Computer Instructions

CISC 3310 — CUNY Brooklyn College
Lecture Notes

Computer Instructions

In this chapter, we will:

1. Find out what 'sections' the CPU is built up of and their purposes.
2. Understand the difference between hardwired and microprogrammed CPU control units.
3. Discuss CPU instructions and what features computer developers need to decide about them.
4. Introduce the idea of instruction pipelining and how it speeds up CPU activity.

The CPU

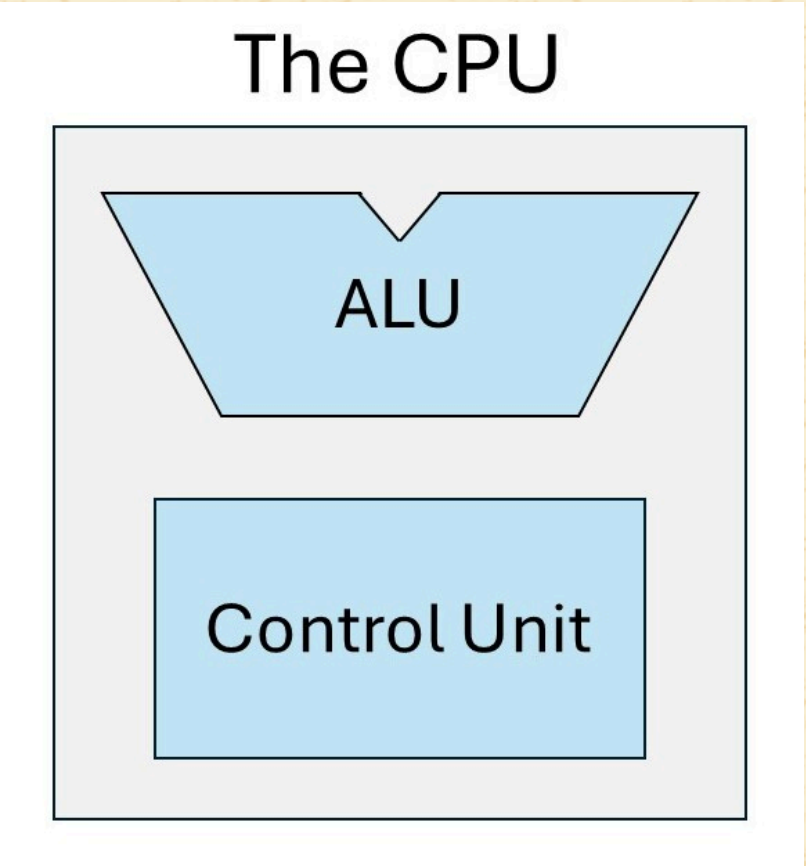
Now that we now how to build meaningful circuits like adders, multiplexers, shifter, multipliers, registers, and other memory storages, we can integrate them to create computer chips or devices, each of which serves a main purpose inside a computer.

We begin with the **CPU**: the Central Processing Unit, whose purpose is to perform calculations with data and make decisions.

A CPU consists of both combinational circuits (which perform calculations) and sequential circuits (to temorarily store the data we work with before it moves to permanent storage or to be output to the user, e.g., via the computer's screen.)

We can divide the CPU into two parts: (1) the **datapath**, which includes the **ALU** (Arithmetic-Logic Unit), which is where computations and decisions take place and where they are stored, and (2) the **Control Unit**, which oversees and controls the activity inside the CPU.

The CPU



The 2 parts of a CPU: ALU and Control Unit. Miriam Briskman, CC BY-NC 4.0.

CPU: Registers

Let's talk about storing data inside the CPU.

Registers are small, high-speed storage locations located directly within the CPU and temporarily store data, instructions, and memory addresses that are actively being used in calculations or program control.

Unlike RAM, which serves as general storage for the computer, registers are accessed much more quickly due to their proximity to the CPU's processing units.

This rapid access to frequently used values allows the CPU to perform operations at a significantly higher speed.

As we shall see next, different types of registers within the CPU fulfill specific roles, all contributing to efficient data handling and processing.

CPU: Registers

The CPU uses various types of registers, each serving a specialized purpose to optimize calculations.

General-purpose registers (GPRs) are versatile and can hold data or memory addresses, depending on the CPU's immediate needs. A programmer can use these 'customizable' registers to store any type of data they choose to.

In contrast, **special-purpose registers** are designed for specific control tasks, such as managing program flow and monitoring system status.

Among these are the **program counter** register, which stores the address to the next instruction of the program, and thus keeps track of the CPU's current position in a program, and the **status register**, which monitor the outcome of operations, keeps track of overflow, and stores carry bits.

This classification of registers helps in organizing and optimizing data flow within the CPU, enhancing overall performance.

CPU: Registers

The **Program Counter** (**PC**) register is crucial for maintaining the CPU's instruction sequence.

It stores the memory address of the next instruction (e.g., adding two numbers, storing a number, etc.,) to be fetched and executed by the CPU.

After each instruction is completed, the PC updates to point to the following instruction, ensuring a smooth and orderly program flow.

In cases where an instruction requires a jump or branch to a different part of the program (think about the moment you want to exit a loop: you want to jump to the code below the loop,) the PC is adjusted accordingly to reflect this new path.

Every program stores its own value in the program counter register, so when programs must 'switch' on the CPU, the previous program's program counter is copied from the CPU to main memory, while the next program's counter is copied from main memory into the CPU.

CPU: Registers

The **Status Register**, sometimes known as the **Flags Register**, monitors and records the outcomes of various CPU operations.

It includes flags for conditions like zero, carry, overflow, and negative values, which reflect specific results from recent calculations.

For instance, the zero flag is raised (= set) if an operation yields a result of zero, which might trigger conditional instructions based on this outcome.

Other flags, such as the carry flag, provide information about arithmetic overflows, crucial for tasks requiring precise calculations.

By tracking these conditions, the Status Register supports decision-making processes within the CPU, allowing it to respond dynamically to changes during program execution.

CPU: Registers

Arithmetic and Boolean (logic) operations in the CPU are supported by registers that hold data and intermediate results throughout the calculations.

A common type is the **accumulator**, a specialized register dedicated to storing results from arithmetic and logical operations. A CPU usually has several accumulator registers.

Some CPU architectures have additional registers, like multipliers and dividers, designed specifically for complex mathematical operations.

By using dedicated arithmetic registers, the CPU can perform calculations quickly without having to access slower memory, which boosts efficiency.

These registers play a critical role in ensuring that arithmetic operations are completed rapidly and accurately, contributing to the CPU's overall processing power.

CPU: Registers

Any questions so far?

Computer Buses

A **computer bus** is a communication system within a computer that transfers data between components or devices. A bus is a wire or a set of wires controlled by rules called the **protocol**.

