ECED3204: Microprocessor

Part II--Assembly and C Language Programming

Jason J. Gu
Department of Electrical and Computer Engineering
Dalhousie University

Outline

i. Part II.1 AVR Assembly Language Programming
ii. Part II.2 Hardware and Software Development Tools for the AVR
iii. Part II.3 Advanced Assembly Programming and Subroutine Calls
iv. Part II.4 C Language Programming

Part II.1 AVR Assembly Language Programming

Outline

i. AVR Assembly Language Program Structure
ii. Memory
iii. Assembler Directives
iv. Writing Program to Perform Arithmetic
v. Write Program Loops
vi. Data Manipulation
vii. Time Delay

AVR Assembly Language Program Structure

- Types of statements: assembly instructions, assembler directives, or comments
- Statement forms: up to four fields
  1. [label:] directive [arguments] [comment]
  2. [label:] instruction [operands] [comment]
  3. Comment
  4. Empty line

Note: items placed in braces ([]) are optional.
AVR assembler is not case sensitive

AVR Assembly Language Program Structure (cont'd.)

- Label field
  - Optional
  - Purpose: identify a location in program memory or identify a location in data memory
  - Must begin with a letter, or _; followed by zero or more letters, digits, and special characters (and _); terminated by colon (:)
  - Examples:
    - Valid label: loop: True: _flight: ques:
    - Invalid label: ,oo: 5 space:

AVR Assembly Language Program Structure (cont'd.)

- Operation field: an AVR mnemonic or an assembler directive
  - Examples: inc, add, st, ld, ldd, std, sub,adc, .def, .org
- Operand field: follows operation field; separated by at least one space
  - Operand may be a constant, a register, or a memory location
  - Example:
    - Adc r10,r11 .org 0100
    - St X,r0 .db 11,12,13,14,15

AVR Assembly Language Program Structure (cont'd.)

- Comment field: used to explain the function or operation of one or more instructions
  - Comment forms
    ; [Text]
    //this is a comment line
    /* this is also a comment and may span multiple lines */
    - Adc r10,r11 ; adding r10 and r11 with carry
    - //following instruction does subtraction without carry
    - Sub r10,r11

Expressions

- To form the operand field of instruction
- To form assembler directive
  - May consist of
    - Operands — add r5,r6
    - Operators — + - * / << >> >=
    - Functions — low(expression)
- Internally 32 bits

Expressions (cont'd.)

- Operands that may be used
  - User defined labels given location counter value at place they appear
  - User defined variables (**set** directive)
  - User defined constants (**equ** directive)
  - Integer constants
  - Current value of PC

Expressions (cont'd.)

- Functions, example: ldi, r28 low(ibuf)

Function	Return value/description
Low (expression)	low byte (least significant byte) of the given expression
High(expression)	second byte of the given expression
Byte2(expression)	same as high
Byte3(expression)	third byte of the given expression
Byte4(expression)	fourth byte (most significant byte) of the given expression
Lwrd (expression)	lower word (bit 0 to 15) of the given expression
Hwrd(expression)	higher word (bit 16 to 31) of the given expression
Page(expression)	bit 16 to 21 of the given expression
Exp2(expression)	2**expression
Log2(expression)	integer part of log2(expression)

Functions (cont'd.)

Function	Return value/description
Int(expression)	truncates a floating-point expression to an integer
Frac(expression)	extracts the fractional part of a floating-point expression
q7(expression)	converts a fractional floating-point expression to a form suitable for the fmul/fmuls/fmulsu instructions (sign + 7-bit fraction)
q15(expression)	converts a fractional floating-point expression to the form returned by the fmul/fmuls/fmulsu instructions (sign+15-bit fraction)
abs(expression)	absolute value of the expression
DEFINED(symbol)	returns true if symbol is previously defined using .equ/.set/.def directives
STRLEN(string)	length of a string constant, in bytes

Expressions (cont'd.)

- Operators ~0X0F

Symbol	Operation
!	Logical not
~	Bitwise not
–	Unary minus
*	Multiplication
/	Division
+	Addition
–	Subtraction
<<	Shift left
>>	Shift right
<	Less than

Symbol	Operation
<=	Less or equal
>	Greater than
>=	Greater or equal
==	Equal
!=	Not equal
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
||	Logical OR
&&	Logical AND

Expressions (cont'd.)

- Formats of constants
  - Accepted bases: binary, octal, decimal and hexadecimal
- Binary constants
  - Lowercase **b** or uppercase **B** suffix: 1010b, 1010B
  - Lowercase **b** or uppercase **B** prefix to single-quoted value: b'101011', B'1110111'

Formats of constants (cont'd.)

- Octal constants
  - Lowercase **q** or uppercase **Q** suffix:
    - 1234q, 1347Q
  - Lowercase **q** or uppercase **Q** prefix to single-quoted value: q'11224', Q'7654321'
- Decimal constants
  - No suffix or prefix
  - Lowercase **d** or uppercase **D** prefix to single-quoted value:
    - d'2345', D'9843210'

Formats of constants (cont'd.)

- Hexadecimal constants
  - Lowercase **h** or uppercase **H** as suffix to value preceded by a 0:
    - 0ABCDEh, 0A8B30H
  - **0x** as a prefix:
    - 0x1234, 0xFFFC
  - Lowercase **h** or uppercase **H** prefix to single-quoted value:
    - h'3344', H'FFFC'

Memory Class

- Class types:
  - code memory, data memory, and EEPROM memory
- Segment within the memory space:
  - data segment, program segment, or EEPROM segment: dseg, cseg, eseg
- Location counter assigned to each segment during assembly process so assembler knows next place to use

Assembler Directives

Directive	Description
byte	Reserve byte
cseg	Code segment
db	Define constant byte(s)
def	Define a symbolic name to a register
device	Define the device to assemble for
dseg	Data segment
dw	Define constant word(s)
endmacro	End macro
equ	Set a symbol equal to an expression

Table 3.2 AVR assembler directives

Assembler Directives (cont'd.)

Directive	Description
eseg	EEPROM segment
exit	Exit from file
include	Read source from the specified file
list	Turn on listfile generation
listmac	Turn on macro expansion
macro	Begin macro
nolist	Turn off listfile generation
org	Set location counter
set	Set a symbol to an expression

Table 3.2 AVR assembler directives (cont'd.)

Assembler Directives: Example

```
        .dseg           ;start a new data seg
buf:    .byte   20      ;reserve 20 bytes
…
        .cseg           ;start a new code seg
        ldi     XL,low(buf)   ;load X register low
        ldi     XH,high(buf) ;load the X register high
.....
//use a symbol to refer to a register. Redefine r5 as myregister
        .def    myregister =r5;
```

Assembler Directives: Example

- .device ATXMega128A1
  - .cseg ; start a new code segment
- array: .db 1,2,3,4,5 ;initialize pmemory with 8-bit value
- String: .db “morning!”,0
- .dw 0, oxEEEE ;initialize pmemory with 16 bit
- // equ directive assigns a value to a label, which is a constant and can not change

. .equ TRUE=1

- // set directive assigns a value to a label, which can be changed

. .set my_offset = 0x10

Macro Directive: Example

- A macro is a name assigned to a group of instructions of directives.
-
- .macro add3
- add @0, @1
- adc @2,@0
- .endmacro
  - Add three number together.
  - @0,@1,@2 are the first,second and third marro parameter.
  - Add3 r15,r16,r17

