Review: CPU Architecture and Program Execution

Lecture 2

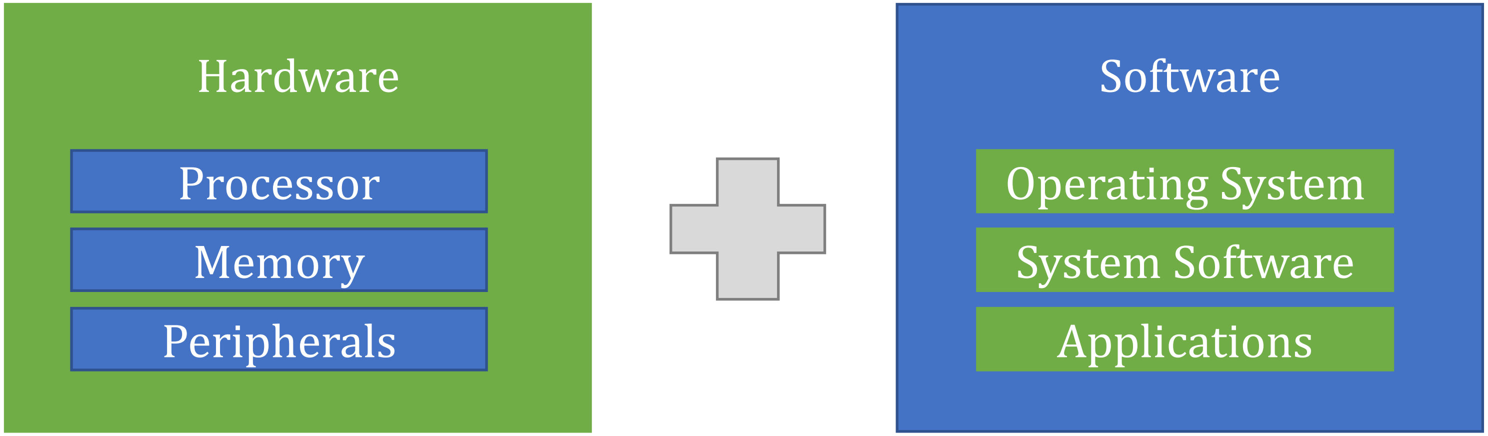
Yeongpil Cho

Hanyang University

Topics

- CPU Architecture Internal
- Program Execution

Structure of Embedded Systems



Hardware

Processor

Memory

Peripherals

Software

Operating System

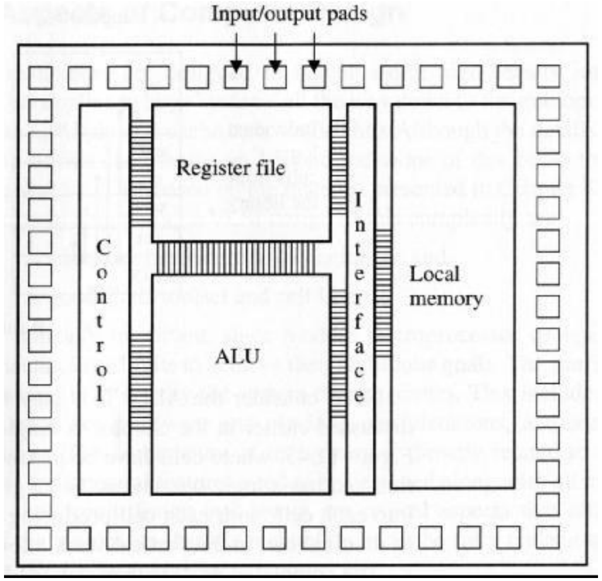
System Software

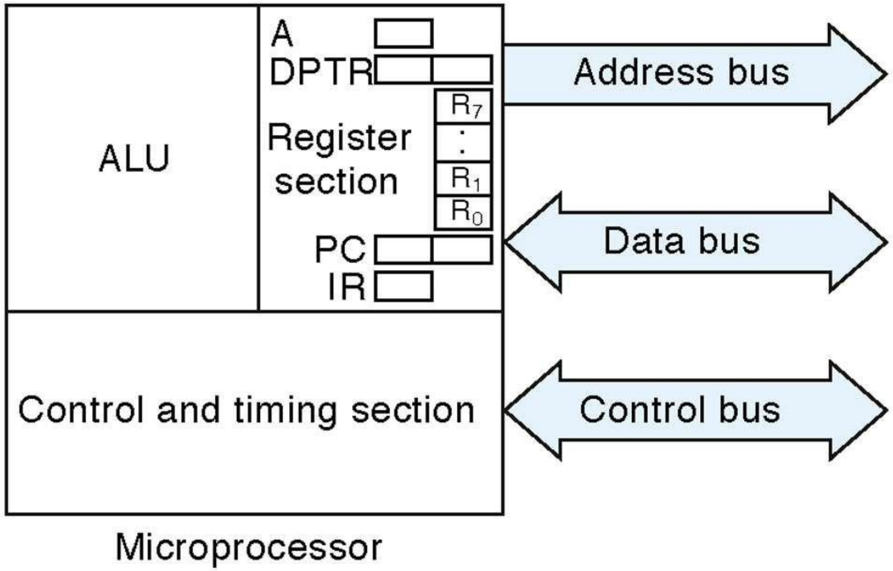
Applications

CPU Architecture Internal

CPU/Processor Architecture

- Control and Timing Section
- Register Section
- ALU (Arithmetic Logic Unit)

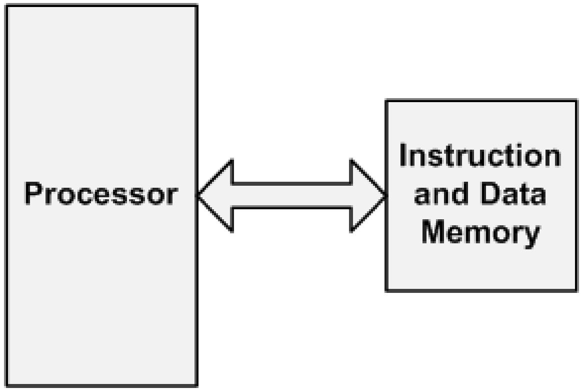




Processor Architectures

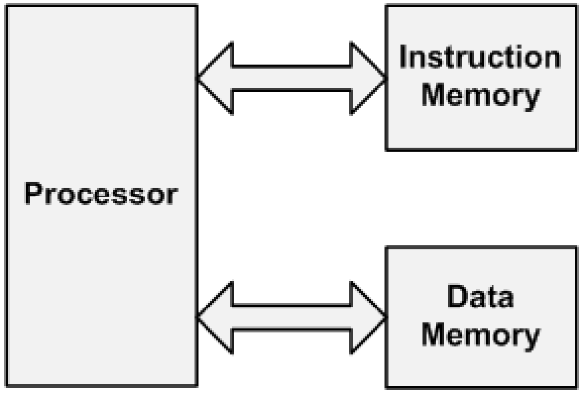
Von-Neumann

Instructions and data are stored in the same memory.



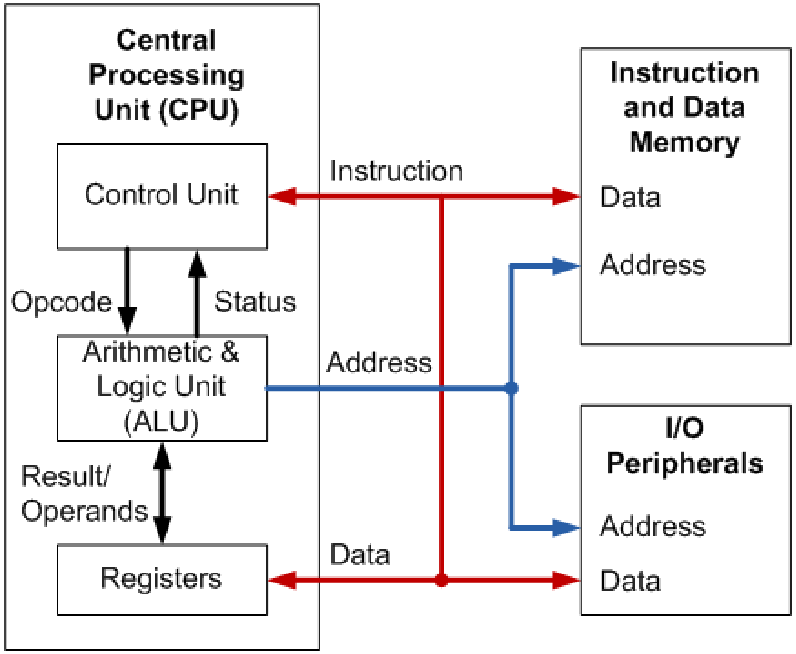
Harvard

Data and instructions are stored into separate memories.

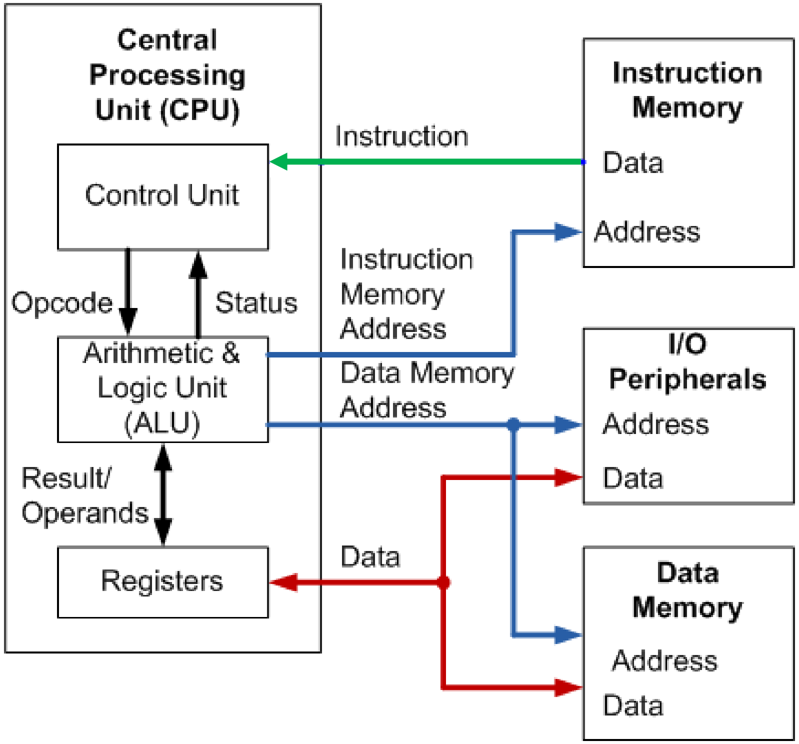


Processor Architectures

Von-Neumann

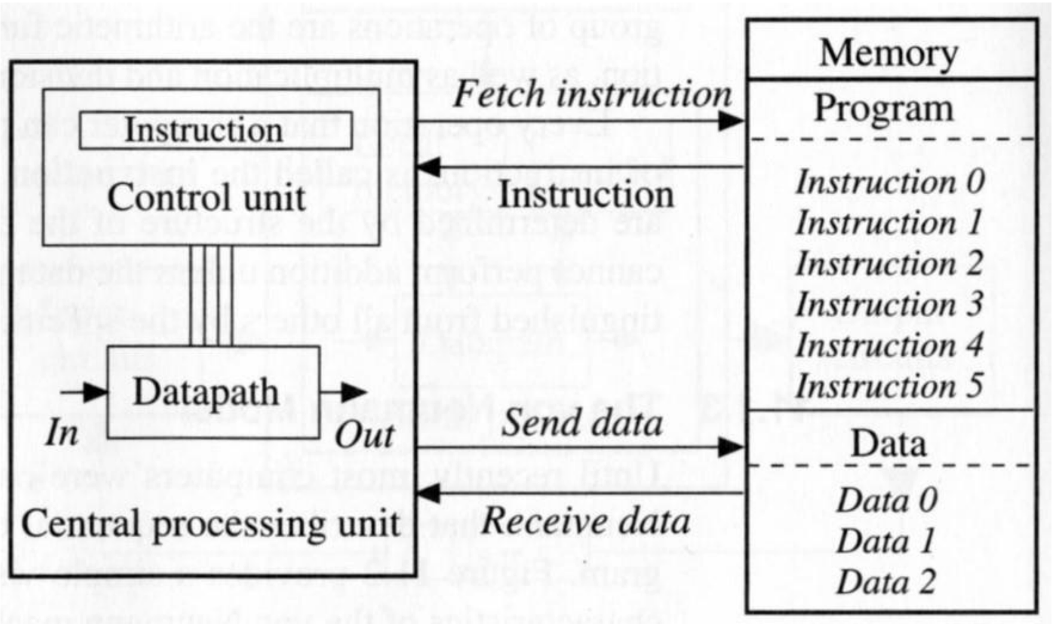


Harvard



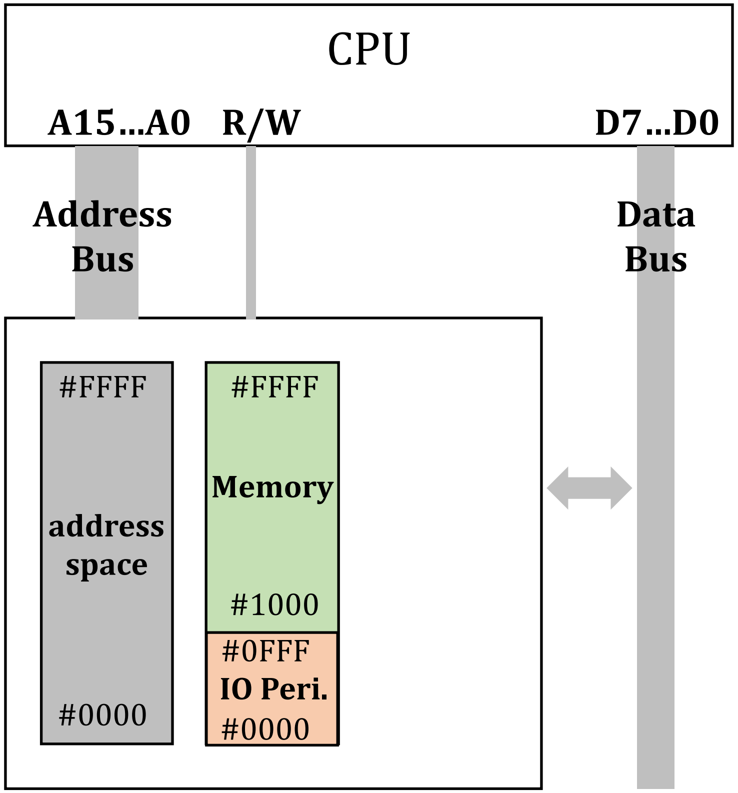
Instruction Execution Process

- **Instruction Fetch:** Reads next instruction into the instruction register (IR). The program counter register (PC) has the instruction address.
- **Instruction Interpretation:** Decodes the op-code, gets the required operands and routes them to ALU.
- **Sequencing:** Determines the address of next instruction and loads it into the PC.
- **Execution:** Generates control signals of ALU for execution.



