Designing RISC-V CPU from scratch – Part 1: Getting hold of the ISA

By chipmunk

Some Background Story

I have been thinking about designing a processor in RTL for a long time with the faintest idea of where to begin with. I even started with a simple ALU processor long ago i.e., one which is capable of doing some basic ALU operations. It was too crude given that it had no standardized ISA, (had to painfully create my own instruction set!) and the execution was sequential i.e., fetch, decode, and execute one instruction at a time. Then move to next instruction and so on. This is an example for a non-pipelined CPU architecture which has sequential operation. Hence, the throughput is very low. So, it was not much of an interesting project to explore and eventually the whole thing was dropped somewhere along the way…

Lately, I have been learning and exploring on ISAs, and RISC-V got most of my attention. The revolutionary ISA hit the market like tidal waves with relentless innovations backed up by countless learning and tool resources, and contributions from around the engineering community. And the ultimate beauty of RISC-V is being an open-source ISA. So, I thought: why not get on the bandwagon!?

Eventually, the idea converged from "*Designing a Processor*" to "*Designing a RISC-V Processor*". The pain of defining your own instruction set is gone now! But a little pain is good for learning. Hence, let's expand the idea to: "*Designing a Pipelined RISC-V Processor*".

Over the next few weeks, I will be demonstrating how to bring this idea to life through the new RISC-V CPU Development blog series which will be published here in multiple parts. We will go through defining specs, designing and refining architecture, identifying and solving challenges, developing RTL, implementing, and testing the CPU on simulation/FPGA board.

> ***Disclaimer****: All the idea to be presented in the blog series, are things I learnt on the way with my "baby steps" by exploring different RTL implementations, standards, documentations of proven RISC-V processors. The architectures presented in the blog are fundamental and the designed CPU is purely for education purpose. Industrial CPU architectures are way more complex beyond the scope of this blog.*

Step 1: Naming the CPU

Naming or branding your idea is mandatory to keep you motivated moving forward until you reach the goal! We are going to build a quite simplistic processor, so I came up with this fancy name "**Pequeno**", which means "tiny" in Spanish; the complete name: **Pequeno RISC-V CPU** aka **PQR5**. Through out the blog series, I will be using this name…

Step 2: Picking up the ISA

RISC-V has different flavors & extensions in the ISA. Let's go with the simplest one. **RV32I** aka 32-bit Base Integer ISA. The ISA is suitable to build a 32-bit CPU which supports integer arithmetic. So here comes the first spec of Pequeno:

> Pequeno is a 32-bit RISC-V CPU which supports RV32I ISA.

Step 3: Getting hold of the ISA

RV32I has 37 base instructions of 32-bit, which we plan to implement in Pequeno. So it's imperative that we should have deep understanding of each instruction. I went the hard way to get complete hold of the ISA. I learned the full spec and designed my own assembler, **pqr5asm**, in the process and validated it against some of the popular RISC-V assemblers.

RISBUJ

The above six letter word summarizes the types of instructions in RV32I. The 37 instructions fall under one of the categories:

- **R-type**: All integer computation instructions on registers.
- **I-type**: All integer computation instructions on registers and immediate values. Also includes JALR, Load instructions.
- **S-type**: All Store instructions.
- **B-type**: All Branch instructions.
- **U-type**: Special instructions like LUI, AUIPC.
- **J-type**: Jump instructions like JAL.

