Memory Layout

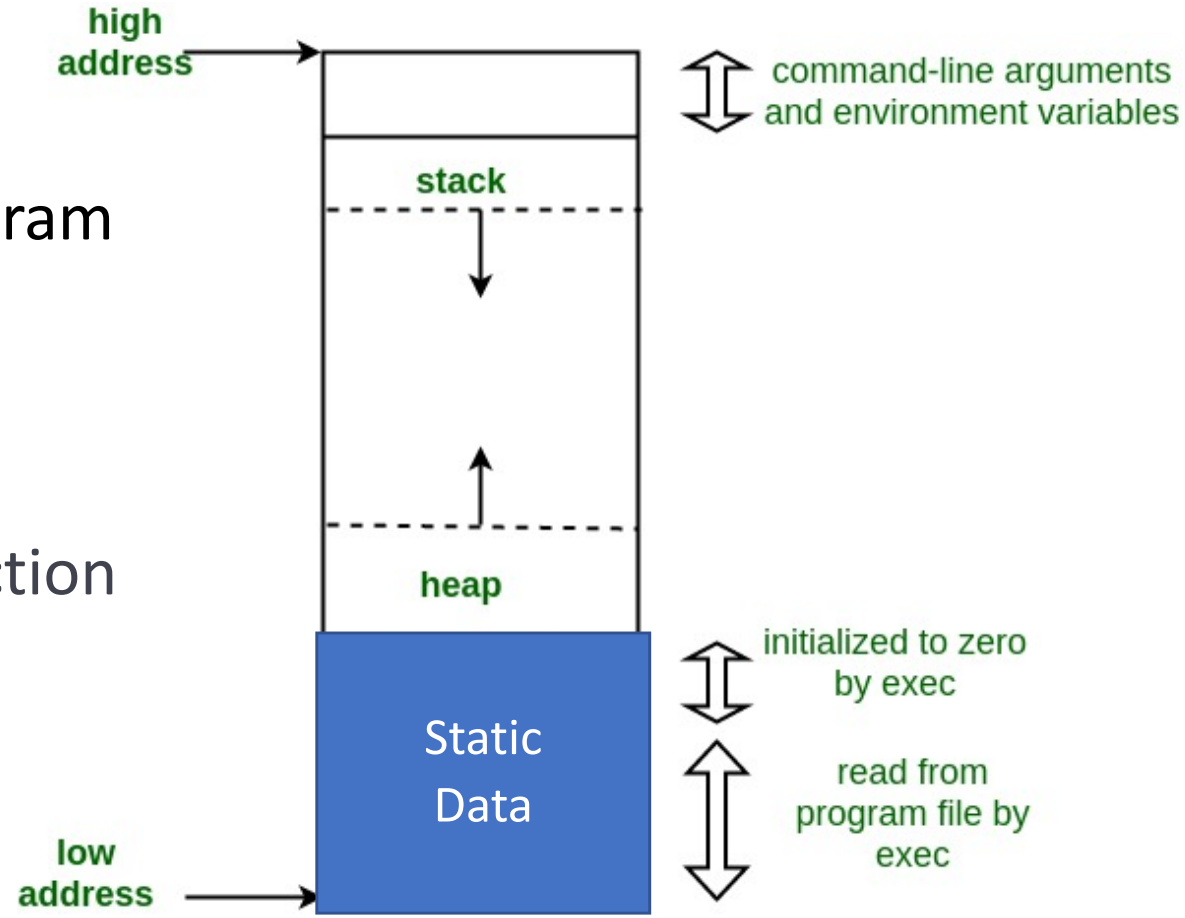
EECS388 Spring 2023

Lecture notes based in part on slides created by Prof. Mohammad Alian and Heechul Yun

Memory Layout of C Programs

A typical memory representation of a C program consists of three broad region.

1. Static Data
   - -fixed size
   - -stays for the entire program
   - -read only
2. Stack
   - -variable size
   - -grows when calling function
3. Heap
   - -variable size
   - -grows when allocated in program

