uOttawa

L'Université canadienne
Canada's university

CEG3136
Computer Architecture II

Module 5 – Developing Software

Notes of

Dr. Voicu Groza

Université d'Ottawa | University of Ottawa

www.uOttawa.ca

Topics of discussion

- Software Development Process
- Design
- Coding
  - Parameter Passing
  - Structured Code
- Program Debugging

- Reading: Cady Chapters 3, 8 and 9, Alarm Simulation design Document

The Software Development Process

- Problem description: What must be done
- Design: structured design with modules and algorithms
  - An algorithm provides details but not for each machine instruction
  - Ex: load variable varA with contents pointed by pointer ptr

    varA ← [ptr] (Pseudo-code)   `varA = *ptr;` (C prog. lang.)
    - varA and ptr can be either registers or locations in memory
  - Shall be using C to design of assembler programs
- Programming: coding the design to a program
- Program Testing
  - Module testing
  - System testing
- Program maintenance
- Documentation

Requirements Specification

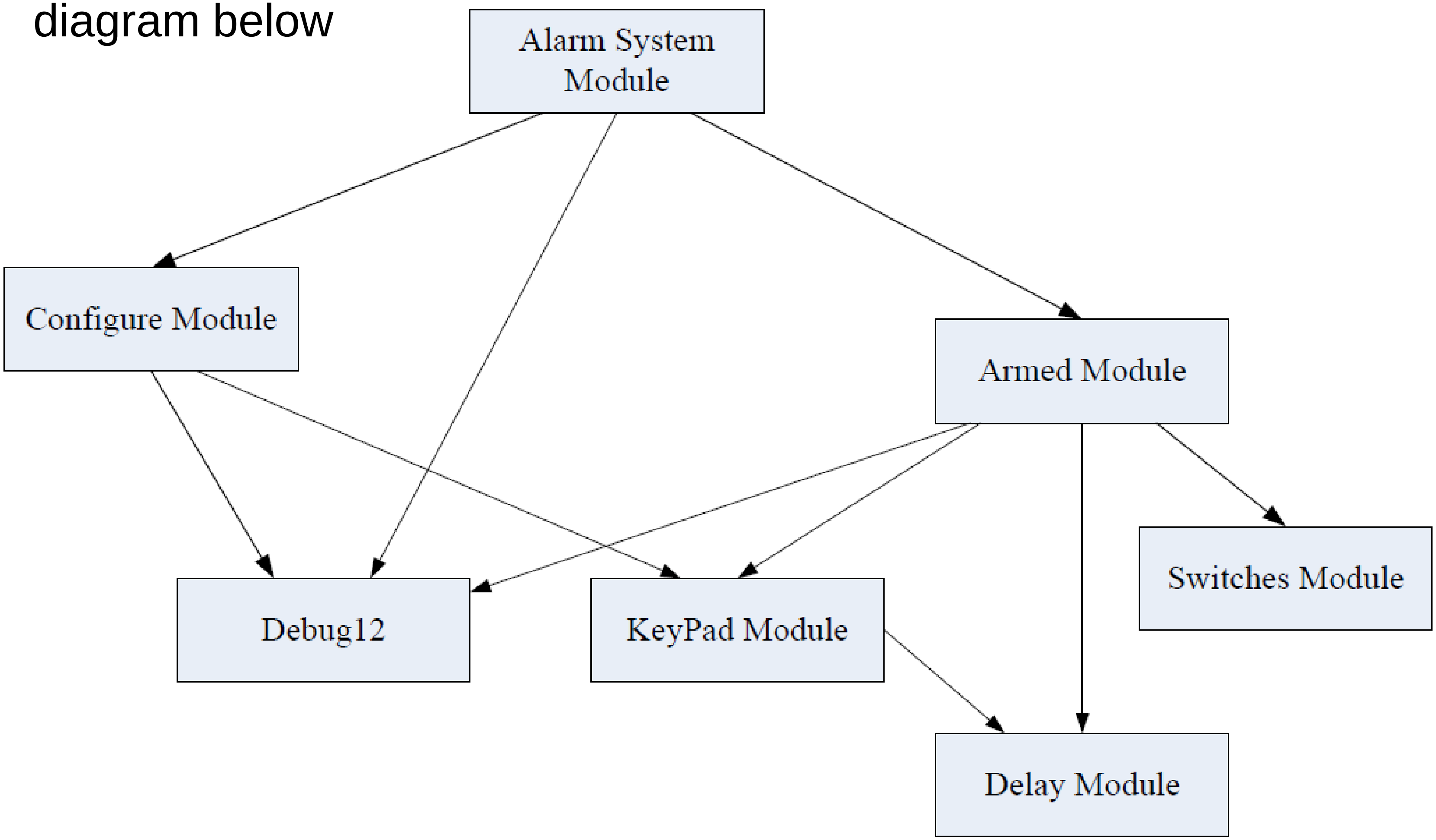
- Understand the problem completely
- Need to know what is to be done
- Communication with the customer
- Alarm System
  - Configuration of a master code, addition of secondary codes
  - Entering any of the codes will set the alarm
  - There should be a delay of 15 seconds before alarm is enabled
  - When the alarm has been triggered, it can be turned off by entering an appropriate code.
  - Disarming the alarm system: Typing the character ‘a’ represents opening the front door, and a 15 seconds are allowed to type in of the alarm code to disable the alarm system.

