

Computer organization and architecture

Lesson 13

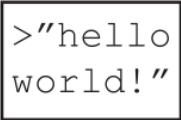


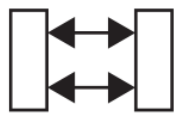
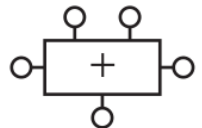

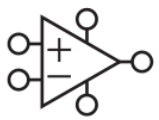


Microarchitecture

Microarchitecture is the specific arrangement of registers, ALUs, finite state machines, memories, and other logic building blocks needed to implement an architecture.

A particular architecture, may have different microarchitectures, each with different trade-offs of performance, cost, and complexity.

They all run the same programs, but their internal designs vary.

A single microarchitecture can execute several different architectures.

Application Software		Programs
Operating Systems		Device Drivers
Architecture		Instructions Registers
Micro-architecture		Datapaths Controllers
Logic		Adders Memories
Digital Circuits		AND Gates NOT Gates
Analog Circuits		Amplifiers Filters
Devices		Transistors Diodes
Physics		Electrons

Microarchitecture is also called **computer organization**.

– abbreviated as **μarch** or **uarch**

Architecture is often abbreviated as **ISA** (**instruction set architecture**).

The microarchitecture is usually represented as diagrams that describe the interconnections of the various **microarchitectural elements**.

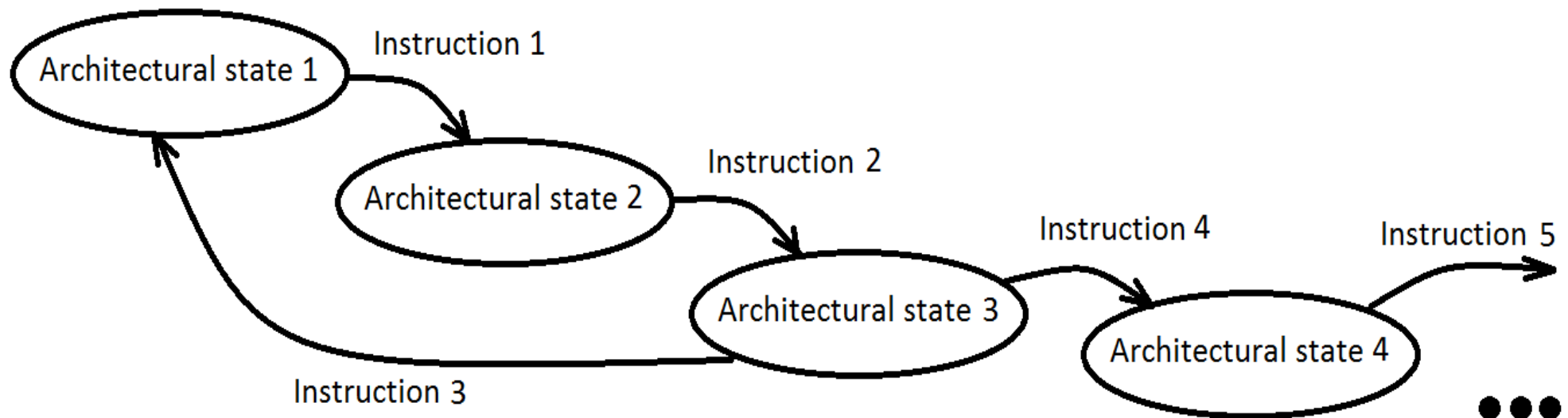
The microarchitecture diagram shows microarchitectural elements such as the arithmetic and logic unit and the register file as a single schematic symbol.

Each microarchitectural element is in turn represented by a circuit diagram describing the connections of the transistors used to implement it in some particular logic family (e.g. CMOS, TTL).

A computer architecture is defined by its instruction set and architectural state.

