# Machine Organization

CMPSCI 230 Computer Systems Principles

UMASSCS

SCHOOL OF COMPUTER SCIENCE

## Announcement

- No quiz until two weeks later
- HW5 is out, due March 7 (Mon)

# Objectives

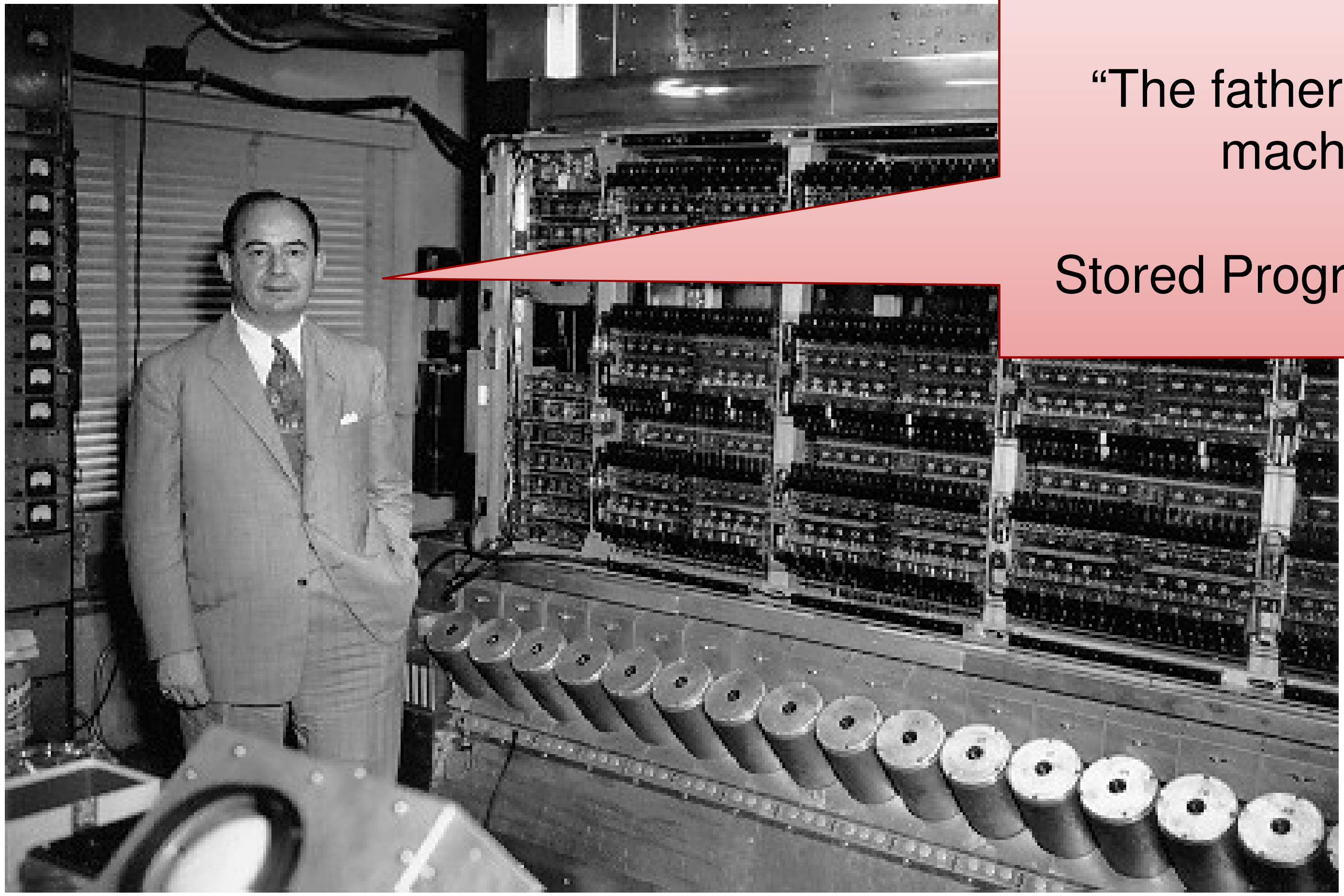
- **Machine Structure**
  - Understand the structure of a machine.
  - Learn about the *central processing unit* (CPU)
  - Learn about the internals of the CPU
- **Machine Execution**
  - Understand the basics of machine execution.
  - Understand how a program represented and interpreted “under-the-hood”
  - *How does a machine execute a program?*
- **Machine Pipelines**
  - Learn how machine’s exploit parallelism to improve performance
  - Understand the basics of *pipelines*
  - Understand how instructions are executed by a pipeline

## Objectives

- **Machine Structure**
  - Understand the structure of a machine.
  - Learn about the *central processing unit* (CPU)
  - Learn about the internals of the CPU
- **Machine Execution**
  - Understand the basics of machine execution.
  - Understand how a program represented and interpreted “under-the-hood”
  - *How does a machine execute a program?*
- **Machine Pipelines**
  - Learn how machine’s exploit parallelism to improve performance
  - Understand the basics of *pipelines*
  - Understand how instructions are executed by a pipeline

## Computer is versatile

## von Neumann



John von Neumann

“The father of modern machines”

Stored Program Concept

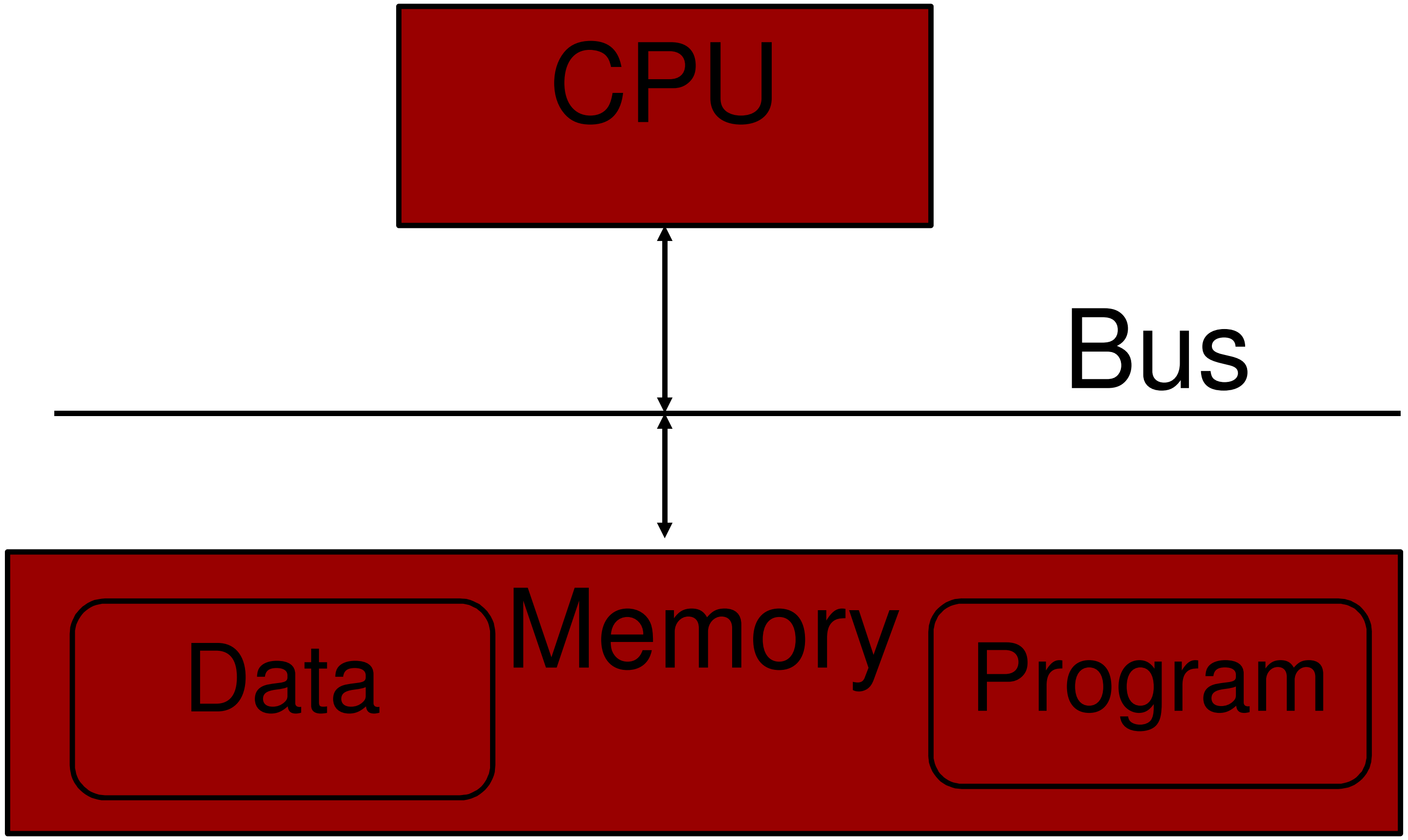
EDVAC 1945

## Stored Program Concept

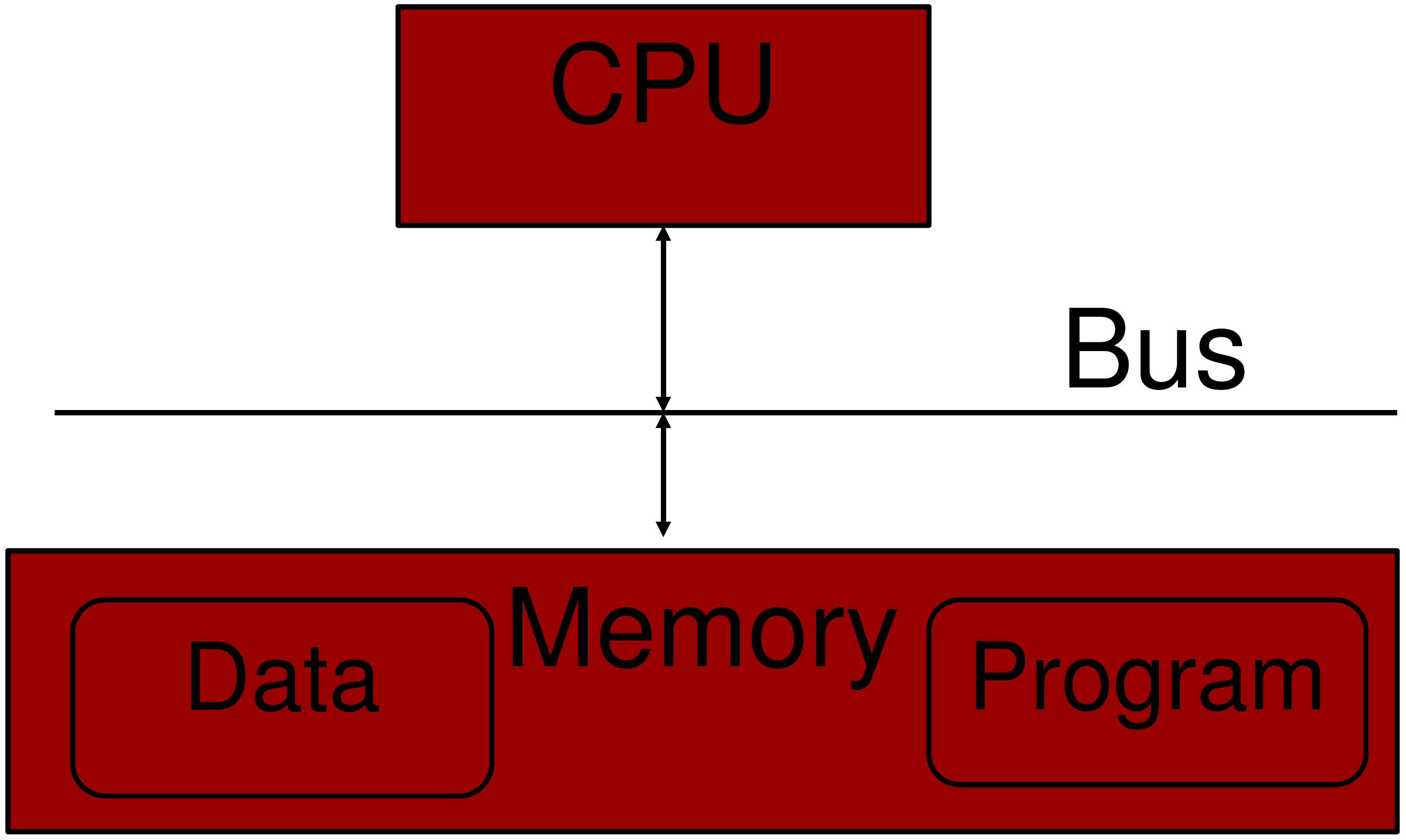
- **Fixed Machines**
  - Early machines had “fixed” programs
  - Can’t be used for other purpose
  - Change required re-design & re-wiring!

## Stored Program Concept

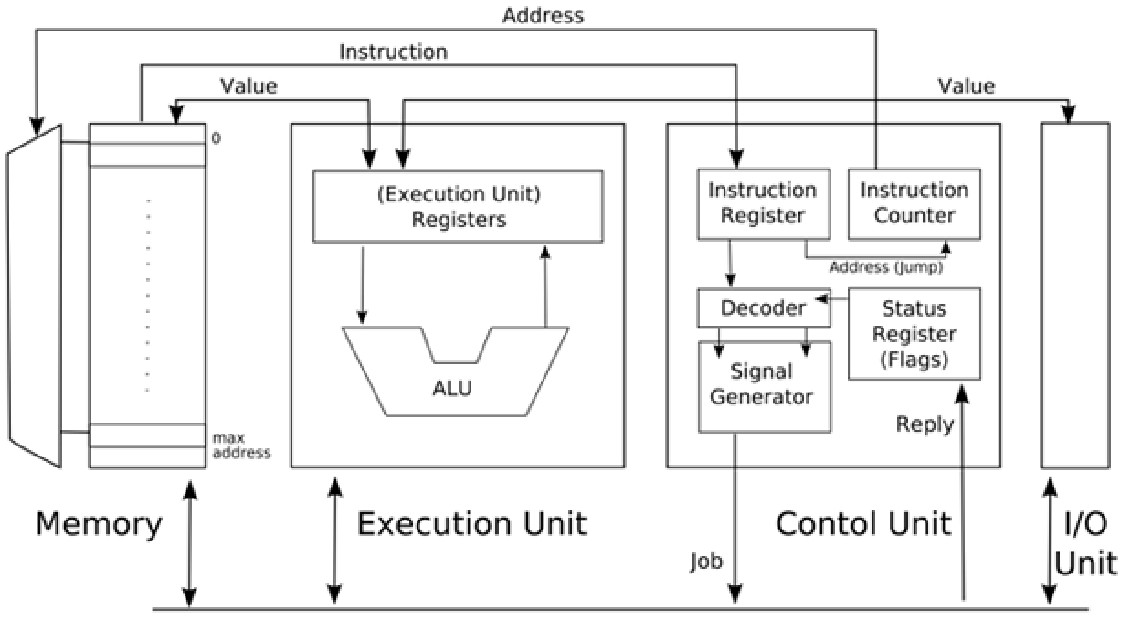
- **Fixed Machines**
  - Early machines had “fixed” programs
  - Can’t be used for other purpose
  - Change required re-design & re-wiring!
- **General Purpose Machine**
  - Need more versatility
  - Programs stored in a “memory”
  - Machine can be re-programmed!

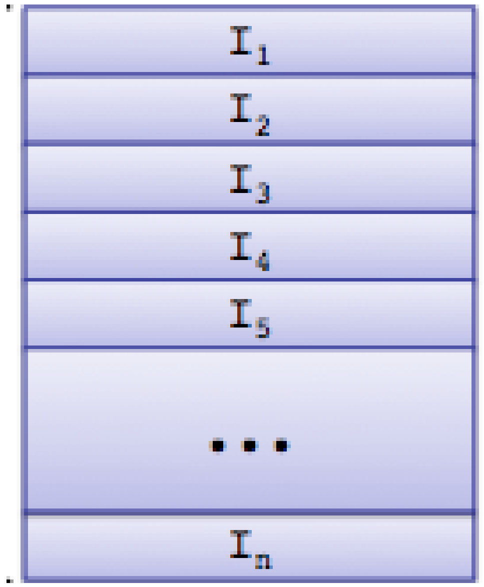

CPU
Bus
Memory
Data
Program

## So, what does the machine look like inside?



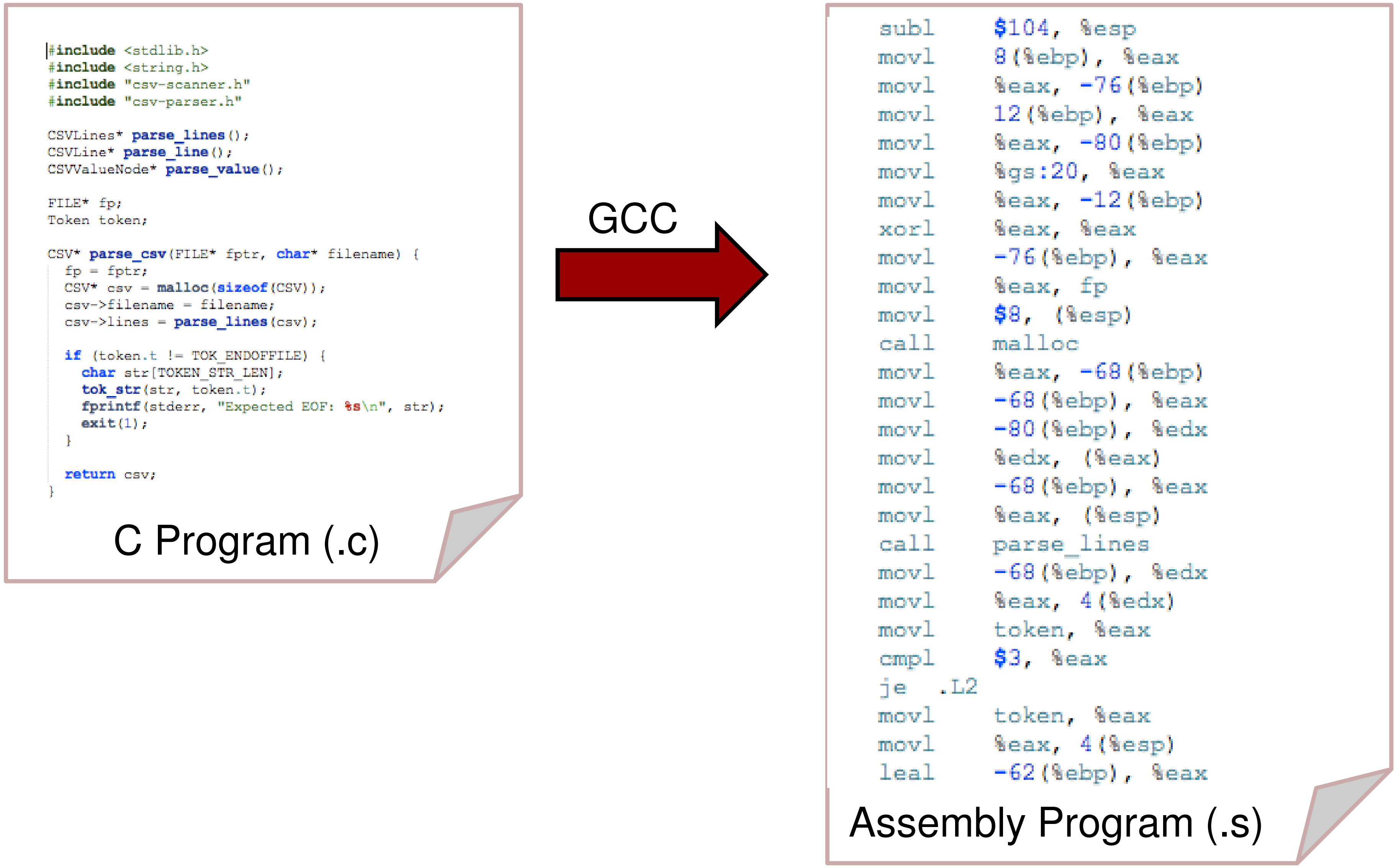
# “von Neumann architecture”





The von-Neumann concept is the basic concept for universal microprocessors.