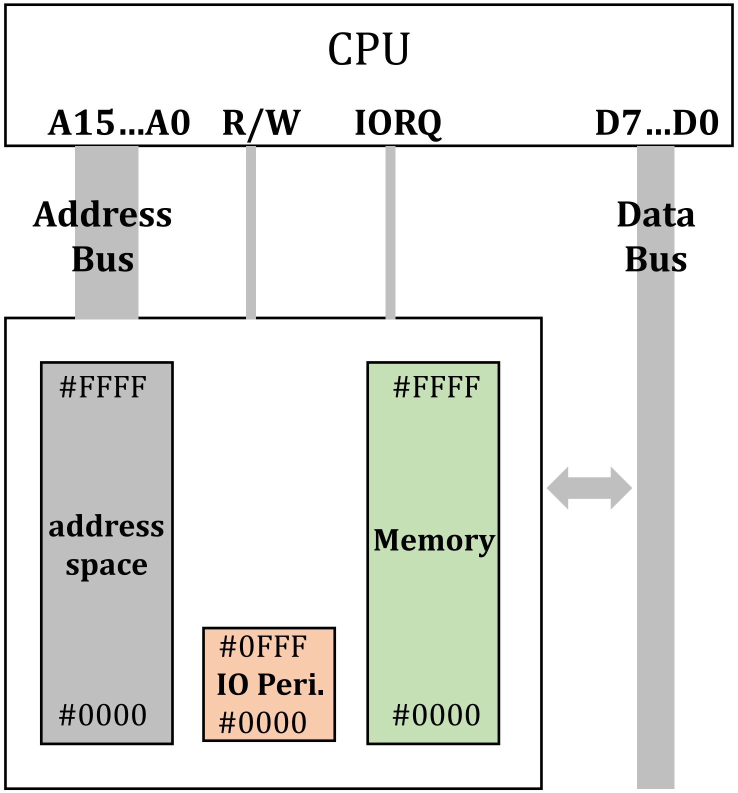
System Organization

- Memory-mapped I/O
  - Memory and I/O have the same address space
    - A: Address, D: Data, R/W: Read/Write



System Organization

- Port-mapped I/O
  - Memory and I/O have different address spaces
    - IORQ: Input/Output Request



Processor Operation Modes

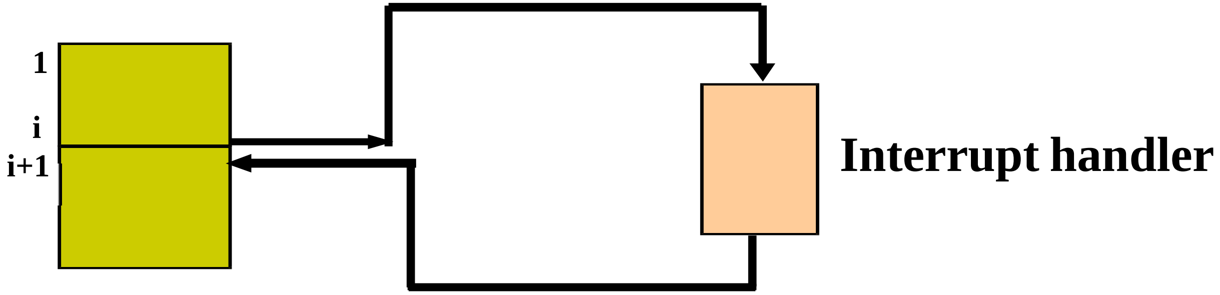
- User mode
    - A user program is running.
    - Certain instructions are not allowed.
    - Memory accesses are restricted
- Supervisor mode
    - The operating system is running.
    - All instructions are allowed.
    - Memory accesses are not restricted
- A single PSW (processor status word) bit sets the above two modes:
    - For instance: PSW-bit =1 for Supervisor mode

Interrupts

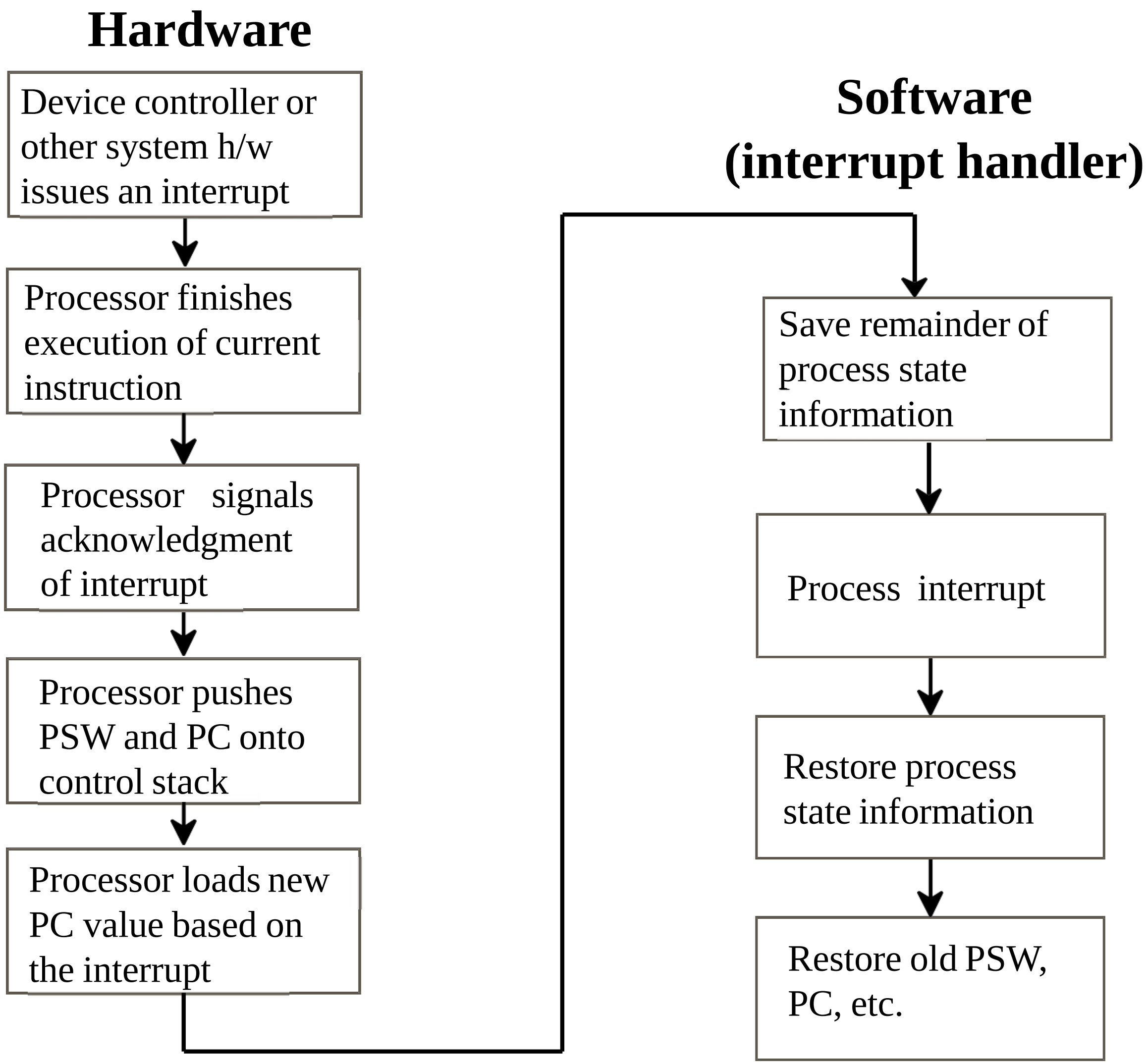
- A computer program has only two ways to determine the conditions that exist in internal and external circuits.
  - One method uses software instructions that jump to subroutine on some flag status.
  - The second method responds to hardware signals called interrupts that force the program to call interrupt-handling subroutines.
  - Interrupts take processor time only when action is required.
  - Processor can respond to an external event much faster by using interrupts.
- Interrupts are often the only way for successful real-time programming.

Instruction Cycle with Interrupts

- Generally CPU checks for interrupts at the end of each instruction and executes the interrupt handler if required.
- **Interrupt Handler** program identifies the nature/source of an interrupt and performs whatever actions are needed.
  - It takes over the control after the interrupt.
  - Control is transferred back to the interrupted program that will resume execution from the point of interruption.
  - Point of interruption can occur anywhere in a program.
  - State of the program is saved.



Interrupt Processing



Hardware

Device controller or other system h/w issues an interrupt

Processor finishes execution of current instruction

Processor signals acknowledgment of interrupt

Processor pushes PSW and PC onto control stack

Processor loads new PC value based on the interrupt

Software (interrupt handler)

Save remainder of process state information

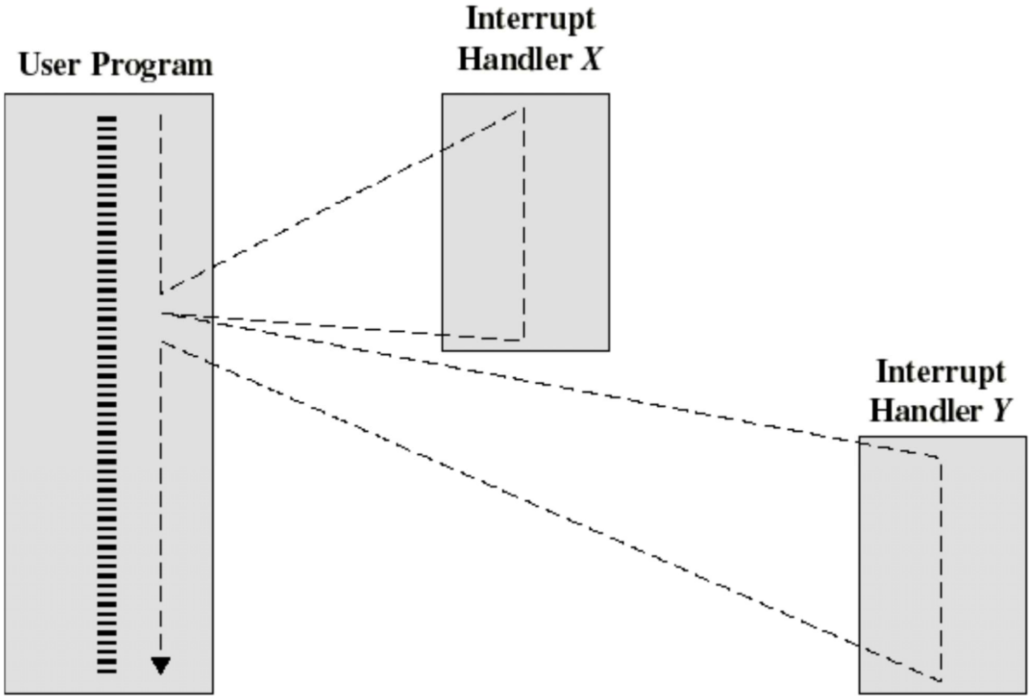
Process interrupt

Restore process state information

Restore old PSW, PC, etc.

