Compiling C to Assembly – the Run-time Stack

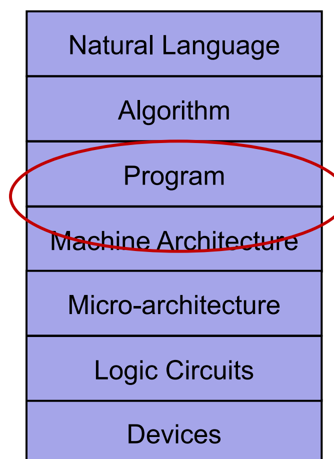
(Chapters 11-13)

1

Problem Transformation - levels of abstraction

How do user programs get executed?

Translation from C to Assembly
- Scope of variables
- Function calls
- Recursion
- Pointers & Arrays
- Data Structures
- Memory Allocation



2

2

Programming Languages

- Assembly language is low-level.
  - It exposes machine instructions and details of the ISA to the programmer.
  - It is ISA-specific.
  - It is useful when the programmer needs fine-grained control of instruction flow and memory usage.
- A high-level language provides a computational abstraction that is machine-independent.
  - Symbolic names (variables) instead of registers and memory locations.
  - High-level operators: multiply, divide, shift, ...

3

3

Why use a High-Level Language? …our choice= C

- Expressiveness: say more with less effort, closer to human-level thinking

```
a = b * c;
```

C statement

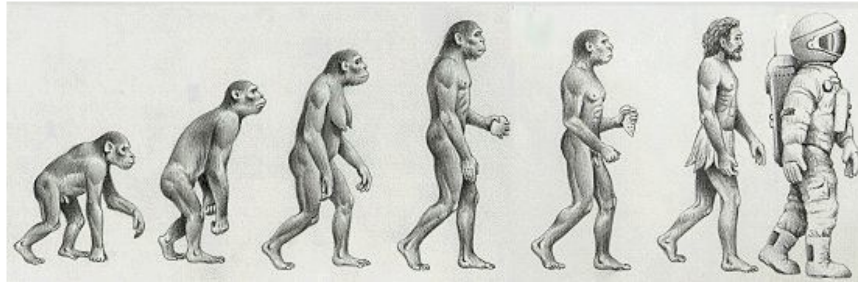
Equivalent LC-3 code

```
      AND R2,R2,#0
      AND R3,R3,#0
      LDR R0,R5,#-1   ; b
      BRz L3
      BRp L1
      NOT R3,R3
      NOT R0,R0
      ADD R0,R0,#1
L1    LDR R1,R5,#-2   ; c
      BRz L5
      BRp L2
      NOT R3,R3
      NOT R1,R1
      ADD R1,R1,#1
L2    ADD R2,R2,R0    ; b * c
      ADD R1,R1,#-1
      BRp L2
      ADD R3,R3,#0
      BRp L3
      NOT R2,R2
      ADD R2,R2,#1
L3    STR R2,R5,#0    ; store to a
```

- Both multiply two values together –
- which is easier to understand?

4

The Evolution of Programming



Machine Language (binary) | Assembly Language | C | Java | Python | Haskell | What's Next??

Simula/C++/etc

5

Quick Note: Compilation vs. Interpretation

•Different ways of translating high-level language

•*Interpretation* *(LISP, Java, Python, Matlab…)*

- interpreter = program that executes program statements (generally one line/command at a time)
  - Called a Virtual Machine
- easy to debug, make changes, view intermediate results

•*Compilation (C, C++, Pascal,…)*

- translates statements into machine language
  - does not execute, but creates executable program
- performs optimization over multiple statements

Example:

X = W+W

Y = X + X

Z= Y + Y

Interpretted language: 3 instructions

Compiler optimized code: 1 instruction Z = 8*W

6

6

Compiling a C Program

•Entire mechanism is usually called the "compiler"

•Preprocessor

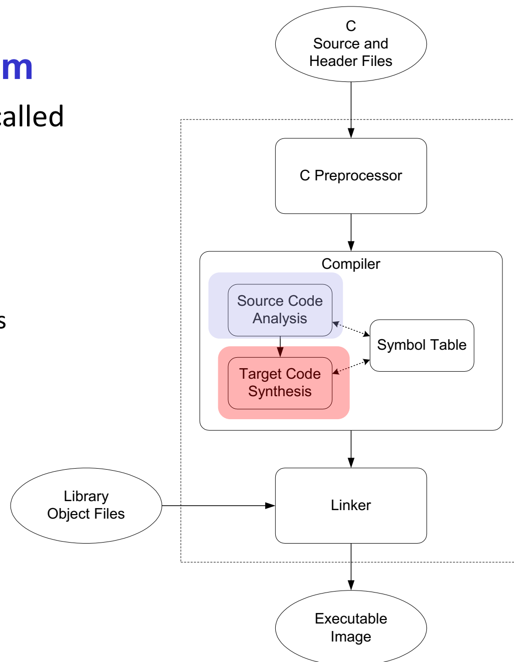
- macro substitution
- conditional compilation
- "source-level" transformations
  - output is still C