# Program Compilation and Assembly



```c
#include <stdlib.h>
#include <string.h>
#include "csv-scanner.h"
#include "csv-parser.h"

CSVLines* parse_lines();
CSVLine* parse_line();
CSVValueNode* parse_value();

FILE* fp;
Token token;

CSV* parse_csv(FILE* fptr, char* filename) {
  fp = fptr;
  CSV* csv = malloc(sizeof(CSV));
  csv->filename = filename;
  csv->lines = parse_lines(csv);

  if (token.t != TOK_ENDOFFILE) {
    char str[TOKEN_STR_LEN];
    tok_str(str, token.t);
    fprintf(stderr, "Expected EOF: %s\n", str);
    exit(1);
  }

  return csv;
}
```

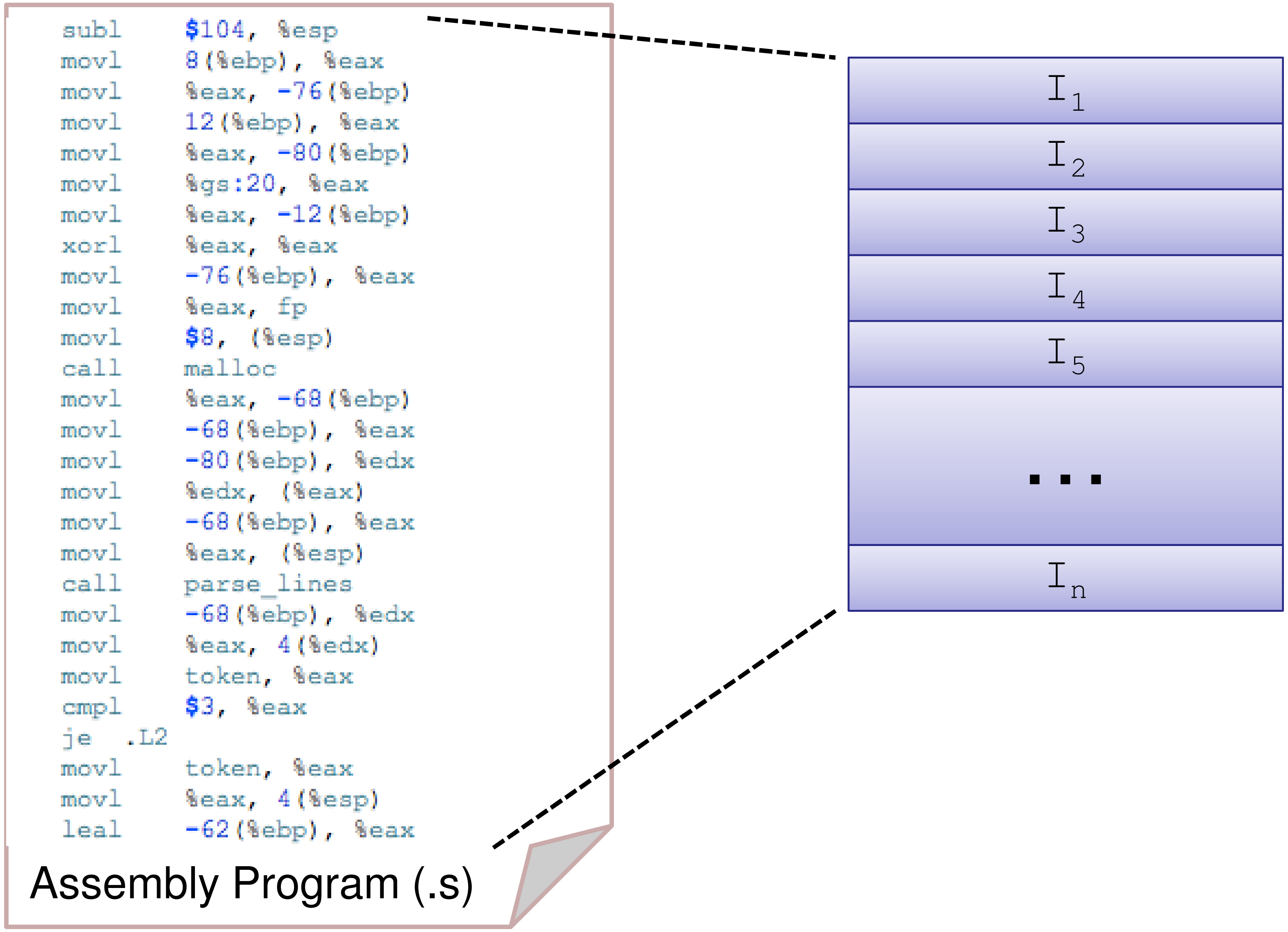
C Program (.c)

GCC

```
subl     $104, %esp
movl     8(%ebp), %eax
movl     %eax, -76(%ebp)
movl     12(%ebp), %eax
movl     %eax, -80(%ebp)
movl     %gs:20, %eax
movl     %eax, -12(%ebp)
xorl     %eax, %eax
movl     -76(%ebp), %eax
movl     %eax, fp
movl     $8, (%esp)
call     malloc
movl     %eax, -68(%ebp)
movl     -68(%ebp), %eax
movl     -80(%ebp), %edx
movl     %edx, (%eax)
movl     -68(%ebp), %eax
movl     %eax, (%esp)
call     parse_lines
movl     -68(%ebp), %edx
movl     %eax, 4(%edx)
movl     token, %eax
cmpl     $3, %eax
je   .L2
movl     token, %eax
movl     %eax, 4(%esp)
leal     -62(%ebp), %eax
```

Assembly Program (.s)

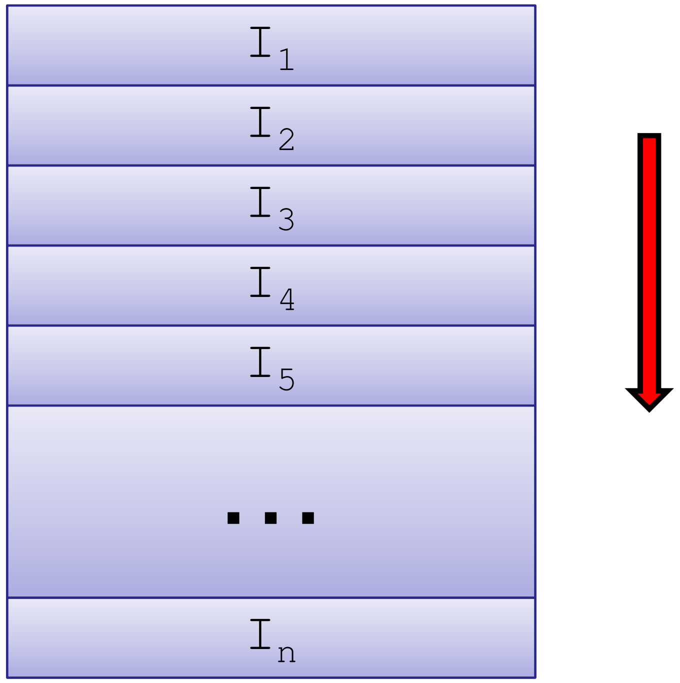
## Assembly and Instructions



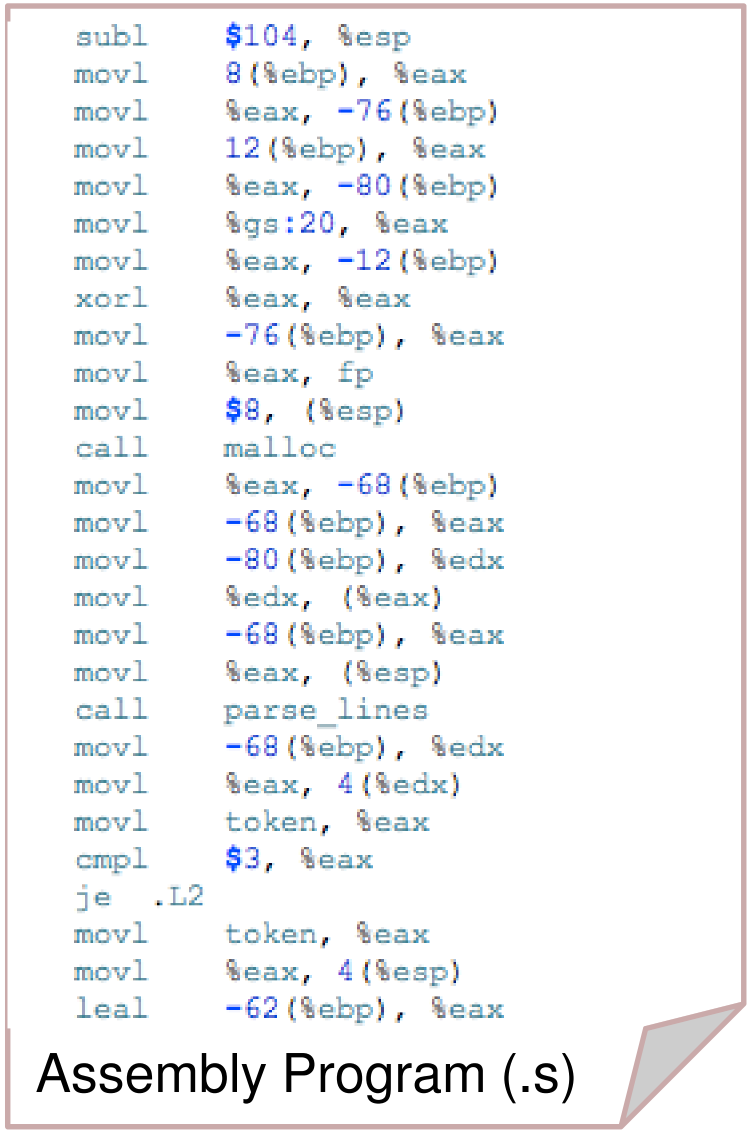
```
    subl     $104, %esp
    movl     8(%ebp), %eax
    movl     %eax, -76(%ebp)
    movl     12(%ebp), %eax
    movl     %eax, -80(%ebp)
    movl     %gs:20, %eax
    movl     %eax, -12(%ebp)
    xorl     %eax, %eax
    movl     -76(%ebp), %eax
    movl     %eax, fp
    movl     $8, (%esp)
    call     malloc
    movl     %eax, -68(%ebp)
    movl     -68(%ebp), %eax
    movl     -80(%ebp), %edx
    movl     %edx, (%eax)
    movl     -68(%ebp), %eax
    movl     %eax, (%esp)
    call     parse_lines
    movl     -68(%ebp), %edx
    movl     %eax, 4(%edx)
    movl     token, %eax
    cmpl     $3, %eax
    je   .L2
    movl     token, %eax
    movl     %eax, 4(%esp)
    leal     -62(%ebp), %eax
```

Assembly Program (.s)

## Executing Instructions



## Assembly and Instructions



```
subl    $104, %esp
movl    8(%ebp), %eax
movl    %eax, -76(%ebp)
movl    12(%ebp), %eax
movl    %eax, -80(%ebp)
movl    %gs:20, %eax
movl    %eax, -12(%ebp)
xorl    %eax, %eax
movl    -76(%ebp), %eax
movl    %eax, fp
movl    $8, (%esp)
call    malloc
movl    %eax, -68(%ebp)
movl    -68(%ebp), %eax
movl    -80(%ebp), %edx
movl    %edx, (%eax)
movl    -68(%ebp), %eax
movl    %eax, (%esp)
call    parse_lines
movl    -68(%ebp), %edx
movl    %eax, 4(%edx)
movl    token, %eax
cmpl    $3, %eax
je  .L2
movl    token, %eax
movl    %eax, 4(%esp)
leal    -62(%ebp), %eax
```

Assembly Program (.s)

Statement:
Assignment

id3 = id1 op id2
id2 = op id1
id2 = id1

# Assembly and Instructions

```
subl     $104, %esp
movl     8(%ebp), %eax
movl     %eax, -76(%ebp)
movl     12(%ebp), %eax
movl     %eax, -80(%ebp)
movl     %gs:20, %eax
movl     %eax, -12(%ebp)
xorl     %eax, %eax
movl     -76(%ebp), %eax
movl     %eax, fp
movl     $8, (%esp)
call     malloc
movl     %eax, -68(%ebp)
movl     -68(%ebp), %eax
movl     -80(%ebp), %edx
movl     %edx, (%eax)
movl     -68(%ebp), %eax
movl     %eax, (%esp)
call     parse_lines
movl     -68(%ebp), %edx
movl     %eax, 4(%edx)
movl     token, %eax
cmpl     $3, %eax
je   .L2
movl     token, %eax
movl     %eax, 4(%esp)
leal     -62(%ebp), %eax
```

Assembly Program (.s)

Statement:
Assignment

id3 = id1 op id2

id2 = op id1

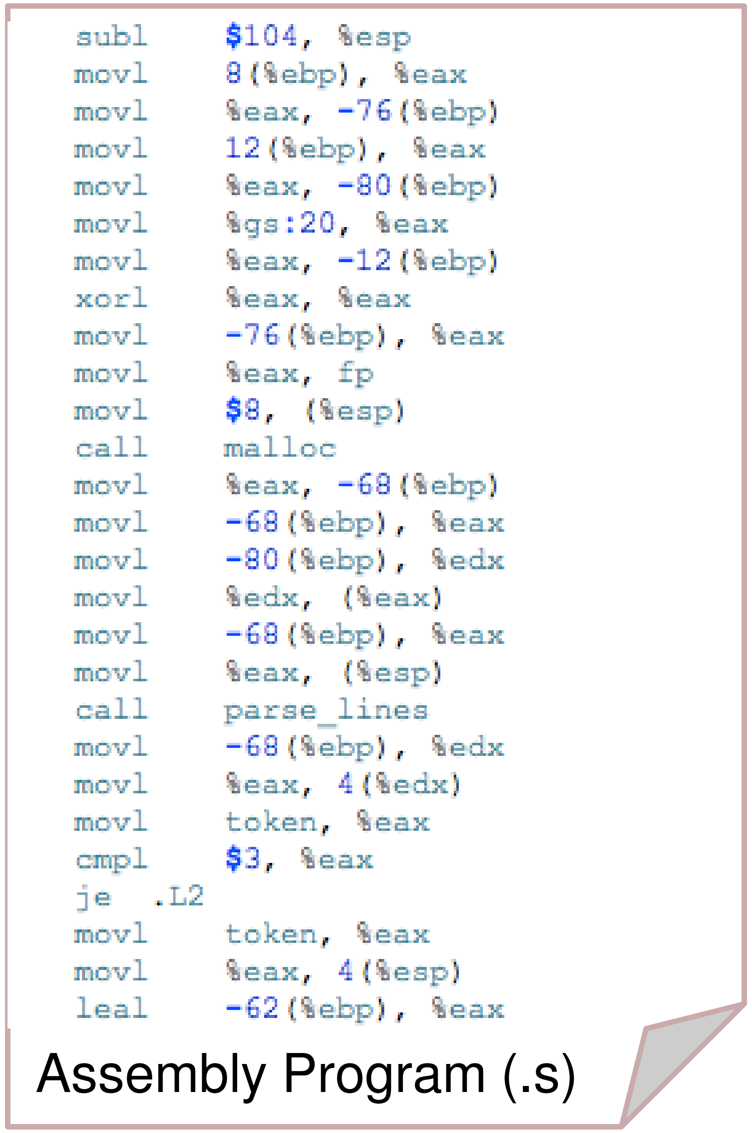
id2 = id1

Stack operation

push id

id = pop()

# Assembly and Instructions



```
subl    $104, %esp
movl    8(%ebp), %eax
movl    %eax, -76(%ebp)
movl    12(%ebp), %eax
movl    %eax, -80(%ebp)
movl    %gs:20, %eax
movl    %eax, -12(%ebp)
xorl    %eax, %eax
movl    -76(%ebp), %eax
movl    %eax, fp
movl    $8, (%esp)
call    malloc
movl    %eax, -68(%ebp)
movl    -68(%ebp), %eax
movl    -80(%ebp), %edx
movl    %edx, (%eax)
movl    -68(%ebp), %eax
movl    %eax, (%esp)
call    parse_lines
movl    -68(%ebp), %edx
movl    %eax, 4(%edx)
movl    token, %eax
cmpl    $3, %eax
je  .L2
movl    token, %eax
movl    %eax, 4(%esp)
leal    -62(%ebp), %eax
```

Assembly Program (.s)

Statement:

Assignment

id3 = id1 op id2

id2 = op id1

id2 = id1

Stack operation

push id

id = pop()

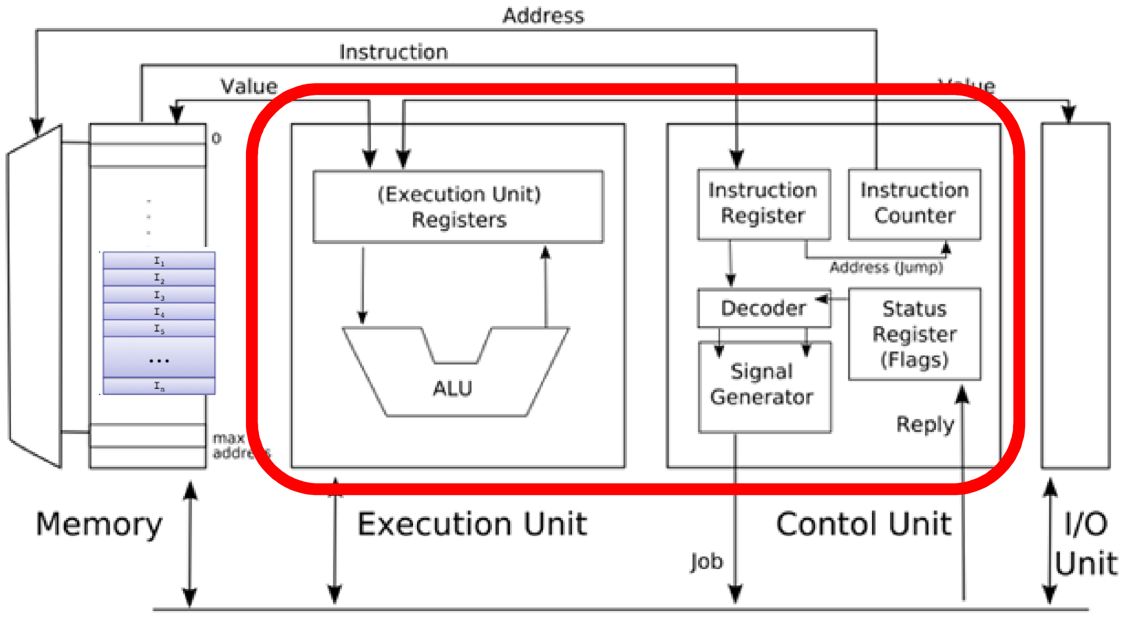
Jump

if id1 op id2 jump L

L

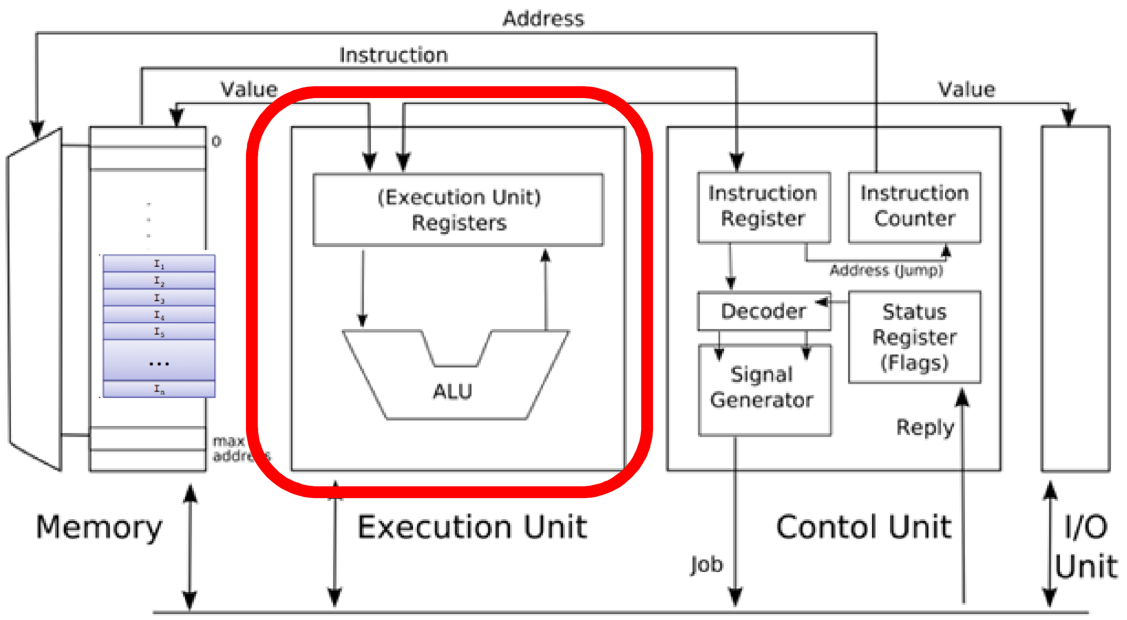
jump L

# “von Neumann architecture”



The von-Neumann concept is the basic concept for universal microprocessors.

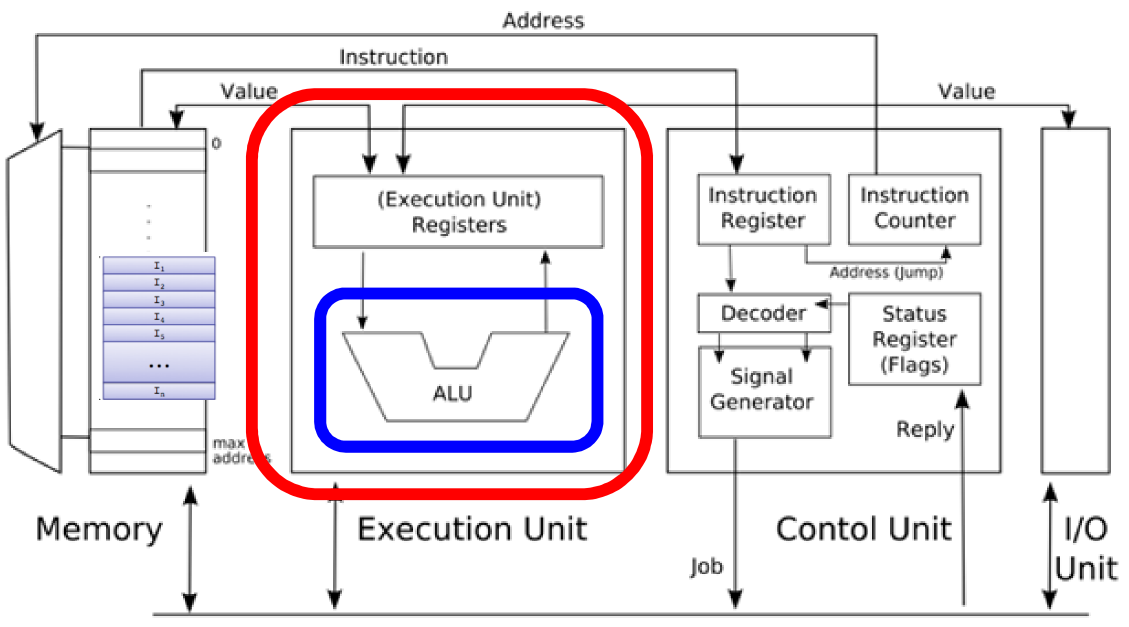
## “von Neumann architecture”



The **Execution Unit** is the core of the processor.

The von-Neumann concept is the basic concept for universal microprocessors.

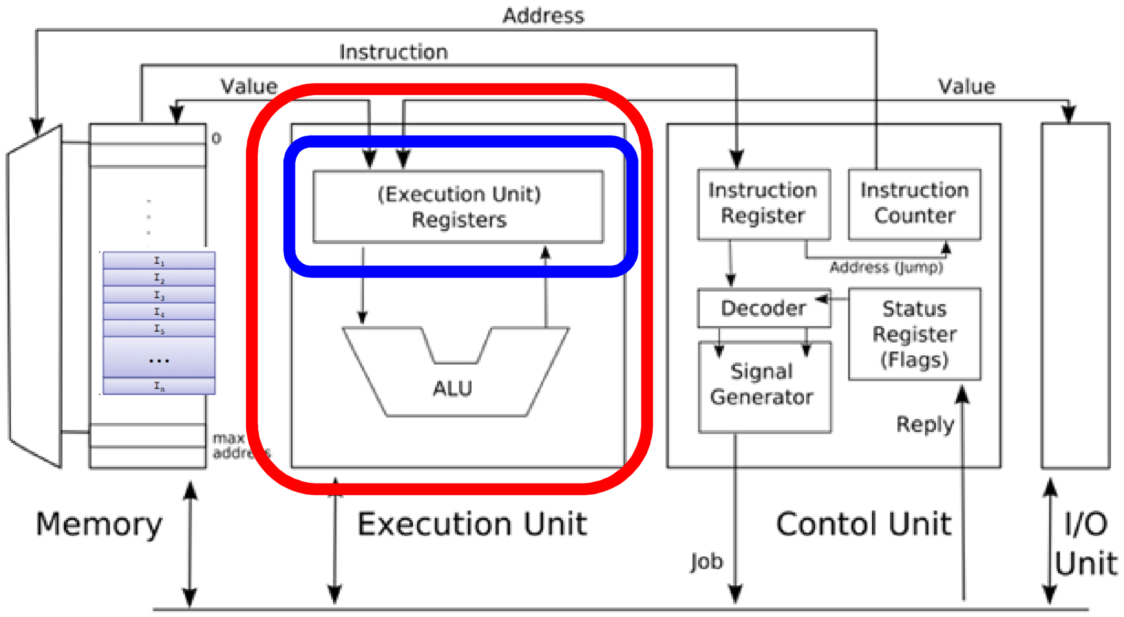
## “von Neumann architecture”



The **Arithmetic Logic Unit (ALU)** calculates!

