Assembly Programming (III)

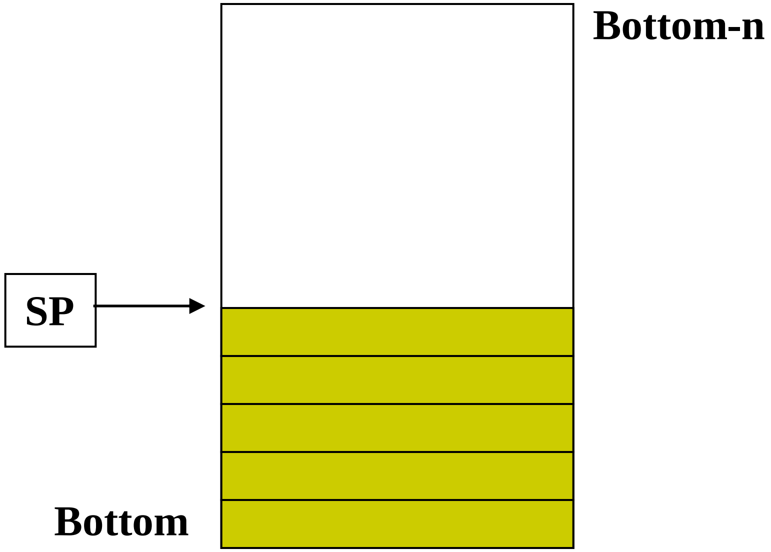
Lecturer: Sri Parameswaran
Notes by: Annie Guo
Dr. Hui Wu

Lecture overview

- Stack and stack operations
- Functions and function calls
  - Calling conventions

Stack

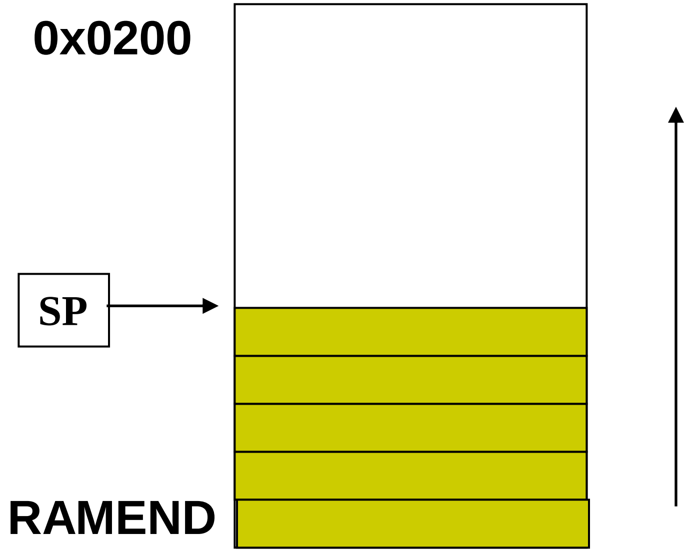
- **What is stack?**
  - **A data structure in which the data item that is Last In is First Out (LIFO)**
- **In AVR, a stack is implemented as a block of consecutive bytes in the SRAM memory**
- **A stack has at least two parameters:**
  - **Bottom**
  - **Stack pointer**



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

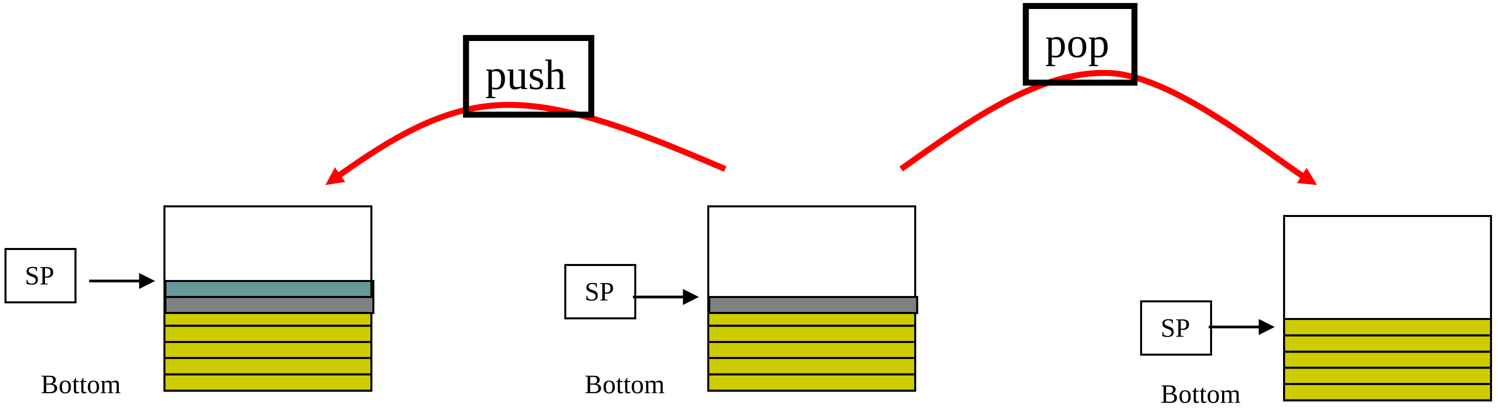


Stack Pointer

- 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 0x0000. Therefore programmers have to initialize a stack before use it.
- The stack pointer always points to the top of the stack
  - Definition of the top of the stack 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: $Rr \in \{r0, r1, \ldots, r31\}$
- Operation: $(SP) \leftarrow Rr$

