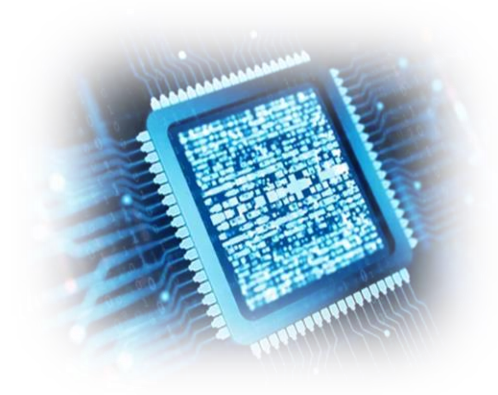


MICROPROCESSOR SYSTEMS

BLG212E

Burak Berk Üstündağ CRN: 11450

Gökhan İnce / Ayşe Yılmazer CRN: 11446

İTÜ Bilgisayar ve Bilişim Fakültesi

Week 10: Programming in ARM Processors, Pointers, Registers, C programming

Overview

- We program in C for convenience
- There are no processor that execute C, only machine code
- So we compile the C into assembly code, a human-readable representation of machine code
- We need to know what the assembly code implementing the C looks like
  - To use the processor efficiently
  - To analyze the code with precision
  - To find performance and other problems
- An overview of what C gets compiled into
  - C start-up module, subroutines calls, stacks, data classes and layout, pointers, control flow, etc.

Programmer's World: The Land of Chocolate!

- As many functions and variables as you want!
- All the memory you could ask for!
- So many data types! Integers, floating point, char, …
- So many data structures! Arrays, lists, trees, sets, dictionaries
- So many control structures! Subroutines, if/then/else, loops, etc.
- Iterators! Polymorphism!

Processor's World

- Data types
  - Integers
  - More if you're lucky!
- Instructions
  - Math: +, -, *
  - Logic: and, or
  - Shift, rotate
  - Move, swap
  - Compare
  - Jump, branch

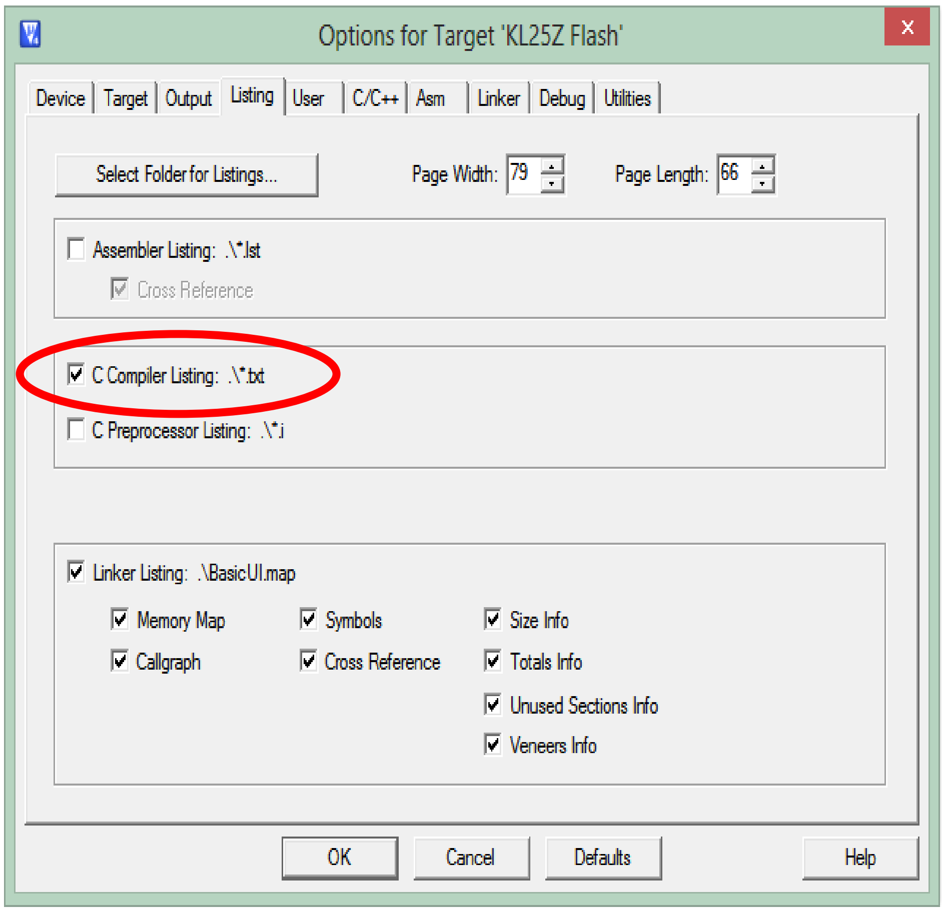
23	251	151	11	3	1	1	1
213	6	234	2	u	1	1	1
2	33	72	1	a	1	1	a
a	4	h	e	l	l	o	1
67	96	a	0	9	9	9	1
6	11	d	72	7	0	0	0
28	289	37	54	42	0	0	0
213	6	234	2	31	1	1	1