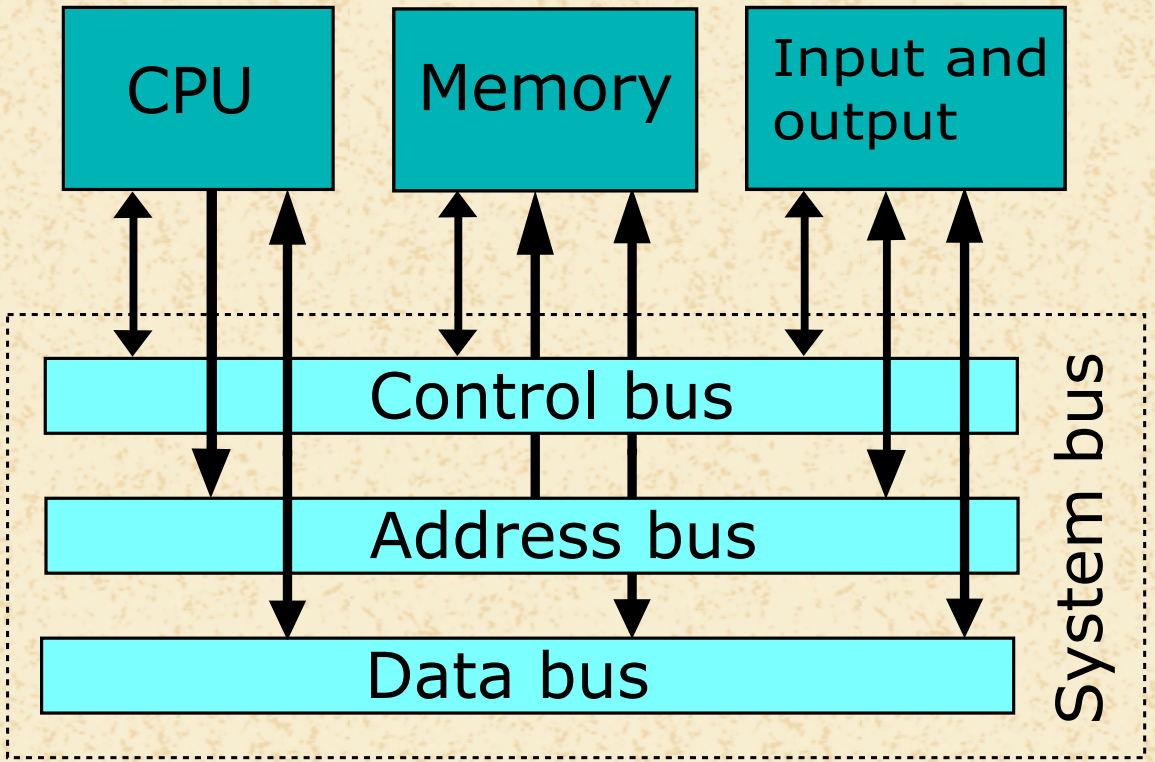
Buses serve as a central highway, connecting the CPU, memory, storage, and peripherals, allowing them to send and receive data efficiently.

The bus architecture is crucial for coordinating communication and maintaining overall system performance.

Bus speed, often measured in megahertz (MHz) or gigahertz (GHz), determines how quickly data travels, impacting the responsiveness of a computer system. A bus's protocol indicates in which speed data should pass through that bus.

High-speed buses enable faster data exchange, supporting demanding applications and system performance.

Computer Buses



The system bus. W Nowicki, CC BY-SA 3.0, via Wikimedia Commons.

Computer Buses

There are several main types of buses, each serving different roles in computer architecture. However, most buses contain the following lines/wires inside them:

- A **data line** carries actual data being processed or transferred between components, enabling communication between the CPU, memory, and peripherals.
- An **address line** sends location addresses to identify where data should be read or written in memory.
- **Control lines** transmit control signals that manage the coordination of data transfer and execution commands across components.
- Additionally, some buses include **power lines** that deliver necessary electrical power to connected devices.

The **system bus**, also called the internal bus, is the one that connects crucial computer devices, including CPU, cache, and main memory. Buses that connect the CPU or RAM with I/O devices are called **external buses**.

Computer Buses

The **data line** is responsible for carrying data back and forth between the CPU, memory, and other components.

- Examples of data include numbers (both integers and floating-point numbers,) characters, strings, and other more complex types or objects.

The **width** of the data line (measured in bits) determines how much data can be transferred simultaneously, impacting overall speed and performance.

A wider data line, such as one of 64 bits, can carry more data in a single operation than a 32-bit line.

The data line directly influences the CPU's ability to access data quickly, making it essential for high-speed computing tasks.

Computer Buses

The **address line** carries memory addresses from the CPU to memory or other devices, pinpointing where data should be retrieved or stored.

- RAM contains thousands storage spaces, which we call **memory locations**. Each such memory location contains a byte of data (8 bits) and has its own address, which is a number that represents its relative location inside main memory and lets the CPU quick find and access that piece of data/variable.

Unlike the data line, the address line generally travels in one direction, from the CPU to the memory or device.

The width of the address line dictates the maximum addressable memory space; for example, a 32-bit address line can address up to 4 GB of memory. A larger address line allows the system to access a larger memory range, which is vital in modern computing.

By isolating address information on a dedicated bus, the system optimizes data transfer and memory management.

Computer Buses

The control lines carry commands and control signals between the CPU and other devices.

These signals coordinate activities within the computer, such as reading and writing data or indicating interrupts and resets.

Control lines ensure that components are synchronized and respond correctly to each other's requests during data transfer.

Examples of control signals include read/write signals and interrupt requests, which help manage the flow and execution of instructions, and the number of the device that is currently using the bus.

The control line is essential for stable and coordinated operations within the system, enhancing reliability and efficiency.

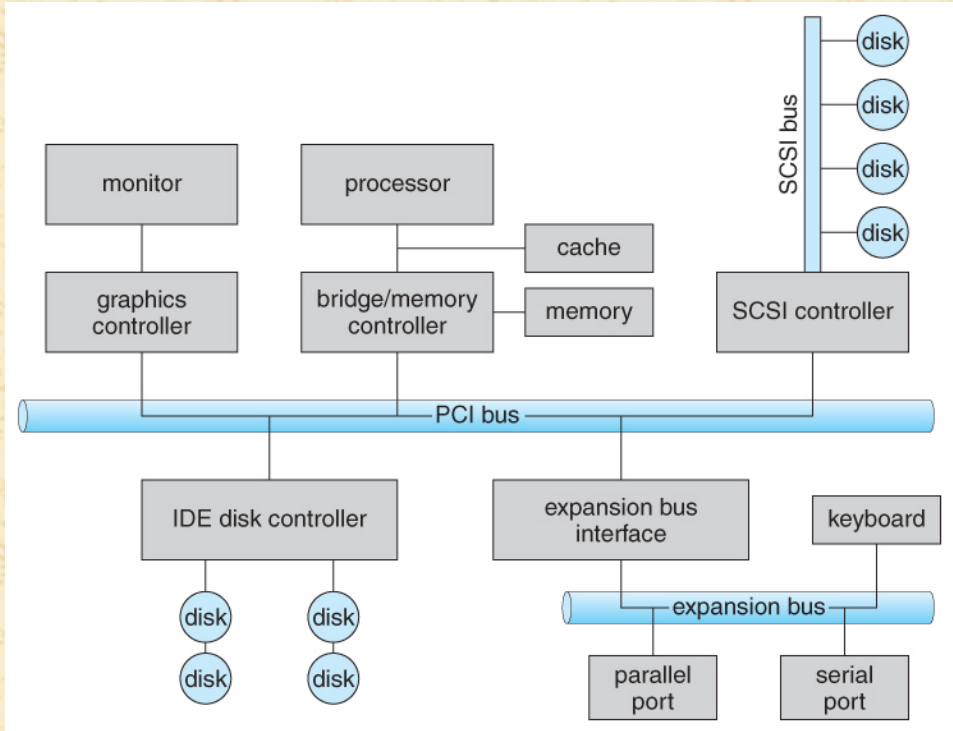
Computer Buses: Examples

Examples of modern buses include **PCIe (Peripheral Component Interconnect Express)**, **USB (Universal Serial Bus)**, **SATA (Serial ATA)**, and the **Expansion** bus.

- **PCI** or **PCIe** ("Peripheral Component Interconnect Express") is known for high data transfer rates and is often used for connecting graphics cards, storage drives, and network cards. The reason for the high speed of this bus is that it is shorter (so its takes less time for data to reach its destination on the bus.)
- **USB** is widely used for peripheral devices like keyboards, mice, and external drives, supporting data transfer and device power.
- **SATA** is a standard interface for connecting hard drives and SSDs, offering reliable speeds for data storage applications.
- **Expansion** buses connect main memory (or the CPU) with I/O devices, such as the screen, the keyboard, the mouse, etc. This bus is longer than the PCIe bus, which means that it transfers data slower than PCIe.

