COMP2121: Microprocessors and Interfacing

AVR Assembly Programming (III)
Functions, Macros, and Assembly Process

http://www.cse.unsw.edu.au/~cs2121

Lecturer: Hui Wu

Term 2, 2019

1

Overview

- Stack
- Variable types
- Memory sections in C
- Parameter passing
- Stack frames
- Implementation of functions
- Recursive functions
- Computing the stack size for function calls
- Macros
- Assembly process

2

2

Stacks

- A stack is a contiguous area of memory that supports two operations:
  - **push**: push a data item on the top of the stack
  - **pop**: pop the data item on top of the stack to a register
- **LIFO**-First In, Last Out
- Every processor has a stack of some kind
  - Used for function (subroutine) calls and interrupts
  - Used to store local variables in C
- A special register named **Stack Pointer** (**SP**) stores the address of the stack top

3

3

Push Register on Stack

- Syntax: push Rr
- Operands: Rr∈{r0, r1, …, r31}
- Operation: (SP) ← Rr
  SP ← SP –1
- Flag affected: None
- Encoding: 1001 001d dddd 1111
- Words: 1
- Cycles: 2
- Example

```
         call routine      ; Call subroutine
         ...
routine: push r14          ; Save r14 on the stack
         push r13          ; Save r13 on the stack
         ...
         pop r13           ; Restore r13
         pop r14           ; Restore r14
         ret               ; Return from subroutine
```

4

4

Pop Register from Stack

- Syntax: pop Rd
- Operands: $Rd \in \{r0, r1, \ldots, r31\}$
- Operation: $SP \leftarrow SP + 1$
  $Rd \leftarrow (SP)$
- Flag affected: None
- Encoding: 1000 000d dddd 1111
- Words: 1
- Cycles: 2
- Example

```
         call routine      ; Call subroutine
         ...
routine: push r14          ; Save r14 on the stack
         push r13          ; Save r13 on the stack
         ...
         pop r13           ; Restore r13
         pop r14           ; Restore r14
         ret               ; Return from subroutine
```

5

5

Stacks (Cont.)

- A stack will grow after push is executed.
- A stack will shrink after pop is executed.
- A stack may grow upwards (from a lower address to a higher address) or downwards (from a higher address to a lower address).
- The direction in which a stack grows is determined by the hardware.

6

6

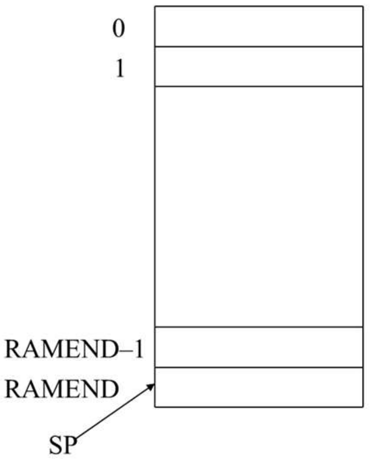
AVR and Stacks

- Stacks are part of SRAM space.
- Stacks grow downwards (from a higher address to a lower address).
- SP needs to hold addresses (therefore 16 bits wide).
  - ❑ Made up of two 8 bit registers
    - o SPH (high byte) (IO register $3E)
    - o SPL (low byte) (IO register $3D)
- First thing to do in any program is to initialize the stack pointer.
  - ❑ Typically stacks use the top of SRAM space.

7

7

AVR Stack Initialization

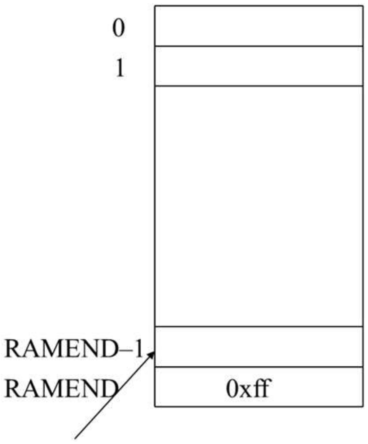



```
.include "m2560def.inc"
.def temp=r20
.cseg
 ldi temp, low(RAMEND)
 out spl, temp
 ldi temp, high(RAMEND)
 out sph, temp
```

