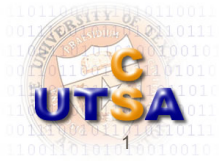
CS4833: Embedded Systems

Architectures – Hardware
(JKP Chapter 1 & web notes)



Review: Embedded Systems

Key points:

- Employs a combination of hardware & software to perform a ***single/specific*** function
- Part of a larger system (not a "computer")
- Works in a *reactive* and *time-constrained* environment.
- Tightly-Constrained: power, size, cost and reliability

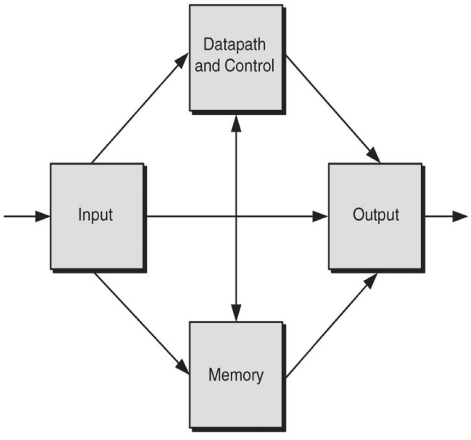
- Software provides features and flexibility
- Hardware = {Processors, ASICs, Memory,...} is used for performance (& sometimes security)

Outline

- Basic components: hardware
  - Transistors and gates
  - Combinational & sequential logic circuits
  - Timers and Counters
- Customized Greatest Common Divisor
- General hardware architectures
  - von Neumann vs. Harvard
- Computing engines in embedded systems
  - Microprocessor, microcomputer, microcontroller
  - Digital signal processors (DSPs)
  - Graphics processing units (GPUs)
- Understand hardware words
  - Data, address or instructions

Basic Architecture: Computing Systems

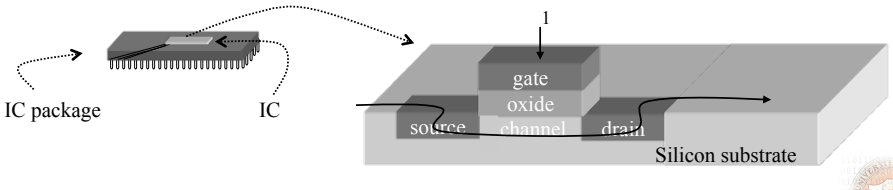
- Processing: **datapath** and **control**
- Interact with outside world: input/output
- Information storage: memory



High-level Block Diagram (at 10,000 feet ☺)

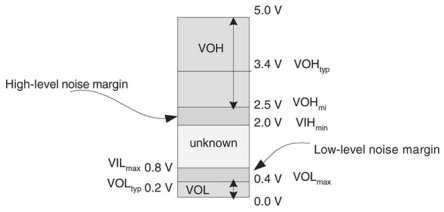
CMOS Transistor on Silicon

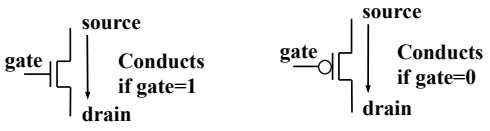
- Complementary Metal Oxide Semiconductor (CMOS)
- Transistor
  - The basic electrical component in digital systems
  - Acts as an on/off switch
  - Voltage at "gate" controls whether current flows from source to drain (by moving electrons within capacitor)
  - Don't confuse this "gate" with a logic gate

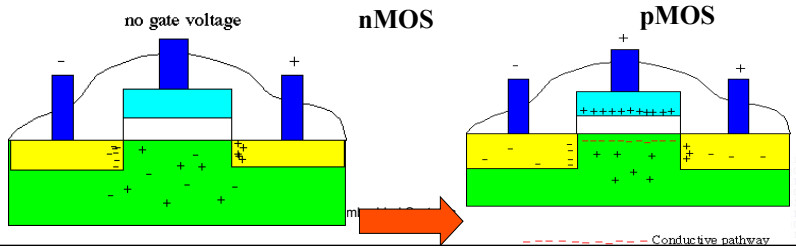


CMOS Transistors

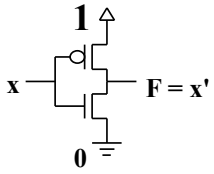
- We refer to logic levels
  - Typically 0 is 0V, 1 is 5V
- Two basic CMOS types
  - nMOS conducts if gate=1
  - pMOS conducts if gate=0
  - Hence "complementary"



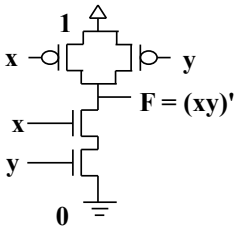




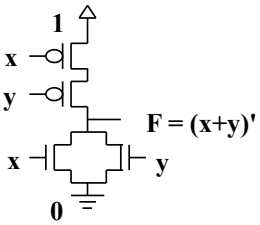
Implementations of CMOS Basic Gates



inverter



NAND gate



NOR gate

x=0 → x'=1
x=1→x'=0

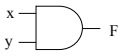
x=0, y=0 → F =1
x=0, y=1 → F =1
x=1, y=0 → F =1
x=1, y=1 → F =0

x=0, y=0 → F =1
x=0, y=1 → F =0
x=1, y=0 → F =0
x=1, y=1 → F =0

Basic Logic Gates

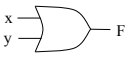
x	F
0	0
1	1

F = x
Driver



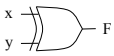
x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

F = x y
AND



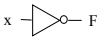
x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

F = x + y
OR



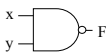
x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

F = x ⊕ y
XOR



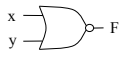
x	F
0	1
1	0

F = x'
Inverter



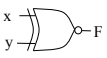
x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

F = (x y)'
NAND



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

F = (x+y)'
NOR



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

F = x ⊙ y
XNOR

Combinational Logic Design

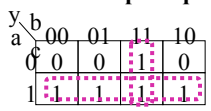
A) **Problem description**
y is 1 if a is to 1, or b and c are 1. z is 1 if b or c is to 1, but not both, or if all are 1.

