Computer Hardware

Aamod Varma

CS - 2110, Fall 2024

Contents

Chapter 7

Assembly

7.1 Introduction

Before a program in high-level language can be executed, it must be translated into a program in the ISA of the computer on which it is to be executed.

7.2 Instructions

An instruction has four parts,

```
Label    Opcode    Operands    ; Comment
```

Where Label and Comment are optional.

7.2.1 Opcodes and Operands

Opcodes and Operands are mandatory. Opcode is the name of the instruction and Operands are the things the it is supposed to do it to.

7.2.2 Labels

Symbolic names used to identify memory locations. Starts with an alphabet but can contain a decimal after. Cannot use Opcodes or Operands as Labels. Labels could be either,

1. Location is a branch of new instruction, eg. WHILE, IF, ELSE
2. Location contains as value, eg. LABEL .fill x0001

7.3 Pseudo-Ops

The assembler is a program that takes the "program" and converts it into a program in the ISA. Pseudo-Ops help in this.

7.3.1 .ORIG

Tells the assembler where in memory to place the LC 3 program. So .ORIG x3050 will place the first LC-3 ISA instruction in that specified location.

```
.orig x3000

.end
```

7.3.2 .FILL

Tells assembler to set aside location and initialize it with the value of the operand (could be either number or label).

```
A .fill 5
SIX .fill x0006
```

7.3.3 .BLKW

Tells assembler to set aside some number of sequential memory locations. The number is the operand of the pseudo-op.

```
ARRAY .blkw 4
```

7.3.4 .STRINGZ

Tells assembler to in initialize sequence of $n + 1$ memory locations for words. Where n is the word length.

```
HELLO .STRINZ "Hello!"
```

We have $n + 1$ because the location after ! would be null or $x0000$

7.3.5 .END

Tells assembler that it has reached the end of the program. Doesn't actually exist at the time of execution.

7.4 Assembly Process

7.4.1 Two-Pass Process

First pass is creating the symbol table. Which is a correspondence of symbolic names with their 16-bit memory addresses.

```
Symbol   Address
TEST     x3004
ASCII    x300E
```

The second pass is for generating the machine language program. So using the help of the symbol table it goes through line by line and produces the machine language instruction (in binary)
For LD instruction or instructions that use PCoffset the assembler calculates it using the symbol table.

```
.orig x3000
    LD R0, VALUE
    HALT
VALUE .fill 10
.end
```

Here VALUE is at 3002 and LD is at 3000. So PCoffset is calculated during the second pass as $3003 - (3000 + 1) = 2$. This is because when *LD* is run the PC is incremented. So label must be not more than +256 or -255 memory locations from the LD instruction itself.

7.5 Execution

7.5.1 More than One Object File

It is possible to have an executable image from more than one object file. We can use, .EXTERNAL to tell the assembler that the absence of a label is not an error.

```
    .EXTERNAL STARTofFILE
```

So it allows references in one module to symbolic locations in another module without a problem. The proper translations are resolved by the linker. At link time, the linker uses the symbol table entry for STARTofFILE in another module to complete the translation of our revised line.

Chapter 8

Data Structures

8.1 Subroutines

We can use program fragments more efficiently. These program fragments are called subroutines/procedures or functions. So instead of writing a fragment again every time we want to use it we write it once and call it whenever we want to use it.
JSR(R) does two things,

1. Loads PC with starting address of the subroutine.
2. Loads R7 with PC+1

JMP R7/RET is the last instruction in the subroutine. It is used to jump back to the next instruction after the subroutine call.

8.1.1 JSR(R)

Bits [15:12] contain the opcode, 0100. Bit [11] specifies the addressing mode, 1 means PC-relative and 0 means Base Register addressing. Lastly Bit [10:0] contain information the actual information depending on Bit[11]. For base register addressing the register is Bit[8:6].

Chapter 9

IO

9.1 Privilege, Priority and Memory Address Space

9.1.1 Privilege

We could either have supervisor privilege or user privilege. Idea is that we don't want everyone to have complete control over things like HALT, some memory locations.

