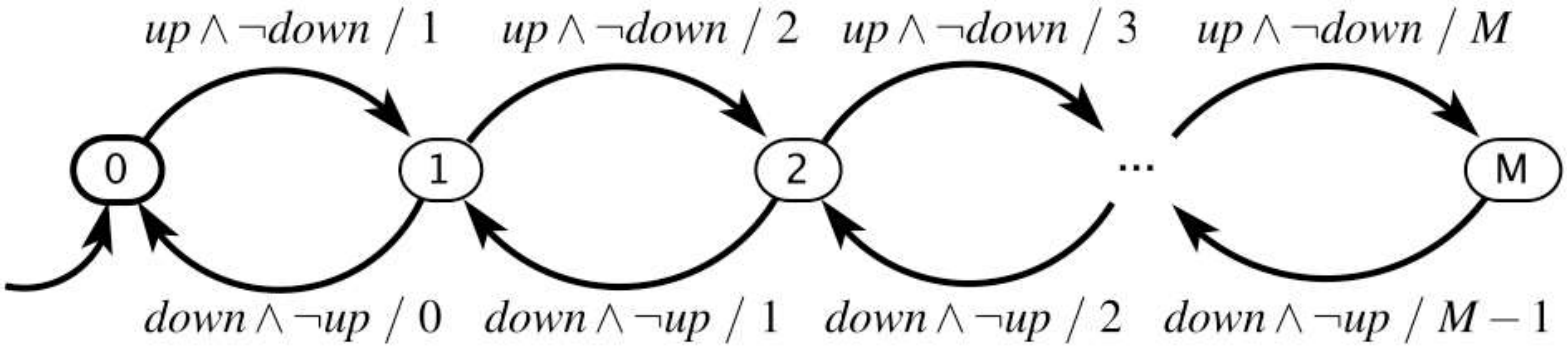


FSM Modeling Languages / Frameworks

- LabVIEW Statecharts
- Simulink Stateflow
- Scade
- …



Garage Counter Mathematical Model

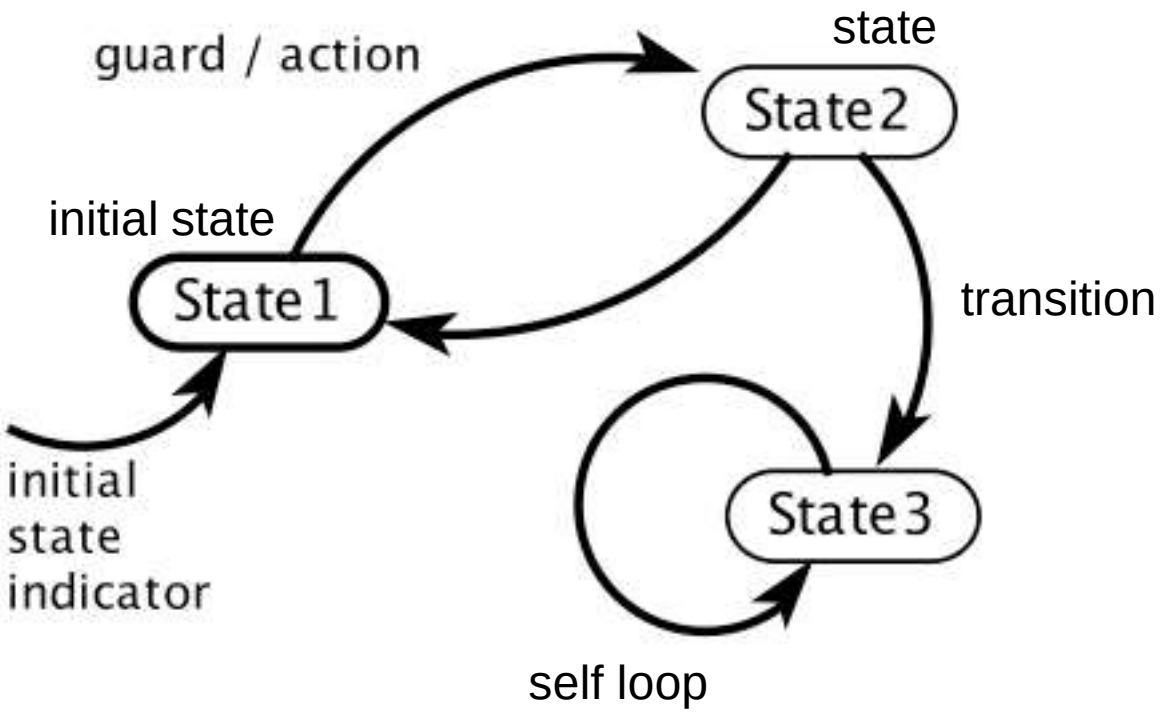


Formally: $(States, Inputs, Outputs, update, initialState)$, where

- $States = \{0, 1, \cdots, M\}$
- $Inputs = (\{up, down\} \rightarrow \{absent, present\}$
- $Outputs = (\{count\} \rightarrow \{absent\} \cup \mathbb{N})$
- $update : States \times Inputs \rightarrow States \times Outputs$
- $initialState = 0$

The picture above defines the update function.

FSM Notation in Lee & Seshia



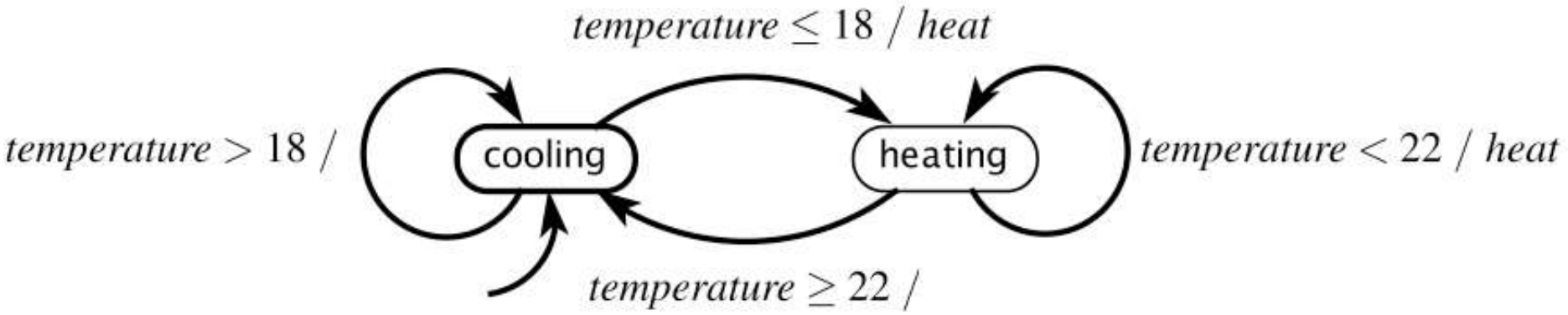
Examples of Guards for Pure Signals

true	Transition is always enabled.
p_1	Transition is enabled if p_1 is *present*.
$\neg p_1$	Transition is enabled if p_1 is *absent*.
$p_1 \wedge p_2$	Transition is enabled if both p_1 and p_2 are *present*.
$p_1 \vee p_2$	Transition is enabled if either p_1 or p_2 is *present*.
$p_1 \wedge \neg p_2$	Transition is enabled if p_1 is *present* and p_2 is *absent*.

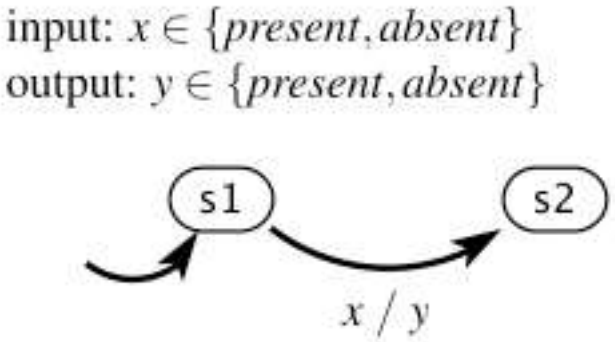
Examples of Guards for Signals with Numerical Values

p_3	Transition is enabled if p_3 is *present* (not *absent*).
$p_3 = 1$	Transition is enabled if p_3 is *present* and has value 1.
$p_3 = 1 \wedge p_1$	Transition is enabled if p_3 has value 1 and p_1 is *present*.
$p_3 > 5$	Transition is enabled if p_3 is *present* with value greater than 5.

Example of *Modal* Model: Thermostat



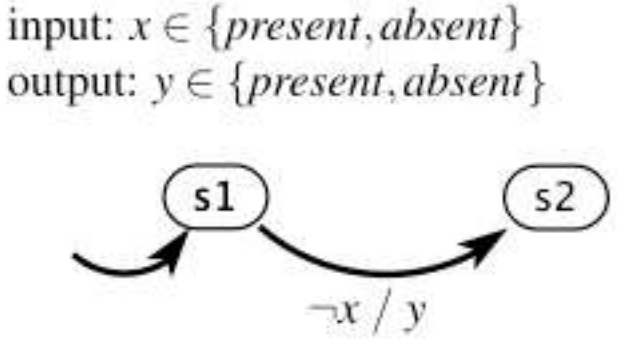
When does a reaction occur?



Suppose all inputs are discrete and a reaction occurs *when any input is present*. Then the above transition will be taken whenever the current state is s1 and *x* is present.

This is an *event-triggered model*.

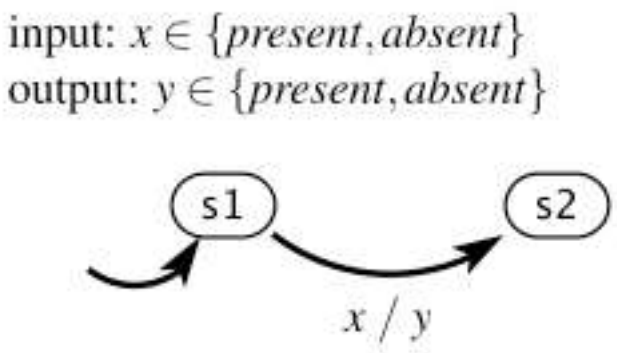
When does a reaction occur?

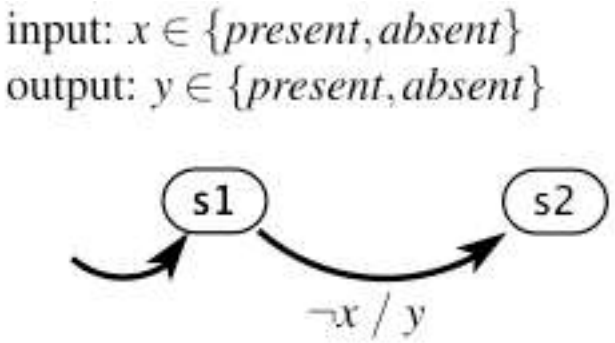


Suppose *x* and *y* are discrete and pure signals.
When does the transition occur?

Answer: when the *environment* triggers a reaction and x is absent. If this is a (complete) event-triggered model, then the transition will never be taken because the reaction will only occur when x is present!

When does a reaction occur?

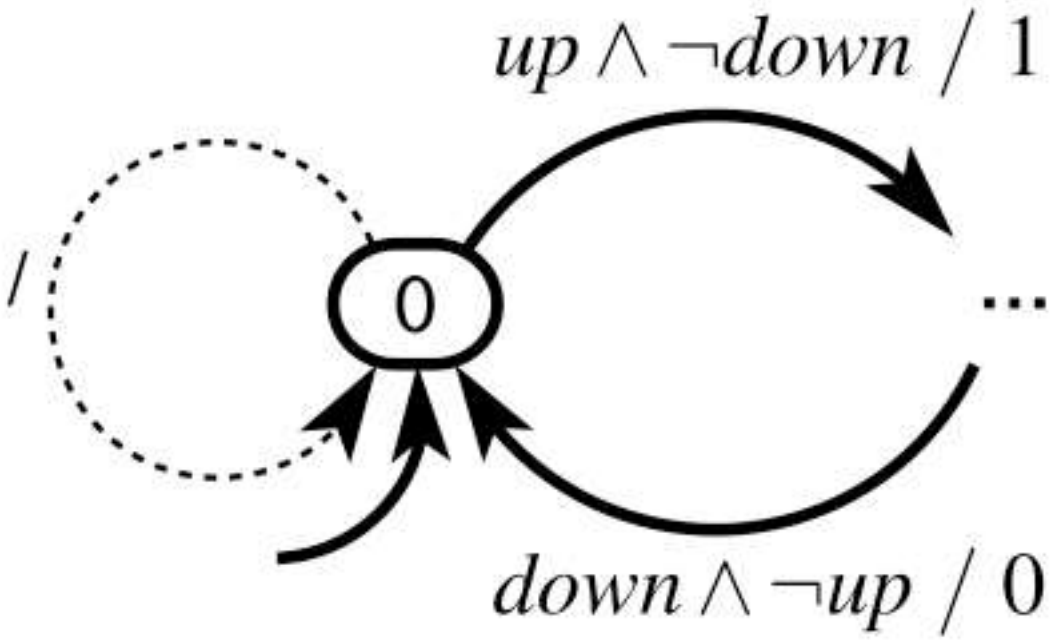




Suppose all inputs are discrete and a reaction occurs *on the tick of an external clock*.

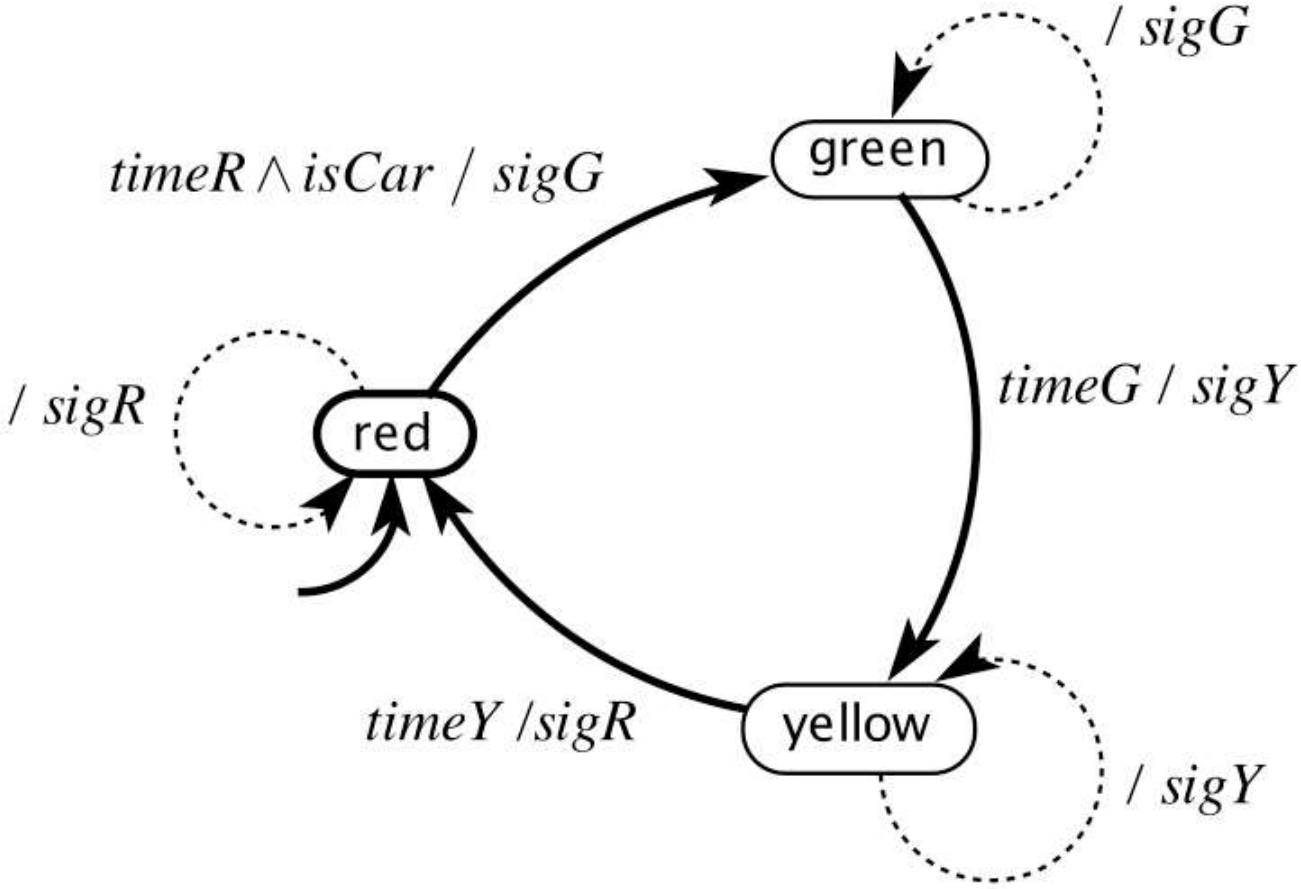
This is a *time-triggered model*.

More Notation: Default Transitions



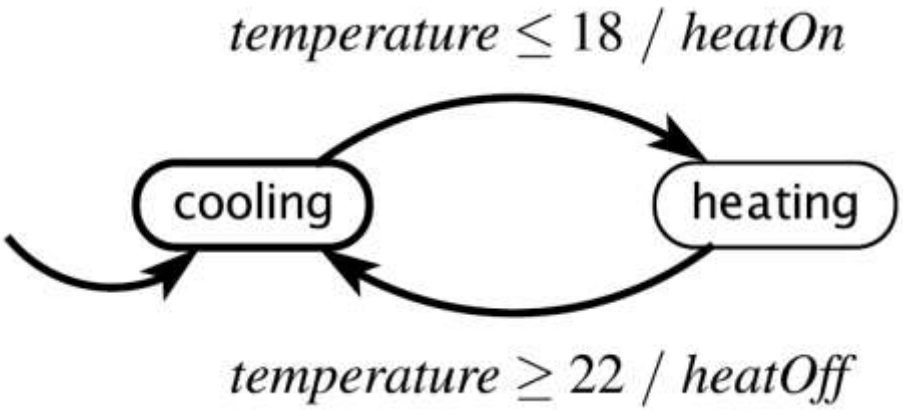
A default transition is enabled if no non-default transition is enabled and it either has no guard or the guard evaluates to true. When is the above default transition enabled?

Only show default transitions if they are guarded or produce outputs (or go to other states)
Example: Traffic Light Controller



Example where default transitions need not be shown

input: *temperature* : $\mathbb{R}$
outputs: *heatOn*, *heatOff* : pure



Exercise: From this picture, construct the formal mathematical model.

Some Definitions

- **Stuttering transition**: (possibly implicit) default transition that is enabled when inputs are absent, that does not change state, and that produces absent outputs.
- **Receptiveness**: For any input values, some transition is enabled. Our structure together with the implicit default transition ensures that our FSMs are receptive.
- **Determinism**: In every state, for all input values, exactly one (possibly implicit) transition is enabled.

Test Your Understanding: Three Kinds of Transitions

Self-Loop

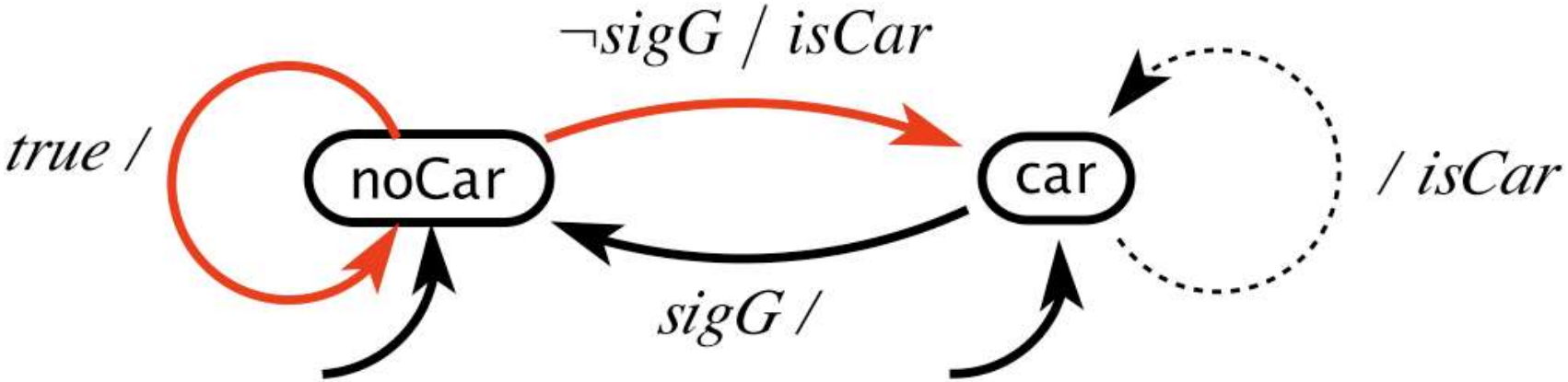
Default Transition

Stuttering Transition

1. Is a default transition always a self-loop?
2. Is a stuttering transition always a self-loop?
3. Is a self-loop always stuttering?

Example: Nondeterministic FSM

Model of the environment for a traffic light, abstracted using nondeterminism:



Formally, the update function is replaced by a function

$$possibleUpdates : States \times Inputs \rightarrow 2^{States \times Outputs}$$

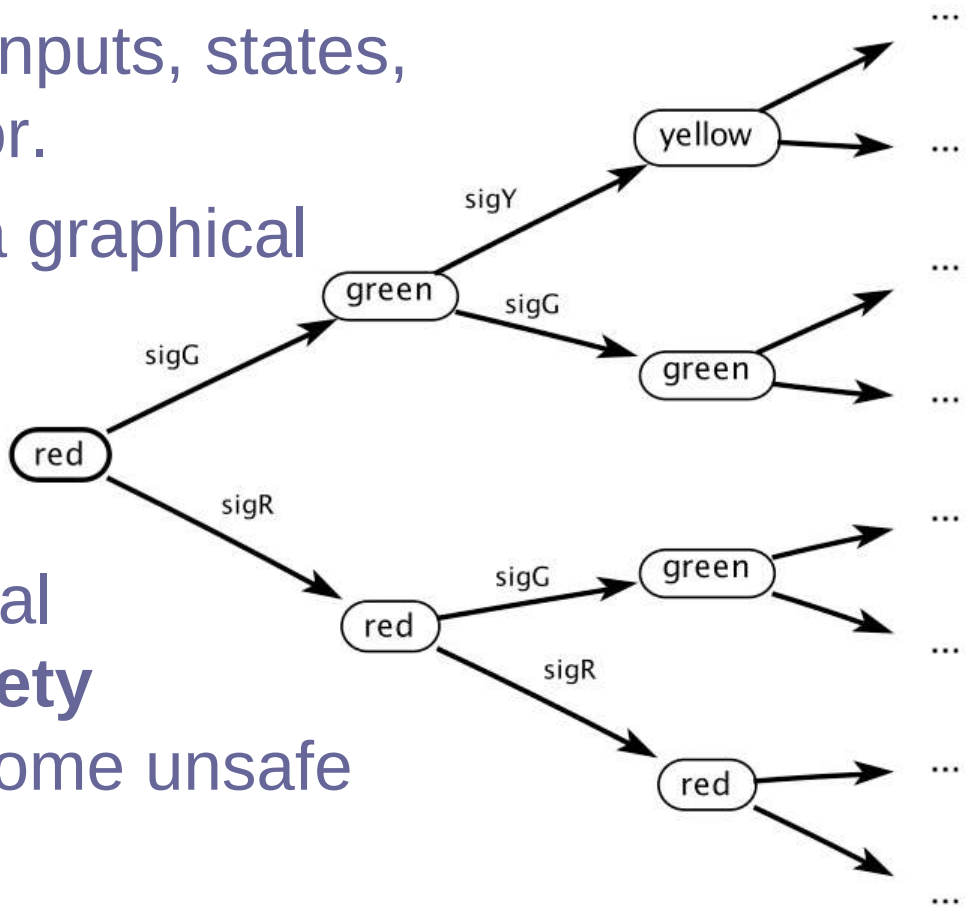
Uses of Nondeterminism

1. Modeling unknown aspects of the environment or system
   - Such as: how the environment changes a robot's orientation
2. Hiding detail in a *specification* of the system
   - We will see an example of this later (see the text)

Any other reasons why nondeterministic FSMs might be preferred over deterministic FSMs?

Behaviors and Traces

- FSM **behavior** is a sequence of (non-stuttering) steps.
- A **trace** is the record of inputs, states, and outputs in a behavior.
- A **computation tree** is a graphical representation of all possible traces.



FSMs are suitable for formal analysis. For example, **safety** analysis might show that some unsafe state is not reachable.

Size Matters

Non-deterministic FSMs are more compact than deterministic FSMs

- A classic result in automata theory shows that a nondeterministic FSM has a related deterministic FSM that is equivalent in a technical sense (language equivalence, covered in Chapter 13, for FSMs with finite-length executions).
- But the deterministic machine has, in the worst case, many more states (exponential in the number of states of the nondeterministic machine, see Appendix B).

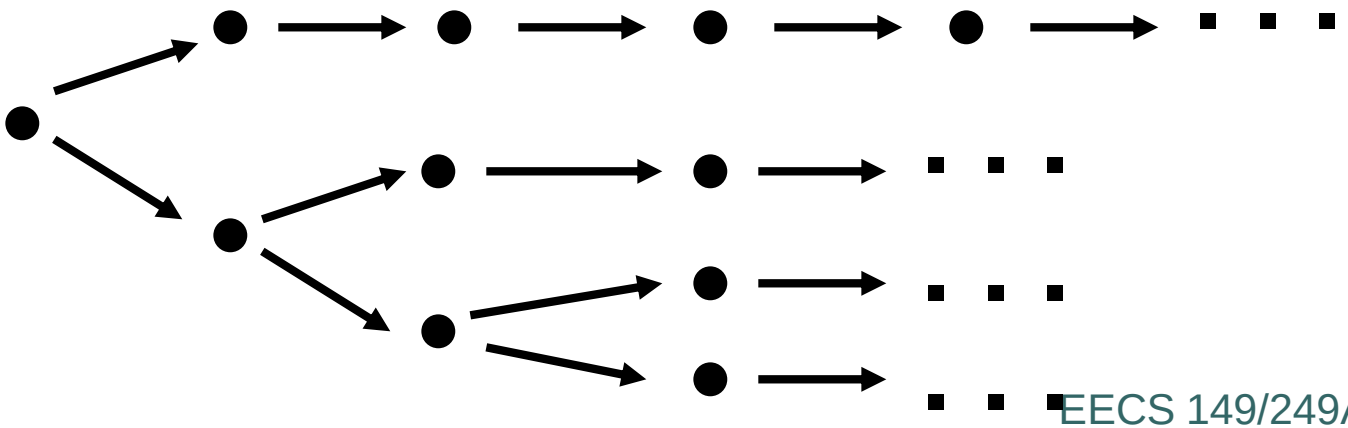
Non-deterministic Behavior: Tree of Computations

For a fixed input sequence:

- A deterministic system exhibits a single behavior
- A non-deterministic system exhibits a **set of behaviors**
  - visualized as a *computation tree*

Deterministic FSM behavior:

Non-deterministic FSM behavior:



Non-deterministic ≠ Probabilistic (Stochastic)

In a probabilistic FSM, each transition has an associated probability with which it is taken.

In a non-deterministic FSM, no such probability is known. We just know that any of the enabled transitions from a state can be taken.

Review: Concepts covered

Models = Programs

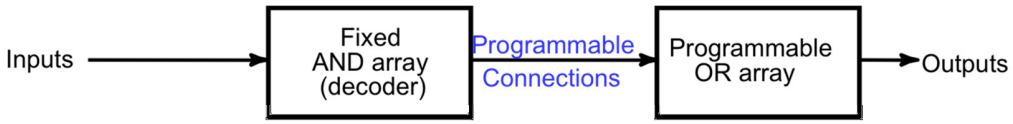
Actor Models of Discrete Systems: Types and Interfaces

States, Transitions, Guards

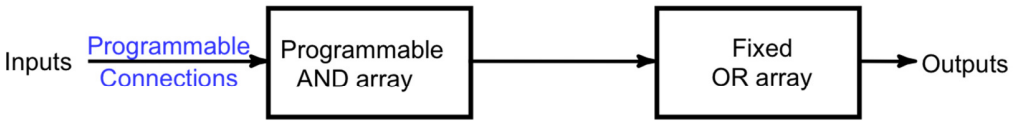
Determinism, Receptiveness, etc.

PROGRAMMABLE LOGIC DEVICES

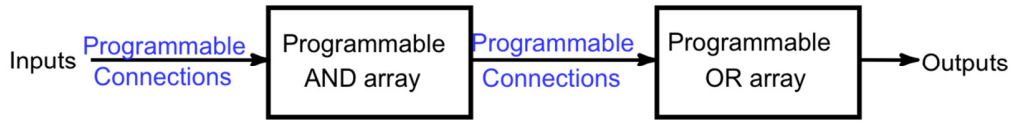
- ***Read Only Memory* (*ROM*)** - a fixed array of AND gates and a programmable array of OR gates
- ***Programmable Array Logic* (*PAL*)** - a programmable array of AND gates feeding a fixed array of OR gates.
- ***Programmable Logic Array* (*PLA*)** - a programmable array of AND gates feeding a programmable array of OR gates.
- ***Complex Programmable Logic Device* (*CPLD*) /*Field- Programmable Gate Array* (*FPGA*)** - complex enough to be called "architectures"



(a) Programmable read-only memory (PROM)



(b) Programmable array logic (PAL) device



(c) Programmable logic array (PLA) device

READ ONLY MEMORY

- Read Only Memories (ROM) or Programmable Read Only Memories (PROM) have:
  - N input lines,
  - M output lines, and
  - 2^N decoded minterms.
- Fixed AND array with 2^N outputs implementing all N-literal minterms.
- Programmable OR Array with M outputs lines to form up to M sum of minterm expressions.
- A program for a ROM or PROM is simply a multiple-output truth table
  - If a 1 entry, a connection is made to the corresponding minterm for the corresponding output
  - If a 0, no connection is made
- Can be viewed as a *memory* with the inputs as *addresses* of *data* (output values), hence ROM or PROM names!

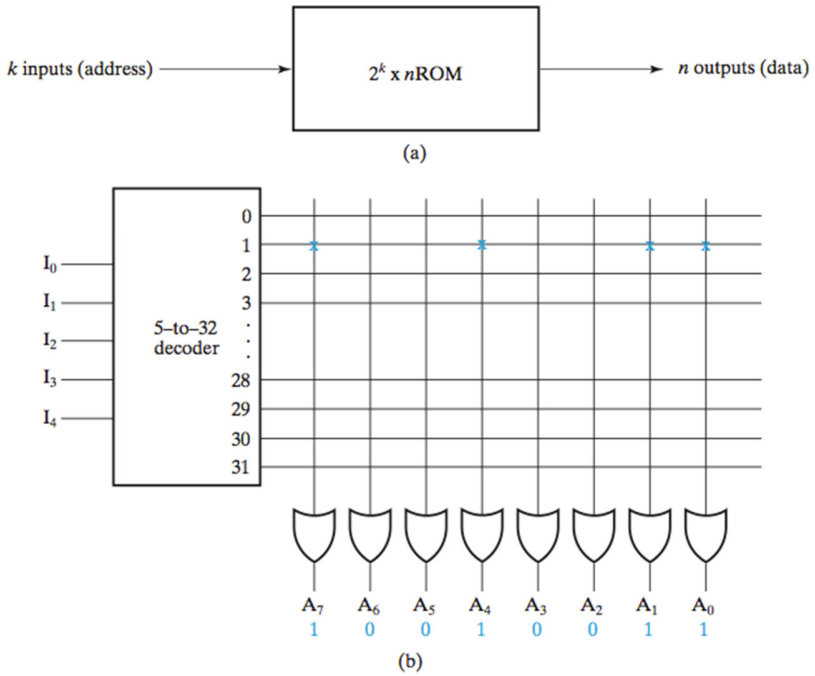
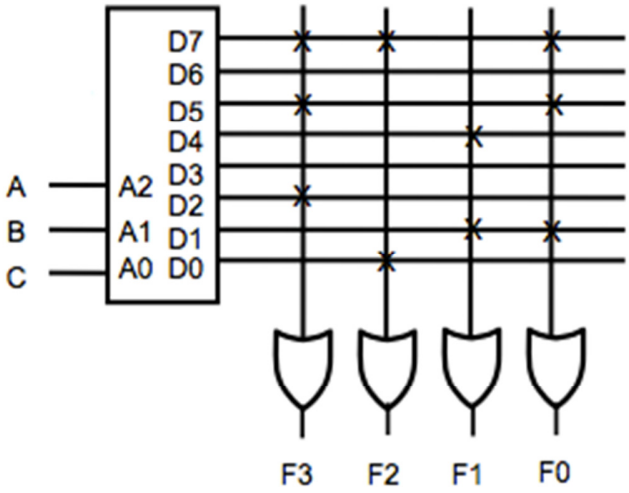


Figure: Block diagram and Internal Logic of a ROM

- Depending on the programming technology and approaches, read-only memories have different names
    1. ROM – mask programmed
    2. PROM – fuse or antifuse programmed
    3. EPROM – erasable floating gate programmed
    4. EEPROM or E^2PROM – electrically erasable floating gate programmed
    5. FLASH memory: electrically erasable floating gate with multiple erasure and programming modes.

- Example: A 8 X 4 ROM (N = 3 input lines, M= 4 output lines)
    - The fixed "AND" array is a “decoder” with 3 inputs and 8 outputs implementing minterms.
    - The programmable "OR“ array uses a single line to represent all inputs to an OR gate. An “X” in the array corresponds to attaching the minterm to the OR
    - Read Example: For input (A_2,A_1,A_0) = 011, output is (F_3,F_2,F_1,F_0) = 0011.
    - What are functions F_3, F_2 , F_1 and F_0 in terms of (A_2, A_1, A_0)?



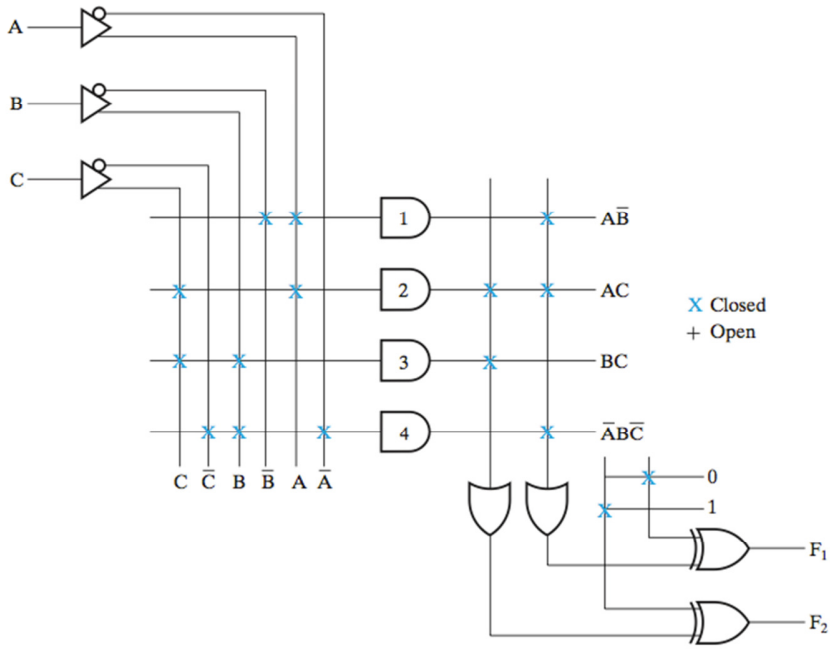
PROGRAMMABLE LOGIC ARRAY (PLA)

- Compared to a ROM and a PAL, a PLA is the most flexible having a programmable set of ANDs combined with a programmable set of ORs.
- Advantages
  - A PLA can have large N and M permitting implementation of equations that are impractical for a ROM (because of the number of inputs, N, required
  - A PLA has all of its product terms connectable to all outputs, overcoming the problem of the limited inputs to the PAL Ors
  - Some PLAs have outputs that can be complemented, adding POS functions
- Disadvantages
  - Often, the product term count limits the application of a PLA.
  - Two-level multiple-output optimization is required to reduce the number of product terms in an implementation, helping to fit it into a PLA.
  - Multi-level circuit capability available in PAL not available in PLA. PLA requires external connections to do multi-level circuits.

Programmable Logic Array Example

F1=AB' + AC + A'BC'

F2= (AC+BC)'

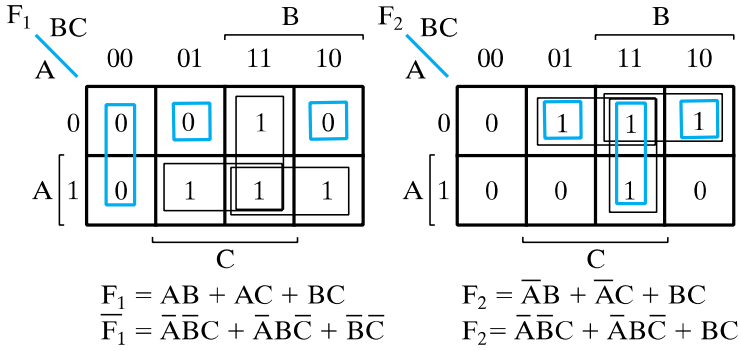


- What are the equations for F_1 and F_2?
- Could the PLA implement the functions without the XOR gates?
- 3-input, 3-output PLA with 4 product terms

Example 6-3 from Mano: Implementing a Combinational Circuit Using a PLA

F1(A,B,C)= Σm(3,5,6,7)

F2(A,B,C)= Σm(1,2,3,7)



$F_1 = AB + AC + BC$

$\overline{F_1} = \bar{A}\bar{B}C + \bar{A}B\bar{C} + B\bar{C}$

$F_2 = \bar{A}B + \bar{A}C + BC$

$F_2 = \bar{A}\bar{B}C + \bar{A}B\bar{C} + BC$

The solution is:

$$F_1 = \overline{\bar{A}\bar{B}C + \bar{A}B\bar{C} + B\bar{C}}$$

$$F_2 = \bar{A}\bar{B}C + \bar{A}B\bar{C} + BC$$

PROGRAMMABLE ARRAY LOGIC (PAL)

- The PAL is the opposite of the ROM, having a programmable set of ANDs combined with fixed ORs.
- Disadvantage
  - ROM guaranteed to implement any M functions of N inputs. PAL may have too few inputs to the OR gates.
- Advantages
  - For given internal complexity, a PAL can have larger N and M
  - Some PALs have outputs that can be complemented, adding POS functions
  - No multilevel circuit implementations in ROM (without external connections from output to input). PAL has outputs from OR terms as internal inputs to all AND terms, making implementation of multi-level circuits easier.