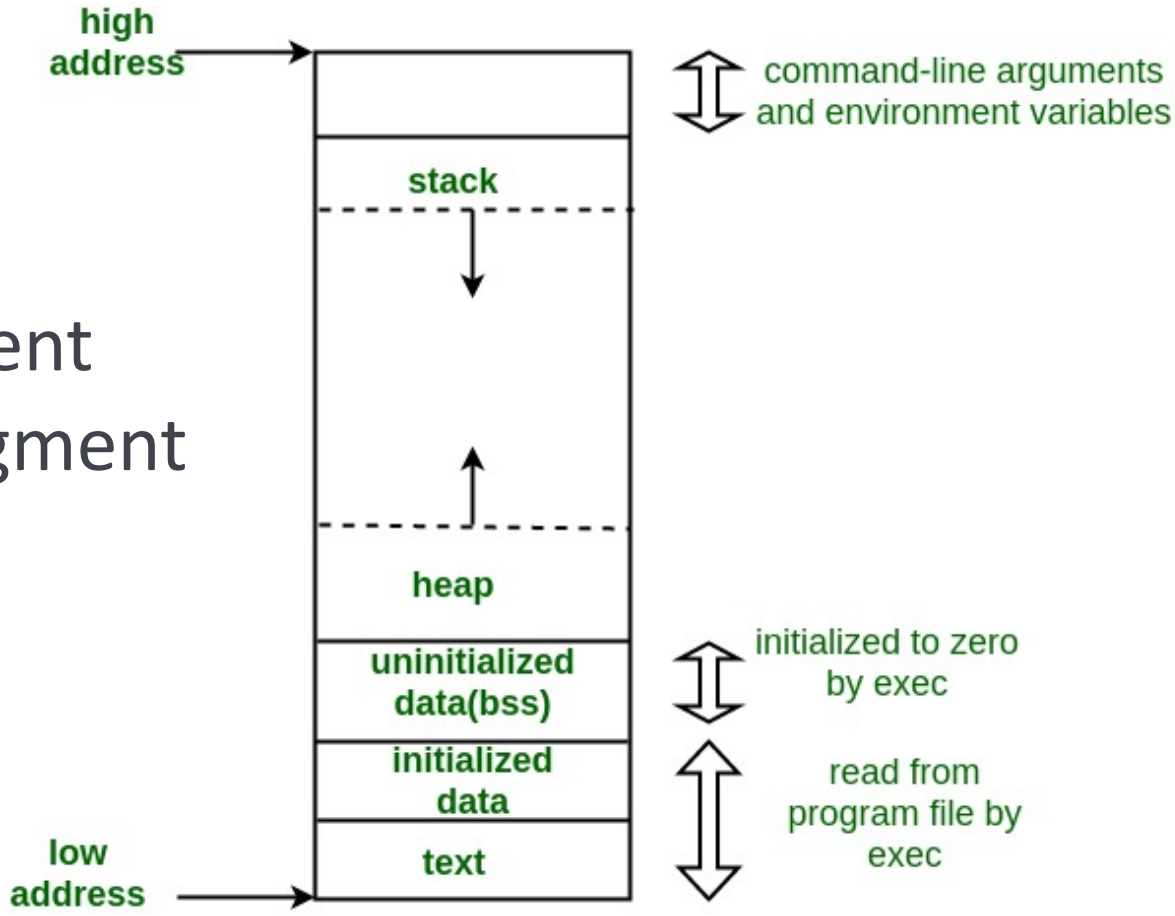


https://courses.engr.illinois.edu/cs225/sp2022/resources/stack-heap/

Memory Layout of C Programs

A typical memory representation of a C program consists of three broad region.

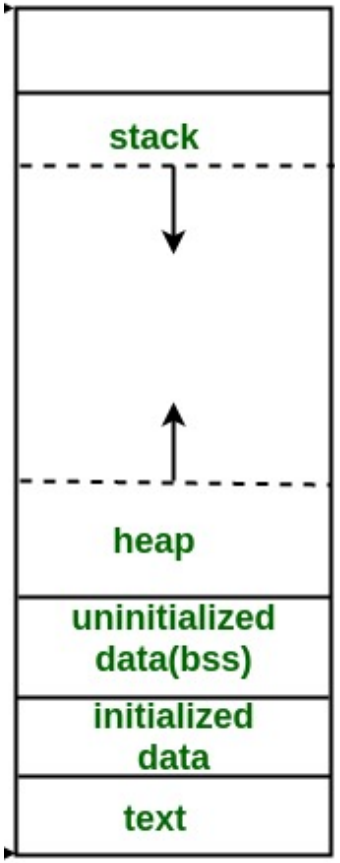
1. Static
   - -Text/code segment
   - -Initialized data segment
   - -Uninitialized data segment
2. Stack
3. Heap