B) **Truth table**

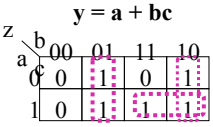
Inputs			Outputs	
a	b	c	y	z
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	1	0
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

C) **Output equations**
y = a'bc + ab'c' + ab'c + abc' + abc
z = a'b'c + a'bc' + ab'c + abc' + abc

D) **Minimize output equations**



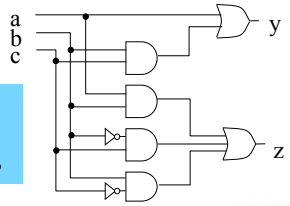
y = a + bc



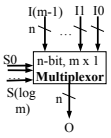
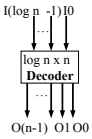
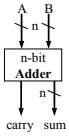
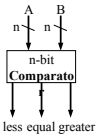
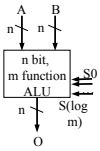
z = ab + b'c + bc'

The output is solely determined by inputs.

E) **Logic Gates**

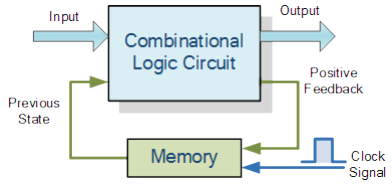


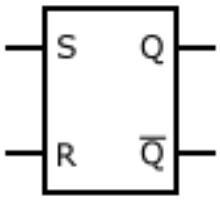
Combinational Components

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

Sequential Logic Circuits: Latch

- **States of circuits: remember past inputs**
  - Store information
- SR *latch* (or simple ***SR flip-flop***)
  - S and R stand for *set* and *reset*
  - The **stored bit** is present on the output marked Q
  - S and R remains low → Q remains unchanged
  - When S is high and R is low: Q is high (1)
  - When R is high and S is low: Q is low (0)



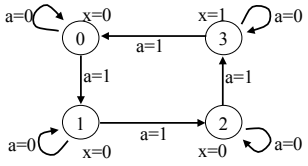


Sequential Logic Design

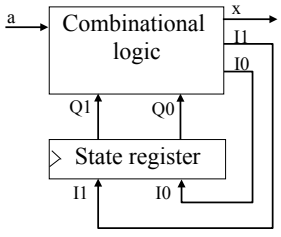
A) Problem Description

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

B) State Diagram



C) Implementation Model



D) State Table (Moore-type)

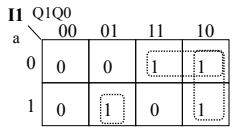
Inputs			Outputs		
Q1	Q0	a	I1	I0	x
0	0	0	0	0	0
0	0	1	0	1	
0	1	0	0	1	0
0	1	1	1	0	
1	0	0	1	0	0
1	0	1	1	1	
1	1	0	1	1	1
1	1	1	0	0	

Convert to combinational logic design

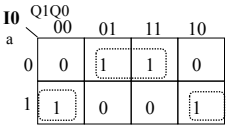
Output depends on both input and previous state.

Sequential Logic Design (cont.)

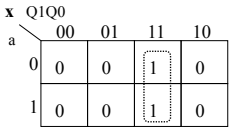
E) Minimized Output Equations



I1 = Q1'Q0a + Q1a' + Q1Q0'

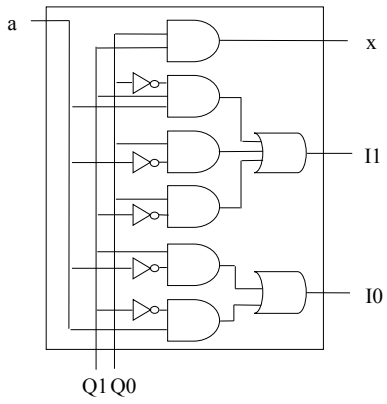


I0 = Q0a' + Q0'a

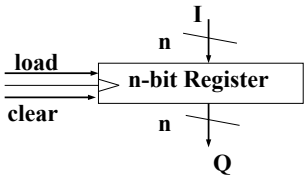


x = Q1Q0

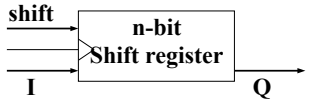
F) Combinational Logic



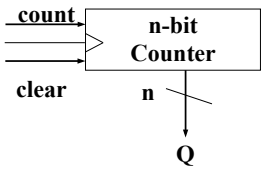
Sequential Components



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



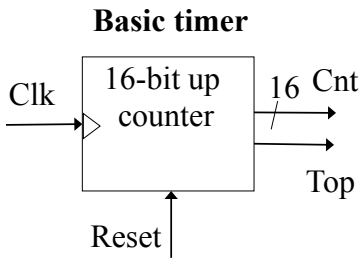
Q = lsb
- Content shifted
- I stored in msb