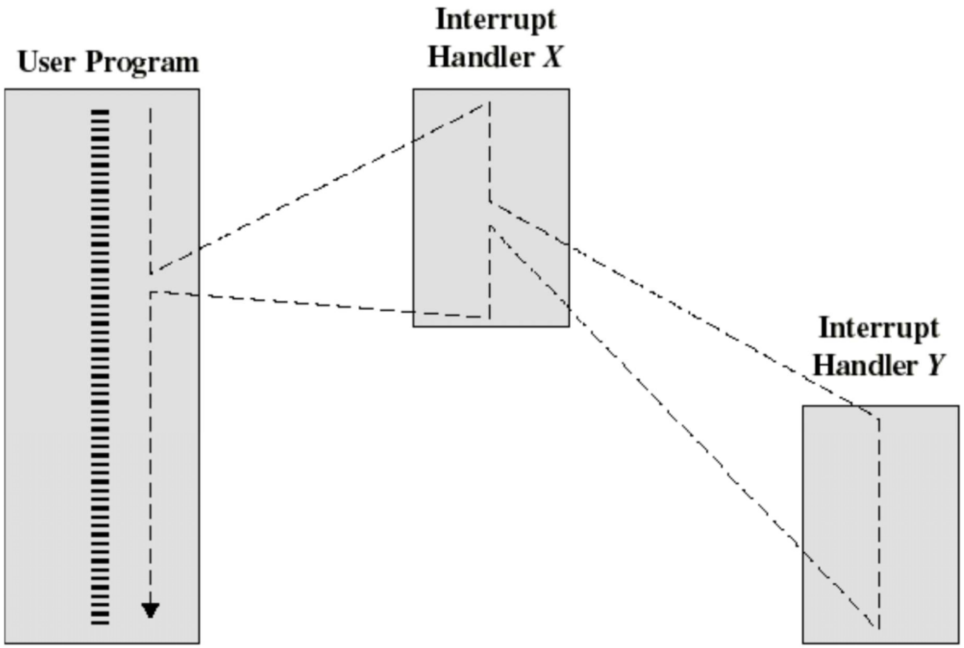
Multiple Interrupts (Sequential Order)

- Disable interrupts to complete the interrupting task at hand.
- Additional interrupts remain pending until interrupts are enabled. Then interrupts are considered in order
- After completing the interrupt handler routine, the processor checks for additional interrupts.



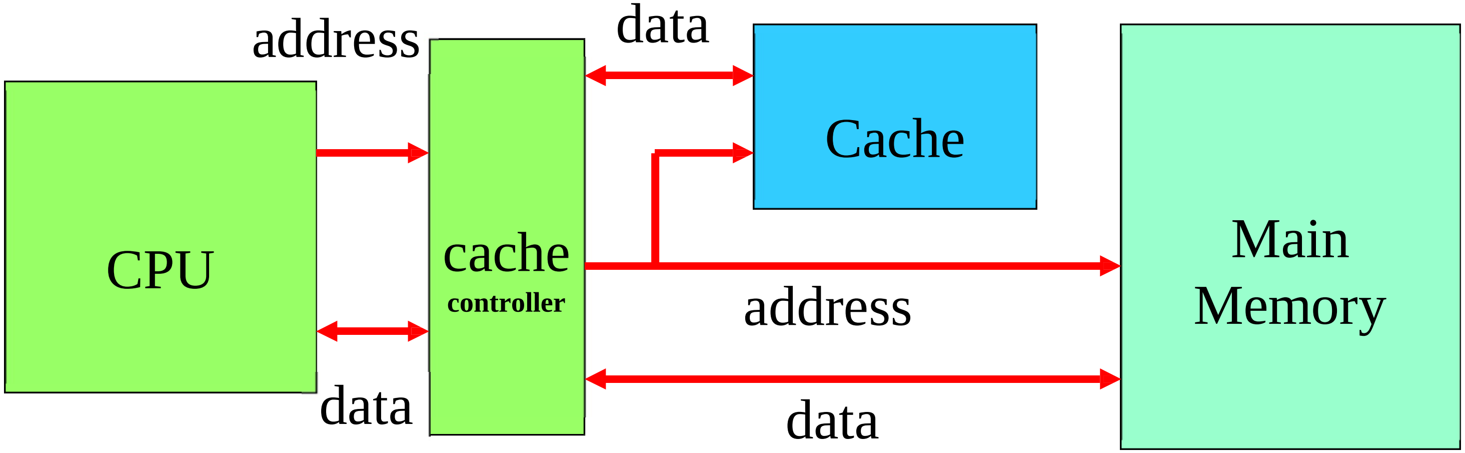
Multiple Interrupts (Nested)

- A higher priority interrupt causes lower-priority interrupts to wait.
- A lower-priority interrupt handler is interrupted.
- For example, when input arrives from a communication line, it needs to be absorbed quickly to make room for additional inputs.



Cache Memory

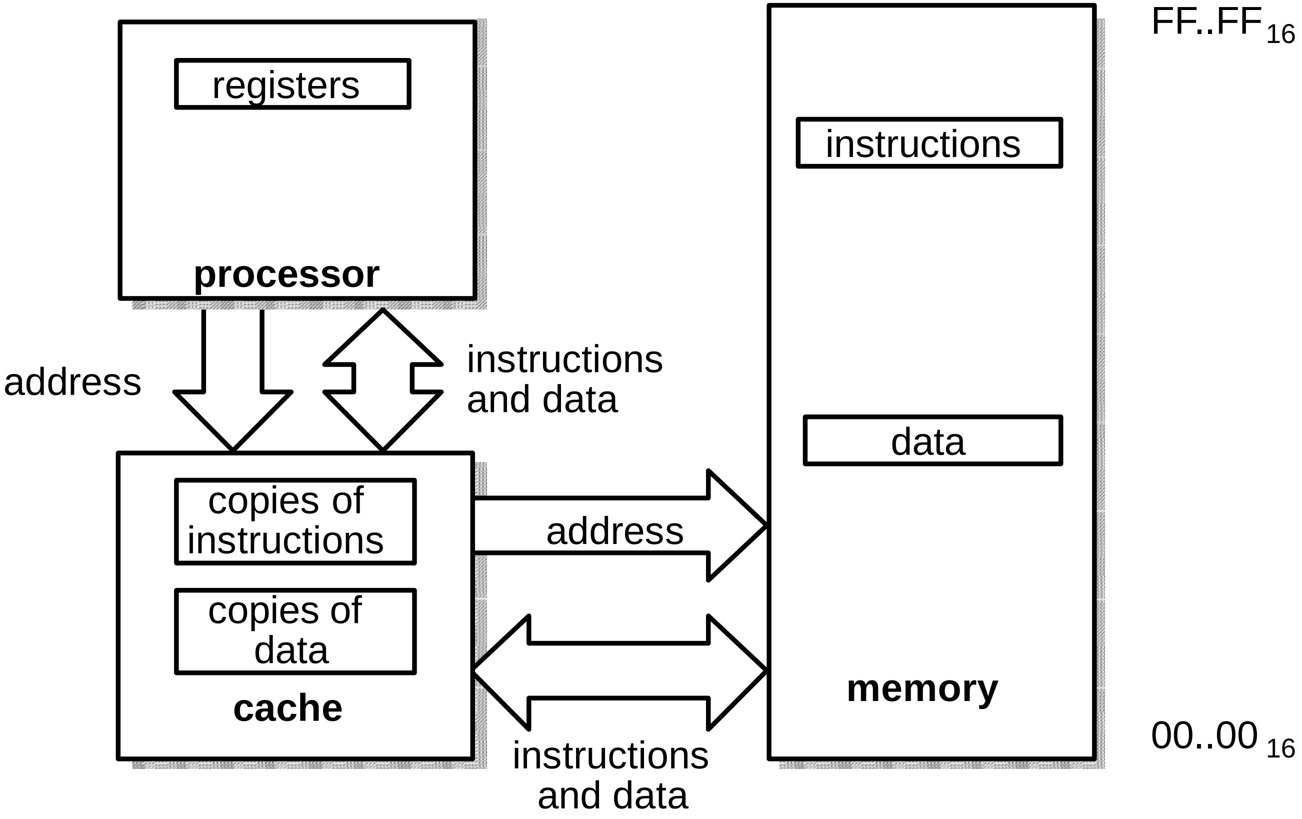
- **Cache:** Expensive but very fast memory directly connected to CPU interacting with slower but much larger main memory.
- Processor first checks if the addresses word is in cache.
- If the word is not found in cache, a block of memory containing the word is moved to the cache.



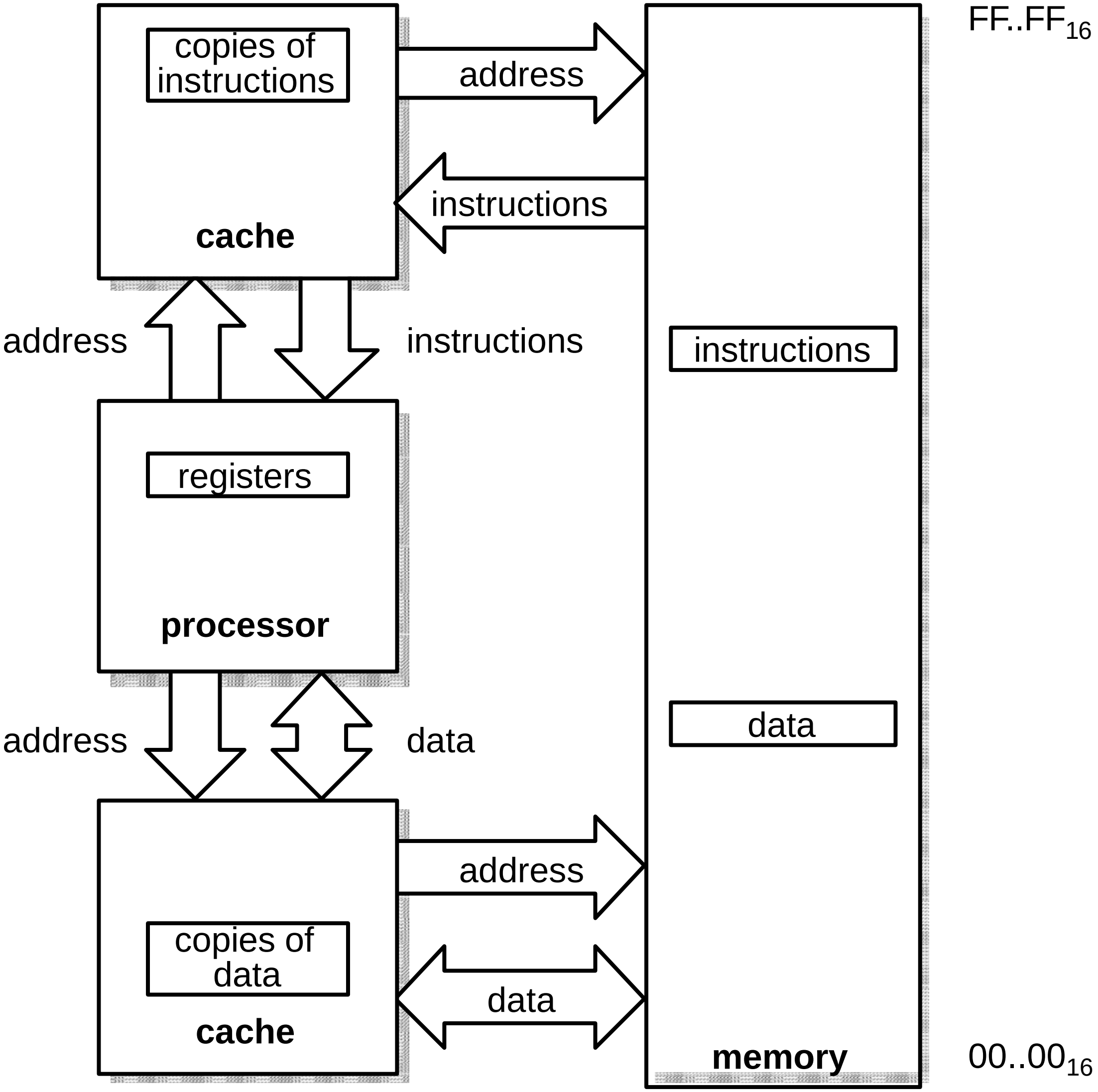
Effectiveness of Caching

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
branch misprediction	5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

A Unified Instruction and Data Cache

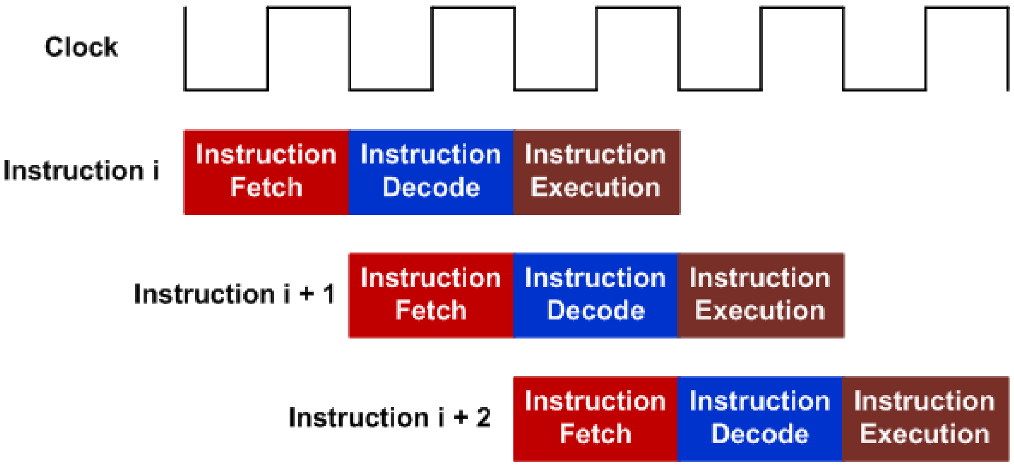


Separate Data and Instruction Caches



CPU Pipelining

- Improve performance by increasing instruction throughput.
- **Pipelining** allows hardware resources to be fully utilized