https://courses.engr.illinois.edu/cs225/sp2022/resources/stack-heap/

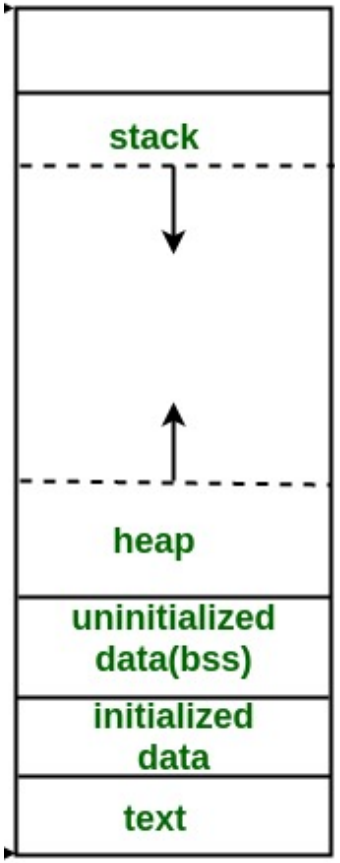
Text Segment

- Contains executable instructions.
- Placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.
- Text segment is often read-only →to prevent a program from accidentally modifying its instructions.



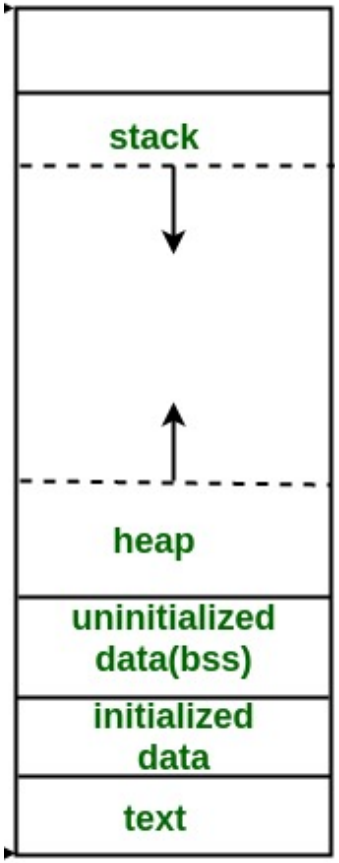
Initialized Data Segment:

- Also called just `data segment’
- contains the global variables and static variables that are initialized in the code
- Data segment is not read-only, since the values of the variables can be altered at run time.
  - Example: Int i=5;



Uninitialized Data Segment:

- often called the “bss”
- contains all global variables and **static variables** that are initialized to zero
- or do not have explicit initialization in source code.
- Example:
  - int i;
  - Int i=0;



Static Variable

- Static variables preserve their previous value in their previous scope and are not initialized again in the new scope.

```c
#include<stdio.h>
int fun()
{
  static int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Output: 1, 2

```c
#include<stdio.h>
int fun()
{
  int count = 0;
  count++;
  return count;
}

int main()
{
  printf("%d ", fun());
  printf("%d ", fun());
  return 0;
}
```

Output: 1, 1

Example

- Let’s take a sample program and see the memory layout

```c
#include <stdio.h>

int main(void)
{
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960        248           8       1216        4c0    memory-layout
```

Example: Global Variable

- Let us add one global variable
- check the size of bss (increased by 4)

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    return 0;
}
```

```
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
 960        248         12       1220        4c4    memory-layout
```

Example: Static Variable

- Let us add one static variable (uninitialized)
- stored in bss (increased by 4).

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i; /* Uninitialized static variable stored in bss */
    return 0;
}
```

```
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
 960        248         16       1224        4c8    memory-layout
```

Example: Initialized Static Variable

- Let us initialize the static variable
- stored in data segment, instead of bss.

```c
#include <stdio.h>

int global; /* Uninitialized variable stored in bss*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

```
[narendra@CentOS]$ gcc memory-layout.c -o memory-layout
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960         252          12       1224        4c8    memory-layout
```