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

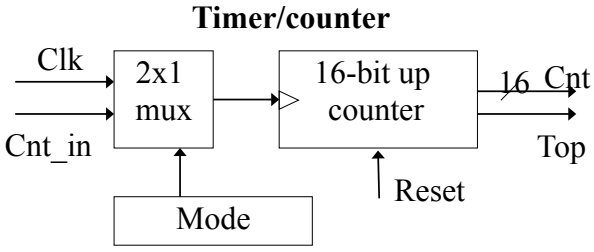
Basic Timers

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



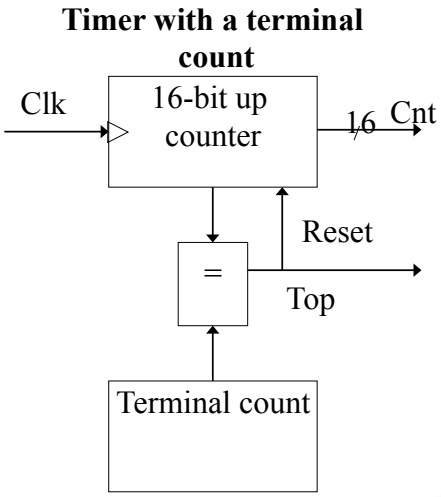
Counters

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



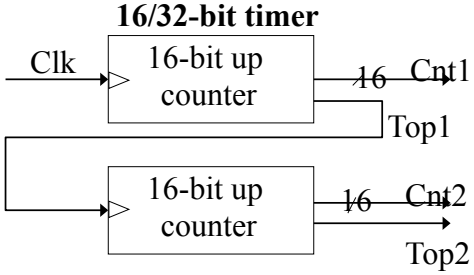
Other Timers

- Interval timer
  - Indicates when desired time interval has passed
  - We set terminal count to desired interval
    - ✓ *Number of clock cycles = Desired time interval / Clock period*

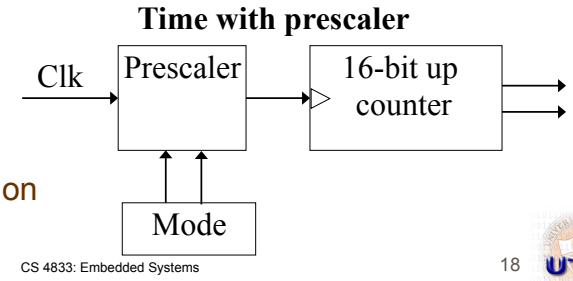


Other Timers (cont.)

- Cascaded counters
  - E.g. larger values



- Prescaler
  - Divides clock
  - Increases range, decreases resolution



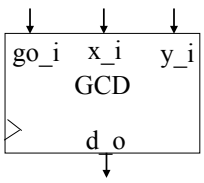
An Example: Greatest Common Divisor

- First create algorithm
- Convert algorithm to "complex" state machine
  - Known as FSMD: **finite-state machine with datapath**

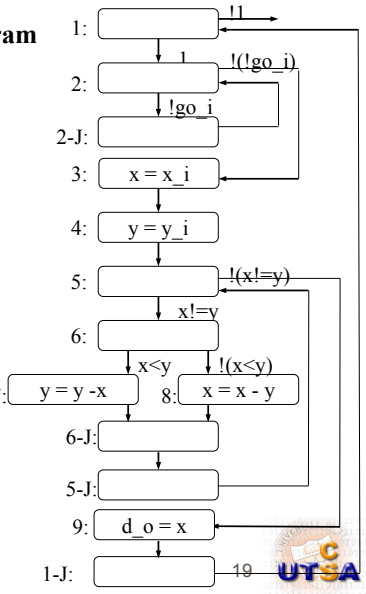
```
0: int  x, y;
1: while (1) {
2:   while (!go_i);
3:   x = x_i;
4:   y = y_i;
5:   while  (x != y)  {
6:      if  (x < y)
7:         y = y - x;
       else
8:         x = x - y;
     }
9:   d_o = x;
   }
```

(b) desired functionality

(a) black-box

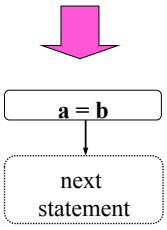


(c) state diagram



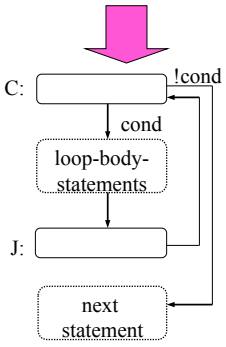
State Diagram Templates

a = b
next statement



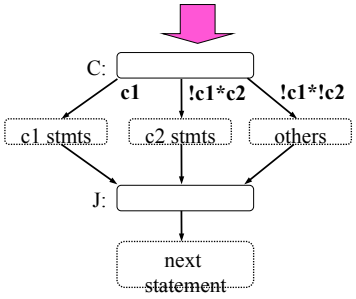
Assignment statement

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



Loop statement

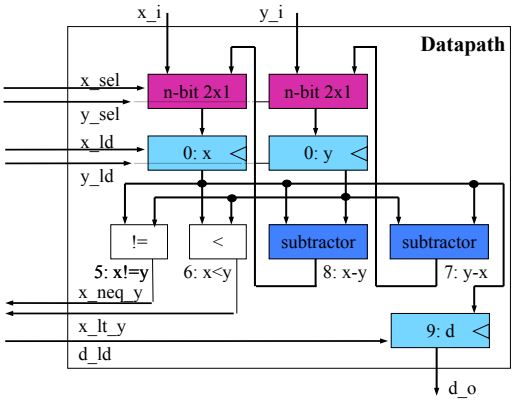
if (c1)
c1 stmts
else if c2
c2 stmts
else
other stmts
next statement



