HOW C CODE IS *REALLY* IMPLEMENTED

Overview

- We program in C for convenience
- There are no MCUs which execute C, only machine code
- So we compile the C to 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 to
  - 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,
- 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)

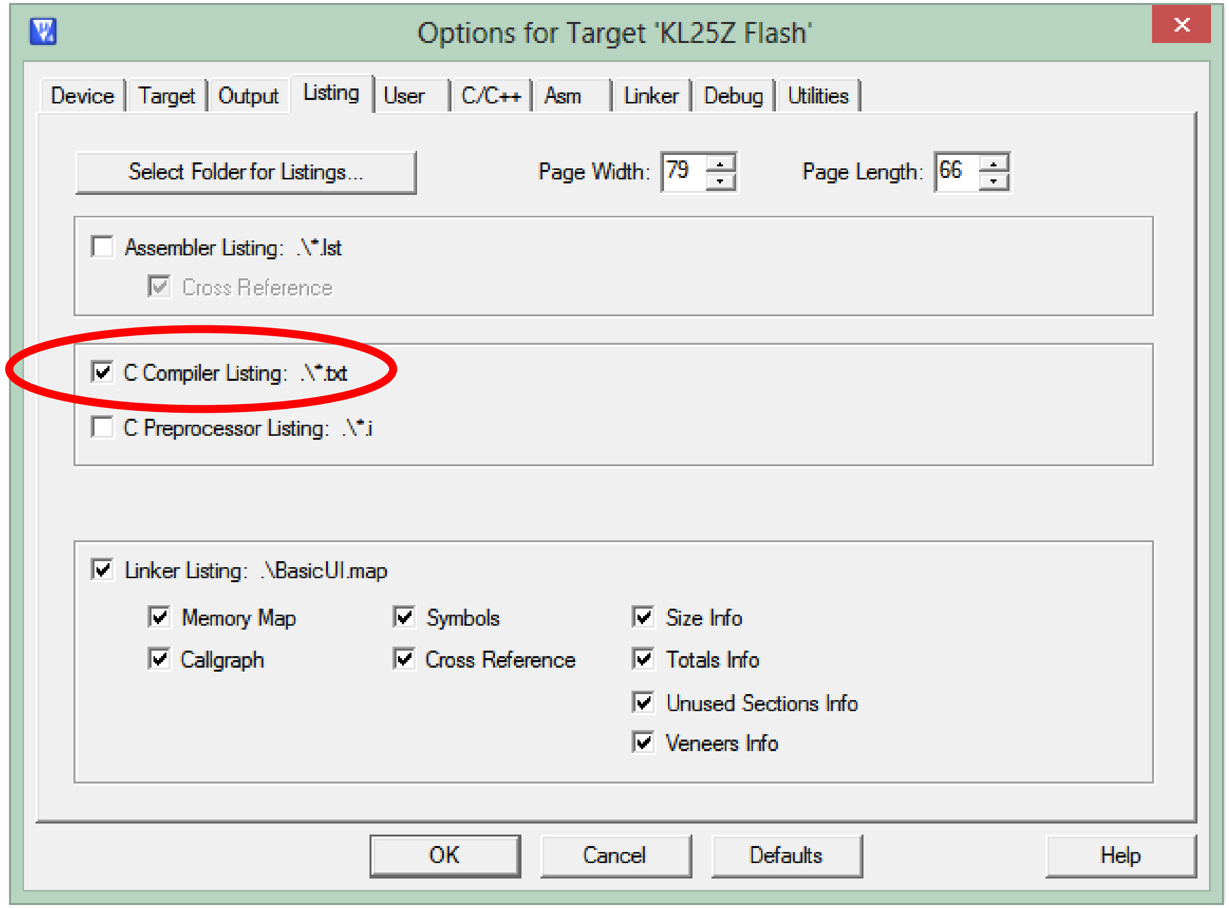
Assembler

- Assembler
  - Creates object code (machine code)

Linker/
Loader

- Linker/Loader
  - Creates executable image from one or more object file

Examining Assembly Code before Debugger



- Some compilers will generate assembly code listing for reference
  - MDK-Pro does, MDK-Lite does not
- Select in project options

Disassemble Code from Command Prompt

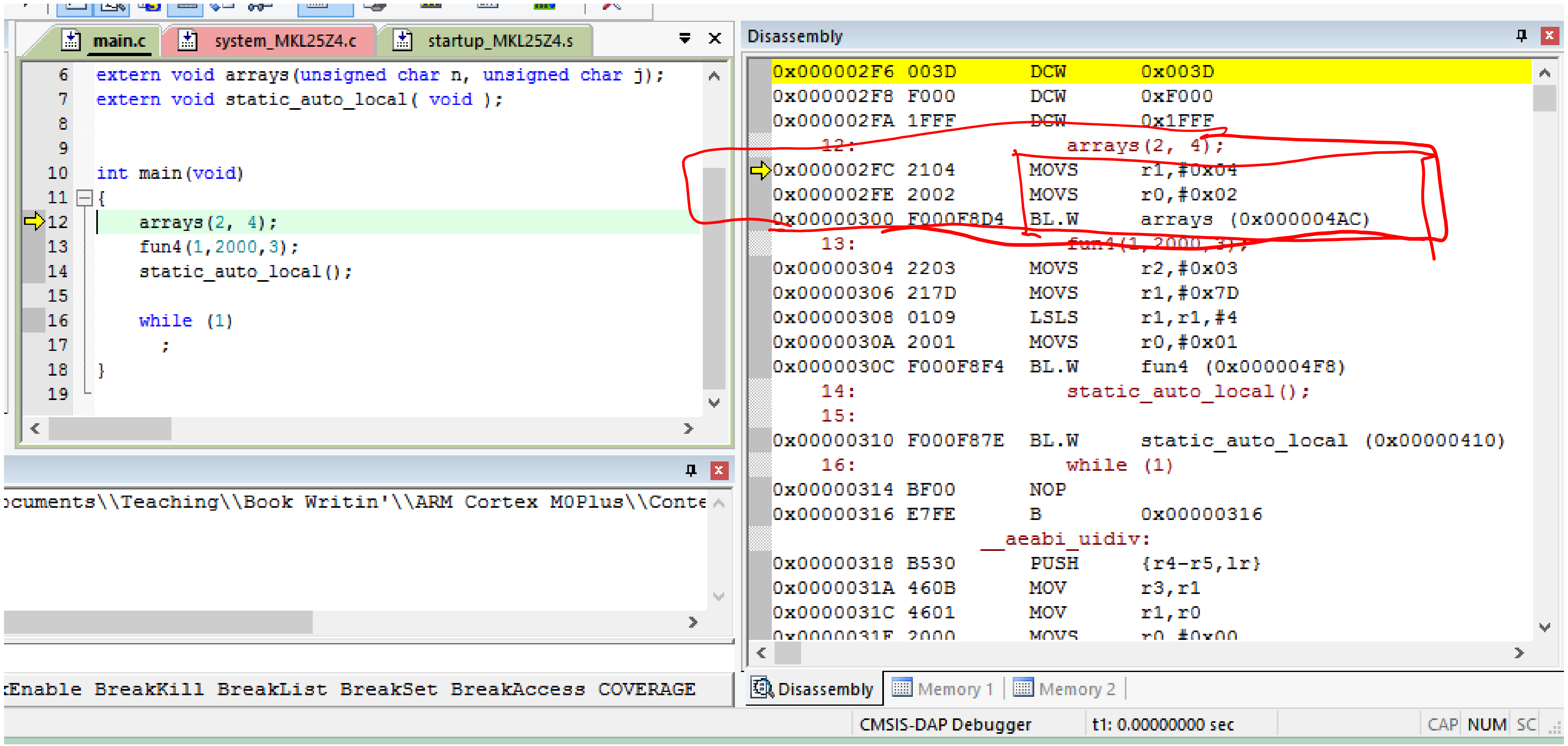
```
C:>C:\Keil_v5\ARM\ARMCC\bin\fromelf
Product: MDK-ARM Lite 5.24
Component: ARM Compiler 5.06 update 5 (build 528)
Tool: fromelf [4d35da]

ARM image conversion utility
fromelf [options] input_file

Options:
       --help          display this help screen
       --vsn           display version information
       --output file   the output file. (defaults to stdout for -text format)
       --nodebug       do not put debug areas in the output image
       --nolinkview    do not put sections in the output image

Binary Output Formats:
       --bin           Plain Binary
       --m32           Motorola 32 bit Hex
       --i32           Intel 32 bit Hex
       --vhx           Byte Oriented Hex format

       --base addr     Optionally set base address for m32,i32

Output Formats Requiring Debug Information
       --fieldoffsets Assembly Language Description of Structures/Classes
       --expandarrays Arrays inside and outside structures are expanded

Other Output Formats:
       --elf          ELF
       --text         Text Information

                Flags for Text Information
                -v          verbose
                -a          print data addresses (For images built with debug)
                -c          disassemble code
                -d          print contents of data section
                -e          print exception tables
                -g          print debug tables
                -r          print relocation information
                -s          print symbol table
                -t          print string table
                -y          print dynamic segment contents
                -z          print code and data size information
```

- Use **fromelf.exe** tool to disassemble ELF object code in .axf or .o file.
  - Located at \Keil_v5\ARM\ARMCC\bin
- Specify text output with disassembled code
  - --text -c
- Capture output in a text file
  - --output filename
  - > filename
- \Keil_v5\ARM\ARMCC\bin\fromelf --text -c HBLED_Test.axf > HBLED_Test.dis
- Could automate this in MDK after build completes

Example Output

```
i.main
main
    0x00000994:    f7fffe68    ..h.    BL      Init_Debug_Signals ; 0x668
    0x00000998:    4c28        (L      LDR     r4,[pc,#160] ; [0xa3c] = 0x40048000
    0x0000099a:    6be0        .k      LDR     r0,[r4,#0x3c]
    0x0000099c:    0421        !.      LSLS    r1,r4,#16
    0x0000099e:    4308        .C      ORRS    r0,r0,r1
    0x000009a0:    63e0        .c      STR     r0,[r4,#0x3c]
    0x000009a2:    6ba0        .k      LDR     r0,[r4,#0x38]
    0x000009a4:    0c8e        ..      LSRS    r6,r1,#18
    0x000009a6:    4330        0C      ORRS    r0,r0,r6
    0x000009a8:    63a0        .c      STR     r0,[r4,#0x38]
    0x000009aa:    4d25        %M      LDR     r5,[pc,#148] ; [0xa40] = 0x4004d040
    0x000009ac:    6ba8        .k      LDR     r0,[r5,#0x38]
    0x000009ae:    2707        .'      MOVS    r7,#7
    0x000009b0:    023f        ?.      LSLS    r7,r7,#8
    0x000009b2:    43b8        .C      BICS    r0,r0,r7
    0x000009b4:    63a8        .c      STR     r0,[r5,#0x38]
    0x000009b6:    6ba8        .k      LDR     r0,[r5,#0x38]
    0x000009b8:    63a8        .c      STR     r0,[r5,#0x38]
    0x000009ba:    4922        "I      LDR     r1,[pc,#136] ; [0xa44] = 0x4003f020
```

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

a = a + 7;

```
Load r0, [a]
add r0, #7
str r0, [a]
```

a += b;

```
load r0, [a]
Load r1, [b]
add r0, r1
str r0, [a]
```

a, b

opt level

0 No Ch.
1
2
3

Application Binary Interface

ABI defines rules which allow functions developed separately 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.