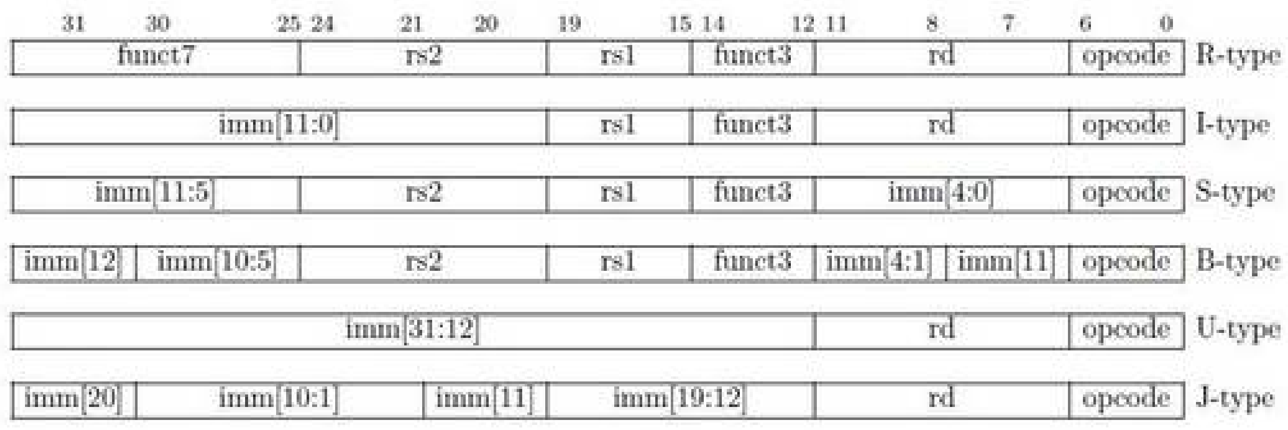
32 general purpose registers are available in RISC-V architectures, x0-x31. All registers are 32-bit. Among these 32 registers, *zero* aka x0 register is a useful special register hardwired to zero, it cannot be written, and is always read as zero. So what is the use of it? You can use x0 as dummy target to dump results you don't care to read, or use as operand zero, or generating NOP instructions to idle CPU.

This Integer computations instructions are ALU instructions performed on register(s) and/or 12-bit immediate values. Load/Store instructions perform storing/loading data between registers and data memory. Jump/Branch instructions are used to transfer the control of program to different location.

Detailed information on each instruction can be found in the RISC-V spec: RISC-V User-Level ISA v2.2
To learn the ISA, the RISC-V spec docs are enough. However, to clarify you can look into the implementation of different open cores in RTL.

Apart from 37 base instructions, I have also added 13 pseudo/custom instructions to pqr5asm and extended the ISA to 50 instructions. These instructions are derived from base instructions and are there to make assembly programmer's life easier… An e.g is: NOP instruction which is same as ADDI x0, x0, 0 which of course does NOTHING in CPU! But it is simpler and easier to interpret in the code.

The expectation before we start designing the processor architecture is the complete understanding of how each instruction is encoded in 32-bit binary and what is its functionality.


31 30 25 24 21 20 19 15 14 12 11 8 7 6 0
funct7 | rs2 | rs1 | funct3 | rd | opcode | R-type
imm[11:0] | rs1 | funct3 | rd | opcode | I-type
imm[11:5] | rs2 | rs1 | funct3 | imm[4:0] | opcode | S-type
imm[12] | imm[10:5] | rs2 | rs1 | funct3 | imm[4:1] | imm[11] | opcode | B-type
imm[31:12] | rd | opcode | U-type
imm[20] | imm[10:1] | imm[11] | imm[19:12] | rd | opcode | J-type

RISC-V Instruction – Encoding (ref: User-Level ISA v2.2)

PQR5ASM – Assembler for Pequeno

PQR5ASM, the RISC-V RV32I assembler which I developed in Python is available in my github here. You can write sample assembly code taking reference from Assembler Instruction Manual. Compile it, and see how it translates to 32-bit binary to cement/verify your understanding before moving on further…