Each bus type is optimized for its specific purpose, with speeds and capabilities suited to its target devices.

Computer Buses: Examples



A typical PC bus structure. Taken from Bell, John T. "I/O Systems." University of Illinois, Chicago.

CPU: Registers

Any questions so far?

Computer Instructions

What is an instruction?

An **instruction** (or **machine instruction**) is a sequence of bits like this one:

```
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

that tells the CPU what action to take (e.g., calculate something, copy data to RAM, fetch data from RAM, make a decision, etc.)

An instruction has 2 parts: the **opcode** (= **operation code**) and 0 or more **operands** (data affected by the operation.)

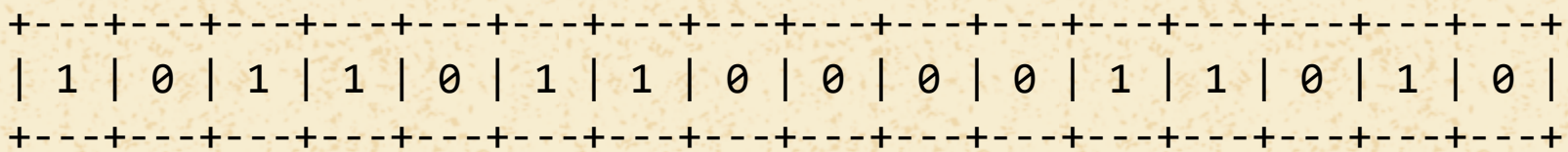
```
   Opcode                          Operand #1
+---+---+---+---+   +---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 1 | 1 | 0 |   | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
+---+---+---+---+   +---+---+---+---+---+---+---+---+---+---+---+---+
```

Computer Instructions

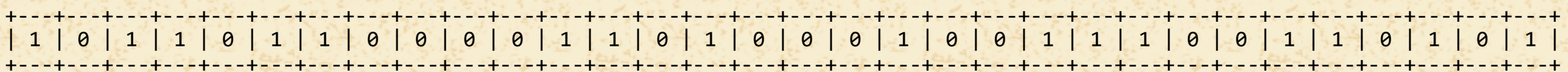
On modern computers, the length of an instruction is usually 32 bits or 64 bits.

The set of rules in a certain computer that tell how a machine instruction is structured is called **ISA**: **Instruction Set Architecture**.

ISA tells how many bits an instruction consists of. For example, whether an instruction must feature 16 bits:



or 32 bits:



Computer Instructions

The length of an instruction depends on how many instructions are supported in total (more instructions = greater length).

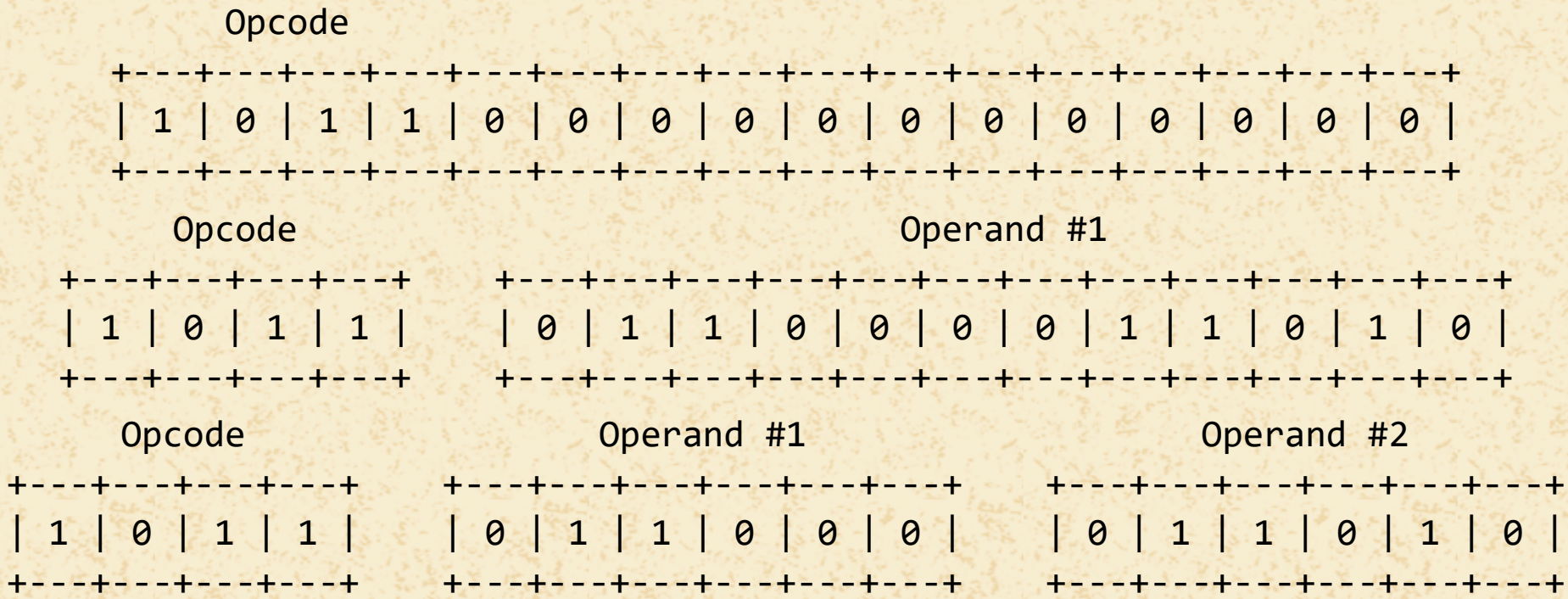
ISA also indicates the:

- Number of operands that an instruction can include.
- Length in bits of the opcode and of the operands.
- Types of operations (e.g., addition, multiplication.)
- Location/source of the data (inside a CPU register or in RAM.)
- Destination of the data (a register or RAM.)
- Types of operands (e.g., number, address, address of address, etc.)
- The storage order of the bits/bytes (left-to-right or right-to-left.)

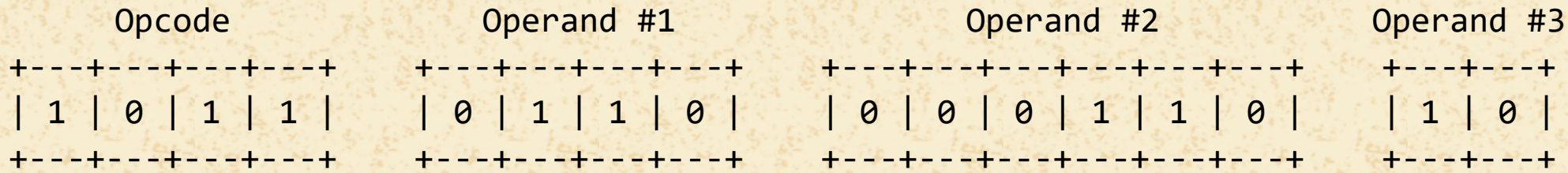
We will expand on those on the next slides.

Computer Instructions: Number & Length of Operands

An instruction may include no operands (just the opcode), 1 operand, 2 operands, or even 3 operands:



Computer Instructions: Number & Length of Operands



Most architectures today support instructions that contain between 0 to 3 operands. The operands don't even have to be all of the same length and don't have to have fixed length:

- Instructions whose operands have **fixed length** (e.g., each operand must contain exactly 4 bits) are fast and result in better execution performace. The cons are that they waste memory and disk space (if certain operands are short, we still must use that fixed size.)
- Instructions whose operands have **variable length** result in space savings but will take longer time to decode and execute.

In reality, a modern computer will support $2-3$ instruction patterns, e.g., one 64-bit instruction with 1 operand of size 32 bits, another 64-bit instruction with 2 16-bit operands , and a 3rd, 86-bit instruction with 1 64-bit operand.

Computer Instructions: Types of Operations

We can divide instructions into several type categories:

1. Arithmetic Operations (involving integers or floating-point numbers):
   - **`ADD`**: Adds up 2 or 3 numbers. Example: `ADD R1, R2` to add the number in register `R2` to the number in register `R1`.
   - **`SUB`**: Subtracts numbers. Example: `SUB R1, X, Y` to subtract the memory variables `X` and `Y` from register `R1`.
   - **`MUL`**: Multiplies numbers. Example: `MUL X` to multiply the variable `X` by a specific accumulator register.
   - **`DIV`**: Divides numbers. Example: `DIV X, Y` to divide the variable `X` by the variable `Y`, and store the result in `X`.
   - **`INC`**: Increments a number (= add `1` to it.) Example: `INC R1` to add `1` to the register `R1`.
   - **`DEC`**: Decrements a number (= subtract `1` from it.) Example: `DEC X` to subtract `1` from the variable `X`.
   - **`NEG`**: Changes the sign of a number from positive to negative and vice versa. Example: `NEG R2` to negate `R2`.

[Note that an instruction's name on one computer may differ from its name on another computer. E.g., a computer may call the multiplication instruction as `MUL`, `MULT`, or `MULTIPLY`: it depends on the decision of the computer's architecturers.]

Computer Instructions: Types of Operations

2. Boolean Logic Operations (involving bits, usually a register's bits):
   - **`NOT`**: flips the bits of a number. Example: `NOT X` flips the bits of variable `X` in memory: X'.
   - **`AND`**: ANDs numbers. Example: `AND R1, R2, R3` will do the bitwise AND operation $R1 \cdot R2 \cdot R3$.
   - **`OR`**: ORs numbers. Example: `OR R1, R2` will do the bitwise OR operation $R1 + R2 + R3$.
   - **`XOR`**: XORs numbers. Example: `XOR R1` will do the bitwise XOR operation $AC \oplus R1$ (`AC` is an accumulator register.)
   - **`TEST`**: ANDs numbers, but only sets bits in the status register: if the result of the ANDing is a 0, the zero flag (bit) is set to 1, and if the result of the ANDing is a negative, the sign flag (bit) is set to 1. The registers or variables themselves don't change. Example: `TEST X, Y`.
   - **`CMP`**: Compares numbers by subtracting one from the other, but only sets bits in the status register: if the result of the subtraction is a 0, the zero flag (bit) is set to 1, and if the result of the subtraction is a negative, the sign flag (bit) is set to 1. The registers or variables themselves don't change. Example: `CMP X, R1`.

Computer Instructions: Types of Operations

3. Other Bit Manipulation Operations (involving bits):
   - **SHL**: shift bits to the left. Example: `SHL R1, 2` shifts `R1` twice to the left (00111000 becomes 11100000).
   - **SHR**: shift bits to the right. Example: `SHR R1, 3` shifts `R1` 3 times to the right (00111000 becomes 00000111).