MEMORY REQUIREMENTS

What Memory Does a Program Need?

```
int a, b, d=31;
const char c=123;
void main(void) {
    int e;
    char * f;
    e = d + 7;
    f = (char *) malloc(e);
    my_strcpy(f,“Hello!”);
    free(f);
}
void my_strcpy(char * d, char * s) {
    int i=0;
    while (s[i] != ‘\0’) {
        d[i] = s[i];
        i++;
    }
    d[i] = ‘\0’;
}
```

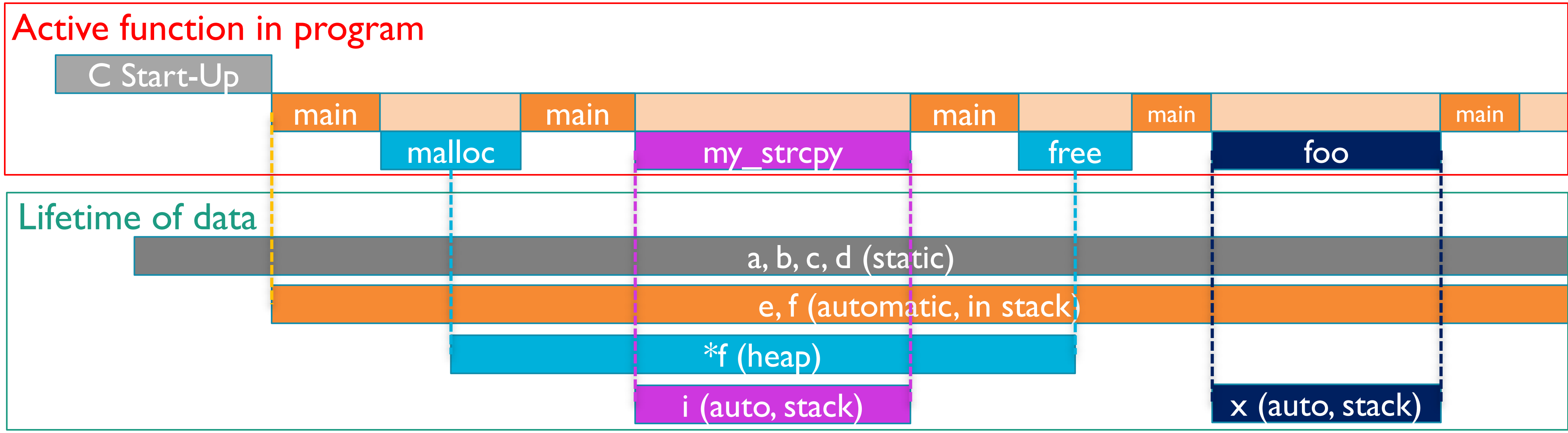
- What does the memory hold?
  - Code
  - Many kinds of data
    - Read-only static data
    - Writable static data
      - Initialized
      - Zero-initialized
      - Uninitialized
    - Heap
    - Stack
- What goes where?
  - Code is obvious – ROM! (at least initially)
  - And the others?

What Memory Does a Program Need?

```
int a, b, d=31;
const char c=123;
void main(void) {
    int e;
    char * f;
    e = d + 7;
    f = (char *) malloc(e);
    my_strcpy(f,"Hello!");
    free(f);
}
void my_strcpy(char * d, char * s) {
    int i=0;
    while (s[i] != '\0') {
        d[i] = s[i];
        i++;
    }
    d[i] = '\0';
}
```

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

Lifetime: When Does the Data Exist?




```
int a, b, d=31;
const char c=123;
void main(void) {
   int e;
   char * f;
   e = d + 7;
   f = (char *) malloc(e);
   my_strcpy(f,"Hello!");
   free(f);
}
void my_strcpy(char * d, char * s){
   int i=0;
   while (s[i] != '\0') {
      d[i] = s[i];
      i++;
   }
   d[i] = '\0';
}
```

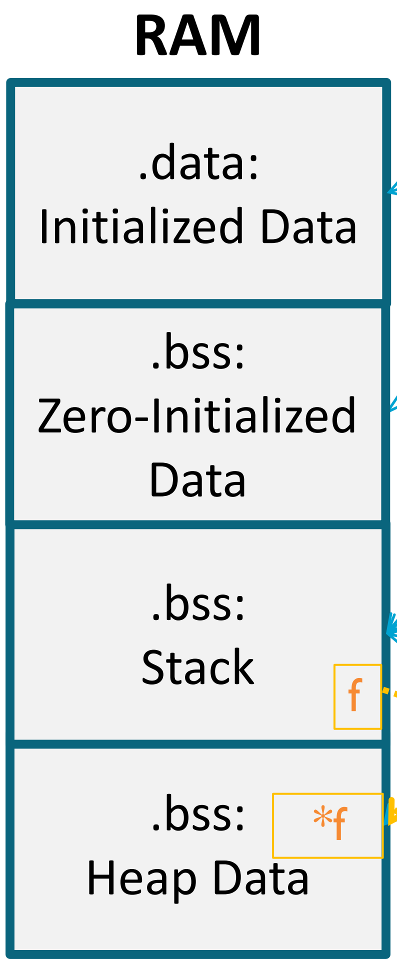
- **Statically allocated: a, b, c, d**
  - Exists from **program start to end**
  - Each variable has its own fixed location
  - Space is not reused
- **Automatically allocated: e, f, i**
  - Exists from **function start to end**
  - Space can be reused
- **Dynamically allocated: *f (allocated space in heap)**
  - Exists from explicit allocation to explicit deallocation
  - Space can be reused

Type and Class Qualifiers for Variables

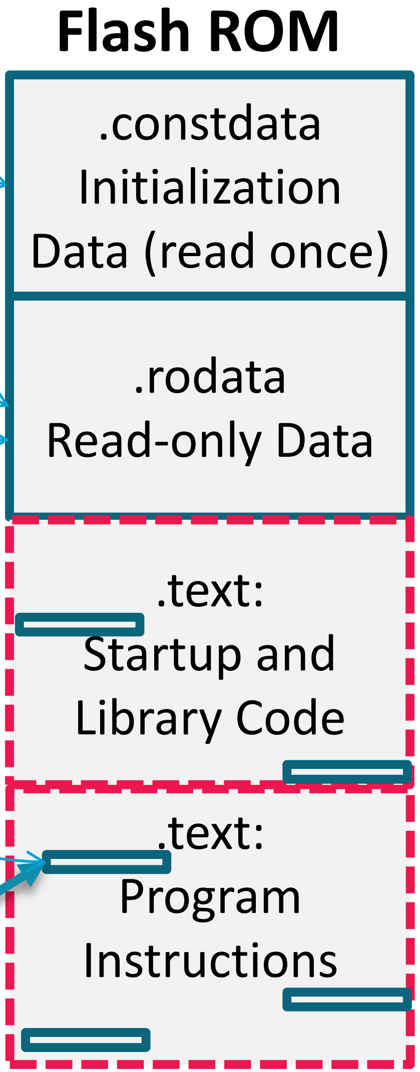
- Modify a variable's declaration so compiler treats it differently

- Const
  - Never written by program, can be put in ROM to save RAM
- Volatile
  - Can be changed outside of normal program flow: by ISR, or variable is a hardware register
  - Compiler must be careful with optimizations. Access memory each time variable appears in program source code
- Static
  - Declared within function, retains value between function invocations
  - Scope is limited to function

Basics of Data Memory Used by Program




```c
int a, b, d=31;
const char c=123;
void main(void) {
    int e;
    char * f;
    e = d + 1234567;
    f = (char *) malloc(e);
    my_strcpy(f,“Hello!”);
    free(f);
}
void my_strcpy(char * d, char * s) {
    int i=0;
    while (s[i] != ‘\0’) {
        d[i] = s[i];
        i++;
    }
    d[i] = ‘\0’;
}
```



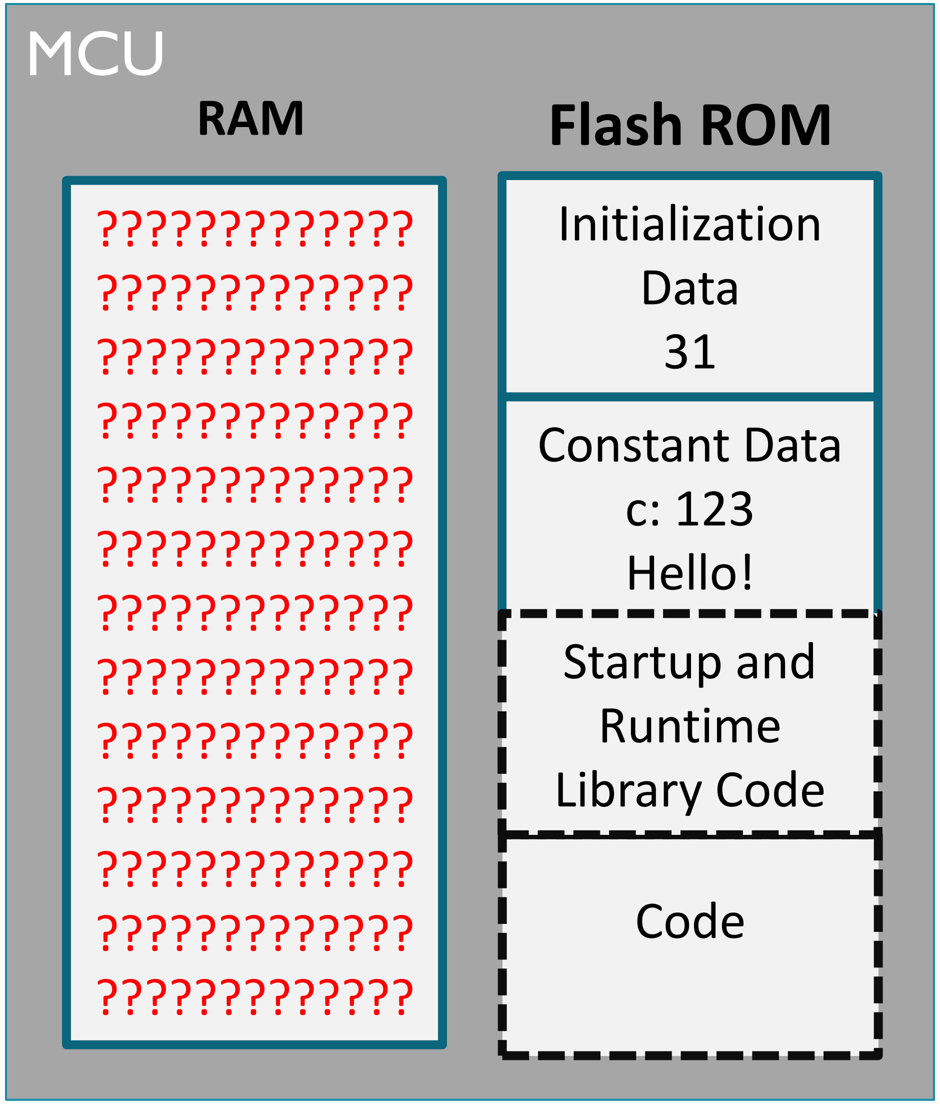
Literal pool of constants at end of function code

Downloading Program to MCU

- Download program to MCU
  - Linker creates object image file (AXF) describing how to program the MCU memory
  - Download operation programs flash ROM as directed by AXF

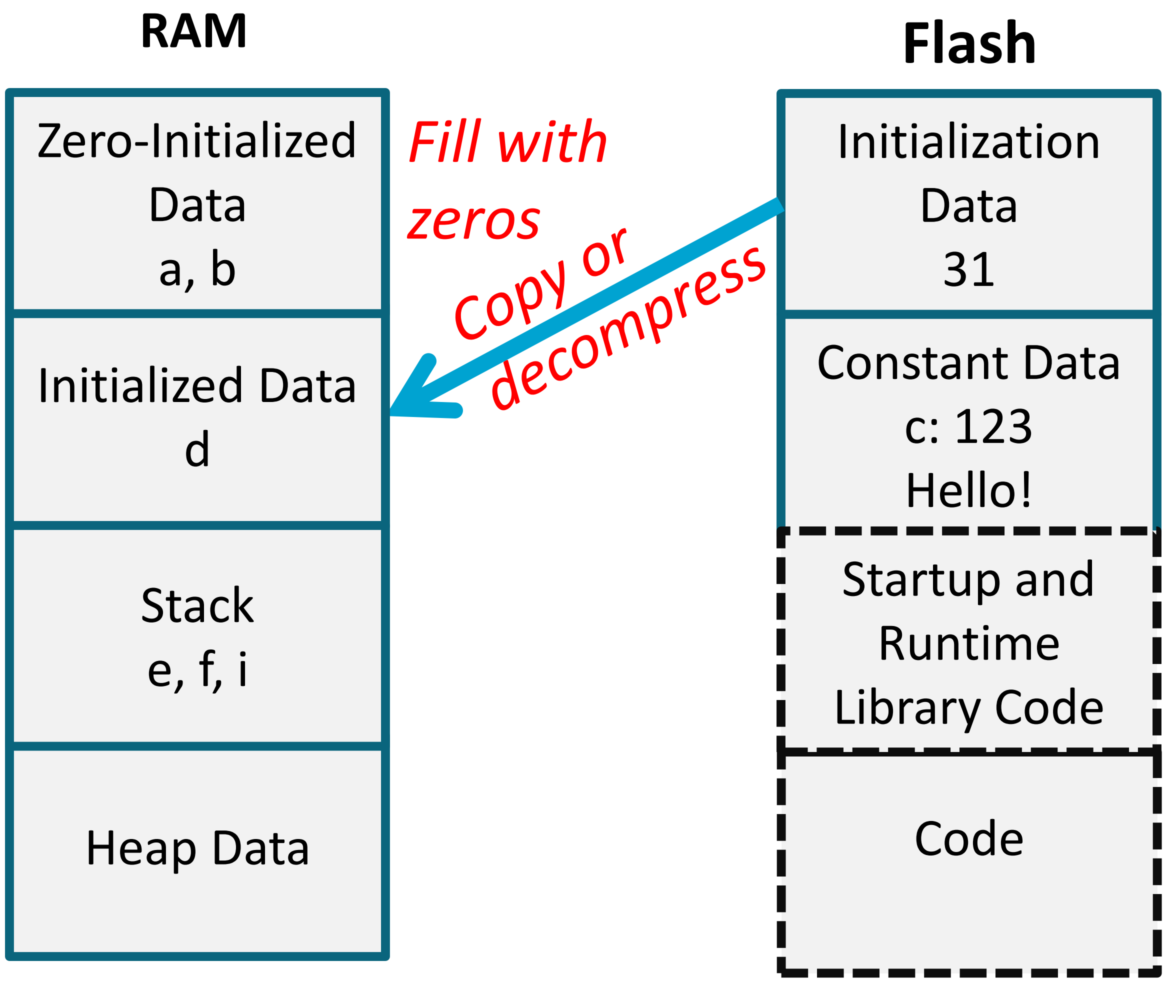
- On start-up (after reflash or power-up), RAM contents are unknown

