April 2013: Traffic Light Heaven in L.A.

Los Angeles syncs up all 4,500 of its traffic lights

Los Angeles is the first city in the world to synchronize all of its traffic lights, hoping to unclog its massive roadway congestion.

It has taken 30 years and $400 million, but Los Angeles has finally synchronized its traffic lights in an effort to reduce traffic congestion, becoming the first city in the world to do so.

Mayor Antonio R. Villaraigosa said with the 4,500 lights now in sync, commuters will save 2.9 minutes driving five miles in Los Angeles. The New York Times reports Villaraigosa also said that the average speed would rise more than two miles per hour on city streets and that carbon emissions would be greatly reduced as drivers spend less time starting and stopping. According to CBS News, less idling will mean a 1-ton reduction in carbon emissions every year.
## Where Are We Now?

- **Where we’ve been:**
  - Embedded Hardware

- **Where we’re going today:**
  - Instruction set & Assembly Language

- **Where we’re going next:**
  - More assembly language
  - Engineering process
  - Embedded C
  - Coding tricks, bit hacking, extended-precision math

---

### Weekly Schedule

<table>
<thead>
<tr>
<th>Wk #</th>
<th>Week of</th>
<th>Mon (Sec B)</th>
<th>Tue (Sec A)</th>
<th>Wed (Sec B)</th>
<th>Thu (Sec C)</th>
<th>Fri (Sec D)</th>
<th>Lab Report Due (Wed)</th>
<th>Prelab Due (Friday)</th>
<th>Fri. Recitation Discussions Labs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11-Jan 2016</td>
<td>No Lab</td>
<td>No Lab</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>None</td>
<td>1</td>
<td>1.2</td>
</tr>
<tr>
<td>2</td>
<td>18-Jan</td>
<td>MILK Day</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>None</td>
<td>2</td>
<td>2.3</td>
</tr>
<tr>
<td>3</td>
<td>25-Jan</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3.4</td>
</tr>
<tr>
<td>4</td>
<td>1-Feb</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4.5</td>
</tr>
<tr>
<td>5</td>
<td>8-Feb</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>5.6</td>
</tr>
<tr>
<td>6</td>
<td>15-Feb</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6.7</td>
</tr>
<tr>
<td>7</td>
<td>22-Feb</td>
<td>5</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>6</td>
<td>None</td>
<td>None</td>
<td>7.8</td>
</tr>
<tr>
<td>8</td>
<td>29-Feb</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>BREAK</td>
<td>5</td>
<td>7 Due Thursday</td>
<td>No Recitation</td>
</tr>
<tr>
<td>9</td>
<td>7-Mar</td>
<td>SPRING</td>
<td>BREAK</td>
<td>SPRING</td>
<td>BREAK</td>
<td>BREAK</td>
<td>None</td>
<td>None</td>
<td>No Recitation</td>
</tr>
<tr>
<td>10</td>
<td>14-Mar</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>8.9</td>
</tr>
<tr>
<td>11</td>
<td>21-Mar</td>
<td>7</td>
<td>7</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td>9</td>
<td>9.10</td>
</tr>
<tr>
<td>12</td>
<td>28-Mar</td>
<td>8</td>
<td>8</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>10</td>
<td>10</td>
<td>10.11</td>
</tr>
<tr>
<td>13</td>
<td>4-Apr</td>
<td>9</td>
<td>9</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>14</td>
<td>11-Apr</td>
<td>10</td>
<td>10</td>
<td>Open Lab</td>
<td>Carnival</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>No Recitation</td>
</tr>
<tr>
<td>15</td>
<td>18-Apr</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>None</td>
<td>None</td>
<td>Optional In-Lab</td>
</tr>
<tr>
<td>16</td>
<td>25-Apr</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>Open Lab</td>
<td>None</td>
<td>None</td>
<td>Optional In-Lab</td>
</tr>
<tr>
<td>16</td>
<td>2-May Finals</td>
<td>TBD</td>
<td>TBD</td>
<td>TBD</td>
<td>TBD</td>
<td>TBD</td>
<td>H Disc (Thursday)</td>
<td>None</td>
<td>No Recitation</td>
</tr>
</tbody>
</table>

*(See blackboard for Lab 11 res/ labs demo & various information)*
Preview

- **Programmer-visible architecture**
  - Registers
  - Addressing modes

- **Branching**
  - Types of branches
  - How condition codes are set

- **Assembly/Disassembly**
  - Review of how instructions are encoded

- **Timing**
  - How long does an instruction take to execute? (simple version)

---

Where Does Assembly Language Fit?