8

8

AVR Stack Operations

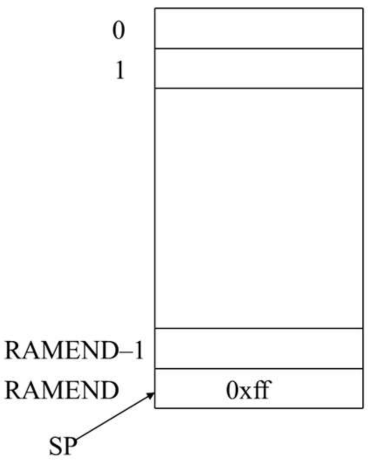



```
.include "m2560def.inc"
.def temp=r20
.cseg
 ldi temp, low(RAMEND)
 out spl, temp
 ldi temp, high(RAMEND)
 out sph, temp
 ldi r1, 0xff
 push r1
```

9

9

AVR Stack Operations (Cont.)




```
.include "m2560def.inc"
.def temp=r20
.cseg
 ldi temp, low(RAMEND)
 out spl, temp
 ldi temp, high(RAMEND)
 out sph, temp
 ldi r1, 0xff
 push r1
 pop r2        ; r2=0xff
```

10

10

Relative Call to Subroutine

- Syntax: rcall k
- Operands: $-2K \leq k < 2K$
- Operation: (i) STACK ← PC + 1 (Store return address)
  (ii) SP ← SP – 2 (2 bytes, 16 bits) for devices with 16 bits PC
  SP ← SP – 3 (3 bytes, 22 bits) for devices with 22 bits PC
  (iii) PC ← PC + k + 1
- Flag affected: None.
- Encoding: 1101 kkkk kkkk kkkk
- Words: 1
- Cycles: 3 (Devices with 16-bit PC)
  4 (Devices with 22-bit PC)

11

11

Relative Call to Subroutine (Cont.)

- Example:

```
         rcall  routine    ; Call subroutine
         ...
routine: push r14          ; Save r14 on the stack
         push r15          ; Save r15 on the stack
         ...               ; Put the code for the subroutine here.
         pop r15           ; Restore r15
         pop r14           ; Restore r14
         ret               ; Return from subroutine
```

12

12

Indirect Call to Subroutine

- Syntax: icall
- Operation: (i) STACK ← PC + 1 (Store return address)
  (ii) SP ← SP – 2 (2 bytes, 16 bits) for devices with 16 bits PC
  SP ← SP – 3 (3 bytes, 22 bits) for devices with 22 bits PC
  (iii) PC(15:0) ← Z(15:0) for devices with 16 bits PC
  PC(15:0) ← Z(15:0) and PC(21:16) ← 0 for devices with 22 bits PC
- Flag affected: None.
- Encoding: 1001 0101 0000 1001
- Words: 1
- Cycles: 3 (Devices with 16-bit PC)
  4 (Devices with 22-bit PC)

13

13

Indirect Call to Subroutine (Cont.)

- Example:

```
        clr  r10                  ; Clear r10
        ldi r20, 2                ; Load call table offset
        ldi r30, low(Lab<<1)      ; High byte of the starting address (base) of call table
        ldi r31, high(Lab<<1)     ; Low byte of the starting address (base) of call table
        add r30, r20
        adc r31, r10              ; Base + offset is the address of the call table entry
        lpm r0, Z+                ; Load low byte of the the call table entry
        lpm r1, Z                 ; Load high byte of the call table entry
        movw r31:r30, r1:r0       ; Set the pointer register Z to point the target function
        icall                     ;  Call the target function
          …
  Lab:  .dw ct_l0                 ; The first entry of the call table
        .dw ct_l1                 ; The second entry of the call table
          …
ct_l0:  nop
ct_l1:  nop
          …
```

14

14

Long Call to Subroutine

- Syntax: call k
- Operands: $0 \le k < 64K$
- Operation: (i) STACK ← PC + 1 (Store return address)
  (ii) SP ← SP – 2 (2 bytes, 16 bits) for devices with 16 bits PC
  SP ← SP – 3 (3 bytes, 22 bits) for devices with 22 bits PC
  (iii) PC ← k
- Flag affected: None.
- Encoding: 1001 010k kkkk 111k
  kkkk kkkk kkkk kkkk
- Words: 2
- Cycles: 4 (Devices with 16-bit PC)
  5 (Devices with 22-bit PC)

Long Call to Subroutine (Cont.)