- Reset interrupt handler runs start-up code, where MCU must initialize RAM and some key hardware

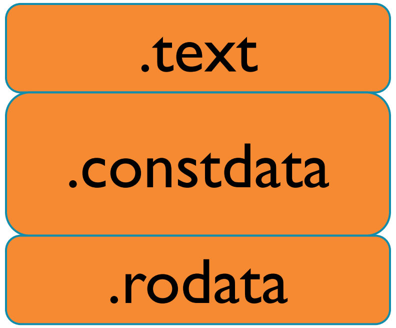


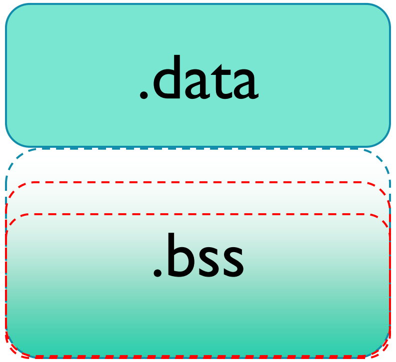
C Run-Time Start-Up Module

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



Executable File Sections (ELF, AXF)





- ROM (RO = Read Only)
  - **.text**
    - Program code (instructions)
    - Never changes, is read directly from ROM by program
    - Exact size known at build time
  - **.constdata**
    - Initialization data for variables
    - Is copied from ROM to RAM on system start-up
    - Exact size known at build time
  - **.rodata**
    - Read-only (*const*-qualified) data
    - Will not change, is read directly from ROM by program
    - Exact size known at build time

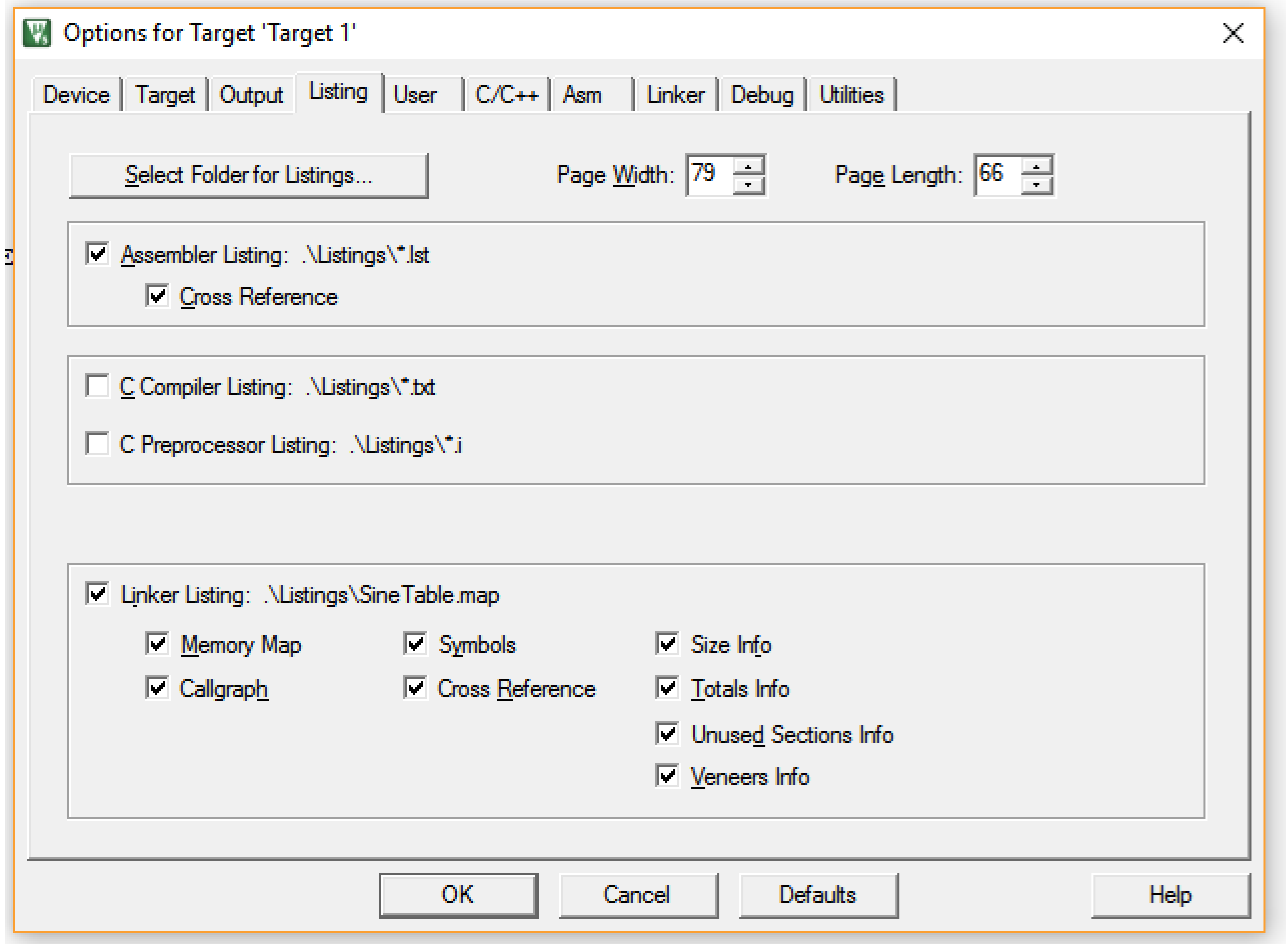
- RAM (RW = Read/Write, ZI = Zero-Initialized)
  - **.data**
    - Holds variables which have been initialized
    - Is loaded from ROM to RAM on system start-up
    - Exact size known at build time
  - **.bss**
    - Uninitialized data, stack, heap
    - Is cleared to zero on start-up before main() begins
    - **Exact size not known at build time** for non-trivial programs since stack and heap depend on program behavior, input data

Linker Map File

- Configure Linker to generate map file (Linker Listing)

- Map file contents
  - 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

- Examine a sample map file



Map File Contents

```
Global Symbols

Symbol Name                              Value      Ov Type        Size   Object(Section)

SystemInit                               0x000000f1    Thumb Code   164   system_mkl25z4.o(.
SystemCoreClockUpdate                    0x00000195    Thumb Code   310   system_mkl25z4.o(.
main                                     0x000002fd    Thumb Code   124   main.o(.text)
Delay                                    0x00000385    Thumb Code    24   delay.o(.text)
__aeabi_uidiv                            0x000003a1    Thumb Code     0   uidiv.o(.text)
__aeabi_uidivmod                         0x000003a1    Thumb Code    44   uidiv.o(.text)
__aeabi_i2d                              0x000003cd    Thumb Code    34   dflti.o(.text)
__I$use$fp                               0x000003f5    Thumb Code     0   iusefp.o(.text)
__scatterload_null                       0x000003f5    Thumb Code     2   handlers.o(i.__sca
__mathlib_zero                           0x000003f8    Data           8   qnan.o(.constdata)
Init_RGB_LEDs                            0x00000411    Thumb Code   124   leds.o(.text)
Control_RGB_LEDs                         0x0000048d    Thumb Code    70   leds.o(.text)
i2c_init                                 0x000004e5    Thumb Code    74   i2c.o(.text)
i2c_start                                0x0000052f    Thumb Code    26   i2c.o(.text)
```

- Map file shows how memory is used
  - Symbol table, memory map, image component sizes
- We might care about function and data sizes

Summary Size Information

```
      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

      5440        342        384         24       1024      12103   Grand Totals
      5440        342        384         24       1024      12103   ELF Image Totals
      5440        342        384         24          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                 5824 (   5.69kB)
    Total RW  Size (RW Data + ZI Data)              1048 (   1.02kB)
    Total ROM Size (Code + RO Data + RW Data)       5848 (   5.71kB)

==============================================================================
```

- ELF image includes zero-initialized data and debug information, not included in ROM
- Sizes: 5824 bytes/1024 bytes per kB = 5.69 kB

Per-Module Information

in ROM   in RAM

Code	(inc. data)	RO Data	RW Data	ZI Data	Debug	Object Name
28	4	0	0	0	764	delay.o
604	12	0	0	0	1559	i2c.o
212	18	0	0	0	773	leds.o
136	12	0	0	0	479	main.o
528	36	0	16	0	1443	mma8451.o
44	24	192	0	1024	840	startup_mkl25z4.o
552	60	0	4	0	5505	system_mkl25z4.o

Code	(inc. data)	RO Data	RW Data	ZI Data	Debug	Library Member Name
544	70	152	0	0	112	atan.o
376	40	0	0	0	144	atan2.o
20	6	0	0	0	68	dunder.o
44	6	0	0	0	60	fpclassify.o
172	0	0	0	0	76	poly.o
0	0	8	0	0	0	qnan.o
72	6	0	0	0	76	sqrt.o

- Includes both **compiled** modules and **library** modules which were linked in
- Includes padding to align symbols

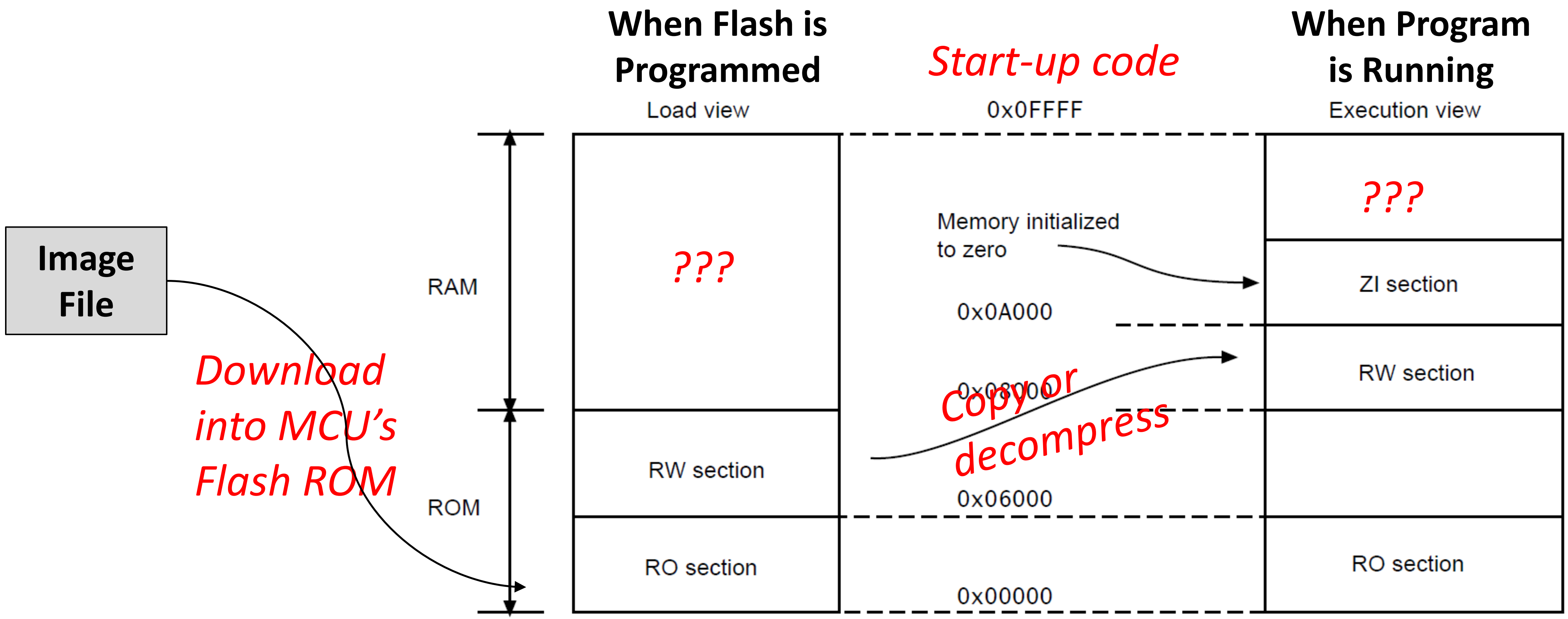
- What's **Code (inc. data)**?
  - Example: the code segment for delay.o is 28 bytes long
    - 24 bytes for instructions
    - 4 bytes ("inc. data") are actually a *Literal Pool* holding constant data (not instructions)
  - Later you'll see some data is stored in code segment for easy access (with PC-relative addressing)

Compression

```
==============================================================================


      Code (inc. data)   RO Data    RW Data    ZI Data      Debug

     23824       1386       2704       5328       2088     437012   Grand Totals
     23824       1386       2704        136       2088     437012   ELF Image Totals (compressed)
     23824       1386       2704        136          0          0   ROM Totals

==============================================================================

    Total RO  Size (Code + RO Data)                26528 (  25.91kB)
    Total RW  Size (RW Data + ZI Data)              7416 (   7.24kB)
    Total ROM Size (Code + RO Data + RW Data)      26664 (  26.04kB)

==============================================================================
```

- Why is RW Data smaller in ELF Image Totals (compressed) and ROM Totals?
- Why is ZI Data zero in ROM Totals?
- Which values are used to determine Total RO, RW, ROM Sizes?