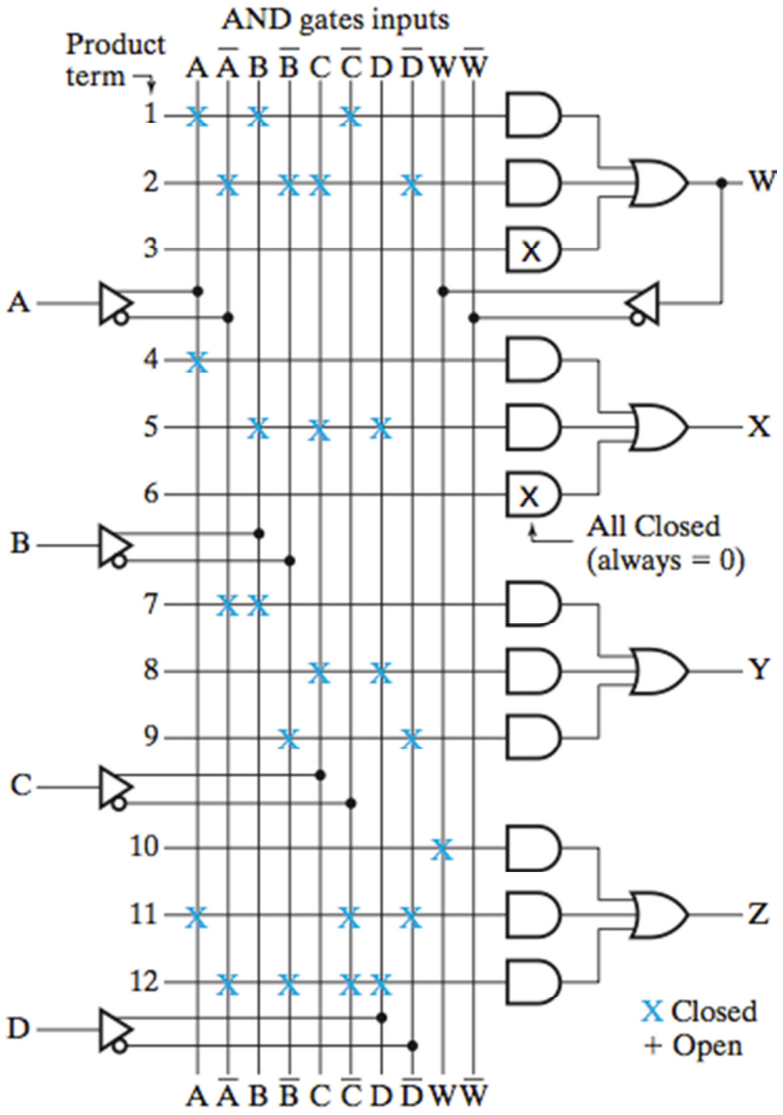
Programmable Array Logic Example

- 4-input, 3-output PAL with fixed, 3-input OR terms
- What are the equations for F1 through F4?

W(A,B,C,D) = Σm (2,12,13)
X(A,B,C,D) = Σm (7,8,9,10,11,12,13,14,15)
Y(A,B,C,D) = Σm (0,2,3,4,5,6,7,8,10,11,15)
Z(A,B,C,D) = Σm (1,2,8,12,13)

Simplifying the four function to a minimum number of terms results in the following Boolean functions

W= ABC'+A'B'CD'
X = A+BCD
Y = A'B+CD+B'D'
Z = ABC'+A'B'CD'+AC'D'+A'B'C'D = W+AC'D'+A'B'C'D


AND gates inputs
Product term
A Ā B B̄ C C̄ D D̄ W W̄
1
2
3
4
5
6
7
8
9
10
11
12
A
B
C
D
W
X
Y
Z
All Closed (always = 0)
X Closed
+ Open

Introduction to Embedded Systems

Edward A. Lee

UC Berkeley

EECS 149/249A

Fall 2016

Chapter 7: Sensors and Actuators

What is a sensor? An actuator?

A sensor is a device that measures a physical quantity

→ Input / “Read from physical world”

An actuator is a device that modifies a physical quantity

→ Output / “Write to physical world”

Sensors and Actuators – The Bridge between the Cyber and the Physical

Sensors:

- Cameras
- Accelerometers
- Gyroscopes
- Strain gauges
- Microphones
- Magnetometers
- Radar/Lidar
- Chemical sensors
- Pressure sensors
- Switches
- …

Actuators:

- Motor controllers
- Solenoids
- LEDs, lasers
- LCD and plasma displays
- Loudspeakers
- Switches
- Valves
- …

Modeling Issues:

- Physical dynamics
- Noise
- Bias
- Sampling
- Interactions
- Faults
- …

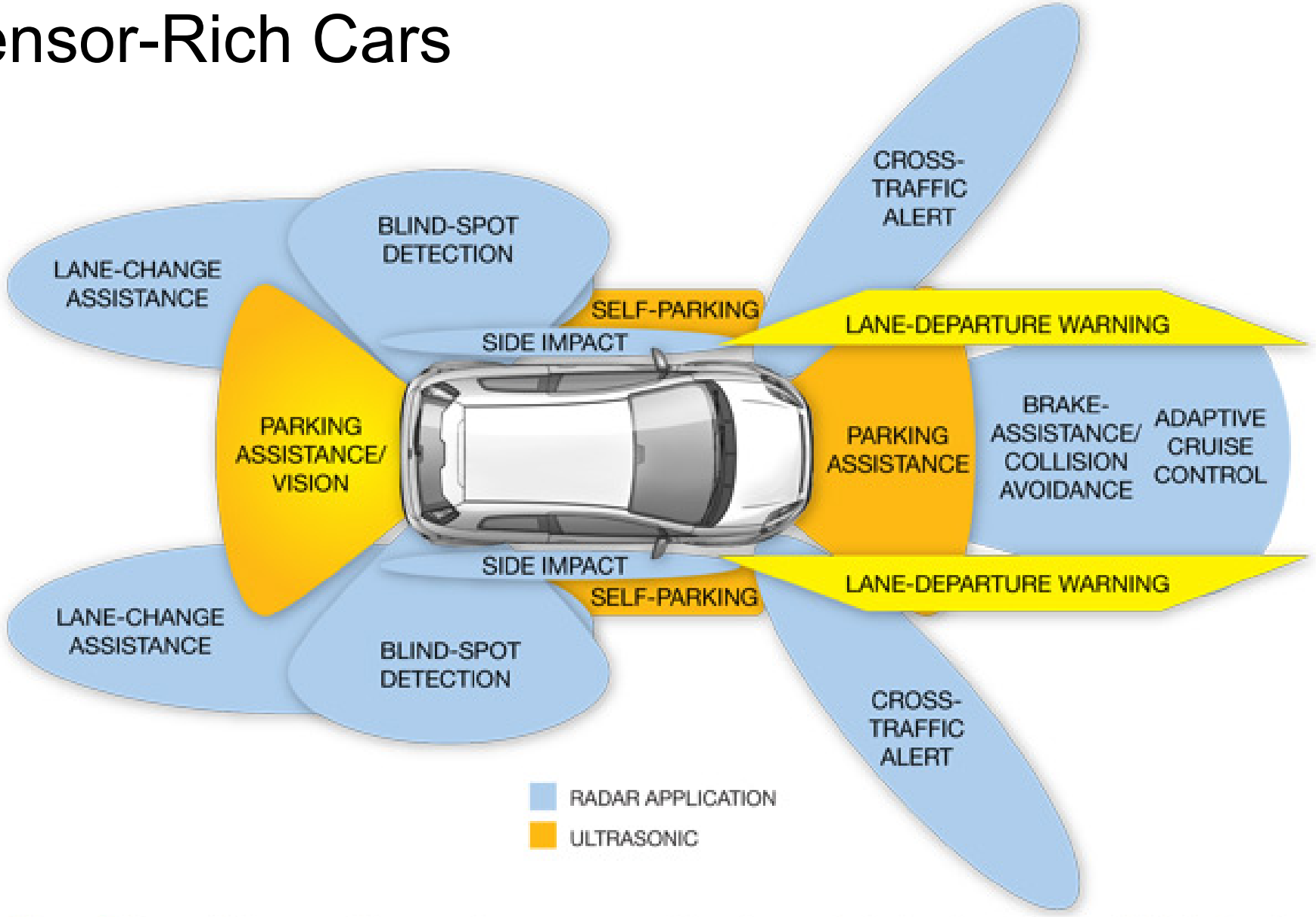
Self-Driving Cars

Berkeley PATH Project Demo,
1999, San Diego.

Google self-driving car 2.0

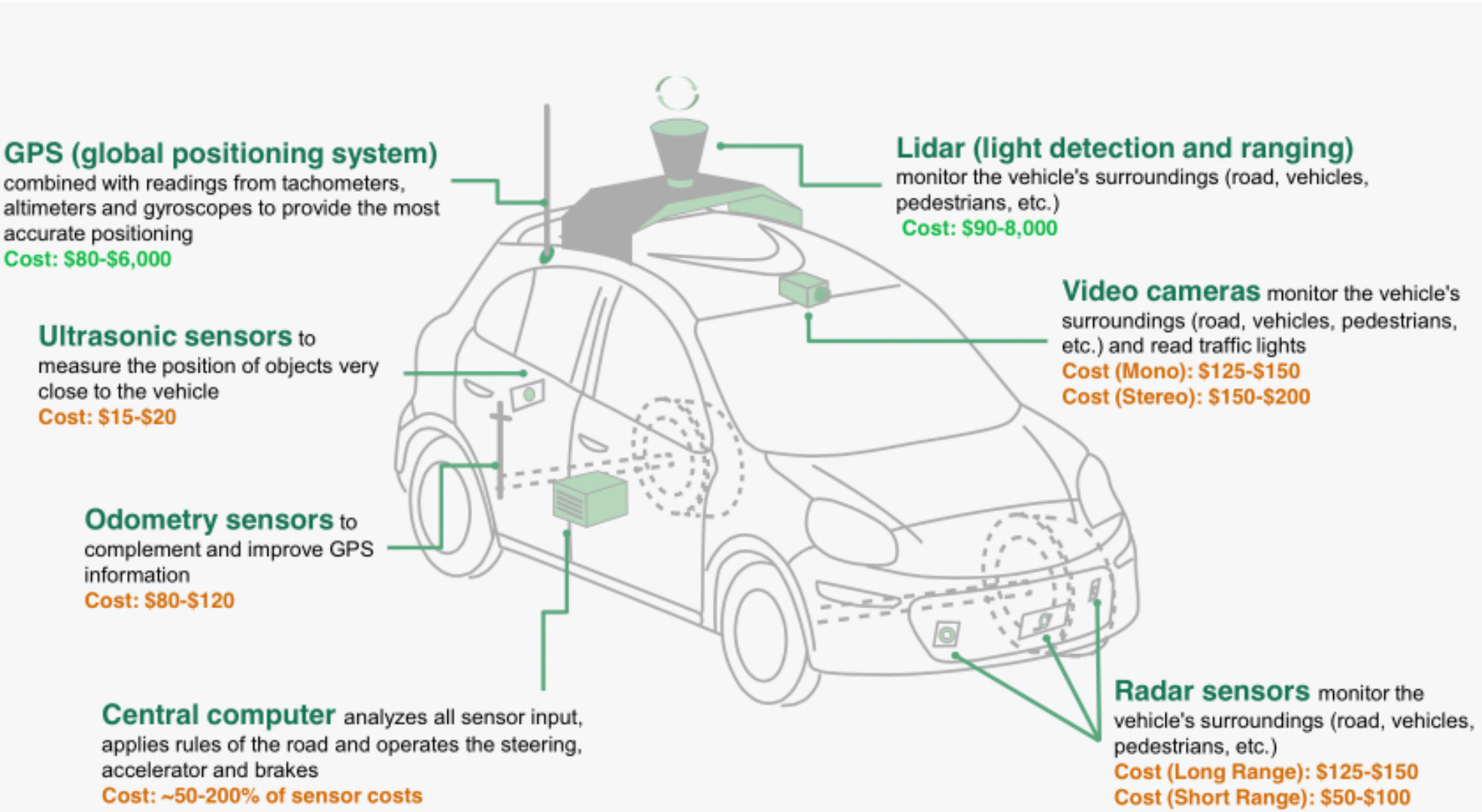
Kingvale Blower
Berkeley PATH Project, March, 2005

Sensor-Rich Cars



Source: Analog Devices

Sensor-Rich Cars



Source: Wired Magazine

Kingvale Blower: Technology Overview
Berkeley PATH Project, March, 2003

Magnetometers

A very common type is the Hall Effect magnetometer.

Charge particles (electrons, 1) flow through a conductor (2) serving as a Hall sensor. Magnets (3) induce a magnetic field (4) that causes the charged particles to accumulate on one side of the Hall sensor, inducing a measurable voltage difference from top to bottom.

The four drawings at the right illustrate electron paths under different current and magnetic field polarities.

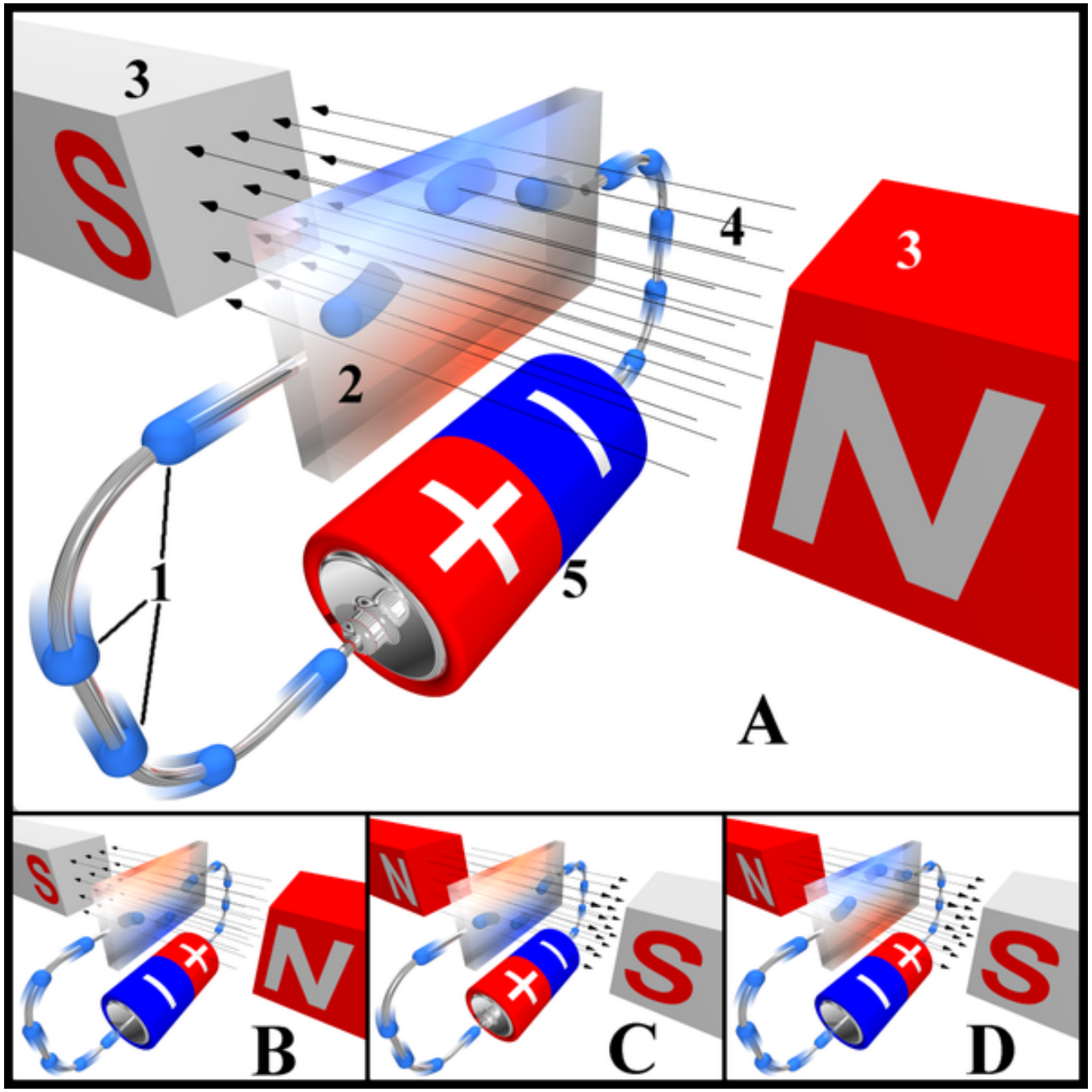


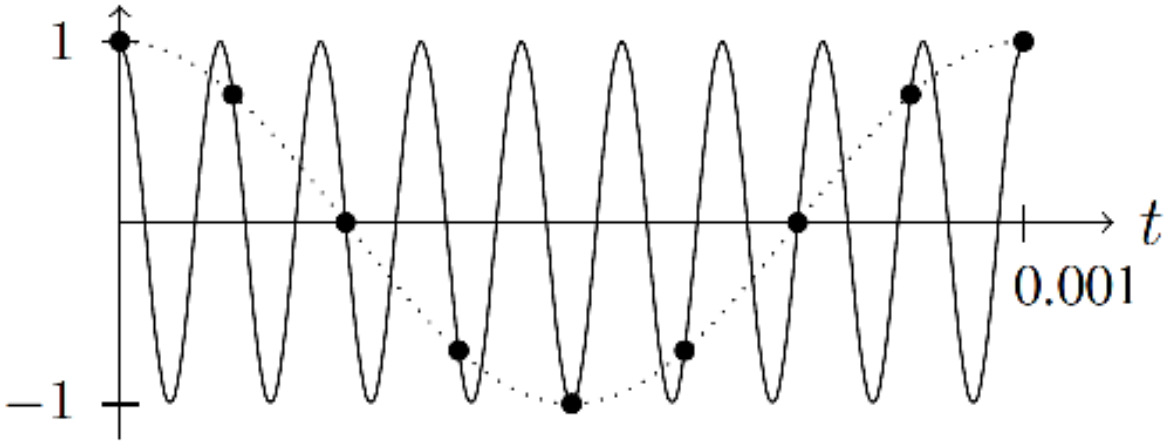
Image source: Wikipedia Commons

Edwin Hall discovered this effect in 1879.

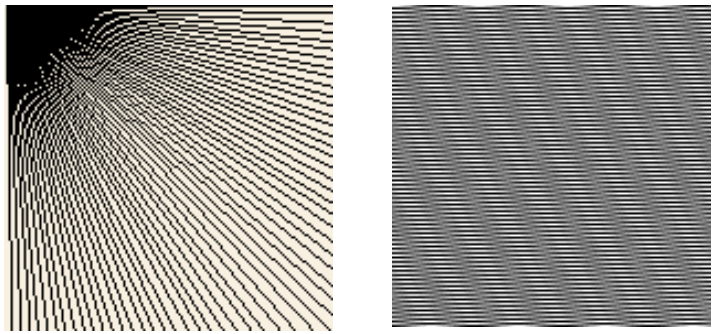
Aliasing

Sampled data is vulnerable to *aliasing*, where high frequency components masquerade as low frequency components.

Careful modeling of the signal sources and analog signal conditioning or digital oversampling are necessary to counter the effect.



A high frequency sinusoid sampled at a low rate looks just like a low frequency sinusoid.



Digitally sampled images are vulnerable to aliasing as well, where patterns and edges appear as a side effect of the sampling. Optical blurring of the image prior to sampling avoids aliasing, since blurring is spatial low-pass filtering.

Roadmap

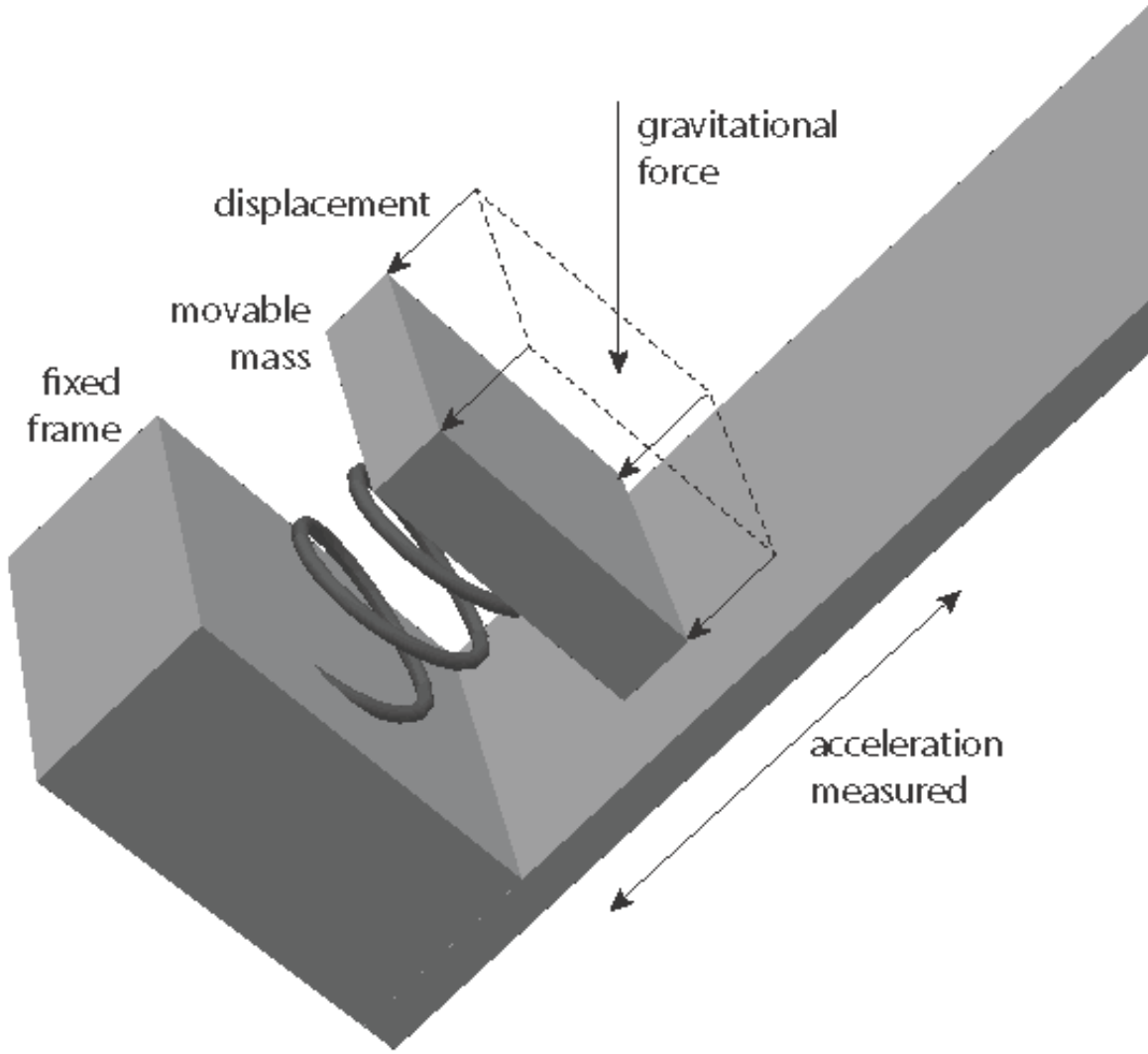
- How Accelerometers work
- Affine Model of Sensors
- Bias and Sensitivity
- Faults in Sensors
- Brief Overview of Actuators

Accelerometers

The most common design measures the distance between a plate fixed to the platform and one attached by a spring and damper. The measurement is typically done by measuring capacitance.

Uses:

- Navigation
- Orientation
- Drop detection
- Image stabilization
- Airbag systems

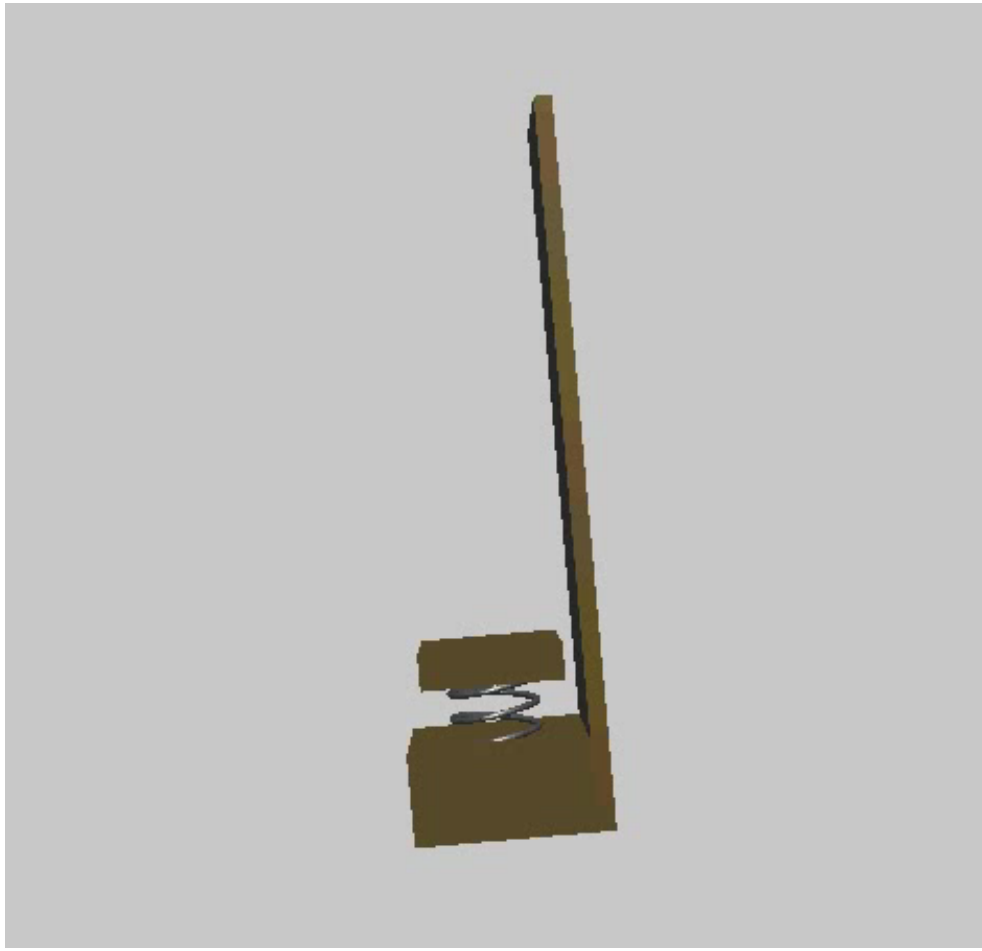


Spring-Mass-Damper Accelerometer

By Newton's second law, F=ma.

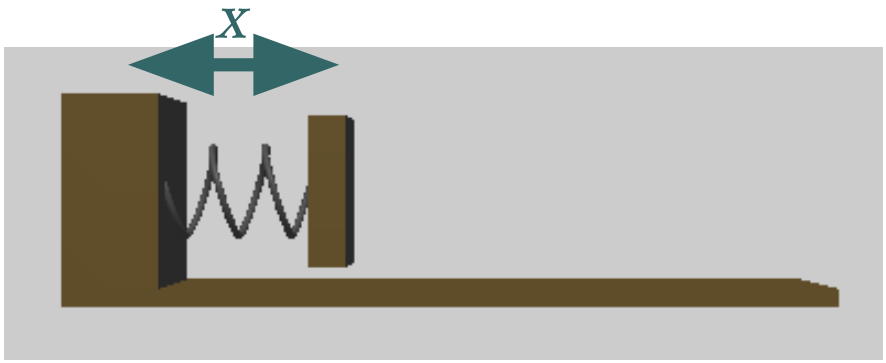
For example, F could be the Earth's gravitational force.

The force is balanced by the restoring force of the spring.



Spring-Mass-Damper System

- mass: M
- spring constant: k
- spring rest position: p
- position of mass: x
- viscous damping constant: c



Force due to spring extension:

$$F_1(t) = k(p - x(t))$$

Force due to viscous damping:

$$F_2(t) = -c\dot{x}(t)$$

Newton's second law:

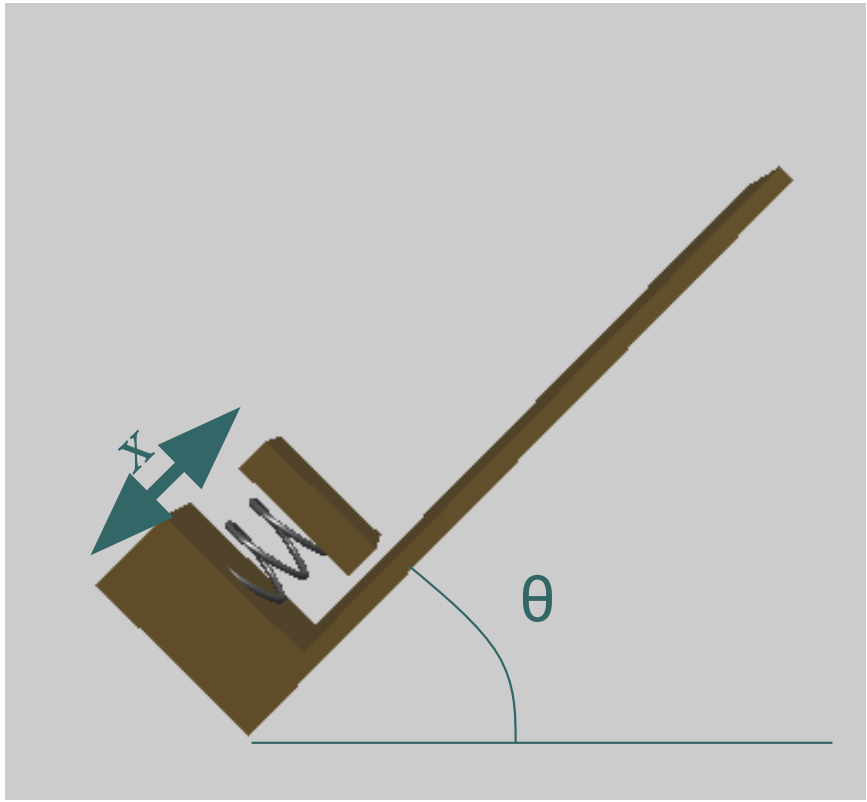
$$F_1(t) + F_2(t) = M\ddot{x}(t)$$

or

$$M\ddot{x}(t) + c\dot{x}(t) + kx(t) = kp.$$

Exercise: Convert to an integral equation with initial conditions.

Measuring tilt



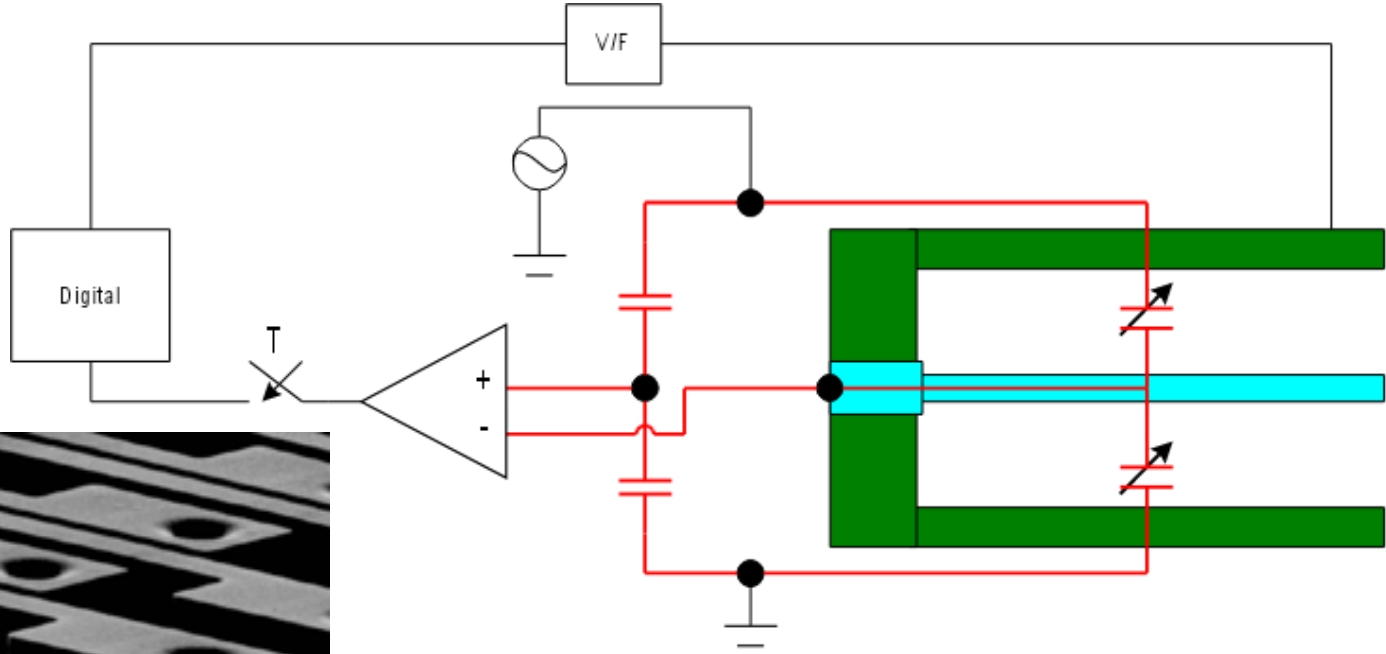
Component of gravitational force in the direction of the accelerometer axis must equal the spring force:

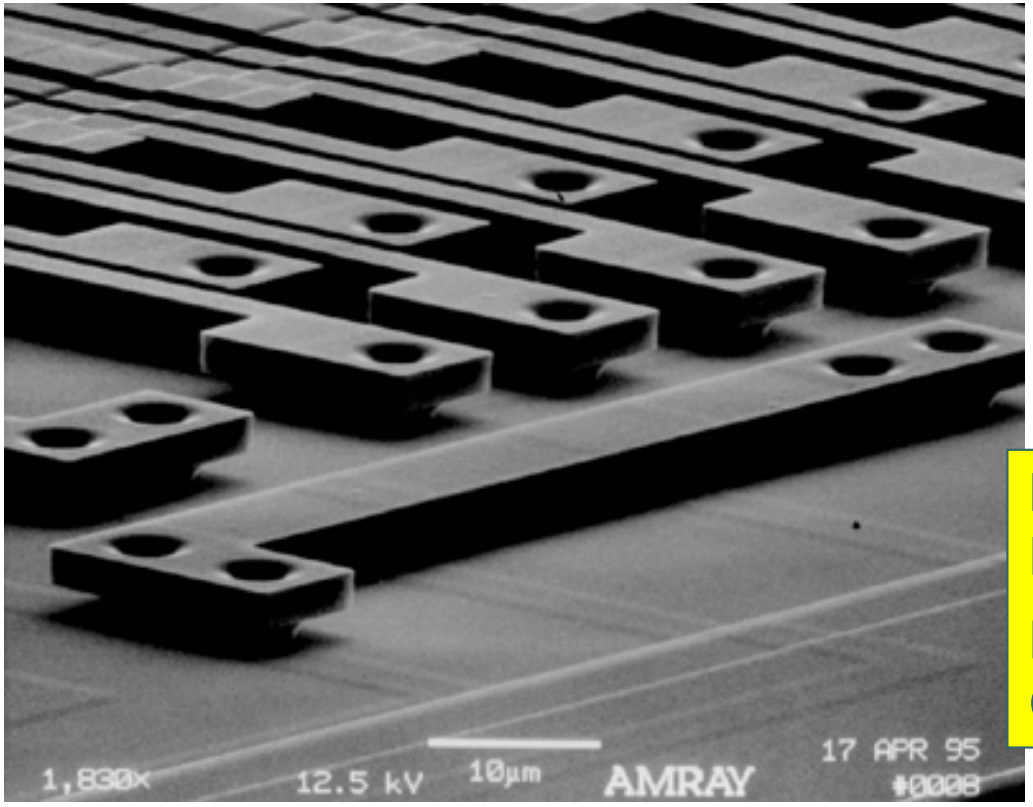
$$Mg\sin(\theta) = k(p - x(t))$$

Given a measurement of x, you can solve for θ, up to an ambiguity of π.

Feedback dramatically improves accuracy and dynamic range of microaccelerometers.

The Berkeley Sensor and Actuator Center (BSAC) created the first silicon microaccelerometers, MEMS devices now used in airbag systems, computer games, disk drives (drop sensors), etc.





M. A. Lemkin, "Micro Accelerometer Design with Digital Feedback Control",

Ph.D. dissertation, EECS, University of California, Berkeley, Fall 1997

Difficulties Using Accelerometers

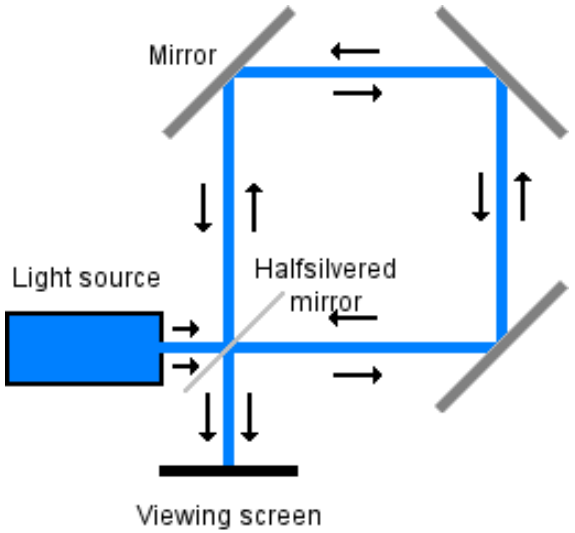
- Separating tilt from acceleration
- Vibration
- Nonlinearities in the spring or damper
- Integrating twice to get position: Drift

$$p(t) = p(0) + \int_0^t v(\tau) d\tau,$$

$$v(t) = v(0) + \int_0^t x(\tau) d\tau.$$

Position is the integral of velocity, which is the integral of acceleration. Bias in the measurement of acceleration causes position estimate error to increase quadraticly.