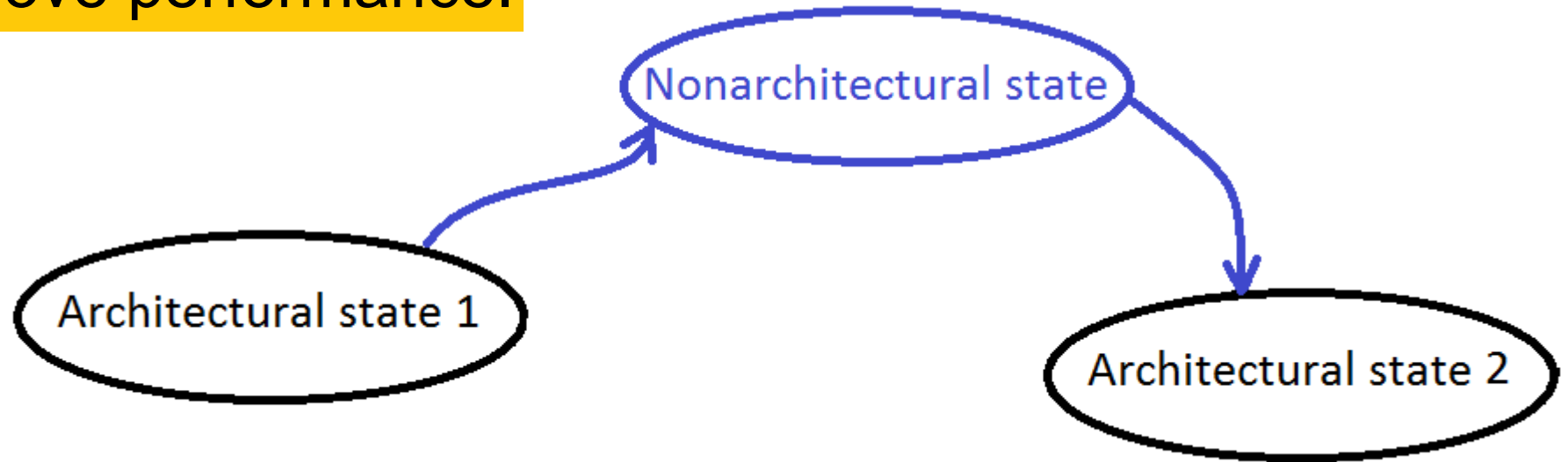
The **architectural state** for the ARM processor consists of 16 32-bit registers and the status register.

Based on the current architectural state, the processor executes a particular instruction with a particular set of data to produce a new architectural state.



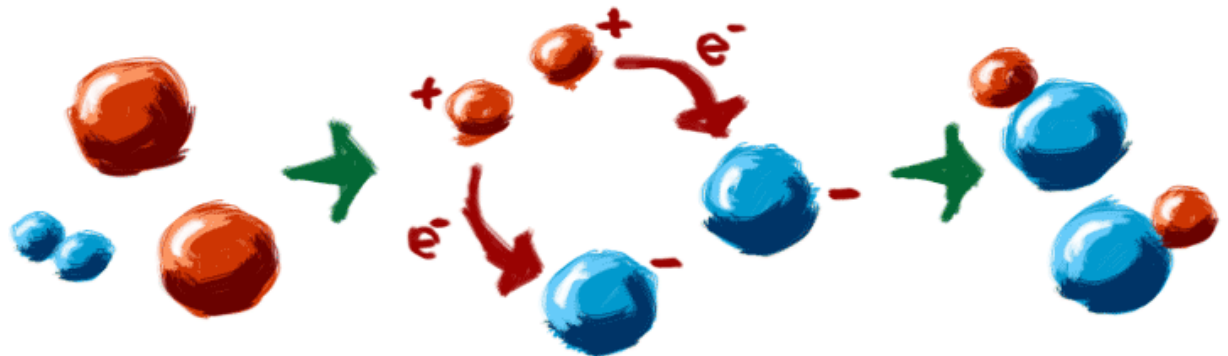
A processor is a finite state machine that jumps between architectural states.

Some microarchitectures contain additional **nonarchitectural states** to either **simplify the logic or improve performance.**



Conjecture:

Any finite state machine (e.g. chemical, biological, etc) can be thought of as a processor, and transitions between states as instructions.



To keep the microarchitectures easy to understand, we consider only a subset of the ARM instruction set:

Data-processing instructions:

ADD SUB AND ORR

with register and immediate addressing modes but
no shifts

Memory instructions:

LDR STR

with positive immediate offset

Branches: B

Once you understand how to implement these instructions, you can expand the hardware to handle others.

