行政小知识

- 2 版本的 CodeWarrior 都在实验室机器上。
- 你应该使用 4.5 版本（CW for HC12 v 4.5）。
- 实验 1 在昨天早上被更新了。
- 两名团队成员都需要出席进行检查。
- 实验报告不需要是"报告"格式。
### Assembly Language Development Process

Source code

```assembly
; MC9S12C32
PTT equ $0240
DDRT equ $0242
org $4000
Main ldaa #30F
staa DDRT
Controller ldaa #5
staa PTT ; 0101
lda #6
staa PTT ; 0110
ldaa #10
staa PTT ; 1010
ldaa #9
staa PTT ; 1001
bra Controller
org $FFFE
fdb Main
```

Assembler

Object code

**Processor**

**RAM**

**ROM**

860F7A02428605
7A024086067A02
40860A7A024086
097A024020BA

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.

<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>ldaa #4</td>
</tr>
<tr>
<td>&lt;expression&gt;</td>
<td>DIR,EXT,REL</td>
<td>ldaa 4</td>
</tr>
<tr>
<td>&lt;expression&gt;,idx</td>
<td>indexed (IND)</td>
<td>ldaa 4,x</td>
</tr>
<tr>
<td>&lt;expr&gt;,#&lt;expr&gt;</td>
<td>bit set or clear</td>
<td>bset 4,#$01,foo</td>
</tr>
<tr>
<td>&lt;expr&gt;,#&lt;expr&gt;,&lt;expr&gt;</td>
<td>bit test &amp; branch</td>
<td>brset 4,#$01,foo</td>
</tr>
<tr>
<td>&lt;expr&gt;,idx,#&lt;expr&gt;,&lt;expr&gt;</td>
<td>bit test &amp; branch</td>
<td>brset 4,x,#$01,foo</td>
</tr>
<tr>
<td>&lt;expression&gt;,idx+</td>
<td>IND, post incr</td>
<td>ldaa 4,x+</td>
</tr>
<tr>
<td>&lt;expression&gt;,idx-</td>
<td>IND, post decr</td>
<td>ldaa 4,x-</td>
</tr>
<tr>
<td>&lt;expression&gt;,+idx</td>
<td>IND, pre incr</td>
<td>ldaa 4,+x</td>
</tr>
<tr>
<td>&lt;expression&gt;,-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>

```
Y $0823
A $56
```

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+ $\rightarrow$ $6A30$
  - First byte is $6A$ - Op code (pg. 254)
  - Second byte is formatted as %rr1pnnn (pg. 33).
  - rr is %00 for register X.
  - nnnn is %0000 for 1.
  - p is %1 for post.
  - %0011 0000 $\rightarrow$ $30$.
  - 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>;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>;Fetch 16-bit address from $2341,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;store $56 at $1234</td>
</tr>
</tbody>
</table>

  ![Diagram](image)
### 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></td>
</tr>
<tr>
<td>stx</td>
<td>[D,Y]</td>
<td>;Store value in RegX at $1234</td>
</tr>
</tbody>
</table>
  ```

### Load Effective 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. |
  ```
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 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>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>$05,X</td>
<td>IND</td>
</tr>
<tr>
<td>1dd</td>
<td>$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:
  \[ \text{movb} \ #w, \text{addr} \ [\text{addr}]=w \]
- Move an 8-bit value memory to memory:
  \[ \text{movb} \ \text{addr1},\text{addr2} \ [\text{addr2}]=[\text{addr1}] \]
- Move a 16-bit constant into memory:
  \[ \text{movw} \ #W,\text{addr} \ \{\text{addr}\}=W \]
- Move a 16-bit value memory to memory:
  \[ \text{movw} \ \text{addr1},\text{addr2} \ \{\text{addr2}\}=[\text{addr1}] \]
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:
  - 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).

Other Data Movement Instructions

- Transfer instructions used to copy data between registers.
  - 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.
  - 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>ldax</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>ldax</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>

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, cpuy.
- 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>ldax</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, tsra, tsrb` - 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`).

\[
\begin{array}{c|c|c}
\text{Register A} & \times & \text{Register B} \\
8 \text{ bits} & & 8 \text{ bits} \\
\hline
\text{Register D} & \text{16 bits}
\end{array}
\]

- 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 = $FE01$`
**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.

\[
\text{Register } D \div \text{Register } X = \text{Register } X \\
\text{Remainder} = \text{Register } D
\]

- Example:

