Heaven's Light Is Our Guide

Rajshahi University of Engineering & Technology

Department of Computer Science & Engineering

Running a Program

CSE 3105 (Computer Interfacing & Embedded Systems)

Md. Nasif Osman Khansur

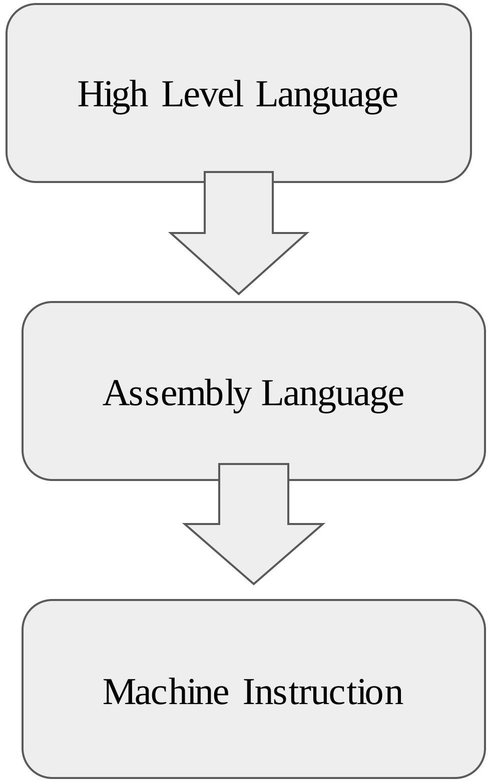
Lecturer

Dept. of CSE, RUET

Reference

Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C 3rd Ed - Yifeng Zhu [Chapter 1]

Translate a C Program into a Machine Program



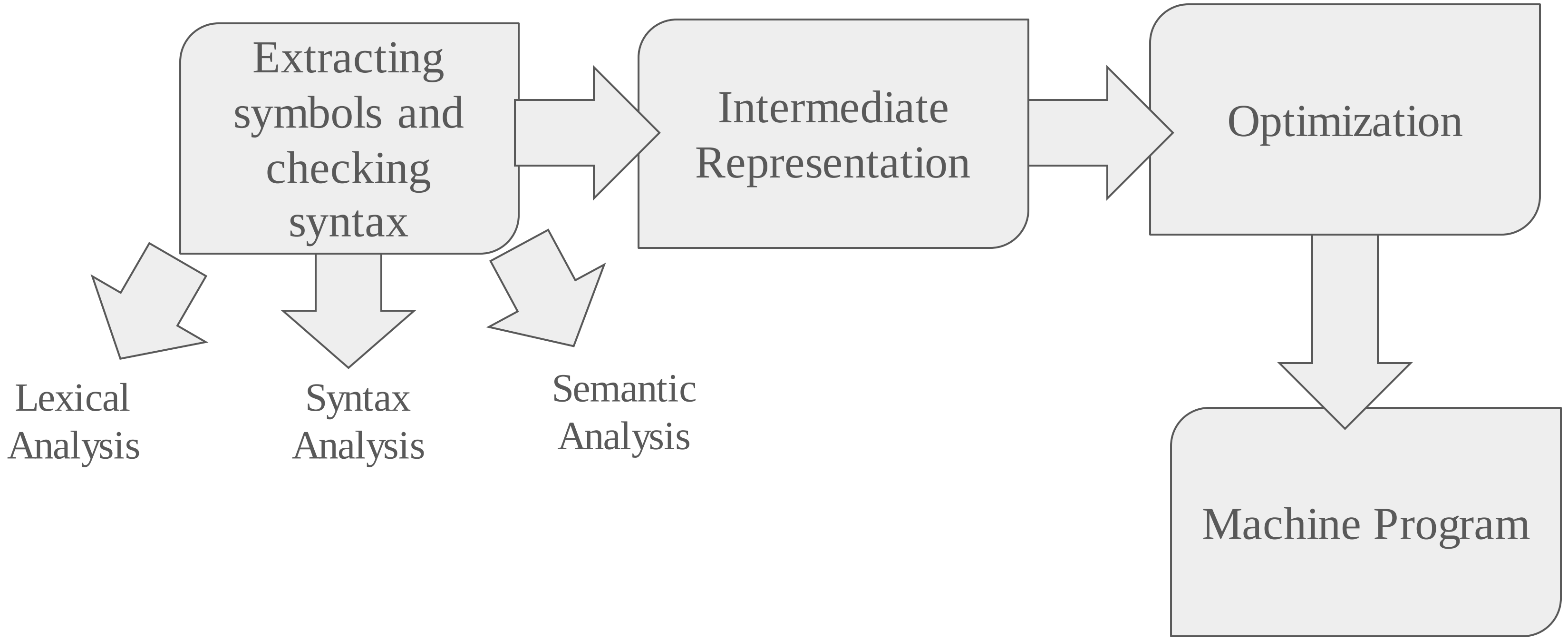
Translate a C Program into a Machine Program