The von-Neumann concept is the basic concept for universal microprocessors.

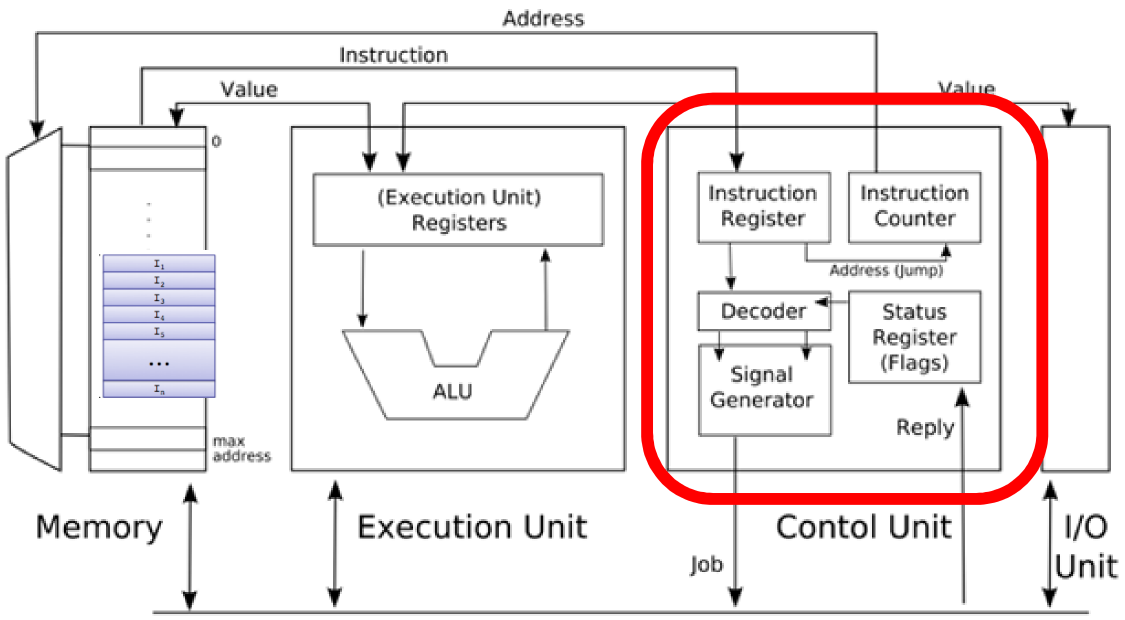
# "von Neumann architecture"



The **Registers** are a very fast memory to keep the operands needed for the actual operations.

The von-Neumann concept is the basic concept for universal microprocessors.

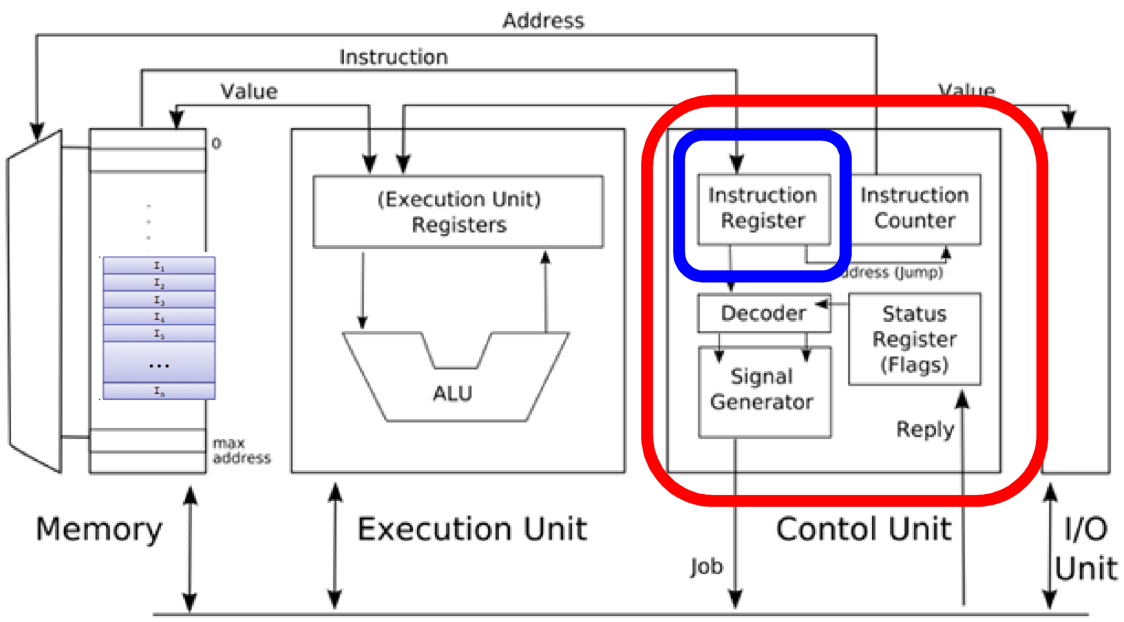
## “von Neumann architecture”



The **Control Unit** interprets instructions of the program and controls the other parts of the processor.

The von-Neumann concept is the basic concept for universal microprocessors.

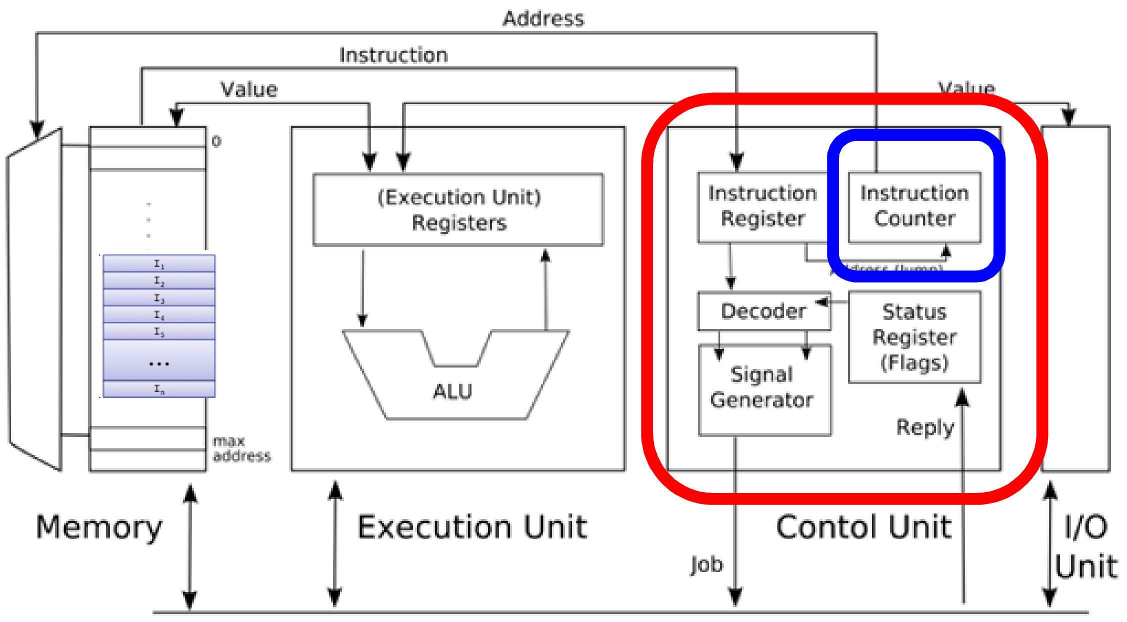
## “von Neumann architecture”



The **Instruction Register** stores the current instruction.

The von-Neumann concept is the basic concept for universal microprocessors.

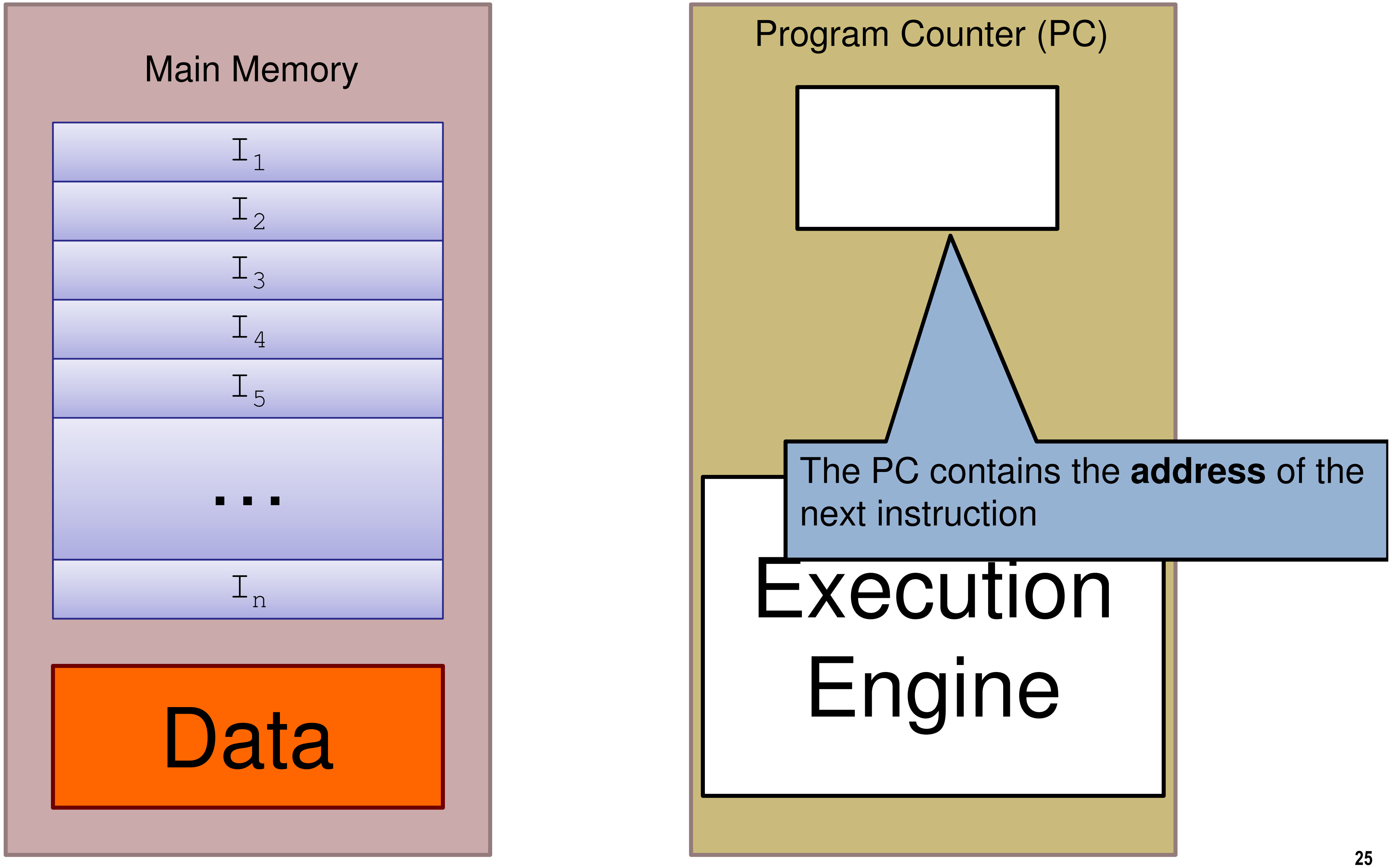
## “von Neumann architecture”



The **Instruction Counter (or Program Counter)** stores the address (pointer) of the next instruction. *Increments automatically*

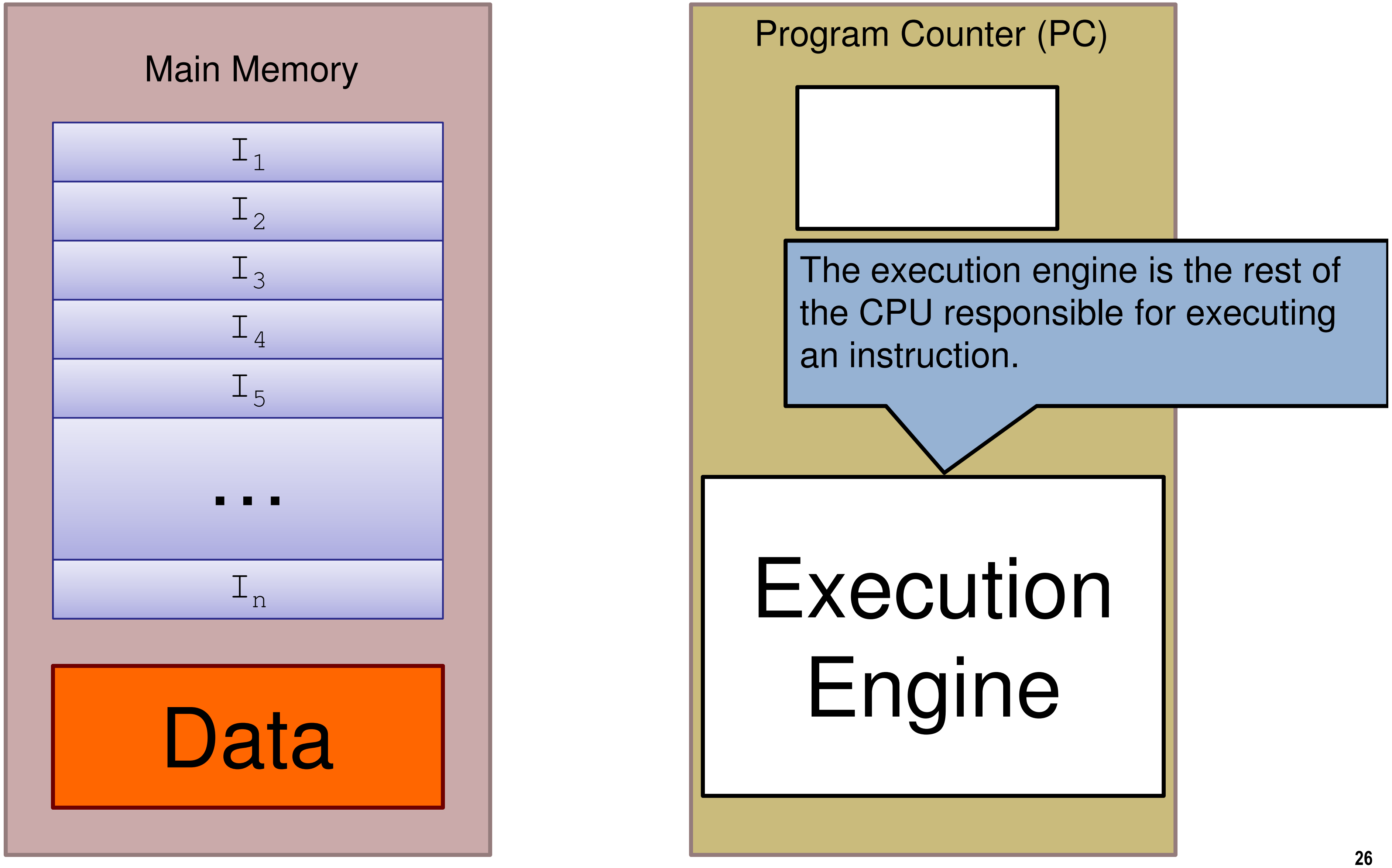
The von-Neumann concept is the basic concept for universal microprocessors.

## Executing Instructions



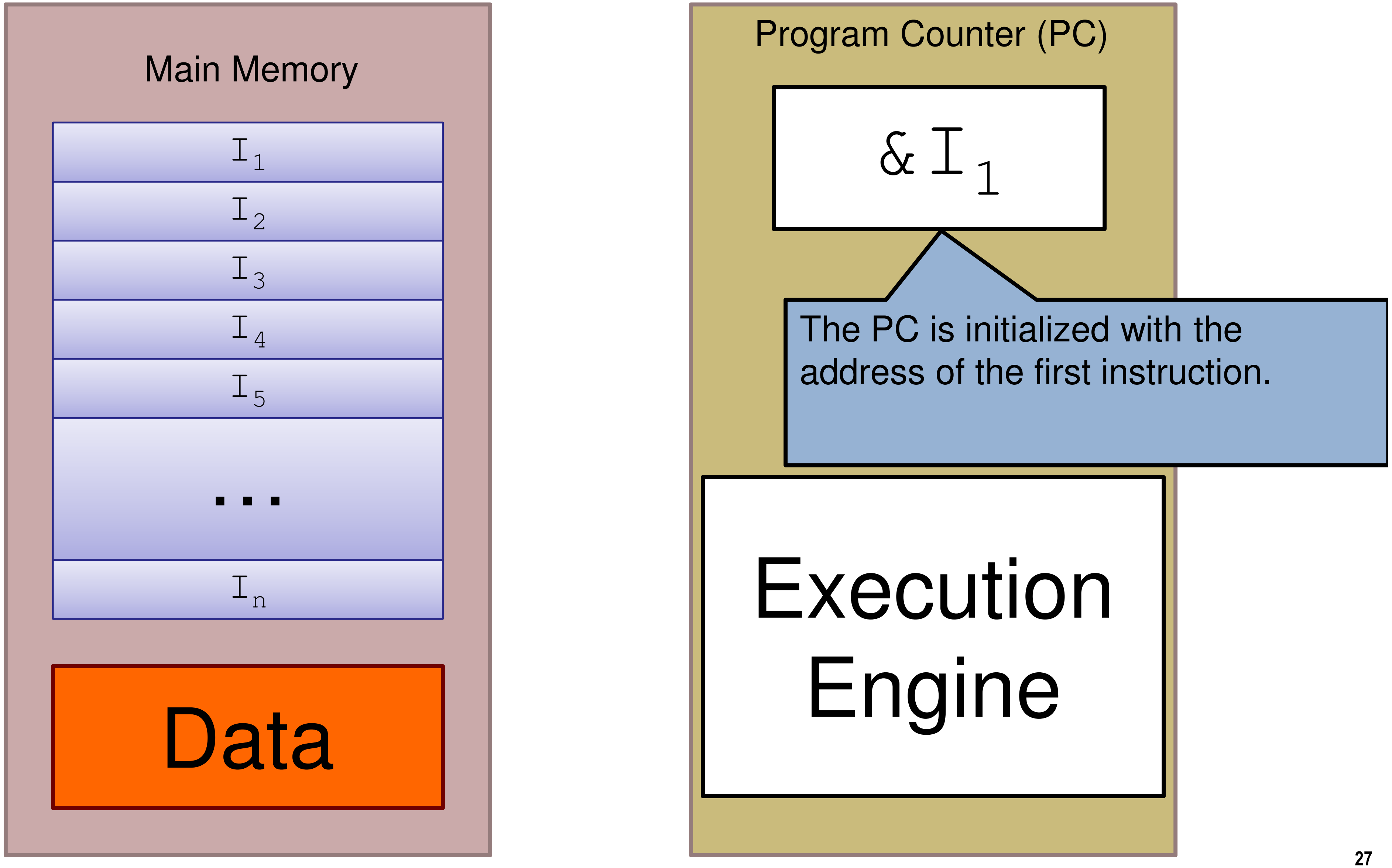
The PC contains the **address** of the next instruction

## Executing Instructions



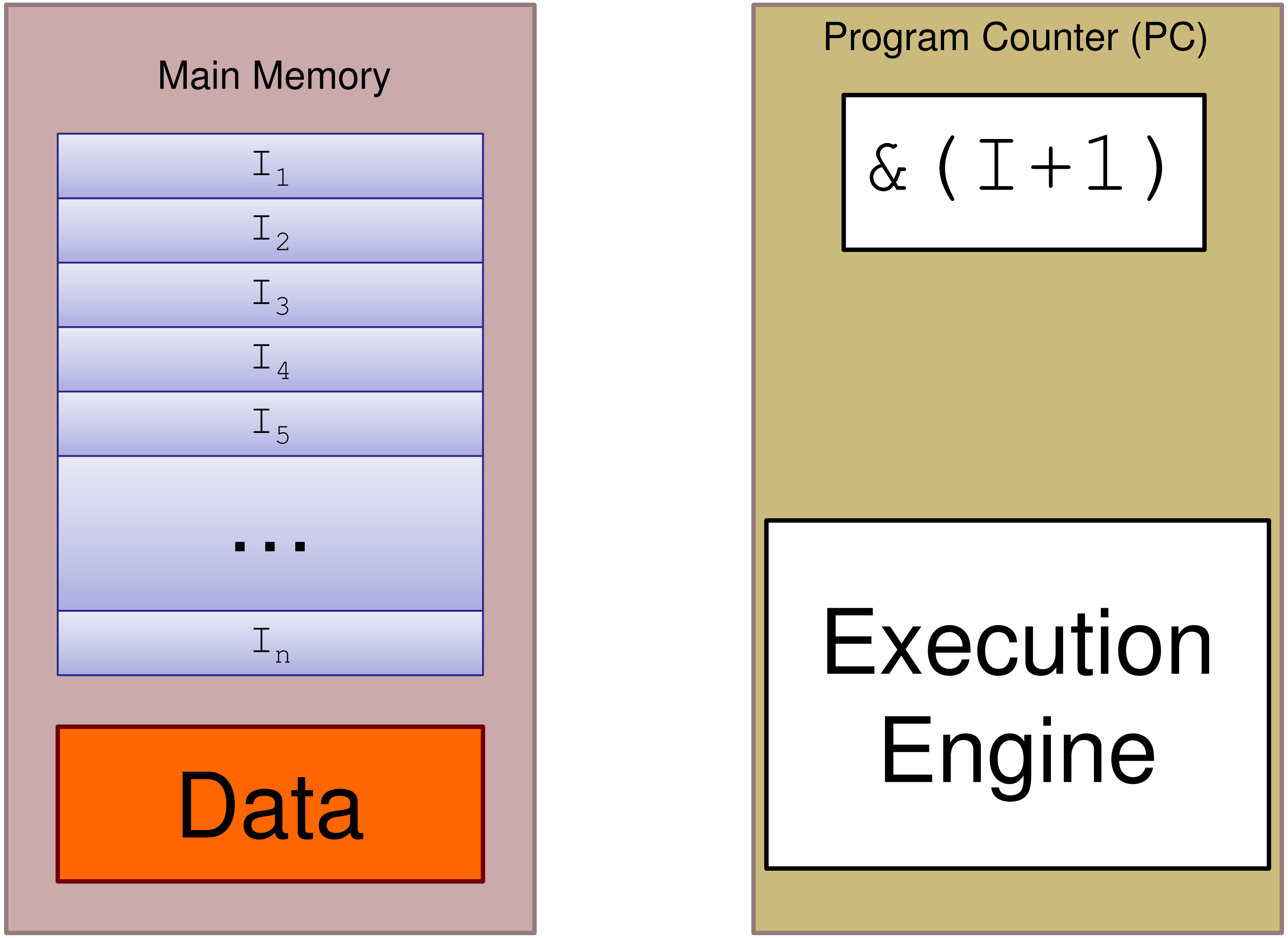
The execution engine is the rest of the CPU responsible for executing an instruction.

