History of the Microprocessor

- 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, $16 \times 16$ integer divide, $16 \times 16$ fractional divide, and $8 \times 8$ unsigned multiply.
- 6812 also supports $16 \times 16$ unsigned/signed multiply, $32 \times 16$ 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

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

External I/O Ports

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

Operating Frequency

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

```c
void PLL_Init(void){
    SYNR = 0x02;
    REFDV = 0x00; // PLLCLK = 2*OSCCLK*(SYNR+1)/(REFDV+1)
    CLKSEL = 0x00;
    PLLCTL = 0x01;
    while((CRGFLG&0x08) == 0){ // Wait for PLLCLK to stabilize.
    }
    CLKSEL_PLLSEL = 1; // Switch to PLL clock
}
```

Registers
**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.

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

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

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

- 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)**

- 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.

  \[
  \text{fixed-point number} = I \cdot \Delta
  \]

  where \( I \) is a Variable integer and \( \Delta \) 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 \( \Delta \) cannot be changed during program execution, and it likely only appears as a comment in the code.

**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:

  \[
  M = (53 \times N) / 100 \text{ versus } M = 53 \times (N / 100)
  \]

  *Promotion* to higher precision avoids overflow.

  Dividing last avoids drop-out.

**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 +5V, 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 \( \Delta = 0.001V \).
Fixed-Point Arithmetic

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

$z = x + y \quad K = I + J$ (addition)
$z = x - y \quad K = I - J$ (subtraction)
$z = x \cdot y \quad K = (I \cdot J) / \Delta$ (multiplication)
$z = x / y \quad K = (I / \Delta) \cdot J$ (division)

If $\Delta$ is different, then must first convert one of the two numbers to use the $\Delta$ of the other.
If $\Delta$ 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)
- $[\text{addr}]$ specifies an 8-bit read from address
- $\text{addr}$ specifies a 16-bit read from address (big endian)
- $\text{< addr >}$ specifies a 32-bit read from address (big endian)
- $\text{[addr]}$ specifies an 8-bit write to address
- $\text{addr}$ specifies a 16-bit write to address (big endian)
- $\text{< addr >}$ specifies a 32-bit write to address (big endian)

Assembly Language

Assembly language instructions have four fields:

<table>
<thead>
<tr>
<th>Label</th>
<th>Opcode</th>
<th>Operand(s)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>here</td>
<td>ldaa</td>
<td>$0000$</td>
<td>RegA = [$0000]</td>
</tr>
<tr>
<td>staa</td>
<td>$3800$</td>
<td>[3800]</td>
<td>RegA = $3800$</td>
</tr>
<tr>
<td>ldx</td>
<td>$3802$</td>
<td>RegX = ($3802$)</td>
<td>RegX = $3802$</td>
</tr>
<tr>
<td>stx</td>
<td>$3804$</td>
<td>($3804$)</td>
<td>RegX = $3804$</td>
</tr>
</tbody>
</table>

Assembly instructions are translated into machine code:

<table>
<thead>
<tr>
<th>Object code</th>
<th>Instruction</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$96$ $00$</td>
<td>ldaa $0000$</td>
<td>RegA = [$0000]</td>
</tr>
</tbody>
</table>

Inherent Addressing Mode

- Uses no operand field.

<table>
<thead>
<tr>
<th>Obj code</th>
<th>Op</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$3F$</td>
<td>swi</td>
<td>Software interrupt</td>
</tr>
<tr>
<td>$87$</td>
<td>clra</td>
<td>RegA = 0</td>
</tr>
<tr>
<td>$32$</td>
<td>pula</td>
<td>RegA = [RegSP]; RegSP=RegSP+1</td>
</tr>
</tbody>
</table>

Immediate Addressing Mode

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

<table>
<thead>
<tr>
<th>Obj code</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8624$</td>
<td>ldaa</td>
<td>$36$</td>
<td>RegA = 36</td>
</tr>
</tbody>
</table>

 immediate addressing mode (IMM)
 Direct page addressing mode (DIR)
 Extended addressing mode (EXT)
 PC relative addressing mode (REL)

- $\text{< addr >}$ specifies a 32-bit write to address (big endian)
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
---|---|---|---
$60801 | ldaa | $0801 | RegA = [$00801]

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.

\[ rr = (\text{destination address}) - (\text{location of branch}) - (\text{size of the branch}) \]

Assume branch located at $F880.

Obj code | Op | Operand | Comment
---|---|---|---
$208E | 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

- Motor
  - Speed
  - Power
- Actuator
  - Voltage
  - Interface
  - ASCII
- Keypad
  - Interface
  - ASCII
- LCD
- Controller software
  - ADC routines
  - Keypad routines
  - LCD routines
  - Actuator routines
- ADC
- Digital control
- Interface
- Digital sample
- ADC routines
- Keypad routines
- LCD routines
- Actuator routines

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.

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.
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.
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