•Compiler

- generates object file
  - machine instructions

•Linker

- combine object files (including libraries) into executable image



Linking, Loading,…Makefiles

•Linking: produce final executable program by combining other code (libraries) used by program

- Dynamic Linking: instead of linking at compile time the linking is done at run-time

• Multi-file development….Makefiles assist in this

- Read tutorials on makefiles

• Loader: load executable image generated by linker & execute

- Part of Operating system
- Memory management system does the actual mapping from user space to physical addresses
  - .ORIG x3000 refers to user space x3000

Compiler

- Source Code Analysis
  - "front end": parses programs to identify its pieces – checks syntax
    - variables, expressions, statements, functions, etc.
  - depends on language (not on target machine)
  - Theory behind parsers = Foundations of Comp. course
- Code Generation… we will cover it implicitly in this course
  - "back end": generates machine code from analyzed source
  - may optimize machine code to make it run more efficiently
  - very dependent on target machine
  - *We will play the role of the code generation component as we discuss how different C concepts are implemented in LC3*
    - This is what the compiler backend does
- Symbol Table
  - map between symbolic names and items
  - like assembler, but more kinds of information

9

9

Lessons from Example – Lab9.c

- **Concept of scope**
  - Global and local
    - Code block (enclosed in { }) defines a scope block (Useful for placing debugging statements)
  - Prints:
    - Question1: Global= 0 Local = 1
    - Question2: Global= 4 Local = 2
    - Question 3: Global= 4 Local = 1
    - Question 5: Value of CC=8, x=5, y=4
    - Question 5a: in foo1: x=25, y=32796
    - Question 5b: value of x=5, value of y=32796 (some junk)
- Constants – computed at start and stays constant
  - CC=8 even when x changes to 5
- concept of local variables and passing arguments by value to function foo
  - Argument x passed to function foo1 is not changed in main
  - Local variables x,y in foo1 different from those in main
  - Local variable y is not initialized – so is set to some strange number

10

10

Implementing C: Translation from C to Assembly

- Translate C to assembly
  - Translating operations is the easy part
  - Y = A*B is: MULT R3, R1, R2
- Implement concept of scope
- Concept of storage class
  - Static, Auto, Register….
- Pointers & Arrays
- Function calls
  - Passing arguments to functions
- …HOW ?
- Starting point (Today): How are C variables allocated to memory ?....*Run-time Stack*

11

11

Translation: Allocation of Variables

- What is compiler's task?

1. Allocate memory for variables in a systematic way.
   Will build a **symbol table** to keep track of variable type, size, location.
2. Generate instruction sequences that carry out the computations specified by operators and statements.

- **For this class, we will compile "by hand" to get a sense of how these translations occur.**

12

12

Starting point for translation/compiler: Concept of Scope of Variable

- In assembly, who has access to a memory location/variable ?
- In high level programs, who has access to a variable ?
  - *Concept of Scope of a variable*
- So how do we make this happen in assembly ???

13

13

Defining a variable in C

- Identifier: references to this translate to locations
  - Name of the variable
  - Example: itslocal
- Type: gives us information on data representation and space needed
  - Type of variable such as int, float, char…
  - Example: int itslocal
- Scope
  - Where can it be accessed
  - Example: global variable itsglobal
- Storage Class
  - How does C compiler allocate the storage
    - Does value persist or not
  - Two main classes in C: *Automatic* and *Static*

14

14

Scope: Global and Local

- Where is the variable accessible?
- Global: accessed anywhere in program
- Local: only accessible in a particular region

- Compiler infers scope from where variable is declared
  - programmer doesn't have to explicitly state
    - **Symbol Table constructs this information**
- Variable is local to the block in which it is declared
  - block defined by open and closed braces { }
  - can access variable declared in any "containing" block
    - Global variable is declared outside all blocks

