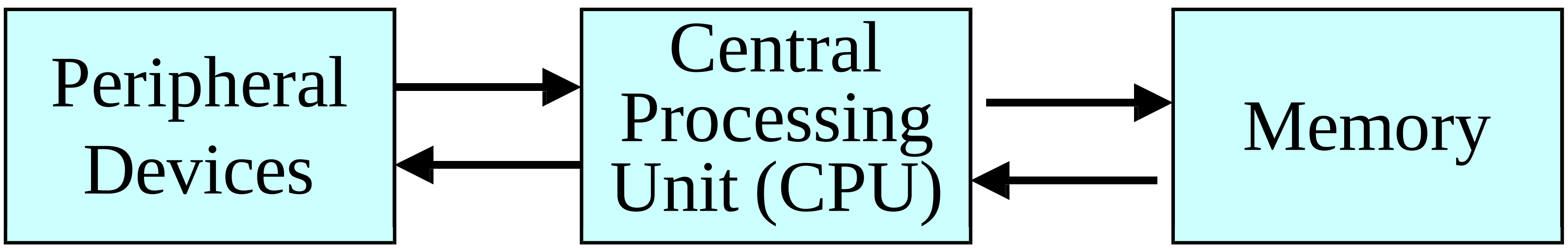
# IT 1206
# Section 5.0

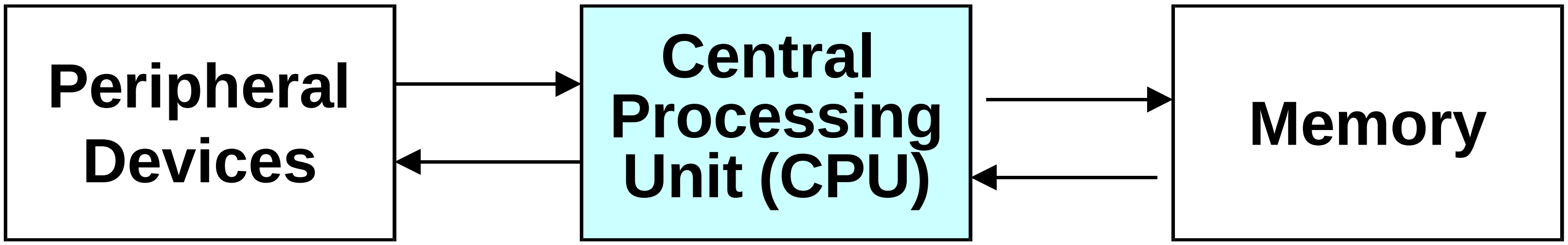
## CPU Organization and Instruction Set Architecture (ISA)

# Hardware Components of a Typical Computer




**Buses allow components to pass data to each other**

# Hardware Components of a Typical Computer - CPU




**Central Processing Unit (CPU)**

- **Performs the basic operations**
- **Consists of two parts:**
  - Arithmetic / Logic Unit (ALU) - data manipulation
  - Control Unit - coordinate machine’s activities

# Central Processing Unit (CPU)

- **Fetches, decodes and executes program instructions**
- **Two principal parts of the CPU**
  - **Arithmetic-Logic Unit (ALU)**
    - Connected to **registers** and **memory** by a data bus
    - All three comprise the **Datapath**
  - **Control unit**
    - Sends signals to CPU components to perform sequenced operations

UCSC

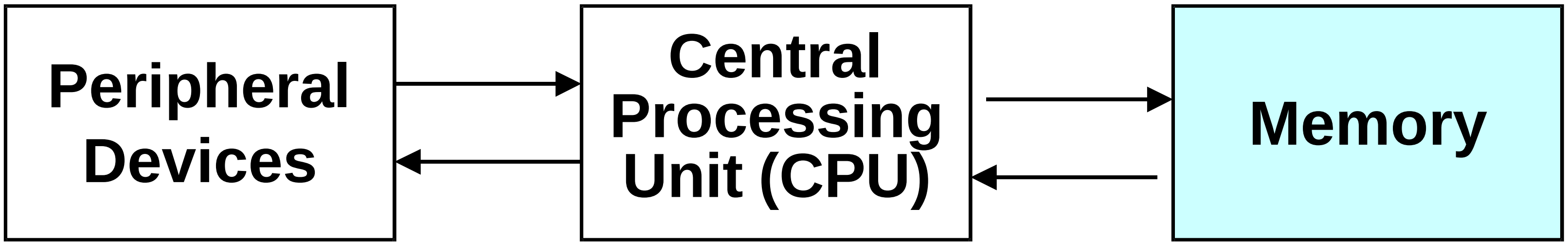


BIT

# CPU: Registers, ALU and Control Unit

- **Registers**
  - **Hold data that can be readily accessed by the CPU**
  - **Implemented using D flip-flops**
    - A 32-bit register requires 32 D flip-flops
- **Arithmetic-logic unit (ALU)**
  - **Carries out logical and arithmetic operations**
  - **Often affects the status register (e.g., overflow, carry)**
  - **Operations are controlled by the control unit**
- **Control unit (CU)**
  - **Policeman or traffic manager**
  - **Determines which actions to carry out according to the values in a program counter register and a status register**

# Hardware Components of a Typical Computer - Memory

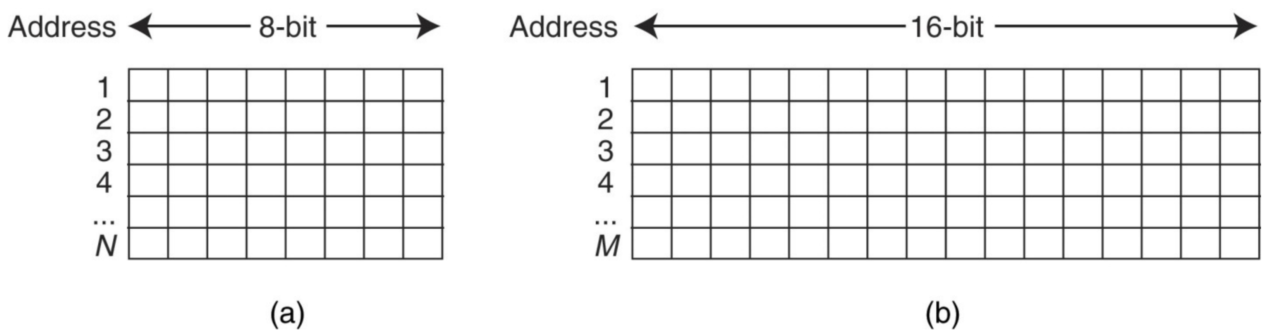



## Main Memory

- Holds programs and data
- Stores bits in fixed-sized chunks: “word” (8, 16, 32 or 64 bits)
- Each word has a unique address
- The words can be accessed in any order € random-access memory or “RAM”

# Memory

- **Consists of a linear array of addressable storage cells**

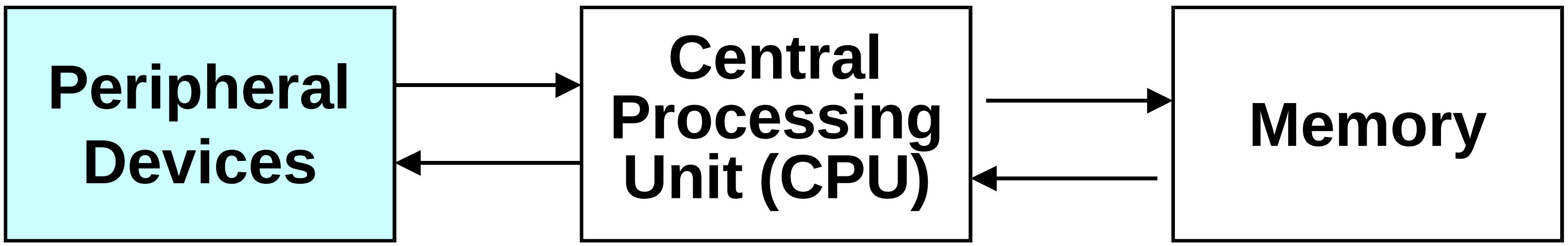



- **A memory address is represented by an unsigned integer**
- **Can be byte-addressable or word-addressable**
  - **Byte-addressable:** each byte has a unique address
  - **Word-addressable:** a word (e.g., 4 bytes) has a unique address

# Memory: Example

- **A memory word size of a machine is 16 bits**
- **A 4MB × 16 RAM chip gives us 4 megabytes of 16-bit memory locations**
  – $4MB = 2^2 * 2^{20} = 2^{22}$ = 4,194,304 unique locations (each location contains a 16-bit word)
  – Memory locations range from 0 to 4,194,303 in unsigned integers
- **$2^N$ addressable units of memory require N bits to address each location**
  – Thus, the memory bus of this system requires at least 22 address lines
  – The address lines “count” from 0 to $2^{22}$ -1 in binary

# Hardware Components of a Typical Computer – Peripheral Devices that Communicate with the Outside World




- **Input/Output (I/O)**
  - **Input:** keyboard, mouse, microphone, scanner, sensors (camera, infra-red), punch-cards
  - **Output:** video, printer, audio speakers, etc
- **Communication**
  - modem, ethernet card

# Hardware Components of a Typical Computer – Peripheral Devices that Store Data Long Term

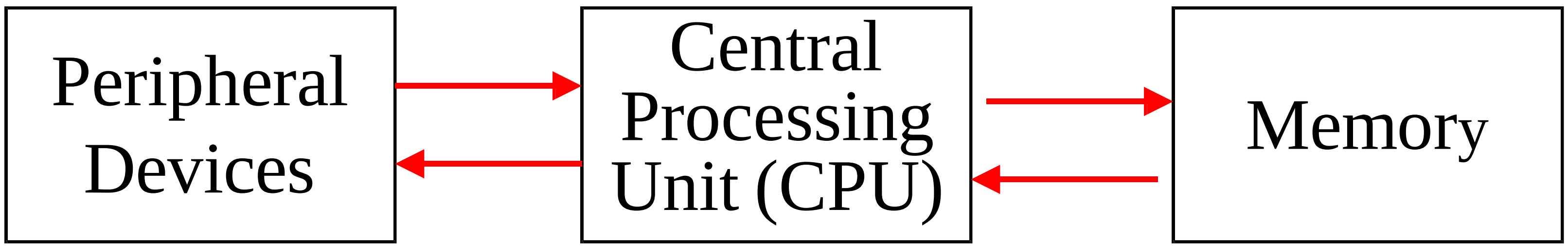
- **Secondary (mass) storage**
- **Stores information for long periods of time as files**
  - Examples: hard drive, floppy disk, tape, CD-ROM (Compact Disk Read-Only Memory), flash drive, DVD (Digital Video/Versatile Disk)

UCSC



BIT

# Hardware Components of a Typical Computer – Buses




## Buses

- **Used to share data between system components inside and outside the CPU**

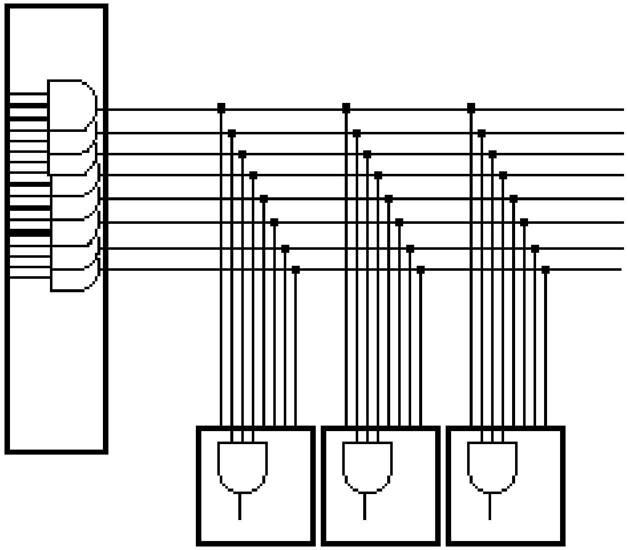
- **Set of wires (lines) that**
  - act as a shared path
  - allow parallel movement of bits

# Typical Bus Transactions

- Sending an address (for performing a read or write)
- Transferring data from memory to register and vice versa
- Transferring data for I/O reads and writes from peripheral devices

# Buses

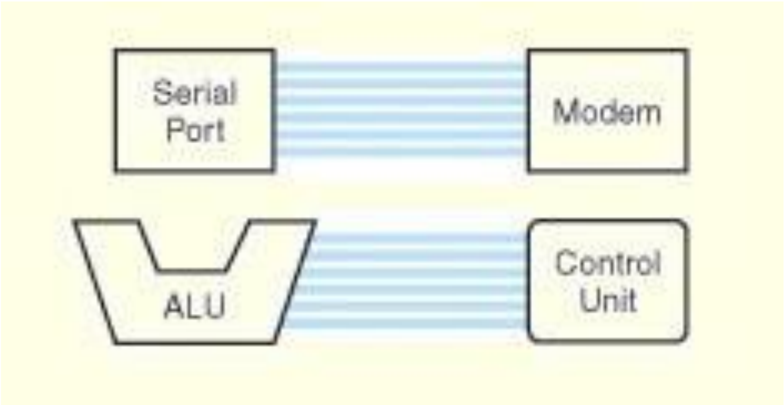
- Physically a bus is a group of conductors that allows all the bits in a binary word to be copied from a *source* component to a *destination* component

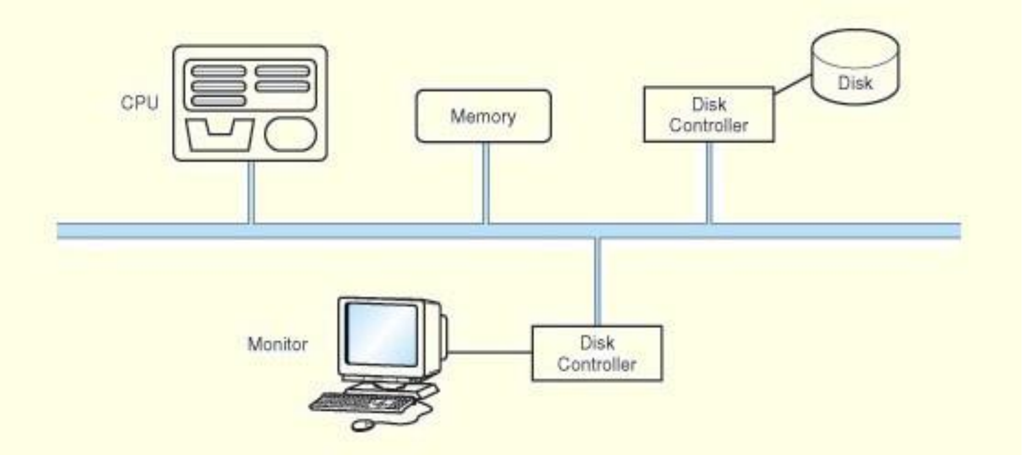


- Buses move binary values inside the CPU between registers and other components
- Buses are also used outside the CPU, to copy values between the CPU registers and main memory, and between the CPU registers and the I/O sub-system

# Types of Buses: Source and Destination

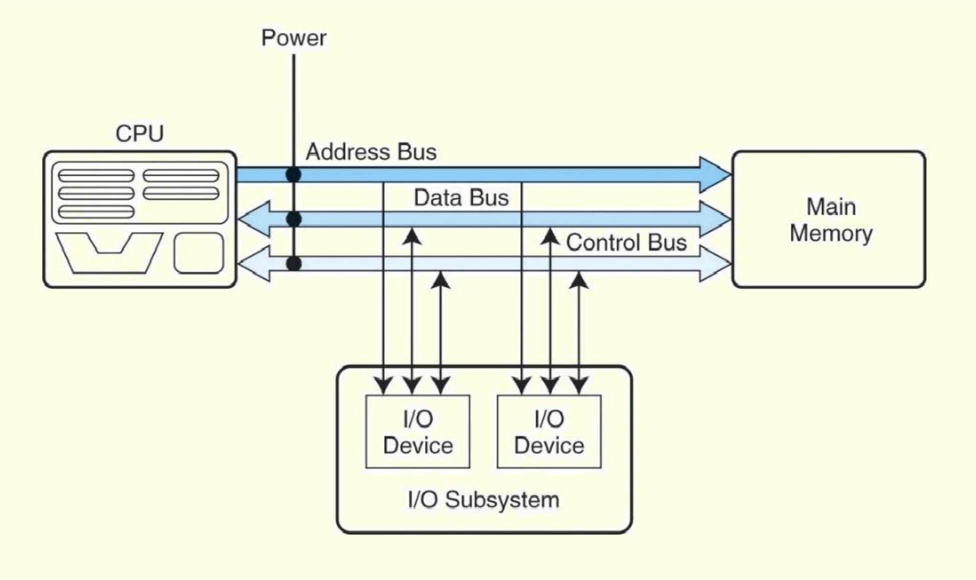
- **Point-to-point: connects two specific components**
- **Multi-point: a shared resource that connects several components**
  - access to it is controlled through protocols, which are built into the hardware



# Types of Buses: Contents

- Data bus: conveys bits from one device to another
- Control bus: determines the direction of data flow and when each device can access the bus
- Address bus: determines the location of the source or destination of the data




UCSC



BIT

# Clock

- **Every computer contains at least one clock that synchronizes the activities of its components**
  - A fixed number of clock cycles are required to carry out each data movement or computational operation
  - The clock frequency determines the speed of all operations
    - Measured in megaHertz or gigaHertz
- **Generally the term clock refers to the CPU (master) clock**
  - Buses can have their own clocks which are usually slower
- **Most machines are synchronous**
  - Controlled by a master clock signal
  - Registers must wait for the clock to tick before loading new data

UCSC



BIT

# Clock Speed (I)

- **Clock cycle time is the reciprocal of clock frequency**
  - Example, an 800 MHz clock has a cycle time of 1.25 ns
    - $1/800{,}000{,}000 = 0.00000000125 = 1.25 * 10^{-9}$
- **Clock-speed ≠ CPU-performance**
  - The CPU time required to run a program is given by the general performance equation:

$$\text{CPU Time} = \frac{\text{seconds}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{avg. cycles}}{\text{instruction}} \times \frac{\text{seconds}}{\text{cycle}}$$

# Clock Speed (II)

- **Therefore, we can improve CPU throughput when we reduce**
  - the number of instructions in a program
  - the number of cycles per instruction
  - the number of nanoseconds per clock cycle
- **But, in general**
  - Multiplication takes longer than addition
  - Floating point operations require more cycles than integer operations
  - Accessing memory takes longer than accessing registers

# Features of Computers: Speed and Reliability

- **Speed**
  - CPU speed
  - System-clock / Bus speed
  - Memory-access speed
  - Peripheral device speed
- **Reliability**

# CPU Speed

- **CPU clock speed: in cycles per second ("hertz")**
  – Example: 700MHz Pentium III, 3GHz Pentium IV
- **but different CPU designs do different amounts of work in one clock cycle**
- **Other measures of speed**
  – "flops" (floating-point operations per second)
  – "mips" (million instructions per second)

# System-Clock / Bus Speed

- **Speed of communication between CPU, memory and peripheral devices**
- **Depends on main board design**
  - Examples:
    - Intel 1.50GHz Pentium-4 works on a 400MHz bus speed

# Memory-Access Speed

- **RAM**
  - about 60ns (1 nanosecond = a billionth of a second), and getting faster
  - may be rated with respect to “bus speed” (e.g., PC-100)
- **Cache memory**
  - faster than main memory (about 20ns access speed), but more expensive
  - contains data which the CPU is likely to use next

# Peripheral Device Speed

- **Mass storage**
  - Examples:
    - 3.5in 1.4MB floppy disk: about 200kb/sec at 300 rpm (revolutions per minute)
    - Hard drive: up to 160 GB of storage, average seek time about 6 milliseconds, and 7,200 rpm
- **Communications**
  - Examples: modems at 56 kilobits per second, and network cards at 10 or 100 megabits per second
- **I/O**
  - Examples: ISA, PCI, IDE, SCSI, ATA, USB, etc....

# Cache Memory and Virtual Memory

- **Cache memory – random access memory that a processor can access more quickly than regular RAM**
- **Virtual memory – an "extension" of RAM using the hard disk**
  - allows the computer to behave as though it has more memory than what is physically available

UCSC



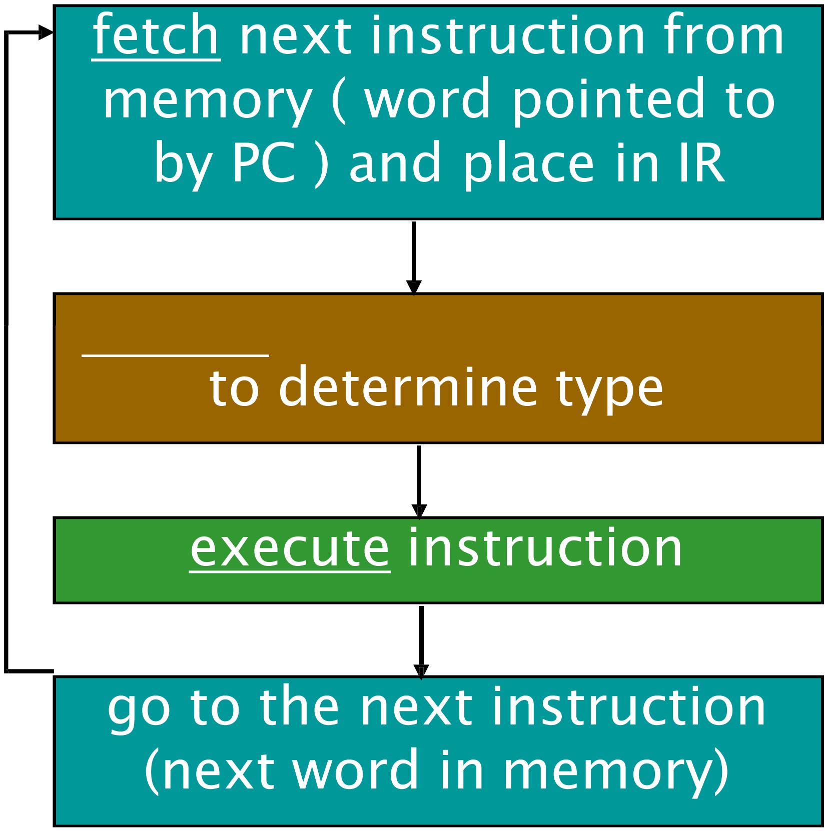
BIT

# Interrupts and Exceptions

- **Events that alter the normal execution of a program**
- **Exceptions are triggered within the processor**
  - Arithmetic errors, overflow or underflow
  - Invalid instructions
  - User-defined break points
- **Interrupts are triggered outside the processor**
  - I/O requests
- **Each type of interrupt or exception is associated with a procedure that directs the actions of the CPU**

# Fetch-decode-execute Cycle

**A computer runs programs by performing fetch-decode-execute cycles**




Example: instruction word at mem[PC] is 0x20A9FFFD

001000 00101 01001 1111111111111101

Opcode 8 is "add immediate", source reg is $5, "target" reg is reg $9, add amount is -3

Send reg $5 and -3 to ALU, add them, put result in reg $9

PC = PC + 4

# Accessing Memory (I)

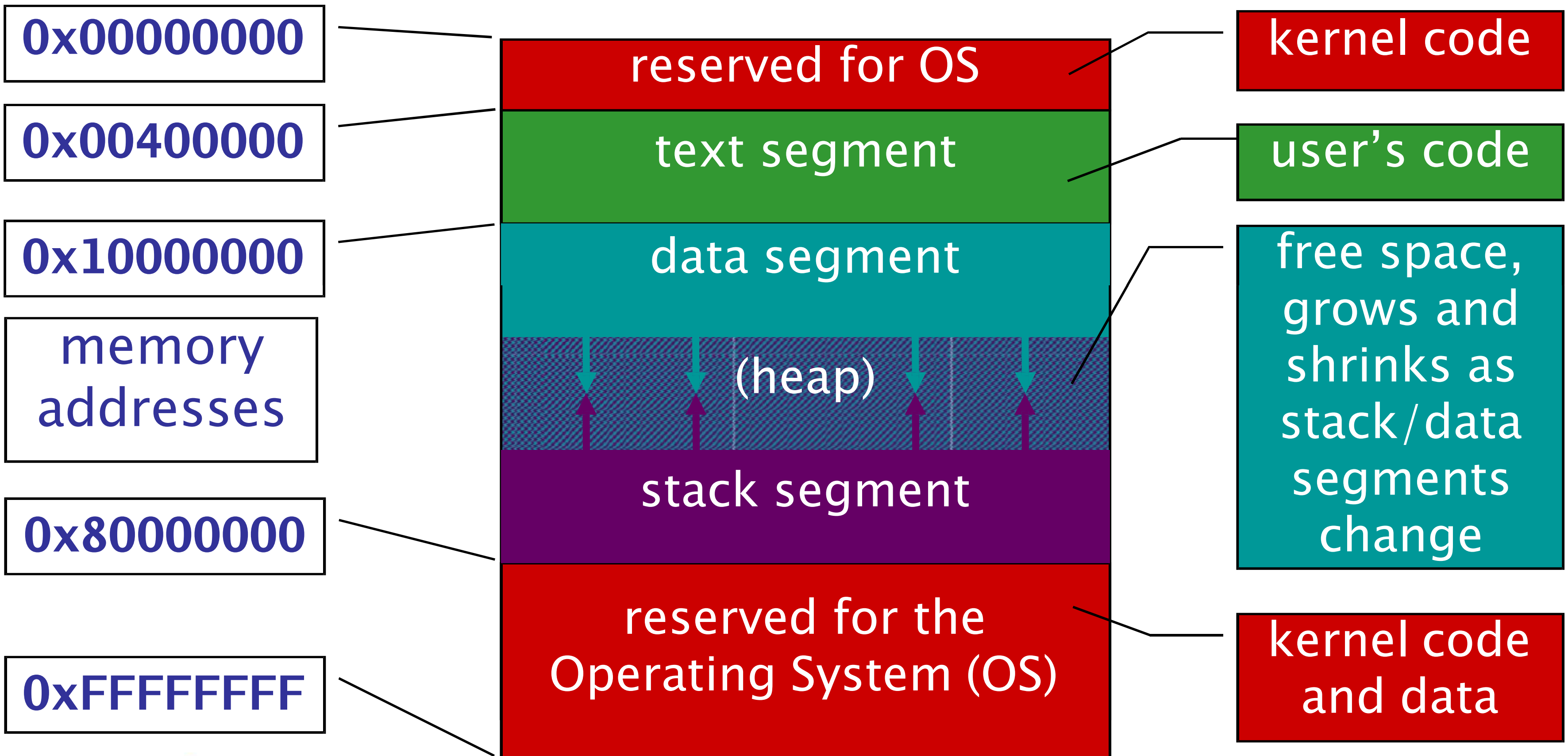
- **Every memory access needs an address word to be sent from CPU to memory**
  - Address range is 0x00000000 to 0xFFFFFFFF
    - about 4 billion bytes of addressable space
- **Addresses output by the CPU go to the Memory Address Register (MAR)**
  - During a **fetch** access, the **PC** value is copied to **MAR**
  - During a **load/store** access, a "computed address" from the ALU is copied to **MAR**

# Accessing Memory (II)

- **Why compute load/store addresses?**
  - **32(instruction bits) – 6(opcode bits) = 26(available bits)**
  - insufficient to hold a full memory address
- **Solution: register based addressing**
  - use 26-bits to specify a base address GPR, a target GPR, plus a 16-bit signed offset
  - ALU computes memory reference address “on the fly” as: **MAR** = base GPR + offset
  - target GPR receives/supplies memory data

# Memory Segments

**Memory is organized into segments, each with its own purpose**



# Text Segment

- Starts at memory address 0x00400000
  – runs up to address 0x0FFFFFFF
- Contains user's *executable program code* (often called the *code segment* )
- PC register value is a CPU "reference" into this memory segment

# Data Segment

- Starts at memory address 0x10000000
  - expands upwards towards stack

- Contains program's *static data*, i.e., data and variables whose location in memory is fixed (and known to the assembler)

| In C | In Java |
| --- | --- |
| global variables<br>string constants | public, static<br>objects |

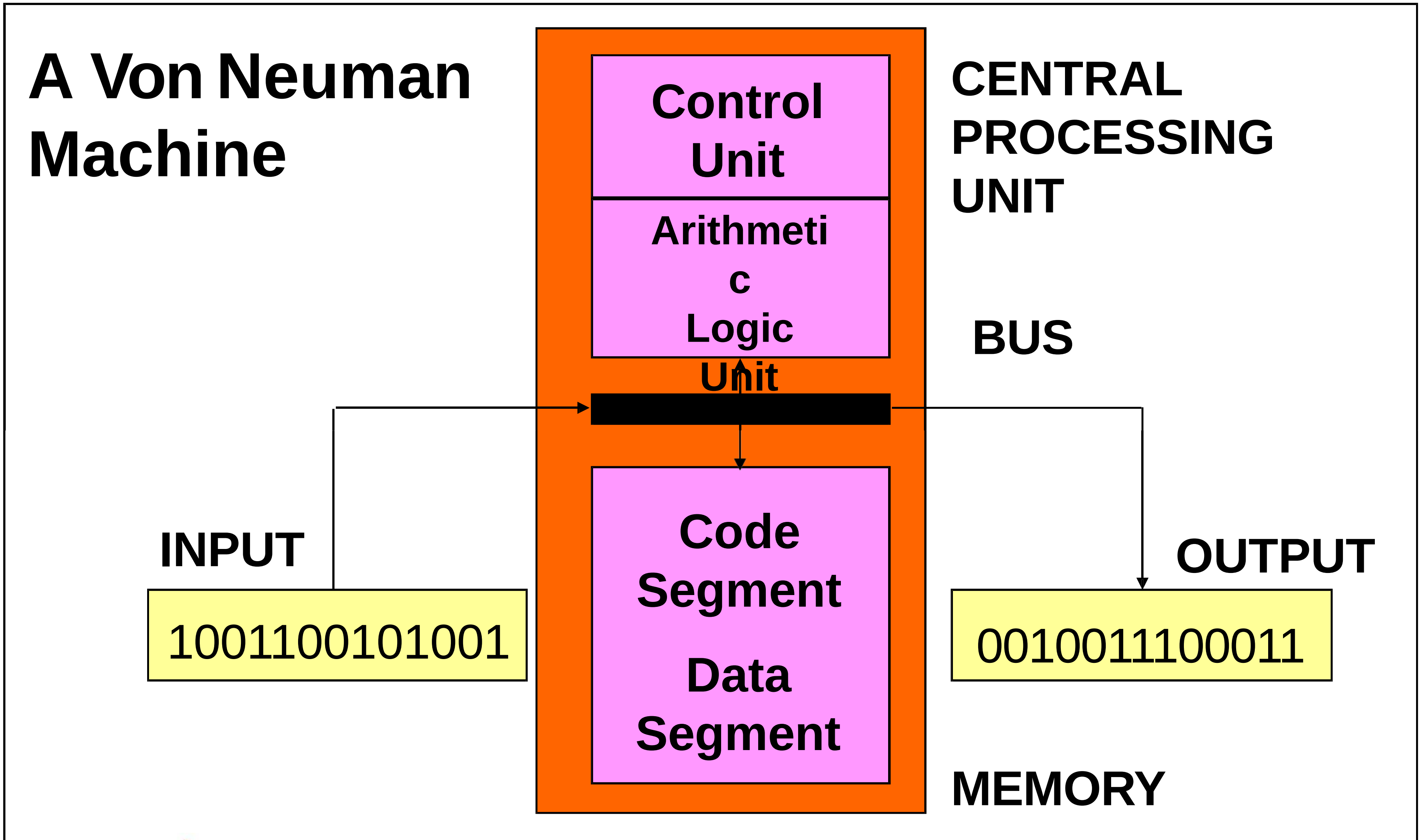
# Stack Segment

- **Starts at memory address 0x7FFFFFFF**
  - grows in the direction of decreasing memory addresses ( i.e., towards the data segment)
- **Contains *system stack***
- **Used for temporary storage of:**
  - local variables of functions
  - function parameter values
  - return addresses of functions
  - saved register values

# Heap

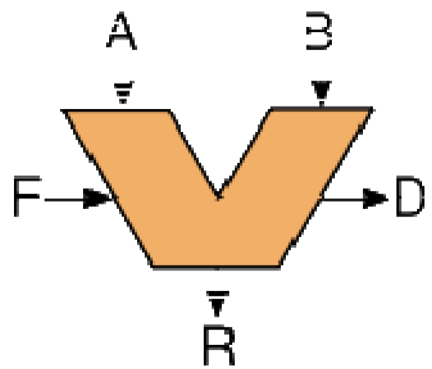
- **Technically part of data segment**
  - located at end of data segment, after all static data
- **Empty at start of program execution**
- **Dynamically allocated memory is taken from heap for program to use**
- **Freed memory (by user or garbage collection) is returned to heap**

# Block Diagram of the System



# Arithmetic Logic Unit

- ALU
  - The part of a computer that performs all arithmetic computations, such as addition and multiplication, and all comparison operations




  - A typical schematic symbol for an ALU: **A** & **B** are operands; **R** is the output; **F** is the input from the Control Unit; **D** is an output status

# Arithmetic Logic Unit…

- The component where data is held temporarily
- Calculations occur here
- It knows how to perform operations such as **ADD**, **SUB**, **LOAD**, **STORE**, **SHIFT**
- It knows the commands that make up the machine language of the CPU
- It is the calculator