  $SP \leftarrow SP - 1$
- Words: 1
- Cycles: 2

POP instruction

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

  Rd ← (SP)
- Words: 1
- Cycles: 2

Stack and 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
    - Economy - common code with parameters; design once and use many times

C code example

```
// int parameters b & e,
// returns an integer

unsigned int pow(unsigned int b, unsigned int e) {
        unsigned int i, p;       // local variables
        p = 1;
        for (i = 0; i < e; i++)  // p = be
                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`
  - Passed parameters
    - b, e
  - Return value/type
    - p/integer

Function Call

- A function call involves
  - Program flow control between caller and callee
    - target/return addresses
  - Value passing
    - parameters/return values
- There are two calling conventions for parameter passing

Calling Conventions

- 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

Passing by value: example

- C program

```
void swap(int x, int y){  // the swap(x, y)
      int temp = x;       // does not work
      x = y;              // since the new x
      y = temp;           // y values are not
}                         // copied back
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

```
void swap(int *px, int *py) {    // call by reference
      int temp;                  // allows callee to change
      temp = *px                 // the caller, since the
      *px = *py;                 // “referenced” memory
      *py = temp;                // is altered
}
int main(void) {
      int a = 1, b = 2;
      swap(&a, &b);
      printf(“a=%d, b=%d”, a, b)
      return 0;
}
```

Register Conflicts

- 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 conflicts.
- Need to save conflict registers on the stack.
- Caller or callee or both can save conflict registers.
  - In WINAVR, callee saves some conflict registers.

Passing Parameters and Return Values

- May use general registers to store part of actual parameters and push the rest of parameters on the stack.
  - WINAVR uses general registers r8 ~ r25 to store actual parameters
  - Actual parameters are eventually stored on the stack to free registers.
- The return value needs be stored in designated registers
  - WINAVR uses r25:r24 to store the return value.

Stack Frames and Function calls

- Each function call creates a new stack frame on the stack.
- The stack frame occupies varied amount of space and has an associated pointer, called the stack frame pointer.
- The stack frame space is freed when the function returns.
- What's inside a stack frame?

Typical Stack Frame Contents

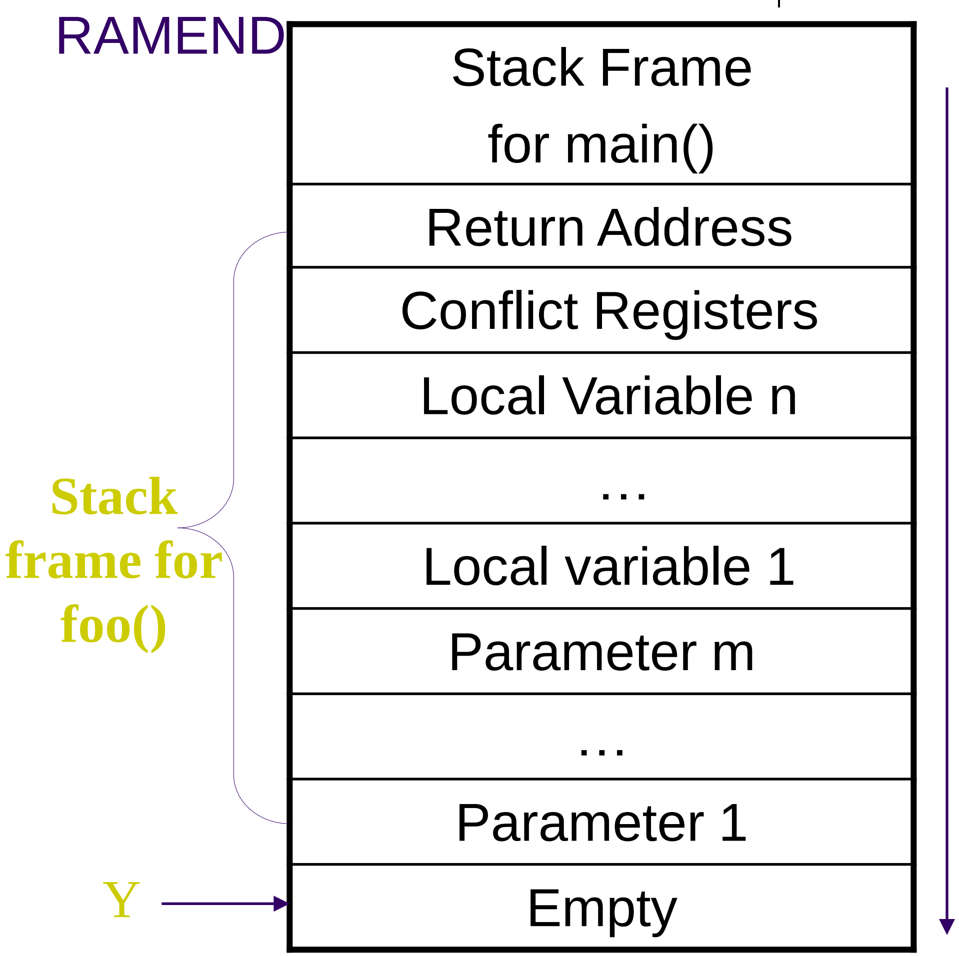
- Return address
  - Used when the function returns
- Conflict registers
  - Need to restore the old contents of these registers when the function returns
  - One conflict register is the stack frame pointer
- Parameters (arguments)
- Local variables

Implementation Considerations

- Local variables and parameters need be stored contiguously on the stack for easy accesses.
- In which order the local variables or parameters stored on the stack? In the order that they appear in the program from left to right? Or the reverse order?
  - WINAVR C compiler uses the reverse order.
- The stack 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. Otherwise, points to the base of the stack frame (Why?)
  - WINAVR uses **Y (r29: r28)** as a stack frame register.