9.1.2 Priority

All about urgency. Everyone program is assigned a priority from 0 to 7. Higher the priority it will be executed first.

9.1.3 Processor Status Register

Each program executing has associated with it two important registers the PC and the PSR. The PSR stores the privilege and priority assigned to the program. PSR[15]=1 means unprivileged and PSR[15]=0 means supervisor. Bits[10:8] specify the priority level from 0 to 7. Bits[2:0] store NZP condition codes.

9.1.4 Organization of Memory

We have privileged memory from $x0000$ to $x2FFFF$. They contain data structures and code of the operating system. Require supervisor privilege to access. Referred to as system space.
Locations $x3000$ to $xFDFF$ are unprivileged memory locations. Referred to as user space.
From $xFE00$ to $xFFFF$ do not correspond to memory at all. So the last address is $xFDFF$. But $xFE00$ to $xFFFF$ are used to identify registers that take part in input and output functions. For instance PSR is assigned address $xFFFC$ and the Master Control Register (MCR) is at $xFFFE$.
Both the supervisor stack and the user stack has a stack pointer that indicates the start of the stack. Only one of the two stacks are activate at a time. Register 6 is used as the stack pointer for active stacks.

9.2 Input/Output

I/O devices are mapped to a set of addresses that are allocated to I/O device registers rather than memory locations. LC-3 uses memory mapped I/O. Where xFE00 to xFFFF are reserved for input/output device registers.

9.2.1 Asynchronous vs Synchronous

An asynchronous mechanism would require some protocol or handshaking. So we need a flag to indicate whether someone has or has not typed a character or whether or not a character has been displayed in the monitor. So every time a typist types something the ready bit is set to 1 and if the character is read by the computer it is set back to 0.
Synchronous is when the computer checks for characters at a specified interval of time. Which requires the typist to be synchronized with the computers time period.

9.2.2 Interrupt Driven vs. Polling

Interrupt-driven means the processor is interrupted and told that a key has been struck. But polling means that the processor will check again and again if something has been struck.

9.2.3 Input from Keyboard

KBDR and KBSR

We have two main registers. KBDR is assigned xFE02 while KBSR is assigned xFE00.

1. For KBDR, Bit[7:0] is used for the ASCII value.
2. For KBSR, Bit[15] (ready bit) determines whether a key has to be read and Bit[14] determines whether we want to interrupt.

Service Routine

When a key is struct the ASCII code is loaded into KBDR[7:0] and KBSR[15] is set to 1. After reading KBSR[15] is cleared.
If polling is used, the processor repeatedly checks KBSR[15] until it is set and then can load the ASCII from KBDR.

```
START
    LDI    R1, A
    BRzp   START
    LDI    R0, B
    BRnzp  NEXT_TASK
A   .fill  xFE00
B   .fill  xFE02
```

Loads the ASCII value inputted into R0.

Memory-Mapped Input

Three steps,

1. MAR is loaded with the address of a device register.
2. Memory is read, MDR is loaded with contents at the memory location.
3. DR is loaded with MDR contents.

9.2.4 Output to Monitor

DDR and DSR

We have two main registers. DDR is assigned xFE06 while DSR is assigned xFE04.

1. For DDR, Bit[7:0] is used for the ASCII value to be shown
2. For DSR, Bit[15] (ready bit) determines whether a key has to be read and Bit[14] determines whether we want to interrupt.

Service routine

When a key is to be shown the ASCII code is loaded into DDR[7:0] and DSR[15] is set to 1. After being outputted by the monitor DSR[15] is cleared.
If polling is used, the processor repeatedly checks DSR[15] until it is set and then can output the ASCII.

```
START
    LDI    R1, A
    BRzp   START
    STI    R0, B
    BRnzp  NEXT_TASK
A   .fill  xFE04
B   .fill  xFE06
```

Displays the ASCII value in R0 in the monitor.