UCSC



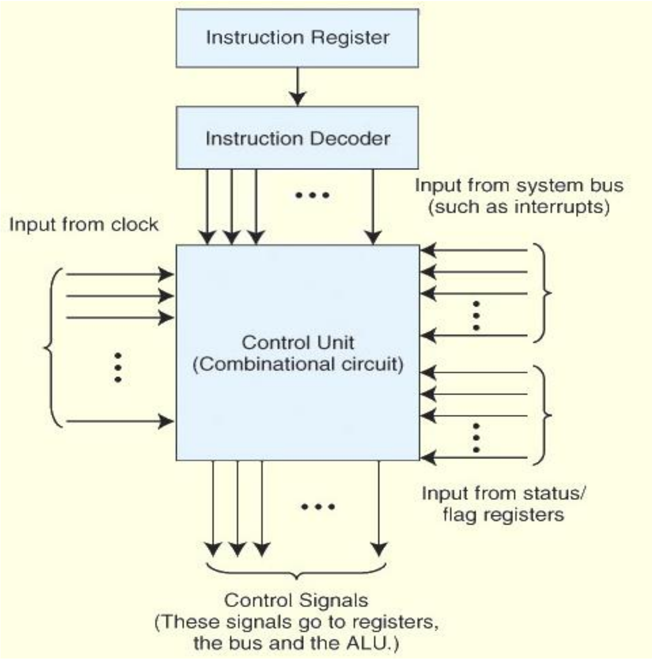
BIT

# Control Unit

- **A computer's control unit keeps things synchronized**
  - Makes sure that the correct components are activated as the components are needed
  - Sends bits down control lines to trigger events
    - E.g., when Add is performed, the control signal tells the ALU to Add
  - How do these control lines become asserted?
    - **Hardwired control**: controllers implement this program using digital logic components
    - **Microprogrammed control**: a small program is placed into read-only memory in the ***microcontroller***

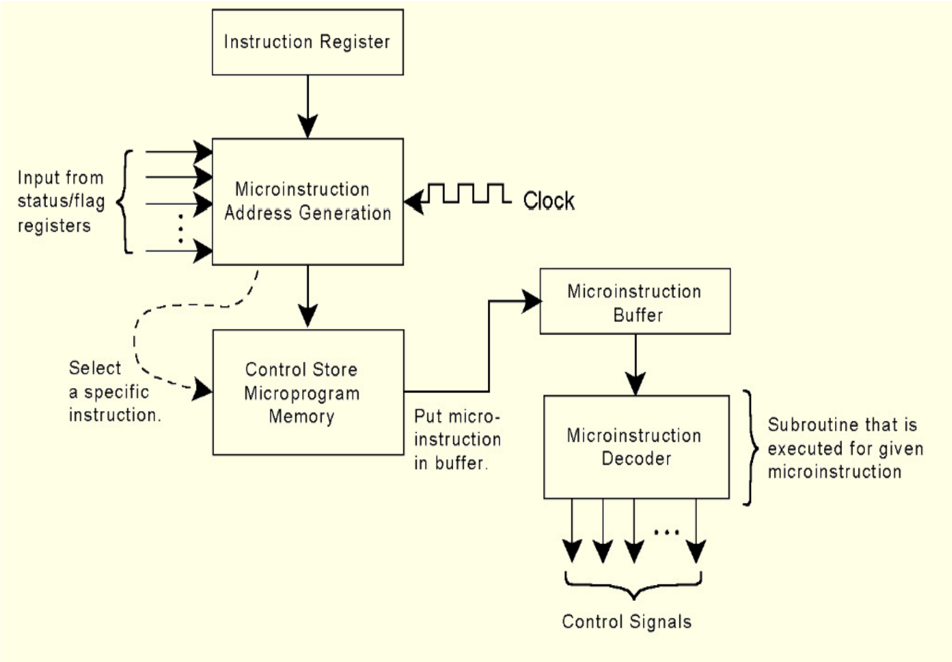
# Control Unit: Hardwired Control

- **Physically connect all of the control lines to the actual machine instruction**
  - Instructions are divided into fields and different bits are combined with various digital logic components (which drive the control line)
- **The control unit is implemented using hardware**
  - The digital circuit uses inputs to generate the control signal to drive various components
- **Advantage: very fast**
- **Disadvantage: instruction set and digital logic are locked**



# Control Unit: Microprogrammed Control

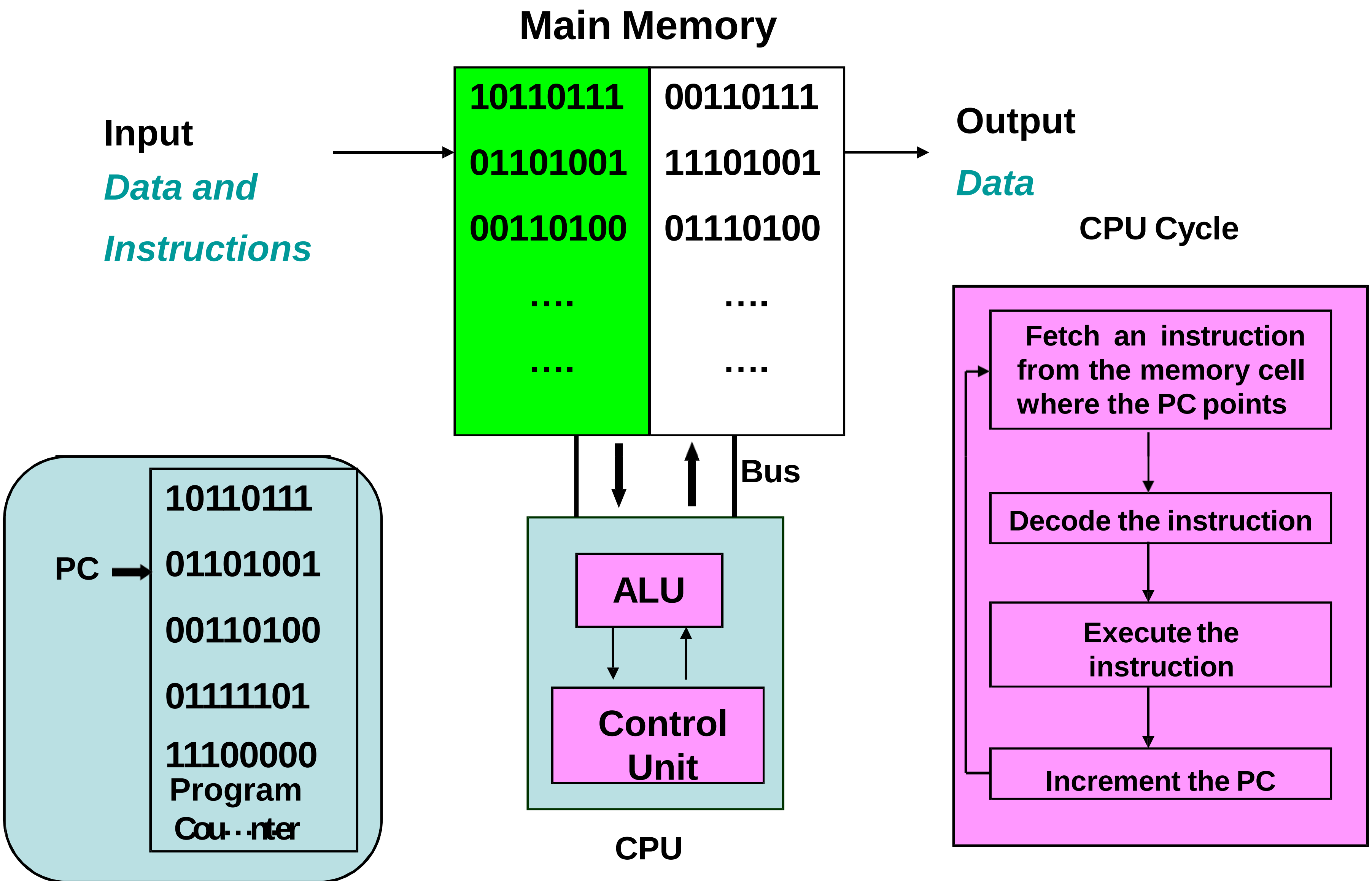
- **Microprogram: software stored in the CPU control unit**
- **Converts machine instructions (binary) into control signals**
- **One subroutine for each machine instruction**
- **Advantage: very flexible**
- **Disadvantage: additional layer of interpretation**



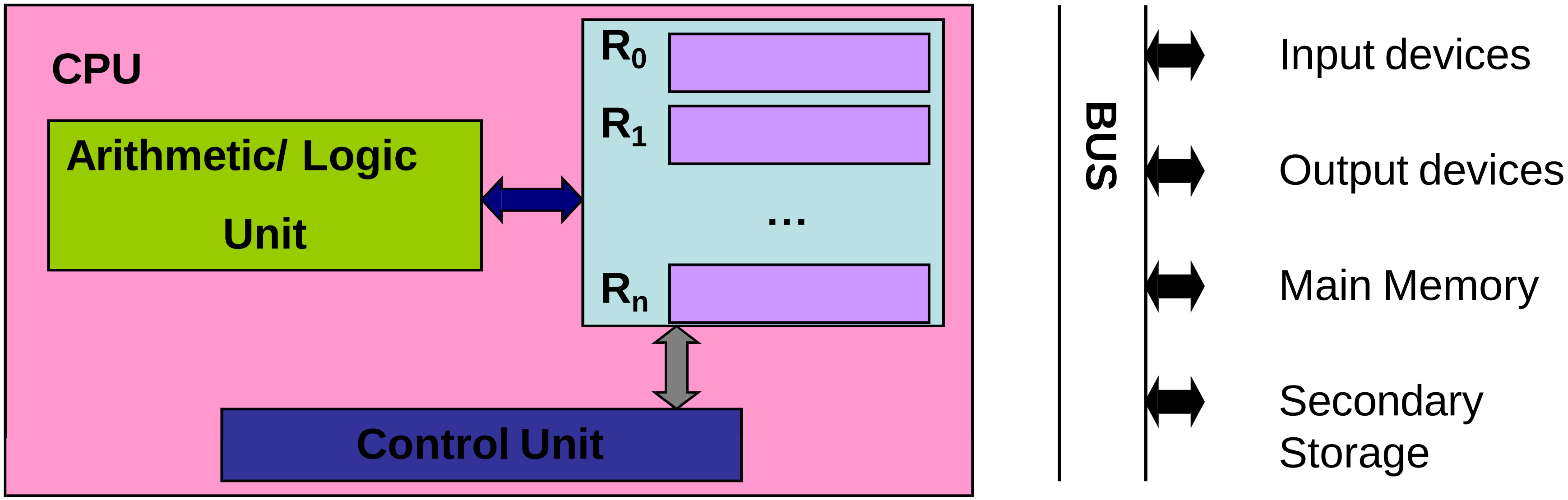
# Registers

- "A register is a single, permanent storage location within the CPU used for a PARTICULAR, defined purpose"
- "A register is used to hold a binary value temporarily for storage, for manipulation, and/or for simple calculations"
- Registers have special addresses

# Von Neuman Machine Model



# Registers




**Registers** are used to hold the data immediately applicable to the operation at hand;

**Main memory** is used to hold the data that will be needed in the near future

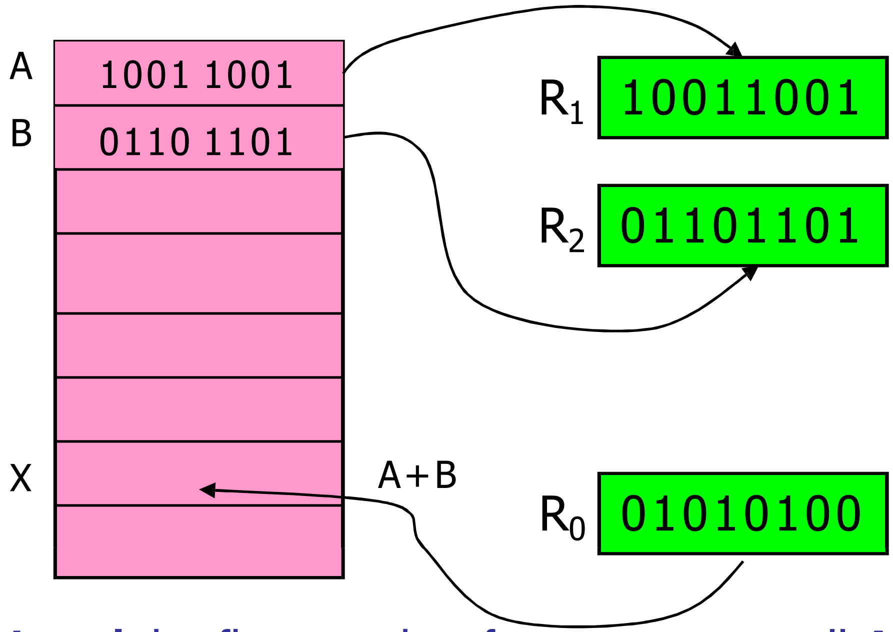
**Secondary storage** is used to hold data that will be likely not be needed in the near future

# Example: Machine Architecture

- Consider a machine with
  - 256 byte Main Memory: 00-FF
  - 16 General Purpose Registers: 0-F
  - 16 Bit Instruction
  - 8 Bit Integer Format (2's Complement)
  - 8 Bit Floating Point Format
    - 1 Sign Bit
    - 3 Exponent Bits
    - 4 Bit Mantissa
  - 16 Instructions: 1-F

| | |
|---|---|
| 00 | 0001 0001 |
| 01 | 0011 0000 |
| 02 | 0001 0010 |
| 03 | 0100 0000 |
| 04 | 0011 0001 |
| | 0100 0000 |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| | |
| ff | 0100 0000 |

# Example: Addition Operation




```
LOAD   R1 , A
LOAD   R2 , B
ADD    R0 , R1 , R2
STORE  R0 , X
```

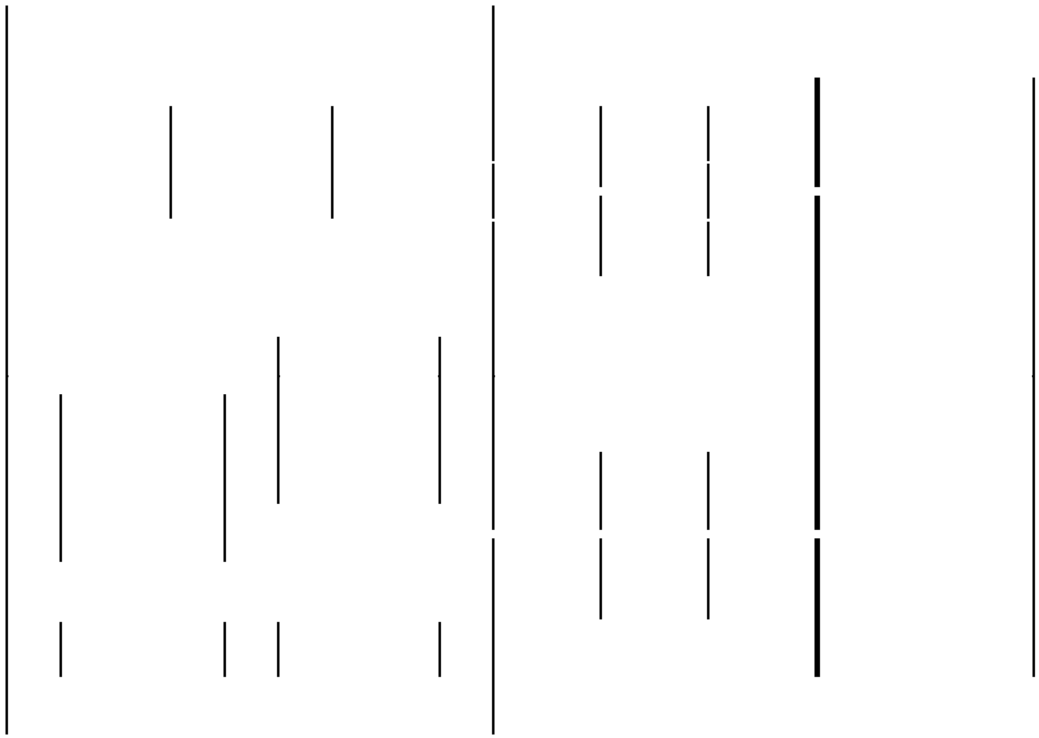
**Load** the first number from memory cell **A** into register $\mathbf{R_1}$

**Load** the second number from memory cell **B** into register $\mathbf{R_2}$

**Adding** the numbers in these two registers and put the result in register $\mathbf{R_0}$

**Store** the result in $\mathbf{R_0}$ into the memory call **X**

# Block Diagram of the CPU



**CPU** - Central Processing Unit

**MAR** - Memory Address Register

**IR** - Instruction Register

**MDR** - Memory Data Register

**PC** - Program Counter

**ALU** - Arithmetic Logic Unit

# Instruction Fetch

- The address in the **Program Counter** is placed in **MAR**
- The addressed instruction is read from memory (through the **MDR**) and placed into the **Instruction Register**

# Instruction Execute

- The **Instruction Decoder** examines the instruction in the **Instruction Register** and sends appropriate signals to other parts of the **CPU** to carry out the actions specified by the instruction. This may include:
  - Reading operands from memory or registers into the **Arithmetic Logic Unit**,
  - Enabling the circuits of the **Arithmetic Logic Unit** to perform arithmetic or other computations,
  - Storing data values into memory or registers,
  - Changing the value of the **Program Counter**

# The CPU Cycle

- The processor endlessly repeats the cycle:

**fetch, execute, fetch, execute, fetch, execute, fetch, execute, fetch, execute, fetch, execute, fetch, execute, fetch, execute, fetch, execute, fetch ...**

# Fetch and Execute Cycle

- At the beginning of each cycle the CPU presents the value of the **program counter** on the **address bus**
- The CPU then fetches the instruction from **main memory** (possibly via a **cache** and/or a **pipeline**) via the **data bus** into the instruction register

# Fetch and Execute Cycle

- From the **instruction register**, the data forming the instruction is decoded and passed to the **control unit**

- It sends a sequence of control signals to the relevant function units of the **CPU** to perform the actions required by the instruction such as reading values from registers, passing them to the **ALU** to add them together and writing the result back to a **register**

# Fetch and Execute Cycle

- The **program counter** is then incremented to address the next instruction and the cycle is repeated

# Instruction Set Architecture (ISA)

- **Instruction sets – definition and features**
  - Instruction types
  - Operand organization
  - Number of operands and instruction length
  - Addressing
  - Instruction execution – pipelining
- **Features of two machine instruction sets (CISC and RISC)**
- **Instruction format**

# Instruction Set Architecture (ISA)

- **Machine instructions**
  - Opcodes and operands
- **High level languages**
  - Hide detail of the architecture from the programmer
  - Easier to program
- **Why learn computer architectures and assembly language?**
  - To understand how the computer works
  - To write more efficient programs

# Instruction Set Architecture (ISA)

**Instruction sets are differentiated by**

- **Instructions**
  - types of instructions
  - instruction length and number of operands
- **Operands**
  - type (addresses, numbers, characters) and access mode
  - location (CPU or memory)
  - organization (stack or register based)
    - number of addressable registers
- **Memory organization**
  - byte- or word-addressable
- **CPU instruction execution**
  - with/without pipelining

UCSC



BIT

# Instruction Set Architecture (ISA)

- **The instruction set format is critical to the machine's architecture**
- **Performance of instruction set architectures is measured by**
  - Main memory space occupied by a program
  - Instruction complexity
  - Instruction length (in bits)
  - Total number of instructions

# Instruction Set Architecture (ISA)

- Instruction types
- Operand organization
- Number of operands and instruction length
- Addressing
- Instruction execution – pipelining

# Instruction Set Architecture (ISA)

- An **instruction set**, or **instruction set architecture (ISA)** describes the aspects of a computer architecture visible to a programmer, including the native data-types, instructions, registers, addressing modes, memory architecture, interrupt and exception handling, and external I/O (if any)
- An ISA includes a specification of the set of all binary codes **(opcodes)** that are the native form of commands implemented by a particular CPU design
- The set of **opcodes** for a particular ISA is also known as the **machine language** for the ISA

# Instruction Set Architecture (ISA)

- **ISAs commonly implemented in hardware**
  - Alpha AXP (DEC Alpha)
  - ARM (Acorn RISC Machine) (Advanced RISC Machine now ARM Ltd)
  - IA-64 (Itanium)
  - MIPS
  - Motorola 68k
  - PA-RISC (HP Precision Architecture)
  - IBM POWER
  - PowerPC
  - SPARC
  - SuperH
  - VAX (Digital Equipment Corporation)
  - x86 (IA-32, Pentium, Athlon) (AMD64, EM64T)

# Machine Instructions

- **Data Transfer**: transfer data between registers and memory cells
- **Arithmetic/Logic Operations**: perform addition, AND, OR, XOR and etc.
- **Control Operations**: control the execution of the program

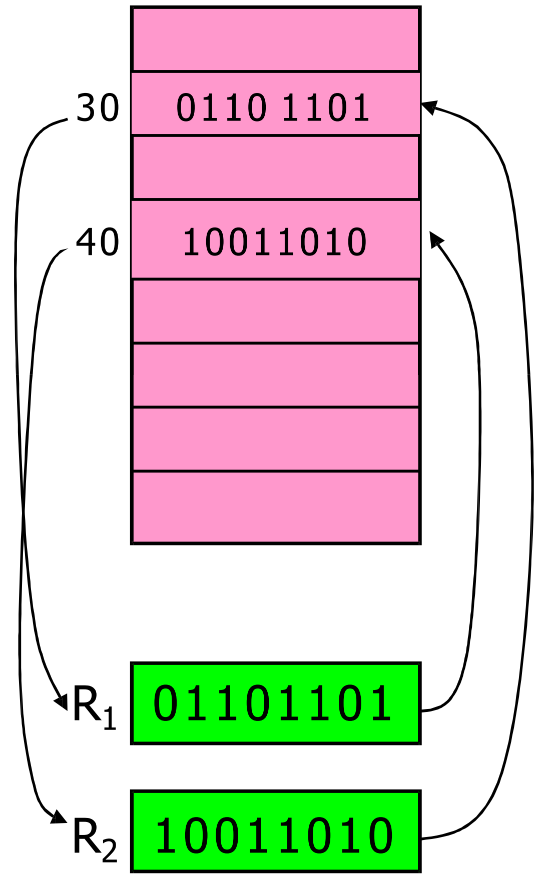
# Data Transfer Instructions

1. **L R , A** **<u>LOAD</u>** the register **R** with the content of memory cell **A**
2. **LI R , I** **<u>LOAD</u>** the register **R** with **I** (**I** is called an <u>immediate</u> number)
3. **ST R , A** **<u>STORE</u>** the content of the register **R** to the memory cell whose address is **A**
4. **LR R1 , R2** **<u>LOAD</u>** the register $\mathbf{R_1}$ with the content of the register $\mathbf{R_2}$

# Example: Data Transfer Instructions

Swap the content of two memory cells $30_{(16)}$ and $40_{(16)}$

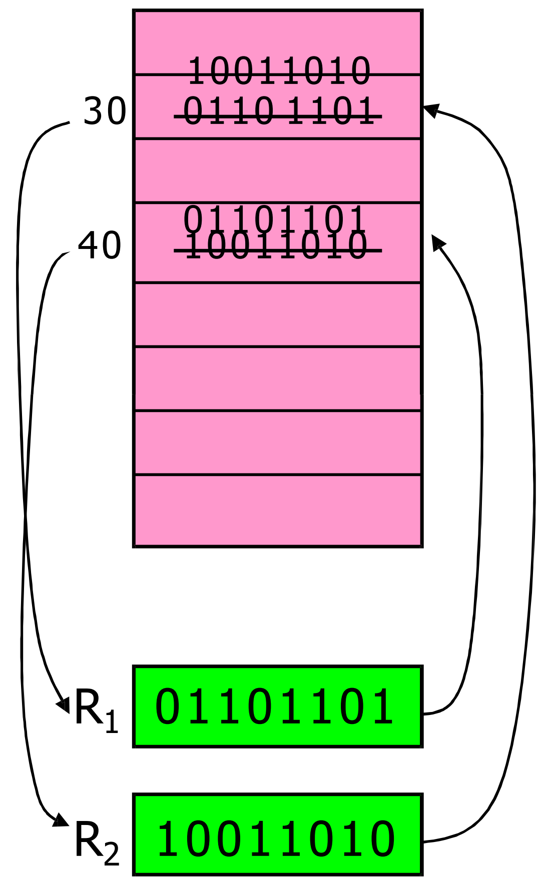
| | |
|---|---|
| L 1 , 30 | /*Load $R_1$ with the content in memory cell 30 */ |
| L 2 , 40 | /* Load $R_2$ with the content in memory cell 40 */ |
| ST 1 , 40 | /* Store $R_1$ to 40 */ |
| ST 2 , 30 | /* Store $R_2$ to 30 */ |



# Example: Data Transfer Instructions

<u>Swap</u> the content of two memory cells $30_{(16)}$ and $40_{(16)}$

| | | |
|---|---|---|
| L | 1 , 30 | /*Load $R_1$ with the content in memory cell 30 */ |
| L | 2 , 40 | /* Load $R_2$ with the content in memory cell 40 */ |
| ST | 1 , 40 | /* Store $R_1$ to 40 */ |
| ST | 2 , 30 | /* Store $R_2$ to 30 */ |



# Arithmetic/Logic Instructions (I)

## Arithmetic Instructions

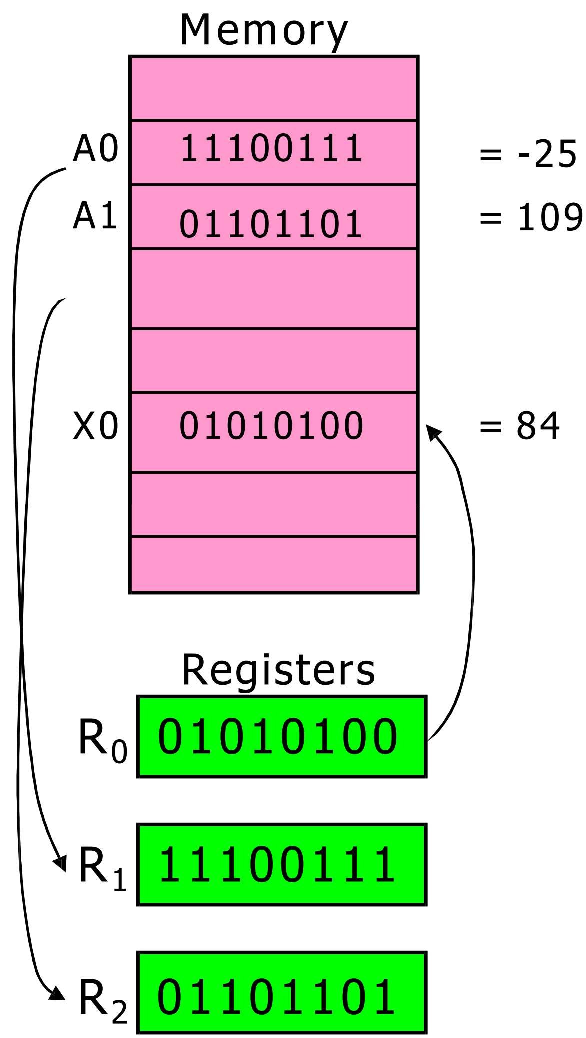
5. **ADD R0, R1, R2** — **ADD** the numbers in $\mathbf{R_1}$ and $\mathbf{R_2}$ representing in 2's complement and place the result in $\mathbf{R_0}$

6. **AFP R0, R1, R2** — **ADD** the numbers in $\mathbf{R_1}$ and $\mathbf{R_2}$ representing in floating-point and place the result in $\mathbf{R_0}$

# Arithmetic/Logic Instructions (I)

## Example: Addition

```
L   1 , A0

L   2 , A1

ADD 0 , 1 , 2

ST 0 , X0
```



# Arithmetic/Logic Instructions (II)

## Logic Instructions

7. **OR R0, R1, R2** — **OR** the bit patterns in $\mathbf{R_1}$ and $\mathbf{R_2}$ and place the result in $\mathbf{R_0}$

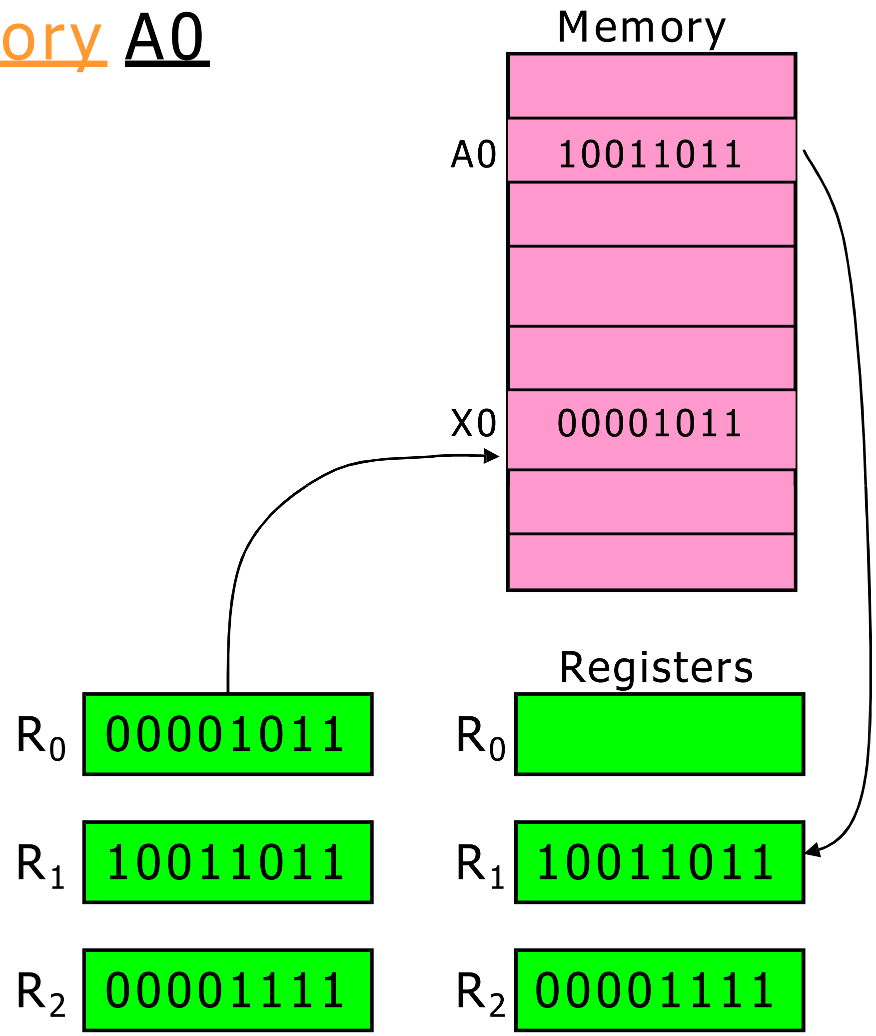
8. **AND R0, R1, R2** — **AND** the bit patterns in $\mathbf{R_1}$ and $\mathbf{R_2}$ and place the result in $\mathbf{R_0}$

9. **XOR R0, R1, R2** — **XOR** the bit patterns in $\mathbf{R_1}$ and $\mathbf{R_2}$ and place the result in $\mathbf{R_0}$

# Arithmetic/Logic Instructions (II)

Example: Mask the first 4 bits of the binary string in memory A0

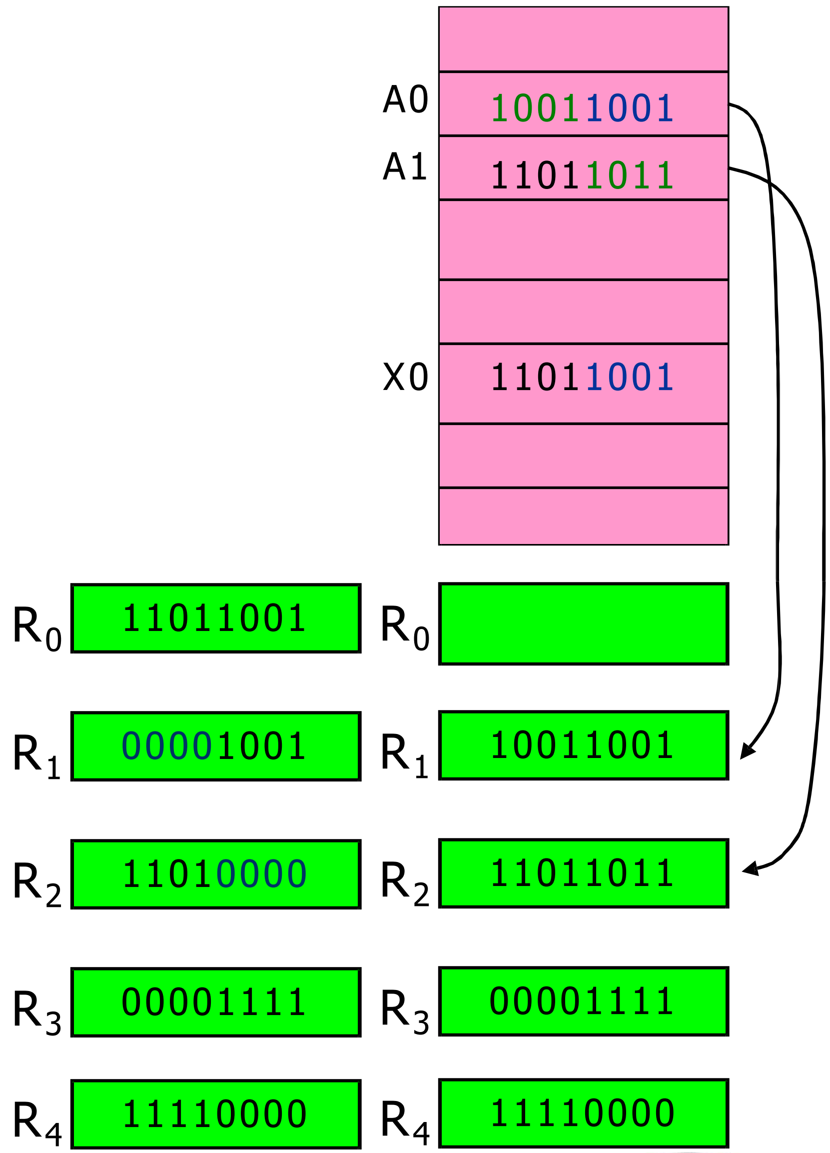
```
L   1 , A0

LI  2 , OF

ADD 0 , 1 , 2

ST  0 , X0
```