- **Source code**
  - High level language (C; Java)
  - Variables and equations
  - One-to-many mapping with assembly language

- **Assembly language**
  - Different for each CPU architecture
  - Registers and operations
  - Usually one-to-one mapping to machine language

- **Machine language**
  - Hex/binary bits
  - Hardware interprets to execute program

```
A = B + C

LOAD R1, B
LOAD R2, C
ADD R3, R1, R2
STORE R3, A
```

```
0x EA 74 27 1
0x B7 32 9 A 2
0x ****
```
Assembler To ROM Process

Figure 2.1
Assembly language development process.

| Source code                                                                 | Assembler                                                                 | Loader                                                                 |
|                                                                           |                                                                           |                                                                        |
| ; MC6812CPU2                                                                   |                                                                           |                                                                        |
| PTT equ 00240                                                               |                                                                           |                                                                        |
| DDRT equ 0042                                                                |                                                                           |                                                                        |
| org 04000                                                                    |                                                                           |                                                                        |
| Main ldaa $0CF                                                                |                                                                           |                                                                        |
| stas DDRT                                                                    |                                                                           |                                                                        |
| Controller                                                                  |                                                                           |                                                                        |
| ldaa $4                                                                        |                                                                           |                                                                        |
| staaS PTT ; 0101                                                             |                                                                           |                                                                        |
| ldaa $6                                                                        |                                                                           |                                                                        |
| staaS PTT ; 0110                                                             |                                                                           |                                                                        |
| ldaa $10                                                                     |                                                                           |                                                                        |
| staaS PTT ; 1010                                                             |                                                                           |                                                                        |
| ldaa $9                                                                       |                                                                           |                                                                        |
| staaS PTT ; 1001                                                             |                                                                           |                                                                        |
| brea Controller                                                             |                                                                           |                                                                        |
| org $FFFF                                                                    |                                                                           |                                                                        |
| fdb Main                                                                    |                                                                           |                                                                        |

Typically simple encoding format to make hardware simpler/faster

Classical Example: MIPS R2000
- R7 <= R27 + 0x1234

RISC Instruction Set Overview

- Typically simple encoding format to make hardware simpler/faster
- Classical Example: MIPS R2000
  - R7 <= R27 + 0x1234
### CISC Instruction Set Overview

- **Complex encoding for small programs**
- **Classical Example: VAX; Intel 8088**
  - REP MOVSB (8088 String move)
  - Up to 64K bytes moved; source in SI reg; dest in DI reg; count in CX

```
REP MOVSB
11110010 10100100
ENCODING: 0xF2 0xA4
```

### Accumulator-Based Microcontrollers

- **Usually one or two “main” registers – “accumulators”**
  - Historically called register “A” or “Acc” or registers “A” and “B”
  - This is where the Pentium architecture gets “AX, BX, CX, DX” from

- **Usually one or more “index” registers for addressing modes**
  - Historically called register “X” or registers “X” and “Y”
  - In the Pentium architecture these correspond to SI and DI registers

- **A typical “H = J + K” operation is usually accomplished via:**
  - Load “J” into accumulator
  - Add “K” to “J”, putting result into accumulator
  - Store “H” into memory
  - Reuse the accumulator for the next operation (no large register file)

- **Usually microcontrollers are resource-poor**
  - E.g., No cache memory for most 16-bit micros! 
"Cpu12" Programming Model – (MC9S12C128)

D is really just A:B NOT a separate register!

