CEG3136
Computer Architecture II

Module 3 – Assembly Language Programming

Notes for
Dr. Voicu Groza

uOttawa
L'Université canadienne
Canada's university

Université d'Ottawa | University of Ottawa

www.uOttawa.ca

Topics of discussion

- Assembly Language Programming
  - The Assembler
  - Code Relocation Problem
  - The Relocatable Assembler and Linker
- MC9S12DG256 Assembly Language Programming
  - The MiniIDE Assembler
- D-Bug12 Monitor
- Reading: Text book – Chapter 5, MiniIDE Documentation

Assembly Language Programming

- Assembly language program manipulates resources in the programmer's model
  - Includes CPU registers
  - Includes CPU hardware resources
  - Includes memory used for data storage
- Compilers create assembler code that is translated into machine code
  - A programmer can create more efficient programs with the assembler than with compilers
    - For example, compiler typically stores intermediate results in memory
    - In an assembler program, can keep intermediate results in a register

The Assembler

- Assembler converts a source file into machine code
- Source file
  - Contains mnemonics with operands that corresponds to machine instructions
- Machine code
  - Actual binary code that are interpreted and executed by the CPU

Assembly Source Instruction Fields

- Each line of assembler source contains four fields
  - Label: supply symbolic memory references and constant definitions
  - Operation Code (op_code) Field: contains either an opcode mnemonic or an assembler directive
  - Operand Field: contains specification of the instruction operands
  - Comment Field: For commenting

Label	Op_code	Operand	Comment
EXAMPLE	mov	a,b	; Restore A Reg

Operation Code Field

- Mnemonics
  - Representation for a machine instruction
  - Example: JMP (Jump instruction)
  - Assembler reads lines of code and translates mnemonics (and operand) into a machine language binary code
  - In example on the following slide, operand TARGET represents an address in memory
- Assembler Directive
  - Instructions for the Assembler
  - Example: ORG (origination)
  - ORG sets an internal location counter to the address given in the operand ($1000 in the example)
  - At the translation of each line of source, the internal location counter is updated
  - Eventually the line with the TARGET label is reached at which point a value for TARGET is assigned according to the internal location counter

Operation Code Field

Label	Op_code	Operand
	ORG	$1000
	op_code	operand
	op_code	operand
	JMP	TARGET
	op_code	operand
	op_code	operand
	….	
TARGET	op_code	operand
	op_code	operand

Operand Field

- Used to determine the operands of an instruction. Can be:
  - Name of a register
  - Numeric or symbolic constants
  - Labels (as seen in previous slides)
  - Algebraic expressions evaluated by the assembler
  - Examples:
    - **`MOVB A,DATA1`** ; A register, symbolic name DATA1
    - **`MOVB A,DATA1+1`** ; expression DATA1+1

Macro Assembler

- Assembler that can collect frequently used instructions into a single statement
  - Macro definition: In the source code, two assembler directives used to indicate start and end of macro, for example DEF_MACRO and END_MACRO
  - Macro invocation: In the source code, macro label used in the operation code field
  - Macro expansion: Assembler expands the macro name into the full code specified in the macro definition

Macro definition

```
Add_B_To_C   DEF_MACRO
             MOV A,C
             ADD A,B
             MOV C,A
             END_MACRO
```

Macro Invocation

op_code	operand
op_code	operand
Add_B_To_C	
op_code	operand
op_code	operand

Macro Expansion

```
op_code         operand
op_code         operand
Add_B_To_C
      MOV A,C   ; Assembler inserts
      ADD A,B   ; these three
      MOV C,A   ; lines
op_code         operand
op_code         operand
```

Macros Versus Subroutines

- Similar since both allow the reuse of code segments
  - But with subroutine, code is included only once
  - Macros are expanded “in-line” at each macro invocation – makes the program longer
- Subroutine requires a call or jump-to-subroutine (macro does not)
  - Thus subroutines are a little slower
- Both macros and subroutines make program easier to read
  - Changes only need to be made in one place
  - Hides details of the segment code – only need to know what it does, not how it does it

Two Pass Assemblers

- Most assemblers allow "*forward referencing*"
  - Use symbols before they are defined, remember the use of TARGET
- Assembler makes two passes during processing of the code
  - First pass is used to create a *symbol table*
  - Symbol table provides the values for symbols
  - Second pass generates the machine code using the values found in the symbol table

Cross Assemblers and Native Assemblers

- Cross Assembler
  - Assembler running on a computer to compile for a different processor
  - For example, using an assembler on a PC running an Intel processor to compile programs for the MC68HC12
- Native Assembler
  - Assembler creates code for the local processor
  - Part of system compilers

The Code Location Problem

- Remember the memory map for the MC9S12DG256
  - RAM
  - EEPROM
- Various parts are located in different sections of memory
  - Data that requires modification should be stored in RAM
  - Code (computer program) can be stored in ROM
- How do we locate code and data when creating an assembly program?