A Sample Stack Frame Structure for AVR

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



A Template for Caller

Caller:

- Before calling the callee, store actual parameters in designated registers.
- Call the 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 devices with 16-bit PC

A Template for Callee

Callee:

- Prologue
- Function body
- Epilogue

A Template for Callee (Cont.)

Prologue:

- Store conflict registers, including the stack frame register Y, on the stack by using `push` instruction
- Reserve space for local variables and passed parameters
- Update the stack pointer and stack frame pointer Y to point to the top of its stack frame
- Pass the actual parameters to the formal 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 designated registers r25:r24.
- De-allocate the stack frame
  - De-allocate 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 are pushed on the stack.
      - The stack frame register of the caller is also restored.
- Return to the caller by using **`ret`** instruction

Return from subroutine

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

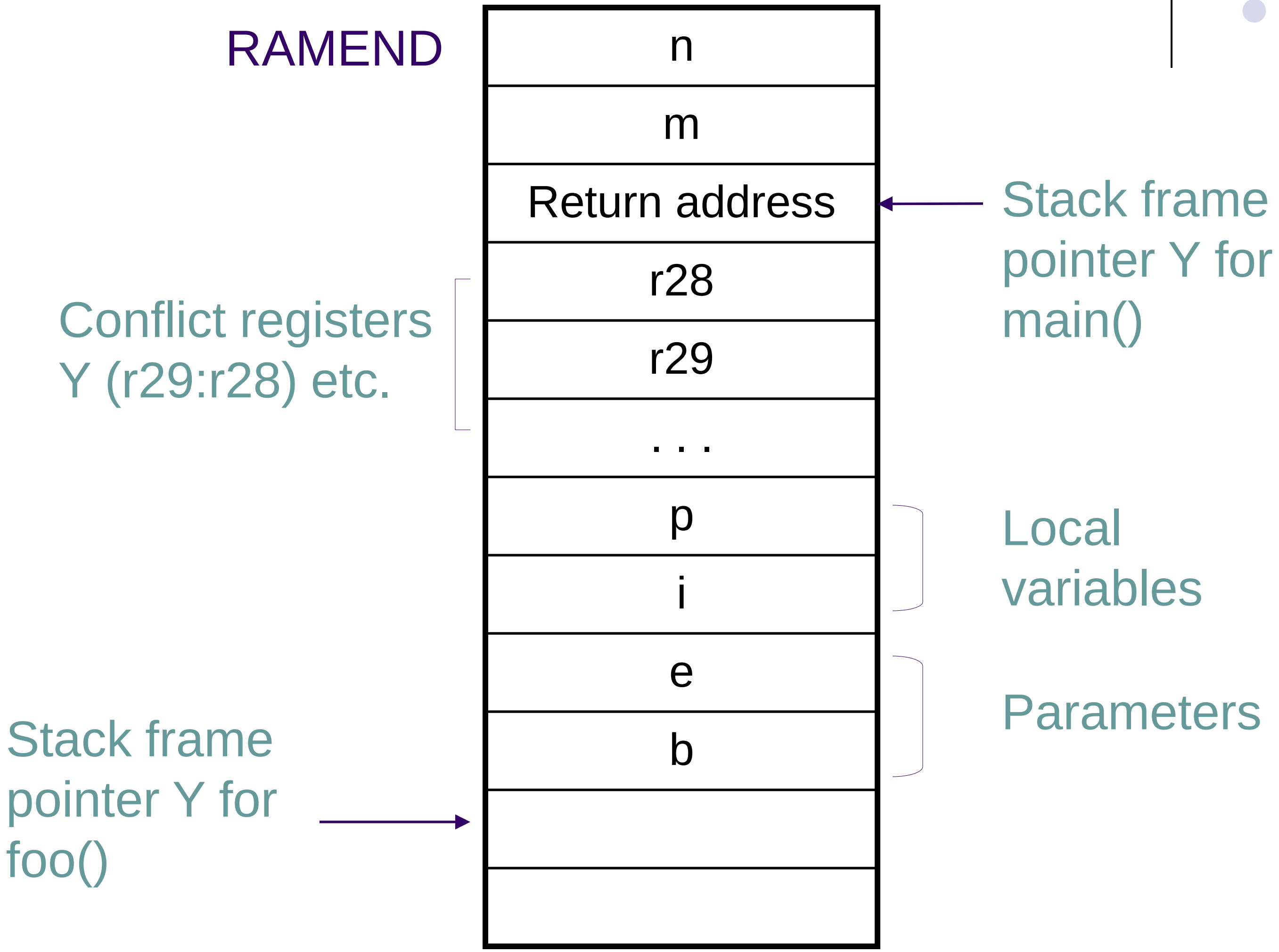
- For devices with 16-bit PC

An Example

- C program

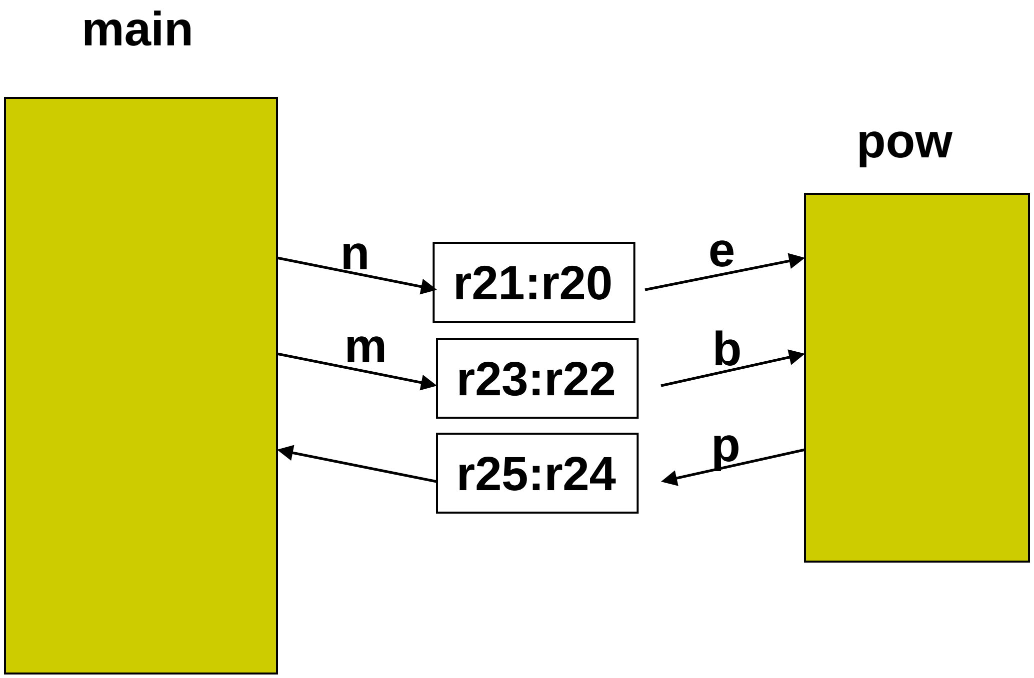
```
// int parameters b & e,
// returns an integer
unsigned int pow(unsigned int b, unsigned int e) {
        unsigned int i, p;    // local variables
        p = 1;
        for (i = 0; i < e; i++) // p = b^e
                p = p*b;
        return p;             // return value of the function
}

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

Stack frames for main() and pow()



High address for high byte

Parameter passing



An example

- Assembly program

```
.include "m2560def.inc"

.def zero = r15             ; To store constant value 0

; 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 they are not r1 and 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

- Assembly program

```
;  continued
main:
        ldi YL, low(RAMEND-4)    ; 4 bytes to store local variables.
        ldi YH, high(RAMEND-4)   ; Assume an integer is 2 bytes.
        out SPH, r29             ; Adjust stack pointer to point
        out SPL, r28             ; to the new stack top.

                                 ; Function body of ‘main’