15

15

16

16

Example: Compiling to LC-3

```
#include <stdio.h>
int itsGlobal;   itsGlobal is global variable
int foo(){
       int xfoo=10;          } xfoo, yfoo are local to foo
       int yfoo=1;
       return(xfoo*10);}
main()
{
  int localA;   /* local to main */  } localA, localB are local to main
  int localB;

  /* initialize */
  localA = 5;
  itsGlobal = 3;
  /* perform calculations */
  localB = localA + itsGlobal;
  localA = foo();  <------------------ call function foo

  /* print results */
  printf("The results are: localA = %d, localB = %d\n",
         localA, localB);
}
```

17

17

Example: The Symbol Table

- Like assembler, compiler needs to know information associated with identifiers
  - in assembler, all identifiers were labels and information is address
  - Symbol table kept track of the addresses of the labels

- Compiler keeps more information
  - Name (identifier)
  - Type
  - Location in memory
  - Scope

Name	Type	Offset	Scope
itsGlobal	int	0	global
localA	int	0	main
localB	int	-1	main
xfoo	int	0	foo
yfoo	int	-1	foo

18

18

Scope & Storage Class: Where can you put variables & how lang will their values persist ?

- Local variable inside a function
  - Lasts only while the function is running
- Local variable inside a function
  - Value persists throughout the life of the program
    - Persistance – use static keyword
- Global variable visible to all functions within a file
  - Persists while program is running
- Global variable visible in more than one file
- Block scope – inside the block (defined by { })
  - Persists while code block is running

19

19

Scope can be global within a file

These are different variables with Global scope within their respective files

```
static int i;

int foo(...)
{
  ...
}

int bar(...)
{
   ...
}
```

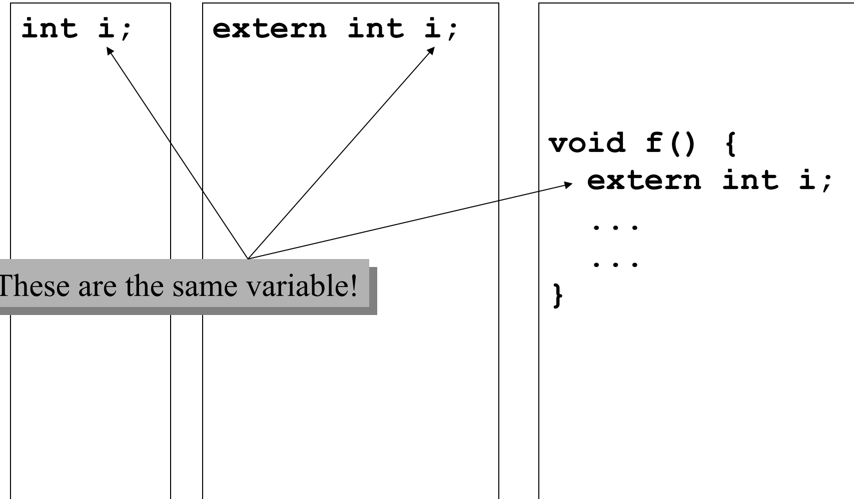
```
static int i;

int main()
{
   ...
}

int baz(...)
{
   ...
}
```

20

Scope can be global across all files



21

Storage Class: Automatic Variables

- Local variable inside a function (lasts only while the function is running)
- `auto` keyword (never used!)
- Located on stack
- Storage Class: AUTO

22

22

Memory

- Many languages have fixed mappings between scopes and lifetimes
- In C, we have the option to decide...

```
void foo(void) {
    int x;
    static int a;
    x = 42;
```

- When the function goes away, x and its value go away since they were *auto storage class* (and placed on stack).
- *Setting a value into a static variable (e.g.* **a***) means that the value is stored in a static area and will persist throughout the entire execution of the program*

25

25

Static Variables (II)

- **`static`**
  - Initialized to zero (like global)
- Located in static area
- Value persists !
- Storage Class: STATIC

26

26

Confused?

```
int x;
static int y;

int main()
{
  static int z;
  int w;
  ...
```

static?

auto?

auto

```
int foo(int z)
{
  int x;
  ...
  if(x == z)
  {

     int y;
     y = ...
```

auto variables (implicitly)

Static Initialization

```
void foo(void)
{
  int x = 10;
  static int y = 20;
  printf("x = %d  y = %d\n", x, y);
  y += 30;
}
```

- What prints the first time foo is called? **x=10 y=20**
- What prints the second time? **x=10 y=50**
  /* value of y from previous call persists */

30

30

Scope vs. Lifetime

Scope	Lifetime
• **File Scope**	• **Life of program**
• **Block Scope**	• **Life of block**

Can be overridden with "**static**"

31

Other goodies

- register
  - Suggests to compiler that this variable should be kept in a register
  - Cannot get address of variable declared register
  - Not as important as it once was with modern compilers
- volatile
  - Type qualifier
  - Tells compiler that value in this variable may change on its own!
  - Used in
    - shared memory applications
    - Memory mapped I/O
- ...
  - Read on your own for now.