Memory-Mapped Output

1. MAR is loaded with the address of a device register.
2. MDR is loaded with contents to be written.
3. Memory is written resulting in the contents of MDR being stored in the address of MAR.

9.3 Trap Routines

9.3.1 Introduction

For I/O we need to know the following,

1. Hardware data registers.
2. Hardware status registers.
3. Asynchronous nature of keyboard input relative to executing program.

9.3.2 Trap Mechanism

1. A set of service routines executed on behalf of user programs. There are 256 service routines.
2. A table of starting addresses stored from x0000 to x00FF.
3. The TRAP instruction. The user uses the TRAP instruction when it wants the OS to execute a specific service routine.
4. A linkage back to the program. The routine must have a mechanism to return control to the user program.

9.3.3 Trap Instruction

The instruction works by,

1. Changing PC to starting address of relevant service routine.
2. Providing a way to get back to the program that executed the TRAP instruction.

Opcode is 1111 and the trap vector is Bits[7:0] which identifies the service routine the program wants the OS to execute.
The EXECUTE phase of the TRAP instruction cycle does three things,

1. PSR and PC are pushed onto stem stack. As PC is incremented the return linkage is saved in the PC. If in user mode R6 is pointing to the user stack. Before PSR and PC is pushed onto system stack, R6 is first stored in SAVED_USP and Saved_SSP is loaded into R6.
2. PSR[15] is set to 0 as the service routine needs supervisor privilege.
3. Trap vector is zero extended to form an address. So x23 is extended to the address x0023. The location x0023 contains the address of the trap handler eg. x04A0.

9.3.4 RTI

RTI means Return from Trap or Interrupt. Opcode is 1000 with no operands. Pops the two top values of the system stack into the PC and PSR. PC will then contain the address following the TRAP instruction. If PSR[15] was in usermode the stack pointers are adjusted back how it was, so Saved_SSP is loaded with R6 and R6 is loaded with Saved_USP.

9.3.5 Summary

Execution of TRAP x23 first causes PSR and incremented PC to be pushed onto system stack. Then contents of x0023 (x04A0) are loaded onto PC.
Next cycle starts with FETCH of the contents of x04A0 which is the first instruction of the service routine. The routine is executed and then ends with RTI which loads PC and PSR with the top two elements in the system stack.
Examples uses,

```
LD    R0, ASCIINewLine
TRAP  x21
LEA   R0, Message
TRAP  x22
LD    R0, ASCIINewLine
TRAP  x21
```

9.3.6 GETC

Invoke TRAP x21 and it displays the ASCII in R0 into the monitor.

9.3.7 PUTS

Invoke TRAP x22 and it displays the string starting from the address in R0 until the end (x0000) into the monitor.

9.4 Interrupts

We see the case where the I/O is controlled by the I/O device rather than processor polling. An I/O device may or may not have anything to do with the program that is currency running can,

1. Force stop the current program
2. Execute a program that carries out needs of the device
3. Resume execution of initial program

9.4.1 Why

Polling wastes a lot of time re executing LDI and BR until the ready bit is set. With interrupt we don't need to do that anymore.

9.4.2 Two parts

1. mechanism that enable the device to interrupt
2. mechanism that handles the interrupt request.

9.4.3 Part 1

The following must be true of the device,

1. Device wants to interrupt
2. Device has the right to interrupt
3. Device has higher priority than current program

Device wants to interrupt: If the ready bit of the KBSR or DSR is set to 1 that means the device can be "used".
Device has the right to interrupt: The interrupt enable bit which can be set or cleared by the processor. Interrupt enable is bit[14] of KBSR or DSR. If 14 is set and as soon as someone types a key (15 is then set) then the device can interrupt.
The interrupt request signal is AND of IE bit and ready bit.
Higher priority: LC-3 has eight priority levels. Higher the number more urgent. Priority of the device must be higher than the current program it wishes to interrupt.