        ldi r24, low(2)          ; m = 2;
        ldi r25, high(2)
        std Y+1, r24
        std Y+2, r25

        ldi r24, low(3)          ; n = 3;
        ldi r25, high(3)
        std Y+3, r24
        std Y+4, r25
                                                         ; continued
```

An example

- Assembly program

```
; continued
                                    ; Prepare parameters for function call.
        ldd r20, Y+3                ; r21:r20 hold the actual parameter n
        ldd r21, Y+4
        ldd r22, Y+1                ; r23:r22 hold the actual parameter m
        ldd r23, Y+2
        rcall pow                   ; Call subroutine ‘pow’

        std Y+1, r24                ; Store the returned result
        std Y+2, r25
end:
        rjmp end
        ; end of main function()
                                                              ; continued
```

An example

- 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

- 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

- Assembly program

```
; continued
                  ; Function body
                  ; Use r23:r22 for i and r21:r20 for p,
                  ; r25:r24 temporarily for e, and r17:r16 for b
        clr zero
        clr r23;                ; Initialize i to 0
        clr r22;
        clr r21;                ; Initialize p to 1
        ldi r20, 1
        ldd r25, Y+4            ; Load e to registers
        ldd r24, Y+3
        ldd r17, Y+2            ; Load b to registers
        ldd r16, Y+1
                                                        ; continued
```

An example

- Assembly program

```
; continued
loop:   cp r22, r24                     ; compare i with e
        cpc r23, r25
        brsh done                       ; if i >= e
        mul2 r20,r21, r16,r17, r18,r19  ; p *= b
        movw r21:r20, r19:r18
        ; AVR does not have add immediate instructions (addi, addci)
        ; but it can be done by subtracting a negative immediate.
        ; Could adiw be used instead?
        subi r22, LOW(-1)               ; i++
        sbci r23, HIGH(-1)
        rjmp loop
done:
        movw r25:r24, r21:r20
        ; End of function body

                                                     ; continued
```