## Executing Instructions

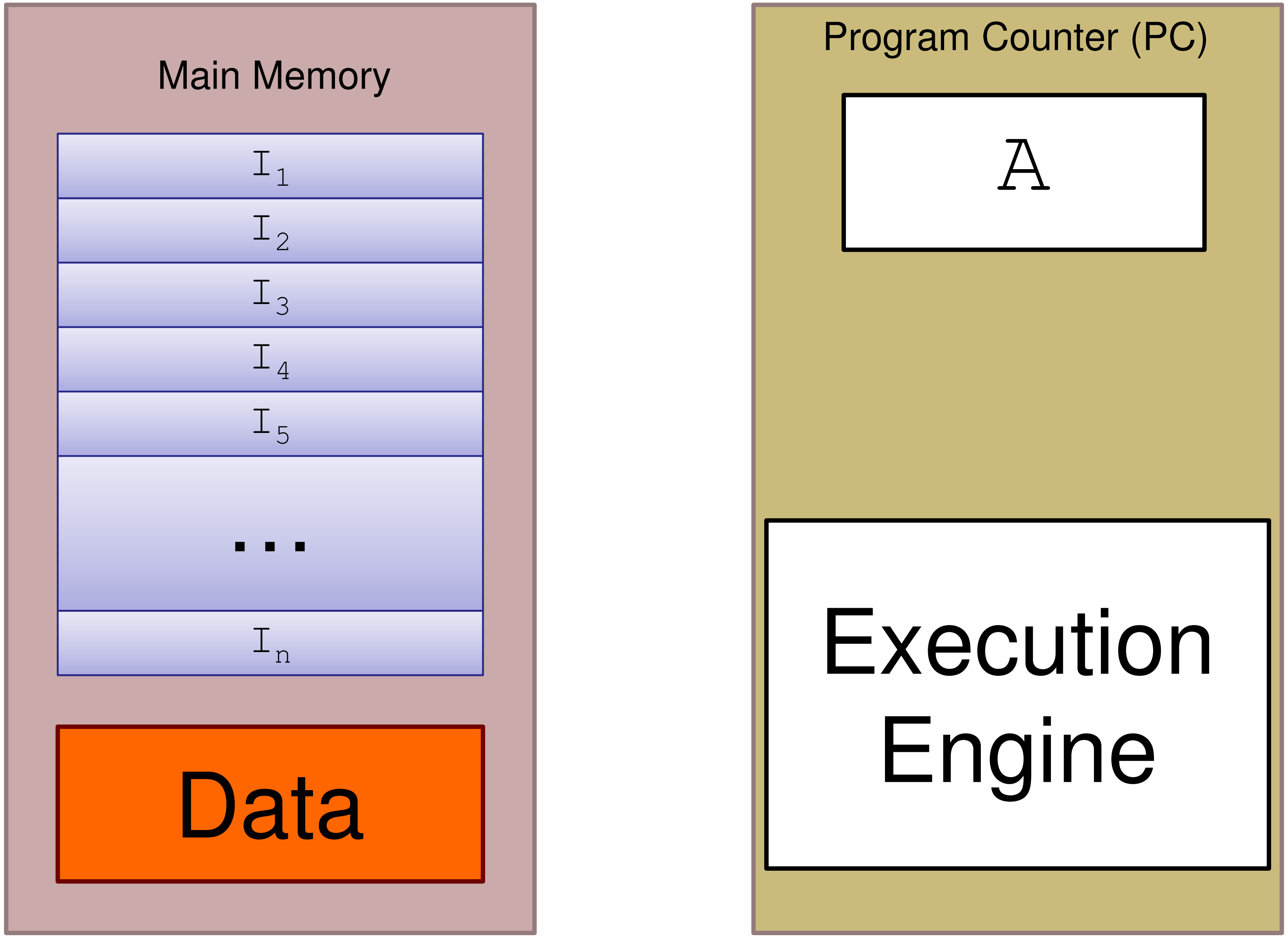


The PC is initialized with the address of the first instruction.

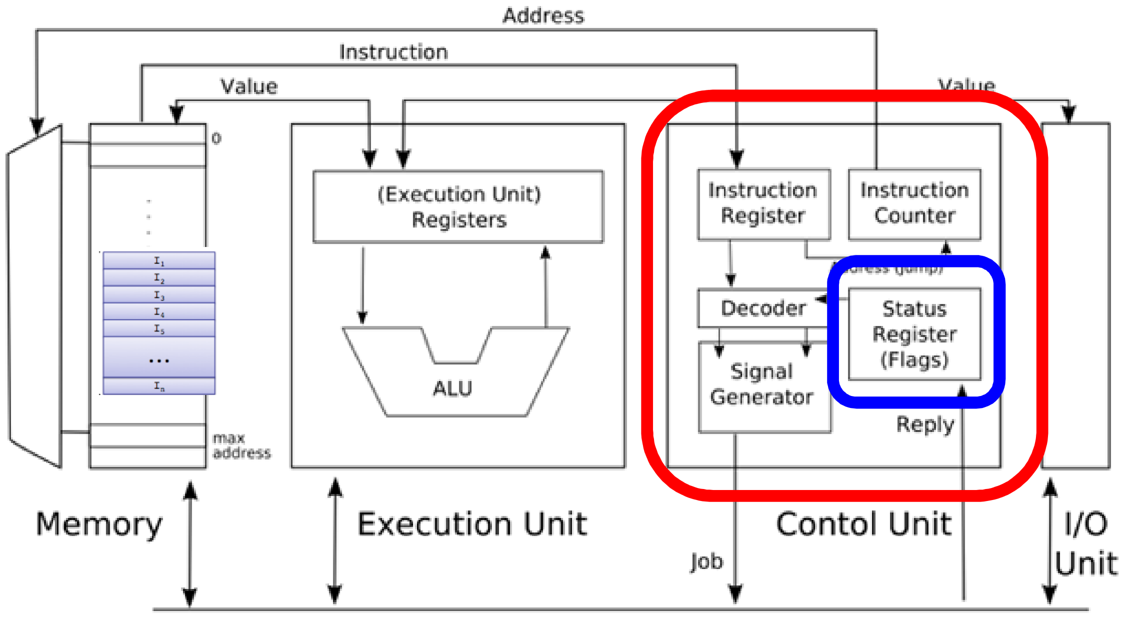
## Executing Instructions



## Executing Instructions



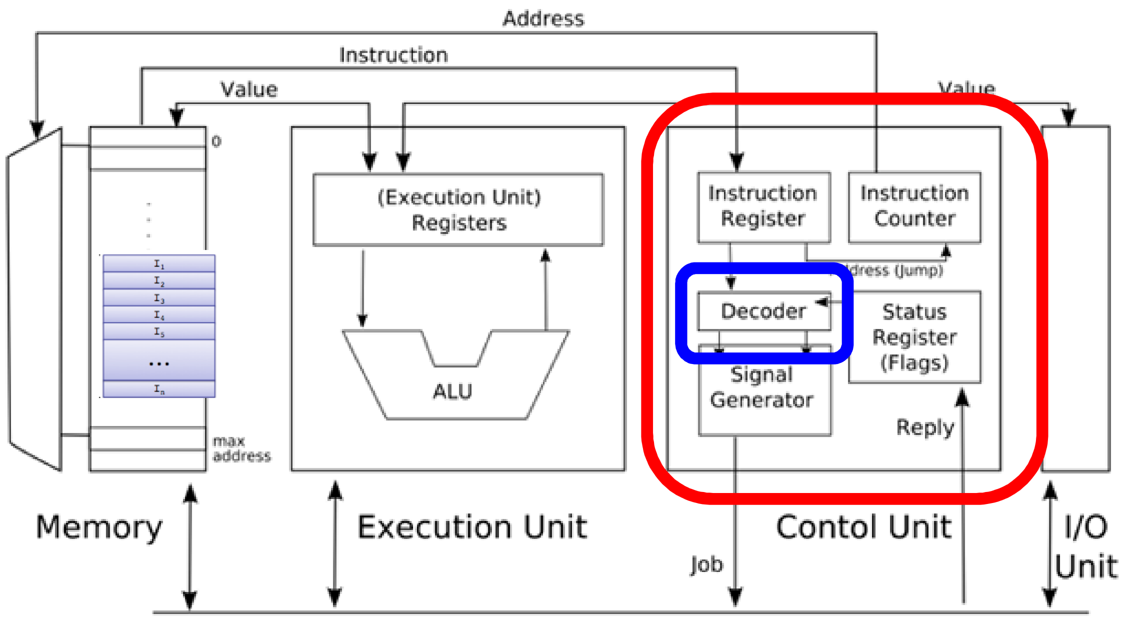
## “von Neumann architecture”



The **Status Register** (flags) stores information about the result of the last operation.

The von-Neumann concept is the basic concept for universal microprocessors.

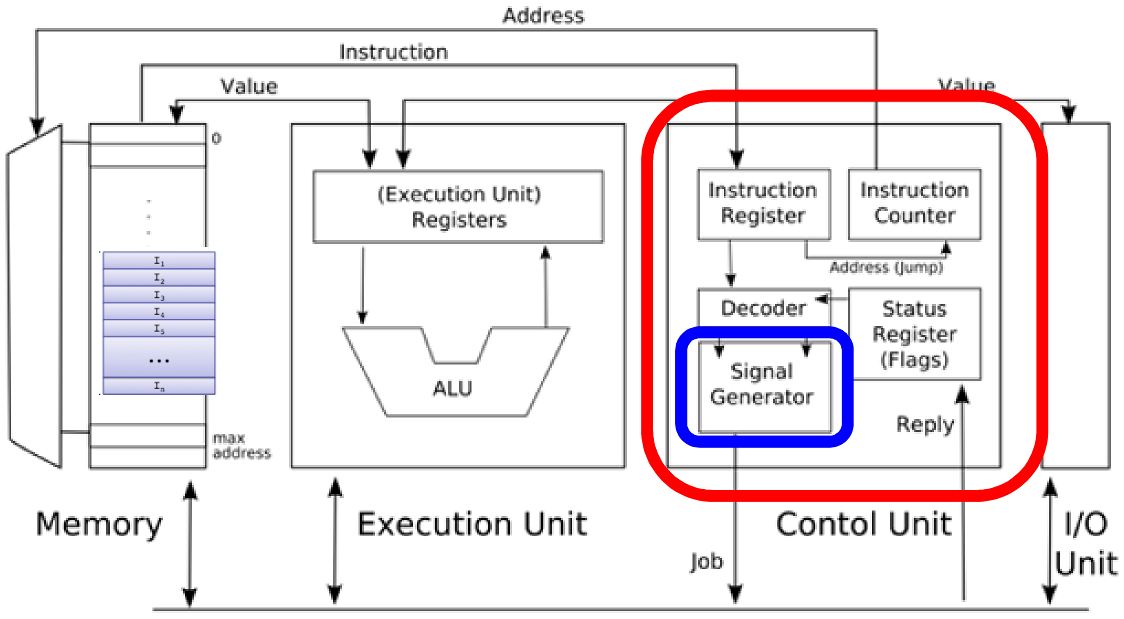
## “von Neumann architecture”



The **Decoder** “reads” the instruction and determines what signals to generate.

The von-Neumann concept is the basic concept for universal microprocessors.

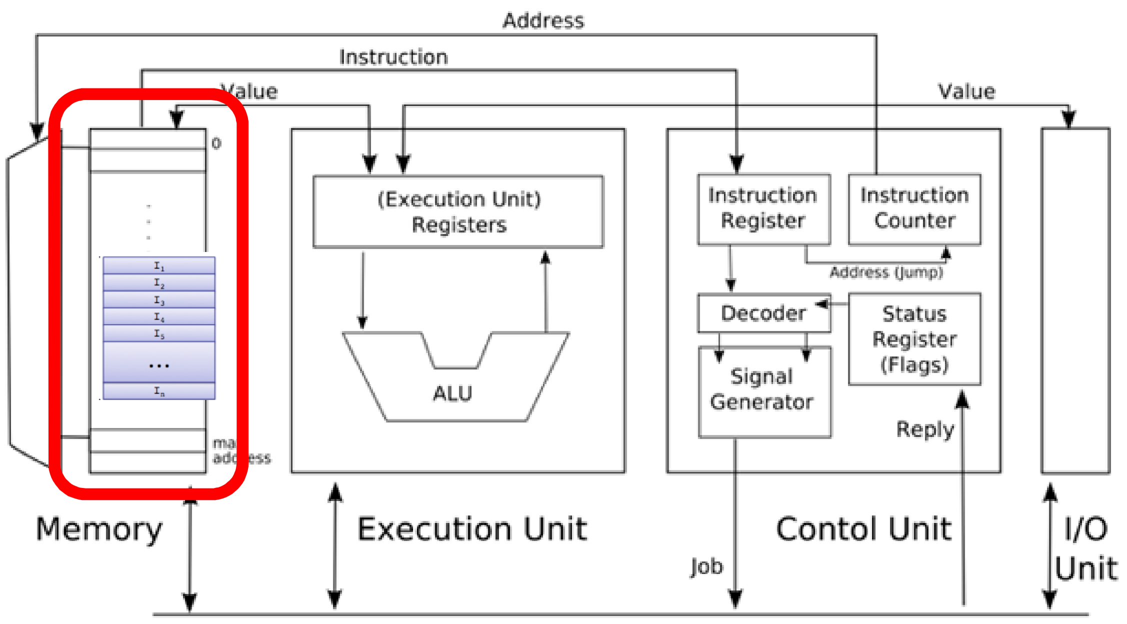
# “von Neumann architecture”



The **Signal Generator** communicates with the rest of the processor.

The von-Neumann concept is the basic concept for universal microprocessors.

## “von Neumann architecture”



The **Memory** stores both data and instructions of the program. Both are accessed by **address** (e.g., pointer).

The von-Neumann concept is the basic concept for universal microprocessors.

## i-clicker question

- Which one of the following about von Neumann machine is wrong?

A. Programs are stored and read from the same medium as data, usually disk and RAM.

B. Instruction Counter (aka PC) is a special purpose register within the CPU that contains the address of next instruction to be executed

C. Decoder is a special purpose register that contains the instruction currently being executed within the CPU

D. Data has to be read into the register before it can be manipulated by the CPU.

## i-clicker question

- Which one of the following about von Neumann machine is wrong? Sol: C

A. Programs are stored and read from the same medium as data, usually disk and RAM.

B. Instruction Counter (aka PC) is a special purpose register within the CPU that contains the address of next instruction to be executed

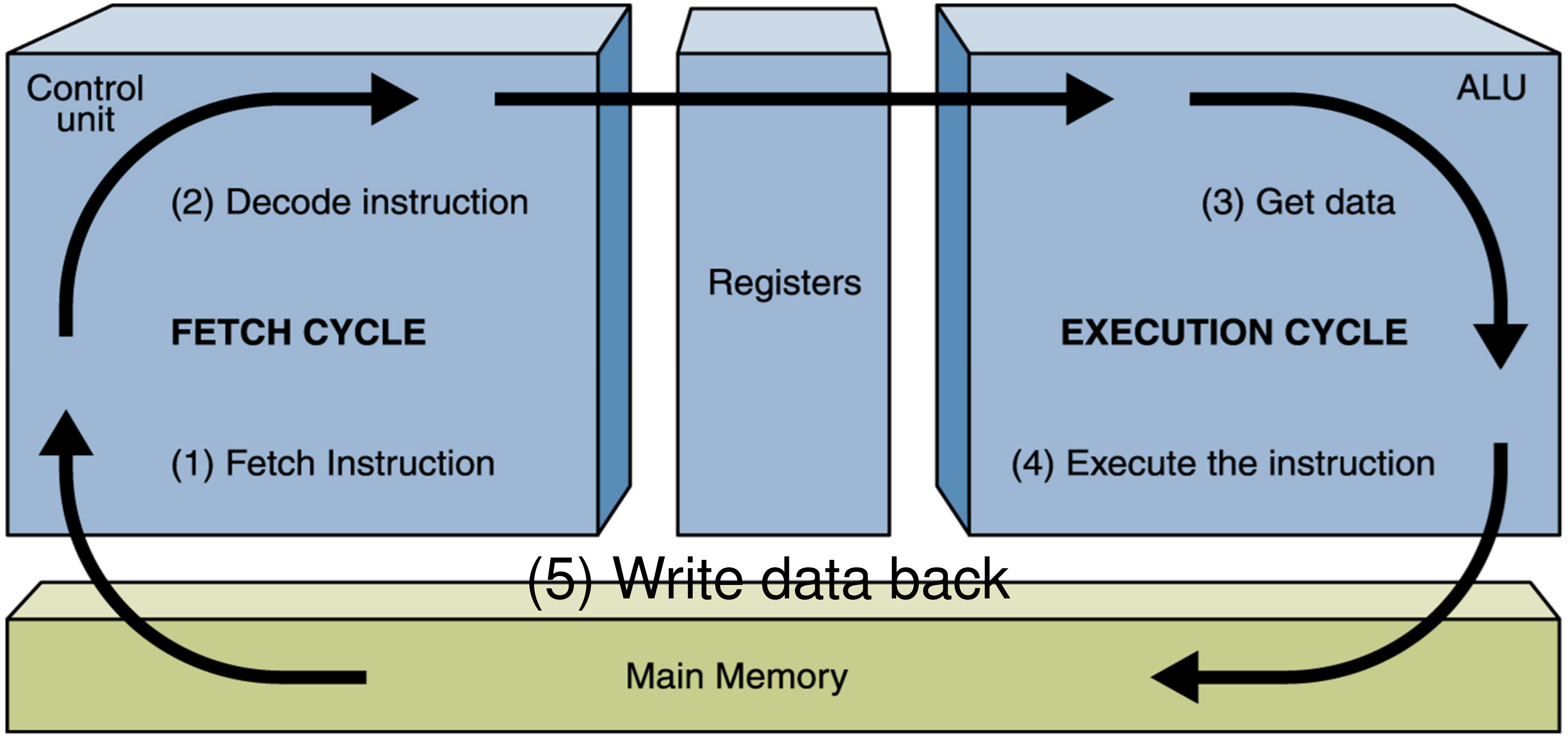
C. Decoder is a special purpose register that contains the instruction currently being executed within the CPU

D. Data has to be read into the register before it can be manipulated by the CPU.

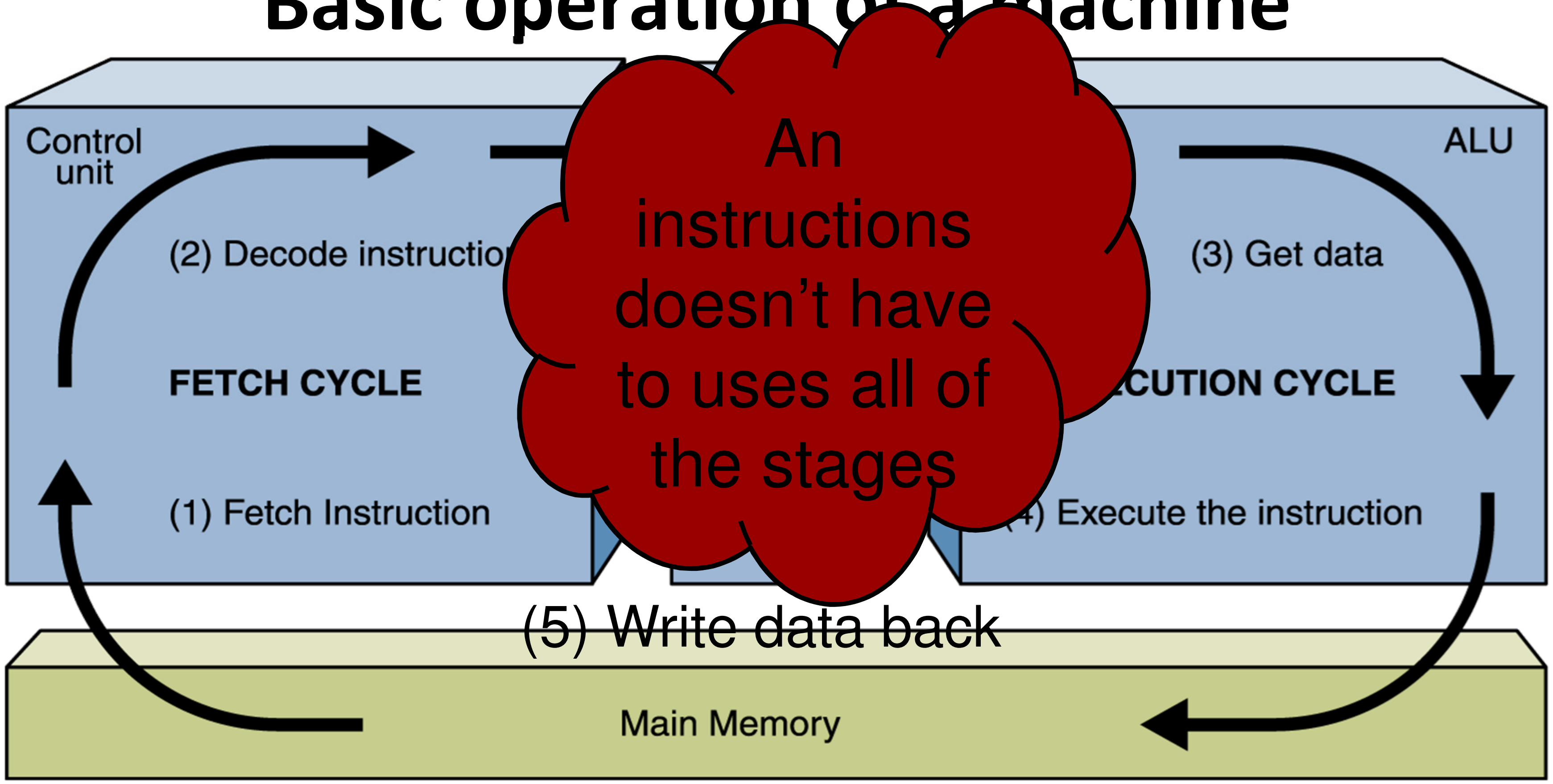
## Objectives

- **Machine Structure**
  - Understand the structure of a machine.
  - Learn about the *central processing unit* (CPU)
  - Learn about the internals of the CPU
- **Machine Execution**
  - Understand the basics of machine execution.
  - Understand how a program represented and interpreted “under-the-hood”
  - *How does a machine execute a program?*
- **Machine Pipelines**
  - Learn how machine’s exploit parallelism to improve performance
  - Understand the basics of *pipelines*
  - Understand how instructions are executed by a pipeline