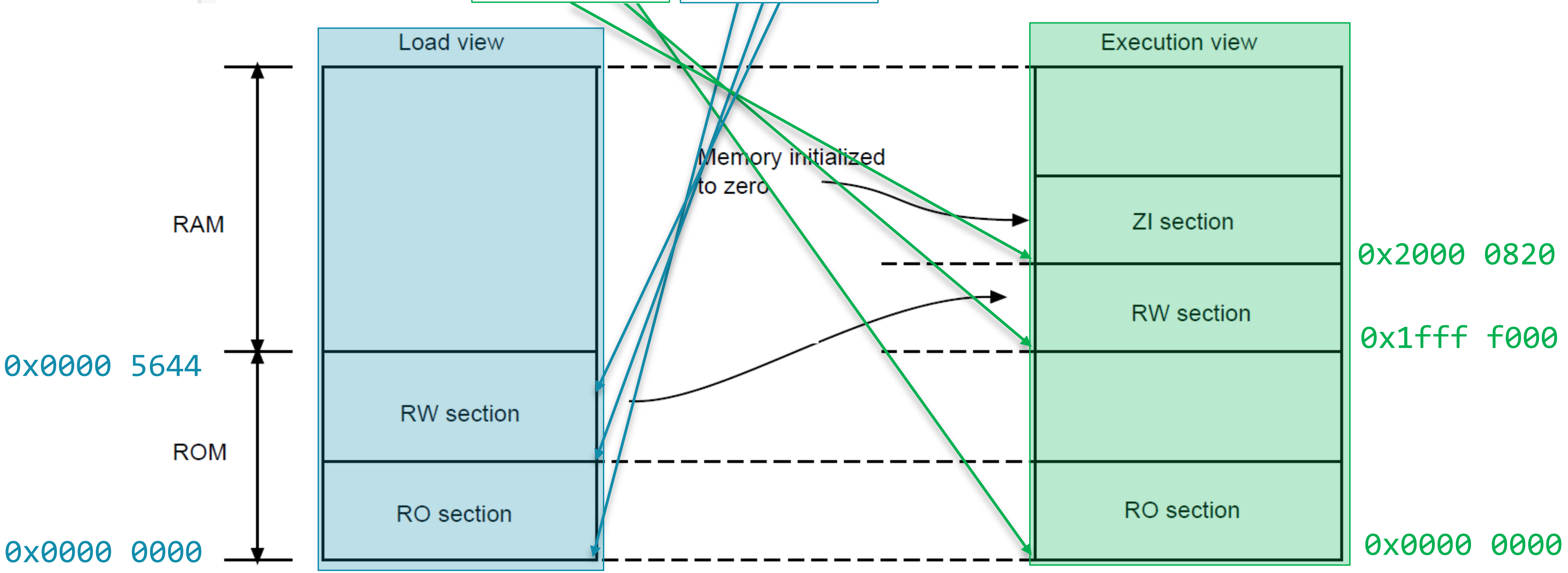
Views (Load and Execution) and Compression



- Image (ELF/AXF) file describes **Load View** – how download tool should program MCU memory
  - RW sections may be compressed to save space
  - No contents stored for ZI sections
- Start-up code initializes RAM to create **Execution View**
  - Zeros out ZI, copies or decompresses RW section

Regions in Map File

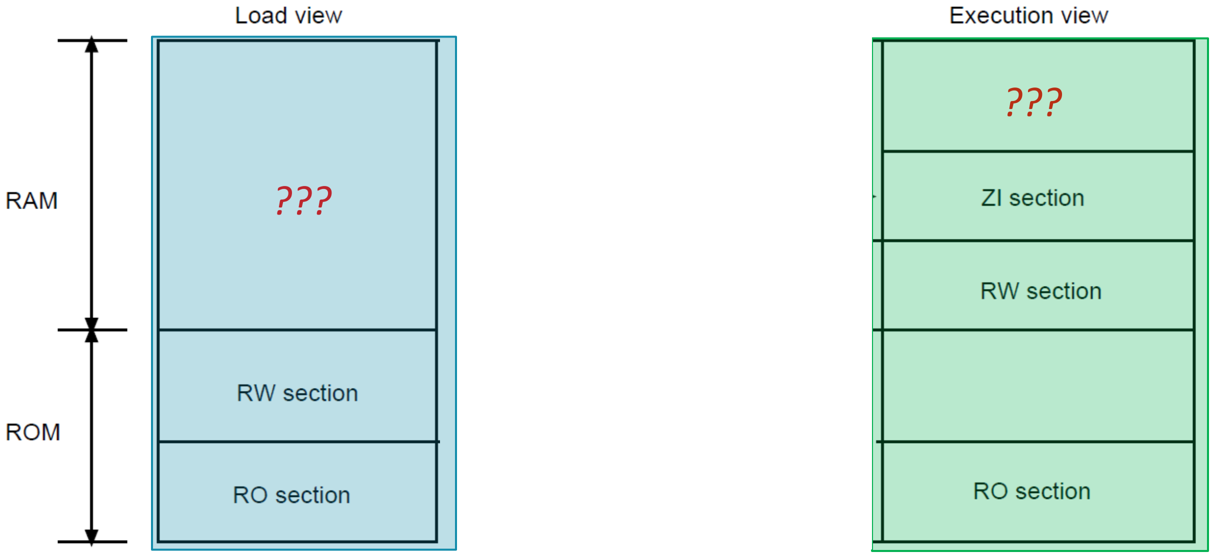
```
==============================================================================

Memory Map of the image

  Image Entry point : 0x000000d5

  Load Region LR_1 (Base: 0x00000000, Size: 0x00006d08, Max: 0xffffffff, ABSOLUTE, COMPRESSED[0x00005644])

    Execution Region ER_RO (Exec base: 0x00000000, Load base: 0x00000000, Size: 0x000054e8, Max: 0xffffffff, ABSOLUTE)

    Execution Region ER_RW (Exec base: 0x1ffff000, Load base: 0x000054e8, Size: 0x00001820, Max: 0xffffffff, ABSOLUTE, COMPRESSED[0x0000015c])

    Execution Region ER_ZI (Exec base: 0x20000820, Load base: 0x00005644, Size: 0x00001368, Max: 0xffffffff, ABSOLUTE)
```



Sections within Regions

- Map file describes Regions
- Regions contain Sections
- Sections contain code and data
  - "Thumb Code" symbols
  - "Data" symbols

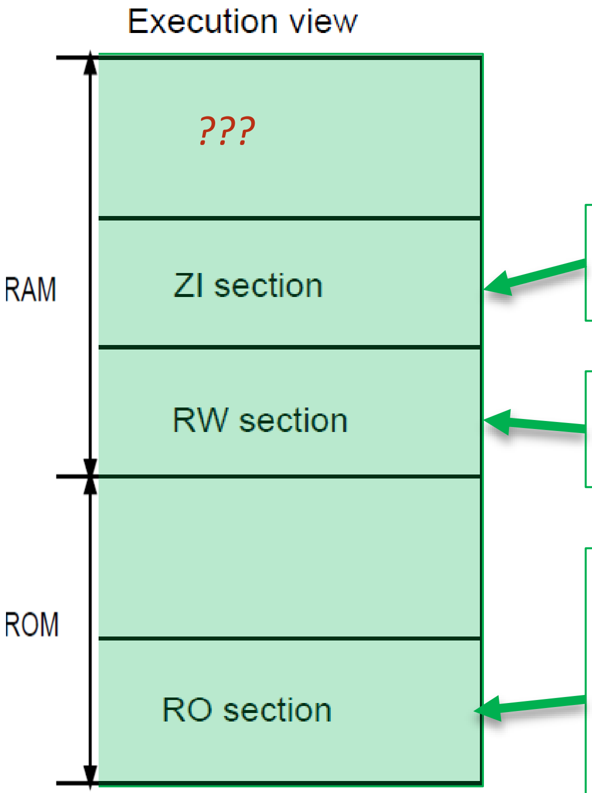
```
==============================================================================

Memory Map of the image

  Image Entry point : 0x000000d5

  Load Region LR_1 (Base: 0x00000000, Size: 0x00006d08, Max: 0xffffffff, ABSOLUTE, COMPRESSED[0x00005644])

    Execution Region ER_RO (Exec base: 0x00000000, Load base: 0x00000000, Size: 0x000054e8, Max: 0xffffffff, ABSOLUTE)

    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name            Object

    0x00000000   0x00000000   0x000000c0   Data   RO        1312     RESET                   startup_mkl25z4.o
    0x000000c0   0x000000c0   0x00000000   Code   RO        1828   * .ARM.Collect$$$$00000000  mc_p.l(entry.o)
    0x000000c0   0x000000c0   0x00000004   Code   RO        2153     .ARM.Collect$$$$00000001  mc_p.l(entry2.o)

    Execution Region ER_ZI (Exec base: 0x20000820, Load base: 0x00005644, Size: 0x00001368, Max: 0xffffffff, ABSOLUTE)

    Exec Addr    Load Addr    Size         Type   Attr      Idx    E Section Name            Object

    0x20000820   0x00005644   0x000000e0   PAD
    0x20000900        -       0x00000f80   Zero   RW          16     .bss                    control.o
    0x20001880        -       0x00000001   Zero   RW        1811     .bss.PendST             RTX_CM0.lib(os_systick.o)
    0x20001881   0x00005644   0x00000007   PAD
    0x20001888        -       0x00000300   Zero   RW        1310     STACK                   startup_mkl25z4.o
```



Code and Data within Sections

Image Symbol Table

Global Symbols



Symbol Name	Value	Ov	Type	Size	Object(Section)
TxQ	0x200004d0		Data	268	uart.o(.bss)
RxQ	0x200005dc		Data	268	uart.o(.bss)
SystemCoreClock	0x20000424		Data	4	system_mkl25z4.o(.data)
osRtxInfo	0x2000042c		Data	164	rtx_kernel.o(.data.os)
Read_TS_attr	0x00005dd0		Data	36	threads.o(.constdata)
Delay	0x000000d5		Thumb Code	18	delay.o(.text)
ShortDelay	0x000000e7		Thumb Code	14	delay.o(.text)
Init_RGB_LEDs	0x000000f9		Thumb Code	84	leds.o(.text)
Control_RGB_LEDs	0x0000014d		Thumb Code	50	leds.o(.text)

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, or dynamic?
  - We'll examine object code generated for this sample function

```
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 – Fixed Address

- Static var can be located anywhere in 32-bit memory space, so 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 value separate from instruction but nearby (in literal pool at end of function's code)
- 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 in literal pool:
32-bit pointer to Variable



Variable
```

Static Variables

CPU r2 LP L1.240 data ||siC|| ||siA||

- Key
  - variable's value
  - variable's address
  - address of copy of variable's address
- Code
  - Loads r2 with address of siA (from |L1.240| in literal pool)
  - Loads r1 with contents of siA (via pointer r2, with offset 0)
  - Same for siC, with address at |L1.244| in literal pool
- Addresses of siA and siC are stored as in literal pool as literal values to be loaded into pointers
- Variables siC and siA are located in .data section with initial values

Code in ROM .text area

```
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
...
```

Data in ROM .text area (literal pool)

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

Data in RAM .data area

```
          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	
	Arguments (optional)	

Higher address

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
  - Maximum range addressable this way is 1024 bytes

sp-4 unused

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

Last used

Full: SP → Data
Descending: grows toward smaller addresses

Automatic Variables

Address	Contents
SP	aiG 6
SP+4	aiF 5
SP+8	aiE 4
SP+0xC	aiB ?
SP+0x10	r0
SP+0x14	r1
SP+0x18	r2
SP+0x1C	r3
SP+0x20	lr

sp

missing sp -= 16 for alloc.