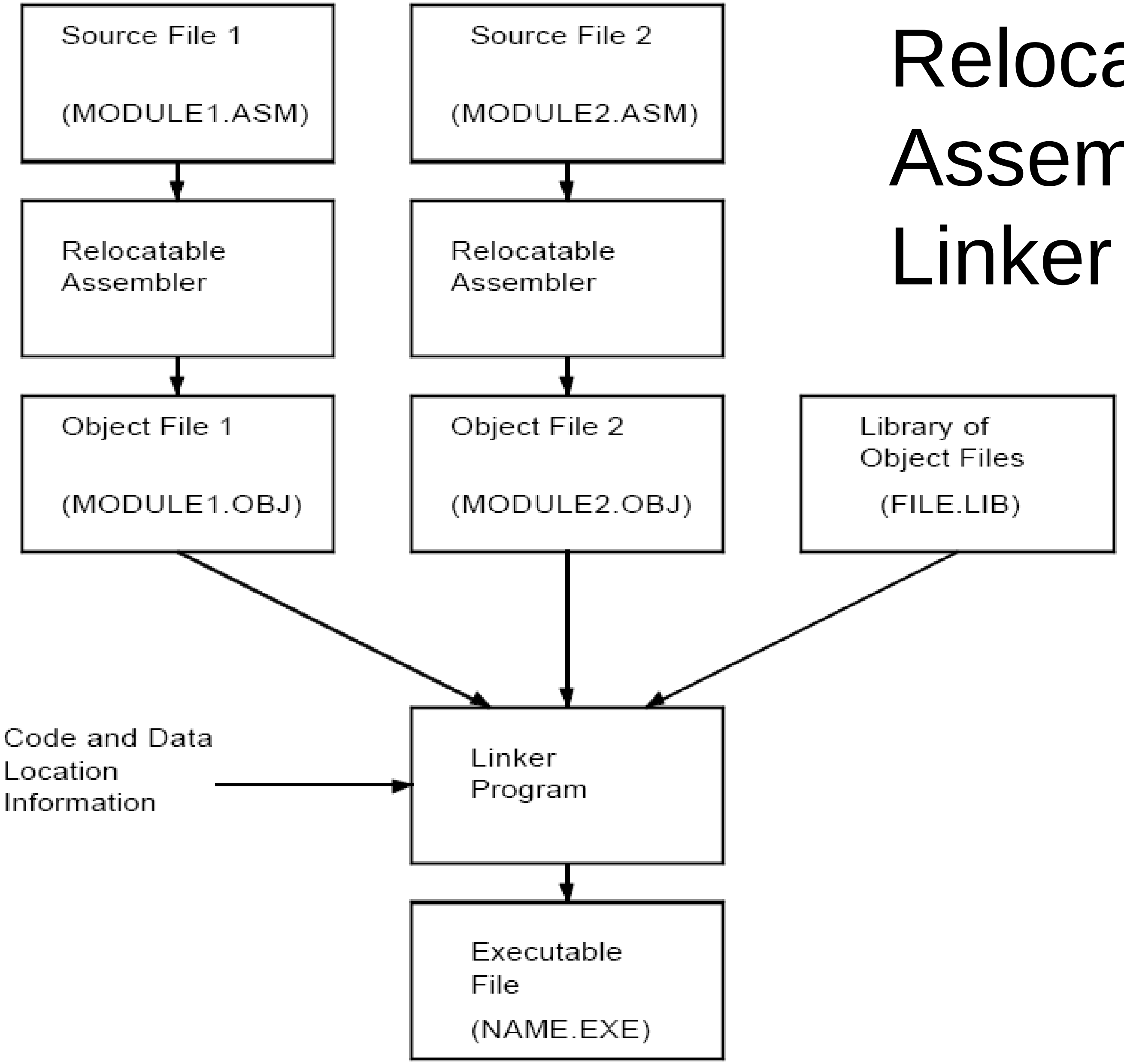
Absolute Assemblers

- Recall Slide 7
  - Compiled by absolute assembler
  - Knows where the program is to reside in memory
- Requires all source code for creating the executable program
- The ORG directive
- Use absolute assembler in this course (MiniIDE)
- Assemblers allow definition of program “sections” to facilitate modular design
  - Thus pieces of sections (data, code, etc.) can be found in separate modules (assembler source files).

Relocatable Assemblers

- Relocatable assembler: Creates *object files* that does not resolve addresses
- Linker: Combines object files to create single executable program (resolves addresses at linking time when creating binary)
- Addresses can also be resolved when the program is loaded into memory or executed
  - Operating systems use this feature to be able to load the program anywhere in memory)

Relocatable Assembler and Linker



Topics of discussion

- Assembly Language Programming
  - The Assembler
  - Code Relocation Problem
  - The Relocatable Assembler and Linker
- MC68HC12 Assembly Language Programming
  - The MiniIDE Assembler
- D-Bug12 Monitor

M68HC12 MiniIDE Assembler

- Public domain assembler
  - Available at http://www.mgtek.com/miniide/
  - Windows application
- Generates two files
  - S19 files – machine code in ASCII format suitable for download to the EVB board
  - LST file – provides a file listing of the source code with the machine code

Example – Hello1.asm

- Consider the source code in file Hello1.asm
  - Recognize the fields: label, Opcode, Operand, and comment
  - Note lines that contain only comments
- Consider the Hello1.lst file with the following fields
  - Source line number
  - Memory address
  - Machine instruction or data bytes
  - Source line
- Consider the contents of the Hello1.S19 file
  - Can you spot the machine code?
  - File can be used to download the code into the Dragon-12
- The logic of the assembler file can be represented in a higher level language like C – see Hello1.c

Example 2-1 Hello World! Program

```
Metrowerks HC12-Assembler
(c) COPYRIGHT METROWERKS 1987-2003
Abs. Loc     Obj. code Source line
 ---  -----  ------   --------
   1          ; Example program to print
   2          ; "Hello World"
   3          ; Constant equates
   4  0000  000D CR:         EQU  $0d    ; Carriage return
   5  0000  000A LF:         EQU  $0a    ; Line feed
   6  0000  0000 EOS:        EQU  0      ; End of string
   7               ; Memory map equates
   8  0000 8000  PROG:       EQU  $8000  ; Flash memory
   9  0000 0A00  STACK:      EQU  $0a00  ; Stack pointer
  10                         ORG  PROG   ; Locate program
  11               Entry:
  12               ; Initialize stack pointer
  13 008000 CF0A 00          lds  #STACK
  14               loop:
  15               ; Print Hello World! string
  16 008003 CC80 0B          ldd  #HELLO
  17 008006 1680 18          jsr  printf
  18               ; Do it forever
  19 008009 20F8             bra  loop
  20               ; Define the string to print
  21 00800B 4865 6C6C HELLO:  DC.B 'Hello World!',EOS
     00800F 6F20 576F
     008013 726C 6421
     008017 00
```

