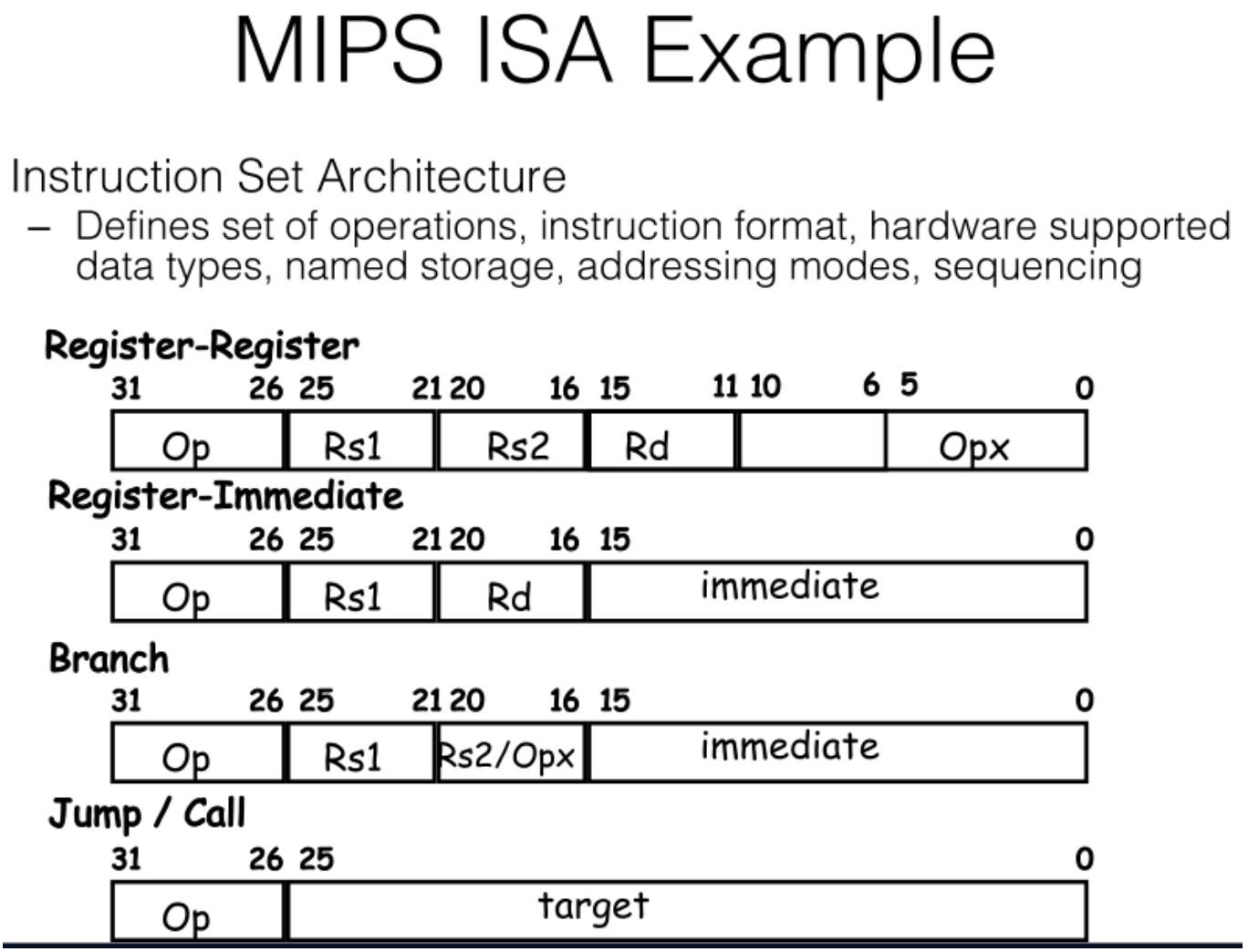
02.MIPS Pipelining

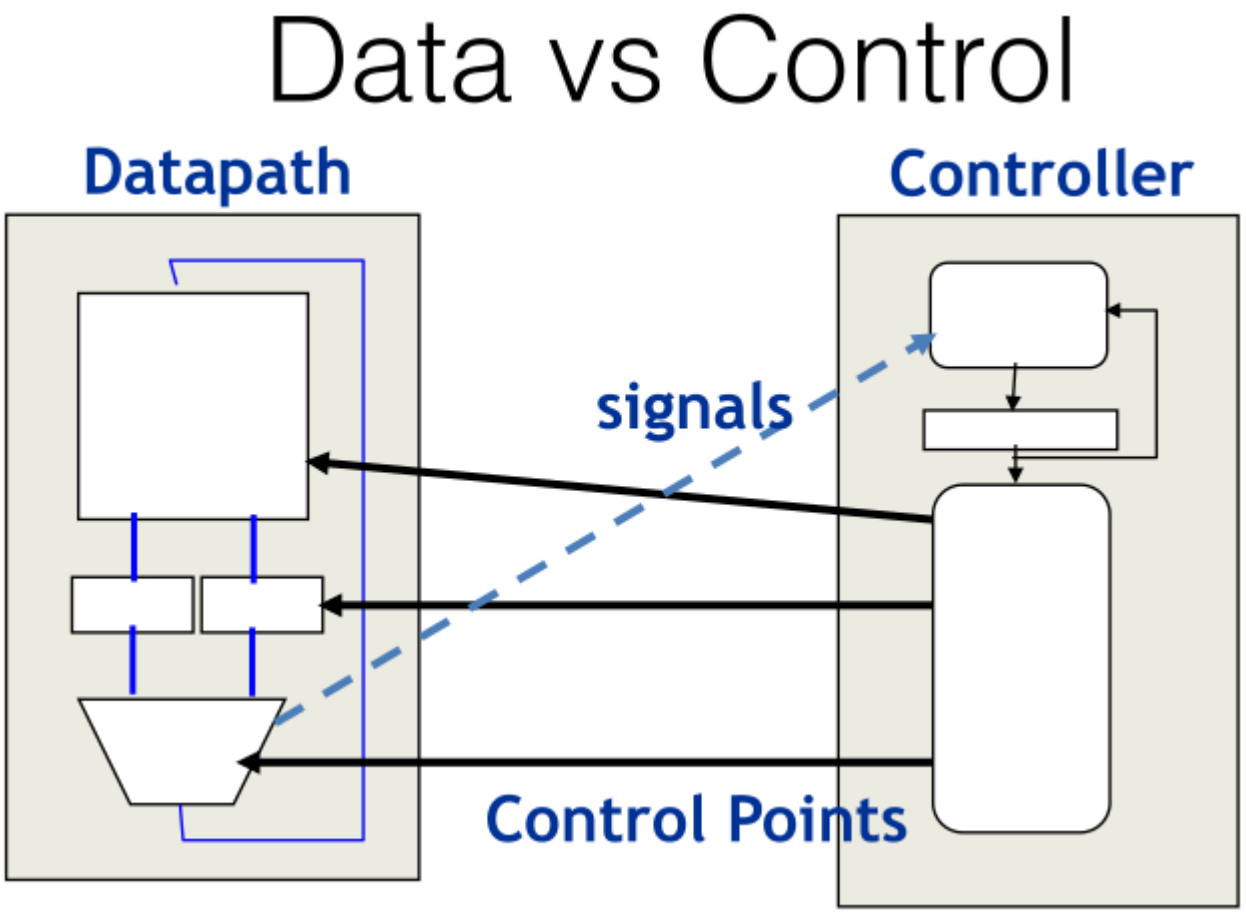
MIPS architecture is a processor architecture based on RISC(Reduced Instruction Set Computer) architecture. Based on the concept of executing only simple instruction in a reduced basic cycle to optimize the performance of CISC CPUs.
Don't need to know all software to be executed as the architecture will translate all instruction received to a language it can comprehend. Very simple instruction cycle(LOAD and Store architecture).
Example: RISC needs only 3 instruction(Load, Add, Store) instead CISC requires all the combination of situation for an addition. Having simpler instructions requires having more instruction but they are simple and faster to execute(Modern architectures are RISC).
The idea of Pipeline Architecture comes from having the idea of overlapping instruction as when executing an instruction you can start to load the next instruction to reduce the time for the execution.



- Immediate: constant value
- Op: operation
- Rs: result

- Opx: operation result
  First two classes are dedicated to manipulation of data the other two to manipulate the instructions.