An example

- Assembly program

```
; continued
        ; Epilogue
        ;ldd r25, Y+8     ; the return value of p is stored in r25,r24
        ;ldd r24, Y+7
        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 calls.
  - Need to know how many times the function is nested (the depth of the calls).
  - And it often depends on the input values of the function.

NOTE: the following section is from the COMP2121 lecture notes by Dr. Hui Wu

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

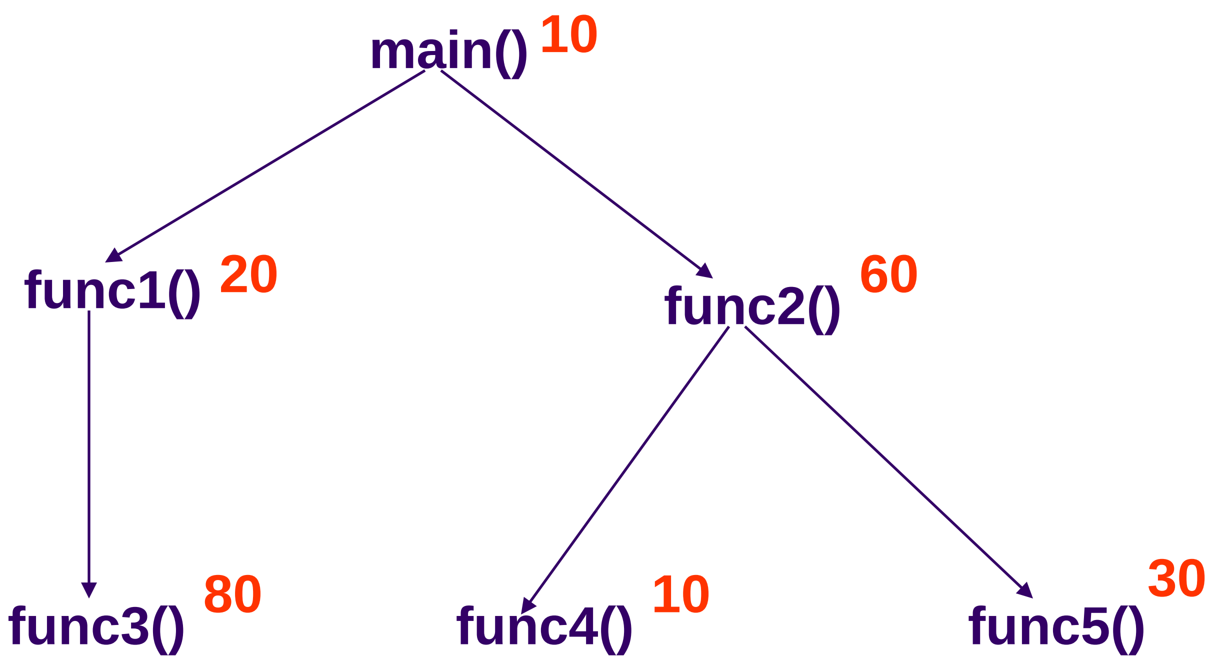
- A call tree is a weighted directed tree G = (V, E, W) where
  - V={v1, v2, …, vn} is a set of nodes each of which denotes an execution of a function;
  - E={vi→vj: vi calls vj} is a set of directed edges each of which denotes the caller-callee relationship, and
  - W={wi (i=1, 2, …, n): wi is the frame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

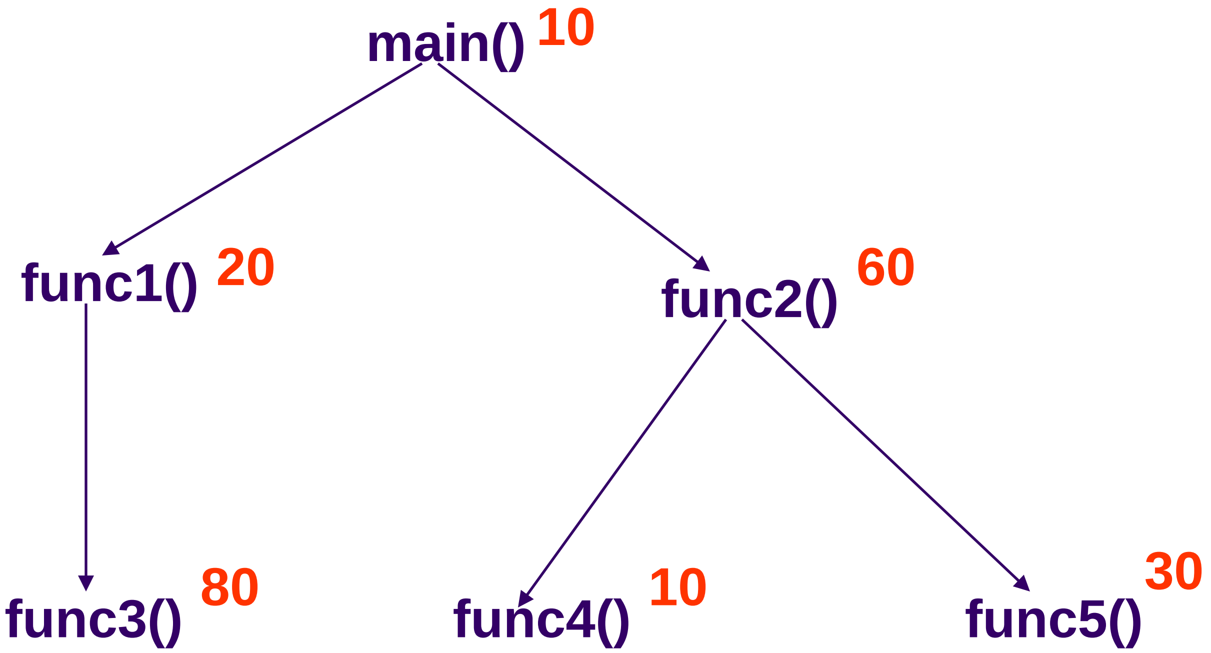


The number in red beside a function is its frame size in bytes.

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.