Top-Down Design

- Design in levels
- Now focus on how to solve the problem
- Break down the problem to be solved into smaller pieces
  - Define smaller tasks
  - Smaller tasks can be subdivided further
  - For large programming projects, tasks can be share among members of a team

Designing the Alarm System

The alarm simulation software is divided into five modules as shown in the diagram below



More on Top-Down Design

- Ensure correctness at each level
  - Will require multiple passes
- Postpone details
  - Work on details progressively going towards lower levels
- Successively refine the design
  - As lower levels are being developed, changes may be made to upper levels
- Design using algorithms
  - Want to focus on algorithms, that is how to solve the problem, without dealing with the details of programming
  - Use pseudo-code or flowcharts
  - In our case, shall use a high-level language (C) to represent algorithms for assembler programs

Bottom-up Design

- Coding before design is complete
  - Do not respect rule of postponing details
  - Make upper levels harder to design
  - Violate successive refinement
  - Cannot optimize low levels base on decisions of upper levels
- Not all bad
  - Use generic tools – code that has general functions

Real-World Approach

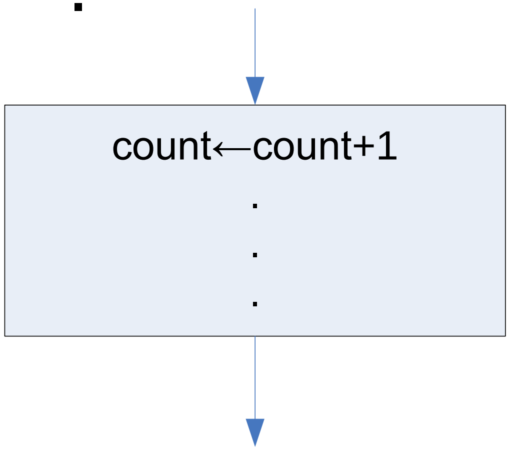
- Difficult to follow ideals of top-down design
  - Use low-level functions that are already available
  - Constraints imposed by using low level functions are offset by time saved in the development of using functioning code
- Real world approach
  - Complete as much design as possible before coding
  - Use previously developed and tested code where possible

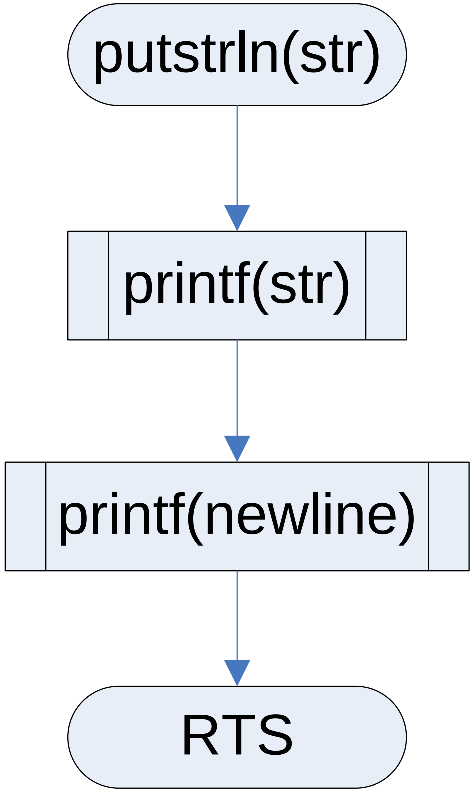
Design Tools

- Structured Programming
  - Use basic structures in a program: sequence, decision, and repetition
  - Do not use GOTO’s (this is a real danger with assembler programming)
  - Keep program segments small to keep them manageable
  - Organize the problem solution hierarchically
  - One input and one output

Pseudo-code: Sequence

```
Begin A
    count←count+1
            .
            .
            .
End A
```





Instruction Sequence

- C Program

```
int qu;  // quotient from divide
int rem; // remainder from divide
qu = num/10;
rem = num%10;
*addr = qu + ASCII_CONV_NUM;
*(addr+1) = rem + ASCII_CONV_NUM;
```

- Note that only variable names and literals are used in the expressions
- This keeps the “pseudo-code” independent of the CPU or MCU being used.

