ECE/CS 5780/6780: Embedded System Design

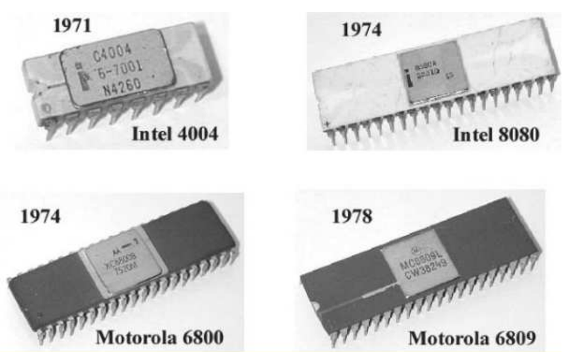
Chris J. Myers

Lecture 1: 68HC12

History of the Microprocessor

- In 1968, Bob Noyce and Gordon Moore left Fairchild Semiconductor and formed Integrated Electronics (Intel).
- At Intel in 1971, Federico Faggin, Ted Hoff, and Stan Mazor invented the first single chip microprocessor, the 4004, a 4-bit microprocessor.
- In 1974, the 8008 and 8080, 8-bit microprocessors, were designed at Intel using NMOS technology.
- In 1974, Motorola also released the MC6800, an 8-bit microprocessor.
- One major difference was that Intel's microprocessors used isolated I/O while Motorola's used memory-mapped I/O.

First Microprocessors



http://www.cpu-world.com

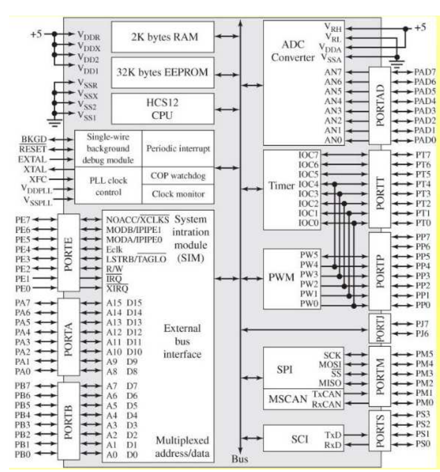
Microcontrollers

- During early 1980s, microcontrollers began to be designed.
- While microprocessors were optimized for speed and memory size, the microcontrollers were optimized for power and physical size.
- Intel produced the 8051 microcontroller.
- Motorola produced the 6805, 6808, 6811, and 6812.
- In 1999, Motorola shipped its 2 billionth MC68HC05 microcontroller.
- In 2004, Motorola spun off its microcontroller division as Freescale Semiconductor.

6811/6812 Architecture

- Instruction sets lend themselves to C compiler implementations.
- Use either two separate 8-bit accumulators (A,B) or one combined 16-bit accumulator (D).
- Have two 16-bit index registers (X,Y).
- Have powerful bit-manipulation instructions.
- Support 16-bit add/subtract, 16×16 integer divide, 16×16 fractional divide, and 8×8 unsigned multiply.
- 6812 also supports 16×16 unsigned/signed multiply, 32×16 unsigned/signed divide, and $32 + (16 \times 16)$ multiply and accumulate.
- 6812 assembly language is a superset of 6811, but they are not machine code compatible and have a different I/O interface.
- Also, their stack pointer operates slightly differently.

MC9S12C32 Block Diagram



Operating Modes

- The 6812 can operate in 1 of 8 modes, but only 3 are important:
  - *Single-chip mode* uses internal memory for program and data.
  - *Expanded narrow mode* allows for use of external 8-bit memory, where PortA is A15-8/D15-8/D7-0 and PortB is A7-A0.
  - *Expanded wide mode* allows for use of external 16-bit memory, where PortA is A15-8/D15-8 and PortB is A7-A0/D7-0.
- NOTE: Our microcontroller can only operate in single-chip mode.