INT Signal

Any device with bits [14] and [15] set assert an interrupt request signal. These are input to a priority encoder that selects the highest priority request from all those asserted. If the PL is higher than the PL of current program then INT signal is asserted.

Test for INT

We don't want to interrupt in the middle of an instruction cycle (FETCH, DECODE, EXECUTE). We'd rather have it check before it beings a new one. So before it executes a next instruction the state depends on the INT signal. So if it is not asserted it continues to FETCH else the next state is the first state of Part II.

9.4.4 Part 2

1. Initiate the interrupt
2. Service the interrupt
3. Return from the interrupt

Initiate the interrupt

1. Save state of interrupted program: PC and PSR are saved. Saved on the supervisor stack the same way how TRAP does it.
2. Load state of Interrupt Service Routine: We load PC and PSR of the interrupt service routine. Similar to TRAP service routines. Interrupt vector table is from x0100 to x01FF.

Service the interrupt

The service routine will execute and the requirement so the device will be serviced.

Return from interrupt

RTI is executed (opcode = 1000) which consist of popping PC and PSR from supervisor stack and restoring them in the rightful places. Condition codes are also restored. If program is user privilege the user stack is restored by setting R6 to contents of Saved_USP.

Chapter 11

Intro to C Programming

11.1 Translating High Level Language Programs

11.1.1 Interpretation

A high level program is a set of "commands" for the interpreter program. The interpreter reads in the commands and carries them out as defined by the language. The interpreter is a virtual machine that executes the program in an isolated sandbox.

11.1.2 Compilation

With compilation the program is translated into machine code that can be directly executed on the hardware. A program can be compiled once and be executed many times. Languages like C, C++ are compiled.

11.1.3 Pros and Cons

With interpretation, developing and debugging are easier. Because it permits the execution one line at a time it is easier to examine intermediate results. However compiled code is quicker.

11.2 C /C++ Language

11.2.1 Compilation Process

First an executable image is created. The compilation involves different components,

1. Preprocessor
2. Compiler
3. Linker

11.2.2 The Preprocessor

It scans through the source files looking for preprocessor directives (similar to pseudo-ops in LC-3). The directives help in transforming the source file. All preprocessor directives start with a # sign.
For instance you can direct the preprocessor to substitute the character string "DAYS" with the string 30 or direct it to insert the contents of stdio.h into the source file at that line.

1. Define,

```
# define add(x,y) x + y

int main() {
    int x = 3 * add(2,1); # x will contain 3 * 2 + 1 = 7 not 3 * 3 = 9
}
```

2. #if/#else/#endif - evaluates a constant expression at compile time; if it's 1, the code between #if and #endif but not in else is kept in the source code; if it's 0, the code between #else and #endif is kept.

11.2.3 The Compiler

After the preprocessor transforms the input source file, the compiler takes over. It compiler creates an object module (machine code for one section of the program). There are two parts,

1. Analysis: The source program is broken down or parsed into its parts.
2. Synthesis: Machine code is generated (also tries to optimize if possible)

The compiler uses the symbol table to translate the program (similar to the LC-3 assembler table)

11.2.4 The Linker

Links together all the object modules to form an executable of the program. Versing of the program that can be loaded into memory and executed by hardware.

11.3 Basic C program

11.3.1 The Function main

Functions are similar to subroutines in LC-3.

- The "main" function serve a special purpose: where the execution of the program begins.
- Every C program requires a main function.
- In ANSI C, main must return an integer value.

11.3.2 The C Preprocessor

Two common directives,

1. # define: Instructs the preprocessor to replace occurrences of any text that matches X with text Y. It helps in creating fixed values within a program.

   For example the following,

   ```
   #define STOP 0
   for (counter = startPoint; counter >= STOP; counter--)
   ```

   is equivalent to,

   ```
   for (counter = startPoint; counter >= 0; counter--)
   ```

2. # include: It instructs the preprocessor to insert another source file into the code at the point include appears.

