SFU SIMON FRASER UNIVERSITY
ENGAGING THE WORLD

ENSC254 – Clock and Pipeline

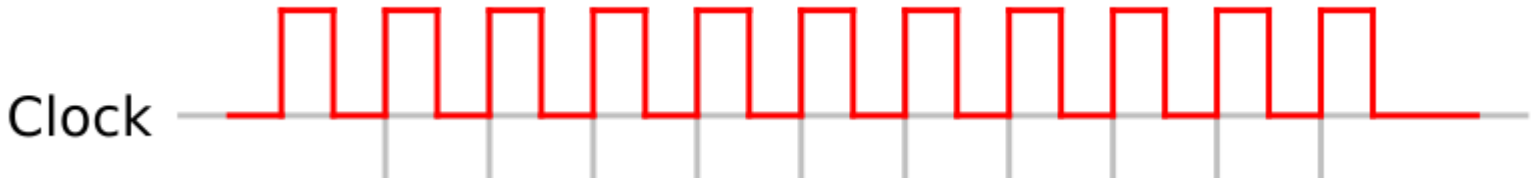
Ensc254

School of Engineering Science
Simon Fraser University
Burnaby, BC, Canada

Clock of an Integrated Circuit

- In a digital integrated circuit (IC) we define CLOCK a signal that oscillates between a high and a low state and ***is utilized like a metronome to coordinate actions of circuits***.
- A circuit regulated by a clock is called SYNCHRONOUS, as opposed to asynchronous circuits

The clock is a signal that does not carry much information, but synchronizes the behavior of different regions of a circuit



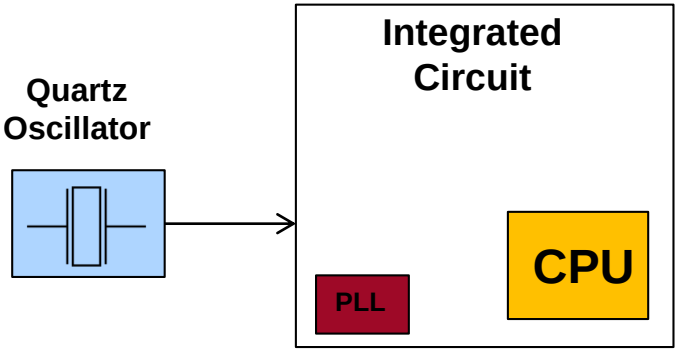
Clock Generation in ICs

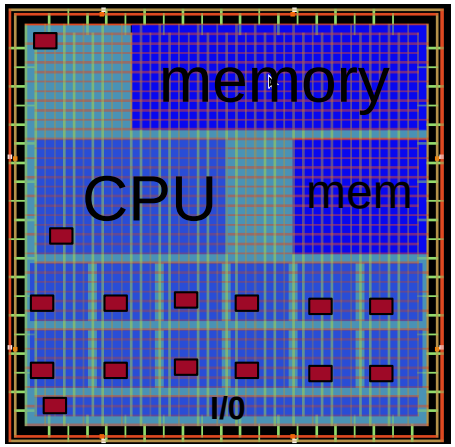
- In electronic systems, Clocks are typically generated OUTSIDE digital ICs, in specific physical devices that have the property of being ***resonant***: in given conditions, (e.g. when subject to an electric voltage) they regularly change state over time

- QUARTZ oscillators are very common: electronic oscillator circuits that use mechanical resonance of a vibrating crystal of piezoelectric material to create an electrical signal with a very precise frequency.

- Oscillators are used to
  - keep track of time
  - provide stable clock signal for digital integrated circuits
  - stabilize frequencies for radio transmitters and receivers.

Frequency Scaling: PLLs

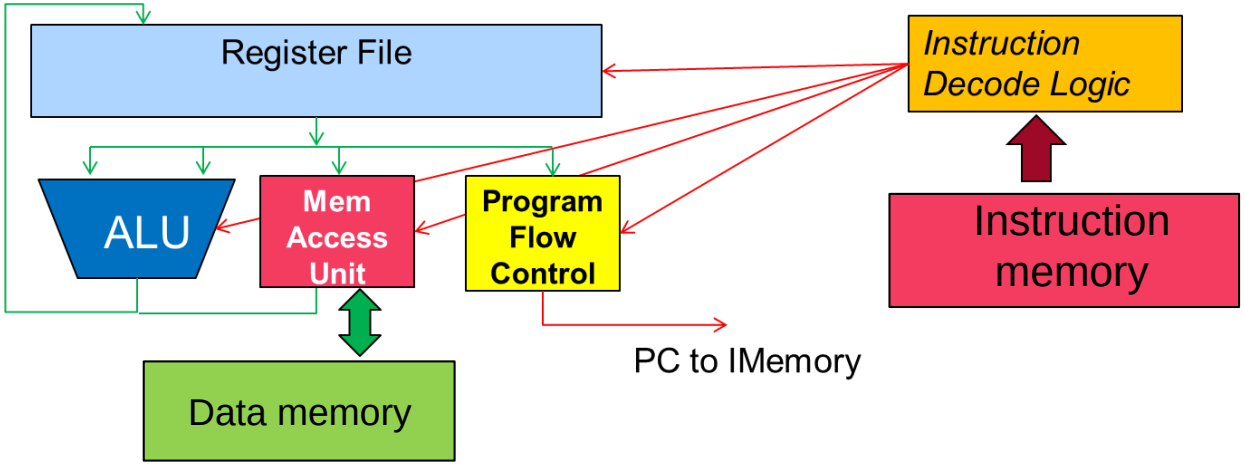




- Oscillators produce very stable fixed frequencies from few KHz to hundreds of megahertz.
- On large microprocessor chips, it may come handy to have higher frequency and *also scale (=change) dynamically the frequency depending on the workload*
- *This is implemented on-chip by an ANALOG clock controlling circuit called a **PLL (Phase Locked Loop)** that can multiply the frequency of the oscillator by a factor that can be chosen by the programmer (or the operating system) writing to the PLL as a peripheral*
- It is increasingly common to have several PLLs on the same chip that can dynamically generate different clocks starting from the same oscillator reference