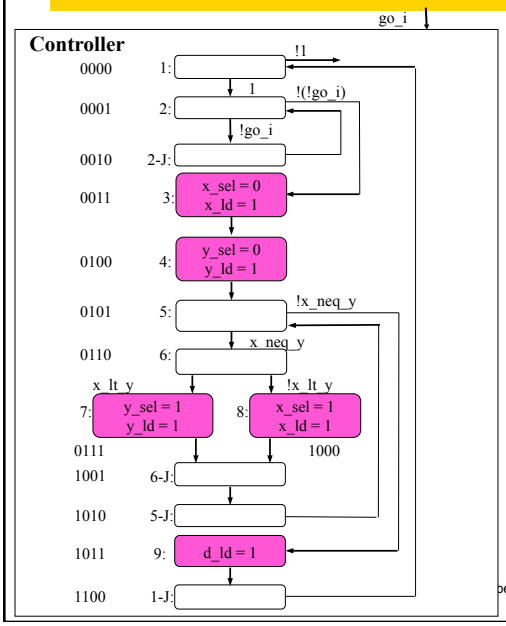
Branch statement

Creating the Datapath

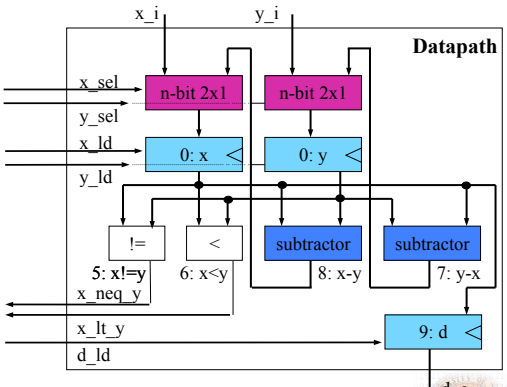
- Create a **register** for any declared variable
- Create a **functional unit** for each arithmetic operation
- Connect the ports, registers and functional units
  - Based on reads and writes
  - Use multiplexors for multiple sources
- Create unique identifier
  - for each datapath component control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



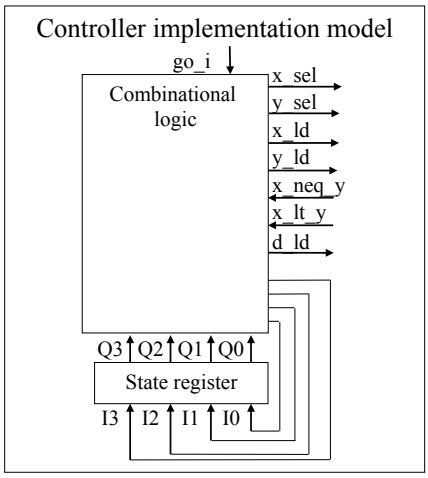
Creating the Controller's FSM

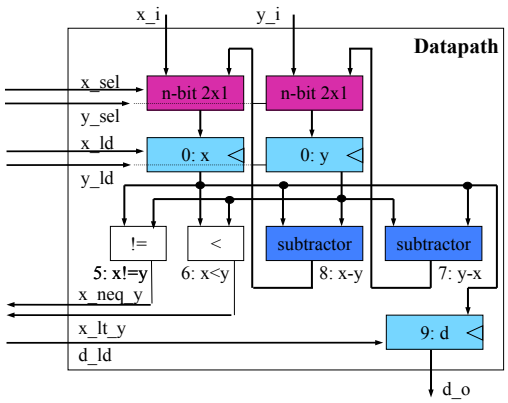


- Same structure as FSMD
- Actions/conditions → **datapath configurations**



Implement & Connect Controller



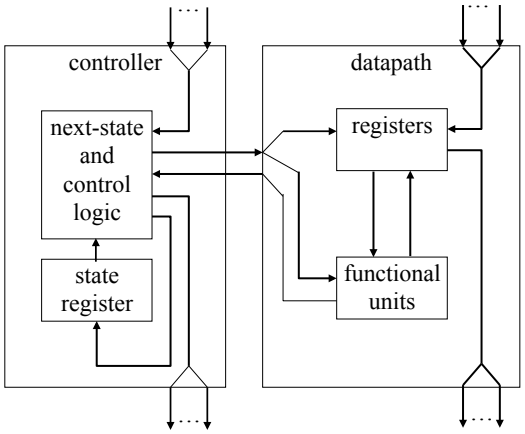


Controller state table for the GCD example

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

Complete GCD Custom Design

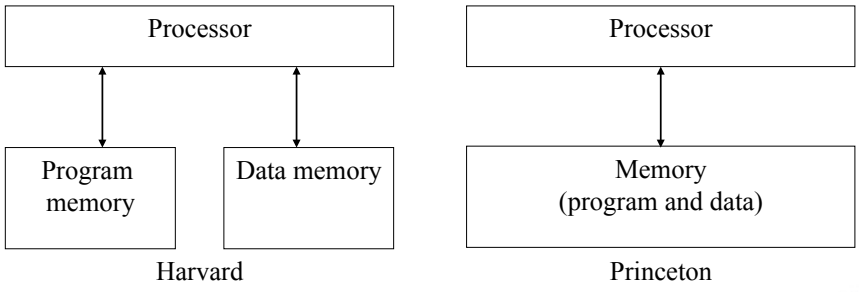
- We finished the datapath
- We have a state table for the next state and control logic
  - All that's left is combinational logic design
- ***Not* an optimized** design, but we see the basic steps



a view inside the controller and datapath

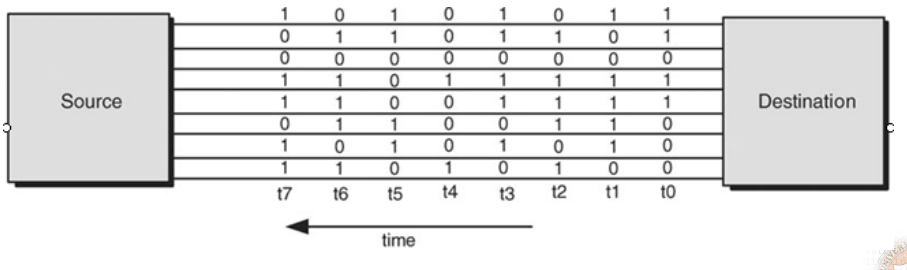
Basic Architectures: Memory

- Princeton (also von Neumann), more common
  - Fewer memory wires (also few pins on processor)
- Harvard: separate memory for instruction and data
  - Simultaneous program and data memory access



