Administrivia

- 2 versions of CodeWarrior are on the lab machines.
- You should use the 4.5 version (CW for HC12 v 4.5).
- Lab 1 was updated yesterday morning.
- Both team members need to be present for check-off.
- Lab reports don’t need to be in "report" format.

Assembly Language Syntax

Label    Operation    Operand    Comment
PORTA    equ        $0000    ; Assembly time constant
Inp      ldaa      PORTA    ; Read data from PORTA
          ldaa      PORTA    ; Read data from PORTA
          bra Controller

- If first character of label is “*” or “;”, then line is a comment.
- If first character is white space, then there is no label.
- Labels composed of characters, digits, “.”, “$”, or “_”, and must start with a character, “.”, or “_”, and are case-sensitive.
- Labels should be defined only once except those defined by set.
- With exception of equ and set, a label is assigned value of program counter for the next instruction or assembler directive.
- A label may have an optional “.” which is ignored or be on a line by itself.
Assembly Language Syntax (cont)

- Operations must be proceeded by at least one white space character, and they are case-insensitive (nop, NOP, NoP).
- Operations can be an opcode or assembler directive (pseudo-op).
- Operand must be proceeded by white space.
- Operands must not contain any white space unless the following comment begins with a semicolon.
- Operands are composed of symbols or expressions.

Operand Types

<table>
<thead>
<tr>
<th>Operand</th>
<th>Format</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>no operand</td>
<td>INH</td>
<td>clra</td>
</tr>
<tr>
<td>#&lt;expression&gt;</td>
<td>IMM</td>
<td>1daa #4</td>
</tr>
<tr>
<td>&lt;expression&gt;</td>
<td>DIR,EXT,REL</td>
<td>1daa 4</td>
</tr>
<tr>
<td>&lt;expression&gt;,idx</td>
<td>indexed (IND)</td>
<td>1daa 4,x</td>
</tr>
<tr>
<td>&lt;expr&gt;,#&lt;expr&gt;,&lt; expr&gt;</td>
<td>bit set or clear</td>
<td>bset 4,#$01</td>
</tr>
<tr>
<td>&lt;expr&gt;,idx,#&lt;expr&gt;,&lt; expr&gt;</td>
<td>bit test &amp; branch</td>
<td>brset 4,#$01,foo</td>
</tr>
<tr>
<td>&lt;expression&gt;,idx+</td>
<td>IND, post incr</td>
<td>1daa 4,x+</td>
</tr>
<tr>
<td>&lt;expression&gt;,idx-</td>
<td>IND, post decr</td>
<td>1daa 4,x-</td>
</tr>
<tr>
<td>&lt;expression&gt;,+idx</td>
<td>IND, pre incr</td>
<td>1daa 4,+x</td>
</tr>
<tr>
<td>&lt;expression&gt;,-idx</td>
<td>IND, pre decr</td>
<td>1daa 4,-x</td>
</tr>
<tr>
<td>acc,idx</td>
<td>accum offset IND</td>
<td>1daa A,x</td>
</tr>
<tr>
<td>[&lt;expression&gt;,idx]</td>
<td>IND indirect</td>
<td>1daa [4,x]</td>
</tr>
<tr>
<td>[D,idx]</td>
<td>RegD IND indirect</td>
<td>1daa [D,x]</td>
</tr>
</tbody>
</table>

Indexed Addressing Mode

- Uses a fixed signed offset with a 16-bit register: X, Y, SP, or PC.
- Offset can be 5-bits, 9-bits, or 16-bits.
- Example (5-bit):
  
  Obj code Op Operand Comment  
  $6A5C staa -4,Y ;[Y-4] = RegA

Building the Object Code

- staa -4,Y → $6A5C
  - First byte is $6A - Op code (pg. 254)
  - Second byte is formatted as %rr0nnnn (pg. 33).
  - rr is %01 for register Y.
  - nnnn is %11100 for -4.
  - %0101 1100 → $5C.
- Information is found in the CPU12 Reference Manual (CPU12RM.pdf).
## Auto Pre/Post Decrement/Increment Indexed

- Can be used with the X, Y, and SP registers, but not PC.
- The register used is incremented/decremented by the offset value (1 to 8) either before (pre) or after (post) the memory access.
- In these examples assume that RegY=2345:

<table>
<thead>
<tr>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>staa</td>
<td>1,Y+</td>
<td>;Store RegA at 2345, then RegY=2346</td>
</tr>
<tr>
<td>staa</td>
<td>4,Y-</td>
<td>;Store RegA at 2345, then RegY=2341</td>
</tr>
<tr>
<td>staa</td>
<td>4,+Y</td>
<td>;RegY=2349, then store RegA at 2349</td>
</tr>
<tr>
<td>staa</td>
<td>1,-Y</td>
<td>;RegY=2344, then store RegA at 2344</td>
</tr>
</tbody>
</table>

## Building the Object Code

- `staa 1,X+ -> $6A30`
  - First byte is $6A - Op code (pg. 254)
  - Second byte is formatted as %rr1pnnn (pg. 33).
  - rr is %00 for register X.
  - nnn is %0000 for 1.
  - p is %1 for post.
  - %0011 0000 → $30.
- Information is found in the CPU12 Reference Manual (CPU12RM.pdf).

## Accumulator Offset Indexed

- Uses two registers, offset is in A, B, or D, while index is in X, Y, SP, or PC.
- Examples:

<table>
<thead>
<tr>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldab</td>
<td>#4</td>
<td></td>
</tr>
<tr>
<td>ldy</td>
<td>#2345</td>
<td></td>
</tr>
<tr>
<td>staa</td>
<td>B,Y</td>
<td>;Store value in RegA at 2349</td>
</tr>
</tbody>
</table>

## Indexed Indirect

- Adds 16-bit offset to 16-bit register (X,Y,SP, or PC) to compute address in which to fetch another address.
- This second address is used by the load or store.
- Examples:

<table>
<thead>
<tr>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldy</td>
<td>#$2345</td>
<td></td>
</tr>
<tr>
<td>staa</td>
<td>[-4,Y]</td>
<td>;Fetch 16-bit address from $2341,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;store $56 at $1234</td>
</tr>
</tbody>
</table>

![Indexed Indirect Diagram](image-url)
### Load and Store Instructions

- Used to move data to (from) registers from (to) memory.
- Load instructions are: 1daa, 1dab, 1dd, 1ds, 1dx, and 1dy.
- Load addressing modes are: IMM, DIR, EXT, IND.
- Store instructions are: staa, stab, std, sds, sdx, and ady.
- Store addressing modes are: DIR, EXT, IND.
- CC bits $N$ and $Z$ are updated based on data loaded or stored.

**Examples:**

<table>
<thead>
<tr>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1daa</td>
<td>#$FF</td>
<td>IMM</td>
</tr>
<tr>
<td>staa</td>
<td>$25</td>
<td>DIR</td>
</tr>
<tr>
<td>1dab</td>
<td>$0025</td>
<td>EXT</td>
</tr>
<tr>
<td>std</td>
<td>$05,X</td>
<td>IND</td>
</tr>
<tr>
<td>1dd</td>
<td>$0C25</td>
<td>EXT</td>
</tr>
</tbody>
</table>

### Memory to Memory Move Instructions

- Used to move constant into memory or the value of one memory location into another.
- CC bits are not affected by these instructions.
- Move an 8-bit constant into memory:
  
  ```
  movb  #w, addr [addr]=w
  ```
- Move an 8-bit value memory to memory:
  
  ```
  movb addr1,addr2 [addr2]=[addr1]
  ```
- Move a 16-bit constant into memory:

  ```
  movw  #W, addr {addr}=W
  ```
- Move a 16-bit value memory to memory:

  ```
  movw addr1,addr2 {addr2}={addr1}
  ```

### Load Effectve Address

- Used with IND addressing modes.
- Calculate the effective address and store it in the specified register: X, Y, or SP.
- CC bits are not affected.

**Example:**

- `leas -4, SP ; SP -= 4 → $1B9C`
- `$1B` is `leas` op code (first byte).
- Second byte is `rr0nnnnn`.
- `%10` is the `rr` code for SP.
- `%1 1100` is `-4`.

### Accumulator D Offset Indexed Indirect

- Offset is in D and index is in another 16-bit register.
- Computed addressed is used to fetch another address from memory.
- Load or store uses the second address.