```
;;;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 = …
…
00001c  9103  STR   r1,[sp,#0xc]
```

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

USING POINTERS

Using Pointers to Automatics

- 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]
```

Using Pointers to Statics

- Load r0 with variable's address from location |L1.244| in literal pool which holds address of ||siC|| (is a pointer to ||siC||)
- Load r1 with value of ||siC|| from memory at address of ||siC||
- 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]
```
Code in ROM .text area

```
|L1.244|
                  DCD       ||siC||
```
Data in ROM .text area

```
          AREA ||.data||, DATA, ALIGN=2
||siC||
                  DCD       0x00000003
```
Data in RAM .data area

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 Static 1-D Array Elements

r3
r4

- Need to calculate element address: sum of…
  - array start address
  - offset: index * element size
- buff2 is static array of unsigned characters

1 byte each

- Move n (argument) from r0 into r2
- Load r3 with address of buff2 from location in literal pool
- Load (byte) r3 with first element of buff2
- Load r4 with address of buff2 from location in literal pool
- Load (byte) r4 with element at address buff2+r2
  - r2 holds argument n
- Add r3 and r4 to form sum (in r0)

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

Code in ROM .text area

```
4602    MOV   r2,r0
;;;76      i = buff2[0] + buff2[n];
4b1b    LDR   r3,|L1.272|
781b    LDRB  r3,[r3,#0]  ;buff2
4c1a    LDR   r4,|L1.272|
5ca4    LDRB  r4,[r4,r2]

1918    ADDS  r0,r3,r4
```

Data in ROM .text area

```
|L1.272|
        DCD   buff2
```

Accessing Static 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	

row 0

row 1

- 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 Static 2-D Array

- Load r3 with row size
- Multiply by row number n to put row offset in r3
- Load r4 with address of buff3 from location in literal pool
- Add buff3 address to row offset in r3 to get address of buff3's n^{th} row
- Shift column number (j, in r1) left by one, multiplying by 2 (bytes/element) to get column offset
- Load (halfword) r3 with element at address r3+r4 (buff3 + row offset + col. offset)
- Add r3 into variable i (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
```

Code in ROM .text area

```
|L1.276|
                DCD   buff3
```

Data in ROM .text area

FUNCTION CALL STACK

Stack and Activation Records

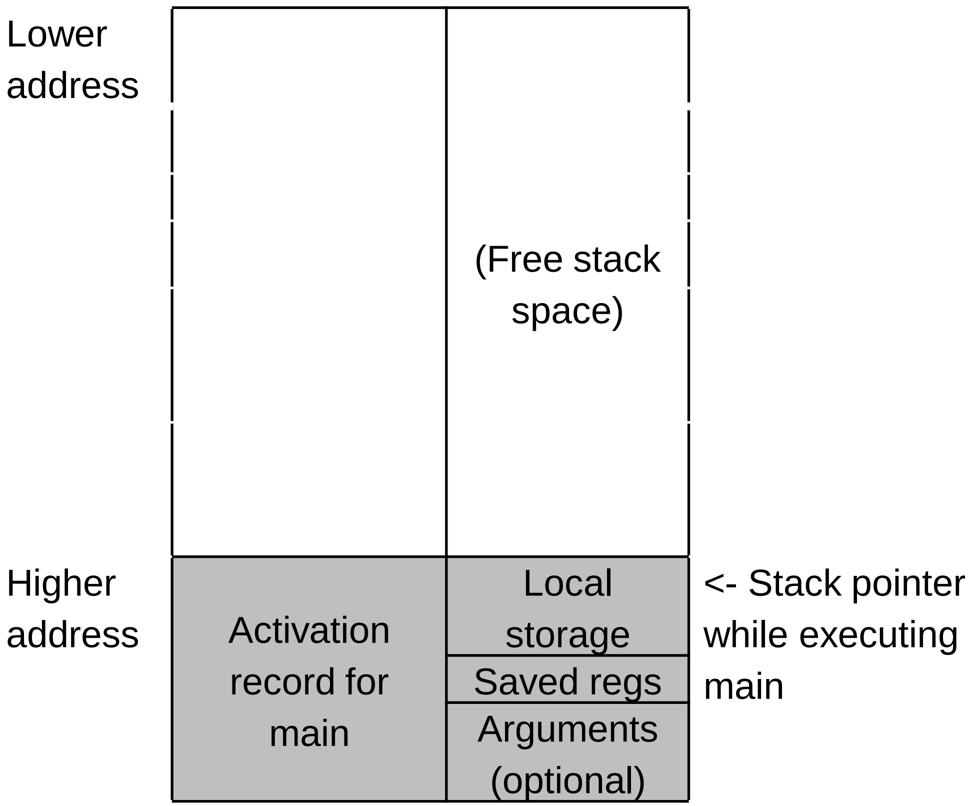
- Activation records are located on the **stack**
  - Calling a function creates an activation record
  - Returning from a function deletes the activation record
- **Automatic variables** and **housekeeping information** are stored in a function's activation record
- Not all fields (LS, RA, Arg) may be present for each activation record

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

Main before Calling function a

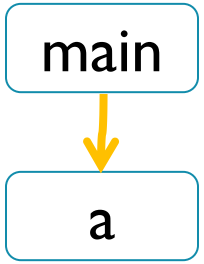
```
int main(void) {
    auto vars
    a();
}

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

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

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

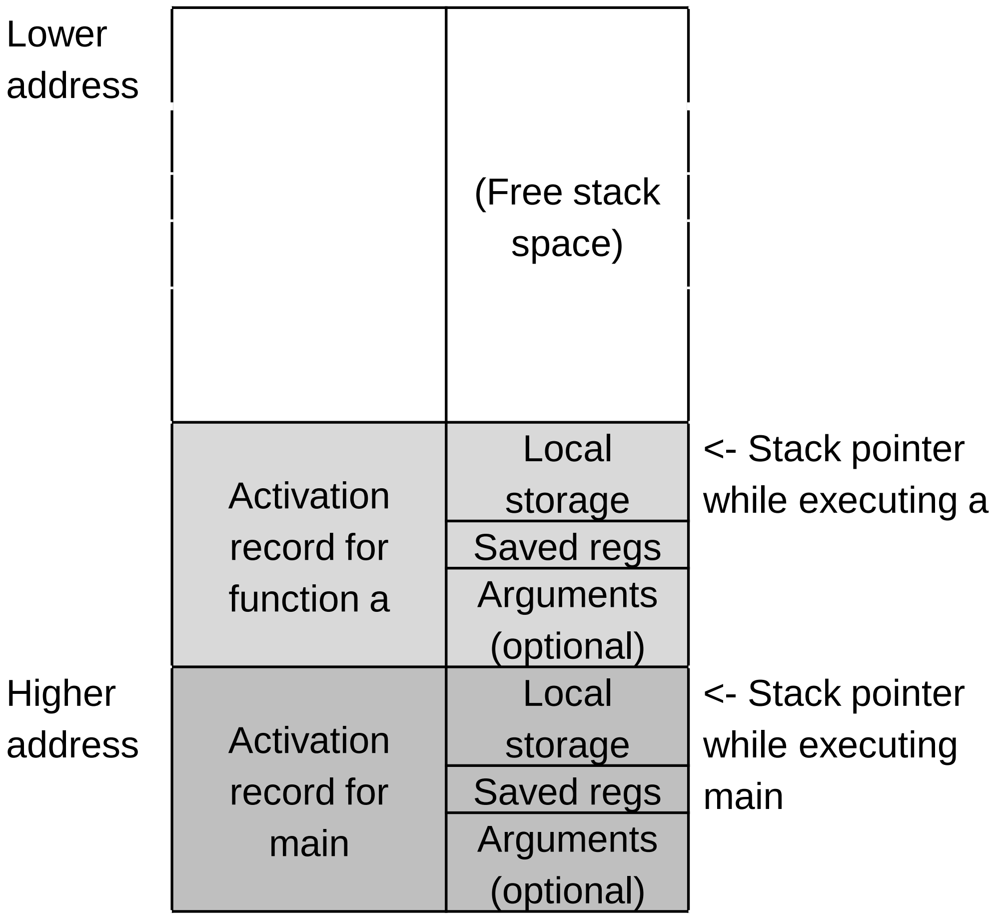
main



Function a before calling function b

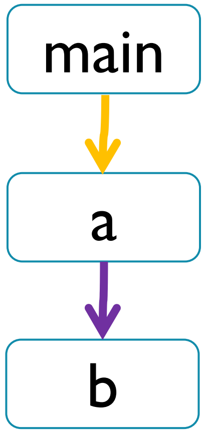
```
int main(void) {
    auto vars
    a();
}

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

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

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

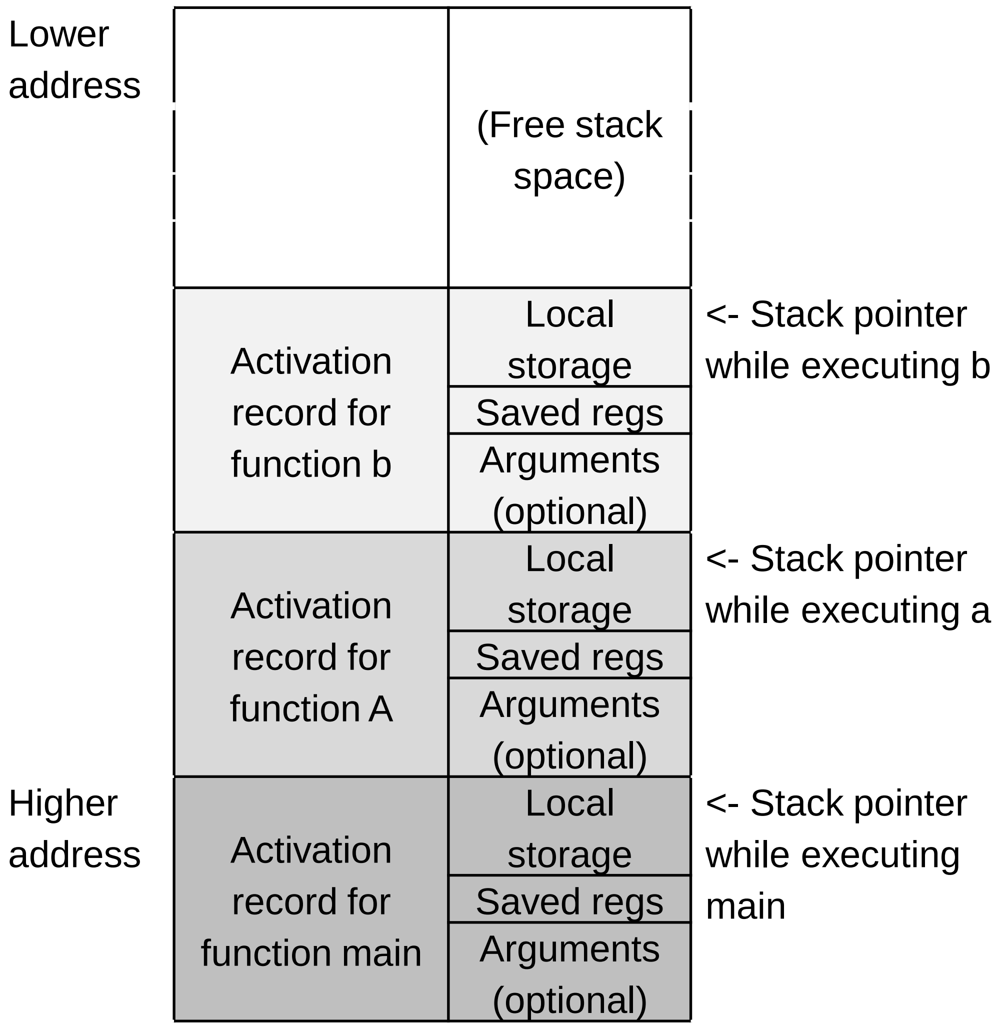




Function b before calling function c

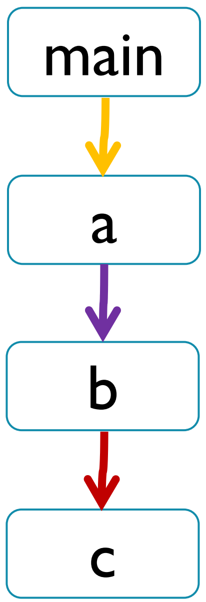
```
int main(void) {
    auto vars
    a();
}

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

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

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





Function c

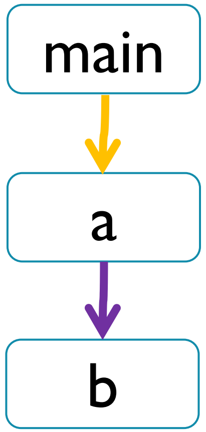
```
int main(void) {
    auto vars
    a();
}

