About Final Exam

- Duration: 2 hours
- Close book exam
  - Instruction set and ASCII table provided
  - Materials of weeks 1-12
- Three sections
  - Multiple choices
  - Small questions
  - Medium-fairly large questions

About Final Exam (cont.)

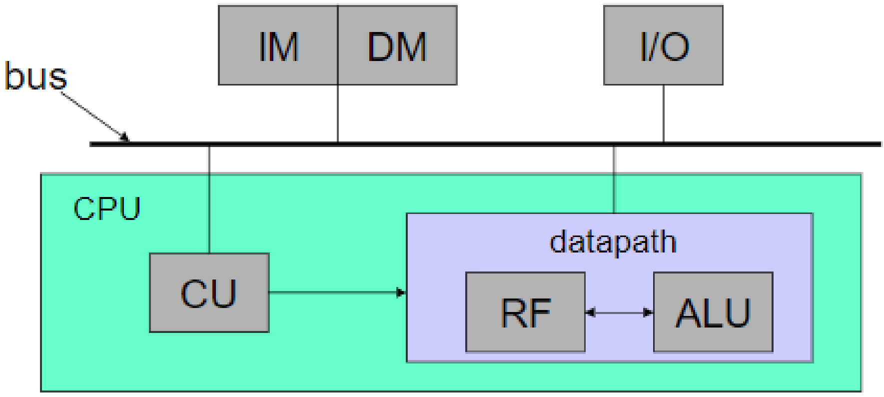
- Things will not be tested
  - Predefined labels for I/O registers
    - Register names
    - Bit names and the bit values for setting a peripheral
  - Using comment lines for any information you need to know related to an **I/O setting**. For example,

    ; insert code here to modify the control register of Timer0 to set up the timer for 1 second duration

    ; insert code here to enable timer0 overflow interrupt

Week 1

Fundamental Hardware Components in Computing System



- **ALU**: Arithmetic and Logic Unit
- **RF**: Register File (a set of registers)
- **CU**: Control Unit
- **IM/DM**: Instruction/Data Memory
- **I/O**: Input/Output Devices

Exercises

- Represent the following decimal numbers using 8-bit 2's complement format
  (a) 7
  (b) 127
  (c) -12
- Can all the above numbers be represented by 4 bits?
- An *n*-bit binary number can be interpreted in two different ways: signed or unsigned. What decimal value does the 4-bit number, 1011, represent for the following two cases?
  (a) if it is a signed number
  (b) if it is an unsigned number

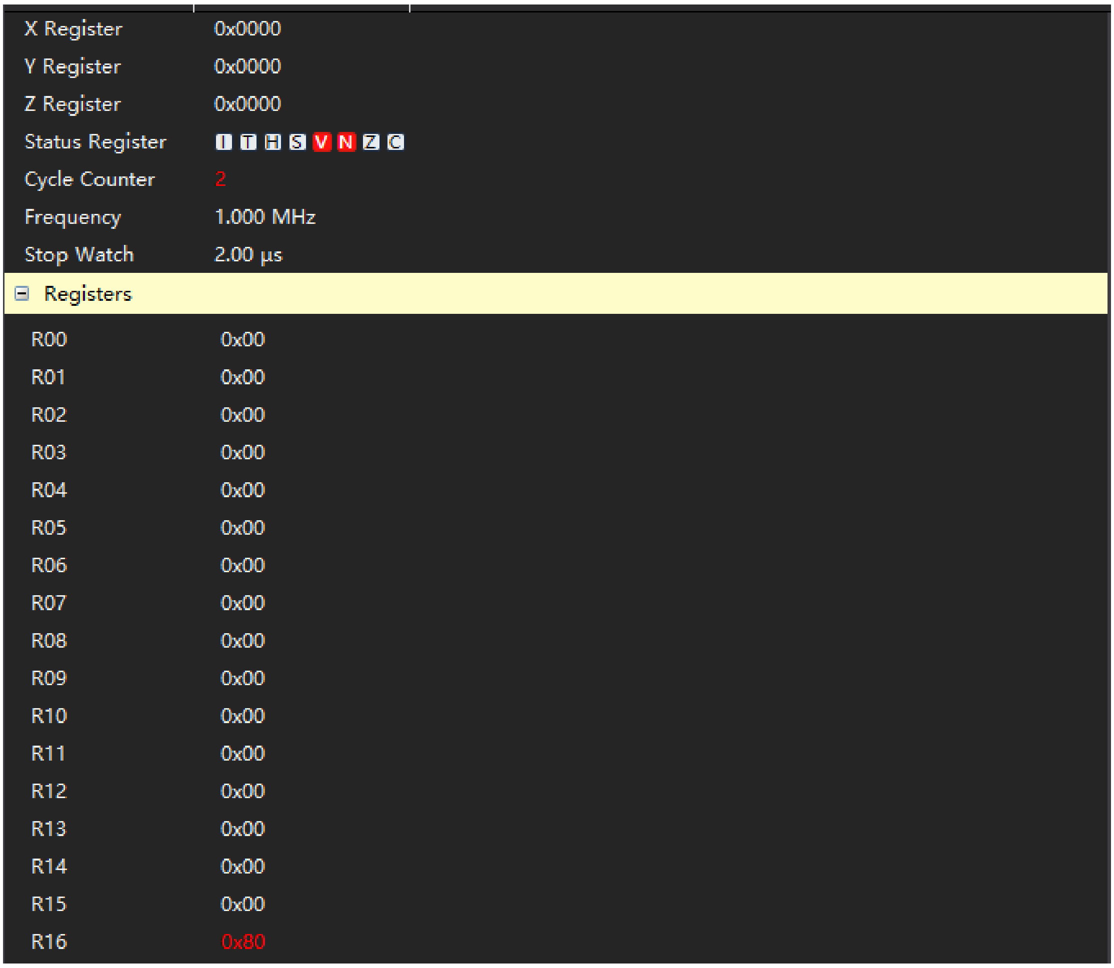
- Overflow happens when the result cannot be represented by the given number of bits.
- Assume *a*, *b* are **positive numbers** in the *n*-bit 2's complement system,
  - For *a+b*
    - If the MSB of *a+b* is *1*, which indicates a negative number; then the addition causes a **positive overflow**.
  - For *-a-b*
    - If the MSB of *-a-b* is *0*, which indicates a positive number; then the addition causes a **negative overflow**.