- Example:

```
        mov r16, r0         ; Copy r0 to r16
        call check          ; Call subroutine
        nop                 ; Continue (do nothing)
        ...
check:  cpi r16, $42        ; Check if r16 has a special value
        breq error          ; Branch if equal
        …
error:  ldi r1, 1
        …                   ; put the code for handling the error here
        ret                 ; Return from subroutine
```

Return from Subroutine

- Syntax: ret
- Operation: (i) SP ← SP + 2 (2 bytes, 16 bits) for devices with 16 bits PC
  SP ← SP + 3 (3 bytes, 22 bits) for devices with 22 bits PC
  (ii) PC(15:0) ← STACK for devices with 16 bits PC
  PC(21:0) ← STACK Devices with 22 bits PC
- Flag affected: None
- Encoding: 1001 0101 0000 1000
- Words: 1
- Cycles: 4 (Devices with 16-bit PC)
  5 (Devices with 22-bit PC)
- Example:

```
routine:  push r14    ; Save r14 on the stack
          ...         ; Put the code for the subroutine here.
          pop r14     ; Restore r14
          ret         ; Return from subroutine
```

17

17

Types of Variables in C

1. Global variables: The variable that are declared outside a function
   - ❑ Exist during the execution of the program
2. Local variables: The variables that are declared in a function.
   - ❑ Exist during the execution of the function only
3. Static variables.
   - ❑ Can be either global or local.
   - ❑ A global static variable is valid only within the file where it is declared
   - ❑ A local static variable still exists after the function returns

18

18

Variable Types and Memory Sections

- Global variables occupy their memory space during the execution of the program
  - ❑ Need the static memory which exists during the program’s lifetime
- Static local variables still occupy their memory space after the function returns.
  - ❑ Also need the static memory which exists after the function returns.
- Local variables occupy their memory space only during the execution of the function.
  - ❑ Need the dynamic memory which exists only during the execution of the function
- So the entire memory space need be partitioned into different sections to be more efficiently utilized.

19

19

An Example (1/3)

```
#inlcude <stdio.h>
int x, y;               /* Global variables */
static int b[10];       /* Static global array */
void auto_static(void)
{
 int autovar=1;         /* Local variable */
 static int staticvar=1;        /* Static local variable */
 printf(autovar = %i, staticvar = %i\n, autovar, staticvar);
 ++autovar;
 ++staticvar;
}
```

20

20

An Example (2/3)

```
int main(void)
{
   int i;            /* Local variable */
   for (i=0; i<5; i++)
       auto_static();
   return 0;
}
```

21

21

An Example (3/3)

Program output:

Autovar = 1, staticvar = 1
Autovar = 1, staticvar = 2
Autovar = 1, staticvar = 3
Autovar = 1, staticvar = 4
Autovar = 1, staticvar = 5

22

22

Memory Sections in C for General Microprocessors

- Heap: Used for dynamic memory applications such as malloc() and calloc()
- Stack: Used to store return address, actual parameters, conflict registers and local variables and other information.
- Uninitialized data section .bss,
  - contains all uninitialized global or static local variables.
- Data section .data.
  - Contains all initialized global or static local variables
- Text section .text
  - Contains code

23

23

Memory Sections in WINAVR (C for AVR)

- Additional EEPROM section .eeprom
  - Contains constants in eeprom
- The text section .text in WINAVR includes two subsections .initN and .finiN
  - .initN contains the startup code which initializes the stack and copies the initialized data section .data from flash to SRAM.
  - .finiN is used to define the exit code executed after return from main() or a call to exit().

24

24

C Functions

```
void main(void) {
  int i, j, k, m;
  i = mult(j,k);
  ... ;
  m = mult(i,i);
  …;
}
int mult (int mcand, int mlier)
{
   int product = 0;
   while (mlier > 0)  {
   product = product + mcand;
   mlier = mlier -1;
 }
  return product;
}
```

Caller

Actual Parameters

Callee

25

25

Two Parameter Passing Approaches

- Pass by value
  - ❑ Pass the value of an actual parameter to the callee
  - ❑ Not efficient for structures and array
    - ❖ Need to pass the value of each element in the structure or array
- Pass by reference
  - ❑ Pass the address of the actual parameter to the callee
  - ❑ Efficient for structures and array passing

26

26

Parameter Passing in C

- Pass by value for scalar variables such as char, int and float.
- Pass by reference for non-scalar variables i.e. array and structures.

27

27

Implementation of C Functions

Issues:

- How to pass the actual parameters by value to a function?
- How to pass the actual parameters by reference to a function?
- Where to get the return value?
- How to allocate stack memory to local variables?
- How to deallocate stack memory after a function returns?
- How to handle register conflicts?

Rules are needed between caller and callee.

28

28

Register Conflicts

- If a register is used in both caller and callee and the caller needs its old value after the return from the callee, then a register conflict occurs. The register is called a conflicting register.
- Compilers or assembly programmers need to check for register conflicts.
- Save conflicting registers on the stack.
- Caller or callee or both can save conflicting registers.
  - ❑ In WINAVR, callee saves conflicting registers.