# Arithmetic/Logic Instructions (II)

## Example: Masking

```
L   1 , A0

L  2 , A1

LI  3, 0F

LI  4, F0

AND 1,1,3

AND 2,2,4

OR  0,1,2

ST 0 , X0
```



# Arithmetic/Logic Instructions (III)

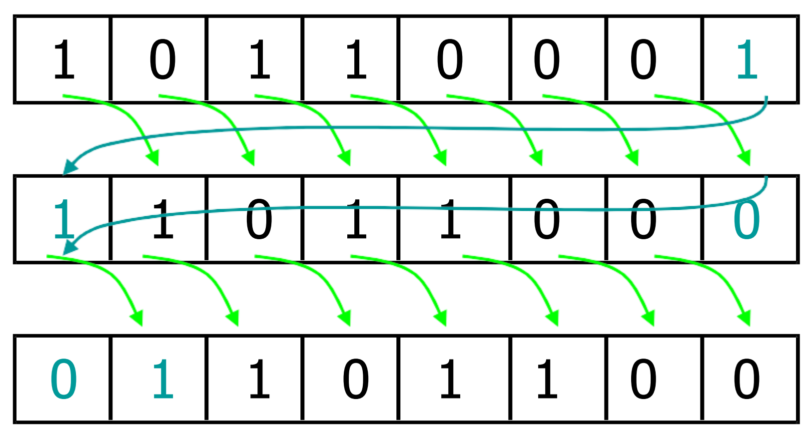
## Bit String Operating Instructions

**B. RR R , I**

**ROTATE** the bit patterns in **R** to right **I** times. Each time place the bit that started at the ***low-order*** end at the ***high-order*** end

**Example RR , 0 , 02**

Original String




Resulting String

# Control Instructions

**E. JMP R , A**

**<u>JUMP</u>** the instruction located in the memory cell **A** if the bit pattern in **R** is equal to the one in **R**

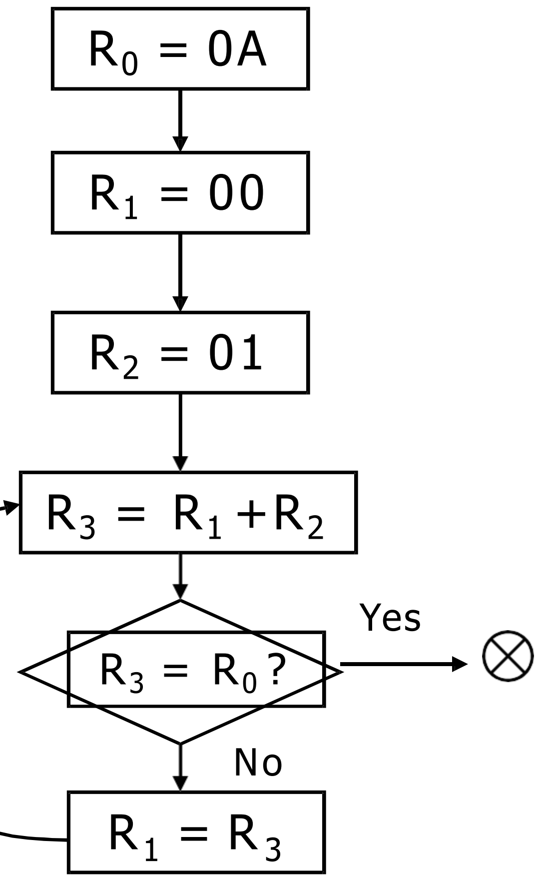
**F. HALT**

**<u>HALT</u>** the execution

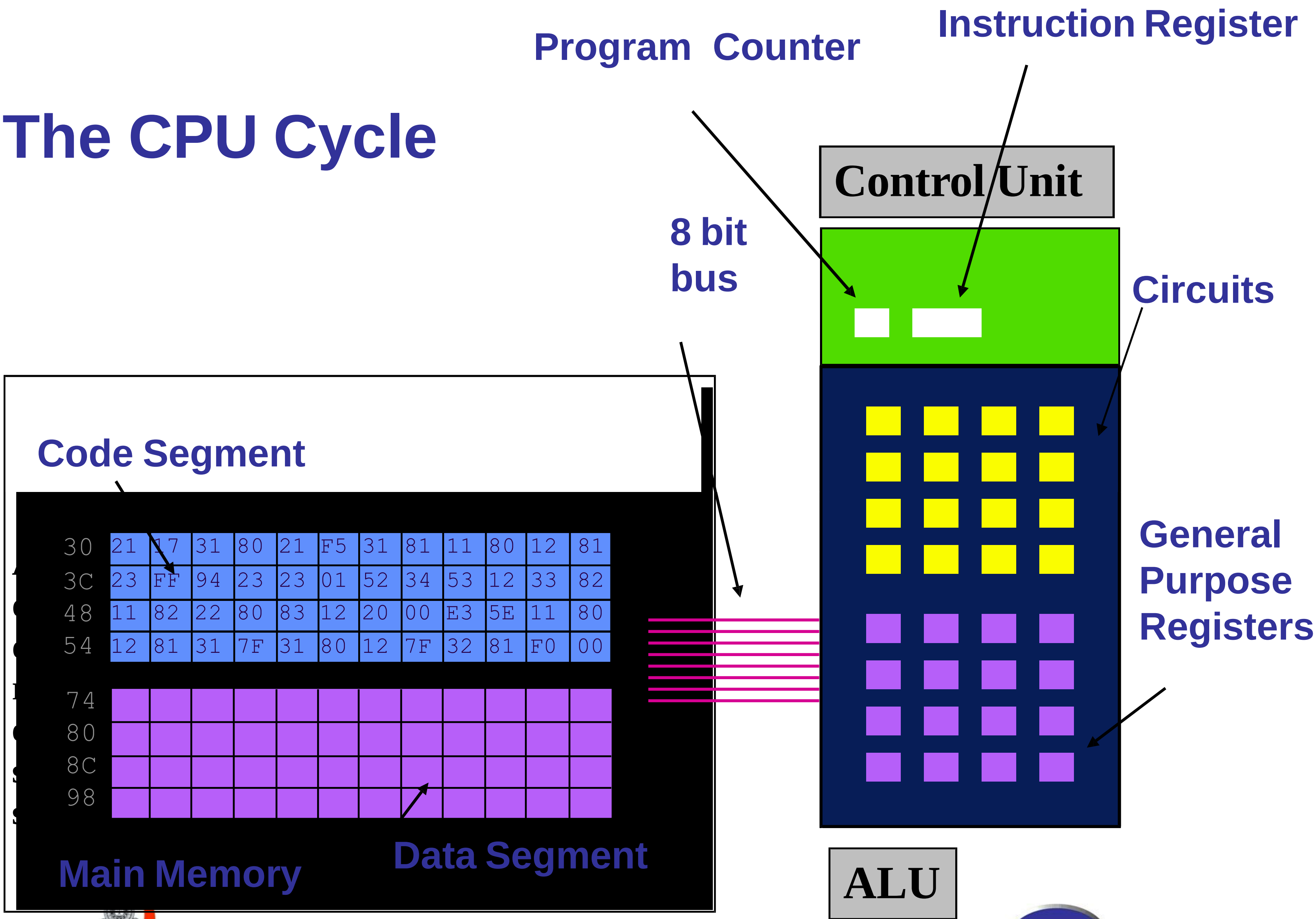
# Example: Control Instructions

```
30  LI   0 , 0A
32  LI   1 , 00
34  LI   2 ,  01
36  ADD  3 , 1, 2
38  JMP  3 , 3E
3A  LR   1 ,3
3C  JMP  0 ,36
3E  HALT
```

| | |
|---|---|
| $R_0$ | 00001010 |
| $R_1$ | 00000000 |
| $R_2$ | 00000001 |
| $R_3$ | 00000001 |



# The CPU Cycle



# Operand Organization

- **Three choices**
  - – Accumulator architecture
  - – General Purpose Register (GPR) architecture
  - – Stack architecture

# Operand Organization – Accumulator Architecture

- **One operand of a binary operation is implicitly in the accumulator**
- **Advantage**
  - Minimizes the internal complexity of the machine
  - Allows for very short instructions
- **Disadvantage**
  - Memory traffic is very high
  - Programming is cumbersome

# Operand Organization – General Purpose Register (GPR) Architecture

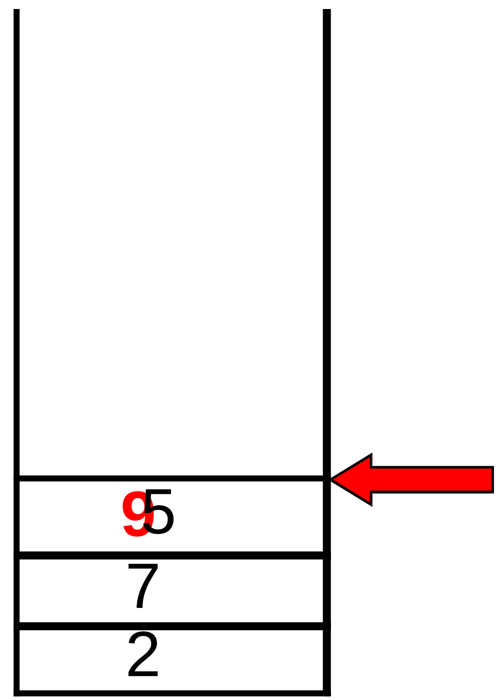
- **Uses sets of general purpose registers**
- **Advantage**
  - Register sets are faster than memory
  - Easy for compilers to deal with
  - Due to low costs large numbers of these registers are being added
- **Disadvantage**
  - Results in longer instructions (longer fetch and decode times)

# Operand Organization – General Purpose Register (GPR) Architecture

- **Three types**
  - **Memory-memory**
    - may have two or three operands in memory
    - an instruction may perform an operation without requiring any operand to be in a register
  - **Register-memory**
    - at least one operand must be in a register and one in memory
  - **Load-store**
    - requires data to be moved into registers before any operation is performed

# Operand Organization – Stack Architecture

- Uses a *stack* to execute instructions
- Operations:
  - PUSH – put a value on top of the stack
  - POP – read top value and move down the "stack pointer"
- Example:
  - POP
  - PUSH 9



# Operand Organization – Stack Architecture

- **Instructions implicitly refer to values at the top of the stack**
  – data can be accessed only from the top of the stack, one word at a time
- **Advantage**
  – Good code density
  – Simple model for evaluation of expressions
- **Disadvantage**
  – Restricts the sequence of operand processing
  – Execution bottleneck (the stack is located in memory)

# Operand Organization – Stack Architecture

- **Stack architecture requires us to think about arithmetic expressions in a new way**
  - We are used to ***Infix notation***
    - E.g., Z = X + Y
  - Stack arithmetic requires ***Postfix notation***:
    - E.g., Z = XY+
    - Postfix notation is also know as ***Reverse Polish Notation***

# Stack Architecture – Postfix Notation

- **Postfix notation doesn't need parentheses**
- **E.g.,**
  - The infix expression $Z = (X * Y) + (W * U)$
    is the postfix expression $Z = X\ Y * W\ U * +$
  - Calculating $Z = X\ Y * W\ U * +$ in a stack ISA

```
PUSH X
PUSH Y
MULT
PUSH W
PUSH U
MULT
ADD
POP  Z
```

Binary operators
- **pop** the two operands on the stack top, and
- **push** the result on the stack

# Number of Operands and Instruction Length

- **The number of operands in each instruction affects the length of the instruction**
- **Instruction length can be**
  - **Fixed –** quick to decode but wastes space
  - **Variable –** more complex to decode but saves space
- **All architectures limit the number of operands allowed per instruction**
  - Stack architecture has 0 or 1 explicit operand
  - Accumulator architecture has 0 or 1 explicit operand
  - GPR architecture has 1, 2 or 3 operands

# Number of Operands - Example

- Calculating the infix expression Z = X * Y + W * U

**One operand**

```
LOAD   X
MULT   Y
STORE  TEMP
LOAD   W
MULT   U
ADD    TEMP
STORE  Z
```

**Two operands**

```
LOAD   R1,X
MULT   R1,Y
LOAD   R2,W
MULT   R2,U
ADD    R1,R2
STORE  Z,R1
```

**Three operands**

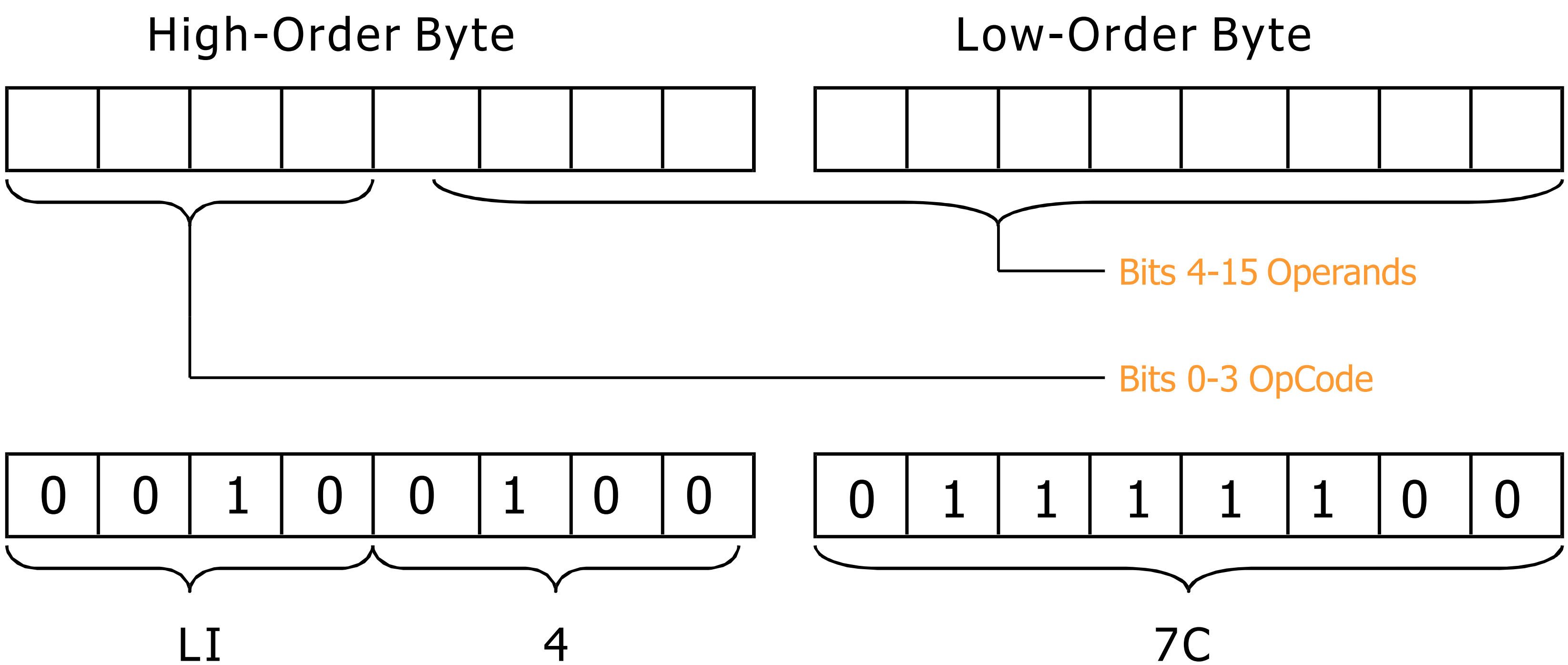
```
MULT R1,X,Y
MULT R2,W,U
ADD  Z,R1,R2
```

**The accumulator is the destination for the result of the instruction**

**The first operand is often the destination for the result of the instruction**

# Coding Instruction

## 16 bit Instruction (2 bytes)



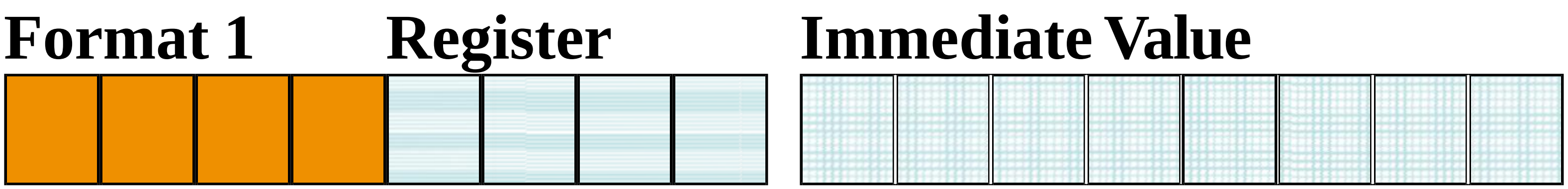

The machine code 0010010001111100 represents the instruction **LI 4 , 7C**

# Instruction Formats

## 16 bit Instruction (2 bytes)

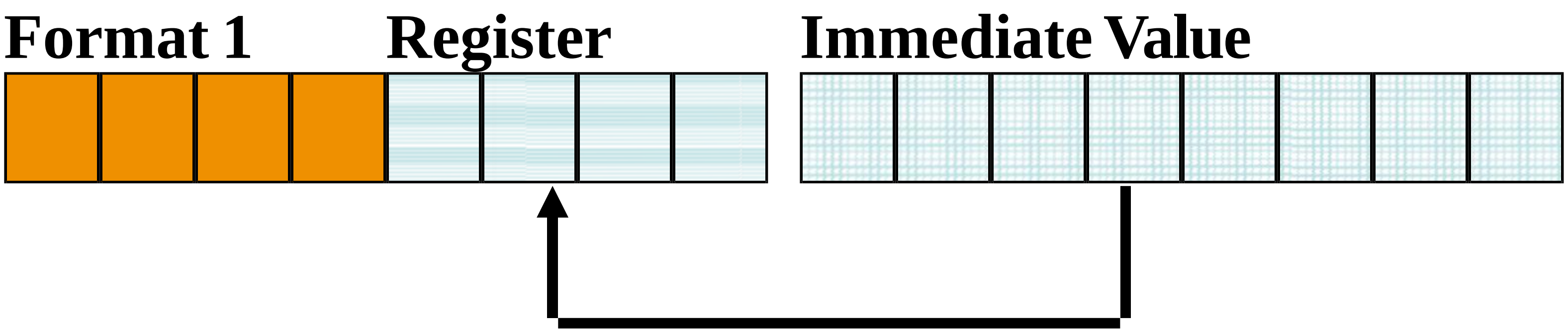
# Format 1 Instruction

## Format 1 Instruction




| Opcode | Instruction | Meaning |
|---|---|---|
| 2 | LI R , I | Load Immediate |
| A | RL R , I | Rotate Left |
| B | RR R , I | Rotate Right |
| C | SL R , I | Shift Left |
| D | SR R , I | Shift Right |

# Format 1 Instruction

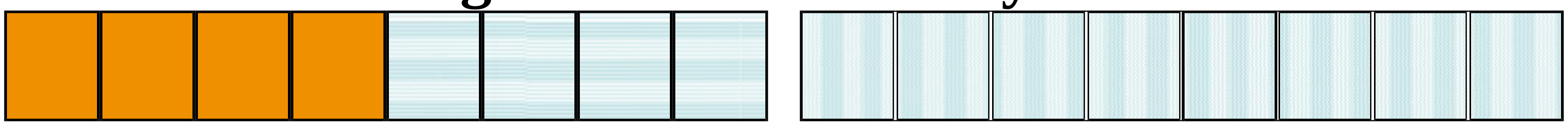



1. COPY THE BIT PATTERN IN THE LOW-ORDER BYTE INTO THE SPECIFIED REGISTER , OR
2. SHIFT/ROTATE THE BITS IN THE SPECIFIED REGISTER THE NUMBER OF PLACES SPECIFIED IN THE LOW-ORDER BYTE.

# Format 2 Instruction

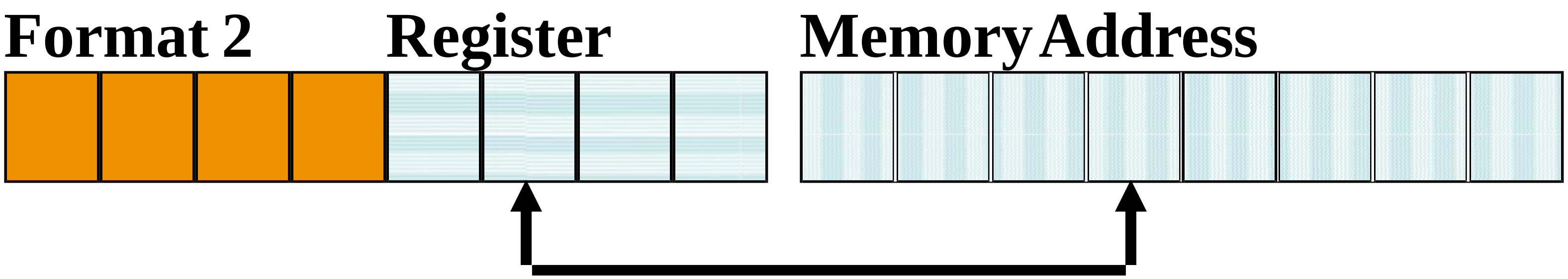
## Format 2 Instruction

Format 2 Register Memory Address



| Opcode | Instruction | Meaning |
|---|---|---|
| 1 | L R , A | Load from Memory |
| 3 | ST R , A | Store to Memory |
| E | JMP R , A | Conditional Jump |

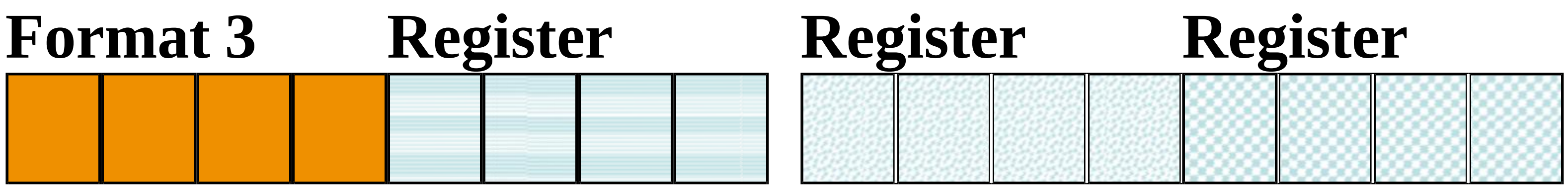
# Format 2 Instruction




1. **Load - Copy the value stored at the Memory Address into the specified register**
2. **Store - Copy the value in the specified register to the Memory Address**
3. **Jump - Compare the contents of the specified register and the contents of Register 0. If equal reset the Program Counter to the Memory Address**

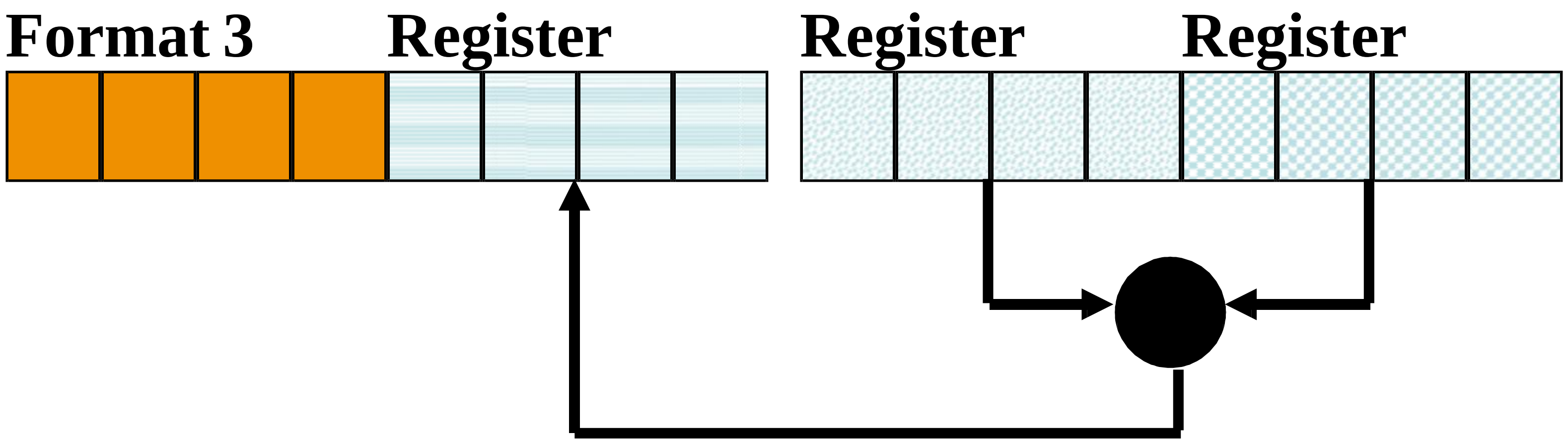
# Format 3 Instruction

## Format 3 Instruction




| Opcode | Instruction | Meaning |
|---|---|---|
| 5 | ADD $R_0, R_1, R_2$ | Load Immediate |
| 6 | AFP $R_0, R_1, R_2$ | Rotate Left |
| 7 | OR $R_0, R_1, R_2$ | Rotate Right |
| 8 | AND $R_0, R_1, R_2$ | Shift Left |
| 9 | XOR $R_0, R_1, R_2$ | Shift Right |

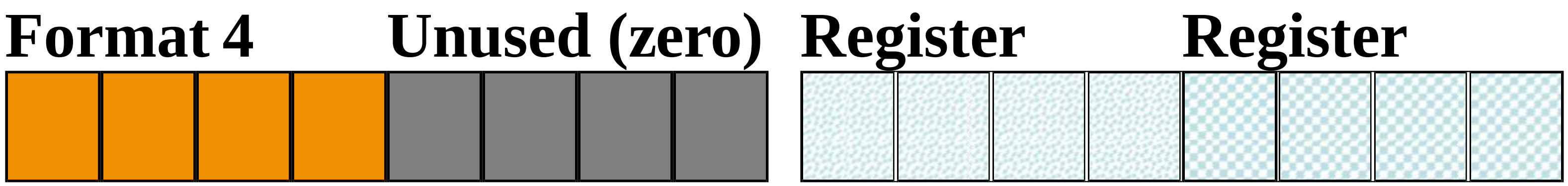
# Format 3 Instruction




Apply the operation to the two values in the registers specified in the Low-Order byte and store the result in the register specified in the High-Order byte

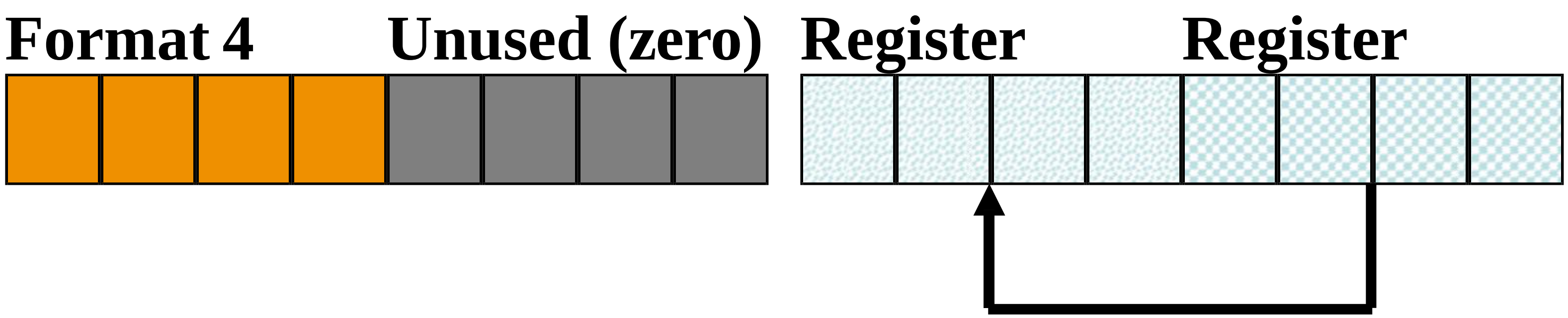
# Format 4 Instruction

## Format 4 Instruction




| Opcode | Instruction | Meaning |
|---|---|---|
| 4 | **LR** $R_1$ , $R_2$ | Load Register |

# Format 4 Instruction




Copy the value in the second register specified in the Low-Order byte to the first register specified in the Low-Order byte

# Full Instruction Set

1. L R , A
2. LI R , I
3. ST R , A
4. LR $R_1$ , $R_2$
5. ADD $R_0$ , $R_1$, $R_2$
6. AFP $R_0$ , $R_1$, $R_2$
7. OR $R_0$ , $R_1$, $R_2$
8. AND $R_0$ , $R_1$, $R_2$
9. XOR $R_0$ , $R_1$, $R_2$

A. RL R , I

B. RR R , I

C. SL R , I

D. SR R , I

E. JMP R , A

F. HALT

# Examples of OpCode

| Name | Comment | Syntax |
|---|---|---|
| | TRANSFER | |
| MOV | Move (copy) | MOV Dest,Source |
| PUSH | Push onto stack | PUSH Source |
| POP | Pop from stack | POP Dest |
| IN | Input | IN Dest, Port |
| OUT | Output | OUT Port, Source |
| | | |
| | ARITHMETIC | |
| ADD | Add | ADD Dest,Source |
| SUB | Subtract | SUB Dest,Source |
| DIV | Divide (unsigned) | DIV Op |
| MUL | Multiply (unsigned) | MUL Op |
| INC | Increment | INC Op |
| DEC | Decrement | DEC Op |
| CMP | Compare | CMP Op1,Op2 |

# Examples of OpCode

| Name | Comment | Syntax |
|---|---|---|
| | LOGIC | |
| NEG | Negate (two-complement) | NEG Op |
| NOT | Invert each bit | NOT Op |
| AND | Logical and | AND Dest,Source |
| OR | Logical or | OR Dest,Source |
| XOR | Logical exclusive or | XOR Dest,Source |
| | | |
| | JUMPS | |
| CALL | Call subroutine | CALL Proc |
| JMP | Jump | JMP Dest |
| JE | Jump if Equal | JE Dest |
| JZ | Jump if Zero | JZ Dest |
| RET | Return from subroutine | RET |
| JNE | Jump if not Equal | JNE Dest |
| JNZ | Jump if not Zero | JNZ Dest |

# Coding Program: Example

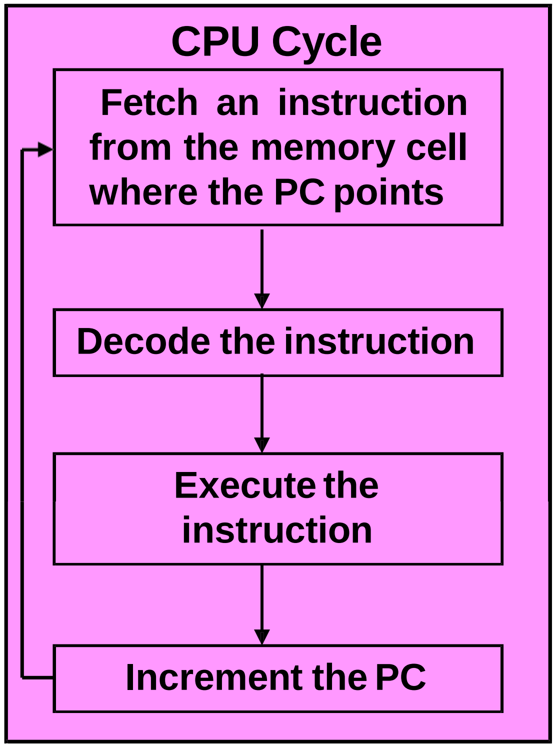
| Assembler | Machine Code | Hexa |
|---|---|---|
| L 1 , 30 | 0001 0001 0011 0000 | 1130 |
| L 2 , 40 | 0001 0010 0100 0000 | 1240 |
| ST 1 , 40 | 0011 0001 0100 0000 | 3140 |
| ST 2 , 30 | 0011 0010 0011 0000 | 3230 |

| Address | Contents |
|---|---|
| 10 | 0001 0001 |
| 11 | 0011 0000 |
| 12 | 0001 0010 |
| 13 | 0100 0000 |
| 14 | 0011 0001 |
| 15 | 0100 0000 |
| 16 | 0011 0010 |
| 17 | 0011 0000 |
| | |
| | |
| | |
| | |
| 30 | 0110 1101 |
| 40 | 1001 1001 |

| Address | Contents | Variable |
|---|---|---|
| | | |
| 30 | 0110 1101 | A |
| | | |
| 40 | 1001 1001 | B |
| | | |

| Register | Contents |
|---|---|
| $R_1$ | 0110 1101 |
| $R_2$ | 1001 1001 |

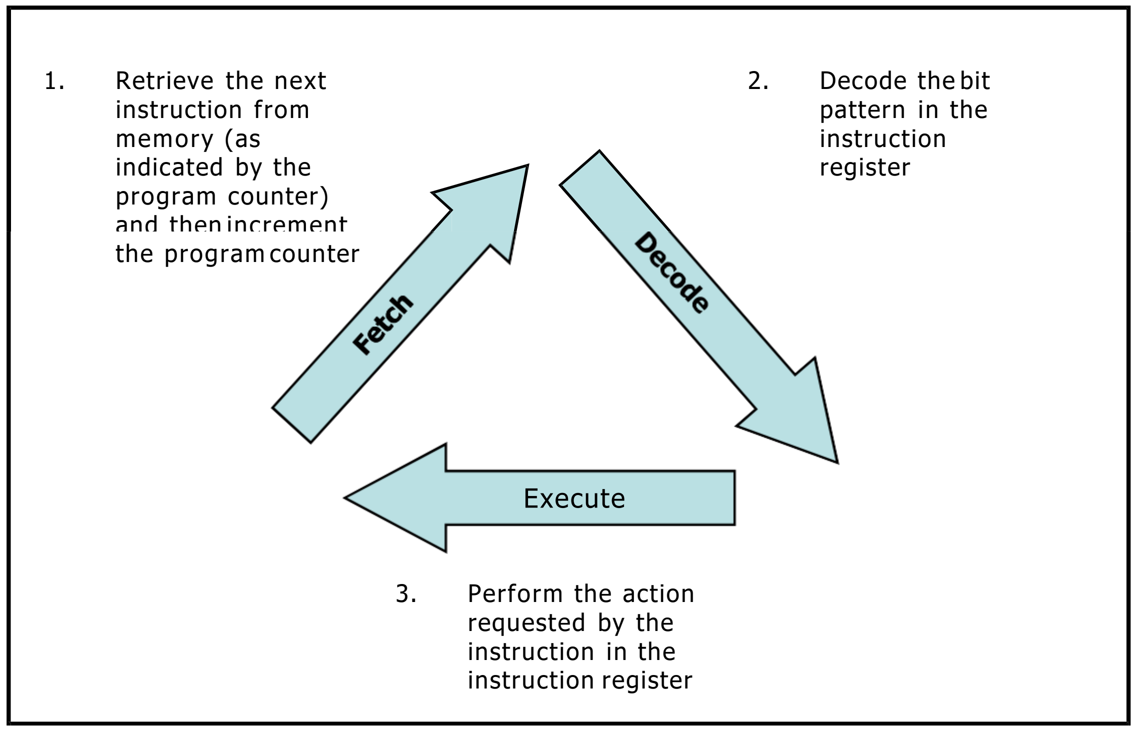
# CPU Cycle (Machine Cycle)