Datapath

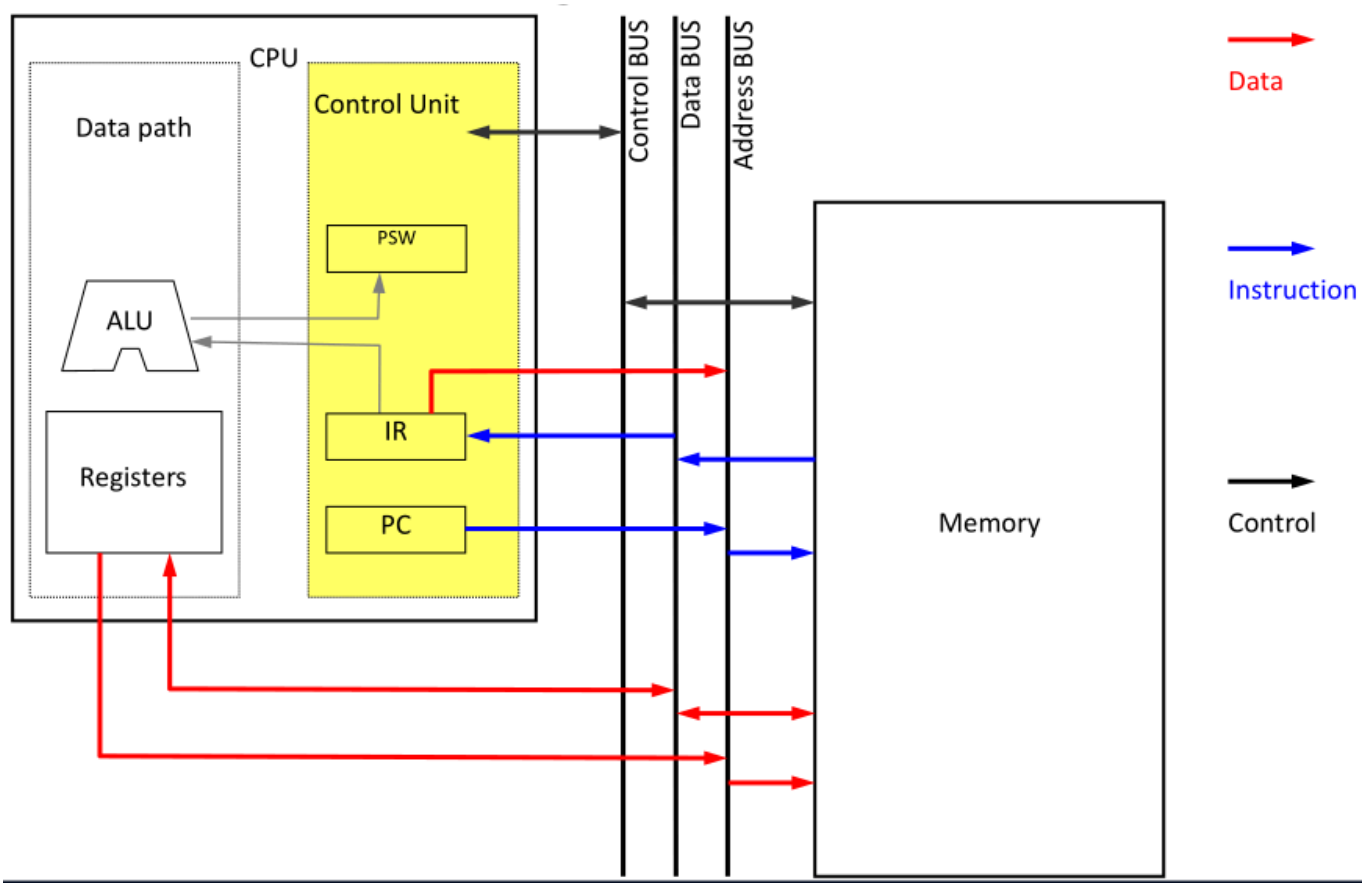


Inputs are signals provided to the CPU as they are commands to the CPU to move the data around.
Datapath: Storage, FU, interconnect sufficient to perform the desired functions

- Inputs are Control Points
- Outputs are signals
  Controller: State machine to orchestrate operation on the data path
- Based on desired function and signals

Memory

It is connected to the CPU with a series of bus to move data, instructions or controls from the memory to the CPU as for every instruction you have to provide an address and data.



This images explains how CPU and memory communicate and work together.

Programs

At HW level CPU doesn't have idea of what is a program, it only know how to compute instruction. At lower level there is the concept of clock cycle to synchronise the work. A program needs to be broken down in instruction to be executed and then it needs to be broken down to clock cycle at lower level. Decreasing CPU frequency decreases the clock cycle time of the CPU decreasing its velocity and capabilities.
You need to translate operations in a way that they don't interfere between them and how the CPU can start the various operation without having operation started before the data they need are ready/read from memory.