Example: Initialized Global Variable

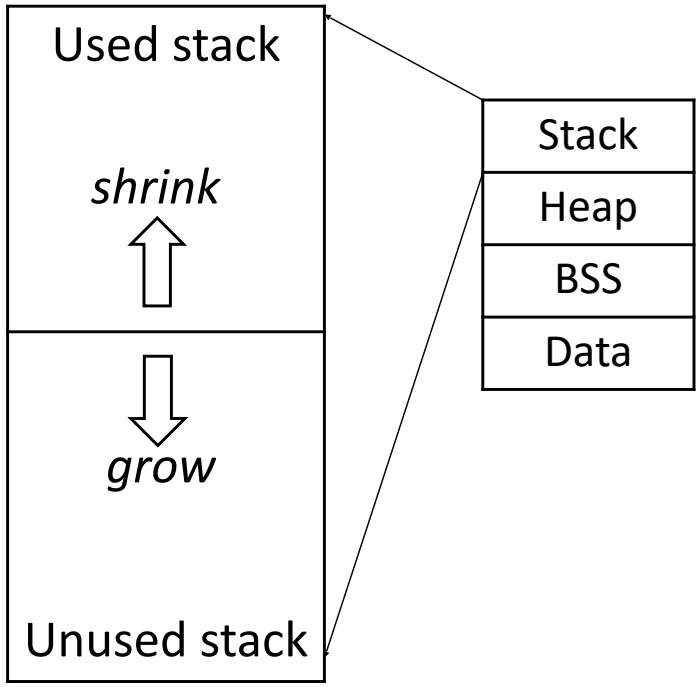
- Let us initialize the global variable
- stored in data segment, instead of bss.

```c
#include <stdio.h>

int global = 10; /* initialized global variable stored in DS*/

int main(void)
{
    static int i = 100; /* Initialized static variable stored in DS*/
    return 0;
}
```

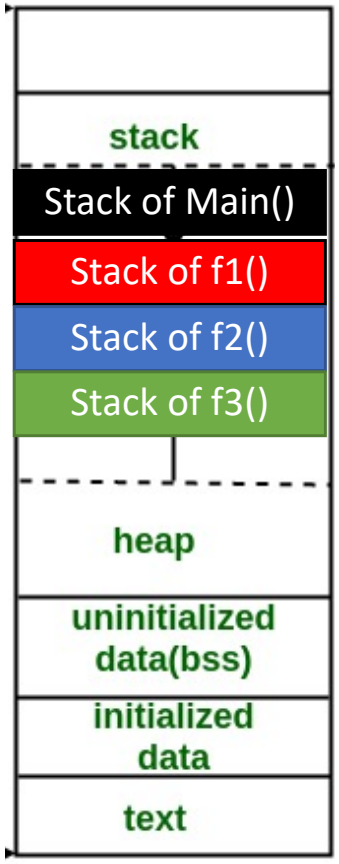
```
[narendra@CentOS]$ size memory-layout
text       data        bss        dec        hex    filename
960         256          8       1224        4c8    memory-layout
```

Stack

- Temporary storage
  - For functions
- Grow/shrink dynamically
  - Call a function → grow
  - Exit a function → shrink
- A stack frame
  - Local variables
  - Input parameters
  - Return address/value
  - Previous stack frame pointer …



Stack Example




```
int main(){
      //…
      f1();
      return 0;
}

int f1(){
      //…
      f2();
}

Void f2(){
      //…
      f3();
}


Void f3(){
      //…
}
```

Sample entry sequence

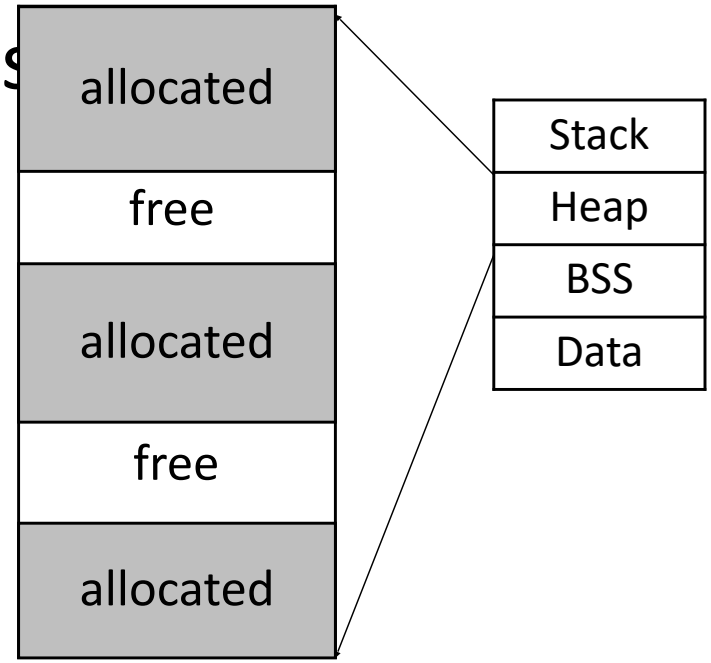
```
addi sp, sp, -8
sw ra, 0(sp)
sw a0, 4(sp)
```