FETCH

DECODE

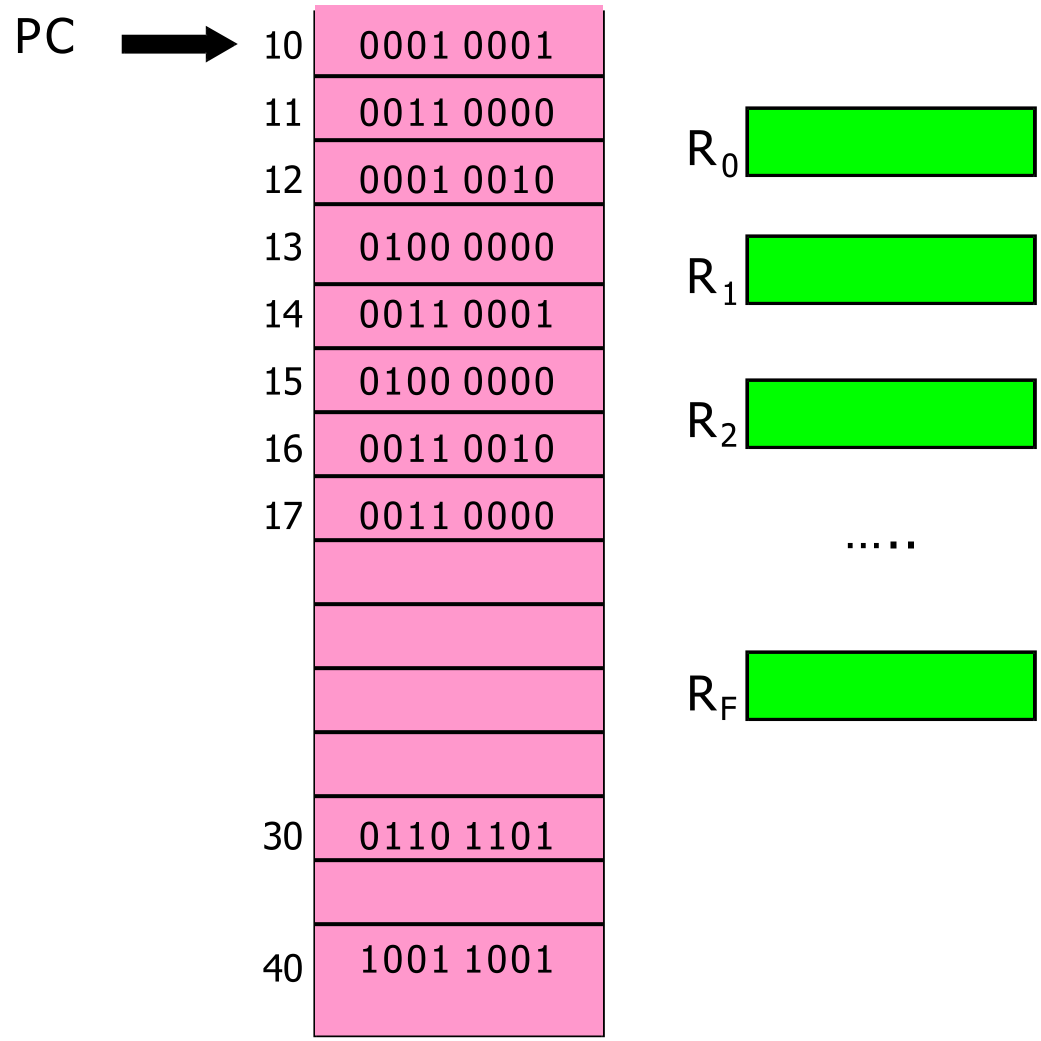
EXECUTE



# Program Execution: Swap Example

- FETCH
- DECODE
- EXECUTE

| | | |
|---|---|---|
| L | 1 , 30 | 1130 |
| L | 2 , 40 | 1240 |
| ST | 1 , 40 | 3140 |
| ST | 2 , 30 | 3230 |



# Execute a Program

- **FETCH**
- **DECODE**
- **EXECUTE**

Instruction:

0001 0001 0011 0000

PC → 10

| | |
|---|---|
| 10 | 0001 0001 |
| 11 | 0011 0000 |
| 12 | 0001 0010 |
| 13 | 0100 0000 |
| 14 | 0011 0001 |
| 15 | 0100 0000 |
| 16 | 0011 0010 |
| 17 | 0011 0000 |
| | |
| | |
| | |
| | |
| 30 | 0110 1101 |
| 40 | 1001 1001 |

$R_0$

$R_1$

$R_2$

.....

$R_F$

L 1 , 30
L 2 , 40
ST 1 , 40
ST 2 , 30

# Execute a Program

- FETCH
- DECODE
- EXECUTE

PC → 10

| Address | Contents |
|---|---|
| 10 | 0001 0001 |
| 11 | 0011 0000 |
| 12 | 0001 0010 |
| 13 | 0100 0000 |
| 14 | 0011 0001 |
| 15 | 0100 0000 |
| 16 | 0011 0010 |
| 17 | 0011 0000 |
| | |
| | |
| | |
| | |
| 30 | 0110 1101 |
| | |
| 40 | 1001 1001 |
| | |

$R_0$

$R_1$

$R_2$

.....

$R_F$

Instruction:

0001 0001 0011 0000

Operation-code : 0001

Register : 0001

Memory address : 0011 0000

L 1 , 30
L 2 , 40
ST 1 , 40
ST 2 , 30

UCSC



BIT 

# Execute a Program

- FETCH
- DECODE
- EXECUTE

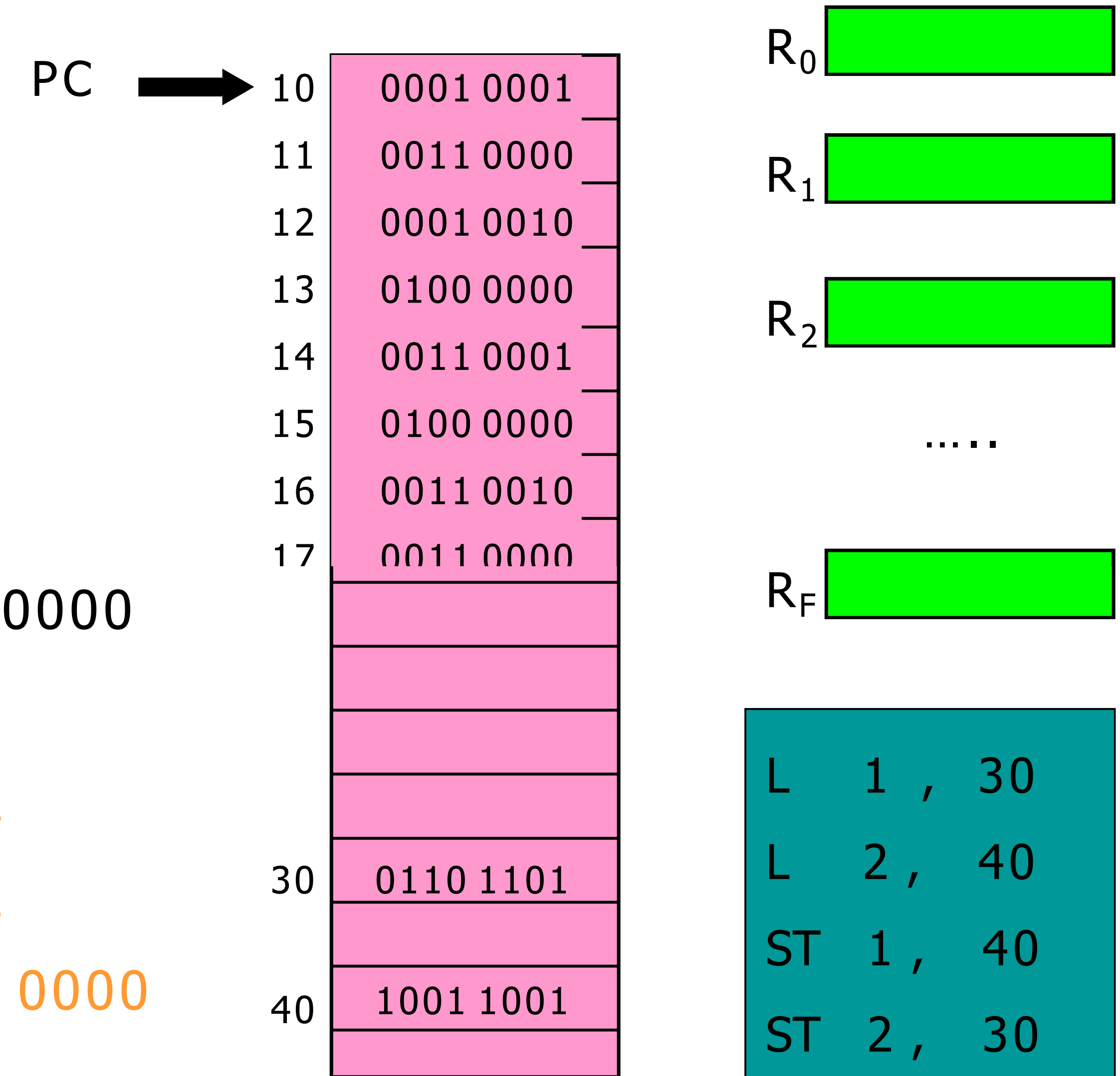
Instruction:

0001 0001 0011 0000

Operation-code : 0001

Register : 0001

Memory address : 0011 0000




```
L  1 , 30
L  2 , 40
ST 1 , 40
ST 2 , 30
```

# Execute a Program

- FETCH
- DECODE
- EXECUTE

PC → 10

| | |
|---|---|
| 10 | 0001 0001 |
| 11 | 0011 0000 |
| 12 | 0001 0010 |
| 13 | 0100 0000 |
| 14 | 0011 0001 |
| 15 | 0100 0000 |
| 16 | 0011 0010 |
| 17 | 0011 0000 |
| 30 | 0110 1101 |
| 40 | 1001 1001 |

$R_0$

$R_1$ 0110 1101

$R_2$

.....

$R_F$

Instruction:

0001 0001 0011 0000

Operation-code : 0001

Register : 0001

Memory address : 0011 0000

```
L   1 , 30
L   2 , 40
ST  1 , 40
ST  2 , 30
```

# Execute a Program

- **FETCH**
- **DECODE**
- **EXECUTE**

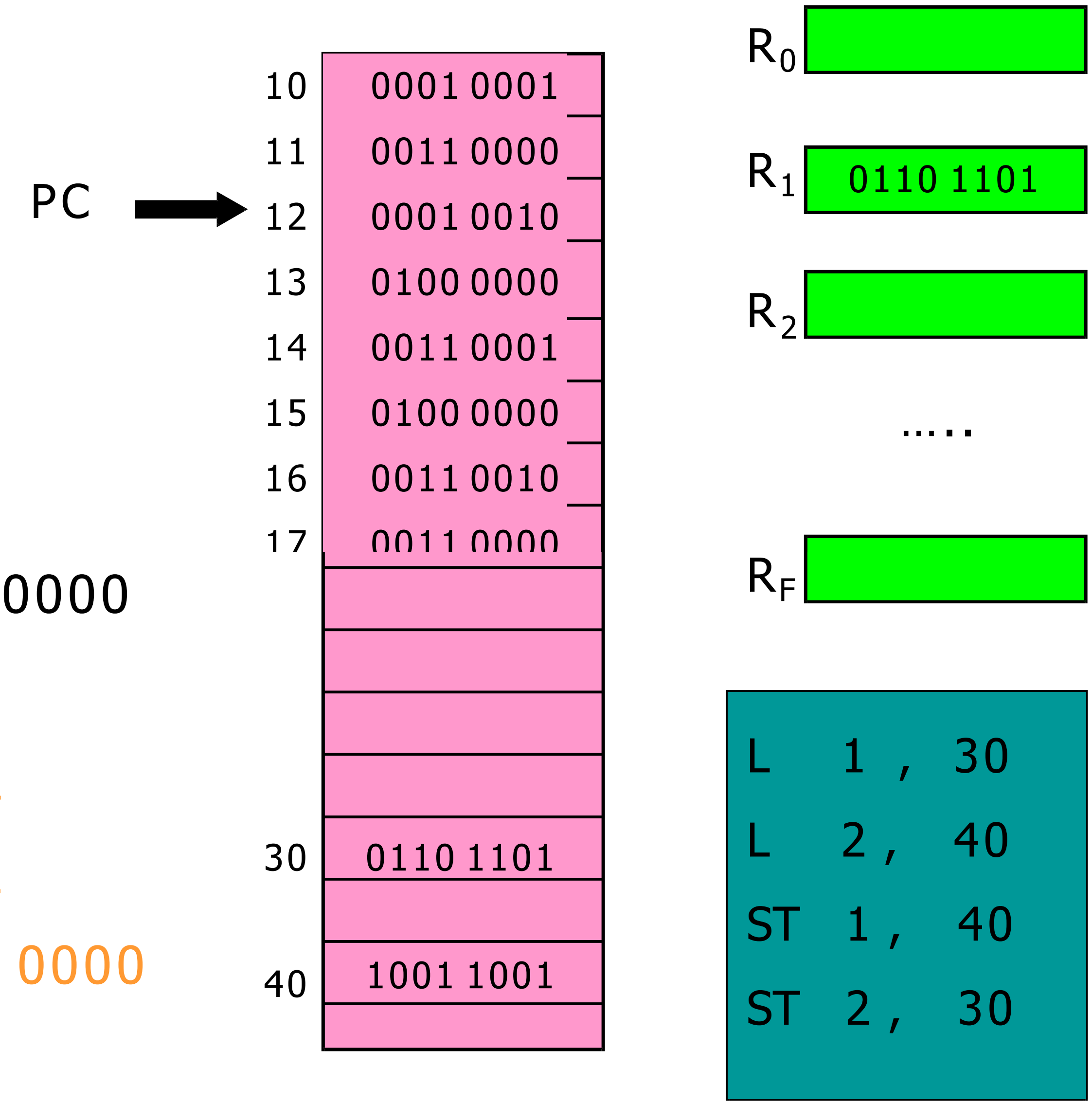
Instruction:

0001 0001 0011 0000

Operation-code : 0001

Register : 0001

Memory address : 0011 0000

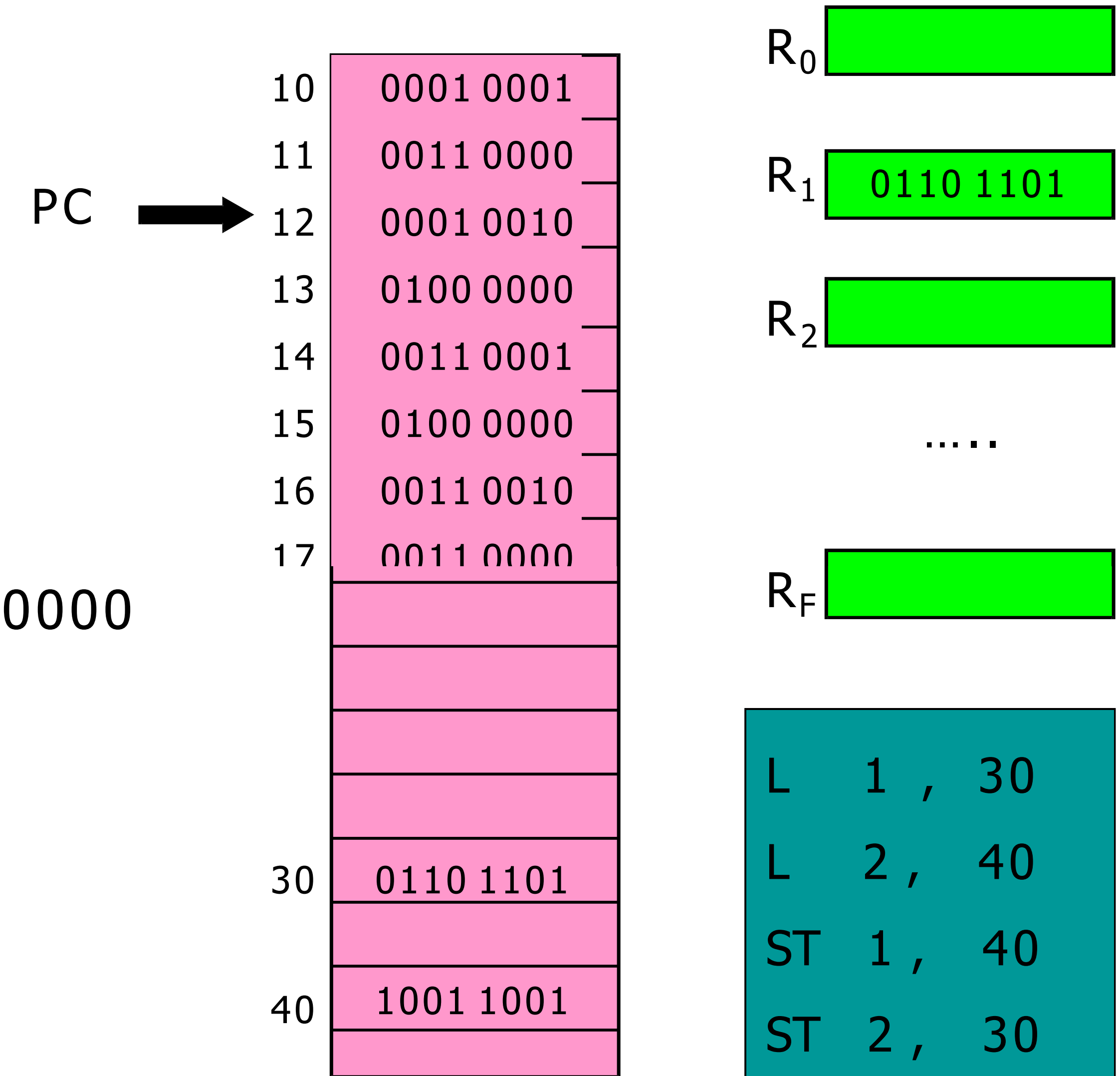


# Execute a Program

- FETCH
- DECODE
- EXECUTE

Instruction:

0001 0010 0100 0000




```
L   1 , 30
L   2 , 40
ST  1 , 40
ST  2 , 30
```

# Execute a Program

- FETCH
- DECODE
- EXECUTE

PC → 12

| | |
|---|---|
| 10 | 0001 0001 |
| 11 | 0011 0000 |
| 12 | 0001 0010 |
| 13 | 0100 0000 |
| 14 | 0011 0001 |
| 15 | 0100 0000 |
| 16 | 0011 0010 |
| 17 | 0011 0000 |
| | |
| | |
| | |
| | |
| 30 | 0110 1101 |
| | |
| 40 | 1001 1001 |
| | |

$R_0$

$R_1$ 0110 1101

$R_2$

.....

$R_F$

Instruction:

0001 0010 0100 0000

Operation-code : 0001

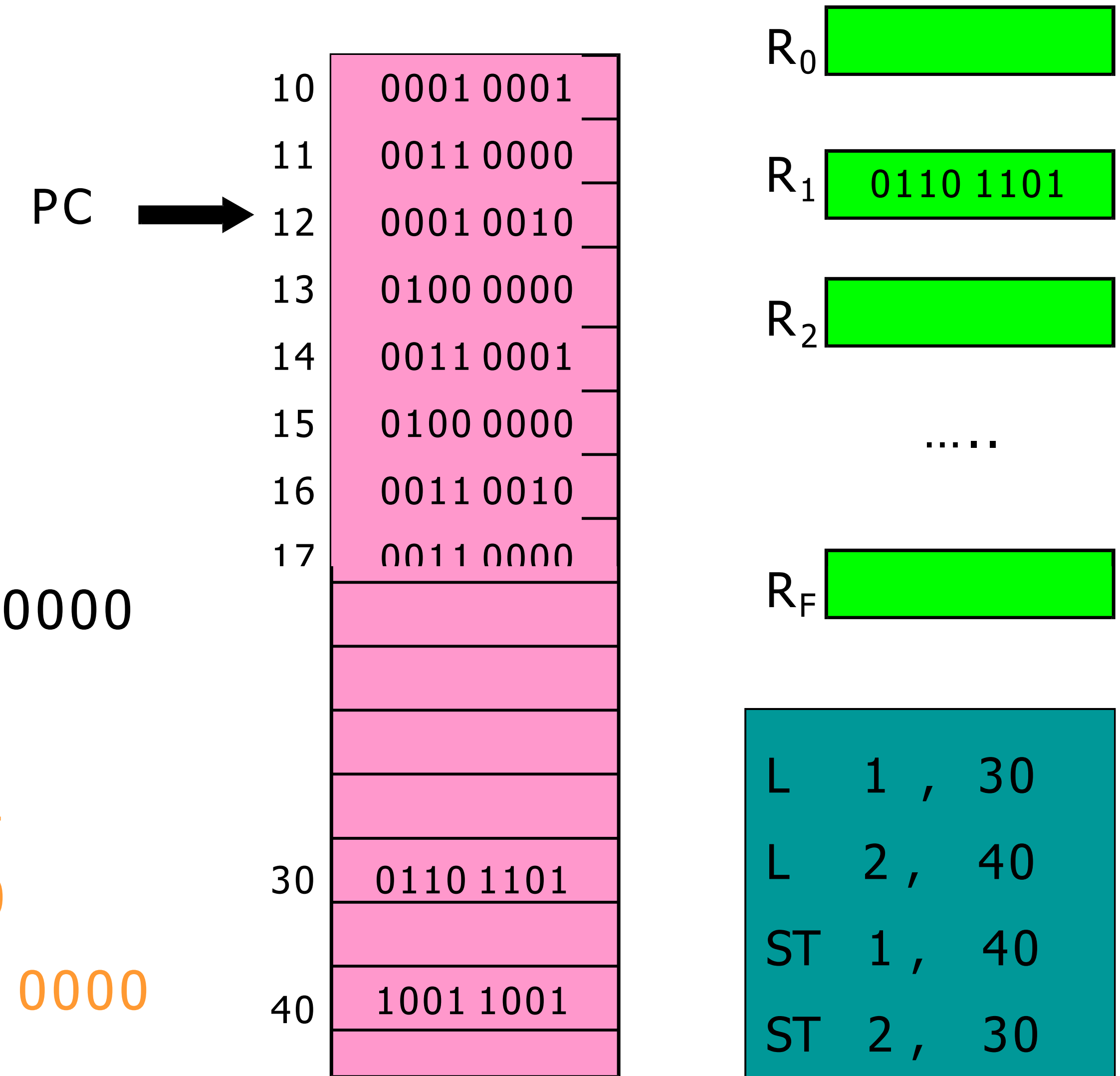
Register : 0010

Memory address : 0100 0000

```
L   1 , 30
L   2 , 40
ST  1 , 40
ST  2 , 30
```

# Execute a Program

- FETCH
- DECODE
- EXECUTE




Instruction:

0001 0010 0100 0000

Operation-code : 0001

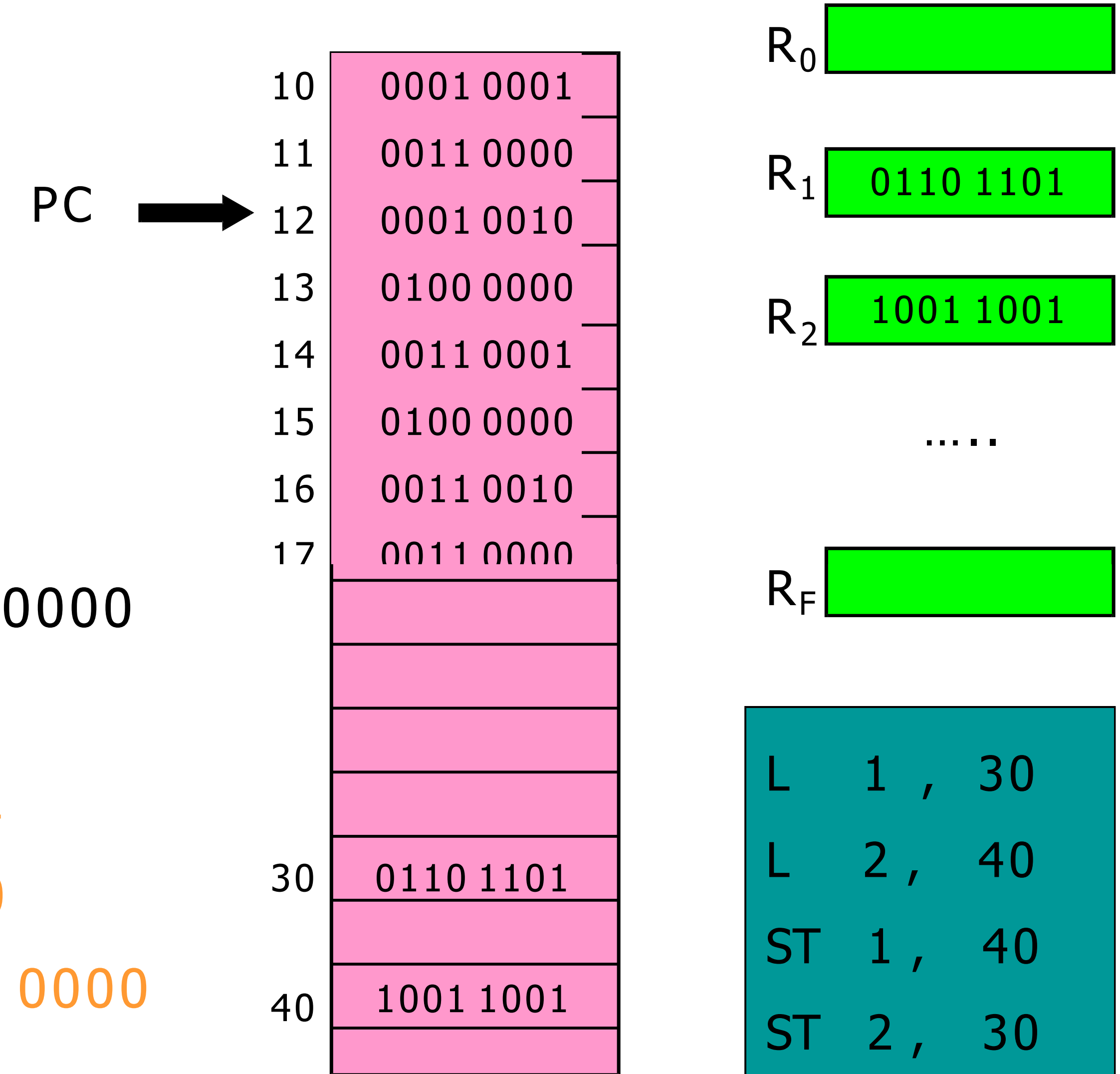
Register : 0010

Memory address : 0100 0000

```
L   1 , 30
L   2 , 40
ST  1 , 40
ST  2 , 30
```

# Execute a Program

- FETCH
- DECODE
- EXECUTE




Instruction:

0001 0010 0100 0000

Operation-code : 0001

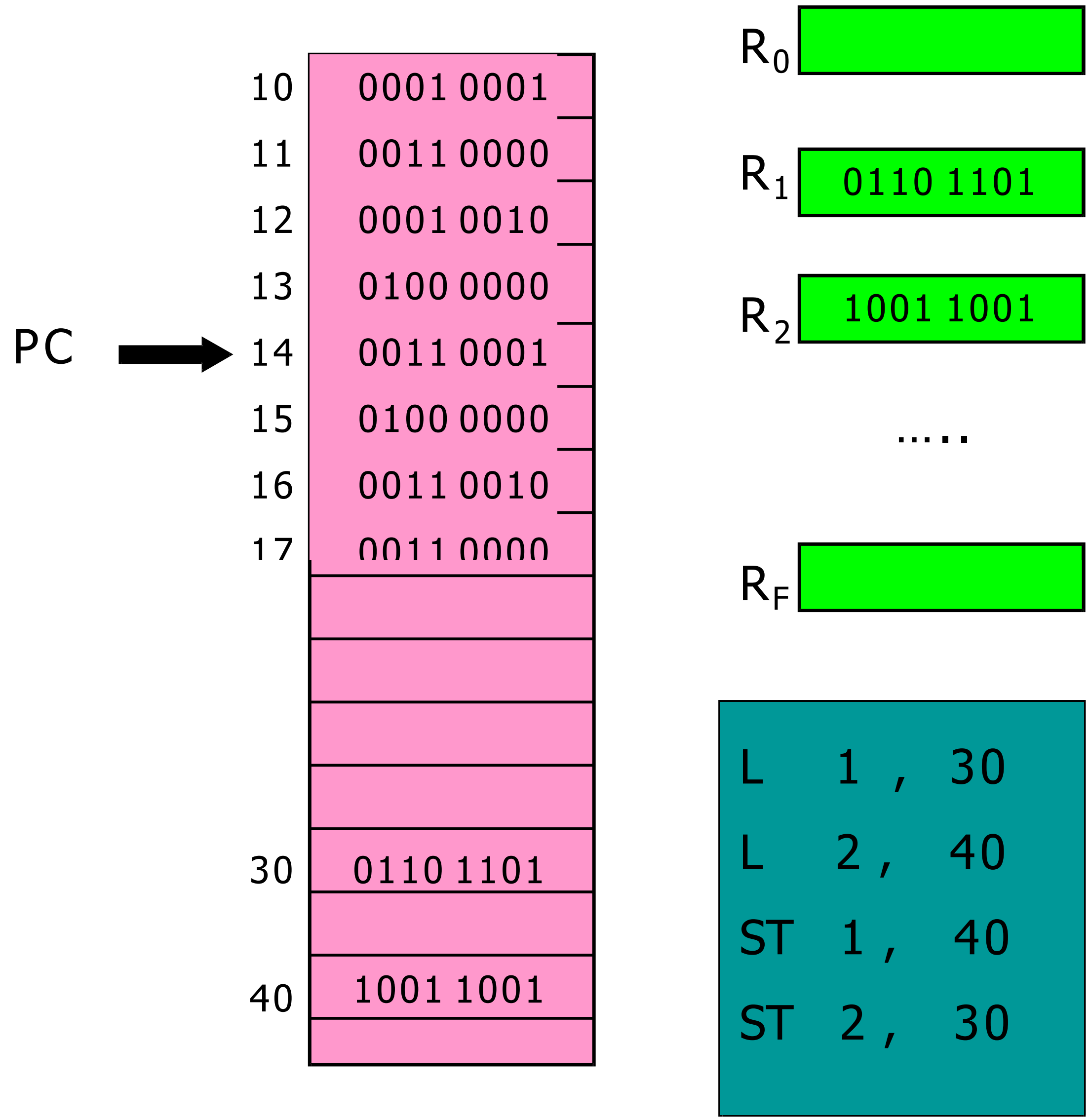
Register : 0010

Memory address : 0100 0000

| | |
|---|---|
| L | 1 , 30 |
| L | 2 , 40 |
| ST | 1 , 40 |
| ST | 2 , 30 |

# Execute a Program

- FETCH
- DECODE
- EXECUTE

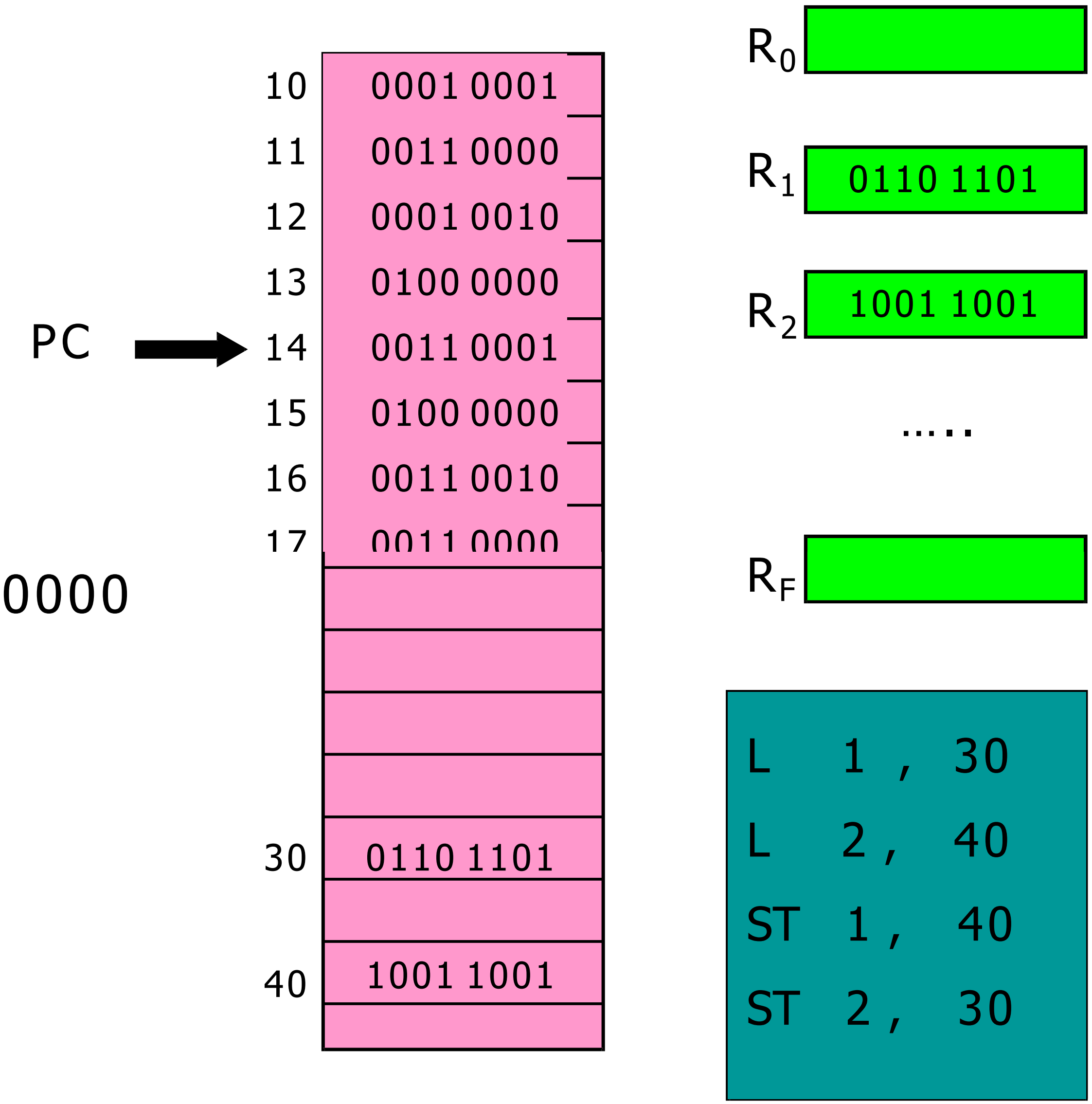


# Execute a Program

- FETCH
- DECODE
- EXECUTE

Instruction:

0011 0001 0100 0000



# Execute a Program

- FETCH
- DECODE
- EXECUTE

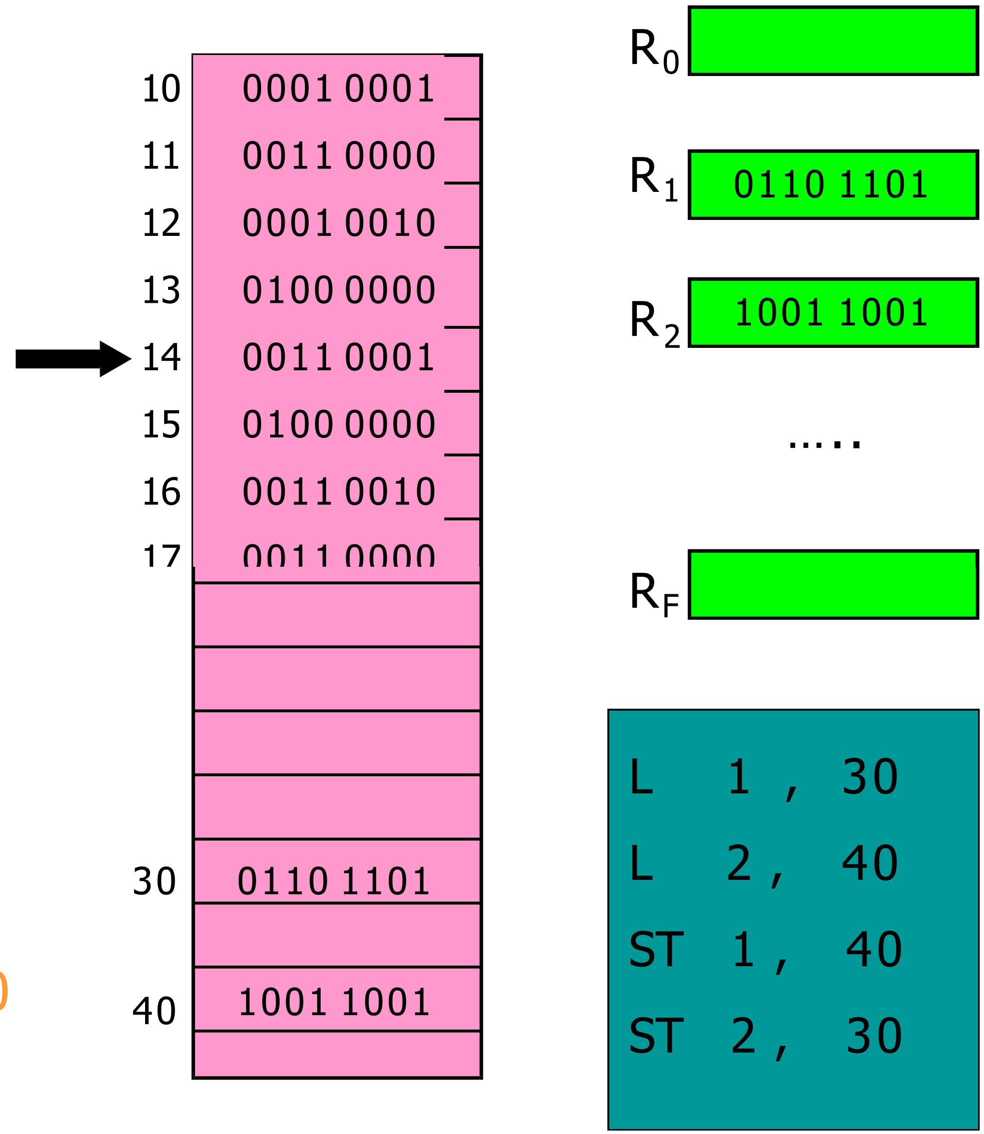
Instruction:

0011 0001 0100 0000

Operation-code : 0011

Register : 0001

Memory address : 0100 0000



# Execute a Program

- FETCH
- DECODE
- EXECUTE

Instruction:

0011 0001 0100 0000

Operation-code : 0011

Register : 0001

Memory address : 0100 0000

PC → 14

| | |
|---|---|
| 10 | 0001 0001 |
| 11 | 0011 0000 |
| 12 | 0001 0010 |
| 13 | 0100 0000 |
| 14 | 0011 0001 |
| 15 | 0100 0000 |
| 16 | 0011 0010 |
| 17 | 0011 0000 |
| | |
| | |
| | |
| | |
| 30 | 0110 1101 |
| | |
| 40 | 1001 1001 |
| | |

| | |
|---|---|
| $R_0$ | |
| $R_1$ | 0110 1101 |
| $R_2$ | 1001 1001 |
| ….. | |
| $R_F$ | |

```
L   1 , 30
L   2 , 40
ST  1 , 40
ST  2 , 30
```

# Execute a Program

- FETCH
- DECODE
- EXECUTE

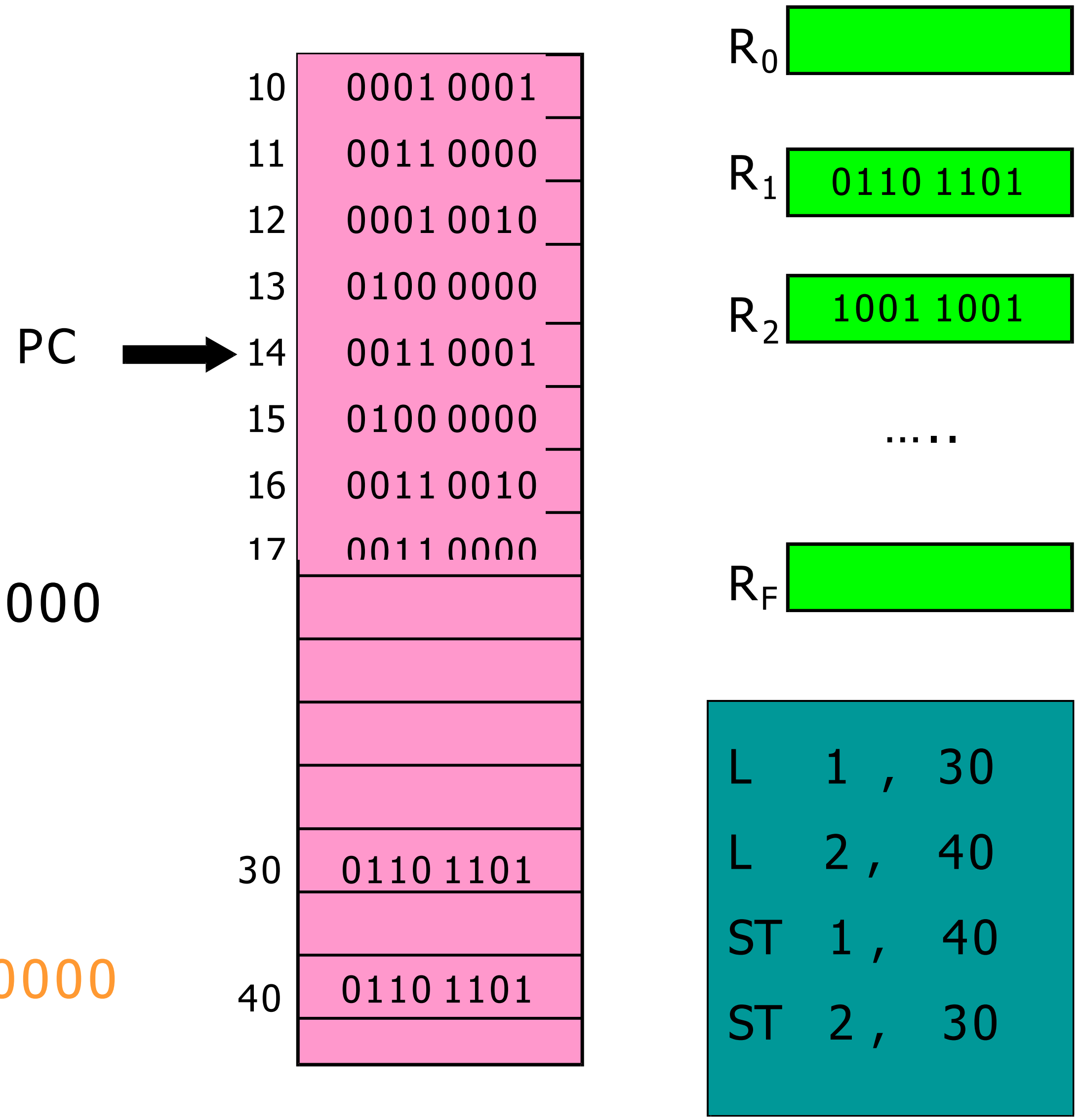
Instruction:

0011 0001 0100 0000

Operation-code : 0011

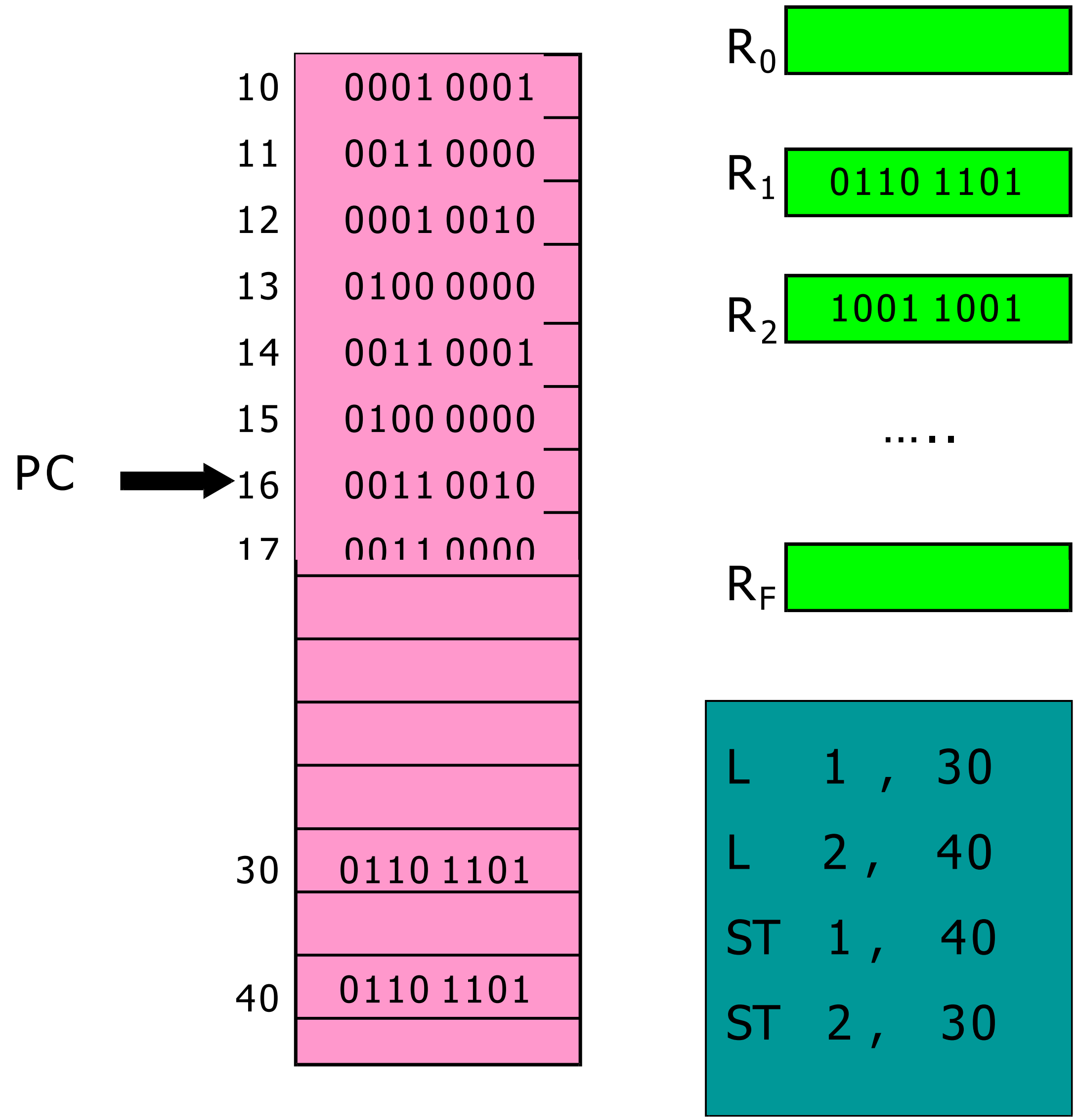
Register : 0001

Memory address : 0100 0000



# Execute a Program

- **FETCH**
- **DECODE**
- **EXECUTE**



# Execute a Program

- FETCH
- DECODE
- EXECUTE

Instruction:

0011 0010 0011 0000

PC → 16

| | |
|---|---|
| 10 | 0001 0001 |
| 11 | 0011 0000 |
| 12 | 0001 0010 |
| 13 | 0100 0000 |
| 14 | 0011 0001 |
| 15 | 0100 0000 |
| 16 | 0011 0010 |
| 17 | 0011 0000 |
| | |
| | |
| | |
| | |
| 30 | 0110 1101 |
| | |
| 40 | 0110 1101 |
| | |

| | |
|---|---|
| $R_0$ | |
| $R_1$ | 0110 1101 |
| $R_2$ | 1001 1001 |
| ….. | |
| $R_F$ | |

```
L   1 , 30
L   2 , 40
ST  1 , 40
ST  2 , 30
```

# Execute a Program

- FETCH
- DECODE
- EXECUTE

Instruction:

0011 0010 0011 0000

Operation-code : 0011

Register : 0010

Memory address : 0011 0000

PC → 16

| | |
|---|---|
| 10 | 0001 0001 |
| 11 | 0011 0000 |
| 12 | 0001 0010 |
| 13 | 0100 0000 |
| 14 | 0011 0001 |
| 15 | 0100 0000 |
| 16 | 0011 0010 |
| 17 | 0011 0000 |
| | |
| | |
| | |
| | |
| 30 | 0110 1101 |
| 40 | 0110 1101 |

| | |
|---|---|
| $R_0$ | |
| $R_1$ | 0110 1101 |
| $R_2$ | 1001 1001 |
| ….. | |
| $R_F$ | |

L 1 , 30
L 2 , 40
ST 1 , 40
ST 2 , 30

# Execute a Program

- **FETCH**
- **DECODE**
- **EXECUTE**

Instruction:

0011 0010 0011 0000

PC → 16

| | |
|---|---|
| 10 | 0001 0001 |
| 11 | 0011 0000 |
| 12 | 0001 0010 |
| 13 | 0100 0000 |
| 14 | 0011 0001 |
| 15 | 0100 0000 |
| 16 | 0011 0010 |
| 17 | 0011 0000 |
| | |
| | |
| | |
| | |
| 30 | 0110 1101 |
| | |
| 40 | 0110 1101 |
| | |

| | |
|---|---|
| $R_0$ | |
| $R_1$ | 0110 1101 |
| $R_2$ | 1001 1001 |
| ….. | |
| $R_F$ | |

Operation-code : 0011

Register : 0010

Memory address : 0011 0000

```
L   1 , 30
L   2 , 40
ST  1 , 40
ST  2 , 30
```

# Execute a Program

- FETCH
- DECODE
- EXECUTE

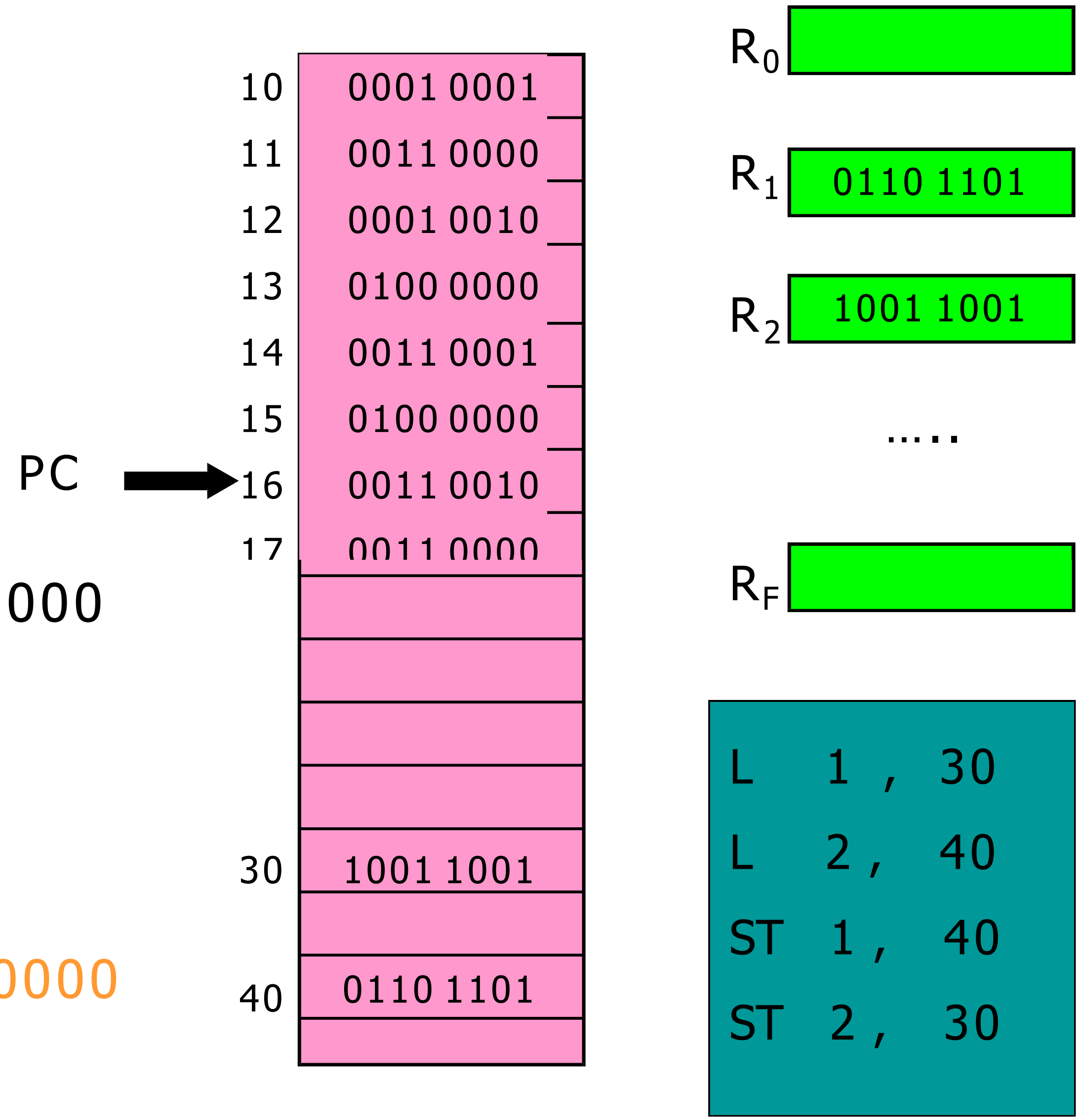
Instruction:

0011 0010 0011 0000

Operation-code : 0011

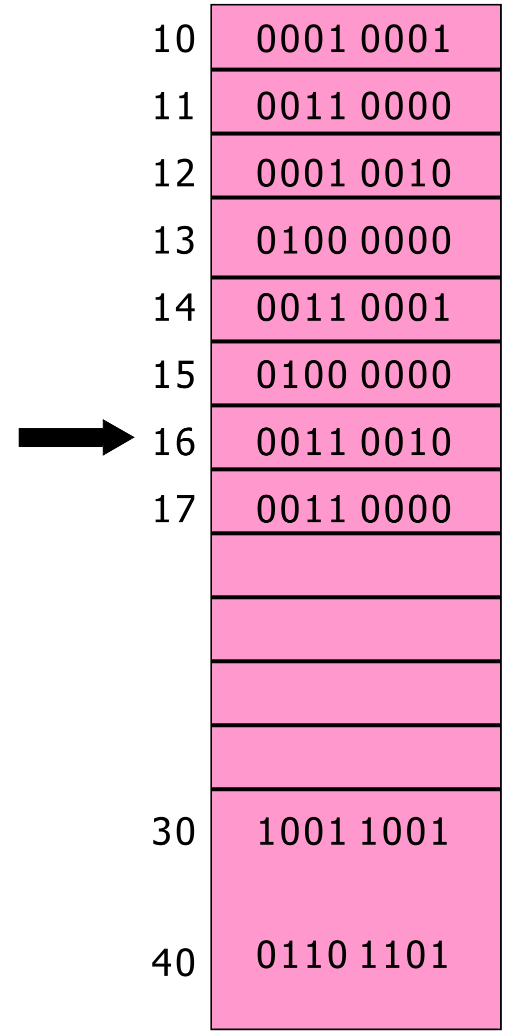
Register : 0010

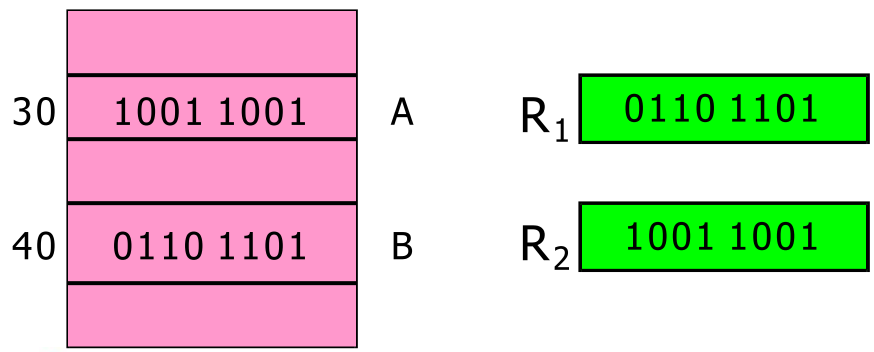
Memory address : 0011 0000



# Coding Program: An Example

| Assembler | Machine Code | Hexa |
|---|---|---|
| L 1 , 30 | 0001 0001 0011 0000 | 1130 |
| L 2 , 40 | 0001 0010 0100 0000 | 1240 |
| ST 1 , 40 | 0011 0001 0100 0000 | 3140 |
| ST 2 , 30 | 0011 0010 0011 0000 | 3230 |



# Assembler Code for A:=23, B:=-11;

| | |
|---|---|
| LI 1 , 17 | LOAD 23 IN HEX INTO R1 |
| ST 1 , A | STORE VALUE AT A |
| LI 1 , F5 | LOAD -11 IN HEX INTO R1 |
| ST 1 , B | STORE VALUE AT B |

# Machine Code for A:=23, B:=-11;

| | | |
|---|---|---|
| LI 1 , 17 | 2117 | 00100001 00010111 |
| ST 1 , A | 3180 | 00110001 10000000 |
| LI 1 , F5 | 21F5 | 00100001 11110101 |
| ST 1 , B | 3181 | 00110001 10000001 |

# Assembler Code for C:=A-B;

| | |
|---|---|
| L 1 , A | LOAD A INTO R1 |
| L 2 , B | LOAD B INTO R2 |
| LI 3 , FF | SET MASK TO FLIP B |
| XOR 4 , 2 , 3 | FLIP B |
| LI 3 , 01 | LOAD 1 INTO R3 |
| ADD 2 , 3 , 4 | ADD 1 TO FLIPPED B |
| ADD 3 , 1 , 2 | NOW DO R3 = A + B |
| ST 3 , C | STORE R3 AT C |

# Machine Code for C:=A-B;

| | | |
|---|---|---|
| L 1 , A | 1180 | 00010001 10000000 |
| L 2 , B | 1281 | 00010010 10000001 |
| LI 3 , FF | 23FF | 00100011 11111111 |
| XOR 4 ,2,3 | 9423 | 10010100 00100011 |
| LI 3 , 01 | 2301 | 00100011 00000001 |
| ADD 2 ,3,4 | 5234 | 01010010 00110100 |
| ADD 3 ,1,2 | 5312 | 01010011 00010010 |
| ST 3 , C | 3382 | 00110011 10000010 |

# Example Program

```
PROGRAM Sort;
   VAR
         A,B,C : INTEGER;
   PROCEDURE Swap (VAR X,Y : INTEGER);
  VAR
         Temp : INTEGER;
BEGIN {Swap}
  Temp := A;
  A := B;
  B  :=  Temp;
END {Swap};
BEGIN {Sort}
  C := A-B;
  IF C = 0 THEN
      Swap (A,B);
END {Sort}.
```

UCSC

BIT

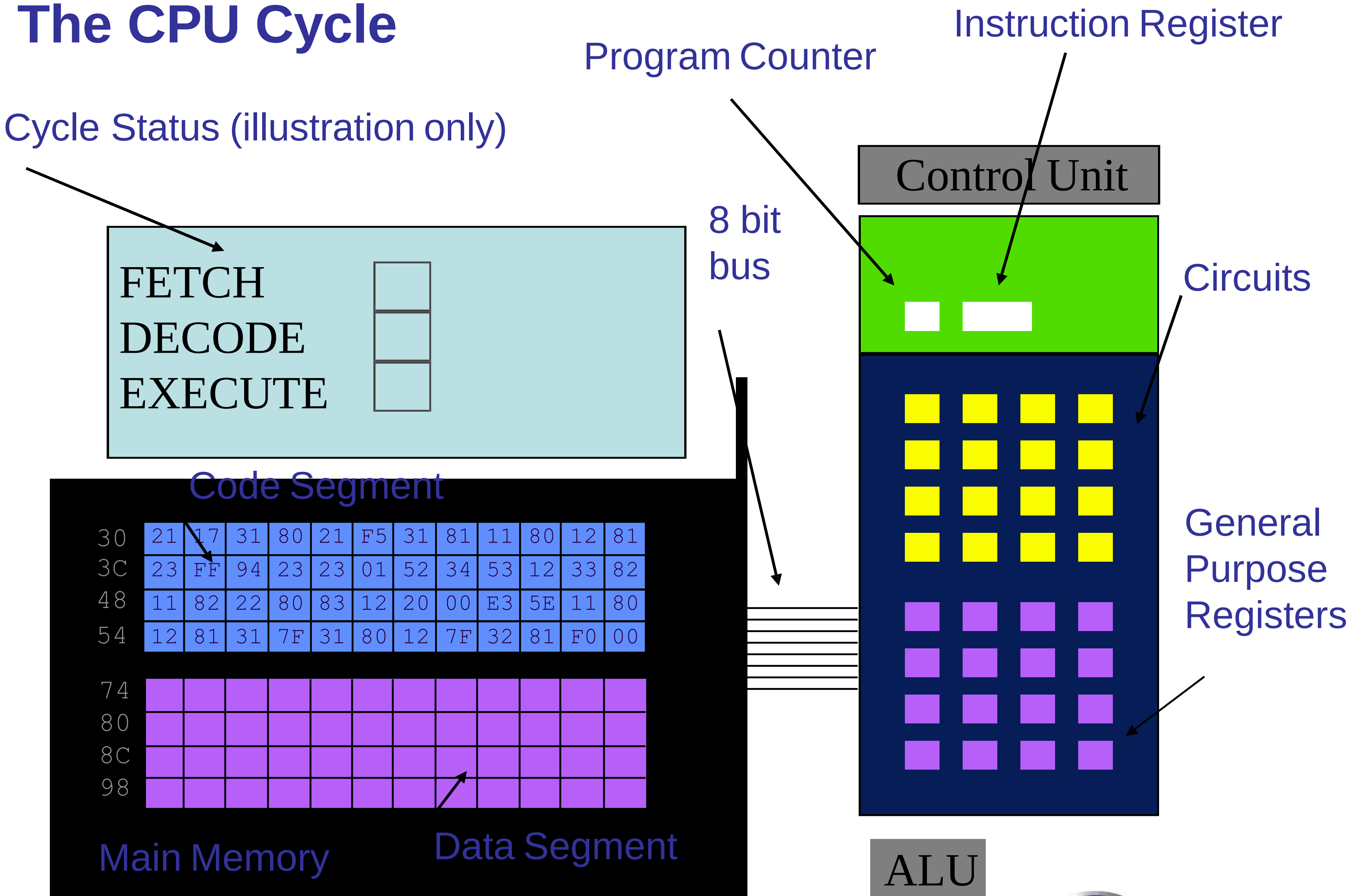
# Assembler and Machine Code

```
30   LI  1,17      2117
32   ST  1,A       3180
34   LI  1,F5      21F5
36   ST  1,B       3181
38   L   1,A       1180
3A   L   2,B       1281
3C   LI  3,FF      23FF
3E   XOR 4,2,3     9423
40   LI  3,01      2301
42   ADD 2,3,4     5234
44   ADD 3,1,2     5312
46   ST  3,C       3382
48   L   1,C       1182
4A   LI  2,80      2280
4C   AND 3,1,2     8312
4E   LI  0,00      2000
50   JMP 3,5E      E35E
52   L   1,A       1180
54   L   2,B       1281
56   ST  1,TEMP    317F
5C   ST  2,B       3281
5E   HALT          F000
```

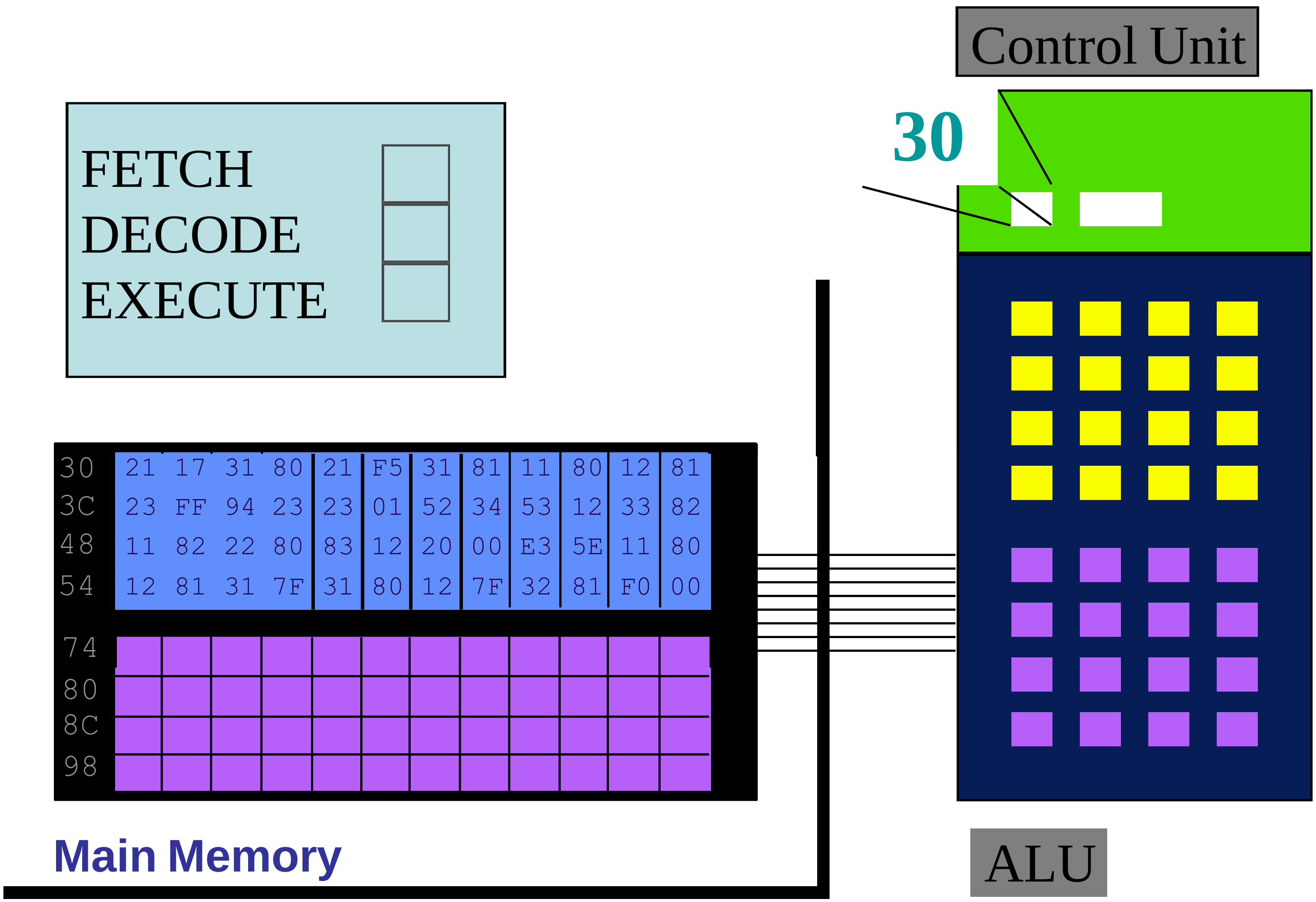
# Code Loaded in Memory

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 30 | 21 | 17 | 31 | 80 | 21 | F5 | 31 | 81 | 11 | 80 | 12 | 81 |
| 3C | 23 | FF | 94 | 23 | 23 | 01 | 52 | 34 | 53 | 12 | 33 | 82 |
| 48 | 11 | 82 | 22 | 80 | 83 | 12 | 20 | 00 | E3 | 5E | 11 | 80 |
| 54 | 12 | 81 | 31 | 7F | 31 | 80 | 12 | 7F | 32 | 81 | F0 | 00 |
| 74 | | | | | | | | | | | | |
| 80 | | | | | | | | | | | | |
| 8C | | | | | | | | | | | | |
| 98 | | | | | | | | | | | | |