Execution of MIPS Instructions

ALU Instructions: op $x, $y, $z

Instr. Fetch & PC Increm.	Read of Source Regs. $y and $z	ALU OP ($y op $z)	Write Back of Destinat. Reg. $x

Load Instructions: lw $x, offset ($y)

Instr. Fetch & PC Increm.	Read of Base Reg. $y	ALU Op. ($y+offset)	Read Mem. M($y+offset)	Write Back of Destinat. Reg. $x

Store Instructions: sw $x, offset ($y)

Instr. Fetch & PC Increm.	Read of Base Reg. $y & Source $x	ALU Op. ($y+offset)	Write Mem. M($y+offset)

Conditional Branch: beq $x, $y, offset

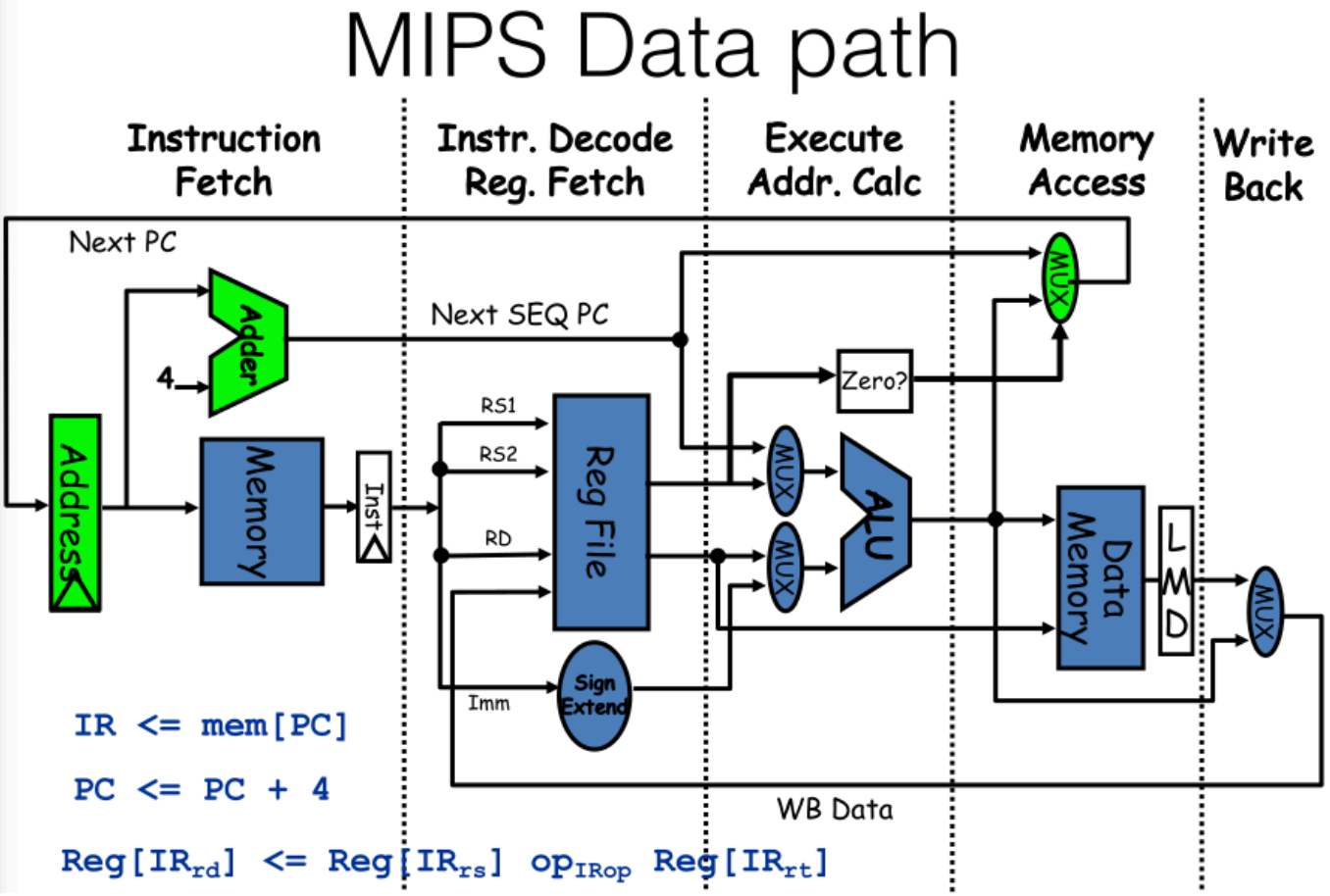
Instr. Fetch & PC Increm.	Read of Source Regs. $x and $y	ALU Op. ($x- $y) & (PC+4+offset)	Write PC