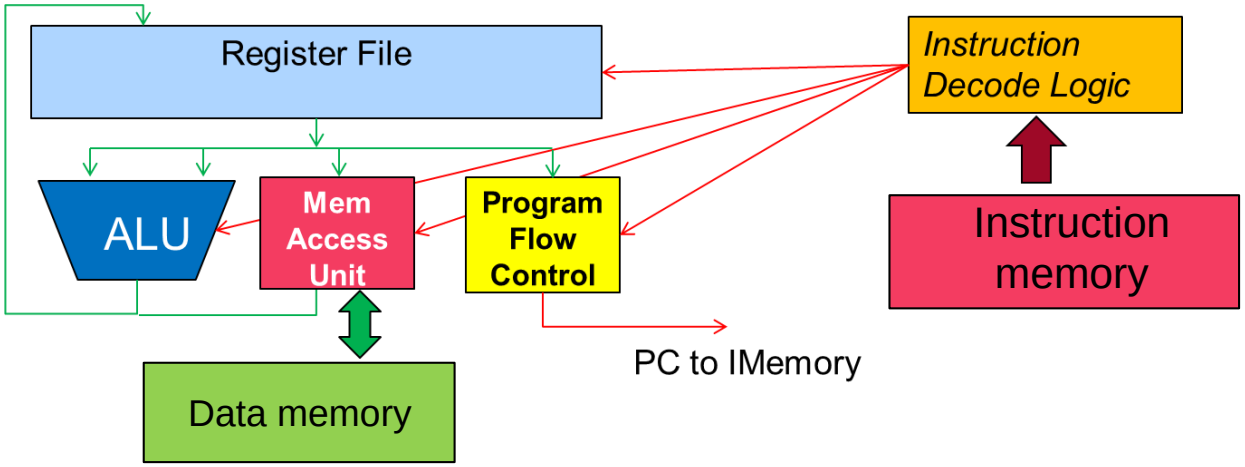
Quiz: What Parts of a Microprocessor Architecture are synchronous (that is, need to be synchronized by a clock)?

- Think carefully to a processor system: *which parts of it need to be synchronized by a clock?*



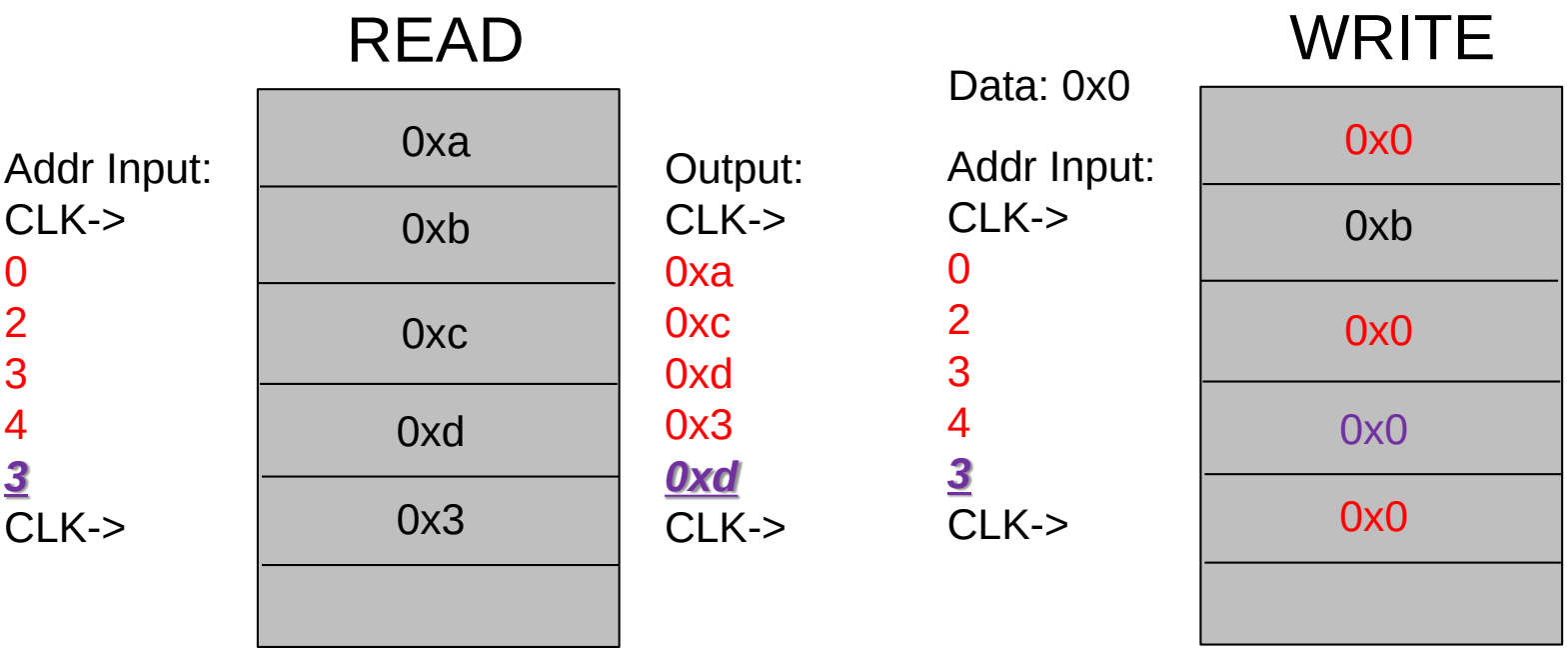
Quiz: What Parts of a Microprocessor Architecture are synchronous (that is, need to be synchronized by a clock?