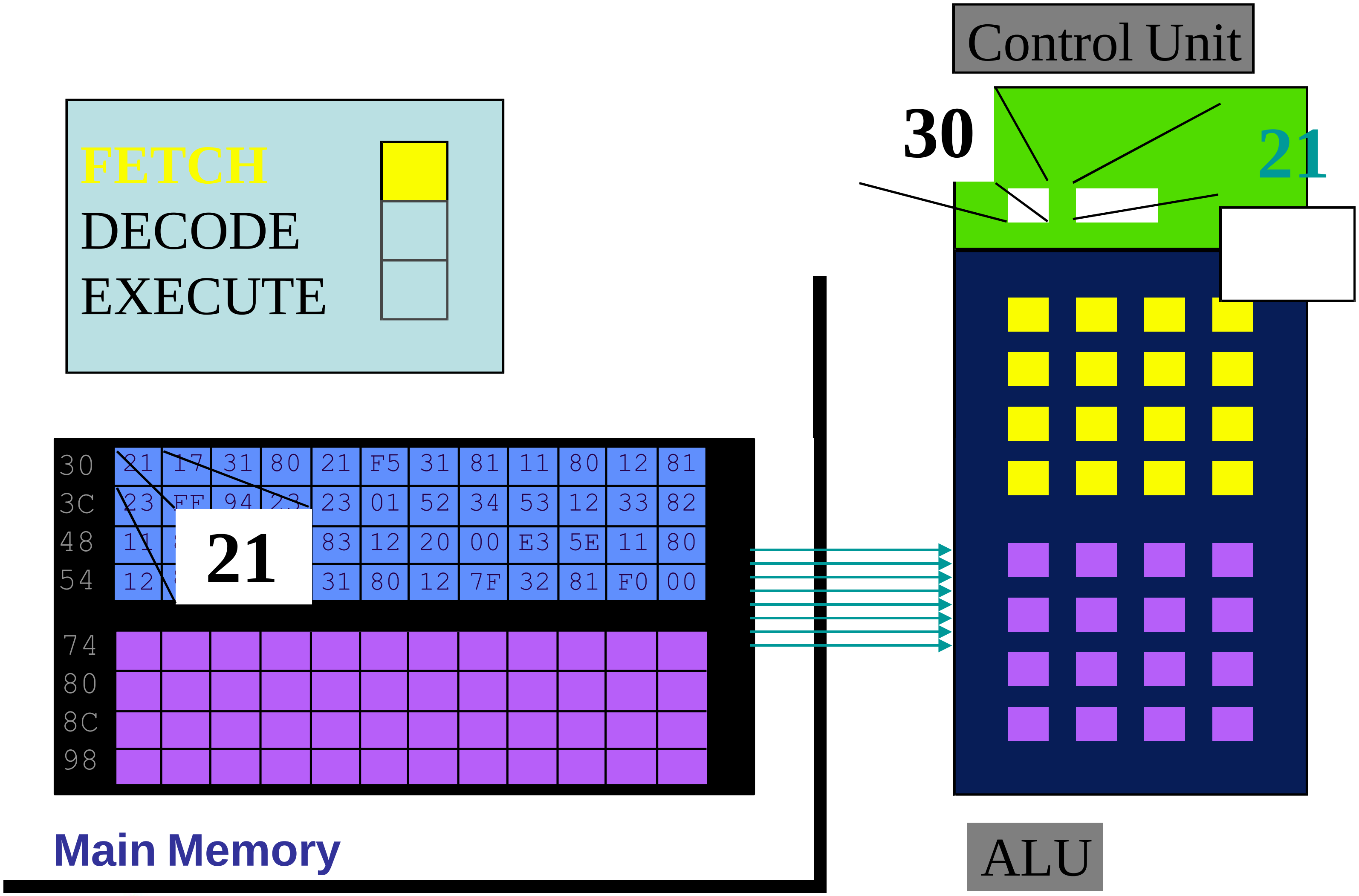
# The CPU Cycle



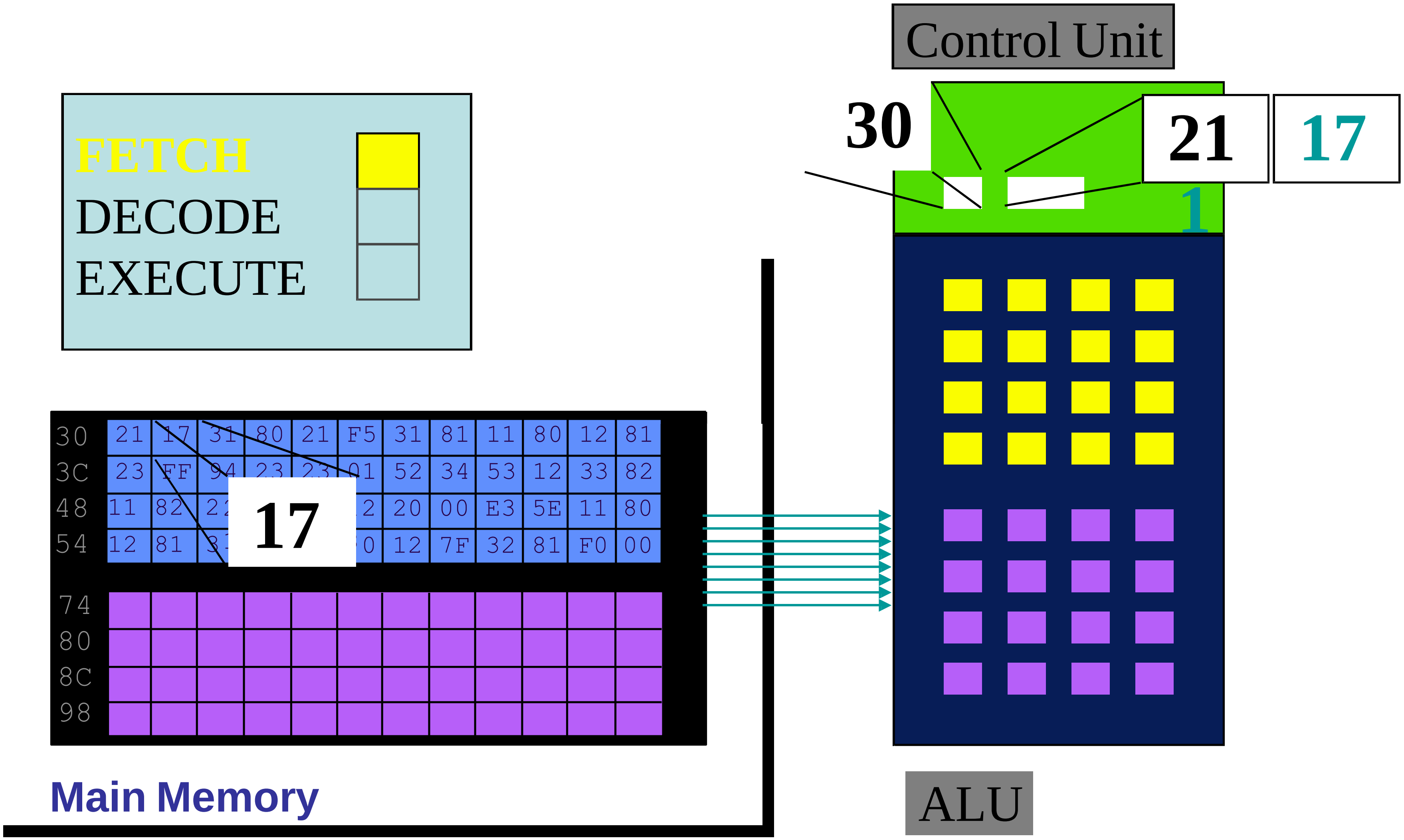
# The CPU Cycle



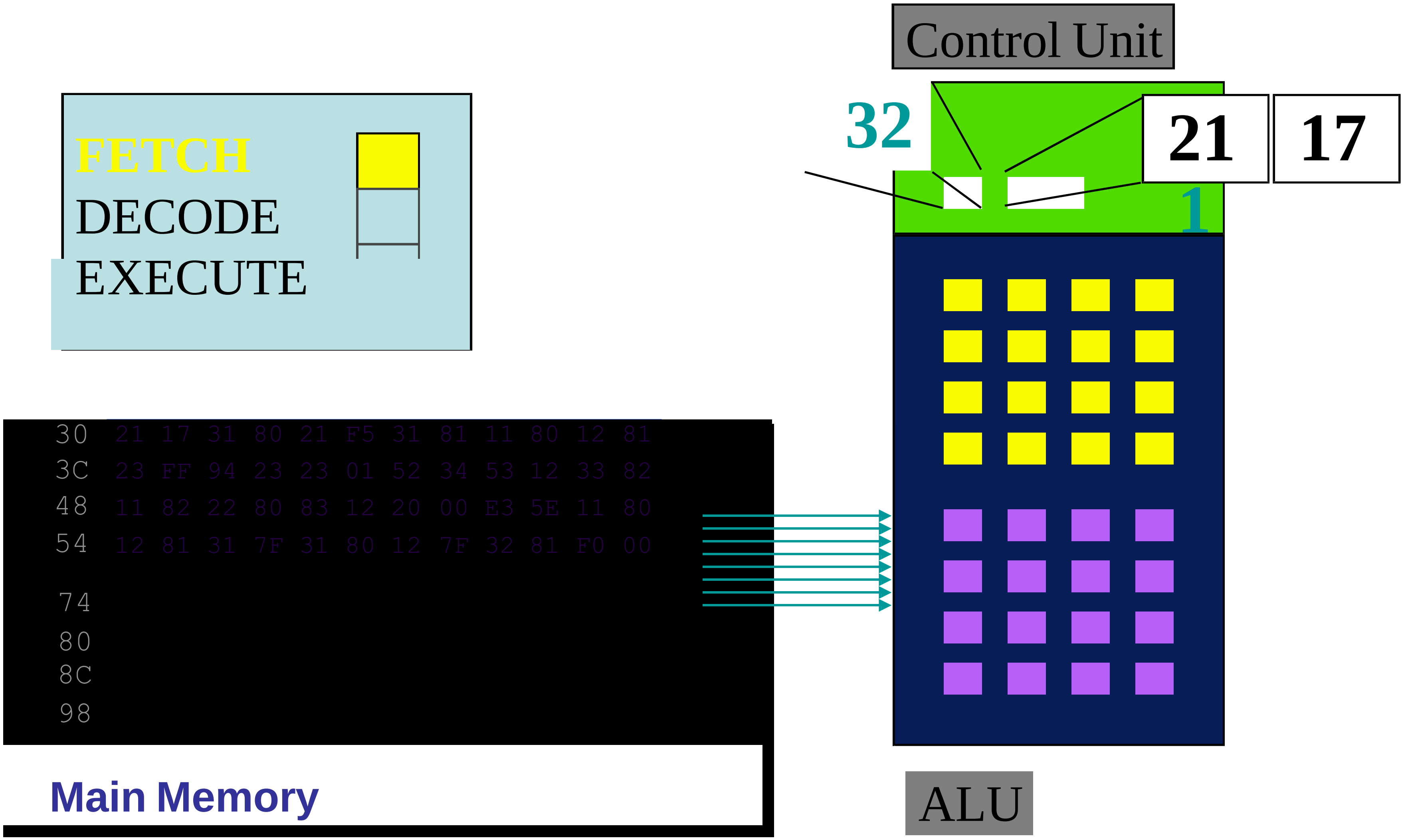
# The CPU Cycle



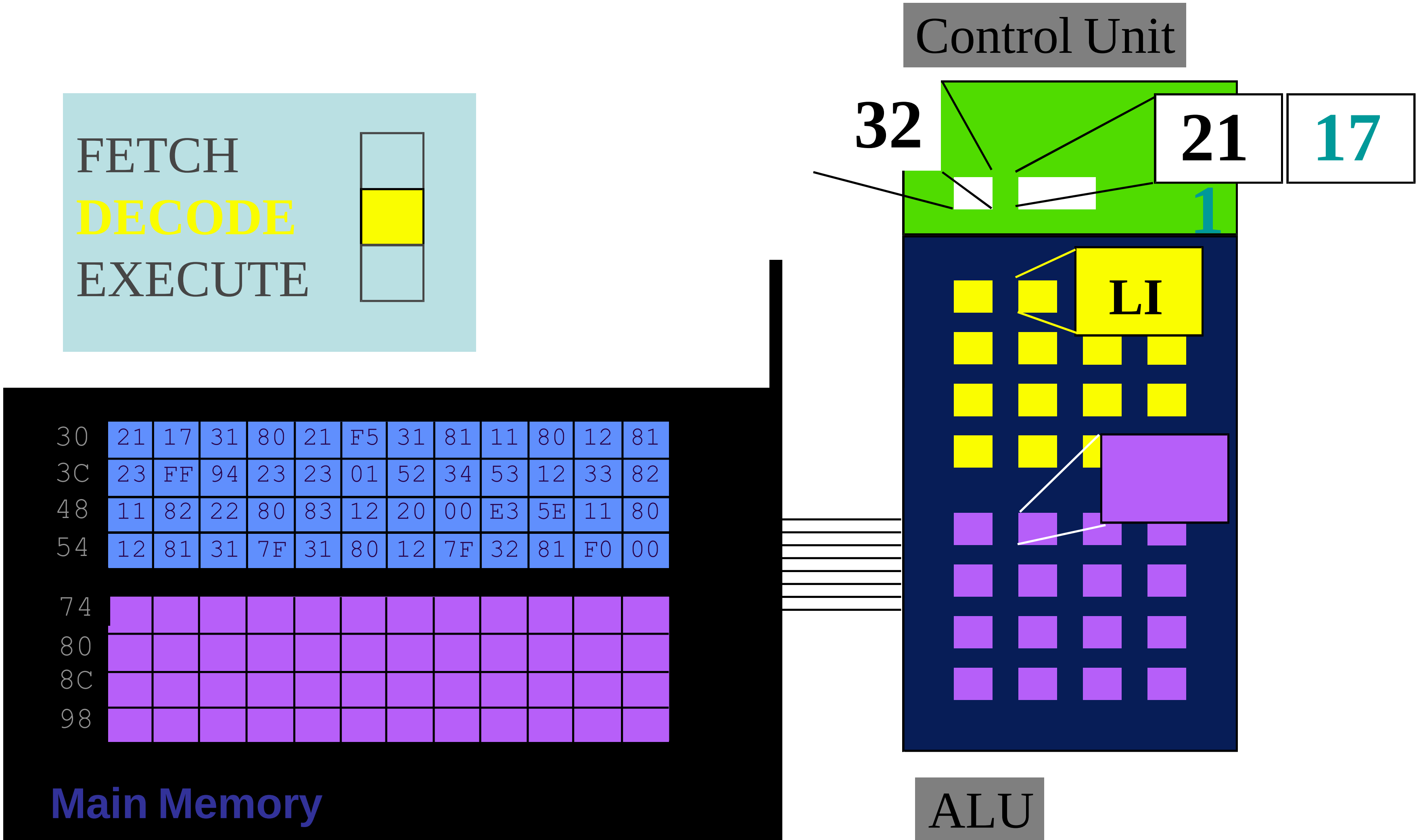
# The CPU Cycle



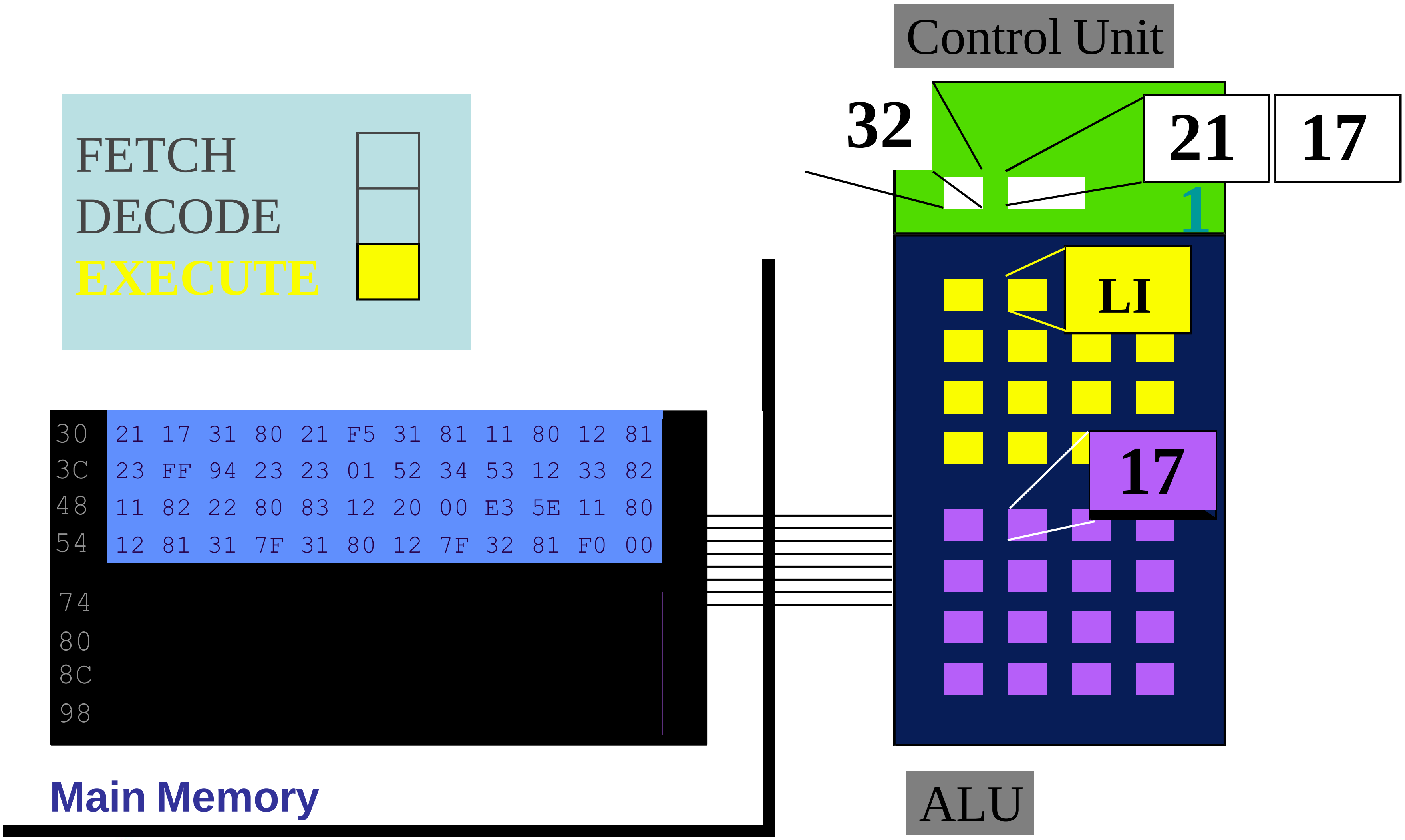
# The CPU Cycle



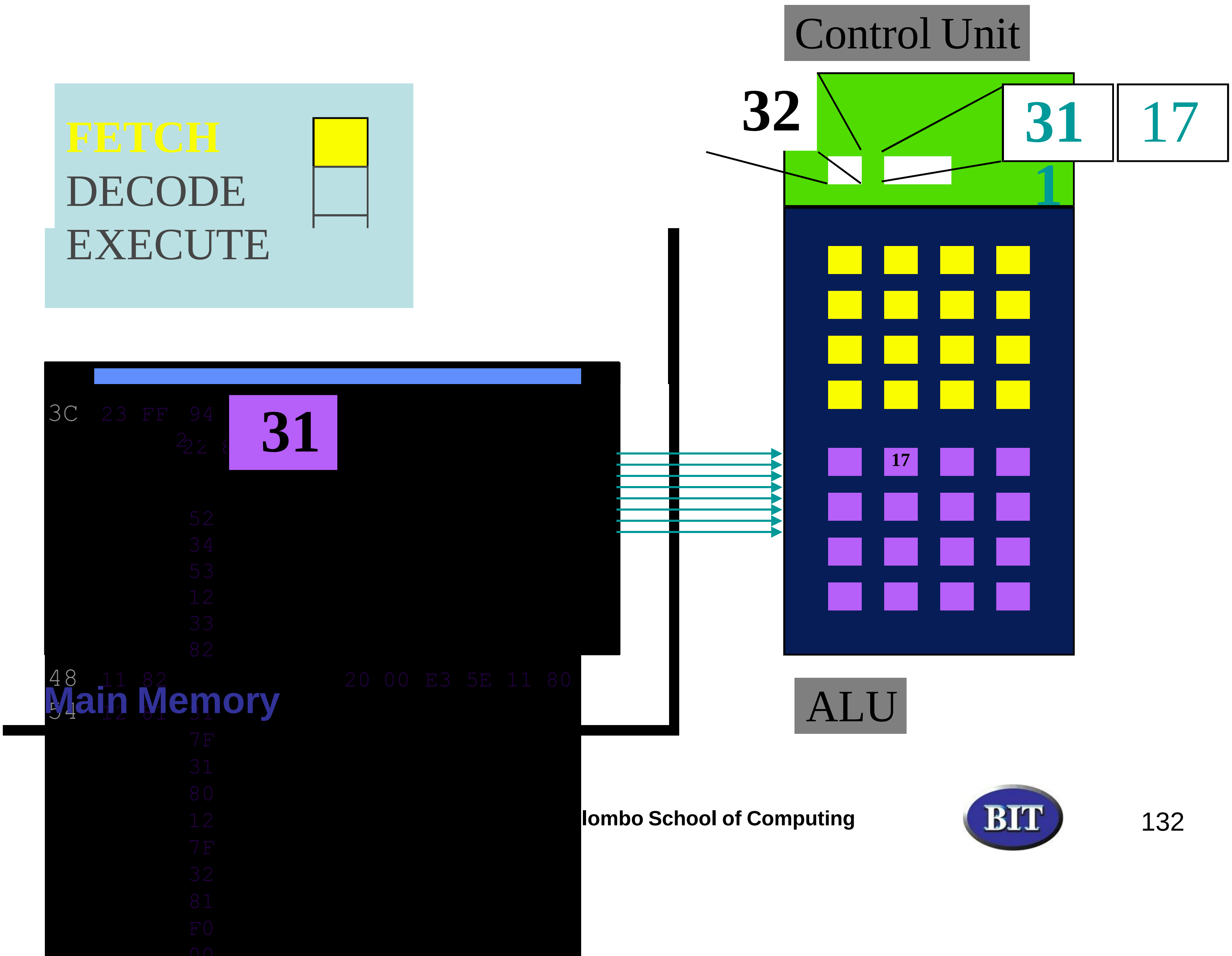
# The CPU Cycle



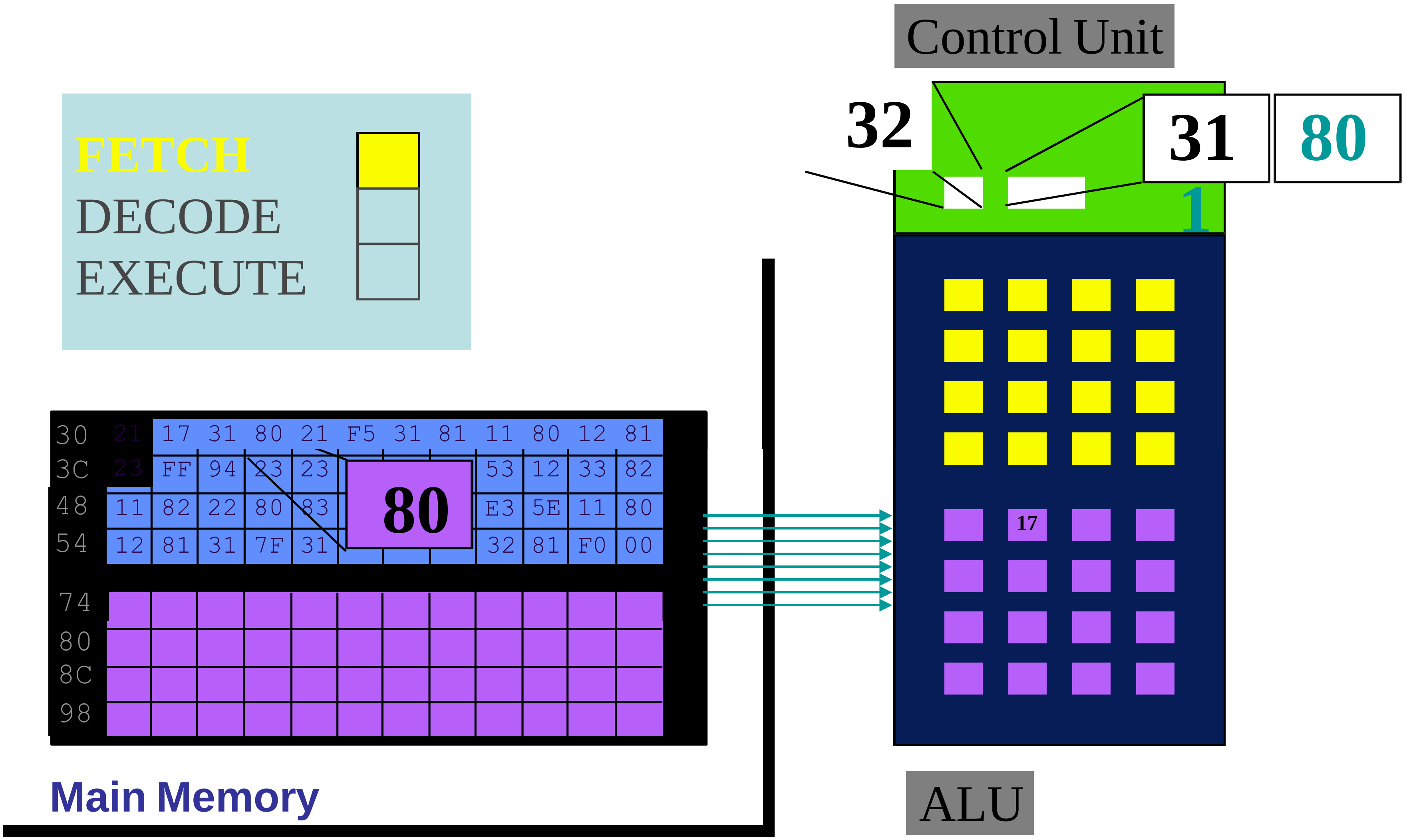
# The CPU Cycle



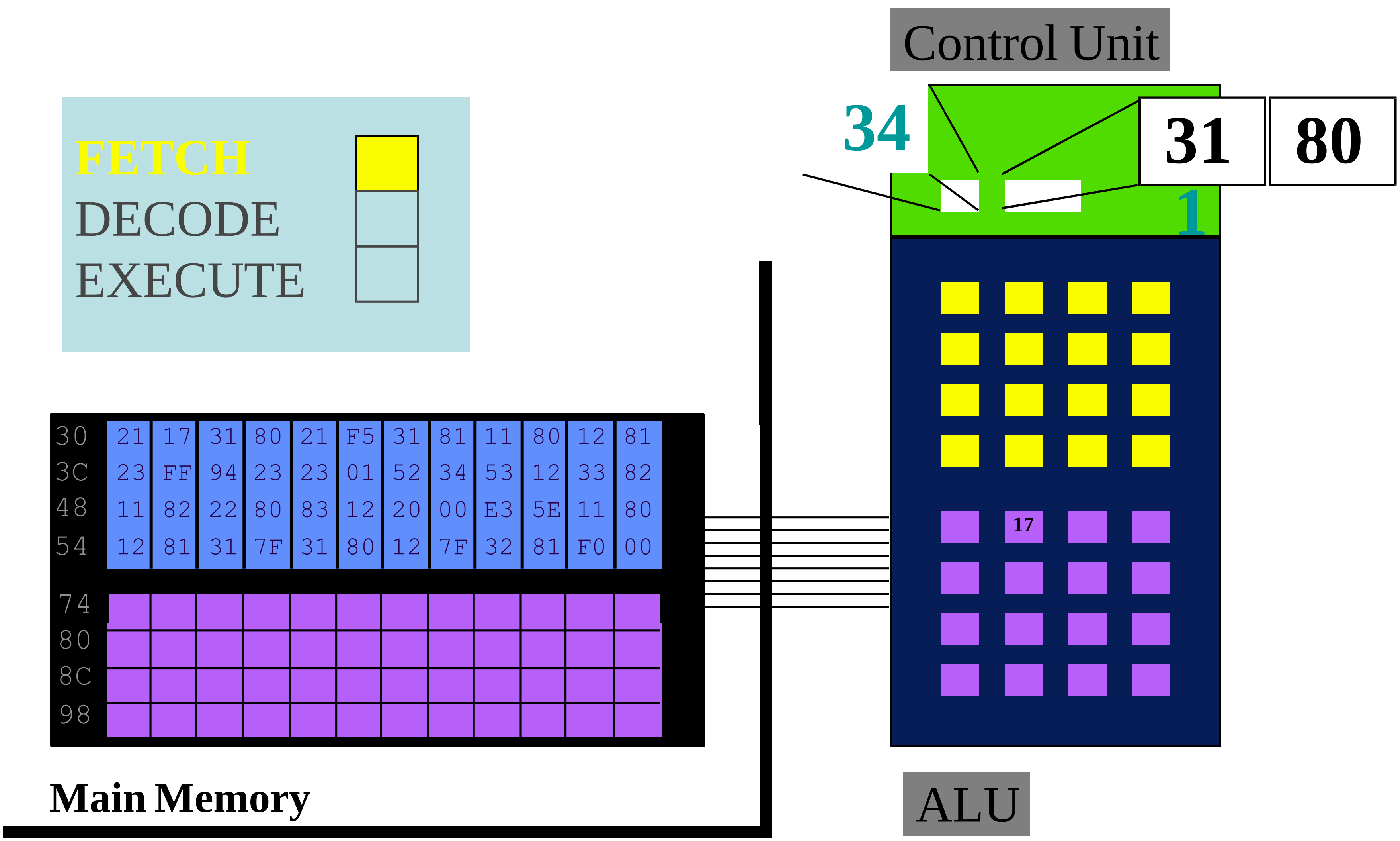
# The CPU Cycle



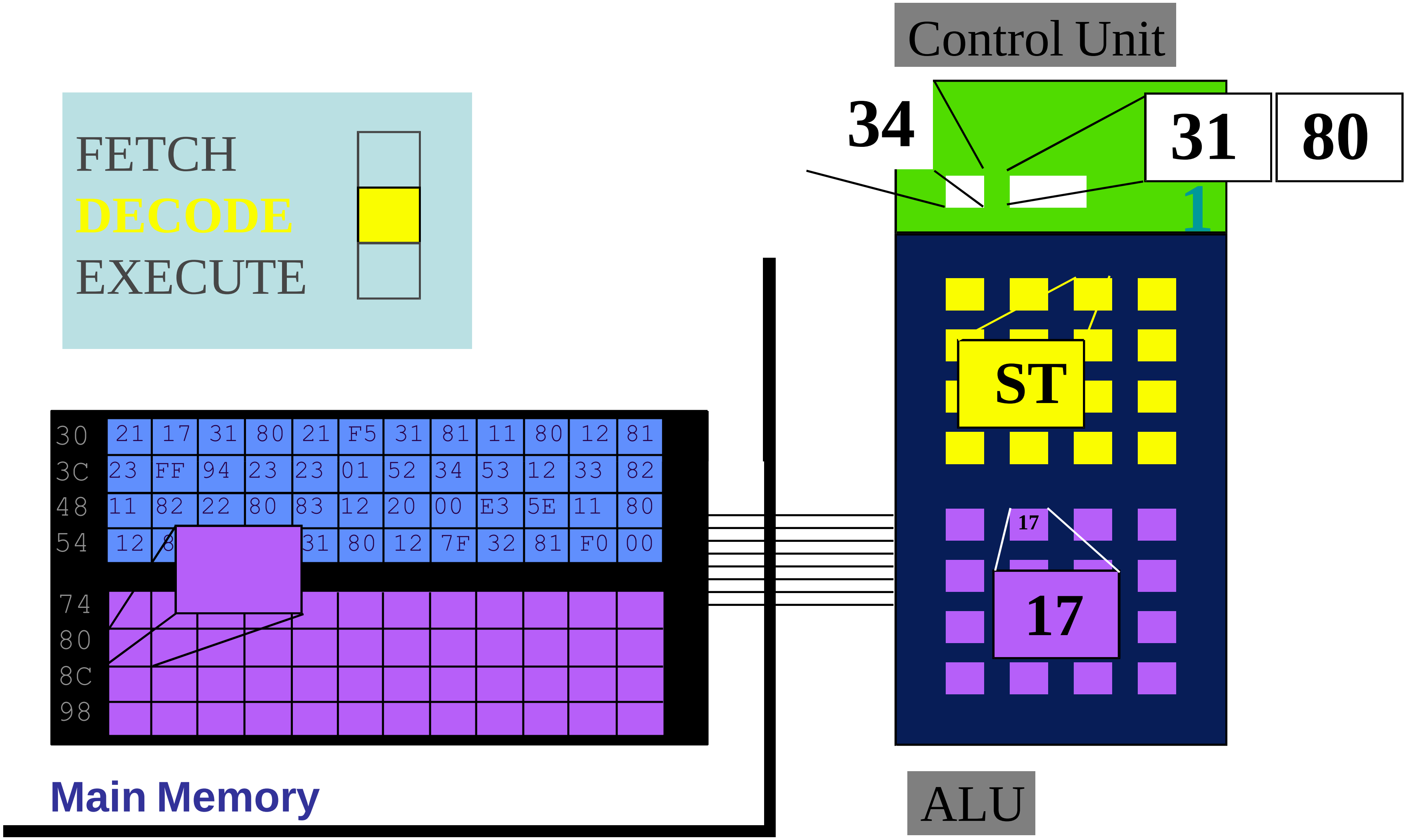
# The CPU Cycle



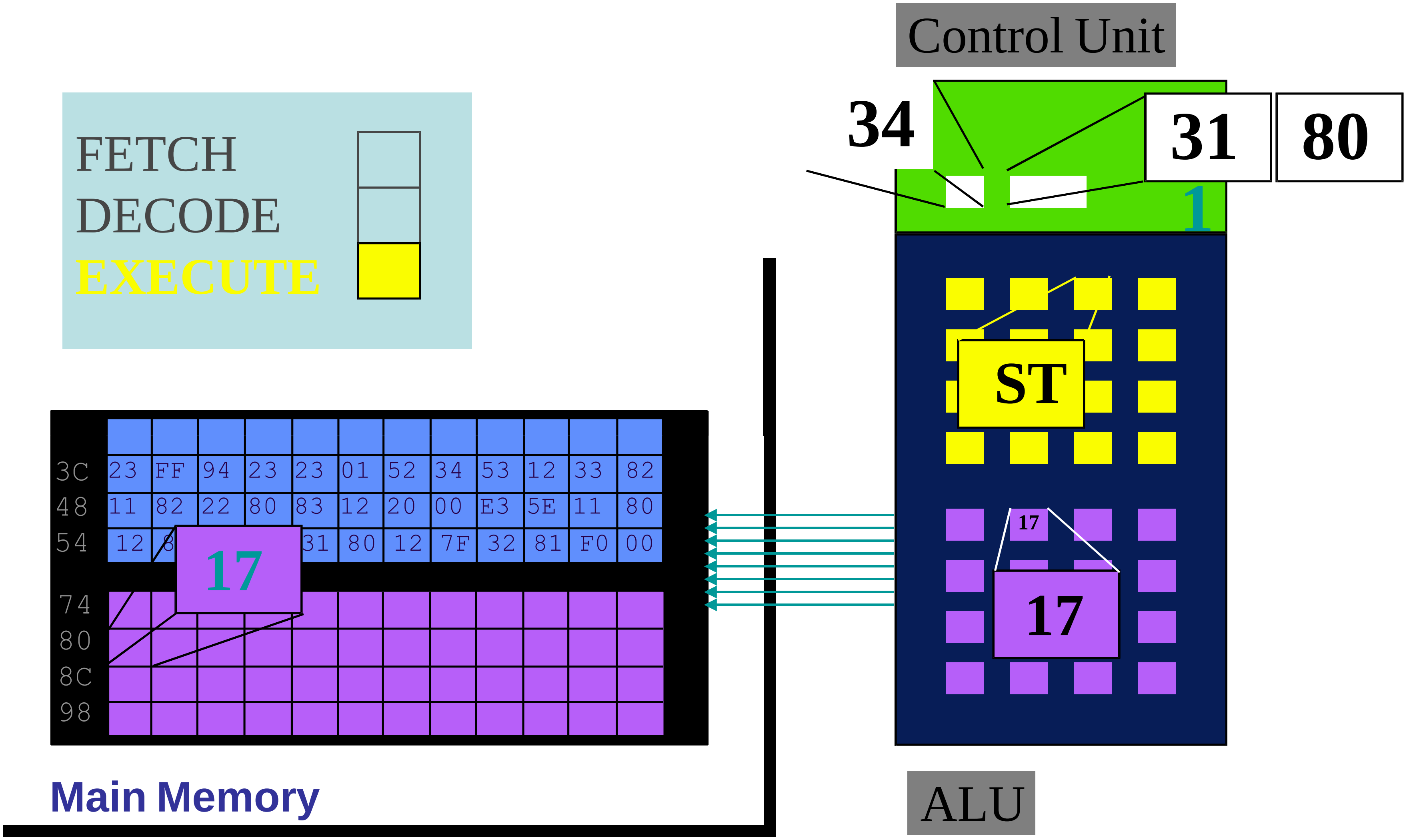
# The CPU Cycle



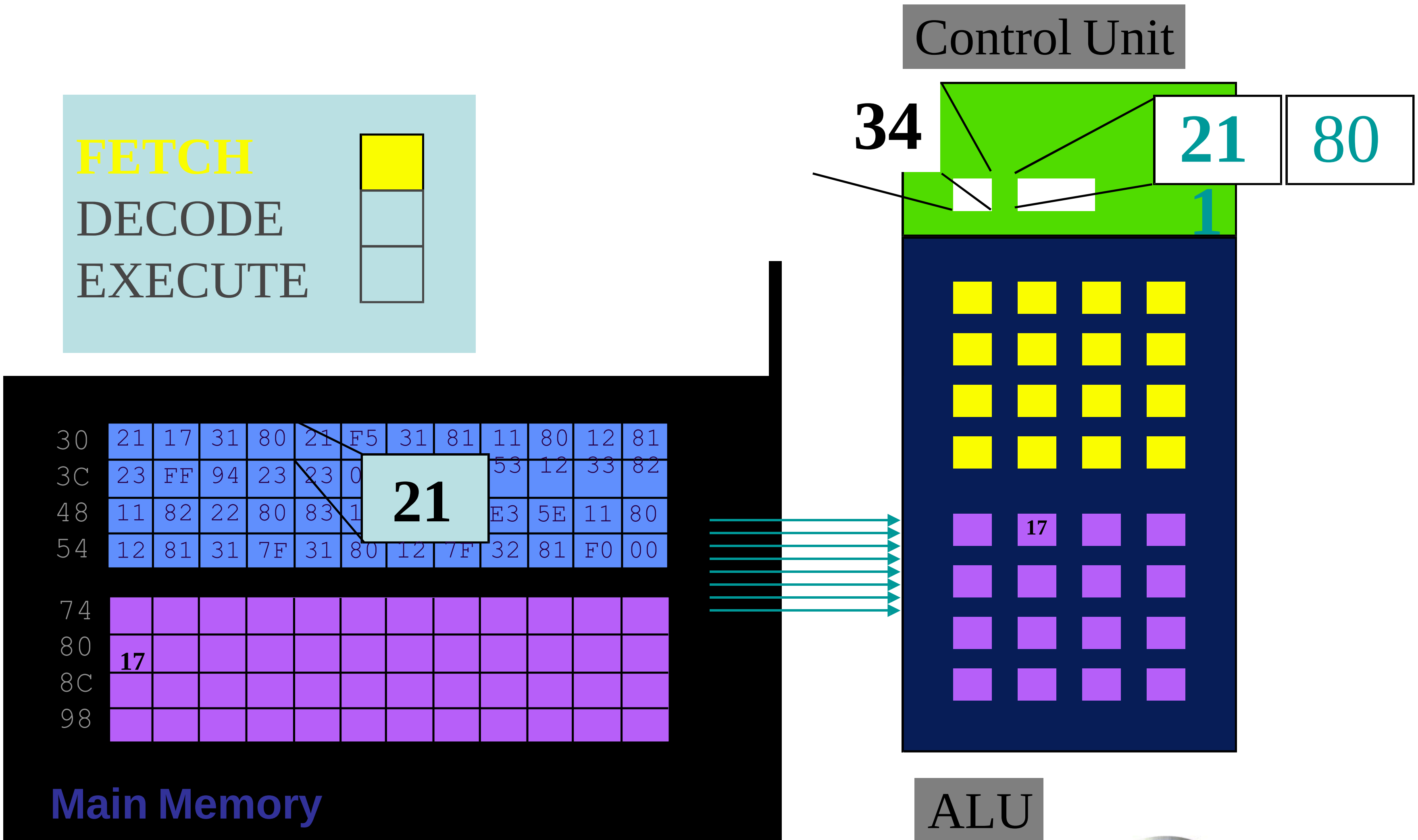
# The CPU Cycle



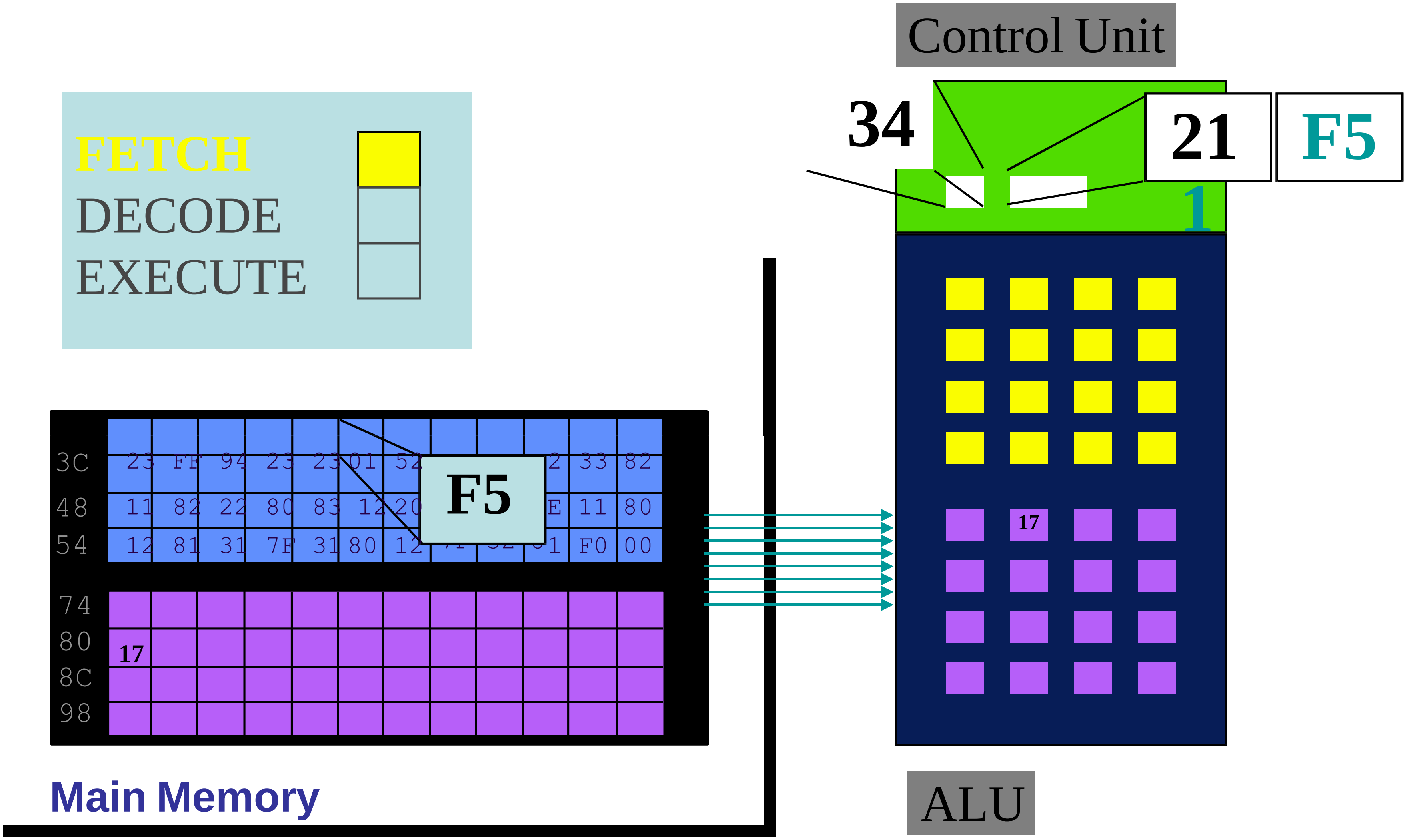
# The CPU Cycle



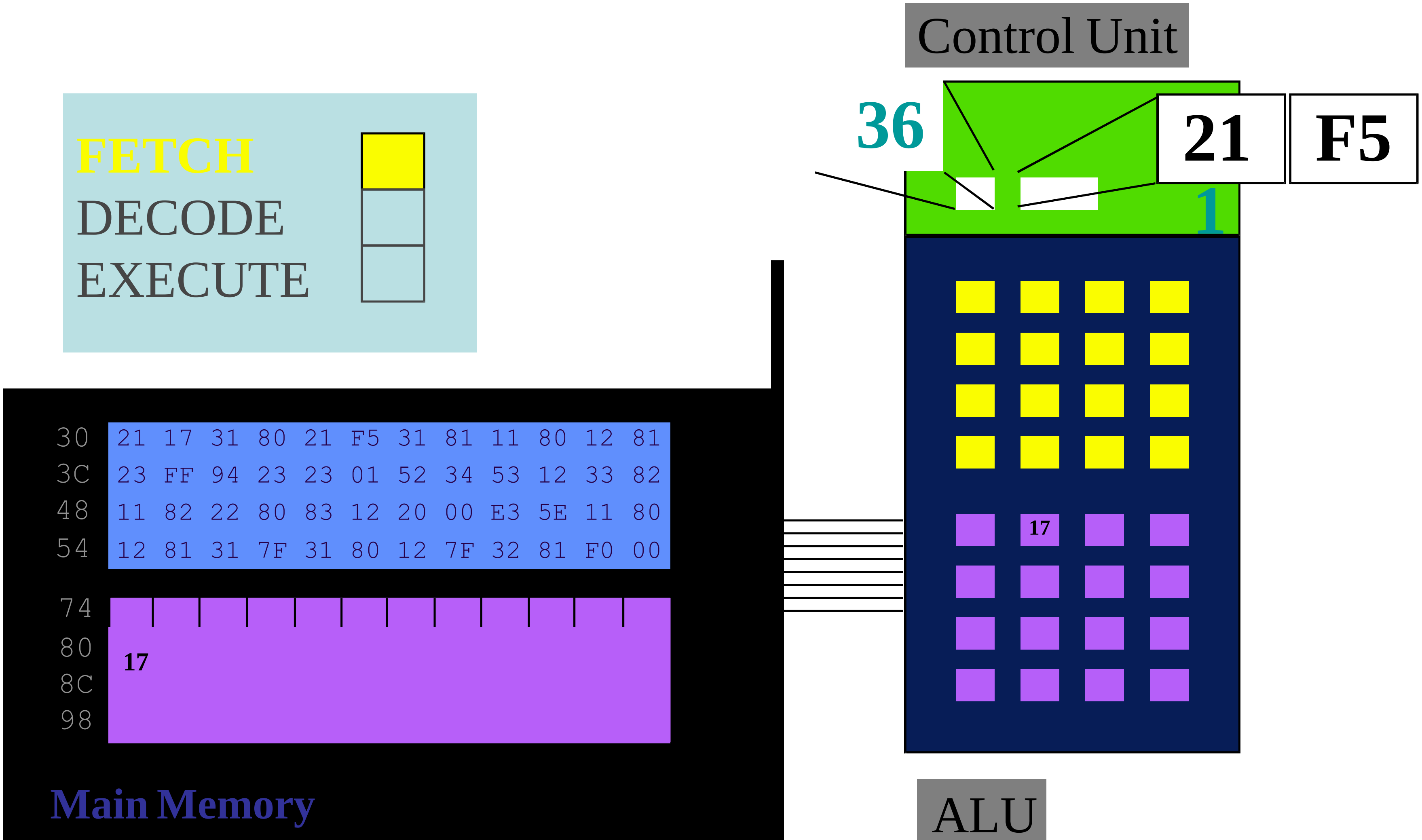
# The CPU Cycle



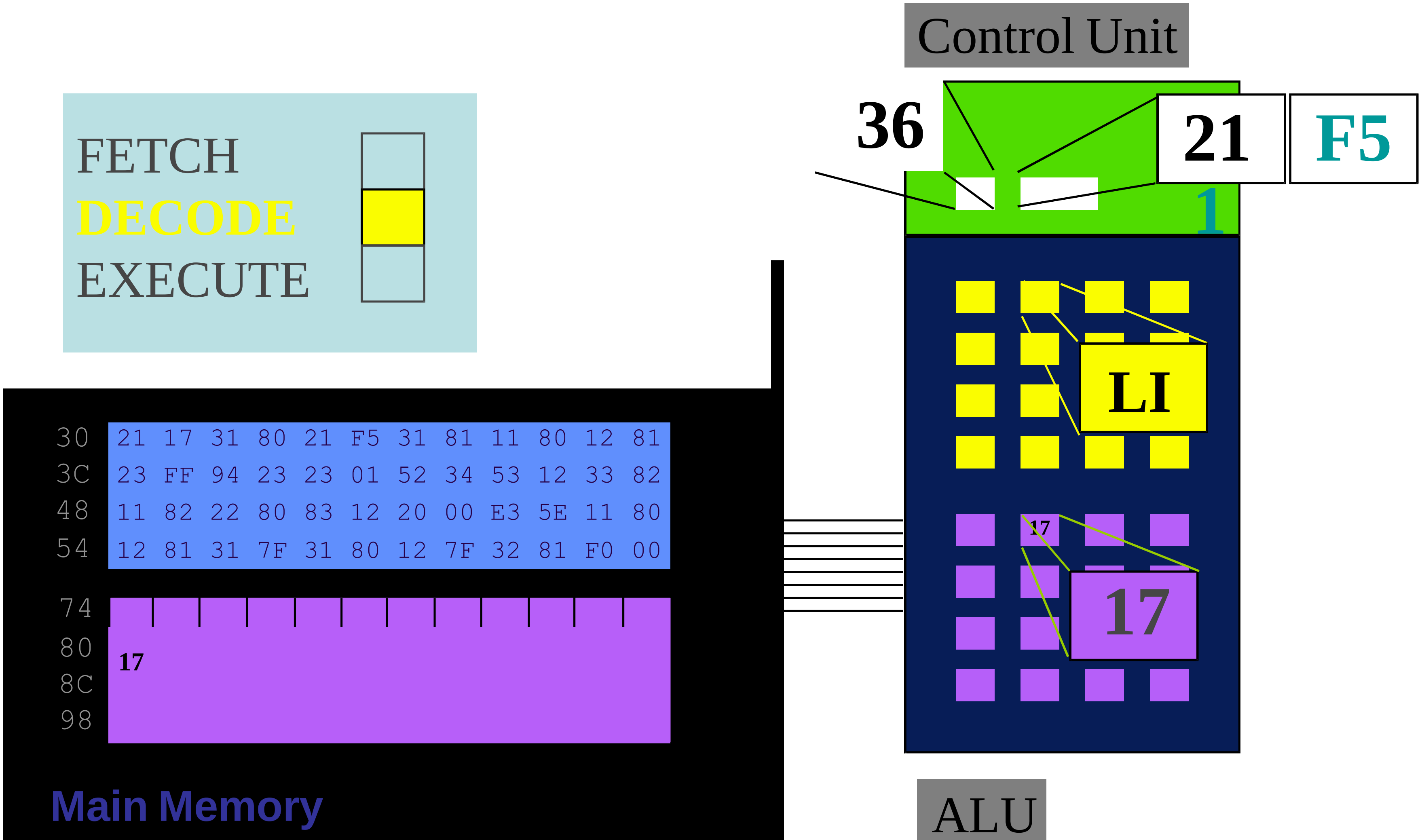
# The CPU Cycle



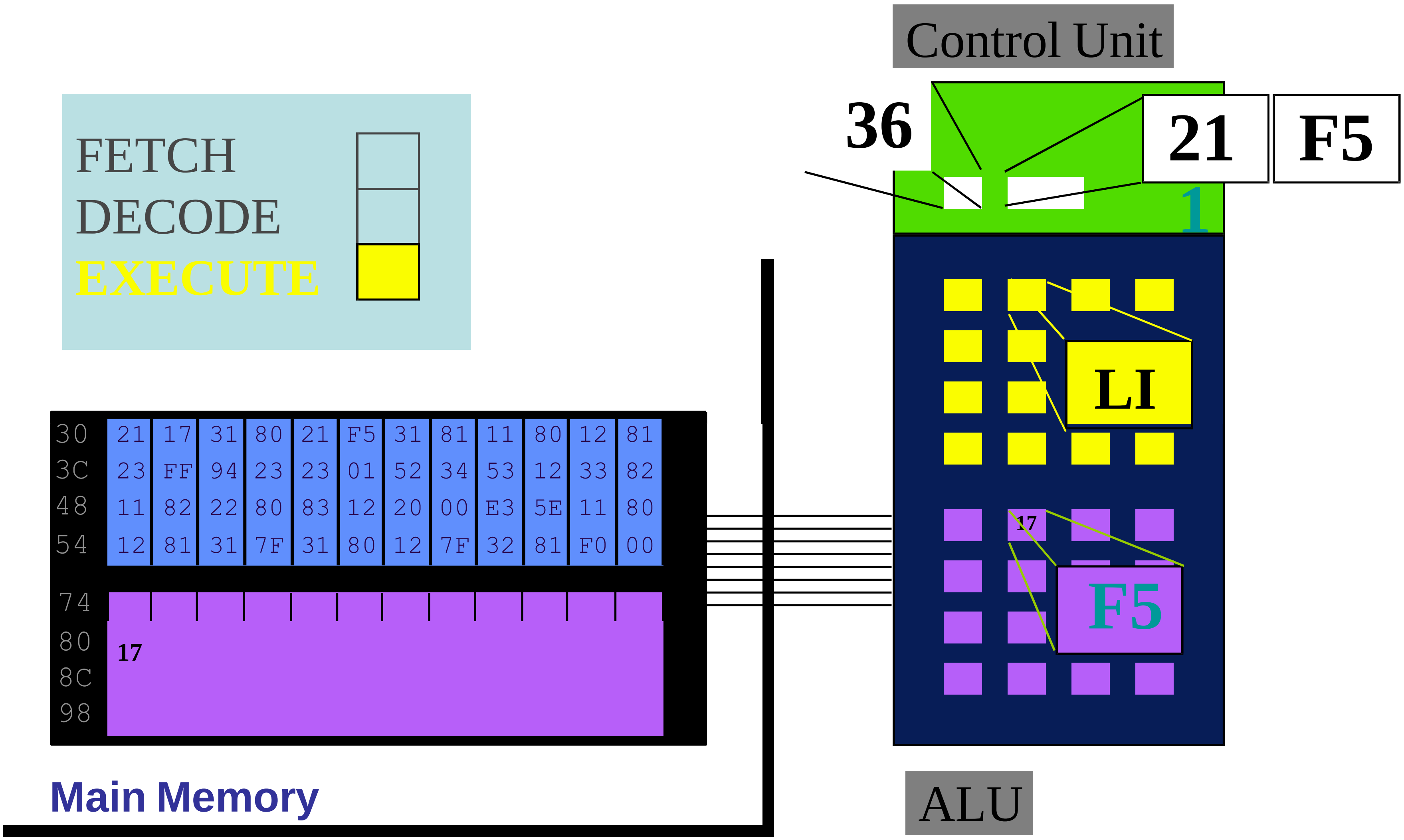
# The CPU Cycle



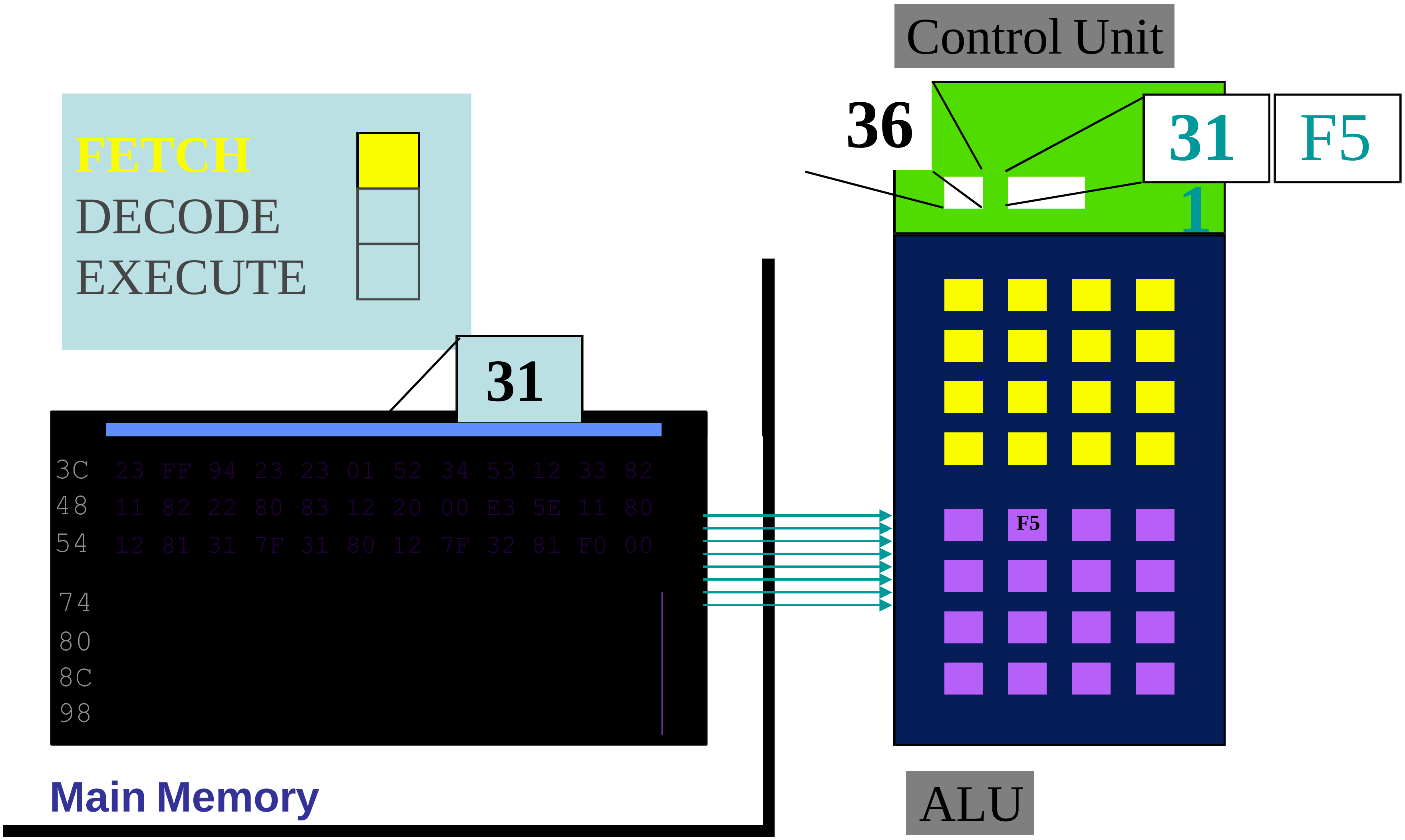
# The CPU Cycle



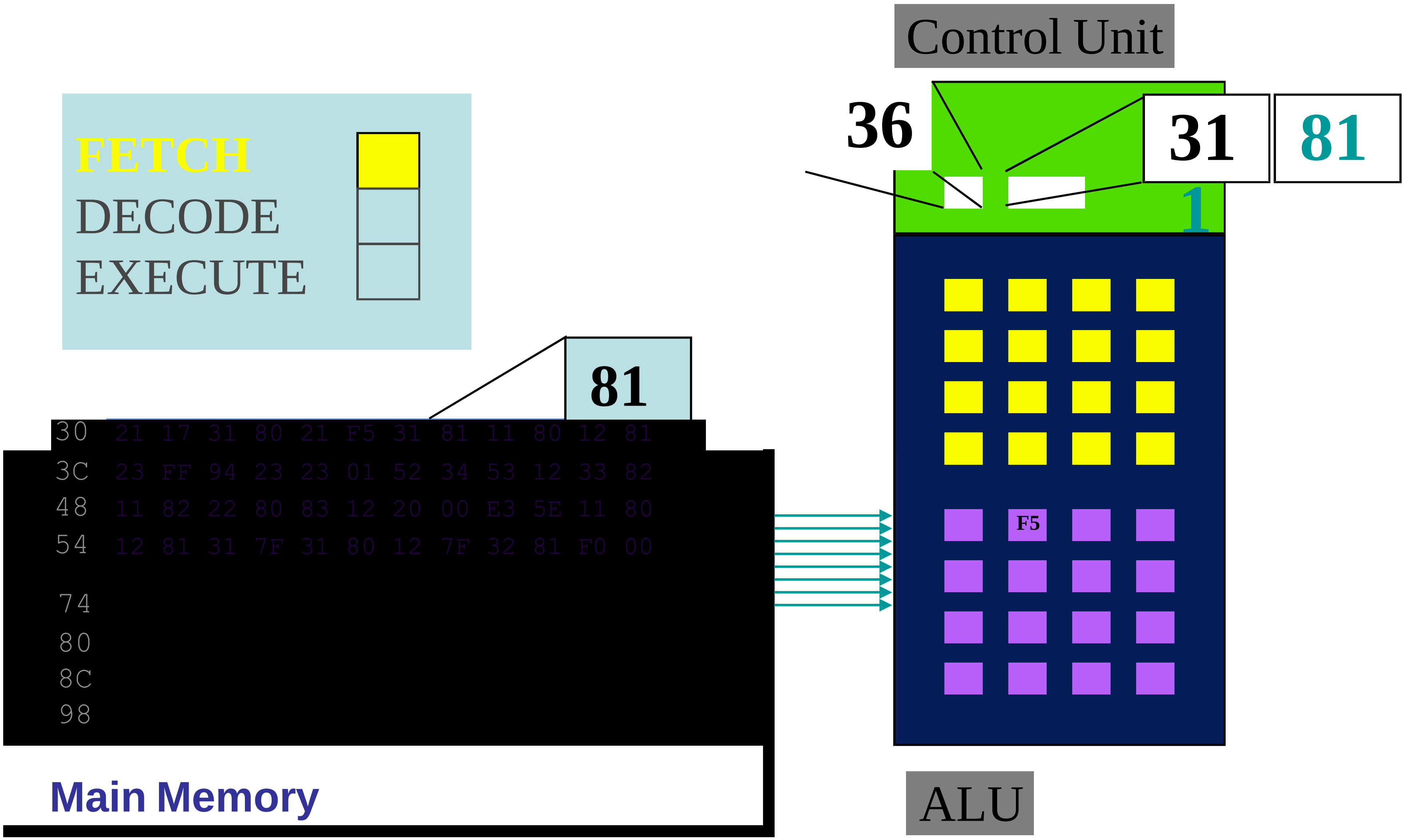
# The CPU Cycle



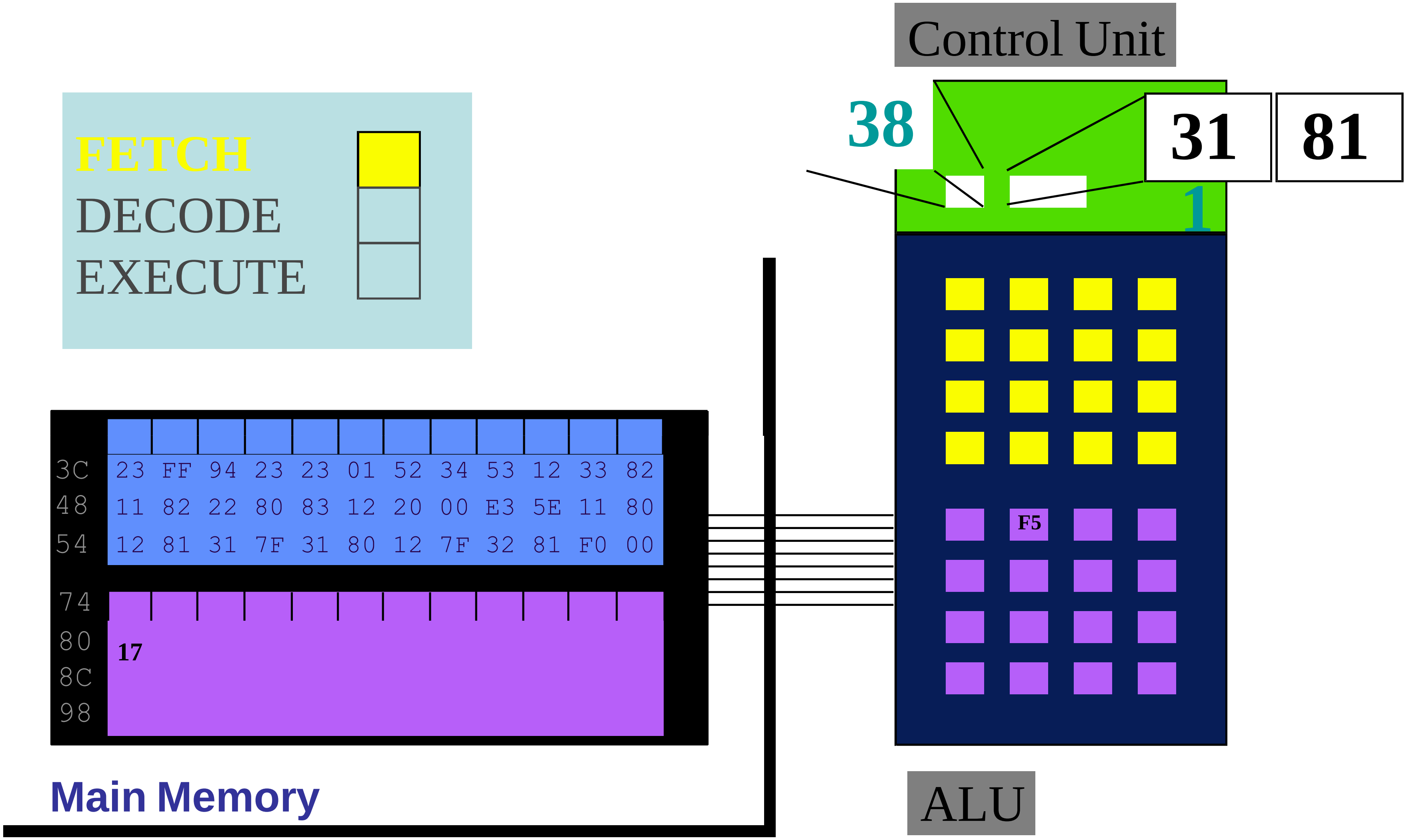
# The CPU Cycle



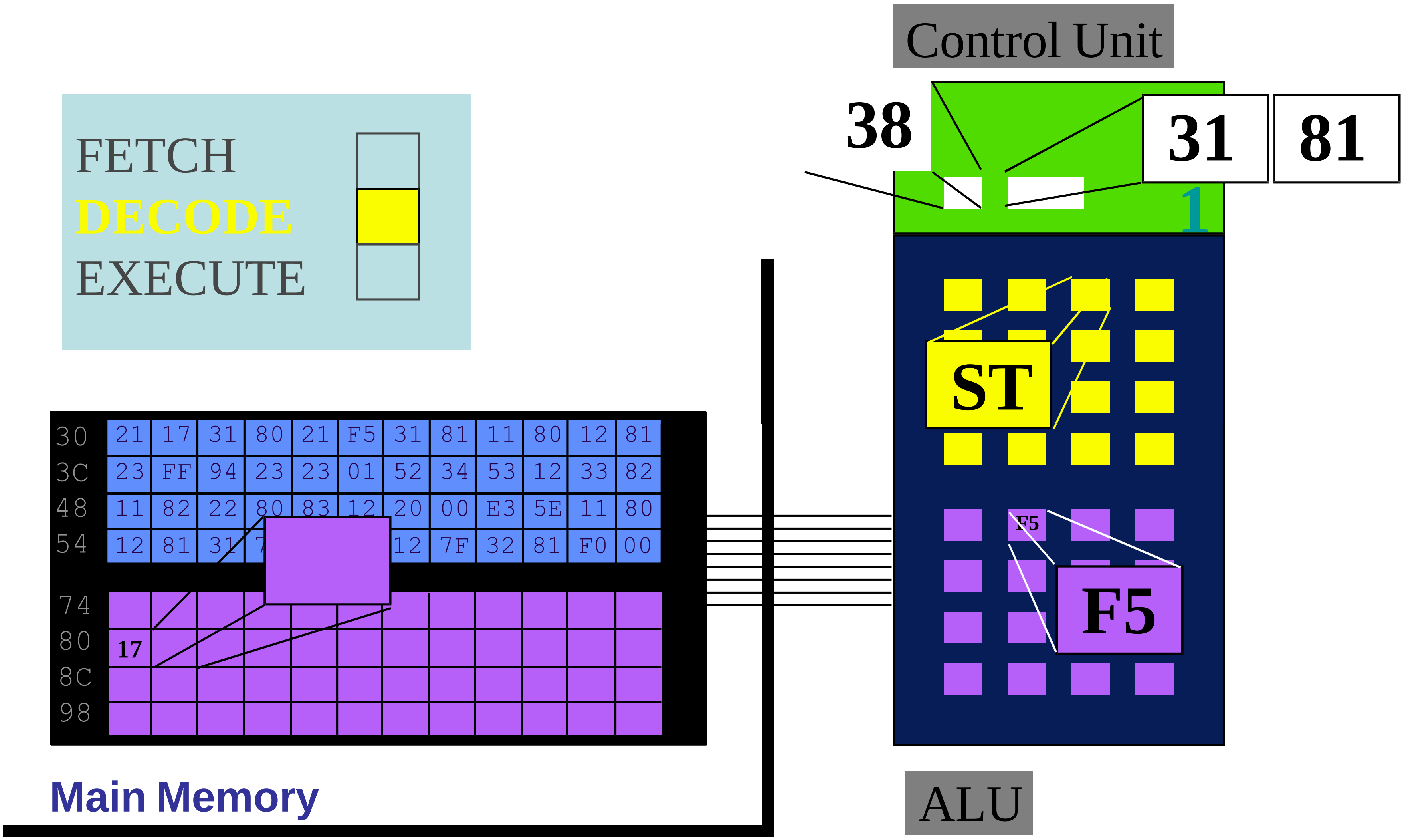
# The CPU Cycle



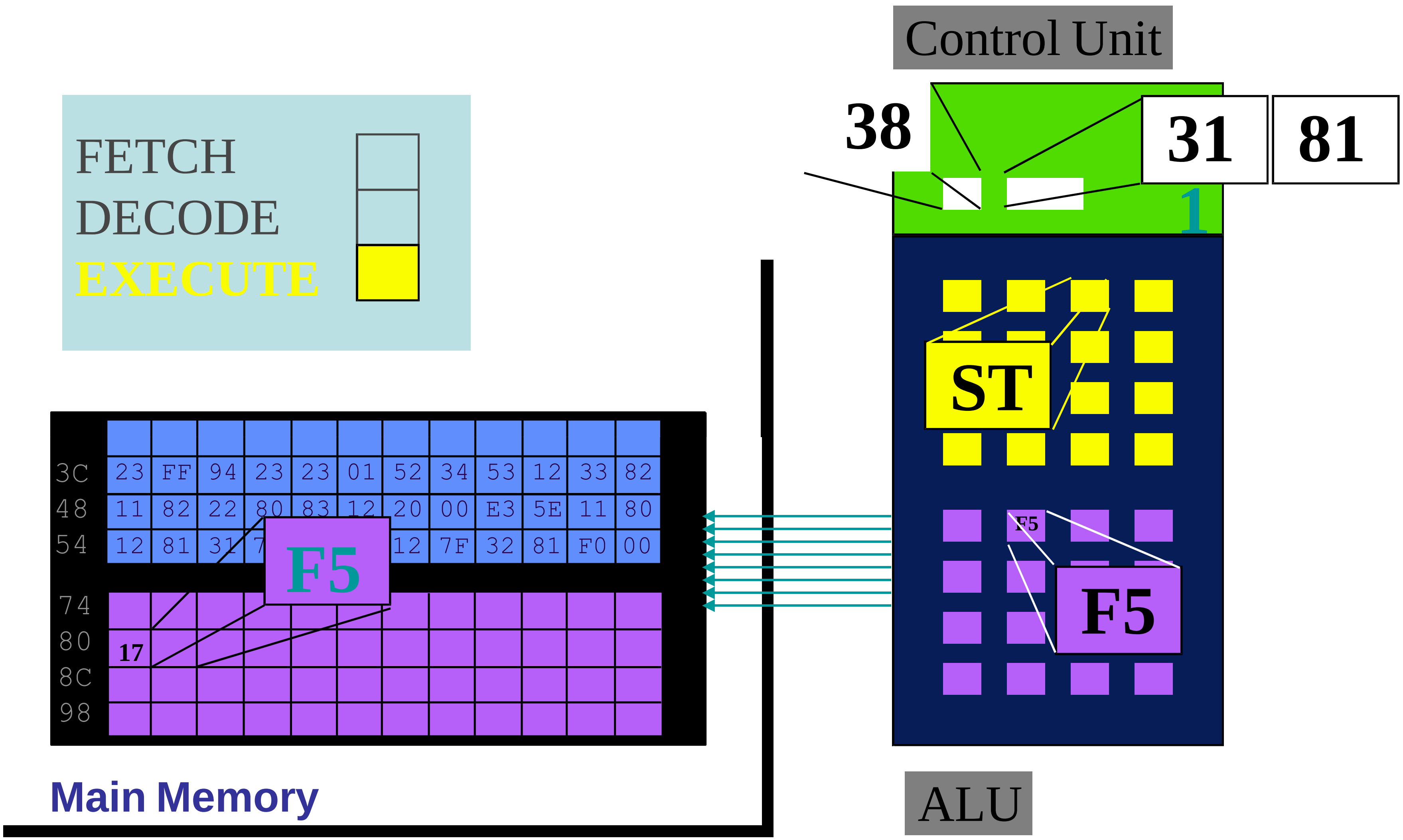
# The CPU Cycle



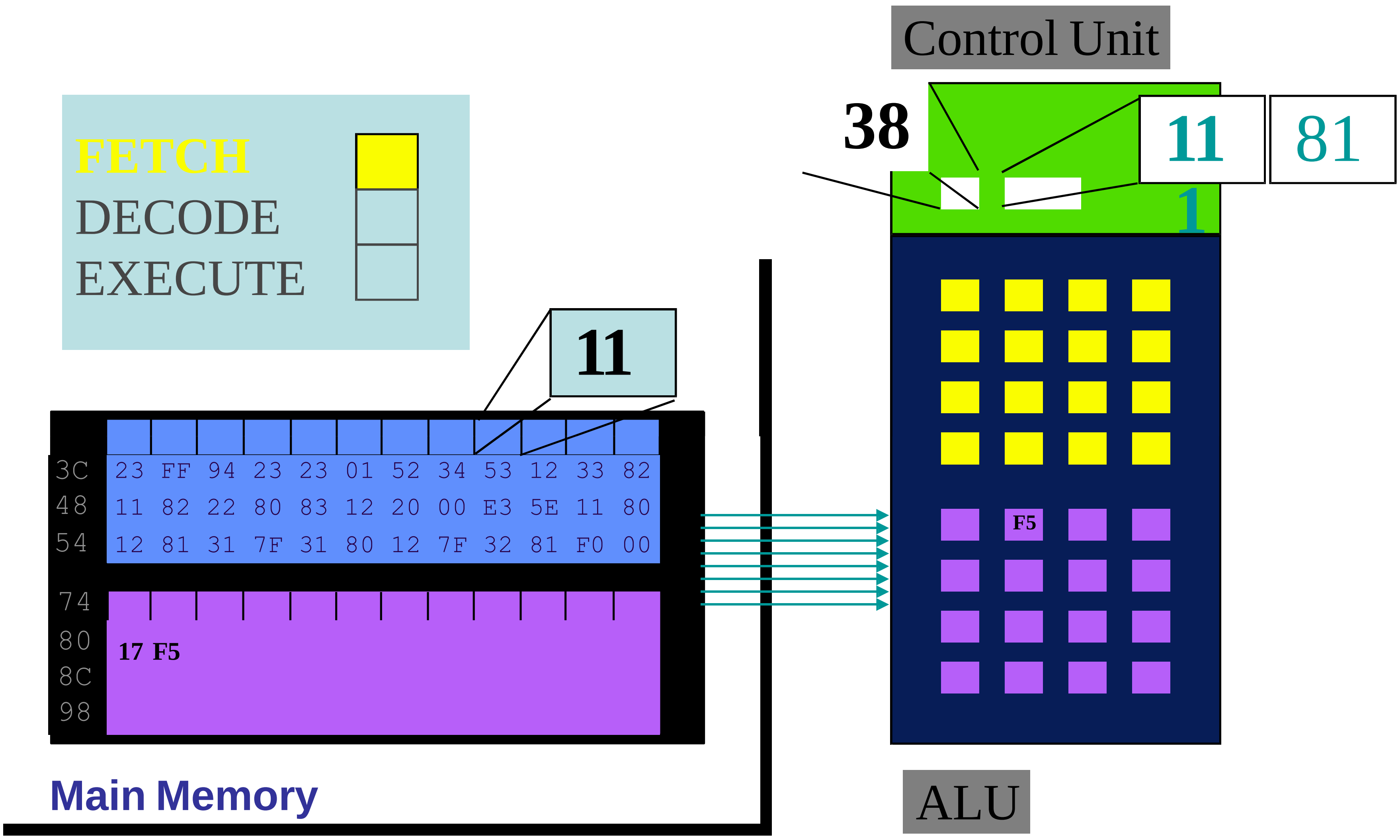
# The CPU Cycle



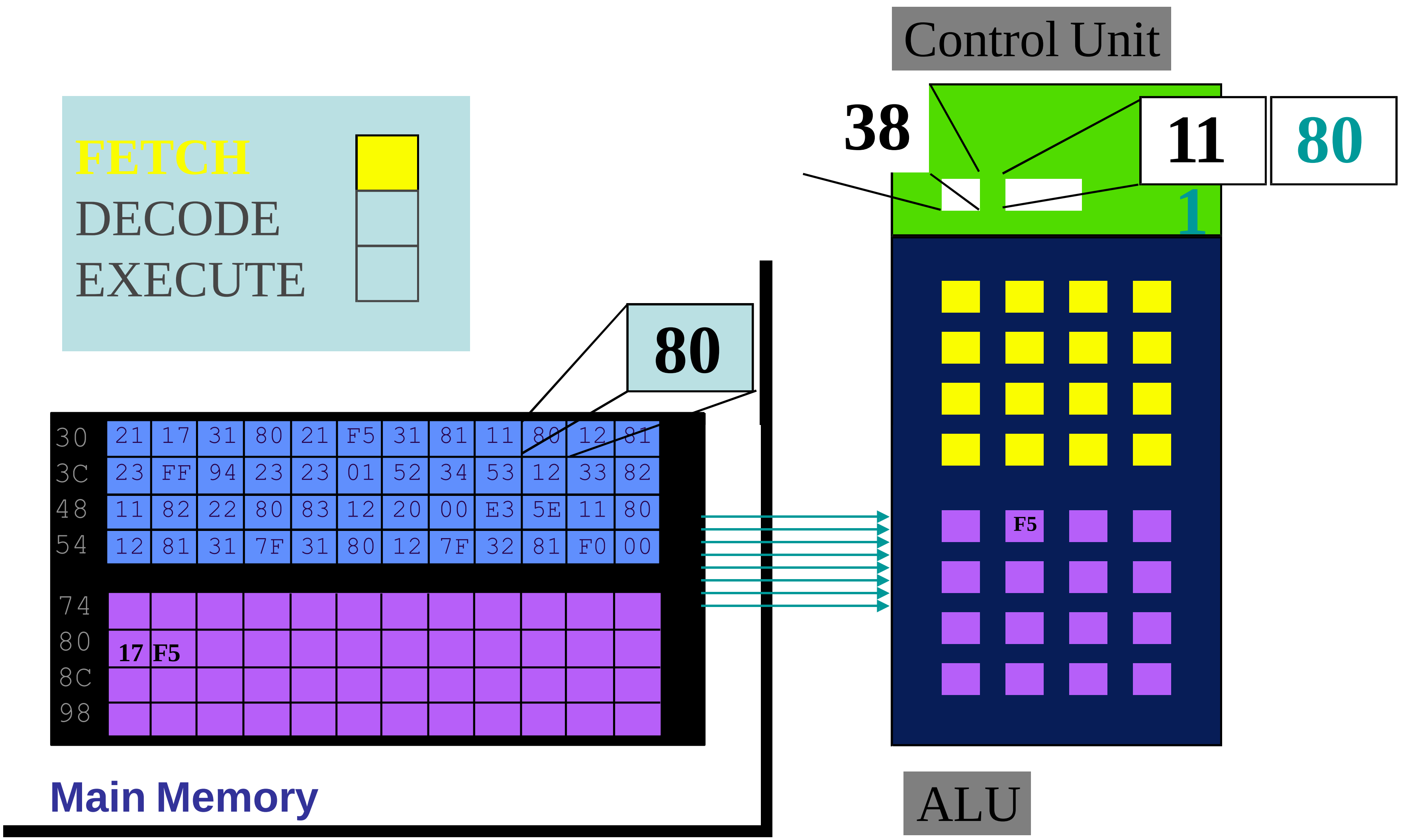
# The CPU Cycle



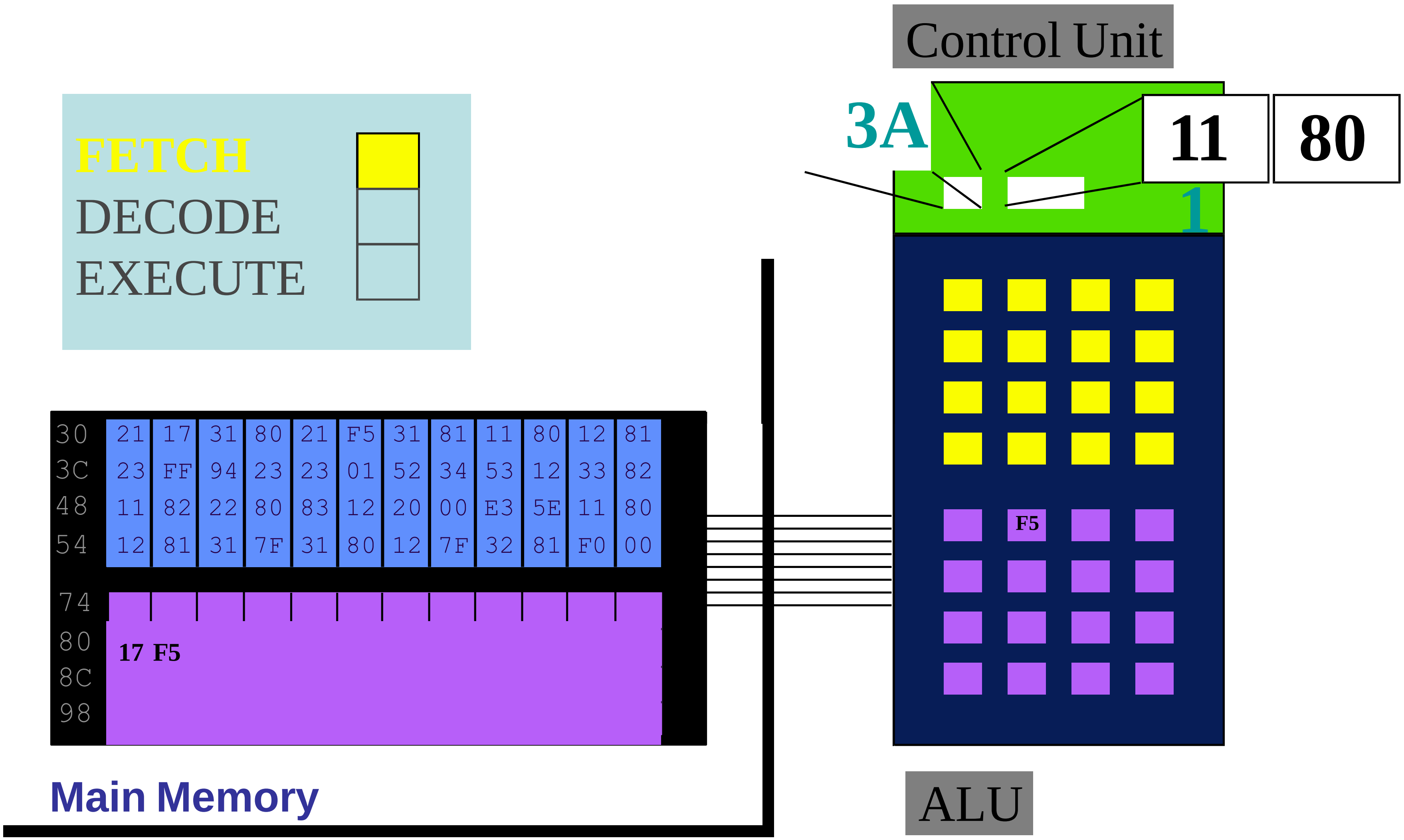