Measuring Changes in Orientation: Gyroscopes



Optical gyros: Leverage the Sagnac effect, where a laser light is sent around a loop in opposite directions and the interference is measured. When the loop is rotating, the distance the light travels in one direction is smaller than the distance in the other. This shows up as a change in the interference.

Images from the Wikipedia Commons

Inertial Navigation Systems

Dead reckoning plus GPS.

Combinations of:

- GPS (for initialization and periodic correction).
- Three axis gyroscope measures orientation.
- Three axis accelerometer, double integrated for position after correction for orientation.

Typical drift for systems used in aircraft have to be:

- 0.6 nautical miles per hour
- tenths of a degree per hour

Good enough? It depends on the application!

Design Issues with Sensors

- Calibration
  - Relating measurements to the physical phenomenon
  - Can dramatically increase manufacturing costs
- Nonlinearity
  - Measurements may not be proportional to physical phenomenon
  - Correction may be required
  - Feedback can be used to keep operating point in the linear region
- Sampling
  - Aliasing
  - Missed events
- Noise
  - Analog signal conditioning
  - Digital filtering
  - Introduces latency
- Failures
  - Redundancy (sensor fusion problem)
  - Attacks (e.g. Stuxnet attack)

Sensor Calibration

Affine Sensor Model

Bias and Sensitivity

Example: Look at ADXL330 accelerometer datasheet

Analog Devices ADXL330 Data Sheet

SPECIFICATIONS

T_A = 25°C, V_S = 3 V, C_X = C_Y = C_Z = 0.1 µF, acceleration = 0 *g*, unless otherwise noted. All minimum and maximum specifications are guaranteed. Typical specifications are not guaranteed.

Table 1.

Parameter	Conditions	Min	Typ	Max	Unit
SENSOR INPUT	Each axis				
Measurement Range		±3	±3.6		*g*
Nonlinearity	% of full scale		±0.3		%
Package Alignment Error			±1		Degrees
Inter-Axis Alignment Error			±0.1		Degrees
Cross Axis Sensitivity[1]			±1		%
SENSITIVITY (RATIOMETRIC)[2]	Each axis				
Sensitivity at X_{OUT}, Y_{OUT}, Z_{OUT}	V_S = 3 V	270	300	330	mV/*g*
Sensitivity Change Due to Temperature[3]	V_S = 3 V		±0.015		%/°C
ZERO *g* BIAS LEVEL (RATIOMETRIC)	Each axis				
0 *g* Voltage at X_{OUT}, Y_{OUT}, Z_{OUT}	V_S = 3 V	1.2	1.5	1.8	V
0 *g* Offset vs. Temperature			±1		mg/°C
NOISE PERFORMANCE					
Noise Density X_{OUT}, Y_{OUT}			280		µg/√Hz rms
Noise Density Z_{OUT}			350		µg/√Hz rms
FREQUENCY RESPONSE[4]					
Bandwidth X_{OUT}, Y_{OUT}[5]	No external filter		1600		Hz
Bandwidth Z_{OUT}[5]	No external filter		550		Hz
R_{FILT} Tolerance			32 ± 15%		kΩ
Sensor Resonant Frequency			5.5		kHz

Design Issues with Sensors

- Calibration
  - Relating measurements to the physical phenomenon
  - Can dramatically increase manufacturing costs
- Nonlinearity
  - Measurements may not be proportional to physical phenomenon
  - Correction may be required
  - Feedback can be used to keep operating point in the linear region
- Sampling
  - Aliasing
  - Missed events
- Noise
  - Analog signal conditioning
  - Digital filtering
  - Introduces latency
- Failures
  - Redundancy (sensor fusion problem)
  - Attacks (e.g. Stuxnet attack)

Faults in Sensors

Sensors are physical devices

Like all physical devices, they suffer wear and tear, and can have manufacturing defects

Cannot assume that *all* sensors on a system will work correctly at *all* times

Solution: Use redundancy

→ However, must be careful *how* you use it!

Violent Pitching of Qantas Flight 72 (VH-QPA)

An Airbus A330 en-route from Singapore to Perth on 7 October 2008

- Started pitching violently, unrestrained passengers hit the ceiling, 12 serious injuries, so counts it as an accident.
- Three Angle Of Attack (AOA) sensors, one on left (#1), two on right (#2, #3) of nose.
- Have to deal with inaccuracies, different positions, gusts/spikes, failures.

[Rushby, 2002]

A330 AOA Sensor Processing

- Sampled at 20Hz
- Compare each sensor to the median of the three
- If difference is larger than some threshold for more than 1 second, flag as faulty and ignore for remainder of flight
- Assuming all three are OK, use mean of #1 and #2 (because they are on different sides)
- If the difference between #1 or #2 and the median is larger than some (presumably smaller) threshold, use previous *average* value for 1.2 seconds
- Failure scenario: two spikes in #1, first shorter than 1 second, second still present 1.2 seconds after detection of first
- Result: flight control computers commanding a nose-down aircraft movement, which resulted in the aircraft pitching down to a maximum of about 8.5 degrees

[Rushby, 2002]

How to deal with Sensor Errors

Difficult Problem, still research to be done

Possible approach: Intelligent sensor communicates an **interval**, not a point value

- Width of interval indicates confidence, health of sensor

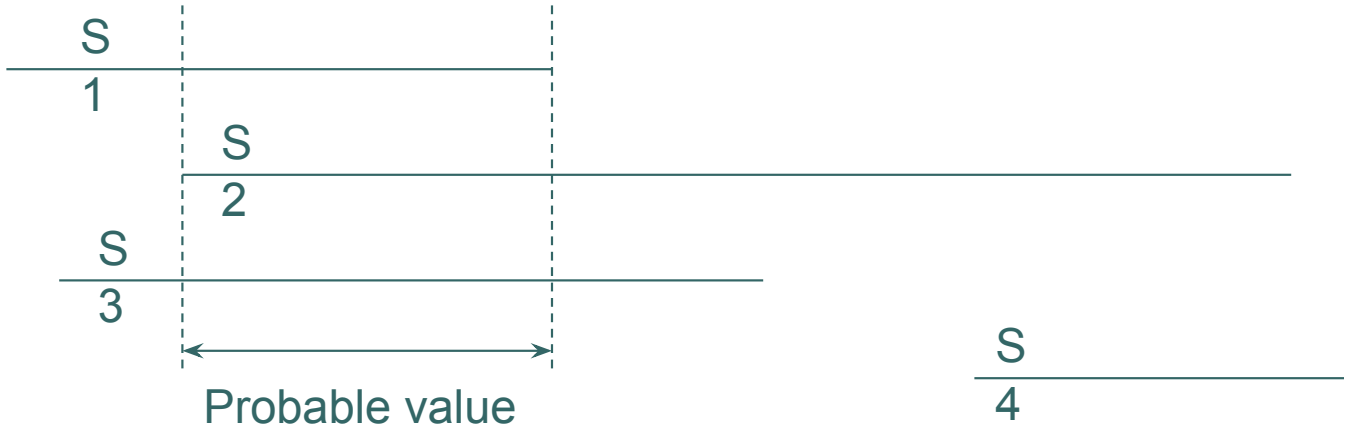
[Rushby, 2002]

Sensor Fusion: Marzullo's Algorithm

- Axiom: if sensor is non-faulty, its interval contains the true value
- Observation: true value must be in overlap of non-faulty intervals
- Consensus (fused) Interval to tolerate f faults in n: Choose interval that contains all overlaps of n − f; i.e., from least value contained in n − f intervals to largest value contained in n − f

[Rushby, 2002]

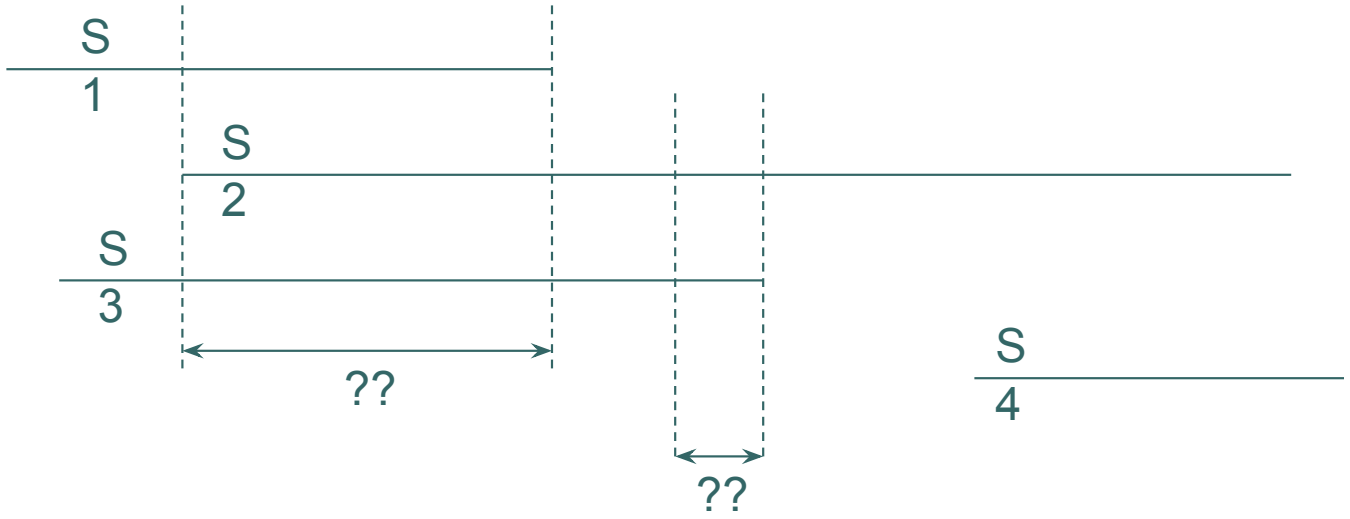
Example: Four sensors, at most one faulty



- Interval reports range of possible values.
- Of S1 and S4, one must be faulty.
- Of S3 and S4, one must be faulty.
- Therefore, S4 is faulty.
- Sound estimate is the overlap of the remaining three.

[Rushby, 2002]

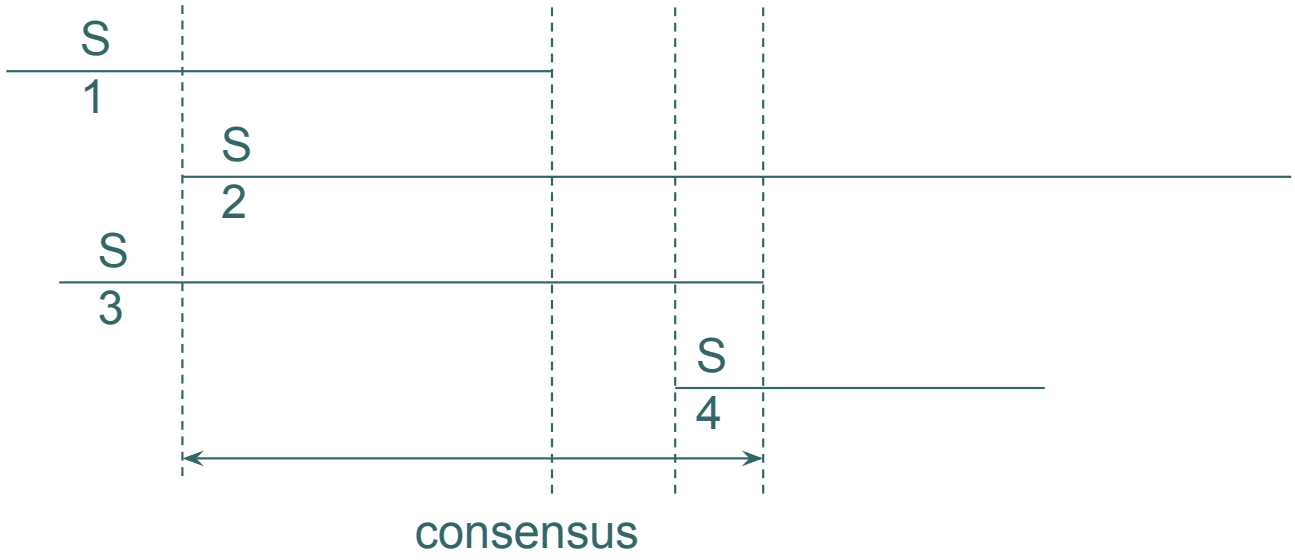
Example: Four sensors, at most one faulty



- Suppose S4's reading moves to the left
- Which interval should we pick?

[Rushby, 2002]

Example: Four sensors, at most one faulty



- Marzullo's algorithm picks the smallest interval that is sure to contain the true value, under the assumption that at most one sensor failed.
- But this yields big discontinuities. Jumps!

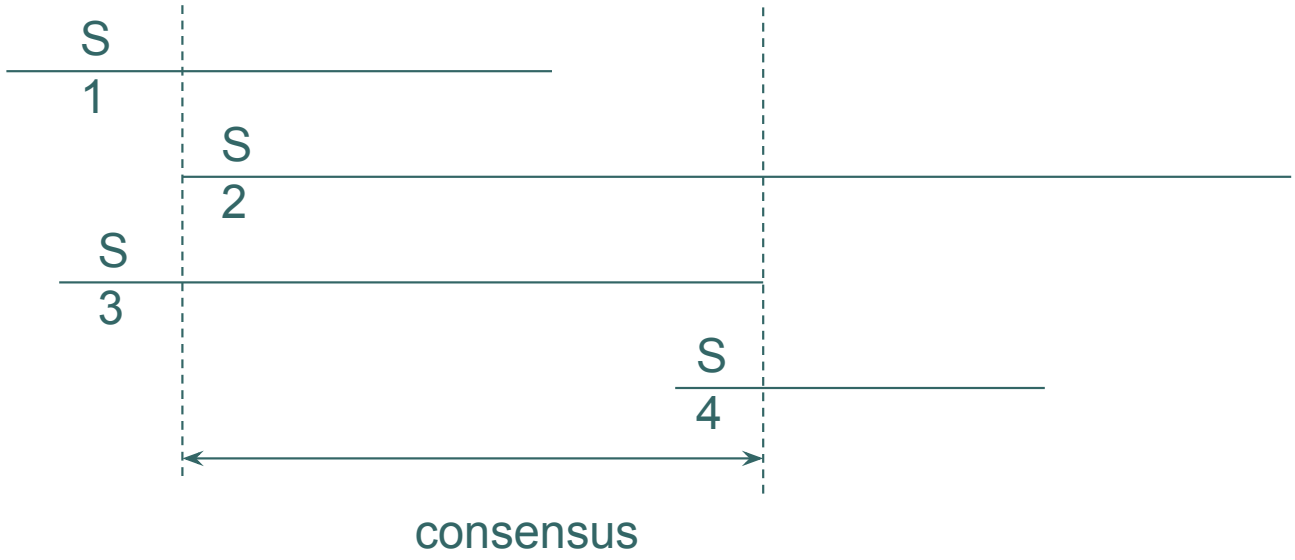
[Rushby, 2002]

Schmid and Schossmaier's Fusion Method

- Recall: n sensors, at most f faulty
- Choose interval from f+1st largest lower bound to f+1st smallest upper bound
- Optimal among selections that satisfy continuity conditions.

[Rushby, 2002]

Example: Four sensors, at most one faulty

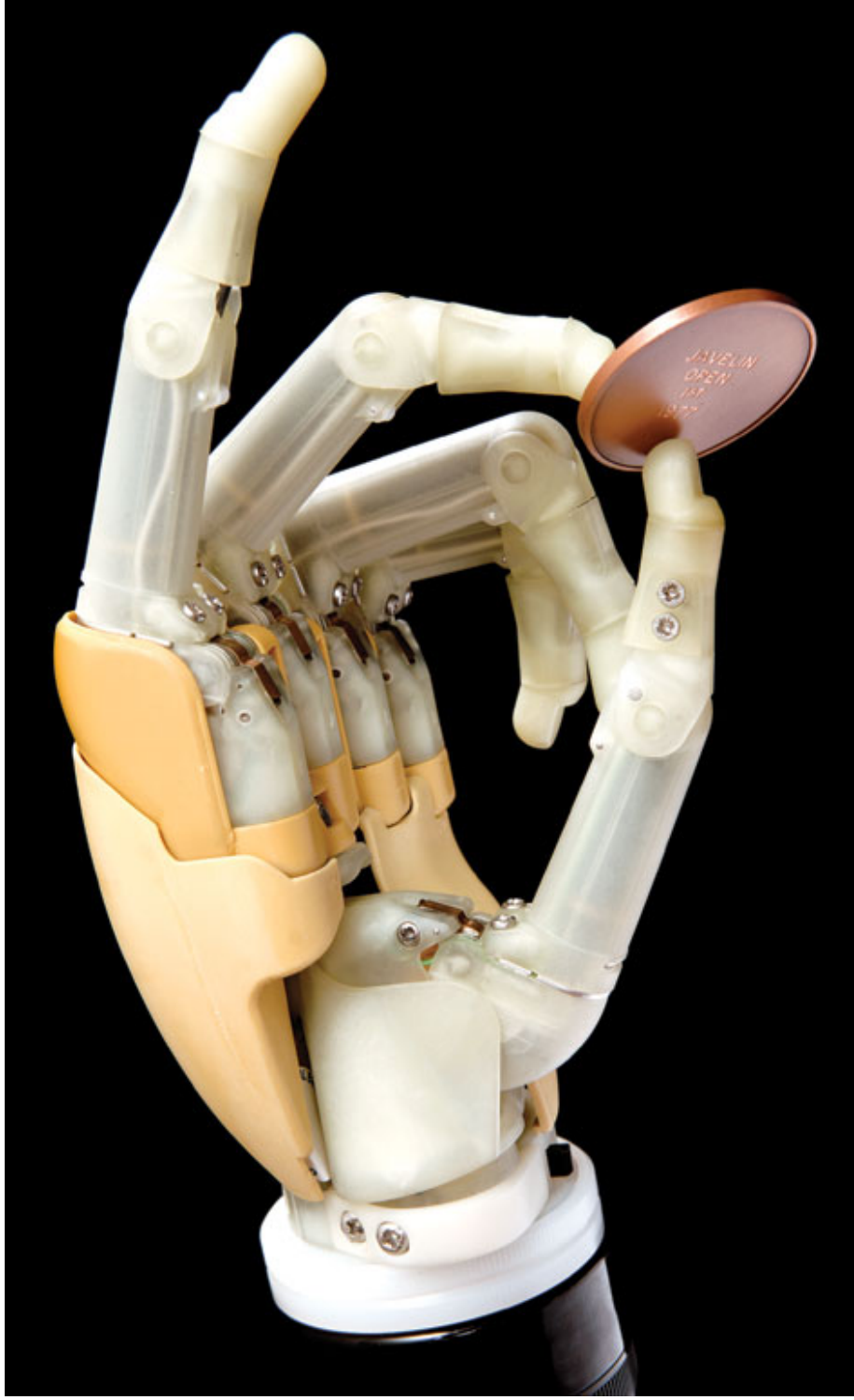


- Assuming at most one faulty, Schmid and Schossmaier's method choose the interval between:
  - Second largest lower bound
  - Second smallest upper bound
  - This preserves continuity, but not soundness

[Rushby, 2002]

Motor Controllers

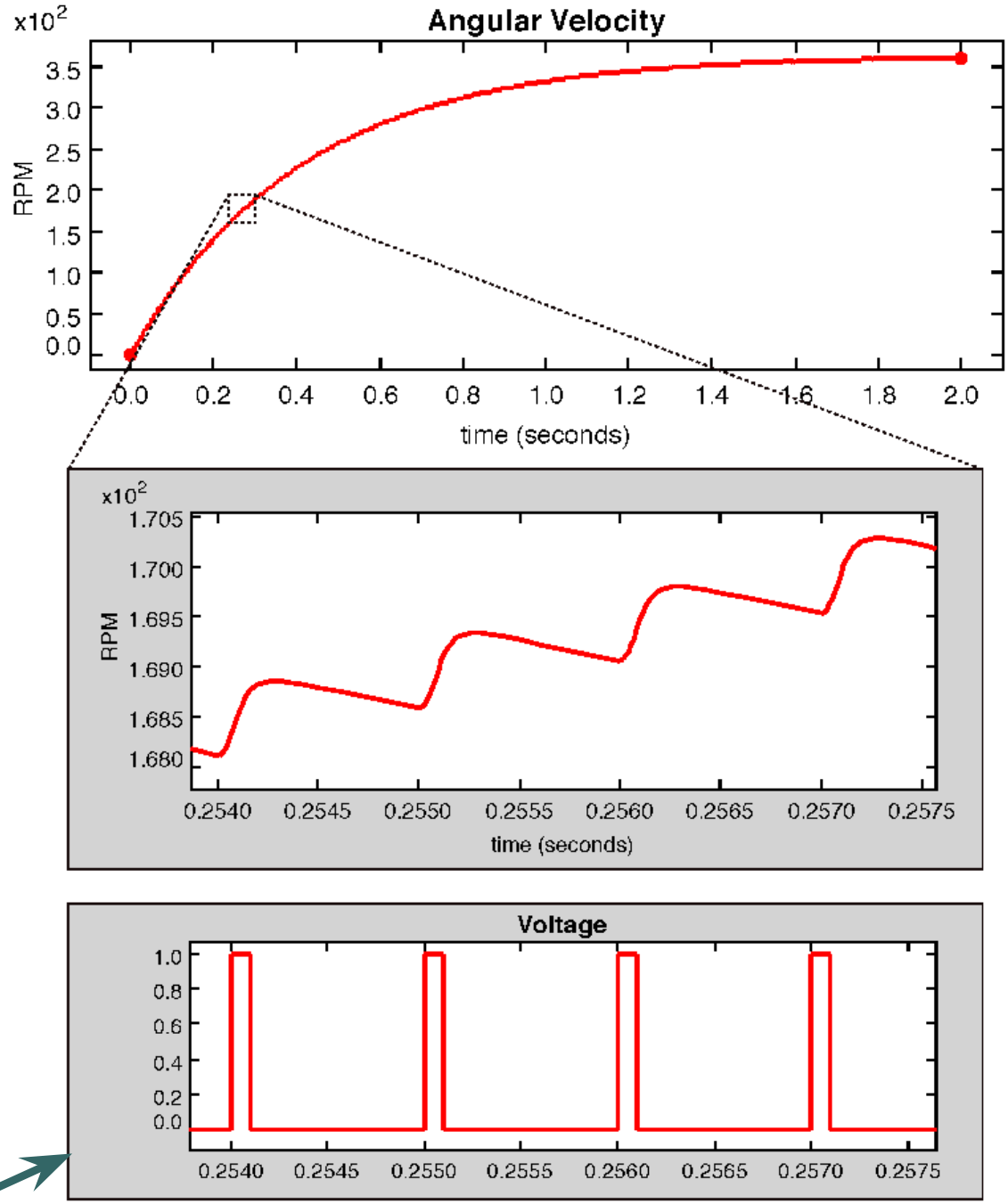
Bionic hand from Touch Bionics costs $18,500, has and five DC motors, can grab a paper cup without crushing it, and turn a key in a lock. It is controlled by nerve impulses of the user's arm, combined with autonomous control to adapt to the shape of whatever it is grasping. Source: IEEE Spectrum, Oct. 2007.



Pulse-Width Modulation (PWM)

Delivering power to actuators can be challenging. If the device tolerates rapid on-off controls ("bang-bang" control), then delivering power becomes much easier.

Duty cycle around 10%



Model of a Motor

Electrical Model:

$$v(t) = Ri(t) + L\frac{di(t)}{dt} + k_b\omega(t)$$

Back electromagnetic force constant

Angular velocity

Mechanical Model (angular version of Newton's second law):

$$I\frac{d\omega(t)}{dt} = k_T i(t) - \eta\omega(t) - \tau(t)$$

Moment of inertia

Torque constant

Friction

Load torque

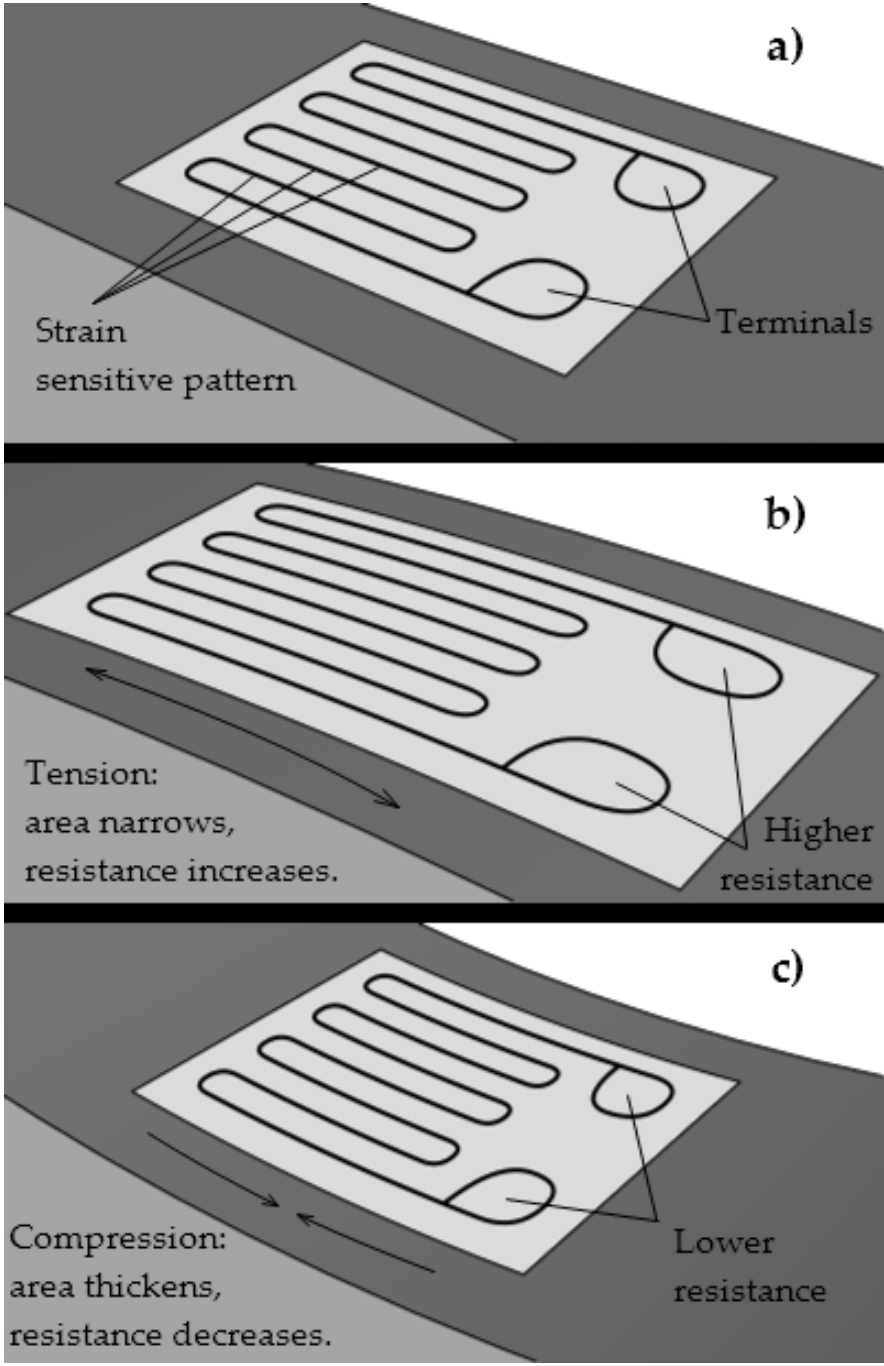
Summary for Lecture

- Overview of Sensors and Actuators
- How Accelerometers work
- Affine Model of Sensors
- Bias and Sensitivity
- Faults in Sensors
- Brief Overview of Actuators

Extra Slides Follow

Strain Gauges

Mechanical strain gauge used to measure the growth of a crack in a masonry foundation. This one is installed on the Hudson-Athens Lighthouse. Photo by Roy Smith, used with permission.



Images from the Wikipedia Commons

Noise & Signal Conditioning

Parsevals theorem relates the energy or the power in a signal in the time and frequency domains. For a finite energy signal x, the energy is

$$\int_{-\infty}^{\infty} (x(t))^2 dt = \frac{1}{2\pi} \int_{-\infty}^{\infty} |X(\omega)|^2 d\omega$$

where X is the Fourier transform. If there is a desired part x_d and an undesired part (noise) x_n,

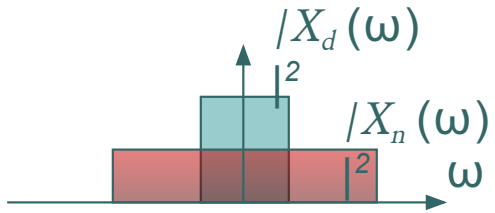
$$x(t) = x_d(t) + x_n(t)$$

then

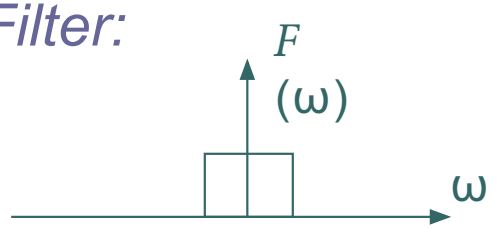
$$X(\omega) = X_d(\omega) + X_n(\omega)$$

Suppose that x_d is a narrowband signal and x_n is a broadband signal. Then the *signal to noise ratio* (SNR) can be greatly improved with filtering.

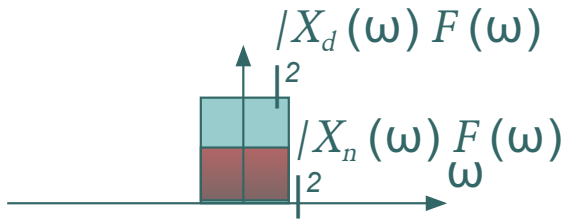
Example:



Filter:



Filtered signal:



A full treatement of this requires random processes.

References

John Rushby, “Formal Verification of Marzullo’s Sensor Fusion Interval,” CSL Technical Report, January 2002, SRI International, Menlo Park, CA.

http://www.csl.sri.com/users/rushby/papers/sensors.pdf

Embedded Processors

Ch8

Dheya Mustafa

Microprocessor

- A microprocessor is a single VLSI chip having a CPU. In addition, it may also have other units such as caches, floating point processing arithmetic unit, and pipelining units that help in faster processing of instructions.
- Earlier generation microprocessors' fetch-and-execute cycle was guided by a clock frequency of order of ~1 MHz. Processors now operate at a clock frequency of 2GHz

microcontroller

- A microcontroller (µC) is a small computer on a single integrated circuit consisting of a relatively simple central processing unit (CPU) combined with peripheral devices such as memories, I/O devices, and timers.
- Microcontrollers are particularly used in embedded systems for real-time control applications with on-chip program memory and devices.

Microprocessor	Microcontroller
Microprocessors are multitasking in nature. Can perform multiple tasks at a time. For example, on computer we can play music while writing text in text editor.	Single task oriented. For example, a washing machine is designed for washing clothes only.
RAM, ROM, I/O Ports, and Timers can be added externally and can vary in numbers.	RAM, ROM, I/O Ports, and Timers cannot be added externally. These components are to be embedded together on a chip and are fixed in numbers.
Designers can decide the number of memory or I/O ports needed.	Fixed number for memory or I/O makes a microcontroller ideal for a limited but specific task.
External support of external memory and I/O ports makes a microprocessor-based system heavier and costlier.	Microcontrollers are lightweight and cheaper than a microprocessor.
External devices require more space and their power consumption is higher.	A microcontroller-based system consumes less power and takes less space.

DSP Processors

- Processors designed specifically to support numerically intensive signal processing applications
- DSP processors normally add an extra stage or two that performs a multiplication, provide separate ALUs for address calculation, and provide a dual data memory for simultaneous access to two operands (this latter design is known as a Harvard architecture).
- Multipy-add instruction ax+b

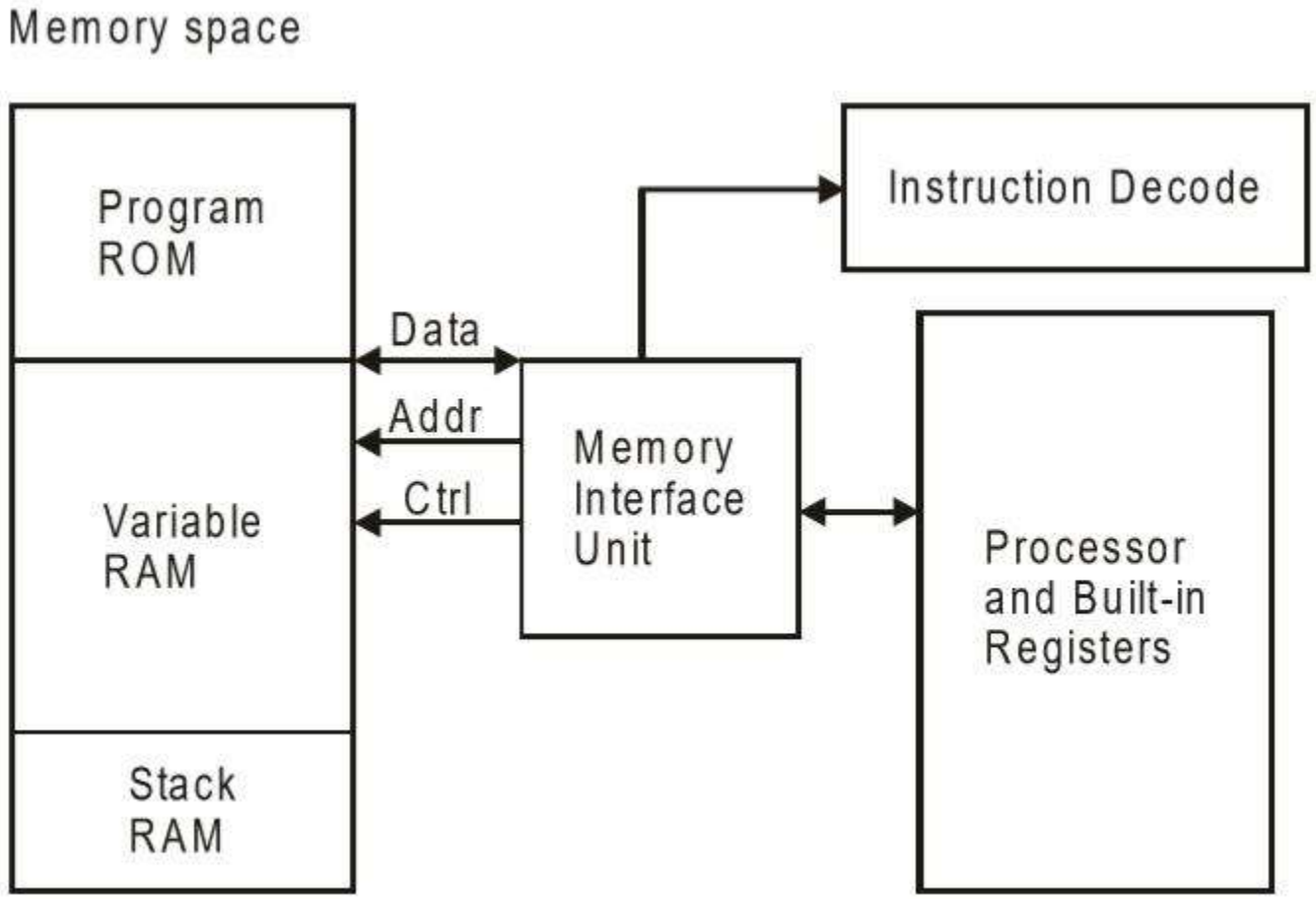
superscalar

- The DSP processors handle the radio, speech, and media processing (audio, images, and video). The other processors handle the user interface, database functions, networking, and downloadable applications. Specifically, the OMAP4440 includes a 1 GHz dual-core ARM Cortex processor, a c64x DSP, a GPU, and an image signal processor

GPUS

- A graphics processing unit (GPU) is a specialized processor designed especially to perform the calculations required in graphics rendering
- Apply same .RGB pixel color: each is one byte bit 64operation on several byte to utilize Vector processor MMX .datapath

Von Neumann Architecture



Harvard Architecture

- The Harvard architecture offers separate storage and signal buses for instructions and data. This architecture has data storage entirely contained within the CPU, and there is no access to the instruction storage as data. Computers have separate memory areas for program instructions and data using internal data buses, allowing simultaneous access to both instructions and data.

Von-Neumann Architecture	Harvard Architecture
Single memory to be shared by both code and data.	Separate memories for code and data.
Processor needs to fetch code in a separate clock cycle and data in another clock cycle. So it requires two clock cycles.	Single clock cycle is sufficient, as separate buses are used to access code and data.
Higher speed, thus less time consuming.	Slower in speed, thus more time-consuming.
Simple in design.	Complex in design.

CISC	RISC
Larger set of instructions. Easy to program	Smaller set of Instructions. Difficult to program.
Simpler design of compiler, considering larger set of instructions.	Complex design of compiler.
Many addressing modes causing complex instruction formats.	Few addressing modes, fix instruction format.
Instruction length is variable.	Instruction length varies.
Higher clock cycles per second.	Low clock cycle per second.
Emphasis is on hardware.	Emphasis is on software.
Control unit implements large instruction set using micro-program unit.	Each instruction is to be executed by hardware.
Slower execution, as instructions are to be read from memory and decoded by the decoder unit.	Faster execution, as each instruction is to be executed by hardware.
Pipelining is not possible.	Pipelining of instructions is possible, considering single clock cycle.

