Embedded Systems -2

ES 2: Focus

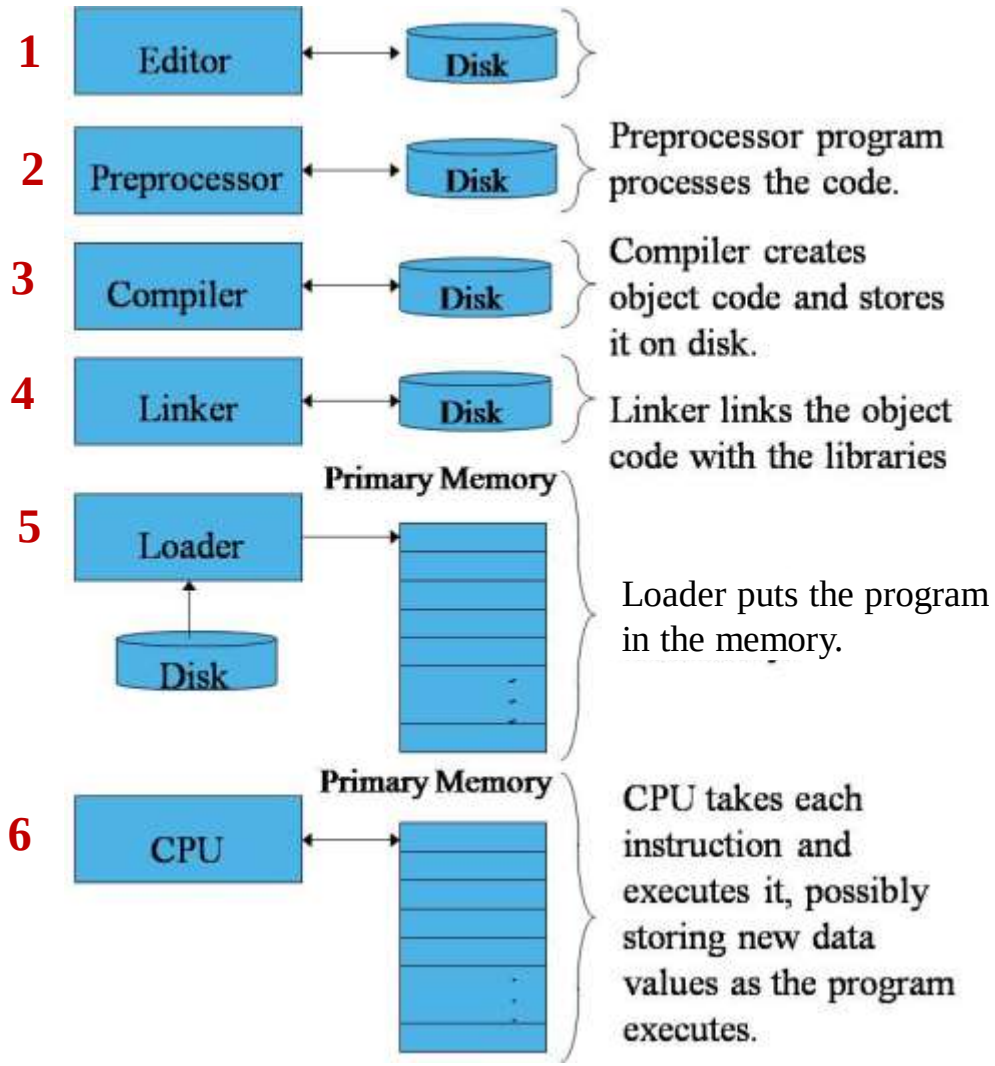
- Memory map of a C program - recap
  - Software Development Tools
  - Program to an Executable
- Architectural View of Computing System - recap
- CPU-DRAM Performance Gap
- Principle of Locality
- Memory/Storage Hierarchies
  - Three Properties of Memory Levels
  - Properties and Issues with Memory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

```
int  count;

count++;
```

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

```
1010011010
```

File: test.o (a binary file)

