Lecture 2: 68HC12 Architecture & Lab 1 Introduction
void foo (int x, int *y, int *z) {
    switch (x % 8) {
    case 0:
        do {
            *y++ = *z++;
        } while ((x -= 8) > 0);
    case 7:
        *y++ = *z++;
    ...  
    case 1:
        *y++ = *z++;
    } while ((x -= 8) > 0);
}
Puzzle Winners

Daniel Stewart Parker
Alex Freshman
Brian faires
Lab Times

Lab times

Tuesday 2:00–5:00
Friday 2:00–5:00
Monday 2:00–5:00

We’ll start labs the week after next
Please send a prioritized list to teach-cs5780 by Friday night
Intel 4004

4-bit BCD
92 kHz
Intel 8008 (1972)

8-bit
500 kHz
Intel 8080 (1974)

2 MHz

Considered to be the first truly usable microprocessor
Intel 8086-8088 (1978)
Intel 286 (1982)
Intel® 386™ (1985)
Intel486™ DX CPU (1989)
Intel® Pentium® (1993)
Intel® Pentium® Pro (1995)
Intel® Pentium® II (1997)
Intel® Pentium® III (1999)
During early 1980s, microcontrollers began to be designed. While microprocessors were optimized for speed and memory size, 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.
Two separate 8-bit accumulators (A,B) or one combined 16-bit accumulator (D).
Two 16-bit index registers (X,Y).
8-bit condition code register.
Powerful bit-manipulation instructions.
Supports 16-bit add/subtract, $32 \times 16$ unsigned/signed divide, $16 \times 16$ fractional divide, $16 \times 16$ unsigned/signed multiply, and $32 + (16 \times 16)$ multiply and accumulate.
Stack pointer points to the top element and grows downward.
Registers

- Register A
- Register B
- CC: 8-bit condition code
- D: Two 8-bit accumulators
- X: 16-bit index register
- Y: 16-bit index register
- SP: 16-bit stack pointer
- PC: 16-bit program counter
Condition Code Register

- Carry/borrow or unsigned overflow
- Signed overflow
- Zero
- Negative
- IRQ interrupt mask
- Half carry from bit 3
- XIRQ interrupt mask
- Stop disable
# 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 $FFFFF</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>
MC9S12C32 Block Diagram
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})

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

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

Programmer must 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)

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

Freescale microcontrollers use the *big endian* approach.
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 \( \Delta \) 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.
**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 $\Delta = 0.001V$. 
Overflow and Drop-Out

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

Example:

\[ M = \frac{53 \times N}{100} \quad \text{versus} \quad M = 53 \times \left( \frac{N}{100} \right) \]

Promotion to higher precision avoids overflow. Dividing last avoids drop-out.
Let $x = I \cdot \Delta$, $y = J \cdot \Delta$, $z = K \cdot \Delta$.

- $z = x + y \quad K = I + J$ \hspace{1cm} (addition)
- $z = x - y \quad K = I - J$ \hspace{1cm} (subtraction)
- $z = x \cdot y \quad K = (I \cdot J)/\Delta$ \hspace{1cm} (multiplication)
- $z = x/y \quad K = (I \cdot \Delta)/J$ \hspace{1cm} (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)

= \([addr]\) 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 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></td>
<td>[$3800] = RegA</td>
</tr>
<tr>
<td>ldx</td>
<td>$3802</td>
<td></td>
<td>RegX = {3802}</td>
</tr>
<tr>
<td>stx</td>
<td>$3804</td>
<td></td>
<td>{3804} = RegX</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>
Addressing Modes

An *addressing mode* is a way for an instruction to locate its operand(s)

About 80% of understanding assembly language is understanding the addressing modes

Some 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.

<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>

What is the difference between ldaa #36 and ldaa #$24?
Direct Page Addressing Mode

Uses an 8-bit address to access from addresses $0000$ to $00FF$.