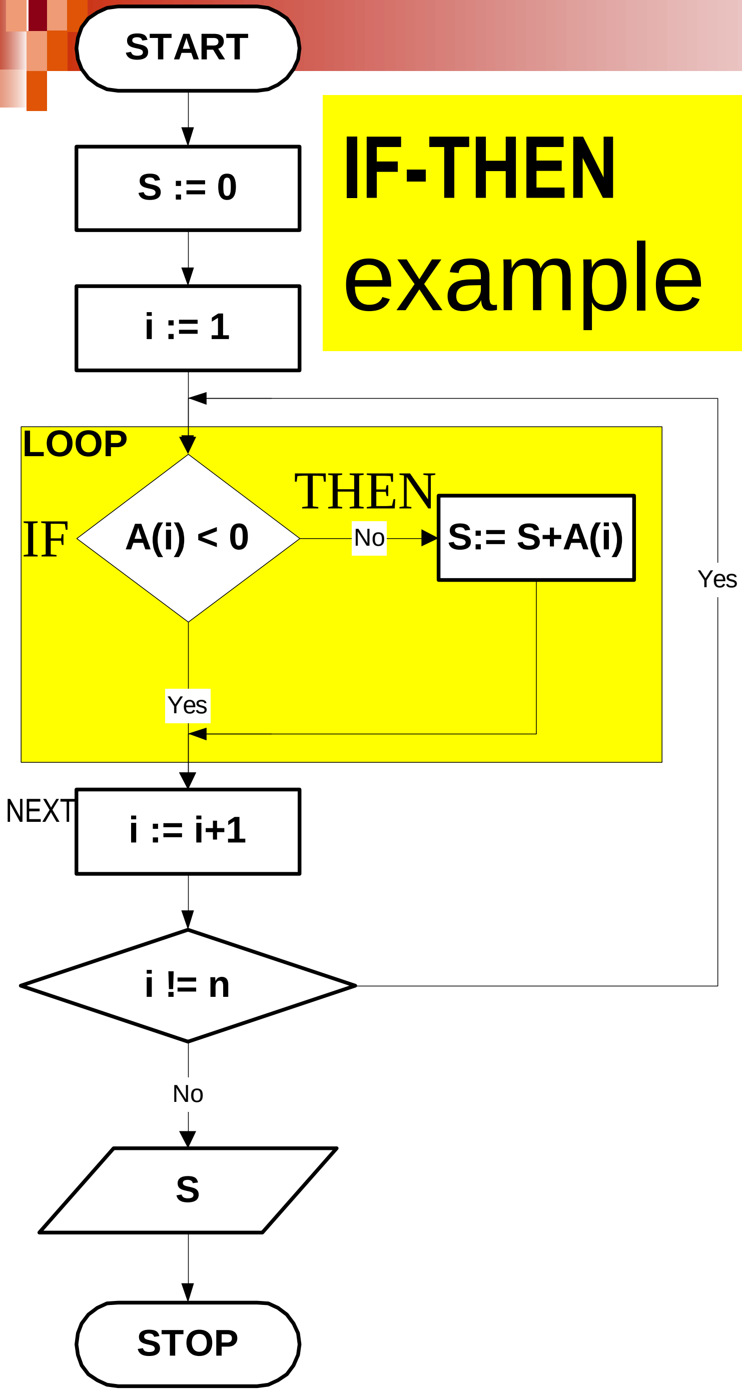
If-Then

```
int *ptr;  // pointer to alarmCodes
byte cnt = NUMCODES;
byte retval = FALSE;
ptr = alarmCodes;
if(*ptr++ == alarmCode)
{
    retval = TRUE;
    break;
}
```

If-Then-Else

```
 if(!dispA)
 {
   setCharDisplay('A',0);
   dispA = TRUE;
}
else
{
   setCharDisplay(' ',0);
   dispA = FALSE;
}
```

IF-THEN example



```
; add all #'s >0 of a N element array stored in
; memory beginning at the address A_start
;and store the result at A_result
;S -> accumulator A ;ptr-> register X ;
;ctr -> accumulator B
N        EQU     4 ;# of elements to be added
         ORG     $800 ; program start address
         clra              ;S <- 0
         ldx     #A_start  ;ptr <- A_start
         ldab    #N        ;ctr <- N
LOOP     tst     0,X     ;test m(ptr) for <0 (IF)
         bmi     NEXT ;(THEN) skip
         adda    0,X     ;(ELSE) S <- S+m(ptr)
NEXT     inx               ;ptr <- ptr+1
         dbne    B,LOOP    ;redo if N-ptr > 0
         staa    A_result
         swi               ;STOP
         ORG     $820
A_result ds 1
A_start  db      1,-2,3,-4
```

If-ElseIf

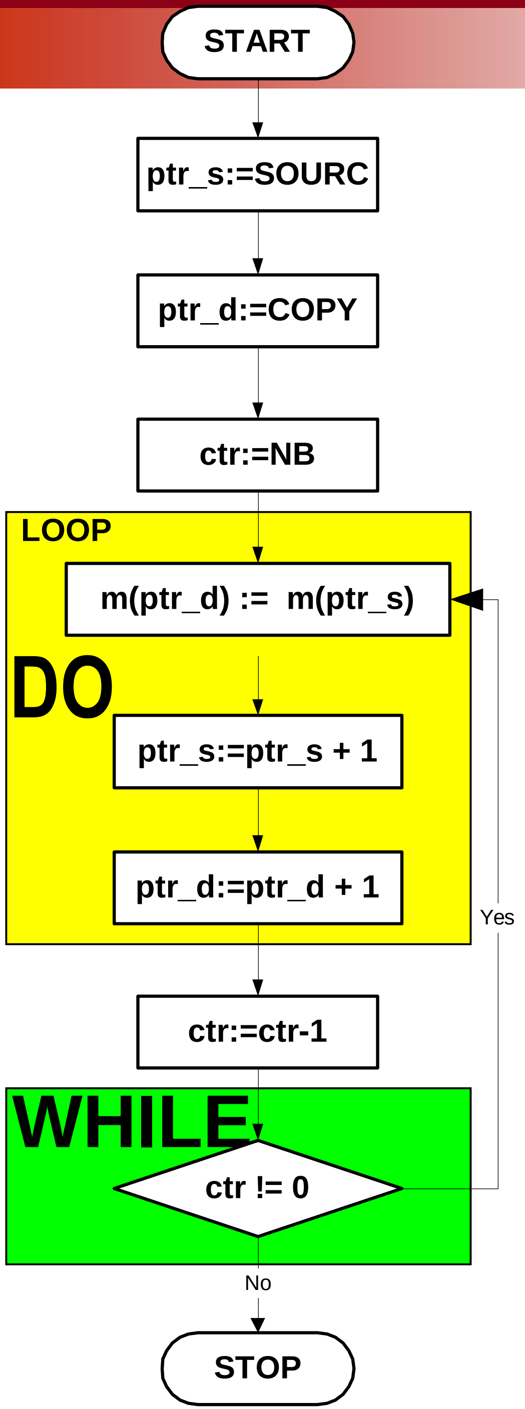
```
byte select;
select = readKey();
if(select == 'c')
     configCodes();
else if(select == 'a')
     enableAlarm();
else /* do nothing */;
```