Output from the ***compiler***

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

```
10111000011010
```

File: test.exe

Output from the ***linker***

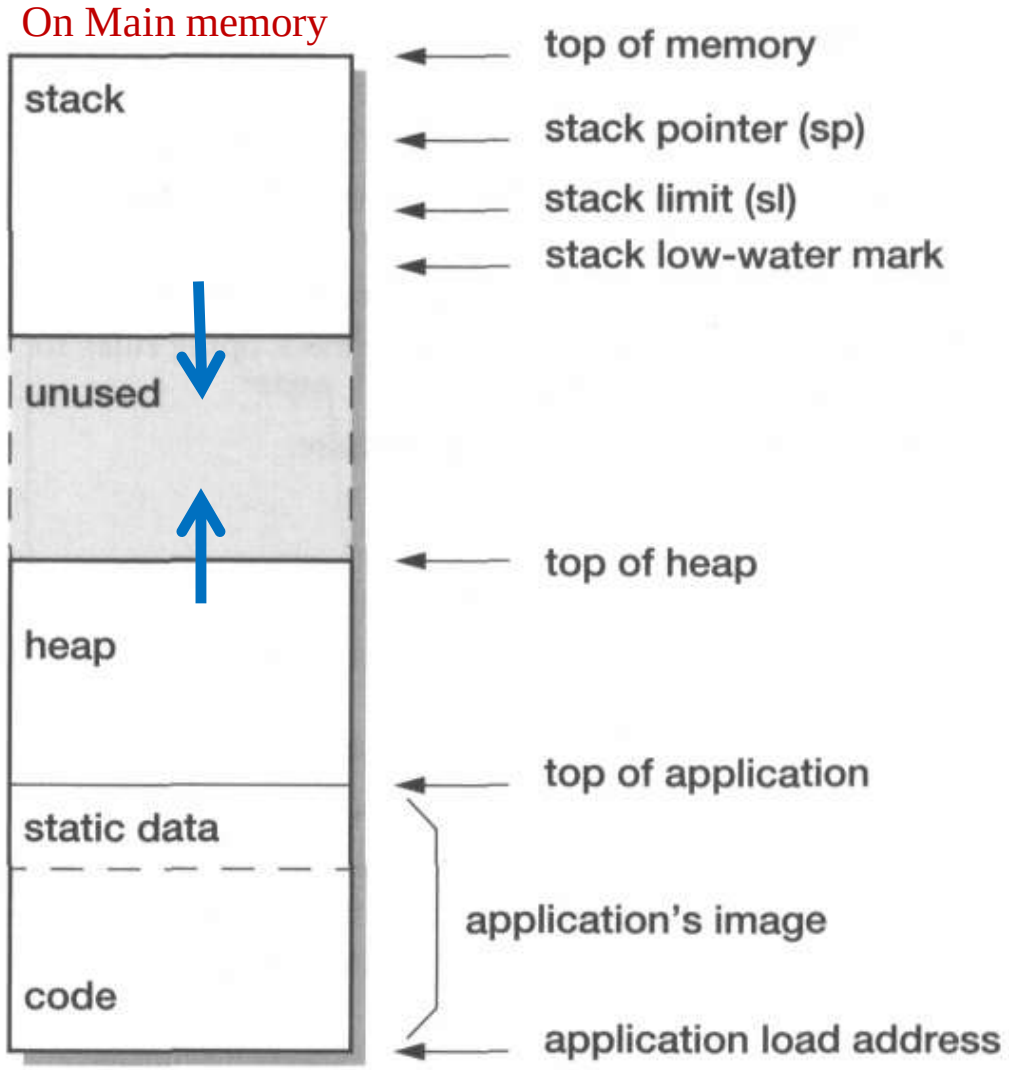
Load the executable into memory using a ***loader*** before the execution.