32

32

Allocation of Variables in Memory and enforcing Scoping rules

- Simply assigning a memory location for each variable is not enough to enforce scope

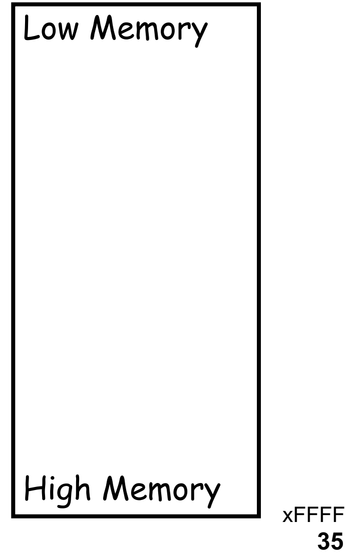
- Need to look at a better scheme to allocate high level program variables to memory in the processor
  - Scope
  - Storage class
  - Allocate space to a variable

34

34

Memory Model (mapping C variables to Memory): Allocating variables in Memory

- How to allocate memory locations to variables
  - *Enforce scope*

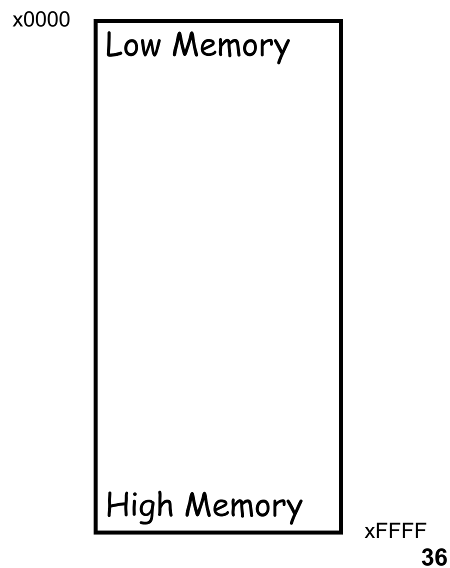


35

35

Memory Model: Compiling & Executing C programs – the Run-time Stack

- Run-time stack – **this** is the KEY!
- Our convention will be that "high-memory" will be on the bottom and "low-memory" on top.
  - drawings are not to scale

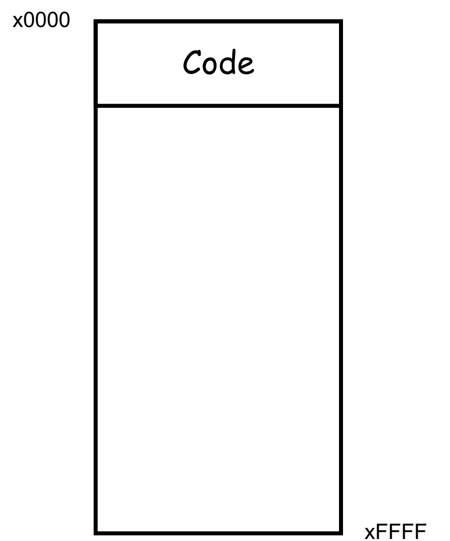


36

36

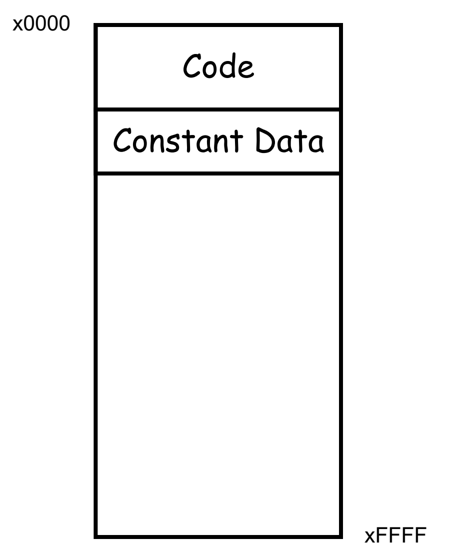
Typical Arrangement

- Normally the actual program code (executable instructions) is placed in low memory
  - Operating System and boot code usually in lowest mem area