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

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

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

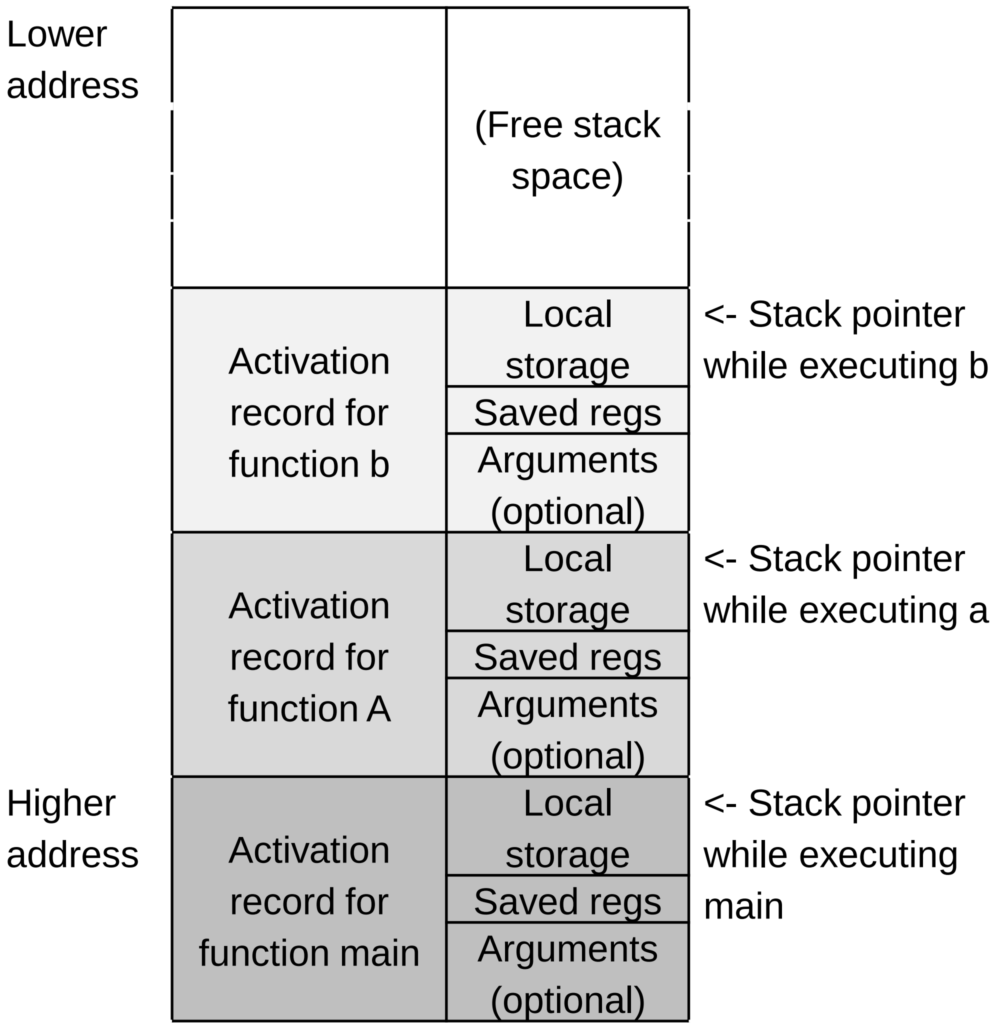


Lower address		(Free stack space)	
	Activation record for current function C	Local storage	<- Stack pointer while executing C
		Saved regs	
		Arguments (optional)	
	Activation record for caller function B	Local storage	<- Stack pointer while executing B
		Saved regs	
		Arguments (optional)	
	Activation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

Function b after calling function c

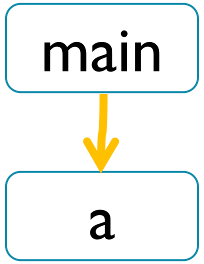
```
int main(void) {
    auto vars
    a();
}

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

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

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

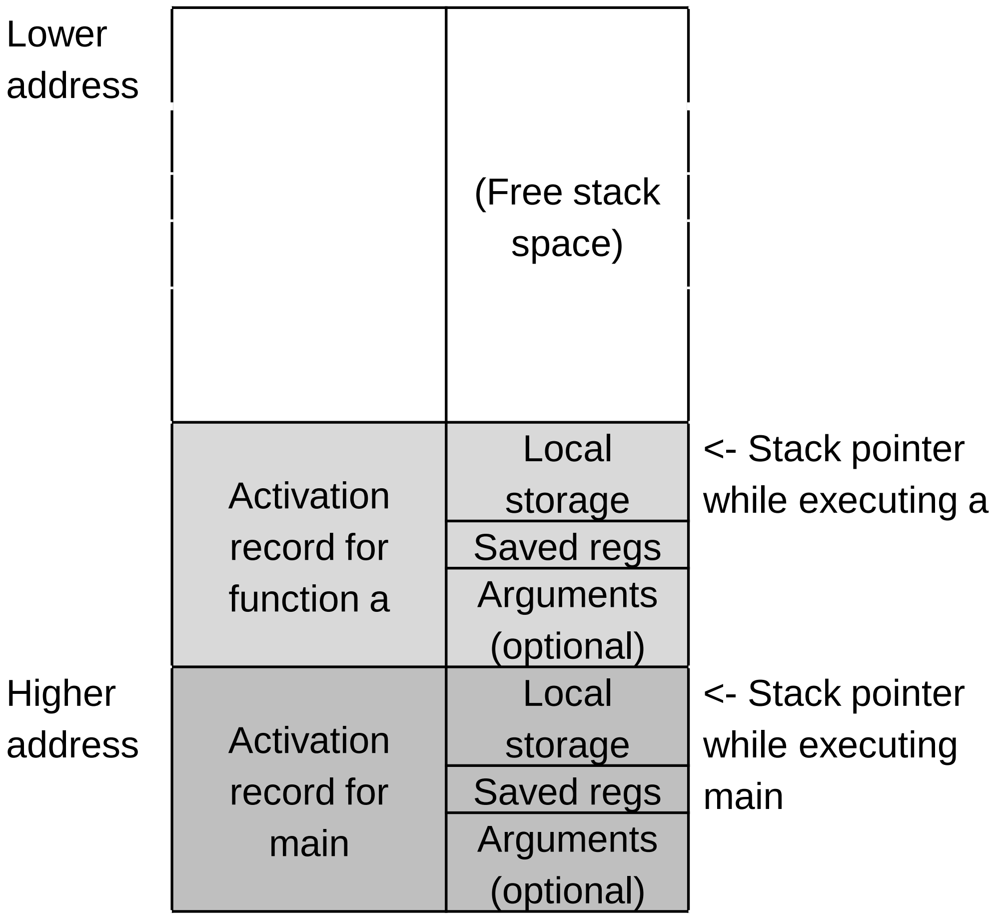




Function a after calling function b

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

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

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

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

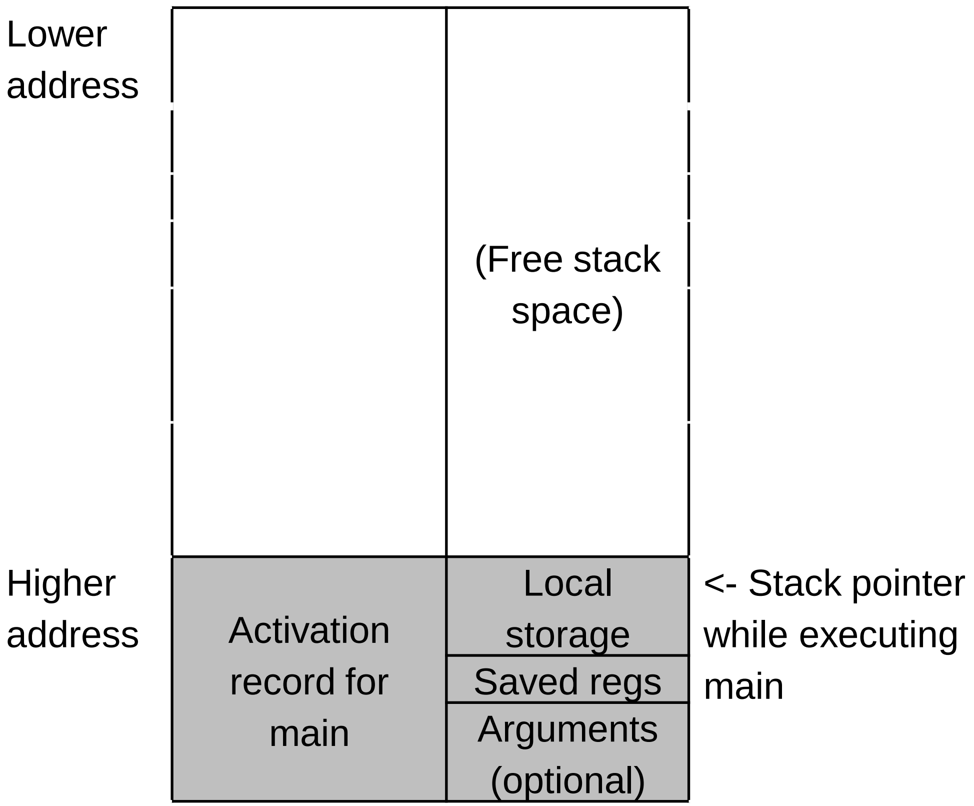




Main after calling function a

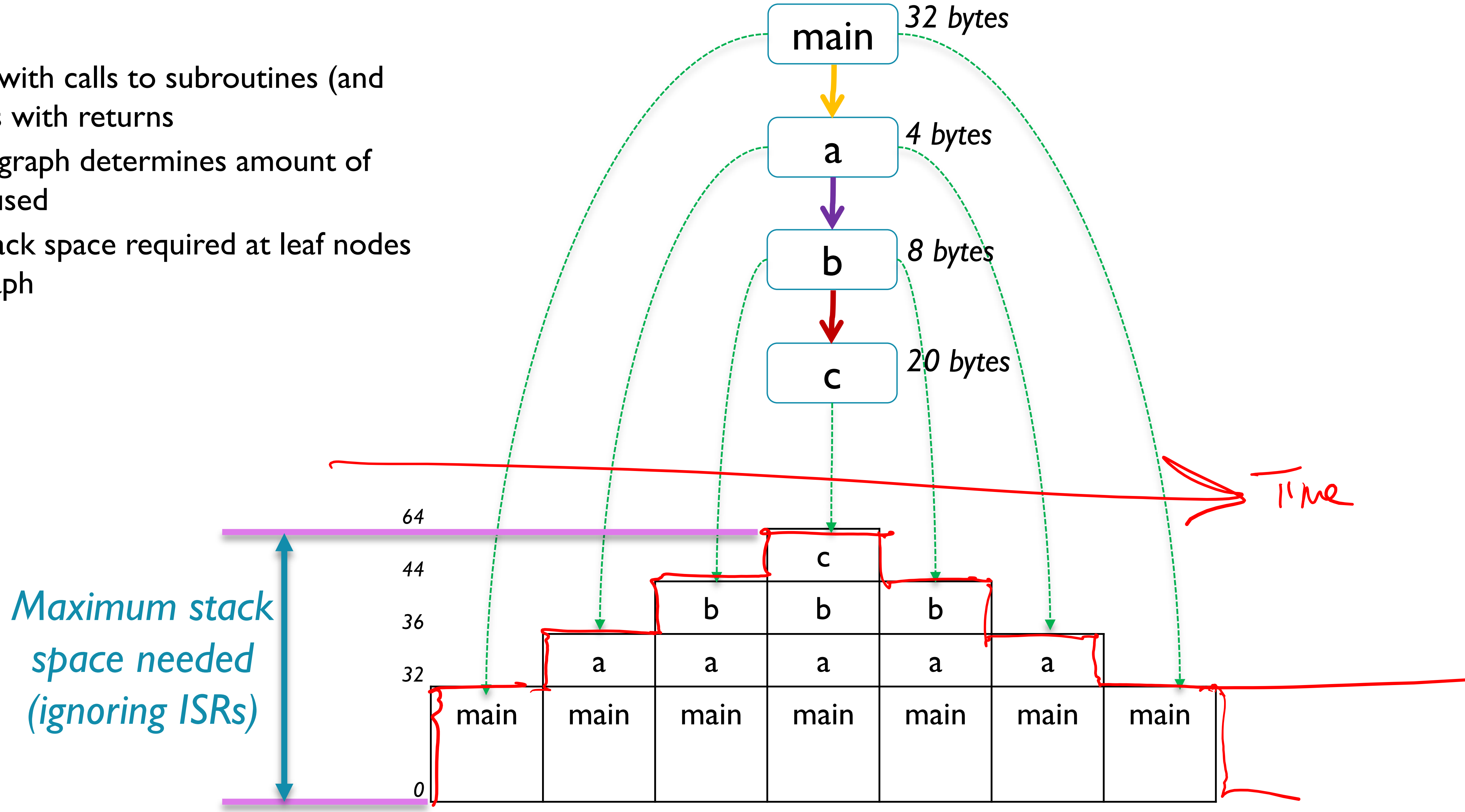
```
int main(void) {
    auto vars
    a();
}