- Think carefully to a processor system: *which parts of it need to be synchronized by a clock?*



- We need to synchronize ONLY IF we are
  - OVERWRITING a saved location (STORAGE AREA): the clock ensures that we are not writing an incomplete result!
  - Clocking a Write operation also ensures that the result of the previous operation has already been completed before reading it
- *So, of the blocks shown above, the only processor areas that need to receive a clock are the REGISTER FILE and writeable MEMORY blocks like DATA MEMORY.*

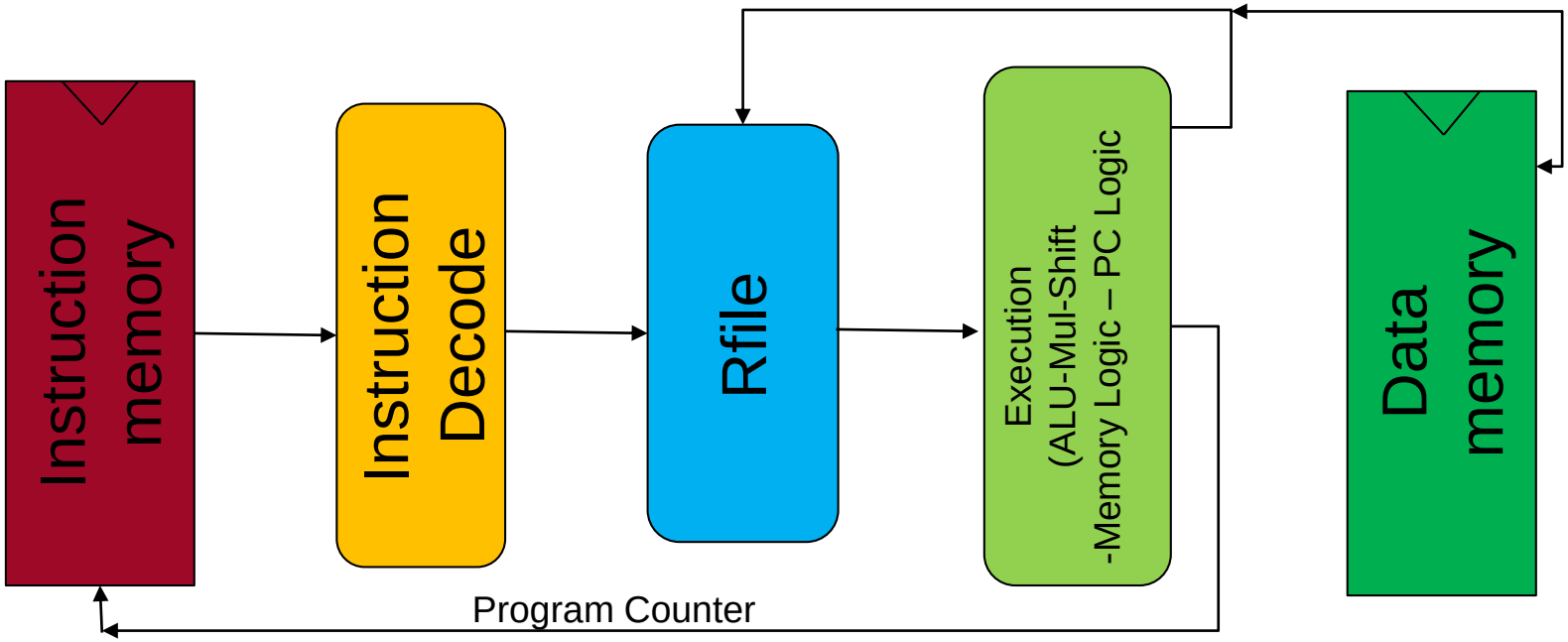
We need to clock only write operations, not read operations



- Please note that we need to clock only WRITE operations
- Read operations can have temporary fluctuations until the input stabilizes, eventually they will provide the right output
- ***Write operation can be performed only when all inputs are stable, or they will destroy existing data like red 0x0 shown in the right of the diagram***

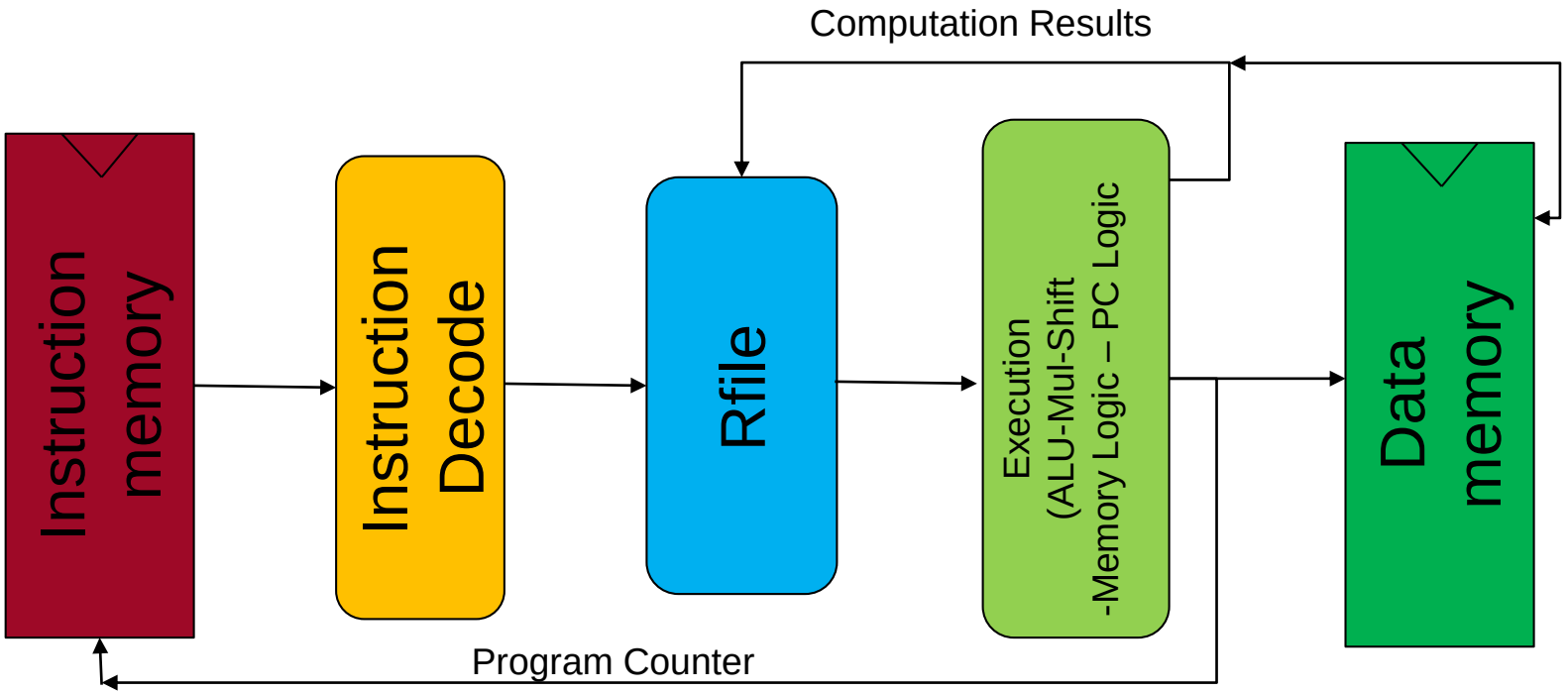
How Does a microprocessor clock work?