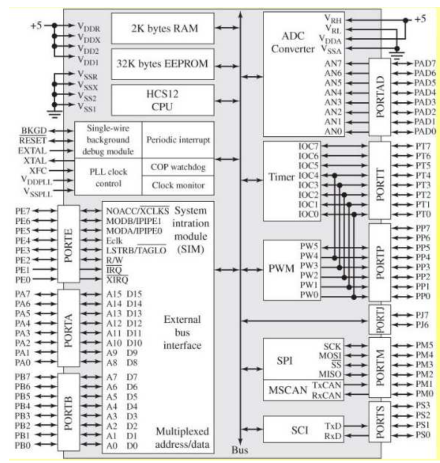
Address Map for MC9S12C32

Address (hex)	Size	Device	Contents
$0000 to $03FF	1K	I/O	
$3800 to $3FFF	2K	RAM	Variables and stack
$4000 to $7FFF	16K	EEPROM	Program and constants
$C000 to $FFFF	16K	EEPROM	Program and constants

External I/O Ports

Port	48-pin	Shared Functions
Port A	PA0	Address/Data Bus
Port B	PB4	Address/Data Bus
Port E	PE7, PE4, PE1, PE0	System Integration Module
Port J	–	Key wakeup
Port M	PM5-PM0	SPI, CAN
Port P	PP5	Key wakeup, PWM
Port S	PS1-PS0	SCI
Port T	PT7-PT0	Timer, PWM
Port AD	PAD7-PAD0	Analog-to-Digital Converter

MC9S12C32 Block Diagram

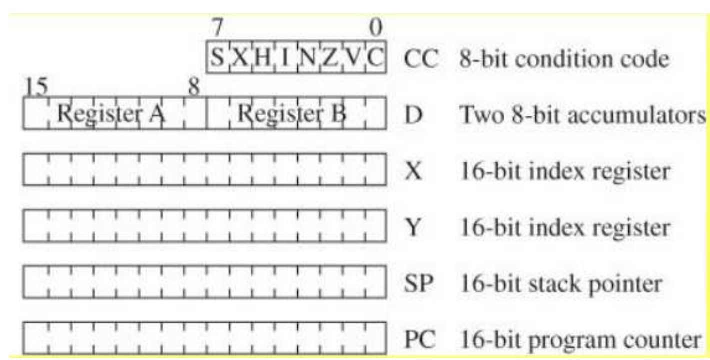


Operating Frequency

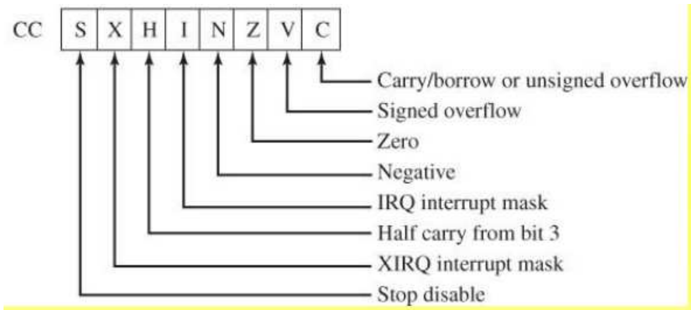
- This program changes the operating frequency from 4 MHz to 24 MHz.

```
void PLL_Init(void){
  SYNR = 0x02;
  REFDV = 0x00;   // PLLCLK = 2*OSCCLK*(SYNR+1)/(REFDV+1)
  CLKSEL = 0x00;
  PLLCTL = 0xD1;
  while((CRGFLG&0x08) == 0){ // Wait for PLLCLK to stabilize.
  }
  CLKSEL_PLLSEL = 1;  // Switch to PLL clock
}
```


Registers



Condition Code Register



Digital Representations of Numbers

- Numbers are represented as a binary sequence of 0's and 1's.
- Each 8-bit byte is stored at a different address.
- A byte can be represented using two hexadecimal digits.

$$\%10110101 \;=\; \$B5 \;\; (0xB5 \text{ in C})$$

b7	b6	b5	b4	b3	b2	b1	b0

$$N = 128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0 \text{ (unsigned)}$$
$$N = -128 \cdot b_7 + 64 \cdot b_6 + 32 \cdot b_5 + 16 \cdot b_4 + 8 \cdot b_3 + 4 \cdot b_2 + 2 \cdot b_1 + b_0 \text{ (signed)}$$

- Only the programmer can keep track if a number is signed or unsigned.
- While addition and subtraction use same hardware, separate hardware is required for multiply, divide, and shift right.
- A byte can also represent a character using the 7-bit ASCII code.