Program Translation Stages

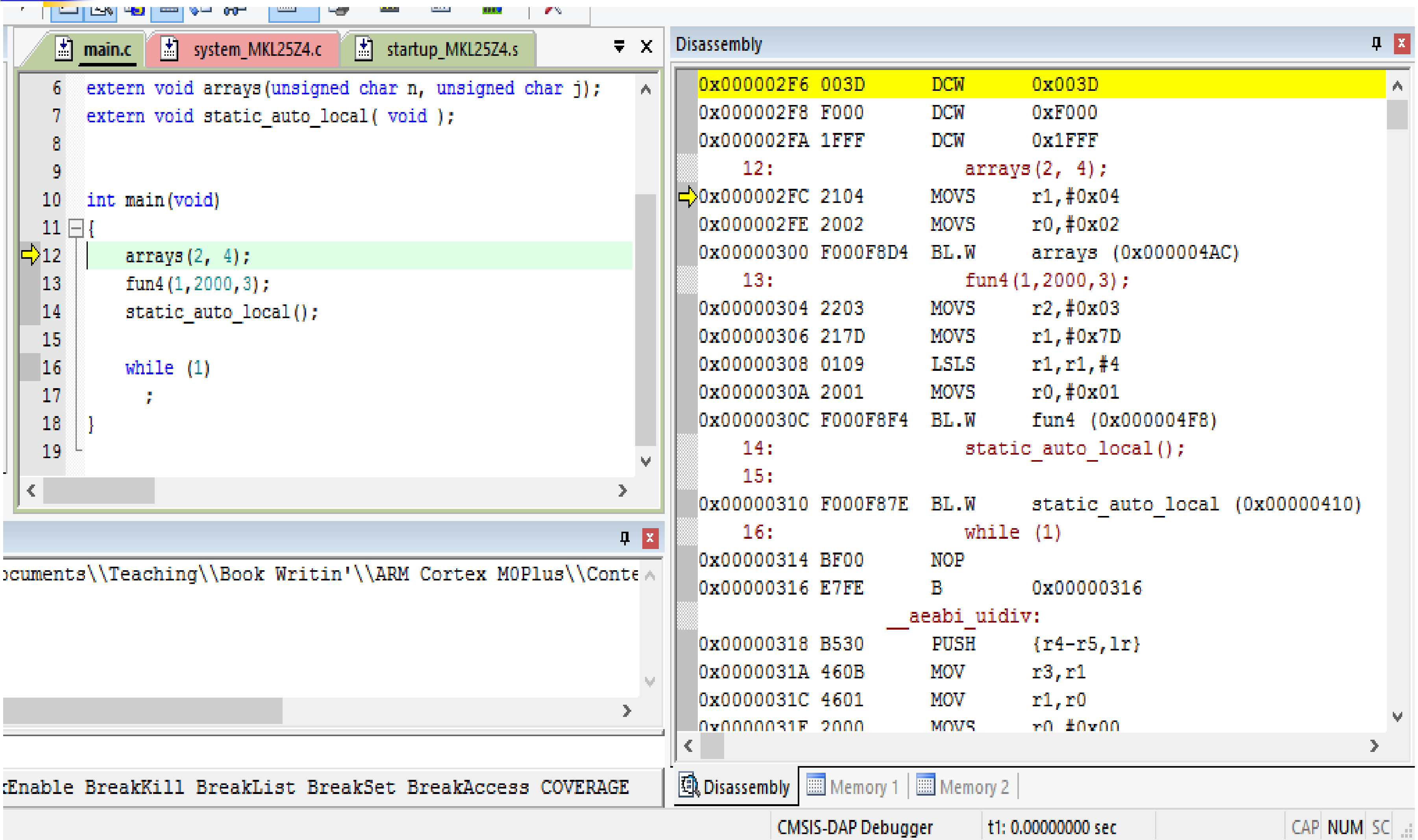
Compiler	■ Parser ▪ reads in C code, ▪ checks for syntax errors, ▪ forms intermediate code (tree representation) ■ High-Level Optimizer ▪ Modifies intermediate code (processor-independent) ■ Code Generator ▪ Creates assembly code from of the intermediate code ▪ Allocates variable uses to registers ■ Low-Level Optimizer ▪ Modifies assembly code (parts are processor-specific)
Assembler	■ Assembler ▪ Creates object code (machine code)
Linker/ Loader	■ Linker/Loader ▪ Creates executable image from one or more object file

Examining Assembly Code before Debugger

- Compiler can generate assembly code listing for reference
- Select in project options



Examining Disassembled Program in Debugger



- View->Disassembly Window

A Warning About Code Optimizations

- Compiler and rest of tool-chain try to optimize code:
  - Simplifying operations
  - Removing "dead" code
  - Using registers
- These optimizations often get in way of understanding what the code does
  - Fundamental trade-off: Fast or comprehensible code?
  - Compilers typically offer a range of optimization levels (e.g. Level 0 to Level 3)
- Code examples here may use "volatile" data type modifier to reduce compiler optimizations and improve readability

Application Binary Interface (ABI)

Defines rules which allow separately developed functions to work together

- ARM Architecture Procedure Call Standard (AAPCS)
  - Which registers must be saved and restored
  - How to call procedures
  - How to return from procedures
- C Library ABI (CLIBABI)
  - C Library functions
- Run-Time ABI (RTABI)
  - Run-time helper functions: 32/32 integer division, memory copying, floating-point operations, data type conversions, etc.

USING REGISTERS

AAPCS Register Use Conventions

- Make it easier to create modular, isolated and integrated code

- Scratch registers are not expected to be preserved upon returning from a called subroutine
  - r0-r3

- Preserved ("variable") registers are expected to have their original values upon returning from a called subroutine
  - r4-r8, r10-r11

AAPCS Core Register Use

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Must be saved, restored by callee-procedure if it will modify them. Calling subroutine expects these to retain their value.

**Must be saved, restored by callee-procedure if it will modify them.
Calling subroutine expects these to retain their value.**

Don't need to be saved. May be used for arguments, results, or temporary values.