   - **SAL**: ("shift arithmetic left") same as `SHL`.
   - **SAR**: ("shift arithmetic right") similar to `SHR`, except that the sign bit remains the same. Example: with `SHR R1, 3`, 10111000 becomes 10000111 (the leftmost bit remains the same.)
   - **ROL**: wraps/rotates around to the left. Example: with `ROL R1, 2`, 01011100 becomes 01110001.
   - **ROR**: wraps/rotates around to the right. Example: with `ROR R1, 3`, 00111101 becomes 10100111.
   - **RCL**: wraps/rotates around to the left through the carry bit. Example: with `RCL R1, 2`, 01011100 w/ carry 1 becomes 01110010 carry 1.
   - **RCR**: wraps/rotates around to the right through the carry bit. Example: with `RCR R1, 3`, 00111101 w/ carry 0 becomes 01000111 w/ carry 1.

Computer Instructions: Types of Operations

4. Data Movement (= Data Flow) Operations:
   - **`MOV`**: ("move") copies the contents of a register to memory, or vice versa. Example: `MOV R1, X` copies the variable `X` to the register `R1`.
   - **`MVR`**: ("move registers") copies the content of one register to another. Example: `MVR R1, R2` copies `R2` to `R1`.
   - **`LOD`**: ("load") copies the content of a memory variable to a register. Example: `LOD X` copies the content of `X` into the accumulator register `Ac`.
   - **`STO`**: ("store") copies the content of a register into a memory variable. Example: `STO X` copies the content of the accumulator register `Ac` into the variable `X`.
   - **`PUSH`**: adds ("pushes") a number onto the stack data structure of the program stored in main memory. Example: `PUSH R1` pushes `R1` to the stack.
   - **`POP`**: removes ("pops") a number from the stack into a register/variable. Example: `POP R1` pops an item from the stack and saves it in `R1`.
   - **`EXC`**: swaps ("exchanges") two contents. Example: `EXC R1, X` exchanges `X` with `R1`.

Computer Instructions: Types of Operations

4. Transfer of Control (= Control Flow) Operations:
   - **`NOP`**: ("no-op: no operation") don't do anything, and go to the next instruction. Example: `NOP`.
   - **`HLT`**: ("halt") stop a program's execution, and exit the program. Example: `HLT`.
   - **`JMP`**: ("jump") unconditionally jump to (= go to) another instruction. Example: `JMP X` will cause the CPU to continue exeuction starting from the instruction inside address `X`; otherwise, it'll continue to the next instruction.
   - **`JZ`**: ("jump on zero") jump to (= go to) another instruction if the zero flag of the status register is on. Example: `JZ Y` will jump to the instruction inside address `Y` if the zero flag is 1; otherwise, it'll continue to the next instruction.
   - **`JNZ`**: ("jump on not zero") jump to (= go to) another instruction if the zero flag of the status register is off. Example: `JNZ Y` will jump to the instruction inside address `Y` if the zero flag is 0; otherwise, it'll continue to the next instruction.
   - **`JN`**: ("jump on negative") jump to (= go to) another instruction if the sign flag of the status register is on. Example: `JN X` will jump to the instruction inside address `X` if the sign flag is 1; otherwise, it'll continue to the next instruction.
   - **`JNN`**: ("jump on not negative") jump to (= go to) another instruction if the sign flag of the status register is off. Example: `JNN X` will jump to the instruction inside address `X` if the sign flag is 0; otherwise, it'll continue to the next instruction.

Computer Instructions: Types of Operations

5. I/O Operations vary greatly from one architecture to another. These have to do with getting data from an input device like the keyboard, sending data to an output device such as the display/screen, pausing the execution of a program when an I/O device has data to transfer to the program, etc.
6. Special-Purpose Operations also vary greatly from one architecture to another and include actions like flag control, cache management, address calculation, and other instructions beyond those listed in the previous categories.

This isn't an exhaustive list of all the instruction types and examples that a computer might have: when exploring an ISA, you will discover plenty other instructions and variations thereof.

Make sure to check not only what kinds of instructions exist but also the way in which you can use them: how many operands (= arguments) they accept, and what they do with these arguments.

An example of the instruction set of a major CPU manufacturer: Intel® 64 and IA-32 Architectures Software Developer's Manual [PDF]: Page 111 (Chapter 5: Instruction Set Summary).

Computer Instructions: Types of Operations

Any questions so far?

Computer Instructions: Locations of Data

The CPU may store data in various places:

1. **Accumulator architecture**: the accumulator register `AC` always implicitely participates in an operation. For instance, `ADD R1` adds the contents of `R1` to `AC`, `LOD X` copy the contents of `X` to `AC`, etc.
2. **Stack architecture**: operands and data is found inside the stack, which is a LIFO data stucture stored in main memory. This architecture features a great number of calls to the `POP` and `PUSH` instructions to extract and put data inside the stack and use that data for calculations. For instance, `POP R1` pops a number from the top of the stack and pastes it into `R1`, `ADD R1` pops a number from the top of the stack and adds it to `R1`, etc.
3. **General-purpose register architecture** (**GPR**): data is located in various general-purpose registers, so we are not limited to only an accumulator or a stack. This is the most common type of architecture, and can further be classified as:
   a. **Memory-Memory architecture**: all the operands are memory variables, e.g., `X`, `VAR`, etc.
   b. **Register-register architecture**: all the operands must be stored inside registers, e.g., `R1`, `AC`, etc.
   c. **Register-Memory architecture**: some operands are memory variables, while others are registers.