Corresponding Exit sequence

```
lw ra, 0(sp)
lw a0, 4(sp)
addi sp, sp, 8
```

Heap

- segment where dynamic memory allocation usually takes place.
- Reserved at compile time
- Allocated/freed at runtime
  - malloc()
  - free()




```
// Dynamically allocate memory using malloc()
ptr = (int*)malloc(n * sizeof(int));
```

Not recommended to use for critical embedded applications (e.g., automotive)

Data Memory (Data Segment)

- **Stack**: temporary data like local variables
- **Heap**: dynamically allocated data
  - `malloc` (similar to `new` in C++)
- **Data**: non-zero initialized global and static data
- **BSS** (Block Started by Symbol): zero initialized and uninitialized global and static data

```
B int globA;
D int globB = 1;
  int main () {
S   int varA;
S   int varB = 10;
B   static int varC = 0;
D   static int varE = 1;
S   char *varD;
H   varD = (char*)malloc(8);
    varA = varB + varC;
    return varA;
  }
```

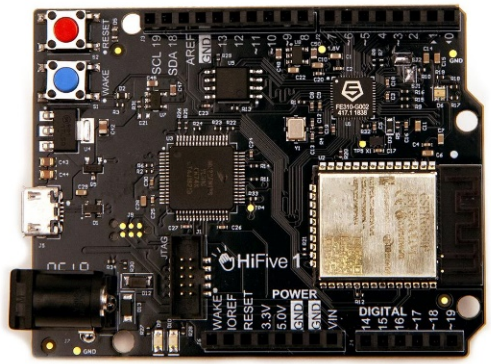
Memory Map of SiFive FE310

Base	Top	Attr.	Description	Notes
0x0000_0000	0x0000_0FFF	RWX A	Debug	Debug Address Space
0x0000_1000				On-Chip Non Volatile Memory
0x0000_2000				
0x0000_3000				
0x0000_4000				
0x0001_0000				
0x0001_2000				
0x0002_0000				
0x0002_2000				
0x0200_0000				
0x0201_0000				
0x0800_0000				
0x0800_2000				
0x0C00_0000				
0x1000_0000				
0x1000_1000				
0x1000_8000				
0x1000_9000				
0x1001_0000				
0x1001_1000				
0x1001_2000				On-Chip Peripherals
0x1001_3000				
0x1001_4000				
0x1001_5000				
0x1001_6000				
0x1001_7000				
0x1002_3000				
0x1002_4000				
0x1002_5000				
0x1002_6000				
0x1003_4000				
0x1003_5000				
0x1003_6000				
0x2000_0000				Off-Chip Non-Volatile Memory
0x4000_0000				
0x8000_0000				On-Chip Volatile Memory
0x8000_4000				

