Students are now assigned to lab sections
   If you’re not in a section mail Rohit right away
   If your team or section assignment is wrong mail Rohit right away
Labs start next week
Last Time

HCS12 architecture
Representing numbers
  Signed and unsigned
  1 or 2 bytes
  Endian issues
Fixed point numbers and arithmetic
Simple addressing modes
Example HCS12 application
Assembly Language Development Process

Source code

; MC9S12C32
PTT equ $0240
DDRT equ $0242
org $4000
Main ldaa #$0F
staa DDRT
Controller
  ldaa #5
  staa PTT ; 0101
  ldaa #6
  staa PTT ; 0110
  ldaa #10
  staa PTT ; 1010
  ldaa #9
  staa PTT ; 1001
  bra Controller
  org $FFFE
  fdb Main

Assembler

Object code

$4000 860F
$4002 7A0242
$4005 8605
$4007 7A0240
$400A 8606
$400C 7A0240
$400F 860A
$4011 7A0240
$4014 8609
$4016 7A0240
$4019 20EA
$FFFE 4000

Loader

Microcontroller

Processor

RAM

ROM

860F7A02428605
7A024086067A02
40860A7A024086
097A024020EA
4000

I/O ports

External circuits and devices
Assembly Language Syntax

<table>
<thead>
<tr>
<th>Label</th>
<th>Operation</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTA</td>
<td>equ</td>
<td>$0000</td>
<td>Assembly time constant</td>
</tr>
<tr>
<td>Inp</td>
<td>ldaa</td>
<td>PORTA</td>
<td>Read data from PORTA</td>
</tr>
</tbody>
</table>

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.
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>#{expression}</td>
<td>IMM</td>
<td>ldaa #4</td>
</tr>
<tr>
<td>{expression}</td>
<td>DIR,EXT,REL</td>
<td>ldaa 4</td>
</tr>
<tr>
<td>{expression},idx</td>
<td>indexed (IND)</td>
<td>ldaa 4,x</td>
</tr>
<tr>
<td>{expr},#{expr}</td>
<td>bit set or clear</td>
<td>bset 4,#$01</td>
</tr>
<tr>
<td>{expr},#{expr},{expr}</td>
<td>bit test &amp; branch</td>
<td>brset 4,#$01,foo</td>
</tr>
<tr>
<td>{expr},idx,#{expr}</td>
<td>bit test &amp; branch</td>
<td>brset 4,x,#$01,foo</td>
</tr>
<tr>
<td>{expression},idx+</td>
<td>IND, post incr</td>
<td>ldaa 4,x+</td>
</tr>
<tr>
<td>{expression},idx-</td>
<td>IND, post decr</td>
<td>ldaa 4,x-</td>
</tr>
<tr>
<td>{expression},+idx</td>
<td>IND, pre incr</td>
<td>ldaa 4,+x</td>
</tr>
<tr>
<td>{expression},-idx</td>
<td>IND, pre decr</td>
<td>ldaa 4,-x</td>
</tr>
<tr>
<td>acc, idx</td>
<td>accum offset IND</td>
<td>ldaa A,x</td>
</tr>
<tr>
<td>[&lt;expression&gt;,idx]</td>
<td>IND indirect</td>
<td>ldaa [4,x]</td>
</tr>
<tr>
<td>[D,idx]</td>
<td>RegD IND indirect</td>
<td>ldaa [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):

<table>
<thead>
<tr>
<th>Obj code</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6A5C</td>
<td>staa</td>
<td>-4,Y</td>
<td>;[Y-4] = RegA</td>
</tr>
</tbody>
</table>

![Diagram showing Indexed Addressing Mode](image)
Building the Object Code

\( \text{staa } -4,Y \rightarrow $6A5C \)

First byte is $6A - \text{Op code (pg. 254)}
Second byte is formatted as \( \%rr0\text{n}nnnnn \) (pg. 33).
\( rr \) is \( \%01 \) for register \( Y \).
\( nnnnn \) is \( \%11100 \) for \(-4\).
\( \%0101 1100 \rightarrow $5C \).