Registers

- Two types
  - General purpose
  - Special purpose
    - e.g.
      - Program Counter (PC)
      - Status Register
      - Stack Pointer (SP)
      - Input/Output Registers
  - Stack Pointer and Input/Output Registers will be discussed in detail later.

Status Register

- Contains a number of bits with each bit being associated with processor (CPU) operations
- Typical status bits
  - V: Overflow
  - C: Carry
  - Z: Zero
  - N: Negative
- Used for controlling the program execution flow



Endianness

- Memory objects
  - Memory objects are basic entities that can be accessed as a function of the **address** and the **length**
    - E.g. bytes, words, longwords
- For large objects (multiple bytes), there are two byte-ordering conventions
  - **Little endian –** little end (least significant byte) stored first (i.e. at the lowest address)
    - Intel microprocessors (Pentium etc)
  - **Big endian –** big end (most significant byte) stored first
    - SPARC, Motorola microprocessors

Big Endian & Little Endian

- Example: 0x12345678—a long word of 4 bytes. It is stored in the memory at address 0x00000100
  - big endian:

Address	data
0x00000100	0x12
0x00000101	0x34
0x00000102	0x56
0x00000103	0x78

  - little endian:

Address	data
0x00000100	0x78
0x00000101	0x56
0x00000102	0x34
0x00000103	0x12

- To base 2
  - To convert $(11.25)_{10}$ to binary
    - For whole number $(11)_{10}$ – repeated division **(by 2)**

11	1
5	1
2	0
1	1
0	

    - For fraction $(0.25)_{10}$ – repeated multiplication **(by 2)**

0.25	
0.5	0
0.0	1

$(11.25)_{10}=(1011.01)_2$

- To base 16
  - To convert $(99.25)_{10}$ to hexadecimal
    - For whole number $(99)_{10}$ – division **(by 16)**

99	3
6	6
0	

    - For fraction $(0.25)_{10}$ – multiplication **(by 16)**

0.25	
0.0	4

$(99.25)_{10}=(63.4)_{hex}$

1. Find the two's complement binary code for the following decimal numbers:

(a) 26

(b) -26

2. Find the binary code words for the following hexadecimal numbers:

(c) C0FFEE

(d) F00D

3. Prove that the two's-complement overflow cannot occur when two numbers of different signs are added.

Week2

AVR Registers

General purpose registers

- 32 8-bit registers, R0 ~ R31 or r0 ~ r31
- Can be further divided into two groups
  - First half group (R0 ~ R15) and second half group (R16 ~ R31)
  - Some instructions work only on the second half group R16~R31
    - Due to the limitation of instruction encoding bits
      - » Will be covered later
    - E.g. *ldi rd, #number* ;rd ∈ R16~R31

AVR Registers (cont.)

- General purpose registers
  - The following register pairs can work together as address registers (or address pointers)
    - X, R27:R26
    - Y, R29:R28
    - Z, R31:R30
  - The following registers can be applied for specific purpose
    - R1:R0 store the result of the multiplication instruction
    - R0 stores the data loaded from the program memory

	Syntax:	Operands:	Program Counter:
(i)	LPM	None, R0 implied	PC ← PC + 1
(ii)	LPM Rd, Z	$0 \leq d \leq 31$	PC ← PC + 1
(iii)	LPM Rd, Z+	$0 \leq d \leq 31$	PC ← PC + 1

AVR Registers (cont.)