- ❑ Executable files created by compilers are usually platform dependent.
- ❑ An executable file compiled for one type of microprocessors, such as ARM Cortex-M3, cannot directly run on a platform with a different kind of microprocessors that support a different set of machine instructions, such as PIC or Atmel AVR microcontrollers.
- ❑ When we migrate a program written in a high-level language to a processor of a different instruction set, we usually have to modify and recompile the source programs for the new target platform.

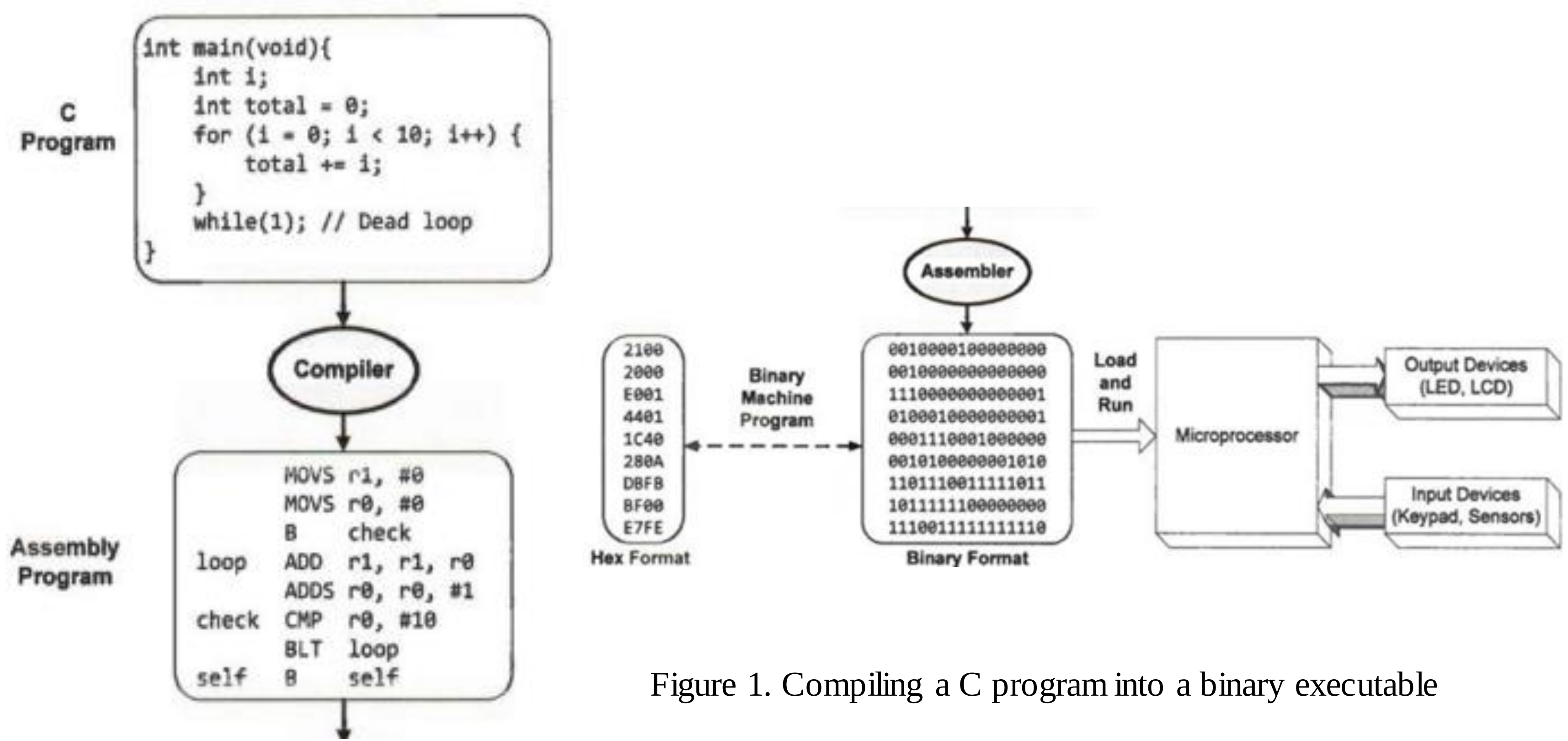
Translate a C Program into a Machine Program

- One exception is Java executables, which are platform independent.
- The Java compiler converts a Java program to bytecodes. A Java virtual machine (JVM) translates bytecode into machine instructions at runtime. Because each platform has its JVM, Java executables become platform-independent.
- Currently, Java is not popular in embedded systems yet because it needs more memory and cannot control peripherals flexibly.