**Examples:**

<table>
<thead>
<tr>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldd</td>
<td>#4</td>
<td></td>
</tr>
<tr>
<td>ldy</td>
<td>#$2341</td>
<td>;Store value in RegX at $1234</td>
</tr>
</tbody>
</table>

**Table:**

<table>
<thead>
<tr>
<th>D</th>
<th>0004</th>
<th>1233</th>
<th>1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>5678</td>
<td>1235</td>
<td>56</td>
</tr>
<tr>
<td>Y</td>
<td>2341</td>
<td>2342</td>
<td>2345</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th>0004</th>
<th>1233</th>
<th>1234</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>5678</td>
<td>1235</td>
<td>56</td>
</tr>
<tr>
<td>Y</td>
<td>2341</td>
<td>2344</td>
<td>2345</td>
</tr>
</tbody>
</table>

`D + Y = 2345`
### Clear/Set Instructions

- Used to initialize memory (clr), accumulators (clra, clrb), or
  bits in the CC (clc, cli, clv).
- clr addressing modes are: EXT, IND.
- clra, clrb, clc, cli, clv are INH.
- Examples:
  
  ```
  Op   Operand  Comment
  clra  INH
  clr  $0025  EXT
  ```

- The carry (C), interrupt mask (I), and overflow (V) bits in the
  CC can also be set (sec, sei, sev).

### Add and Subtract Instructions

- Registers: aba, abx, aby, sba (all INH).
- With carry to memory: adca, adcb, sbca, sbcb.
- w/o carry to memory: adda, addb, addd, suba, subb, subd.
- Addressing modes are: IMM, DIR, EXT, IND.
- Examples: 16-bit addition using only A

  ```
  Op   Operand  Comment
  ldaa $25  load least sig byte
  adda $35  add data at $35 to A
  sta $45  store least sig byte
  lda $24  load most sig byte
  adca $34  add data at $34 to A
  staa $44  store most sig byte
  ```

### Other Data Movement Instructions

- Transfer instructions used to copy data between registers.
  - tab, tap, tba, tpa, tax, tay, txs, tys (all INH).
- Exchange instructions used to exchange data between
  accumulator D and index registers X and Y.
  - xgd, xgy (all INH).

### Compare Instructions

- Perform a subtraction to update the CC, but do not alter data
  register values.
- Typically used just before a branch instruction.
- Compare registers: cba (INH).
- Compare to memory: cmpa, cmpb, cpd, cpx, cpy.
- Addressing modes: IMM, DIR, EXT, IND
- Example: comparing with a known set point

  ```
  Obj code  Op   Operand  Comment
  $8650  ldaa  #$50  load set point into A
  $B11031 cmpa  $1031  compare A to memory
  ```

- If Z flag is 1 then the contents of $1031 equals $50.
### Miscellaneous Arithmetic Instructions

- `dec`, `deca`, `decb`, `des`, `dex`, `dey` - decrement
- `inc`, `inca`, `incb`, `ins`, `inx`, `iny` - increment
- `neg`, `nega`, `negb` - two's complement.
- `tst`, `tsta`, `tstb` - subtracts 0 from memory or register and sets `Z` and `N` flags in the CC.

### Integer Divide Instruction

- Use `D` register for the dividend and `X` register for the divisor.
- Resultant placed in `X` register and remainder in `D` register.
- `idiv` performs integer division.

$$\begin{align*}
\text{Register } D & \div \text{ Register } X = \text{ Register } X \\
\text{Remainder} & = \text{ Register } D
\end{align*}$$

- Example:
  - Obj code | Op | Operand | Comment
  - $CCFFFF$ | ldd | #$FFFF$ (65535) | After `idiv` executes
  - $CE2710$ | ldx | #$2710$ (10000) | X contains $0006$
  - $02$ | idiv | | D contains $159F$ (5535)

### Multiply Instructions

- Multiplies two unsigned 8-bit values in `A` and `B` to produce a 16-bit unsigned product stored in `D` (i.e., `A \times B \rightarrow D`).

$$\begin{align*}
\text{Register } A & \times \text{ Register } B = \text{ Register } A | \text{ Register } B \\
\text{8 bits} & \times \text{8 bits} = \text{16 bits}
\end{align*}$$

- Example:
  - Op | Operand | Comment
  - ldaa | #$FF$ (255) | IMM
  - ldab | #$14$ (20) | IMM
  - mul | | INH
  - At the end, accumulator `D` contains `13EC` (5100).
  - `$FF \times FF = $FE01$

### Fractional Divide Instruction

- `fdiv` performs fractional division resulting in binary weighted fraction between 0 and 0.999998 (i.e., $(65536 \times D)/X \rightarrow X$).