<table>
<thead>
<tr>
<th>Obj code</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$9624$</td>
<td>ldaa</td>
<td>36</td>
<td>RegA = [$0024]</td>
</tr>
</tbody>
</table>

What is the difference between `ldaa #36` and `ldaa 36`?
Extended Addressing Mode

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

<table>
<thead>
<tr>
<th>Obj code</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B60801</td>
<td>ldaa</td>
<td>$0801</td>
<td>RegA = [$0801]</td>
</tr>
</tbody>
</table>
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 $F880$.

<table>
<thead>
<tr>
<th>Obj code</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$20BE$</td>
<td>bra</td>
<td>$F840$</td>
<td>$F840 - F880 - 2 = -42 = BE$</td>
</tr>
<tr>
<td>$2046$</td>
<td>bra</td>
<td>$F8C8$</td>
<td>$F8C8 - F880 - 2 = 46$</td>
</tr>
</tbody>
</table>
Lab1Example.c requirements

SW1 and PB2 light up LED1 (MCU board) and LED1 and LED2 (project board) when pressed.
SW2 and PB1 light up LED2 (MCU board) and LED3 and LED4 (project board) when pressed.
Application Module Student Learning Kit Users Guide (APS12C32SLKUG.pdf) contains the necessary information.

User jumpers table states that jumpers User1-4 must be on to enable the switches and LEDs (pg. 11).

Switches are active low (pg. 11).

SW1 and SW2 provide input on PORTE0 (PE0) and PORTP5 (PP5) respectively (pg. 11).

LEDs are active low (pg. 12).

LED1 and LED2 are driven by PORTA0 (PA0) and PORTB4 (PB4) respectively (pg. 12).
MCU Project Board Student Learning Kit User Guide (PBMCUSLKUG.pdf) contains the necessary information.

Push button switches are active low (pg. 17).
PB1 and PB2 are connected to the MCU via ports 9 and 11 respectively (pg. 20).

Push buttons are enabled by a '0' on port 36 (pg. 21).

LEDs are active high (pg. 18).

LED1-LED4 are connected to the MCU via ports 33, 35, 37, and 39 respectively (pg. 20).

LEDs are enabled by a '0' on port 34 (pg. 21).
## MCU port mappings

<table>
<thead>
<tr>
<th>Board port</th>
<th>MCU Port</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>PP5</td>
<td>PB1</td>
</tr>
<tr>
<td>11</td>
<td>PE0</td>
<td>PB2</td>
</tr>
<tr>
<td>33</td>
<td>PAD4</td>
<td>LED1</td>
</tr>
<tr>
<td>35</td>
<td>PAD5</td>
<td>LED2</td>
</tr>
<tr>
<td>37</td>
<td>PAD6</td>
<td>LED3</td>
</tr>
<tr>
<td>39</td>
<td>PAD7</td>
<td>LED4</td>
</tr>
<tr>
<td>34</td>
<td>PT4</td>
<td>LED_EN</td>
</tr>
<tr>
<td>36</td>
<td>PT5</td>
<td>PB_EN</td>
</tr>
</tbody>
</table>

Mapping found in Application Module Student Learning Kit Users Guide (APS12C32SLKUG.pdf) (pg. 11).
## MCU Port configurations