Computer Instructions: Types of Operands (Address Modes)

What kinds of operands can we pass?

Address Mode	Explanation	Example: `LOD 100`
Immediate	The operand itself is present in the instruction.	100 itself is the operand/data.
Direct	The operand is an address to the data.	100 is the RAM address.
Indirect	The operand is an address to an address.	100 is the address of the address.
Register	The operand is a register storing the data.	100 is the register's name/serial number.
Register Indirect	The operand is a register with an address.	100 is a register containing an address.
Indexed	Operand + index register are the address.	100 + `IND` is the RAM address.
Based	Operand + base register are the address.	100 + `BASE` is the RAM address.
Stack	The operand is implicitly on the stack	`LOD` pop from the stack and loads to `AC`.

Computer Instructions: Types of Operands (Address Modes)

Example: Suppose main memory and the CPU registers contain the following data:

RAM Address	Content
100	200
200	400
300	100
400	300
500	1000
600	700

CPU Registers	Content
100	500
IND	500
BASE	300

and that the stack contains the number 600 on its top.

On the next slide, we'll explain what number the instruction `LOD 100` will load into the accumulator `AC`.

Computer Instructions: Types of Operands (Address Modes)

According to the data in the previous example, `LOD 100` will result in the following number to be stored in `AC`:

Address Mode	Result	Explanation
Immediate	100	100 itself is the operand/data.
Direct	200	100 is the RAM address pointing to 200.
Indirect	400	100 is the address pointing to 200, which points to 400.
Register	500	Because register 100 contains 500.
Register Indirect	1000	Because register 100 contains the address 500 pointing to 1000.
Indexed	700	$100 + 500$ (`IND`) $= 600$ is the RAM address pointing to 700.
Based	300	$100 + 300$ (`BASE`) $= 400$ is the RAM address pointing to 300.
Stack	600	600 is popped from the stack and put in `AC`.

Computer Instructions: Types of Operands (Address Modes)

Any questions so far?

Computer Instructions: Storage Order

In most computer architectures, every byte stored in main memory has its own address.

- This means that the CPU can easily retrieve any byte from RAM, instantly.

Many data structures, even integers, store more than 1 byte of data.

- For example, an `int` data type in C/C++ usually consists of 2 or 4 bytes.

The question is, in what order should we store the bytes: left-to-right, or right-to-left?

This kind of storage order is called **Endianess** (based on the word 'end'):

1. On a **big endian** device, the bytes of a data type are stored left-to-right (or top to bottom.)
2. On a **little endian** device, the bytes of a data type are stored right-to-left (or bottom to top.)

Computer Instructions: Storage Order

The term "endian" is based on the book Gulliver's Travels, in which the citizens of Lilliput were debating over two customs: breaking an egg either from its small (little) end or big end. Just like them, computer engineers are also divided into two groups: either those preferring to store bytes left-to-right, or those storing bytes right-to-left.

Example: the 2-byte integer 10010111_01101011 will be stored either as

Address	100	101
Content	10010111	01101011

if we follow the big endian principle, or as

Address	100	101
Content	01101011	10010111

if we follow the little endian principle.

Computer Instructions: Storage Order

Big-Endian Pros:

- Big-endian systems store the most significant byte first, making it easier to understand and compare data when looking at memory in a human-readable format.
- This ordering aligns well with numeric systems, like those commonly used in network protocols (e.g., IP addresses), enhancing compatibility in network data transfer.
- Many older systems use big-endian format, so there is widespread legacy support.
- Debugging in big-endian systems can be simpler since the byte ordering often matches expected numeric notation.

Big-Endian Cons:

- Such systems can be less efficient for certain arithmetic operations on smaller data types, as they often require additional shifts to access the least significant byte.
- This format may require byte-swapping when exchanging data with little-endian systems, adding complexity in cross-platform development.
- Memory handling may sometimes be slower on modern processors, especially when bitwise operations are involved.

Computer Instructions: Storage Order

Little-Endian Pros:

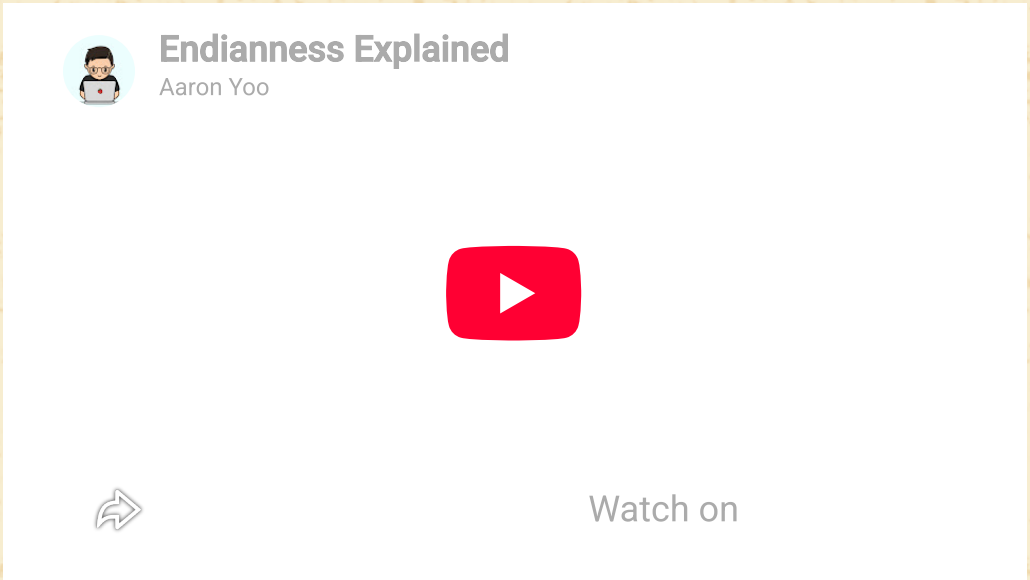
- Little-endian systems store the least significant byte first, which can simplify certain arithmetic and Boolean operations.
- This approach allows for more efficient data access on some modern processors, as they often begin computations with least significant bits.
- Little-endian formats are widely used in x86 architectures, making them common in personal computing and widely supported in modern applications.

Little-Endian Cons:

- Such systems can be less intuitive when viewing data in memory, as the least significant byte appears first, which can be confusing for human readability.
- Data exchange with big-endian systems often requires byte-swapping, complicating network programming.
- This format may be less compatible with network protocols that expect big-endian order, requiring extra handling in network applications.

Computer Instructions: Storage Order

A beautiful explanation of big vs. little endianess:



https://www.youtube.com/watch?v=LxvFb63OOs8

Computer Instructions: Storage Order

Any questions so far?

Fetch–Decode–Execute Cycle

The **fetch–decode–execute cycle** is the basic activity of a CPU. It describes how the CPU retrieves an instruction (copies it from main memory into a register,) understands what to do with it, and then carries out (= executes) the instruction.

In each cycle, the CPU processes one instruction by going through these 3 main stages: fetch, decode, and execute.