Flags used for conditional branches

The CPU12 Reference Guide

Summarizes assembly language programming info

Lots of info there …. This lecture is an intro to that material

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Operation</th>
<th>Addr. Mode</th>
<th>Decode Coding (bits)</th>
<th>Access-From</th>
<th>S X H I Z V C</th>
</tr>
</thead>
<tbody>
<tr>
<td>AXA</td>
<td>A.x + B.y</td>
<td>INH</td>
<td>1001 0000</td>
<td>MC12</td>
<td>1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>AXK</td>
<td>B.x (D) → X</td>
<td>EXR</td>
<td>1100 0100</td>
<td>MC12</td>
<td>1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>AXY</td>
<td>B.y (D) → Y</td>
<td>EXE</td>
<td>1010 0110</td>
<td>MC12</td>
<td>1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>AAXA</td>
<td>A.x + B.y</td>
<td>INH</td>
<td>1001 0000</td>
<td>MC12</td>
<td>1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>AAXK</td>
<td>B.x (D) → X</td>
<td>EXR</td>
<td>1100 0100</td>
<td>MC12</td>
<td>1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>AAXY</td>
<td>B.y (D) → Y</td>
<td>EXE</td>
<td>1010 0110</td>
<td>MC12</td>
<td>1 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
ALU Operations – Addition as an Example

◆ “Inherent” address modes:
  • ABA  (B) + (A) => A     Add accumulator B to A
    – Encoding: 18 06
  • ABX  (B) + (X) => X     Add accumulator B to X
    – Encoding: 1A E5

◆ Immediate Operand:
  • ADDD  #value  (D) + jj:kk => D  Add to D
    – Add constant value to D (example: D <= D + 1234)
    – Encoding: C3 jj kk
    – Example: ADDD #$534  Adds hex 534 (0x534) to D reg

◆ “Extended” operand – location in memory at 16-bit address:
  • ADDD  address  (D) + [HH:LL] => D  Add to D
    – Fetch a memory location and add to D
    – Encoding: F3 HH LL
    – Example: ADDD $5910  Adds 16-bit value at $5910 to D

  • NOTE: “[xyz]” notation means “Fetch from address xyz”

Address Modes

Address Modes

IMM — Immediate
IDX — Indexed (no extension bytes) includes:
      5-bit constant offset
      Pre/post increment/decrement by 1 . . . 8
      Accumulator A, B, or D offset
IDX1 — 9-bit signed offset (1 extension byte)
IDX2 — 16-bit signed offset (2 extension bytes)
[D, IDX] — Indexed indirect (accumulator D offset)
[IDX2] — Indexed indirect (16-bit offset)
INH — Inherent (no operands in object code)
REL — 2’s complement relative offset (branches)
DIR — Direct (8-bit memory address with zero high bits)
EXT — Extended (16-bit memory address)
Instruction Description Notation

abc — A or B or CCR
abcdys — A or B or CCR or D or X or Y or SP. Some assemblers also allow T2 or T3.
abdf — A or B or D
abdyxs — A or B or D or X or Y or SP
dyxs — D or X or Y or SP
mask — 5-bit mask, some assemblers require # symbol before value
opr# — 8-bit immediate value
opr16 — 16-bit immediate value
opr8a — 8-bit address used with direct address mode
opr16a — 16-bit address value
opr2y — Indexed addressing postbyte code:
    oprx3—yys — Preincrement X or Y or SP by 1 . . . 8
    oprx3,yys — Preincrement X or Y or SP by 1 . . . 8
    oprx3,yys — Postincrement X or Y or SP by 1 . . . 8
    oprx3,yys+ — Postincrement X or Y or SP by 1 . . . 8
    oprx5,yys — 5-bit constant offset from X or Y or SP or PC
    atxy — Accumulator A or B or D offset from X or Y or SP or PC
opr3 — Any positive integer 1 . . . 6 for pre/post increment/decrement
opr9 — Any value in the range –16 . . . +15
opr2 — Any value in the range –256 . . . +255
opr16 — Any value in the range –32,768 . . . 65,535
page — 8-bit value for PPAGE, some assemblers require # symbol before this value
rel9 — Label of branch destination within –256 to +255 locations
rel11 — Label of branch destination within –512 to +511 locations
rel19 — Any label within 64K memory space
span — Any 8-bit value in the range $30–$39 or $40–$FF
yys — X or Y or SP
xysp — X or Y or SP or PC