- I/O registers
  - 64+ 8-bit registers
    - Their names are defined in the *m2560def.inc* file
  - Used in input/output operations
    - Mainly storing data/addresses and control signal bits
  - Will be covered in detail later
- Status register (SREG)
  - A special I/O register

SREG

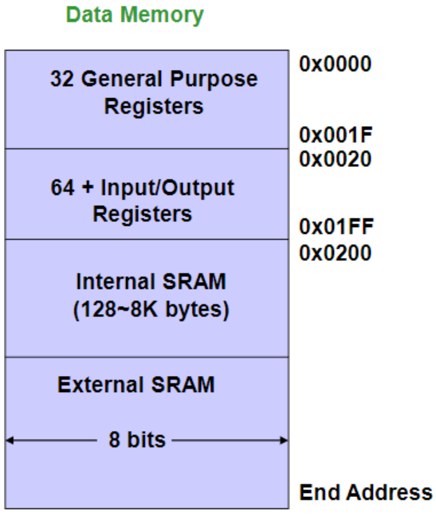
- The Status Register (SREG) contains information about the result of the most recently executed AL instruction. This information can be used for altering program flow in order to perform conditional operations.
- SREG is updated by hardware after an AL operation.
  - Some instructions such as load do not affect SREG.
- SREG is not automatically saved when entering an interrupt routine and restored when returning from an interrupt. This must be handled by software.
  - Using in/out instruction to store/restore SREG
  - To be covered later

AVR Address Spaces

- Three address spaces
  - Data memory
    - Storing data to be processed
  - Program memory
    - Storing program code and constants
  - EEPROM memory
    - Large permanent data storage

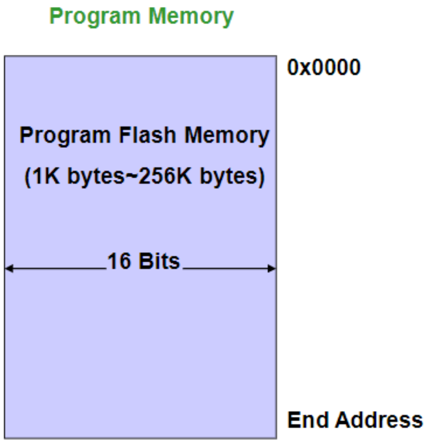
Data Memory Space

- Covers
  - Register file
    - i.e. registers in the register file also have memory addresses
  - I/O registers
    - I/O registers have two versions of addresses
      - I/O addresses
      - Memory addresses
  - SRAM data memory
    - The highest memory location is defined as RAMEND



Program Memory Space

- Covers
  - 16 bit flash memory
    - Mainly for read only
      - Instructions are retained when power off
  - Can be accessed with special instructions
    - LPM
    - SPM



Compare

- Syntax: *cp Rd, Rr*
- Operands: Rd $\in$ {r0, r1, …, r31}
- Operation: Rd - Rr (Rd is not changed)
- Flags affected: H, S, V, N, Z, C
- Words: 1
- Cycles: 1
- Example:

```
        cp r4, r5           ; Compare r4 with r5
        brne noteq          ; Branch if r4 ≠ r5
        ...
noteq: nop                  ; Branch destination (do nothing)
```

Conditional Branch

- Syntax: ***brge k***
- Operands: $-64 \le k < 64$
- Operation: If Rd≥Rr ($N \oplus V=0$) then PC←PC+k+1,
  else PC ← PC+1 if condition is false
- Flag affected: None
- Words: 1
- Cycles: 1 if condition is false; 2 if condition is true

Selection (2/2)

- IF-THEN-ELSE control structure

```
if(a<0)
        b=1;
else
        b=-1;
```

  – Numbers a, b are 8-bit signed integers and stored in registers. You need to decide which registers to use.

```
.def    a=r16
.def    b=r17
        cpi     a, 0        ;a-0
        brge    ELSE        ;if a≥0, go to ELSE
        ldi     b, 1        ;b=1
        rjmp    END         ;end of IF statement
ELSE:   ldi     b, -1       ;b=-1
END:    ...
```

Iteration (1/2)

- WHILE loop

```
sum =0;
i=1;
while (i<=n){
        sum += i*i;
        i++;
}
```

  – Numbers *i, sum* are 8-bit unsigned integers and stored in registers. You need to decide which registers to use.

Iteration (2/2)

- WHILE loop

```
.def    i = r16
.def    n = r17
.def    sum = r18

        ldi  i, 1                ;initialization
        clr  sum
loop:
        cp  n, i
        brlo end
        mul  i, i
        add  sum, r0
        inc  i
        rjmp  loop
end:
        rjmp  end
```

1. Refer to the AVR Instruction Set document (available at http://www.cse.unsw.edu.au/~cs9032, under the link References → Documents → AVR-Instruction-Set.pdf).

   Study the following instructions:

   – Arithmetic and logic instructions
     - add, adc, adiw, sub, subi, sbc, sbci, sbiw, mul, muls, mulsu
     - and, andi, or, ori, eor
     - com, neg

Homework

1. Refer to the AVR Instruction Set document (available at http://www.cse.unsw.edu.au/~cs9032, under the link References → Documents → AVR-Instruction-Set.pdf).