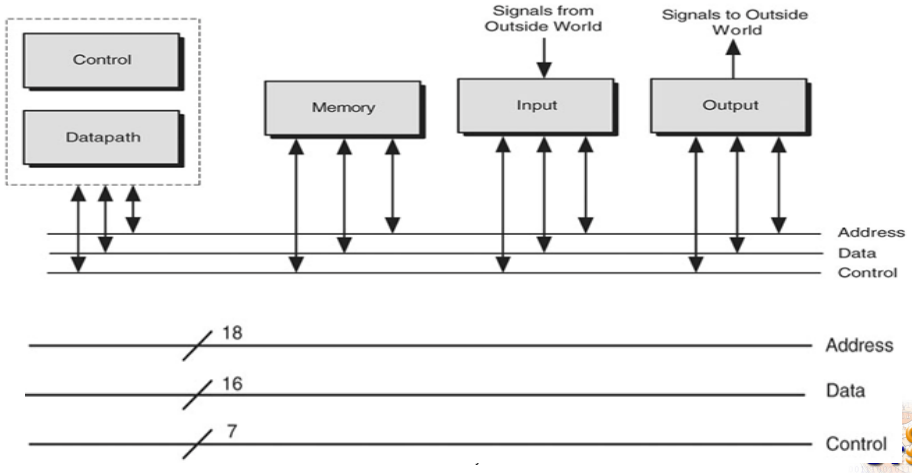
Buses: Connection Between Components

- Bus: a **set of wires** to move signals between components
  - Width: bits/signals can be moved simultaneously
  - Large width → more bits can be moved at the same time
  - 4 → 8 → 16 → 32 → 64 (?)

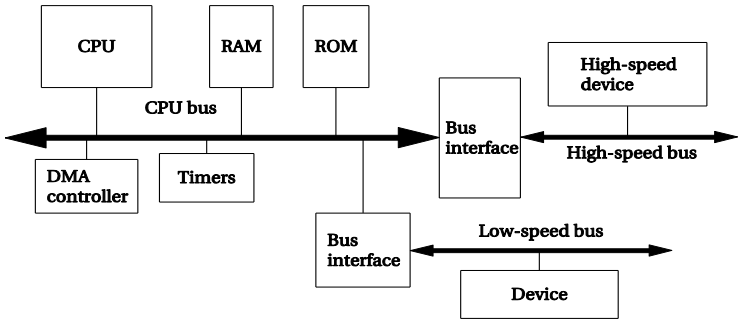


Typical Bus Structure

- Three types of bus for different signals
  - **Address**, **Data** and **Control** signals



Architecture with Multiple Buses



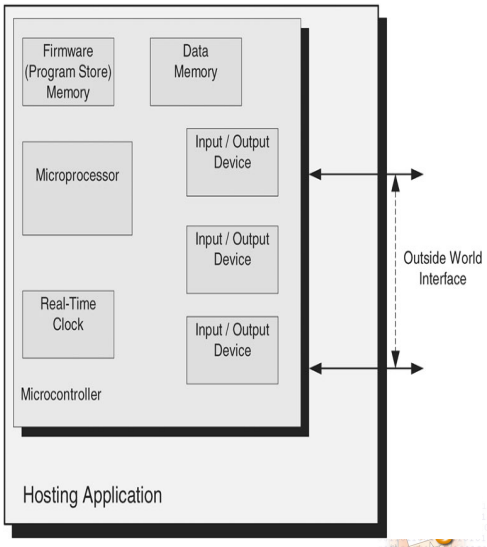
- Multiple busses connect CPU, memory to devices
  - Different speeds: fast vs. slow devices
- DMA provides direct memory access.
- Timers used by OS, devices.

Computing Engines in Embedded Systems

- **Microprocessors**: CPU
  - integrated implementation of central processing unit portion (control and arithmetic and logic)
  - Registers: high speed memory
- **Microcomputers**
  - complete system with basic peripheral functions
- **Microcontrollers**

Microcontroller-Based Systems

- Single integrated circuit: contains rich collection of peripherals and I/O capability
  - Timers
  - A/D and D/A
  - digital I/O,
  - serial or parallel communication
  - DMA control
  - Memory (optional)
  - Eg., ATmega328P (PRIZM)

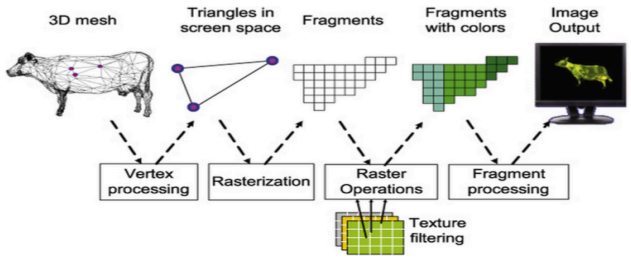


Digital Signal Processors (DSP)