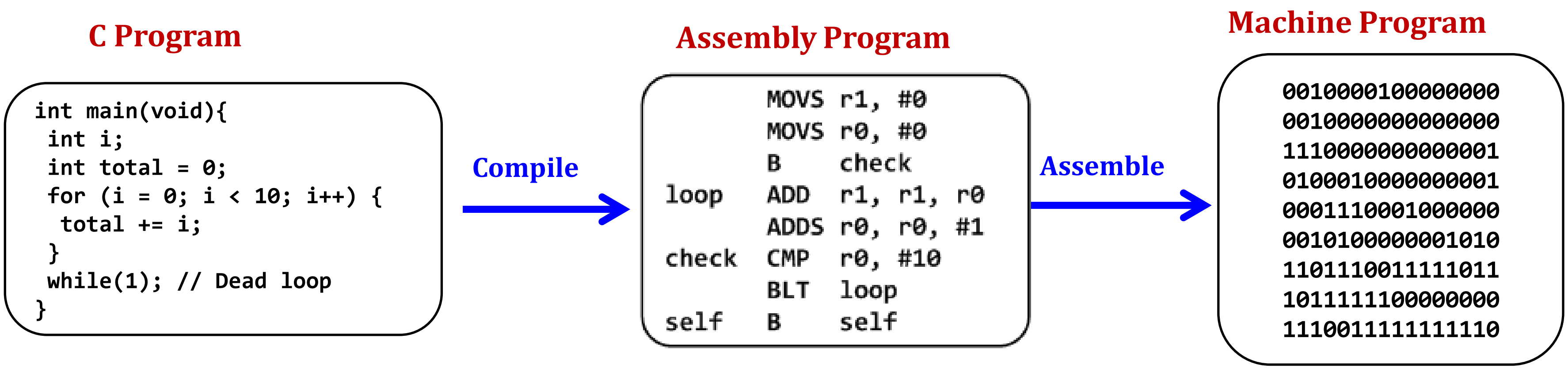
3-stage Pipelining

CPU Pipelining

- What makes pipelining easy?
  - When all instructions are of the same length.
  - Few instruction formats.
  - etc.
- What makes pipelining hard?
  - Structural Hazards:
    - They arise from resource conflicts when the hardware cannot support all possible combinations of instructions in simultaneous overlapped execution.
  - Data Hazards:
    - They arise when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.
  - Control Hazards:
    - They arise from the pipelining of branches and other instructions that change the PC

Program Execution

Levels of Program Code



C Program

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

Assembly Program

```
        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
```

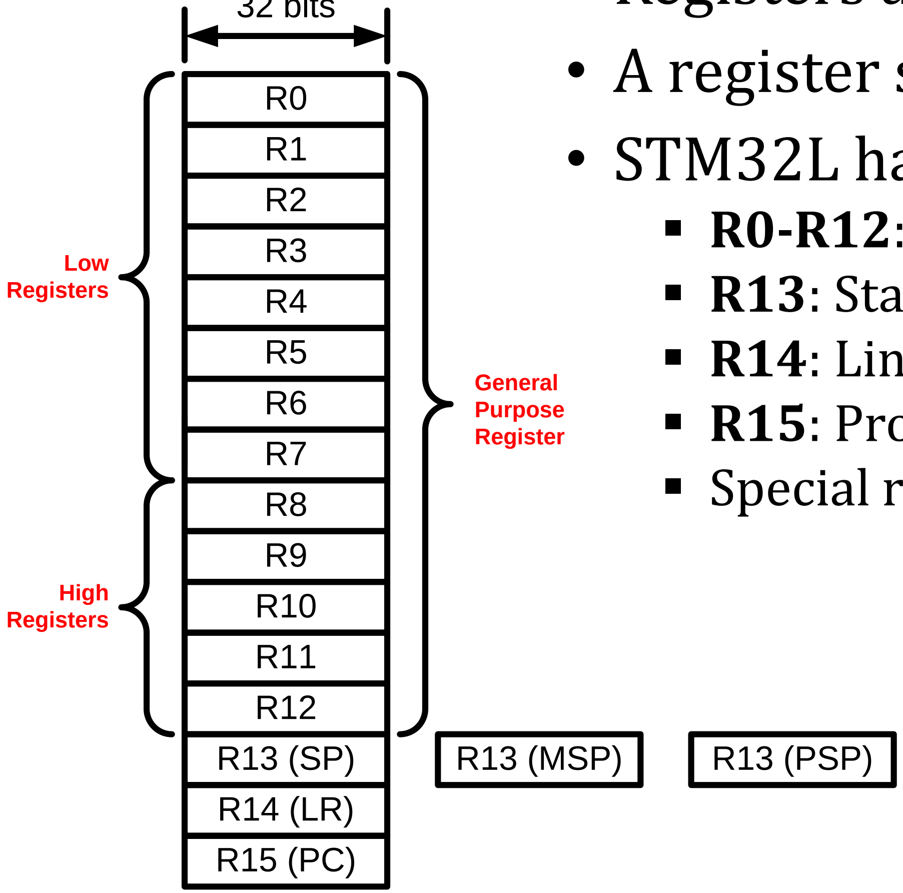
Machine Program

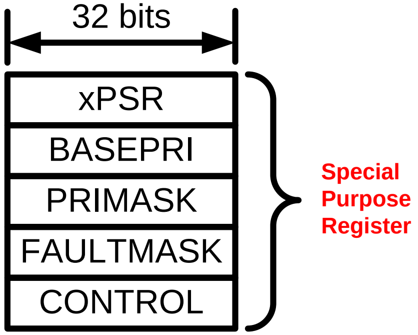
```
0010000100000000
0010000000000000
1110000000000001
0100010000000001
0001110001000000
0010100000001010
1101110011111011
1011111100000000
1110011111111110
```

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data

Processor Registers

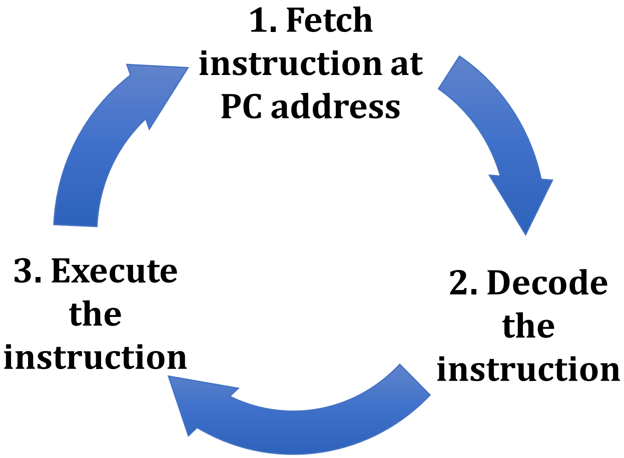
- Fastest way to read and write
- Registers are within the processor chip
- A register stores 32-bit value
- STM32L has
  - **R0-R12**: 13 general-purpose registers
  - **R13**: Stack pointer (Shadow of MSP or PSP)
  - **R14**: Link register (LR)
  - **R15**: Program counter (PC)
  - Special registers (xPSR, BASEPRI, PRIMASK, etc)





Program Execution

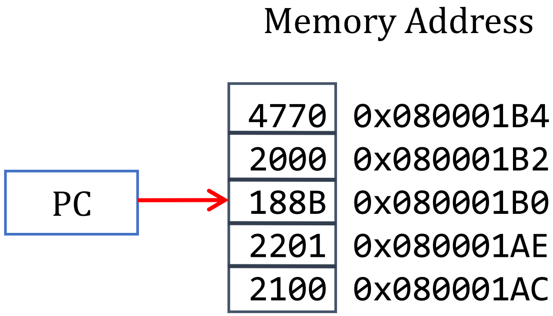
- Program Counter (PC) is a register that holds the memory address of the next instruction to be fetched from the memory.



1. Fetch instruction at PC address

2. Decode the instruction

3. Execute the instruction

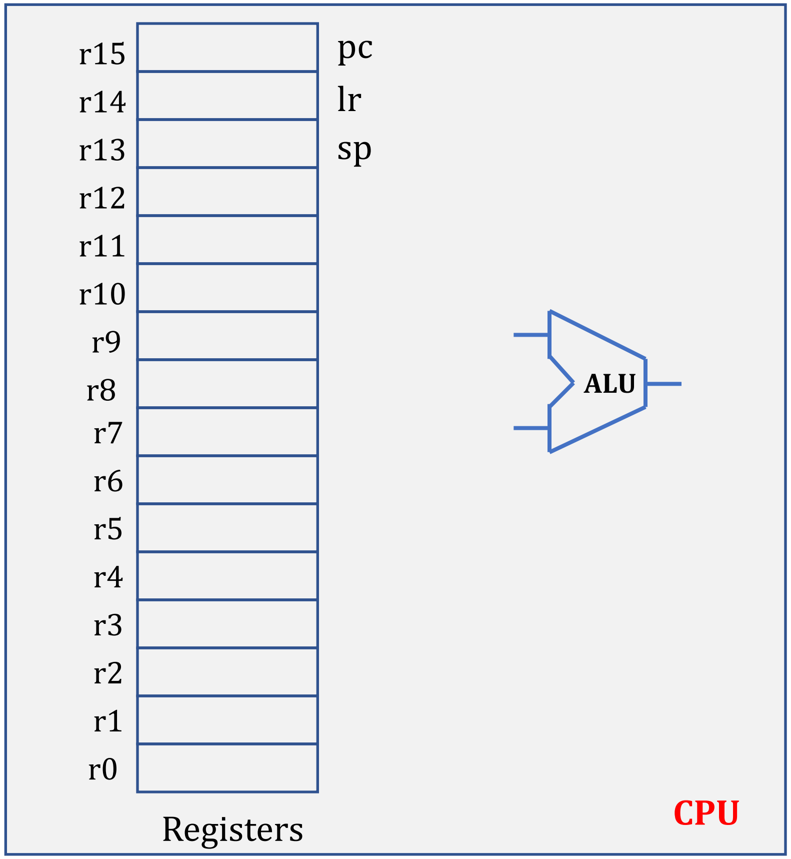


PC

	Memory Address
4770	0x080001B4
2000	0x080001B2
188B	0x080001B0
2201	0x080001AE
2100	0x080001AC

PC = 188B
Instruction = 0x080001B0

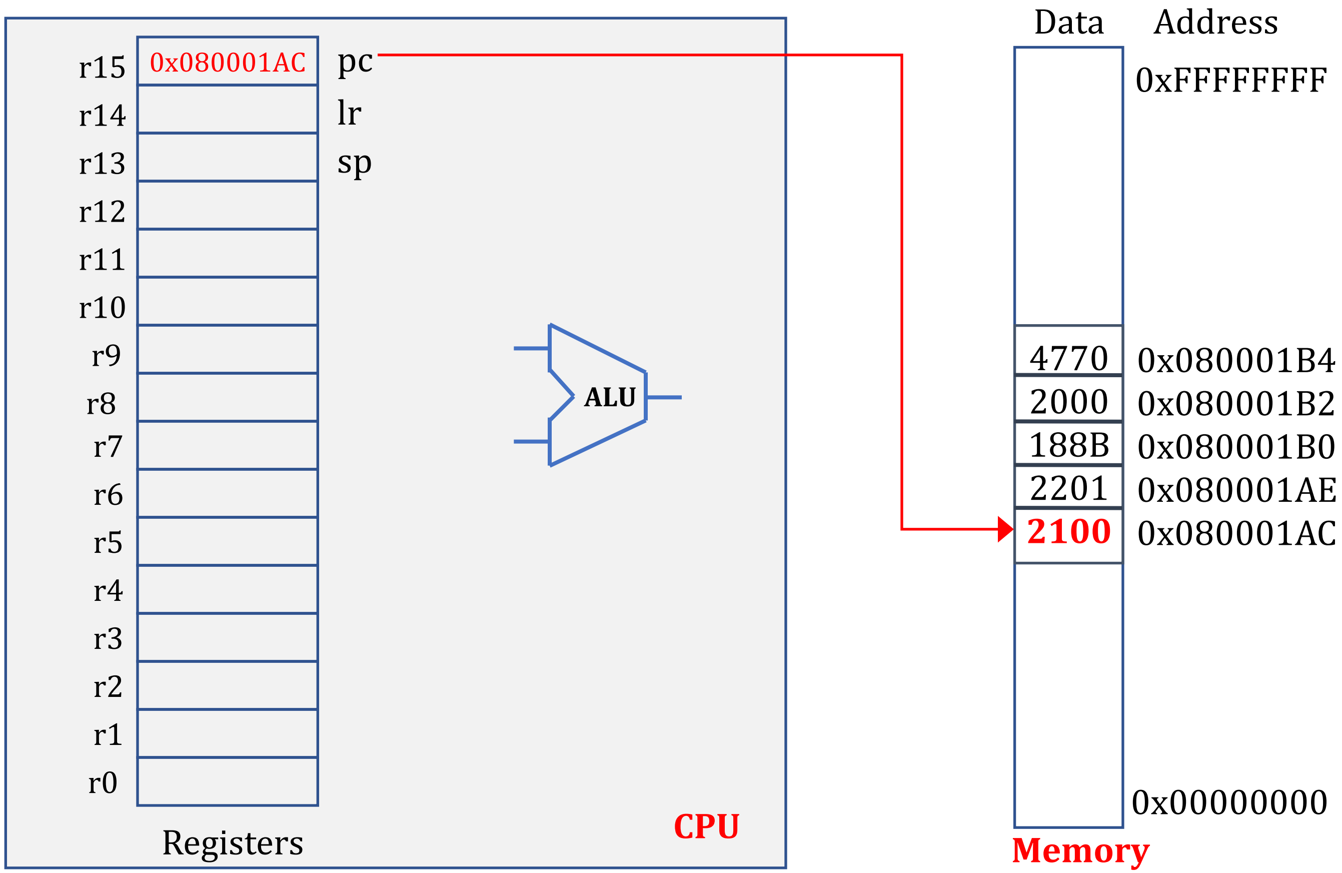
Machine codes are stored in memory



Data	Address
	0xFFFFFFFF
4770	0x080001B4
2000	0x080001B2
188B	0x080001B0
2201	0x080001AE
2100	0x080001AC
	0x00000000