16-Bit Words (Double Bytes)

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0

- Endian comparison for the 16-bit number \$03E8:

Big Endian

Address	Contents
\$0050	\$03
\$0051	\$E8

Little Endian

Address	Contents
\$0050	\$E8
\$0051	\$03

- Freescale microcomputers use the *big endian* approach.

Fixed-Point Numbers

- In embedded systems, *fixed-point* is often preferred over floating point since it is simpler, more memory efficient, and often all that is required.

$$\text{fixed-point number} \equiv I \cdot \Delta$$

where I is a *Variable integer* and Δ is a *Fixed constant*.

- If $\Delta = 10^n$, then called *decimal fixed-point*.
- If $\Delta = 2^n$, then called *binary fixed-point*.
- The value of Δ cannot be changed during program execution, and it likely only appears as a comment in the code.

Precision, Resolution, and Range

- *Precision* is the total number of distinguishable values.
- *Resolution* is the smallest difference that can be represented.
- *Range* is the minimum and maximum values.
- Example: A 10-bit ADC with a range of 0 to $+5$V, has a precision of $2^{10} = 1024$ values, and a resolution of 5V/1024 or about 5mV.
- This could be accurately stored in a 16-bit fixed-point number with $\Delta = 0.001$V.

Overflow and Drop-Out

- *Overflow* occurs when result of calculation is outside of the range.
- *Drop-out* occurs when an intermediate result cannot be represented.
- Example:

$$M = (53 * N)/100 \quad \text{versus} \quad M = 53 * (N/100)$$

- *Promotion* to higher precision avoids overflow.
- Dividing last avoids drop-out.

Fixed-Point Arithmetic

- Let $x = I \cdot \Delta$, $y = J \cdot \Delta$, $z = K \cdot \Delta$.

$$
\begin{array}{lll}
z = x + y & K = I + J & (\textit{addition}) \\
z = x - y & K = I - J & (\textit{subtraction}) \\
z = x \cdot y & K = (I \cdot J)/\Delta & (\textit{multiplication}) \\
z = x/y & K = (I \cdot \Delta)/J & (\textit{division})
\end{array}
$$

- If Δ is different, then must first convert one of the two numbers to use the Δ of the other.
- If Δ is different, binary fixed-point is more convenient as conversion can be done with shifting rather than multiplication/division.

Notation

- w is 8-bit signed (-128 to +127) or unsigned (0 to 255)
- n is 8-bit signed (-128 to +127)
- u is 8-bit unsigned (0 to 255)
- W is 16-bit signed (-32787 to +32767) or unsigned (0 to 65535)
- N is 16-bit signed (-32787 to +32767)
- U is 16-bit unsigned (0 to 65535)
- $= [addr]$ specifies an 8-bit read from address
- $= \{addr\}$ specifies a 16-bit read from address (big endian)
- $= < addr >$ specifies a 32-bit read from address (big endian)
- $[addr] =$ specifies an 8-bit write to address
- $\{addr\} =$ specifies a 16-bit write to address (big endian)
- $< addr > =$ specifies a 32-bit write to address (big endian)

Assembly Language

- Assembly language instructions have four fields:

Label	Opcode	Operand(s)	Comment
`here`	`ldaa`	`$0000`	`RegA = [$0000]`
	`staa`	`$3800`	`[$3800] = RegA`
	`ldx`	`$3802`	`RegX = {$3802}`
	`stx`	`$3804`	`{$3804} = RegX`

- Assembly instructions are translated into machine code:

Object code	Instruction	Comment
`$96 $00`	`ldaa $0000`	`RegA = [$0000]`

Simple Addressing Modes

- Inherent addressing mode (INH)
- Immediate addressing mode (IMM)
- Direct page addressing mode (DIR)
- Extended addressing mode (EXT)
- PC relative addressing mode (REL)

Inherent Addressing Mode

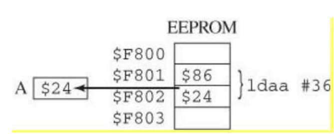
- Uses no operand field.