Hello1.c

```
/* Hello World Program in C */
#define HELLO "Hello world!"
void main()
{
   putstr(HELLO);
}
/*------------------------------
Function: putstr
Parameters: char *str – string to display
Description: displays the string by calling putchar()
--------------------------------*/
void putstr(char *str)
{
   while(*str != '\0')
      putchar(*str++);
}
```

MiniIDE Label Field

- A label is a symbol composed of alphanumeric characters, underscores (_), periods (.), dollar signs ($), or question marks (?).
- The first character of the label must not be a numeric character.
- The length of a label is limited to 128 characters; longer labels are truncated.
- An optional colon (:) may be added to the end of the label.
- Assembler is case insensitive.
- Lines with no labels start with white space character (space or tab)
- Cannot use reserved words as labels.

Examples of labels (symbols)

TEST:	; Legal label
TEST$:	; Legal
TEST$DATA:	; Legal. Sometimes the $ is used as a
	; separator to make the label more
	; readable
TestData:	; Legal, more readable
Test_Data:	; Legal, more readable yet
Label	; Legal. A label does not need a colon

MiniIDE Opcode and Operands

- OpCode field
  - M68HC12 mnemonic, assembler directive or macro name
- Operands
  - Symbol, constant, or expression to be converted to instruction operand
  - Symbols: Used to represent 8 or 16 bit integer values (see printf in the Hello World example)
  - Constants: Numerical values in one of the following formats: decimal, hexadecimal, octal, binary, and ASCII.
    - Prefix or suffix indicates the format (see next slide)
  - Expressions
    - Combination of symbols, constants and operators

Base Designators for Constants

Base	Prefix	Suffix
Binary	%	B
Decimal		D
Octal	@	O,Q
Hexadecimal	$	H
Ascii	‛	

Examples of constants

```
;Decimal Constants
0000 8664      1    ldaa   #100    ; 100 -> A
0002 8664      2    ldaa   #100D   ; 100 -> A
0004 CE04D2    3    ldx    #1234   ; 1234 -> X

;Hexadecimal Constants
0000 8664      1    ldaa   #$64    ; $64 = 100 -> A
0002 869C      2    ldaa   #$9C    ; $9c = -100 -> A
0004 CE1234    3    ldx    #$1234  ; $1234 -> X

;Binary Constants
0000 8664      1    ldaa   #%01100100          ; 100 -> A
0002 8664      2    ldaa   #01100100b          ; 100 -> A
0004 8664      3    ldaa   #01100100B          ; 100 -> A
0006 869C      4    ldaa   #%10011100          ; -100 -> A
0008 CE091A    5    ldx    #%0001001000110100  ; $1234 -> X
000B 86F0      6    ldaa   #%11110000          ; MS nibble mask
```

Expression operators

- Arithmetic:
  - * (multiplication) / (division) + (addition) – (subtraction) % (modulus)
- Bit operators
  - ~ (one's complement), << (shift left), >> (shift right), & (bitwise AND), ^ (bitwise XOR), | (bitwise OR)
- See MiniIDE documentation for a complete list that includes logical and comparison operators

Examples of Expressions

```
                1  ;  Test of all expression operators
0000            2  ONE:       EQU     1
0000            3  TWO:       EQU     2
0000            4  SMALL:     EQU     $FF
                5  ;
0000 03         6  ADD:       DC.B    {ONE+TWO}     ; Addition
0001 01         7  SUB:       DC.B    {TWO-ONE}     ; Subtraction
0002 02         8  ASL:       DC.B    {ONE<<1}      ; shift left
0003 7F         9  LSR:       DC.B    {SMALL>>1}    ; shift right
0004 3F         10            DC.B    {SMALL>>2}    ; shift right
0005 00         11  AND:      DC.B    {ONE&TWO}     ; Bitwise AND
0006 03         12  OR:       DC.B    {ONE|TWO}     ; Bitwise OR
0007 03         13  XOR:      DC.B    {ONE^TWO}     ; Bitwise XOR
0008 04         14  MULT:     DC.B    {TWO*TWO}     ; Multiplication
0009 01         15  DIV:      DC.B    {TWO/TWO}     ; Division
```