- Which of the below resources require a clock?
- Most every instruction changes the processor status (PC, Status Register, Register File, Memory). We need to ensure that
  1. *The operation is fully completed before we save its results*
  2. *We don't start the next instruction until the results of the first are completely saved*



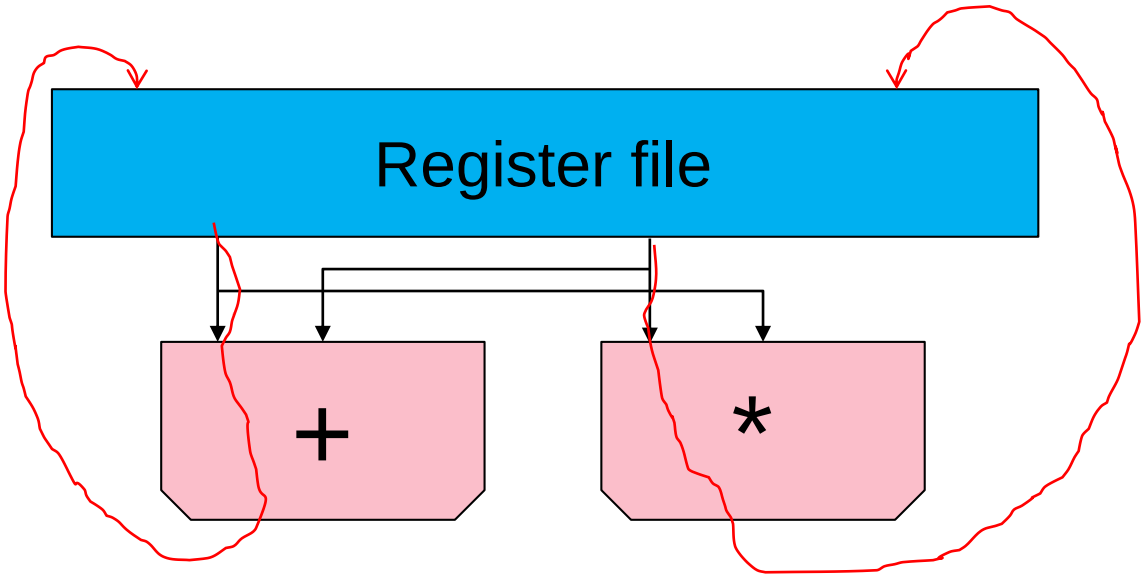
How Does a microprocessor clock work?

- The clock frequency of a microprocessor can be described as f=1/T
- *T=Period= T(read instr)+T(decode instr)+T(read rfile)+T(execution)+T(write results)*



What is the frequency of a clock?

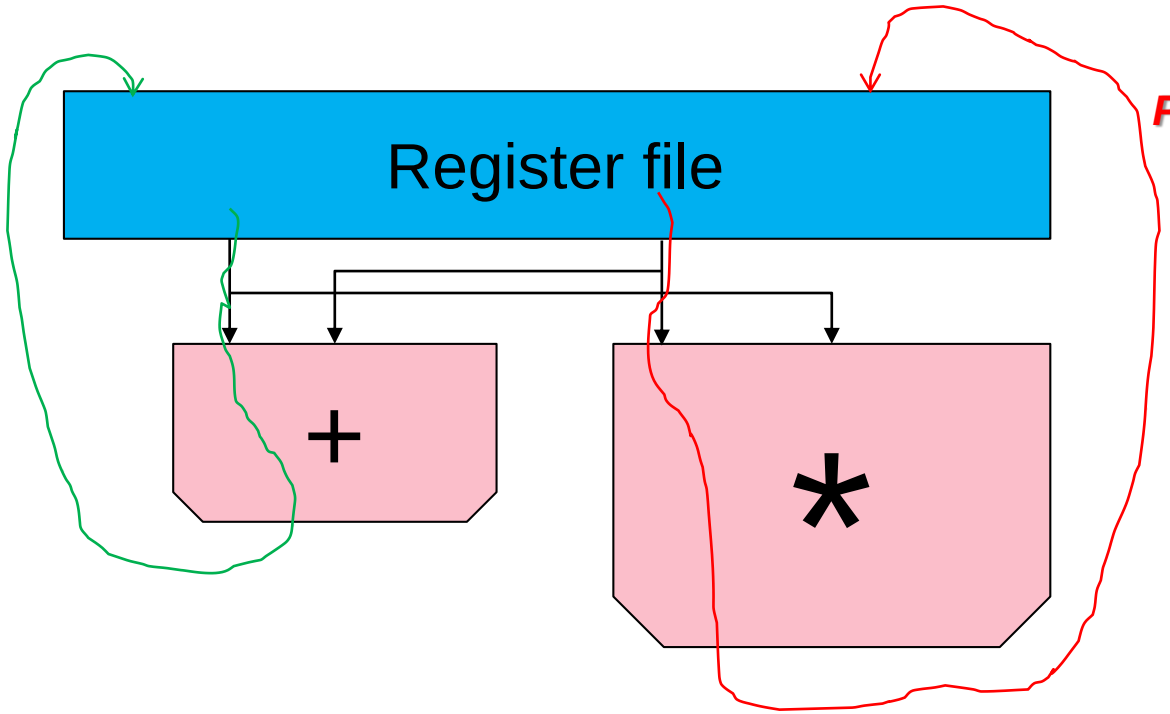
- A clock must synchronize the behavior of different regions of hardware, ensuring that every operation has concluded before starting new ones: suppose every other delay except computation is negligible …



What would be the frequency of the clock synchronizing the circuit above?