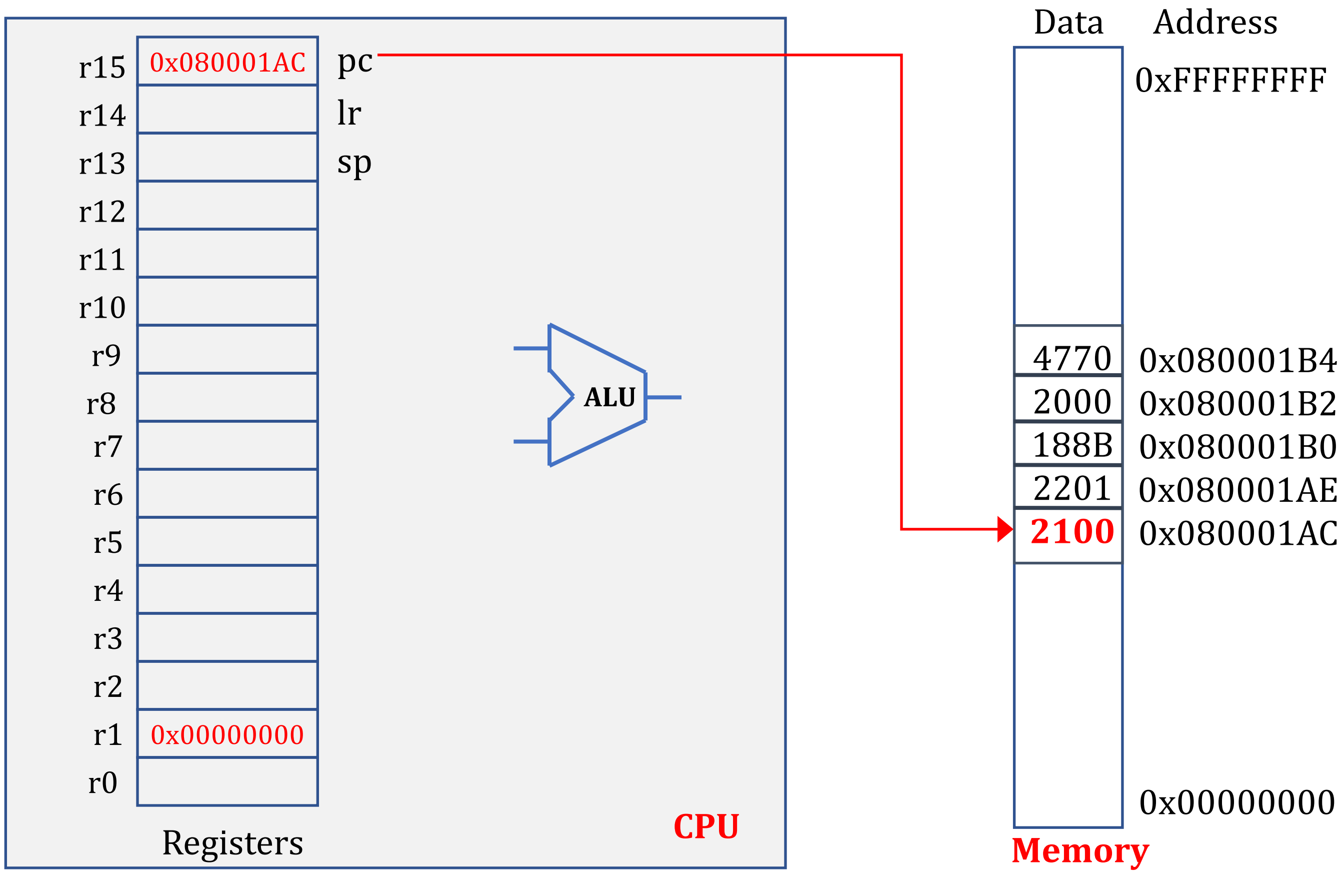
Memory

Fetch Instruction: pc = 0x08001AC
Decode Instruction: 2100 = MOVS r1, #0x00



Data	Address
	0xFFFFFFFF
4770	0x080001B4
2000	0x080001B2
188B	0x080001B0
2201	0x080001AE
2100	0x080001AC
	0x00000000

Execute Instruction:
MOVS r1, #0x00



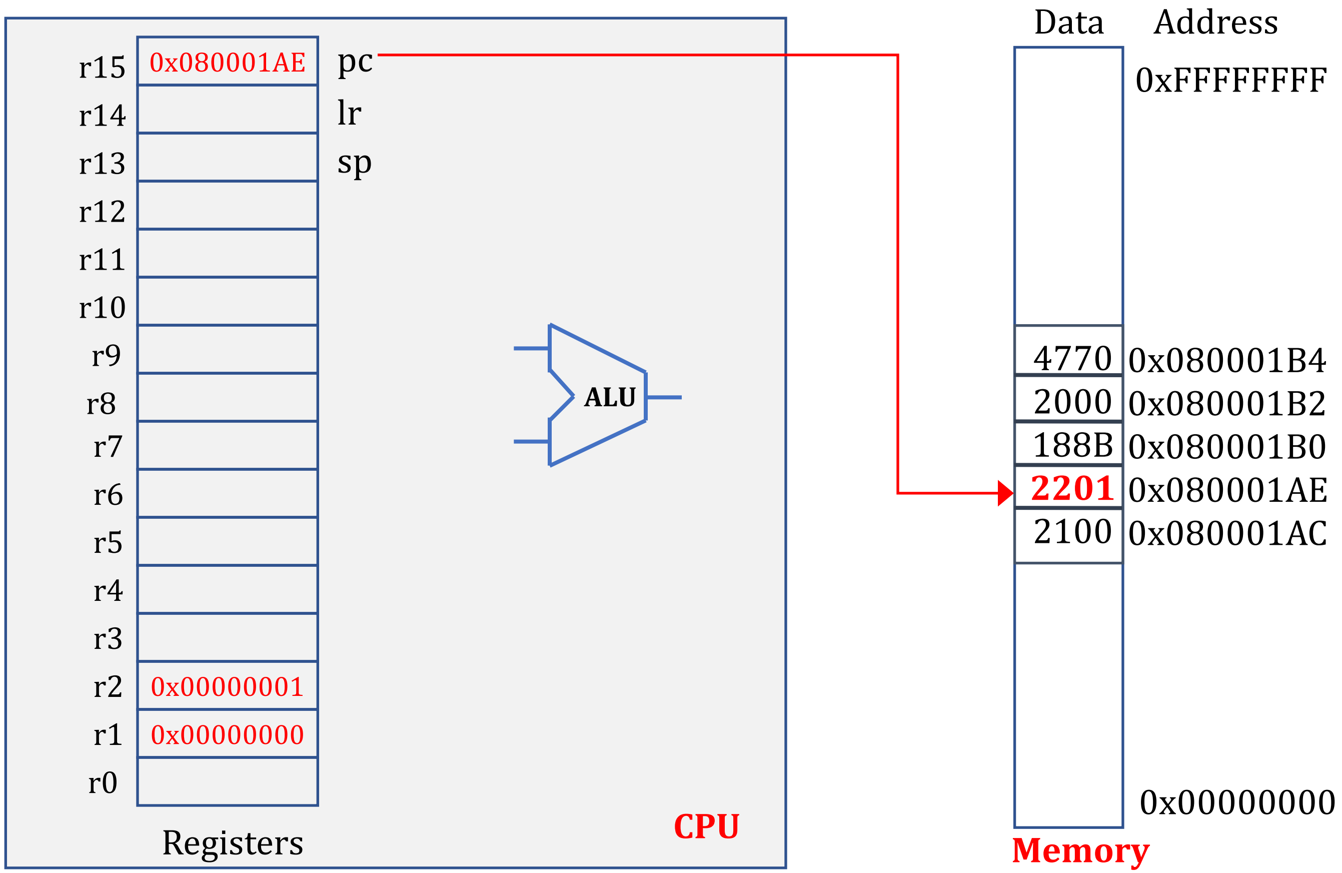
CPU

Registers		
r15	0x080001AC	pc
r14		lr
r13		sp
r12		
r11		
r10		
r9		
r8		
r7		
r6		
r5		
r4		
r3		
r2		
r1	0x00000000	
r0		

Memory

Data	Address
	0xFFFFFFFF
4770	0x080001B4
2000	0x080001B2
188B	0x080001B0
2201	0x080001AE
2100	0x080001AC
	0x00000000

Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2201 = MOVS r2, #0x01

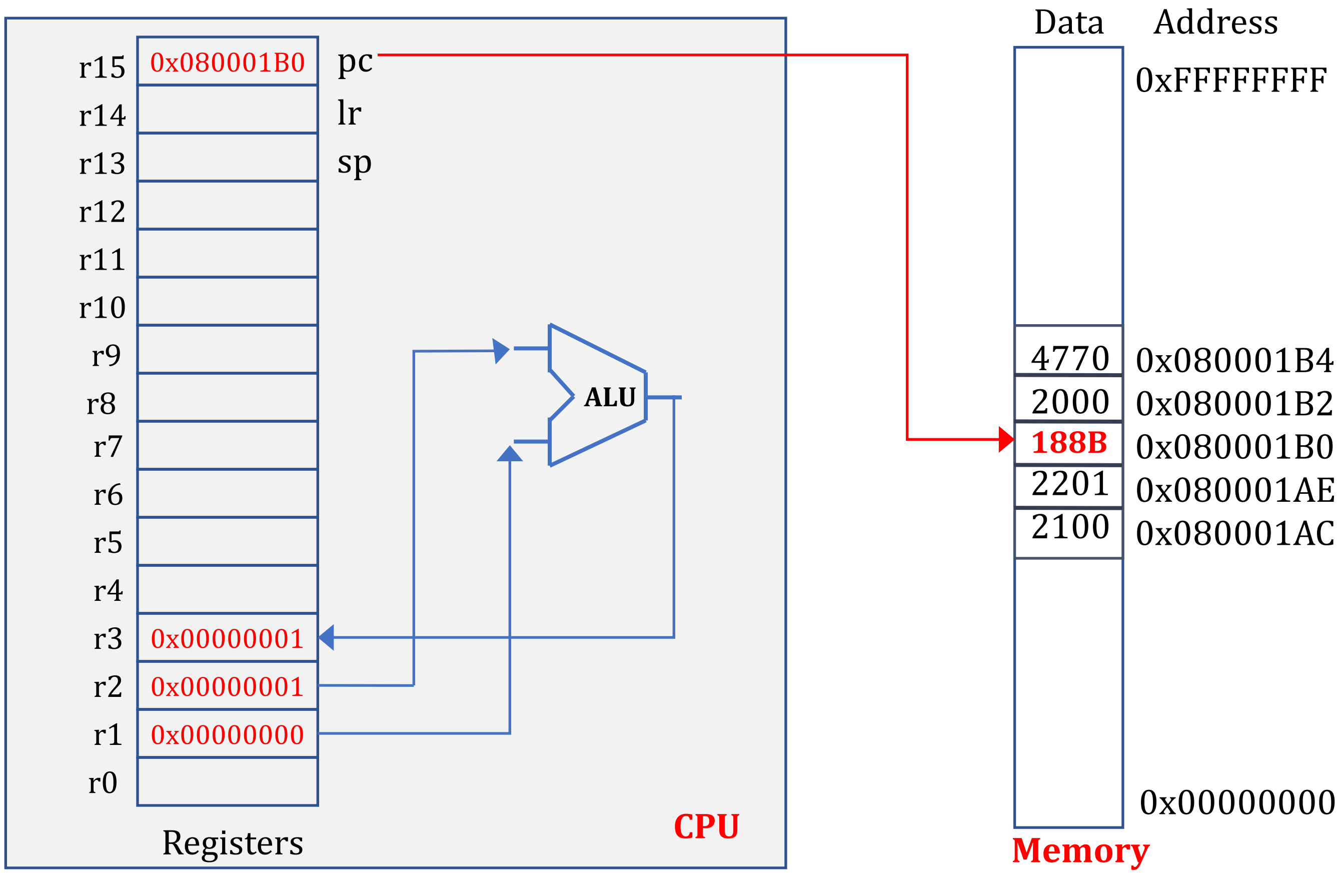


	Registers	
r15	0x080001AE	pc
r14		lr
r13		sp
r12		
r11		
r10		
r9		
r8		
r7		
r6		
r5		
r4		
r3		
r2	0x00000001	
r1	0x00000000	
r0		