This cycle allows a computer to perform complex tasks by breaking down instructions into smaller, manageable steps.

This approach to processing instructions is foundational for all modern computing, enabling rapid and continuous task execution.

On the next slides, we'll examine each stage in detail to understand how the CPU completes an instruction.

Fetch–Decode–Execute Cycle

In the **fetch** stage ("fetch" is a synonym of "bring",) the CPU copies an instruction from main memory in the following manner:

1. The **program counter** (**PC**) register of the CPU's control unit stores the address (location) in RAM of the next instruction that should be fetched.
2. The CPU sends a signal to main memory over the control line to indicate that the CPU wants to read data from main memory. The CPU then also sends the address (i.e., the value of the program counter) over the address line.
3. Once receiving that electric signal, main memory responds by sending the content of that address (which is the instruction the CPU is looking for) to the CPU over the data line.
4. After the CPU receives the instruction, it increments the program counter to point to the next instruction in memory.

This process ensures that instructions are fetched in the correct order, allowing the program to run smoothly.

The fetched instruction is temporarily stored in another register, called the **Instruction Register** (**IR**), which is also located inside the CPU's control unit.

Fetch–Decode–Execute Cycle

In the **decode** stage, the CPU interprets the instruction that was fetched, determining what actions are required to complete it.

The CPU's control unit breaks down the instruction into parts, identifying the opcode (operation code) and the operands.

The opcode tells the CPU what operation to perform, while the operands provide any data or addresses required for that operation.

- As the name suggests it, a **decoder** circuit is used during the decode stage to direct the CPU to the right activity: it looks at the n opcode bits, and directs the CPU to one of the 2^n possible computation/action types.

This breakdown helps the CPU understand exactly what needs to be done and prepares it for the execution stage.

Decoding is crucial because it translates the instruction into signals that the CPU's components can understand and act upon.

Fetch–Decode–Execute Cycle

During the **execute** stage, the CPU performs the operation specified by the instruction.

This might involve arithmetic calculations, logic operations, data movement, or other tasks defined by the opcode.

The CPU uses its ALU (Arithmetic Logic Unit) (including the various combinational circuits installed in the ALU) or other components to complete the task as required.

After the operation is done, any result produced may be stored in a register or in memory, depending on what the instruction indicated.

The CPU then starts a new fetch–decode–execute cycle when it begins to process the next instruction in the program.

Fetch–Decode–Execute Cycle

Example: consider some computer program that has an instruction to add two numbers; this instruction, along with the other instructions of the program, is stored in main memory.

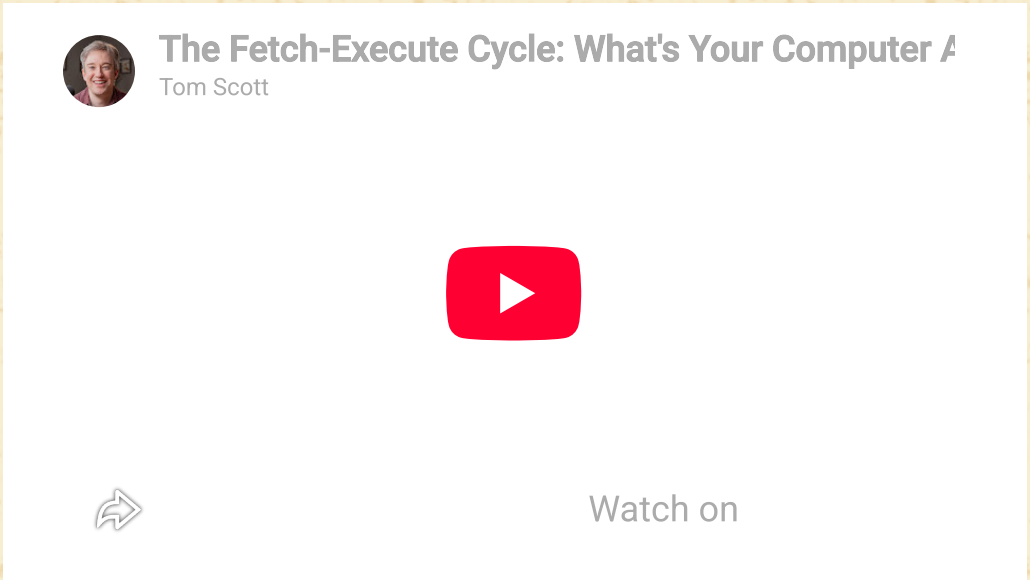
1. First, the CPU **fetches** the addition instruction from memory, using the program counter to find its address in memory.
2. Next, the CPU **decodes** the instruction, identifying it as an addition operation with two operands.
3. Finally, the CPU **executes** the operation using a Ripple-Carry Adder circuit in the ALU: it adds the numbers and stores the result in a specified memory location or CPU register.

This simple example illustrates how the fetch–decode–execute cycle enables the CPU to perform even complex tasks step-by-step.

Why can't we store all of the instructions of a program directly on the CPU? Why do we use RAM?
Answer: The CPU has only several register in it, so its space is limited. Programs, especially long ones, simply won't fit into these registers, so they must be kept in a larger container: main memory.

Fetch–Decode–Execute Cycle

Tom Scott explaining the fetch–decode–execute cycle:



https://www.youtube.com/watch?v=Z5JC9Ve1sfI

Fetch–Decode–Execute Cycle

Any questions so far?

Pipelining

Pipelining is a technique used in **CPU** design to improve the performance of instruction execution.

It allows multiple instructions to be processed at different stages of the fetch–decode–execute cycle simultaneously.

By dividing the instruction processing into stages and processing them in parallel, pipelining can significantly reduce the time it takes to complete a sequence of instructions.

This process is often compared to an **assembly line**, where each stage of instruction processing is completed independently and in parallel with other stages, to create more goods in a shorter amount of time.

To implement pipelining, a computer would take the fetch–decode–execute cycles of several instructions and divide them into further, smaller steps (an example is shown on the next slide.) A step from one instruction could be done at the same time as a step of another instruction: one CPU core can fetch an instruction X, decode an instruction Y, and execute an instruction Z all at the same time!

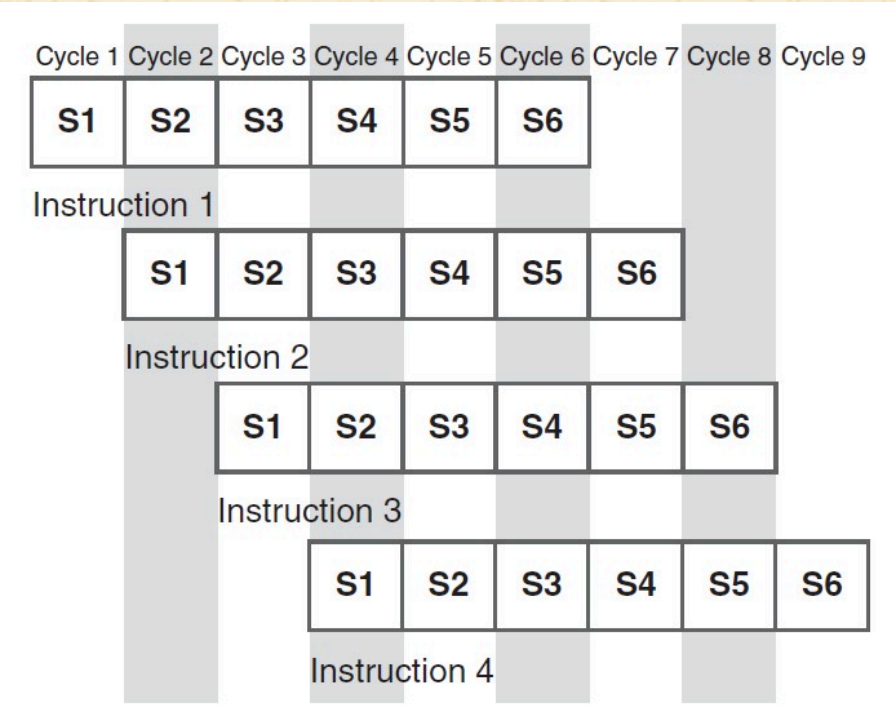
Pipelining

Pipelining divides the fetch–decode–execute cycle into separate stages, which may include:

1. **Instruction fetch**: The CPU fetches the instruction from memory.
2. **Instruction decode**: The instruction is decoded to understand what actions to perform.
3. **Find operand addresses**: calculate the address in main memory where each operand is stored.
4. **Fetch Operands**: The CPU fetches the instruction from main memory.
5. **Execute Instruction**: The CPU performs the operation specified by the instruction.
6. **Write Back**: The result of the operation is written back to the register or memory.

