The Macro Assembler
Macro Assembler

- Definition from “http://www.computeruser.com”

- “A program that translates assembly language instructions into machine code and which the programmer can use to define macro instructions”.

Comments

• A Comment is:
  – Any text after all operands for a given mnemonic have been processed
  – A line beginning with * (in the first column) up to the end of the line
  – An empty line

* This is a comment

LDAA #$5C  * This is a comment
LDAA #$AA  This is a comment
Labels

• The Assembler has the facility to generate symbolic labels during assembly

• This is a two pass process
  – Step 1, the lengths of instructions are determined so that they may be assigned specific addresses
  – Step 2, instructions that use labels have the label replaced by an address or offset

• The Label must begin in the first column of the text file, and may optionally end in a colon (:).
Label Example

* Looping program

LDAA #$FF  * load A with the number $FF

DECA       * decrement accumulator A

BNE Loop   * branch to ‘Loop’ if not zero
Addressing Modes

- The Assembler will identify what addressing mode each instruction is in, and assign the appropriate opcode.

<table>
<thead>
<tr>
<th>LDAA (opr)</th>
<th>Load Accumulator A</th>
<th>M → A</th>
<th>A IMM</th>
<th>A DIR</th>
<th>A EXT</th>
<th>A IND,X</th>
<th>A IND,Y</th>
<th>86</th>
<th>96</th>
<th>B6</th>
<th>A6</th>
<th>18 A6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

- Labels are automatically de-referenced as extended mode addressing
- Exceptional Behaviour - BRANCHES
Arithmetic Expressions

• The Motorola assembler supports several arithmetic operations which can be used to form values of labels or instruction arguments

• Addition +
• Subtraction −
• Multiplication *
• Division /
• Remainder after division %
• Bitwise AND &
• Bitwise OR |
• Bitwise XOR ^
Constants

• Hexadecimal $ followed by hex number
• Binary % followed by binary number
• Characters ‘ followed by an ascii character
Psuedo-Ops

- **ORG (opr)** Short for ORiGin. This command specifies the location in memory for the assembler to place the following machine code.

- For the EVBPlus board, the address $8800 is the beginning of the largest free block of memory:

  
<table>
<thead>
<tr>
<th>Command</th>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORG</td>
<td>$8800</td>
<td>*start of memory block</td>
</tr>
<tr>
<td>LDS</td>
<td>#$FF</td>
<td>*init stack pointer</td>
</tr>
</tbody>
</table>
Psuedo-Ops 2

- **EQU**: Used to EQUate a symbol to a numeric value (similar to `#define` in C programming)

- Useful for naming memory locations, so names rather than numbers can be remembered when programming.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>portA</td>
<td>EQU</td>
<td>$1000</td>
<td>* port a register location</td>
</tr>
<tr>
<td>ORG</td>
<td></td>
<td>$8800</td>
<td></td>
</tr>
<tr>
<td>LDAA</td>
<td></td>
<td>portA</td>
<td></td>
</tr>
</tbody>
</table>
Psuedo-Ops 3

- FCC Form Constant Character, forms a string of ascii characters in memory.

```
label:    FCC    "Hello World."
*
label points to start of string
```
Psuedo-Ops 4

- FCB Form Constant Byte
- FDB Form Double Byte

Label_1: FCB $12
Label_2: FDB $1007
Psuedo-Ops 5

• RMB Reserve Memory Block – takes an operand, and then skips that many bytes ahead to place code into memory

• Often used to leave a gap in memory that will be filled in by the program at run-time

Label: RMB $03 * reserve 3 bytes of memory
Program Style - Documentation

• Assembly language programs are very hard to read and interpret

• Document all major code blocks (e.g. subroutines). Include a description of purpose, input and output parameters, registers and memory locations used, and perhaps a sample case

• Comment each line

• Macros are provided in the text editor ConText to make this process easier (CTRL-J to select)
Program Style - Subroutines

• Use subroutines to break your program into logical units

• This results in slightly slower execution time (JSR, RTS) but much more readable, understandable and reusable code
Program Style - Design

• Design your code before implementation

• Use flow charts to show the logical flow of the program – then convert to tests and branch instructions

• Break the program into functional reusable modules (normally implemented as subroutines)
Program Style – Debugging/Testing

• Test each module independently

• Use the debugging tools (single stepping/breakpoints) to identify areas of program failure

• Examine the program logic (it may be wrong)

• Eliminate possibilities in a logical, ordered manner