AVR Assembly Program Template

- After reserving space for handling reset and interrupts, assembly program (using AVR assembler) looks like:

```
        .include    <xxx.inc>       ; xxx is the AVR device name
        .org        0x00
        jmp         start
        ...
        org         0xF6
start:  ...
```

Software Development Issue

- Effective software development involves
  - Systematic software development methodology
  - Developing algorithms for solving problems
  - Developing reusable software
  - Using software tools for debugging
  - Implementing good programming style
  - Top down design with hierarchical refinement
  - Testing

Writing Programs to Perform Arithmetic

- Example: Write a program to add the 8-bit numbers stored at data memory locations 0x20 and 0x21, and save the sum at data memory location 0x22.
- Example: Write a program that adds 20 to data memory locations 0x21.

```
.include <XXX.inc>
.org 0x00
Jmp start
.org 0xF6

Start:  lds r1,0x20
        lds r2,0x21
        add r2,r1
        sts 0x22,r2
```

```
.include <XXX.inc>
.org 0x00
Jmp start
.org 0xF6

Start:  ldi r16,20
        lds r2,0x21
        add r2,r16
        sts 0x21,r2
```

Writing Programs to Perform Arithmetic (cont'd.)

- The carry/borrow flag: SREG register bit 7
  - Affected by all addition/subtraction instructions
  - Set to 1 or 0 dependent on carry out
- Multiprecision addition
  - Two numbers longer than 8 bits are added one byte at a time proceeding from least significant byte (lsb) to most significant byte (msb)

Writing Programs to Perform Arithmetic (cont'd.)

- The C flag and subtraction
  - Set to 1 when subtrahend larger than minuend
  - Set to 0 when minuend larger than subtrahend
- Multiprecision subtraction:
  - subtraction of numbers longer than one byte for an 8-bit microcontroller
- Multiplication and division

Writing Programs to Perform Arithmetic (cont'd.)

Example Write a program to add the 32-bit numbers stored at data memory locations 0x200-0x203 and 0x204-0x207, respectively, and store the sum at 0x208-0x20B.

```
.include <m2560def.inc>
        .cseg
        .org    0x00
        rjmp    start
        .org    0xF6
start:  lds     r0,0x200    ; fetch the lsbs
        lds     r1,0x204    ;   "
        add     r0,r1       ; add them
        sts     0x208,r0    ; save the lsb of sum
        lds     r0,0x201
        lds     r1,0x205
        adc     r0,r1
        sts     0x209,r0
        lds     r0,0x202
        lds     r1,0x206
        adc     r0,r1
        sts     0x20A,r0
        lds     r0,0x203    ; fetch the msbs
        lds     r1,0x207    ;   "
        adc     r0,r1       ; add them
        sts     0x20B,r0    ; save the msb of sum
// end of program
```

Multiplication and Division (cont'd.)

Mnemonics	Description	Operation
mul Rd, Rr(1)	Multiply unsigned	R1:R0 ← [Rd] × [Rr] (UU)(4)
muls Rd, Rr(2)	Multiply signed	R1:R0 ← [Rd] × [Rr] (SS)(4)
mulsu Rd, Rr(3)	Multiply signed with unsigned	R1:R0 ← [Rd] × [Rr] (SU)(4)
fmul Rd, Rr(3)	Fractional multiply unsigned	R1:R0 ← [Rd] × [Rr] << 1 (UU)(4)
fmuls Rd, Rr(3)	Fractional multiply signed	R1:R0 ← [Rd] × [Rr] << 1 (SS)(4)
fmulsu Rd, Rr(3)	Fractional multiply signed with unsigned	R1:R0 ← [Rd] × [Rr] << 1 (SU)(4)

Note: 1. $0 \le d \le 31$; $0 \le r \le 31$
2. $16 \le d \le 31$; $16 \le r \le 31$
3. $16 \le d \le 23$; $16 \le r \le 23$
4. U: unsigned; S: signed

Table 3.3 AVR multiply instructions

Multiplication

Example Write a program to multiply the 8-bit numbers stored at data memory locations 0x200 and 0x204, respectively, and store the sum at 0x208-0x209.

```
.include <m2560def.inc>
        .cseg
        .org    0x00
        rjmp    start
        .org    0xF6
start:  lds     r2,0x200    ; fetch the number
        lds     r3,0x204    ;   "
        mul     r2,r3       ; multiply them
        sts     0x208,r0    ; save the lsb of sum
        sts     0x209,r1    ; save the msb of sum
// end of program
```

Multiplication and Division (cont'd.)

- Fractional multiplication often used in digital signal processing
- AVR does not provide divide instruction
  - Programmer must write a subroutine to implement division

Accessing Data in Data and Program Memory

- Accessing data stored in program memory

  Step 1: Place address of data in program memory in the Z register

  Step 2: Execute an **lpm** or **elpm** instruction

  - lpm r1,z ;load memory content into r1
- Note:
  - To access data in program memory, we need to use z register. Since the location counter of program memory counts words instead of bytes, a label in the program memory represents a word address(=byte address/2). We should multiply a word address by 2 to translate it to byte address (num<<1)

Writing Programs to Perform Arithmetic (cont'd.)

Example: Write a program to load data from program memory and store at 0x208

```
.include <m2560def.inc>
        .cseg
        .org    0x00
        rjmp    start
        .org    0xF6
start:  ldi     ZL, low(Parray <<1)
        ldi     ZH, high(Parray <<1)
//      setpointer ZL,ZH, (parrary<<1)
        lpm     r0,Z
        sts     0x208,r0

Parray: .db     11,12,13,14,15,16
        .db     18,21,11,13,15,55
        .db     56,78,99,77,66,55
// end of program
```

Writing Program Loops

- The infinite loop: sequence of instructions in which microcontroller stays forever
  - Implemented with unconditional jump instructions: rjmp, ijmp, eijmp, and jmp
- The for-loop: finite loop
  - Syntax
    For i = n1 to n2 do S *or*
    For i = n2 downto n1 do S, where i is loop index

Writing Program Loops

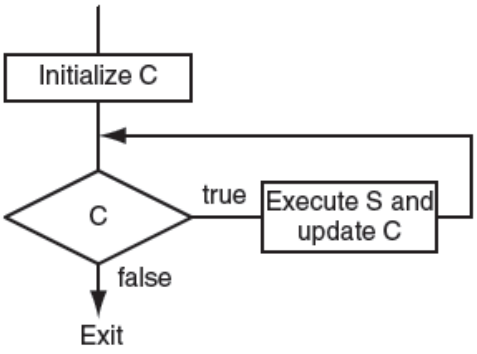
```
Loop:   ......
        ......
        rjmp    loop
```

Mnemonics	Description	Operation
rjmp k	Relative jump	PC ← [PC] + k + 1
ijmp	Indirect jump to (Z)	PC(15:0) ← [Z], PC(21:16) ← 0
eijmp	Extended indirect jump to (Z)	PC(15:0) ← [Z], PC(21:16) ← EIND
jmp k	Jump	PC(15:0) ← k

Table 3.4 ■ AVR unconditional jump instructions

Writing Program Loops (cont'd.)

- The while-loop: finite loop



Note:
- logical expression C
- statement S

Figure 3.3 The WHILE … DO construct

Writing Program Loops (cont'd.)

- The repeat-until loop: finite loop
  - Repeat-S-until-C: instruction sequence below performs operation N times

```
        .equ    N       = 20
        .def    lpcnt   = r16
        ...
        ldi     lpcnt,0
loop:   ...
        ...
        inc     lpcnt               ; increment loop count
        cpi     lpcnt,N             ; compare loop count with N
        brne    loop                ; loop count not equal to N, continue
```

Writing Program Loops (cont'd.)