In a **sequential** fetch–decode–execute cycle, each instruction must complete all stages before the next instruction can start. This means that only one instruction is being processed at a time, which slows down overall performance.
If the CPU uses pipelining, however, each of these stages operates independently, allowing the CPU to process different parts of multiple instructions at the same time, thereby boosting the overall performance.

Pipelining



Pipelining with a 6-stage cycle and 4 instructions. Figure 5.5 on page 329 of *Essentials of Computer Organization and Architecture* by Linda Null, 2023.

Pipelining

Let's do some math now to find how fast pipelining works compared to sequential exeuction.

Suppose that some code has n instructions, and that the CPU can break the fetch-decode-execute cycle into k small steps (the previous slide introduced 6 steps, for example.) Also, suppose that each small step needs only one clock cycle to complete.

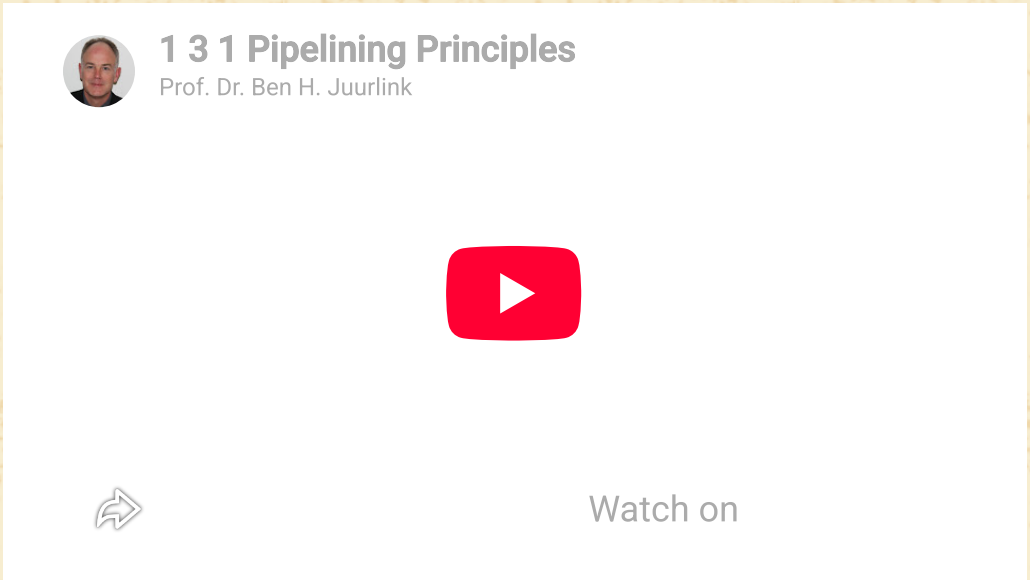
- If we run our instructions one-by-one, without pipelining, we will need a total of $n \cdot k$ clock cycles to run these n instructions.
- However, with pipelining, we notice from the previous image that the number of clock cycles we used was $6 + 4 - 1 = 9$ clock cycles (6 was the number of small steps, and 4 was the number of instructions.
  In general, if the CPU uses pipelining, the number of clock cycles will be equal to $k + n - 1$.

Note that, if $k \geq 3$ and $n \geq 3$, the quantity $k + n - 1$ is much smaller than $n \cdot k$, which means that pipelining is indeed faster comparing to sequential exeuction! (Remember: fewer clock cycles needed = faster execution.)

Pipelining

Fun class activity: How many clock cycles are needed to run a program (a) without and (b) with pipelining if:

1. The program has 25 instructions, and the fetch-decode-execute cycle has 6 steps?
2. The program has 100 instructions, and the fetch-decode-execute cycle has 11 steps?



https://www.youtube.com/watch?v=zPmfprtdzCE

Pipelining: Challenges and Solutions

As we just learned, in pipelining, multiple instructions are processed simultaneously, which can improve CPU efficiency.

Nevertheless, this parallel processing can lead to issues called **pipelining hazards**, which may cause incorrect results or delays (= slower computations).

The main types of pipelining hazards are **resource conflicts**, **data dependencies**, and **conditional branch statements**.

Each of these hazards presents unique challenges to ensuring correct and efficient execution of instructions.

In the following slides, we will define each hazard, discuss its impact, and explore solutions for managing it in an effort to still allow a computer to perform pipeline.

Pipelining: Challenges and Solutions

Resource conflicts occur when multiple instructions require the same hardware resource simultaneously.

This conflict can happen if instructions need to use the ALU, memory, or another hardware component at the same time. Note that a hardware component can be used only for 1 task at a time.

Resource conflicts can cause delays or incorrect calculations if instructions are not managed properly.

One solution is to introduce **additional hardware units**, allowing more instructions to be processed without conflict.

Another approach is **resource scheduling**, which organizes instruction timing to avoid simultaneous demands on the same resource. An example would be to allow a reading into main memory while pausing a writing operation into main memory.

We have yet another solution: design the RAM device to feature two cables: one for reading data from RAM (= data is copied from RAM to the CPU) and another one for writing data to RAM (= data is copied from the CPU to RAM.)

Pipelining: Challenges and Solutions

Data dependencies occur when one instruction relies on the result of a previous instruction, creating a dependency.

If the dependent instruction is processed before the first instruction completes, it may lead to incorrect results.

This type of hazard is common in arithmetic and logic operations where one result is needed by the next instruction.

A common solution is **data forwarding**, where the result is sent directly from one stage to another without waiting for the cycle to complete.

Another technique is **stalling**, where the CPU temporarily pauses the pipeline until the required data is available, ensuring accurate calculations.

Pipelining: Challenges and Solutions

Conditional branch statements introduce hazards by altering the normal flow of instructions based on specific conditions.

If the CPU does not know which branch to follow, it may fetch instructions that are later discarded, causing pipeline delays.

This uncertainty can lead to wasted processing cycles and reduced efficiency.

One solution is **branch prediction**, where the CPU predicts the branch it will likely need to take and fetches those instructions in advance.

If the prediction is correct, execution continues smoothly; if incorrect, **rollback mechanisms** are used to correct the pipeline and fetch the right instructions.

Pipelining: Challenges and Solutions

Any questions so far?

CPU Control Types

How do control lines work? One of two general methods is used by every computer architecture.

In a **hardwired control unit**, control signals are generated by hardware using fixed logic circuits. This approach relies on combinational and sequential logic gates to directly interpret each machine instruction into specific signals for CPU components.

Hardwired control units are typically faster than other control methods because there is no intermediate decoding or microinstruction process.

These control units, however, are complex to design, as each CPU instruction requires dedicated logic circuitry, making changes or upgrades challenging.

Hardwired control is often found in systems where speed is critical and instructions are limited, such as in specialized processors.