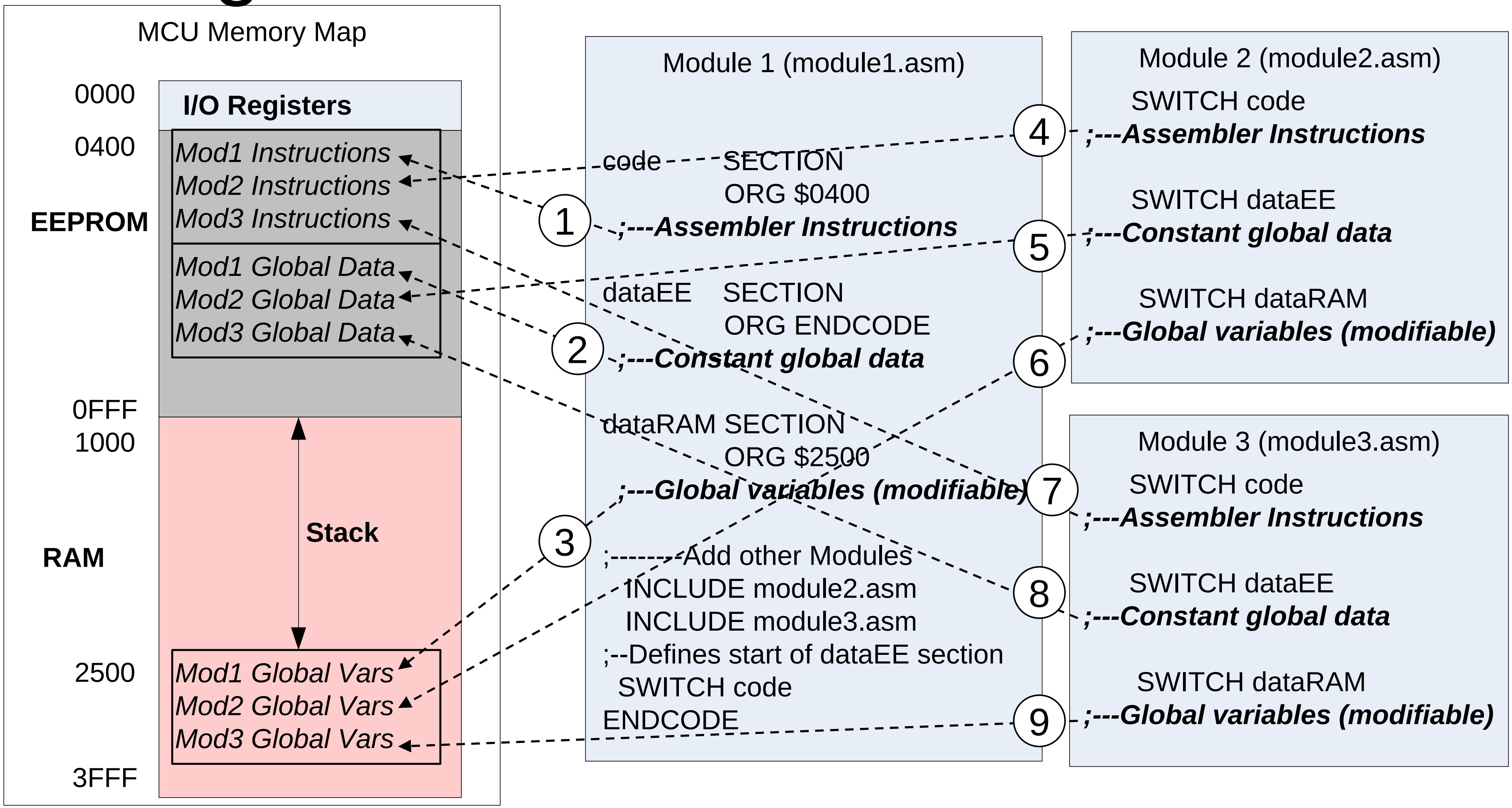
Comment Field

- Started by a ; character and goes to the end of the line
- A complete line can be a comment
  - Line starts with ; or * character
- Blank lines can be present in the source code

Assembler Directives

- ORG – sets the location counter
- EQU – equates a symbol to an expression
  - Can also use SET. With SET, can redefine symbols
- Data definition directives
  - DC – define constant data (.B, .W, .L)
  - DCB – define constant block (.B, .W, .L)
  - DS – define storage (.B, .W, .L) (i.e. variables)
  - OFFSET – define offsets for data structures
- Macro Control Directives
  - MACRO – start macro
  - ENDM – end macro
  - EXITM – exit macro
- Directives to support modules:
  - Define sections: SECTION, SWITCH
  - Include files (i.e. modules): INCLUDE
- Conditional assembly directives (see MiniIDE documentation)
- List control directives (see MiniIDE documentation)

Program Sections



Note: Absolute addresses are defined only on the second pass

- ENDCODE value can change when code changes

OFFSET

- OFFSET <value>
  - Assembler moves to the OFFSET section and sets its location counter to value.
  - Exits the section at first encounter of instruction that generates code
  - Can define offsets into data structures
- Example structure C

```
struct varStr
{
   int n1;
   int n2;
   long n3;
} var;
…
va = var.n1;
```

... in assembly:

```
                OFFSET 0
n1              DS.W   1
n2              DS.W   1
n3              DS.L   1
VAR_STR_LEN EQU * ; the size of the structure
                …
               ldx #var
               ldaa n1,x
               …
var            ds.b VAR_STR_LEN
```

Defining Stack Usage

- Use OFFSET to define offset labels into the stack
- Example:

```
             OFFSET 0
RTN_VAR1     DS.B 1   ; single byte var.
RTN_VAR2     DS.W 1   ; two byte var.
RTN_ARR      DS.B  10 ; 10 byte array
RTN_PR_B     DS.B 1   ; preserve B
RTN_PR_X     DS.W 1   ; preserve X
RTN_RA       DS.W 1   ; Return Address
RTN_ARG1     DS.W 1   ; 2 byte argument
RTN_ARG2     DS.B 1   ; 1 byte argument

routine      pshx ; subroutine label with code instr.
             pshb
             ldd RTN_ARG1,SP    ; load ARG1
              …
             stab RTN_VAR1,SP   ; assign value to VAR1
```

The Software Development Process

- Problem description: What must be done
- Design: structured design includes modules and algorithms
- Programming: coding the design and algorithms
- Program Testing
- Program maintenance
- Documentation

Software Design

- Development of modules
  - Modules divide the problem into manageable tasks
  - Each module is again sub-divided into subtasks for which algorithms are defined
  - What are the advantages of modules?
- Development of algorithms
  - Algorithms are developed for each subroutine
  - Options
    - Flowcharts
    - Pseudo-code
  - For the first half of the course – shall use C as the pseudo-code
    - Translate MANUALLY C program into Assembler

Simple example

- Problem:
  - Make the following calculation:
    - z = a + b – c
- Algorithm
  - In C:

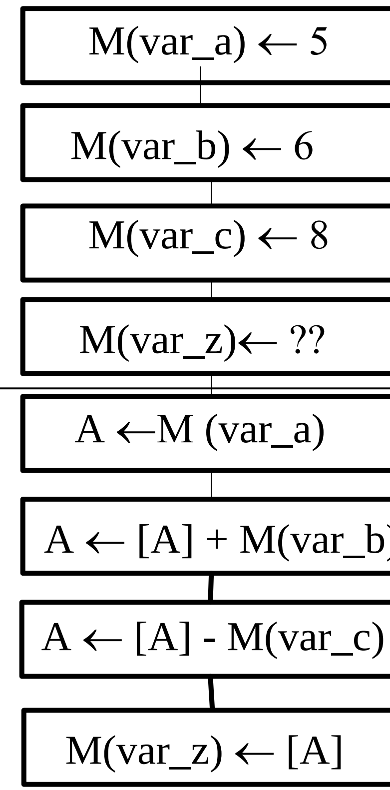
```
int z, a=5, b=6, c=8;
z = a + b - c;
```

Simple Example (continued)

- The code needs to assemble and test

.lst

```
 1:; Define the addresses to the variables
 3: =00002500                 ORG $2500
 4: 2500 0005   var_a DC.W 5     ; a
 5: 2502 0006   var_b DC.W 6     ; b
 6: 2504 0008   var_c DC.W 8     ; c
 7: 2506 +0002  var_z DS.W 1     ; z
 8:; Code z = a + b - c
 9:  =00001000                 org  $1000
10:       1000 B6 2500 ldaa  var_a    ; Load a
11:       1003 BB 2502 adda  var_b    ; add  b
12:       1006 B0 2504 suba  var_c    ; subtract c
13:       1009 7A 2506 staa  var_z    ; save z
```



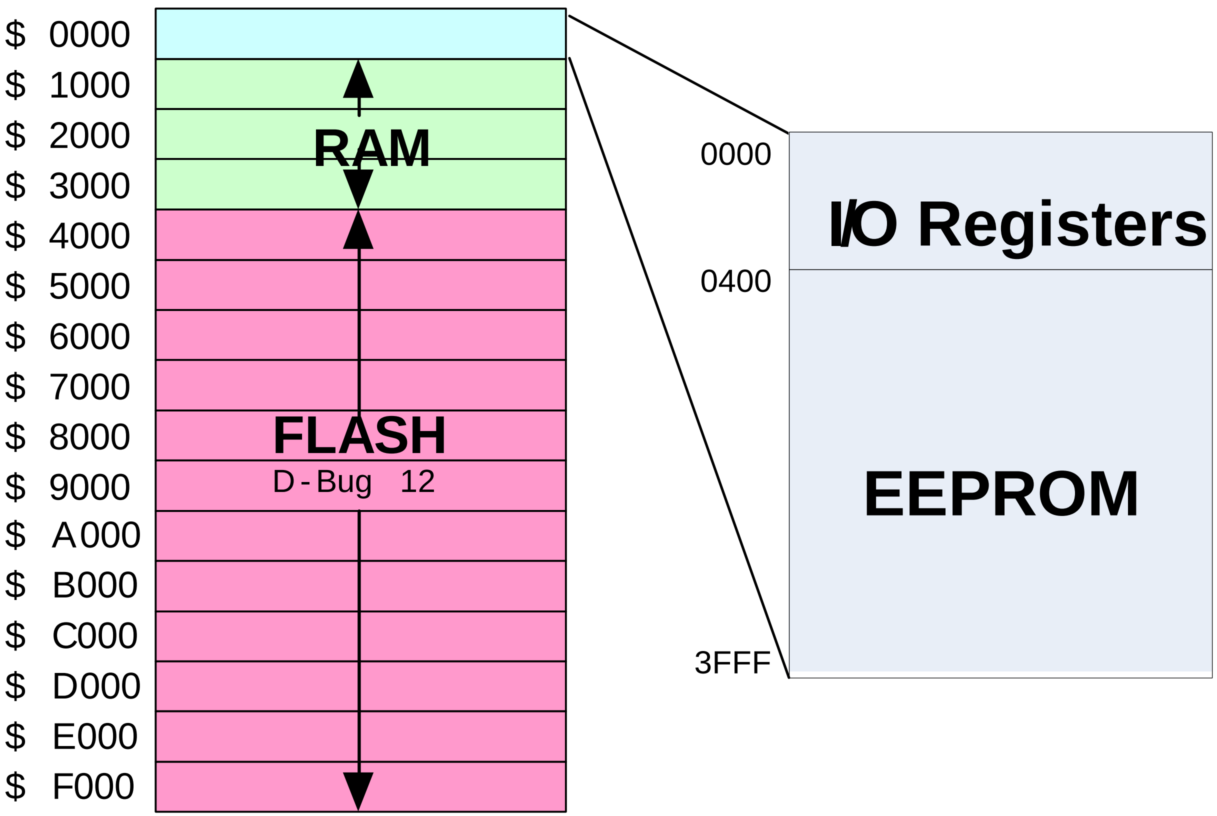
Topics of discussion

- Assembly Language Programming
  - The Assembler
  - Code Relocation Problem
  - The Relocatable Assembler and Linker
- MC68HC12 Assembly Language Programming
  - The MiniIDE Assembler
- D-Bug12 Monitor