<table>
<thead>
<tr>
<th>Obj code</th>
<th>Op</th>
<th>Operand</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CCFFFF$</td>
<td>1dd</td>
<td>#$FFFF (65535)</td>
<td>After <code>idiv</code> executes</td>
</tr>
<tr>
<td>$CE2710$</td>
<td>1dx</td>
<td>#$2710 (10000)</td>
<td>$X$ contains $0006$</td>
</tr>
<tr>
<td>$02$</td>
<td><code>idiv</code></td>
<td></td>
<td>$D$ contains $159F (5535)$</td>
</tr>
</tbody>
</table>

**Fractional Divide Instruction**

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

\[
\text{Register } D \div \text{Register } X = \text{Register } X \\
\text{Remainder} = \text{Register } D
\]

- 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.

\[
\begin{array}{c}
\text{Reg Y} \\
16 \text{ bits}
\end{array} \times \begin{array}{c}
\text{Reg D} \\
16 \text{ bits}
\end{array} = \begin{array}{c}
\text{Reg Y} \\
32 \text{ bits}
\end{array} \quad \begin{array}{c}
\text{Reg D}
\end{array}
\]

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

\[
\begin{array}{c}
\text{Reg Y} \\
32 \text{ bits}
\end{array} \quad \begin{array}{c}
\text{Reg D}
\end{array} \div \begin{array}{c}
\text{Reg X} \\
16 \text{ bits}
\end{array} = \begin{array}{c}
\text{Reg Y} \\
16 \text{ bits}
\end{array} \
\begin{array}{c}
\text{Remainder}
\end{array} \begin{array}{c}
\text{Reg D}
\end{array}
\]
Example: \( M = (53 \times N + 50)/100 \)

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

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 (lsl, lsra, lsrb, lsr) shifts 0's into MSB.
  ![LSR Diagram]

- Arithmetic shift right (asr, asra, asrb) retains MSB value.
  ![ASR Diagram]

- Logical shift left (lsl, ls1a, lslb, ls1d) and arithmetic shift left (asl, as1a, aslb, as1d) are equivalent (same op code).
  ![LSL/ASL Diagram]

Rotate Instructions

- Rotate right (ror, rora, rorb) put carry bit into the MSB.
  ![ROR Diagram]

- Rotate left (rol, rola, rolb) put carry bit into the LSB.
  ![ROL Diagram]
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>
<tr>
<td></td>
<td></td>
<td></td>
<td>Result in A is %00000100</td>
</tr>
</tbody>
</table>

Data Test and Bit Manipulation

- 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.

```
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></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 $FFFE
fdb main
```
### bsr and rts Execution

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</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>SP</td>
<td>$0900</td>
</tr>
<tr>
<td>PC</td>
<td>$C000</td>
</tr>
<tr>
<td>A</td>
<td>$XX</td>
</tr>
</tbody>
</table>

- **main:**
  - `lds #0900`
  - `clra`
- **loop:**
  - `bsr Add1`
  - `bra loop`
- **Add1:**
  - `inca`
  - `rts`
## bsr and rts Execution

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>org $C000</td>
<td></td>
<td>$C00</td>
</tr>
<tr>
<td>main: lds #$0900</td>
<td></td>
<td>$CF09</td>
</tr>
<tr>
<td>clra</td>
<td></td>
<td>$C002</td>
</tr>
<tr>
<td>loop: bsr Add1</td>
<td></td>
<td>$0087</td>
</tr>
<tr>
<td>bra loop</td>
<td></td>
<td>$C004</td>
</tr>
<tr>
<td>Add1: inca</td>
<td></td>
<td>$0702</td>
</tr>
<tr>
<td>rts</td>
<td></td>
<td>$C006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$20FC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$423D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$XXX</td>
</tr>
</tbody>
</table>

### Stack Values

- SP: $08FE
- PC: $C004
- A: $00

---

### Loop Execution

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>org $C000</td>
<td></td>
<td>$C00</td>
</tr>
<tr>
<td>main: lds #$0900</td>
<td></td>
<td>$CF09</td>
</tr>
<tr>
<td>clra</td>
<td></td>
<td>$C002</td>
</tr>
<tr>
<td>loop: bsr Add1</td>
<td></td>
<td>$0087</td>
</tr>
<tr>
<td>bra loop</td>
<td></td>
<td>$C004</td>
</tr>
<tr>
<td>Add1: inca</td>
<td></td>
<td>$0702</td>
</tr>
<tr>
<td>rts</td>
<td></td>
<td>$C006</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$20FC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$423D</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$XXX</td>
</tr>
</tbody>
</table>