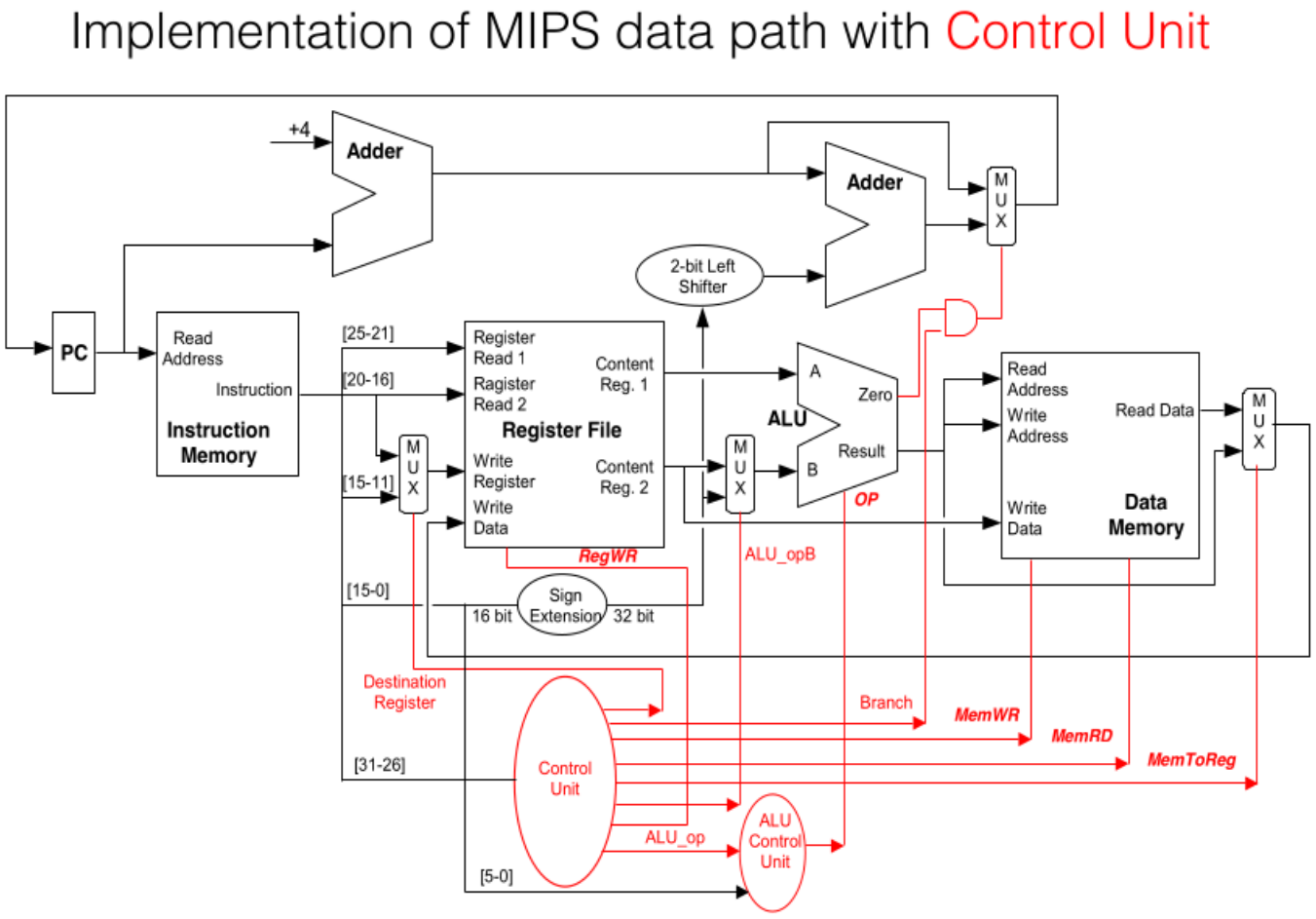
Every instruction in the MIPS subset can be implemented in at most 5 clock cycles as follows:

- Instruction Fetch Cycle:
  – Send the content of Program Counter register to Instruction Memory and fetch the current instruction from Instruction Memory.
  Update the PC to the next sequential address by adding 4 to the PC (since each instruction is 4 bytes(32 bit)).
- Instruction Decode and Register Read Cycle:
  – Decode the current instruction (fixed-field decoding) and read from the Register File of one or two registers corresponding to the registers specified in the instruction fields.
  – Sign-extension of the offset field of the instruction in case it is needed.
- Execution Cycle:
  The ALU operates on the operands prepared in the previous cycle depending on the instruction type:
  – Register-Register ALU Instructions:
  • ALU executes the specified operation on the operands read from the RF
  – Register-Immediate ALU Instructions:
  • ALU executes the specified operation on the first operand read from the RF and the sign-extended immediate operand
  – Memory Reference:
  • ALU adds the base register and the offset to calculate the effective address.
  – Conditional branches:

• Compare the two registers read from RF and compute the possible branch target address by adding the sign-extended offset to the incremented PC.

- Memory Access (ME)
  Result of operation aren't immediately inserted in the memory but they have to execute the load instruction
- Write-Back Cycle (WB)
  Value is written back in the register
  Not all the stages are required for every instruction so these steps are a guideline to complete all the instructions but they can be not needed for an instruction.