<table>
<thead>
<tr>
<th>MCU Port</th>
<th>Direction</th>
<th>Config Register</th>
<th>Value</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTE0</td>
<td>Input</td>
<td>DDRE0 (pg. 140)</td>
<td>0</td>
<td>SW1</td>
</tr>
<tr>
<td>PORTP5</td>
<td>Input</td>
<td>DDRP5 (pg. 94)</td>
<td>0</td>
<td>SW2</td>
</tr>
<tr>
<td>PORTA0</td>
<td>Output</td>
<td>DDRA0 (pg. 136)</td>
<td>1</td>
<td>LED1</td>
</tr>
<tr>
<td>PORTB4</td>
<td>Output</td>
<td>DDRB0 (pg. 137)</td>
<td>1</td>
<td>LED2</td>
</tr>
<tr>
<td>PORTP5</td>
<td>Input</td>
<td>DDRP5 (pg. 94)</td>
<td>0</td>
<td>PB1</td>
</tr>
<tr>
<td>PORTE0</td>
<td>Input</td>
<td>DDRE0 (pg. 140)</td>
<td>0</td>
<td>PB2</td>
</tr>
<tr>
<td>PORTAD4</td>
<td>Output</td>
<td>DDRAD4 (pg. 102)</td>
<td>1</td>
<td>LED1</td>
</tr>
<tr>
<td>PORTAD5</td>
<td>Output</td>
<td>DDRAD5 (pg. 102)</td>
<td>1</td>
<td>LED2</td>
</tr>
<tr>
<td>PORTAD6</td>
<td>Output</td>
<td>DDRAD6 (pg. 102)</td>
<td>1</td>
<td>LED3</td>
</tr>
<tr>
<td>PORTAD7</td>
<td>Output</td>
<td>DDRAD7 (pg. 102)</td>
<td>1</td>
<td>LED4</td>
</tr>
<tr>
<td>PORTT4</td>
<td>Output</td>
<td>DDRT4 (pg. 82)</td>
<td>1</td>
<td>LED_EN</td>
</tr>
<tr>
<td>PORTT5</td>
<td>Output</td>
<td>DDRT5 (pg. 82)</td>
<td>1</td>
<td>PB_EN</td>
</tr>
</tbody>
</table>

void main(void) {
    //Set the direction of ports A, B, E, and P.
    DDRA = 0xFF;
    DDRB = 0xFF;
    DDRE = 0x00;
    DDRP = 0x00;
    //Set the direction of ports T and AD
    DDRT = PTT_PTT4_MASK | PTT_PTT5_MASK;
    DDRAD = PTAD_PTAD7_MASK | PTAD_PTAD6_MASK | PTAD_PTAD5_MASK |
            PTAD_PTAD4_MASK;
    //Enable project board push buttons and LEDs
    PTT = ~(PTT_PTT4_MASK | PTT_PTT5_MASK);
}

Macro definitions are found in mc9s12c32.h.
Alternatively...

```c
void main(void) {
    //Set the direction of ports A,B,T,AD,E, and P.
    DDRA = 0xFF;
    DDRB = 0xFF;
    DDRE = 0x00;
    DDRP = 0x00;
    DDRT = 0xFF;
    DDRAD = 0xFF;
    //Enable project board push buttons and LEDs
    PTT = 0x00;
}
```
void main(void) {
    // Set the direction of ports A, B, T, AD, E, and P.
    DDRA = 0xFF;
    DDRB = 0xFF;
    DDRE = 0x00;
    DDRP = 0x00;
    DDRT = 0x30;
    DDRAD = 0xF0;
    // Enable project board push buttons and LEDs
    PTT = 0xCF;
}
void main(void) {
    ...
    for(;;) {
        // Checks the current status of SW1.
        if((PORTE & PORTE_BIT0_MASK) == 0) {
            // Turns on the LEDs
            PORTA = ~PORTA_BIT0_MASK;  \MCU
            PTAD = PTAD|(PTAD_PTAD5_MASK|PTAD_PTAD4_MASK);  \PB
        } else {
            // Turn off the LEDs.
            PORTA = PORTA_BIT0_MASK;  \MCU
            PTAD = PTAD&~(PTAD_PTAD5_MASK|PTAD_PTAD4_MASK);  \PB
        }
    }
}
Basic MCU programming:
  Extremely simple (or no) data structures
  Simple control structures
  Lots of bit-twiddling

These basics are almost exactly the same in C or assembly

Key skills:
  Debugging with very little feedback
  Getting the details right
  Putting together pieces of information that are scattered across hundreds or thousands of pages of documentation