Microarchitectures consist of two parts:

- Datapath
- Control unit

The **datapath** operates on words of data.

It contains memories, registers, ALUs, multiplexers, etc

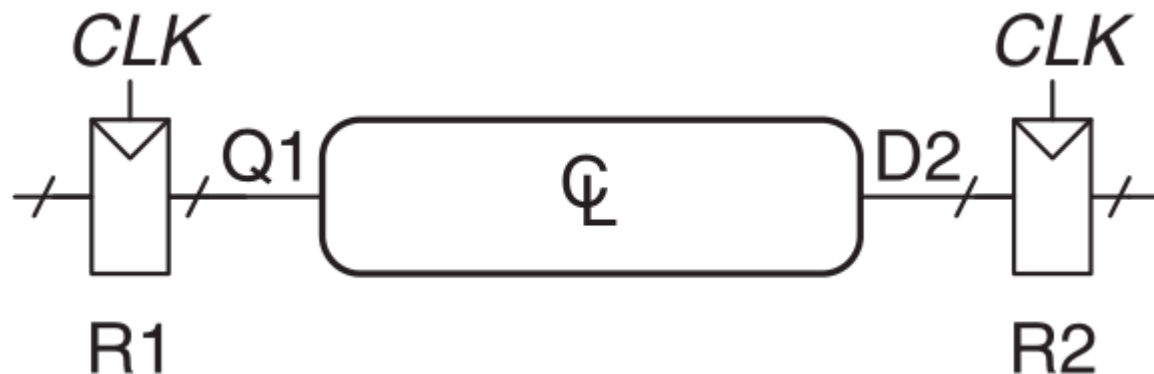
The **control unit** receives the current instruction from the datapath and tells the datapath how to execute that instruction.

Specifically, the control unit produces **multiplexer select, register enable, and memory write signals** to control the operation of the datapath.

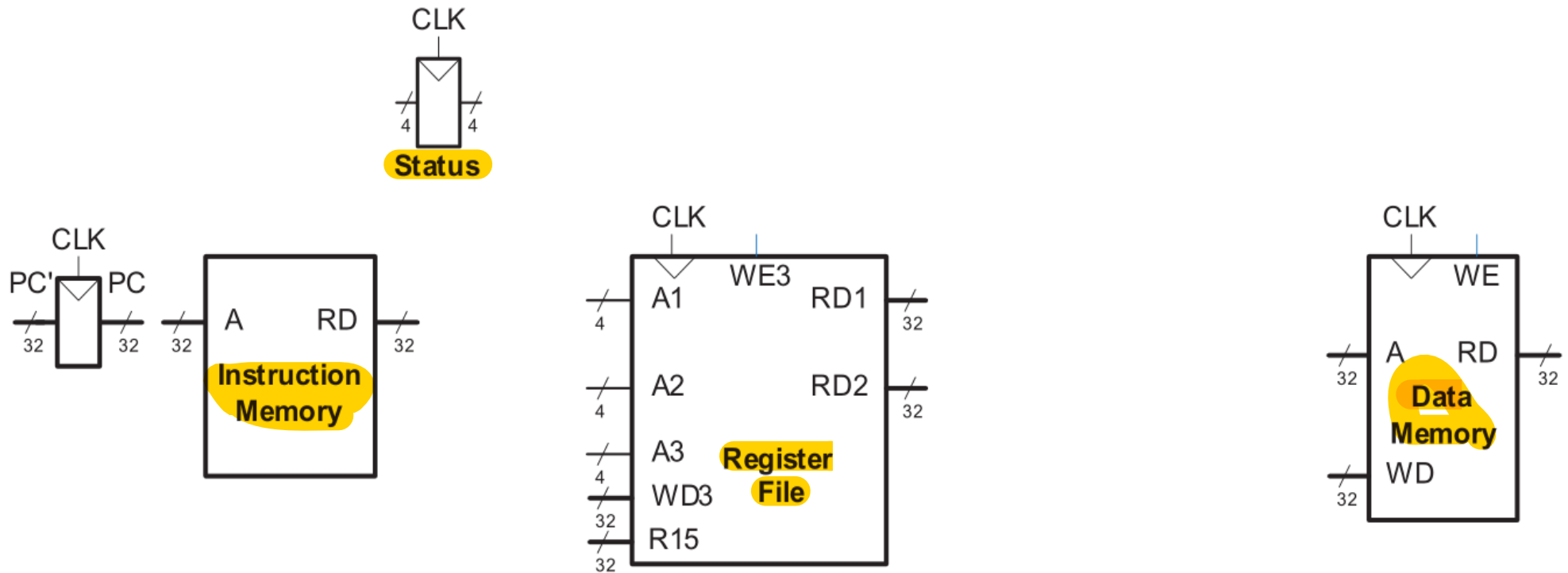
We start with hardware containing the state elements.