Introduction to Embedded Systems

Edward A. Lee

UC Berkeley

EECS 149/249A

Fall 2016

Chapter 9: Memory Architectures

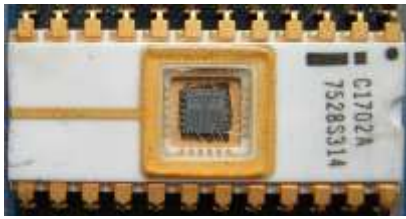
Memory Architecture: Issues

These issues loom larger in embedded systems than in general-purpose computing.

- Types of memory
  - volatile vs. non-volatile, SRAM vs. DRAM
- Memory maps
  - Harvard architecture
  - Memory-mapped I/O
- Memory organization
  - statically allocated
  - stacks
  - heaps (allocation, fragmentation, garbage collection)
- The memory model of C
- Memory hierarchies
  - scratchpads, caches, virtual memory)
- Memory protection
  - segmented spaces

Non-Volatile Memory

Preserves contents when power is off



- EPROM: erasable programmable read only memory
  - Invented by Dov Frohman of Intel in 1971
  - Erase by exposing the chip to strong UV light
- EEPROM: electrically erasable programmable read-only memory
  - Invented by George Perlegos at Intel in 1978
- Flash memory
  - Invented by Dr. Fujio Masuoka at Toshiba around 1980
  - Erased a “block” at a time
  - Limited number of program/erase cycles (~ 100,000)
  - Controllers can get quite complex
- Disk drives
  - Not as well suited for embedded systems

USB Drive

Images from the Wikimedia Commons

Volatile Memory

Loses contents when power is off.

- SRAM: static random-access memory
  - Fast, deterministic access time
  - But more power hungry and less dense than DRAM
  - Used for caches, scratchpads, and small embedded memories
- DRAM: dynamic random-access memory
  - Slower than SRAM
  - Access time depends on the sequence of addresses
  - Denser than SRAM (higher capacity)
  - Requires periodic refresh (typically every 64msec)
  - Typically used for main memory
- Boot loader
  - On power up, transfers data from non-volatile to volatile memory.

Example:

Die of a STM32F103VGT6 ARM Cortex-M3 microcontroller with 1 megabyte flash memory by STMicroelectronics.

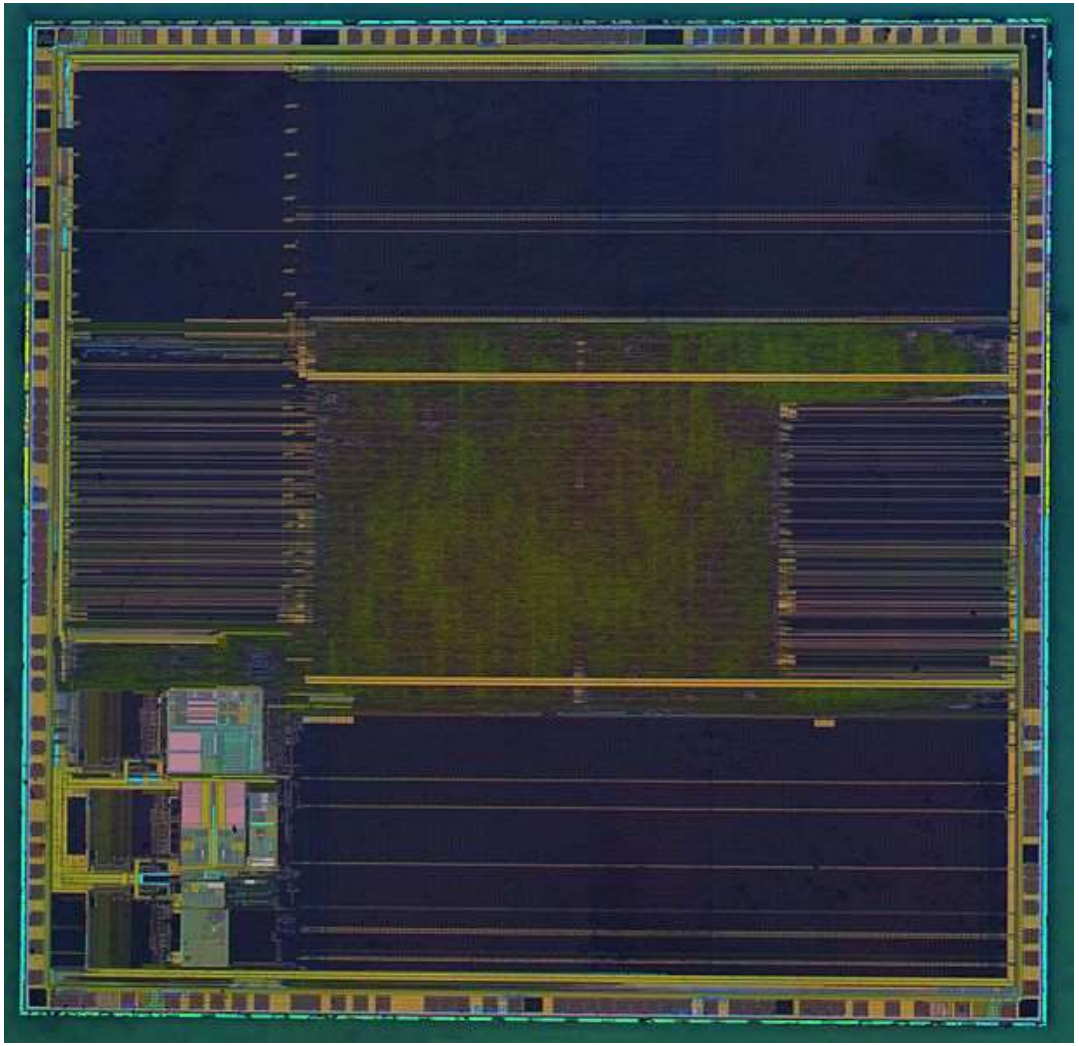
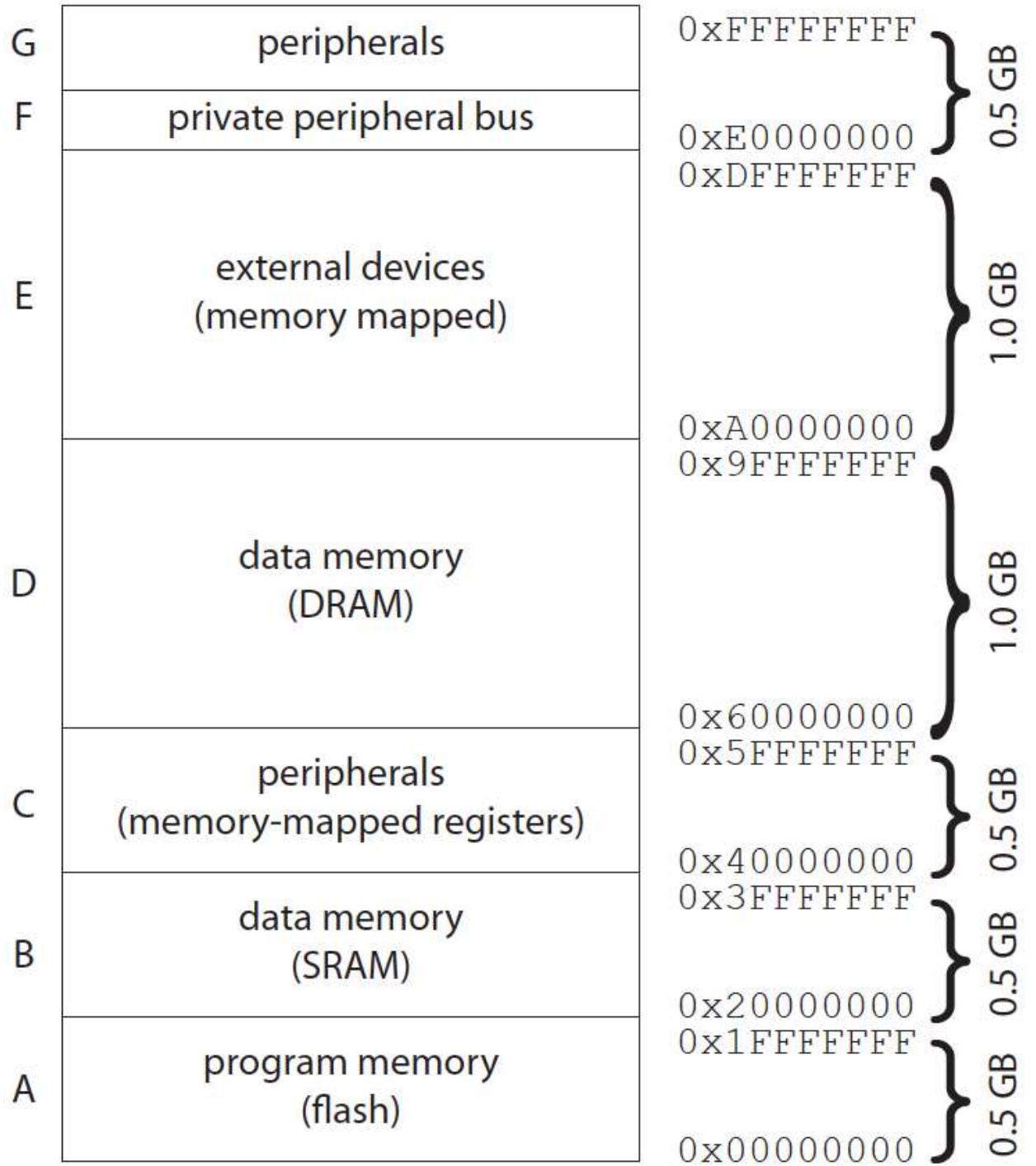


Image from Wikimedia Commons

Memory Map of an ARM Cortex™ - M3 architecture

Defines the mapping of addresses to physical memory.

Note that this does not define how much physical memory there is!



Another Example: AVR

The AVR is an 8-bit single chip microcontroller first developed by Atmel in 1997. The AVR was one of the first microcontroller families to use on-chip flash memory for program storage. It has a modified Harvard architecture.[1]

AVR was conceived by two students at the Norwegian Institute of Technology (NTH) Alf-Egil Bogen and Vegard Wollan, who approached Atmel in Silicon Valley to produce it.

[1] A Harvard architecture uses separate memory spaces for program and data. It originated with the Harvard Mark I relay-based computer (used during World War II), which stored the program on punched tape (24 bits wide) and the data in electro-mechanical counters.

A Use of AVR: Arduino

Arduino is a family of open-source hardware boards built around either 8-bit AVR processors or 32-bit ARM processors.

Example:
Atmel AVR
Atmega328
28-pin DIP on an
Arduino Duemilanove
board

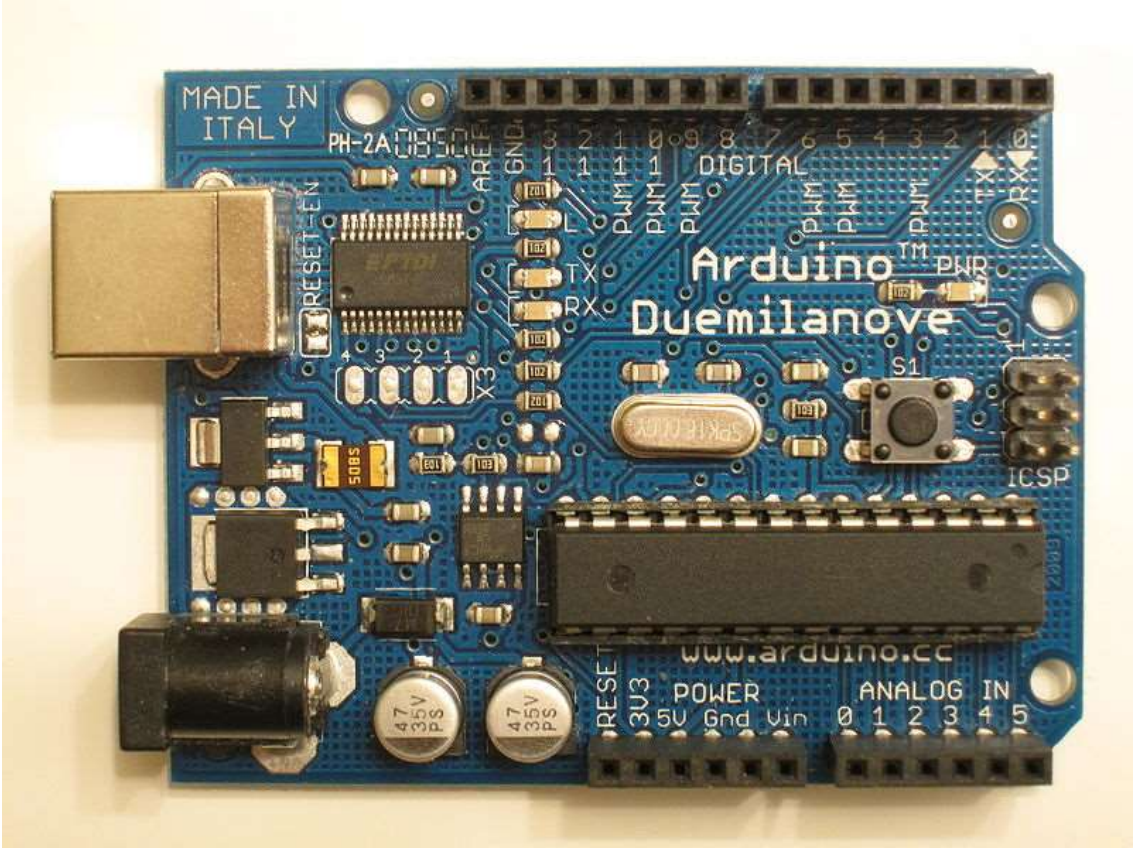


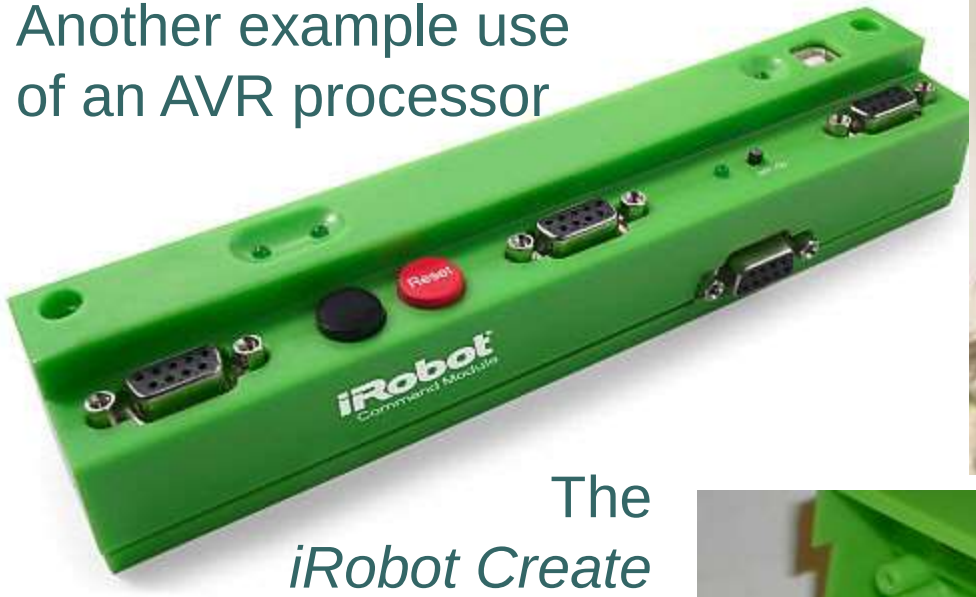
Image from Wikimedia Commons

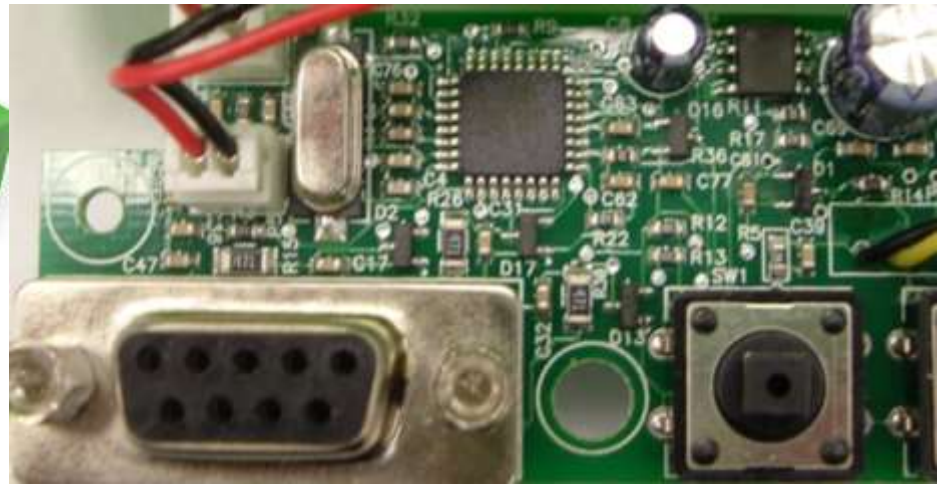
Open-Source Hardware and the maker movement

Massimo Banzi, founder of the Arduino project at Ivrea, Italy, and Limor Fried, owner and founder of Adafruit, showing one of the first board Arduino Uno from the production lines of Adafruit.

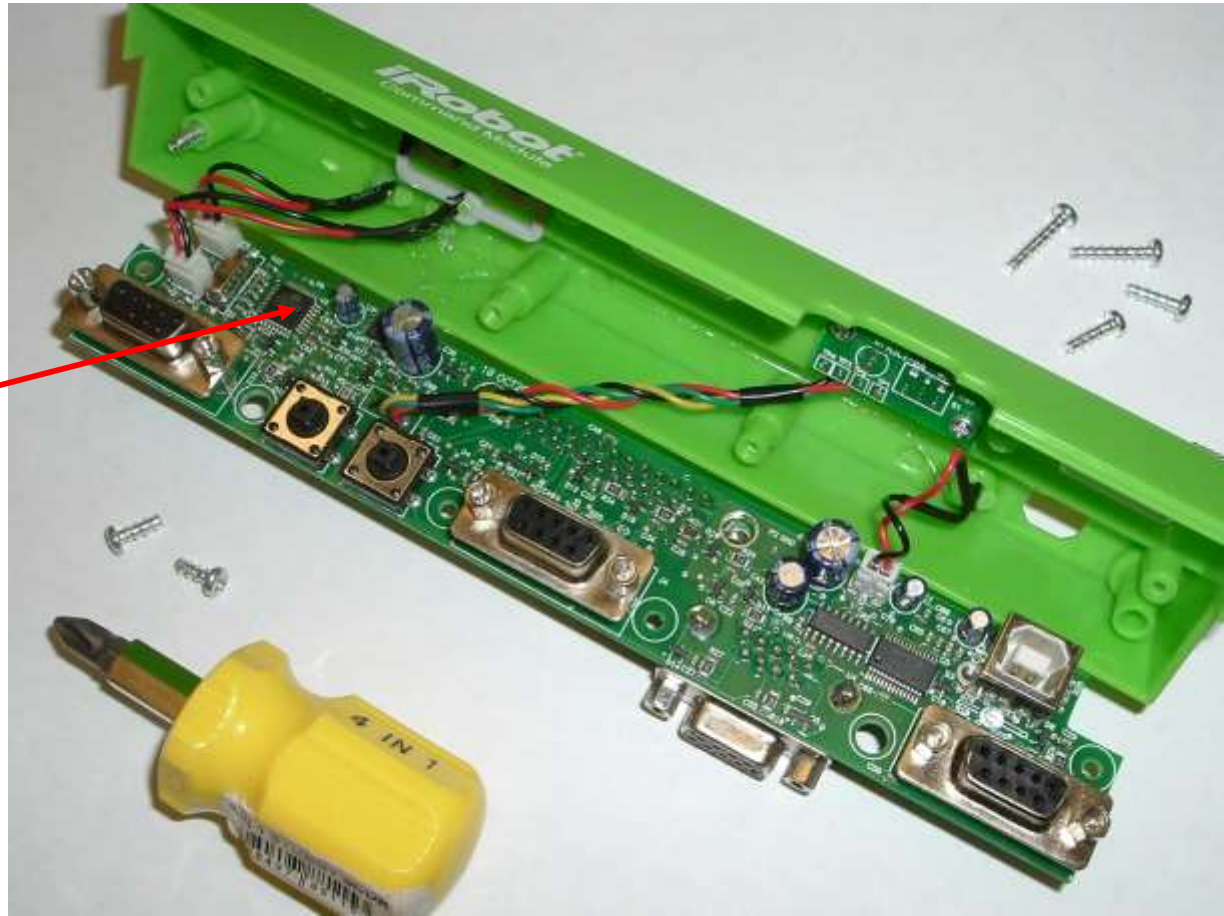
[http://www.open-electronics.org]

Another example use of an AVR processor



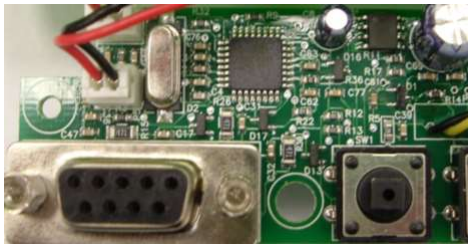


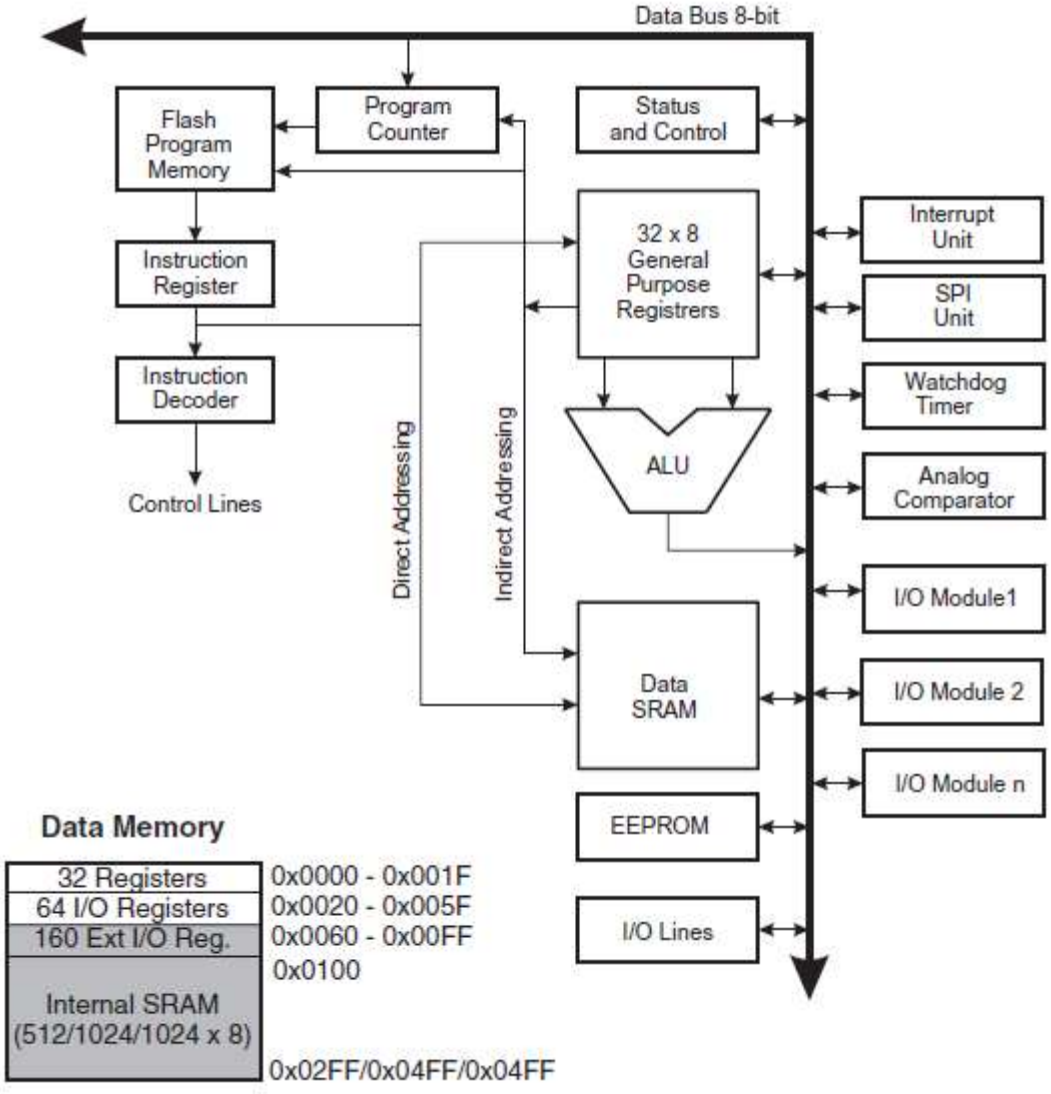
The *iRobot Create* Command Module



Atmel ATMega 168 Microcontroller

ATMega 168: An 8-bit microcontroller with 16-bit addresses



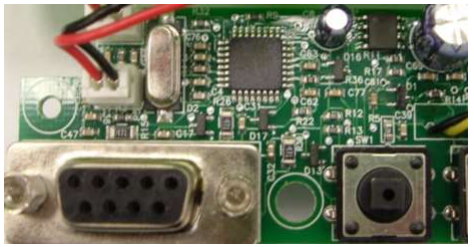


AVR microcontroller architecture used in iRobot command module.

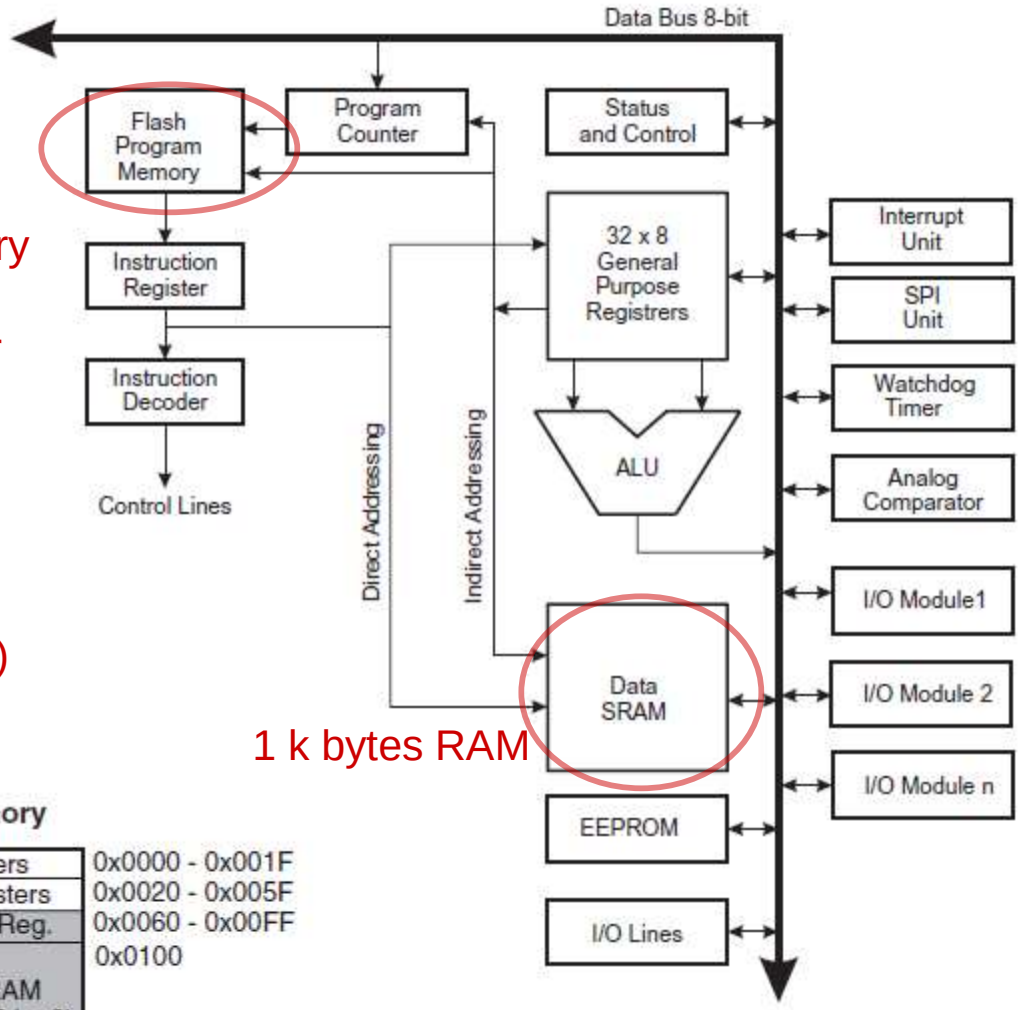
Why is it called an 8-bit microcontroller?

ATMega168 Memory Architecture

An 8-bit microcontroller with 16-bit addresses



iRobot command module has 16K bytes flash memory (14,336 available for the user program. Includes interrupt vectors and boot loader.)



1 k bytes RAM

The "8-bit data" is why this is called an "8-bit microcontroller."

Additional I/O on the command module:

- Two 8-bit timer/counters
- One 16-bit timer/counter
- 6 PWM channels
- 8-channel, 10-bit ADC
- One serial UART
- 2-wire serial interface

Source: ATmega168 Reference Manual

Memory Organization for Programs

- Statically-allocated memory
  - Compiler chooses the address at which to store a variable.
- Stack
  - Dynamically allocated memory with a Last-in, First-out (LIFO) strategy
- Heap
  - Dynamically allocated memory

Statically-Allocated Memory in C

```
char x;
int main(void) {
      x = 0x20;
      …
}
```

Compiler chooses what address to use for x, and the variable is accessible across procedures. The variable's lifetime is the total duration of the program execution.

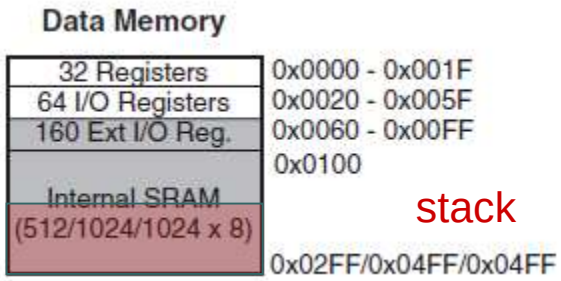
Statically-Allocated Memory with Limited Scope

```
void foo(void) {
     static char x;
     x = 0x20;
     …
}
```

Compiler chooses what address to use for x, but the variable is meant to be accessible only in foo(). The variable's lifetime is the total duration of the program execution (values persist across calls to foo()).

Variables on the Stack ("automatic variables")

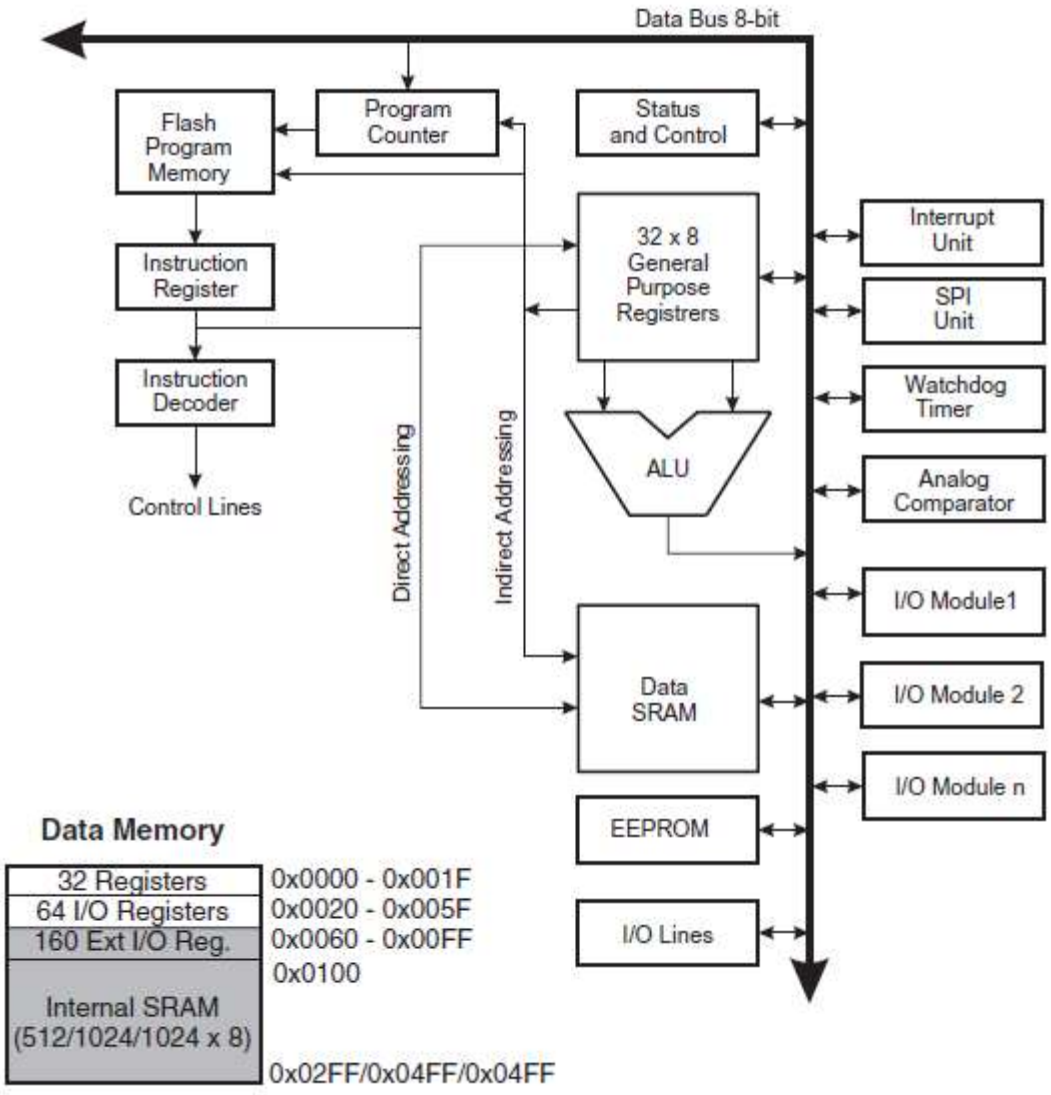
```
void foo(void) {
      char x;
      x = 0x20;
      …
}
```



As nested procedures get called, the stack pointer moves to lower memory addresses. When these procedures, return, the pointer moves up.

When the procedure is called, x is assigned an address on the stack (by decrementing the stack pointer). When the procedure returns, the memory is freed (by incrementing the stack pointer). The variable persists only for the duration of the call to foo().