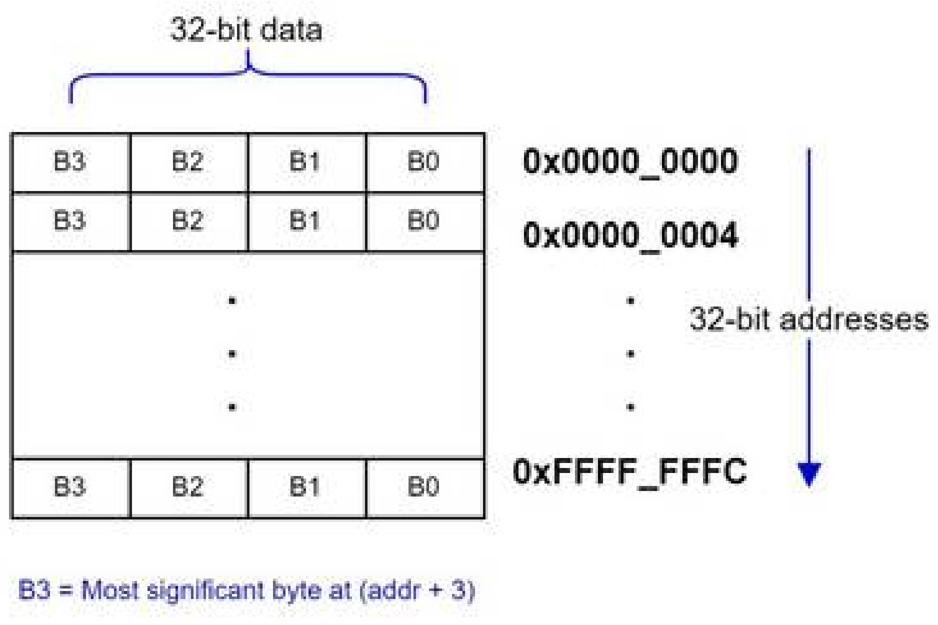
Pequeno RISC-V CPU – Specifications

- 32-bit CPU, *single-issue*.
- Classic five-stage RISC pipeline. Strictly *in-order* pipeline.
- Compliant to RV32I User-Level ISA v2.2. Supports all 37 base instructions + 13 custom instructions.
- Separate bus interface for Instruction & Data memory access. (*Why? To be discussed in future…*)

As said in the previous blog, we would be supporting RV32I ISA. So, the CPU supports only integer arithmetic.

All registers in the CPU are 32-bit. Address and data buses are also 32-bit. The CPU assumes the classic little endian byte-addressable memory space. Each address corresponds to a byte in the address space of the CPU. 0x00 - byte[7:0], 0x01 - byte[15:8] ...

32-bit word can be accessed at 32-bit aligned addresses i.e., addresses which are multiples of four: 0x00 - word0, 0x04 - word1 ...



Pequeno – Address Space

Pequeno is a single-issue CPU, i.e., only one instruction is fetched at a time from memory, and issued to be decoded and executed. Pipelined processors with single-issue can have max. IPC = 1 (or least/best CPI = 1) i.e., the ultimate goal is to execute at the rate of 1 instruction per clock cycle. This is theoretically the maximum performance achievable.

Classic five-stage RISC pipeline is the fundamental architecture to understand any other RISC architectures. This would make the ideal and simplest choice for our CPU. The architecture of Pequeno is built around this five-stage pipeline. Let's take a deeper dive into the underlying concepts.