   Study the following instructions:

   – Arithmetic and logic instructions
     - add, adc, adiw, sub, subi, sbc, sbci, sbiw, mul, muls, mulsu
     - and, andi, or, ori, eor
     - com, neg

Homework

2. Write the assembly code for the following functions
   1) 2-byte addition (i.e, addition on 16-bit numbers)
   2) 2-byte signed subtraction
   3) 2-byte signed multiplication

```
   xxxxxxxx xxxxxxxx 00000000 00000000
  --------xxxxxxxx xxxxxxxx 00000000
  --------xxxxxxxx xxxxxxxx 00000000
+ 00000000 00000000 xxxxxxxx xxxxxxxx
-------------------------------------
x xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
     r19      r18      r17      r16
```

```
               ah       al
x              bh       bl
--------------------------
          (ah*bl) (al*bl)
(ah*bh) (al*bh)
--------------------------
                  r17:r16
          r18:r17
          r18:r17
r19:r18
```

```
muls16x16:
    clr    r2
    muls   r23,  r21              ; (signed) ah * (signed) bh
    movw   r19 : r18,  r1 : r0
    mul    r22,  r20              ; (unsigned) al * (unsigned) bl
    movw   r17 : r16,  r1: r0
    mulsu  r23,  r20              ; (signed) ah * (unsigned) bl
    sbc    r19,  r2               ;Trick here (Hint: what does the carry mean here?)
    add    r17,  r0
    adc    r18,  r1
    adc    r19,  r2
    mulsu  r21,  r22              ; (signed) bh * (unsigned) al
    sbc    r19,  r2               ; Trick here
    add    r17,  r0
    adc    r18,  r1
    adc    r19,  r2
```

Week3

- An assembly program basically consists of
  - Assembler directives
    - E.g. ***.def temp = r15***
  - Executable instructions
    - E.g. ***add r1, r2***
- An input line in an assembly program takes one of the following forms :
  - [label:] directive [operands] [comment]
  - [label:] instruction [operands] [comment]
  - Comment
  - Empty line

  Note: [ ] indicates optional

Example

```
; The program performs
; 2-byte addition: sum=a+b;

      .def   a_high = r2;
      .def   a_low = r1;
      .def   b_high = r4;
      .def   b_low = r3;
      .def   sum_high = r6;
      .def   sum_low = r5;

      mov   sum_low, a_low
      mov   sum_high, a_high
      add   sum_low, b_low
      adc   sum_high, b_high
end:  rjmp end
```

Two comment lines

Empty line

Six assembler directives

Five executable instructions

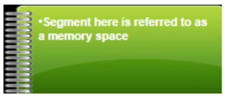
Typical AVR Assembler directives

Directive	Description
BYTE	Reserve byte to a variable
CSEG	Code Segment
DB	Define constant byte(s)
DEF	Define a symbolic name on a register
DEVICE	Define which device to assemble for
DSEG	Data Segment
DW	Define constant word(s)
ENDMACRO	End macro
EQU	Set a symbol equal to an expression
ESEG	EEPROM Segment
EXIT	Exit from file
INCLUDE	Read source from another file
LIST	Turn listfile generation on
LISTMAC	Turn macro expansion on
MACRO	Begin macro
NOLIST	Turn listfile generation off
ORG	Set program origin
SET	Set a symbol to an expression

NOTE: All directives must be preceded by **a period, '.'**

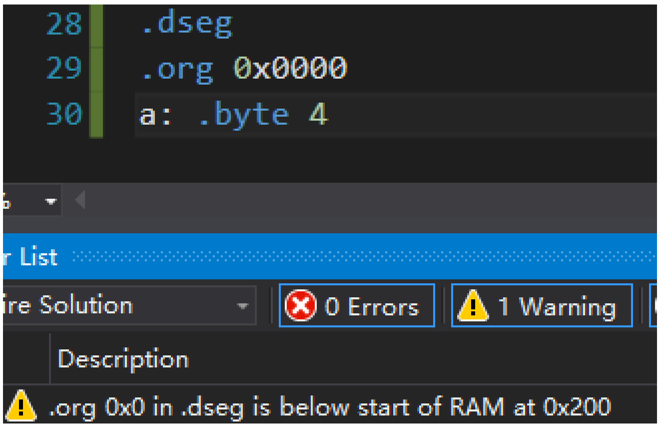
Program/Data Memory Organization

- AVR has three different memories
  - Data memory
  - Program memory
  - EPROM memory
- The three memories are corresponding to three memory segments to the assembler:
  - Data segment
  - Program segment (or Code segment)
  - EEPROM segment



Program/Data Memory Organization Directives

- Memory segment directive specifies which physical memory to use
  - **.dseg**
    - Data memory
  - **.cseg**
    - Code/Program memory
  - **.eseg**
    - EPROM memory
- The default segment is cseg
- The **.org** directive specifies the start address for the related code/data to be saved



Operators in Assembler Expression

Same meanings as in C

Symbol	Description
!	Logical Not
~	Bitwise Not
-	Unary Minus
*	Multiplication
/	Division
+	Addition
-	Subtraction
<<	Shift left
>>	Shift right
<	Less than
<=	Less than or equal
>	Greater than
>=	Greater than or equal
==	Equal
!=	Not equal
&	Bitwise And
^	Bitwise Xor
|	Bitwise Or
&&	Logical And
||	Logical Or

Directives for Macro

- .macro
  - Tells the assembler that this is the start of a macro
  - Takes the macro name and (implicitly) parameters
    - Up to 10 parameters
      - Which are referenced by @0, …@9 in the macro definition body
- .endmacro
  - Specifies the end of a macro definition.