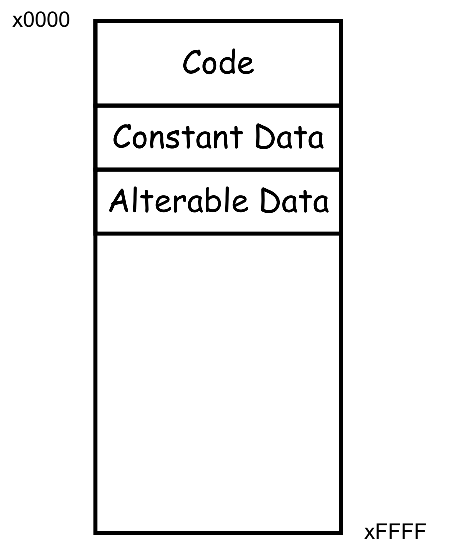
Typical Arrangement

- Next we have an area for storage of constant data



Typical Arrangement

- Data that may be changed follows

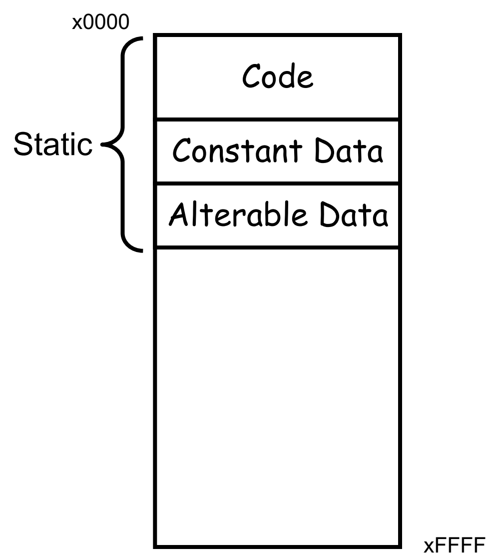


39

39

Typical Arrangement

- These three items comprise what is considered the static area of memory. The static area details (size, what is where, etc.) are *known at translation or compile time*.

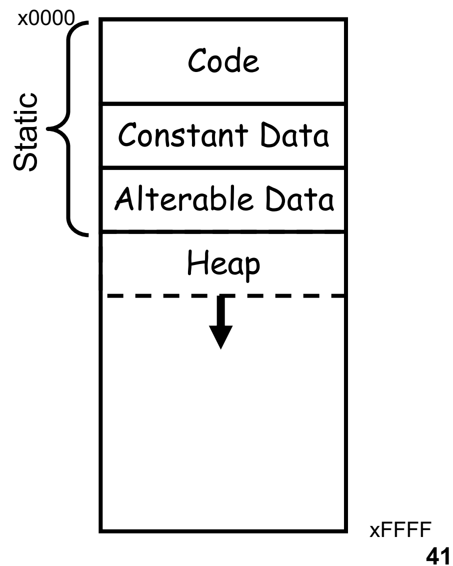


40

40

Typical Arrangement: Heap

- Immediately above the static area the heap is located.
- The heap can expand upward as the program dynamically requests additional storage space
  - `malloc()`
- In most cases, the runtime environment manages the heap for the user

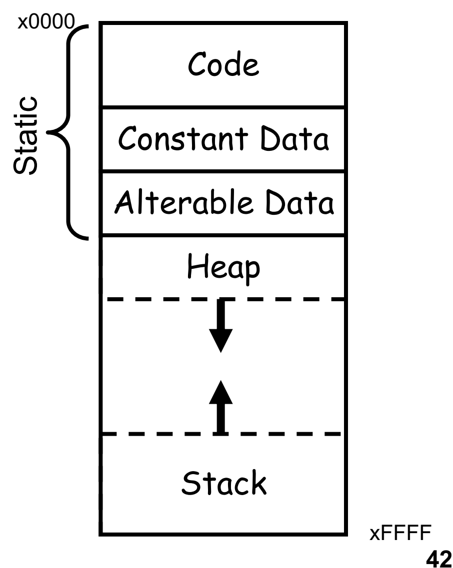


41

41

Typical Arrangement: stack for local variables

- Finally, the activation/run-time stack starts in high memory and can grow to lower addresses as space is needed.
- Items maintained in the stack include
  - Local variables
  - Function parameters
  - Return values