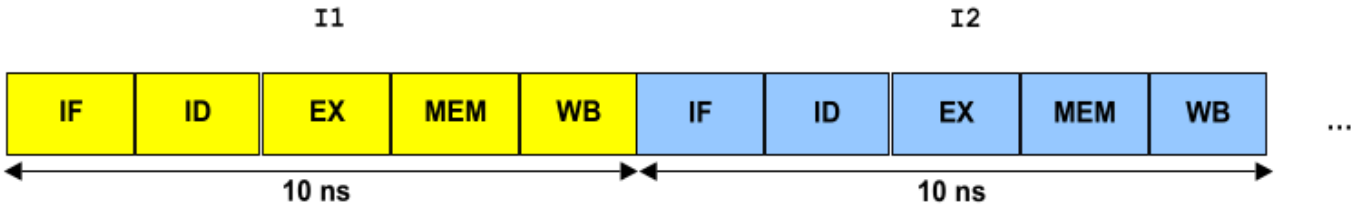


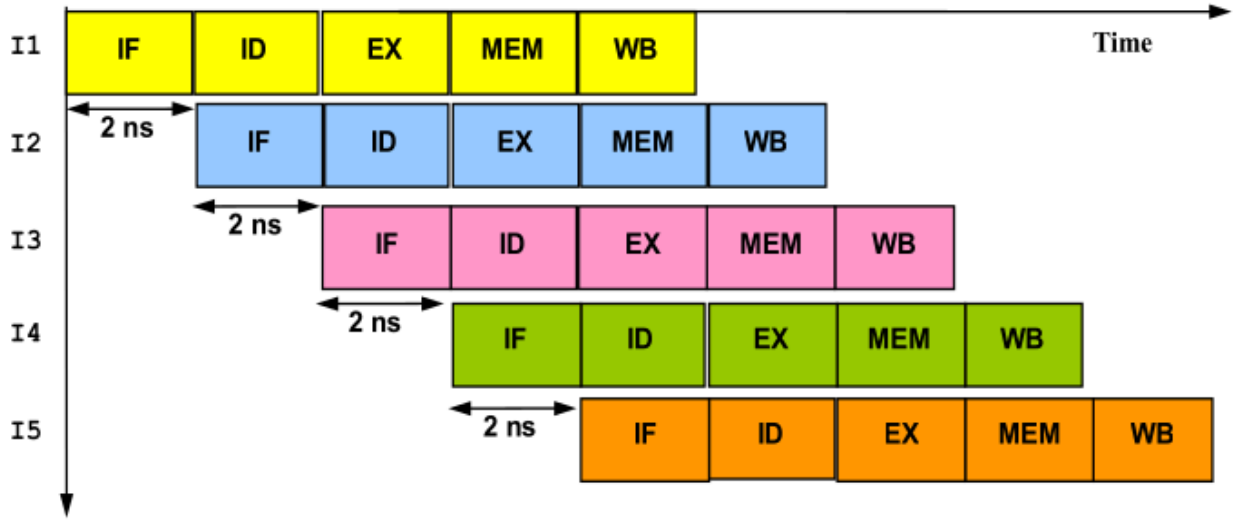


Some instruction have a grater latency as they have to fetch data from memory, but in any case the speed of a CPU is based on the speed of the slowest instruction. You can decompose the micro-architecture identifying smaller tokens(previous five steps) enabling the CPU to running faster(using a faster clock).

Pipelining

In each clock cycle we can determine what is the instruction to be executed.
We have the possibility to overlap instruction as if we have a stage we know is idling we can instruct it to do something speeding up our operation process speed. We are moving from a sequential execution were for 2 instruction we need 10 cycle but now we can use only 6 cycle as we can start the fetch step of the next operation the next cycle.





We cannot have two operation in the same logical stage it is an ERROR

We want to balance the length for each stage to achieve the maximum speedup(given from the number of stages).

Pipelining doesn't change the latency of an instruction but helps to speedup the process.