D-Bug12

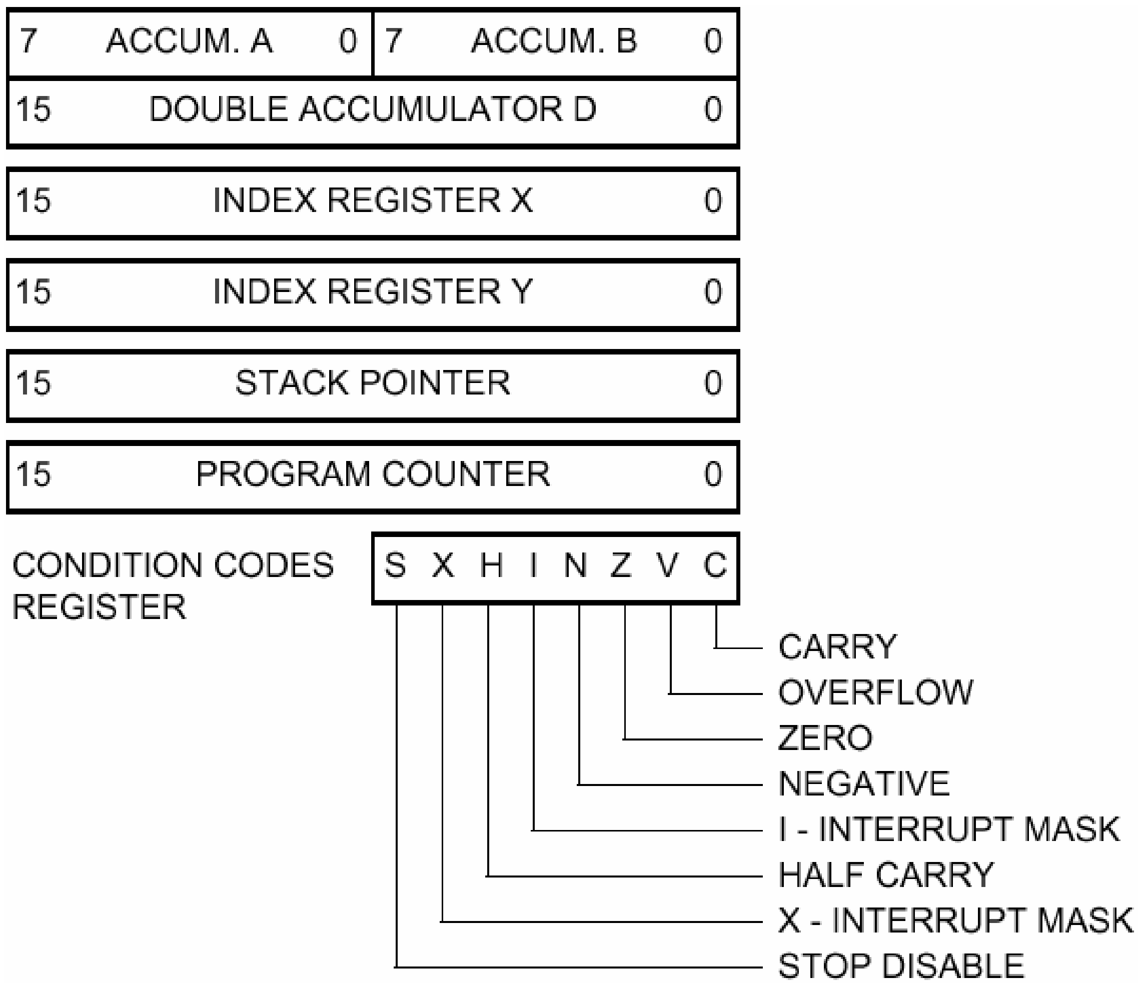
- Will use D-Bug12 Monitor
  - Version 4.xx on the Dragon-12 board (using the MC9S12DG256)
  - Consists of a program in ROM used for loading and running assembler programs
  - Supports debugging
  - Command line program

Recall the MC9S12DG256 Memory Map



- On-chip registers to I/O interfaces ($0000 to $03FF)
- 3 KBytes EEPROM ($0400 to $0FFF)
- 12 KByte RAM ($1000 to $30FF)
  - $3C00 to $3FFF is used by the D-Bug12 Monitor
- 48 KByte EEPROM Memory ($4000 to $FFFF)

Recall the CPU12 Programmer's Model



D-Bug12 Monitor Commands

- **ASM**: Assembler/Disassembler
- **BF**: Block fill memory
- **BR**: Set Breakpoint
- **CALL**: Call and execute subroutine
- **G**: Go (run) to a user program
- **GT**: Go till an address
- **HELP**: Prints command summary
- **LOAD**: Download program
- **MD**: Memory display
- **MM**: Memory modify.
- **MOVE**: Copy block of memory
- **NOBR**: Remove breakpoints
- **RD**: Display register contents
- **RM**: Register modify
- **T**: Trace program.
- **UPLOAD**: Display a block of memory in S-record format
- **VERIF**: Compare memory to downloaded file.

ASM Command

- ASM <address>
  - Assemble/disassemble
  - Parameter “address” is the starting address
  - Type in assembler instruction to enter code
  - Type <Enter> to accept/view current code
  - Default type for operands are decimal
  - Use $ prefix to specify hexadecimal values

Monitor Utility Routines

- Provides a set of subroutines available for use by user programs (use JSR)
  - D-Bug12 provides a vector table to the subroutines
  - Use JSR with indirect indexed addressing, (or PC relative addressing)
- Were developed in C and uses C parameter passing
  - First function parameter passed in the D register
  - Other parameters are pushed on the stack in right to left order (note the calling program must clean up the stack after the subroutine is called).
  - Function returns a value in the D register
- Consider the call to the following function:

  int function_call(int param1, char param2, int parm3)

Example of D-Bug12 Utility

```
=0000FE0A   VECTOR  EQU     $FE0A   ; Address of the routine
0000 DC 13              ldd     param3  ; Retrieve param 3 from memory
0002 3B                 pshd            ; Put it on the stack
0003 D6 12              ldab    param2  ; Retrieve param 2 from memory
0005 B7 14              sex     b,d     ; Convert to 16-bits
0007 3B                 pshd            ; Put it on the stack
0008 DC 10              ldd     param1  ; Retrieve param 1 from memory
000A 15 FB FDFC         jsr     [VECTOR,PCR] ; JSR to the subroutine
000E 1B 84              leas    4,sp    ; Clean up the stack
                        ; At this point the D register contains the int variable
                        ; returned by the function.
                        ;           - - -
  0010 +0002            param1: DS   2      ; Storage for parameter 1
  0012 +0001            param2: DS   1      ; Storage for parameter 2
  0013 +0002            param3: DS   2      ; Storage for parameter 3
```

