

## Embedded Systems -2

#### ES 2: Focus

- Memory map of a C program recap
  - Software Development Tools
  - Program to an Executable
- Architectural View of Computing System recap
- CPU-DRAM Performance Gap
- Principle of Locality
- Memory/Storage Hierarchies
  - Three Properties of Memory Levels
  - Properties and Issues with Memory Levels

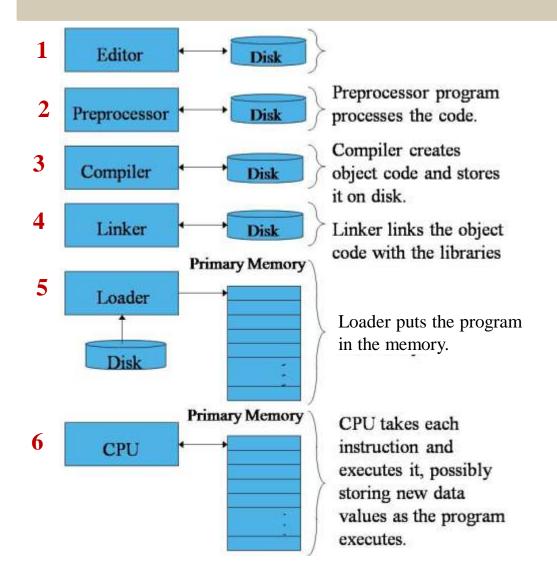


### **SW Development Tools**

#### **Phases of SW Program Development**

- 1. Edit Source files (.c and .h)
- 2. **Preprocess** Part of compilation (include files)
- 3. Compile Get the .obj file of each source file
- **4.** Link Link libraries and your own object files
- **5.** Load Load the executable (along with static data)
- **6.** Execute Run the program on the CPU

#### **SW Program Development Environment**



- 1. Edit
- 2. Preprocess
- 3. Compile
- 4. Link
- 5. Load
- 6. Execute



### Program to an Executable

#### From a C program to an Executable



File: test.c

Snippet of a C program test.c

test.c is compiled using a C compiler

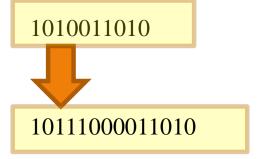
ADR R1, count; load the address of count in R1
LDR R0, [R1]; copy the content of count into R0
ADD R0, R0, #1; increment R0 by 1
STR R0, [R1]; copy the incremented value to count

**File**: test.asm (a text file with assembly instructions)

This is a readable format of .o file

File: test.o (a binary file)

Output from the *compiler* 



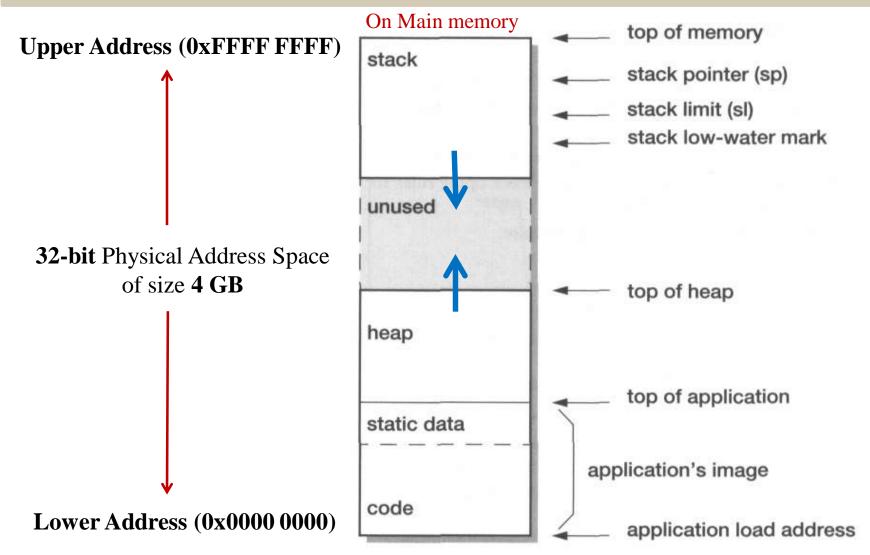
test.o is linked with lib files using a *linker* 

File: test.exe

Output from the *linker* 

**Load** the executable into memory using a *loader* before the execution.

#### Standard 32 bit Processor Address Space Model: An Example



Let us now view the memory layout of a simple C program next ...