Pipeline Execution of MIPS Instructions

IF Instruction Fetch	ID Instruction Decode	EX Execution	ME Memory Access	WB Write Back

ALU Instructions: `op $x,$y,$z`

Instr. Fetch & PC Increm.	Read of Source Regs. `$y` and `$z`	ALU Op. (`$y op $z`)		Write Back Destinat. Reg. `$x`

Load Instructions: `lw $x,offset($y)`

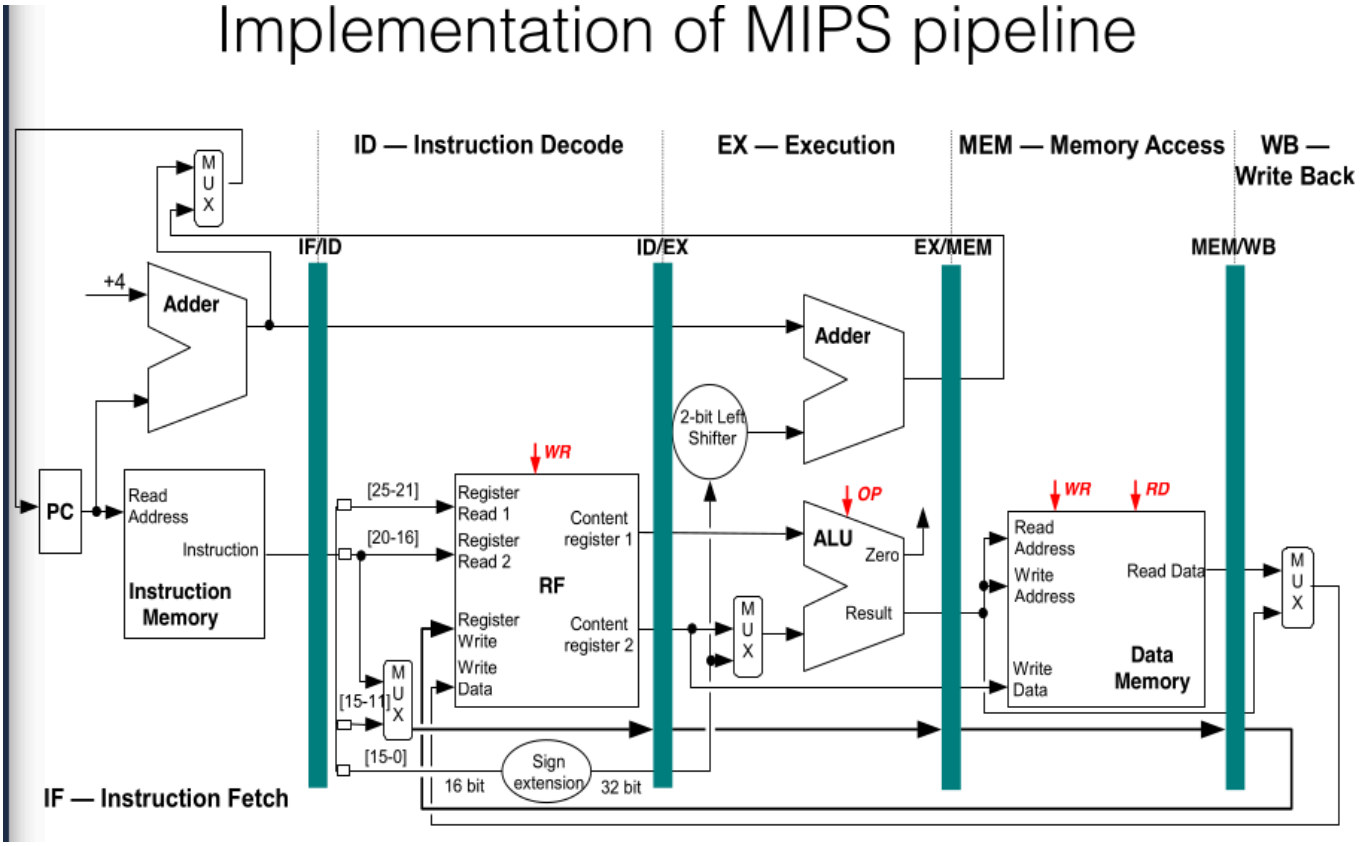
Instr. Fetch & PC Increm.	Read of Base Reg. `$y`	ALU Op. (`$y+offset`)	Read Mem. `M($y+offset)`	Write Back Destinat. Reg. `$x`