Notation for Encoding of Instruction Bytes

Machine Coding

cd — 8-bit direct address $0000$ to $00FF$. (High byte assumed to be $00$).
eg — High-order byte of a 16-bit constant offset for indexed addressing.
eb — Exchange/Transfer post-byte. See Table 3 on page 23.
ff — Low-order eight bits of a 9-bit signed constant offset for indexed addressing,
or low-order byte of a 16-bit constant offset for indexed addressing.
hh — High-order byte of a 16-bit extended address.
i — 8-bit immediate data value.
j — High-order byte of a 16-bit immediate data value.
k — Low-order byte of a 16-bit immediate data value.
lb — Loop primitive (DBNE) post-byte. See Table 4 on page 24.
ll — Low-order byte of a 16-bit extended address.
m — 8-bit immediate mask value for bit manipulation instructions.
Set bits indicate bits to be affected.
p — Program page (bank) number used in CALL instruction.
qq — High-order byte of a 16-bit relative offset for long branches.
t — Trap number $30–$39 or $40–$FF.
rr — Signed relative offset $580 (-128)$ to $7FF (+127)$.
Offset relative to the byte following the relative offset byte, or
low-order byte of a 16-bit relative offset for long branches.
x — Indexed addressing post-byte. See Table 1 on page 21
and Table 2 on page 22.
**ALU Operations – Addition Example Revisited**

- "Inherent" address modes:
  - ABA (B) + (A) => A  
    - Encoding: 18 06  
    - Add accumulator B to A
  - ABX (B) + (X) => X  
    - Encoding: 1A E5  
    - Add accumulator B to X

- Immediate Operand:
  - ADDD #opr16i (D) + jj:kk => D  
    - Encoding: C3 jj kk  
    - Example: ADDD #$534 Adds hex 534 (0x534) to D reg
  - What C code would result in this instruction?
    ```c
    register int16 T;  // assume that X is kept in machine register D
    T = T + 0x534;
    ```

- "Extended" operand – location in memory at 16-bit address:
  - ADDD opr16a (D) + [HH:LL] => D  
    - Encoding: F3 HH LL  
    - Example: ADDD $5910 Adds 16-bit value at $5910 to D
  - What C code would result in this instruction?
    ```c
    static int16 B;  // B is a variable that happens to be at address $5910
    T = T + B;
    ```

**ALU Operations – Addition – 2**

- Immediate Operand:
  - ADDD #opr16i (D) + jj:kk => D  
    - Add constant value to D (example: D <= D + 1234)
    - Encoding: C3 jj kk
    - Example: ADDD #$534 Adds hex 534 (0x534) to D reg

- "Extended" operand – location in memory at 16-bit address:
  - ADDD opr16a (D) + [HH:LL] => D  
    - Fetch a memory location and add to D
    - Encoding: F3 HH LL
    - Example: ADDD $5910 Adds 16-bit value at $5910 to D

- What C code would result in this instruction?
  ```c
  register int16 T;  // assume that X is kept in machine register D
  T = T + 0x534;
  ```

- What C code would result in this instruction?
  ```c
  static int16 B;  // B is a variable that happens to be at address $5910
  T = T + B;
  ```
ALU Operations – Addition – 2

“Direct” operand – location in memory at 8-bit address:
- ADDD opr8a (D) + [00:LL] => D  **Add to D**
  - Fetch a memory location and add to D; address is 0..FF (“page zero” of memory)
  - Encoding: D3 LL
  - Example: ADDD $0038

- Special optimized mode for smaller code size and faster execution
  - Especially for earlier 8-bit processors, but still can be useful
  - Gives you 256 bytes of memory halfway between “memory” and “register” in terms of ease & speed of access
  - Assembler knows to use this mode automatically based on address being $00xx

- Result – programs often optimized to store variables in first 256 bytes of RAM
  - If you have very limited RAM, this is worth doing to save time & space!
  - But it also promotes use of shared RAM for variables, which is bug prone

- What C code would result in this instruction?
  static int16 B;  // B is a variable that happens to be at address $0038
  T = T + B;