# The CPU Cycle



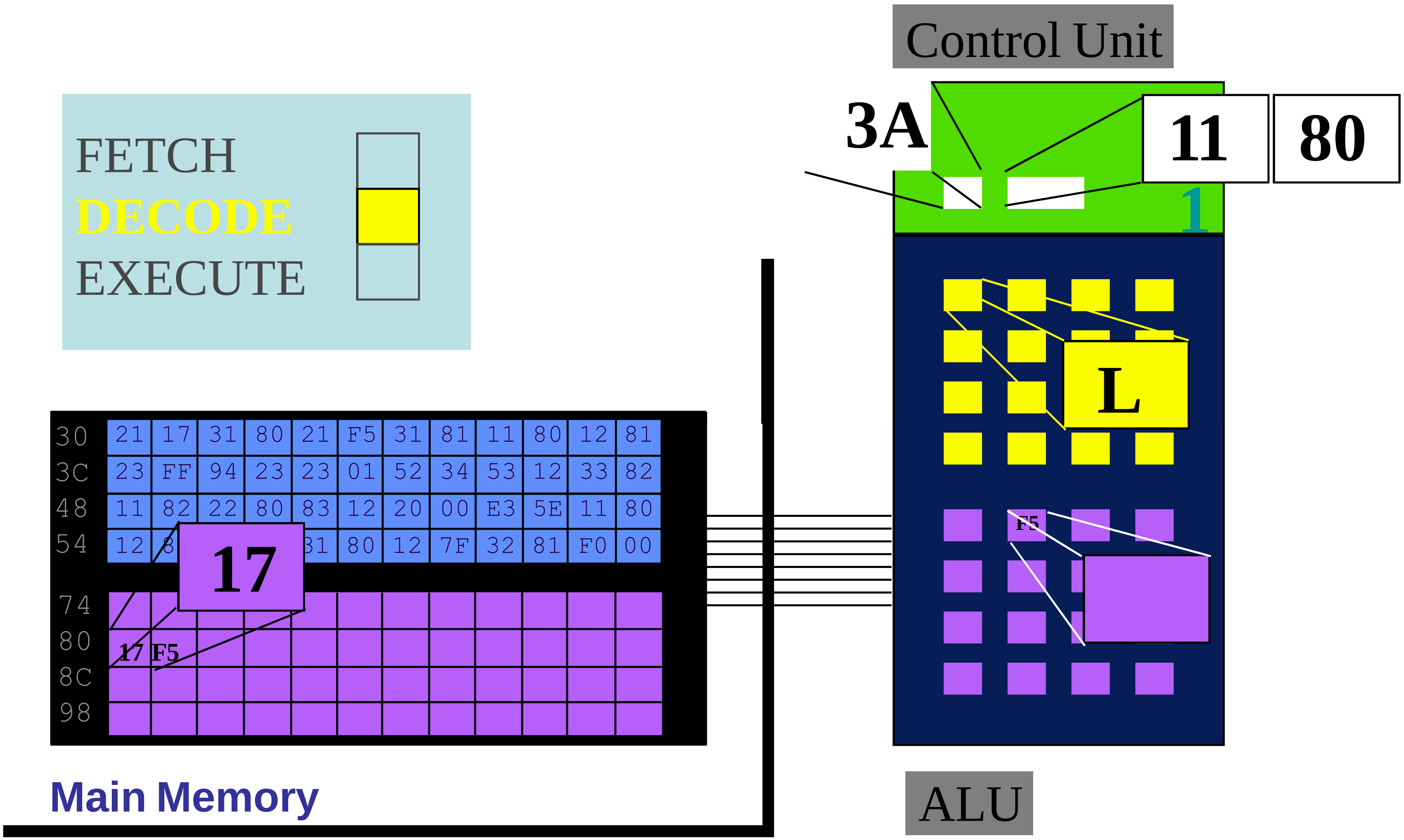
# The CPU Cycle



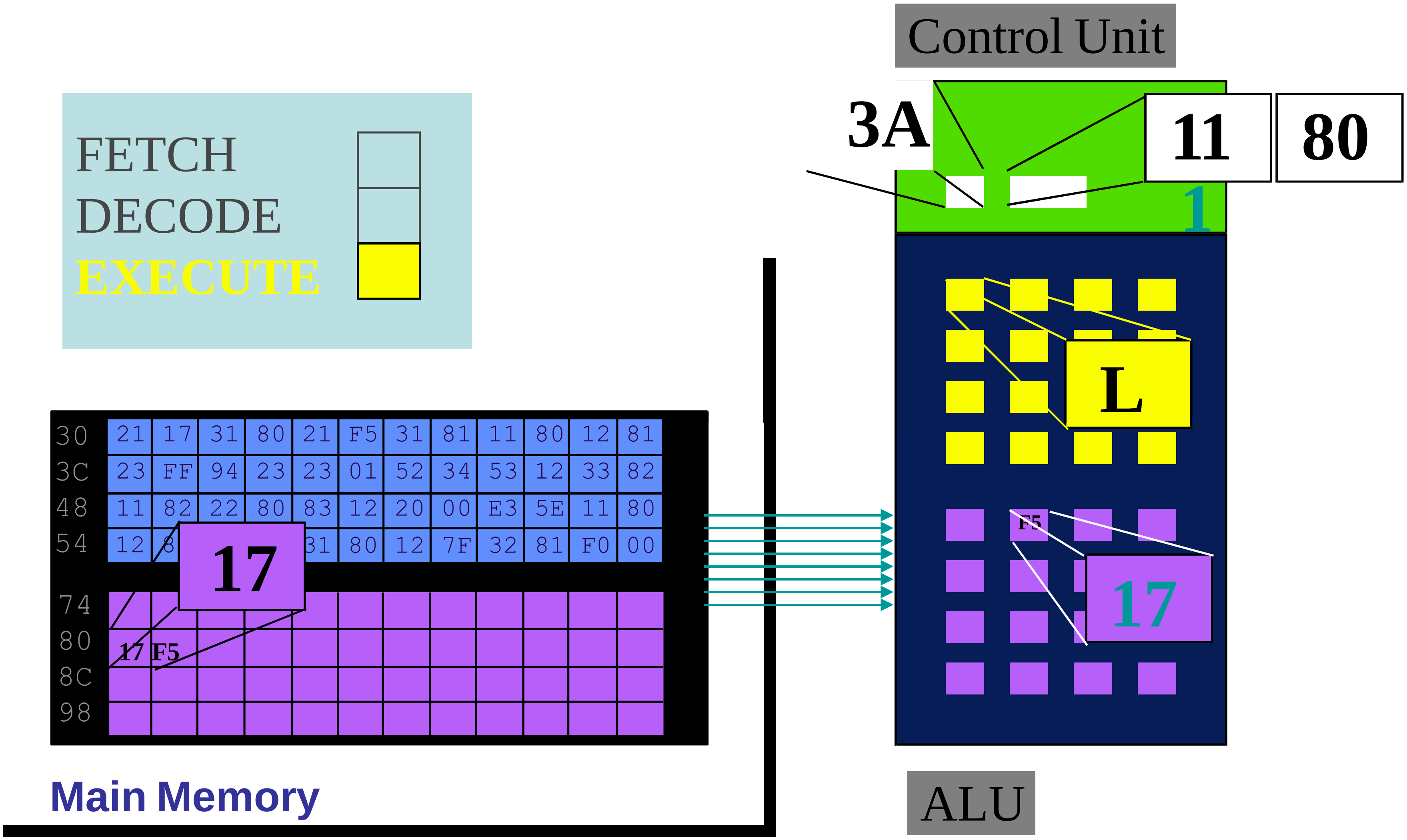
# The CPU Cycle



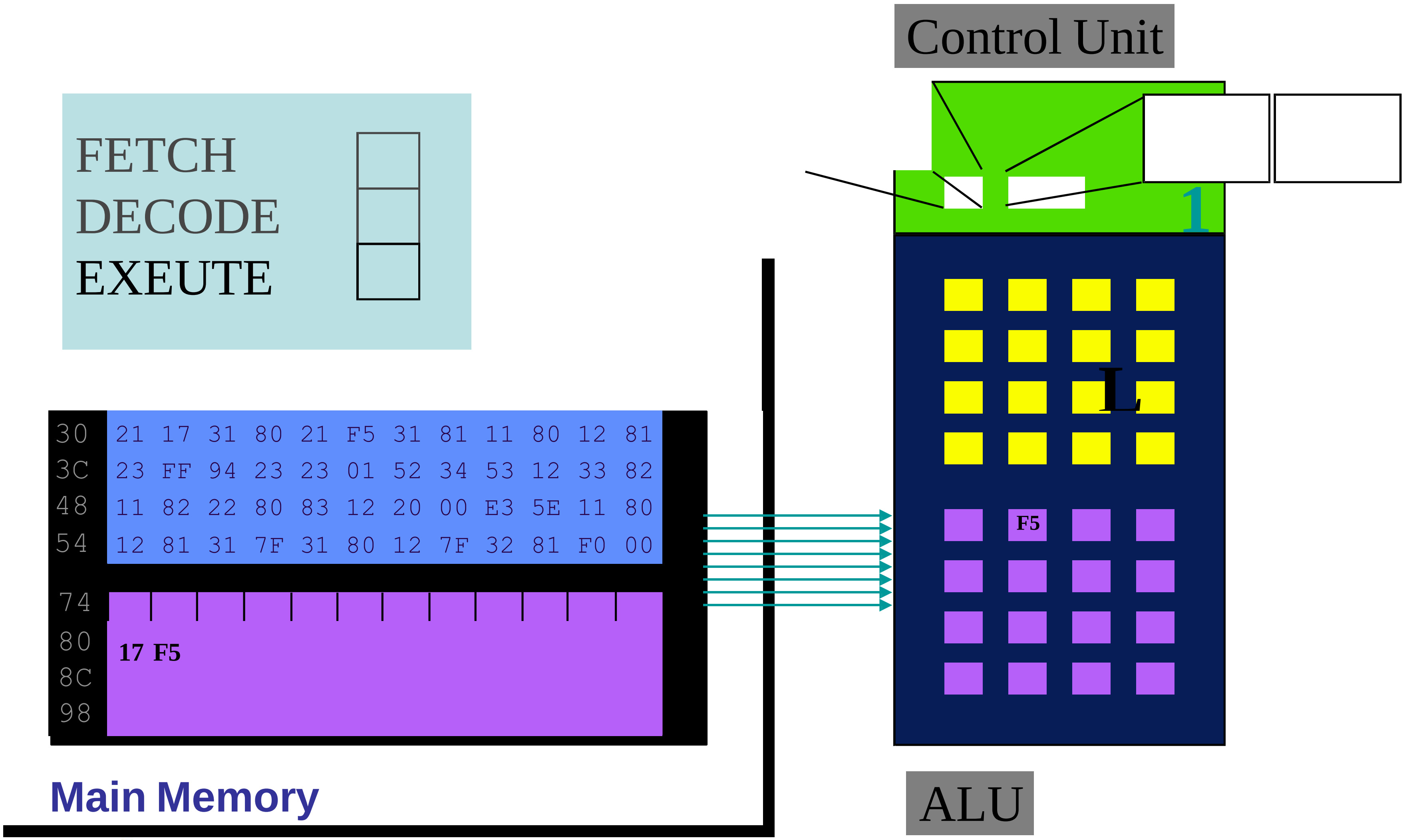
# The CPU Cycle



# The CPU Cycle



# The CPU Cycle – and so on…



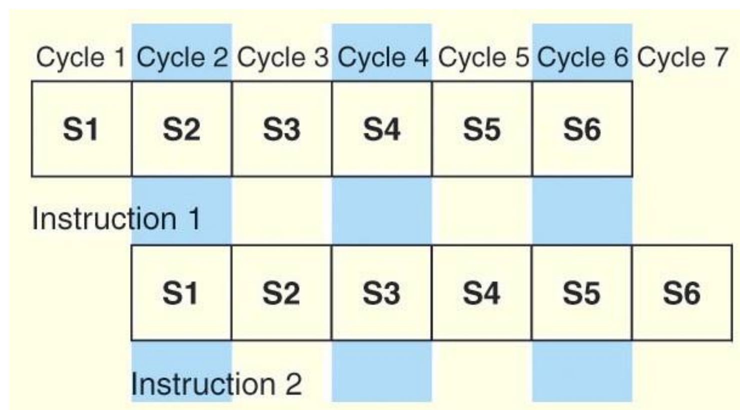
# Instruction Execution - Pipelining

- **Some CPUs divide the fetch-decode-execute cycle into smaller steps**

- ***Instruction Level Pipelining* overlaps these smaller steps <u>for consecutive instructions</u> in order to increase throughput**
  - Need to balance the time taken by each pipeline stage

# Instruction Level Pipelining - Example

- **Suppose a fetch-decode-execute cycle were broken into the following smaller steps:**
  1. Fetch instruction
  2. Decode opcode
  3. Calculate the address of operands
  4. Fetch operands
  5. Execute instruction
  6. Store result




- **For every clock cycle, one small step is carried out, and the stages are overlapped**

# Instruction Level Pipelining - Speed

- **There are *n* instructions**
- **There are *k* stages in the pipeline, and the time per stage is $t_p$**
  - The first instruction requires $k$ x $t_p$ time to complete
- **The remaining *(n – 1)* instructions emerge from the pipeline one per stage**
  - The total time to complete the remaining instructions is $(n - 1)\ t_p$