MEMORY REQUIREMENTS

What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,“Hello!”);
}
```

- Five possible types
  - Code
  - Read-only static data
  - Writable static data
    - Initialized
    - Zero-initialized
    - Uninitialized
  - Heap
  - Stack
- What goes where?
  - Code is obvious
  - And the others?

What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,“Hello!”);
}
```

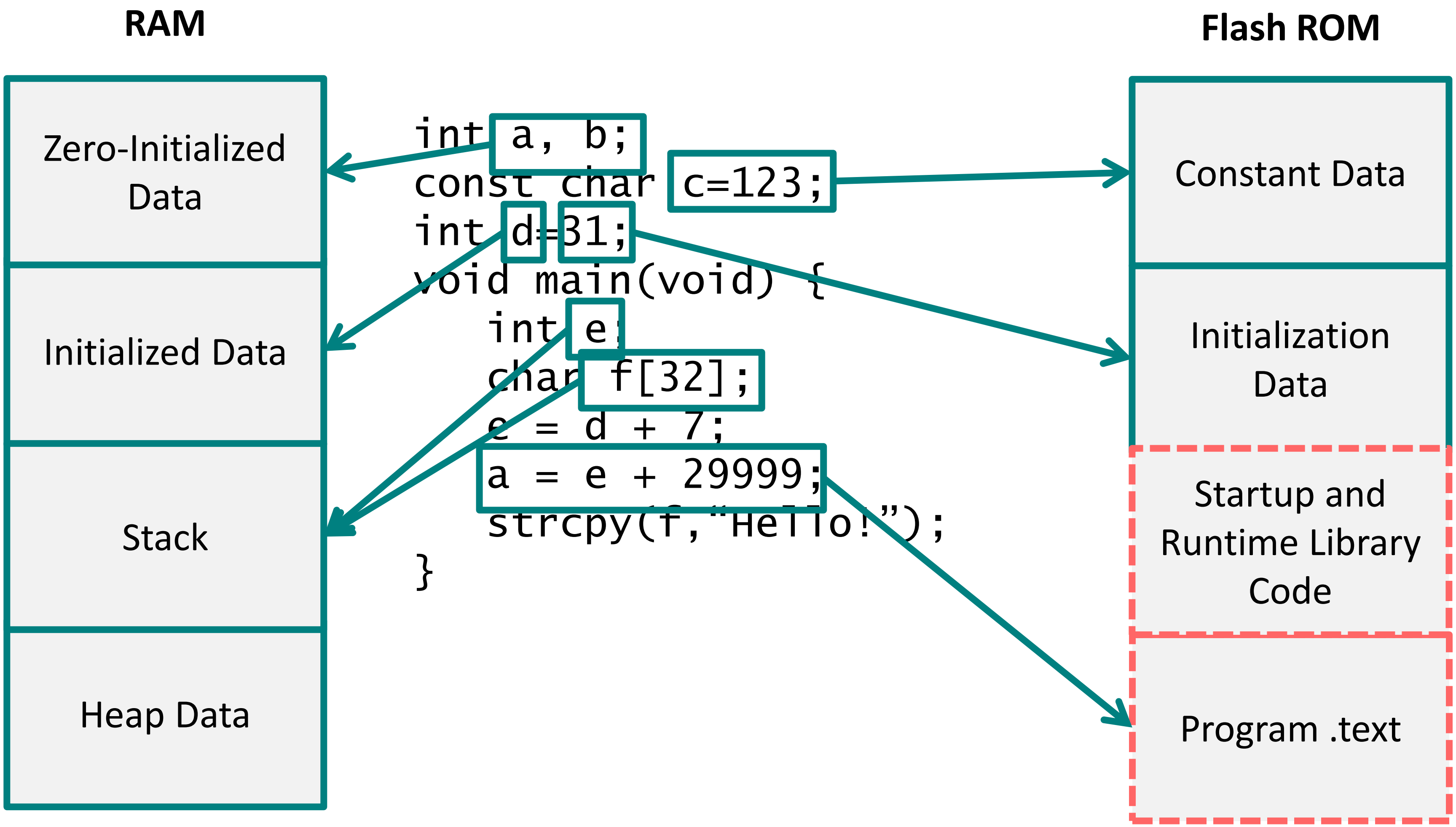
- Can the information change?
  - **If No → Put it in read-only, nonvolatile memory**
    - Instructions
    - Constant strings
    - Constant operands
    - Initialization values
  - **If Yes → Put it in read/write memory**
    - Variables
    - Intermediate computations
    - Return address
    - Other housekeeping data

What Memory Does a Program Need?

```
int a, b;
const char c=123;
int d=31;
void main(void) {
   int e;
   char f[32];
   e = d + 7;
   a = e + 29999;
   strcpy(f,"Hello!");
}
```

- How long does the data need to exist? Reuse memory if possible.
  - **Statically allocated**
    - Exists from **program start to end**
    - Each variable has its own fixed location
    - Space is not reused
  - **Automatically allocated**
    - Exists from **function start to end**
    - Space can be reused
  - **Dynamically allocated**
    - Exists from explicit allocation to explicit deallocation
    - Space can be reused

Program Memory Use



Activation Record

- Activation records are located on the **stack**
  - Calling a function creates an activation record
  - Returning from a function deletes the activation record

- **Automatic variables** and **housekeeping information** are stored in a function's activation record

Lower address		(Free stack space)	
	Activation record for current function	Local storage	<- Stack ptr
		Return address	
		Arguments	
	Activation record for caller function	Local storage	
		Return address	
		Arguments	
	Activation record for caller's caller function	Local storage	
		Return address	
		Arguments	
Higher address	Activation record for caller's caller's caller function	Local storage	
		Return address	
		Arguments	

- Not all fields (Local storage, Return Address, Arguments) may be present for each activation record

Type and Class Qualifiers

- Used to modify a variable's declaration so compiler treats it slightly differently
- Const
  - Never written by program, can be put in ROM to save RAM
- Volatile
  - Can be changed outside of normal program flow: Interrupt Service Routine (ISR), hardware register
  - Compiler must be careful with optimizations
- Static
  - Declared within function, retains value between function invocations
  - Scope is limited to function