What is the frequency of a clock?

- A clock must synchronize the behavior of different regions of hardware, ensuring that every operation has concluded before starting new ones: ***So the clock PERIOD must be long enough for ALL operations to be concluded in time for the next operation to start!***



Path A=5ns

Path B=15ns

FREQUENCY = 1/Max_Path=1/15ns = 66 MHz

What is the frequency of the clock?

- It is impractical to change the frequency of the clock depending on the instruction that we are running, as the system needs a reliable synchronization mechanism to provide reliable results

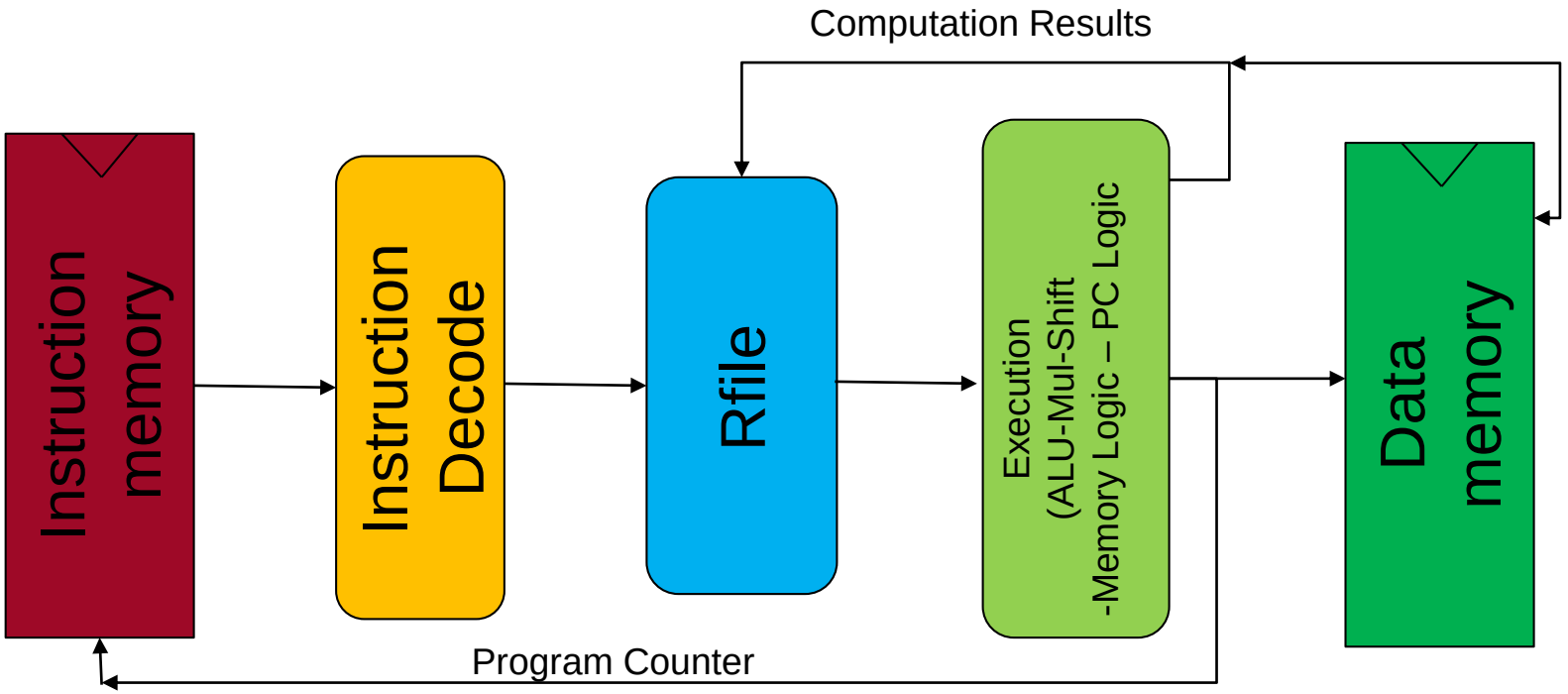
(There is a lot of literature on asynchronous CPUs, but they never really had commercial success!!)

- ***So the frequency of a digital circuit is always driven by its slowest component or, phrased better, the SLOWEST PATH***

- In the previous example, it would not make sense to optimize the ALU, as the longest delay is for the multiplier