29

29

Parameter Passing and Return Value

- May use general registers to store part of actual parameters and push the rest of parameters on the stack.
  - ❑ WINAVR uses general registers up to r24 to store actual parameters
  - ❑ Actual parameters are eventually passed to the formal parameters stored on the stack.
- The return value need be stored in designated registers
  - ❑ WINAVR uses r25:r24 to store the return value.

30

30

Stack Structure

- A stack consists of stack frames.
- A stack frame is created whenever a function is called.
- A stack frame is freed whenever the function returns.
- What's inside a stack frame?

31

31

Stack Frame

A typical stack frame consists of the following components:

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

32

32

Implementation Considerations

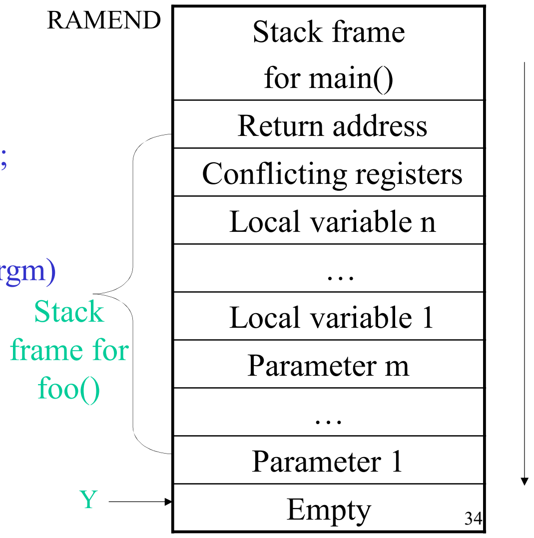
- Local variables and parameters need be stored contiguously on the stack for easy accesses.
- In which order the local variables or parameters stored on the stack? In the order that they appear in the program from left to right? Or the reverse order?
  - ❑ C compiler uses the reverse order.
- Need a stack frame register to point 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.

33

33

An Sample Stack Frame Structure for AVR

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



34

34

A Template for Caller

Caller:

1. Store a subset of actual parameters in designated registers and the rest of actual parameters on the stack.
2. Call the callee.

35

35

A Template for Callee (1/3)

Callee:

1. Prologue
2. Function body
3. Epilogue

36

36

A Template for Callee (2/3)

Prologue:

- Store conflicting registers, including the stack frame register Y, on the stack by using push
- Pass the actual parameters to the formal parameters on the stack
- Update the stack frame register Y to point to the top of its stack frame

Function body:

Does the normal task of the function.

37

37

A Template for Callee (3/3)

Epilogue:

1. Store the return value in designated registers r25:r24.
2. Deallocate local variables and parameters by updating the stack pointer SP.
   - SP=SP + the size of all parameters and local variables.
3. Restore conflicting registers from the stack by using pop
   - The conflicting registers must be popped in the reverse order in which they are pushed on the stack.
   - The stack frame register of the caller is also restored.
   - Step 2 and Step 3 together deallocate the stack frame.
4. Return to the caller by using ret.

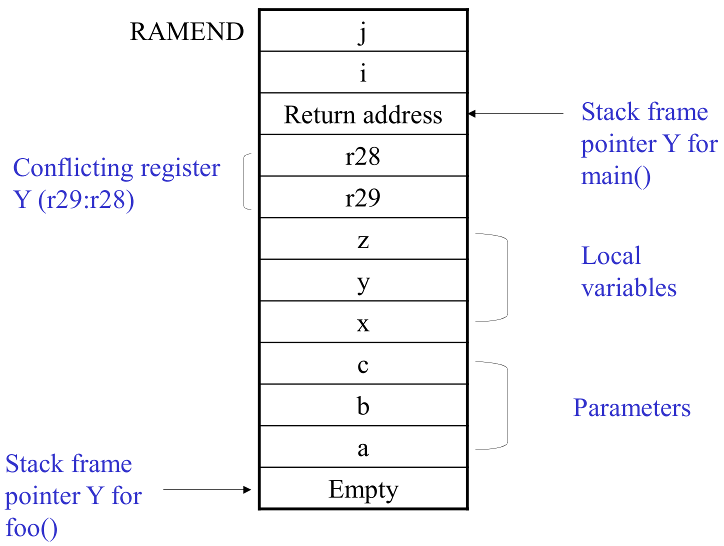
38

38

An Example

```
int foo(char a, int b, int c);

 int main()
{ int i, j;
  i=0;
  j=300;
  foo(1, i, j);
  return 0;
}

int foo(char a, int b, int c)
  { int x, y, z;
    x=a+b;
    y=c–a;
    z=x+y;
    return z;
  }
```