# Typical Memory Layout of C Program (Data)

```
Main Memory (MM)
 #include <stdio.h>
                                                                          (DRAM)
 #include <stdlib.h>
                                         Higher address
                                                                                                    Cmd line
 /* These global variables are not used here */
                                                                No cmd line args here
 const float pi = 3.14;
                                                                                                       args
 static int s1 = 10;
                                                                     local, ip allotted on main()
                                                                     i when inc() is called
 static int s2;
                                                                          Stack
 int g = 2;
                                                                                                     Changes
                                                                                                        At
                     Note: OS prepares this program
 int inc(int i) {
                                                                                                     run-time
                     layout on MM by reading the executable
     return i++:
                     from the hard disk, before giving
                     the control to the program.
                                                                           Heap
                                                                  Allotted space for 10 integers,
 main(void) {
                                                                     after malloc() is called
                                                                                                      Loaded
     int local = 10; int *ip = NULL;
                                                                      Uninitialized
                                                              s2, s3
     static int s3;
                                                                                                      before
                                                                       Data (BSS)
     ip = (int *) malloc(sizeof(int) * 10);
                                                                                                    execution
                                                                  This is filled with all zeros
                                                              pi, s1, g
     local = inc(local);
                                                                     Initialized Data
                                                                                                   Read From
     free(ip); /* ip not used by the program */
                                                                        Code/text
                                                                                                     .exe file
     return 0:
                                                                    Assembly opcode
 } /* end of main() */
                                                                     of the program
                                           Lower address
BSS: Block Started by Symbol
```



### **Architectural View of Computing System**

#### **Architectural View of a Computer System** (from the architecture point of view) **DRAM** State transitions of a CPU Main memory Computing System I/O System bus PC MAR Instruction Instruction movement apparatus Instruction MBR IR Stack I/O AR Data Execution Data I/O BR unit Control Data mechanism Data Data **Processing** I/O Module n-2Data storage processing facility **Needed for Data Processing** Data Data **Storage Processing** PC Program counter IR Instruction register MAR Memory address register Buffers MBR Memory buffer register I/O AR Input/output address register **Data Storage** I/O BR Input/output buffer register

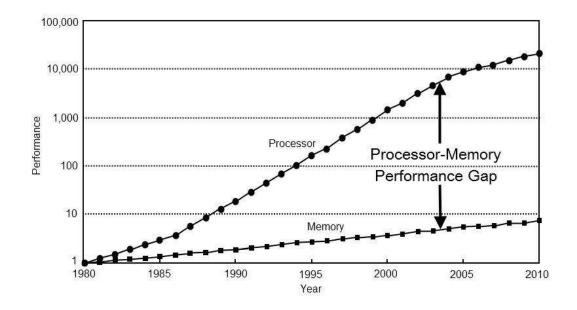
Control module coordinates the interactions between all the modules. The CPU needs to read each instruction and read/write data from/to the memory, is there an issue with it?



### **CPU-DRAM Performance Gap**

#### **CPU-DRAM Performance Gap**

- The **performance** gap between **CPU** and **DRAM** is shown below:
  - CPU performance increases by 60% every year
  - **DRAM** performance **increases** by less than **10% every year**



Next, let us see one important property of software that helps in reducing the impact on software efficiency due to this performance gap between CPU and memory ...



**Principle of Locality** 

#### **Principle of Locality**

- Programs tend to reuse **data** and **instructions** that are **closer** to **those** that have been **used recently**, or that were **recently referenced** themselves.
- **Temporal locality: Recently referenced** items are **likely** to be **referenced** in the near future.
- **Spatial locality: Items** with **nearby addresses** tend to be **referenced** more often with higher probability.

Sample code ...



Identify: Temporal or Spatial locality?

int sum = 0;for (i = 0; i < n; i++)

sum += a[i]; return sum;

For Data

Access of array elements in succession

Spatial locality - Access of **sum** on each iteration Temporal locality

For Instructions

Access of instructions in a sequence

 Cycling through the loop repeatedly Temporal locality

How can this property be leveraged to fill the gap between the speeds of CPU and memory? ... let us see next ...