These elements include the memories and the architectural state (the program counter, registers, and status register).

Then, we add blocks of combinational logic between the state elements to compute the new state based on the current state.



State elements of ARM processor



State elements usually have a reset input to put them into a known state at start-up.

To save clutter, this reset is not shown.

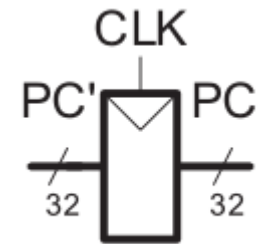
The instruction is read from part of memory.

Load and store instructions then read or write data from another part of memory.

Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data.

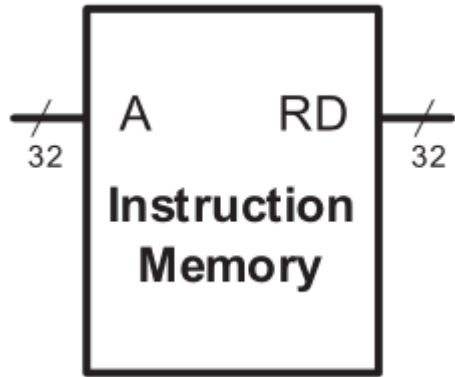
Although the **program counter** (PC) is logically part of the register file, it is read and written on every cycle independent of the normal register file operation and is more naturally built as a stand-alone 32-bit register.

Its output, PC, points to the current instruction.



Its input, PC', indicates the address of the next instruction.

ARM processors normally initialize the PC to 0x00000000 on reset.

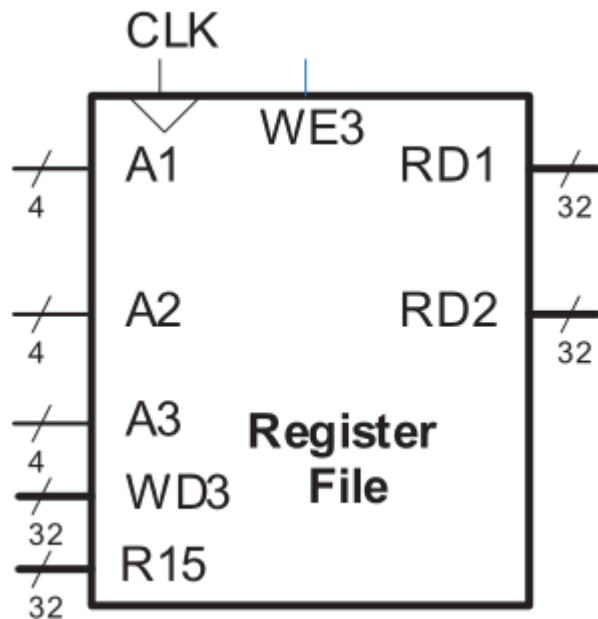


The **instruction memory** has a single read port.

It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD.

This is an oversimplification used to treat the instruction memory as a ROM.

In most real processors, the instruction memory must be writable so that the operating system can load a new program into memory.