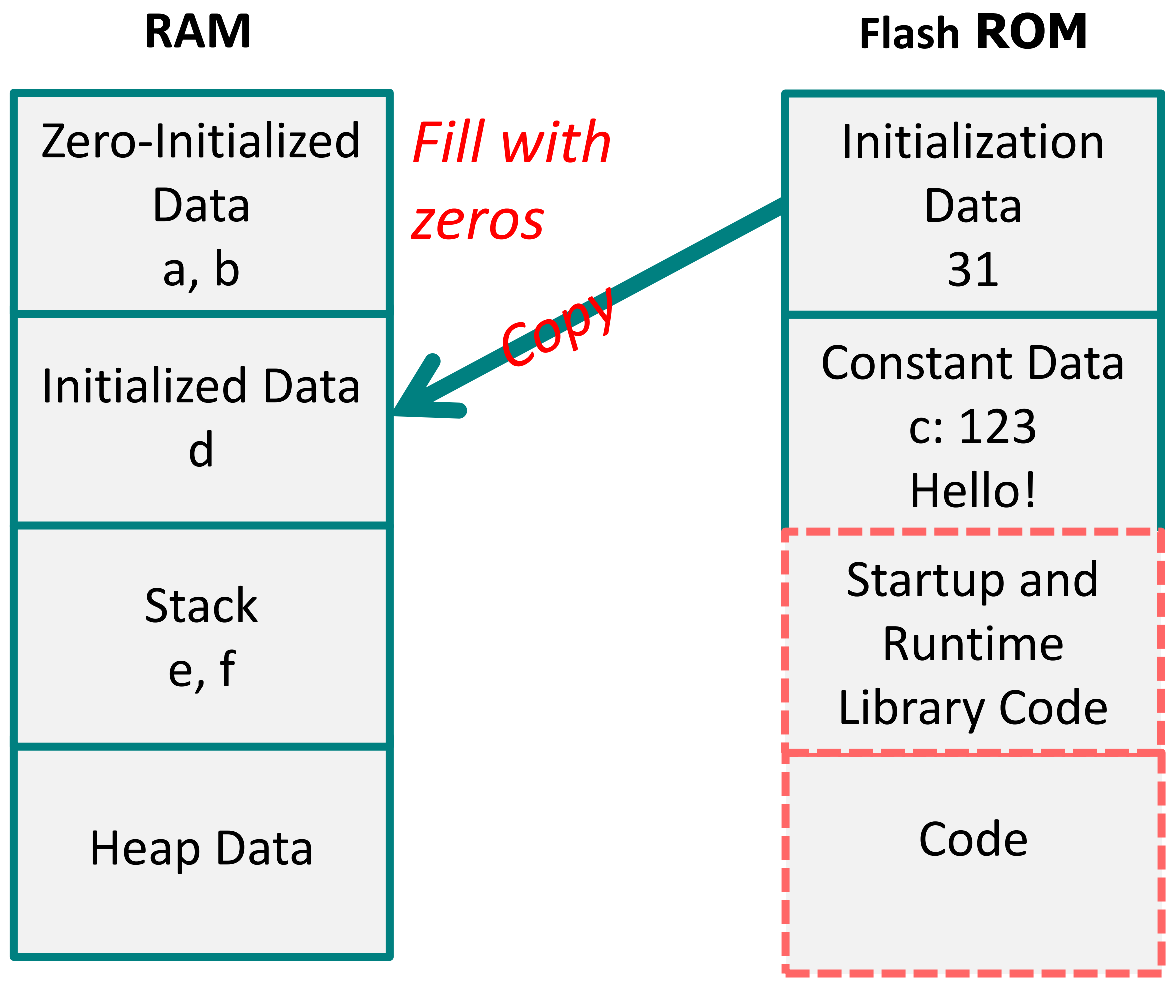
Linker Map File

- Contains extensive information on functions and variables
  - Value, type, size, object

- Cross references between sections
- Memory map of image
- Sizes of image components
- Summary of memory requirements

C Run-Time Start-Up Module

- After reset, processor must…
- Initialize hardware
  - Peripherals, etc.
  - Set up stack pointer
- Initialize C or C++ run-time environment
  - Set up heap memory
  - Initialize variables



ACCESSING DATA IN MEMORY

Accessing Data

- What does it take to get at a variable in memory?
  - Depends on location, which depends on storage type (static, automatic, dynamic)

```
int siA;
void static_auto_local() {
        int aiB;
        static int siC=3;
        int * apD;
        int aiE=4, aiF=5, aiG=6;

        siA = 2;
        aiB = siC + siA;
        apD = & aiB;
        (*apD)++;
        apD = &siC;
        (*apD) += 9;
        apD = &siA;
        apD = &aiE;
        apD = &aiF;
        apD = &aiG;
        (*apD)++;
        aiE+=7;
        *apD = aiE + aiF;
}
```

Static Variables

- Static var can be located anywhere in 32-bit memory space, so to access it, you need a 32-bit pointer
- Can't fit a 32-bit pointer into a 16-bit instruction (or a 32-bit instruction), so save the pointer separate from instruction, but nearby so we can access it with a short PC-relative offset
- Load the pointer into a register (r0)
- Can now load variable's value into a register (r1) from memory using that pointer in r0
- Similarly can store a new value to the variable in memory

```
Load r0 with pointer to variable
Load r1 from [r0]
Use value of variable

Label:
32-bit pointer to Variable



Variable
```

Static Variables

- Key
  - variable's value
  - variable's address
  - address of copy of variable's address
- Code
  - Loads r2 with address of siA (from |L1.240|)
  - Loads r1 with contents of siA (via pointer r2, with offset 0)
  - Same for siC, with address at |L1.244|

```
AREA ||.text||, CODE, READONLY, ALIGN=2
;;;20              siA = 2;
00000e   2102   MOVS      r1,#2
000010   4a37   LDR       r2,|L1.240|
000012   6011   STR       r1,[r2,#0]   ; siA
;;;21          aiB = siC + siA;
000014   4937   LDR       r1,|L1.244|
000016   6809   LDR       r1,[r1,#0]   ; siC
000018   6812   LDR       r2,[r2,#0]   ; siA
00001a   1889   ADDS      r1,r1,r2
...
```

- Addresses of siA and siC are stored as literals to be loaded into pointers

```
|L1.240|
                 DCD       ||siA||
|L1.244|
                 DCD       ||siC||
```