CPU Control Types

Advantages of hardwired control:

- **Speed:** Hardwired control is generally faster because control signals are generated directly from hardware without an intermediate microinstruction layer.
- **Efficiency:** It can be highly optimized for specific tasks, making it suitable for specialized applications or high-performance tasks.
- **Reliability:** Because it relies on fewer stages and direct hardware connections, hardwired control can be more reliable with fewer points of failure.

Disadvantage of hardwired control:

- **Complexity:** It can be complex to design and is challenging to modify, making upgrades difficult if instruction sets change. Adding or changing even the smallest thing will require you to re-wire the entire control unit.

CPU Control Types

In a **microprogrammed control unit**, control signals are generated by a sequence of microinstructions stored in a special memory called the control memory.

Each microinstruction defines a set of control signals that the CPU needs to perform an operation or execute an instruction.

The control unit reads each microinstruction, generating signals to the CPU's components in a step-by-step manner, enabling more flexible control.

Unlike hardwired control, microprogrammed control can easily be modified by updating the microinstructions, making it adaptable to new instruction sets.

Microprogrammed control is often used in general-purpose CPUs, where flexibility and ease of upgrade are essential.

CPU Control Types

Advantages of microprogrammed control:

- **Flexibility:** Microprogrammed control allows for easier updates by modifying microinstructions, making it adaptable to new instructions or architectures.
- **Scalability:** It is simpler to add new instructions, making this approach suitable for complex, general-purpose processors.
- **Reduced Design Complexity:** Microprogrammed control is easier to design and debug, as changes can be made by updating the control memory rather than reconfiguring circuits.

Disadvantage of microprogrammed control:

- Despite these advantages, microprogrammed control may be slower than hardwired control due to the additional layer of microinstruction decoding.

CPU Control Types

Modern CPUs commonly use a **hybrid approach** that combines elements of both hardwired and microprogrammed control for optimized performance and flexibility.

Many high-performance CPUs use hardwired control for common, speed-critical operations while using microprogrammed control for complex or less frequent instructions.

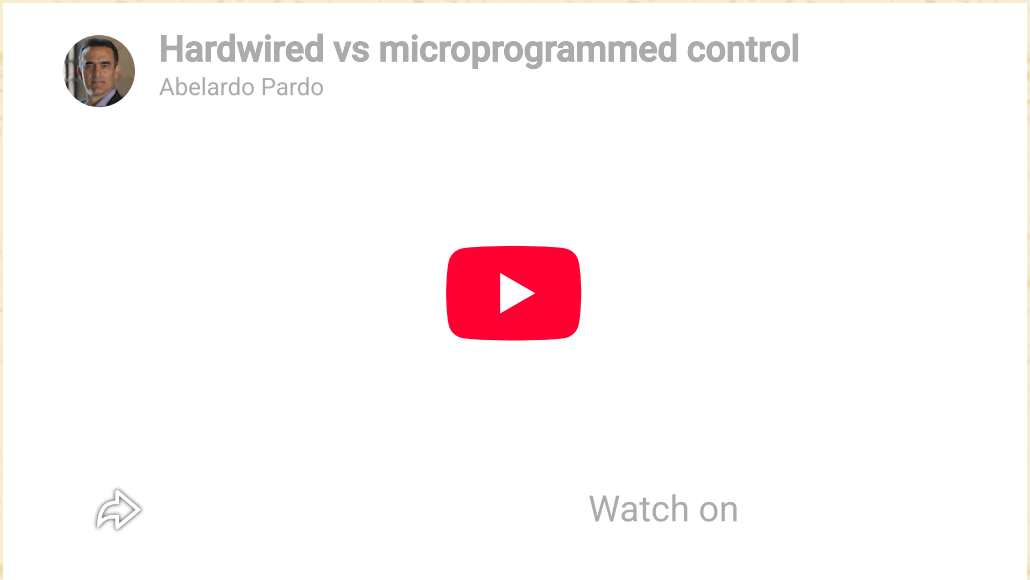
This hybrid approach enables efficient execution of a wide range of instructions while maintaining flexibility for future updates or changes.

The combination provides a balance of speed and adaptability, making it well-suited for today's multipurpose and high-performance computing needs.

As a result, modern CPUs achieve a balance that supports both high-speed processing and flexibility in adapting to evolving instruction sets.

CPU Control Types

An excellent explanation on hardwired vs. microprogrammed control methods:

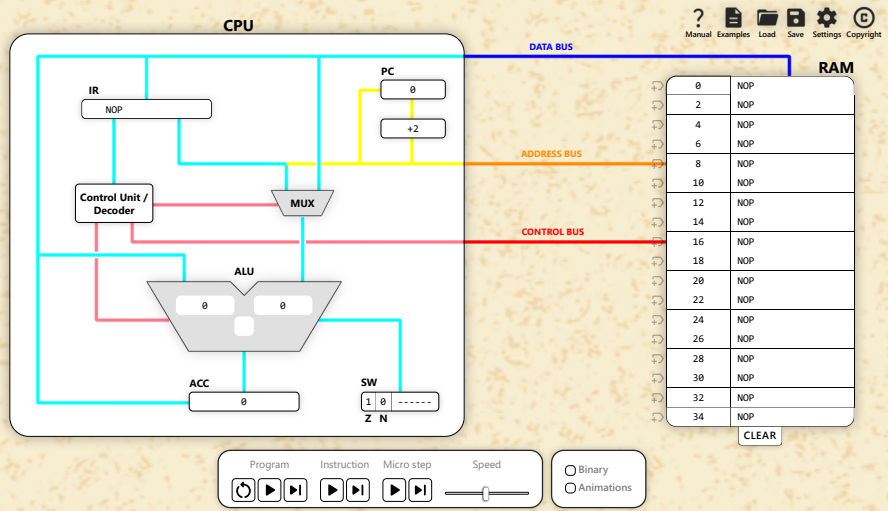


https://www.youtube.com/watch?v=pI7mDvUwiWE

CPU Control Types

Any questions so far?

CPU Simulator



Citation: Cancelli, Jonathan et al. "CPU Visual Simulator (CPUVSIM)". *GitHub*, 2021. URL: https://cpuvisualsimulator.github.io/.

CPU Simulator

Usage: To watch the simulator in action, click "Examples" on the top-right menu, choose one of the given examples, and then press the Play button (►) under "Program" at the middle-bottom of the screen. The simulator will start running the underlying program and show how each of the instructions of that program travels through the CPU and its buses, resulting in the correct answer at the end. Alternatively, press the Play button (►) under "Micro step", instead, to see the execution of just one instruction at a time.

For more information about what each example program does, and to learn what each of the computer instructions inside RAM (Random Access Memory = computer's memory) means, please view the manual page of the guide on the following slide or here: https://cpuvisualsimulator.github.io/manual.

MANUAL

Introduction

CPU Visual Simulator allows you to enter and visualize the execution of assembly language code. Instructions and numeric data can be inserted or modified directly in RAM. It is possible to define "labels" (identifiers to be used in place of memory addresses): these labels can then be used as parameters in jump instructions, or as variable identifiers. At any time, it is possible to switch between symbolic and binary representations. It is also possible to directly modify the Program Counter, the Accumulator or the Status Word's Negative and Zero flags. The program can be executed: normally (the CPU continues to execute instructions until execution is paused or the program ends), one statement at a time, or one step at a time. The text-to-speech function is also made available which, if enabled, activates a synthetic voice that explains what is being executed by the CPU.

Citation: Cancelli, Jonathan et al. "CPU Visual Simulator: Manual". *GitHub*, 2021. URL: https://cpuvisualsimulator.github.io/manual.

CPU Simulator: Manual

Any questions?

Computer Instructions: Sources

Bergmann, Seth D. (2018). Section 6.5 and Section 6.6, Chapter 7, Section 9.1, Section 9.2, and Appendix: MIPS Instruction Set. *Computer Organization with MIPS*. License: CC BY-NC-ND: Attribution-NonCommercial-NoDerivs.

CS301: Computer Architecture. Unit 2 and Unit 5. *Saylor Academy*, 2010. License: CC BY: Attribution.

Viswanath, Divakar. Chapter 3. *Scientific Programming and Computer Architecture*. The MIT Press, 2017. License: CC BY-NC-ND: Attribution-NonCommercial-NoDerivs.

Wienand, Ian. (2019). Section 3.1. *Computer Science from the Bottom Up*. License: CC BY-SA: Attribution-ShareAlike.