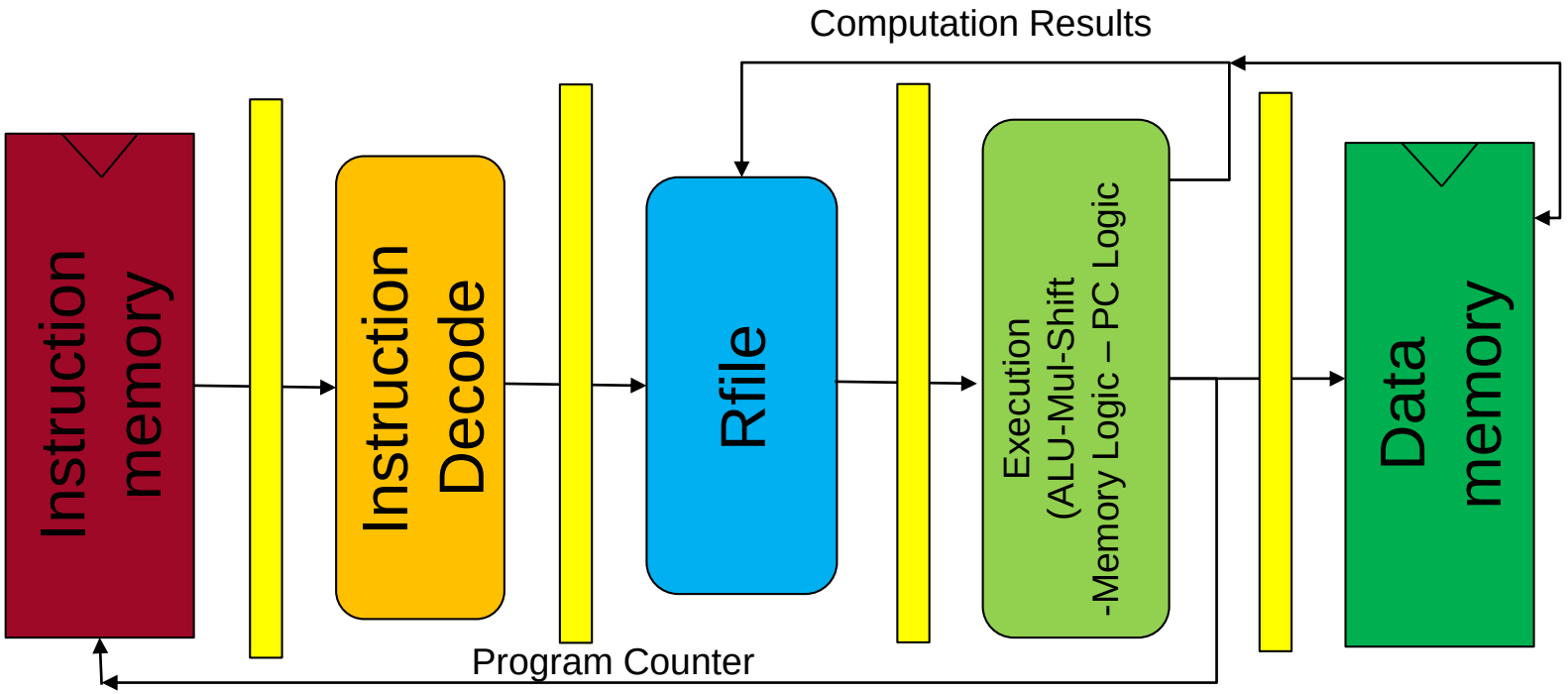
Pipelining

The standard CPU architecture described below is clearly very inefficient: the complete computation is strongly serialized and the electrical path needs to traverse a long list of ordered operations: *T=Period= T(read instr)+T(decode instr)+T(read rfile)+T(execution)+T(write results)*



Pipelining (2)

- A very appealing alternative would be to run every single one of the stages below concurrently, so that *T=Period= MAX(T(read instr), T(decode instr), T(read rfile), T(execution), T(write results)*
- *This is done by dividing instruction processing into STAGES and starting instruction 2 when instruction 1 has completed only the first "stage"*



Pipelining (3)

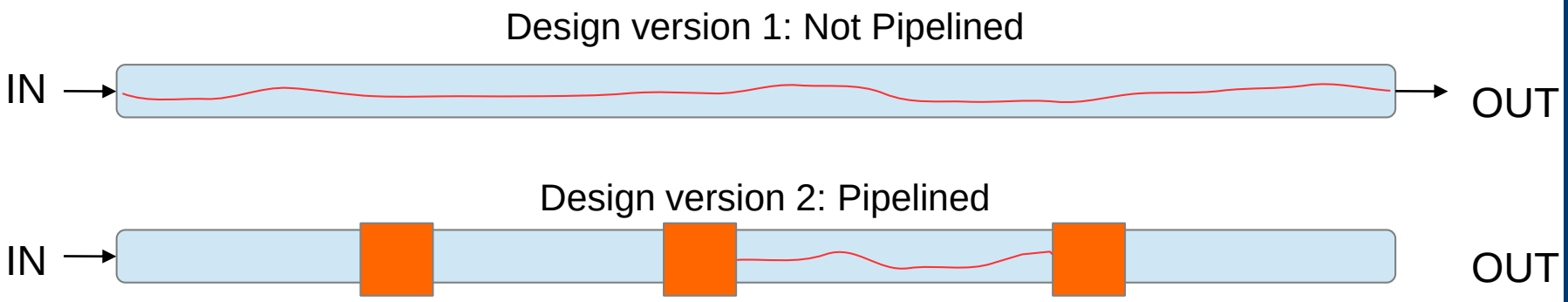
	Inst Fetch	Instr Decode and read RFILE	Execution (ALU/MUL/ Shift/ PC / Memory Address	Memory Read or Write	Write on RFILE
Cycle 0	Instr 1				
Cycle 1	Instr 2	Instr 1			
Cycle 2	Instr 3	Instr 2	Instr 1		
Cycle 3	Instr 4	Instr 3	Instr 2	Instr 1	
Cycle 4		Instr 4	Instr 3	Instr 2	Instr 1
Cycle 5			Instr 4	Instr 3	Instr 2
Cycle 6				Instr 4	Instr 3

Pipelining : DEFINITION

- **Pipelining** is a hardware *Design Practice*, normally applied to gain performance in a design
- Pipelining can be applied to any digital design, increasing speed at the price of higher chip area and power consumption
- The critical path of a pipelined design with N stages is the critical path of the worst stage:

  Critical Path(pipe) = Max(CriticalPath(stage1), CriticalPath(stage2), ..., CriticalPath(stageN))

- For this reason, to gain maximum speedup *pipelines should be BALANCED*: all stages should have comparable Max Delay. In the example below, the red line describes the critical path



Pipelining: Stalls

Pipelining works well if all blocks work independently: sometimes relations between successive instruction require to STALL the pipeline

Subs r2,r3
Bnz label

	Inst Fetch	Instr Decode and read RFILE	Execution (ALU/MUL /Shift/ PC / Memory Address	Memory Read or Write	Write on RFILE
Cycle 0	Subs r2,r3				
Cycle 1	Bne label	Subs r2,r3			
Cycle 2		Bne label (?)	Subs r2,r3		
Cycle 3		Bne label (?)		Subs r2,r3	
Cycle 4		Bne label (?)			Subs r2,r3
Cycle 5			Bne label		
Cycle 6				Bne label	
Cycle 7					Bne label

Pipelining: Stalls

If an instruction such as a MUL is part of a pipeline, it may require a significant delay, that we may not want to apply to every other instruction.