- Variables siC and siA are located in .data section with initial values

```
          AREA ||.data||, DATA, ALIGN=2
||siC||
                 DCD       0x00000003
||siA||
                 DCD       0x00000000
```

Automatic Variables Stored on Stack

- Automatic variables are stored in a function’s activation record (unless optimized and promoted to register)
- Activation records are located on the stack
- Calling a function creates an activation record, allocating space on stack
- Returning from a function deletes the activation record, freeing up space on stack

```
int main(void) {
    auto vars
    a();
}

void a(void) {
    auto vars
    b();
}

void b(void) {
    auto vars
    c();
}

void c(void) {
    auto vars
    …
}
```

Automatic Variables

```
int main(void)
{
    auto vars
    a();
}

void a(void) {
    auto vars
    b();
}

void b(void) {
    auto vars
    c();
}

void c(void) {
    auto vars
    …
}
```

Lower address		(Free stack space)	
	Activation record for current function C	Local storage	<- Stack pointer while executing C
		Saved regs	
		Arguments (optional)	
	Activation record for caller function B	Local storage	<- Stack pointer while executing B
		Saved regs	
		Arguments (optional)	
	Activation record for caller's caller function A	Local storage	<- Stack pointer while executing A
		Saved regs	
		Arguments (optional)	
	Activation record for caller's caller's caller function main	Local storage	<- Stack pointer while executing main
		Saved regs	
Higher address		Arguments (optional)	

Addressing Automatic Variables

- Program must allocate space on stack for variables
- Stack addressing uses an offset from the stack pointer: [sp, #offset]
  - One byte used for offset, is multiplied by four
  - Possible offsets: 0, 4, 8, …, 1020 bytes
  - Maximum range addressable this way is 1024 bytes

Address	Contents
SP	
SP+4	
SP+8	
SP+0xC	
SP+0x10	
SP+0x14	
SP+0x18	
SP+0x1C	
SP+0x20	

Example Code

```
int siA;
void static_auto_local() {
        int aiB;
        static int siC=3;
        int * apD;
        int aiE=4, aiF=5, aiG=6;

        siA = 2;
        aiB = siC + siA;
        apD = & aiB;
        (*apD)++;
        apD = &siC;
        (*apD) += 9;
        apD = &siA;
        apD = &aiE;
        apD = &aiF;
        apD = &aiG;
        (*apD)++;
        aiE+=7;
        *apD = aiE + aiF;
}
```

Automatic Variables

Address	Contents
SP	aiG
SP+4	aiF
SP+8	aiE
SP+0xC	aiB
SP+0x10	r0
SP+0x14	r1
SP+0x18	r2
SP+0x1C	r3
SP+0x20	lr

```
;;;14      void static_auto_local( void ) {
000000  b50f   PUSH  {r0-r3,lr}
;;;15  int aiB;
;;;16  static int siC=3;
;;;17  int * apD;
;;;18  int aiE=4, aiF=5, aiG=6;
000002  2104 MOVS    r1,#4
000004  9102   STR   r1,[sp,#8]
000006  2105   MOVS  r1,#5
000008  9101   STR   r1,[sp,#4]
00000a  2106   MOVS  r1,#6
00000c  9100   STR   r1,[sp,#0]
…
;;;21        aiB = siC + siA;
…
00001c  9103   STR   r1,[sp,#0xc]
```

- Initialize aiE
- Initialize aiF
- Initialize aiG
- Store value for aiB

USING POINTERS

Example Code

```
int siA;
void static_auto_local() {
        int aiB;
        static int siC=3;
        int * apD;
        int aiE=4, aiF=5, aiG=6;

        siA = 2;
        aiB = siC + siA;
        apD = & aiB;
        (*apD)++;
        apD = &siC;
        (*apD) += 9;
        apD = &siA;
        apD = &aiE;
        apD = &aiF;
        apD = &aiG;
        (*apD)++;
        aiE+=7;
        *apD = aiE + aiF;
}
```

Using Pointers to Automatic Variables

- C Pointer: a variable which holds the data's address
- aiB is on stack at SP+0xc
- Compute r0 with variable's address from stack pointer and offset (0xc)
- Load r1 with variable's value from memory
- Operate on r1, save back to variable's address

```
;;;22          apD = & aiB;
00001e  a803  ADD   r0,sp,#0xc
;;;23          (*apD)++;
000020  6801  LDR   r1,[r0,#0]
000022  1c49  ADDS  r1,r1,#1
000024  6001  STR   r1,[r0,#0]
```

Example Code

```
int siA;
void static_auto_local() {
        int aiB;
        static int siC=3;
        int * apD;
        int aiE=4, aiF=5, aiG=6;

        siA = 2;
        aiB = siC + siA;
        apD = & aiB;
        (*apD)++;
        apD = &siC;
        (*apD) += 9;
        apD = &siA;
        apD = &aiE;
        apD = &aiF;
        apD = &aiG;
        (*apD)++;
        aiE+=7;
        *apD = aiE + aiF;
}
```

Using Pointers to Static Variables

- Load r0 with variable's address from address of copy of variable's address
- Load r1 with variable's value from memory
- Operate on r1, save back to variable's address

```
;;;24               apD = &siC;
000026  4833 LDR r0,|L1.244|
;;;25               (*apD) += 9;
000028  6801      LDR    r1,[r0,#0]
00002a  3109      ADDS   r1,r1,#9
00002c  6001      STR    r1,[r0,#0]
|L1.244|
                   DCD          ||siC||
            AREA ||.data||, DATA, ALIGN=2
||siC||
                   DCD       0x00000003
```

ARRAY ACCESS

Array Access

- What does it take to get at an array element in memory?
  - Depends on how many dimensions
  - Depends on element size and row width
  - Depends on location, which depends on storage type (static, automatic, dynamic)

```
unsigned char buff2[3];
unsigned short int buff3[5][7];

unsigned int arrays(unsigned char n,
unsigned char j) {
  volatile unsigned int i;

  i = buff2[0] + buff2[n];
  i += buff3[n][j];

  return i;
}
```