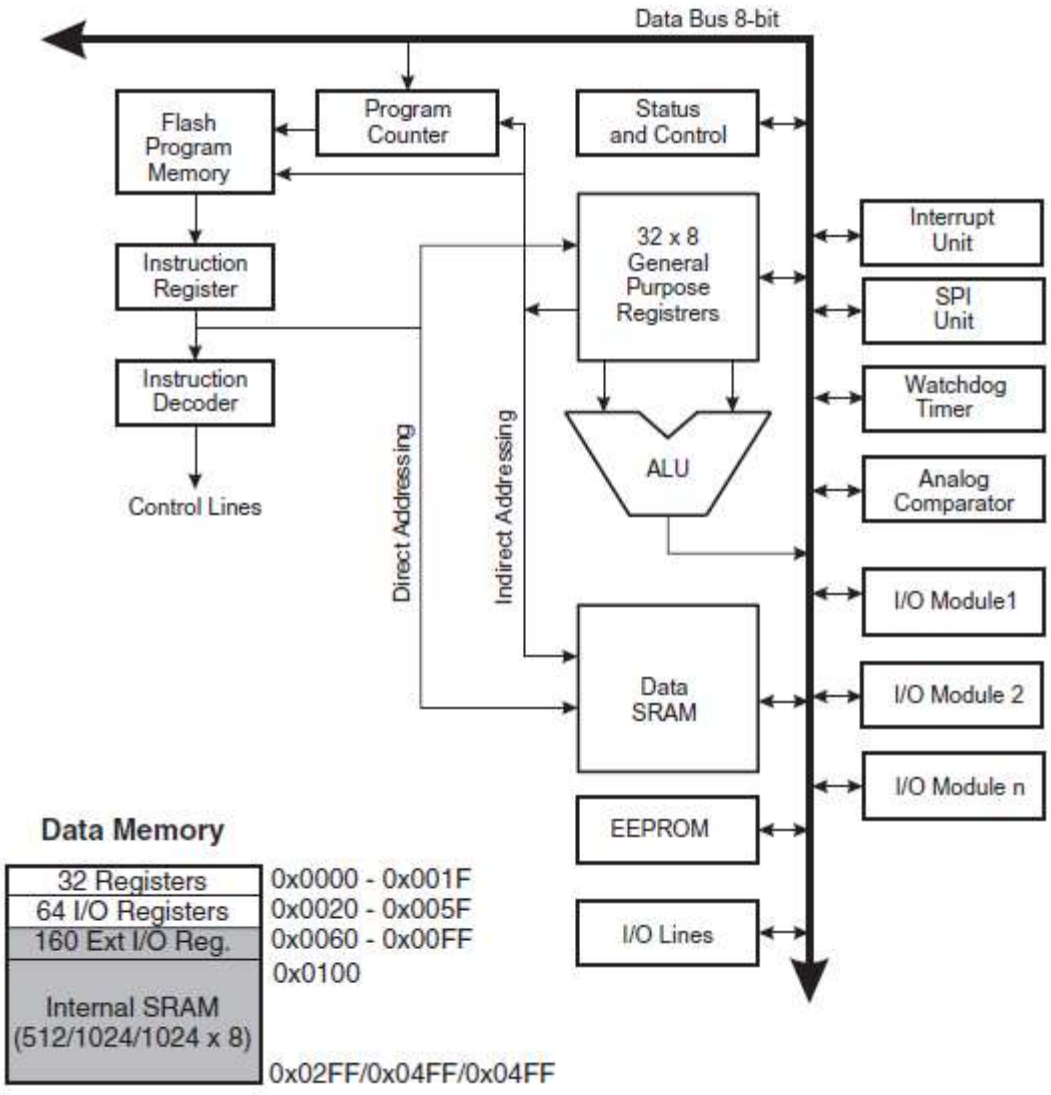
Question 1



What is meant by the following C code:

```
char x;
void foo(void) {
   x = 0x20;
   …
}
```

Answer 1

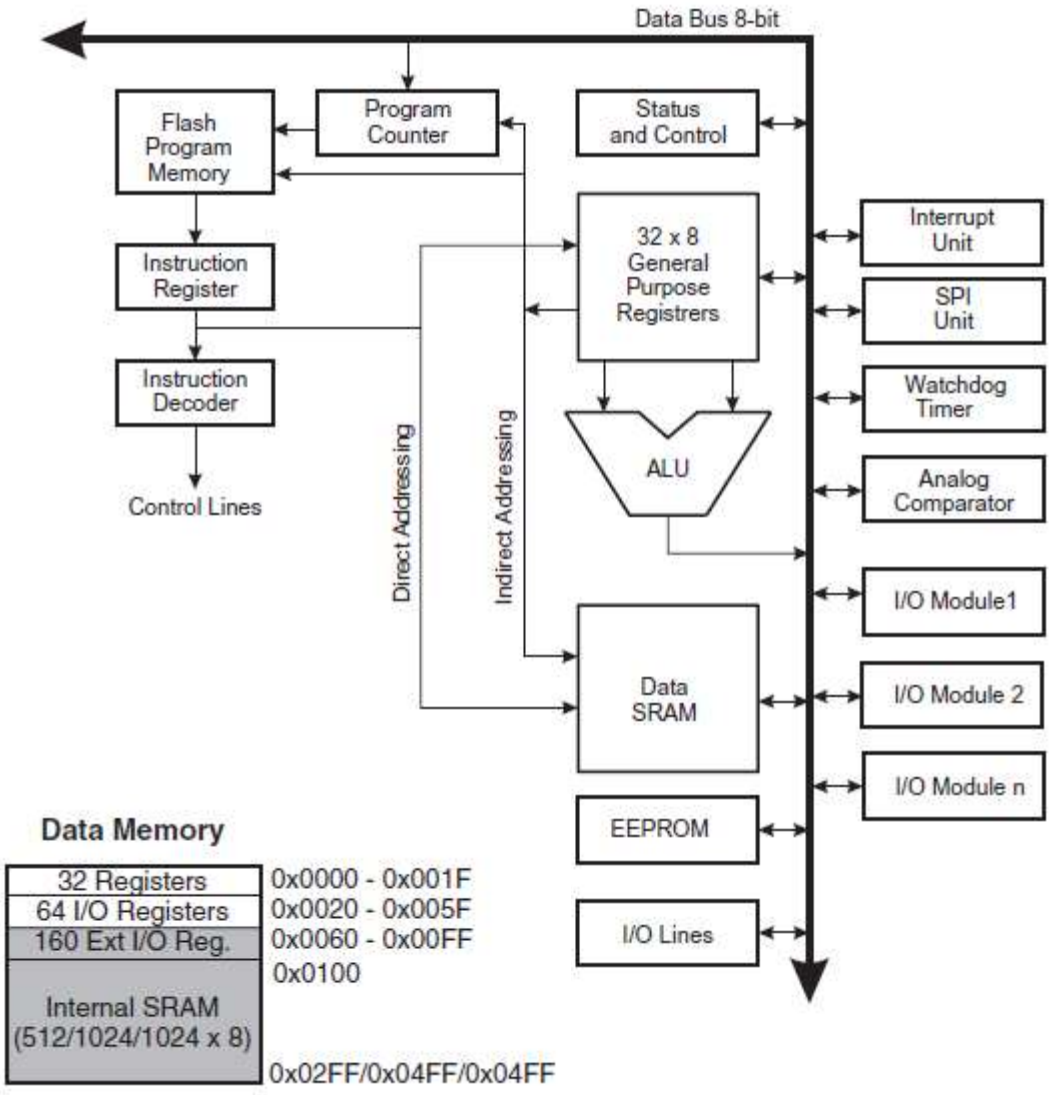


What is meant by the following C code:

```
char x;
void foo(void) {
   x = 0x20;
   …
}
```

An 8-bit quantity (hex 0x20) is stored at an address in statically allocated memory in internal RAM determined by the compiler.

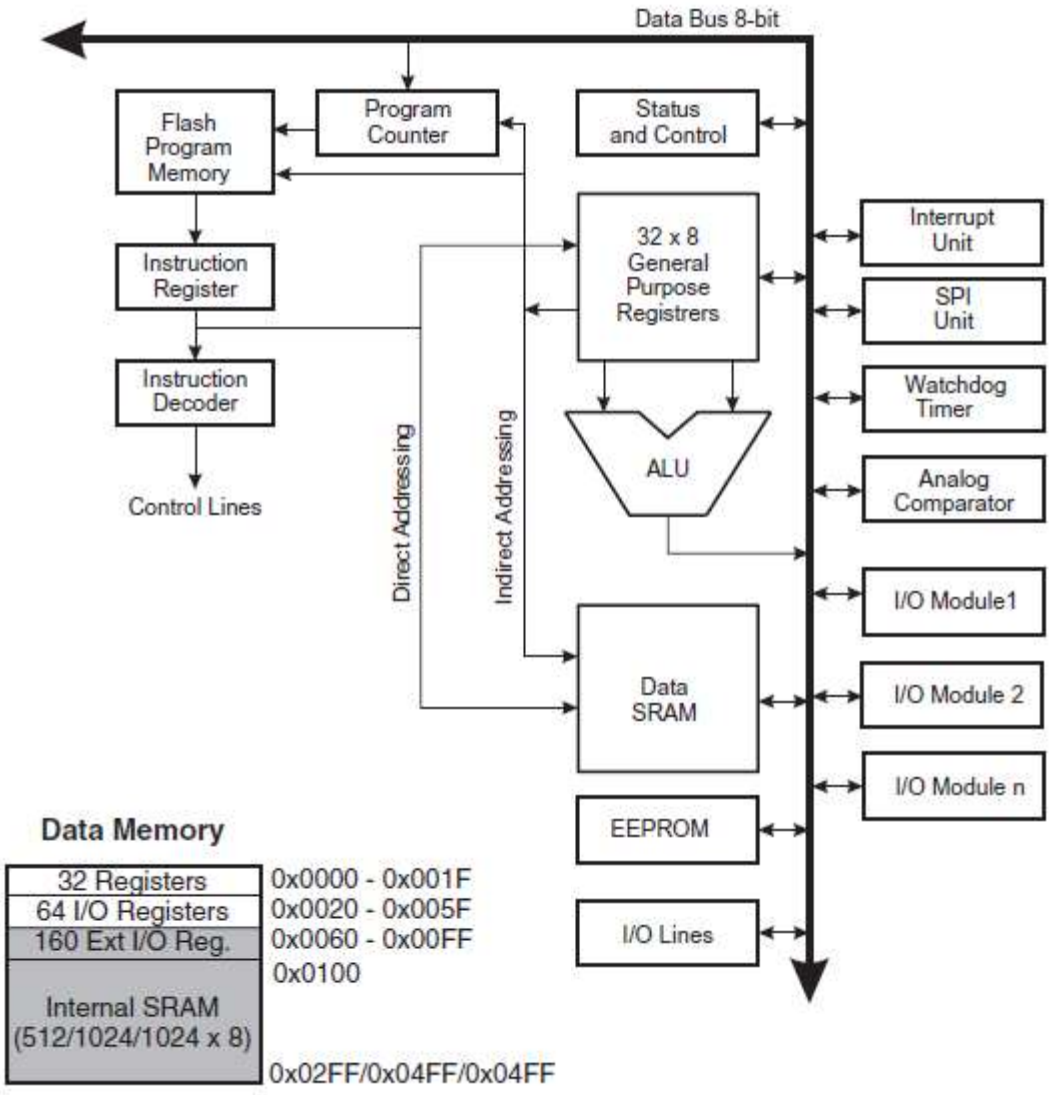
Question 2



What is meant by the following C code:

```
char *x;
void foo(void) {
   x = 0x20;
   …
}
```

Answer 2

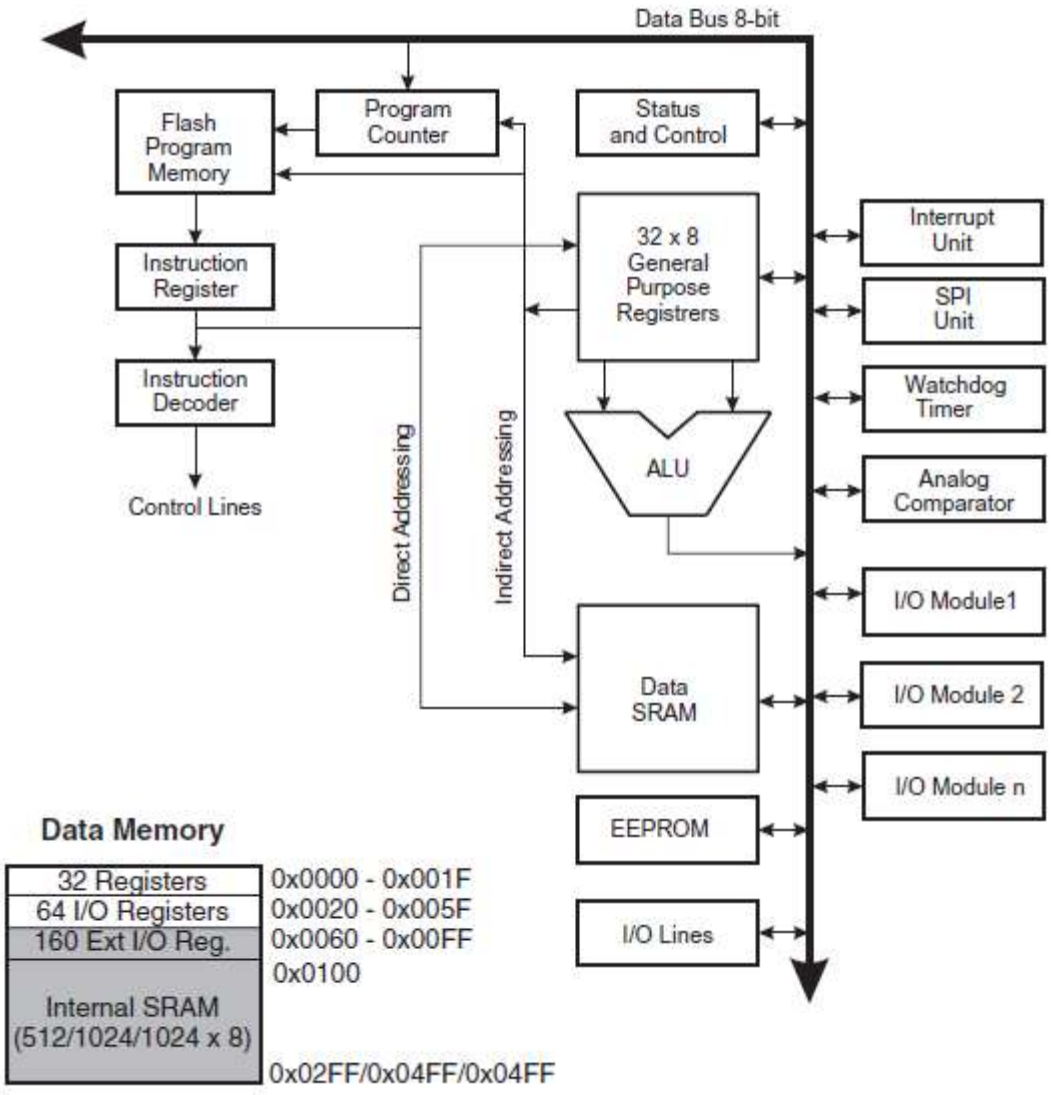


What is meant by the following C code:

```
char *x;
void foo(void) {
   x = 0x20;
   …
}
```

An 16-bit quantity (hex 0x0020) is stored at an address in statically allocated memory in internal RAM determined by the compiler.

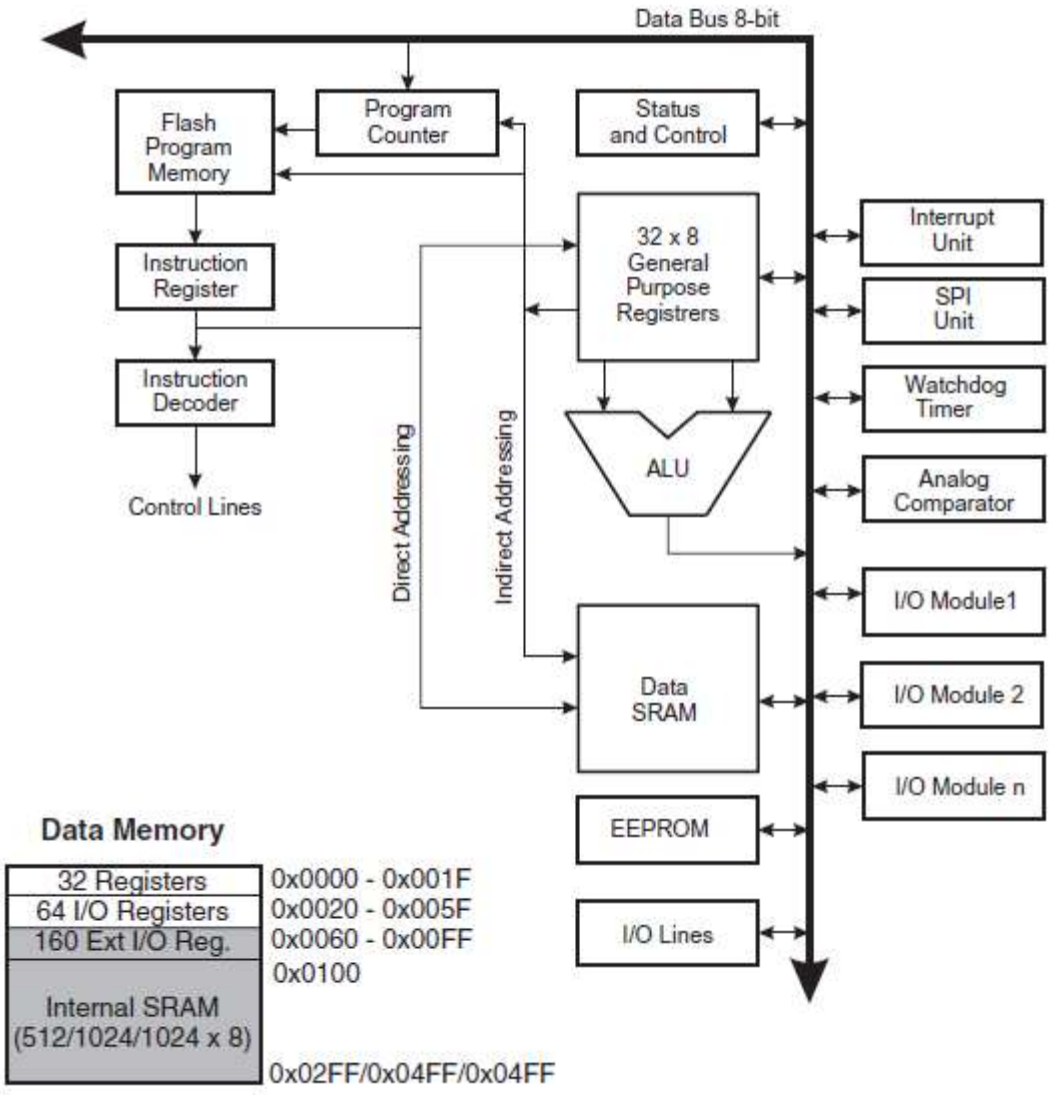
Question 3



What is meant by the following C code:

```
char *x, y;
void foo(void) {
   x = 0x20;
   y = *x;
   …
}
```

Answer 3

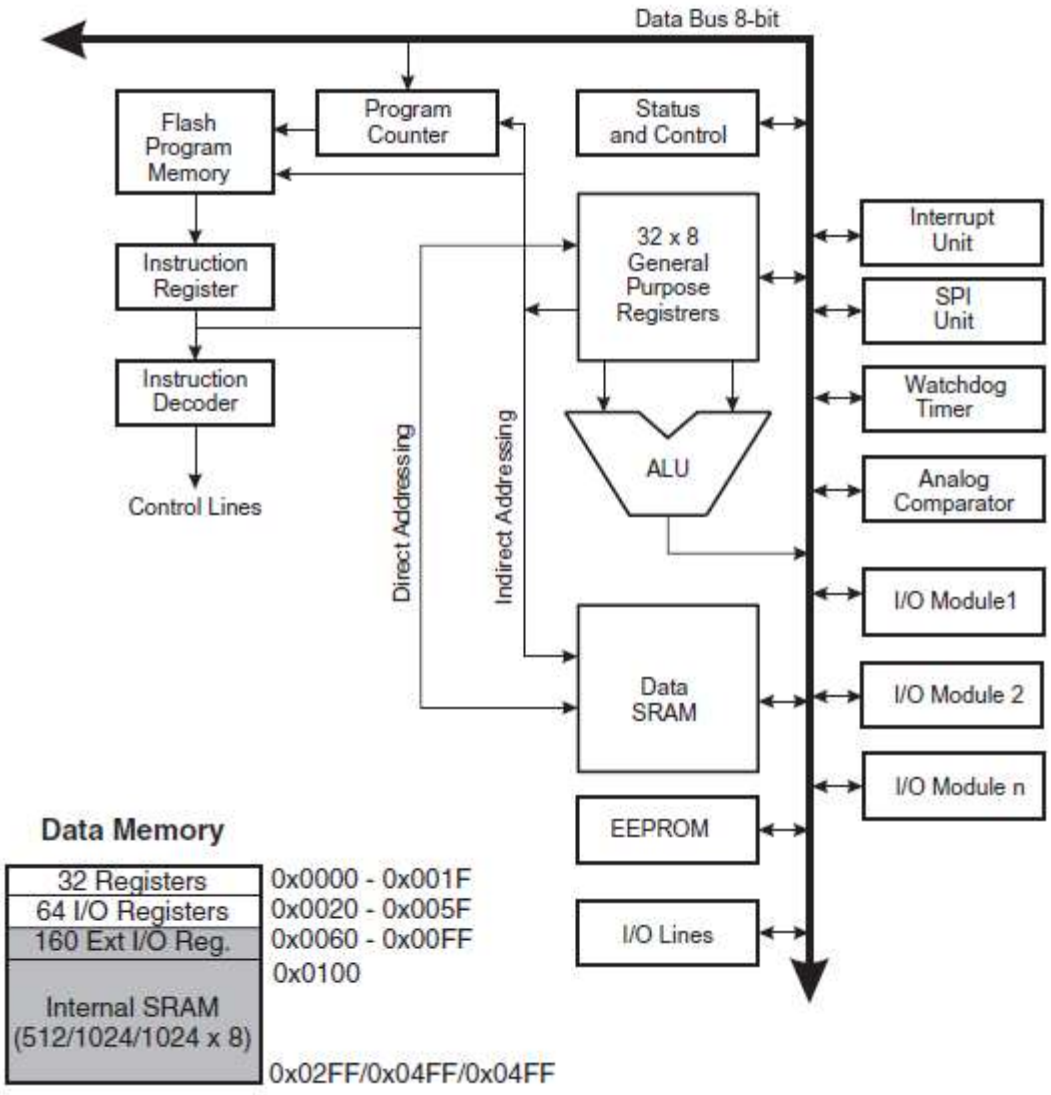


What is meant by the following C code:

```
char *x, y;
void foo(void) {
   x = 0x20;
   y = *x;
   …
}
```

The 8-bit quantity in the I/O register at location 0x20 is loaded into y, which is at a location in internal SRAM determined by the compiler.

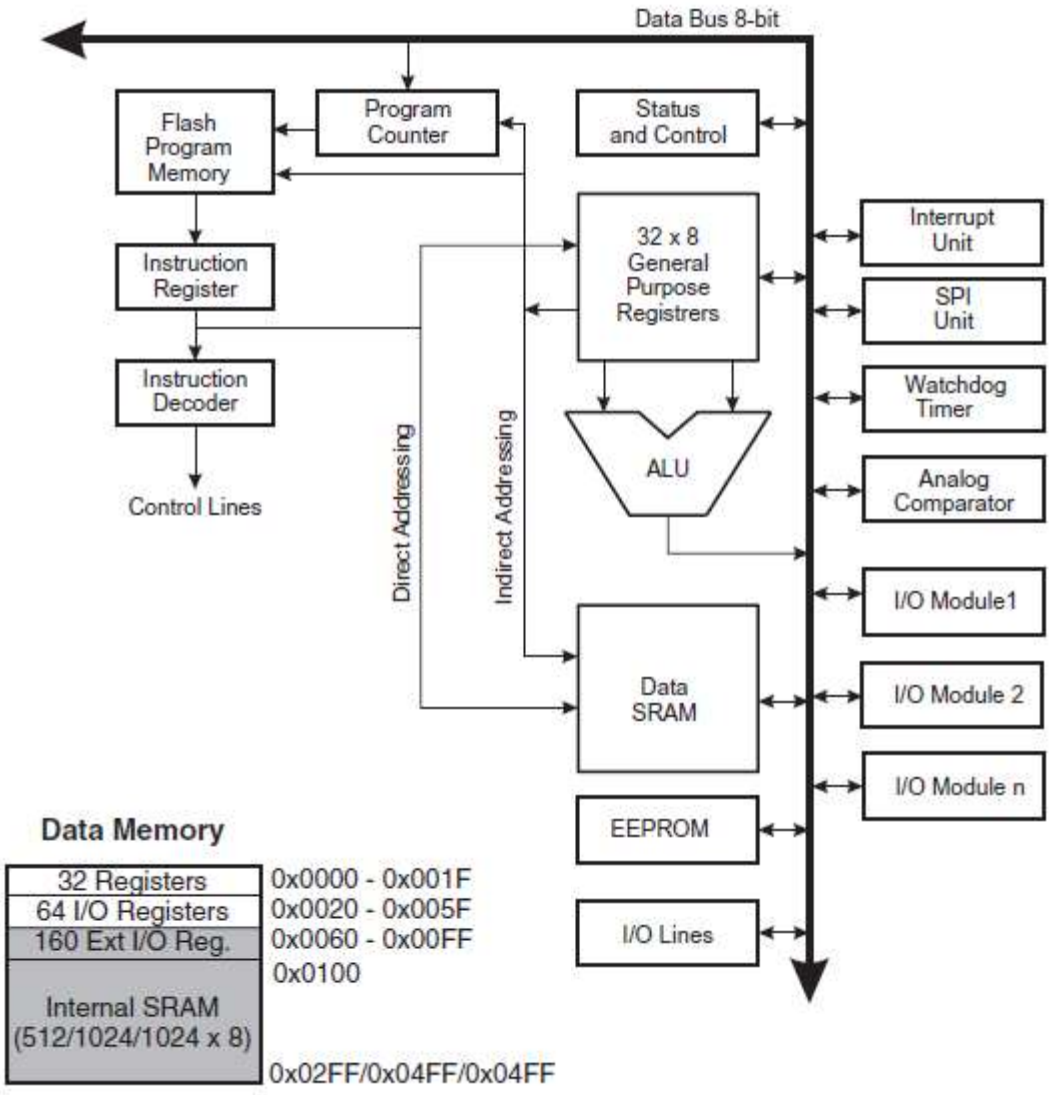
Question 4




```
char foo() {
   char *x, y;
   x = 0x20;
   y = *x;
   return y;
}
char z;
int main(void) {
   z = foo();
   …
}
```

Where are x, y, z in memory?

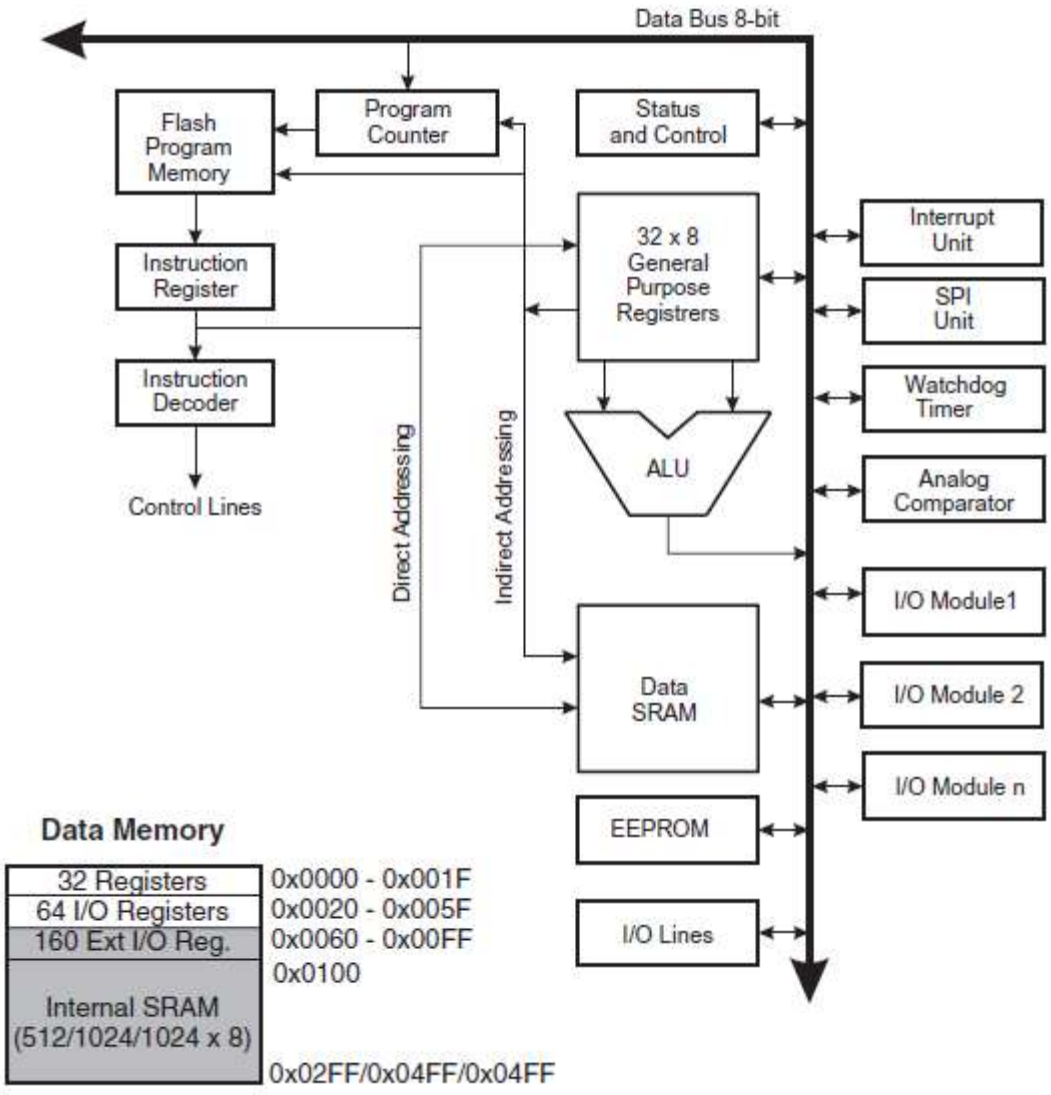
Answer 4




```
char foo() {
   char *x, y;
   x = 0x20;
   y = *x;
   return y;
}
char z;
int main(void) {
   z = foo();
   …
}
```

x occupies 2 bytes on the stack, y occupies 1 byte on the stack, and z occupies 1 byte in static memory.

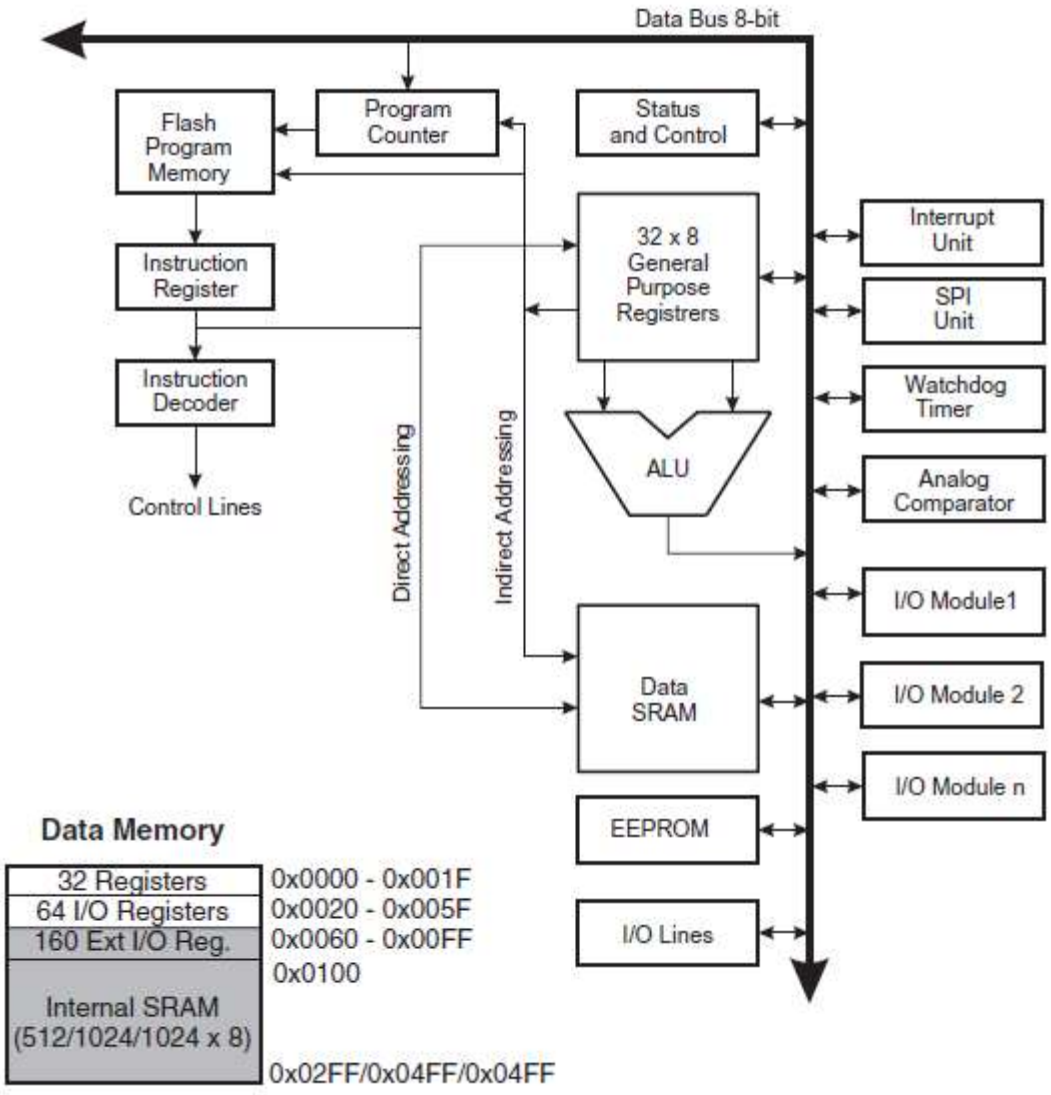
Question 5



What is meant by the following C code:

```
void foo(void) {
   char *x, y;
   x = &y;
   *x = 0x20;
   …
}
```

Answer 5



What is meant by the following C code:

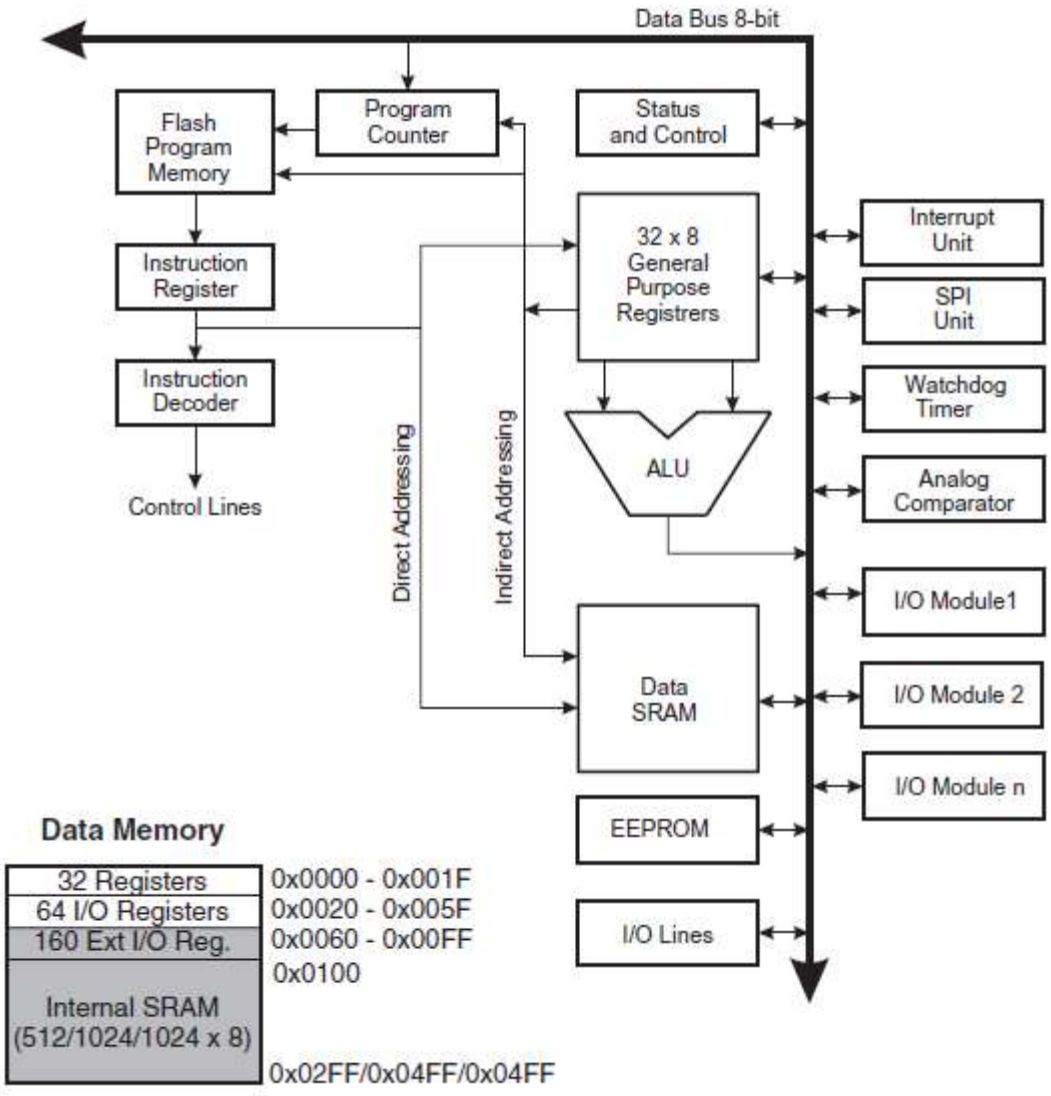
```
void foo(void) {
   char *x, y;
   x = &y;
   *x = 0x20;
   …
}
```

16 bits for x and 8 bits for y are allocated on the stack, then x is loaded with the address of y, and then y is loaded with the 8-bit quantity 0x20.

Question 6

What goes into z in the following program:

```
char foo() {
    char y;
    uint16_t x;
    x = 0x20;
    y = *x;
    return y;
}
char z;
int main(void) {
    z = foo();
    …
}
```

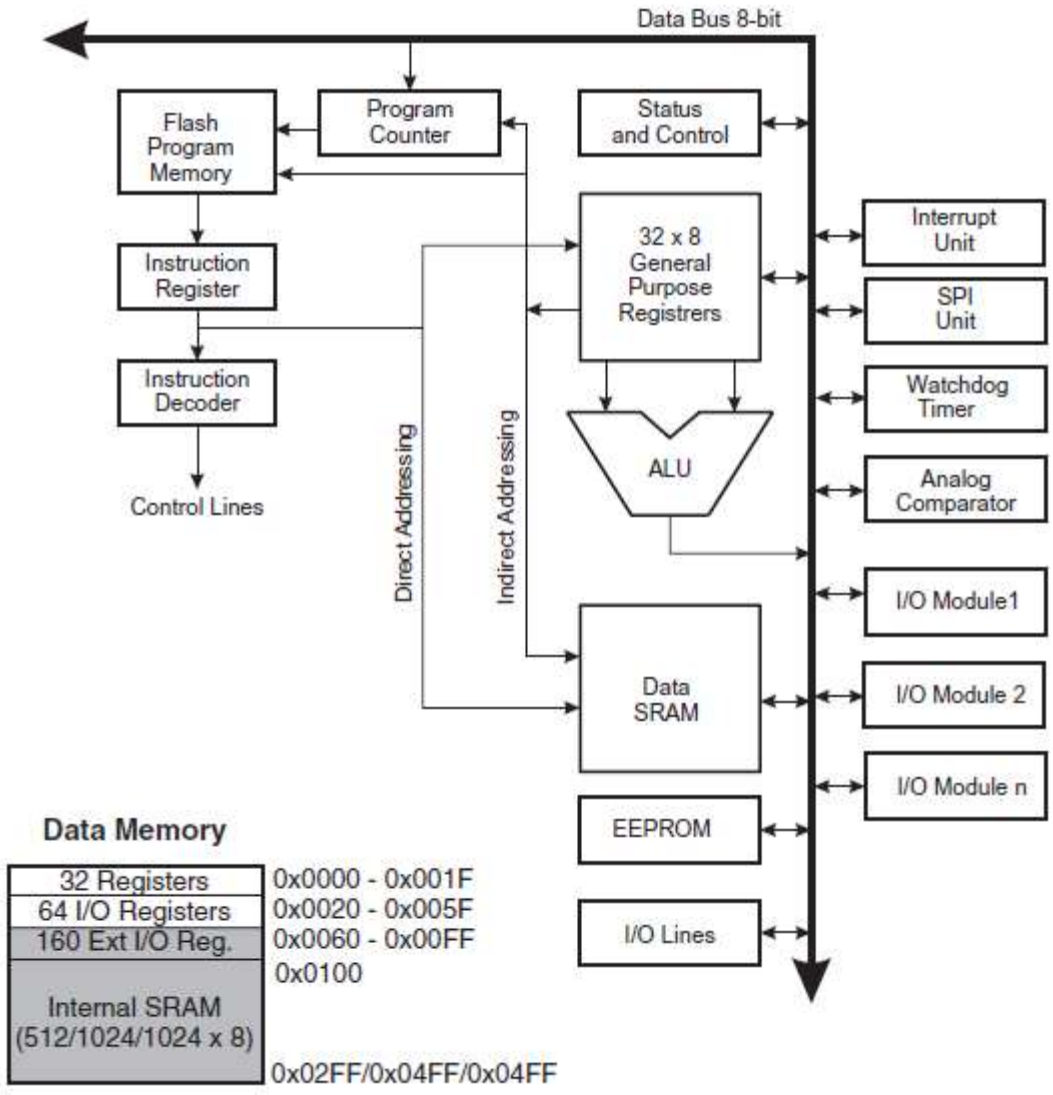


Answer 6

What goes into z in the following program:

```
char foo() {
    char y;
    uint16_t x;
    x = 0x20;
    y = *x;
    return y;
}
char z;
int main(void) {
    z = foo();
    …
}
```

z is loaded with the 8-bit quantity in the I/O register at location 0x20.