D-Bug12 Character Routines

Function	Vector	Description
int getchar(void)	$EE84	B contains character fetched from the serial I/O port. Waits until char is available.
int putchar(char c)	$EE86	Sends the single char in B to the serial port (SCI). Returns the character sent.
int isalpha(int c)	$EE96	Returns non-zero value in D, if the char c (in D) is an alphabetic char, and zero in D otherwise.
int isxdigit(int c)	$EE92	Returns non-zero value in D, if the char c (in D) is a hexadecimal digit, and zero in D otherwise.
int toupper(int c)	$EE94	Converts char in B to upper case.
void out2hex(unsigned int num)	$EE9C	Displays the byte in the B register as two hexadecimal characters.
void out4hex(unsigned int num)	$EEA0	Displays the 2 bytes in the D register as four hexadecimal characters.

D-Bug12 String Routines

Function	Vector	Returns
int printf(char *format, arg1,…)	$EE88	Implements the C printf function that prints a formatted string (does not support floating type). Returns number of char's printed in D.
int getCmdLine(char *strpt, int len)	$EE8A	Always returns D=0. Obtains a line from the user (with editing), and stores at address found in D.
char *sscanhex(char *hexStr, unsigned int *binN)	$EE8E	Converts the ASCII Hexadecimal string at address "hexStr" to a binary integer (stored at address "binN"). D returns NULL if error occurred, otherwise points to the terminating character of the hex string.
char *strcpy(char *s1, char *s2)	$EE9A	Copies the null-terminated string addressed by s1 to memory address s2. D returns pointer to S2
int strlen(char *s)	$EE98	Returns in D the number of characters in the string addressed by s.

D-Bug12 Miscellaneous Routines

Function	Vector	Description
void main(void)	$EE80	Starts the D-Bug12 monitor. Allows some initialization before starting the monitor.
setUserVector(int vectN, address UsrAdr)	$EEA4	Sets the interrupt vector address “UsrAdr” for the interrupt associated to vectN. Returns D=-1 if vector number is invalid, and 0 otherwise.
int eraseEE(void)	$EEAA	Bulk erase EEPROM. If not erased, D=0 and Z bit is set, otherwise, D non-zero and Z=0.
int WriteEEByte(address EEAdr, byte EEData)	$EEA6	Program a byte in to on-chip EEPROM.

References

- Reference Guide for D-Bug12 Version 4.x.x, Gordon Dougham
- Fredrick M. Cady, James M. Sibigtroth; "Software and Hardware Engineering: Assembly and C Programming for the FreeScale HCS12 Microcontroller"
- Above are sources for most of the figures, tables and examples in the course notes

Some Instructions

Load Instructions:

LDAA #*ii* (86 *ii*) Put *ii* into **A** (N,Z); **LDAB** #*ii* (C6 *ii*) Put *ii* into **B**

LDX #*ea1 ea0* (CE *ea1 ea0*) Load Index Register X with address *ea1 ea0*
CCR affected: (N,Z)

Arithmetic Instructions:

ADDA (AB *ea*) ; A := (A) + m(*ea*) ; CCR affected: (H,N,ZV,C)

Decrement and Increment Instructions

DECB (53) Subtract one from the content of accumulator B. (N,Z,V)

INX (08) Add one to index register X. (Z)