The 15-element $\times 32$ -bit **register file** holds registers R0–R14

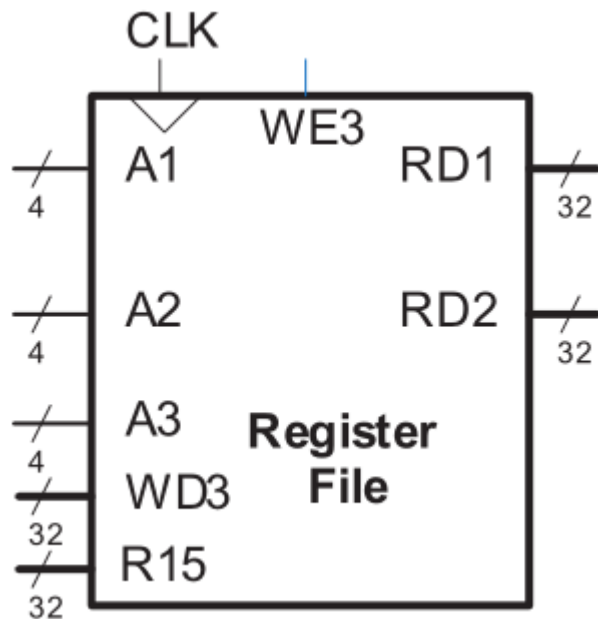
It has an additional input to receive R15 from the PC

2 read ports, 1 write port

The read ports take 4-bit address inputs, A1 and A2, each specifying one of 16 registers as source operands.

They read the 32-bit register values onto read data outputs RD1 and RD2, respectively.

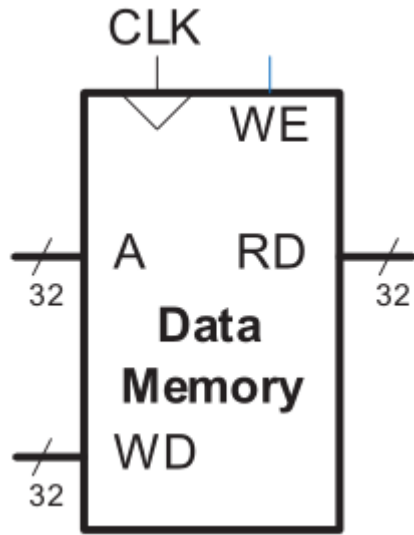
The write port takes a 4-bit address input, A3; a 32-bit write data input, WD3; a write enable input, WE3; and a clock.



If the write enable is asserted, then the register file writes the data into the specified register on the rising edge of the clock.

A read of R15 returns the value from the PC plus 8.

Writes to R15 must be specially handled to update the PC because it is separate from the register file.



The **data memory** has a single read/write port.

If its write enable, WE, is asserted, then it writes data WD into address A on the rising edge of the clock.

If its write enable is 0, then it reads address A onto RD.

The instruction memory, register file, and data memory are all read **combinationally**.

If the address changes, then the new data appears at RD after some propagation delay – no clock is involved

They are written only on the rising edge of the clock.

The state of the system is changed only at the clock edge.

The address, data, and write enable must **setup** before the clock edge and must remain stable until a **hold time** after the clock edge.

Because the state elements change their state only on the rising edge of the clock, they are **synchronous sequential circuits**.

The microprocessor is built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit.

Microarchitectures

We study 3 microarchitectures for the ARM architecture:

- single-cycle
- multicycle
- pipelined

They differ in the way that the state elements are connected together and in the amount of nonarchitectural state.

The single-cycle microarchitecture

The single-cycle microarchitecture executes an entire instruction in one cycle.

It has a simple control unit.

Because it completes the operation in one cycle, it does not require any nonarchitectural state.

However, the cycle time is limited by the slowest instruction.

The processor requires separate instruction and data memories, which is generally unrealistic.

The multicycle microarchitecture

The multicycle microarchitecture executes instructions in a series of shorter cycles.

Simpler instructions execute in fewer cycles than complicated ones.

The multicycle microarchitecture reduces the hardware cost by reusing expensive hardware blocks such as adders and memories.

Example: the adder may be used on different cycles for several purposes while carrying out a single instruction.

The multicycle microprocessor accomplishes this by adding several nonarchitectural registers to hold intermediate results.

The multicycle processor executes only one instruction at a time, but each instruction takes multiple clock cycles.

The multicycle processor requires only a single memory, accessing it on one cycle to fetch the instruction and on another to read or write data.

Therefore, multicycle processors were the historical choice for inexpensive systems.

Multicycle microarchitectures are used in inexpensive microcontrollers such as the 8051, the 68HC11, and the PIC16-series found in appliances, toys, and gadgets.



The pipelined microarchitecture

The pipelined microarchitecture applies pipelining to the single-cycle microarchitecture.

It can execute several instructions simultaneously, improving the throughput.