Quiz: Find the flaw in this program
(begin by thinking about where each variable is allocated)

```
int x = 2;

int* foo(int y) {
  int z;
  z = y * x;
  return &z;
}

int main(void) {
  int* result = foo(10);
  ...
}
```

Solution: Find the flaw in this program

```
int x = 2;

int* foo(int y) {
  int z;
  z = y * x;
  return &z;
}

int main(void) {
  int* result = foo(10);
  ...
}
```

statically allocated: compiler assigns a memory location.

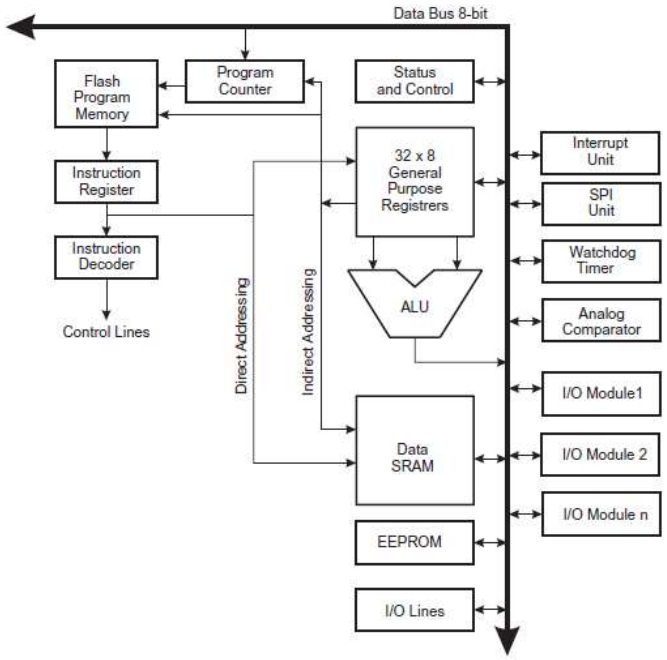
arguments on the stack

automatic variables on the stack

program counter, argument 10, and z go on the stack (and possibly more, depending on the compiler).

The procedure foo() returns a pointer to a variable on the stack. What if another procedure call (or interrupt) occurs before the returned pointer is de-referenced?

Watch out for Recursion!!
Quiz: What is the Final Value of z?

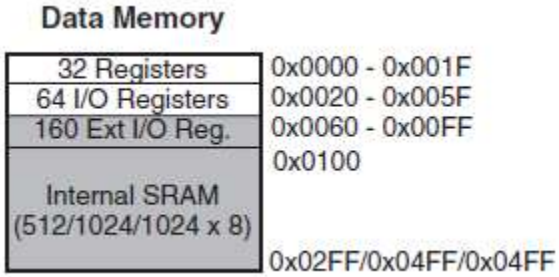



```
void foo(uint16_t x) {
    char y;
    y = *x;
    if (x > 0x100) {
        foo(x - 1);
    }
}
char z;
void main(…) {
    z = 0x10;
    foo(0x04FF);
    …
}
```

Data Memory

32 Registers	0x0000 - 0x001F
64 I/O Registers	0x0020 - 0x005F
160 Ext I/O Reg.	0x0060 - 0x00FF
Internal SRAM (512/1024/1024 x 8)	0x0100 … 0x02FF/0x04FF/0x04FF

Dynamically-Allocated Memory
The Heap



An operating system typically offers a way to dynamically allocate memory on a "heap".

Memory management (malloc() and free()) can lead to many problems with embedded systems:

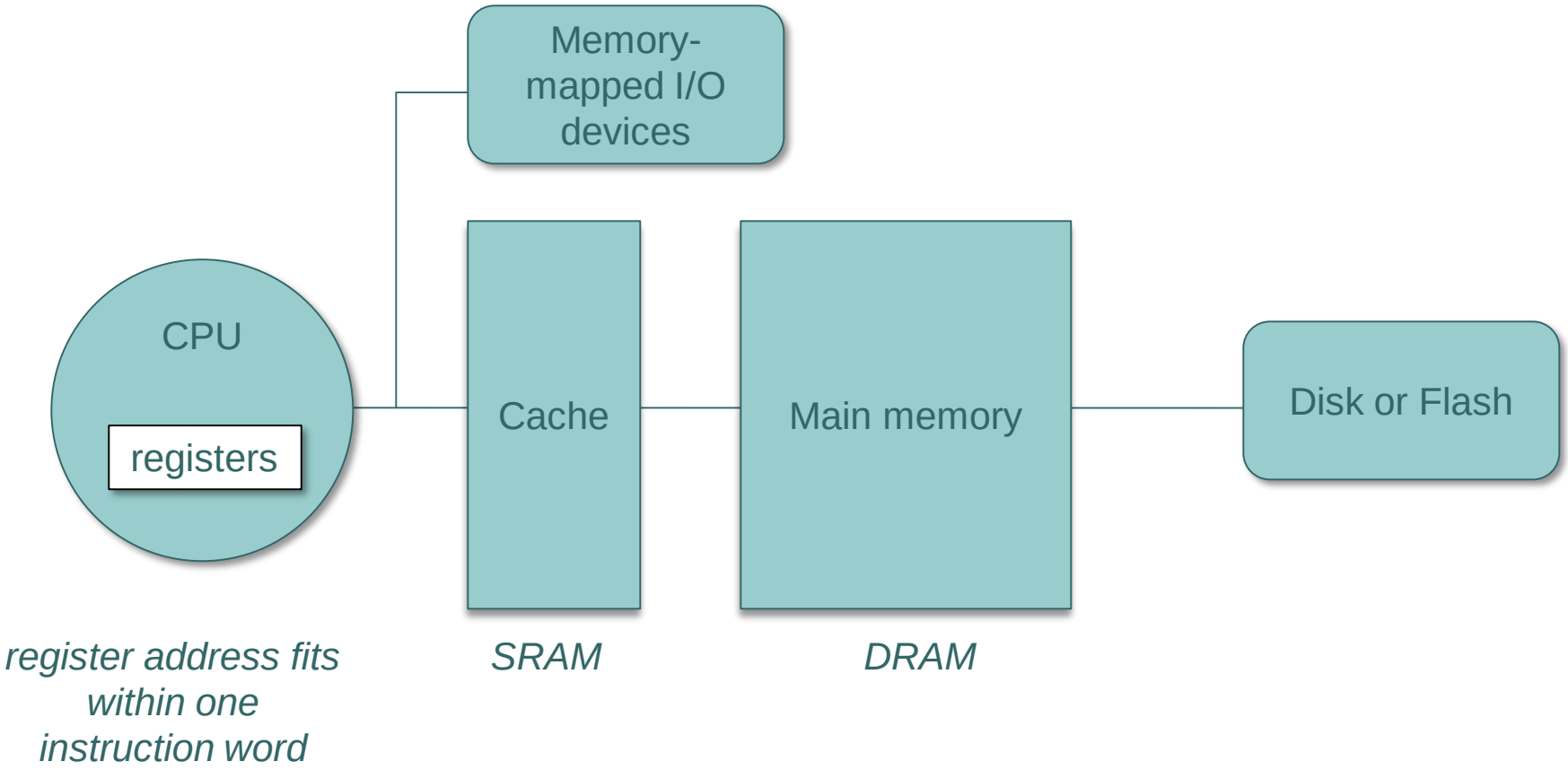
- Memory leaks (allocated memory is never freed)
- Memory fragmentation (allocatable pieces get smaller)

Automatic techniques ("garbage collection") often require stopping everything and reorganizing the allocated memory. This is deadly for real-time programs.

Memory Hierarchies

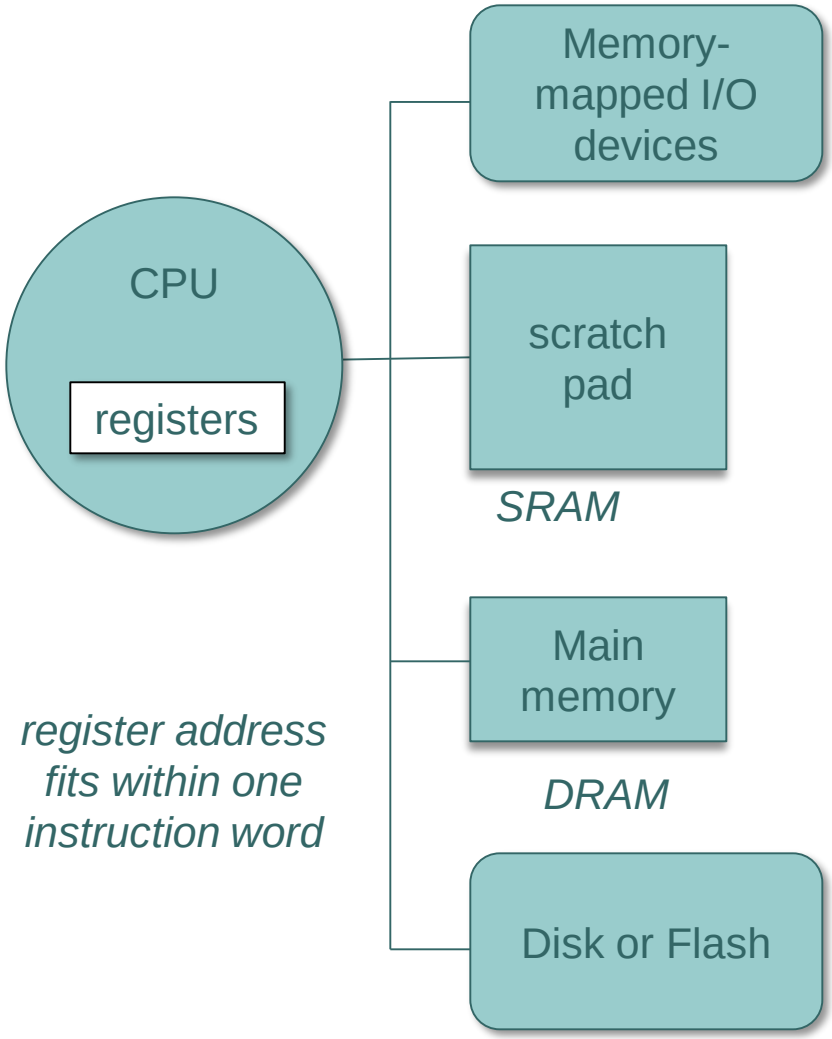
- Memory hierarchy
  - Cache:
    - A subset of memory addresses is mapped to SRAM
    - Accessing an address not in SRAM results in *cache miss*
    - A miss is handled by copying contents of DRAM to SRAM
  - Scratchpad:
    - SRAM and DRAM occupy disjoint regions of memory space
    - Software manages what is stored where
- Segmentation
  - Logical addresses are mapped to a subset of physical addresses
  - Permissions regulate which tasks can access which memory

Memory Hierarchy



Here, the cache or scratchpad, main memory, and disk or flash share the same address space.

Memory Hierarchy

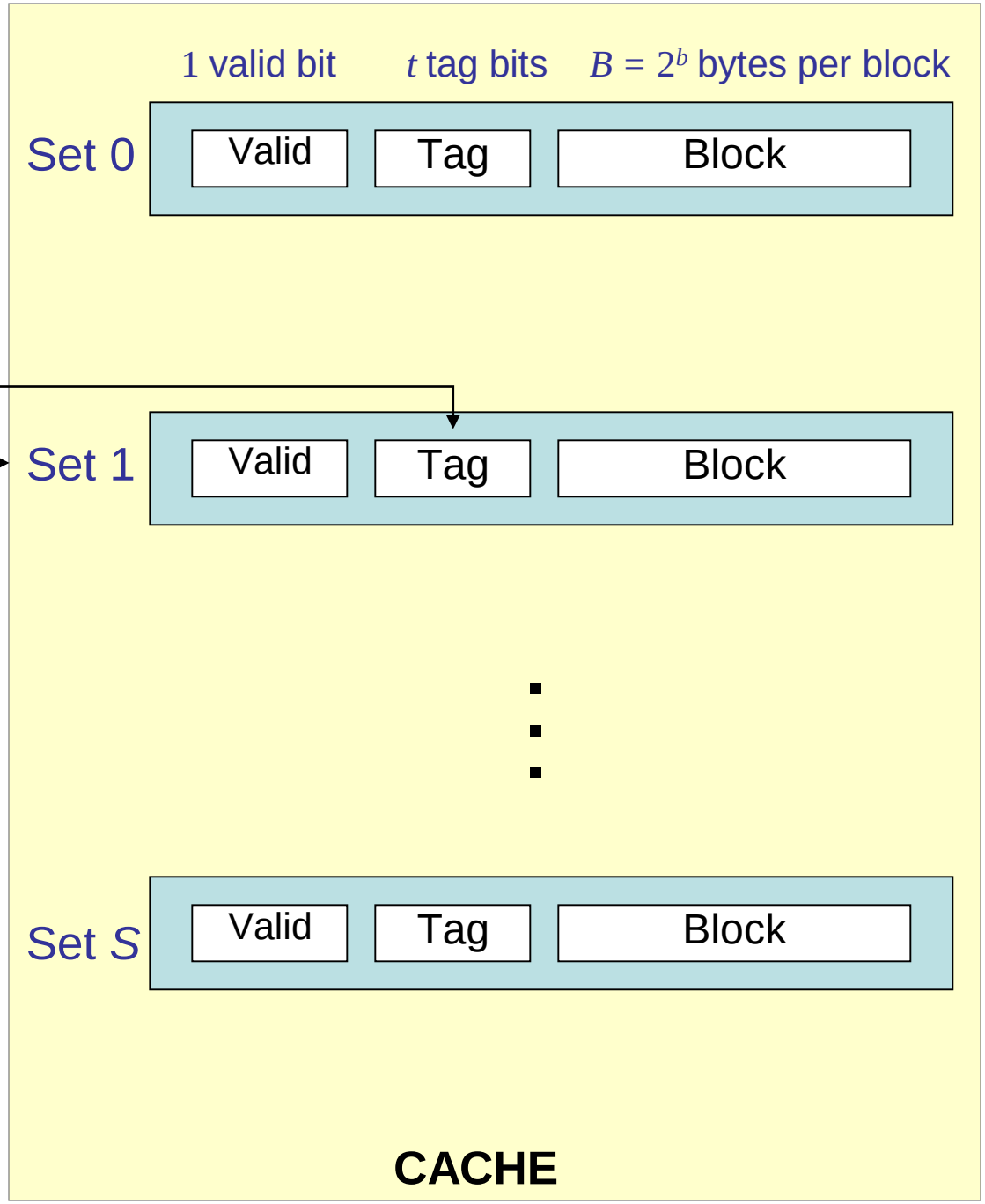


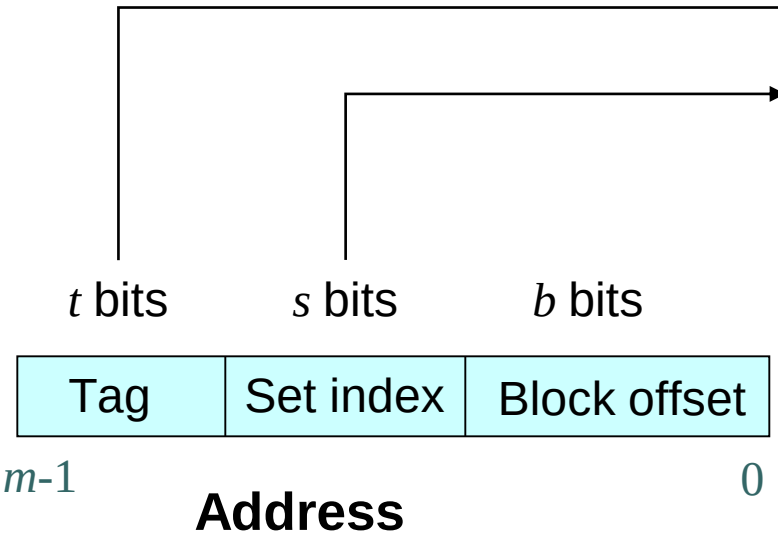
Here, each distinct piece of memory hardware has its own segment of the address space.

This requires more careful software design, but gives more direct control over timing.

Direct-Mapped Cache

A "set" consists of one "line"

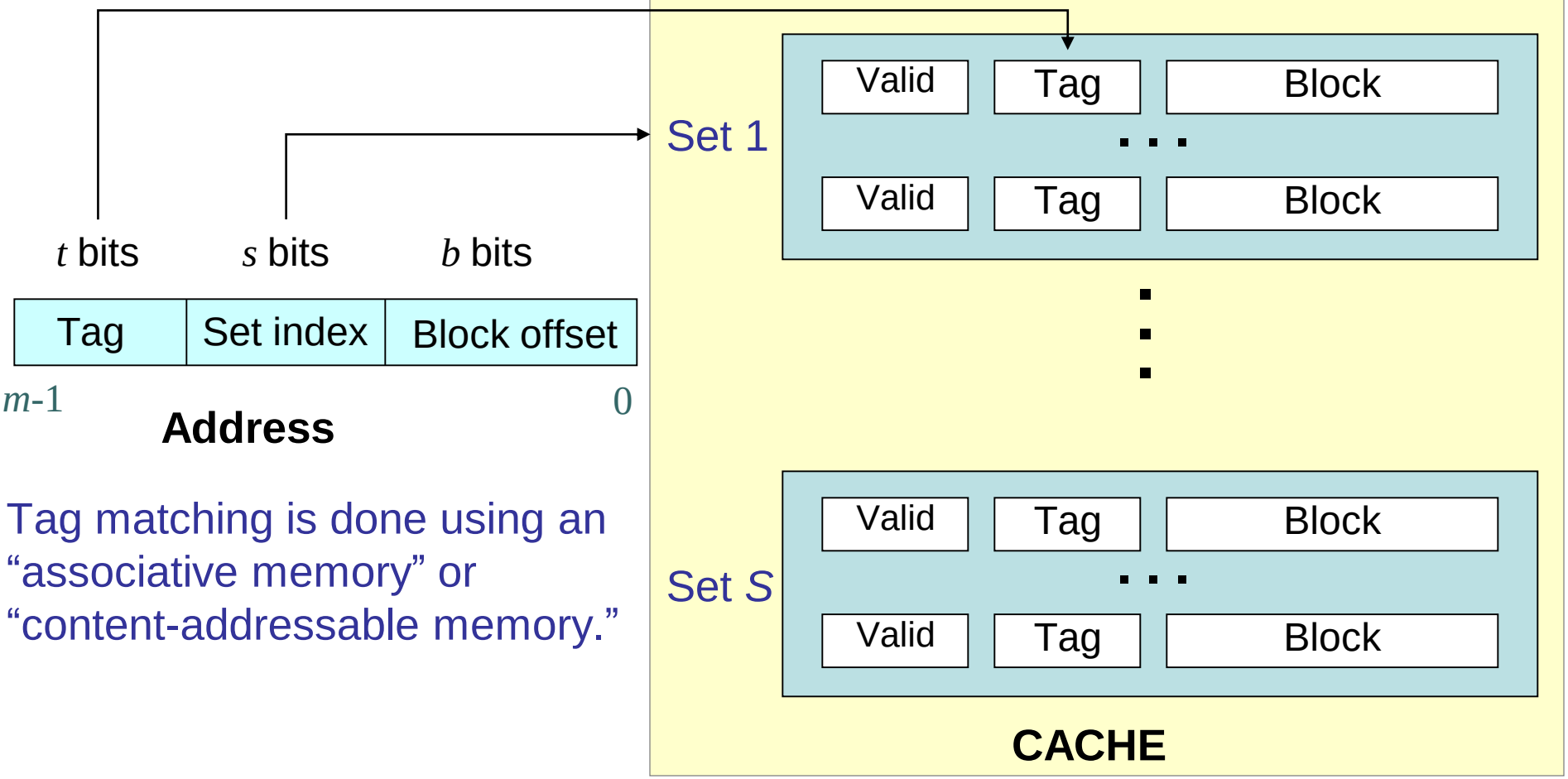




If the tag of the address matches the tag of the line, then we have a "cache hit." Otherwise, the fetch goes to main memory, updating the line.

Set-Associative Cache

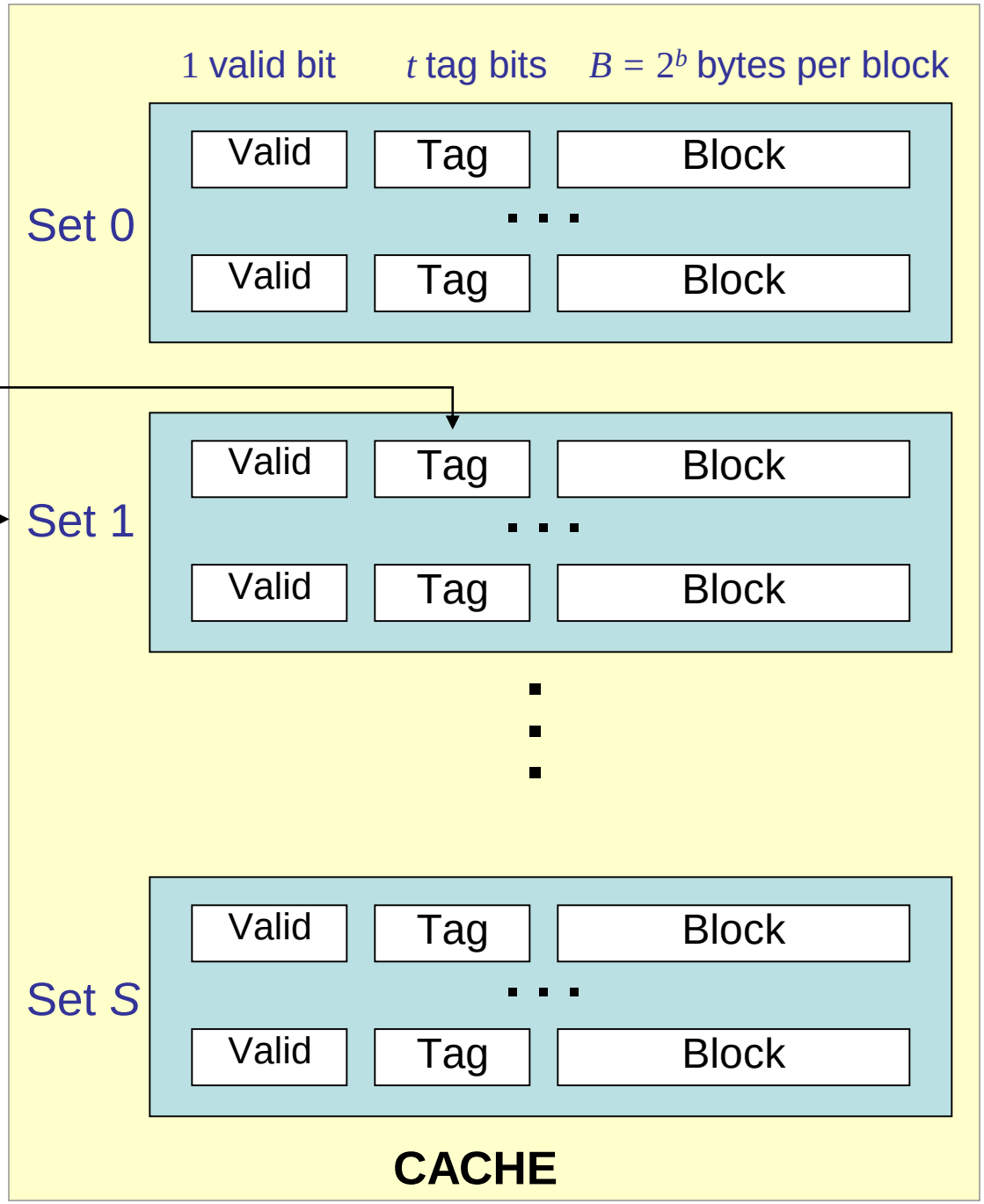
A "set" consists of several "lines"

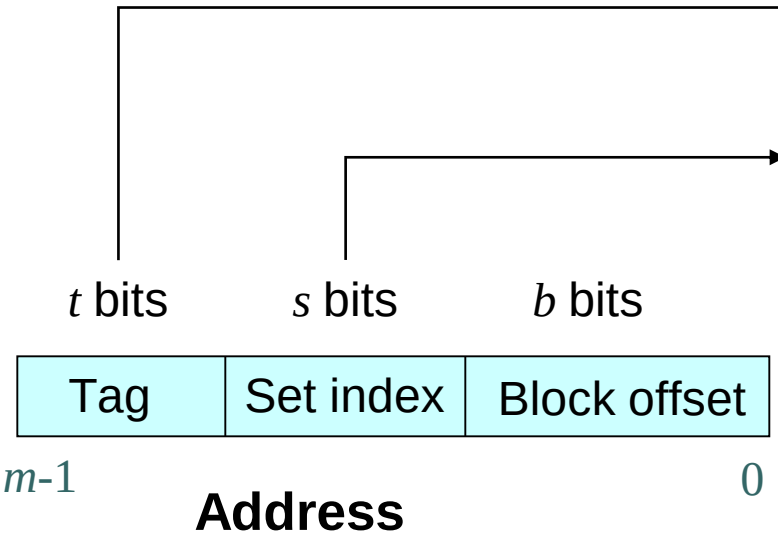


Tag matching is done using an "associative memory" or "content-addressable memory."

Set-Associative Cache

A "set" consists of several "lines"





A "cache miss" requires a replacement policy (like LRU or FIFO).

Your Lab Hardware (2014 - 2016)

myRIO 1950/1900

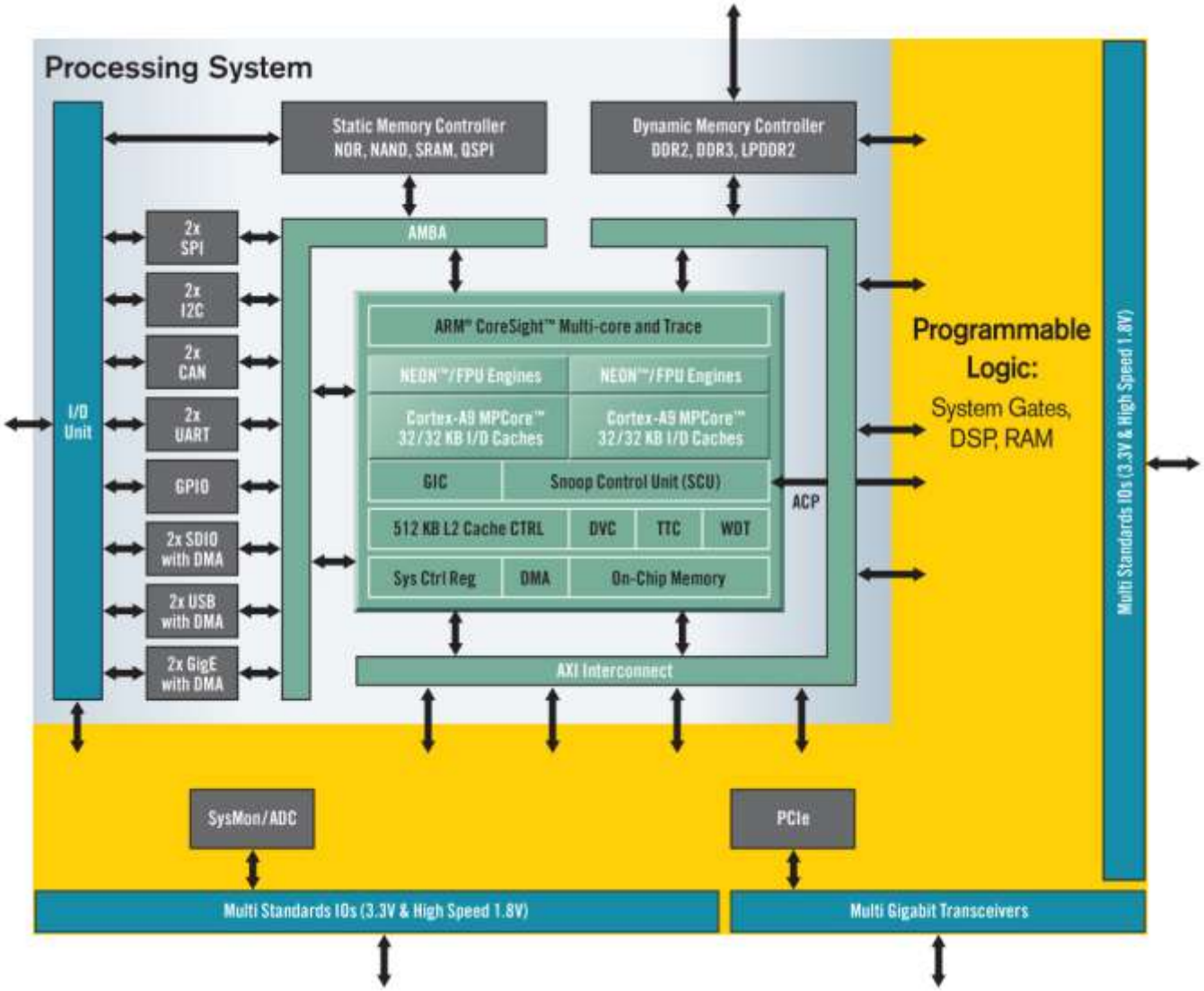
(National Instruments)

Xilinx Zynq Z-7010

- ARM Cortex-A9 MPCore dual core processor
  - Real-time Linux
- Xilinx Artix-7 FPGA
  - Preconfigured with a 32-bit MicroBlaze microprocessor running without an operating system (“bare metal”).

Xilinx Zynq

Dual-core ARM processor
+ FPGA
+ rich I/O
on a single chip.



Microblaze I/O Architecture

Source: Xilinx

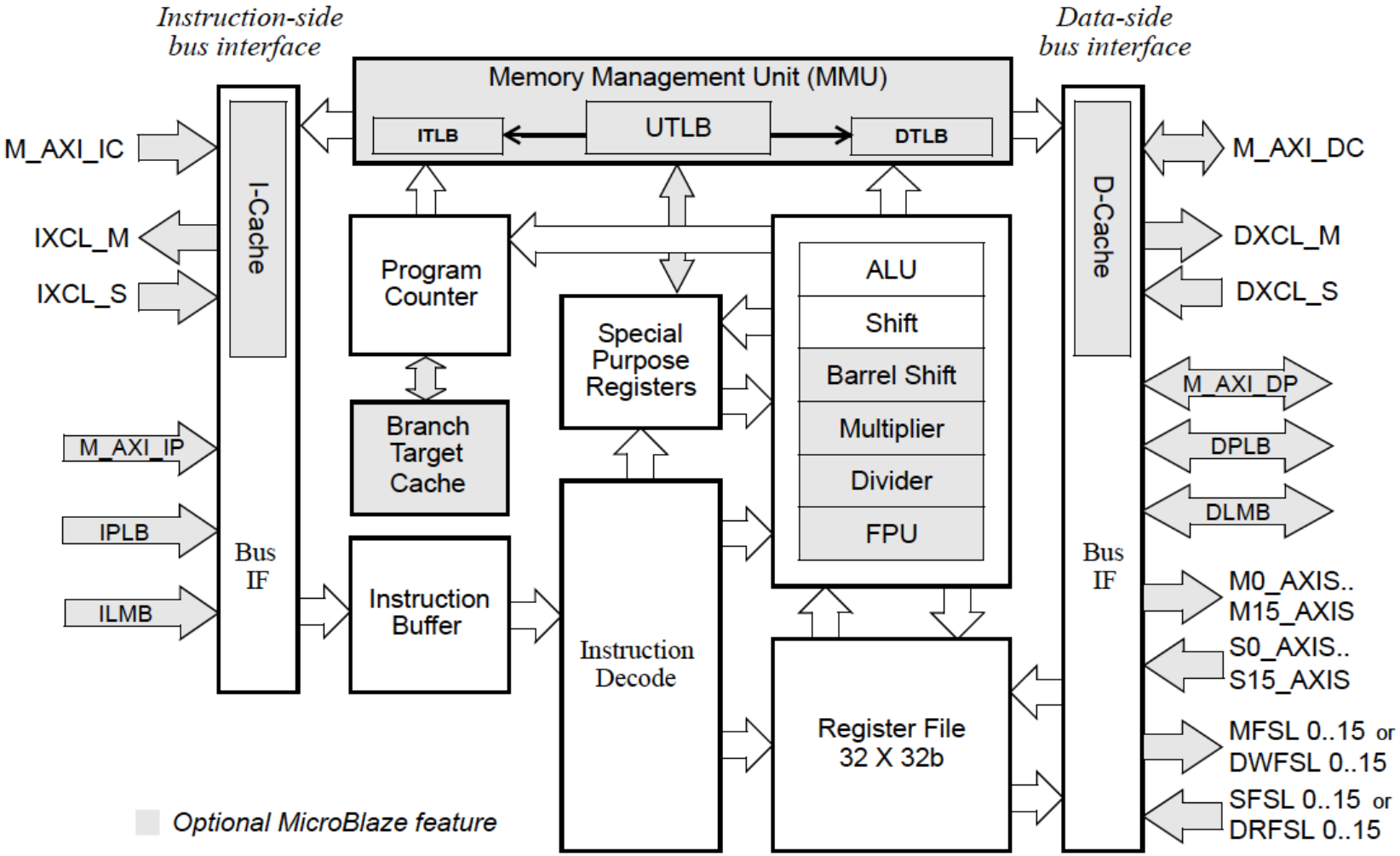
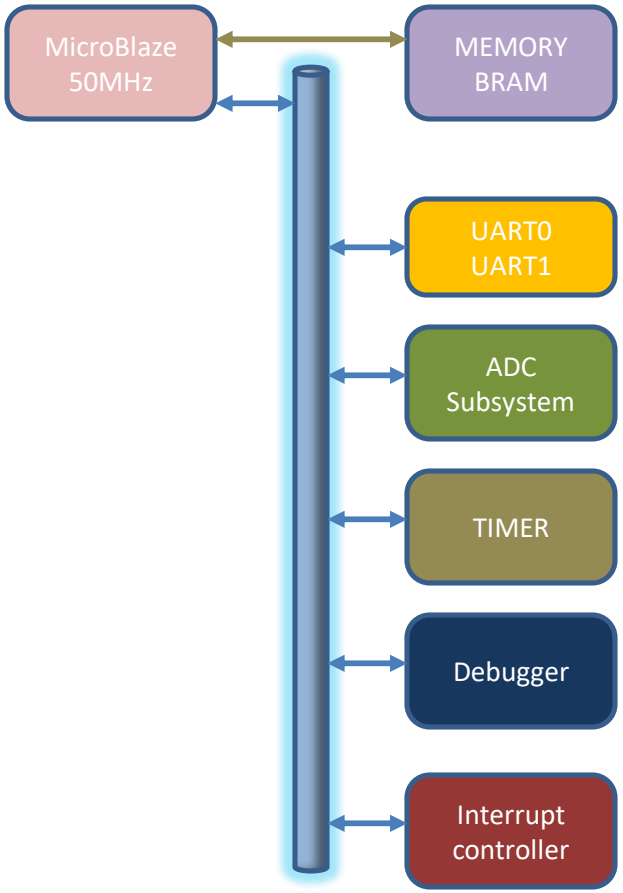
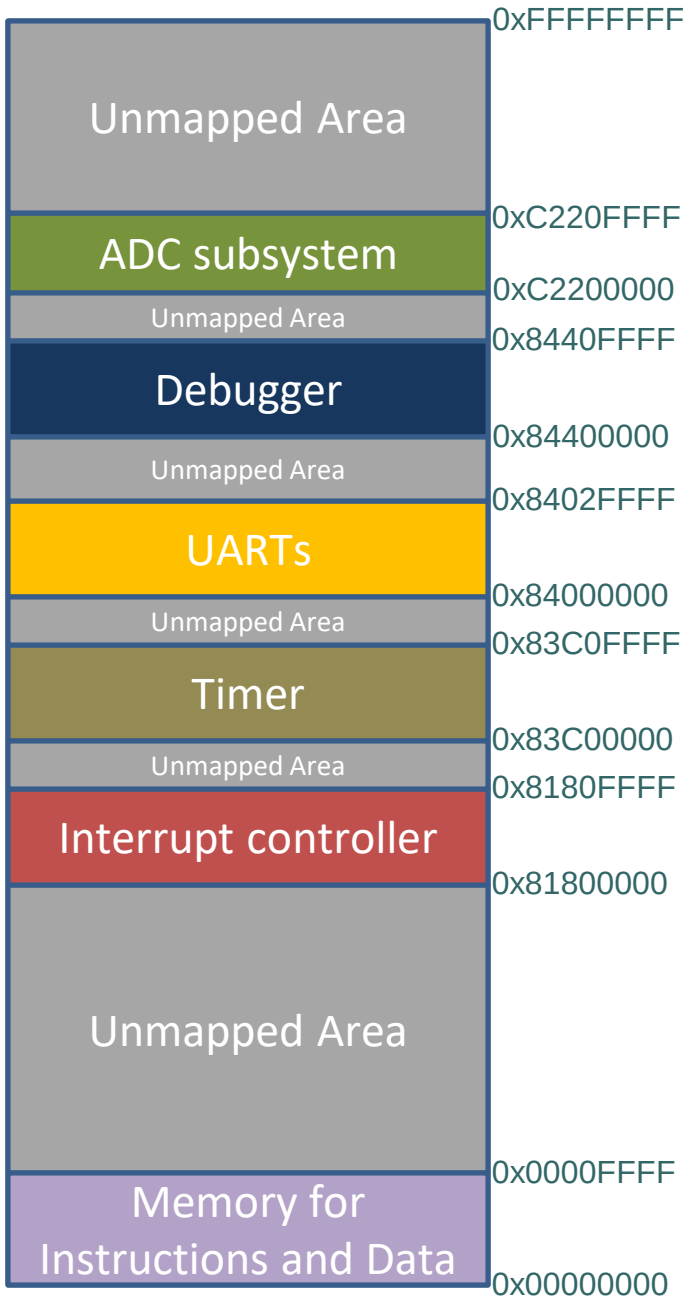


Figure 2-1: **MicroBlaze Core Block Diagram**

Berkeley Microblaze Personality Memory Map





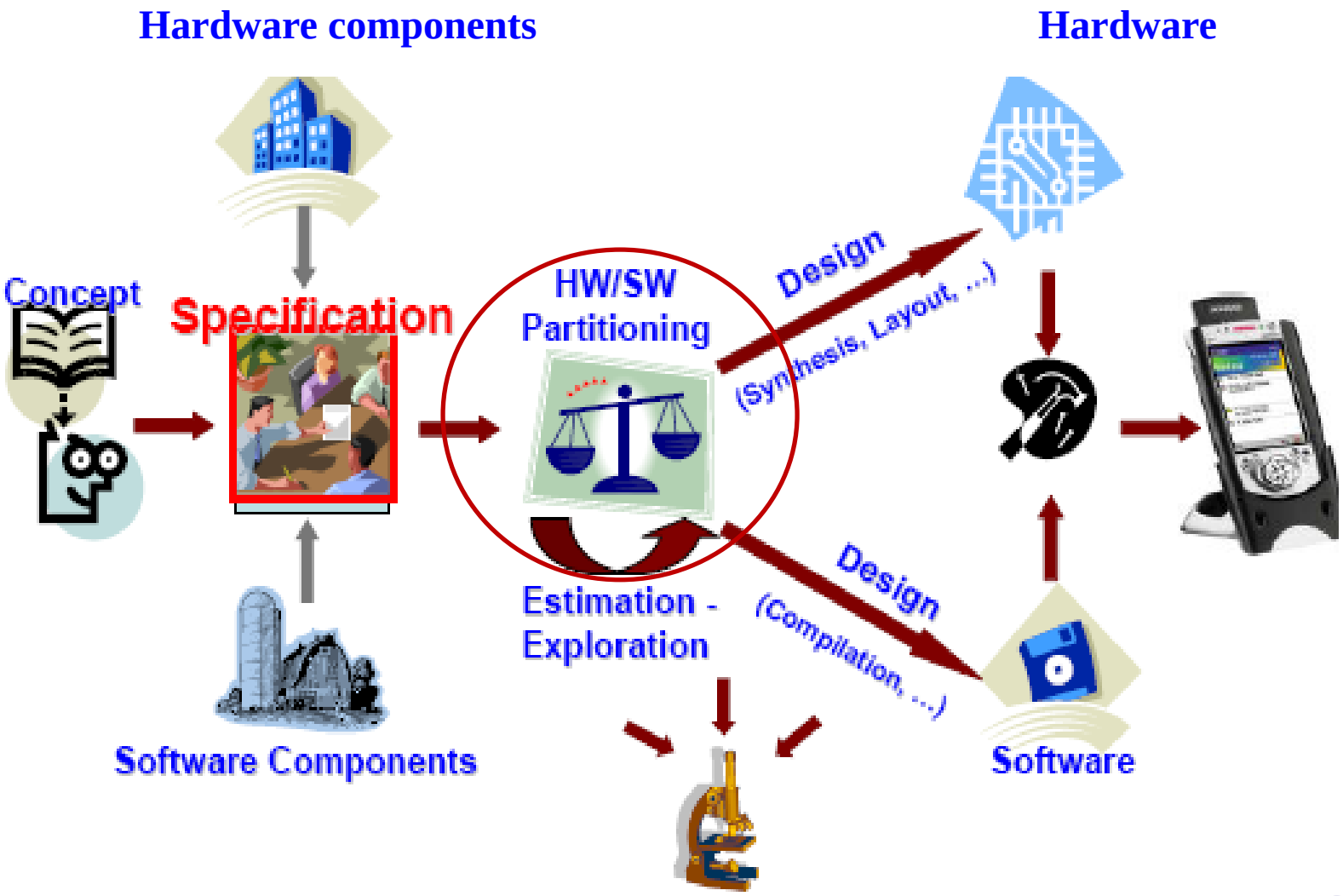
Conclusion

Understanding memory architectures is essential to programming embedded systems.