Mnemonics	Description	Operation
cpse Rd, Rr	Compare, skip if equal	If ([Rd] == [Rr]) PC ← [PC] + 2 or 3
cp Rd, Rr	Compare	[Rd] − [Rr], and update Z, C, N, V, S, H
cpc Rd, Rr	Compare with carry	[Rd] − [Rr] − C, and update Z, C, N, V, S, H
cpi Rd, k	Compare with immediate	[Rd] − k, and update Z, C, N, V, S, H; 0 ≤ k ≤ 255

Table 3.5 ■ AVR compare instructions

Shift and Rotate Instructions

Mnemonics	Description	Operation
lsl Rd	Logical shift left	Rd(n + 1) ← Rd(n), Rd(0) ← 0, C ← Rd(7), n = 0..6
lsr Rd	Logical shift right	Rd(n) ← Rd(n + 1), Rd(7) ← 0, C ← Rd(0), n = 0..6
rol Rd	Rotate left through carry	Rd(n + 1) ← Rd(n), Rd(0) ← C, C ← Rd(7), n = 0..6
ror Rd	Rotate right through carry	Rd(n) ← Rd(n + 1), Rd(7) ← C, C ← Rd(0), n = 0..6
asr Rd	Arithmetic shift right	Rd(n) ← Rd(n + 1), n = 0..6; Rd(7) ← Rd(7)
swap Rd	Swap nibbles	Rd(3..0) ↔ Rd(7..4)

Table 3.7 AVR shift and rotate instructions

Example on Board

Boolean Instructions

```
in   r16, PORTB
ori  r16, 0x02
out PORTB, r16
```

SER Rd 

Mnemonics	Description	Operation
and Rd, Rr	Logical AND	Rd ← [Rd] • [Rr]
andi Rd, k	Logical AND with immediate	Rd ← [Rd] • k
or Rd, Rr	Logical OR	Rd ← [Rd] ∨ [Rr]
ori Rd, k	Logical OR with immediate	Rd ← [Rd] ∨ k
eor Rd, Rr	Exclusive OR	Rd ← [Rd] ⊕ [Rr]
com Rd	One's complement	Rd ← 0×FF − [Rd]
neg Rd	Two's complement	Rd ← 0×00 − [Rd]
sbr Rd, k	Set bit (s) in register	Rd ← [Rd] ∨ k
cbr Rd, k	Clear bit (s) in register	Rd ← [Rd] • (0×FF − k)
tst Rd	Test for zero or minus	Rd ← [Rd] • [Rd]
clr Rd	Clear register	Rd ← [Rd] ⊕ [Rd]
set Rd	Set register	Rd ← 0×FF

Table 3.8 A summary of AVR Boolean instructions

Boolean Instructions

- Example:
  - in r16, portB ;read portB
  - andi r16,0x0F ;clear upper 4bit
  - out portB, r16 ;store back

Bit Manipulating Instructions

- Used to change the value of a bit or copy a bit from one register to another
- See Table 3.9 A summary of the AVR bit manipulation instructions

Bit Manipulating Instructions

Mnemonics	Description	Operation
bset s	Flag set	SREG(s) ← 1
bclr s	Flag clear	SREG(s) ← 0
sbi A, b	Set bit in I/O register	I/O(A, b) ← 1
cbi A, b	Clear bit in I/O register	I/O(A, b) ← 0
bst Rr, b	Bit store from register to T	T ← Rr(b)
bld Rr, b	Bit load from T to register	Rd(b) ← T
sec	Set carry	C ← 1
clc	Clear carry	C ← 0
sen	Set negative flag	N ← 1
cln	Clear negative flag	N ← 0
sez	Set zero flag	Z ← 1
clz	Clear zero flag	Z ← 0
sei	Enable global interrupt	I ← 1
cli	Disable global interrupt	I ← 0
ses	Set signed test flag	S ← 1
cls	Clear signed test flag	S ← 0
sev	Set 2's complement overflow	V ← 1
clv	Clear 2's complement overflow	V ← 0
set	Set T in SREG	T ← 1
clt	Clear T in SREG	T ← 0
seh	Set half carry flag in SREG	H ← 1
clh	Clear half carry flag in SREG	H ← 0

Table 3.9 ■ A summary of the AVR bit manipulation instructions

Create Time Delay Using Program Loops

- Overview
  - CPU clock controls AVR instruction execution
  - Each AVR instruction is completed in one to five CPU clock cycles (See Appendix A)
  - A time delay can be created by repeating a sequence of instructions for a certain number of times.

Create Time Delay Using Program Loops (cont'd.)

- Using a program loop to create time delay

  Step 1: Select sequence of instructions that take a certain number of CPU clock cycles to execute

  Step 2: Repeat the instruction sequence for an appropriate number of times

- Loop time delays are inaccurate due to some overhead

Create Time Delay: Example

Example: create a time delay of 1 second:

Note: 16 MHz

```
        ldi     r18,10
loop2:  ldi     r17,200         ; set external loop count to 200
loop1:  ldi     r16,250         ; set loop count to 250
loop0:  push    r0              ; 2 CPU clock cycles
        pop     r0              ; 2 CPU clock cycles
        push    r0
        pop     r0
        push    r0
        pop     r0
        push    r0
        pop     r0
        push    r0
        pop     r0
        push    r0
        pop     r0
        push    r0
        pop     r0
        push    r0
        pop     r0
        nop                     ; 1 CPU clock cycle
        dec     r16             ; 1 CPU clock cycle
        brne    loop0           ; take 2 (1) CPU clock cycles when branch is
taken (not taken)
        dec     r17
        brne    loop1
        dec     r18
        brne    loop2
```

Part II.2 Hardware and Software Development Tools for the AVR

Outline

i. Hardware Development Tools
ii. Software Development Tools
iii. AVR IDE
iv. Development Tips

Development Tools for the Atmel AVR

- Microcontroller development tools categories
  - Hardware tools to support device programming, program execution, and software debugging
  - Software tools to allow programs to be entered, assembled/compiled, linked, and executed.

Hardware Development Tools

- Most useful hardware tools for learning AVR MCU: microcontroller demo board, programmer, and debug adapter
- Choosing a Demo Board for Learning the AVR
  - Desirable peripheral functions: I/O ports, interrupts, timer functions, universal asynchronous receiver transmitter (UART) port, serial peripheral interface (SPI), two-wire interface (TWI), A/D converter

Choosing a Demo Board for Learning the AVR (cont'd.)

- Signal pins available to the user so that he or she can experiment with other peripheral chips
- Some people recommend using a bare kit that only makes I/O signals available to the user
  - Forces the user to do more wiring; learn more
  - Difficulty may frustrate user
- Connector to connect a debug adapter or programmer so that the user can download the program onto the demo board for execution

Hardware Development Tools (cont'd.)