Translate a C Program into a Machine Program



Translate a C Program into a Machine Program



```
int main(void){
    int i;
    int total = 0;
    for (i = 0; i < 10; i++) {
        total += i;
    }
    while(1); // Dead loop
}
```

```
        MOVS r1, #0
        MOVS r0, #0
        B    check
loop    ADD  r1, r1, r0
        ADDS r0, r0, #1
check   CMP  r0, #10
        BLT  loop
self    B    self
```

```
2100    0010000100000000
2000    0010000000000000
E001    1110000000000001
4401    0100010000000001
1C40    0001110001000000
280A    0010100000001010
DBFB    1101101111111011
BF00    1011111100000000
E7FE    1110011111111110
```

Figure 1. Compiling a C program into a binary executable

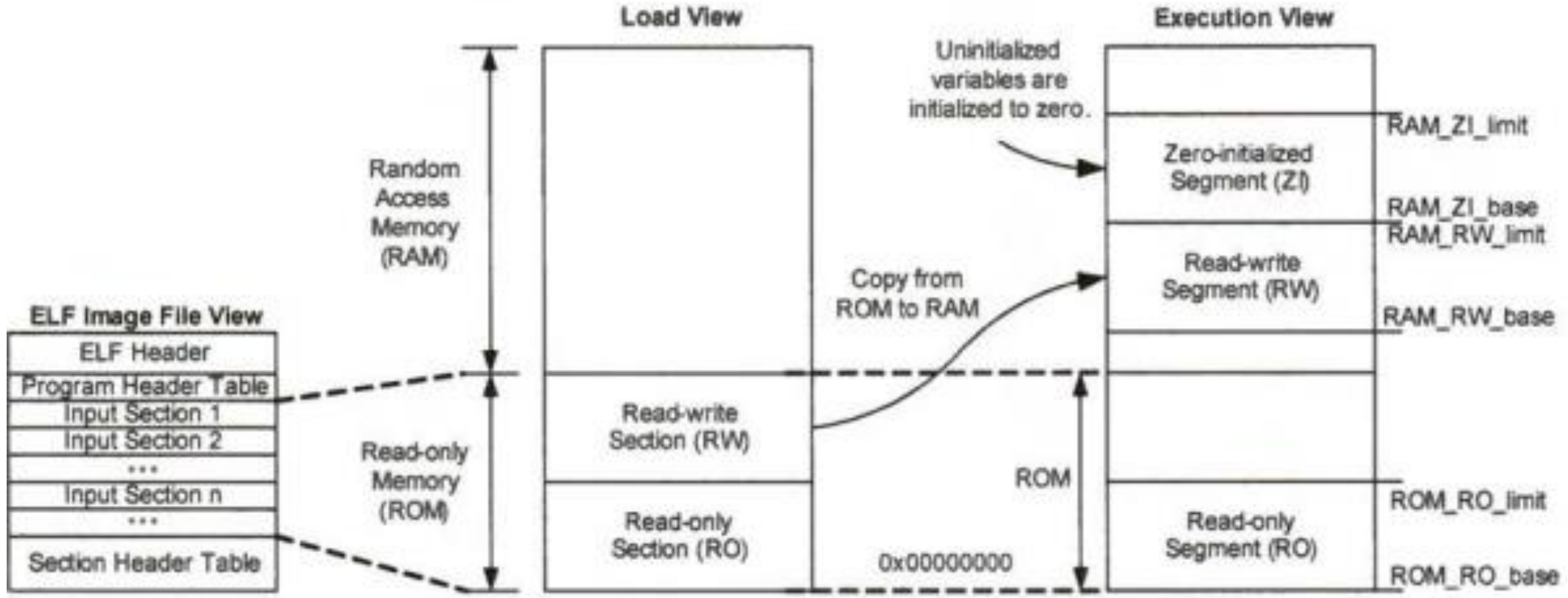
Translate a C Program into a Machine Program

- ❖ An assembly program includes the following **five** key components:

  1. A ***label*** represents the memory address of a data variable or an assembly instruction.
  2. An ***instruction mnemonic*** is an operation that the processor should perform, such as "ADD" for adding integers.
  3. ***Operands*** of a machine operation can be numeric constants or processor registers.
  4. A ***program comment*** aims to improve inter-programmer communication and code readability by explicitly specifying programmers' intentions, assumptions, and hidden concepts.
  5. An ***assembly directive*** is not a machine instruction, but it defines the data content or provides relevant information to assist the assembler.