### Stack Values

- SP: $08FE
- PC: $C008
- A: $01
bsr and rts Execution

<table>
<thead>
<tr>
<th>org</th>
<th>main:</th>
<th>loop:</th>
<th>Add1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C000</td>
<td>lds</td>
<td>bsr</td>
<td>inca</td>
</tr>
<tr>
<td></td>
<td>#$0900</td>
<td>Add1</td>
<td>rts</td>
</tr>
<tr>
<td>clra</td>
<td></td>
<td>bra</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>$C002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0087</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0702</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$20FC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$423D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$08FE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0900</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$XXXX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scott R. Little (Lecture 3: Assembly)  ECE/CS 5780/6780  41 / 59

bsr and rts Execution

<table>
<thead>
<tr>
<th>org</th>
<th>main:</th>
<th>loop:</th>
<th>Add1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C000</td>
<td>lds</td>
<td>bsr</td>
<td>inca</td>
</tr>
<tr>
<td></td>
<td>#$0900</td>
<td>Add1</td>
<td>rts</td>
</tr>
<tr>
<td>clra</td>
<td></td>
<td>bra</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>$C002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0087</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C004</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0702</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$20FC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C008</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$423D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>$08FE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$C006</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$0900</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$XXXX</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scott R. Little (Lecture 3: Assembly)  ECE/CS 5780/6780  42 / 59
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., \( C = 0 \)).
- `bc\( s\)` - 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) {
    ld   G2
    cmp  G1
}  bne next ;skip if not equal
    jsr  isEqual ;G2==G1
next

====================================================================
if (G2!=G1) {
    ld   G2
    cmp  G1
}  beq next ;skip if equal
    jsr  isNotEqual ;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., $Z \cdot (N \oplus V) = 0$).
- bge - branch if greater or equal '>=' (i.e., $N \oplus V = 0$).
- blt - branch if less '<' (i.e., $N \oplus V = 1$).
- ble - branch if less or equal '<=' (i.e., $Z \cdot (N \oplus V) = 1$).

Example: Unsigned Tests

<table>
<thead>
<tr>
<th>C Code</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned int G1;</td>
<td>ldaa G2</td>
</tr>
<tr>
<td>unsigned int G2;</td>
<td>cmpa G1</td>
</tr>
<tr>
<td>if (G2 &gt; G1) {</td>
<td>bls next ;skip if G2&lt;=G1</td>
</tr>
<tr>
<td>isGreater();</td>
<td>jsr isGreater ;G2&gt;G1</td>
</tr>
<tr>
<td>}</td>
<td>next</td>
</tr>
<tr>
<td>unsigned int G1;</td>
<td>ldaa G2</td>
</tr>
<tr>
<td>unsigned int G2;</td>
<td>cmpa G1</td>
</tr>
<tr>
<td>if (G2 &gt; G1) {</td>
<td>blo next ;skip if G2&lt;G1</td>
</tr>
<tr>
<td>isGreaterEq();</td>
<td>jsr isGreaterEq ;G2&gt;=G1</td>
</tr>
<tr>
<td>}</td>
<td>next</td>
</tr>
</tbody>
</table>
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            |
  - ...     |     |         |                             |
  - C3F2    | jmp | PI      | PI may be any valid address.

- Branch loops can be used to insert a time delay:
  - Label | Op  | Operand | Comment                   |
  - ldam  | #$Count | Load B with count       |
  - Delay | deb |         | decrement B               |
  - bne   | Delay | If $B \neq 0$ goto Delay |

- $DelayTime = t(ldam) + (t(decb) + t(bne)) \times count$
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):
  
  \( \text{org<expression>} \)

- Equate symbol to a value (equ, =):
  
  \(<\text{label}>\text{equ<expression}>\)

- Redefinable equate symbol to a value (set):
  
  \(<\text{label}>\text{set<expression}>\)

- Reserve multiple bytes (rmb, ds, ds.b, .blkb):
  
  \(<\text{label}>\text{rmb<expression}>\)

- Reserve multiple words (ds.w, .blkw):
  
  \(<\text{label}>\text{ds.w<expression}>\)

- Reserve multiple 32-bit words (ds.1, .blkl):
  
  \(<\text{label}>\text{ds.1<expression}>\)

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
```
Assembler Pseudo-ops (cont)

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

A 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
```
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 ldaa cnt
   staa PTT ;output
   adda const
   staa cnt
   bra loop

org $FFFE ;EEPROM
fdb main ;reset vector
```

Scott R. Little  (Lecture 3: Assembly)   ECE/CS 5780/6780