$$\begin{align*}
\text{Register } D & \div \text{ Register } X = \text{ Register } X \\
\text{Remainder} & = \text{ Register } D
\end{align*}$$

- Numerator must be less than denominator or overflow occurs.
- Next 16-bits of the fraction can be obtained by reloading the denominator and doing `fdiv` again.
### fdiv Example

**Op** Operand  
**ldd** #1  
**ldx** #3  
**fdiv**

- Result: $X: 5555$ and $D: 0001$.
- Assuming decimal point is to the left of the MSB $5555 = 0.333328247...$
- Another fdiv refines the value to: $0.3333333325572$

### Extended Precision Arithmetic Instruction

- **emul** and **emuls** perform $16 \times 16$ unsigned and signed multiplication.

```
\[
\begin{array}{ccc}
\text{Reg Y} & \times & \text{Reg D} = \text{Reg Y} & \text{Reg D} \\
16 \text{ bits} & 16 \text{ bits} & 32 \text{ bits}
\end{array}
\]
```

- **ediv** and **edivs** perform $32 \times 16$ unsigned and signed division.

```
\[
\begin{array}{ccc}
\text{Reg Y} & \text{Reg D} & \div \text{Reg X} = \text{Reg Y} \\
32 \text{ bits} & 16 \text{ bits} & 16 \text{ bits}
\end{array}
\]\[\text{Remainder: Reg D}\]

### Example: $M = (53 \times N + 50)/100$

```
\begin{align*}
\text{ldd } N & \quad ; \text{Reg}A=N \ (\text{between } 0 \text{ and } 65535) \\
\text{ldy } #53 & \quad ; \text{Reg}Y:D=53 \times N \ (\text{between } 0 \text{ and } 347355) \\
\text{emul} & \quad ; \text{Reg}Y:D=53 \times N + 50 \ (\text{between } 0 \text{ and } 347355) \\
\text{add} \ #50 & \quad ; \text{Reg}D=53 \times N + 50 \ (\text{between } 0 \text{ and } 347355) \\
\text{bcc } & \quad \text{skip iny} \\
\text{skip } \text{ldx } #100 & \quad ; \text{Reg}Y=(53 \times N + 50)/100 \ (\text{between } 0 \text{ and } 34734) \\
\text{ediv} & \quad ; \text{Reg}Y=_{\text{M}} \quad (\text{between } 0 \text{ and } 34734) \\
\text{sty } & \\
\end{align*}
```

### Multiply and Accumulate

- **emacs** performs a $16 \times 16$ signed multiply followed by a 32-bit signed addition.
- Uses indexed addressing to access two 16-bit inputs and extended addressing to access the 32-bit sum.

$$< U > = < U > + \{X\} \times \{Y\}$$
Shift Instructions

- Logical shift right (lsr, lsra, lsr b, lsr d) shifts 0’s into MSB.
  \[ \text{LSR} \quad 0 \]
- Arithmetic shift right (asr, asra, asrb) retains MSB value.
  \[ \text{ASR} \]
- Logical shift left (lsl, ls1a, lslb, lsld) and arithmetic shift left (as1, as1a, aslb, asld) are equivalent (same op code).
  \[ \text{LSL/ASL} \quad C \quad 0 \]

Rotate Instructions

- Rotate right (ror, rora, rorb) put carry bit into the MSB.
  \[ \text{ROR} \quad C \]
- Rotate left (rol, rola, rolb) put carry bit into the LSB.
  \[ \text{ROL} \quad C \]

Logical Operation Instructions

- AND - anda, andb (IMM, DIR, EXT, IND).
- Inclusive OR - oraa, orab (IMM, DIR, EXT, IND).
- Exclusive OR - eora, eorb (IMM, DIR, EXT, IND).
- 1’s complement - com, coma, comb.
- Example: masking unwanted bits
  
<table>
<thead>
<tr>
<th>Obj code</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8634</td>
<td>1daa</td>
<td>#$34</td>
<td>%00110100</td>
</tr>
<tr>
<td>$840F</td>
<td>anda</td>
<td>#$0F</td>
<td>%00001111</td>
</tr>
</tbody>
</table>
  
  Result in A is %00000100

Data Test and Bit Manipulation

- bitsa and bitb instructions perform an AND operation and update N and Z flags of the CC w/o altering the operand.
- bclr and bset instructions are used to clear or set bit(s) in a given memory location.
  
  ```
  bclr addr, mm
  bset addr, mm
  ```
  
  where addr is a memory location specified using DIR, EXT, or IND addressing mode and mm is a mask byte.
Stack Instructions

- Stack pointer (RegSP) defines the top of the stack.
- Should be loaded with a RAM memory address early in any assembly language program.
- Push and pull instructions put data onto and take data off the stack.
  - psha, pshb, pshx, pshy, pula, pulb, pulx, puly (all INH).

Example: Saving State to Stack

<table>
<thead>
<tr>
<th>Op</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>pshy</td>
<td>INH</td>
</tr>
<tr>
<td>pshx</td>
<td>INH</td>
</tr>
<tr>
<td>pshb</td>
<td>INH</td>
</tr>
<tr>
<td>psha</td>
<td>INH</td>
</tr>
<tr>
<td>tpa</td>
<td>INH</td>
</tr>
<tr>
<td>psha</td>
<td>INH</td>
</tr>
<tr>
<td>pula</td>
<td>body of subroutine INH</td>
</tr>
<tr>
<td>tap</td>
<td>INH</td>
</tr>
<tr>
<td>pulb</td>
<td>INH</td>
</tr>
<tr>
<td>pulx</td>
<td>INH</td>
</tr>
<tr>
<td>puly</td>
<td>INH</td>
</tr>
</tbody>
</table>

Subroutine Calls and Return

- **bsr** - branch to subroutine using REL addressing.
- **jsr** - jumps to subroutine using DIR, EXT, or IND addressing.
- On either bsr or jsr, PC is automatically pushed onto the stack (least significant byte first).
- **rts** - return from subroutine, PC automatically pulled off the stack and jumps to that location.

Example Using bsr and rts

