Microprocessors & Interfacing

AVR Programming (III)

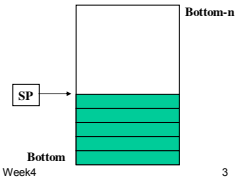
Lecturer : Annie Guo

Lecture Overview

- Stack and stack operation
- Function and function call
  - Calling convention
  - Examples

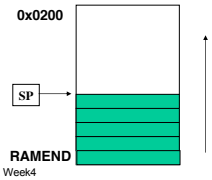
Stack

- 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 SRAM memory
- A stack has at least two parameters:
  - Bottom
  - Stack pointer



Stack Bottom

- The stack usually *grows from higher addresses to lower addresses*
- The stack bottom is the location with the highest address in the stack
- In AVR, 0x0200 is the lowest address for stack
  - i.e. in AVR, stack bottom >=0x0200

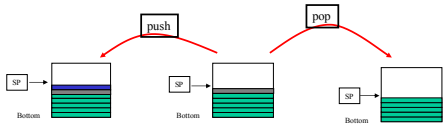


Stack Pointer

- In AVR, the stack pointer, *SP*, is an I/O register pair, *SPH:SPL*, they are defined in the device definition file
  - m2560def.inc
- Default value of the stack pointer is 0x21FF
- The stack pointer always points to the top of the stack
  - Definition of the stack top varies:
    - the location of Last-In element;
      - E.g, in 68K
    - the location available for the next element to be stored
      - E.g. in AVR

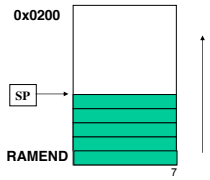
Stack Operations

- There are two stack operations:
  - push
  - pop



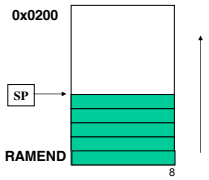
PUSH Instruction

- Syntax: *push Rr*
- Operands: Rr∈{r0, r1, …, r31}
- Operation: (SP) ← Rr
  SP ← SP – 1
- Words: 1
- Cycles: 2



POP Instruction

- Syntax: *pop Rd*
- Operands: Rd∈{r0, r1, …, r31}
- Operation: SP ← SP + 1
  Rd ← (SP)
- Words: 1
- Cycles: 2



Functions

- Stack is used in function/subroutine 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

C Code Example

```
unsigned int pow(unsigned int b, unsigned int e) {      // int parameters b & e,
                                                        // returns an integer
        unsigned int i, p;                              // local variables
        p = 1;
        for (i=0; i<e; i++)                             // p = b^e
                p = p*b;
        return p;                                       // return value of the function
}

int main(void) {
        unsigned int m, n;
        m = 2;
        n = 3;
        m = pow(m, n);
        return 0;
}
```


C Code Example (cont.)

- In this program:
  - Caller
    - main
  - Callee
    - pow
  - Passing parameters
    - b, e
  - Return value
    - p

Function Call

- A function call involves
  - Program flow control between caller and callee
    - target/return addresses
  - Value passing
    - parameters/return values
- Certain rules/conventions are used for implementing functions and function calls.

Rules (I)

- Using stack for parameter passing for reentrant subroutine
  - A reentrant subroutine can be called at any point of a program (or inside the subroutine itself) safely.
- Registers can be used as well for parameter passing
  - For example, WINAVR uses
    - registers r8 ~ r25 to store actual parameters
    - r25:r24 to store the return value
  - Actual parameters may eventually be stored on the stack to free registers.
- Some parameters that have to be used in several places in the program must be saved in the stack.