   For instance,

   ```
   #include <stdio.h>
   ```

   defines some relevant information about the I/O functions in the library. The preprocessor directive is used to insert the header file before the compilation begins.

   There are two variations of how it could be used,

   ```
   #include <stdio.h>
   #include "program.h"
   ```

   (a) The first variation uses angle branches and tells the preprocessor that the header file can be found in a predefined system directory.

   (b) The second is used to include header files we create ourselves, the file should be found in the same directory.

   None of the directives end with a semicolon as they are not C statements but directives.

11.3.3 Input and Output

I/O is accomplished through a set of I/O functions. Similar to the IN and OUT trap routines by the LC-3 system software. It requires a format string that needs two things,

printf

(a) Text to print out

(b) Specifications on how to print program values.

For example,

```
printf("%d is a prime number.", 43);
```

prints out,

```
43 is a prime number.
```

Other variants are,

```
printf("43 plus 59 in decimal is %d.", 43 + 59); # 102
printf("43 plus 59 in hexadecimal is %x.", 43 + 59); # 66
printf("43 plus 59 as a character is %c.", 43 + 59); # 'f'
```

The special sequence backslash n is used for a new line character,

```
printf("Hello\n")
printf("%d %d\n", counter, startPoint - counter);
```

scanf

scanf can be used to input from the user. For instance,

```
// Reads in a character and stores it in nextChar
scanf("%c", &nextChar);
// Reads in a floating point number into radius
scanf("%f", &radius);
// Reads two decimal numbers into length and width
scanf("%d %d", &length, &width);
```

Chapter 12

Variables and Operators

Variables hold the values upon which a program acts and operators are used to manipulate these values.

12.1 Variables

Most basic type of memory object

12.1.1 Basic Data types

int

```
int echo;
```

The representing and range of value of an int depend on the ISA of the underlying hardware. In LC-3 an int would be a 16 bit 2's complement integer and on a x86 system an int is likely to be a 32-bit 2's complement.

char

Declares a character. It stores the ASCII value of the character,

```
char lock;
char key = 'Q';
```

char variables will occupy 16 bits on the LC-3.

float / double

Single-precision floating point number and double declares a double precision floating point number.

```
double twoPointOne = 2.1; // This is 2.1
double twoHundredTen = 2.1E2; // This is 210.0
double twoHundred = 2E2; // This is 200.0
double twoTenths = 2E-1; // This is 0.2
double minusTwoTenths = -2E-1; // This is -0.2
double extTemp = -0.2; // This is -0.2
```

A double may have more bits for the fraction than a float but never fewer. Usually a double is 64 bits long and a float is 32 bits long.

bool

Takes on the values 0 or 1 to represent true or false. The specifier is "_Bool"

```
_Bool flag = 1; // Initialized to 1 or true
bool test = false; // Initialized to false, which is symbolically
```

12.1.2 Initialization of values

Local variables are initialized with garbage values. But global variables are initialized to 0.

```
double width;
double pType = 9.44;
double mass = 6.34E2;
double verySmallAmount = 9.1094E-31;
double veryLargeAmount = 7.334553E102;
int average = 12;
int windChillIndex = -21;
int unknownValue;
int mysteryAmount;
bool flag = false;
char car = 'A'; // single quotes specify a single ASCII character
char number = '4'; // single quotes specify a single ASCII character
```

12.1.3 Bitwise Operator

1. & is bitwise AND
2. | is bitwise OR
3. ^ is bitwise XOR
4. ~ is bitwise NOT
5. « is leftshift, vacated bit positions are filled with zeroes.
6. » is rightshift, vacated bits are sign extended.

12.1.4 Logical Operator

Logical operators only generate integer values 1 or 0.

1. &&: 3 && 4 evaluates to 1 but 3 && 0 will be 0
2. ||: 3 || 4 is 1 and 3 || 0 is 1 but 0 || 0 is 0
3. !: !x is 1 only if x = 0 else its always 0.