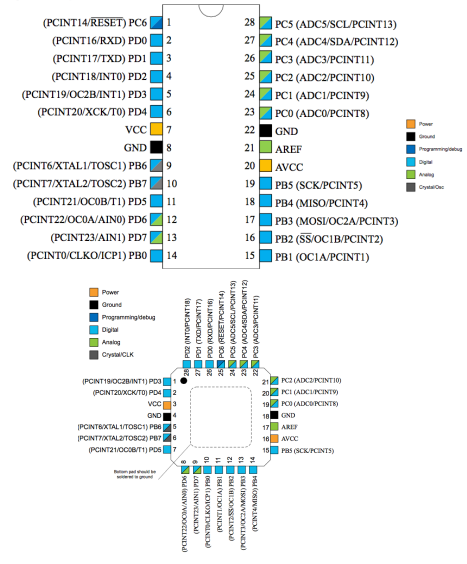
- Special processing of signals w. digital techniques
  - Physical: audio and video, speech, medical (e.g., EKG)
- Fundamental DSP operations
  - Arithmetic computations: multiply, shift, add & division etc.
  - Finite impulse response (FIR)
  - Fast Fourier transformation (FFT)
  - Discrete cosine transformation (DCT)
  - Inverse discrete cosine transformation (IDCT)
- Adopt Harvard architecture: instruction memory
  - Separate high bandwidth data bus

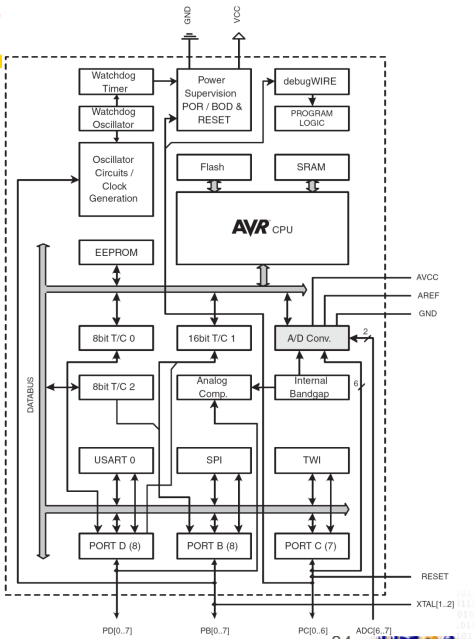
Graphics Processing Units (GPUs)

- Specialized processors for images to display
  - Primary elements: vertex and pixel shaders
- Compute Unified Device Architecture (CUDA)
  - GPGPUs: used for general purpose processing
  - Normally hundreds of simple cores
- Single program multiple data (SPMD)
  - Threading directives and memory addressing



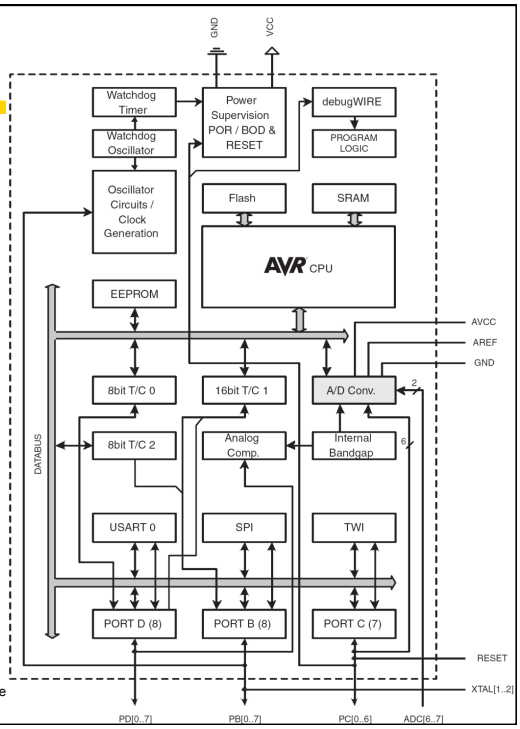
Example: ATmega328P





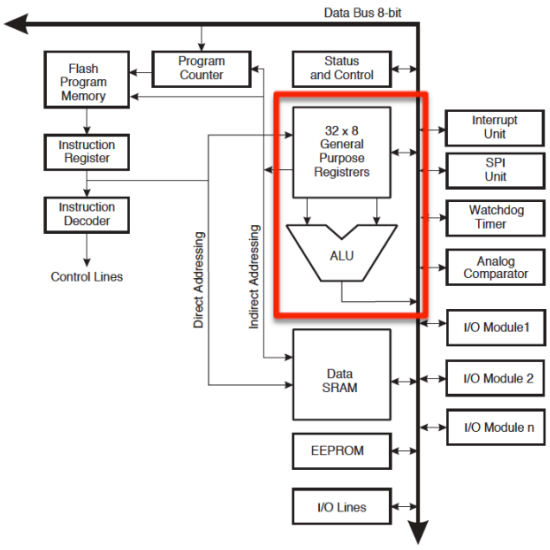
ATmega328P (cont.)

- Clocks and Power
- Flash – program memory: 32 K
- SRAM – data memory: 2K
- I/O etc.