Standard 32 bit Processor
Address Space Model: An Example

Upper Address (0xFFFF FFFF)

32-bit Physical Address Space of size **4 GB**

Lower Address (0x0000 0000)

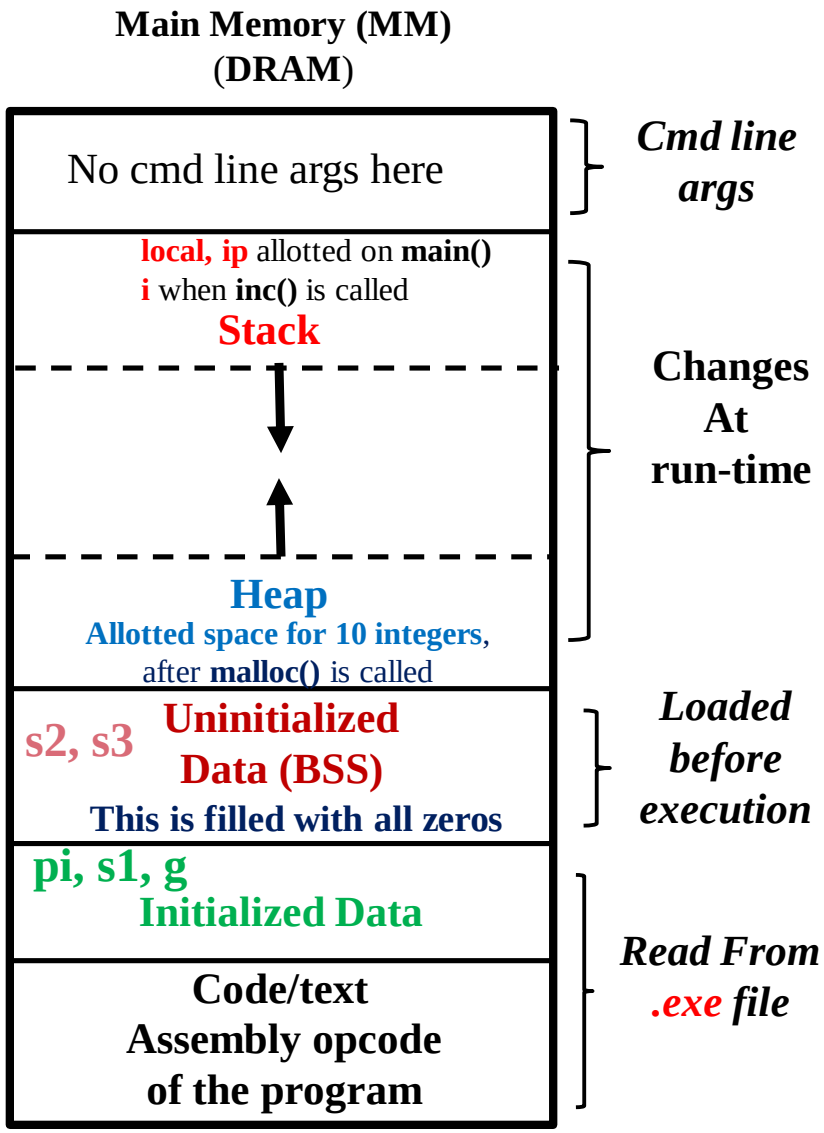


Let us now view the memory layout of a simple C program next …

Typical Memory Layout of C Program (Data)

```
#include <stdio.h>
#include <stdlib.h>
/* These global variables are not used here */
const float pi = 3.14;
static int s1 = 10;
static int s2;
int g = 2;

int inc(int i) {
    return i++;
}

main(void) {
    int local = 10; int *ip = NULL;
    static int s3;
    ip = (int *) malloc(sizeof(int) * 10);

    local = inc(local);

    free(ip); /* ip not used by the program */
    return 0;
} /* end of main() */
```

Note: OS prepares this program layout on MM by reading the executable from the hard disk, before giving the control to the program.