Rules (II)

- Parameters can be passed by *value* or *reference*
  - Passing by value
    - Pass the value of an actual parameter to the callee
      - Not efficient for structures and arrays
        - » Need to pass the value of each element in the structure or array
  - Passing by reference
    - Pass the address of the actual parameter to the callee
    - Efficient for structures and array passing
    - Using *passing by reference* when the parameter is to be changed by the subroutine
      - Example is given in the next two slides

Passing by Value: Example

- C program

```
void swap(int x, int y){          // the swap(x,y)  in fact
        int temp = x;             // does not work  since
        x = y;                    // the new x,  y values
        y = temp;                 // are not  copies back.

        return;
}

int main(void) {
        int a = 1, b = 2;
        swap(a,b);
        printf("a=%d, b=%d", a, b)
        return 0;
}
```


Passing by Reference: Example

- C program

```
swap(int *px, int *py){           // call by reference
        int temp;                 // allows callee to change
        temp = *px                // the value in caller, since the
        *px = *py;                // "referenced" memory
        *py = temp;               // is altered.
        return;
}

int main(void) {
        int a = 1, b = 2;
        swap(&a,&b);
        printf("a=%d, b=%d", a, b)
        return 0;
}
```


Rules (III)

- If a register is used in both caller and callee functions and the caller needs its old value after the callee returns, then a *register conflict* occurs.
- Compilers or assembly programmers need
  - to check for register conflict.
  - to save conflict registers on the stack.
- Caller or callee or both can save conflict registers.
  - In WINAVR, callee saves conflict registers.

Rules (IV)

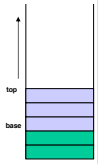
- Local variables and parameters need to be stored contiguously on the stack for easy accesses.
- How are the local variables or parameters stored on the stack?
  - In the order that they appear in the high-level program from left to right, or the reverse order.
  - Either is OK. But the consistency should be maintained.

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

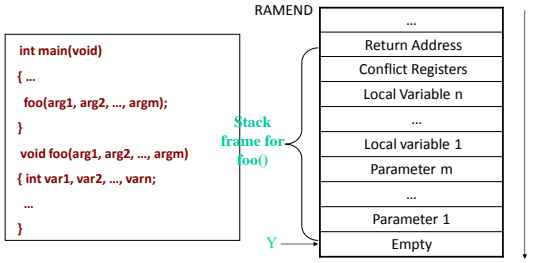
- Each function call creates a *stack frame* in the stack.
- The stack frame occupies varied amount of space and has an associated pointer, called *stack frame pointer*.
  - WINAVR uses **Y (r29: r28)** as the stack frame pointer
- The stack frame space is freed when the function returns.
- The stack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



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

An Example of Stack Frame Structure



A Template for Caller

Basic operations by caller:

- Before calling the callee, store actual parameters in designated registers
- Call callee.
  - Using instructions for subroutine call
    - rcall, icall, call.

Relative Call to Subroutine

- Syntax: ***rcall k***
- Operands: $-2K \le k < 2K$
- Operation: stack ← PC+1, SP ← SP-2
  PC ← PC+k+1
- Words: 1
- Cycles: 3

- For device with 16-bit PC

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

Return from Subroutine Instruction

- Syntax: *ret*
- Operands: none
- Operation: SP ← SP+1, PC ← (SP), SP ← SP+1
- Words: 1
- Cycles: 4

- For device with 16-bit PC

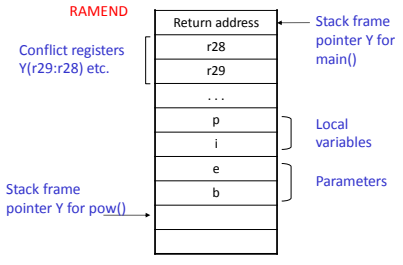
An Example

- C program

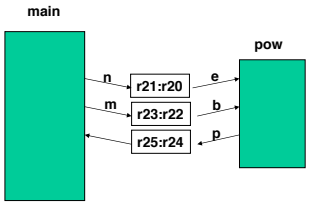
```
unsigned int pow(unsigned int b, unsigned int e) {      // int parameters b & e,
                                                        // returns an integer
        unsigned int i, p;                              // local variables
        p = 1;
        for (i=0; i<e; i++)                             // p = b^e
                p = p*b;
        return p;                               // return value of the function
}

int main(void) {
        unsigned int m, n;
        m = 2;
        n = 3;
        m = pow(m, n);
        return 0;
}
```


Stack frame for pow()



Parameter Passing



An Example (cont.)