- The EasyAVR M1280 Demo Boards: made by AVRVI
- Stingray XMega Demo Board: from Xbit Inc.
- Arduino Demo Kits: open-source electronics prototyping platform
- Debug Adapters from Atmel: AVR Dragon, JTAGICE3, JTAGICE mkII, and AVR ONE!
- Dalhousie home made board (Lab kit)

Software Development Tools

- Text editor to enter the program
- Assembler and compiler to assemble and compile the program
- Linker to resolve variables and subroutines cross-referencing and memory assignment
- Simulator and debugger to debug the software
- Project manager to coordinate the overall debug activities

Software Development Tools

- Integrated development environment (IDE): single software package that includes assembler, compiler, linker, librarian, simulator, debugger, and project manager

Using the AVR Studio IDE (Lab1)

- Steps for developing a program
  - **Step 1:** Create a project
    Start→All Programs→Atmel AVR tools→AVR Studio 6.2
  - **Step 2:** Enter the program
  - **Step 3:** Assemble (or build) the project
    Syntax or semantic errors listed
  - **Step 4:** Program execution and debugging

Program Execution and Debugging (cont'd.)

- Methods to start program execution and debugging in the AVR Studio
  - Start debugging: immediately starts an emulation session
  - Start debugging and break: breakpoint set at the entry point of the produced executable file
  - Start without debugging: launches the debugging tool, but not the emulation session

Program Execution and Debugging (cont'd.)

- Program window: yellow arrow at the left-hand side of the points to the instruction (jmp start) at address 0x00
- Watch window: available only in debug session
  - Setup watch list: track changes of variables' values

Program Execution and Debugging (cont'd.)

- Target MCU: reset before running the program
- Breakpoint:
  - address of an instruction where program execution stops
  - User can set breakpoints in start without debugging mode and the start-debugging-and-break mode
  - Properties: hit count, condition, and action

Program Execution and Debugging (cont'd.)

- Run to cursor:
  - quickly find out if program executes correctly up to the cursor position
- Stepping over instructions
  - Step into: causes the MCU to execute one instruction at a time
  - Step over: if the instruction being stepped is not a call instruction, the MCU only executes one instruction and stops

Stepping Over Instructions (cont'd.)

- Step over (cont'd.): if instruction being stepped is a call instruction, then the MCU completes the execution of the whole subroutine and stops at the instruction after the call instruction
- Step out: if the MCU is executing a subroutine, this command causes the MCU to complete the execution of the current subroutine and return to the caller of this subroutine

Tips for Assembly Program Debugging

- Syntax and semantic errors
  - Misspelling of instruction mnemonics
  - Symbol not terminated with a colon character when it is defined
  - Missing operands
  - Invalid register
  - Undefined Symbols
- Logical errors: HARD TO FIND

Tips for Assembly Program Debugging (cont'd.)

- General debug strategy
  - Determine whether the program runs correctly:
    - may use the Run-to-Cursor feature to test the program
  - Locate any error:
    - Run-to-Cursor, single-stepping instructions, and Watch Window may assist user
  - Fix the error:
    - must first identify type of error

Tips for Assembly Program Debugging (cont'd.)

- Common program logical errors
  - Forgetting initializing program variables
  - Using the same register for multiple variables at the same time
  - Using the wrong instruction for a certain purpose
  - Operand size mismatch

Common Program Logical Errors (cont'd.)

- Missing a return instruction in a subroutine or interrupt service routine
- Forgetting to pass parameters to the subroutine being called
- Incorrect program algorithm

Project File Structure

- AVR Studio IDE: creates a directory of the project name under the selected directory
  - Example: c:\books\avr\programs\ch04\tutor1
  - AVR Studio Solution file and directory created within this directory c:\books\avr\programs\ch04\tutor1\tutor1
    - Under this directory: Debug directory, assembly program file (.asm extenstion), and assembler project file

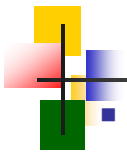
Part II.3 Advanced Assembly Programming and Subroutine Calls

Outline

i. Introduction
ii. Subroutine calls
iii. Stacks
iv. Examples

Introduction

- Same sequence of instructions commonly executed in several places of the same program
  - Macros: each call duplicates the same sequence of instructions in places where the macro is invoked increasing the program size
  - Subroutine: call causes change to program flow



Introduction (cont'd.)

- Most popular software development methodology: "top-down design with hierarchical refinement"
  - Made possible by subroutine mechanism: subroutine-call and return-from-subroutine instructions
- Macros and subroutines support software reuse

Subroutine (cont'd.)

- When subroutine-call instruction is executed, the processor:
  - Saves the return address in the stack
  - Loads the starting address of the subroutine into the program counter
  - Processor control is transferred to the subroutine
  - After execution, control resumes with instruction following subroutine call

The Stack Data Structure

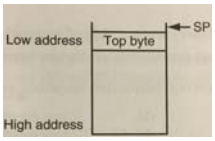
- Stack: data structure from which elements can be accessed only from its top
  - Uses push and pull (or pop) operations
  - AVR stack grows from high address toward lower address; stack pointer (SP) points to the byte immediately above the top byte of the stack
  - Stack memory space is limited: danger of stack overflow and stack underflow
    - Overflow: pushes data too many times, thus sp outside the stack area
    - Underflow: pop too many times, thus sp points below bottom

The Stack Data Structure (cont'd.)

- Initializing the stack pointer: must occur before stack can be used
  - Stack data structure must be created inside the on-chip SRAM area
    - Ldi temp, low(RAMED); temp is one register r16-r31
    - Out SPL, temp ;initialize SP
    - Ldi temp, high(RAMED)
    - Out SPH, temp
- Note: SPL and SPH for Mega and CPU_SPL and CPU_SPH for Xmega.

Instructions for Stack Operation (cont'd.)

- Instructions for stack operation
  - Push Rd instruction places the contents of the Rd register at the location pointed to by the SP and then decrements SP by 1
  - Pop Rd instruction increments SP by 1 and then copies the contents of the memory location pointed to by the SP to the Rd register
  - Example:
    - Push r1
    - Pop r0



An Example of Subroutine

- Subroutine is a sequence of instructions that can be called from many places of a program
- Sequence of instructions can be converted to a subroutine by
  - Adding a label (name of the subroutine) to the first instruction of the sequence.
  - Adding a ret instruction as the last instruction of the sequence

An Example of Subroutine (cont'd.)

Instruction sequence to swap r16, r17 contents:

```
push    r16       ; save the contents of r16 in stack
mov     r16,r17   ; copy the contents of r17 to r16
pop     r17       ; place the contents of r16 in r17
```

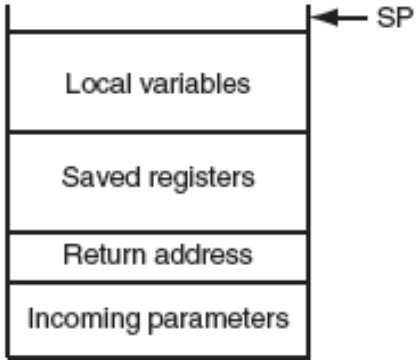
Converted to a subroutine:

```
swapRegs:  push    r16       ; save the contents of r16 in stack
           mov     r16,r17   ; copy the contents of r17 to r16
           pop     r17       ; place the contents of r16 in r17
           ret
```

Issues Related to Subroutine Calls

- Parameter passing
  - Caller of the subroutine may pass parameters, using registers (r0 to r31 used with the AVR), stack, or global memory
- Local variable allocation and deallocation
  - Use CPU registers for local variables when possible; use stack if more space is needed
  - Subroutine should deallocate stack space used by local variables before returning to the caller

Issues Related to Subroutine Calls (cont'd.)




```
allocate k byte from stack:

in      r18,SPL
in      r19,SPH   ;Get the SP
sbiw    r18,k     ;allocate K byte
out     SPL, r18
out     SPH,r29
```

Figure 5.2 Structure of the AVR stack frame

Issues Related to Subroutine Calls (cont'd.)