Hardware/Software Codesign

Tajana Simunic Rosing
Department of Computer Science and Engineering
University of California, San Diego.

ES Design


Hardware components
Hardware
Concept
Specification
HW/SW
Partitioning
Design
(Synthesis, Layout, ...)
Design
(Compilation, ...)
Estimation -
Exploration
Software Components
Software
Verification and Validation

ES Application Classes

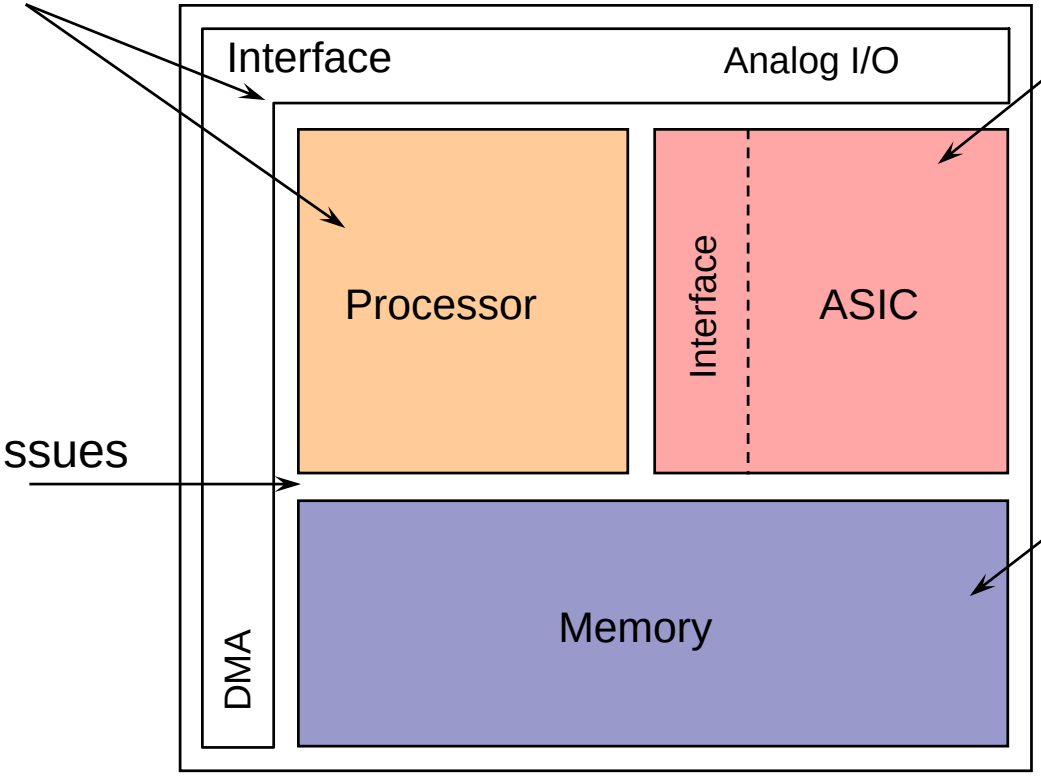
Class	Application	Processor	Requirements
Data flow	laser printers, X-terminals, routers, bridges, image processing	R4600, I960, 29k, Coldfire, PPC (403, 605)	Processes data and passes it on. High memory bw, high throughput.
Interactive video & portable	set-top boxes, video games, PDAs, portable info appliances	R3900, R4100/ 4300/ 4600, ARM 6xx/ 7xx, V851, SH1/ 2/ 3	Interactive, low cost, low power, high throughput.
Classic embedded	controllers, disk controllers, automotive, industrial control	Piranha, ARM, MIPS, Cores	mix of CPU power, low cost, low power, peripherals

Time-constrained computing systems.

System Design Problem Areas

1. Design environment, co-simulation constraint analysis.

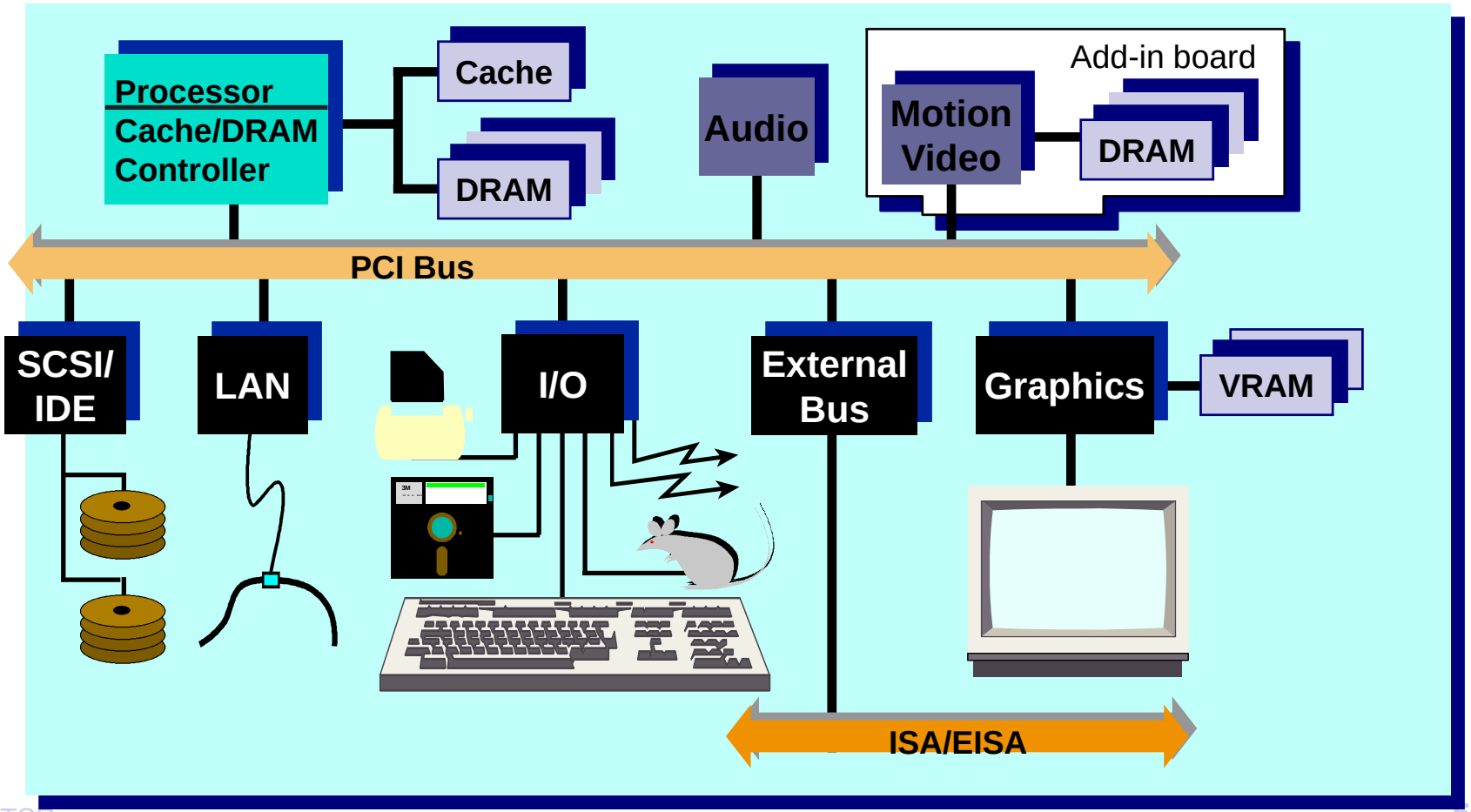
2. HDL Modeling
Architectural synthesis
Logic synthesis
Physical synthesis



3. Software synthesis, Optimization, Retargetable code gen., Debugging & Programming environ.

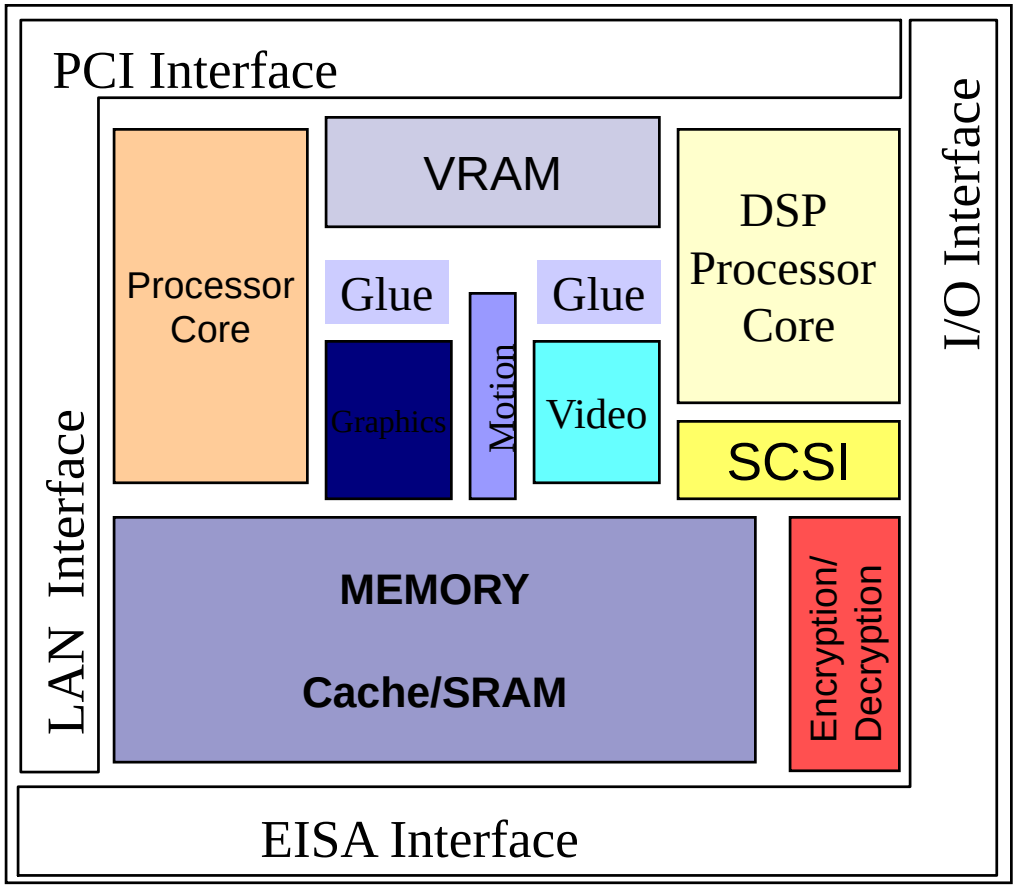
4. Test Issues

System Architecture: Yesterday PCB design

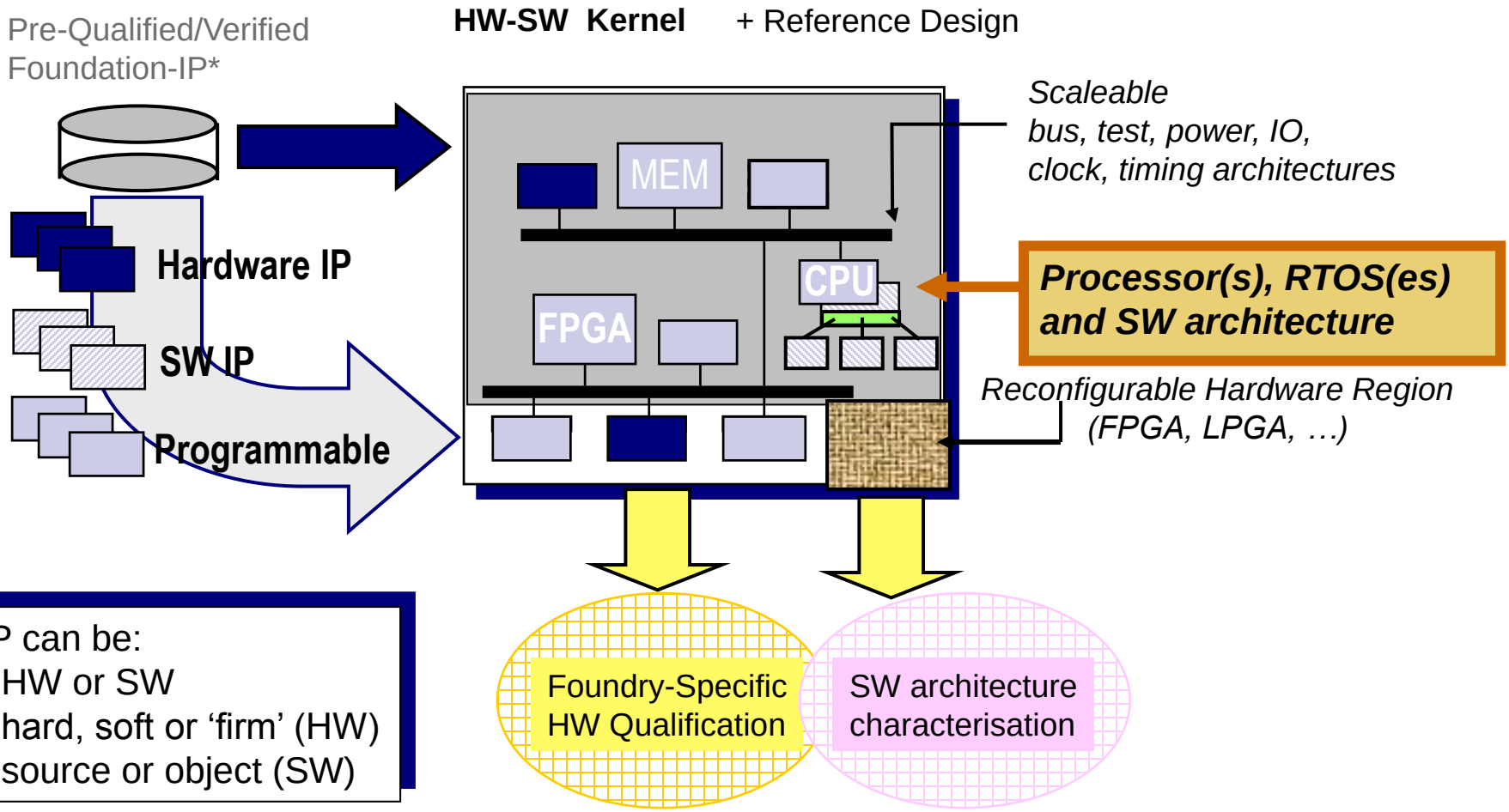


A System Architecture: Today

HW/SW Codesign of a SoC

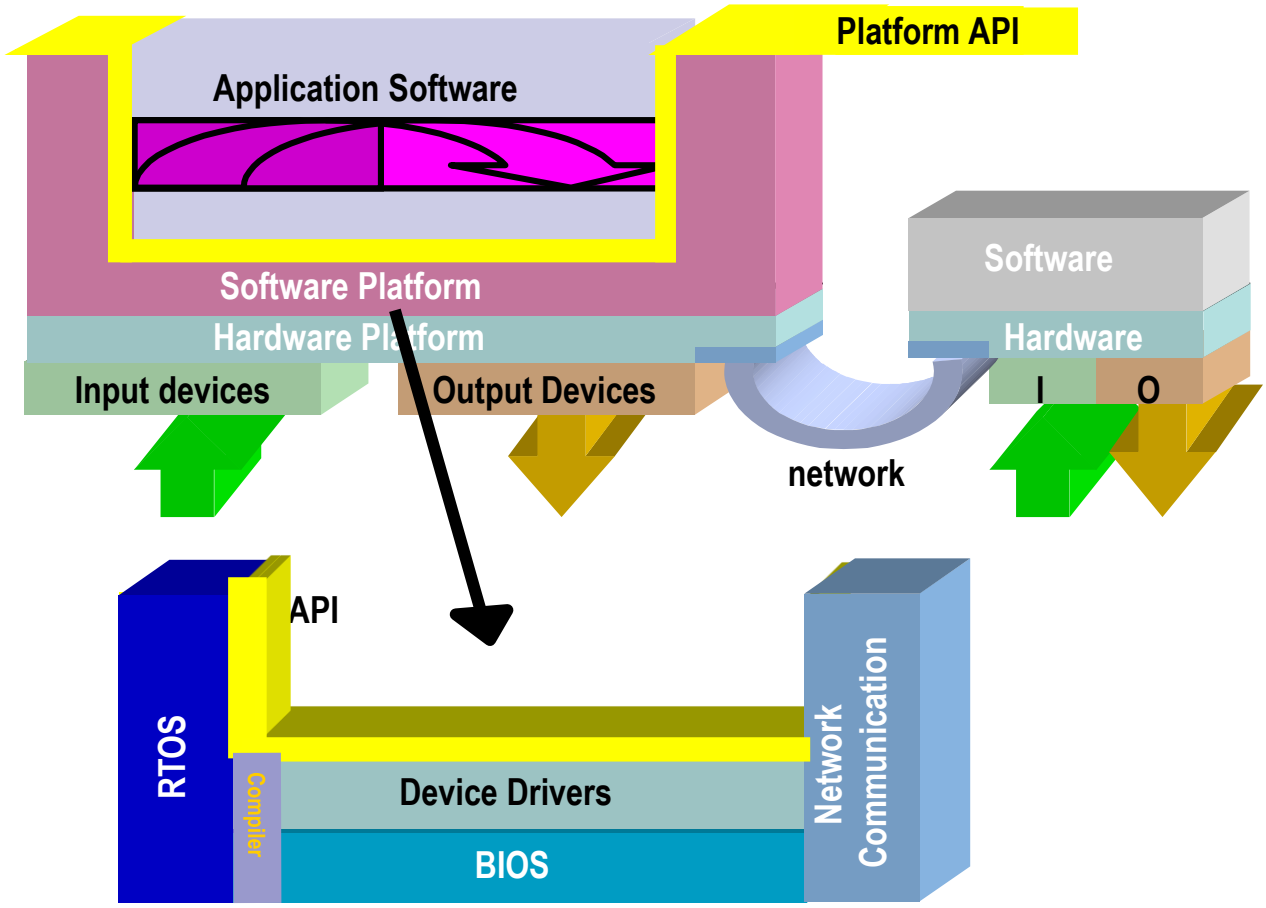


HW-centric view of a Platform



Source: Grant Martin and Henry Chang, "Platform-Based Design: A Tutorial," ISQED 2002, 18 March 2002, San Jose, CA.

SW-Centric View of Platforms

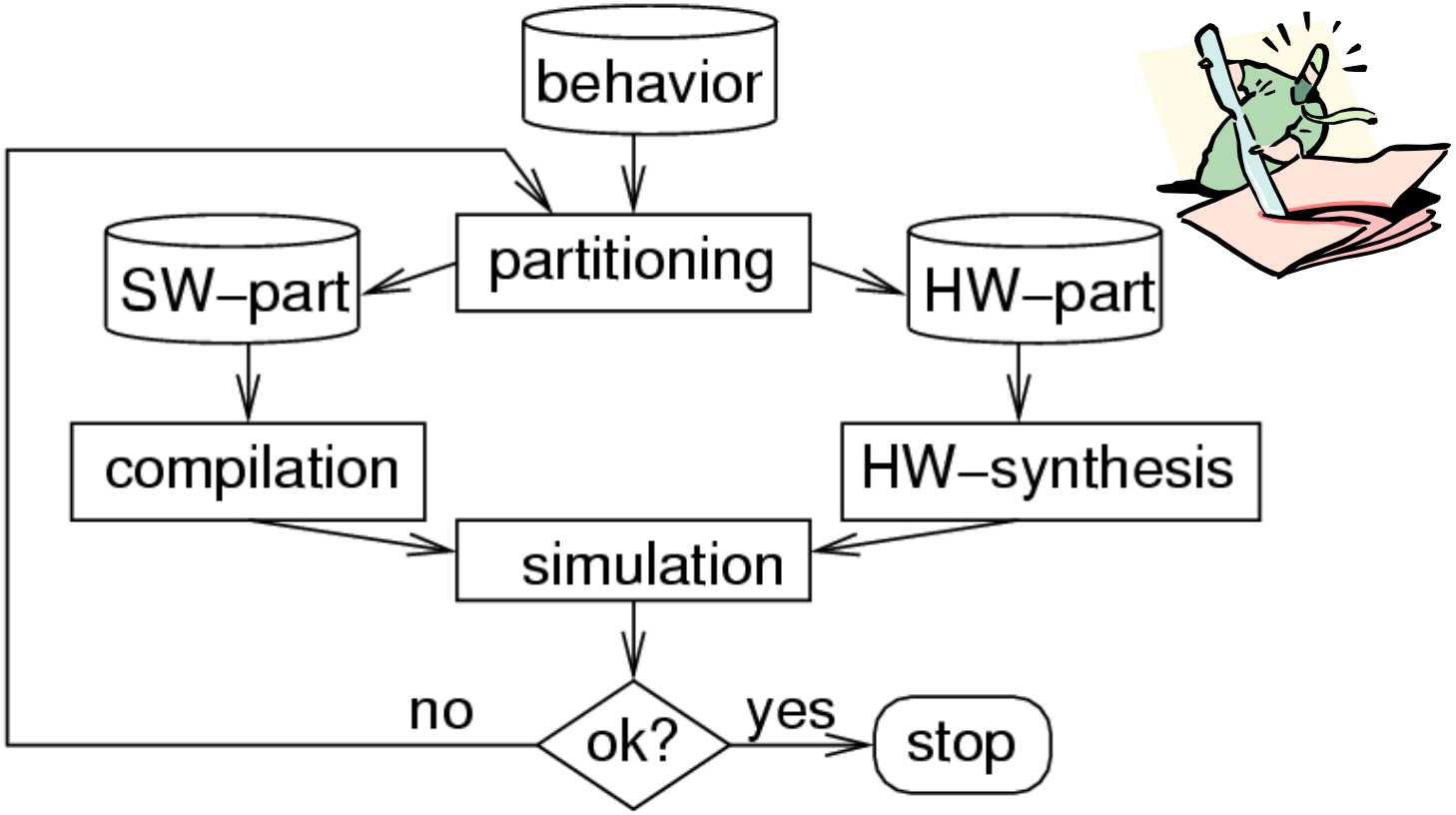


Source: Grant Martin and Henry Chang, "Platform-Based Design: A Tutorial," ISQED 2002, 18 March 2002, San Jose, CA.

HW/SW Codesign: Motivations

- Benefit from both HW and SW
  - HW:
    - Parallelism -> better performance, lower power
    - Higher implementation cost
  - SW
    - Sequential implementation -> great for some problems
    - Lower implementation cost, but often slower and higher power

Software or hardware?

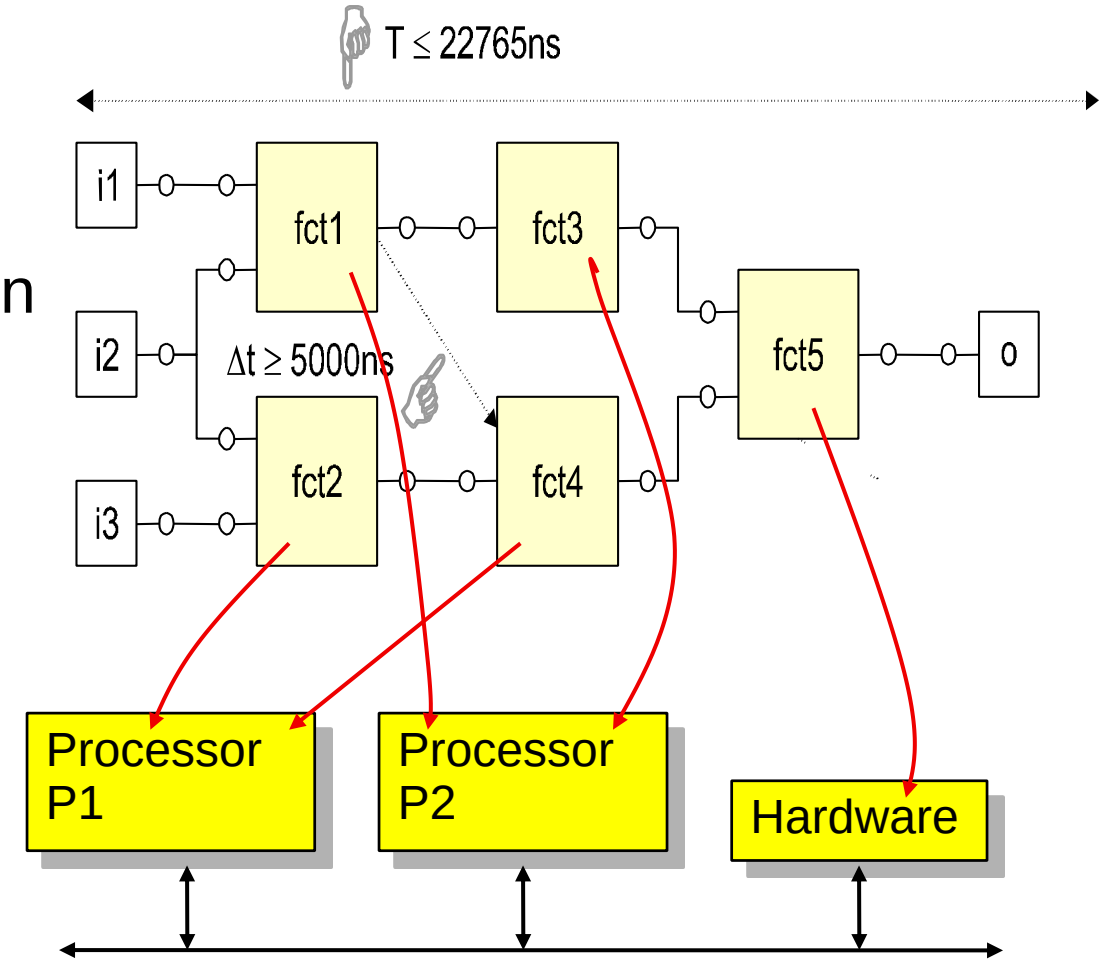


- Decision based on hardware/ software partitioning,

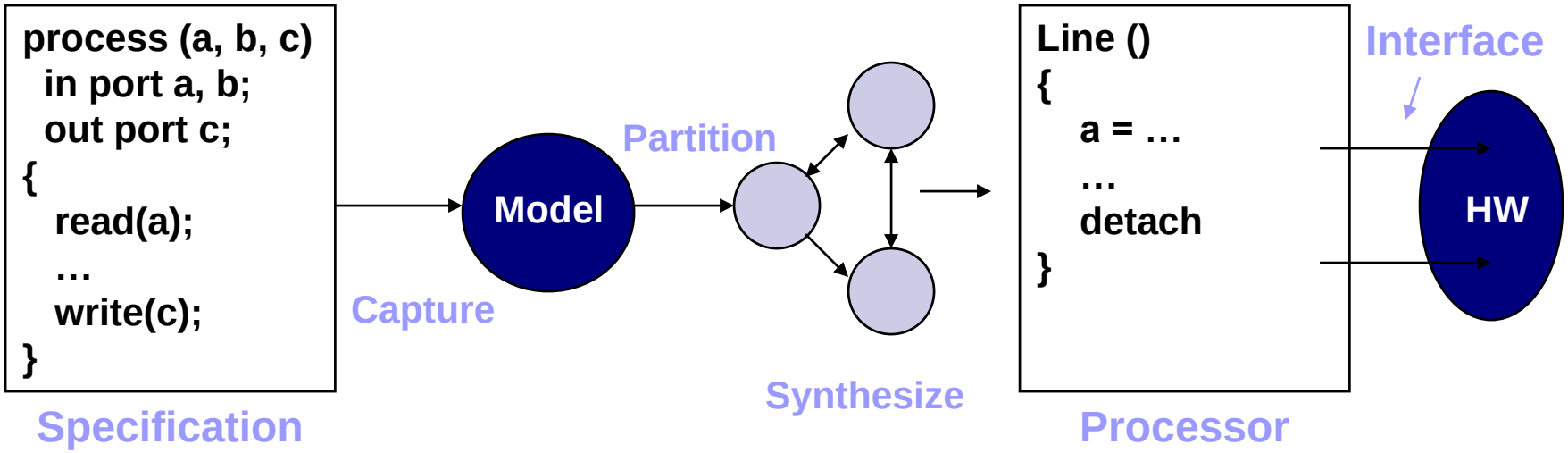
Hardware/software codesign

Specification

Mapping

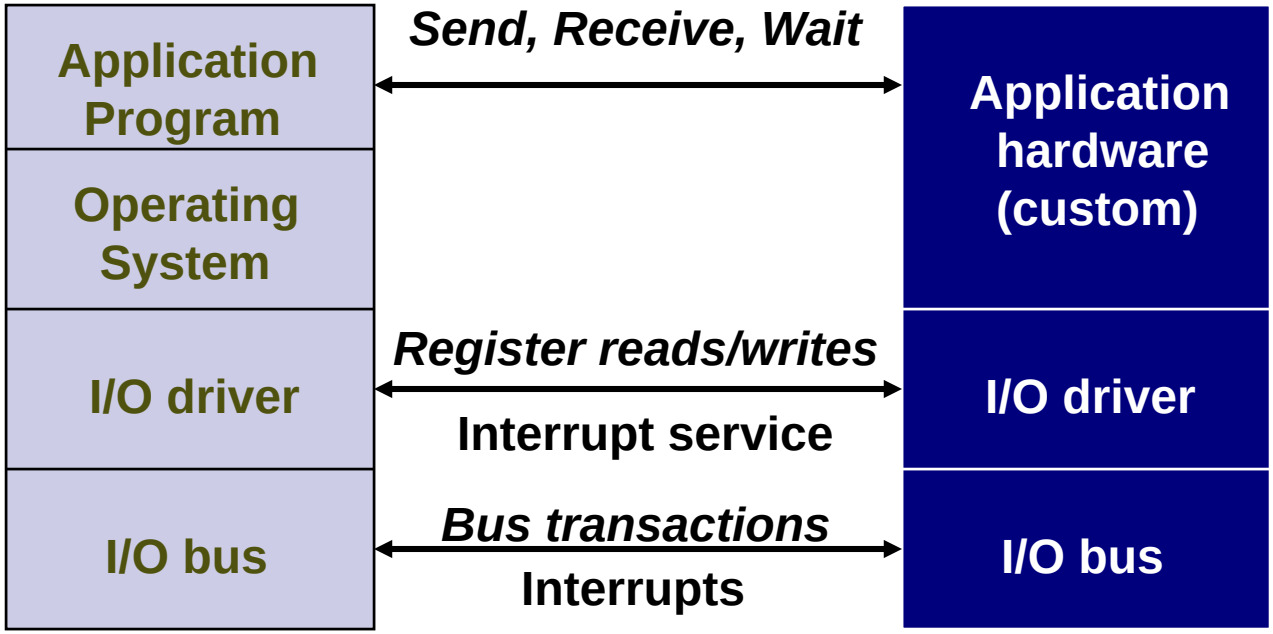


System Partitioning



- Good partitioning mechanism:
  1) Minimize communication across bus
  2) Allows parallelism -> both HW & CPU operating concurrently
  3) Near peak processor utilization at all times