An 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.

Fibonacci Rabbits

- Suppose a newly-born pair of rabbits, one male, one female, are put in a field. Rabbits are able to mate at the age of one month so that at the end of its second month a female can produce another pair of rabbits. Suppose that our rabbits never die and that the female always produces one new pair (one male, one female) every month from the second month on.
- How many pairs will there be in one year?
  - Fibonacci’s Puzzle
  - Italian, mathematician Leonardo of Pisa (also known as Fibonacci) 1202.

Fibonacci Rabbits (Cont.)

- The number of pairs of rabbits in the field at the start of each month is 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- In general, the number of pairs of rabbits in the field at the start of month n, denoted by F(n), is recursively defined as follows.

$$F(n) = F(n - 1) + F(n - 2)$$

Where $F(0) = F(1) = 1$.

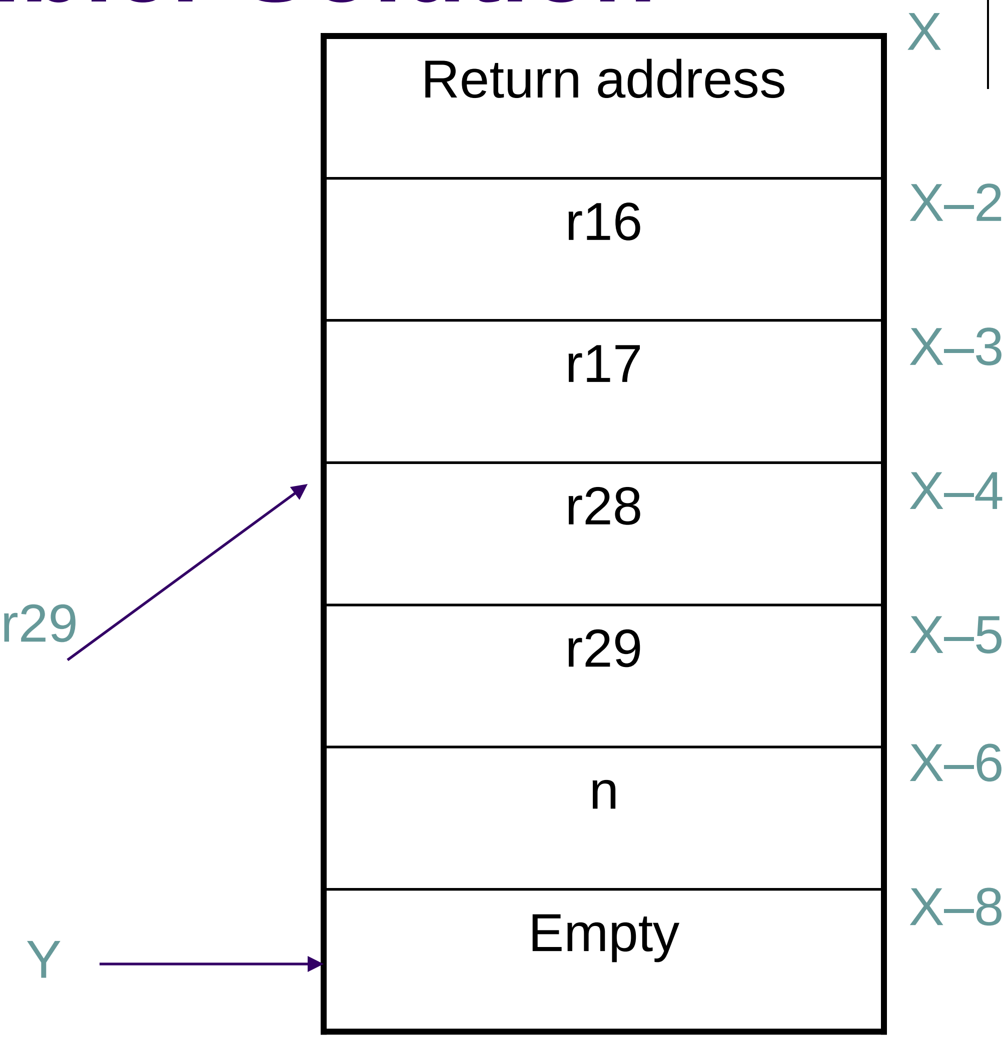
F(n) (n = 1, 2, …,) are called Fibonacci numbers.

C Solution of Fibonacci Numbers