Accessing 1-D Array Elements

- Need to calculate element address, that is sum of:
  - array start address
  - offset: index * element size
- buff2 is array of **unsigned characters**
- Move n (argument) from r0 into r2
- Load r3 with pointer to buff2
- Load (byte) r3 with first element of buff2
- Load r4 with pointer to buff2
- Load (byte) r4 with element at address buff2+r2
  - r2 holds argument n
- Add r3 and r4 to form sum

Address	Contents
buff2	buff2[0]
buff2 + 1	buff2[1]
buff2 + 2	buff2[2]

```
;;;74 unsigned int arrays(unsigned char
n, unsigned char j) {
00009e  4602      MOV     r2,r0
;;;75  volatile unsigned int i;
;;;76    i = buff2[0] + buff2[n];
0000a0  4b1b      LDR     r3,|L1.272|
0000a2  781b      LDRB    r3,[r3,#0]
0000a4  4c1a      LDR     r4,|L1.272|
0000a6  5ca4      LDRB    r4,[r4,r2]
0000a8  1918      ADDS    r0,r3,r4
|L1.272|
                  DCD     buff2
```

Accessing 2-D Array Elements

`short int buff3[5][7]`

Address	Contents
buff3	buff3[0][0]
buff3+1	
buff3+2	buff3[0][1]
buff3+3	
(etc.)	
buff3+10	buff3[0][5]
buff3+11	
buff3+12	buff3[0][6]
buff3+13	
buff3+14	buff3[1][0]
buff3+15	
buff3+16	buff3[1][1]
buff3+17	
buff3+18	buff3[1][2]
buff3+19	
(etc.)	
buff3+68	buff3[4][6]
buff3+69	

- var[rows][columns]
- Sizes
  - Element: **2 bytes**
  - Row: 7*2 bytes = 14 bytes (0xe)
- Offset based on row index and column index
  - column offset = column index * element size
  - row offset = row index * row size

Code to Access 2-D Array

- Load r3 with row size
- Multiply by row number (n, r2) to put row offset in r3
- Load r4 with address of buff3
- Add buff3 address to row offset in r3
- Shift column number (j is mapped to r1) left by one
  - Which is multiplying by 2 (bytes/element)
- Load (halfword) r3 with element at address r3+r4 (buff3 + row offset + col. offset)
- Add r3 into variable i (variable i is mapped to r0)

```
;;;77          i += buff3[n][j];
0000aa   230e    MOVS   r3,#0xe
0000ac   4353    MULS   r3,r2,r3

0000ae   4c19    LDR    r4,|L1.276|

0000b0   191b    ADDS   r3,r3,r4

0000b2   004c    LSLS   r4,r1,#1

0000b4   5b1b    LDRH   r3,[r3,r4]

0000b6   1818    ADDS   r0,r3,r0

|L1.276|
                 DCD    buff3
```

FUNCTION PROLOG AND EPILOG

Prolog and Epilog

- A function's P&E are responsible for creating and destroying its activation record
- Remember AAPCS
  - **Scratch** registers r0-r3 are not expected to be preserved upon returning from a called subroutine, can be overwritten
  - **Preserved** ("variable") registers r4-r8, r10-r11 must have their original values upon returning from a called subroutine
  - Prolog must **save** preserved registers on stack
  - Epilog must **restore** preserved registers from stack
- Prolog also may
  - Handle function arguments
  - Allocate temporary storage space on stack (subtract from SP)
- Epilog
  - May deallocate stack space (add to SP)
  - Returns control to calling function

Return Address

- Return address stored in LR by bl, blx instructions
- Consider case where a() calls b() which calls c()
  - On entry to b(), LR holds return address in a()
  - When b() calls c(), LR will be overwritten with return address in b()
  - After c() returns, b() will have lost its return address
- If a function calls a subroutine,
  - Yes-> must save and restore LR on stack just like other preserved registers, but LR value is popped into PC rather than LR
  - No-> don't need to save or restore LR, as it will not be modified

Function Prolog and Epilog

- Save r4 (preserved register) and link register (return address)
- Allocate 32 (0x20) bytes on stack for array x by subtracting from SP
- Compute return value, placing in return register r0
- Deallocate 32 bytes from stack
- Pop r4 (preserved register) and PC (return address)

```
 fun4 PROC
;;;102  int fun4(char a, int b, char c) {
00010a  b510      PUSH   {r4,lr}
;;;103    volatile int x[8];
00010c  b088      SUB    sp,sp,#0x20
...

;;;106             return a+b+c;
00011c  1858      ADDS   r0,r0,r1
00011e  1880      ADDS   r0,r0,r2
;;;107    }
000120  b008      ADD    sp,sp,#0x20

000122  bd10      POP    {r4,pc}
                  ENDP
```

Activation Record Creation by Prolog

Smaller address	space for x[0]		**<- 3. SP after sub sp,sp,#0x20**
	space for x[1]		
	space for x[2]		
	space for x[3]	Array x	
	space for x[4]		
	space for x[5]		
	space for x[6]		
	space for x[7]		
	lr	Return address	**<- 2. SP after push {r4,lr}**
	r4	Preserved register	
Larger address		Caller's stack frame	**<- 1. SP on entry to function, before push {r4,lr}**

Activation Record Destruction by Epilog

Smaller address	space for x[0]	Array x	<- 1. SP before add sp,sp,#0x20
	space for x[1]		
	space for x[2]		
	space for x[3]		
	space for x[4]		
	space for x[5]		
	space for x[6]		
	space for x[7]		
	lr	Return address	<- 2. SP after add sp,sp,#20
	r4	Preserved register	
Larger address		Caller's stack frame	<- 3. SP after pop {r4,pc}