- Assembly program
  - Assume an integer takes two bytes

```
.include "m2560def.inc"
.def zero = r15                ; to store constant value 0
.equ m = 2
.equ n = 3
; Macro mul2: multiplication of two 2-byte unsigned numbers with a 2-byte result
; All parameters are registers, @5:@4 should be in the form: rd+1:rd, where d is
; the even number, and rd+1:rd are not r1:r0
; Operation: (@5:@4) = (@1:@0)*(@3:@2)

.macro mul2                    ; a * b
    mul    @0, @2              ; al * bl
    movw  @5:@4, r1:r0
    mul    @1, @2              ; ah * bl
    add    @5, r0
    mul    @0, @3              ; bh * al
    add    @5, r0
.endmacro
                                ; continued
```

An Example (cont.)

- Assembly program

```
; continued
        ;ldi YL, low(RAMEND)            ; set up the stack
        ;ldi YH, high(RANEND)
        ;out SPL, YL
        ;out SPH, YH

        ; main
        ldi r22, low(m)                 ; m = 2
        ldi r23, high(m)
        ldi r20, low(n)                 ; n = 3
        ldi r21, high(n)
        rcall pow                       ; Call subroutine 'pow'
        movw r23:r22, r25:r24           ; Get the return result

end:
        rjmp end
        ; end of main
                                        ; continued
```

An Example (cont.)

- Assembly program

```
; continued
pow:
        ; Prologue:
                                        ; r29:r28 will be used as the frame pointer
        push YL                         ; Save r29:r28 in the stack
        push YH
        push r16                        ; Save registers used in the function body
        push r17
        push r18
        push r19
        push zero
        in YL, SPL                      ; Initialize the stack frame pointer value
        in YH, SPH
        sbiw Y, 8                       ; Reserve space for local variables
                                        ; and parameters.
                                        ; continued
```


An Example (cont.)

- Assembly program

```
; continued
        out SPH, YH         ; Update the stack pointer to
        out SPL, YL         ; point to the new stack top

                            ; Pass the actual parameters
        std Y+1, r22        ; Pass m to b
        std Y+2, r23
        std Y+3, r20        ; Pass n to e
        std Y+4, r21
        ;  End of prologue

                                        ; continued
```


An Example (cont.)

- Assembly program

```
; continued
        ; Function body
                                        ; Use r23:r22 for i and r25:r24 for p,
                                        ; r21:r20 temporarily for e and r17:r16 for b
        clr zero
        clr r23;                        ; Initialize i to 0
        clr r22;
        clr r25;                        ; Initialize p to 1
        ldi r24, 1
        …                               ; Store the local values to the stack
                                        ; if necessary

        ldd r21, Y+4                    ; Load e to registers
        ldd r20, Y+3
        ldd r17, Y+2                    ; Load b to registers
        ldd r16, Y+1
                                        ;  continued
```

An Example (cont.)

- Assembly program

```
; continued
loop:   cp r22, r20                 ; Compare i with e
        cpc r23, r21
        brsh done                   ; If i >= e
        mul2 r24,r25, r16, r17, r18, r19    ; p *= b
        movw r25:r24, r19:r18
        ;std Y+8, r25               ; store p
        ;std Y+7, r24
        ;inc r22                    ; i++, (can we use adiw?)
        ;adc r23, zero
        subi r22, LOW(-1)
        sbci r23, HIGH(-1)
        ;std Y+6, r23               ; store i
        ;std Y+5, r22
        rjmp loop
done:
        ; End of function body
                                                ; continued
```


An Example (cont.)