Macro (cont.)

- Macro definition structure:

```
.macro macro_name
        ; macro body
.endmacro
```

- Usage

```
macro_name   [para0, para1, ...,para9]
```

- Swapping any two memory data

```
.macro swap2
        lds r2, @0          ; load data from provided
        lds r3, @1          ; two locations
        sts @1, r2          ; interchange the data and
        sts @0, r3          ; store data back
.endmacro

swap2 a, b                  ; a is @0, b is @1.
swap2 c, d                  ; c is @0, d is @1.
```

- Register bit copy
  - copy a bit from one register to a bit of another register

```
; Copy bit @1 of register @0
; to bit @3 of register @2

.macro  bitcopy
        bst @0, @1
        bld @2, @3
.endmacro

bitcopy r4, 2, r5, 3
bitcopy r5, 4, r7, 6
```

Functions in Assembler Expression

- LOW(expression)
  - Returns the low byte of an expression
- HIGH(expression)
  - Returns the second (low) byte of an expression
- BYTE2(expression)
  - The same function as HIGH
- BYTE3(expression)
  - Returns the third byte of an expression
- BYTE4(expression)
  - Returns the fourth byte of an expression
- LWRD(expression)
  - Returns low word (bits 0-15) of an expression
- HWRD(expression):
  - Returns bits 16-31 of an expression
- PAGE(expression):
  - Returns bits 16-21 of an expression
- EXP2(expression):
  - Returns 2 to the power of expression
- LOG2(expression):
  - Returns the integer part of log2(expression)

Example 1

- Translate the following C variables. Assume each integer takes four bytes.

```
int  a;
unsigned int  b;
char  c;
char*  d;
```

Example 1: Solution

- Translate the following variables. Assume each integer takes four bytes.

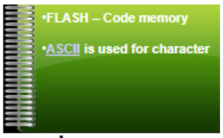
```
.dseg                ; in data memory

.org 0x200           ; start from address 0x200

a:   .byte  4        ; 4 byte integer
b:   .byte  4        ; 4 byte unsigned integer
c:   .byte  1        ; 1 character
d:   .byte  2        ; address pointing to the string
```

  - All variables are allocated in data memory (SRAM)
  - Labels are given the same name as the variable for convenience and readability.

Example 2



- Translate the following C constants and variables.

C code:

```
int  a;
const char b[ ] = "COMP9032";
const int c = 9032;
```

Assembly code:

```
.dseg
a:  .byte 4

.cseg
;b:  .db  'C', 'O', 'M', 'P', '9', '0', '3', '2', 0
b:  .db  "COMP9032", 0
c:  .dw  9032
```

  - All variables are in SRAM and constants are in FLASH

Example 2 (cont.)

- Program memory mapping
  - In the program memory, data are packed in words. If only a single byte left, that byte is stored in the first (left) byte and the second (right) byte is filled with 0, as highlighted in the example.

0x0000	'C'	'O'
0x0001	'M'	'P'
0x0002	'9'	'0'
0x0003	'3'	'2'
0x0004	0	0
0x0005	0x48	0x23

Hex values

43	4F
4D	50
39	30
33	32
0	0
48	23

Example 4

- Translate variables with structured data type
  - with initialization

```
struct STUDENT_RECORD
{
        int student_ID;
        char name[20];
        char WAM;
};

typedef struct STUDENT_RECORD student;

struct student s1 = {123456, "John Smith", 75};
struct student s2;
```

Example 4: Solution

- Translate variables with structured data type

```
.set    student_ID=0
.set    name = student_ID+4
.set    WAM = name + 20
.set    STUDENT_RECORD_SIZE = WAM + 1

.cseg
s1_value:  .dw   LWRD(123456)
           .dw   HWRD(123456)
           .db   "John Smith     ", 0
           .db   75
.dseg
s1:     .byte   STUDENT_RECORD_SIZE
s2:     .byte   STUDENT_RECORD_SIZE

; copy the data from instruction memory to s1
...
```

Memory Access Operations

- Access to data memory
  - Using instructions
    - ld, lds, st, sts
- Access to program memory
  - Using instructions
    - lpm
    - spm
      - Not covered in this course
  - Most of time, that we access the program memory is to load data

Load Program Memory Instruction

- Syntax: *lpm Rd, Z*
- Operands: Rd∈{r0, r1, …, r31}
- Operation: Rd ← (Z)