ALU Operations – Addition – 3

“Indexed” operand – memory indexed; pre/post increment/decrement
- ADDD oprx,xysp (D) + [EE:FF+XYSP] => D
  - Add oprx to X, Y, SP or PC; use address to fetch from memory; add value into D
  - Encoding: E3 xb  // E3 xb ff  // E3 xb ee ff
    (Signed offset value; encoding varies – 5 bits, 9 bits; 16 bits)
  - Example: ADDD $FFFF, X  add value at (X-16 10) to D
    Encoding: E3 10
    (see Table 1 of CPU12 reference guide for xb byte encoding)

- Special optimized mode for smaller code size and faster execution
  - “xb” can do many tricks, including support for post/pre-increment/decrement to access arrays

- What C code would result in this instruction?
  static int16 B[100];
  register int16 *p = &B[50];  // assume “p” is stored in register X
  T = T + *(p-8);  // adds B[42] to T
Indexed Examples

**Figure 2.2**
Example of the 6811 indexed addressing mode.

| X | $0023 | $0026 |
| A | $056 | $0029 |


**Figure 2.3**
Example of the 6812 indexed addressing mode.

| Y | $0823 | $001E |
| A | $056 | $0820 |


**Figure 2.4**
Another example of the 6812 indexed addressing mode.

| Y | $0823 | $0862 |
| A | $056 | $0864 |


**Figure 2.5**
A third example of the 6812 indexed addressing mode.

| Y | $0823 | $0A22 |
| A | $056 | $0A24 |

---

ALU Operations – Addition – 4

◆ “Indexed Indirect” operand – use memory value as address, with offset

- **ADD [oprx16,xysp]**
  
  $D = (D) + [EE:FF+XYSP]$

  - Add oprx to X, Y, SP or PC; use address to fetch from memory; use the value fetched from memory to fetch from a different memory location; add value into D

  - Encoding: \( E3 \) \( xa \) \( ee \) \( ff \)

  - Example: ADDD \([S8, X]\) add value at \((X+8)\) to D

  - Encoding: \( E3 \) \( E3 \) \( 00 \) \( 08 \) 16-bit constant offset

  (see Table 1 of CPU12 reference guide for \( xa \) byte encoding)

  - What C code would result in this instruction?

    ```c
    static int16 vart;
    register int16 *p;
    static int16 *B[100]; // B is a variable that happens to be at address $38
    
    p = &B[0]; // assume "p" is stored in register X
    T = T + "*(p+4)"; // adds vart to T
    ```

24
Indexed Indirect Example

LDAA #$56
LDY #$2345
STAA [-4,Y] ; Fetch 16-bit address from $2341, store A at $1234

Figure 2.6
Example of the 6812 indexed-indirect addressing mode.

Had Enough Yet?

- Really, all these modes get used in real programs
  - You’ve already seen very similar stuff in 18-240, but that’s more RISC-like
  - We expect you to be able to tell us what a short, simple program we’ve written does if it uses any of the modes described during lecture
  - There are even trickier modes – seldom used but nice to have

- See Valvano Section 2.2 for more discussion
### Other Math & Load/Store Instructions

**Math**
- ADD – integer addition (2’s complement)
- SBD – integer subtraction (2’s complement)
- CMP – compare (do a subtraction to set flags but don’t store result)

**Logic**
- AND – logical bit-wise and
- ORA – logical bit-wise or
- EOR – bit-wise exclusive or (xor)
- ASL, ASR – arithmetic shift left and right (shift right sign-extends)
- LSR – logical shift right

**Data movement**
- LDA, LDX, … – load from memory to a register
- STA, STX, … – store from register to memory
- MOV – memory to memory movement

**Bit operations and other instructions**
- Later…

### Control Flow Instructions

**Used to go somewhere other than the next sequential instruction**
- Unconditional branch – always changes flow (“goto instruction x”)
- Conditional branch – change flow sometimes, depending on some condition

**Addressing modes**
- REL: Relative to PC – “go forward or backward N bytes”
  - Uses an 8-bit offset rr for the branch target
  - Most branches are short, so only need a few bits for the offset
  - Works the same even if segment of code is moved in memory

- EXT: Extended hh:ll – “go to 16-bit address hh:ll”
  - Takes more bits to specify
  - No limit on how far away the branch can be
Relative Addressing