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 %rr1pnnnn (pg. 33).
rr is %00 for register X.
nnnn 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, ;store $56 at $1234</td>
</tr>
</tbody>
</table>

![Diagram showing memory access and addressing]
Accumulator D Offset Indexed Indirect

Offset is in D and index is in another 16-bit register. Computed address 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></td>
</tr>
<tr>
<td>stx</td>
<td>[D,Y]</td>
<td>;Store value in RegX at $1234</td>
</tr>
</tbody>
</table>

Diagram:

- D: 0004
- X: 5678
- Y: 2341
- D+Y=2345
- 1233: 56
- 1234: 78
- 2344: 12
- 2345: 34
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.
```
Load and Store Instructions

Used to move data to (from) registers from (to) memory.
Load instructions are: ldaa, ldab, ldd, lds, ldx, and ldy.
Load addressing modes are: IMM, DIR, EXT, IND.
Store instructions are: staa, stab, std, sds, sdx, and sdy.
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>ldaa</td>
<td>#$FF</td>
<td>IMM</td>
</tr>
<tr>
<td>staa</td>
<td>$25</td>
<td>DIR</td>
</tr>
<tr>
<td>ldab</td>
<td>$0025</td>
<td>EXT</td>
</tr>
<tr>
<td>std</td>
<td>$05,X</td>
<td>IND</td>
</tr>
<tr>
<td>ldd</td>
<td>$C025</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}
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:

<table>
<thead>
<tr>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>clra</td>
<td></td>
<td>INH</td>
</tr>
<tr>
<td>clr</td>
<td>$0025</td>
<td>EXT</td>
</tr>
</tbody>
</table>

The carry (C), interrupt mask (I), and overflow (V) bits in the CC can also be set (sec, sei, sev).
Transfer instructions used to copy data between registers.

\[ \text{tab, tap, tba, tpa, tsx, tsy, txs, tys (all INH)} \]

Exchange instructions used to exchange data between accumulator D and index registers X and Y.

\[ \text{xgdx, xgdy (all INH)} \]
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

<table>
<thead>
<tr>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldaa</td>
<td>$25</td>
<td>load least sig byte</td>
</tr>
<tr>
<td>adda</td>
<td>$35</td>
<td>add data at $35 to A</td>
</tr>
<tr>
<td>staa</td>
<td>$45</td>
<td>store least sig byte</td>
</tr>
<tr>
<td>ldaa</td>
<td>$24</td>
<td>load most sig byte</td>
</tr>
<tr>
<td>adca</td>
<td>$34</td>
<td>add data at $34 to A</td>
</tr>
<tr>
<td>staa</td>
<td>$44</td>
<td>store most sig byte</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Obj code</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$8650</td>
<td>ldaa</td>
<td>#$50</td>
<td>load set point into A</td>
</tr>
<tr>
<td>$B11031</td>
<td>cmpa</td>
<td>$1031</td>
<td>compare A to memory</td>
</tr>
</tbody>
</table>

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.
Multiply Instructions

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

Example:

<table>
<thead>
<tr>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldaa</td>
<td>#$FF (255)</td>
<td>IMM</td>
</tr>
<tr>
<td>ldab</td>
<td>#$14 (20)</td>
<td>IMM</td>
</tr>
<tr>
<td>mul</td>
<td></td>
<td>INH</td>
</tr>
</tbody>
</table>

At the end, accumulator $D$ contains $13EC$ (5100).

$FF \times FF = FE01$
Use \( D \) register for the dividend and \( X \) register for the divisor. Resultant placed in \( X \) register and remainder in \( D \) register. \textit{idiv} performs integer division.

\[
\begin{array}{ccc}
\text{Register } D & / & \text{Register } X \\
\hline
\text{Remainder} & = & \text{Register } D
\end{array}
\]

Example:

<table>
<thead>
<tr>
<th>Obj code</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CCFFFFF</td>
<td>ldd</td>
<td>#$FFFF (65535)</td>
<td>After \textit{idiv} executes</td>
</tr>
<tr>
<td>$CE2710</td>
<td>ldx</td>
<td>#$2710 (10000)</td>
<td>( X ) contains $0006</td>
</tr>
<tr>
<td>$02</td>
<td>idiv</td>
<td></td>
<td>( D ) contains $159F (5535)</td>
</tr>
</tbody>
</table>
\( \text{fdiv} \) performs fractional division resulting in binary weighted fraction between 0 and 0.999998 (i.e., \( (65536 \times D)/X \rightarrow X \)).

\[
\begin{array}{|c|c|}
\hline
\text{Register D} & 0 \\
\hline
\end{array}
\bigg/\bigg\
\begin{array}{|c|c|}
\hline
\text{Register X} & = \text{Register X} \\
\hline
\end{array}
\bigg/\bigg\
\text{Remainder} = \begin{array}{|c|}
\hline
\text{Register D} \\
\hline
\end{array}
\]

Numerator must be less than denominator or overflow occurs. Next 16-bits of the fraction can be obtained by reloading the denominator and doing \( \text{fdiv} \) again.
fdiv Example

<table>
<thead>
<tr>
<th>Op</th>
<th>Operand</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldd</td>
<td>#1</td>
</tr>
<tr>
<td>ldx</td>
<td>#3</td>
</tr>
<tr>
<td>fdiv</td>
<td></td>
</tr>
</tbody>
</table>

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: 0.333333333325572
emul and emuls perform $16 \times 16$ unsigned and signed multiplication.

ediv and edivs perform $32 \times 16$ unsigned and signed division.
Example: \[ M = \frac{(53 \times N + 50)}{100} \]

