# Seamless Transition

Computational processes smoothly execute on computing systems, seamlessly transitioning from one step to the next step.

The three key questions on this topic are:

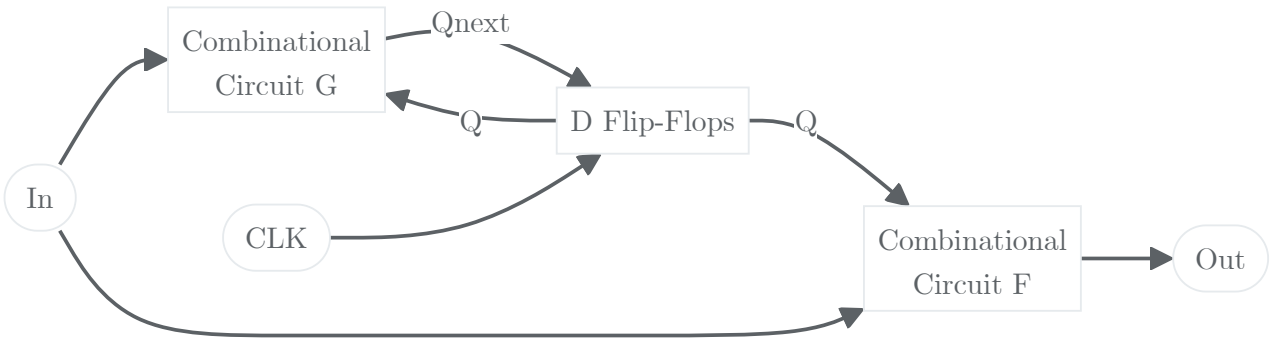
- How to identify the first instruction?
- How to ensure a single instruction's correct execution?
- How to find the next instruction and transition to it?

Computer systems capable of solving the above three problems are said to have the capability of *seamless transition*. It turns out that computer science has established four principles to support seamless transition

# Yang's Cycle Principle

A system executes a computational process in a sequence of *cycles*. The system finishes one cycle and automatically returns to the beginning (of the next cycle), so that different computational processes preserve their respective kinds.

How can it be done to automatically return to the beginning of the next cycle? Because, at the basic level, a computing system is a sequential circuit. It uses the current state $\mathbf{Q}$ to generate the next state $\mathbf{Q}_{\text{next}}$



For instance, recall the typical organization of sequential circuit as drawn above. At step $k$, the system is in state Q, which in the output of the D flip-flops. The system uses Q and the current input In to generate $\mathrm{Q}_{\mathrm{next}}$ through combinational circuit G, and to generate the current output through circuit F. This is the functionality of step $k$. When step $k$ finishes, $\mathrm{Q}_{\mathrm{next}}$ replaces Q to become the current sate via the D flip-flops, and the system returns to the beginning of step $k+1$

To generalize, working for cycle could be at different *granularities*

- A small granularity is **clock cycle**. For instance, a 1-GHz processor has the clock cycle of 1 ns. At this granularity, the multi-step computational process is a sequence of clock cycles. The system finishes one clock cycle and returns to the beginning of the next clock cycle.
- At the **instruction granularity**, a computer finishes a step by executing an instruction cycle. At this granularity, the multi-step computational process is a sequence of instruction cycles. The system finishes one instruction cycle and returns to the beginning of the next instruction cycle. Note that an instruction cycle is itself implemented by a sequence of clock cycles, to realize pipeline stages of Instruction Fetch, Instruction Decode, and Instruction Execution. Each stage of the instruction pipeline may need one or more clock cycles.
- At the **program granularity**, a computer finishes a step by executing a program cycle. At this granularity, the multi-step computational process is a sequence of program cycles. The system finishes one program cycle and returns to the beginning of the next program cycle. Note that a program cycle is itself implemented by a sequence of instruction cycles.

# Postel's Robustness Principle

This principle is often shortened to: **be tolerant of input and strict on output**. It has an important implication: accumulation of errors, drifts, and distortions can often be avoided.

# von Neumann's Exhaustiveness Principle

The *exhaustiveness principle* is due to John von Neumann. In the *First Draft of a Report on the EDVAC*, he stated right at the beginning of the document the following principle, when defining an automatic computing system (called the *device*) for problem-solving such as to solve a non-linear partial differential equation (called this *operation*).

> The instructions which govern this operation must be given to the device in absolutely exhaustive detail. They include all numerical information which is required to solve the problem under consideration . . . Once these instructions are given to the device, it must be able to carry them out completely and without any need for further intelligent human intervention.
>
> John von Neumann, 1945

Note that instructions in the above quote are not only binary instructions of program code, but all numerical information. Obviously, the computer must be given the input data and the processing program code. The computer must also be given information such as the library of functions, context information, etc.

For example, when the power is turned on in a computer with an x86 processor, the first instruction to execute is at memory address `0×FFFFFFF0`, and it contains a jump instruction such as `JUMP 000F0000`. Address `000F0000` contains the entry instruction for the BIOS code.

## Amdahl's Law

Amdahl's law was originally proposed by Gene Amdahl, a designer of the famous IBM S/360 general-purpose computer. The modern form of this law can be stated succinctly as follows: **After enhancing a portion of a system, the speedup obtained is upper bounded by the reciprocal of the other portion's time.**

More precisely, suppose a system's execution time is broken into two portions $1 - f$ and $f$, such that $1 - f > f$. Enhancement on the $1 - f$ portion can lead to a *speedup* no more than $1/f$.

Here, *speedup* is (time before enhancement) / (time after enhancement).