Translate a C Program into a Machine Program

An ELF file provides load view and execution view. The load view specifies how to load data into the memory. The execution view instructs how to initialize data regions at runtime.

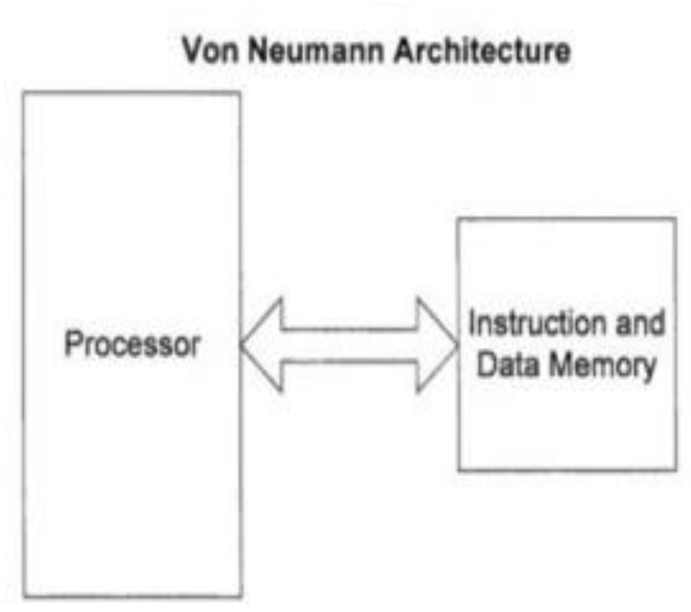


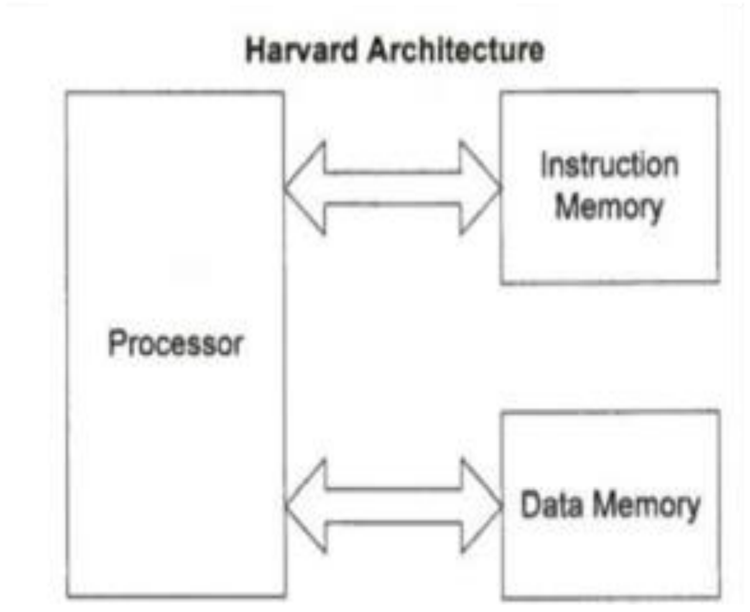
Translate a C Program into a Machine Program

A binary machine program includes four critical sections, including:

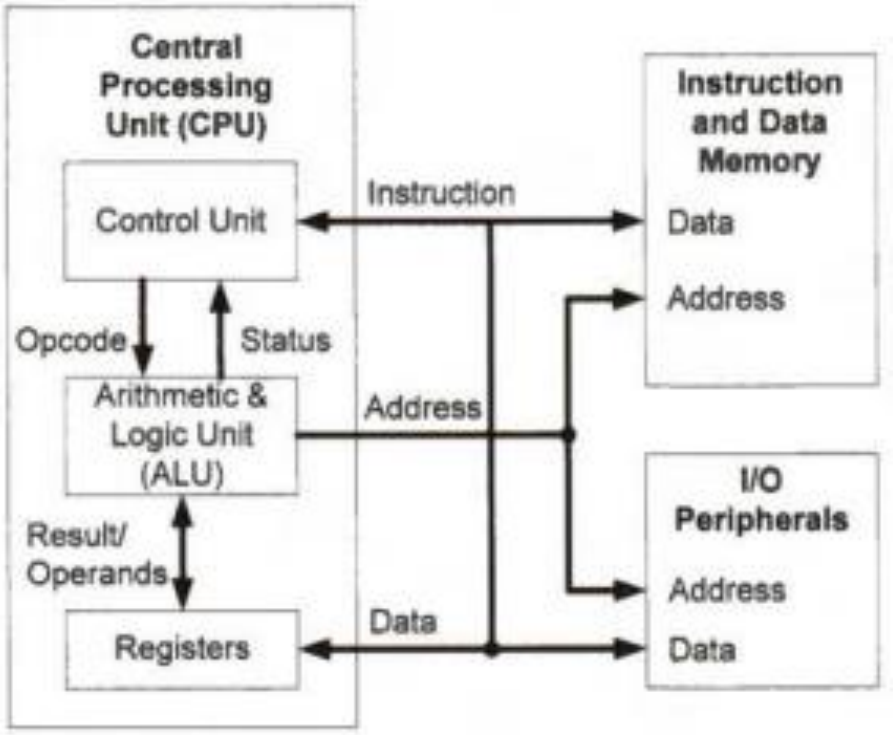
- a text segment that consists of binary machine instructions,
- a read-only data segment that defines the value of variables unalterable at runtime,
- a read-write data segment that sets the initial values of statically allocated and modifiable variables, and
- a zero-initialized data segment that holds all uninitialized variables declared in the program