Peripherals

Code memory

Data memory



CPU: 32 bit RISC-V

Clock: 320 MHz

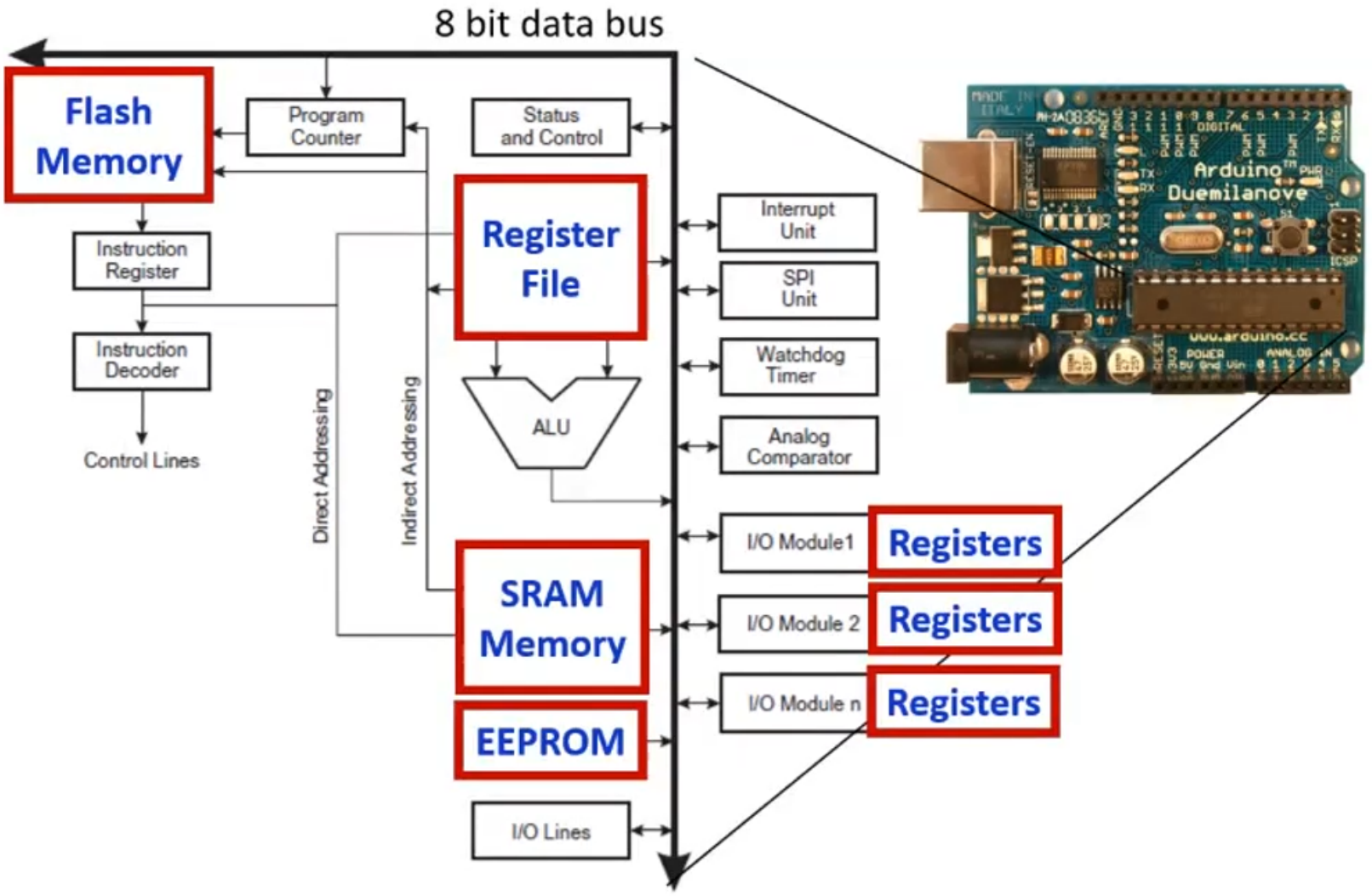
SRAM: 16 KB (D)