BSS: Block Started by Symbol

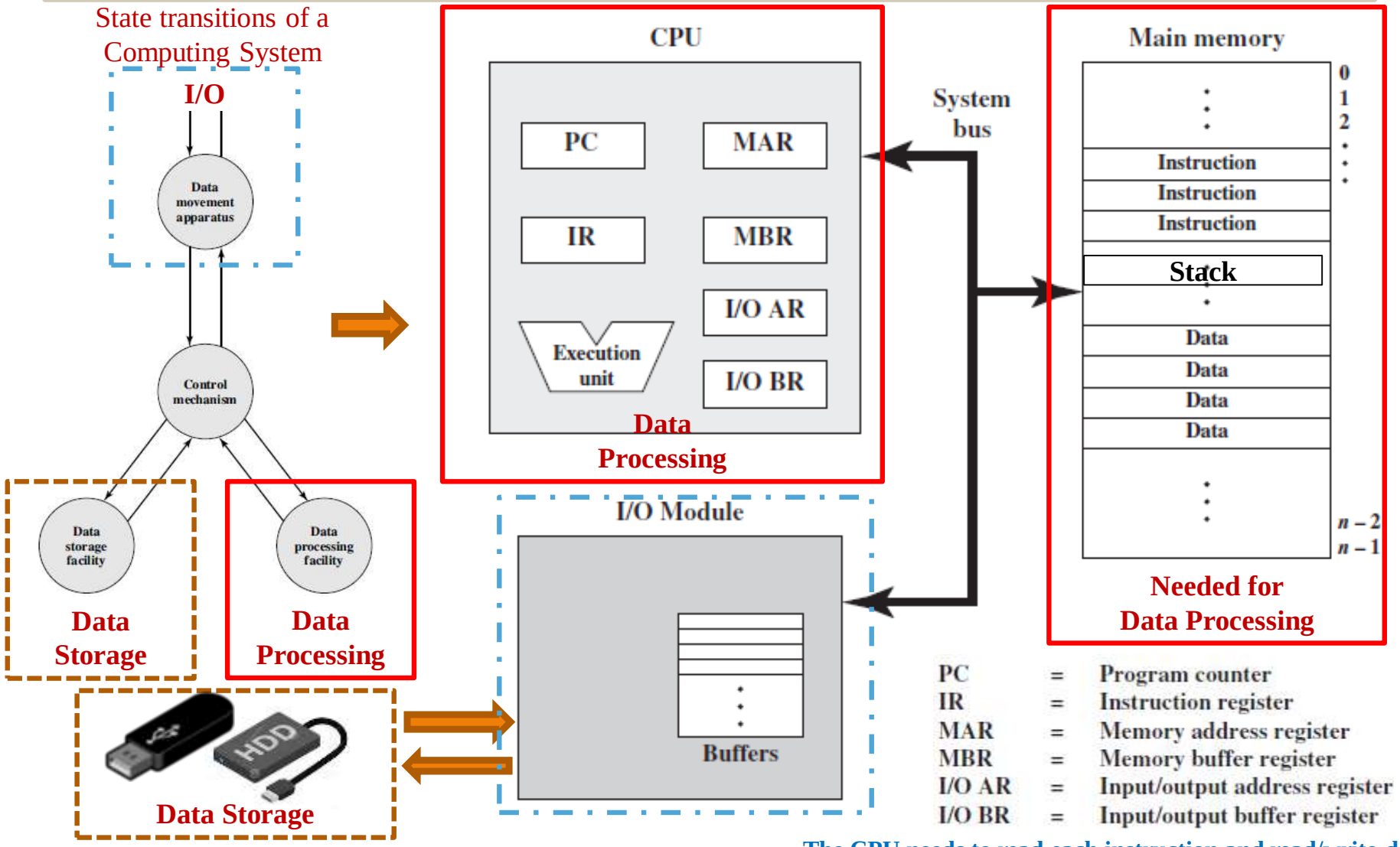
Architectural View of Computing System

Architectural View of a Computer System

(from the architecture point of view)