Obj code	Op	Comment
`$3F`	`swi`	`Software interrupt`
`$87`	`clra`	`RegA = 0`
`$32`	`pula`	`RegA = [RegSP]; RegSP=RegSP+1`

Immediate Addressing Mode

- Uses a fixed constant.
- Data is included in the machine code.

Obj code	Op	Operand	Comment
`$8624`	`ldaa`	`#36`	`RegA = 36`

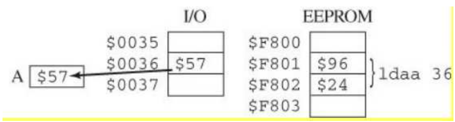


- What is the difference between `ldaa #36` and `ldaa #$24`?

Direct Page Addressing Mode

- Uses an 8-bit address to access from addresses 0 to $00FF.
- This is RAM in 6811 and I/O in 6812.

Obj code	Op	Operand	Comment
`$9624`	`ldaa`	`36`	`RegA = [$0036]`

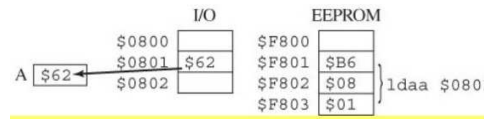


- What is the difference between `ldaa $12` and `ldx $12`?

Extended Addressing Mode

- Uses a 16-bit address to access all memory and I/O devices.

Obj code	Op	Operand	Comment
`$B60801`	`ldaa`	`$0801`	`RegA = [$0801]`



- < forces direct addressing and > forces extended addressing.
- What is the difference between `ldaa $0801` and `ldaa <$0801`?
- What is the difference between `ldaa $01` and `ldaa >$01`?

PC Relative Addressing Mode

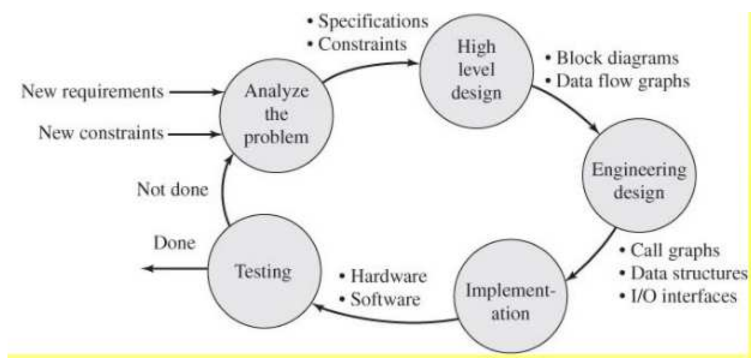
- Used for branch and branch-to-subroutine instructions.
- Stores 8-bit signed relative offset from current PC rather than absolute address to branch to.

rr = (destination address) − (location of branch) − (size of the branch)

- Assume branch located at $F880.

Obj code	Op	Operand	Comment
`$20BE`	`bra`	`$F840`	`$F840` − `$F880` − 2 = −`$42` = `$BE`
`$2046`	`bra`	`$F8C8`	`$F8C8` − `$F880` − 2 = `$46`