- Result returning
  - Subroutine may return its computation result in registers, stack, or global memory
- Accessing local variables in the stack
  - Data-indirect-with displacement addressing mode
- Register usage issue
  - Must avoid caller-callee interference
    - Callee: subroutine being called

Register Usage Issue (cont'd.)

Name	Usage
r4~r7, r12~r15, r28, r29	Callee saved
r0~r3, r8~r11, r20, r21	Caller saved
r16~r19, r30, r31	Parameter passing
r22~r27	Result returning

Table 5.1 Recommendation for register usage

Issues Related to Subroutine Calls (cont'd.)

Instruction	Operation
call k	PC ← k; stack ← PC + 2; SP ← SP – 2 (devices with 16-bit PC) SP ← SP – 3 (devices with 22-bit PC)
eicall	PC(15:0) ← Z(15:0); PC(21:16) ← EIND; stack ← PC + 1; SP ← SP – 3
icall	PC(15:0) ← Z(15:0); PC(21:16) ← 0; (devices with 22-bit PC); stack ← PC + 1; SP ← SP – 2 (devices with 16-bit PC); SP ← SP – 3 (devices with 22-bit PC)
rcall k	PC ← PC + K + 1; stack ← PC + 1; SP ← SP – 2 (devices with 16-bit PC); SP ← SP – 3 (devices with 22-bit PC)
ret	PC(15:0) ← stack (devices with 16-bit PC); SP ← SP + 2 PC(21:0) ← stack (devices with 22-bit PC); SP ← SP + 3

Table 5.2 Subroutine call instructions

Issues Related to Subroutine Calls (cont'd.)

```
//call  K subroutine
        Ldi     r18,0x10
        call    loopk
        ....
        ....
Loopk:  adi r18, 5
        ...
        ret
```

```
// EICALL subroutine
/* calls a subroutine pointed to by the Z pointer and the EIND resister in the I/O space
*/
        Ldi     r16,0x10
        out     EIND, r16
        ldi     r30,0x12
        ldi     r31,0x11
        eicall
//call the subroutine at 0x101112
```

```
// ICALL subroutine
/* calls a subroutine pointed to by the Z pointer
*/
        ldi     r30,0x12
        ldi     r31,0x11
        icall
//call the subroutine at 0x1112
```

```
// RCALL subroutine
/* the range of constant k 11bit is between -2048 and 2048. The instruction allows the program to calls a subroutine k words away.
*/
        rcall   looprcall
        ...
        ...
Looprcall:sub   r30,0x12
        ...
        ...
        ret
```

Writing Subroutines to Perform Multiprecision Arithmetic

- Writing subroutines to perform 16-bit unsigned multiplication
  - To multiply two 16-bit unsigned numbers, the multiplier and the multiplicand must be broken down into 8-bit chunks, and four 8-bit by 8-bit multiplications are performed
  - Example:

$$P = P_H P_L = P_H \times 2^8 + P_L$$
$$Q = Q_H Q_L = Q_H \times 2^8 + Q_L$$

Writing Subroutines to Perform Multiprecision Arithmetic

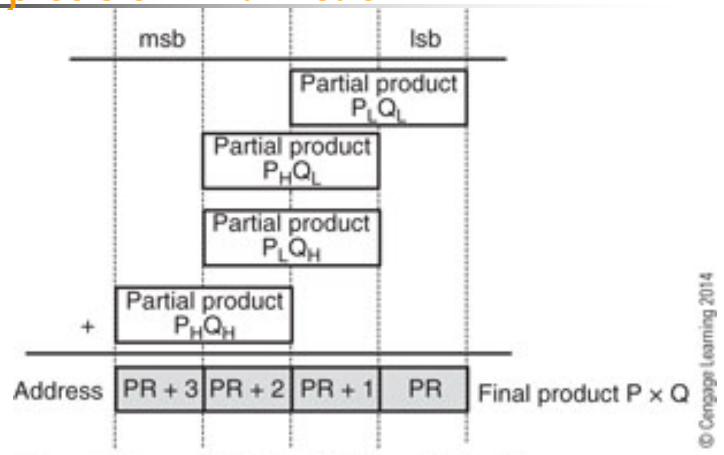


Figure 5.4 ■ 16-bit by 16-bit multiplication

Writing Subroutines to Perform Multiprecision Arithmetic

Example : **eg05_01.asm**

Write a subroutine to multiply two unsigned 16 bit integers passed in r16:r17 and r18:r19 and return the production in r22,r23,r24,r25. r17 and r19 hold the upper byte of two numbers to be multiplied and r25 down to r22 hold the most significant to the least significant bytes of the product

Writing Subroutines to Perform Multiprecision Arithmetic (cont'd.)

- Writing a subroutine to perform 16-bit signed multiplication
  - n-bit MCU: the result of any arithmetic operation is equal to the remainder of the initial result divided by 2^n (i.e., it is performing a modulo-2^n operation)

Writing a Subroutine to Perform 16-bit Signed Multiplication (cont'd.)

- Four possibilities for the multiplication of signed numbers

Case 1: Both operands are positive, op1=P,op2=Q

Use unsigned multiplication

Case 2: First operand is negative -P, second operand is positive Q

$-P*Q=(2^n-P)*Q=2^n*Q-P*Q=2^{2n}-P*Q+2^n*Q$

Case 3:First operand is positive P, second operand is negative -Q

$P*(-Q)=P*(2^n-Q)=2^n*P-P*Q=2^{2n}-P*Q+2^n*P$

Case 4: Both operands are negative –P,-Q

$(-P)*(-Q)=(2^n-P)*(2^n-Q)=2^{2n}-2^n*P-2^n*Q+P*Q=P*Q+2^{2n}-2^n*P+2^{2n}-2^n*Q$

$=P*Q+2^n*(2^n-P)+2^n*(2^n-Q)$

Writing a Subroutine to Perform 16-bit Signed Multiplication (cont'd.)

- N bit multiplication steps
- Step 1: Multiply two operands disregard their signs
- Step 2: If op1 is negative, then subtract op2 from the upper half of product
- Step3: if op2 is negative, then subtract op1 from the upper half of the product

Writing Subroutines to Perform Multiprecision Arithmetic

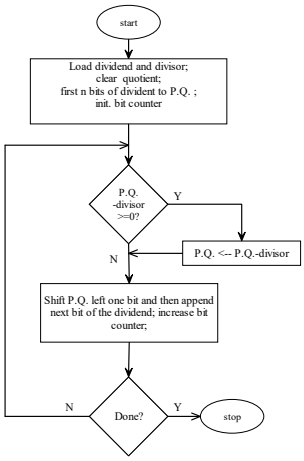
Example : **eg05_02.asm**

Write a subroutine to multiply two signed 16 bit integers passed in r16:r17 and r18:r19 and return the production in r22,r23,r24,r25. r17 and r19 hold the upper byte of two numbers to be multiplied and r25 down to r22 hold the most significant to the least significant bytes of the product

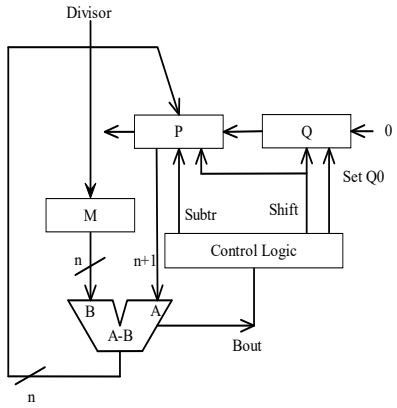
Writing Subroutines to Perform Multiprecision Arithmetic (cont'd.)