In this case we may use stalls to give the Mul multiple cycles to complete

Mla r1,r2,r3
Subs r4,1

	Inst Fetch	Instr Decode and read RFILE	Execution (ALU/MUL/ Shift/ PC / Memory Address	Memory Read or Write	Write on RFILE
Cycle 0	Mla r1,r2,r3				
Cycle 1	Subs r4,#1	Mla r1,r2,r3			
Cycle 2		Subs r4,#1	Mla r1,r2,r3		
Cycle 3			Mla r1,r2,r3		
Cycle 4			Mla r1,r2,r3		
Cycle 5			Subs r4,#1	Mla r1,r2,r3	
Cycle 6				Subs r4,#1	Mla r1,r2,r3
Cycle 7					Subs r4,#1

Pipeline stages in ARM7TDMI

This is the reason why MUL takes 3 cycles in the uVision profiling, and LDR take 2 cycles:

- The CPU reserves more cycles for these specific operations (introducing stalls in the system)

- *The ARM7TDMI we use in, for example, uVision has 3 pipeline stages, so can process up to three instructions at the same time.*

Some versions of ARM 11 can have 8 pipeline stages, INTEL PROCESSORS can have 12 or more.

- Please note that the number of stages is not strictly related to the ISA definition, so we may have different versions of the same processor family with different numbers of stages

Ex1: Non-Pipelined (Sequential) Computation

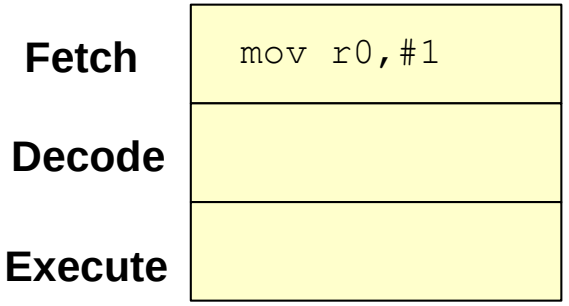
PIPELINE STAGES IN ARM7TDMI

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : RFILE, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
mov     r0,#1
cmp     r0,#17
```



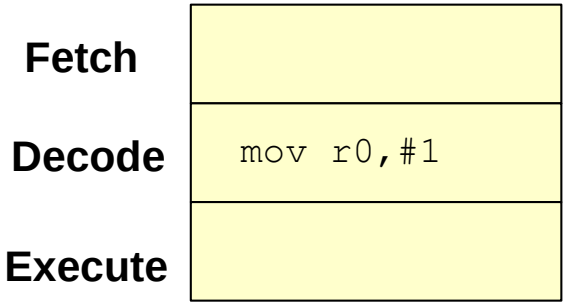
Ex1: Non-Pipelined (Sequential) Computation

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : Rfile, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
mov     r0,#1
cmp     r0,#17
```



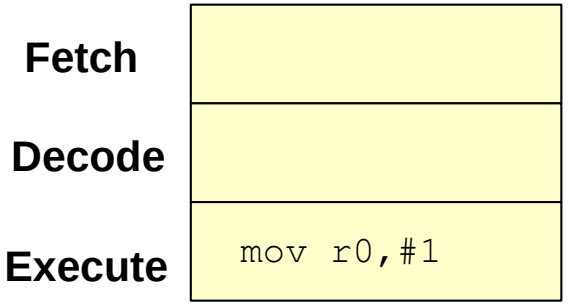
Ex1: Non-Pipelined (Sequential) Computation

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : Rfile, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
mov     r0,#1
cmp     r0,#17
```



The time required to complete the two instructions above would be 3+3 = 6 clock cycles

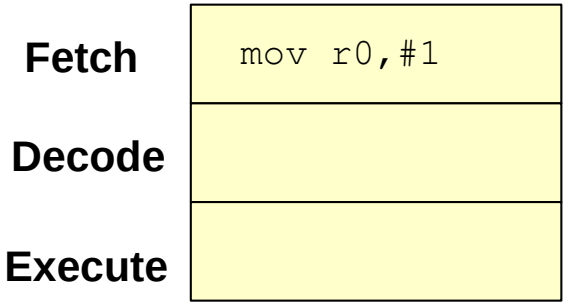
Ex2: Pipelined (Concurrent) Computation

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : Rfile, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
mov     r0,#1
cmp     r0,#17
```



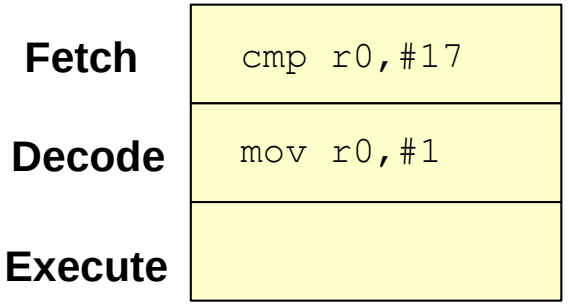
Ex2: Pipelined (Concurrent) Computation

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : Rfile, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
mov r0,#1
cmp     r0,#17
```



Why should we wait to complete the MOV before fetching the CMP? Since the hardware blocks performing Fetch and Decode are independent, we should start the second operation before completing the first

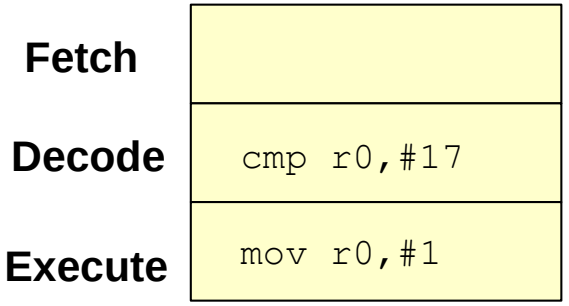
Ex2: Pipelined (Concurrent) Computation

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : Rfile, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
mov     r0,#1
cmp     r0,#17
```