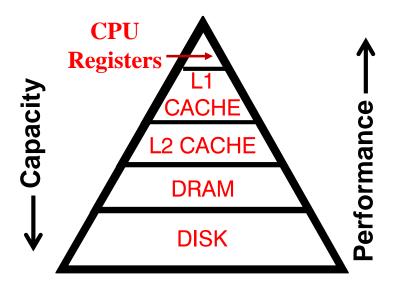
Spatial locality



### **Memory/Storage Hierarchies**

#### **Memory/Storage Hierarchies**

- Balancing performance with cost
  - Smaller memories are faster but expensive.
  - Larger memories are slower but cheaper.
- Exploit locality to get the best of both worlds
  - Reuse of nearness of accesses.
  - Results in most accesses using the smaller and faster memories.
  - Along with larger memory available which costs cheaper.



There other means of improving the performance of program execution by parallel processing techniques using multi-core processors.

our **RP2040** is itself a dual core processor.

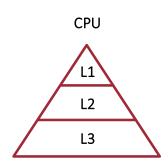
This enables sequential programs to run faster on a processor system ...

#### **Three Properties of Memory Levels**

- **Inclusion**: Any data that is part of a lower level (closer to the processor) needs to be present at the higher level
- Coherence (consistency): Multiple copies of the same data are available at each level (Registers, L1 cache, L2 cache, Main Memory, etc.)
  - All copies need to be identical or maintained consistent

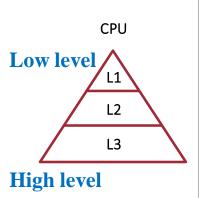


- **Temporal**: Will be used in the near future
- **Spatial**: Adjacent data are likely to be used often
- **Sequential**: Execution of instructions



#### **Properties and Issues with Memory Levels**

- The caches which are farther to the CPU are of larger sizes than the ones that are closer to the CPU
  - Size (L3) > Size (L2) > Size (L1)
- The sizes of caches are always in multiples of powers of two
  - Sizes of higher-level caches would also be multiples of lower-level cache sizes
  - Example: Size (L1) = 32 KB, Size (L2) = 256 KB, Size (L3) = 2 MB.
- If some code or data has been brought into L1 cache, the same copy of code or data will also be present at all the higher-level caches
  - Having multiple copies of the same data has associated issues
  - Maintaining consistency across multiple copies of data during the execution of a program is a challenge
- This issue is called 'cache coherency', which is to be addressed in multiprocessor systems



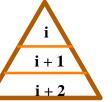
### Quiz: Relationship between Memory Levels

$$t_i < t_{i+1}$$

• Cost per Byte 
$$(c_i)$$

$$c_i > c_{i+1}$$

$$s_i \ < \ s_{i+1}$$



• Transfer Bandwidth (**b**<sub>i</sub>):

$$b_i > b_{i+1}$$

• Unit of transfer 
$$(x_i)$$

$$x_i < x_{i+1}$$

$$f_i > f_{i+1}$$

Notes: a) i is closer to the processor than i+1

b) Unit of Bandwidth is Bytes/Second

Access time: Time taken to access data from the memory

### ES 2: Summary

- Memory map of a C program recap
  - Software Development Tools
  - Program to an Executable
- Architectural View of Computing System recap
- CPU-DRAM Performance Gap
- Principle of Locality
- Memory/Storage Hierarchies
  - Three Properties of Memory Levels
  - Properties and Issues with Memory Levels

#### References - 1

Ref 0 Ref 1 Ref 2

SPECIAL ASSESSMENT OF THE PARTY P.

Getting started with Raspberry Pi Pico

C/C++ development with Raspberry Pi Pico and other RP2040-based microcontroller boards arm Education Media

Fundamentals of System-on-Chip Design on Arm Cortex-M Microcontrollers

TEXTBOOK

René Beuchat, Florian Depraz, Andrea Guerrieri, Sahand Kashani arm Education Media

Modern System-on-Chip Design on Arm

TEXTBOOK

David J. Greaves



#### References - 2

Ref 3 Ref 4 Ref 5

Cortex-M0+

Revision: r0p1

**Technical Reference Manual** 

Cortex - M0+ Devices

Generic User Guide

PRESA TOTAL CONTRACTOR SUSPENSION IN

RP2040 Datasheet

A microcontroller by Raspberry Pi

ARM

Copyright © 2012 ARM. All rights retented. ARM DOLOHOUT (DOLTETS) ARM

Copyright © 2012 ARM, 62 rights reserved. ARM DUI 100036 (DIO11713):

.

Ref 6

Raspberry Pi Pico C/C++ SDK

Libraries and tools for C/C++ development on RP2040 microcontrollers