- Writing subroutines to perform unsigned multiprecision division
  - Repeated-shift-and-subtract method: widely used division method
    - Initialization phase
    - Shift-and-subtract phase

Writing Subroutines to Perform Multiprecision division: Comparison Method



Writing Subroutines to Perform Multiprecision division



01011 | 100010010

M=01011

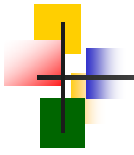
Writing Subroutines to Perform Multiprecision division: Comparison Method

B	P					Q									operation
1	0	0	0	0	0	1	0	0	0	1	0	0	1	0	load M,Q, clear P
1	0	0	0	0	1	0	0	0	1	0	0	1	0	0	shift P.Q
1	0	0	0	1	0	0	0	1	0	0	1	0	0	0	shift
1	0	0	1	0	0	0	1	0	0	1	0	0	0	0	shift
1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	shift
0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	subtr.+set Q0
1	0	0	1	1	0	0	0	1	0	0	0	0	0	1	shift
0	0	1	1	0	0	0	1	0	0	0	0	0	1	0	subtr. +set Q0
1	0	0	0	0	1	0	1	0	0	0	0	0	1	1	shift
1	0	0	0	1	0	1	0	0	0	0	0	1	1	0	shift
1	0	0	1	0	1	0	0	0	0	0	1	1	0	0	shift
1	0	1	0	1	0	0	0	0	0	1	1	0	0	0	

Writing Subroutines to Perform Multiprecision Arithmetic

Example : **eg05_03.asm**

Write a subroutine to divide an unsigned 16 bit integer into another 16 bit unsigned integer. Dividend and divisor are passed in r16:r17 and r18:r19, respectively. Reminder and quotient are to be returned in r22:r23,r24:r25, respectively.



Writing a Subroutine

- Converting an internal binary number into a BCD string
  - Binary number in the computer memory must be converted into a BCD string before being output
  - BCD string uses the ASCII code to represent each decimal digit (Skip)

Writing a Subroutine to Perform 16-bit Signed Multiplication (cont'd.)

- Signed Division Operation
  - One complication for signed division: must also set the sign of the remainder
  - Equation: Dividend = Quotient X Divisor + Remainder
  - Correctly signed division algorithm negates the quotient if the signs of the operands are opposite and makes the sign of the nonzero remainder match that of the dividend (Skip)

Writing Subroutines to Perform Multiprecision Arithmetic (cont'd.)

- Finding the square root: method based on successive approximation

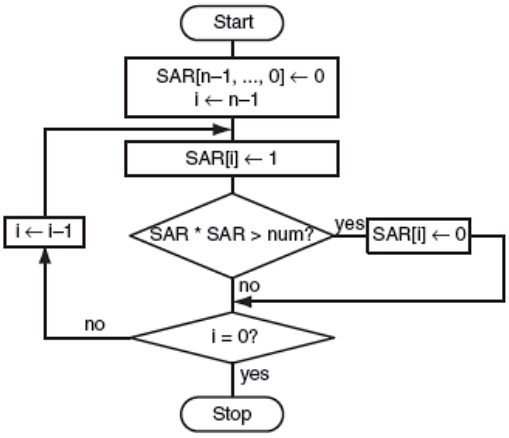
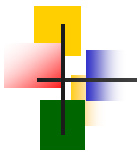


Figure 5.5 Successive-approximation method for finding square root

Finding the square root

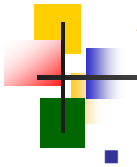
Example : **eg05_05.asm**

Write a subroutine to implement the square root algorithm. This algorithm will find the square root of a 32-bit unsigned integer. The integer is passed in r19-r16, and the square root is returned in r22-r23.



Writing a Subroutine

- Prime test subroutine
  - Not extremely useful for embedded applications
  - Good example for software reuse



Prime Test Subroutine (cont'd.)

- Algorithm

Step 1: Let num, k, and isprime represent the number to be tested, the loop index, and the flag to indicate if num is a prime number

Step 2: isprime ← 0; tlimit ← square root of num;

Step 3:

```
for k = 2 to tlimit do
    if ((num % k) == 0) then return;
isprime ← 1;
return;
```

Prime Test Subroutine (cont'd.)

Example : eg05_06.asm

Write a subroutine to determine whether an unsigned 16 bit integer is a prime number

In this example, sqroot16 and div16u will be reused.

Subroutines with Local Variables in Stack

- Subroutine to convert a BCD string to a binary number
  - To allow the MCU to manipulate text data that represent numeric values, they must first be converted into binary values
  - Conversion subroutine must also perform error checking for invalid text data characters (Skip)

Subroutines with Local Variables in Stack (cont'd.)

- Bubble sort: simple, widely known, but inefficient, sorting method
  - Basic underlying idea: go through the array or file sequentially several times; each iteration consists of comparing each element in the array or file with its successor (x[i] with x[i+1]) and interchanging them if they are not in proper order (either ascending or descending)

Bubble Sort (cont'd.)

Each number bubbles up to its proper position with each successive iteration until entire array is sorted

Iteration	0 (original array)	157 13 35 9 98 810 120 54 10 30
1		13 35 9 98 157 120 54 10 30 810
2		13 9 35 98 120 54 10 30 157 810
3		9 13 35 98 54 10 30 120 157 810
4		9 13 35 54 10 30 98 120 157 810
5		9 13 35 10 30 54 98 120 157 810
6		9 13 10 30 35 54 98 120 157 810
7		9 10 13 30 35 54 98 120 157 810
8		9 10 13 30 35 54 98 120 157 810
9		9 10 13 30 35 54 98 120 157 810

Bubble Sort (Improved)

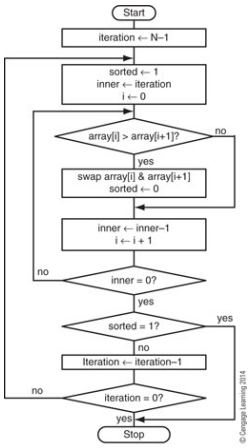


Figure 5.9 ■ Logic flow of bubble sort

Example :
eg05_08.asm

Bubble sort subroutine

Example

E5.8 Write a subroutine that tests whether a number has a specified property.
The square of the sum of the upper half and the lower half of the given number is equal to the original number. The number to be tested is in r16-r17, and the results (0 or 1) is returned in register r22.