CALLING FUNCTIONS

Function Arguments and Return Values

- First, pass the arguments
  - **How to pass them?**
    - Much faster to use registers than stack
    - But quantity of registers is limited
  - **Basic rules**
    - Process arguments in order they appear in source code
    - Round size up to be a multiple of 4 bytes
    - Copy arguments into core registers (r0-r3), aligning doubles to even-numbered registers
    - Copy remaining arguments onto stack, aligning doubles to even addresses
    - Specific rules in AAPCS, Section 5.5
- Second, call the function
  - Usually as subroutine with branch link (bl) or branch link and exchange instruction (blx)
  - Exceptions in AAPCS

AAPCS Core Register Use

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Return Values

- Callee passes Return Value in register(s) or stack
- Registers
- Stack
  - Caller function allocates space for return value, then passes pointer to space as an argument to callee
  - Callee stores result at location indicated by pointer

Return value size	Registers used for passing	
	Fundamental Data Type	Composite Data Type
1-4 bytes	r0	r0
8 bytes	r0-r1	stack
16 bytes	r0-r3	stack
Indeterminate size	n/a	stack

Call Example: Calling Function

```c
int fun2(int arg2_1, int arg2_2) {
  int i;
  arg2_2 += fun3(arg2_1, 4, 5, 6);
  …
}
```

```
fun2 PROC
;;;85      int fun2(int arg2_1, int
arg2_2) {
...
0000e0  2306      MOVS   r3,#6
0000e2  2205      MOVS   r2,#5
0000e4  2104      MOVS   r1,#4
0000e6  4630      MOV    r0,r6


0000e8  f7fffffe  BL     fun3

0000ec  1904      ADDS   r4,r0,r4
```

- Argument 4 into r3
- Argument 3 into r2
- Argument 2 into r1
- Argument 0 into r0
- Call fun3 with BL instruction
- Result was returned in r0, so add to r4 (arg2_2 += result)

r4 ve r6: arbitrary values

Call and Return Example

```
int fun3(int arg3_1, int arg3_2,
  int arg3_3, int arg3_4) {
  return   arg3_1*arg3_2*
           arg3_3*arg3_4;
}
```

- Save r4 and Link Register on stack
- r0 = arg3_1*arg3_2
- r0 *= arg3_3
- r0 *= arg3_4
- Restore r4 and return from subroutine
- Return value is in r0

```
 fun3 PROC
;;;81     int fun3(int arg3_1, int arg3_2, int arg3_3, int arg3_4) {

0000ba  b510      PUSH   {r4,lr}


0000c0  4348      MULS   r0,r1,r0
0000c2  4350      MULS   r0,r2,r0
0000c4  4358      MULS   r0,r3,r0

0000c6  bd10      POP    {r4,pc}
```

CONTROL FLOW

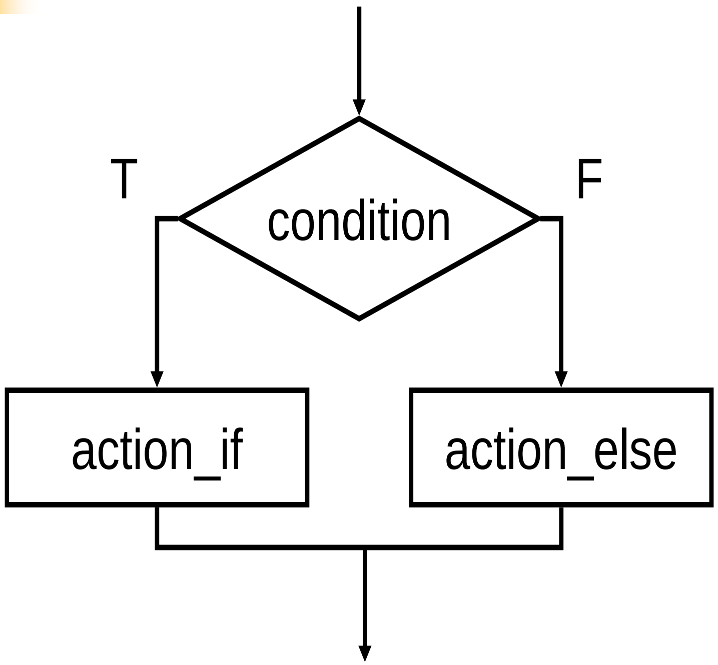
Control Flow: Conditionals and Loops

- How does the compiler implement conditionals and loops?

```
if (x){
    y++;
} else {
    y--;
}

switch (x) {
  case 1:
    y += 3;
    break;
  case 31:
    y -= 5;
    break;
  default:
    y--;
    break;
  }
```

```
while (x<10) {
    x = x + 1;
}

for (i = 0; i < 10;
i++){
    x += i;
}

do {
    x += 2;
} while (x < 20);
```

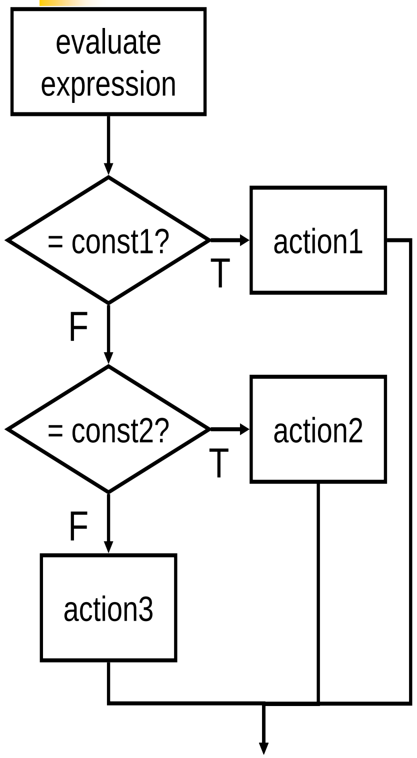
Control Flow: If/Else




```
;;;39        if (x){
000056  2900   CMP   r1,#0
000058  d001   BEQ   |L1.94|
;;;40           y++;
00005a  1c52   ADDS  r2,r2,#1
00005c  e000   B     |L1.96|

   |L1.94|
;;;41        } else {
;;;42          y--;
00005e  1e52   SUBS  r2,r2,#1

   |L1.96|
;;;43        }
```

```
if (x){
    y++;
} else {
    y--;
}
```