The time required to complete these two instructions above would be 4 clock cycles in this particular case, and 2+N in the case of N similar instructions. Basically, we would have 1 instruction per cycle except for the first instruction

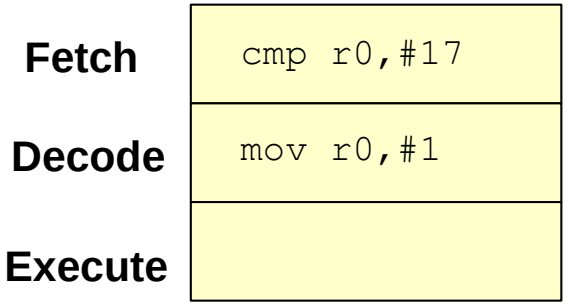
Impact of Pipelining on branches

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : Rfile, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
Loop
0x10 ….
[…….]
0x24 mov    r0,#01
0x28 cmp    r0,#17
0x30 Bne    loop
```



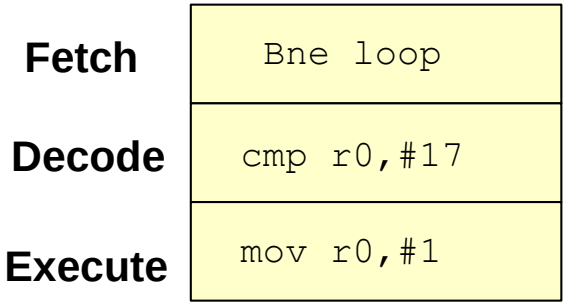
Impact of Pipelining on branches

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : Rfile, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
Loop
0x10 …….
[…….]
0x20 mov    r0,#01
0x24 cmp    r0,#17
0x28 Bne    loop
0x2c Trash1
0x30 Trash2
0x34 Trash3
```



Note: *At this stage we are just fetching the BNE from memory, we don't even know yet that it will be a branch. We execute our mov, so according to the ARM rules PC=PC+4. We will start fetching code* ***that is after our loop, so we really don't know what that is!!!!!***

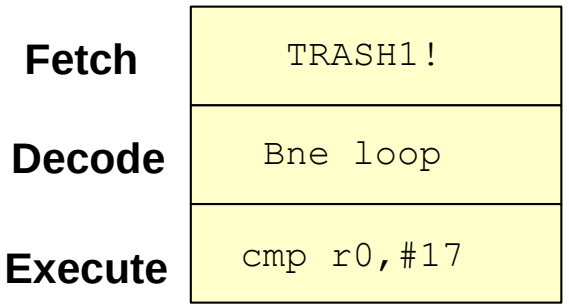
Impact of Pipelining on branches

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : Rfile, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
Loop
0x10 …….
[…….]
0x20 mov    r0,#01
0x24 cmp    r0,#17
0x28 Bne    loop
0x2c Trash1
0x30 Trash2
0x34 Trash3
```



Note: *We start realizing that BNE is a branch, but we still don't know WHERE to jump, as that will only be calculated during the Execute Stage!*

Executing CMP, we can only set PC=PC+4 and keep on loading trash!

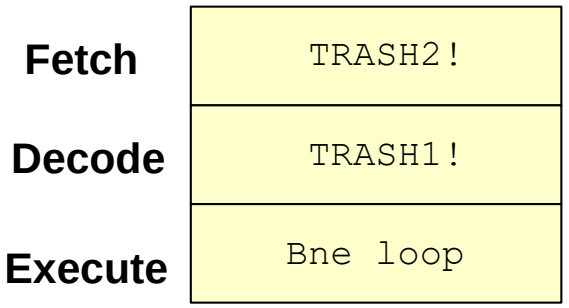
Impact of Pipelining on branches

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : Rfile, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
Loop
0x10 …….
[……]
0x20 mov    r0,#01
0x24 cmp    r0,#17
0x28 Bne    loop
0x2c Trash1
0x30 Trash2
0x34 Trash3
```



Note: *Only now we can execute the Branch and calculate our target address that is 0x10 . At the next Fetch stage we will have the correct instruction*

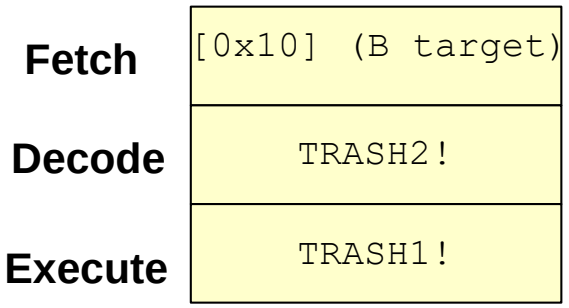
Impact of Pipelining on branches

FETCH : An instruction is loaded from the Instruction memory

DECODE : A specific logic loads the instruction, and understands its meaning giving appropriate commands to RFILE, ALU, SHIFTER MEMORY, etc

EXECUTE : Rfile, ALU, SHIFTER, MEMORY communicate data between themselves performing the required operation

```
Loop
0x10 …….
[……..]
0x20 mov    r0,#01
0x24 cmp    r0,#17
0x28 Bne    loop
0x2c Trash1
0x30 Trash2
0x34 Trash3
```



As you see, ANY TAKEN BRANCH WILL CAUSE OUR COMPUTATION TO MISS TWO CLOCKS! This is why in the simulator you will see a time count 3 times larger in case of branches. ***And that is Why when optimizing code we should avoid branches whenever reasonable!***