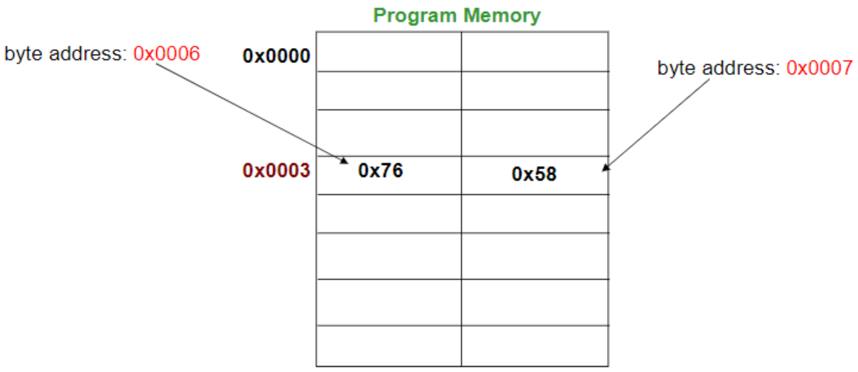
- Words: 1
- Cycles: 3

Load Data From Program Memory

- The address label in the program memory is ***word address***
  - Used by the PC register
- To access constant data in the program memory with *lpm*, ***byte address*** should be used.
- Address register, Z, is used to point bytes in the program memory

Byte Address vs Word Address

- First-byte-address (in a word) = 2 * word-address
- Second-byte-address (in a word) = 2 * word-address +1




```
.include "m2560def.inc"   ; include definition for Z

ldi ZH, high(Table_1<<1)  ; initialize Z
ldi ZL, low(Table_1<<1)

lpm r16, Z                ; load constant from the program
                          ; memory pointed to by Z (r31:r30)

table_1:
        .dw 0x5876        ; 0x76 is the value when Z_LSB = 0
                          ; 0x58 is the value when Z_LSB = 1
```

Complete Example 1 (cont.)

- C description

```
struct STUDENT_RECORD
{
        int student_ID;
        char name[20];
        char WAM;
};

typedef struct STUDENT_RECORD student;

student s1 = {123456, “John Smith”, 75};
```

Complete Example 1 (cont.)

- Assembly translation

```
.set    student_ID=0
.set    name = student_ID+4
.set    WAM = name + 20
.set    STUDENT_RECORD_SIZE = WAM + 1

.cseg
start:  ldi zh, high(s1_value<<1)       ; pointer to student record
        ldi zl, low(s1_value<<1)        ; value in the program memory

        ldi yh, high(s1)                ; pointer to student record holder
        ldi yl, low(s1)                 ; in the data memory

        clr r16
```

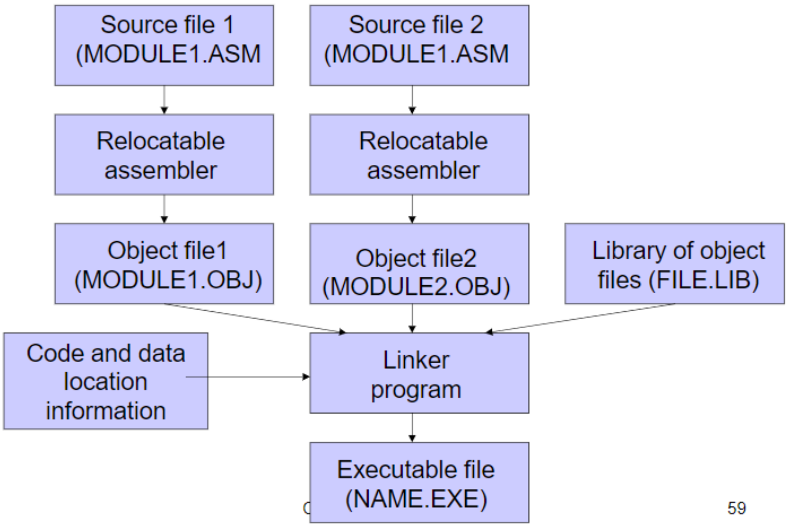
Complete Example 1 (cont.)

- Assembly translation (cont.)

```
load:
                cpi r16, STUDENT_RECORD_SIZE
                brge end
                lpm r10, z+
                st  y+, r10
                inc r16
                rjmp load
end:
                rjmp end

s1_value:       .dw     LWRD(123456)
                .dw     HWRD(123456)
                .db     "John Smith      ", 0
                .db     75
.dseg
.org 0x200
s1:     .byte   STUDENT_RECORD_SIZE
```

Relocatable Assembly - workflow



59

1. Refer to the AVR Instruction Set manual, study the following instructions:
   - Arithmetic and logic instructions
     - clr
     - inc, dec
   - Data transfer instructions
     - movw
     - sts, lds
     - lpm
     - bst, bld
   - Program control
     - jmp
     - sbrs, sbrc