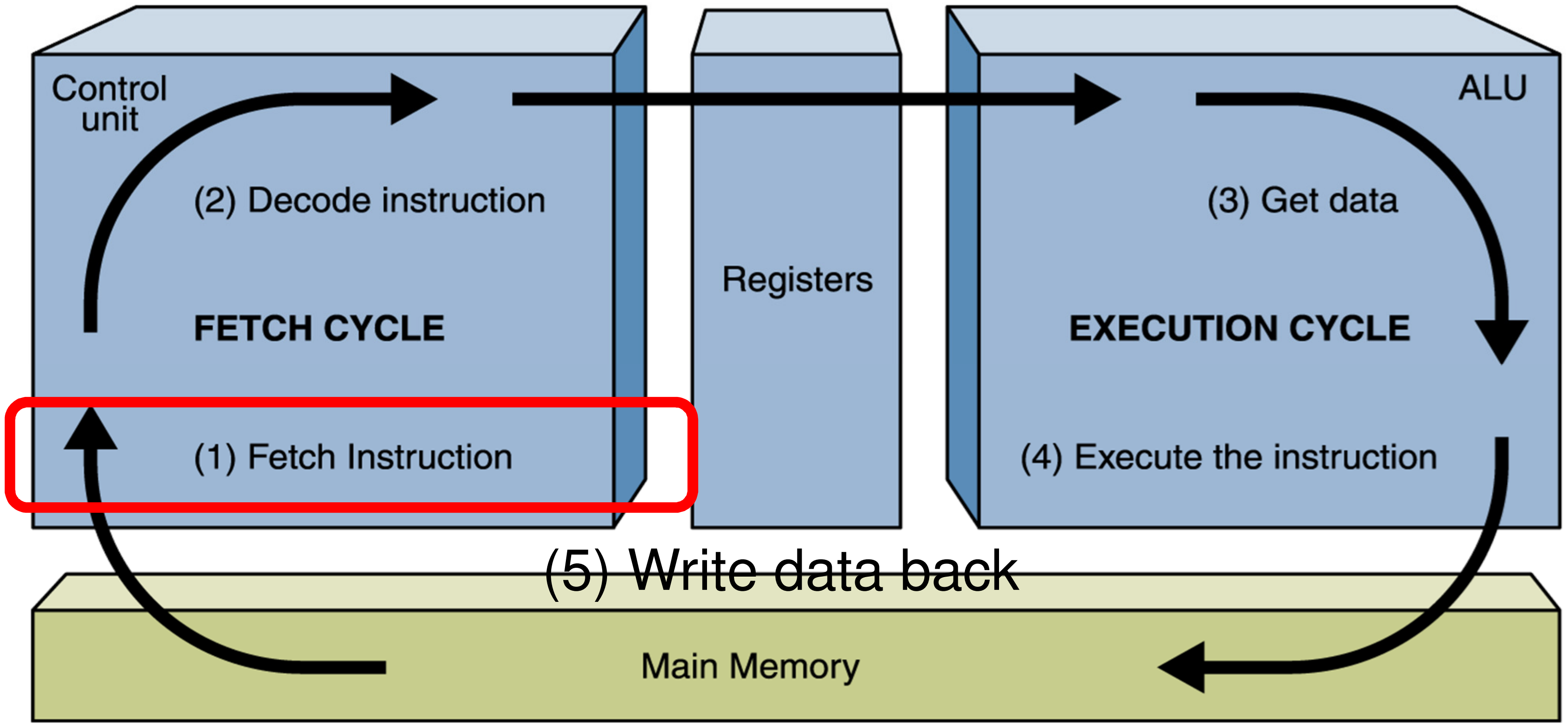
## Basic operation of a machine



## Basic operation of a machine



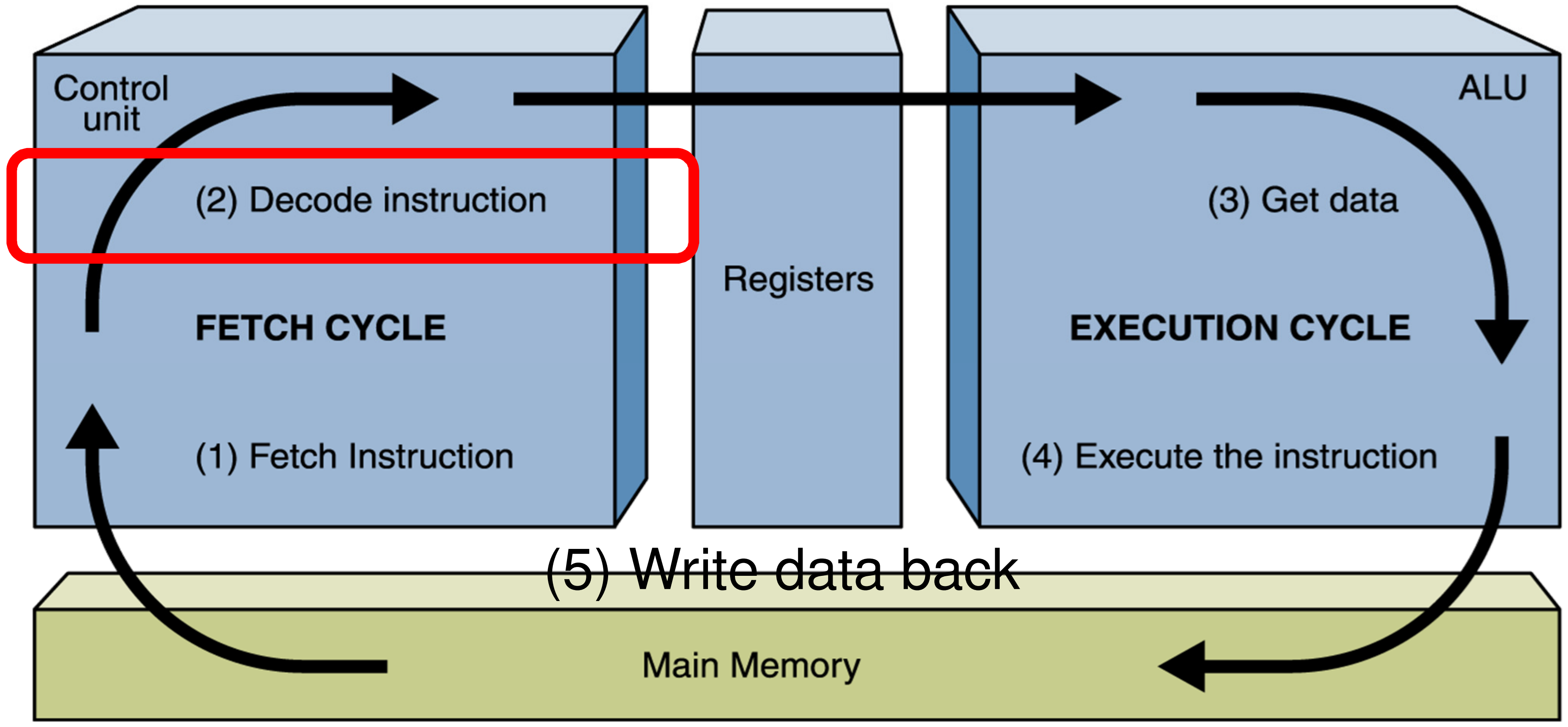
## Basic operation of a machine



**FETCH**

The address of the next instruction is read from the instruction counter

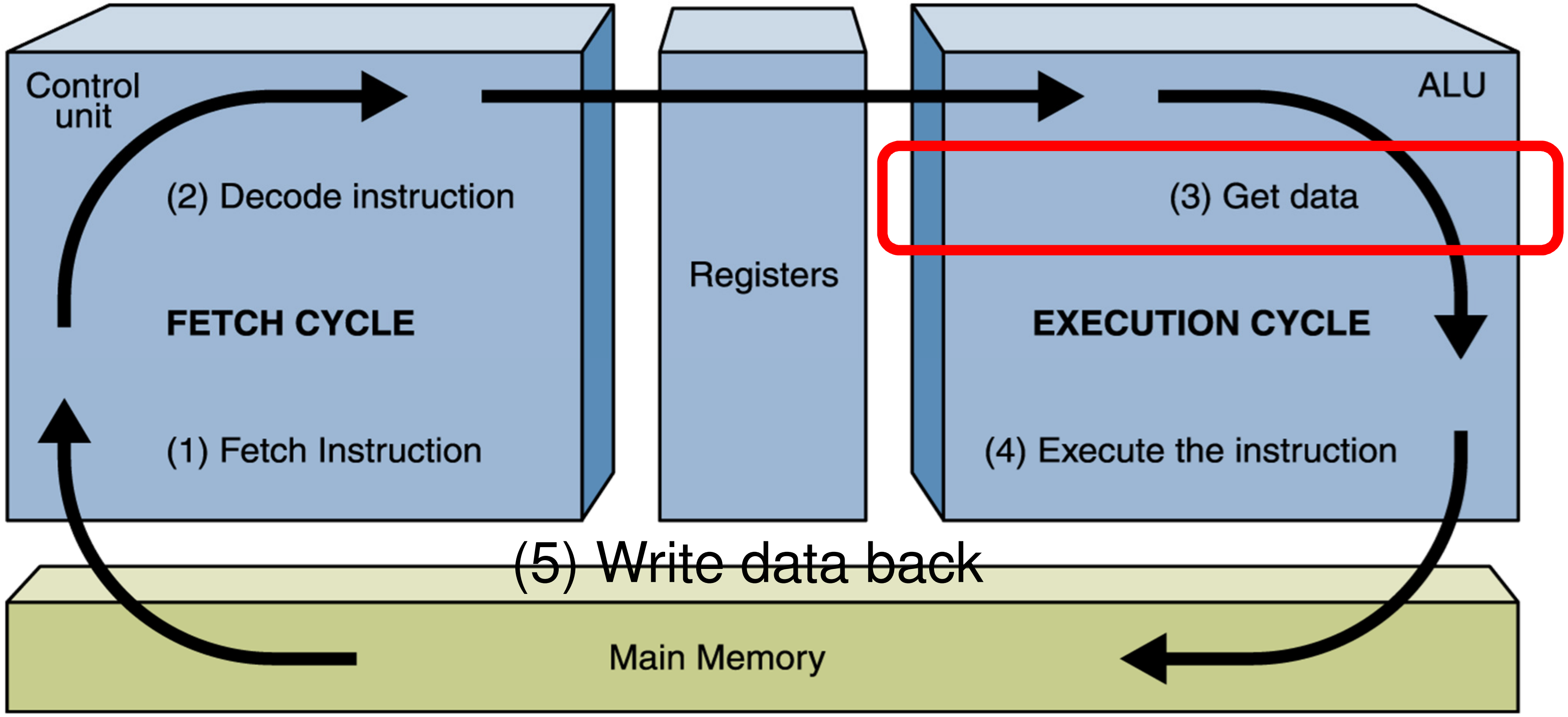
# Basic operation of a machine



**DECODE**

The instruction is examined to determine what operation to perform and which operands to operate on.

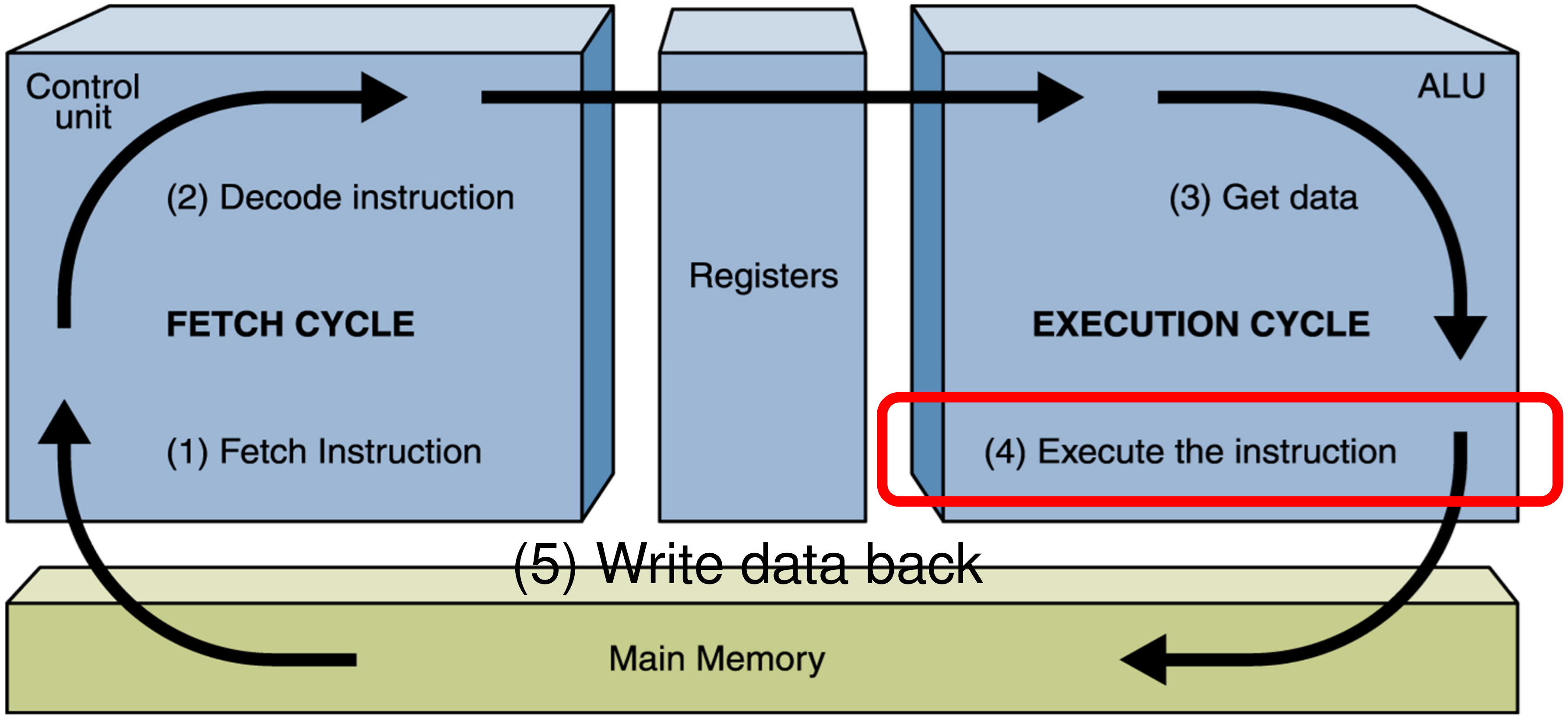
## Basic operation of a machine



**GET DATA**

Depending on the instruction operands, they may need to be fetched from a memory location (e.g., address, pointer).

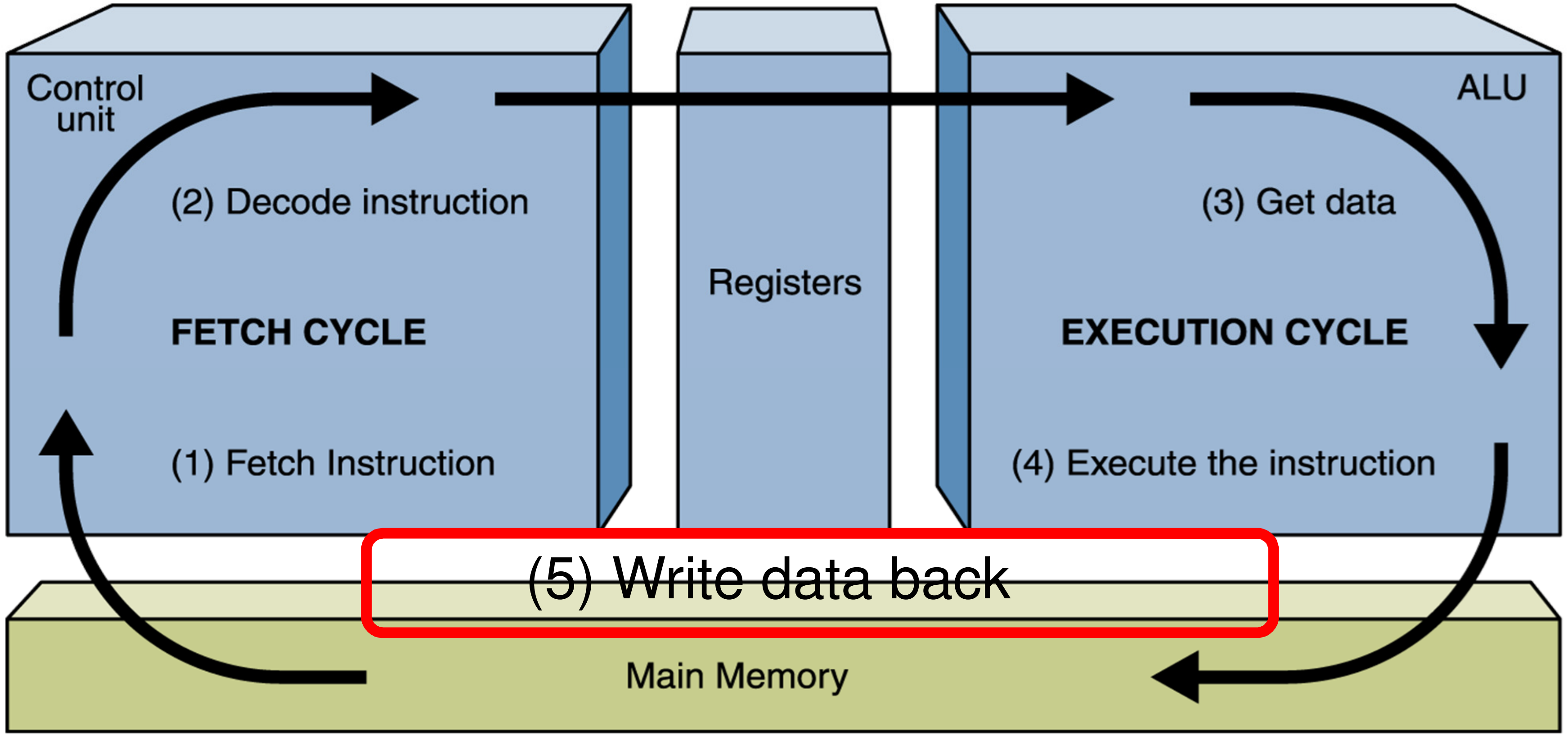
# Basic operation of a machine



**EXECUTE**

The ALU performs the operation and writes results to registers or memory depending on the instruction.