- **Thus, the time required to complete *n* tasks using a *k*-stage pipeline is**

$$(k * t_p) + (n - 1)\ t_p = (k + n - 1)\ t_p$$

# Instruction Level Pipelining - Speed

- **Speedup gained by using a pipeline**

time without pipeline

time with pipeline

$$Speedup = \frac{n \times k\, t_p}{(k + n - 1)t_p}$$

- **As *n* approaches infinity, *(k + n – 1)* approaches *n*, which results in a theoretical speedup of**

$$Speedup = \frac{n \times k\, t_p}{n\, t_p} = k$$

# Instruction Level Pipelining - Issues

- **Assumptions**
  - the architecture supports fetching instructions and data in parallel
  - the pipeline can be kept filled at all times
    - This is not always the case due to pipeline conflicts
- **It may appear that more stages imply faster performance, but**
  - the amount of control logic increases with the number of stages
  - pipeline conflicts affect the execution of instructions

UCSC

BIT

# Instruction Level Pipelining – Pipeline Conflicts

- **Resource conflicts**
  - One instruction is storing a value to memory while another instruction is being fetched from memory
- **Data dependencies**
  - When the not-yet-available result of one instruction is the operand of a subsequent instruction
- **Conditional branch statements**
  - Several instructions can be fetched and decoded before the execution of a preceding branch instruction is finished

# Thank You