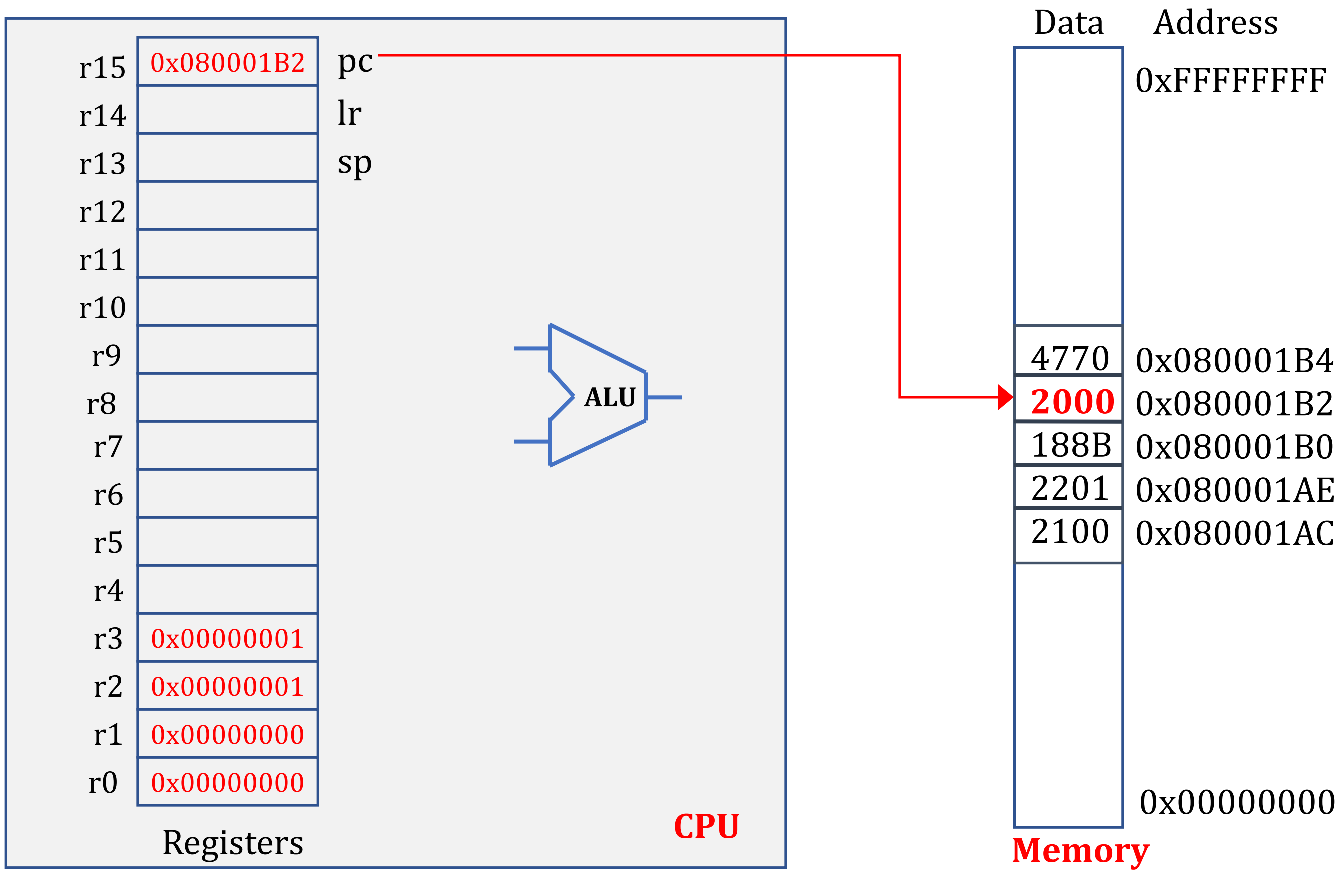
Memory

Data	Address
	0xFFFFFFFF
4770	0x080001B4
2000	0x080001B2
188B	0x080001B0
2201	0x080001AE
2100	0x080001AC
	0x00000000

Fetch Next Instruction: pc = pc + 2
Decode & Execute: 188B = ADDS r3, r1, r2



Fetch Next Instruction: pc = pc + 2
Decode & Execute: 2000 = MOVS r0, #0x00

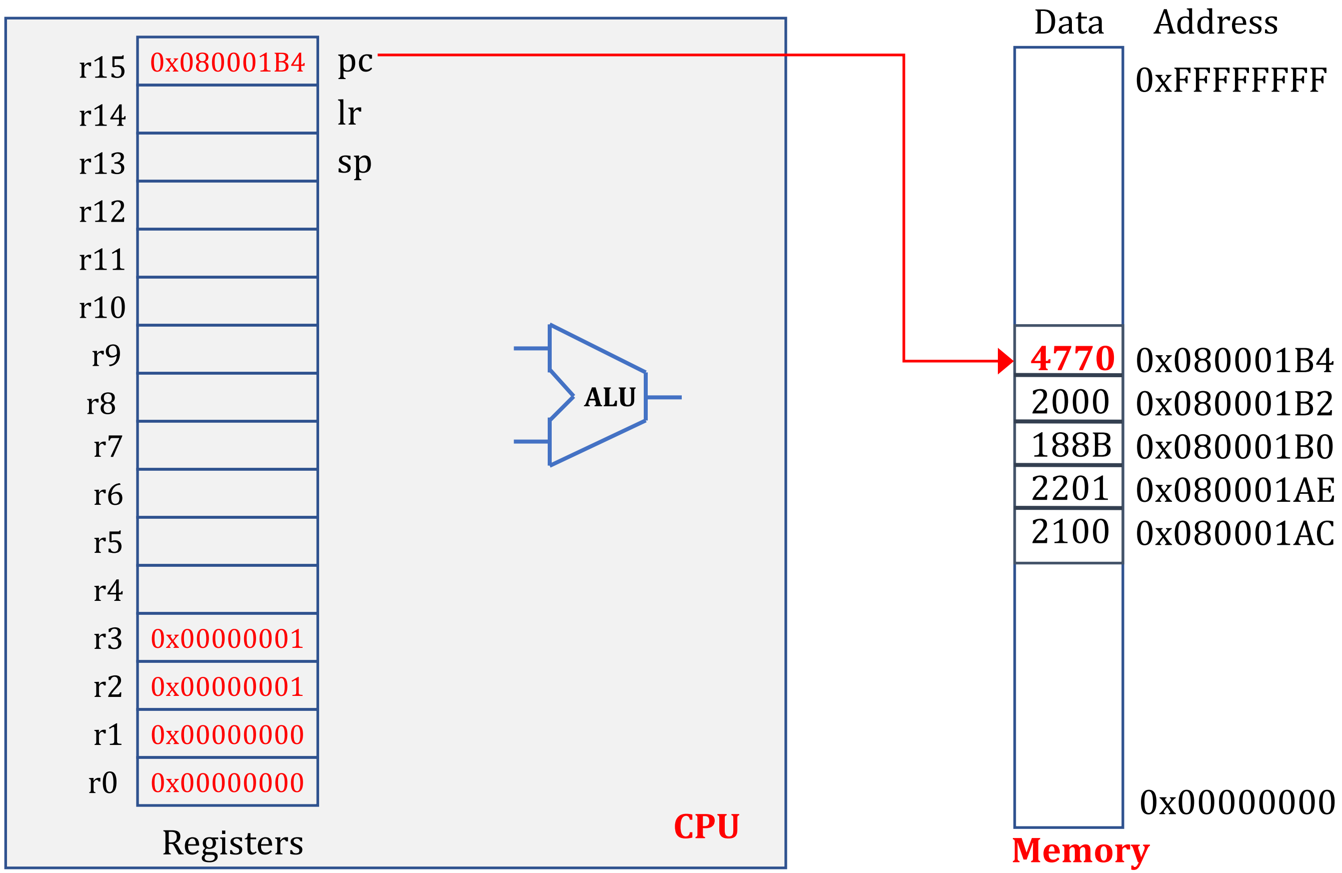


r15	0x080001B2	pc
r14		lr
r13		sp
r12		
r11		
r10		
r9		
r8		
r7		
r6		
r5		
r4		
r3	0x00000001	
r2	0x00000001	
r1	0x00000000	
r0	0x00000000	

Data	Address
	0xFFFFFFFF
4770	0x080001B4
2000	0x080001B2
188B	0x080001B0
2201	0x080001AE
2100	0x080001AC
	0x00000000

Fetch Next Instruction: pc = pc + 2
Decode & Decode: 4770 = BX lr

(BX: branch and exchange)

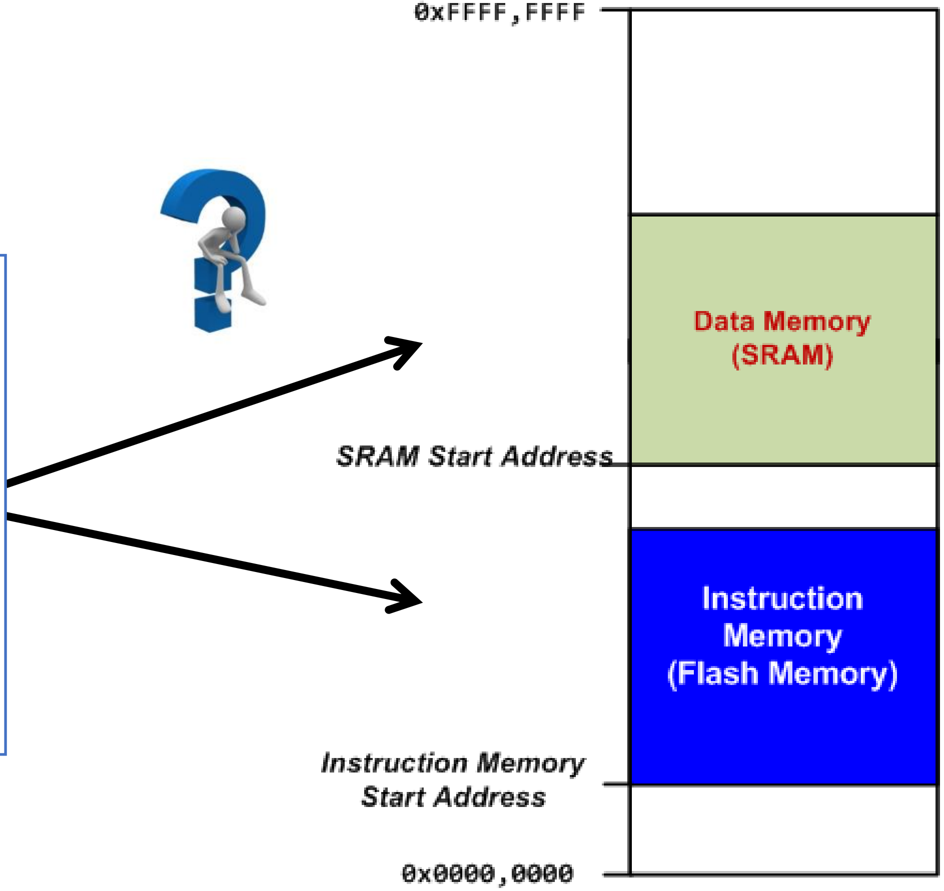


	Registers	
r15	0x080001B4	pc
r14		lr
r13		sp
r12		
r11		
r10		
r9		
r8		
r7		
r6		
r5		
r4		
r3	0x00000001	
r2	0x00000001	
r1	0x00000000	
r0	0x00000000	

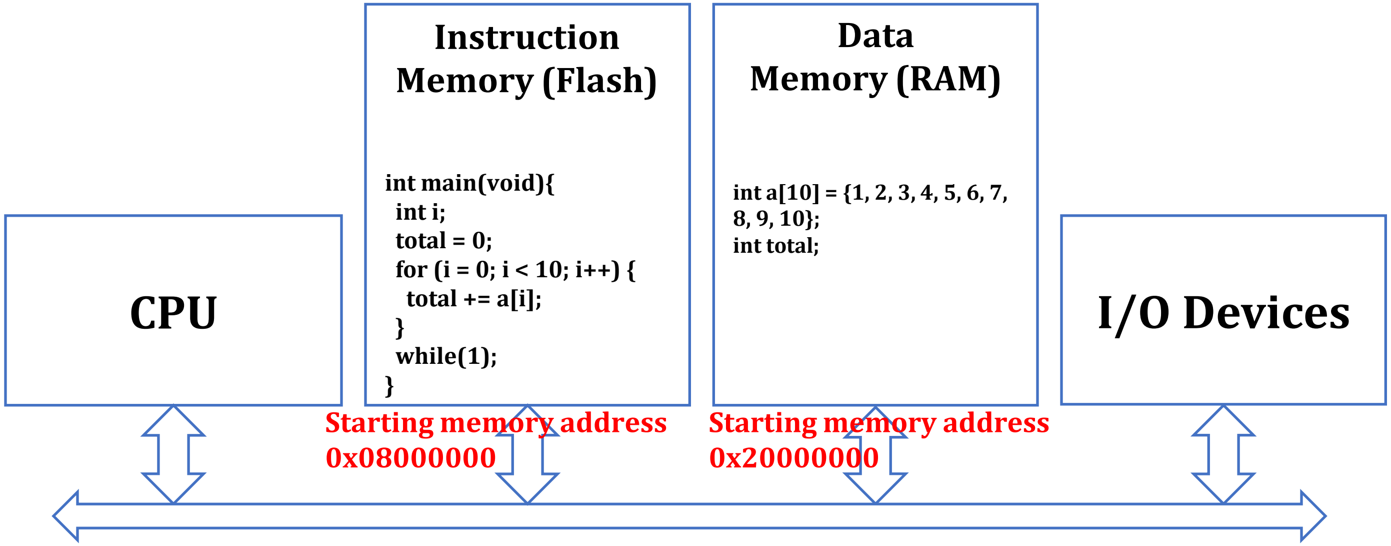
Data	Address
	0xFFFFFFFF
4770	0x080001B4
2000	0x080001B2
188B	0x080001B0
2201	0x080001AE
2100	0x080001AC
	0x00000000