Stack frames for main() and foo()



An Example (1/3)

```
.include  "m2560def.inc"    ; Include definition file for ATmega2560
.cseg
main:  ldi r28, low(RAMEND-4)    ; 4 bytes to store local variables i and j
       ldi r29, high(RAMEND-4)   ; The size of each integer is 2 bytes
       out SPH, r29              ; Adjust stack pointer so that it points to
       out SPL, r28              ; the new stack top.
       clr r0                    ; The next three instructions implement i=0
       std Y+1, r0               ; The address of i in the stack is Y+1
       std Y+2, r0
       ldi r24, low(300)         ; The next four instructions implement j=300
       ldi r25, high(300)
       std Y+3, r24
       std Y+4, r25
       ldd r20,Y+3               ; r21:r20 keep the actual parameter j
       ldd r21,Y+4
       ldd r22,Y+1               ; r23:r22 keep the actual parameter i
       ldd r23,Y+2
       ldi  r24,low(1)           ; r24 keeps the actual parameter 1
       rcall foo                 ; Call foo
       ...
```

41

41

An Example (2/3)

```
foo:                  ; Prologue: frame size=11 (excluding the stack frame
                      ; space for storing return address and registers)
      push r28        ; Save r28 and r29 in the stack
      push r29
      in r28, SPL
      in r29, SPH
      sbiw r28, 11    ; Compute the stack frame top for foo
                      ; Notice that 11 bytes are needed to store
                      ; the actual parameters a, i, j and local
                      ; variables x, y and z
      out SPH, r29    ; Adjust the stack frame pointer to point to
      out SPL, r28    ; the new stack frame
      std Y+1, r24    ; Pass the actual parameter 1 to a
      std Y+2, r22    ; Pass the actual parameter i  to b
      std Y+3, r23
      std Y+4, r20    ; Pass the actually parameter j to c
      std Y+5, r21    ; End of prologue
```

42

42

An Example (3/3)

```
foo:
        …                   ; Function body here
                            ; Epilogue starts here
        ldd r24, Y+10       ; The return value of z is store in r25:r24
        ldd r25, Y+11
        adiw r28, 11        ; Deallocate the stack frame
        out  SPH,  r29
        out SPL, r28
        pop r29             ; Restore Y
        pop r28
        ret                 ; Return to main()
```

43

43

Recursive Functions

- A recursive function is both a caller and a callee of itself.
- Need to check both its source caller (that is not itself) and itself for register conflicts.
- 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).

44

44

An Example of Recursive Function Calls

```
int sum(int n);
int main(void)
{ int n=100;
  sum(n);
  return 0;
}
void 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

45

45

Call Trees

- A call tree is a weighted directed tree G = (V, E, W) where
  - ❑ $V=\{v_1, v_2, \ldots, v_n\}$ is a set of nodes each of which denotes an execution of a function;
  - ❑ $E=\{v_i \rightarrow v_j: v_i \text{ calls } v_j\}$ is a set of directed edges each of which denotes the caller-callee relationship, and
  - ❑ $W=\{w_i\ (i=1, 2, \ldots, n): w_i \text{ is the frame size of } v_i\}$ 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.

46

46

An Example of Call Trees (1/2)

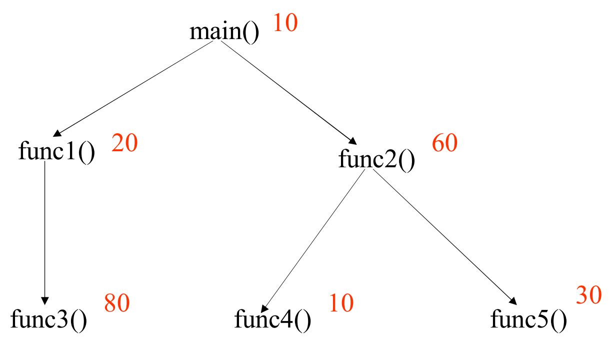
```
int main(void)
{ …
  func1();
  …
  func2();
}

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

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

47

47

An Example of Call Trees (2/2)



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

48

48

Computing the Maximum Stack Size for Function Calls

Step 1: Draw the call tree.

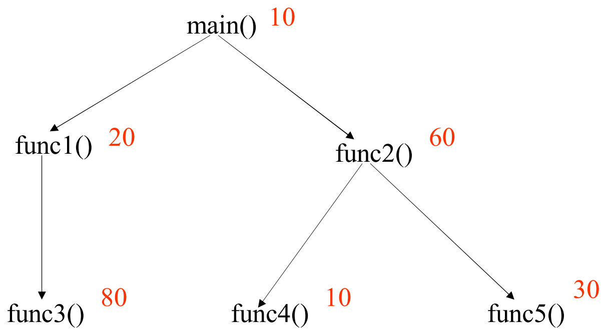
Step 2: Find the longest weighted path in the call tree.