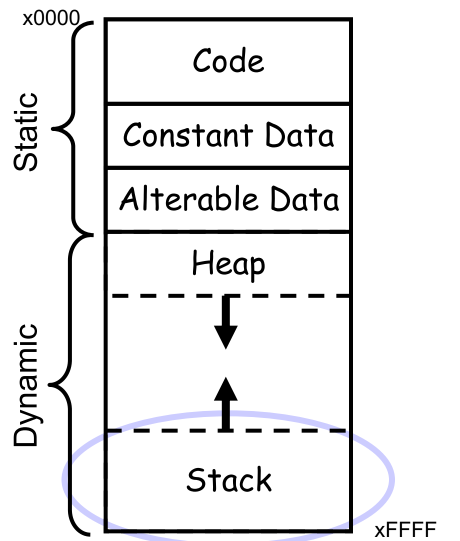


42

42

auto variables

- auto, short for automatic variables are those that exist on the stack. The auto keyword is not normally used.
- Automatic means that space is allocated and deallocated on the stack automatically without the programmer having to do any special operations.



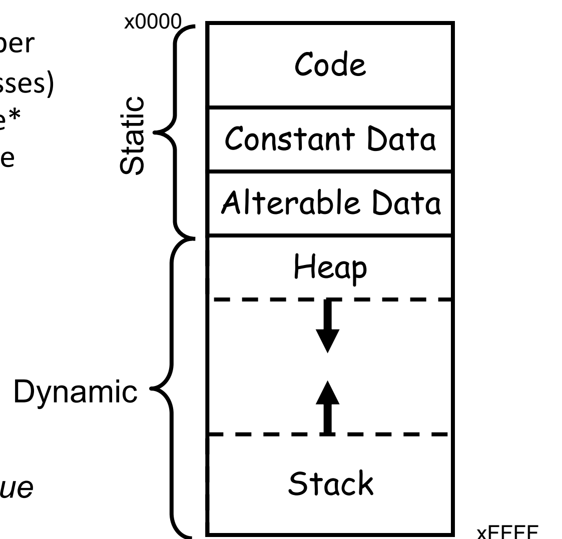
43

43

Summary of Typical Arrangement

- These items in the upper portion (higher addresses) of the memory change* during execution of the program.
- Thus they are called dynamic

**Not just their value*



44

44

Compiler Magic

- The compiler has the job of converting your C program into assembly code
- Thus it must convert the symbolic variable names into addresses
- How does it keep track of what is where?
  - Keep track of scope
  - Storage class
- Symbol table provides much of this information

45

45

Activation Records!!!

- Two main areas (for now) in memory:
  - Global data section
  - Run-time stack
- Local variables exist only during lifetime of function
  - De-allocated after function completes

- How to define area of memory for a code block/function ?
- ......Activation Record
  - Also called Stack Frame
  - Local variables allocated in the activation record
  - Activation record is portion of run-time stack
    - Function can only access a valid portion of the stack
    - Should not access another functions activation records!
  - When function returns…POP the record
    - The local variables can no longer be accessed!

46

46

Symbol Table

- For each variable keeps track of
  - Type
  - Scope
  - Location (as an offset)
    - Either in global area or local area
    - If in local area, then based on activation record
  - Other info (const, etc.)

47

Example: The Symbol Table

- Like assembler, compiler needs to know information associated with identifiers
  - in assembler, all identifiers were labels and information is address
  - Symbol table kept track of the addresses of the labels

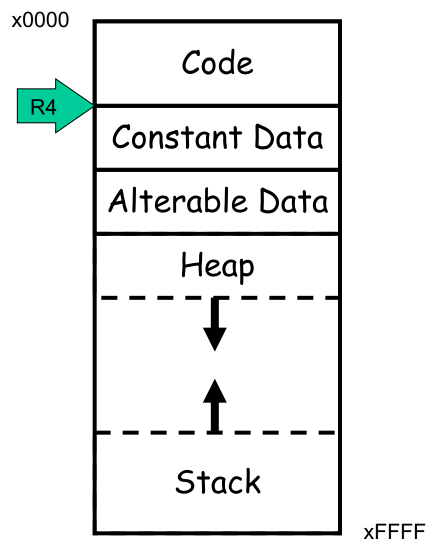
- Compiler keeps more information
  - Name (identifier)
  - Type
  - Location in memory
  - Scope

Name	Type	Offset	Scope
`itsGlobal`	`int`	`0`	`global`
`localA`	`int`	`0`	`main`
`localB`	`int`	`-1`	`main`
`xfoo`	`int`	`0`	`foo`
`yfoo`	`int`	`-1`	`foo`

48

Offset?