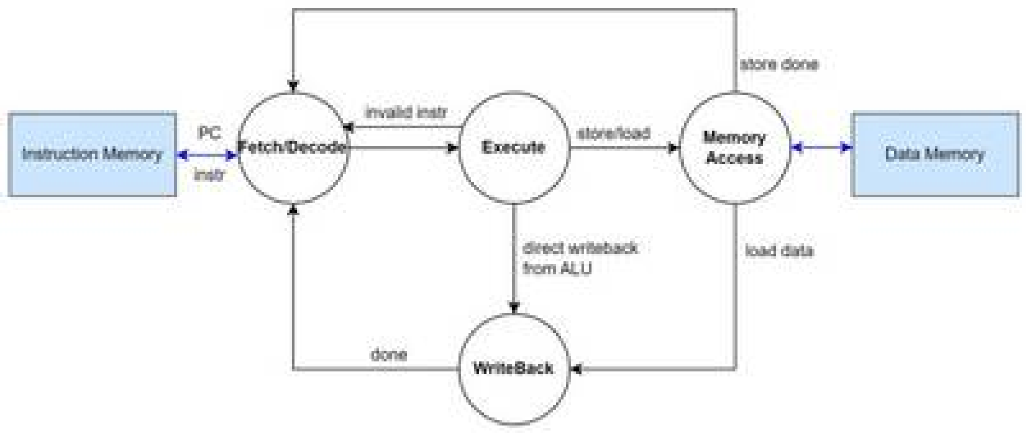
> *For simplicity, we will not be supporting timers, interrupts, and exceptions in the CPU pipeline. Hence, CSRs and privilege levels need not be implemented as well. RISC-V Privileged ISA is therefore not part of the current implementation of Pequeno.*

Non-pipelined CPU

Simplest approach to design a CPU is the non-pipelined way. Let's see couple of design approaches for a non-pipelined RISC CPU and understand its drawbacks.

Let's assume the classic sequence of steps followed by a CPU for instruction execution: **Fetch**, **Decode**, **Execute**, **Memory Access**, and **Writeback**.

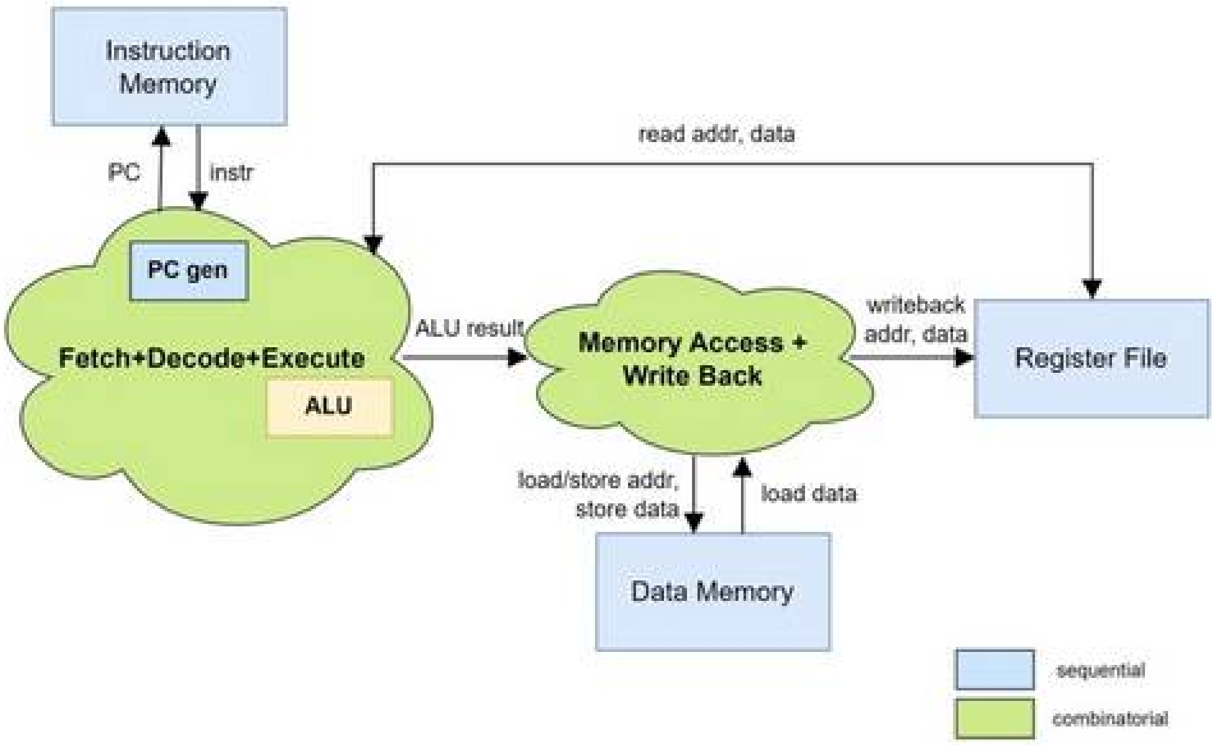
First design approach is: designing CPU like an FSM with four or five states which does every operation sequentially. For eg:

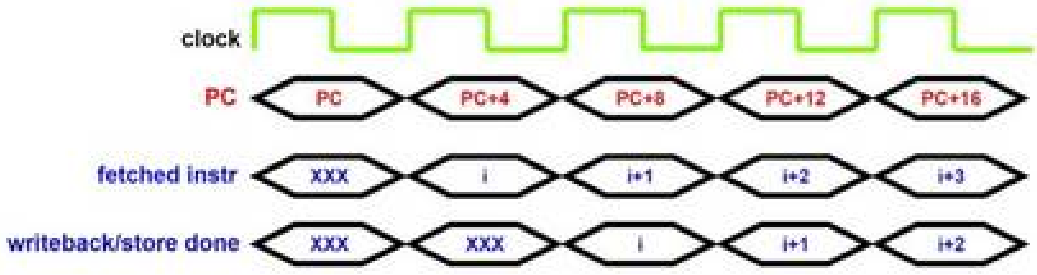


CPU like an FSM

But this architecture takes a bad hit on instruction execution rate. As it will take multiple clock cycles to complete the execution of a single instruction. Say, a register write would take 3 cycles. If load/store instruction, memory latency comes into picture as well. This is bad and primitive approach to design a CPU. Let's dump this for good!

Second approach is: instructions may be fetched from instruction memory, decoded, and executed by a fully combinatorial logic. The result from ALU is then written back to register file. This whole process up to writeback may be done in a single clock cycle. Such a CPU is called single-cycle CPU. If the instruction requires data memory access, read/write latency should be taken into account. If read/write latency is one clock cycle, store instructions may still finish execution in one clock cycle like all other instructions, but load instructions may take one clock cycle extra, as the loaded data has to be written back to register file. PC generation logic has to take care of the implications of this latency. If the data memory read interface is combinatorial (asynchronous read), the CPU becomes truly single-cycle for all instructions.



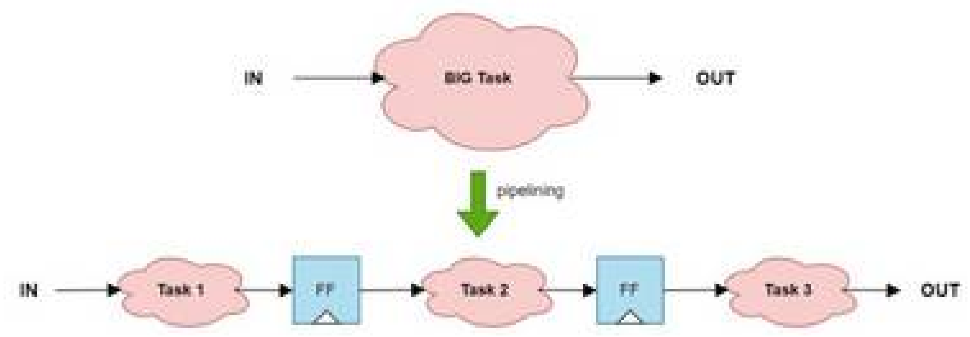


Single Cycle RISC-V CPU

Main drawback of the architecture is obviously the long critical path through the combinatorial logic from fetch to memory/register file write, which constraints the timing performance. However, this design approach is simple and suitable for CPUs in low-end microcontrollers where low clock speed, power consumption, and area is desirable.