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

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

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

main



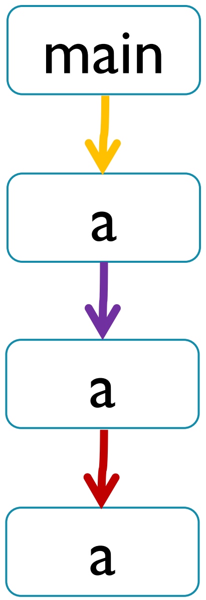
Summary of Stack Memory Use

- Stack grows with calls to subroutines (and ISRs), shrinks with returns
- Depth in callgraph determines amount of stack space used
- Maximum stack space required at leaf nodes (c) of call graph



Example of Recursion

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

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



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

CALLING FUNCTIONS

Function Arguments and Return Values

- First, pass the arguments
  - **How to pass them?**
    - Much faster to use registers than stack memory – for recursion
    - 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 registers
    - Copy remaining arguments onto stack, aligning doubles to even addresses doubleword
    - 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

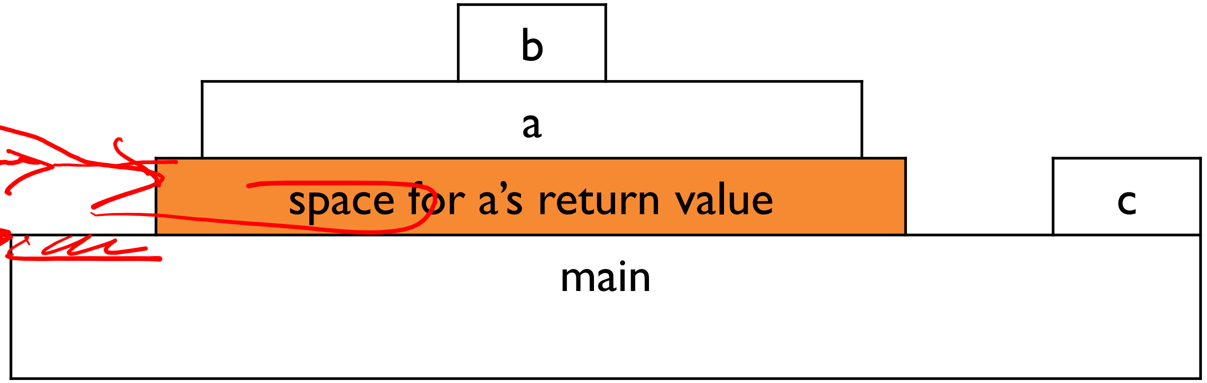
Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this 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: r0, r0-r1, or r0-r3
- Stack
  - Caller function allocates space on stack for return value, then passes pointer to space as an argument to callee
  - Callee stores result at location indicated by pointer
  - Example: a's result is passed on stack

```
int main(…) {
    a(…);
    c(…);
}
big_return_type a(…){
    b(…);
}
small_return_type b(…) {
}
small_return_type c(…) {
}
```

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

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

- Argument 4 into r3
- Argument 3 into r2
- Argument 2 into r1
- Argument 1 into r0
- Call fun3 with BL instruction
- Result was returned in r0, so add to r4 (arg2_2 += result)

```
fun2 PROC
;;;    arg2_2 += fun3(arg2_1, 4, 5, 6);
...
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
```

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}

;;; return arg3_1*arg3_2*arg3_3*arg3_4;

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

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

FUNCTION PROLOG AND EPILOG

Prolog and Epilog

fun2

fun3

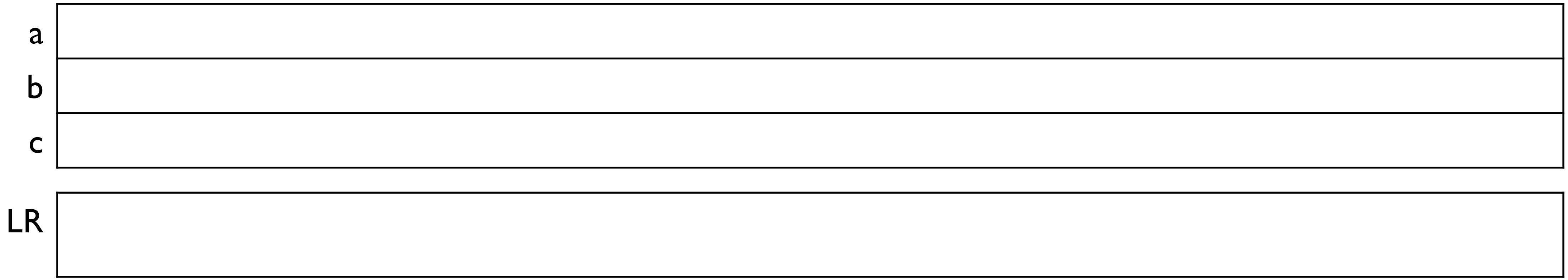
stack

- Each active function has an activation record (AR) on the stack (unless optimized out)
- Code in function manages the AR
  - Prolog code creates AR, epilog destroys it
- Remember the AAPCS
  - **Scratch** registers r0-r3
    - Not expected to be preserved upon returning from a called subroutine
    - Can be overwritten and not restored
  - **Preserved** ("variable") registers r4-r8, r10-r11
    - Expected to have their same values before and after calling a subroutine
    - If changed in subroutine, need to be restored
- Prolog
  - **Save** preserved registers on stack
  - May handle function arguments
  - May allocate temporary storage space on stack (subtract from SP)
- Epilog
  - **Restore** preserved registers from stack
  - May deallocate stack space (add to SP)
  - May handle function return value
  - Return control to calling function by writing to PC

Prolog and Epilog

- Each active function has an activation record (AR) on the stack (unless optimized out)
- Code in function manages the AR
  - Prolog code creates AR, epilog destroys it
- Remember the AAPCS
  - **Scratch** registers r0-r3
    - Not expected to be preserved upon returning from a called subroutine
    - Can be overwritten and not restored
  - **Preserved** (“variable”) registers r4-r8, r10-r11
    - Expected to have their same values before and after calling a subroutine
    - If changed in subroutine, need to be restored
- Prolog
  - **Save** preserved registers on stack
  - May handle function arguments
  - May allocate temporary storage space on stack (subtract from SP)
- Epilog
  - **Restore** preserved registers from stack
  - May deallocate stack space (add to SP)
  - May handle function return value
  - Return control to calling function by writing to PC

Return Address

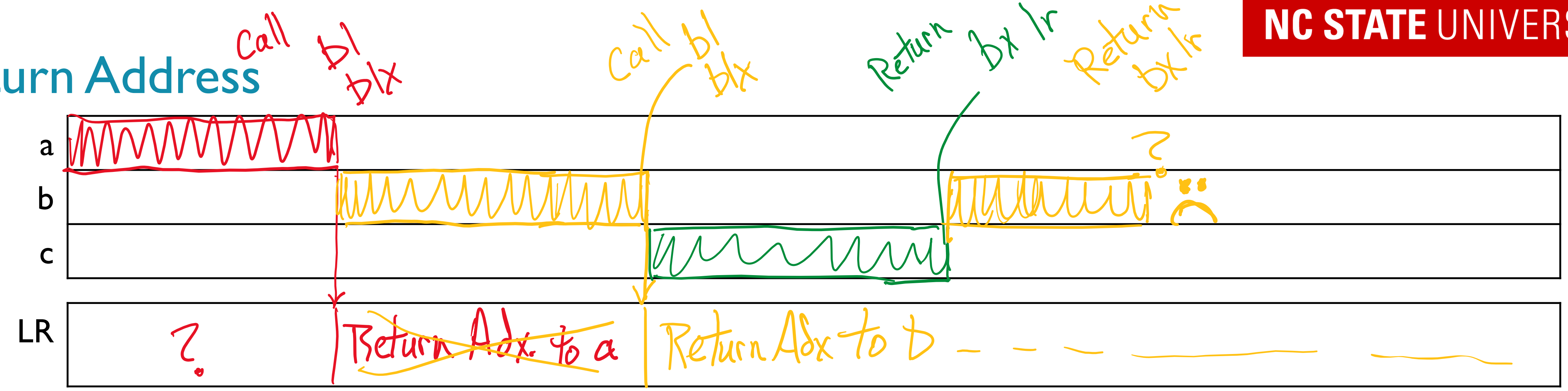


- Return address written into link register (LR) by bl, blx instructions

- 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

- Two code versions: Could this function call a subroutine?
  - Yes
    - Prolog saves, epilog restores LR on stack just like other preserved registers
    - Epilog returns by restoring LR value into PC (not LR) with a pop {…, pc}
  - No
    - Prolog doesn't save LR, as it will not be modified
    - Epilog returns by copying LR value into PC with a bx LR (branch exchange registers)

Return Address



- Return address written into link register (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
- Two code versions: Could this function call a subroutine?
  - Yes
    - Prolog saves, epilog restores LR on stack just like other preserved registers
    - Epilog returns by restoring LR value into PC (not LR) with a pop {…, pc}
  - No
    - Prolog doesn't save LR, as it will not be modified
    - Epilog returns by copying LR value into PC with a bx LR (branch exchange registers)

Example Function Prolog and Epilog

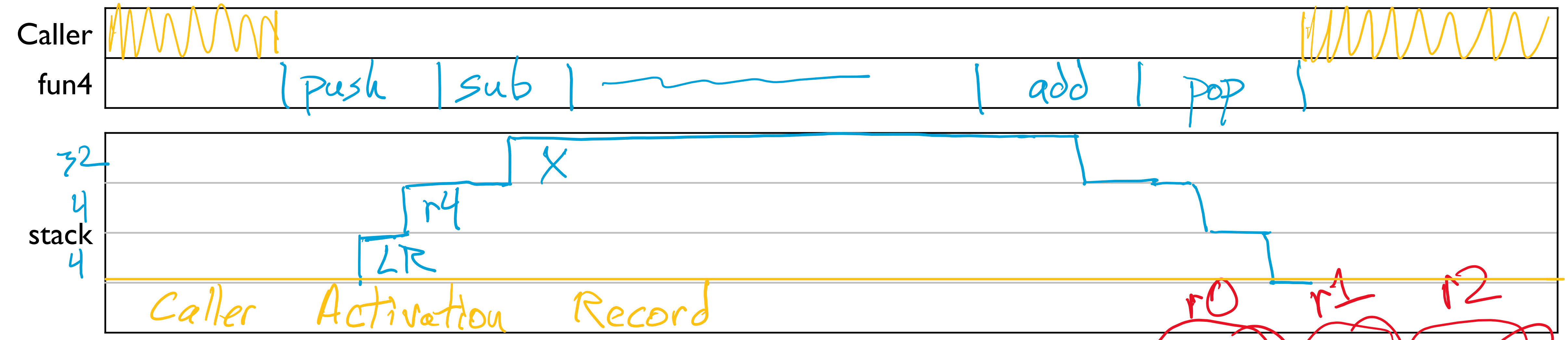
Caller	
fun4	

stack	

- Save r4 (preserved) and link register (return address)
  - Value of r4 (not lr/r14) goes in smaller address, since 4 < 14
- 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)
  - Value from smaller address goes into r4 (not pc/r15), since 4 < 15

```
;;;102  int fun4(char a, int b, char c) {
;;;103    volatile int x[8];
00010a  b510      PUSH   {r4,lr}
00010c  b088      SUB    sp,sp,#0x20
...
;;;106             return a+b+c;
00011c  1858      ADDS   r0,r3,r1
00011e  1880      ADDS   r0,r0,r2
;;;107    }
000120  b008      ADD    sp,sp,#0x20
000122  bd10      POP    {r4,pc}
```