Harvard Architecture and Von Neumann Architecture



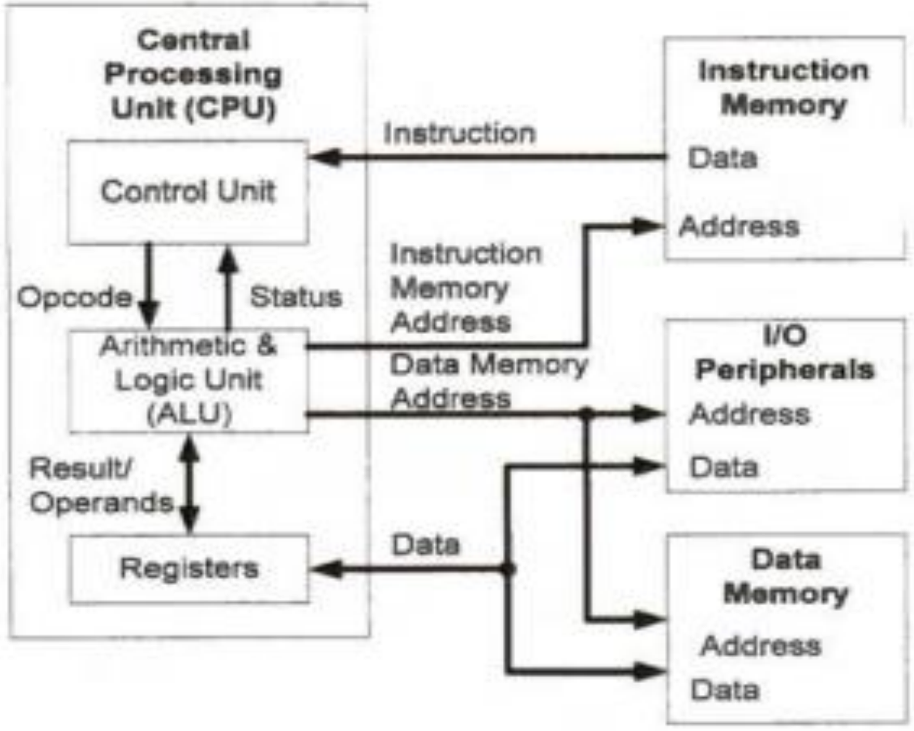


Von Neumann Architecture



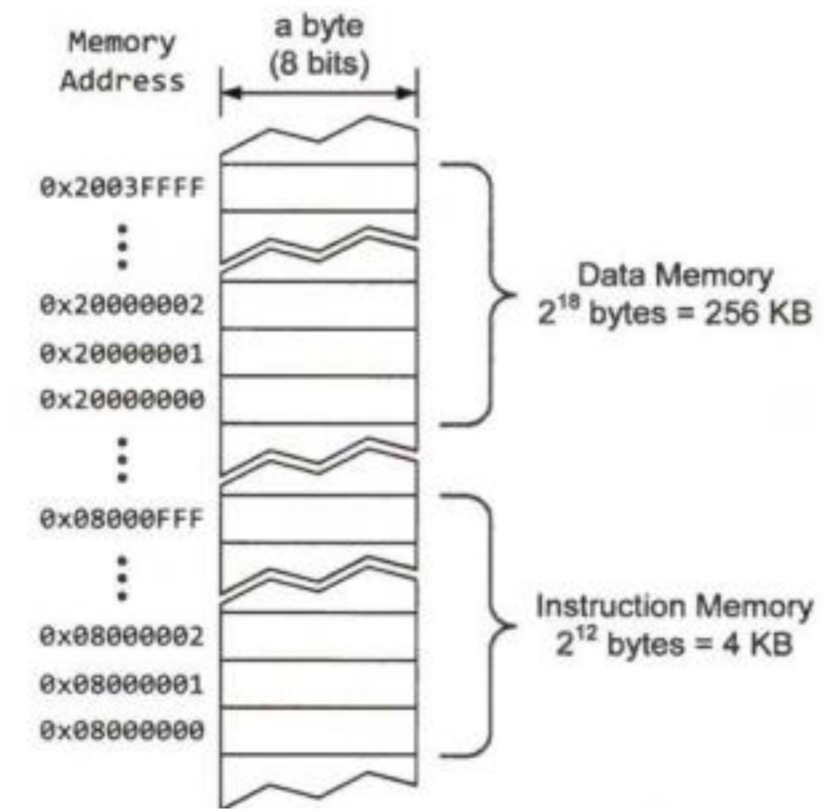
Instructions and data share the memory device. It has only one set of data bus and address bus shared by the instruction memory and the data memory. The data stream and the instruction stream share the memory bandwidth.

Harvard Architecture



Instructions and data are stored in different memory devices. It has a dedicated set of data bus and address bus for the instruction memory and the data memory. The instruction stream and the data stream transfer information on separate sets of data and address buses.

Harvard Architecture



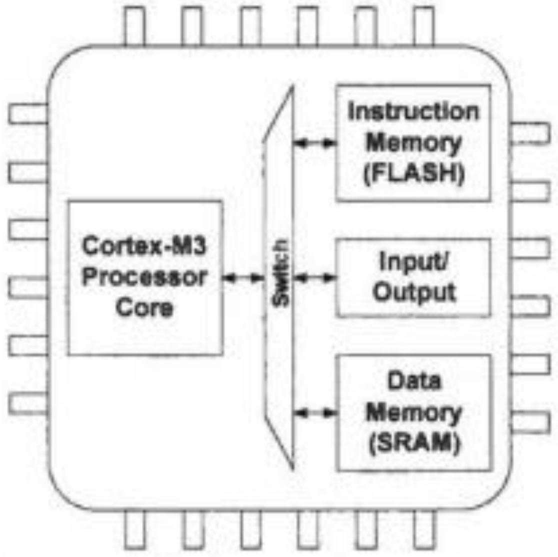
Data memory and instruction memory are in the same memory address space in Harvard Architecture in many embedded systems. Accordingly, the instruction memory and the data memory can share the same address bus.

Harvard Architecture and Von Neumann Architecture

- The Von Neumann architecture is relatively inexpensive and simple.
- The Harvard architecture allows the processor to access the data memory and the instruction memory concurrently. By contrast, the Von Neumann architecture allows only one memory access at any time instant; the processor either read an instruction from the instruction memory or accesses data in the data memory. Accordingly, the Harvard architecture often offers faster processing speed at the same clock rate.
- The Harvard architecture tends to be more energy efficient. Under the same performance requirement, the Harvard architecture often needs lower clock speeds, resulting in a lower power consumption rate.
- In the Harvard architecture, the data bus and the instruction bus may have different widths. For example, digital signal processing processors can leverage this feature to make the instruction bus wider to reduce the number of clock cycles required to load an instruction.

Creating Runtime Memory Image

- ❑ ARM Cortex-M3/M4/M7 microprocessors are Harvard computer architecture, and the instruction memory (flash memory) and the data memory (SRAM) are built into the processor chip.
- ❑ Separating the instruction and data memories allows concurrent accesses to instructions and data, thus improving the memory bandwidth and speeding up the processor performance.
- ❑ Typically, the instruction memory uses a slow but nonvolatile flash memory, and the data memory uses a fast but volatile SRAM.