While-Do

```
codeValid = FALSE;
while(!codeValid)
{
  input = readKey();
  codeValid = checkCode(input);
}
```

Do-While

```
int *ptr;  // pointer to alarmCodes
byte cnt = NUMCODES;
byte retval = FALSE;
ptr = alarmCodes;
do
{
   if(*ptr++ == alarmCode)
   {
      retval = TRUE;
      break;
   }
   cnt--;
} while(cnt != 0);
```



START

ptr_s:=SOURC

ptr_d:=COPY

ctr:=NB

LOOP

DO

m(ptr_d) := m(ptr_s)

ptr_s:=ptr_s + 1

ptr_d:=ptr_d + 1

ctr:=ctr-1

WHILE

ctr != 0

Yes

No

STOP

ptr_s	X
ptr_d	Y
ctr	B

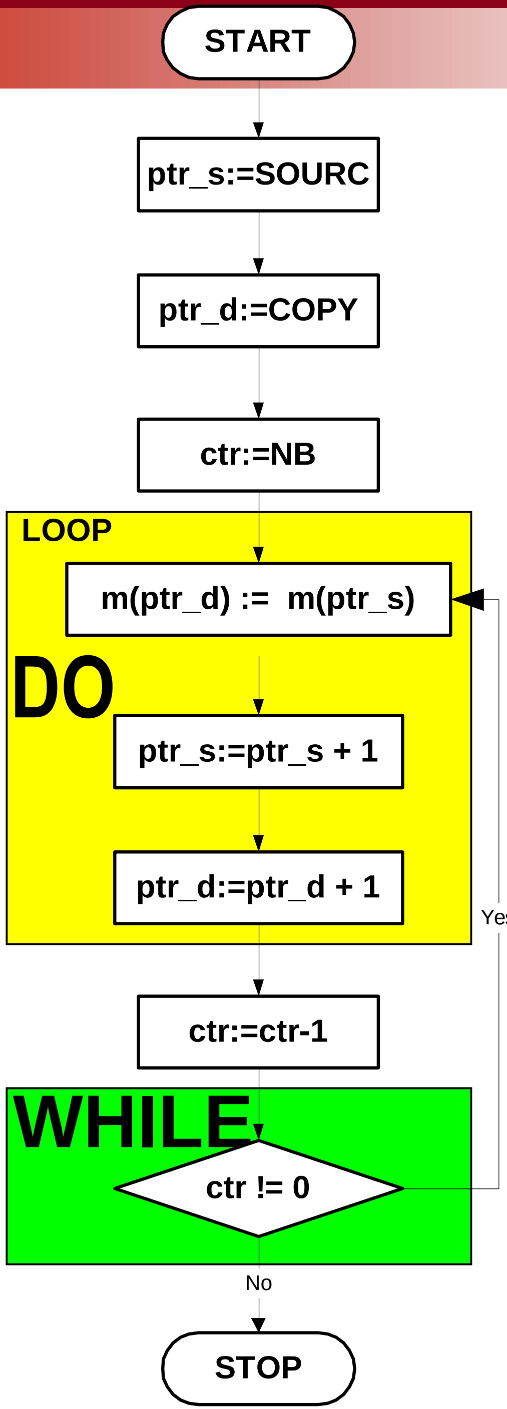
```
1:        =0000A000     SOURC   EQU $A000
2:        =0000A100     COPY          EQU $A100
3:        =0000000A     NBELEM  EQU 10
4:        =00000800             ORG     $800
5:    0800 CE A000              LDX     #SOURC
6:    0803 CD A100              LDY     #COPY
7:    0806 C6 0A                LDAB    #NB
8:    0808 180A 30 70   LOOP MOVB 1,X+,1,Y+
9:    080C 53                    DECB
10:    080D 26 F9                BNE     LOOP
11:    080F 3F                   SWI
```

```
Symbols:
copy                 *0000a100
loop                 *00000808
nb                 *0000000a
sourc                *0000a000
```

Do-While-Break

```
int *ptr;  // pointer to alarmCodes
byte cnt = NUMCODES;
byte retval = FALSE;
ptr = alarmCodes;
do
{
   if(*ptr++ == alarmCode)
   {
      retval = TRUE;
      break;
   }
   cnt--;
} while(cnt != 0);
```

- Attention – use minimally the `break`
- Use it to simplify the logic when the logical expression in the `while` is not sufficient



START

ptr_s:=SOURC

ptr_d:=COPY

ctr:=NB

LOOP

m(ptr_d) := m(ptr_s)