- **Relative address computed as:**
  - Address of next in-line instruction after the branch instruction
    - Because the PC already points to the next in-line instruction at execution time
  - Plus relative byte $rr$ treated as a signed value
    - $rr$ of 0..$7F$ is a forward relative branch
    - $rr$ of $80$..$FF$ is a backward relative branch

- **Example: BCC cy_clr**
  - Next instruction is at $0009$; $rr = 03$
  - $0009 + 03 = 000C$ (cy_clr)

- **Example: BRA asm_loop**
  - Next instruction is at $000F$; $rr = 07$
  - $000F + 07 = 000F + 0FFF7 = 000F - 0009 = 0006$ (asm_loop)

Unconditional Branch

- **JMP instruction – Jump**
  - JMP $1256$ -- jump to address $1256$
  - JMP Target_Name

  - JMP also supports indexed addressing modes – why are they useful?

- **BRA $12$ -- jump to $12$ past current instruction
  - Relative addressing (“$rr$”) to save a byte and make code relocatable

- **JSR instruction – Jump to Subroutine**
  - JSR $7614$ -- jump to address $7614$, saving return address
  - JSR Subr_Name

  - Supports DIRect (8 bit offset to page 0) and EXTended, as well as indexed addressing
  - More about how this instruction works in the next lecture
Conditional Branch

- **Branch on some condition**
  - Always with RELative (rr 8-bit offset) addressing
    - Look at detailed instruction set description for specifics of exactly what address the offset is added to
  - Condition determines instruction name
    - BCC $08 – branch 8 bytes ahead if carry bit clear
    - BCS Loop – branch to label “Loop” if carry bit set
    - BEQ / BNE – branch based on Z bit (“Equal” after compare instruction)
    - BMI / BPL – branch based on N bit (sign bit)

- **Other complex conditions that can be used after a CMP instruction**
  - BGT – branch if greater than
  - BLE – branch if less than or equal
  - …

Condition Codes

- **Status bits inside CPU that indicate results of operations**
  - C = carry-out bit
  - Z = whether last result was zero
  - N = whether last result was “negative” (highest bit set)
  - V = whether last result resulted in an arithmetic overflow

- **Set by some (but not all instructions)**
  - CMP – subtracts but doesn’t store result; sets CC bits for later “BGE, BGT” etc
  - ADD and most arithmetic operations – sets CC bits
  - MOV instructions – generally do NOT set CC bits on this CPU
    - But, on a few other CPUs they do – so be careful of this!
C & V flags

- **Carry**: did the previous operation result in a carry out bit?
  - $\text{FFFF} + 1 = \text{0000} + \text{Carry out}$
  - $\text{7FFF} + \text{8000} = \text{FFFF} + \text{No Carry out}$
  - Carry-in bit, if set, adds 1 to sum for ADC
    - we’ll do multi-precision arithmetic later
  - Carry bit is set if there is an *unsigned* add or subtract overflow
    - Result is on other side of $\text{0000}/\text{FFFF}$ boundary

- **Overflow (V)**: did the previous operation result in a signed overflow?
  - $\text{FFFF} + 1 = \text{0000}$ no signed overflow (-1 + 1 = 0)
  - $\text{7FFF} + 1 = \text{8000}$ has signed overflow (32767 + 1 \(\rightarrow\) -32768)
  - This is overflow in the normal signed arithmetic sense that you are used to
    - Result is on other side of $\text{8000}/\text{7FFF}$ boundary

- **Note that the idea of “overflow” depends on signed vs. unsigned**
  - Hardware itself is sign agnostic – software has to keep track of data types
  - Carry flag indicates unsigned overflow
  - V flag indicates signed overflow