12.1.5 Increment-Decrement operators

```
x = 10
y = x++ # y = 10, x = 11
z = ++x # z = 12, x = 12
```

12.2 Tying it Together

12.2.1 Symbol Table

Symbols table entry for a variable contains,

1. Identifier
2. Type
3. Place in memory
4. Scope

12.2.2 Allocating Space for Variables

Two regions of mem in which variables in C are allocated storage.

1. Global data section: global variables
2. Run-time stack: local variables

A stack frame is a region of contiguous memory locations that contains all the local variables for a given function. When a function is executing the highest numbered memory address of its stack frame is stored in R5 - the frame pointer. The variables are allocated in the reverse of the order in which they are declared. So the first declared is the closest to R5 and the last declared would be on the top everything else.
The first local variable is R5, next is R5 - 1 and so on. During execution, R4 points to beginning of global data section, R5 within the run-time stack, and R6 to the top of the run-time stack.

12.3 Additional Topics

12.3.1 Variation of Basic Types

The modifiers long and short can be attached to int with the intent of extending or shortening the default size.
Also use unsigned int where all bits are used to represent nonnegative numbers.

12.3.2 Literals, Constants and Symbolic Values

We can use const quantifier to declare a constant. They are read-only. Three types of constants,

1. Literal constants - unnamed values that appear literally. For instance writing x = 2 * y. In this case 2 is a literal constant.

2. We can define constant values using the const qualifier. Eg. const int x = 10;

3. We can use #define and do #define RADIUS 15.0

Chapter 13

Functions

13.1 Implementation Functions in C

Four phases,

1. Argument values from the caller are passed to callee
2. Control is transferred to the callee
3. The callee executes its task
4. Control is passed back to the caller with return value

13.1.1 Run-Time stack

A functions is just a sequence of instructions that is called during a JSR instruction. The RET instruction returns control back to the caller.
When a functions starts executing the local variables must be allocated somewhere in memory. There are two options,
Option 1: The compiler could systematically assign places in memory for each function to place its local variables. Function A might be assigned memory starting at X while B might be assigned starting at Y. However this **doesn't** work if the function calls itself.
Option 2: The space is allocated once the function starts executing. When the function returns to the caller the space is reclaimed and freed. If function calls itself it gets its own space somewhere else.
In this method, each function has a memory template where it stores its local variables, bookkeeping information and its parameter variables. This template is called it stack frame or activation record. When function is called its stack frame will be allocated somewhere in memory.

1. R5: Frame pointer - points to the base of the local variables for the function.
2. R6: Stack pointer - points to the top of the stack

13.1.2 How it works

Four steps,

1. Caller copies arguments for the callee onto the run-time stack and passes control to the callee.
2. Callee pushes space for local variables and other info onto the run-time stack, creating its stack frame on top of the stack.
3. Callee executes
4. Callee them removes its stack frame off the run-time stack and returns the return value and control to the caller.

The Call

Consider,

```
w = Volt(w, 10);
```

The compiler generate LC-3 code that does the following,

1. Transmits the value of the two arguments to the function Volt by pushing them onto the top of the run-time stack. R6 points to the top of the run-time stack (stack pointer). It contains the address of the memory location that is actively holding the topmost data item on the stack. To push we decrement R6 and then store the data value using R6 as the base address. The argument are pushed onto the stack from **right to left** in the order in which they appear. In our example first 10 is pushed then w.
2. Transfers control to Volt via the JSR instruction.

```
AND R0, R0, #0 ; R0 <- 0
ADD R0, R0, #10 ; R0 <- 10
ADD R6, R6, #-1 ;
STR R0, R6, #0 ; Push 10 onto stack

LDR R0, R5, #0 ; Load w
ADD R6, R6, #-1 ;
STR R0, R6, #0 ; Push w

JSR Volt
```

Starting the Callee Function

The instruction JSR points to is the first instruction in the callee. But first we need to set stage for Volt to execute.
Right now the stack contains the two arguments for Volt at the top. We also need to prepare the stack such that it contains a spot for the **return value of Volt, return address of Watt, Watt's frame pointer, all local variables of Volt**