DO

ptr_s:=ptr_s + 1

ptr_d:=ptr_d + 1

ctr:=ctr-1

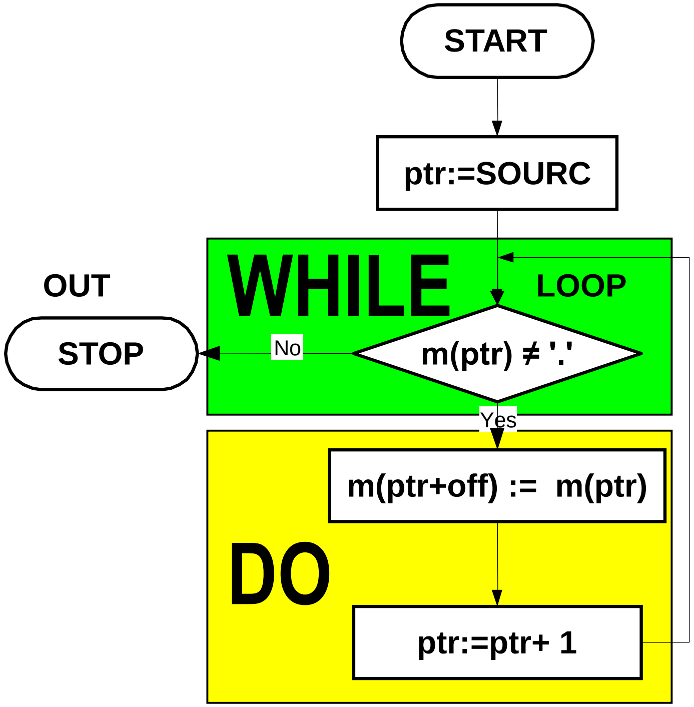
WHILE

ctr != 0

Yes

No

STOP



START

ptr:=SOURC

WHILE

LOOP

OUT

STOP

No

m(ptr) ≠ '.'

Yes

m(ptr+off) := m(ptr)

DO

ptr:=ptr+ 1

Subroutines – C functions

- Defining functions
  - □ Function declaration

```
<type> nameFunction( <parameter list> )
{
    // instructions
}
```

  - □ <*type*> can be a simple type or pointer type (i.e. address to a simple type, array, or structure)
  - □ <*parameter list*>: variable declaration to receive argument values
  - □ Ex: `int strlen(char *str) { … }`
  - □ In assembler, it is possible to return multiple values, it is C sets this limit
- Only one exit point: use only a single `return` instruction at the end of the function.
- Calling functions
  - □ Ex: `num = strlen(“A string”);`

Coding – Using Modules

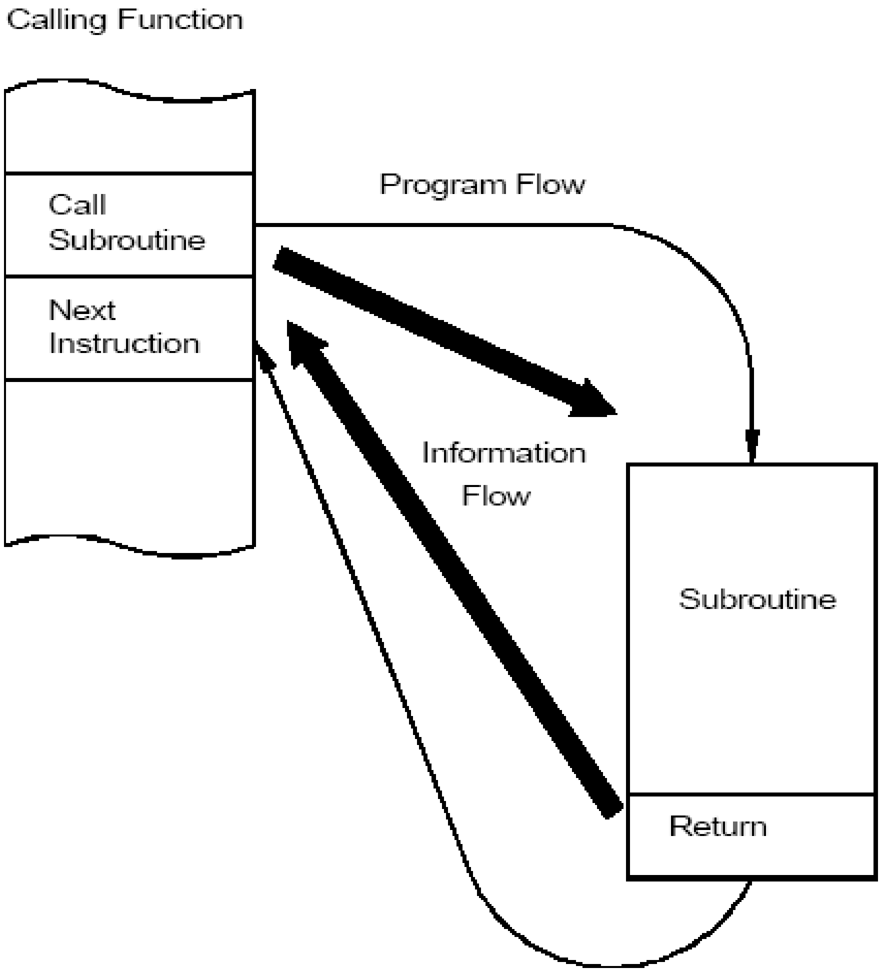
- Top-down design leads to programs that can be written into modules
  - Assign development of module to a team member (example – lab 2 – “Keyboard” Module)
  - Goal – try to create independent modules
  - Three attributes: function, coupling/linking to other modules, its logic.