Example

; The sumSqEqu subroutine tests whether the square of the sum of the upper and lower half equals the original number. The number is passed in r16~r17 and the result is returned in r22.

```
sumSqEqu:push   r17          ; save the number to be tested
         push   r16
         ldi    r18,100      ; separate the upper and lower two digits
         clr    r19          ; by dividing the number by 100
         call   div16U       ;   "
         add    r22,r24      ; add the upper and lower halves
         mul    r22,r22      ; compute the square of the sum
         pop    r16          ; get back the number to be tested
         pop    r17          ;   "
         cp     r0,r16       ;
         cpc    r1,r17
         brne   false
         ldi    r22,1
         ret
false:   clr    r22
         ret
         .include "div16U.asm"
//end of program
```

Example

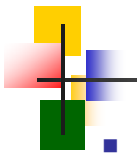
The subroutine divides the 16-bit unsigned numbers contained in r18~r19 into the 16-bit unsigned number contained in r16~r17 and returns the quotient in r25~r24 and the remainder in r23~r22.

```
         .def   dlpcnt = r20
         .def   tmpL = r0
         .def   tmpH = r1
div16U:  ldi    dlpcnt,16    ; set up divide iteration count
         movw   r24,r16      ; place dividend in Q register
         clr    r23          ; load 0 in R register
         clr    r22          ;   "
dvloop:  lsl    r24          ; shift R:Q to the left one place
         rol    r25          ;   "
         rol    r22          ;   "
         rol    r23          ;   "
         movw   tmpL,r22     ; transfer R to temporary registers
         sub    tmpL,r18     ; compute R - S
         sbc    tmpH,r19     ;   "
         brmi   less         ; branch if divisor is larger
         movw   r22,tmpL     ; save the difference in R
         ori    r24,0x01     ; set the lsb of Q to 1
         rjmp   nxtb
less:    andi   r24,0xFE     ; set the lsb of Q to 0
nxtb:    dec    dlpcnt
         brne   dvloop
         ret
```

Part II.4 C Language Programming

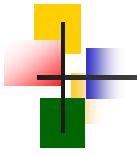
Outline

i. Introduction to C
ii. Types, Operators, and Expressions
iii. Control Flow
iv. Input and Output
v. Functions and Program Structure
vi. Pointers, Arrays, Structures, Unions, and Type Definition
vii. Using the AVR Studio IDE to Develop C Programs (Lab1 & Lab2)



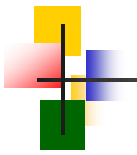
Introduction to C

- Review of the C language
- Introduction to using the C language to program the AVR peripheral functions
  - C language is quickly replacing assembly language in every embedded application
    - Improved programming productivity: allows the user to work on program logic at a level much higher than that of the assembly language



Introduction to C: Example

```
(1)   #include <avr/io.h>
(2)   #include <util/delay.h>
(3)   int main(void)              //program begins
(4)   {
(5)   DDRB = 1<<0;                //configure PORTB
(6)
(7)   while(1)                    //stay in an infinite loop
(8)   {
(9)   PORTB = 0x00;               //output to PORTB
(10)  _delay_ms(500);             //Delay 500ms
(11)  PORTB = 1<<0;               //output to PORTB
(12)  _delay_ms(500);             //delay 500ms
(13)  }
(14)  }
```



Types, Operators, and Expressions

- Variable naming conventions:
  - Start with a letter or underscore character followed by zero or more letters, digits, or underscore characters
  - Cannot contain arithmetic signs, dots, apostrophes, C keywords, or special symbols
  - Underscore not advisable as first character; uppercase and lowercase letters are distinct
- Note:
  - Variable must be declared before they can be used.

Types, Operators, and Expressions (cont'd.)

- Data types: void, char, int, float (32bit), and double (64 bit)
- Variable declarations: type and a list of one or more variables of that type
  - Int i,j,k;
- Constants: characters, integers, floating-point numbers, and strings
  - Constants int c=0;

Types, Operators, and Expressions (cont'd.)

- Arithmetic operators

Symbol	Operation	Example
+	Add and unary plus	3 + 5; +4
–	Subtract and unary minus	60 – 13; –10
*	Multiply	a * b
/	Divide	100 / 13
%	Modulus	300 % 19
++	Increment	++y; y++
––	Decrement	––ptr; ptr––

Table 6.3 Arithmetic operators in C

Example:

```
x= ++y;
K=5+j++;
```

Types, Operators, and Expressions (cont'd.)

- Bitwise operators: & AND; | OR; ^ XOR; ~ NOT; >> right shift; << left shift
- Example:
  - P1 = P1 & 0x40; P1 & = 0x40; //P1 8bit
  - P2 = P2 | 0xFE; P2 | = 0xFE; // P2 is 8 bit
  - P3 = P3 ^ 0xFF; P3 ^ = 0xFF;
  - P4 = ~ P4; P4 ~ =P4;
  - P5 = P5 >> 2; P5 >> =2;
  - P6 = P6 << 1; P6 << =1;

Relational and Logical Operators (cont'd.)

Relational and logical operators: used in expressions to compare the values of two operands

Symbol	Operation	Example
==	equal to	ax == bx
!=	not equal to	ax != 10
>	greater than	ax >60
>=	greater than or equal to	ax >= 20
<	less than	ax <40
<=	less than or equal to	ax <= 80
&&	and	(ax > 3) && (bx < 10)
||	or	(ax == 0) || (bx !=3)
!	not (one's complement)	!ax

Table 6.5 Relational and logical operators in C

Example:

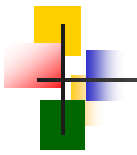
```
if(!(PORTA & 0x80))
Statement;
```

Types, Operators, and Expressions (cont'd.)

- Precedence of operators: order in which operators are processed
  - Operators at the same level are evaluated from left to right

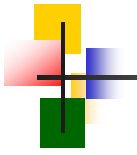
Precedence	Operator	Associativity
Highest	() [] → .	left to right
	! ~ ++ -- - * & (type) sizeof	right to left
	* / %	left to right
	+ -	left to right
	<< >>	left to right
	< <= > >=	left to right
	== !=	left to right
	&	left to right
	^	left to right
	|	left to right
	&&	left to right
	||	left to right
	?:	right to left
	= += -= *= /= %= &= ^= |= <<= >>=	right to left
Lowest	,	left to right

Table 6.6 ■ Table of precedence of operators



Control Flow

- If statement: statement associated with the *if statement* executed based upon the outcome of a condition
- If-else statement: one statement executed if a condition is nonzero and a different statement if the condition is zero
- Multiway conditional statement: cascaded series of if-else statements



Control Flow: Example

- if (expression)
  - Statement1
- else
  - Statement 2

- if (k>0) return 1;
- else if(k==0) return 0;
- else return -1;

Control Flow (cont'd.)

- Switch statement: multiway decision based on the value of a control expression
- For-loop statement
  - Syntax: for (expr1; expr2; expr3)
    statement;

  Where expr1 and expr3 are assignments or function calls, and expr2 is a relational expression

Control Flow: Example

- Switch (index){
  - Case 10:
    - Out=100;
  - Case 20:
    - Out=200;
  - Case 30:
    - Out=400;
  - default:
    - Out=0;
- }

- sum=0;
- for(i=1;i<100;i++)
  - sum=sum+i;

Control Flow (cont'd.)

- While statement: value of the expression is checked prior to each execution of the statement within the loop
- Do-while statement: body statements execute at least once; tests the termination condition at the end of the statement
- Goto statement: transfers control to a labeled statement

Control Flow: Example

- while (expression)
  - statement
  - How about this one?
- while(1)
- do
  - statement
- while(expression)

Input and Output

- Input and output functions
  - int getchar ( ): returns a character when it is called
    - char abc;
    - abc = getchar();
  - int putchar (int): outputs a character on the standard output device
    - putchar('d');
  - int puts (const char *s): outputs the string pointed to by s on the standard output device
    - putchar("dddddere\n");

Input and Output Functions (cont'd.)