```
ldd N ;RegD=N (between 0 and 65535)
ldy #53
emul ;RegY:D=53*N (between 0 and 3473355)
add #50 ;RegD=53*N+50 (between 0 and 3473355)
bcc skip
iny
skip ldx #100
ediv ;RegY=(53*N+50)/100 (between 0 and 34734)
sty M
```
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.

$$< U > = < U > + \{ X \} \ast \{ Y \}$$
Logical shift right (lsr, lsra, lsrb, lsrd) shifts 0’s into MSB.

Arithmetic shift right (asr, asra, asrb) retains MSB value.

Logical shift left (lsl, lsla, lslb, lsld) and arithmetic shift left (asl, asla, aslb, asld) are equivalent (same op code).
Rotate Instructions

Rotate right (ror, rora, rorb) put carry bit into the MSB. Rotate left (rol, rola, rolb) put carry bit into the LSB.
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>ldaa</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
bita 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.

\begin{align*}
\text{bclr} & \text{ addr, mm} \\
\text{bset} & \text{ addr, mm}
\end{align*}

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></td>
<td>body of subroutine</td>
</tr>
<tr>
<td>pula</td>
<td>INH</td>
</tr>
<tr>
<td>tap</td>
<td>INH</td>
</tr>
<tr>
<td>pula</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

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 $FFFFE
fdb main
bsr and rts Execution

org $C000
main:  lds #$0900
clda
loop:  bsr Add1
bra  loop
Add1:  inca
rts

<table>
<thead>
<tr>
<th>$C000</th>
<th>$CF09</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C002</td>
<td>$0087</td>
</tr>
<tr>
<td>$C004</td>
<td>$0702</td>
</tr>
<tr>
<td>$C006</td>
<td>$20FC</td>
</tr>
<tr>
<td>$C008</td>
<td>$423D</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$08FE</td>
<td>$XXXX</td>
</tr>
<tr>
<td>$0900</td>
<td>$XXXX</td>
</tr>
</tbody>
</table>

SP $0900
PC $C000
A $XX
bsr and rts Execution

org $C000

main: lds #$0900
   clra
loop: bsr Add1
   bra loop
Add1: inca
   rts

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C000</td>
<td>$CF09</td>
<td></td>
</tr>
<tr>
<td>$C002</td>
<td>$0087</td>
<td></td>
</tr>
<tr>
<td>$C004</td>
<td>$0702</td>
<td></td>
</tr>
<tr>
<td>$C006</td>
<td>$20FC</td>
<td></td>
</tr>
<tr>
<td>$C008</td>
<td>$423D</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>$0900</td>
<td></td>
</tr>
<tr>
<td>PC</td>
<td>$C003</td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>$00</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$08FE</td>
<td>$XXXX</td>
</tr>
<tr>
<td>$0900</td>
<td>$XXXX</td>
</tr>
</tbody>
</table>
bsr and rts Execution

org $C000
main: lds #$0900
clra
loop: bsr Add1
bra loop
Add1: inca
rts

<table>
<thead>
<tr>
<th>location</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C000</td>
<td>$C000</td>
</tr>
<tr>
<td>$C002</td>
<td>$C002</td>
</tr>
<tr>
<td>$C004</td>
<td>$C004</td>
</tr>
<tr>
<td>$C006</td>
<td>$C006</td>
</tr>
<tr>
<td>$C008</td>
<td>$C008</td>
</tr>
<tr>
<td>$08FE</td>
<td>$08FE</td>
</tr>
<tr>
<td>$0900</td>
<td>$0900</td>
</tr>
<tr>
<td>$XXXX</td>
<td>$XXXX</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>register</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>$08FE</td>
</tr>
<tr>
<td>PC</td>
<td>$C004</td>
</tr>
<tr>
<td>A</td>
<td>$00</td>
</tr>
</tbody>
</table>
### bsr and rts Execution

**org** $C000

**main:** lds #$0900

clra

**loop:** bsr Add1

bra loop

**Add1:** inca

rts

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C000</td>
<td>lds #$0900</td>
<td>$0900</td>
</tr>
<tr>
<td>$C002</td>
<td>clra</td>
<td>$0087</td>
</tr>
<tr>
<td>$C004</td>
<td>bsr Add1</td>
<td>$0702</td>
</tr>
<tr>
<td>$C006</td>
<td>bra loop</td>
<td>$20FC</td>
</tr>
<tr>
<td>$C008</td>
<td>inca</td>
<td>$423D</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>$08FE</td>
<td>sp</td>
<td>$08FE</td>
</tr>
<tr>
<td>$C000</td>
<td>rts</td>
<td>$01</td>
</tr>
<tr>
<td>$0900</td>
<td></td>
<td>$XXXX</td>
</tr>
</tbody>
</table>
bsr and rts Execution