Module Attributes

- Function:
  - Description of what the module does.
  - Some provide a collection of unrelated functions, for example a utilities Module
  - Some provide a number of related logical functions – for example a keypad Module
- Module coupling
  - Modules interact with each other
  - Challenge: keep modules as independent as possible
  - For example, use local, rather than global variables
- Module Logic
  - How the module does it’s job

Parameter Passing

- Data passed between subroutine and calling code
  - □ Data is passed to the subroutine
  - □ Data is returned from the subroutine
- A number of approaches can be used for passing parameters



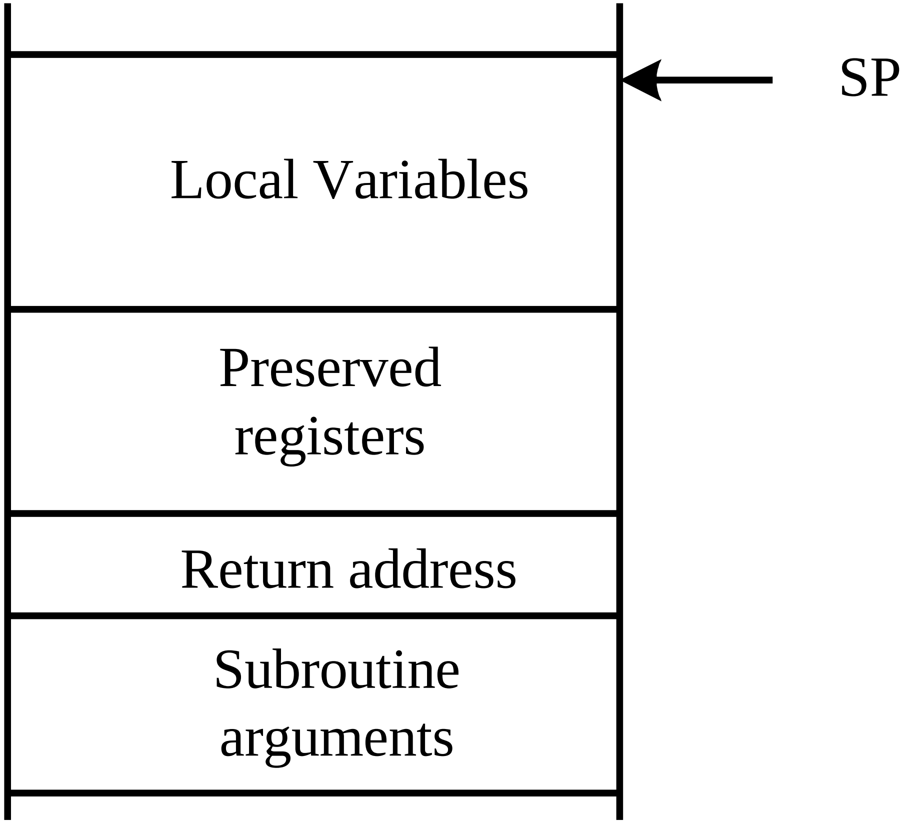
Techniques for parameter passing

- Using registers
  - Efficient and fast
  - General (data in memory not affected)
  - Example: `printf` uses the D register
  - But only a few registers available
- Using Condition Code Bits
  - Condition Code bits can be used to return Boolean values

Techniques for parameter passing

- Global Data Areas
  - Can be reached from any part of the program
  - May be difficult to find offending code when bug is detected
  - Increases coupling between modules
  - Passing addresses to global data is common
- Using the stack
  - Powerful and general
  - Restricts parameters and return data to the subroutine
  - Can also use stack for local variables
  - Use with care – balanced operations and good documentation

Using the Stack with Subroutines



- Stack contents
  - □ Arguments for subroutine (can be used also to return values)
  - □ Return address (placed on stack by BSR/JSR)
  - □ Contents of saved registers (to preserve their values while being used)
  - □ Space for local variables
  - □ Note that when RTS is called, SP must be pointing to the return address

Translating C Function Call

- A call to a C function requires the exchange of the following data
  - □ 0 or more arguments (values, maximum 16 bits)
  - □ 0 or 1 return value (maximum 16 bits)
  - □ Arguments and return value are limited to the following: simple type (8 or 16 bits), pointer (to an array or structure)
- Arguments are placed on the stack and D register
  - □ One single argument – passed in register D
  - □ Two or more arguments
    - Stack arguments on the stack starting with the rightmost argument in the argument list
    - Leftmost argument is placed in register D
- Return value is placed in register D
- Function local variables are stored either on the stack or in a register

Assembler Coding Standard

- Source code style
  - Adopt some standard format for code
- Program elements
  - Program Header
  - Assembler Equates
  - Main program
    - Location
    - Initialisation
    - Main Body
    - Program end
  - Modules – collection of subroutines
  - Constant data definitions
  - Variable Data
    - Location
    - Allocation
  - Sections – to support linking of modules

Program Header

Program Element **Program Example**

Program Header

```
;  MC68HC12 Assembler Example
;
;   This program is to demonstrate a
;   readable programming style.
;   It counts the number of characters
;   in a buffer and stores the result in
;   a data location.   It then prints
;   the number of characters using
;   D-Bug12 Monitor routines.
;   Source File:  M6812EX1.ASM
;   Author:  F. M. Cady
;   Created: 5/15/97
;   Modifications:  None
```