DRAM

State transitions of a Computing System



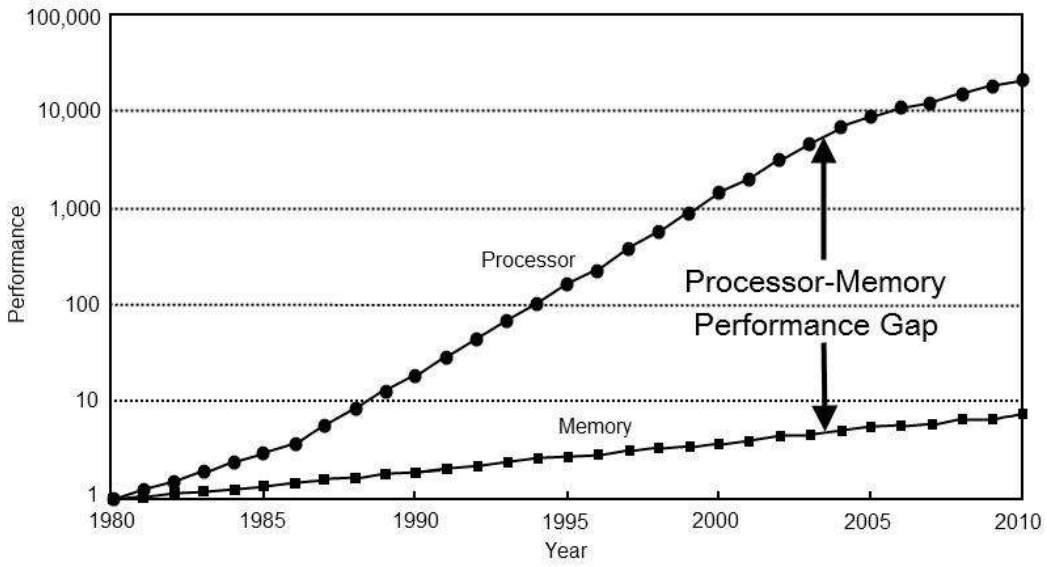
Control module coordinates the interactions between all the modules.

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 — Spatial locality
  - Cycling through the loop repeatedly — Temporal locality

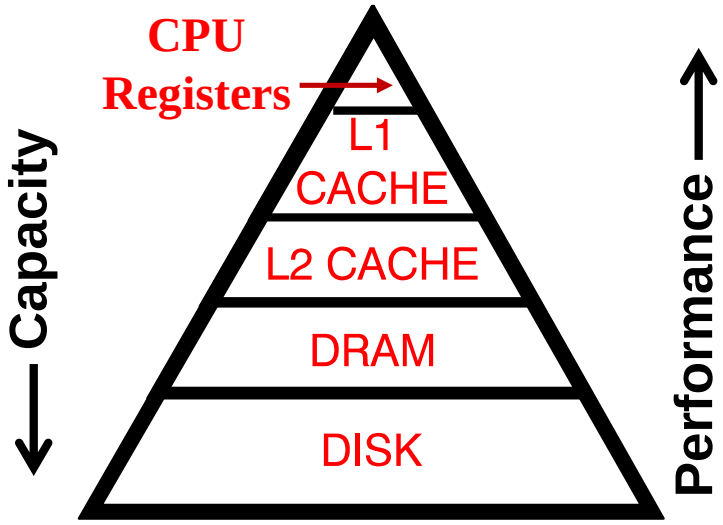
How can this property be leveraged to fill the gap between the speeds of CPU and memory? … let us see next …

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.**

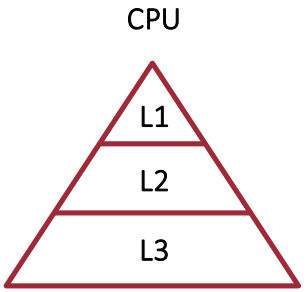
This enables sequential programs to run faster on a processor system …



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.

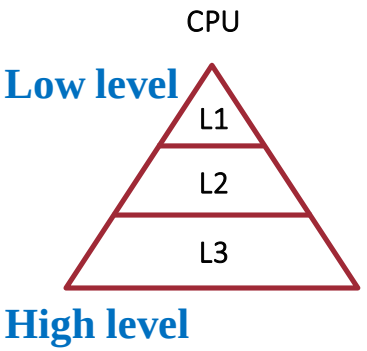
Three Properties of Memory Levels

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



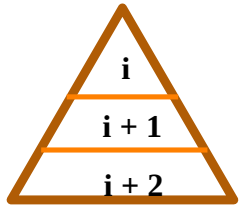
Properties and Issues with Memory Levels