```
org $C000
main: lds #$0900
clra
loop:  bsr Add1
       bra  loop
Add1:  inca
       rts
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$C000</td>
<td>$CF09</td>
</tr>
<tr>
<td>$C002</td>
<td>$0087</td>
</tr>
<tr>
<td>$C004</td>
<td>$0702</td>
</tr>
<tr>
<td>$C006</td>
<td>$20FC</td>
</tr>
<tr>
<td>$C008</td>
<td>$423D</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$08FE</td>
<td>$C006</td>
</tr>
<tr>
<td>$0900</td>
<td>$XXXX</td>
</tr>
<tr>
<td>SP</td>
<td>$0900</td>
</tr>
<tr>
<td>PC</td>
<td>$C009</td>
</tr>
<tr>
<td>A</td>
<td>$01</td>
</tr>
</tbody>
</table>
```
bsr and rts Execution

**main:**
org $C000
lds #$0900
clla

**loop:**
bsr Add1
bra loop

**Add1:**
inca

```
<table>
<thead>
<tr>
<th>SP</th>
<th>$0900</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>$C006</td>
</tr>
<tr>
<td>A</td>
<td>$01</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>$C000</th>
<th>$CF09</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C002</td>
<td>$0087</td>
</tr>
<tr>
<td>$C004</td>
<td>$0702</td>
</tr>
<tr>
<td>$C006</td>
<td>$20FC</td>
</tr>
<tr>
<td>$C008</td>
<td>$423D</td>
</tr>
<tr>
<td>$08FE</td>
<td>$C006</td>
</tr>
<tr>
<td>$0900</td>
<td>$XXXX</td>
</tr>
</tbody>
</table>
```
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.
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
                isEqual();   cmpa G1
    }
                bne next ;skip if not equal
                jsr isEqual ;G2==G1

next

if (G2!=G1) {    ldaa G2
                isEqual();   cmpa G1
    }
                beq next ;skip if equal
                jsr isNotEqual ;G2!=G1

next
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 '\(\geq\)' (i.e., \(C = 0\)).

*blo* - branch if lower '<[i.e., \(C = 1\)].

*bls* - branch if lower or same '\(\leq\)' (i.e., \(C + Z = 1\)).
These branches usually follow $\text{cba, cmp(A,B,D), cp(X,Y), sba, sub(A,B,D)}$ instructions.

- **bgt** - branch if greater ‘$>$’ (i.e., $Z + (N \oplus V) = 0$).
- **bge** - branch if greater or equal ‘$\geq$’ (i.e., $N \oplus V = 0$).
- **blt** - branch if less ‘$<$’ (i.e., $N \oplus V = 1$).
- **ble** - branch if less or equal ‘$\leq$’ (i.e., $Z + (N \oplus V) = 1$).
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>
Use branch and jump to branch outside of range:

<table>
<thead>
<tr>
<th>Address</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>C3B0</td>
<td>bcs</td>
<td>$40</td>
<td></td>
</tr>
<tr>
<td>C3B2</td>
<td></td>
<td></td>
<td>Next instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C3F2</td>
<td>jmp</td>
<td>PI</td>
<td>PI may be any valid address.</td>
</tr>
</tbody>
</table>