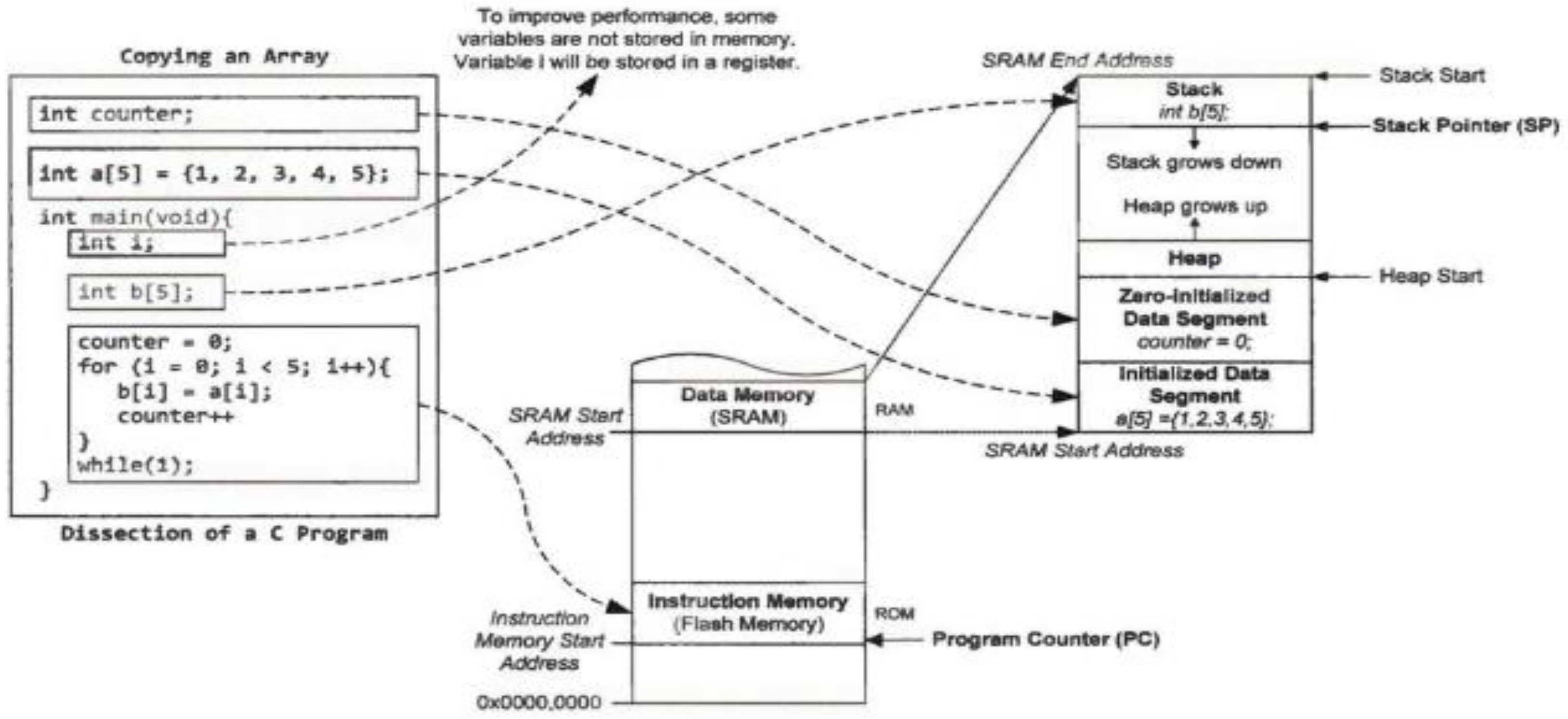
Creating Runtime Memory Image

- At runtime, the data memory is divided into four segments: initialized data segment, uninitialized data segment, heap, and stack. The processor allocates the first two data segments statically, and their size and location remain unchanged at runtime. The size of the last two segments changes as the program runs.
- The **initialized data segment** contains global and static variables that the program gives some initial values. For example, in a C declaration, "int capacity = 100;", if it appears outside any function (i.e., it is a global variable), the processor places the variable capacity in the initialized data segment with an initial value when the processor creates a running time memory image for this C program.
- The **zero-initialized data segment** contains all global or static variables that are uninitialized or initialized to zero in the program. For example, a globally declared string "char name[20];" is stored in the uninitialized data segment.