Assembler Equates

Program Element **Program Example**

System Equates.

```
;   Monitor Equates
out2hex:EQU     $FE16   ; Output 2 hex nibbles
putchar:EQU     $FE04   ; Print a character
;   I/O Ports
PORTH:  EQU     $24     ; Port H address
PORTJ:  EQU     $28     ; Port J address
```

Constant Equates

```
;   Constant Equates
CR:     EQU     $0d     ; CR code
LF:     EQU     $0a     ; LF code
NULL:   EQU     $00     ; End of ASCII string
NIL:    EQU     0       ; Initial data value
```

Memory Map Equates

```
;   Memory Map Equates
PROG:   EQU     $4000   ; Locate the program
DATA:   EQU     $6000   ; Variable data areas
STACK:  EQU     $8000   ; Top of stack
```

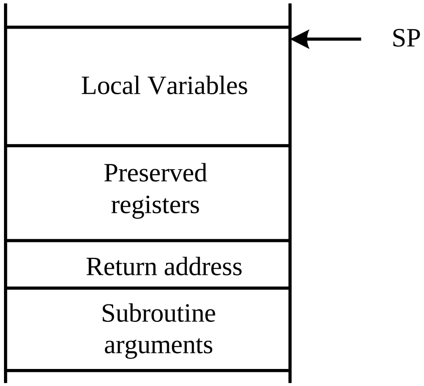
Main Program

- Location
  - ORG or SWITCH Directive to locate the code
- Initialisation
  - For example, the stack
  - Global variables
- Main Body
  - Code for main
  - Upper level algorithm
- Program end
  - For example, SWI

Modules - Subroutines

```
; Subroutine - updateRow(ptr,rptr) - Display Module
; Parameters: ptr - pointer to game matrix array (in D)
;           rptr - pointer to a row (on stack PBL_RPTR)
; Returns: nothing
; Variables: none

; Stack Usage
  OFFSET 0 ; to setup offsets into stack
PBL_PR_Y DS.W 1   ; preserve Y - used as rptr
PBL_PR_X DS.W 1   ; preserve X - used as ptr
PBL_RA   DS.W 1   ; return address
PBL_RPTR DS.W 1   ; rptr
```



- Draw a diagram showing the contents of the stack?
- Where would you place a local variable, say `cnt`?

Constant Data

- Location
  - Use ORG or SWITCH Directive
  - Typically placed at the end of code (destined for ROM)
  - Define constant data section using SECTION directive (see Module 3)
- Allocation
  - Use assembler directive DC

Variable Data

- Location
  - Use ORG or SWITCH Directive to place variables in RAM
  - Define Variable data section using SECTION directive (see Module 3)
- Allocation
  - Use assembler directive DS

Other Coding Principles

- Indent or not to Indent
  - Typically assembler source code is not indented
- Upper and Lower case
  - Assemblers are not case sensitive
  - Useful for making labels easier to read
  - All upper case for constants
- Use equates, not magic numbers
  - Makes source code self-documenting
- Use include files
  - Insert include files using “include” directive
  - For example, define common equates and include in a number of assembler files
- Commenting Style
  - Comment each line
  - Comment block of code
  - Include high-level pseudo-code design statements as comments

Structured Programming

- IF-ELSE
- IF-ELSEIF
- DO-WHILE and WHILE-DO
- Using Breaks
- See Cady Chapter 8.2 for more examples

Complex Test Expressions

- do { … } while((var1<10) && (var2>3));

```
    do:
       …           ;{  …  }
       ldaa var1      ;while( (var1 < 10)
     cmpa #10
     bhs endwhile
     ldaa var2 ;&& (var2 > 3)
     cmpa #3
     bhi do
  endwhile :    ;   );
```