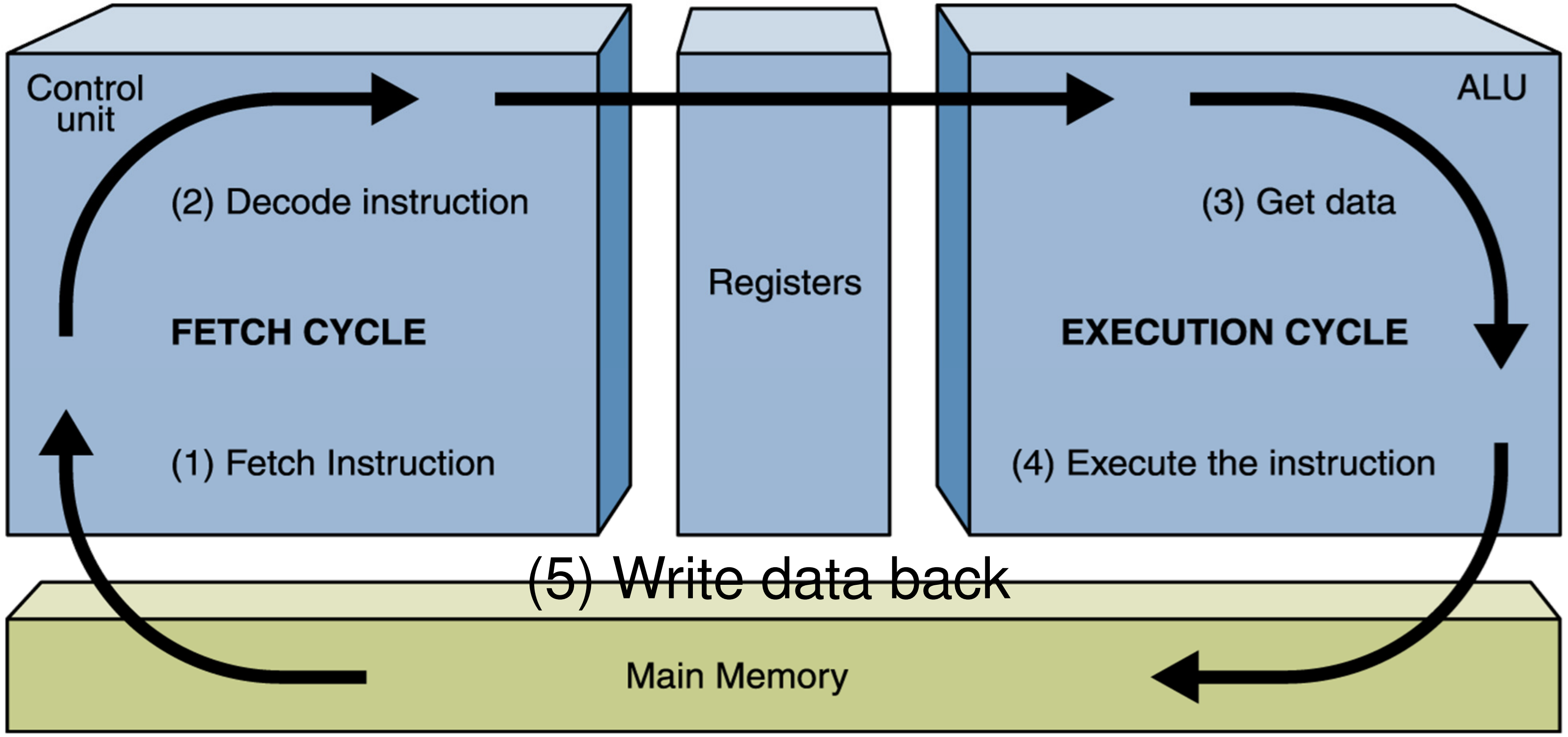
# Basic operation of a machine



**Write back**

Write results into register/memory

## Basic operation of a machine

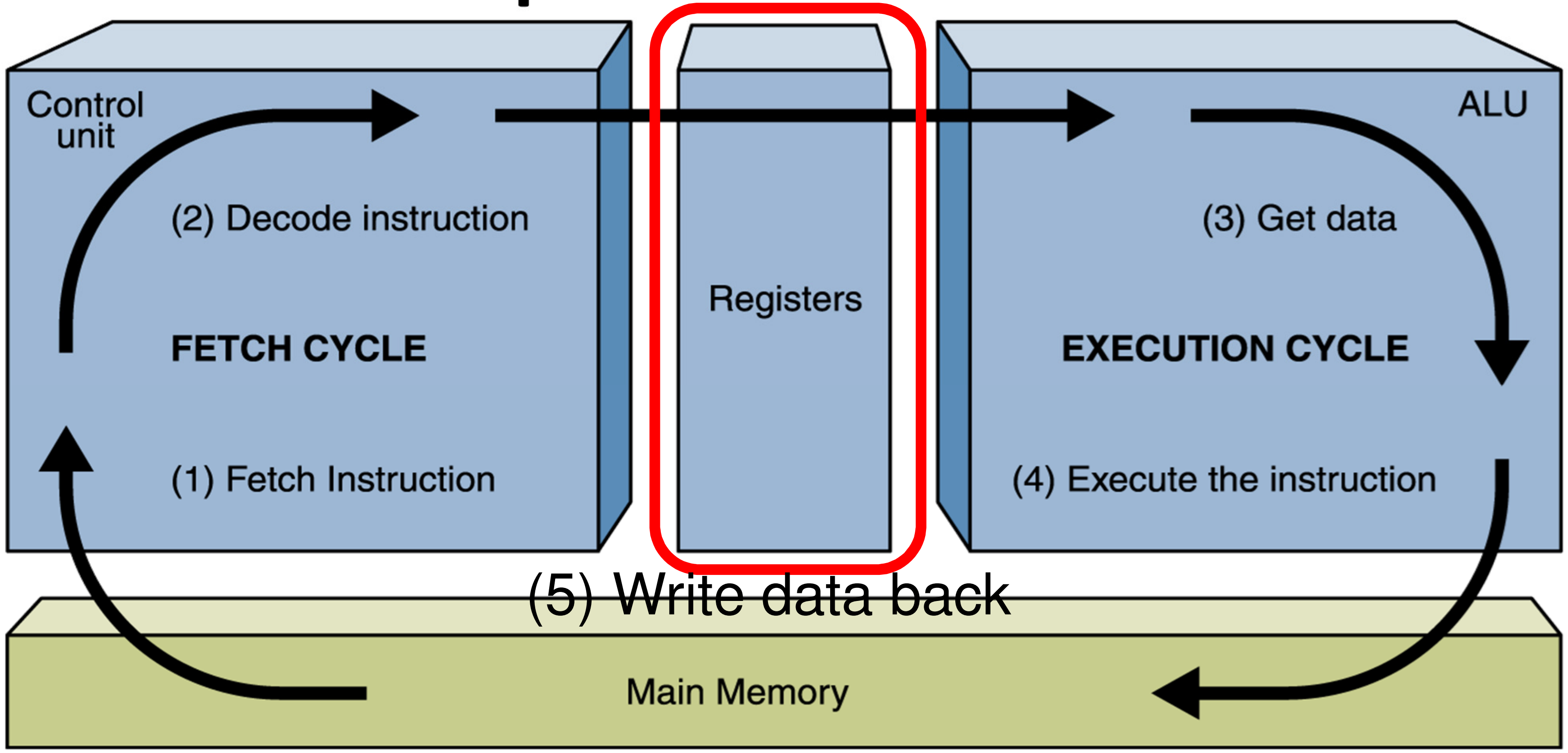


**UPDATE INSTRUCTION COUNTER**

The instruction counter is incremented for the next cycle.

**REPEAT**

## Basic operation of a machine



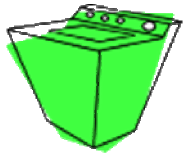
**REGISTERS**

Closest memory locations to the execution core!

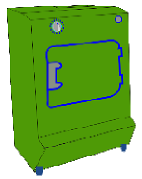
## Objectives

- **Machine Structure**
  - Understand the structure of a machine.
  - Learn about the *central processing unit* (CPU)
  - Learn about the internals of the CPU
- **Machine Execution**
  - Understand the basics of machine execution.
  - Understand how a program represented and interpreted "under-the-hood"
  - *How does a machine execute a program?*
- **Machine Pipelines**
  - Learn how machine's exploit parallelism to improve performance
  - Understand the basics of *pipelines*
  - Understand how instructions are executed by a pipeline

## Laundry

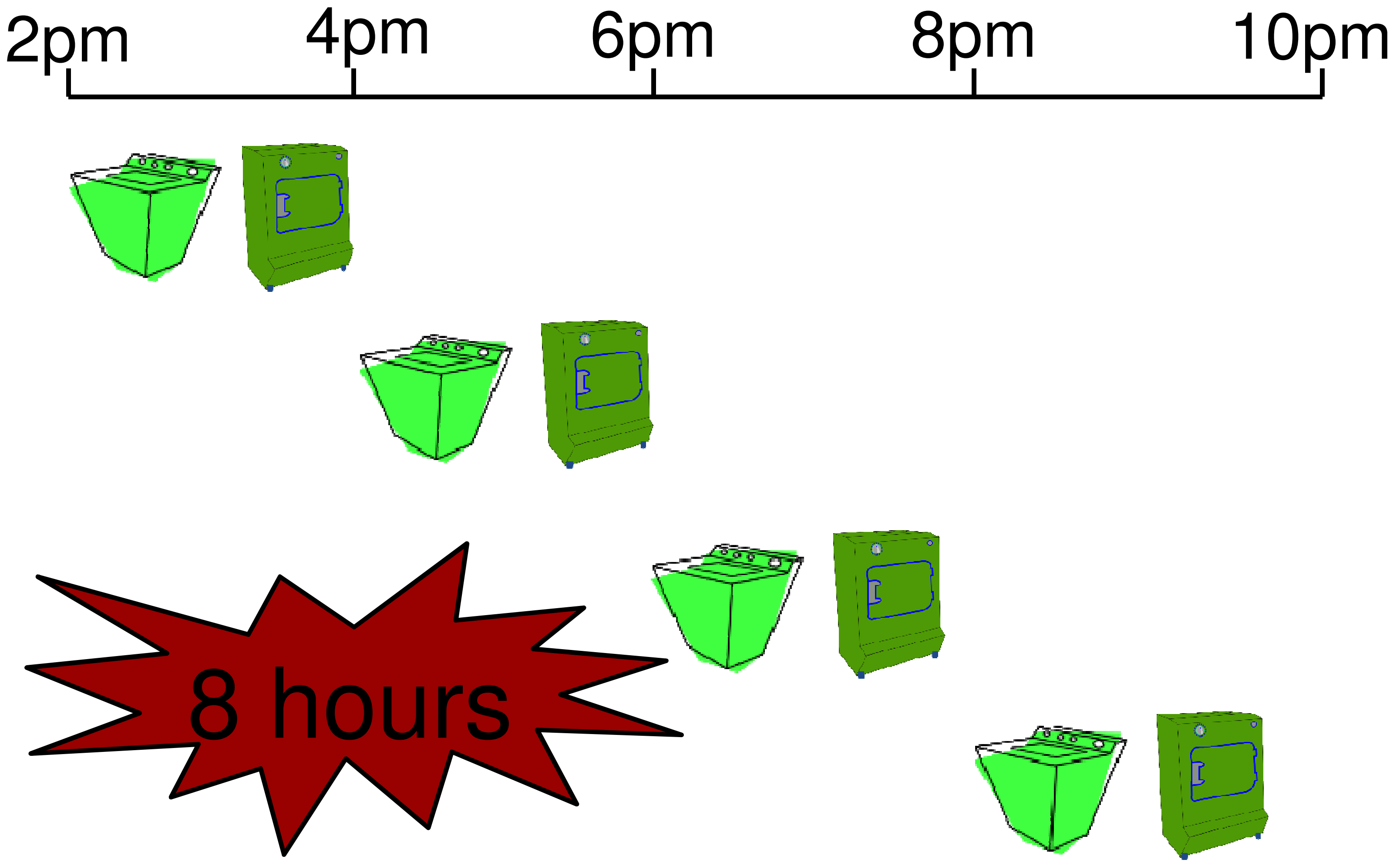


1 hour

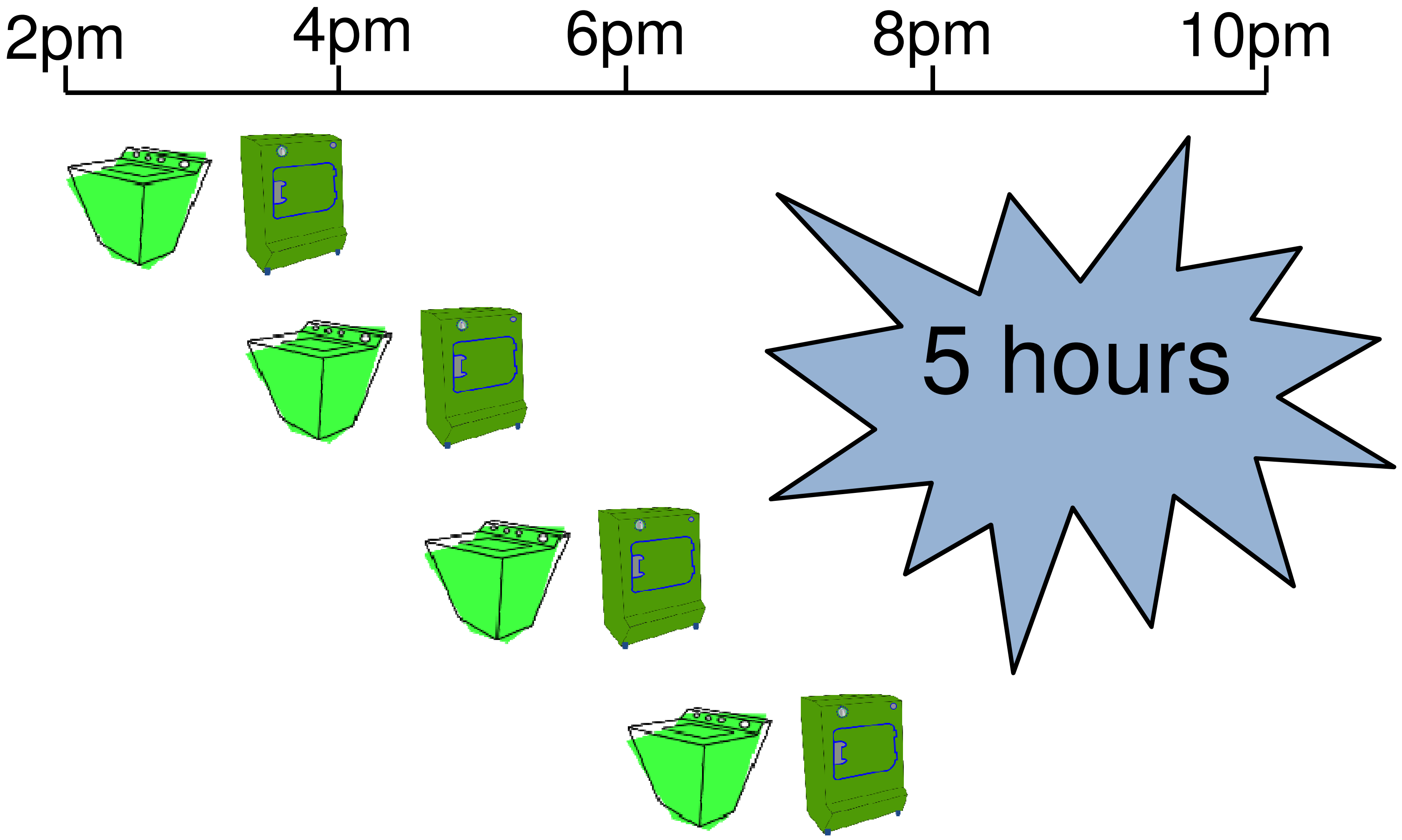


1 hour

## Unpipelined laundry



## Pipelined laundry

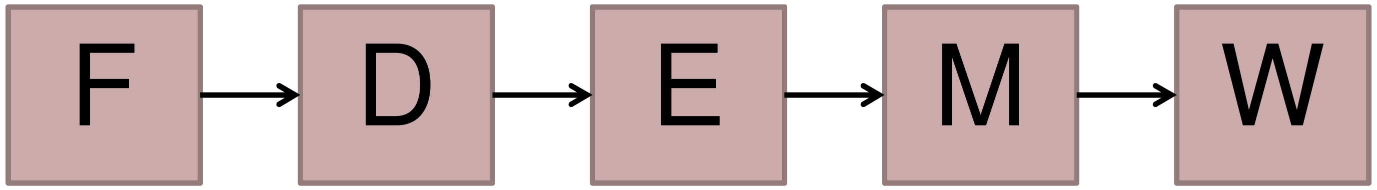


## Unpipelined Architectures

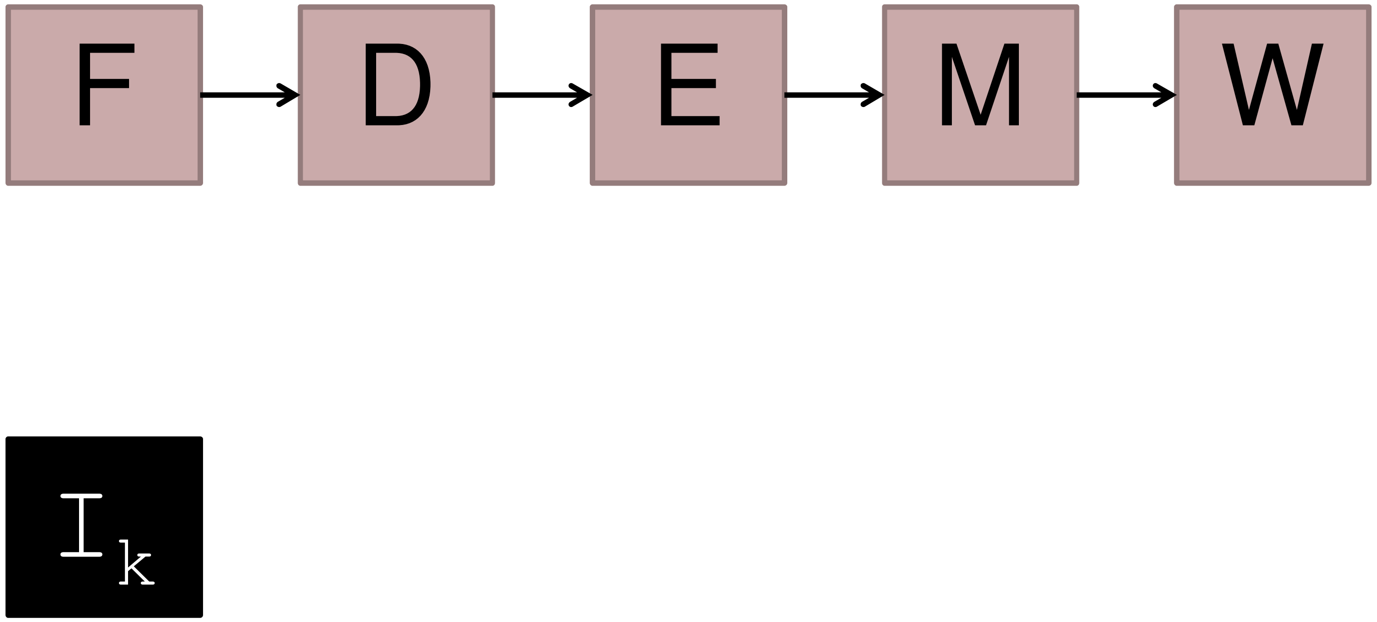
***Straight up sequential execution of instructions***

- **Fetch instruction (F)**
  - Machine fetches/copies the instruction from memory
- **Decode instruction (D)**
  - Machine decodes the instruction to understand what the instruction is and how it should be executed
- **Execute instruction (E)**
  - Execute the instruction
- **Access memory (M)**
  - Read values from memory if instruction has operands
- **Write back results back (W)**
  - Write any results generated by the instruction back to memory/register

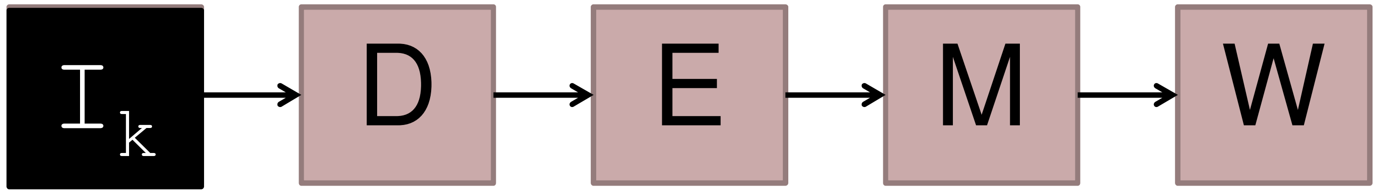
## Unpipelined Execution



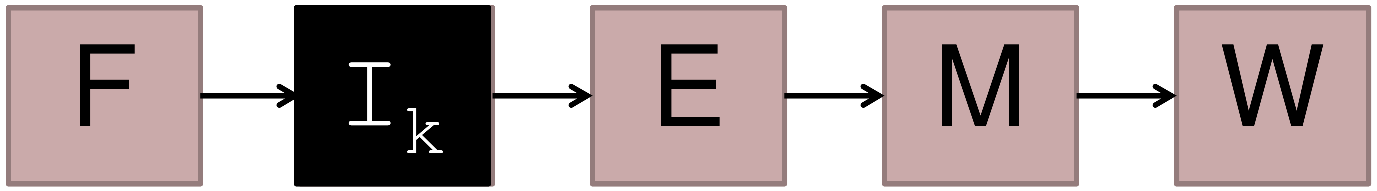
## Unpipelined Execution



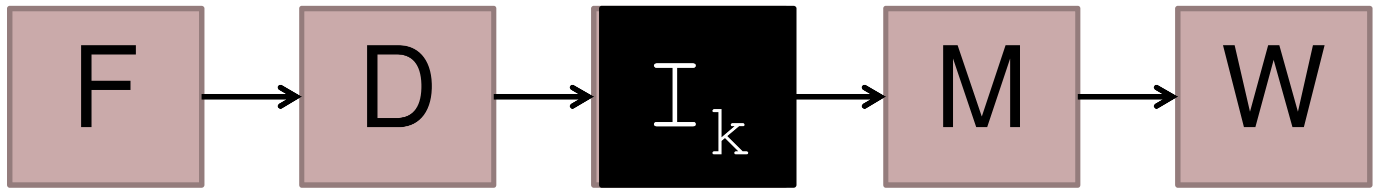
## Unpipelined Execution



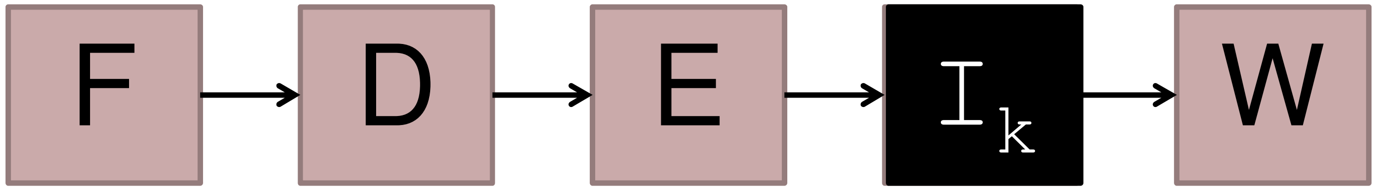
## Unpipelined Execution



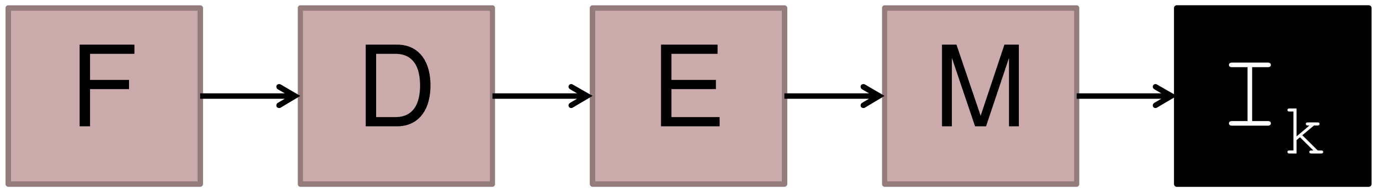
## Unpipelined Execution



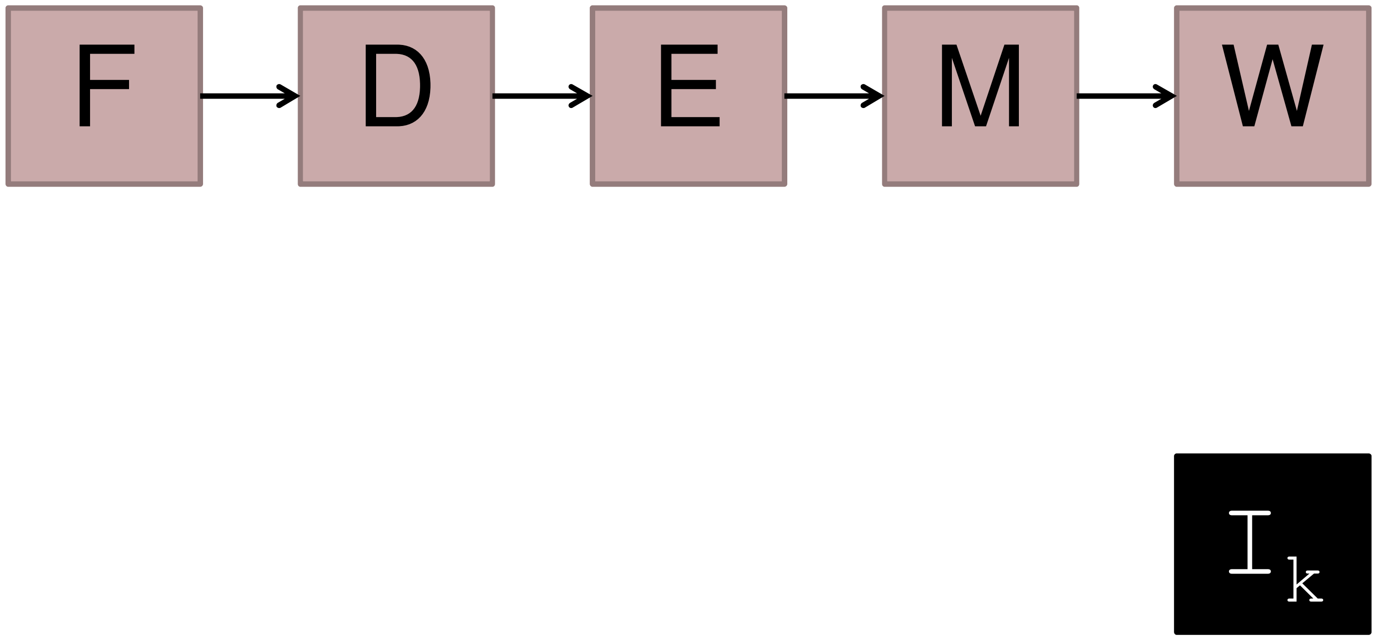
## Unpipelined Execution



## Unpipelined Execution

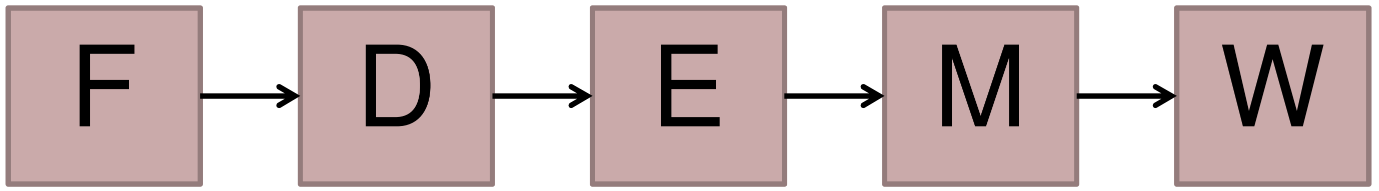


## Unpipelined Execution



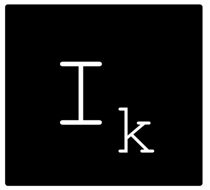
The instruction is then “retired”

## Unpipelined Execution



No great from a time perspective!