Creating Runtime Memory Image

- The **heap** holds all data objects that an application creates dynamically at runtime. For example, all data objects created by dynamic memory allocation library functions like malloc() or calloc() in C or by the new operator in C++ are placed in the heap. A free() function inC or a delete operator in C++ removes a data object from the heap.
- The **stack** stores local variables of subroutines, including main(), saves the runtime environment and passes arguments to a subroutine. A stack is a first-in, last-out (FILO) memory region, and the processor places it on the top of the data memory. When a subroutine declares a local variable, the variable is saved in the stack. When a subroutine returns, the subroutine should pop from the stack all variables it has pushed.

Creating Runtime Memory Image



```
int counter;

int a[5] = {1, 2, 3, 4, 5};
int main(void){
    int i;
    int b[5];
    counter = 0;
    for (i = 0; i < 5; i++){
        b[i] = a[i];
        counter++
    }
    while(1);
}
```

A processor of Harvard architecture loads a program into the instruction memory and the data memory.

Reusing Registers to Improve Performance

Question: Why some variable is not stored in memory rather stored in register?

Answer: Variables stored in registers rather than memory are typically those that are heavily used and require fast access. Registers are much faster to access than memory because they are part of the CPU itself, whereas accessing memory involves traversing buses and interacting with potentially slower components. When a variable is stored in a register, it means that the CPU can directly manipulate the variable's value without having to fetch it from memory. This can significantly speed up the execution of code that heavily uses these variables, particularly in tight loops or performance-critical sections of code. In languages like C or C++, the register keyword can be used as a hint to the compiler that a particular variable should be stored in a register if possible. However, modern compilers are often able to make these decisions automatically based on their optimization algorithms and the characteristics of the target hardware.

Reusing Registers to Improve Performance

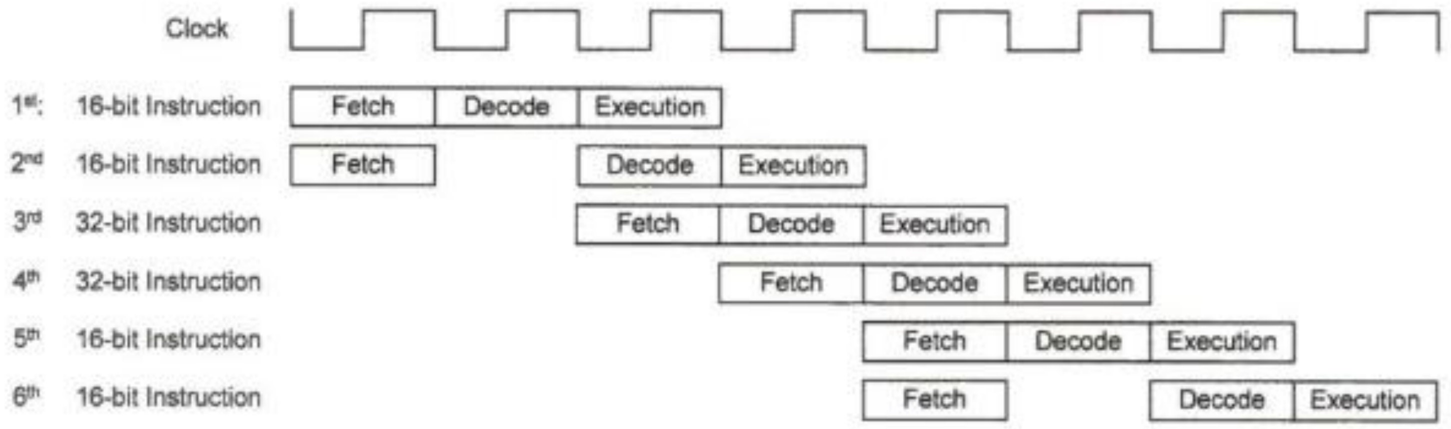
Self Study*: Temporal, Spatial Locality, Register Allocation, Processor Registers (Article 1.3.1 & 1.3.2)*

Life cycle of an instruction:

- At the first stage, the processor fetches 4 bytes from the instruction memory and increments the program counter by 4 automatically. After each instruction fetch, the program counter points to the next instruction(s) to be fetched.
- At the second stage, the processor decodes the instruction and finds out what operations are to be carried out.
- At the last stage, the processor reads operand registers, carries out the designated arithmetic or logic operation, accesses data memory (if necessary) and updates target registers (if needed)

Reusing Registers to Improve Performance

Pipelining allows multiple instructions to run simultaneously. Thus, it increases the utilization of hardware resources and improves the processor's overall performance.



Executing a Machine Program

Extended Study*: Loading the program (1.4.1), Starting the Execution (1.4.2), Program Completion (1.4.3)*

Thank
YOU