Determining Communication Level

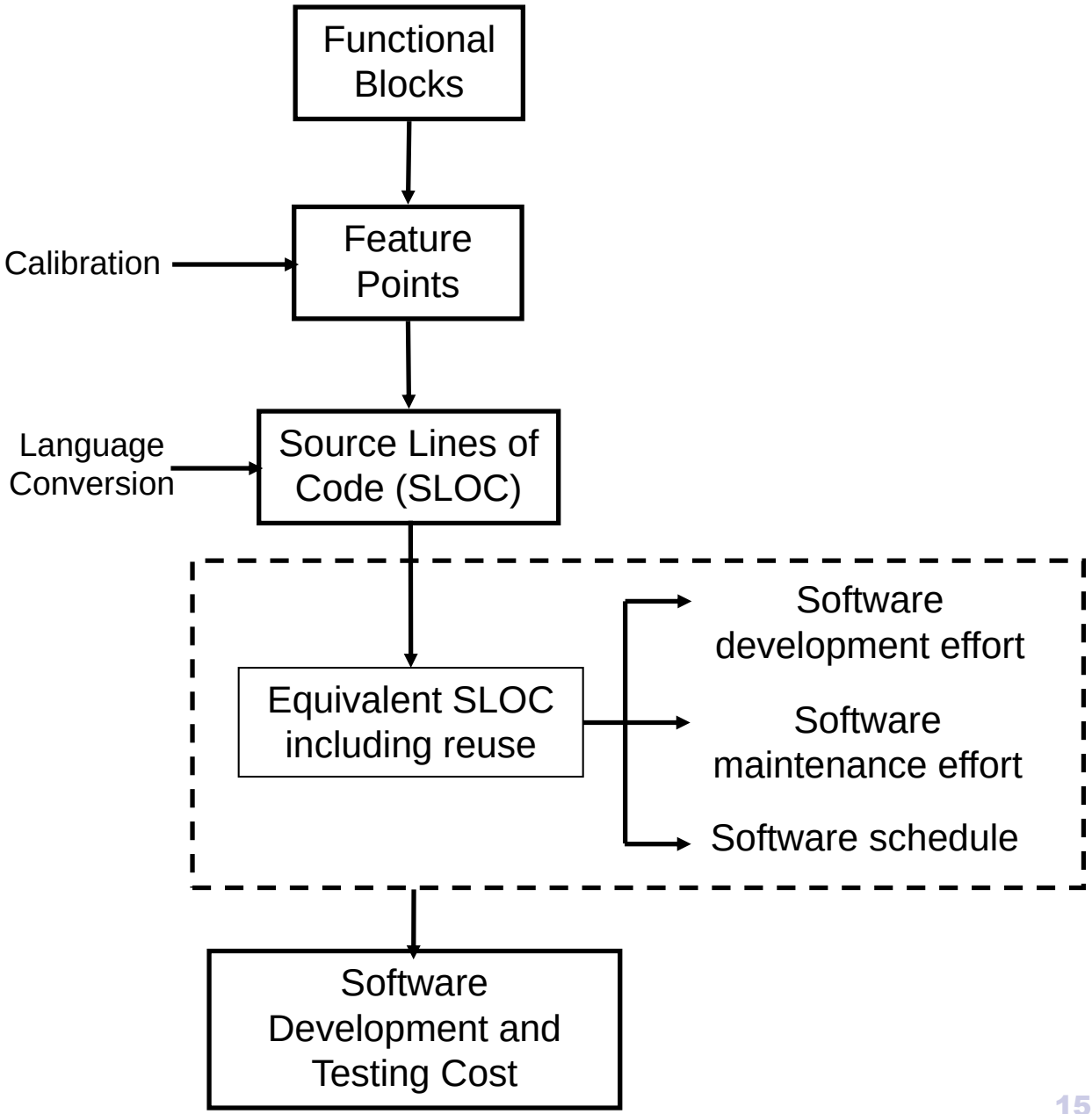


- Easier to program at application level
  - (send, receive, wait) but difficult to predict
- More difficult to specify at low level
  - Difficult to extract from program but timing and resources easier to predict

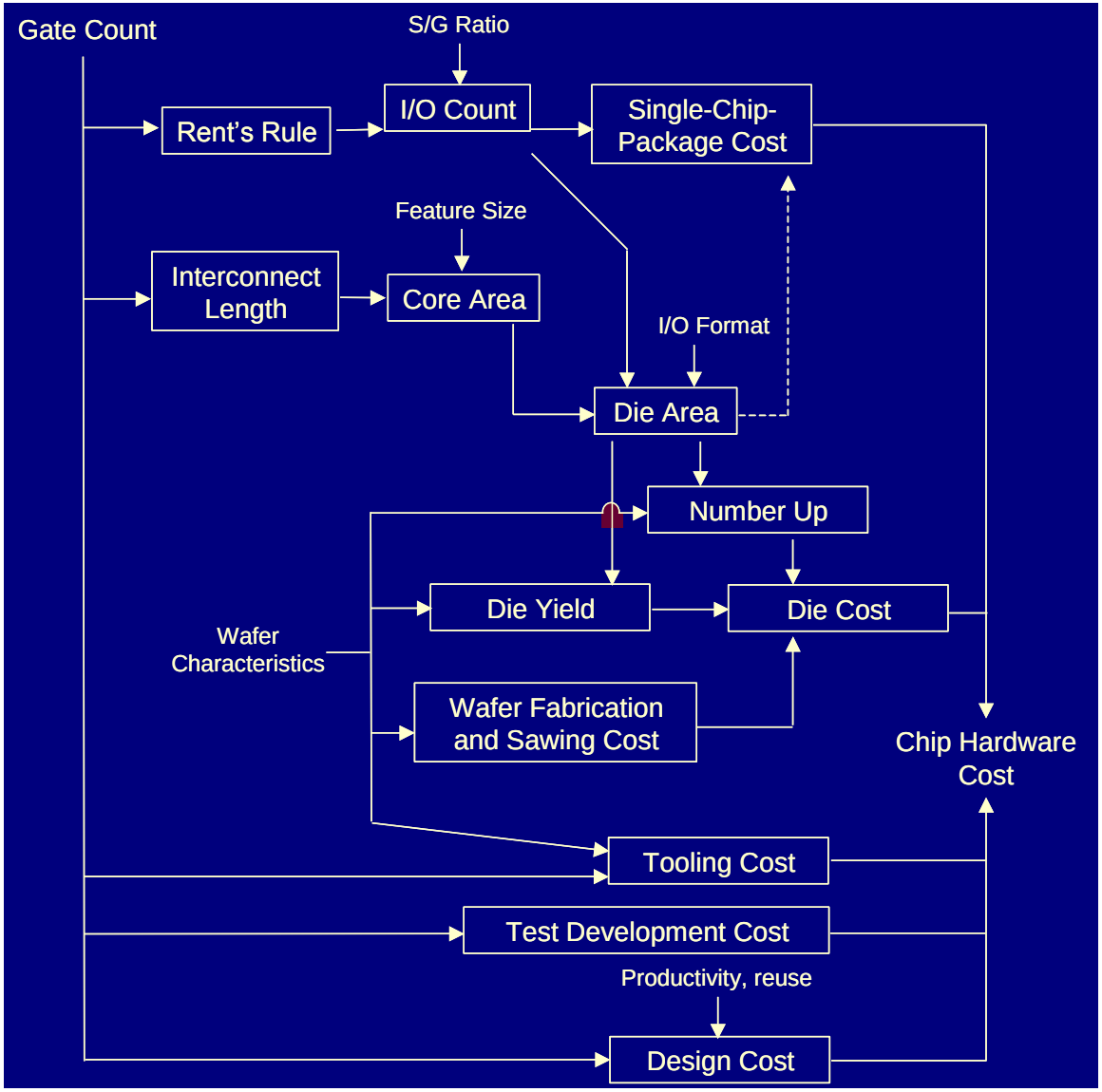
Partitioning Costs

- Software Resources
  - Performance and power consumption
  - Lines of code – development and testing cost
  - Cost of components
- Hardware Resources
  - Fixed number of gates, limited memory & I/O
  - Difficult to estimate timing for custom hardware
  - Recent design shift towards IP
    - Well-defined resource and timing characteristics

Software Cost Analysis Process

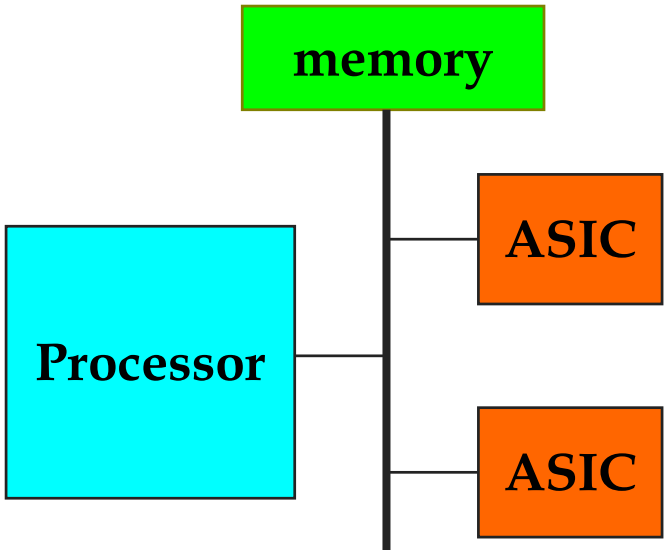


Hardware Cost Analysis Process



Hardware/Software Partitioning

Simple architectural model: CPU + 1 or more ASICs on a bus



- Properties of classic partitioning algorithms
  - Single rate; Single-thread: CPU waits for ASIC
  - Type of CPU is known; ASIC is synthesized

HW/SW Partitioning Styles

- HW first approach
  - start with all-ASIC solution which satisfies constraints
  - migrate functions to software to reduce cost
- SW first approach
  - start with all-software solution which does not satisfy constraints
  - migrate functions to hardware to meet constraints

Partitioning - ILP

Ingredients:

- Cost function
- Constraints

} Involving linear expressions of integer variables from a set X

Cost function $$C = \sum_{x_i \in X} a_i x_i \text{ with } a_i \in R, x_i \in \mathbb{N} \quad (1)$$

Constraints: $$\forall j \in J : \sum_{x_i \in X} b_{i,j} x_i \geq c_j \text{ with } b_{i,j}, c_j \in \mathbb{R} \quad (2)$$

Def.: The problem of minimizing (1) subject to the constraints (2) is called an **integer programming (IP) problem**.

If all x_i are constrained to be either 0 or 1, the IP problem said to be a **0/1 integer programming problem**.

FAQ on integer programming

- Maximizing the cost done by setting $C' = -C$
- Integer programming is NP-complete.
  - Running times increase exponentially with problem size
  - Commercial solvers can solve for thousands of variables
- IP models are a good starting point for modelling even if in the end heuristics have to be used to solve them.

IP model for HW/SW partitioning

- **Notation:**
- Index set *I* denotes task graph nodes.
- Index set *L* denotes task graph node **types**
  e.g. square root, DCT or FFT
- Index set *KH* denotes hardware component **types**.
  e.g. hardware components for the DCT or the FFT.
- Index set *J* of hardware component instances
- Index set *KP* denotes processors.
  All processors are assumed to be of the same type
- T is a mapping from task graph nodes to their types
  $T: I \rightarrow L$

- **Therefore:**
  - $X_{i,k}$: =1 if node v_i is mapped to HW component type $k \in KH$
  - $Y_{i,k}$: =1 if node v_i is mapped to processor $k \in KP$
  - $NY_{\ell,k}$ =1 if at least one node of type ℓ is mapped to processor $k \in KP$

Constraints

- Operation assignment constraints

$$\forall i \in I : \sum_{k \in KH} X_{i,k} + \sum_{k \in KP} Y_{i,k} = 1$$

All task graph nodes have to be mapped either in software or in hardware.

Variables are assumed to be integers.

Additional constraints to guarantee they are either 0 or 1:

$$\forall i \in I : \forall k \in KH : X_{i,k} \leq 1$$

$$\forall i \in I : \forall k \in KP : Y_{i,k} \leq 1$$

Operation assignment constraints

$\forall\forall\ \ell \in L,\ \forall\ i{:}T(v_i)=c_\ell,\ \forall\ k \in KP{:}\ NY_{\ell,k} \geq Y_{i,k}$

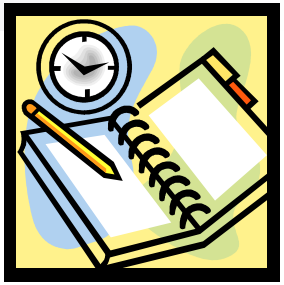
- For all types ℓ of operations and for all nodes *i* of this type:
  - if *i* is mapped to some processor *k*, then that processor must implement the functionality of ℓ.
- Decision variables must also be 0/1 variables:

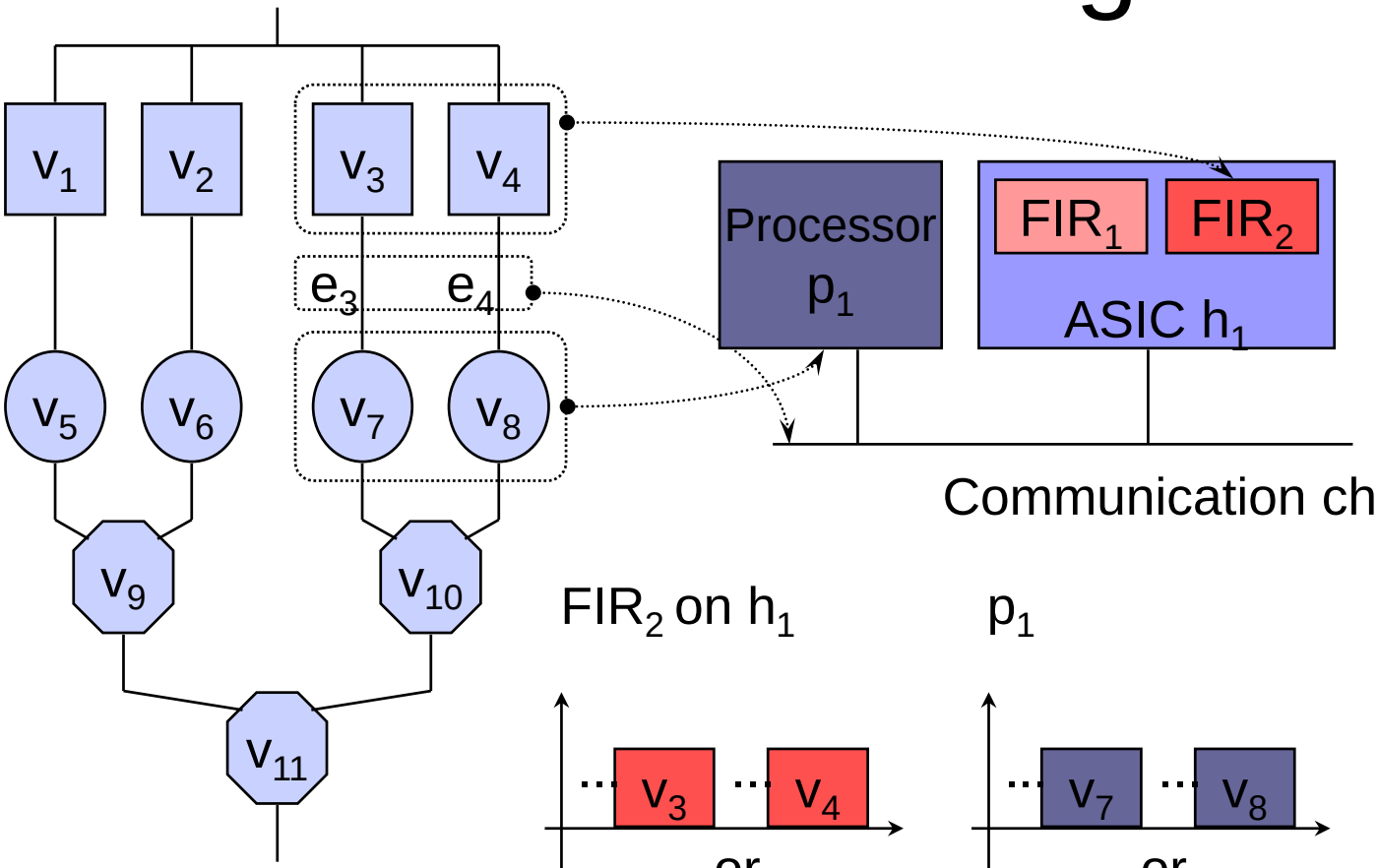
$\forall\forall\ \ell \in L,\ \forall\ k \in KP{:}\ NY_{\ell,k} \leq 1.$

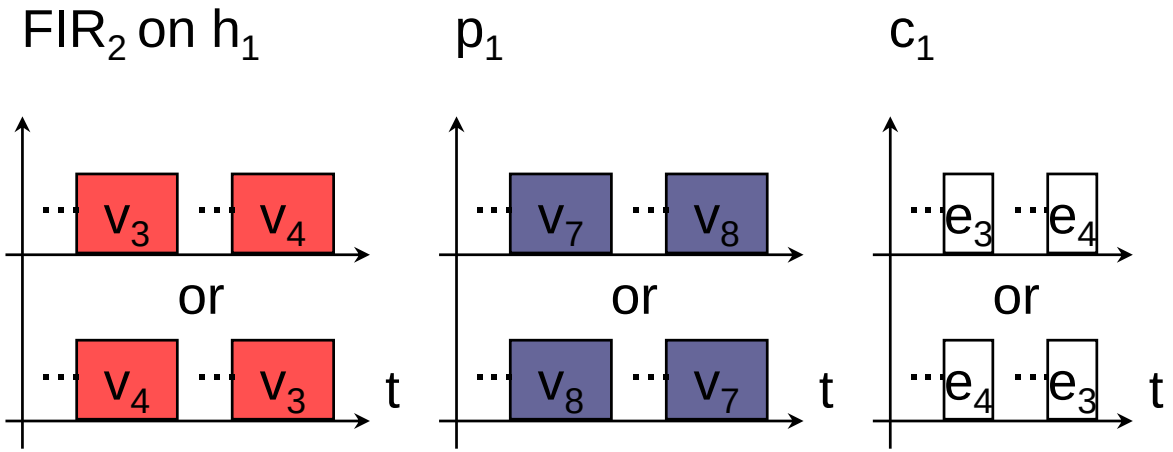
Resource & design constraints

- $\forall\ k \in KH$, the cost for components of that type should not exceed its maximum.
- $\forall\ k \in KP$, the cost for associated data storage area should not exceed its maximum.
- $\forall\ k \in KP$ the cost for storing instructions should not exceed its maximum.
- The total cost ($\Sigma_{k \in KH}$) of HW components should not exceed its maximum
- The total cost of data memories ($\Sigma_{k \in KP}$) should not exceed its maximum
- The total cost instruction memories ($\Sigma_{k \in KP}$) should not exceed its maximum

Scheduling



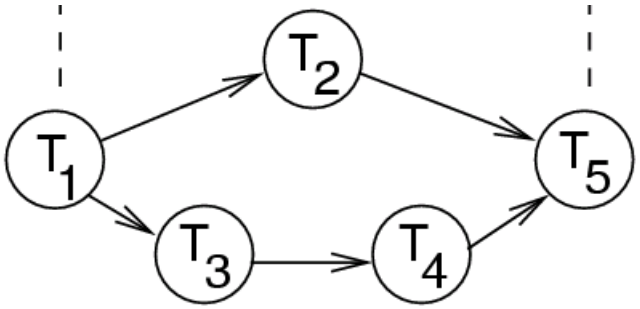

v1
v2
v3
v4
e3
e4
v5
v6
v7
v8
v9
v10
v11
Processor
p1
FIR1
FIR2
ASIC h1
Communication channel c1


FIR2 on h1
p1
c1
v3
v4
or
v4
v3
t
v7
v8
or
v8
v7
t
e3
e4
or
e4
e3
t

Scheduling / precedence constraints

- For all nodes v_{i1} and v_{i2} that are potentially mapped to the same processor or hardware component instance, introduce a binary decision variable $b_{i1,i2}$ with
  $b_{i1,i2}$=1 if v_{i1} is executed before v_{i2} and
  = 0 otherwise.
  Define constraints of the type
  (end-time of v_{i1}) $\leq$ (start time of v_{i2}) if $b_{i1,i2}$=1 and
  (end-time of v_{i2}) $\leq$ (start time of v_{i1}) if $b_{i1,i2}$=0
- Ensure that the schedule for executing operations is consistent with the precedence constraints in the task graph.
- Timing constraints need to be met

Example



- HW types H1, H2 and H3 with costs of 20, 25, and 30.
- Processors of type P.
- Tasks T1 to T5.
- Execution times:

T	H1	H2	H3	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

Operation assignment constraint

T	H1	H2	H3	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

$$\forall i \in I : \sum_{k \in KH} X_{i,k} + \sum_{k \in KP} Y_{i,k} = 1$$

$X_{1,1}+Y_{1,1}=1$ (task 1 mapped to H1 or to P)
$X_{2,2}+Y_{2,1}=1$
$X_{3,3}+Y_{3,1}=1$
$X_{4,3}+Y_{4,1}=1$
$X_{5,1}+Y_{5,1}=1$

Operation assignment constraint

- Assume types of tasks are ℓ =1, 2, 3, 3, and 1.

$\forall\forall\ \ell \in L,\ \forall\ i{:}T(v_i)=c_{\ell},\ \forall\ k \in KP{:}\ NY_{\ell,k} \geq Y_{i,k}$

$$NY_{1,1} \geq Y_{1,1}$$
$$NY_{2,1} \geq Y_{2,1}$$
$$NY_{3,1} \geq Y_{3,1}$$
$$NY_{3,1} \geq Y_{4,1}$$
$$NY_{1,1} \geq Y_{5,1}$$

Functionality 3 to be implemented on processor if node 4 is mapped to it.

Other equations

- Time constraint: Application specific hardware required for time constraints under 100 time units.

T	H1	H2	H3	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

Cost function:

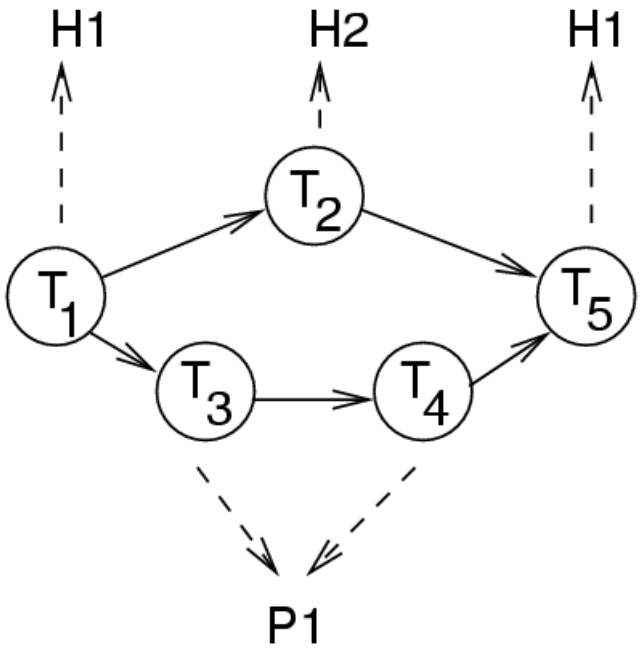
C=20 #(H1) + 25 #(H2) + 30 # (H3) + cost(processor) + cost(memory)

Result

- For a time constraint of 100 time units and cost(P)<cost(H3):

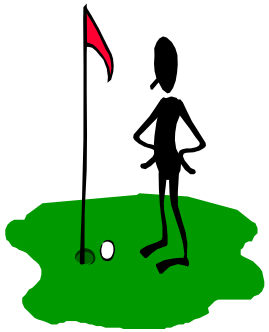
T	H1	H2	H3	P
1	20			100
2		20		100
3			12	10
4			12	10
5	20			100

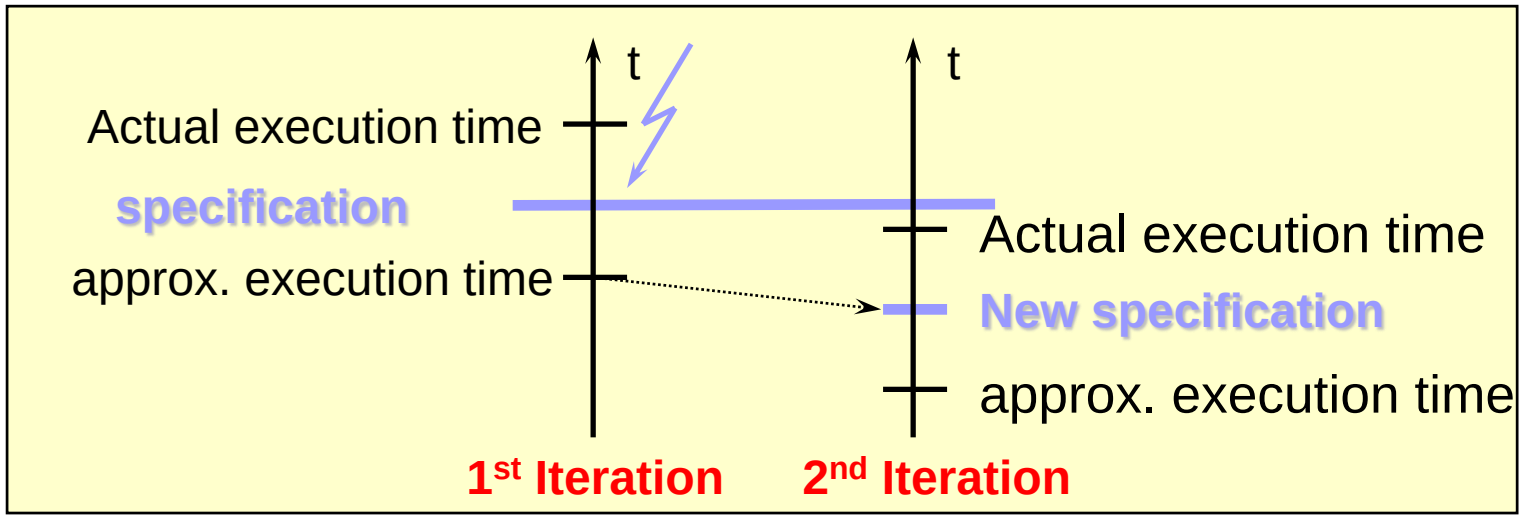
Solution:
T1 → H1
T2 → H2
T3 → P
T4 → P
T5 → H1



Separation of scheduling and partitioning

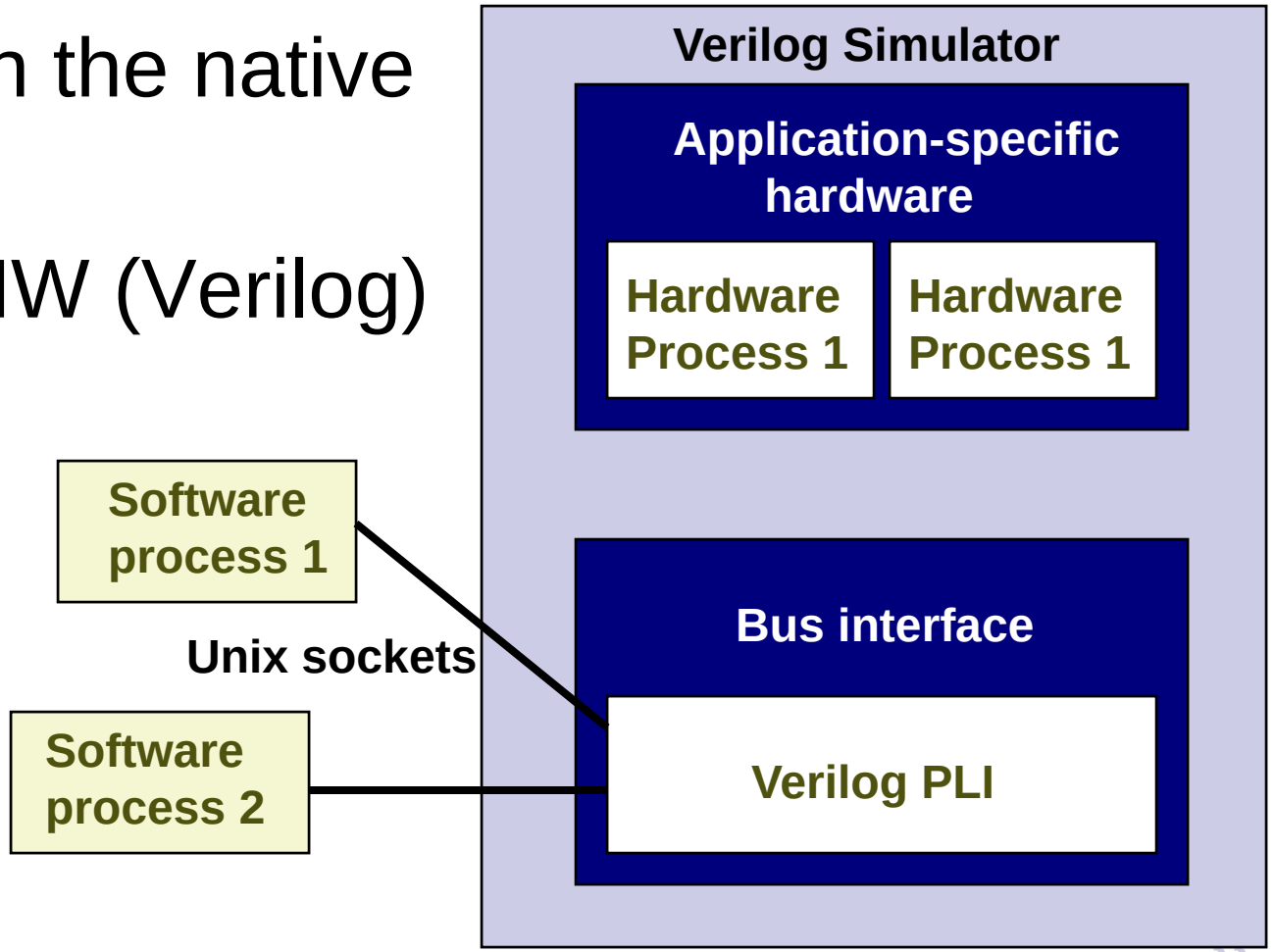
- Combined scheduling/partitioning very complex;
  ☞ Heuristic: Compute estimated schedule
- Perform partitioning for estimated schedule
- Perform final scheduling
- If final schedule does not meet time constraint, go to 1 using a reduced overall timing constraint.



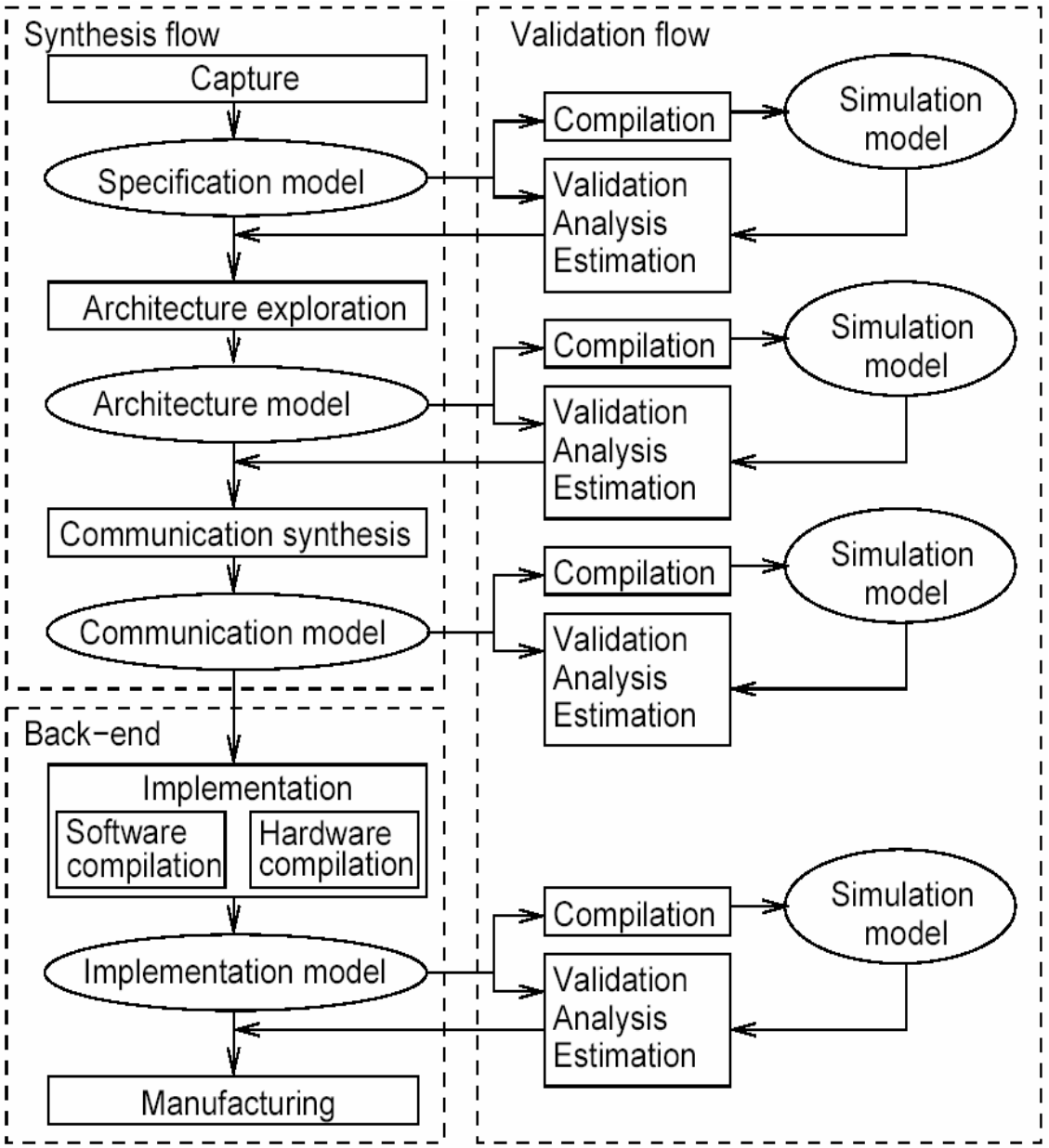


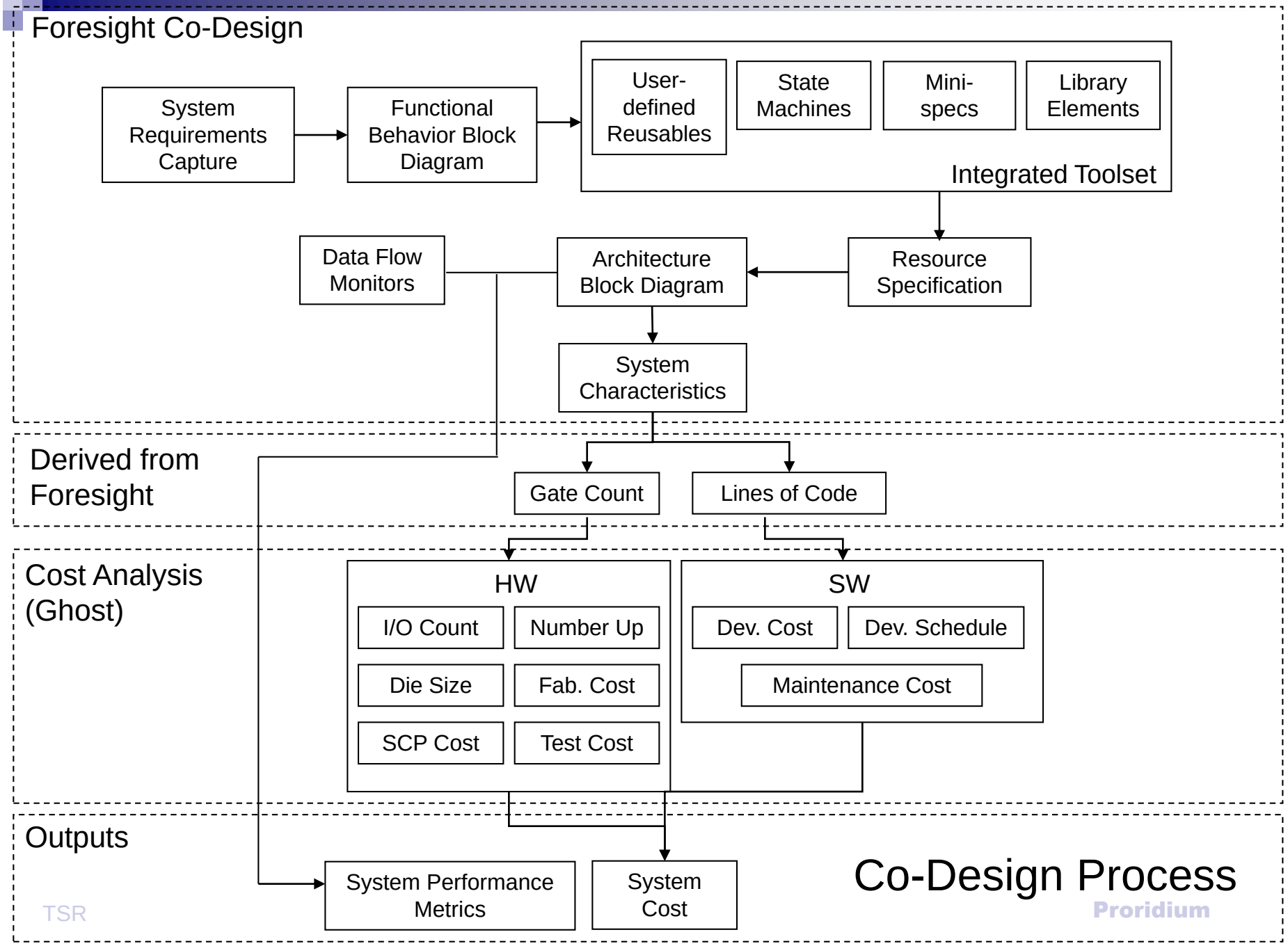
Codesign Verification

- Run SW on the native processor
- Simulate HW (Verilog)



SpecC model


Synthesis flow
Capture
Specification model
Architecture exploration
Architecture model
Communication synthesis
Communication model
Back-end
Implementation
Software compilation
Hardware compilation
Implementation model
Manufacturing
Validation flow
Compilation
Simulation model
Validation Analysis Estimation
Compilation
Simulation model
Validation Analysis Estimation
Compilation
Simulation model
Validation Analysis Estimation
Compilation
Simulation model
Validation Analysis Estimation


Foresight Co-Design
System Requirements Capture
Functional Behavior Block Diagram
User-defined Reusables
State Machines
Mini-specs
Library Elements
Integrated Toolset
Data Flow Monitors
Architecture Block Diagram
Resource Specification
System Characteristics
Derived from Foresight
Gate Count
Lines of Code
Cost Analysis (Ghost)
HW
I/O Count
Number Up
Die Size
Fab. Cost
SCP Cost
Test Cost
SW
Dev. Cost
Dev. Schedule
Maintenance Cost
Outputs
System Performance Metrics
System Cost
Co-Design Process
TSR
Proridium

Industry Initiatives

- Seamless Co-Verification Environment-CVE
- Proridium (Foresight)
  - Customers: Boeing, Microsoft, Raytheon, Oracle etc.
- CoWare (now in Synopsys)
  - Cosimulation and IP integration
  - One of founding members of SystemC (language)
- New FPGA synthesis tools incorporate CPUs
- Platform-based design
  - Platform: predesigned architecture that designers can use to build systems for a given range of applications

Summary

- HW/SW codesign is complicated and limited by performance estimates
- Algorithms are in research and development,
  - much of the work is still done by expert designers

Sources and References

- Peter Marwedel, “Embedded Systems Design,” 2004.
- Giovanni De Micheli @ EPFL
- Vincent Mooney @ Gatech
- Nikil Dutt @ UCI