Good from a perspective of simplicity!



The instruction is then “retired”

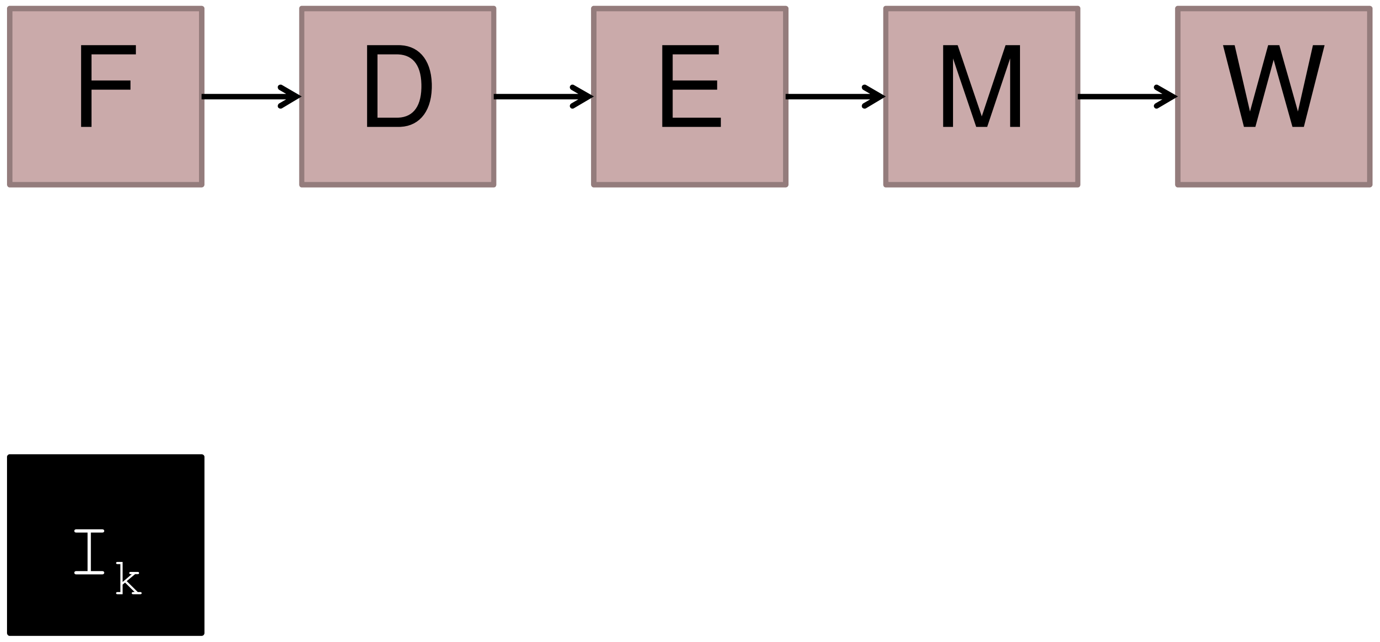
## Pipelined Architectures

***Use a "pipeline" to execute instructions in parallel***

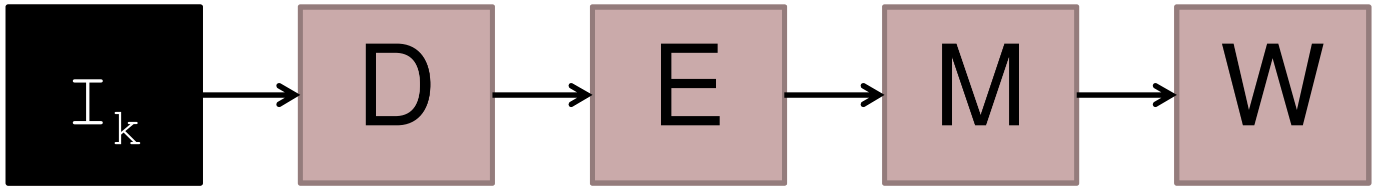
**Rather then deal with a single instruction at a time – pump instructions into the "pipeline" to allow multiple *instructions in flight*. That is, as soon as a stage is empty (e.g., fetch), fill it with a new instruction.**

**This allows *instruction-level parallelism* to be achieved. This will keep the machine busy in all stages as much as possible.**