- 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

- Access time (t_i) : $t_i < t_{i+1}$
- Cost per Byte (c_i) : $c_i > c_{i+1}$
- Memory size (s_i) : $s_i < s_{i+1}$
- Transfer Bandwidth (b_i) : $b_i > b_{i+1}$
- Unit of transfer (x_i) : $x_i < x_{i+1}$
- Frequency of access (f_i) : $f_i > f_{i+1}$



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

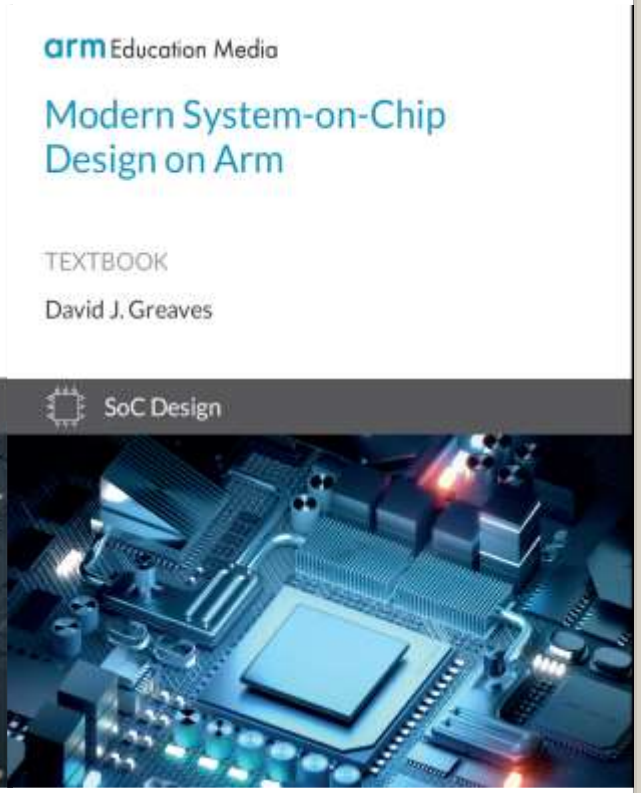
- Memory map of a C program - recap
  - Software Development Tools
  - Program to an Executable
- Architectural View of Computing System - recap
- CPU-DRAM Performance Gap
- Principle of Locality
- Memory/Storage Hierarchies
  - Three Properties of Memory Levels
  - Properties and Issues with Memory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

References - 1

Ref 0

Ref 1

Ref 2

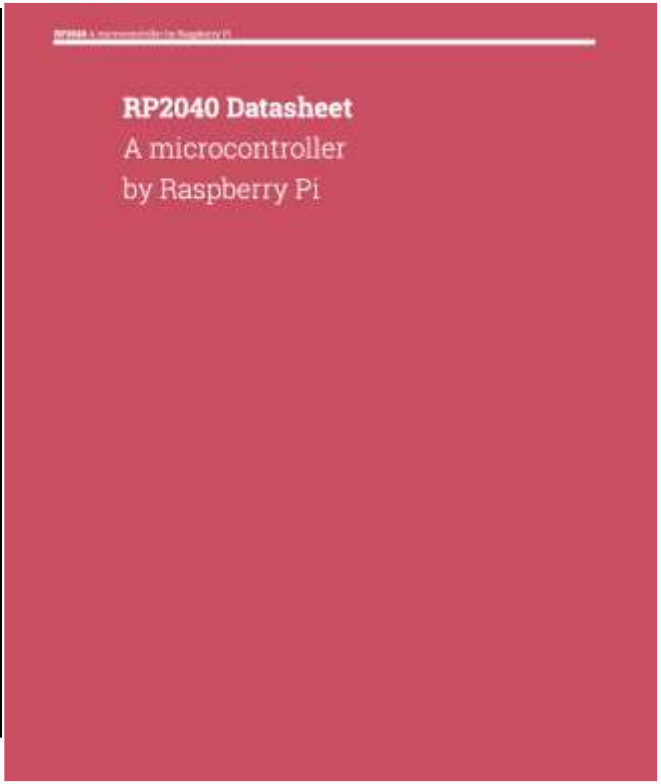


References - 2

Ref 3

Ref 4

Ref 5



Ref 6