```assembly
org $C000
main lds #$0900 ($CF0900)
    clra ($87)
loop bsr Add1 ($0702)
    bra loop ($20FC)

;********Add1**********
;Purpose: Subtract one from RegA
; Input: RegA
;Output: RegA=Input+1
Add1 inca ($42)
    rts ($3D)
org $FFFE
fdb main
```
Jump and Branch Always

- `jmp` and `bra` instructions are unconditional.
- `bra` uses relative addressing (REL) so it can only be used to jump −128 or 127 instructions.
- `jmp` can use EXT and IND addressing so it can be used to jump anywhere in the 64K address space.
- `bra` $ stops progress of CPU, but it continues to execute this instruction.

Single Condition Branches

- `bcc` - branch if carry clear (i.e., $C = 0$).
- `bcs` - branch if carry set (i.e., $C = 1$).
- `bne` - branch if not equal to zero (i.e., $Z = 0$).
- `beq` - branch if equal to zero (i.e., $Z = 1$).
- `bpl` - branch if positive or zero (i.e., $N = 0$).
- `bmi` - branch if negative (i.e., $N = 1$).
- `bvc` - branch if overflow clear (i.e., $V = 0$).
- `bvs` - branch if overflow set (i.e., $V = 1$).
- `brn` - branch never
Example: Tests for Equality

C Code           Assembly Code
-----------------------------
if (G2==G1) {
    ldaa G2
    cmpa G1
}  
    beq next ;skip if equal
    jsr isEqual ;G2==G1
next
-----------------------------
if (G2!=G1) {
    ldaa G2
    cmpa G1
}  
    bne next ;skip if not equal
    jsr isEqual ;G2!=G1
next

Unsigned Number Branches

- These branches usually follow cba, cmp(A,B,D), cp(X,Y), sba, sub(A,B,D) instructions.
- bhi - branch if higher '>=' (i.e., C + Z = 0).
- bhs - branch if higher or same '>=' (i.e., C = 0).
- blo - branch if lower '<=' (i.e., C = 1).
- bls - branch if lower or same '<=' (i.e., C + Z = 1).

Signed Number Branches

- These branches usually follow cba, cmp(A,B,D), cp(X,Y), sba, sub(A,B,D) instructions.
- bgt - branch if greater '>' (i.e., Z ∙ (N ⊕ V) = 0).
- bge - branch if greater or equal '>=' (i.e., N ⊕ V = 0).
- bit - branch if less '<' (i.e., N ⊕ V = 1).
- ble - branch if less or equal '<=' (i.e., Z ∙ (N ⊕ V) = 1).

Example: Unsigned Tests

C Code           Assembly Code
-----------------------------
unsigned int G1;    ldaa G2
unsigned int G2;    cmpa G1
if (G2 > G1) {
    bls next ;skip if G2<=G1
        isGreater();  jsr isGreater ;G2>G1
}  
next
-----------------------------
unsigned int G1;    ldaa G2
unsigned int G2;    cmpa G1
if (G2 > G1) {
    blo next ;skip if G2>G1
        isGreaterEq();  jsr isGreaterEq ;G2>=G1
}  
next
Bit Masking Branches

- `brset` - performs logical AND of memory address and mask provided and branches only when all bits in the mask are set.
- `brclr` - performs logical AND of memory address and mask provided and branches only when all bits in the mask are clear.

<table>
<thead>
<tr>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>brset</td>
<td>5,$10,$F889</td>
<td>goto $F889 if bit 4 of loc 5 is 1</td>
</tr>
<tr>