Pipelined processors split the execution of each instruction into multiple phases and allow different instructions to be processed in different phases simultaneously.

Early processor designs would carry out all of the steps for one instruction before moving onto the next.

Large portions of the circuitry were idle at any one step.

Example: the instruction decoding circuitry would be idle during execution.

Pipelining must add logic to handle dependencies between simultaneously executing instructions.

It also requires nonarchitectural pipeline registers.

Pipelined processors must access instructions and data in the same cycle.

They generally use separate instruction and data caches for this purpose.

Improvements in pipelining and caching are the two major microarchitectural advances that have enabled processor performance to keep pace with the circuit technology on which they are based.

The total execution time for each individual instruction is **not** changed by pipelining.

Pipelining **doesn't speed up** instruction execution time.

But it **does speed up** program execution time (the time it takes to execute an entire program) by increasing the number of instructions finished per unit time.

Pipelining allow a processor to execute programs in a shorter amount of time even though each individual instruction still spends the same amount of time traveling through the CPU.

Pipelining makes more efficient use of the CPU's existing resources by putting all of its units to work simultaneously, thereby allowing it to do more total work each nanosecond.

Execution units are also essential to microarchitecture.

Execution units include

- arithmetic logic units (ALU)
- floating point units (FPU)
- load/store units
- branch prediction
- SIMD

The choice of the number of execution units, their latency and throughput is a central microarchitectural design task.

The size, latency, throughput and connectivity of memories within the system are also microarchitectural decisions.

Performance analysis

A particular processor architecture can have many microarchitectures with different cost and performance trade-offs.

The cost depends on the amount of digital building blocks.

There are many ways to measure the performance of a computer system.

Microprocessor makers often market their products based on the clock frequency and the number of cores.

But some processors accomplish more work than others in one clock cycle and this varies from program to program.

The only true way to measure performance is by measuring the execution time of your program.

The computer that executes your program fastest has the highest performance.

The next best choice is to measure the total execution time of a collection of programs that are similar to those you plan to run.

Such collections of programs are called **benchmarks**.

Popular benchmarks:

Whetstone (for floating point operations)

Dhrystone (for integer operations)

CoreMark

SPECint2006

The execution time of a program:

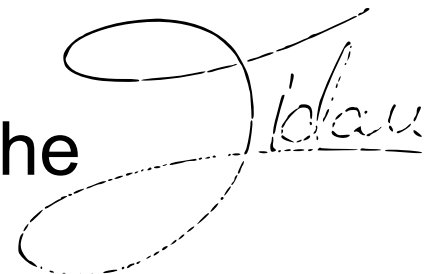
$$\text{Execution Time} = \left(\#instructions \right) \left(\frac{\text{cycles}}{\text{instruction}} \right) \left(\frac{\text{seconds}}{\text{cycle}} \right)$$

The number of instructions in a program depends on the processor architecture.

Some architectures have complicated instructions that do more work per instruction, thus reducing the number of instructions in a program.

However, these complicated instructions are often slower to execute in hardware.

The number of instructions also depends on the cleverness of the programmer.



The **cycles per instruction** (CPI) is the number of clock cycles required to execute an average instruction.

It is the reciprocal of the **throughput** (**instructions per cycle**, or IPC).

First, we assume we have an ideal memory that does not affect the CPI.

Later in the course, we examine how the processor sometimes has to wait for the memory, which increases the CPI.

The number of seconds per cycle is the clock period, T_c

The clock period is determined by the critical path through the logic on the processor.

Different microarchitectures have different clock periods.

Logic and circuit designs also significantly affect the clock period.

Example: a carry-lookahead adder is faster than a ripple-carry adder

Many other factors affect overall computer performance.

For example, the hard disk, the memory, the graphics system, and the network connection may be limiting factors.

Microarchitectural design pays closer attention not only to performance but to other constraints as well:

- power consumption

- logic complexity

- ease of connectivity

- manufacturability

- ease of debugging

- testability

Exercise: review the following topics from the Digital Logic Design course:

combinational circuits

sequential circuits

critical path 

short path 

propagation delay 

contamination delay 

setup time 

hold time 

setup time constraint 

hold time constraint 

clock skew 

latency 

throughput

spatial parallelism

temporal parallelism

logic families