Example Function Prolog and Epilog



- Save r4 (preserved) and link register (return address)
  - Value of r4 (not lr/r14) goes in smaller address, since 4 < 14
- 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)
  - Value from smaller address goes into r4 (not pc/r15), since 4 < 15

```
;;;102   int fun4(char a, int b, char c) {
;;;103     volatile int x[8];
00010a   b510      PUSH   {r4,lr}
00010c   b088      SUB    sp,sp,#0x20
...
;;;106             return a+b+c;
00011c   1858      ADDS   r0,r3,r1
00011e   1880      ADDS   r0,r0,r2
;;;107     }
000120   b008      ADD    sp,sp,#0x20
000122   bd10      POP    {r4,pc}
```

Activation Record Creation by Prolog

Smaller address	space for x[0]	Array x	<- 3. SP after sub 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]		
	r4	Preserved register	<- 2. SP after push {r4,lr}
	lr	Return address	
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]		
	r4	Preserved register	<- 2. SP after add sp,sp,#20
	lr	Return address	
Larger address		Caller’s stack frame	<- 3. SP after pop {r4,pc}

CONTROL FLOW

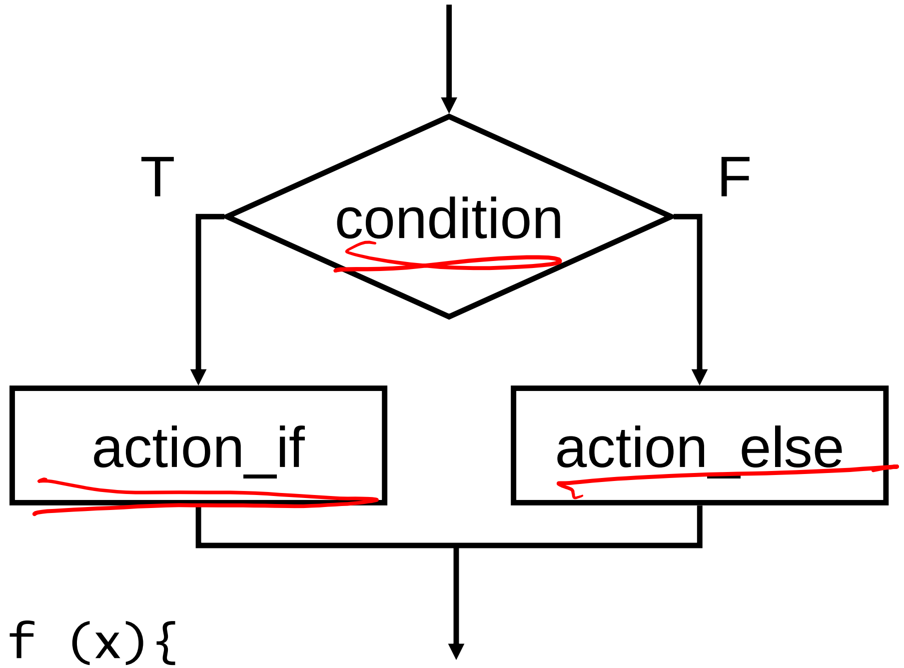
Control Flow: Conditionals and Loops

- 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




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

```
;;;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            }
```

x

y

y

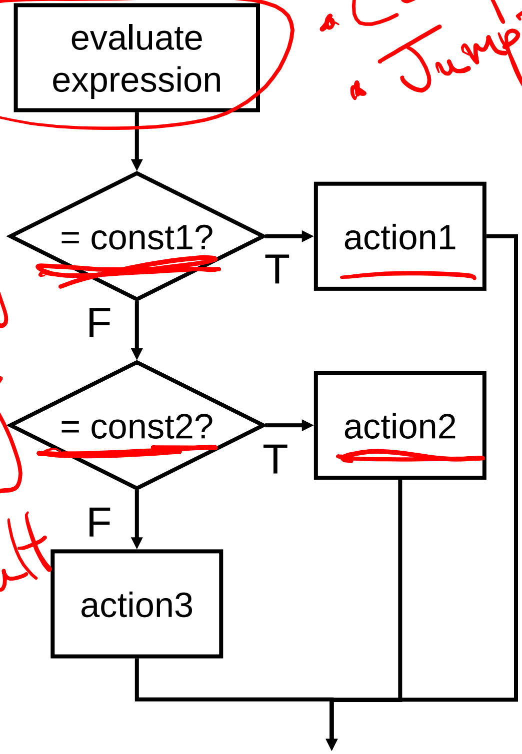
sets N,V,Z,C

Z==1 if result is 0

Control Flow: Switch

• Computed jump
• Jump table

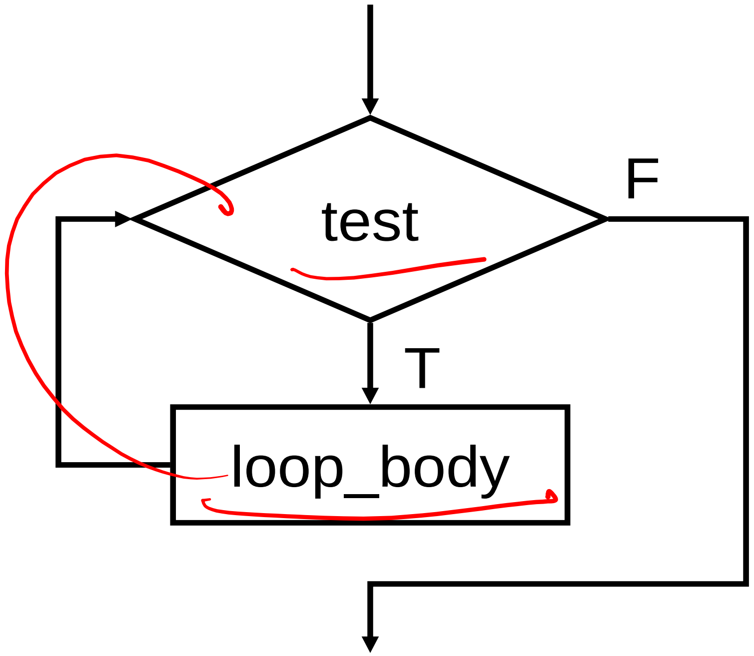
|L1.106|




```
;;;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



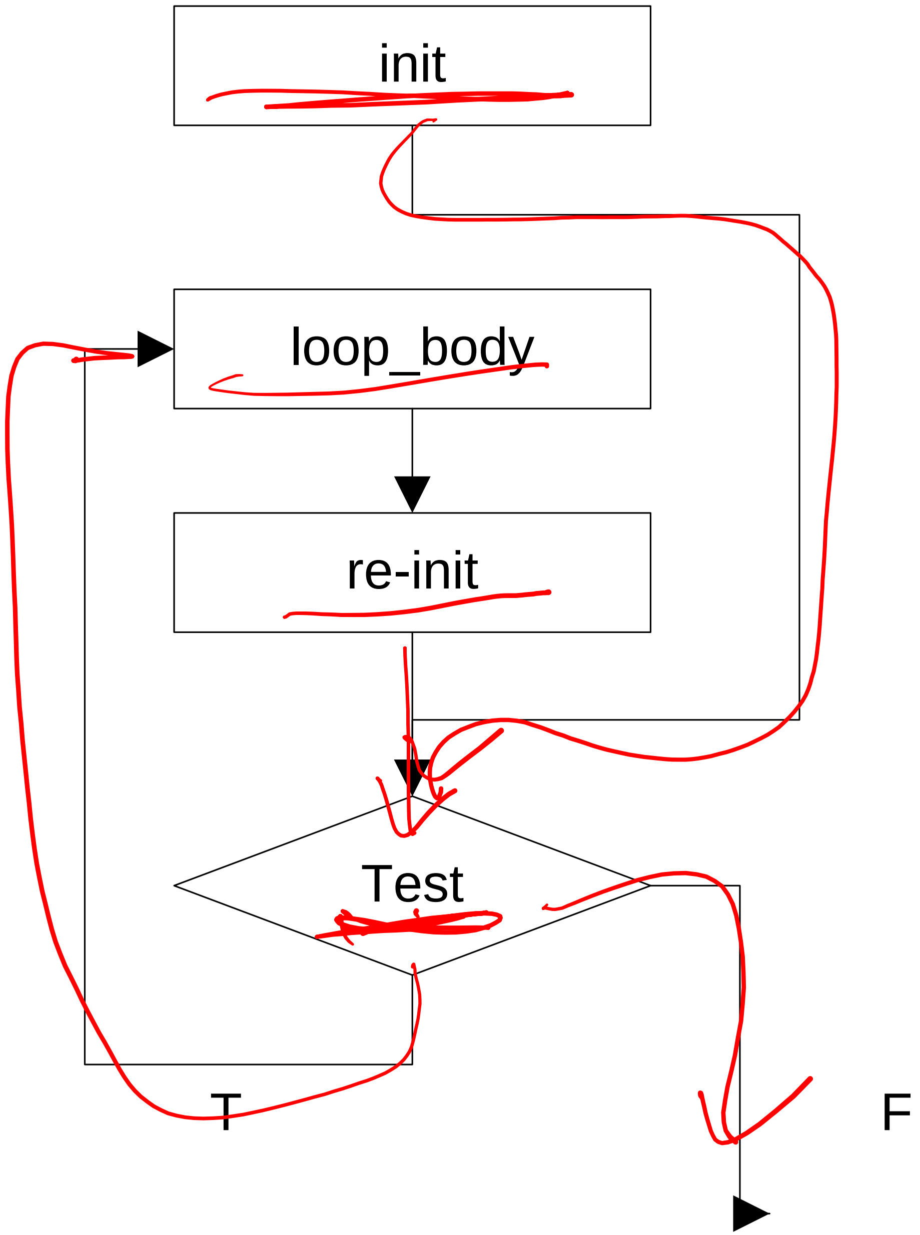

```
;;;57          while (x<10) {
000078   e000  B        |L1.124|
               |L1.122|
;;;58           x = x + 1;
00007a   1c49  ADDS  r1,r1,#1
               |L1.124|
00007c   290a  CMP   r1,#0xa
;57
00007e   d3fc  BCC   |L1.122|
;;;59          }
```

x 10

False

Exit

Iteration: For



init test re-init

```
;;;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            }
```

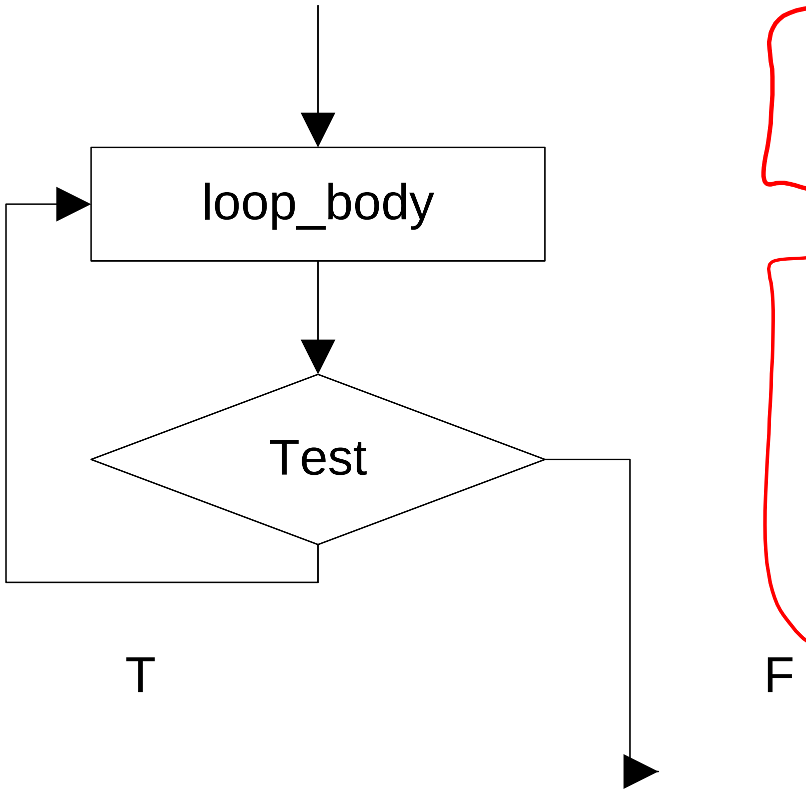
x i

i++

T

F, not taken

Iteration: Do/While




```
;;;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|
```

dec

hex

T

F