1. Save space for callee return value. Immediately on top of the parameters for the callee. Written value is written onto this location later.
2. Push a copy of the return address in R7 onto the stack. R7 will hold the return address when JSR is used to initiate a subroutine call.
3. Push caller frame pointer in R5 onto the stack. So we'll be able to restore the stack frame of the caller so that it can easily resume execution after the function call is complete.
4. Allocate space for callee local variables and adjust R5 to point to the base of the local variables and R6 to the top of the stack.

```
Volt:   ADD R6, R6, #-1 ; Allocate spot for the return value

        ADD R6, R6, #-1 ;
        STR R7, R6, #0 ; Push R7 (Return address)

        ADD R6, R6, #-1 ;
        STR R5, R6, #0 ; Push R5 (Caller's frame pointer)

        ADD R5, R6, #-1 ; Set frame pointer for Volt
        ADD R6, R6, #-2 ; Allocate memory for Volt's local variable
```

Ending the Callee function

Once the callee has completed its work, we need to return to the caller. So tearing down the stack frame and restoring the run-time stack.

1. If there is a return value, write it into the return value entry of the active stack frame.
2. Local variables for callee are popped off.
3. Caller frame pointer and return address are restored
4. Control to the caller function via the RET instruction.

```
LDR R0, R5, #0 ; Load local variable k
STR R0, R5, #3 ; Write it in return value slot, which will always ; be at
location R5 + 3

ADD R6, R5, #1 ; Pop local variables

LDR R5, R6, #0 ; Pop the frame pointer
ADD R6, R6, #1 ;

LDR R7, R6, #0 ; Pop the return address
ADD R6, R6, #1 ;
RET
```

Returning to the Caller Function

After executing RET control is passed back to the caller.

1. Return value is popped off the stack.
2. Arguments are popped off the stack.

```
JSR Volt
LDR R0, R6, #0 ; Load the return value
STR R0, R5, #0 ; w = Volt(w, 10);
ADD R6, R6, #1 ; Pop return value

ADD R6, R6, #2 ; Pop arguments
```

Caller Save / Callee Save

- R0 to R3 can contain temporary values part of an ongoing computation.
- R4 to R7 are reserved.
    1. R4: pointer to the global data section
    2. R5: frame pointer
    3. R6: stack pointer
    4. R7: return address

13.1.3 Summary

1. Caller pushes each argument onto stack and performs JSR to callee.
2. Callee allocates space for return value and saves frame pointer and return address.
3. Space is allocated for local variables.
4. Callee carries out its task.
5. Callee writes return value into space reserved.
6. Frame pointer and return value for caller is restored and returns to caller.
7. Caller pops the return value and arguments from stack and resumes.

Chapter 16

Pointers and Arrays

A pointer is the address of a memory object. We can indirectly access variables/objects. With pointers we can create functions that modify the arguments passed by the caller.
An array is a list of data objects of the same type arranged sequentially in memory.

16.1 Pointers

Arguments are always passed from caller to callee by value.

16.1.1 Declaring Pointer Variables

A pointer variables contains a bit pattern traded as an address of a memory object. A pointer is said to point to the variable whose address it contains. Eg.

```
int *ptr;
char *cp;
double *dp;
```

The variable named ptr points to an integer

16.1.2 Pointer Operators

Address operator &

& generates the memory address of it operand.

```
int object;
int *ptr;

object = 4;
ptr = &object;
```

Here ptr will point to the integer variable object.

Indirection operator *

Used to dereference. We can indirectly manipulate the value of a memory object. If we have Then *ptr refers to the value pointed to by the variable ptr.

```
int object;
int *ptr;

object = 4;
ptr = &object;
*ptr = *ptr + 1;
```

The last statement is the same as writing object = object + 1;

16.1.3 Passing reference using pointers

```
void Swap(int *firstVal, int *secondVal){
    int tempVal;

    tempVal = *firstVal;
    *firstVal = *secondVal;
    *secondVal = tempVal;
}
```

Now because we're passing a pointer, we're technically doing call by reference (the stack stores the address not the value). So now if we use * we can dereference and modify the value within that address.