- int printf (formatting string, arg_1, arg_2, …, arg_n):
  - Converts, formats, and prints its arguments on the standard output under the control of a formatting string
  - arg_1, arg_2, …, arg_n represent individual output data items

- printf("%d,%2.1d, %3.2f, %i\n",x1,x2,x3,x4);

Conversion character	Meaning
c	Data item is displayed as a single character .
d	Data item is displayed as a signed decimal number .
e	Data item is displayed as a floating-point value with an exponent.
f	Data item is displayed as a floating-point value without an exponent.
g	Data item is displayed as a floating-point value using either e-type or f-type conversion, depending on value; trailing zeros, trailing decimal point will not be displayed.
i	Data item is displayed as a signed decimal integer.
o	Data item is displayed as an octal integer ,without a leading zero.
s	Data item is displayed as a string.
u	Data item is displayed as an unsigned decimal integer.
x	Data item is displayed as a hexadecimal integer, without the leading 0x.

Table 6.8 Commonly used conversion characters for data output

Functions and Program Structure

- Syntax of a function definition

```
return_type function_name (declarations of arguments)
{
    declarations and statements
}
```

- Function prototype: used to declare a function

```
return_type function_name (declarations of arguments)
```

- Note:
  - A function can not be called before it has been defined. This dilemma is solved by using the function prototype statement.
    - int sqroot(unsigned int y); //before main( ) is a function prototype statement.

Functions and Program Structure (cont'd.)

- Writing a C program with multiple functions
  - Calling a function: write the name of the function and replace the argument declarations by actual arguments or values
  - Example

```
#include <avr\io.h>
int sqroot(unsigned int y);
void main(void)
{
int k=100;
sqroot(k);
}
int sqroot(unsigned int y);
{
}
```

Pointers, Arrays, Structures, Unions, and Type Definition

- Pointers and addresses
  - Pointer: holds the address of another variable
- Example show how to declare pointer and how to use it
  - char cx, cy; //define cx and cy of char type
  - char *cp; //define cp pointer of a character
  - cp = &cx; //assigns the address of the variable cx to cp
  - cy = *cp; //cy gets the value of cp

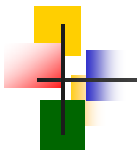
Pointers, Arrays, Structures, Unions, and Type Definition (cont'd.)

- Arrays: consist of multiple elements of the same type and same storage class
  - Pointers and arrays
    - Array subscripting operations can also be done with pointers
  - Passing arrays to a function
    - Array name used as an argument to pass an array to a function
  - Initializing arrays
    - Syntax for initializing an array:
      array_declarator = { value-list };

Pointers, Arrays, Structures, Unions, and Type Definition

- Pointer & arrays
  - int x, a[10];
  - int *ap;
  - ap = &a[0];
  - x = *ap;
- /*ap is the address of a[0] and x is the content of a[0]*/

- Initializing array
  - char grade[4] = {'A','B','C','D'};
  - int level[5] = {1,2,3,4,5};
  - cmyaddress[] ="1360 barrington street";
  - char *myaddress ="1360 barrington street";

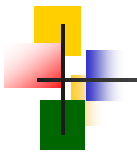


Pointers, Arrays, Structures, Unions, and Type Definition (cont'd.)

- Structures: group of related variables that can be accessed through a common name
  - Each item within a structure has its own data type
    - struct point {
      - int x;
      - int y;
      - int z;
      - };

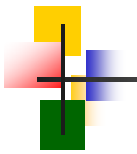
Pointers, Arrays, Structures, Unions, and Type Definition (cont'd.)

- Unions: variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements
  - union point {
    - int x;
    - int y;
    - int z;
    - int color;
    - int thinkness;
    - }



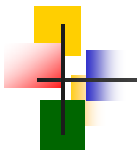
Pointers, Arrays, Structures, Unions, and Type Definition (cont'd.)

- The typedef statement: provides a capability that enables assigning an alternate name to a data type
- example
  - typedef char letter;
  - letter ax,bx,cx;



Pointers, Arrays, Structures, Unions, and Type Definition (cont'd.)

- Enumerated data types: defines a variable and specifies the valid values that can be stored into that variable
- example
  - enum level(high, middle, low);
- To declare an enumerate type:
  - enum level mylevel, hislevel, herlevel;
  -



Miscellaneous Items

- Automatic/external/static/volatile
  - Automatic variables: come into existence when the function is entered and disappear when it is left
  - External variables: defined outside of any function and thus are available potentially to many functions

Miscellaneous Items (cont'd.)

  - Static: when used with a local variable declaration inside a block or a function, causes the variable to maintain its value between entrances to the block or function
  - Volatile variable: value can be changed by something other than user code, particularly related to PORT definitions
- Scope rules
  - The scope of a name is the part of the program within which the name can be used

Miscellaneous Items (cont'd.)

- Type casting: causes the program to treat a variable of one type as though it contains data of another type
  - int a,b,c,d;
  - double e,f;
  - e=((double)a)*((double)c);

Miscellaneous Items (cont'd.)

- Pointer to functions
  Example: int (*funcPtr) (int kx);
  - Variable funcPtr of type "pointer to function that returns an int and that takes one int argument"

funcPtr = primeTest; // assign function name to pointer

The C Preprocessor

- Preprocesor
  - Part of C compilation process
  - Recognizes and analyzes special statements that begin with a pound sign, #, in first nonspace character on a line
- The #define statement: used to assign symbolic names to program constants
  - #define TOP 1000
  - #define not !
  - #define xor ^
- Can be used to define a macro definition
  - #define CUBIC(x) x*x*x;
  - y = CUBIC(x);

The C Preprocessor (cont'd.)

- The ## operator: used in macro definitions to join two tokens (a token can be a character or a string)
- the #include statement
  - #include <avr\io.h>

The C Preprocessor (cont'd.)

- Conditional compilation: often used to create one program that can be compiled to run on different computer systems
  - The #IFDEF, #ENDIF,#ELSE,#IFNDEF,#IF, and #ELIF statements
    - #if defined(YY) && YY
    - #define XX 24
    - #else
    - #define XX 23
    - #endif

Using the AVR Studio C Compiler

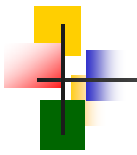
- ATMEL AVR module types: AVR CPU core, SRAM, Flash, EEPROM, I/O ports, and a number of peripheral modules
- AVR peripheral register naming convention
  - Peripheral register categories: control, status, and data registers; names reflect category, e.g., CTRL and STATUS
    - counter register is names CNT
    - two bytes are named CNTL, CNTH

Using the AVR Studio C Compiler (cont'd.)

- AVR peripheral register bit naming convention
  - Bits of a group will always have a number suffix
  - Bits not part of a bit group will never have a number suffix
    - Example
      - Timer/counter register D-CTRLD:
      - EVACT2,EVACT1,EVACT0, EVDLY, EVSEL3,EVSEL2,EVSEL1,EVSEL0

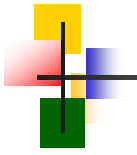
Using the AVR Studio C Compiler (cont'd.)

- Accessing AVR Mega device peripheral registers in C: gcc compiler uses #define statement to associate every peripheral register with its I/O address or memory address#
- example: definition for SPCR register:
  - #define SPIE 7 //bit 7
  - #define DORD 5//bit5
  - #define CPOL 3//bit3
  - #define SPR0 1//bit 1



Using the AVR Studio IDE to Develop C Programs

- Step 1: Create a new project to manage the programming task
- Step 2: Enter C functions in file(s)
- Step 3: Add C files into the project
- Step 4: Compile (build) project; eliminate syntax/semantic errors
- Step 5: Execute and debug the project
- Example



Multiple-File Project

- Reasons for using multiple files
  - Code reuse to reduce application development time
  - Divide and conquer: allows a complex project to be split into several smaller and manageable subprojects