---

Look For Annotations Showing CC Bits Set

**Instruction Set Summary (Sheet 5 of 14)**

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Operation Description</th>
<th>Add Mode</th>
<th>Machine Coding (Hex)</th>
<th>Access Details</th>
<th>$S$</th>
<th>$X$</th>
<th>$R$</th>
<th>$E$</th>
<th>$V$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNL, S1=0, xM</td>
<td>(optional) 1 (\Rightarrow) 0 M Location and Branch</td>
<td>(optional)</td>
<td>(\text{F} | \text{011} | \text{XX} | \text{XX} )</td>
<td>(000 | \text{YY} )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(1 )</td>
</tr>
<tr>
<td>DEC, $xM$</td>
<td>Decrement Memory Location</td>
<td>(optional)</td>
<td>(\text{E} | \text{XX} | \text{XX} )</td>
<td>(000 | \text{YY} )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(1 )</td>
</tr>
<tr>
<td>DEC, $a,xM$</td>
<td>Decrement Accumulator</td>
<td>(optional)</td>
<td>(\text{E} | \text{XX} | \text{XX} )</td>
<td>(000 | \text{YY} )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(1 )</td>
</tr>
<tr>
<td>DEC, $xM,xM$</td>
<td>Decrement Memory Location</td>
<td>(optional)</td>
<td>(\text{E} | \text{XX} | \text{XX} )</td>
<td>(000 | \text{YY} )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(1 )</td>
</tr>
<tr>
<td>DECA</td>
<td>(optional) A (\Rightarrow) A</td>
<td>(optional)</td>
<td>(\text{E} | \text{XX} | \text{XX} )</td>
<td>(000 | \text{YY} )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(1 )</td>
</tr>
<tr>
<td>DECB</td>
<td>(optional) B (\Rightarrow) B</td>
<td>(optional)</td>
<td>(\text{E} | \text{XX} | \text{XX} )</td>
<td>(000 | \text{YY} )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(1 )</td>
</tr>
<tr>
<td>DES</td>
<td>GP1 (\Rightarrow) GP1</td>
<td>(optional)</td>
<td>(\text{E} | \text{XX} | \text{XX} )</td>
<td>(000 | \text{YY} )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(1 )</td>
</tr>
<tr>
<td>DEK</td>
<td>GP (\Rightarrow) GP</td>
<td>(optional)</td>
<td>(\text{E} | \text{XX} | \text{XX} )</td>
<td>(000 | \text{YY} )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(1 )</td>
</tr>
<tr>
<td>DEV</td>
<td>GP (\Rightarrow) GP</td>
<td>(optional)</td>
<td>(\text{E} | \text{XX} | \text{XX} )</td>
<td>(000 | \text{YY} )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(000 )</td>
<td>(1 )</td>
</tr>
</tbody>
</table>
Assembler to Hex

- Sometimes (less often these days, but sometimes) you have to write your own assembler!

- In this course, we want you to do just a little by hand to get a feel
  - LDAB #254

<table>
<thead>
<tr>
<th>Addressing mode</th>
<th>Opcode</th>
<th>Operand</th>
<th>Full encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>LDAB</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Addressing mode is:___________________
- Opcode is:___________________
- Operand is:___________________
- Full encoding is:_____   _____

[Motorola01]

Hex to Assembler (Dis-Assembly)

- If all you have is an image of a program in memory, what does it do?
  - Important for debugging
  - Important for reverse engineering (competitive analysis; legacy components)

- Start with Hex, and figure out what instruction is
  - AA E2 23 CC

<table>
<thead>
<tr>
<th>ORAA</th>
<th>ORAA</th>
<th>E2 23 CC with Memory</th>
</tr>
</thead>
</table>

- ORAA – one of the indexed versions
- Need to look up XB value =>

[Motorola01]
Easier Way To Find Op-Code Information

Performance – How Many Clock Cycles?

◆ This is not so easy to figure out
  • See pages 73-75 of the CPU 12 reference manual

◆ In general, factors affecting speed are:
  • Does the chip have an 8-bit or 16-bit memory bus? (Ours has a 16-bit bus)
    – 8-bit bus needs one memory cycle per byte
    – 16-bit bus needs one memory cycle per 2 bytes, but odd addresses only get 1 byte
  • How many bytes in the encoded instruction itself?
    – AA E2 23 CC takes 4 bytes of fetching
      » 2 bus cycles if word aligned
      » 3 bus cycles if unaligned (but get next instruction byte “for free” on 3rd cycle)
  • How many bytes of data
    – Need to read data and, potentially write it
  • Is there an instruction prefetch queue that can hide some fetch delay?
  • Is it a complicated computation that consumes clock cycles (e.g., division)?