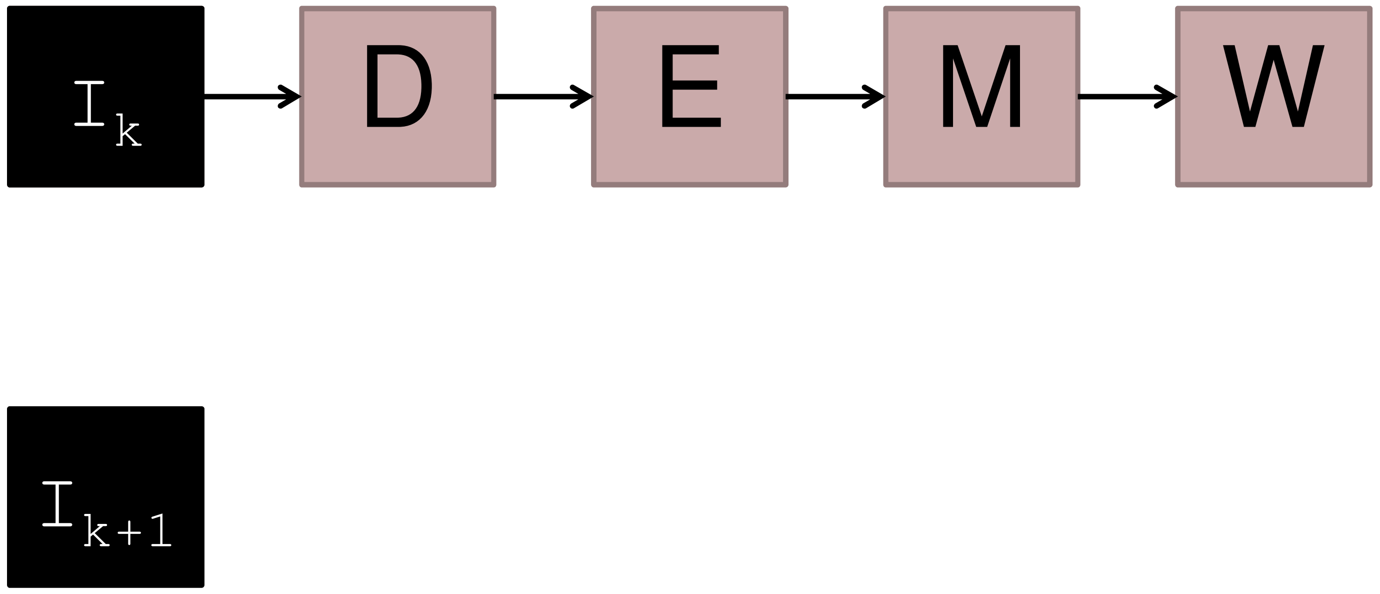
## Pipelined Execution



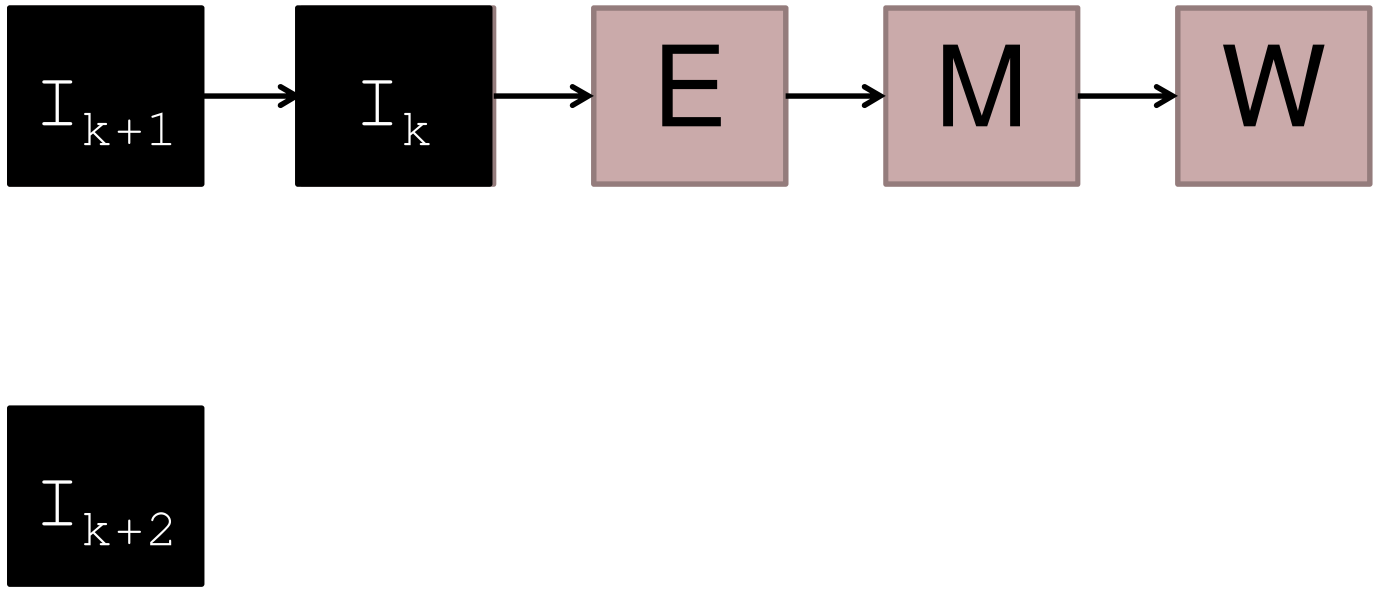
## Pipelined Execution



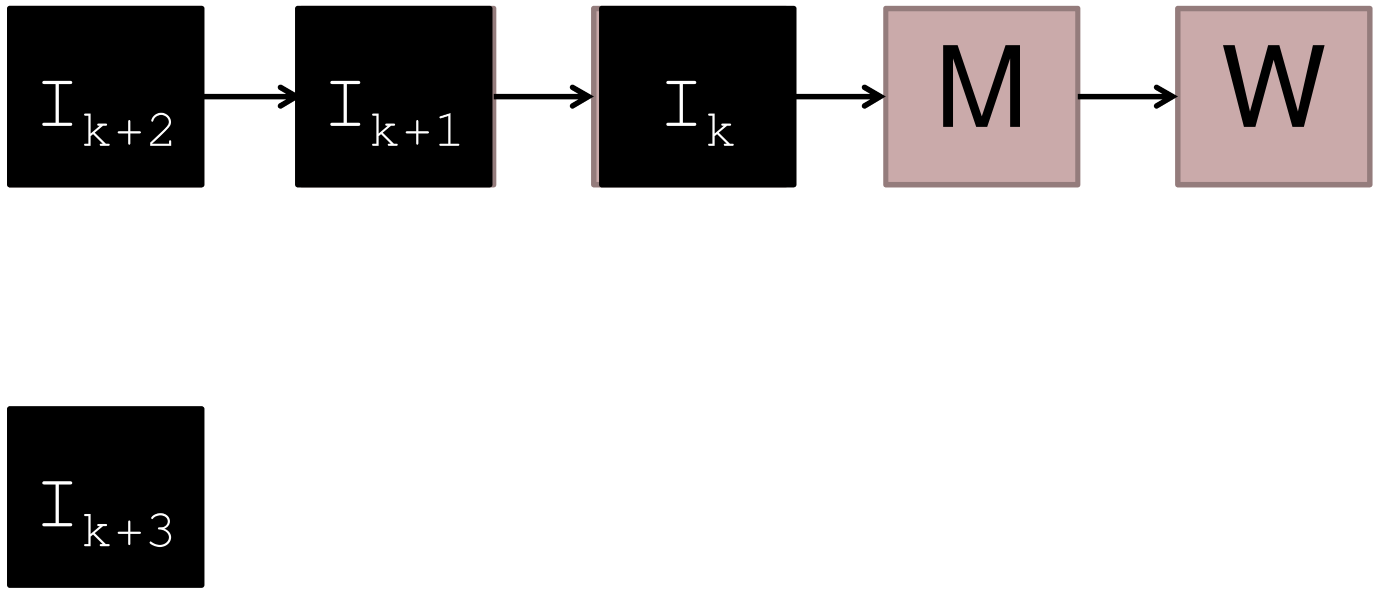
## Pipelined Execution



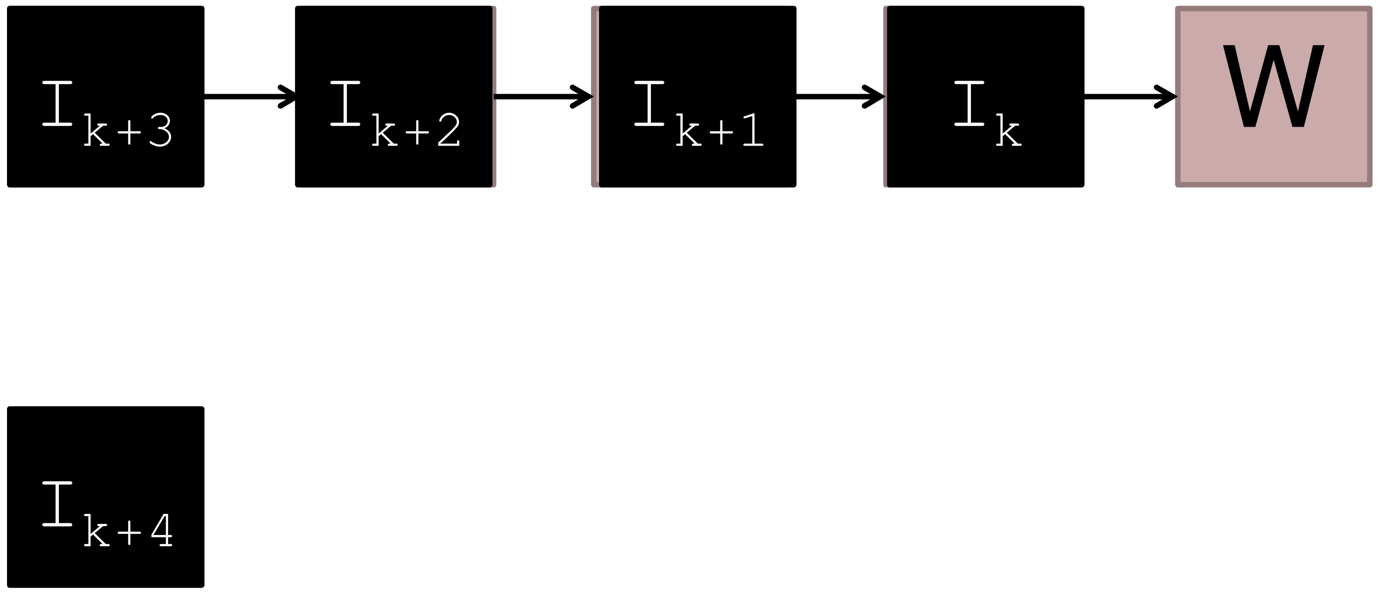
## Pipelined Execution



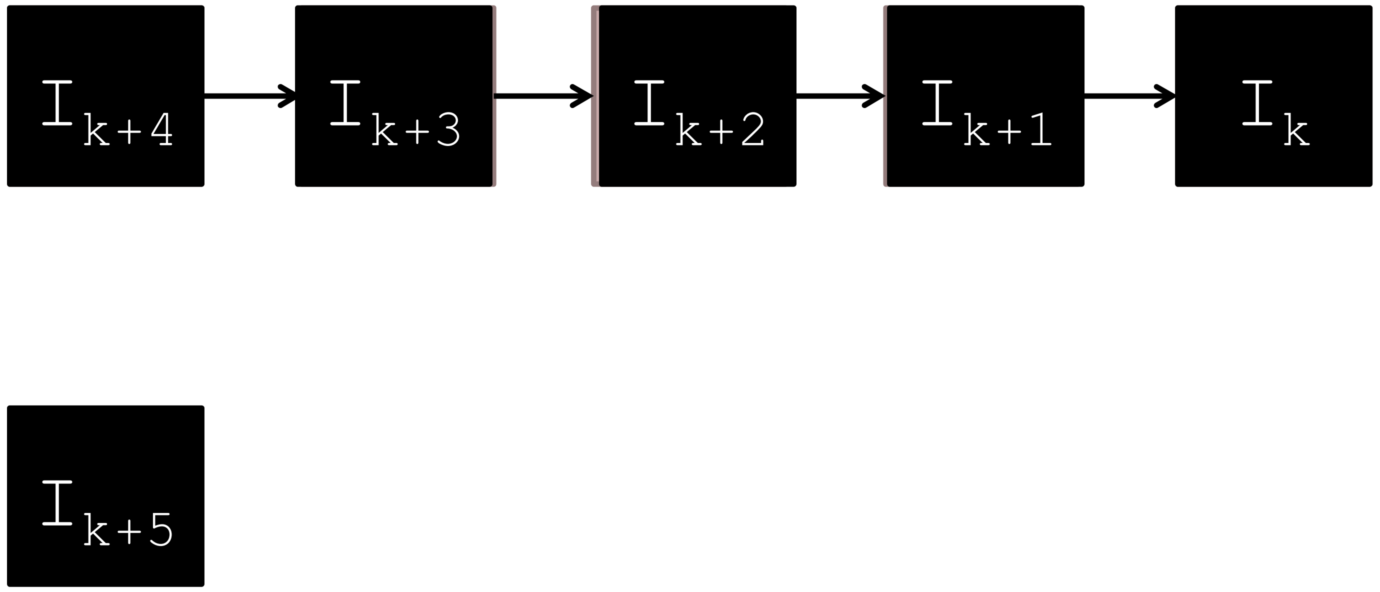
## Pipelined Execution



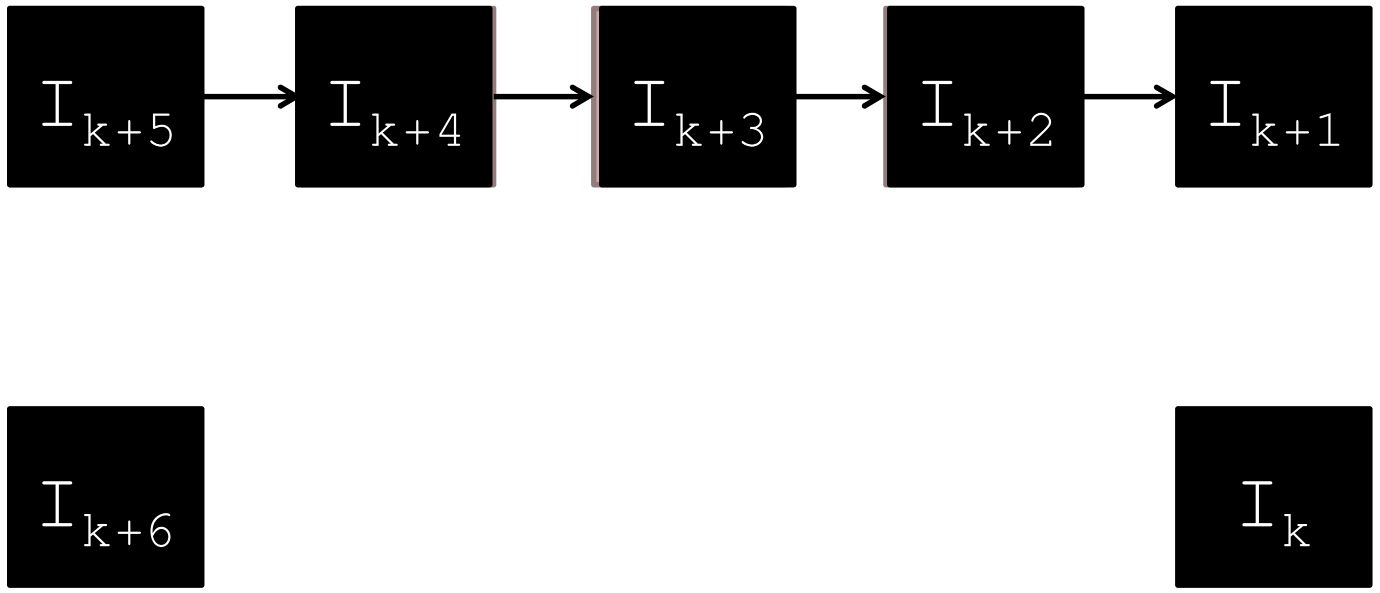
## Pipelined Execution



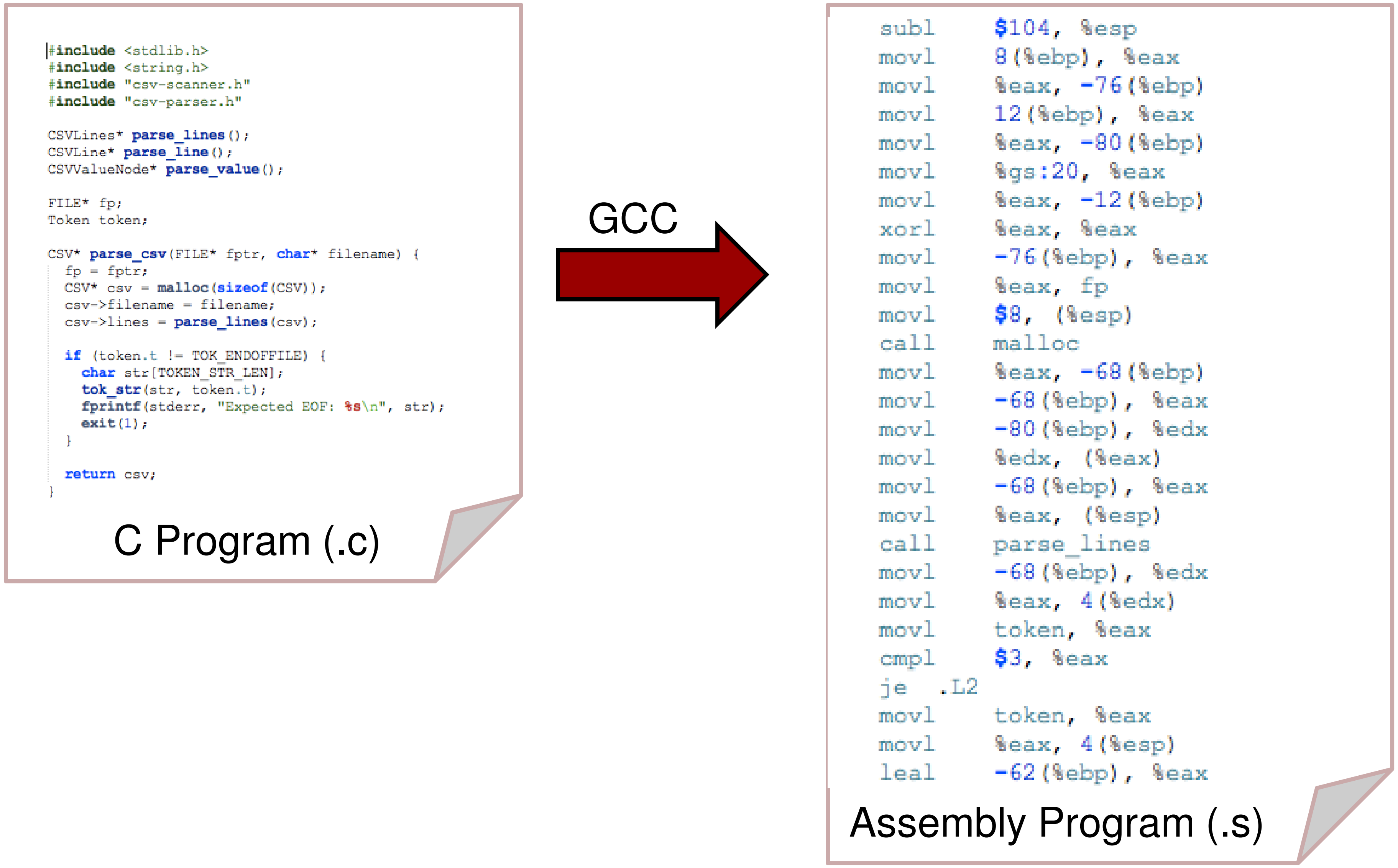
## Pipelined Execution



# Pipelined Execution



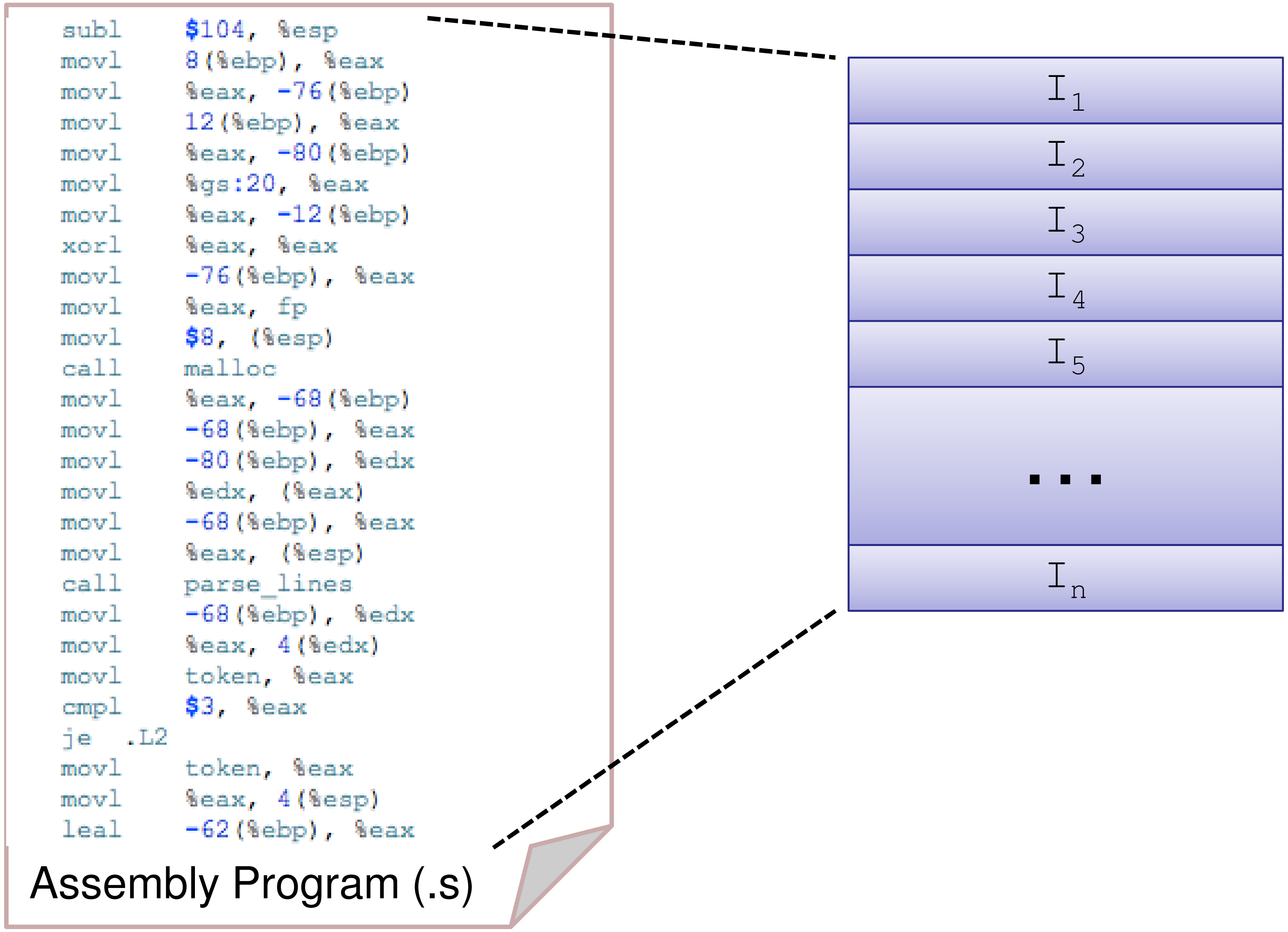
# Program Compilation and Assembly



```
#include <stdlib.h>
#include <string.h>
#include "csv-scanner.h"
#include "csv-parser.h"

CSVLines* parse_lines();
CSVLine* parse_line();
CSVValueNode* parse_value();

FILE* fp;
Token token;

CSV* parse_csv(FILE* fptr, char* filename) {
  fp = fptr;
  CSV* csv = malloc(sizeof(CSV));
  csv->filename = filename;
  csv->lines = parse_lines(csv);

  if (token.t != TOK_ENDOFFILE) {
    char str[TOKEN_STR_LEN];
    tok_str(str, token.t);
    fprintf(stderr, "Expected EOF: %s\n", str);
    exit(1);
  }

  return csv;
}
```

```
subl    $104, %esp
movl    8(%ebp), %eax
movl    %eax, -76(%ebp)
movl    12(%ebp), %eax
movl    %eax, -80(%ebp)
movl    %gs:20, %eax
movl    %eax, -12(%ebp)
xorl    %eax, %eax
movl    -76(%ebp), %eax
movl    %eax, fp
movl    $8, (%esp)
call    malloc
movl    %eax, -68(%ebp)
movl    -68(%ebp), %eax
movl    -80(%ebp), %edx
movl    %edx, (%eax)
movl    -68(%ebp), %eax
movl    %eax, (%esp)
call    parse_lines
movl    -68(%ebp), %edx
movl    %eax, 4(%edx)
movl    token, %eax
cmpl    $3, %eax
je  .L2
movl    token, %eax
movl    %eax, 4(%esp)
leal    -62(%ebp), %eax
```

# Assembly and Instructions



```
    subl    $104, %esp
    movl    8(%ebp), %eax
    movl    %eax, -76(%ebp)
    movl    12(%ebp), %eax
    movl    %eax, -80(%ebp)
    movl    %gs:20, %eax
    movl    %eax, -12(%ebp)
    xorl    %eax, %eax
    movl    -76(%ebp), %eax
    movl    %eax, fp
    movl    $8, (%esp)
    call    malloc
    movl    %eax, -68(%ebp)
    movl    -68(%ebp), %eax
    movl    -80(%ebp), %edx
    movl    %edx, (%eax)
    movl    -68(%ebp), %eax
    movl    %eax, (%esp)
    call    parse_lines
    movl    -68(%ebp), %edx
    movl    %eax, 4(%edx)
    movl    token, %eax
    cmpl    $3, %eax
    je  .L2
    movl    token, %eax
    movl    %eax, 4(%esp)
    leal    -62(%ebp), %eax
```

Assembly Program (.s)

## Assembly Instruction

- ADD S, D := D<-D+S

## Assembly Instruction

- ADD S, D := D<-D+S
    - C syntax: D += S

## Assembly Instruction

- ADD S, D := D<-D+S
  - C syntax: D += S
  - Example: ADD R1, R2

## Assembly Instruction

- ADD S, D := D<-D+S
  - C syntax: D += S
  - Example: ADD R1, R2
  - Example: ADD $10, R1

## Assembly Instruction

- ADD S, D := D<-D+S
  - C syntax: D += S
  - Example: ADD R1, R2
  - Example: ADD $10, R1
- MOV S, D := D ←S

# Assembly Instruction

- ADD S, D := D<-D+S
  - C syntax: D += S
  - Example: ADD R1, R2
  - Example: ADD $10, R1
- MOV S, D := D $\leftarrow$S
  - Example: MOV M1, R1
  - Example: MOV R2, M2

## Assembly Instruction

- ADD S, D := D<-D+S
  - C syntax: D += S
  - Example: ADD R1, R2
  - Example: ADD $10, R1
- MOV S, D := D ←S
  - Example: MOV M1, R1
  - Example: MOV R2, M2
  - Example: MOV $10, R1

# Assembly Instruction

- ADD S, D := D<-D+S
  - C syntax: D += S
  - Example: ADD R1, R2
  - Example: ADD $10, R1
- MOV S, D := D ←S
  - Example: MOV M1, R1
  - Example: MOV R2, M2
  - Example: MOV $10, R1
- JMP Label

## Assembly Instruction

- ADD S, D := D<-D+S
  - C syntax: D += S
  - Example: ADD R1, R2
  - Example: ADD $10, R1
- MOV S, D := D ←S
  - Example: MOV M1, R1
  - Example: MOV R2, M2
  - Example: MOV $10, R1
- JMP Label
- JE Label, JLE Label, JNE Label, ….

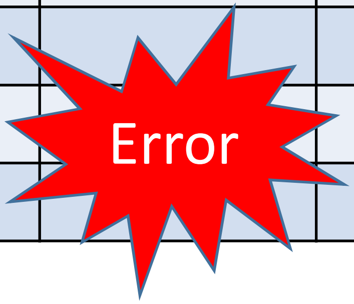
| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

|  | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 |  |  |  |  |  |
| t2 |  |  |  |  |  |
| t3 |  |  |  |  |  |
| t4 |  |  |  |  |  |
| t5 |  |  |  |  |  |

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | | | MOV M1, R1 | | |
| t2 | | | | | |
| t3 | | | | | |
| t4 | | | | | |
| t5 | | | | | |

|  | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | ADD R1, R2 | ADD  $12, R1 | MOV M1, R1 |  |  |
| t2 |  |  |  |  |  |
| t3 |  |  |  |  |  |
| t4 |  |  |  |  |  |
| t5 |  |  |  |  |  |

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | ADD R1, R2 | ADD  $12, R1 | MOV M1, R1 | | |
| t2 | … | ADD R1, R2 | ADD  $12, R1 | MOV M1, R1 | |
| t3 | | | | | |
| t4 | | | | | |
| t5 | | | | | |



## Data hazard

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | ADD R1, R2 | ADD $12, R1 | MOV M1, R1 | | |
| t2 | … | ADD R1, R2 | ADD $12, R1 | MOV M1, R1 | |
| t3 | | | | | |
| t4 | | | | | |
| t5 | | | | | |

Error

When data dependencies have the potential to cause an erroneous computation by the pipeline, they are called data hazards.

## Avoid Data hazard by Stalling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | ADD R1, R2 | ADD $12, R1 | MOV M1, R1 | | |
| t2 | | | | | |
| t3 | | | | | |
| t4 | | | | | |
| t5 | | | | | |

Stalling: the processor holds back one more instructions in the pipeline until the hazard condition no longer holds.

## Avoid Data hazard by Stalling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| **t1** | **ADD R1, R2** | **ADD  $12, R1** | **MOV M1, R1** | | |
| **t2** | **ADD R1, R2** | **ADD  $12, R1** | **bubble** | **MOV M1, R1** | |
| **t3** | | | | | |
| **t4** | | | | | |
| **t5** | | | | | |

## Avoid Data hazard by Stalling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| **t1** | **ADD R1, R2** | **ADD $12, R1** | **MOV $5, R1** | | |
| **t2** | **ADD R1, R2** | **ADD $12, R1** | **bubble** | **MOV $5, R1** | |
| **t3** | | | | | |
| **t4** | | | | | |
| **t5** | | | | | |

bubble == nop instruction

## Avoid Data hazard by Stalling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| **t1** | **ADD R1, R2** | **ADD $12, R1** | **MOV M1, R1** | | |
| **t2** | **ADD R1, R2** | **ADD $12, R1** | **bubble** | **MOV M1, R1** | |
| **t3** | | | | | **MOV M1, R1** |
| **t4** | | | | | |
| **t5** | | | | | |

## Avoid Data hazard by Stalling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| **t1** | **ADD R1, R2** | **ADD $12, R1** | **MOV M1, R1** | | |
| **t2** | **ADD R1, R2** | **ADD $12, R1** | **bubble** | **MOV M1, R1** | |
| **t3** | | **ADD $12, R1** | **bubble** | **bubble** | **MOV M1, R1** |
| **t4** | | | | | |
| **t5** | | | | | |

## Avoid Data hazard by Stalling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | ADD R1, R2 | ADD  $12, R1 | MOV M1, R1 | | |
| t2 | ADD R1, R2 | ADD  $12, R1 | bubble | MOV M1, R1 | |
| t3 | ADD R1, R2 | ADD  $12, R1 | bubble | bubble | MOV M1, R1 |
| t4 | ADD R1, R2 | ADD  $12, R1 | bubble | bubble | bubble |
| t5 | | | | | |

# Avoid Data hazard by Stalling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | ADD R1, R2 | ADD  $12, R1 | MOV M1, R1 | | |
| t2 | ADD R1, R2 | ADD  $12, R1 | bubble | MOV M1, R1 | |
| t3 | ADD R1, R2 | ADD  $12, R1 | bubble | bubble | MOV M1, R1 |
| t4 | ADD R1, R2 | ADD  $12, R1 | bubble | bubble | bubble |
| t5 | MOV  M2, R1 | ADD R1, R2 | ADD  $12, R1 | bubble | bubble |

## Avoid Data hazard by Forwarding

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | ADD R1, R2 | ADD $12, R1 | MOV M1, R1 | | |
| t2 | | | | | |
| t3 | | | | | |
| t4 | | | | | |
| t5 | | | | | |

Forwarding: passing a result value directly from one pipeline stage to an earlier one

## Avoid Data hazard by Forwarding

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | ADD R1, R2 | ADD $12, R1 | MOV M1, R1 | | |
| t2 | | | | | |
| t3 | | | | | |
| t4 | | | | | |
| t5 | | | | | |

The ADD has read the wrong register value for R1.

Forwarding: passing a result value directly from one pipeline stage to an earlier one

## Avoid Data hazard by Forwarding

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | ADD R1, R2 | ADD  $12, R1 | MOV M1, R1 | | |
| t2 | MOV  M2, R1 | ADD R1, R2 | ADD  $12, R1 | MOV M1, R1 | |
| t3 | | | | | |
| t4 | | | | | |
| t5 | | | | | |

Forwarding: passing a result value directly from one pipeline stage to an earlier one

## Avoid Data hazard by Forwarding

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | ADD R1, R2 | ADD $12, R1 | MOV M1, R1 | | |
| t2 | MOV M2, R1 | ADD R1, R2 | ADD $12, R1 | MOV M1, R1 | |
| t3 | | | | | |
| t4 | | | | | |
| t5 | | | | | |

Forwarding: fix what we have read

## i-clicker question

- Which of following programs' data hazards cannot be dealt with forwarding?

A.

```
mov $10, r1
mov $3, r2
nop
nop
add r1, r2
```

B.

```
mov $10, r1
mov $3, r2
nop
add r1, r2
```

C.

```
mov $10, r1
mov $3, r2
add r1, r2
```

D.

```
mov M1, r1
add $2, r1
```

## i-clicker question

- Which of following programs' data hazards cannot be dealt with forwarding? Sol: D

A.

```
mov $10, r1
mov $3, r2
nop
nop
add r1, r2
```

B.

```
mov $10, r1
mov $3, r2
nop
add r1, r2
```

C.

```
mov $10, r1
mov $3, r2
add r1, r2
```

D.

```
mov M1, r1
add $2, r1
```

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | JMP Label | | | | |
| t2 | | | | | |
| t3 | | | | | |
| t4 | | | | | |
| t5 | | | | | |

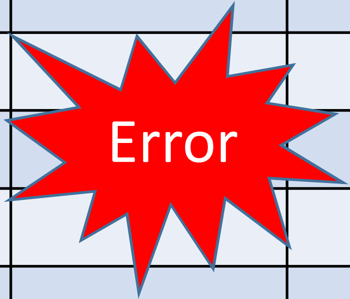
| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | JMP Label | | | | |
| t2 | ADD $12, R1 | JMP Label | | | |
| t3 | | | | | |
| t4 | | | | | |
| t5 | | | | | |

|  | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| **t1** | JMP Label |  |  |  |  |
| **t2** | ADD  $12, R1 | JMP Label |  |  |  |
| **t3** | ADD R1, R2 | ADD  $12, R1 | JMP Label |  |  |
| **t4** |  |  |  |  |  |
| **t5** |  |  |  |  |  |

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | JMP Label | | | | |
| t2 | ADD $12, R1 | JMP Label | | | |
| t3 | ADD R1, R2 | ADD $12, R1 | JMP Label | Error | |
| t4 | | | | | |
| t5 | | | | | |

## Control Hazard

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | JMP Label | | | | |
| t2 | ADD  $12, R1 | JMP Label | | | |
| t3 | ADD R1, R2 | ADD  $12, R1 | JMP Label | | |
| t4 | | | | | |
| t5 | | | | | |



When control dependencies have the potential to cause an erroneous computation by the pipeline, they are called control hazards.

## Avoid Control Hazard by stalling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| **t1** | **JMP Label** | | | | |
| **t2** | **bubble** | **JMP Label** | | | |
| **t3** | **bubble** | **bubble** | **JMP Label** | | |
| **t4** | | | | | |
| **t5** | | | | | |

## Avoid control hazards by cancelling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| t1 | JMP Label | | | | |
| t2 | ADD  $12, R1 | JMP Label | | | |
| t3 | ADD R1, R2 | ADD  $12, R1 | JMP Label | | |
| t4 | | | | | |
| t5 | | | | | |

## Avoid control hazards by cancelling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| **t1** | **JMP Label** | | | | |
| **t2** | **ADD  $12, R1** | **JMP Label** | | | |
| **t3** | **bubble** | **bubble** | **JMP Label** | | |
| **t4** | | | | | |
| **t5** | | | | | |

## Avoid control hazards by cancelling

| | Fetch | Decode | Execute | Memory | Write back |
|---|---|---|---|---|---|
| **t1** | **JMP Label** | | | | |
| **t2** | **ADD $12, R1** | **JMP Label** | | | |
| **t3** | **bubble** | **bubble** | **JMP Label** | | |
| **t4** | **ADD R1, R2** | **bubble** | **bubble** | **JMP Label** | |
| **t5** | | | | | |