Control Flow: Switch

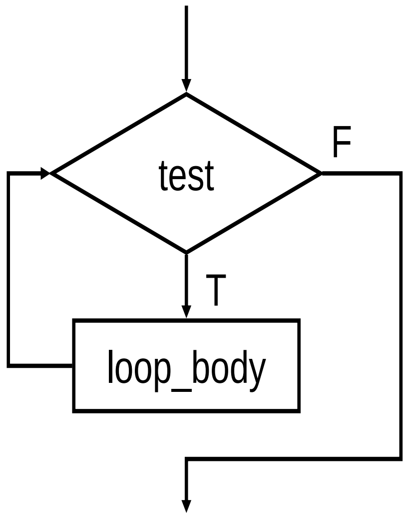



```
switch (x) {
  case 1:
    y += 3;
    break;
  case 31:
    y -= 5;
    break;
  default:
    y--;
    break;
  }
```

```
;;;45        switch (x) {
000060  2901      CMP      r1,#1
000062  d002      BEQ      |L1.106|
000064  291f      CMP      r1,#0x1f
000066  d104      BNE      |L1.114|
000068  e001      B        |L1.110|
```

```
       |L1.106|
;;;46           case 1:
;;;47             y += 3;
00006a  1cd2      ADDS     r2,r2,#3
;;;48             break;
00006c  e003      B        |L1.118|
|L1.110|
;;;49           case 31:
;;;50             y -= 5;
00006e  1f52      SUBS     r2,r2,#5
;;;51             break;
000070  e001      B        |L1.118|
|L1.114|
;;;52           default:
;;;53             y--;
000072  1e52      SUBS     r2,r2,#1
;;;54             break;
000074  bf00      NOP
|L1.118|
000076  bf00      NOP
;;;55           }
```

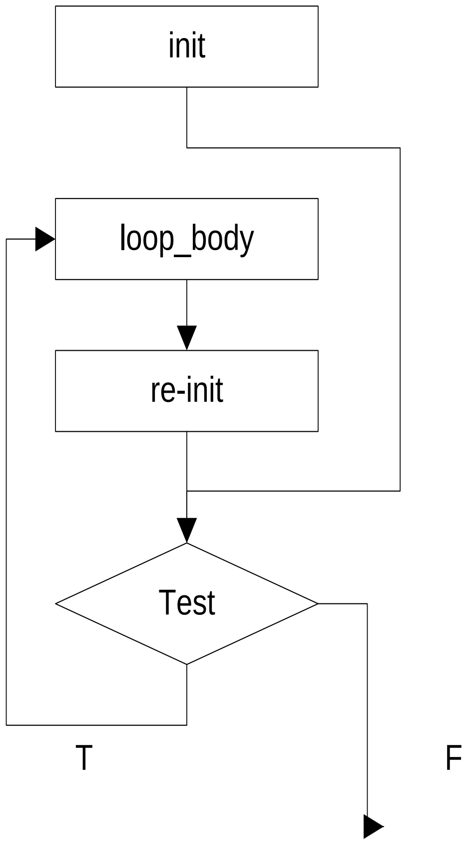
Iteration: While




```
;;;57        while (x<10) {
000078  e000     B        |L1.124|
                 |L1.122|
;;;58            x = x + 1;
00007a  1c49     ADDS   r1,r1,#1
                 |L1.124|
00007c  290a     CMP    r1,#0xa      ;57
00007e  d3fc     BCC    |L1.122|
;;;59        }
```

```
while (x<10) {
    x = x + 1;
}
```

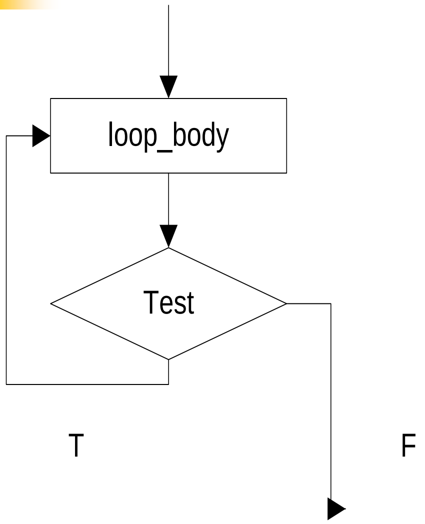
Iteration: For




```
for (i = 0; i < 10; i++){
    x += i;
}
```

```
;;;61        for (i = 0; i < 10; i++){
000080  2300     MOVS    r3,#0
000082  e001     B       |L1.136|

                 |L1.132|
;;;62               x += i;
000084  18c9     ADDS    r1,r1,r3
000086  1c5b     ADDS    r3,r3,#1     ;61

                 |L1.136|
000088  2b0a     CMP     r3,#0xa      ;61
00008a  d3fb     BCC     |L1.132|
;;;63        }
```

Iteration: Do/While




```
do {
    x += 2;
} while (x < 20);
```

```
;;;65        do {
00008c  bf00      NOP

                  |L1.142|
;;;66          x += 2;
00008e  1c89      ADDS    r1,r1,#2
;;;67        } while (x < 20);
000090  2914      CMP     r1,#0x14
000092  d3fc      BCC     |L1.142|
```

Acknowledgement

- Slides are adapted from Arm Teaching Kit

References

- Lecture Slides: Dr. Şule Gündüz Öğüdücü
- Lecture Slides: Dr. Erdem Matoğlu
- Lecture Slides: Dr. Feza Buzluca
- Lecture Slides: Dr. Bassel Soudan
- Lecture Slides: Dr. Gökhan İnce
- Lecture Slides: Dr. B.Berk Üstündağ