16.1.4 Null Pointers

We can also have, pointer point to nothing,

```
int *ptr = NULL;
```

16.2 Arrays

16.2.1 Declaring and Using Arrays

```
int grid[10];
```

This declares an array of 10 integers. So memory locations are allocated for grid[0] to grid[9].

16.2.2 Arrays as parameters

When an array is passed as a parameter we just pass the name,

```
int numbers[10];
mean = Average(numbers); # only name is passed
```

In this case the type of "numbers" is similar to int * in that the name numbers is equivalent to &numbers[0]. So numbers is a pointer to something of integer type. So the address of the array numbers is pushed onto the stack.

16.2.3 Strings in C

Strings are sequences of characters. They are arrays of char.

```
char word[10];
```

declares an array that can contain a string up to 10 characters. However because we need the null terminator we will need to allocate 11 locations.
We can print using %s.

```
char word[10] = "Hello";
printf("%s", word);
```

To input a string we can use scanf,

```
char word[10];
scanf("%s", word); # no need to use & beacuse word is an address
printf("This is entered: %s", word);
```

16.2.4 Relation between Arrays and Pointers in C

```
char word[10];
char *cptr;
cptr = word
```

We can access the 4th character using,

```
word[3] or *(cptr + 3) or *(word + 3)
```

16.2.5 Common Pitfalls

C does not have protection against exceeding the size of an array. So an expression like a[i] can access a memory location beyond the end of the array.

Chapter 19

Dynamic Data Structures

Structures are used to represent objects that are represented by multiple basic data types.

19.1 Structures

```
struct flightType{
    char ID[7];
    int altitude;
}
```

We can declare a variables using,

```
struct flightType plane;
```

To access individual elements we can do,

```
plane.ID = "test";
plane.altitude = 100;
```

19.1.1 typedef

We can use typedef to define our own aggregate types,

```
typedef struct flightType Flight;
```

19.1.2 Array of structure

```
Flight aircraft[100];
```

We have an array of Flight which can store 100 Flights.
We can access using,

```
aircraft[0].heading
```

We can also create pointers to structures,

```
Flight *airPtr;
airPtr = &aircraft[30];
```

to access any field we can do,

```
(*airPtr).longitude;
airPtr->longitude
```

19.2 Dynamic Memory Allocation

Fixed data has problems, (1) We can't go over. (2). Adding, removing etc can take a toll.
Memory objects in C are allocated to,

1. run-time stack: variables declared local to functions.
2. global data section: global variables.
3. heap: dynamically allocated data objects.

At high level, a memory allocated (malloc) manages an area of memory called the heap. A program can make a request to malloc for blocks of memory of a particular size. malloc then locates a block of the size and reserves it by marking it as allocated and returns a pointer to the block. Once a block is allocated it will stay allocated unless we deallocate it by calling "free" in C unlike the stack where after the function call it is technically deallocated.

Dynamically Sized Arrays

Dynamic allocation and reallocation are handled by malloc and free.

```
int numAircraft;
Flight *planes;

planes = (Flight *) malloc(24 * numAircraft);
```

malloc allocates a contiguous region of memory on the heap of the size in bytes. If the heap has enough unclaimed memory, malloc returns a pointer to the allocated region. In this case we have 24 because one Flight takes up 24 bytes of memory (depends on the int, char, double etc in the struct).
malloc returns NULL if the current allocation cannot be accomplished. Because malloc needs to be compatible with any data type we need to type cast it depending on what we're using it for.

```
int num;
Flight *planes;

planes = (Flight *) malloc(sizeof(Flight) * num);
if (planes == NULL) {
    printf("Error in malloc");
}
```