The total weight of the longest weighted path is the maximum stack size needed for the function calls.

49

49

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.

50

50

Fibonacci Rabbits (1/2)

- 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 be there in one year?

  Fibonacci's Puzzle

  Italian, mathematician Leonardo of Pisa (also known as Fibonacci) 1202.

51

51

Fibonacci Rabbits (2/2)

- 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.

52

52

C Solution of Fibonacci Numbers

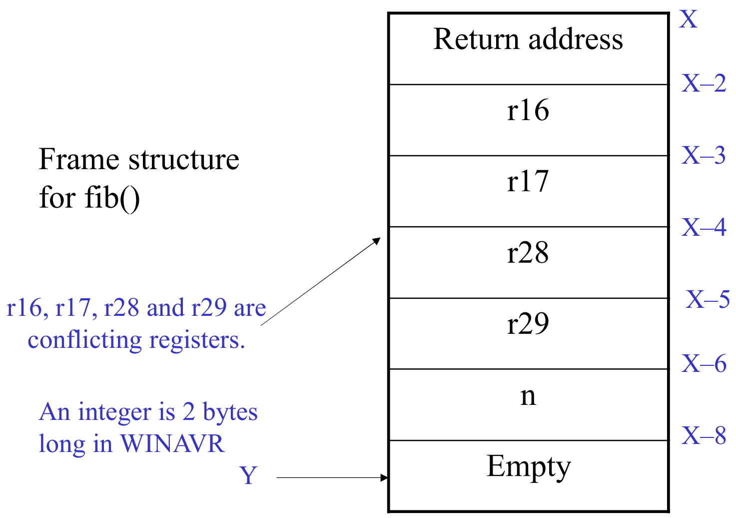
```
int month=4;
void main()
 {
    fib(month);
  }
int fib(int n)
{
   if(n == 0)  return 1;
   if(n == 1)  return 1;
   return (fib(n - 1) + fib(n - 2));
}
```

53

53

AVR Assembler Solution

Frame structure
for fib()



54

54

Assembly Code for main()

```
.include  "m2560def.inc"    ; Include definition file for ATmega2560
.cseg
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
```

55

55

Assembly Code for fib() (1/3)

```
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 to the formal parameter
     std Y+2, r25
     clr r0
     cpi r24, 0         ; Compare n with 0
     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
```

56

56

Assembly Code for fib() (2/3)

```
L3:     clr r0
        cpi r24, 1          ; Compare n with 1
        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 r24, 1         ; Pass n-1 to the callee
        rcall fib           ; call fib(n-1)
        mov r16, r24        ; Store the return value in r17:r16
        mov r17, r25
        ldd r24, Y+1        ; Load the actual parameter n
        ldd r25, Y+2
        sbiw r24, 2         ; Pass n-2 to the callee
        rcall fib           ; call fib(n-2)
        add r24, r16        ; r25:r24=fib(n-1)+fib(n-2)
        adc r25, r17
```

57

57

Assembly Code for fib() (3/3)

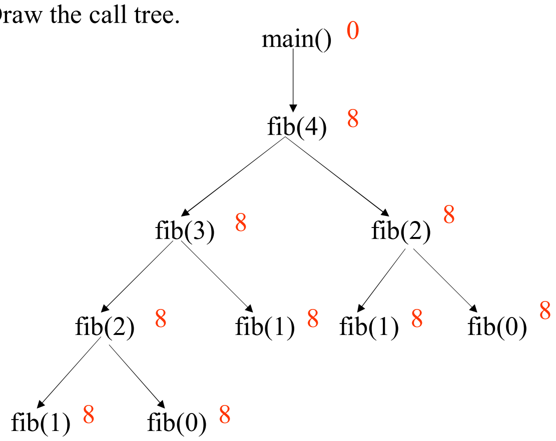
```
L2:
                            ; Epilogue
        adiw r29:r28, 2     ; Deallocate the stack frame for fib()
        out SPH, r29        ; Restore SP
        out SPL, r28
        pop r29             ; Restore Y
        pop r28
        pop r17             ; Restore r17 and r16
        pop r16
  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

58

58

Computing the Maximum Stack Size (1/2)

Step 1: Draw the call tree.



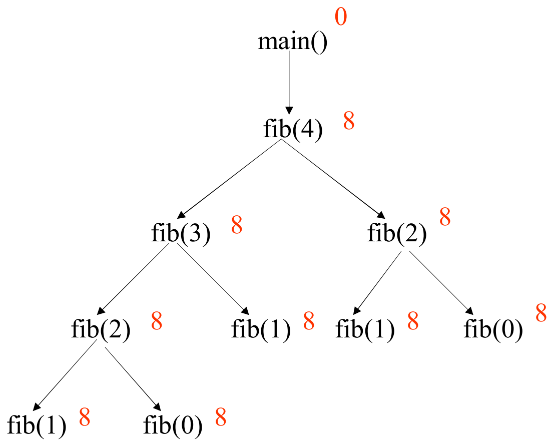
The call tree for n=4

59

59

Computing the Maximum Stack Size (2/2)

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.

60

60

Macros (1/2)

- Assembler programmers often need to repeat sequences of instructions several times
- Could just type them out – tedious
- Could just copy and paste - then the specializations are often forgotten or wrong
- Could use a subroutine, but then there is the overhead of the call and return instructions
- Macros solve this problem
- Consider code to swap two bytes in memory:

```
lds r2, p
lds r3, q
sts  q, r2
sts  p, r3
```

61

61

Macros (2/2)

- Swapping p and q twice
- Without macro

```
lds r2, p
lds r3, q
sts q, r2
sts p, r3

lds r2, p
lds r3, q
sts q, r2
sts p, r3
```

- With macro

```
.macro      myswap
   lds r2, p
   lds r3, q
   sts q, r2
   sts p, r3
.endmacro

   myswap
   myswap
```

62

62

AVR Macro Parameters

- There are up to 10 parameters
  - ❑ Indicated by @0 to @9 in the macro body
  - ❑ @0 is the first parameter, @1 the second, and so on
- Other assemblers let you give meaningful names to parameters

63

63

AVR Parameterised Macro

- Without macro

```
lds r2, p
lds r3, q
sts q, r2
sts p, r3

lds r2, r
lds r3, s
sts s, r2
sts r, r3
```

- With macro

```
.macro     change
    lds r2, @0
    lds r3, @1
    sts @1, r2
    sts @0, r3
.endmacro
   change p, q
   change r, s
```

64

64

Another Example

- Subtract 16-bit immediate value from 16 bit number stored in two registers

```
.MACRO SUBI16                ; Start macro definition
      subi @1,low(@0)        ; Subtract low byte
      sbci @2,high(@0)       ; Subtract high byte
.ENDMACRO                    ; End macro definition

.CSEG                        ; Start code segment
      SUBI16 0x1234,r16,r17  ; Sub.0x1234 from
                             ; r17:r16
```

- Useful for other 16-bit operations on an 8-bit processor

65

65

Two Pass Assembly Process

- We need to process the file twice
- Pass One
  - Lexical and syntax analysis: checking for syntax errors
  - Record all the symbols (labels etc) in a *symbol table*
  - Expand macro calls
- Pass Two
  - Use the symbol table to substitute the values for the symbols and evaluate functions.
  - Assemble each instruction
    - i.e. generate machine code

66

66

An Example (1/4)

```
.include "m2560def.inc"    ; Include definition file for ATmega2560
.equ bound =5
.def counter =r17
.dseg
 Cap_word:.byte 5
.cseg
      rjmp start                        ; Interrupt vector tables starts at 0x00
.org 0x003E                             ; Program starts at 0x003E
Low_word: .db "hello"
start:
      ldi zl, low(Low_word<<1)      ; Get the low byte of the address of "h"
      ldi zh, high(Low_word<<1)     ; Get the high byte of the address of "h"
      ldi yh, high(Cap_word)
      ldi yl, low(Cap_word)
      clr counter                       ; counter=0
```

67

67

An Example (2/4)

```
main:
        lpm  r20,  z+     ; Load a letter from flash memory
        subi r20,  32     ; Convert it to the capital letter
        st   y+,  r20     ; Store the capital letter in SRAM
        inc  counter
        cpi  counter, bound
        brlt main