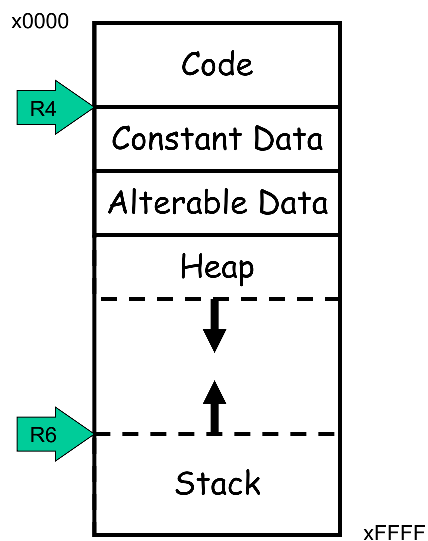
- Assembly code written by a compiler usually looks a little different from assembly code written by hand
- *Registers are dedicated to point to key areas of memory*
- R4 is the Global Pointer
    - Points to start of global static area



49

Keeping Track of `auto` variables

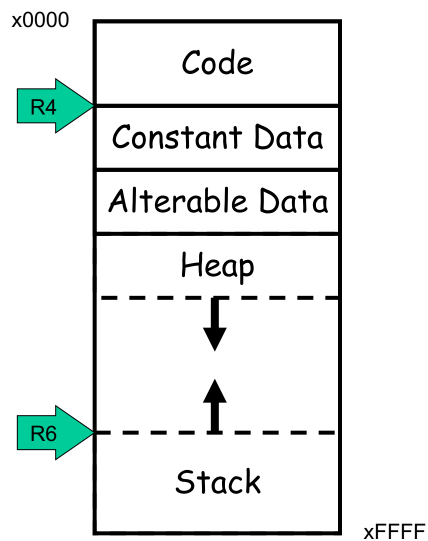
- Stack pointer is obvious but the compiler writer needs more info...
- Where is the activation stack frame?
- R6 is Top of Stack (TOS) pointer



50

Why do we care?

- We would like to know where a variable is throughout the execution of a function
- But, wait you say, I can just reference the variable from the stack pointer
- HA!



51

Can we use TOS

```
int f(int a, int b) {
   int c;
   c = a + b;
   return c;
}

int main() {
   int x;
   int y = 4;
   x = f(7, y);
   printf("%d\n", x);
   return 0;
}
```

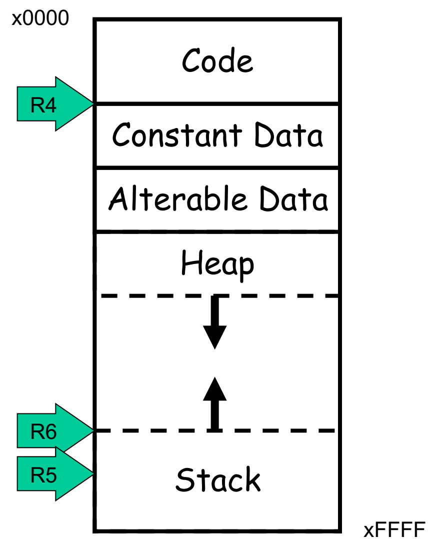
- **What do we need to keep track of?**

52

Frame Pointer

Strictly speaking: we can use stack ptr and forgo FP – some compilers do this!

- The Frame Pointer designates a fixed spot in the activation stack which can be used as a reference throughout execution of the function.
- Note: Frame pointer also called dynamic link
- Store Frame pointer in register….R5
  - Points to 'start' of set of local variables



53

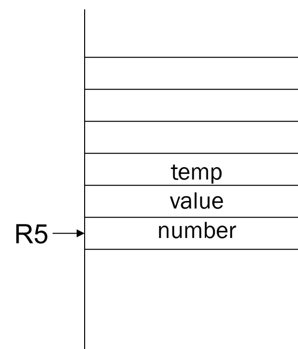
LC3: Local Variable Storage

•Local variables are stored in an *activation record*, for each code block also known as a *stack frame*.

- Cannot afford to forget about the Stack ☺

•*Symbol table "offset" gives the distance from the base of the frame*.

- R5 is the frame pointer – holds address of the base of the current frame.
- A new frame is pushed on the run-time stack each time a block is entered.
- Because stack grows downward, base is the highest address of the frame, and variable offsets are <= 0.