Flash: 4MB

Code/text

Stack, Heap, BSS, data

Context



Atmel ATmega168 Datapath

Context

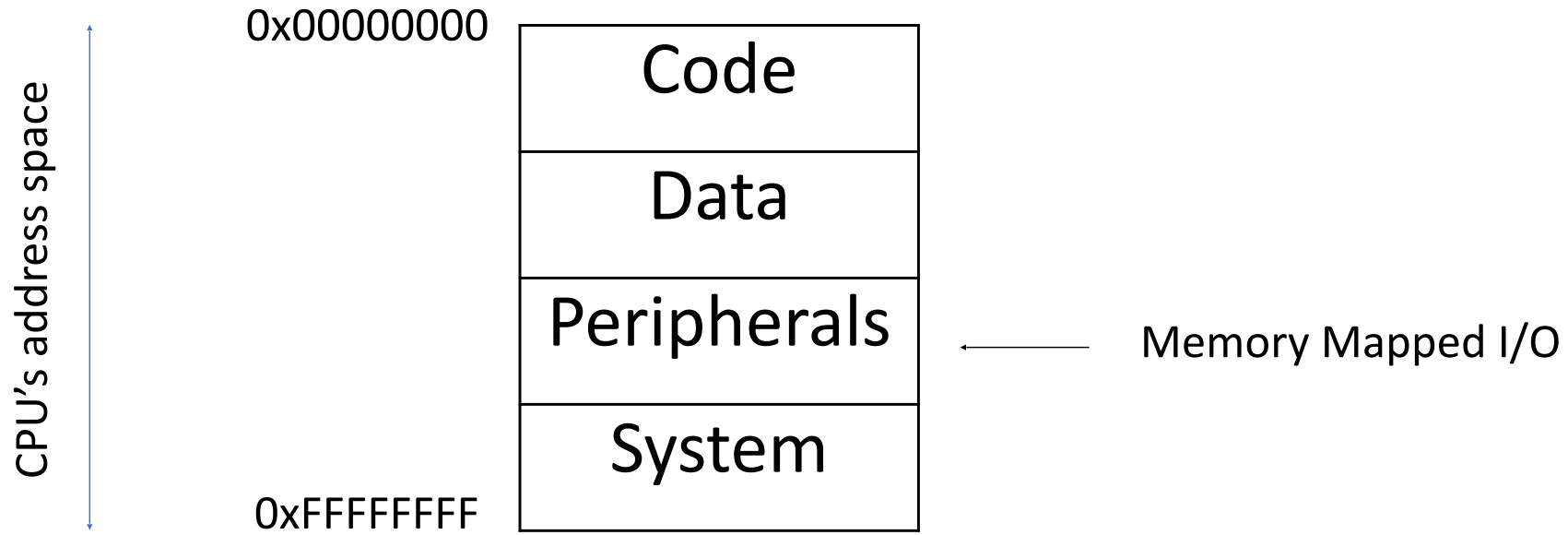
- Memory Mapped I/O
- Memory Segments

https://www.youtube.com/watch?v=D0VKuZuuvW8&ab_channel=LiveOverflow

Memory Map

CPU's view of the physical memory

- Segregated with multiple regions
- Different memory type for each region
- Each platform may have different mappings



Interfacing with I/O Devices

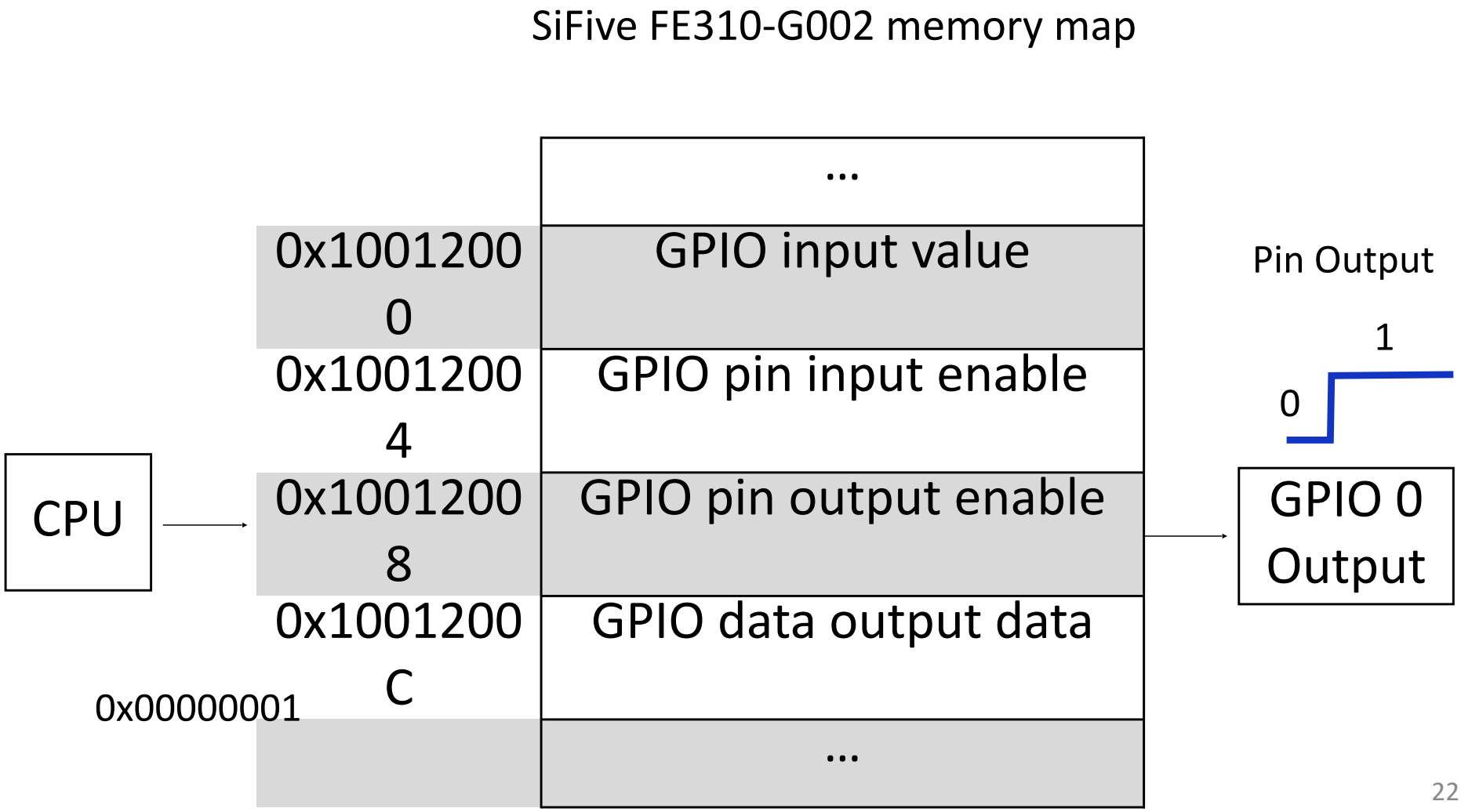
Port-mapped I/O

- Use a separate address space for I/O devices and use **special instructions** to access the I/O memory