Homework

2. Design a checking strategy that can find the endianness of AVR machine.

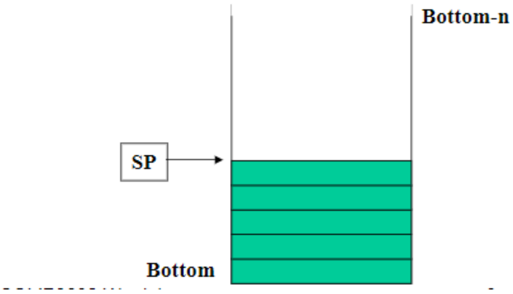
Homework

3. Convert lowercase to uppercase for a string (for example, “hello”)
   – The string is stored in the program memory
   – The resulting string after conversion is stored in the data memory.
     - In ASCII, uppercase letter + 32 = lowercase letter
       – e.g. ‘A’+32=‘a’

Week4

Stack

- What is stack?
  - A data structure in which a data item that is Last In is First Out (LIFO)
- In AVR, a stack is implemented as a block of consecutive bytes in the data memory
- A stack has at least two parameters:
  - Bottom
  - Stack pointer



PUSH

- Syntax: ***push Rr***
- Operands: $Rr \in \{r0, r1, \ldots, r31\}$
- Operation: $(SP) \leftarrow Rr$
  $SP \leftarrow SP - 1$
- Words: 1
- Cycles: 2

POP

- Syntax: ***pop Rd***
- Operands: $Rd \in \{r0, r1, \ldots, r31\}$
- Operation: $SP \leftarrow SP + 1$
  $Rd \leftarrow (SP)$
- Words: 1
- Cycles: 2

Functions

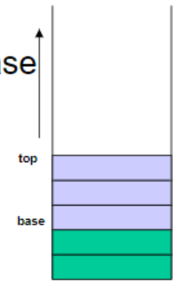
- Stack is used in function calls
- Functions are used
  - in top-down design
    - Conceptual decomposition - easy to design
  - for modularity
    - Readability and maintainability
  - for reuse
    - Design once and use many times
      - Common code with parameters
    - Store once and use many times
      - Saving code size, hence memory space

Three Typical Calling Conventions

- Default C calling convention
  - Push parameters on the stack in reverse order
  - Caller cleans up the stack
    - Larger caller code size
- Pascal calling convention
  - Push parameters on the stack in reverse order
  - Callee cleans up the stack
    - Save caller code size
- Fast calling convention
  - Parameters are passed in registers when possible
    - Save stack size and memory operations
  - Callee cleans up the stack
    - Save caller code size

Stack Frames and Function Calls

- Each function call creates a *stack frame* in the stack.
- The stack frame occupies varied amount of space and has an associated pointer, called *stack frame pointer*.
  - WINAVR uses **Y (r29: r28)** as the stack frame pointer
- The stack frame space is freed when the function returns.
- The stack frame pointer can point to either the base (starting address) or the top of the stack frame
  - In AVR, it points to the top of the stack frame

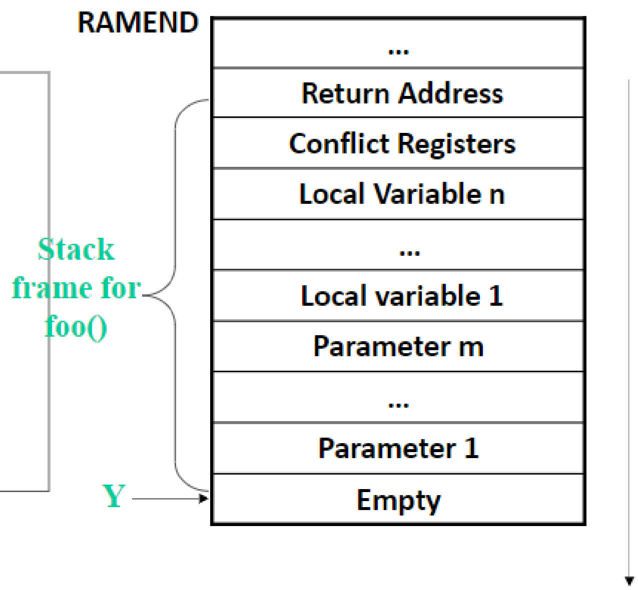


Typical Stack Frame Contents

- Return address
  - Used when the function returns
- Conflict registers
  - One conflict register is the stack frame pointer
  - The original contents of these registers need to be restored when the function returns
- Parameters (arguments)
- Local variables

Stack Frame Structure: an example

```
int main(void)
{ …
  foo(arg1, arg2, …, argm);
}
void foo(arg1, arg2, …, argm)
{ int var1, var2, …, varn;
  …
}
```



A Template for Callee

Callee (function):

- Prologue
- Function body
- Epilogue

A Template for Callee (cont.)

Prologue:

- Save conflict registers, including the stack frame pointer on the stack by using *push* instruction
- Reserve space for local variables and passing parameters
  - by updating the stack pointer SP
    - SP = SP - the size of all parameters and local variables.
    - Using *OUT* instruction
- Update the stack pointer and stack frame pointer Y to point to the top of its stack frame
- Pass the actual parameters' values to the parameters on the stack

Function body:

- Do the normal task of the function on the stack frame and general purpose registers.