Loading Code and Data into Memory

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total;

int main(void){
        int i;
        total = 0;
        for (i = 0; i < 10; i++) {
                total += a[i];
        }
        while(1);
}
```



Loading Code and Data into Memory



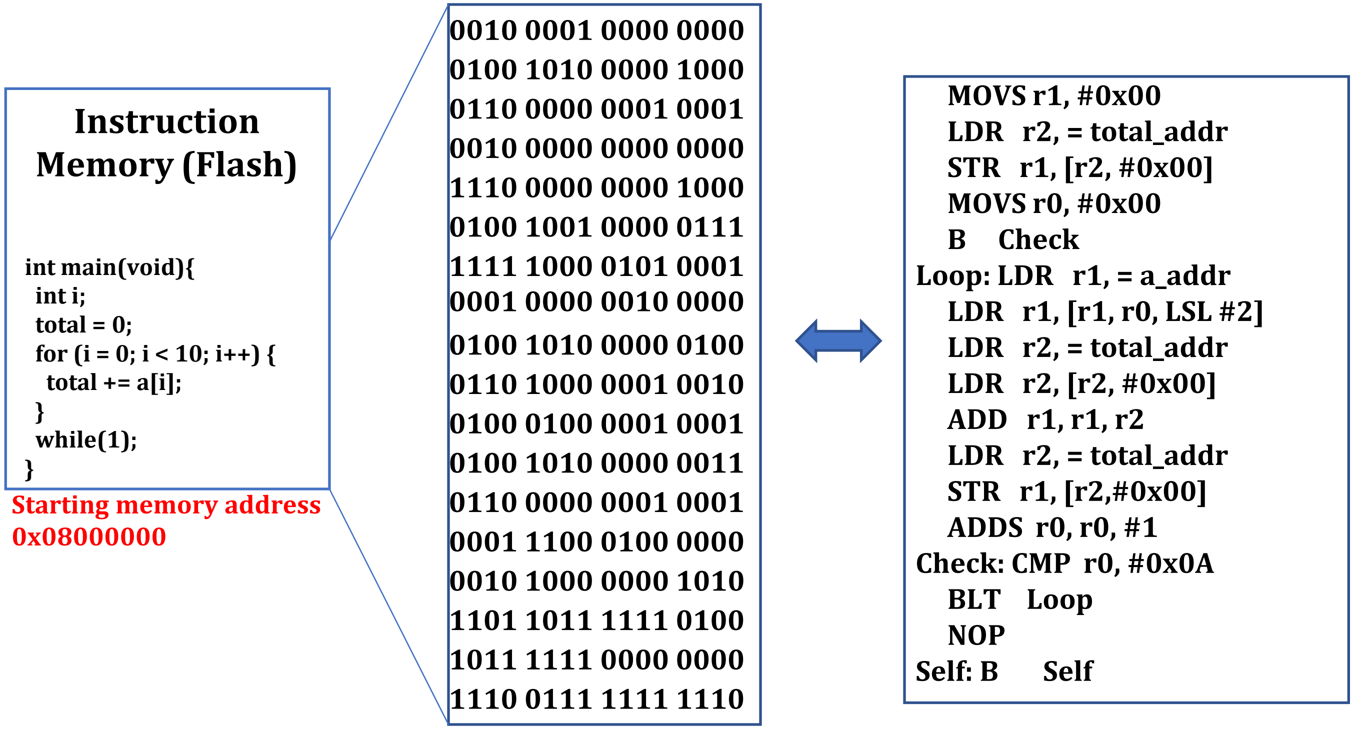
```
int main(void){
 int i;
 total = 0;
 for (i = 0; i < 10; i++) {
  total += a[i];
 }
 while(1);
}
```

Starting memory address 0x08000000

```
int a[10] = {1, 2, 3, 4, 5, 6, 7,
8, 9, 10};
int total;
```

Starting memory address 0x20000000

Loading Code and Data into Memory



Instruction Memory (Flash)

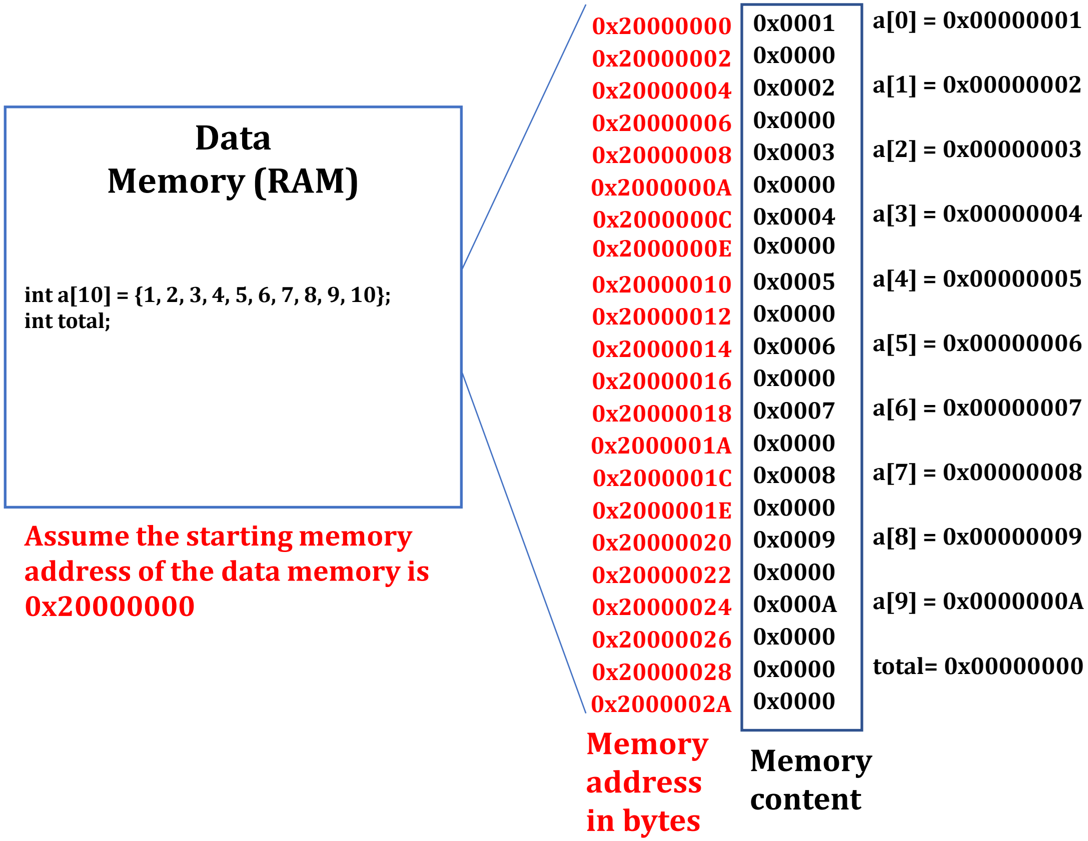
```
int main(void){
 int i;
 total = 0;
 for (i = 0; i < 10; i++) {
  total += a[i];
 }
 while(1);
}
```

Starting memory address 0x08000000

```
0010 0001 0000 0000
0100 1010 0000 1000
0110 0000 0001 0001
0010 0000 0000 0000
1110 0000 0000 1000
0100 1001 0000 0111
1111 1000 0101 0001
0001 0000 0010 0000
0100 1010 0000 0100
0110 1000 0001 0010
0100 0100 0001 0001
0100 1010 0000 0011
0110 0000 0001 0001
0001 1100 0100 0000
0010 1000 0000 1010
1101 1011 1111 0100
1011 1111 0000 0000
1110 0111 1111 1110
```

```
      MOVS r1, #0x00
      LDR  r2, = total_addr
      STR  r1, [r2, #0x00]
      MOVS r0, #0x00
      B    Check
Loop: LDR  r1, = a_addr
      LDR  r1, [r1, r0, LSL #2]
      LDR  r2, = total_addr
      LDR  r2, [r2, #0x00]
      ADD  r1, r1, r2
      LDR  r2, = total_addr
      STR  r1, [r2,#0x00]
      ADDS r0, r0, #1
Check: CMP r0, #0x0A
      BLT  Loop
      NOP
Self: B    Self
```

Loading Code and Data into Memory



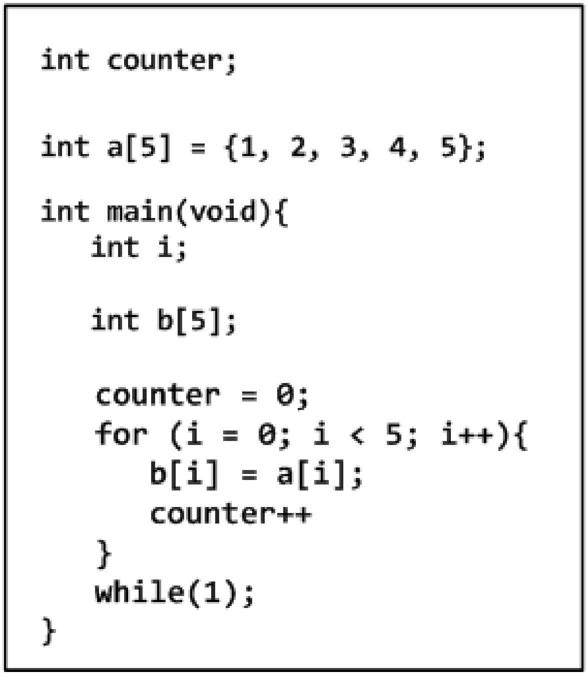
Data Memory (RAM)