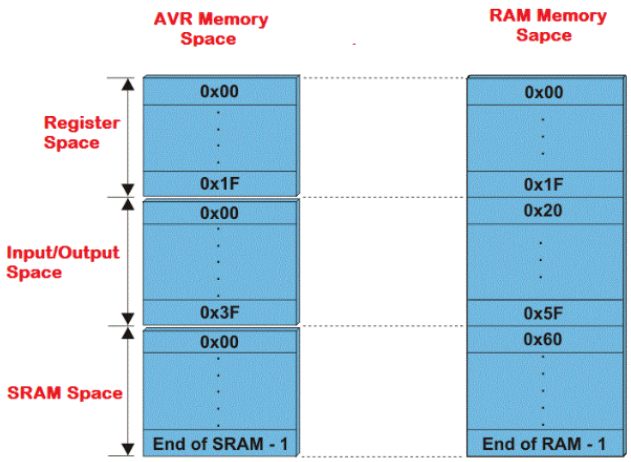


ATmega328P: Inside the CPU

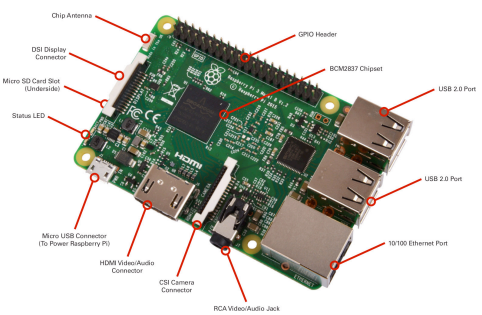
- Instruction fetch and decode
- ALU instructions
  - 32 X 8 bit GP reg.
  - Part SRAM addr.
  - 16 bit reg. X, Y, Z use the last 6 reg.
- I/O & special functions

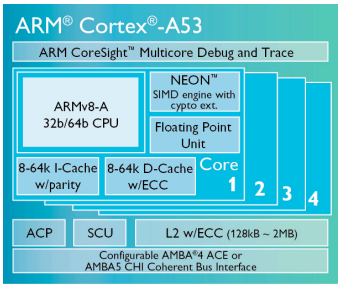


ATmega328P: Memory Space



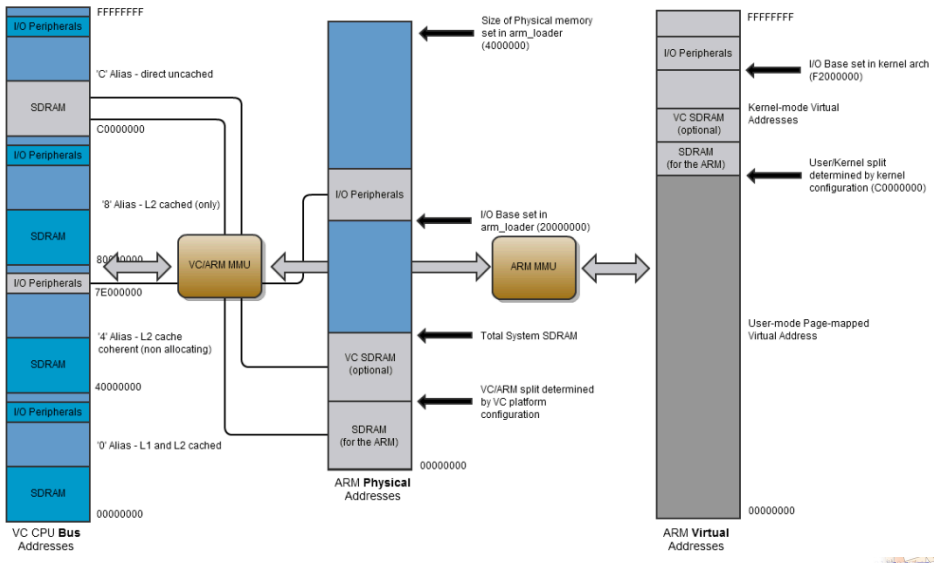
Example: RP3B - BCM2837 (SoC)





- Quad core Cortex-A53 (ARMv8) 1.2 GHz
- Cache: 32kB L1 and 512kB L2
- Dual Core VideoCore IV GPU @ 400 MHz
- 1GB LPDDR2 memory

BCM2837 Address Space



Metric for Processors

- Processors
  - ATmega328P, ARM Cortex-A53, TI C54X, 60X DSPs
- Clock frequency: speed of computation
- Memory: indeterminacy of execution
  - Cache miss: compulsory, conflict, or capacity
- Chip size: small form factor requirements
- Power consumption: dynamic, sleep/idle power
- Specialization: FPGAs, ASICs
- Non-Technical considerations
  - prior expertise, development environment, licensing etc.

Metric for Processors (cont.)

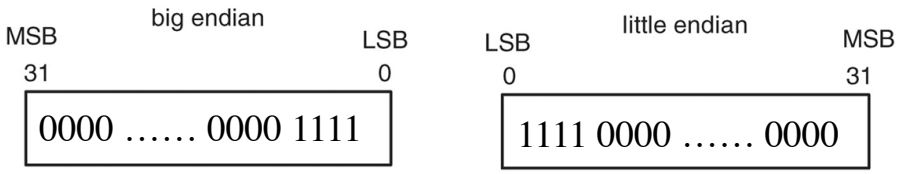
- Parallelism:
  - superscalar pipeline:
    depth & width → latency & throughput
  - Multithreading
    GPU workload requires programming efforts
- Instruction set architecture
  - Complex instruction set computer (CISC)
    - ✓Many addressing modes, more operations per inst.
    - ✓E.g., TI C54x
  - Reduced instruction set computer (RISC)
    - ✓Load/store, easy to pipeline
    - ✓E.g. ARM series
  - Very long instruction word (VLIW)

Performance Evaluations

- Processing speed ?
  - Clock speed: instr. per cycle may differ
  - IPC (instr. Per cycle): work per instr. may differ
- Practical evaluation: benchmarks
  - Dhrystone: synthetic, normalized instr. Count RISC/CISC
  - SPEC: more realistic, oriented to desktops
  - EEMBC: EDN embedded benchmark consortium
    - ✓Suites of benchmarks: automotive, consumer electronics, networking, office automation, telecommunications
  - 0xBehcn: integrated Android benchmarks
    - ✓C library, sys calls, Javascript, graphics etc.