Short Branch Instructions

BNE rel (26 rel) Tests the Z status bit and branches if Z = 0 to PC:=(PC)+2+rel

NOP

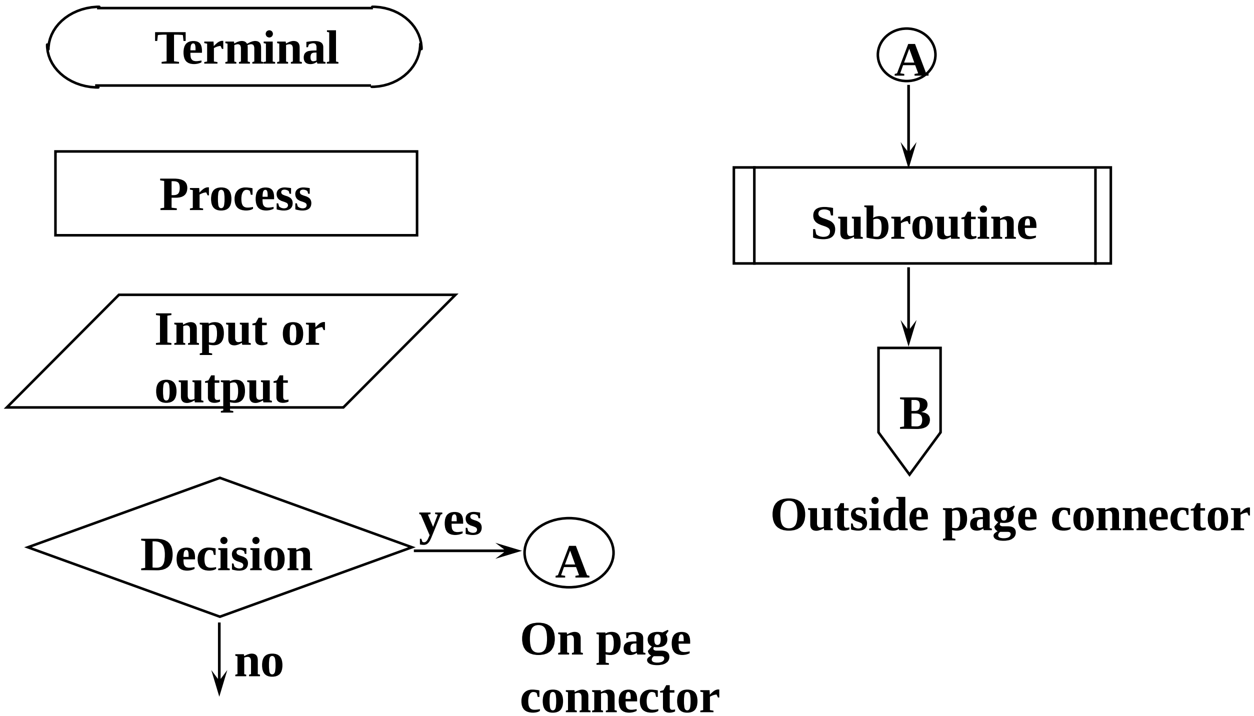
Stack Operation Instructions

PSHA The stack pointer is decremented by one.
The content of A is then stored at the address the SP points to.

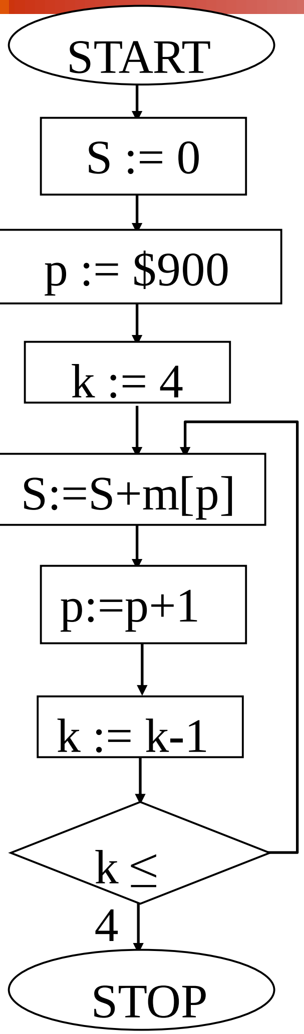
PULA Accumulator A is loaded from the address indicated by the SP
The SP is then incremented by one.

SWI Software Interrupt (Used to end all our HC12 programs)

Flowchart Symbols



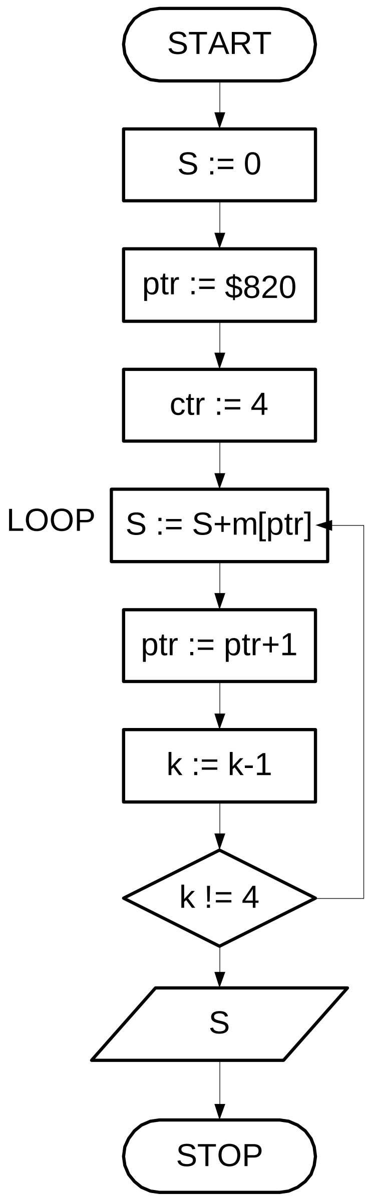
A Simple Program



A 4-element (1-dimensional) array is stored in memory starting from address $900. You have the D-Bug12 only … and you have to write a program to adding all array's elements and put their sum into A.

Instruction		Operation	Comments	Address	Memory Content
LDAA	#0	A ← 0	S:=0	$800	86
					0
LDX	#$900	X ← $900	Pointer to the first element	$802	CE
					9
					00
LDAB	#$04	B ← 4	Init. Counter (k := 4)	$805	C6
					04
ADDA	0,X	A ← (A) + m[(X)]	S:=S+m[p]	$807	AB
					00
INX		X ← (X)+1	Point to next element	$809	08
DECB		B ← (B) - 1	Decrement counter	$80A	53
BNE	$807	PC ← (PC) - 5	Redo loop if more to be added	$80B	26
					FA
SWI		SW Interrupt	Back to D-Bug12	$80D	3F

Variable	Register	
S	A	sum
p	X	ptr
k	B	ctr




```
C:\Program Files\MiniIDE\sum_array.lst - MGTEK Assembler
                                       ;S       -> accumulator A
                                       ;ptr     -> register X
                                       ;ctr     -> accumulator B
 =00000004          N          EQU       4 ;# of array elements
 =0000F684          putchar    EQU $F684   ;start of print sub
 =00000800                     ORG   $800    ;our program start
 0800 87                       clra          ;S <- 0
 0801 CE 0820                  ldx #A_start;ptr <- A_start
 0804 C6 04                    ldab #N       ;ctr <- N
 0806 AB 00         LOOP       adda 0,x      ;S <- S+m(ptr)
 0808 08                       inx           ;ptr <- ptr+1
 0809 53                       decb          ;ctr <- ctr-1
 080A 26 FA                    bne LOOP      ;redo if N-ptr > 0
 080C FE F684                  ldx putchar ;Print sum putchar
 080F 15 00                    jsr 0,x
 0811 3F                       swi           ;STOP
 =00000820                     ORG $820
 0820 01 02 03 04  A_start    db  1,2,3,4
Symbols:
a_start                             *00000820
loop                                *00000806
n                                   *00000004
putchar                             *0000f684
```