- Assembly program

```
; continued
        ; Epilogue
        adiw Y, 8               ; De-allocate the reserved space
        out SPH,  YH
        out SPL, YL
        pop zero
        pop r19
        pop r18                 ; Restore registers
        pop r17
        pop r16
        pop YH
        pop YL
        ret                     ; Return to main()
        ; End of epilogue
                                        ; End
```


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 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

An Example of Recursive Function Calls

```
int sum(int n);
int main(void)
{
        int n = 100;
        sum(n);
        return 0;
}
int sum(int n)
 {
  if (n <= 0) return 0;
  else return (n + sum(n - 1));
 }
```

main() is the caller of sum()

sum() is the caller and callee of itself

Stack Space

- Stack space of functions calls in a program can be determined by call tree

Call Trees

- A call tree is a weighted directed tree G = (V, E, W) where
  - V={v1, v2, …, vn} is a set of nodes each of which denotes an execution of a function;
  - E={vi→vj: vi calls vj} is a set of directed edges each of which denotes the caller-callee relationship, and
  - W={wi (i=1, 2, …, n): wi is the frame size of vi} is a set of stack frame sizes.
- The maximum size of stack space needed for the function calls can be derived from the call tree.

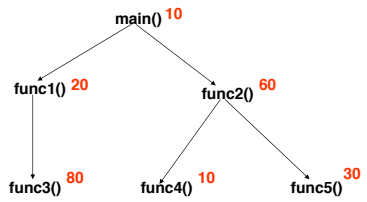
An Example of Call Trees

```
int main(void)
{ ...
  func1();
  ...
  func2();
}

void func1()
{ ...
  func3();
  ...
}
```

```
void func2()
{ ...
  func4();
  ...
  func5();
  ...
}
```


An Example of Call Trees (cont.)

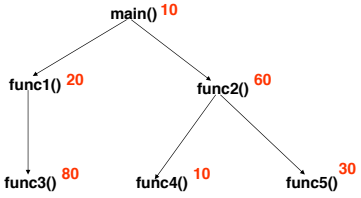


The number in red beside a function is its frame size in bytes.

Computing the Maximum Stack Size for Function Calls

- Step 1: Draw the call tree.
- Step 2: Find the longest weighted path in the call tree.
- The total weight of the longest weighted path is the maximum stack size needed for the function calls.

Example

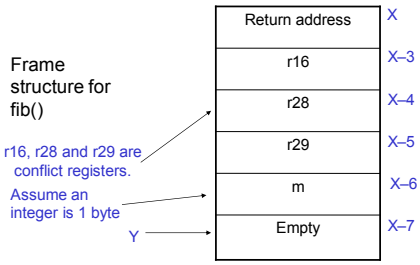


The longest path is main() → func1() → func3() with the total weight of 110. So the maximum stack space needed for this program is 110 bytes.

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, 0      ; Compare n with 0
    brne L2         ; If n!=0 or 1, go to L2
    ldi r24, 1      ; n==0 or 1, return 1
    rjmp L1         ; Jump to the epilogue
                                    ; continued
```


Assembly Code for fib() (cont.)

```
;continued

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)
                                    ; continued
```


Assembly Code for fib() (cont.)

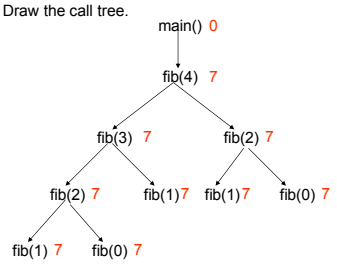
```
; continued

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

                                    ; END
```


Computing the Maximum Stack Size

Step 1: Draw the call tree.



The call tree for n=4

Reading Material

- AVR ATmega2560 data sheet
  - Stack, stack pointer and stack operations

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. In AVR, why register Y is used as the stack frame pointer? And why is the stack frame pointer set to point to the top of the stack frame?
3. What are the differences between using functions and using macros?

Homework

4. Write a macro that can clear a range of data memory locations. The range is given as the macro parameters.