loop: nop
        rjmp loop
```

68

68

An Example (3/4)

- Pass 1: Lexical and syntax analysis

Symbol Table

Symbol	Value
bound	5
counter	17
Cap_word	0x0000
Low_word	0x003E
start	0x0041
main	0x0046
loop	0x004c

69

69

An Example (4/4)

- Pass 2: code generation.

Program address	Machine code	Assembly code
0x00000000:	40C0	rjmp start
	…	
0x0000003E:	6865	"he" ; Little endian
0x0000003F:	6C6C	"ll"
0x00000040:	6F00	"o"
0x00000041:	ECE7	ldi zl, low(Low_word<<1)
0x00000042:	F0E0	ldi zh, high(Low_word<<1)
0x00000043:	D2E0	ldi yh, high(Cap_word)
0x00000044:	C0E0	ldi yl, low(Cap_word)
0x00000045:	1127	clr counter
	…	

70

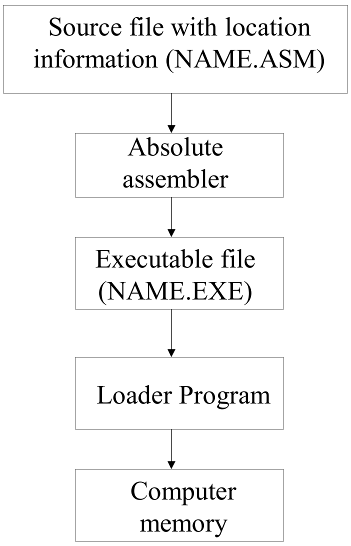
70

Absolute Assemblers (1/2)

- The only source file contains all the source code of the program
- Programmers use .org to tell the assembler the starting address of a segment (data segment or code segment)
- Whenever any change is made in the source program, all code must be assembled.
- A downloader transfers an executable file (machine code) to the target system.

Absolute Assemblers (2/2)

Absolute Assembler Operation



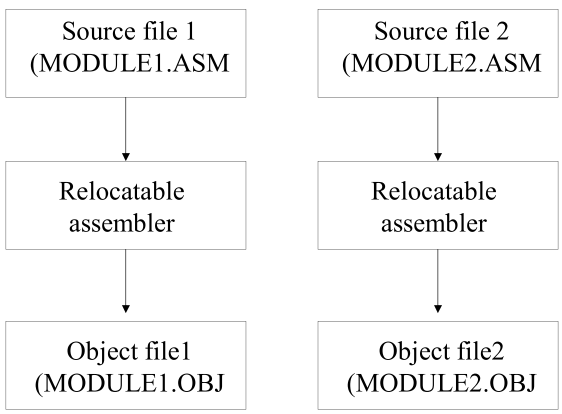
Relocatable Assemblers (1/2)

- The program may be split into multiple source files
- Each source file can be assembled separately
- Each file is assembled into an object file where some addresses may not be resolved
- A linker program is needed to resolve all unresolved addresses and make all object files into a single executable file

73

73

Relocatable Assemblers (2/2)



74

74

Linker (1/2)

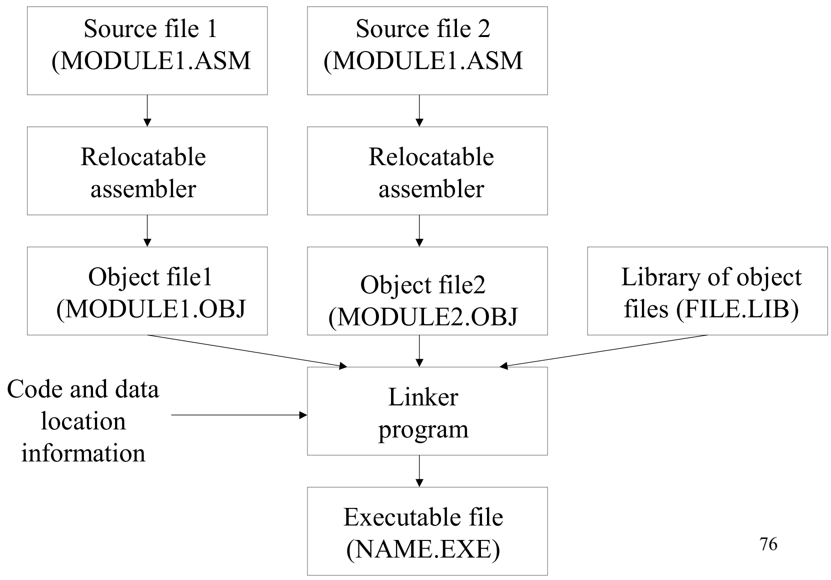
• Takes all object files and links them together and locates all addresses

• Works together with relocatable assembler

75

75

Linker (2/2)



76

76

Loader

- Puts an executable file into the memory of the computer.
- May take many forms.
  - ❑ Part of an operating system.
  - ❑ A downloader program that takes an executable file created on one computer and puts it into the target system.
  - ❑ A system that burns a programmable read-only memory (ROM).

77

77

Reading

1. Chap. 5. Microcontrollers and Microcomputers
2. AVR Assembler
   http://www.atmel.com/webdoc/avrassembler/

78

78