```
int a[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int total;
```

Assume the starting memory address of the data memory is 0x20000000

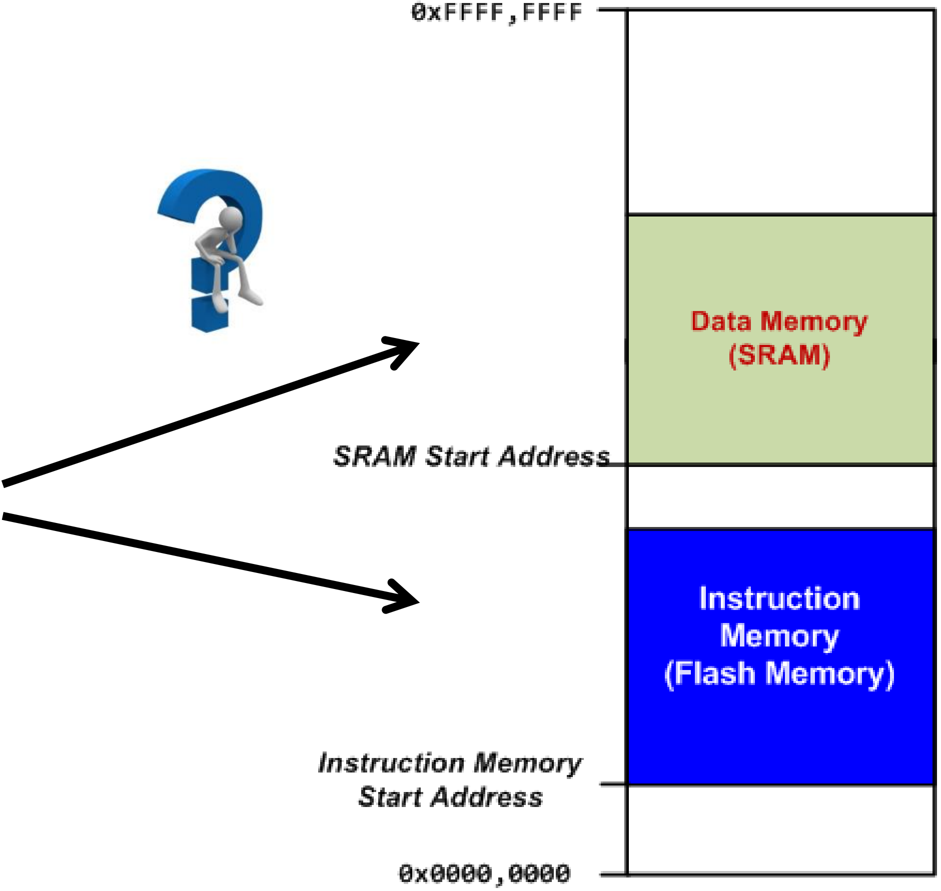
Memory address in bytes	**Memory content**	
0x20000000	0x0001	a[0] = 0x00000001
0x20000002	0x0000	
0x20000004	0x0002	a[1] = 0x00000002
0x20000006	0x0000	
0x20000008	0x0003	a[2] = 0x00000003
0x2000000A	0x0000	
0x2000000C	0x0004	a[3] = 0x00000004
0x2000000E	0x0000	
0x20000010	0x0005	a[4] = 0x00000005
0x20000012	0x0000	
0x20000014	0x0006	a[5] = 0x00000006
0x20000016	0x0000	
0x20000018	0x0007	a[6] = 0x00000007
0x2000001A	0x0000	
0x2000001C	0x0008	a[7] = 0x00000008
0x2000001E	0x0000	
0x20000020	0x0009	a[8] = 0x00000009
0x20000022	0x0000	
0x20000024	0x000A	a[9] = 0x0000000A
0x20000026	0x0000	
0x20000028	0x0000	total= 0x00000000
0x2000002A	0x0000	

Loading Code and Data into Memory

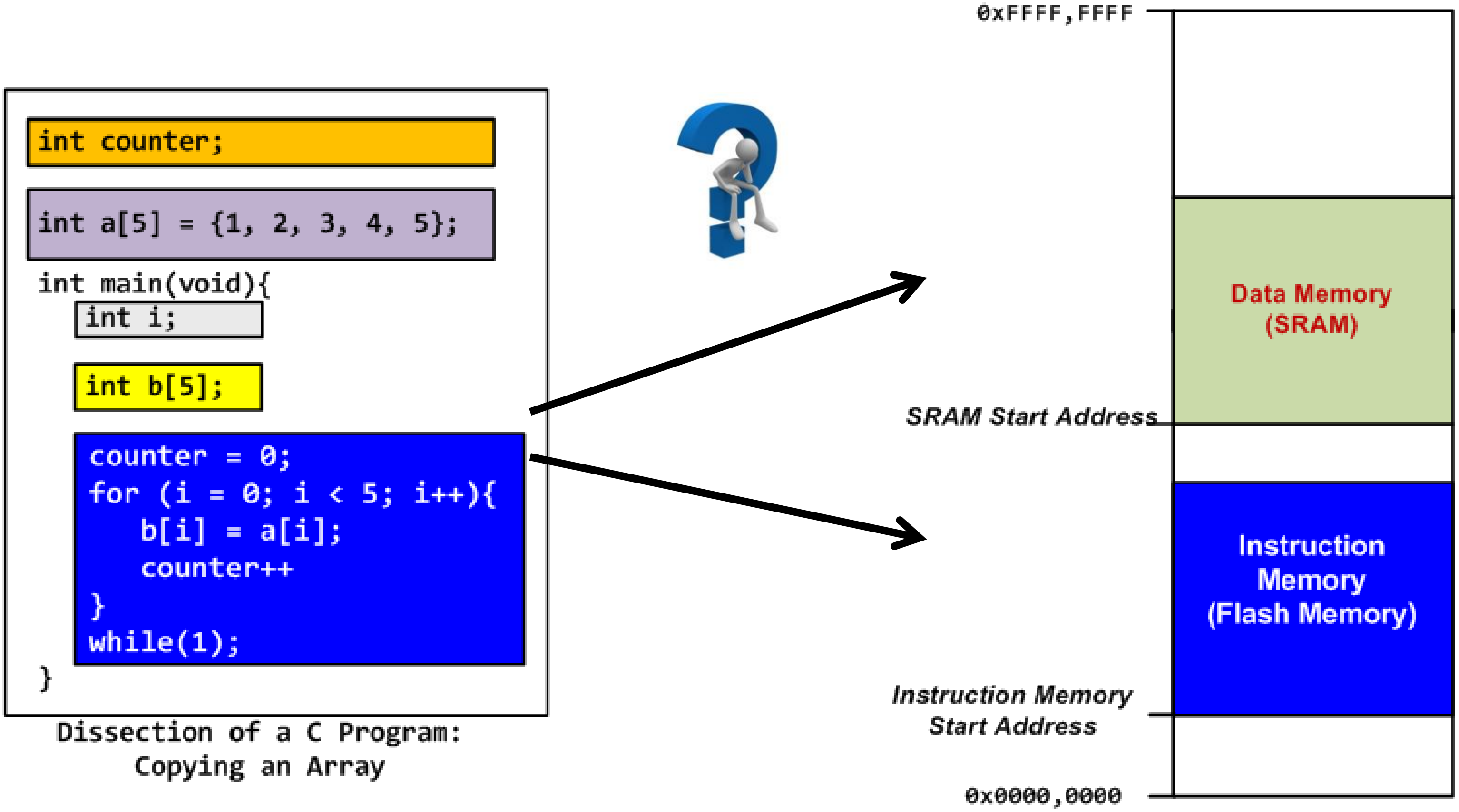


```
int counter;

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);
}
```

C Program:
Copying an Array



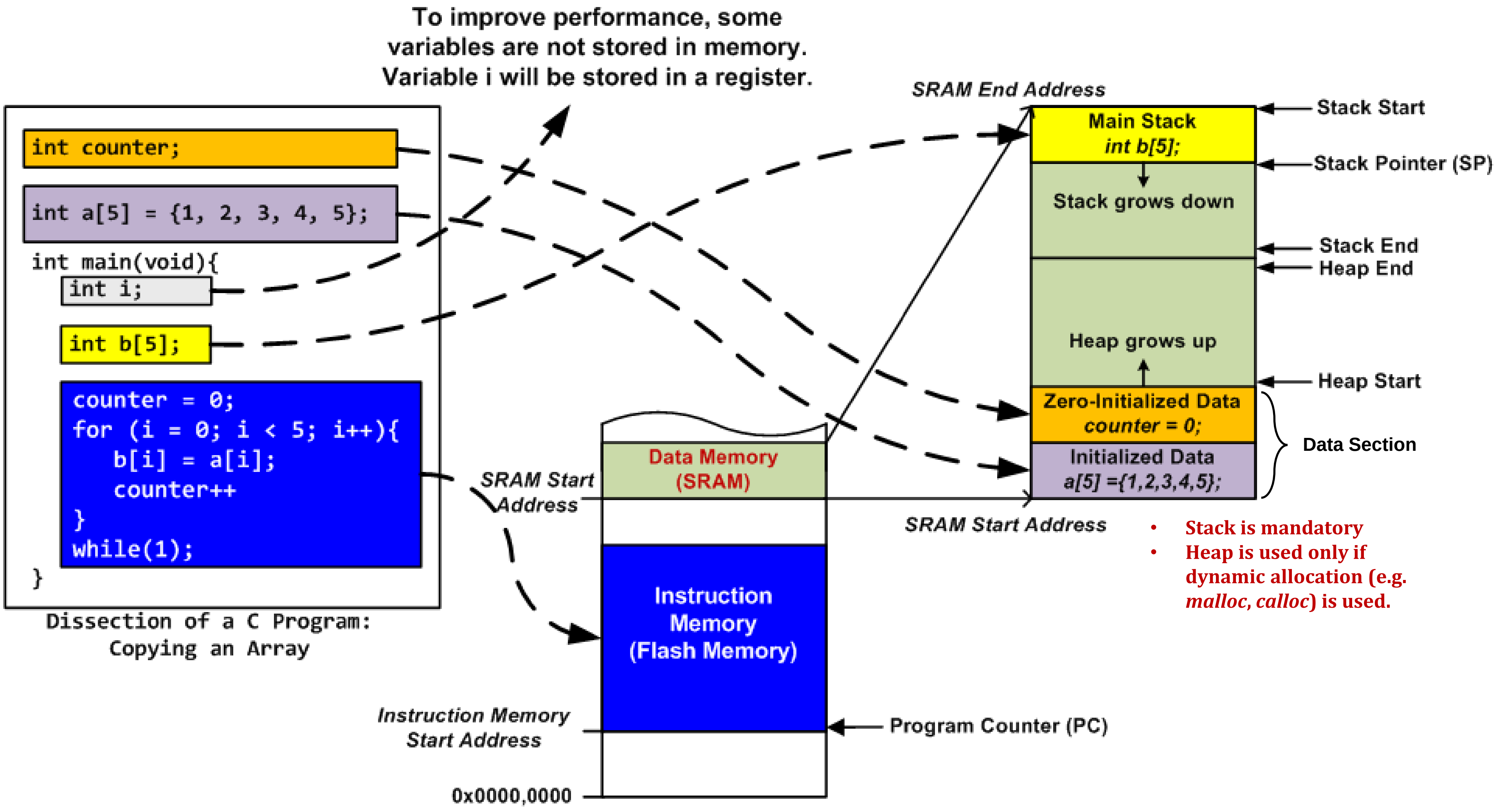
Loading Code and Data into Memory



```
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);
}
```

Dissection of a C Program:
Copying an Array

Loading Code and Data into Memory



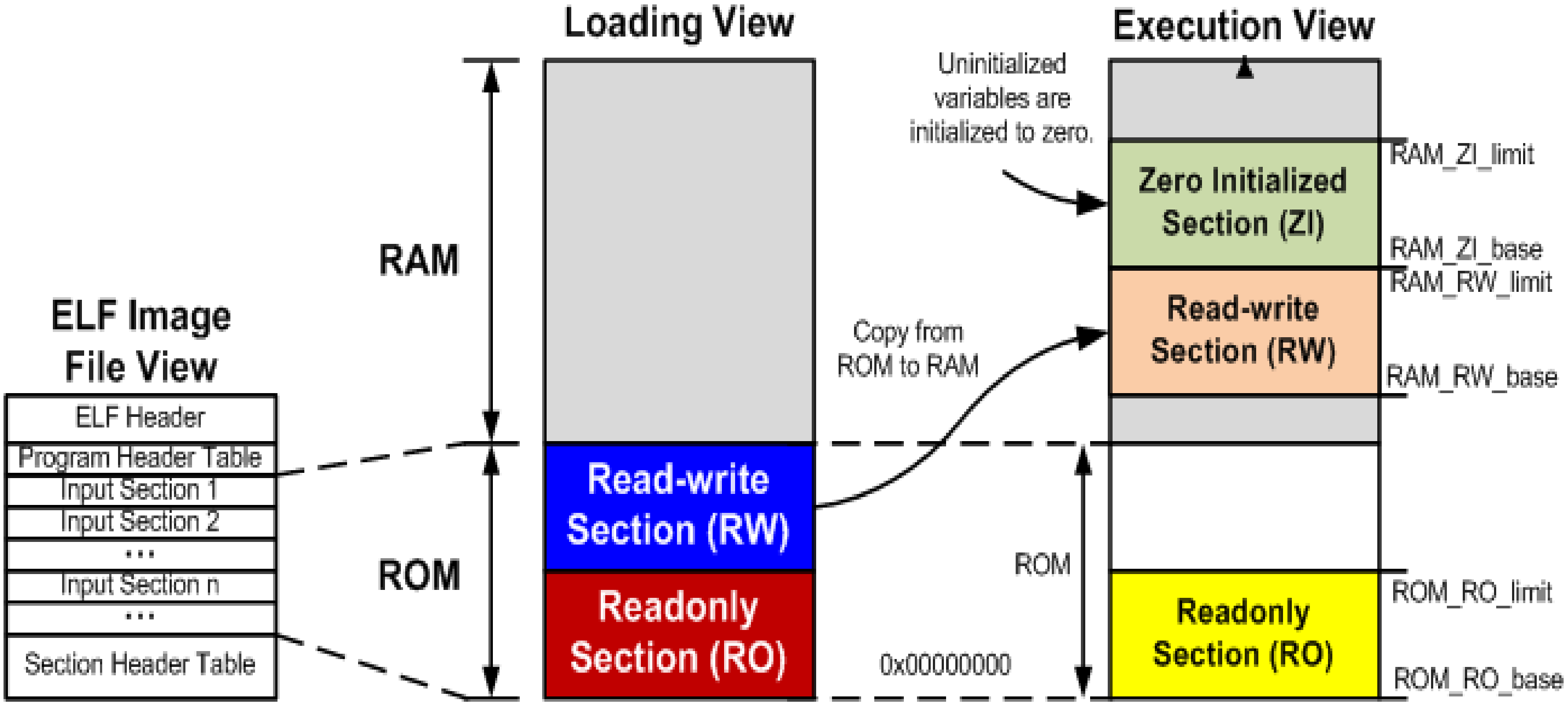
To improve performance, some variables are not stored in memory. Variable i will be stored in a register.

```
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);
}
```

Dissection of a C Program: Copying an Array

- **Stack is mandatory**
- **Heap is used only if dynamic allocation (e.g. *malloc, calloc*) is used.**

View of a Binary Program


ELF Image
File View
ELF Header
Program Header Table
Input Section 1
Input Section 2
...
Input Section n
...
Section Header Table
RAM
ROM
Loading View
Read-write
Section (RW)
Readonly
Section (RO)
Uninitialized
variables are
initialized to zero.
Copy from
ROM to RAM
ROM
0x00000000
Execution View
Zero Initialized
Section (ZI)
Read-write
Section (RW)
Readonly
Section (RO)
RAM_ZI_limit
RAM_ZI_base
RAM_RW_limit
RAM_RW_base
ROM_RO_limit
ROM_RO_base