```
int month = 4;
int main(void)
{
    fib(month);
}
int fib(int n)
{
   if (n == 0) return 1;
   if (n == 1) return 1;
   return (fib(n - 1) + fib(n - 2));
}
```

AVR Assembler Solution

Frame structure for fib()



r16, r17, r28 and r29 are conflict registers.

An integer is 2 bytes long in WINAVR

Assembly Code for main()

```
.cseg
    rjmp main
month:
    .dw 4
main:
                                ; Prologue
    ldi r28, low(RAMEND)
    ldi r29, high(RAMEND)
    out SPH, r29                ; Initialise the stack pointer SP to point to
    out SPL, r28                ; the highest SRAM address
                                ; End of prologue
    ldi r30, low(month << 1)    ; Let Z point to month
    ldi r31, high(month << 1)
    lpm r24, Z+                 ; Actual parameter 4 is stored in r25:r24
    lpm r25, Z
    rcall fib                   ; Call fib(4)
                                ; Epilogue: no return

loopforever:
    rjmp loopforever
```

Assembly Code for fib()

```
fib:
    push r16            ; Prologue
    push r17            ; Save r16 and r17 on the stack
    push r28            ; Save Y on the stack
    push r29
    in r28, SPL
    in r29, SPH
    sbiw r29:r28, 2     ; Let Y point to the bottom of
                        ; the stack frame
    out SPH, r29        ; Update SP so that it points to
    out SPL, r28        ; the new stack top
    std Y+1, r24        ; Pass the actual parameter
    std Y+2, r25        ; to the formal parameter
    cpi r24, 0          ; Compare n with 0
    clr r0
    cpc r25, r0
    brne L3             ; If n != 0, go to L3
    ldi r24, 1          ; n == 0
    ldi r25, 0          ; Return 1
    rjmp L2             ; Jump to the epilogue
```

Assembly Code for fib() (Cont.)

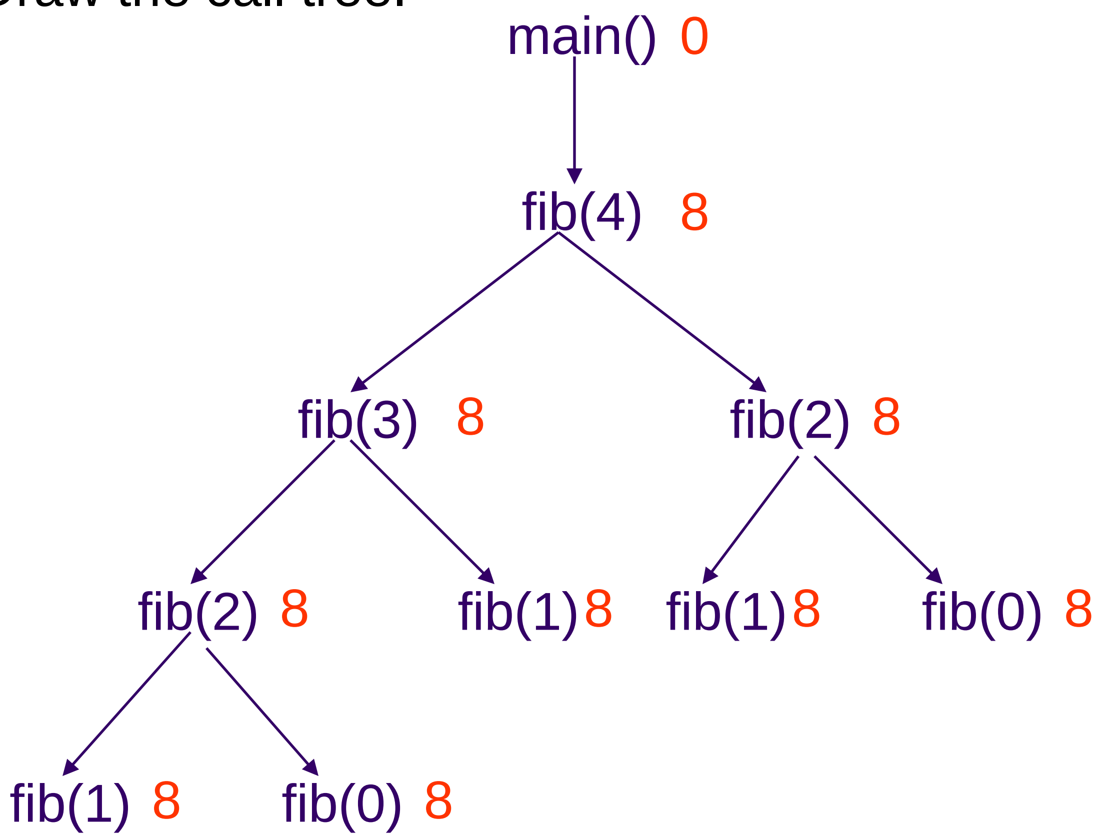
```
L3: cpi r24, 1          ; Compare n with 1
    clr r0
    cpc r25, r0
    brne L4             ; If n != 1 go to L4
    ldi r24, 1          ; n == 1
    ldi r25, 0          ; Return 1
    rjmp L2             ; Jump to the epilogue
L4: ldd r24, Y+1        ; n >= 2
    ldd r25, Y+2        ; Load the actual parameter n
    sbiw r25:r24, 1     ; Pass n - 1 to the callee
    rcall fib           ; call fib(n - 1)
    movw r16, r24       ; Store the return value in r17:r16
    ldd r24, Y+1        ; Load the actual parameter n
    ldd r25, Y+2
    sbiw r25:r24, 2     ; Pass n - 2 to the callee
    rcall fib           ; call fib(n-2)
    add r24, r16        ; r25:r25 = fib(n - 1) + fib(n - 2)
    adc r25, r17
```

Assembly Code for fib() (Cont.)