Top-Down Design Process



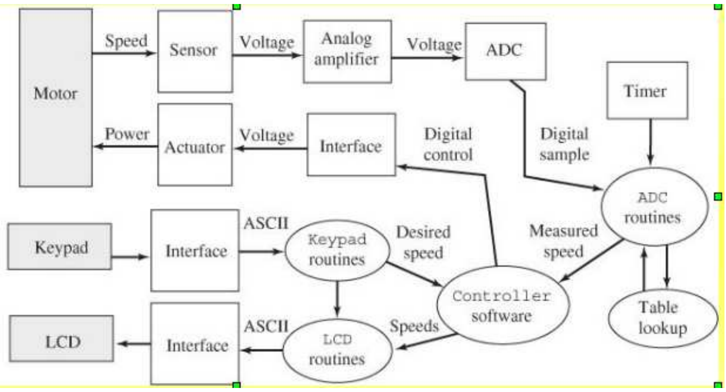
Analysis Phase

- Discover the requirements and constraints for our proposed system.
- *Requirements* are general parameters that the system must satisfy.
- *Specification* are detailed parameters.
- *Constraints* are limitations under which the system must operate.
- Issues that should be considered are:
  - Safety.
  - Accuracy, precision, resolution.
  - Response time, bandwidth.
  - Maintainability, testability, compatibility.
  - Mean time between failure.
  - Size, weight, power.
  - Nonrecurring engineering cost (NRE cost), unit cost.
  - Time-to-prototype, time-to-market
  - Human factors

High-Level Design Phase

- Build a conceptional model of the hardware and software system.
- Design broken into modules or subcomponents.
- Estimate cost, schedule, and expected performance.
- Develop a *data flow graph* for the system.

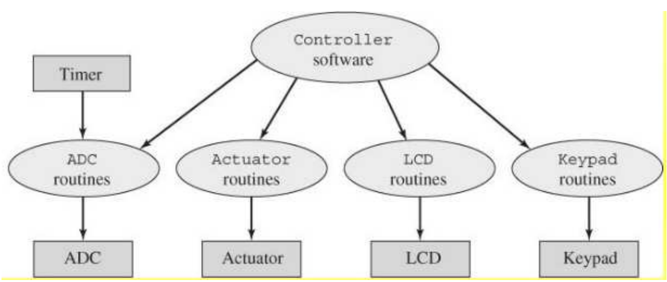
Data Flow Graph for a Motor Controller



Engineering Design Phase

- Construct a preliminary design.
- This should include the hierarchical structure, basic I/O signals, shared data structures, and overall software scheme.
- Build mock-ups of mechanical parts and user software interface.
- *Call graphs* can be used to show how software and hardware interact.

Call Graph for a Motor Controller



Implementation Phase

- During this phase, the design is actually built.
- Implementation of subcomponents may actually be started during the earlier phases.
- Debugging embedded systems can be very difficult.
- Therefore, extensive use of hardware/software simulation is essential.

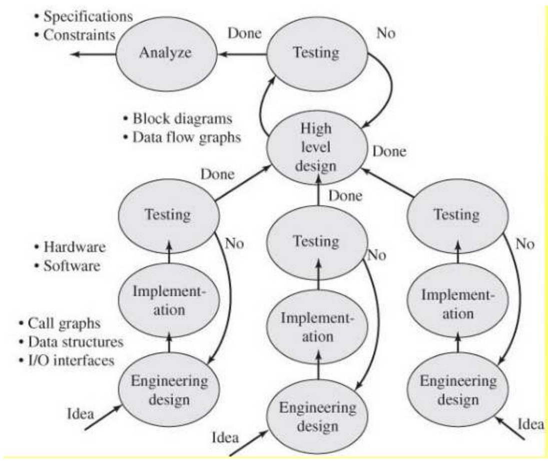
Testing Phase

- During this phase, we evaluate the performance.
- First, debug and validate the basic functions of the system.
- Next, evaluate and optimize various performance parameters such as execution speed, accuracy, and stability.

Maintenance Phase

- During this phase, we:
  - Correct mistakes,
  - Add new features,
  - Optimize execution speed or program size,
  - Port to new computers or operating systems, and
  - Reconfigure the system to solve a similar problem.
- Must be able to deal with changes in requirements or constraints.
- Not actually another phase, but more loops through the entire cycle.