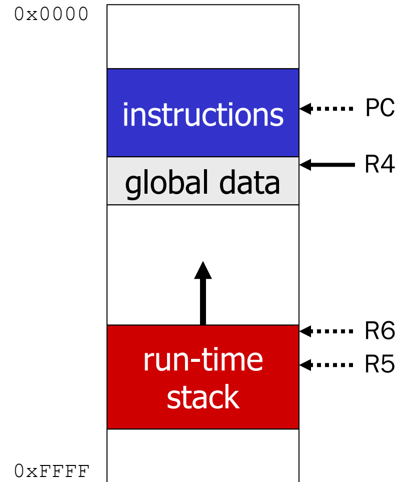


54

54

Summary: LC3 Allocation for Variables

- Global data section
  - All global variables stored here (actually all static variables)
  - R4 points to beginning
- Run-time stack
  - Used for local variables
  - R6 points to top of stack
  - R5 points to top frame on stack
  - New frame for each block (goes away when block exited)
- Offset = distance from beginning of storage area
  - Global: `LDR R1, R4, #4`
  - Local: `LDR R2, R5, #-3`

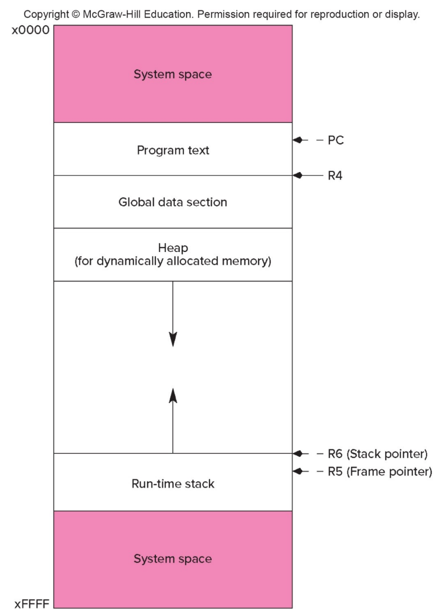


55

LC-3 Memory Map – The Complete Picture

- The LC-3 operating system reserves some of the memory address space for trap vectors, service routine code, and memory-mapped I/O.
- Program instructions will be placed in the "program text" section.
- Global variables are allocated next. R4 is set to the first allocated address for globals.
- Local variables are stored on the run-time stack. R5 points to the local variables of the currently-executing function.

Copyright © McGraw-Hill Education. Permission required for reproduction or display.



Variables and Memory Locations

•In our examples, a variable is always stored in memory.

•When assigning to a variable, must store to memory location.

•A real compiler would perform code optimizations that try to keep variables allocated in registers.

- Why?

57

57

Example: Compiling to LC-3

```
#include <stdio.h>
int itsGlobal;   itsGlobal is global variable
int foo(){
       int xfoo=10;
       int yfoo=1;
       return(xfoo*10);}        xfoo, yfoo are local to foo
main()
{
  int localA;   /* local to main */   localA, localB are local to main
  int localB;

  /* initialize */
  localA = 5;
  itsGlobal = 3;
  /* perform calculations */
  localB= localA + itsGlobal
  localA = foo();           <---------- call function foo
  /* print results */
  printf("The results are: localA = %d, localB = %d\n",
         localA, localB);
}
```

58

58

Example: The Symbol Table

Name	Type	Offset	Scope
itsGlobal	int	0	global
localA	int	0	main
localB	int	-1	main
xfoo	int	0	foo
yfoo	int	-1	foo

59

59

Example: Code Generation

```
; main
; initialize variables
      AND R0, R0, #0
      ADD R0, R0, #5   ; localA = 5
      STR R0, R5, #0   ; (offset = 0)

      AND R0, R0, #0
      ADD R0, R0, #3   ; itsGlobal = 3
      STR R0, R4, #0   ; (offset = 0)
```

address R5+offset=0 is address of localA => accessing local var localA

address R4+offset=0 is address of itsGlobal => accessing R4 implies Global variable

60

60

Example (continued)

```
; statement:
   localB= localA + itsGlobal;
;address of localA = R5 + #0
; address of localB = R5 + ( - #1)
; address of itsGlobal = R4 + #0
     LDR R0, R5, #0   ; load/read localA into R0
     LDR R1, R4, #0   ; load/read itsGlobal to R1
     ADD R2, R1, R0   ; add two values, put in R2
     ; this result has to be written to localB
     STR R2, R5, # -1  ; store value in R2 to
                           localB
```

61

61

Example: C to LC3 Translation

•What is the C code corresponding to these LC3 code segments

Y= A+X;

•First identify accesses/addresses for variables A, X, Y:

- R4, #0   R5, #0   R5, # -1

•Symbol Table:

Identifier	Type	Offset	Scope
A	int	0	Global
B	int	2	Global
X	int	0	main
Y	int	-1	main
Z	int	-2	main

62

62