Understand Hardware Words

- Word in hardware
  - Size: number of bits, (for presenting integer)
  - 4, 8, 16, 32 and 64 bits
- Two basic interpretations
  - Big endian: most significant bit on the left (normal number)
  - Little endian: MSB on the right



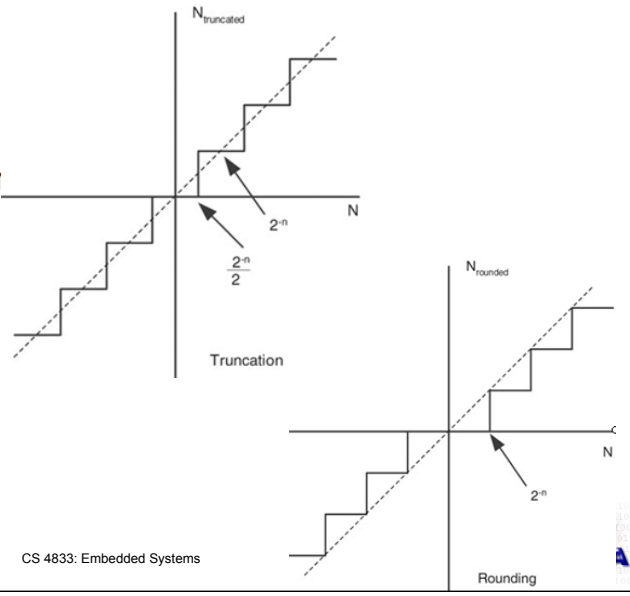
Two different presentation of the number 15

Understand Numbers

- Word size is limited (even 64 bits)
  - Range, resolution, accuracy, errors (and propagation)
- Resolution: smallest value can be represented
  - **Fractional part**: one bit for 0.5; two bits for 0.25 etc
  - 4 bits word to represent value 2.3
  - If 1 bit for fractional part: either 010.0 (2.0) or **010.1 (2.5)**
  - If 2 bits for fractional part: either **10.01 (2.25)** or 10.10 (2.5)
  - N bits for fractional part: $\mathbf{2^{-N}}$
- Errors: representation and propagation

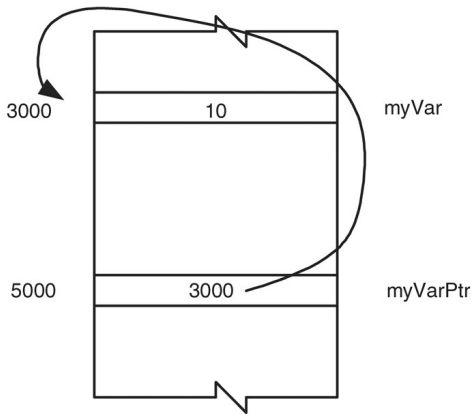
Error Handling: Round or Truncate

- Errors
  - Round:
    $E_R=N_{rounded}-N;$
    $-2^{-(n+1)} < E_R < 2^{-(n+1)}$
    Better propagation
    (average case)
  - Truncation:
    $E_T=N_{truncate}-N;$
    $-2^{-n} < E_T < 0$



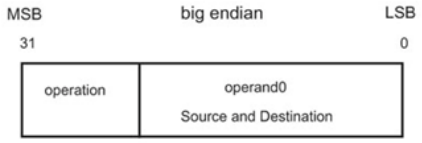
Hardware Word for Address

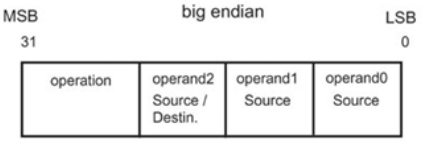
- Address: memory location
- Word size → how much memory can be addressed
- Example
  - *int myVar = 10;*
  - *int* myVarPtr=&myVar;*



Hardware Word for Instruction

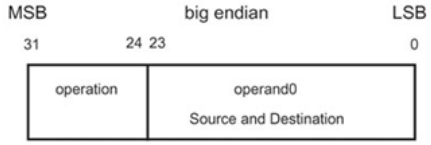
- Instruction
  - Actions/Operation (op-code)
- **Operands**
  - One , two or three
- Types of instructions
  - Arithmetic
  - Logical
  - Data movement
  - Flow of control

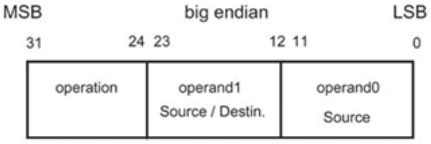


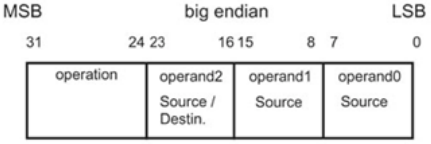


Instructions: Operands

- Represent operands
  - 32 bits instructions
  - 32 bits data: how?!
- Registers
  - Quick hardware components for storing data
  - Set of **registers**: 16 to 256 → **8 bits** to represent one
  - Index by register numbers







Summary

- Basic components: hardware
  - Transistors and gates
  - Combinational and sequential logic circuits
  - Timers and counters
- Customized GCD
- General hardware architectures
  - von Neumann vs. Harvard
- Processing cores in embedded systems
  - Microprocessor, microcomputer, microcontroller
  - Digital signal processors (DSPs)
  - Graphics processing units (GPUs)
- Understand hardware words
  - Data, address or instructions