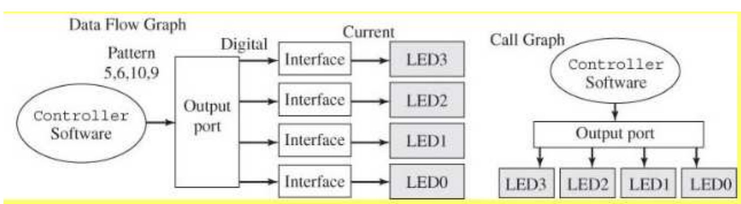
Bottom-Up Design Process



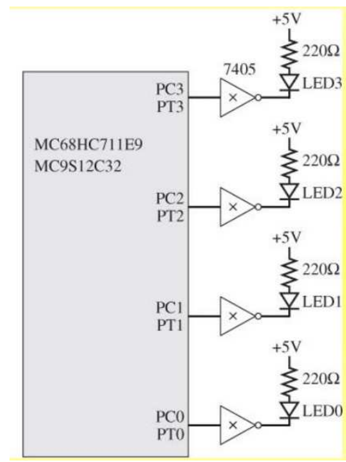
Our First Design Problem: Specifications and Constraints

- Specifications:
  - Design an embedded system that flashes LEDs in a 0101, 0110, 1010, 1001 binary repeating pattern.
  - Use four 2.2V 10mA red LEDs.
  - Use a +5V power supply.
- Constraints:
  - Use a 6812.
  - Minimize cost.
  - Use standard 5% resistors.

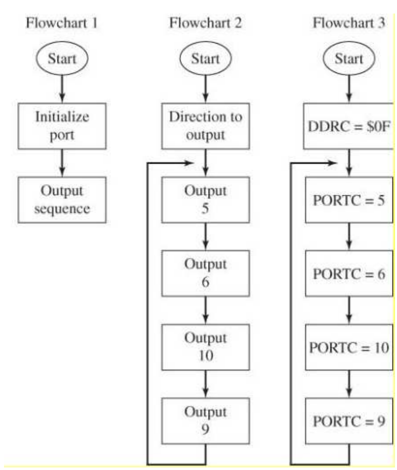
Data Flow Graph and Call Graph for LED Output System



Hardware Circuit for LED Output System



Software Design for LED Output System



I/O Port Definitions

```
; Assembly definitions for an I/O port
PTT  equ $0240
PTIT equ $0241
DDRT equ $0242

// C definitions for an I/O port.
#define PTT  *(unsigned char volatile *)(0x0240)
#define PTIT *(unsigned char volatile *)(0x0241)
#define DDRT *(unsigned char volatile *)(0x0242)
```

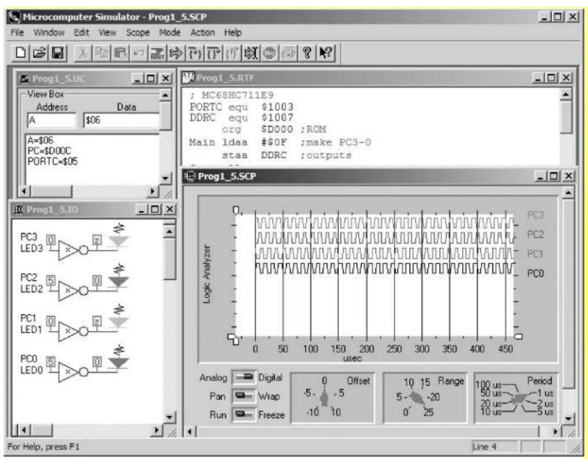
Assembly Software for the LED Output System

```
     org   $4000 ;ROM
Main ldaa  #$0F  ;make PT3-0
     staa  DDRT  ;outputs
Ctrl ldaa  #5
     staa  PTT   ;set 0101
     ldaa  #6
     staa  PTT   ;set 0110
     ldaa  #10
     staa  PTT   ;set 1010
     ldaa  #9
     staa  PTT   ;set 1001
     bra   Ctrl
     org   $FFFE
     fdb   Main  ;Reset vector
```


C Software for the LED Output System

```
void main(void){// make PT3-0
  DDRT = 0x0F;  // outputs
  while(1){
    PTT = 5;    // 0101
    PTT = 6;    // 0110
    PTT = 10;   // 1010
    PTT = 9;    // 1001
  }
}
```


TExaS Simulation of LED Output System



Oscilloscope Waveforms for LED Output System