A Template for Callee (cont.)

Epilogue:

- Store the return value in the designated registers
- De-allocate the stack frame
  - Deallocate the space for local variables and parameters by updating the stack pointer SP.
    - SP = SP + the size of all parameters and local variables.
    - Using *OUT* instruction
  - Restore conflict registers from the stack by using *pop* instruction
    - The conflict registers must be popped in the reverse order that they were pushed on the stack.
      - The stack frame pointer register of the caller is also restored.
- Return to the caller by using *ret* instruction

Recursive Functions

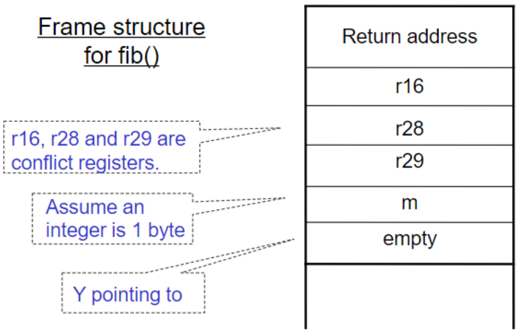
- A recursive function is both a caller and a callee of itself.
- Can be hard to compute the maximum stack space needed for a recursive function call.
  - Need to know how many times the function is nested (the depth of the call).
  - And it often depends on the input values of the function

C Code of Fibonacci Number Calculation

```
int n = 12;
void main(void)
 {
   fib(n);
 }

int fib(int m)
{
  if(m == 0)  return 1;
  if(m == 1)  return 1;
  return (fib(m - 1) + fib(m - 2));
}
```

AVR Assembly Solution



r24 stores the passing parameter value and return value

Assembly Code for main()

```
.include "m2560def.inc"
.cseg
        rjmp main
n:      .db 12

main:
        ldi ZL, low(n <<1)          ; Let Z point to n
        ldi ZH, high(n <<1)
        lpm r24, z                  ; Actual parameter n is stored in r24
        rcall fib                   ; Call fib(n)

halt:
        rjmp halt
```

Assembly Code for fib()

```
fib:                        ; Prologue
    push r16                ; Save r16 on the stack
    push YL                 ; Save Y on the stack
    push YH
    in YL, SPL
    in YH, SPH
    sbiw Y, 1               ; Let Y point to the top of the stack frame
    out SPH, YH             ; Update SP so that it points to
    out SPL, YL             ; the new stack top

    std Y+1, r24            ; get the parameter
    cpi r24, 2              ; Compare n with 0
    brsh L2                 ; If n!=0 or 1
    ldi r24, 1              ; n==0 or 1, return 1
    rjmp L1                 ; Jump to the epilogue
```

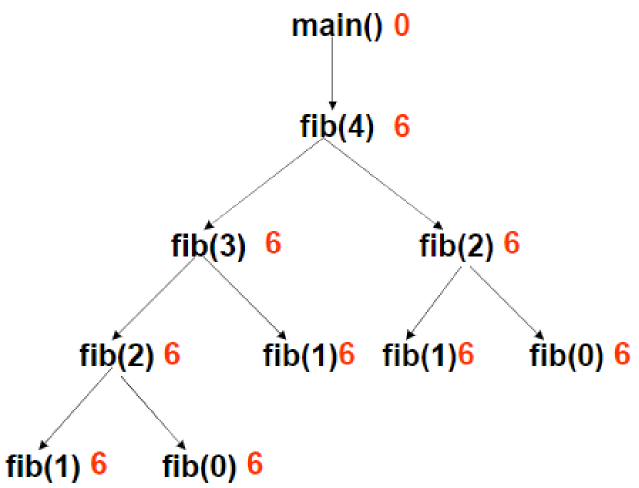
Assembly Code for fib() (cont.)

```
L2:     ldd r24, Y+1        ; n>=2, load the actual parameter n
        dec r24             ; Pass n-1 to the callee
        rcall fib           ; call fib(n-1)
        mov r16, r24        ; Store the return value in r16
        ldd r24, Y+1        ; Load the actual parameter n
        subi r24, 2         ; Pass n-2 to the callee
        rcall fib           ; call fib(n-2)
        add r24, r16        ; r24=fib(n-1)+fib(n-2)
```

Assembly Code for fib() (cont.)

```
L1:
        ; Epilogue
        adiw Y, 1           ; Deallocate the stack frame for fib()
        out SPH, YH         ; Restore SP
        out SPL, YL
        pop YH              ; Restore Y
        pop YL
        pop r16             ; Restore r16
        ret
```

Stack Size



The call tree for n=4

The longest path: fib(4)→fib(3)→fib(2)→fib(1)

Homework

1. Refer to the AVR Instruction Set manual, study the following instructions:
   - Arithmetic and logic instructions
     - sbci
     - lsl, rol
   - Data transfer instructions
     - pop, push
     - in, out
   - Program control
     - rcall
     - ret
   - Bit
     - clc
     - sec

Homework

2. What are the differences between using functions and using macros?