Memory Mapped I/O

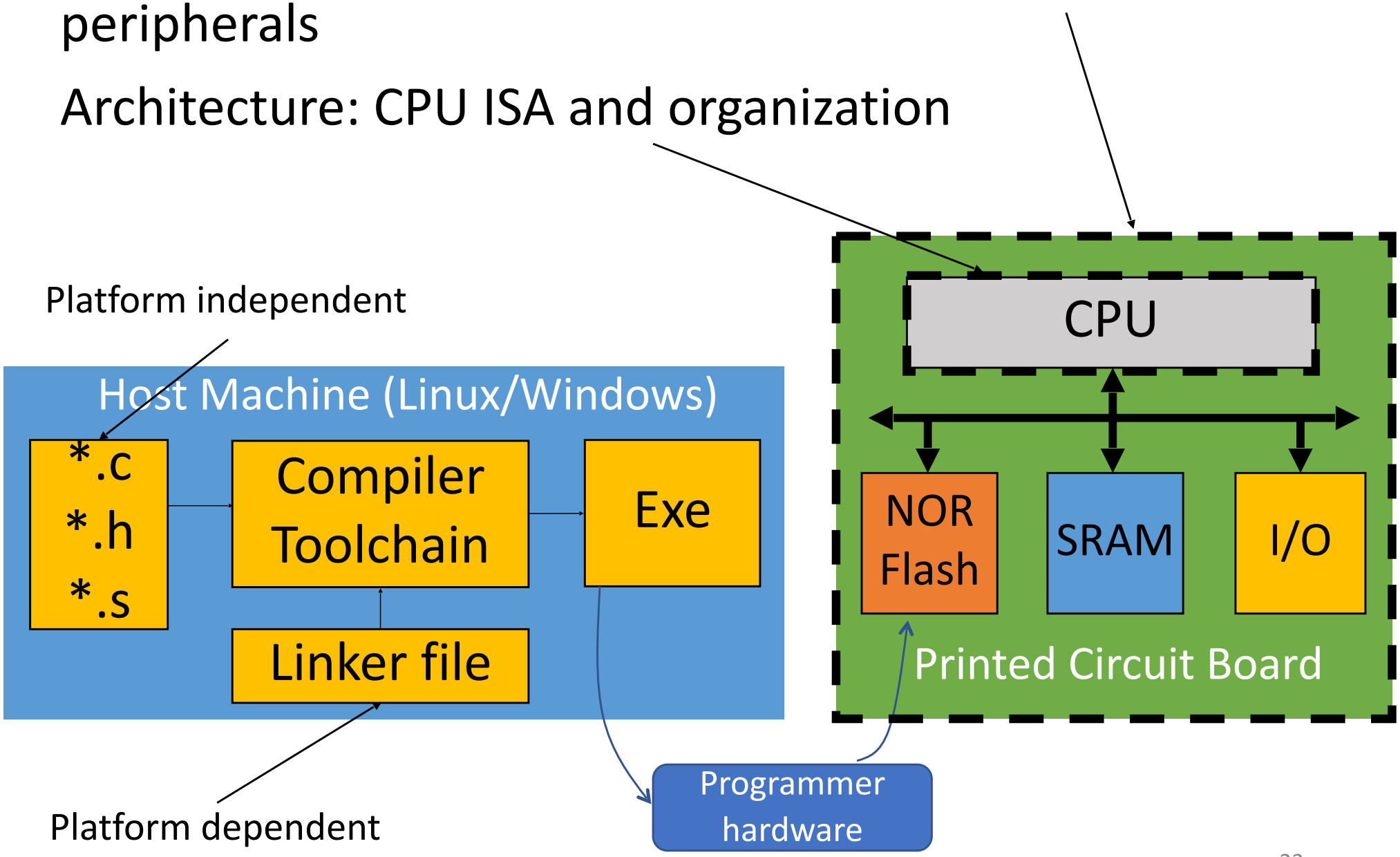
- I/O memory is mapped into the CPU address space
- **Use load/store instruction** to communicate with I/O devices

Example of Memory Mapped I/O



Platform: Components on the PCB, CPU and peripherals

Architecture: CPU ISA and organization



Linker Script

- Maps executable to physical memory locations on the platform