```
L2:
    ; Epilogue
    adiw r29:r28, 2    ; Deallocate the stack frame for fib()
    out SPH, r29       ; Restore SP
    out SPL, r28
    pop r29            ; Restore Y
    pop r28
    pop r17            ; Restore r17 and r16
    pop r16
    ret
```

Computing the Maximum Stack Size

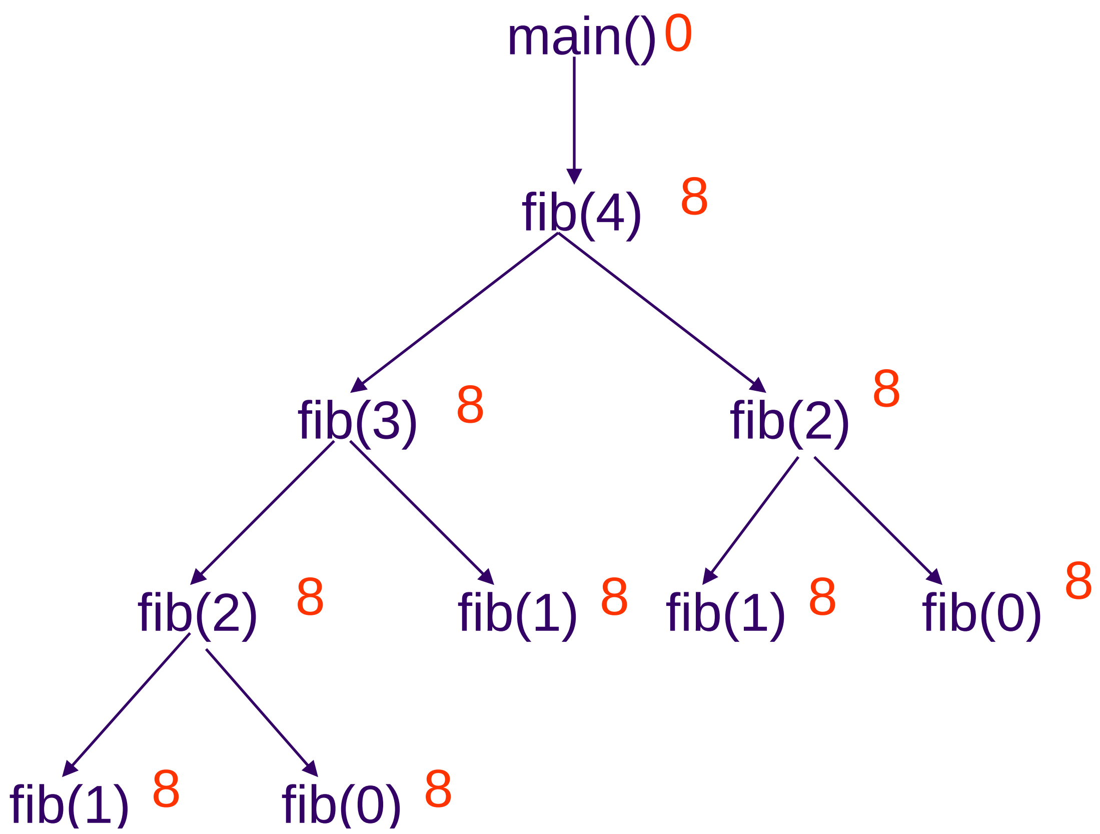
Step 1: Draw the call tree.



The call tree for n = 4

Computing the Maximum Stack Size (Cont.)

Step 1: Find the longest weighted path.



The longest weighted path is main() → fib(4) → fib(3) →fib(2) → fib(1) with the total weight of 32. So a stack space of 32 bytes is needed for this program.

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
     - `adiw`
     - `lsl, rol`
   - Data transfer instructions
     - `movw`
     - `pop, push`
     - `in, out`
   - Program control
     - `rcall`
     - `ret`

Homework

2. In AVR, why is register Y used as the stack frame pointer? And why is the stack frame pointer set to point to the top of the stack frame?

Homework

3. What is the difference between using functions and using macros?

Homework

4. When would you use macros and when would you use functions?

Homework

5. Write an assembly routine for *a x 5*, where *a* is a 2-byte unsigned integer.

Homework

6. Write an assembly code for the following C program. Assume an integer takes one byte.

```
void swap(int *px, int *py) {    // Call by reference
       int temp;                  // allows the callee to
       temp = *px                 // change the caller, since
       *px = *py;                 // the “referenced” memory
       *py = temp;                // is altered.
}
int main(void) {
       int a = 1, b = 2;
       swap(&a, &b);
       printf(“a=%d, b=%d”, a, b)
       return 0;
}
```