◆ Usual lower bound estimate
  • Count up clock cycles for memory touches and probably it takes that or longer
Simple Timing Example

- **ADCA $1246**
  - EXT format – access detail is “rPO” for HCS12
    - r – 8-bit data read
    - P – 16-bit program word access to fetch next instruction
    - O – either prefetch cycle or free cycle (memory bus idle) based on alignment
  - Total is 3 clock cycles
    - (lower case letters are 8-bits; upper case letters are 16-bit accesses)
    - Simple rule – count letters for best case # of clock cycles

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
<th>Access Detail</th>
<th>HCS12</th>
<th>M88HC12</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADCA #op06</td>
<td>IMM</td>
<td>39 11</td>
<td>P</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>ADCA op16a</td>
<td>DIR</td>
<td>39 11 11</td>
<td>r</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>ADCA op16a</td>
<td>EXT</td>
<td>39 11 11</td>
<td>r</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>ADCA [op16,xyp]</td>
<td>IDX</td>
<td>39 11 11 11</td>
<td>r</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>ADCA [op16,xyp]</td>
<td>[IDX]</td>
<td>39 11 11 11</td>
<td>r</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

Another Timing Example

- Recall that “D” is a 16-bit register comprised of A:B
- **ADDD $1247, X**
  - IDX2 format – access detail is “fRPP” for HCS12
    - f – free cycle (to add address to computation performed, memory bus idle)
    - R – 16-bit data read
    - P – 16-bit program word access to fetch next instruction
    - P – 16-bit program word access to fetch next instruction
  - Total is 4 or 5 clock cycles
    - 4 for minimum; plus 1 if value of X+$1247 is odd (straddles word boundaries)

<table>
<thead>
<tr>
<th>Source Form</th>
<th>Address Mode</th>
<th>Object Code</th>
<th>Access Detail</th>
<th>HCS12</th>
<th>M88HC12</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDD #op06</td>
<td>IMM</td>
<td>39 31 11</td>
<td>R</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>ADDD op16a</td>
<td>DIR</td>
<td>39 31 11 11</td>
<td>R</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>ADDD op16a</td>
<td>EXT</td>
<td>39 31 11 11</td>
<td>R</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>ADDD [op16,xyp]</td>
<td>IDX</td>
<td>39 31 11 11</td>
<td>R</td>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>ADDD [op16,xyp]</td>
<td>[IDX]</td>
<td>39 31 11 11</td>
<td>R</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>
Preview of Labels for Prelab 2

- **Labels are a convenient way to refer to a particular address**
  - Can be used for program addresses as well as data addresses
  - You know it is a label because it starts in column 1 (":" is optional)

- **Assume you are currently assembling to address $4712**
  - (how you do that comes in the next lecture)

```
Mylabela:
  ABA ; this is at address $4712

Mylabelb:

Mylabelc
  PSHA ; this is at address $4713
```

- The following all do EXACTLY the same thing:
  - JMP $4713
  - JMP Mylabelb
  - JMP Mylabelc

Preview of Assembler Psuedo-Ops

- **The following are assembler directives, not HC12 instructions**
  - Labels – refer to an address by name instead of hex number
  - ORG: define the address where data/code starts
  - DS: Define Storage (allocate space in RAM)
  - DC: Define Constant (allocate space in ROM/flash)
  - EQU: Equate (like an equal sign for assembler variables)

- **This is for orientation when looking at code**
  - Specifics in the next lecture
**Lecture 3 Lab Skills**

- Write an assembly language program and run it
- Manually convert assembly language to hex
- Manually convert hex program to assembly language

**Lecture 3 Review**

- **CPU12 programmer model**
  - Registers
  - Condition codes

- **Memory Addressing modes**
  - Given an instruction using one of the modes described and some memory contents, what happens?

- **Assembly**
  - Given some assembly language, what is the hex object code?
  - Given some hex object code, what is the assembly language

- **Simple timing**
  - Given an encoded instruction, what is the minimum number of clocks to execute?
    - Be able to count number of letters in the timing column
    - We do not expect you to figure out all the rules for straddling word boundaries etc.
  - Branch cycle counting covered in next lecture