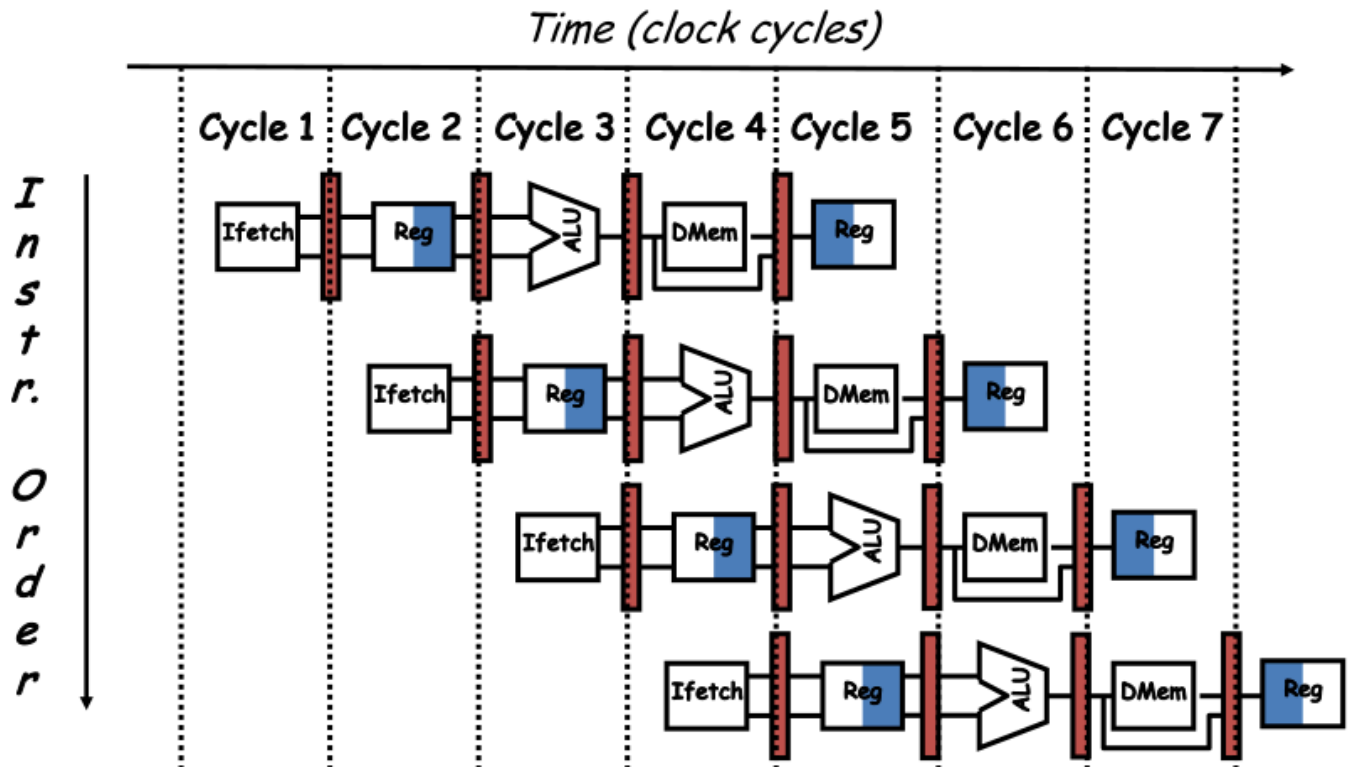
Store Instructions: `sw $x,offset($y)`

Instr. Fetch & PC Increm.	Read of Base Reg. `$y` & Source `$x`	ALU Op. (`$y+offset`)	Write Mem. `M($y+offset)`	

Conditional Branches: `beq $x,$y,offset`

Instr. Fetch & PC Increm.	Read of Source Regs. `$x` and `$y`	ALU Op. (`$x-$y`) & (`PC+4+offset`)	Write PC	





A possible issue is that in the register file you can have access to two different stages(ID and WB). Usually this is not a problem as usually you can write and read in two *different* registry. Problem arise when you read a register written by the WB. It is needed a stall(needs to read the new value). It s worse if the instruction is in the Execution step and not WB.
If you have to read and write on the same register you can use the clock cycle to your advantages reading the register in the second half of the cycle and write in the first half(Only if you are reading and writing in the same clock cycle). You HAVE to put a stall if the reading

happens before the execution of the previous instruction is not completed.
So the maximum speedup is basically impossible.

Hazard Classes

- Structural Hazards: attempt to use the same resource from different instructions simultaneously(Impossible in MIPS architecture)
- Data Hazards: attempt to use a result before it is ready. Can arise when instructions are too close. Possible solution is to insert two stalls. They can be a Read after Write or caused by a Dependence(Derives from an actual need for communication). Can be resolved with Compilation Techniques(insertion of nop instructions, instruction scheduling to avoid that correlating instructions are too close) or Hardware Techniques(insertion of bubbles or stall in the pipeline, data forwarding(Use temporary data stored in the pipeline registers instead of waiting the result of the WB. Needs to add multiplexers at the inputs of ALU to fetch inputs from pipeline registers to avoid the insertion of stalls in the pipeline) or bypassing). Other hazards as Write after Write(second instruction tries to write before the first has writes in the same register, WB of the second instruction as to be after the WB of the first instruction. Cannot happen in the MIPS pipeline as all registers write operations occur in the WB stage), Write after Read()
- Control Hazards: attempt to make a decision on the next instruction to execute before the condition is evaluated

Implementation of MIPS with Forwarding Unit