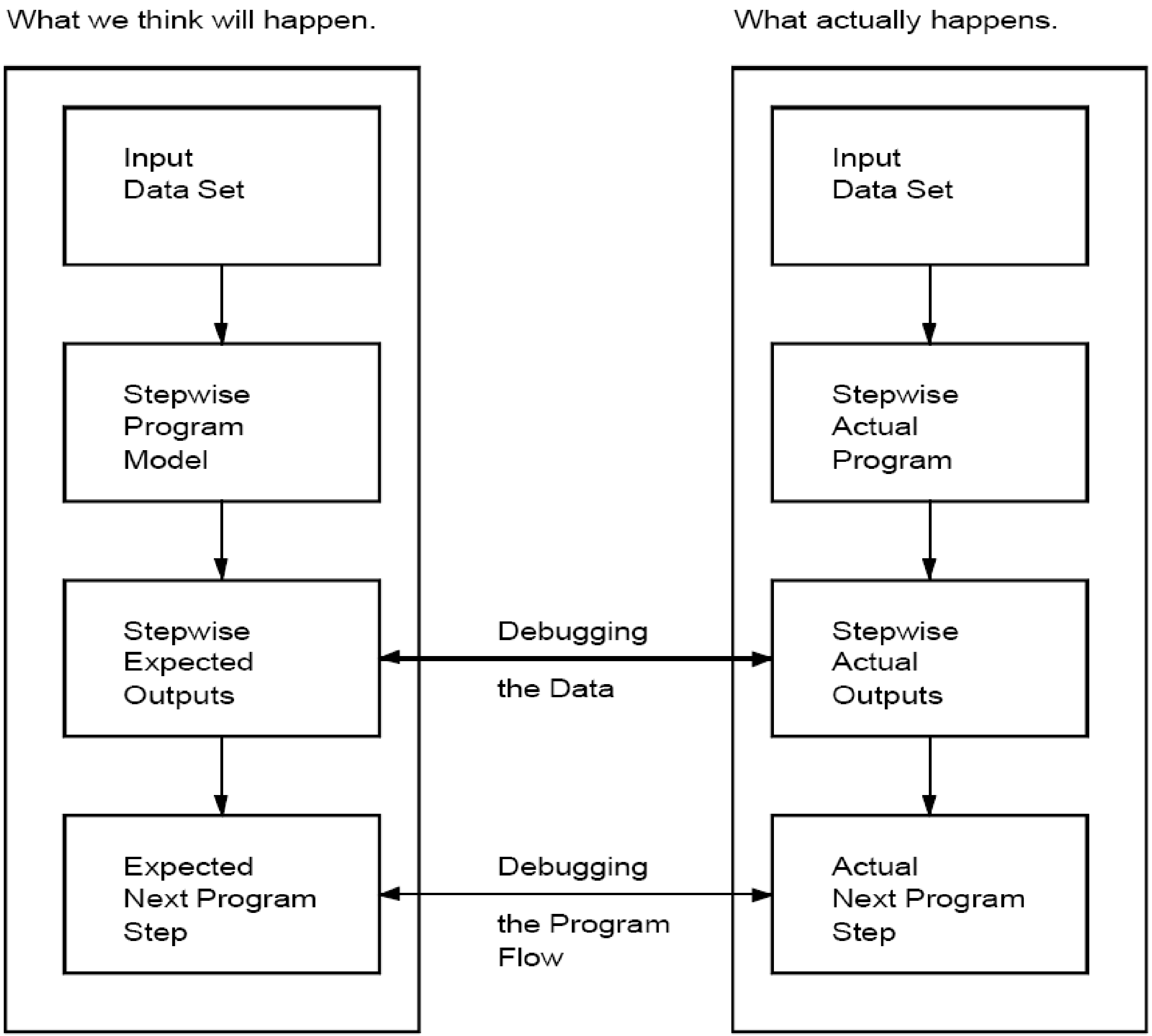
Complex Test Expressions

- if((var1=3) || (var1=2)) {…} else {…}

```
  ldaa var1   ; if((var=3)
  cmpa #3
  beq then
  ldaa var1   ;    || (var1=2)
  cmpa #2
  bne else    ;              )
then:
  …
  bra endif
else:
  …
endif:
```

Debugging – Using Analysis

- Use an analysis approach to correct faulty programs
  - Find out what the program is doing before making corrects
  - Try to match what you think the program should do to what it actually does



Fgiure 5-6 Analytical debugging model.

Code Walkthroughs

- Also called peer code reviews
- Effective debugging technique
- Eliminate problems before running the code
- Invite other experts both familiar and unfamiliar with the project
  - Get feedback on errors and even improving the design/code
- Use walkthroughs in your lab preparation with your partner

Debugging Plan

- Use structured design
  - Code will be divided into well defined blocks (using subroutines)
  - Using C for design will help with producing a structured design
- Allows possibility to isolate the problem to a block of code
  - Can set breakpoints after each block to see which block (or blocks) are causing the bug.

Debugging Tools

- Debuggers
  - Software that controls the execution of a program
  - Can be quite sophisticated (on development boards, *monitor* programs provide debugging functions)
- Program trace
  - Step through the program one instruction at a time
- Breakpoints
  - Define conditions to stop the program
  - Typically set at a program statement
    - Some debuggers allow conditions based on the value of data elements
    - Others allow hardware breakpoints (pattern on computer bus)

Debugging Data Elements

- What to examine during the program flow:
  - Registers: Contents of the CPU registers, including the CCR.
  - Memory:
    - Usually high-level debuggers can be used to examine declared variables.
    - With low-level debuggers, must examine memory in hexadecimal format
- Use the Source Code Listing
  - The source code listing is required to follow the program.
  - Use the assembler list file which shows the machine code
    - Can spot errors by examining this listing.
    - Useful to know where to set breakpoints

Typical Bugs

- Improper transfer to subroutines
- Forgetting to initialise the stack pointer
- Not allocating enough memory to the stack
- Unbalanced stack operations
- Subroutines that wipe out registers
- Transposed registers
- Not initialising pointer registers
- Not initialising registers and data
- Modifying condition codes before branch instructions
- Using the wrong conditional branch instruction
- Using the wrong memory addressing mode
- Using a 16-bit counter in memory
- Not stopping the program properly

Tricks of the Trade

- Use register addressing when possible
  - Faster and use less memory
- Use register indirect or indexed addressing:
  - Next most efficient addressing mode
  - Addresses can be calculated at run time
- Stack for temporary storage
  - In subroutines, care about balanced stack operations
- Do not use hard coded numbers
  - Use labels to represent constants
- Use assembler features
  - Use labels, assembler expressions, and macros
- Use but do not abuse comments
  - TFR A,B ; Transfer a register to b register ---- Not significant
  - TFR A,B ; Restore loop counter