Branch loops can be used to insert a time delay:

<table>
<thead>
<tr>
<th>Label</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldab</td>
<td>#$Count</td>
<td></td>
<td>Load B with count</td>
</tr>
<tr>
<td>Delay</td>
<td>decb</td>
<td></td>
<td>decrement B</td>
</tr>
<tr>
<td>bne</td>
<td>Delay</td>
<td></td>
<td>If $B \neq 0$ goto Delay</td>
</tr>
</tbody>
</table>

\[
\text{DelayTime} = t(\text{ldab}) + (t(\text{decb}) + t(\text{bne})) \times \text{count}
\]
**Miscellaneous**

**nop** — no operation, creates a 2-cycle delay.

**swi** — trigger a software interrupt.

**rti** — called at the end of an *interrupt service routine* to restore the CPU registers.

**wai** — puts CPU into standby mode waiting for an interrupt; CPU clock is stopped but other MCU clocks can continue to run.

**stop** — stop all clocks to save power; start on *RESET*, *XIRQ*, or unmasked *IRQ*; RAM, I/O space, and registers are preserved.
Set the location to put the following object code (org, .org):

\[
\text{org} \ <\text{expression}> \\
\]

Equate symbol to a value (equ, =):

\[
<\text{label}> \ \text{equ} \ <\text{expression}> \\
\]

Redefinable equate symbol to a value (set):

\[
<\text{label}> \ \text{set} \ <\text{expression}> \\
\]

Reserve multiple bytes (rmb, ds, ds.b, .blkb):

\[
<\text{label}> \ \text{rmb} \ <\text{expression}> \\
\]

Reserve multiple words (ds.w, .blkw):

\[
<\text{label}> \ \text{ds.w} \ <\text{expression}> \\
\]

Reserve multiple 32-bit words (ds.l, .blkl):

\[
<\text{label}> \ \text{ds.l} \ <\text{expression}> \\
\]
Using equ for Constants

org $3800
size equ 5
data rmb size
org $4000
sum ldaa #size
ldx #data
clrb
loop addb 1,x+
dbne A,loop
rts
Form constant byte (fcb, dc.b, db, .byte):

    <label> fcb <expression>

Form double byte (fdb, dc.w, dw, .word):

    <label> fdb <expression>

Define 32-bit constant (dc.l, dl, .long):

    <label> fqb <expression>

Form constant character string (fcc):

    hello fcc ‘‘Hello World’’,0
Simple Stepper Motor Controller

```
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
```
Assembler is the program that converts assembly language into object code that can be executed by the processor.

Addressing modes are key to reading and writing assembly language.

Condition codes and subsequent branches are also very important.

For efficiency and convenience, HCS12 provides extensive math operations wider than 8 bits.

Otherwise, math on the HCS12 is totally standard.

Assembly language permits very basic, very serious mistakes such as failing to save the right registers on a function call, creating mismatched stack frames, improper use of condition code registers, etc.