<td>brclr</td>
<td>5,$10,$F889</td>
<td>goto $F889 if bit 4 of loc 5 is 0</td>
</tr>
</tbody>
</table>

Long Branches and Delays

- Use branch and jump to branch outside of range:
  ```
  Address    Op    Operand     Comment
  C3B0       bcs    $40        Next instruction
  C3B2       jmp    PI         PI may be any valid address.
  ...
  C3F2       jmp    R1        PI may be any valid address.
  ```
- Branch loops can be used to insert a time delay:
  ```
  Label      Op   Operand     Comment
  ldab       #$,Count  Load B with count             
  Delay      decb       decrement B
  bne        Delay      If $B \neq 0$ goto Delay    
  ```

  \[ DelayTime = t(1dab) + (t(decb) + t(bne)) \times count \]

Interrupt Handling

- `wai` instruction puts CPU into standby mode waiting for an external interrupt.
- `swi` instruction executes a software interrupt.
- `rti` instruction is called at the end of an interrupt service routine to restore the CPU registers.

Miscellaneous

- `nop` - no operation, creates a 2-cycle delay.
- `stop` - if $S$ flag in CCR is 0 then stop clocks to save power, recover after `RESET`, `XIRQ`, or unmasked `IRQ`.
Assembler Pseudo-ops

- Set the location to put the following object code (org, .org):
  ```assembly
  org <expression>
  ```
- Equate symbol to a value (equ, =):
  ```assembly
  <label> equ <expression>
  ```
- Redefinable equate symbol to a value (set):
  ```assembly
  <label> set <expression>
  ```
- Reserve multiple bytes (rmb, ds, ds.b, .blkb):
  ```assembly
  <label> rmb <expression>
  ```
- Reserve multiple words (ds.w, .blkw):
  ```assembly
  <label> ds.w <expression>
  ```
- Reserve multiple 32-bit words (ds.1, .blkl):
  ```assembly
  <label> ds.1 <expression>
  ```

Assembler Pseudo-ops (cont)

- Form constant byte (fcb, dc.b, db, .byte):
  ```assembly
  <label> fcb <expression>
  ```
- Form double byte (fdb, dc.w, dw, .word):
  ```assembly
  <label> fdb <expression>
  ```
- Define 32-bit constant (dc.l, dl, .long):
  ```assembly
  <label> fdb <expression>
  ```
- Form constant character string (fcc):
  ```assembly
  hello fcc ‘Hello World’,0
  ```

Using equ for Constants

```assembly
org $3800
size equ 5
data rmb size
org $4000
sum ldaa #size
ldx #data
clrb
loop addb 1,x+
dbne A,loop
rts
```

A Stepper Motor Controller

```assembly
size equ 4
PTT equ $0240
DDRT equ $0242
org $4000
main movb #$FF,DDRT ;PT3-0 outputs
run ldaa #size
ldx #steps
step movb 1,x+,PTT ;step motor
dbne A,step
bra run
steps fcb 5,6,10,9 ;out sequence
org $FFFE
fdb main
```
Memory Allocation in Intel x86

Memory Allocation in Embedded System

Memory Allocation in Software

```
org $3800 ;RAM
cnt rmb 1 ;global
org $4000 ;EEPROM
const fcb 5 ;amount to add
init movb #$FF,DDRT ;outputs
clr cnt
rts
main lds #$4000 ;sp=>RAM
bsr init
loop ldd cnt
staa PTT ;output
adda const
staa cnt
bra loop
org $FFFE ;EEPROM
fdb main ;reset vector
```