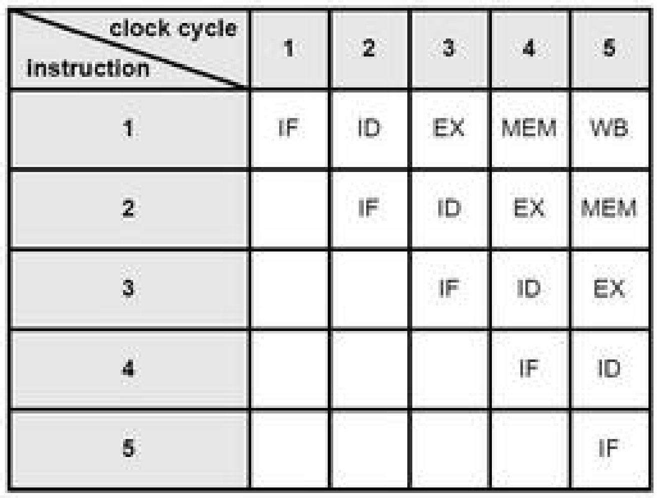
Pipelining CPU

To achieve higher clock speeds and performance, we can segregate the sequential processing of instructions by CPU. Each sub-process is then assigned to independent processing units. These processing units are cascaded sequentially to form a *Pipeline*. All units work in parallel and operate upon different parts of the instruction execution. Multiple instructions can be processed parallelly in this way. This technique to implement instruction-level parallelism is called Instruction Pipelining. This execution pipeline constitutes the core of a pipelined CPU.



Pipelining breaks the critical path by segregating the combo logic and adding registers in between

Classic five-stage RISC pipeline has five processing units aka *Pipeline Stages*. The stages are: **Fetch** (IF), **Decode** (ID), **Execute** (EX), **Memory Access** (MEM), **WriteBack** (WB). The working of the pipeline can be visualized as:



clock cycle / instruction	1	2	3	4	5
1	IF	ID	EX	MEM	WB
2		IF	ID	EX	MEM
3			IF	ID	EX
4				IF	ID
5					IF

Each clock cycle, different part of an instruction is processed, and each stage processes different instruction. If you observe closely, only at @5th cycle: instruction-1 finishes execution. This latency is called *Pipeline Latency*, Δ. This latency is same as the number of pipeline stages. After this latency, @6th cycle: instruction-2 finishes execution, @7th cycle: instruction-3, and so on…. We can theoretically compute the throughput (*Instructions Per Cycle*, **IPC**) as:

N instructions take $(\Delta+N-1)$ cycles to execute.

$\therefore IPC=(\Delta+N-1)N$

Theoretical max. IPC achievable is when $N\rightarrow\infty$

$\lim N\rightarrow\infty(\Delta+N-1)N=1$ instruction per cycle

Thus, pipelining CPU guarantees an execution rate of one instruction per clock cycle. This is the max. possible IPC in a single-issue processor.

By demarcating the critical path to multiple pipeline stages, CPU can now run in much higher clock speed as well. Mathematically, this boosts the throughput of pipelined CPU over an equivalent non-pipelined CPU by a factor, $S=(\Delta+N-1)N.\Delta=\Delta$, for $N\rightarrow\infty$.

This is called *Pipeline Speed-up*. In simpler words, a pipelined CPU with S stages can work at clock speed of S times compared to the non-pipelined one.

> *Pipelining normally increases area/power consumption but the performance gain is worth it.*

The mathematical computations assume that the pipeline never stalls, i.e., the data keeps moving forward from one stage to another every clock cycle. But in an actual CPU, the pipeline can stall due to multiple reasons, primarily due to *Structural / Control / Data Dependency*.

An example: register X cannot be read by Nth instruction because X is not written back yet by (N−1)th instruction which modified the value of X. This is an example for *Data Hazard* in pipelines.

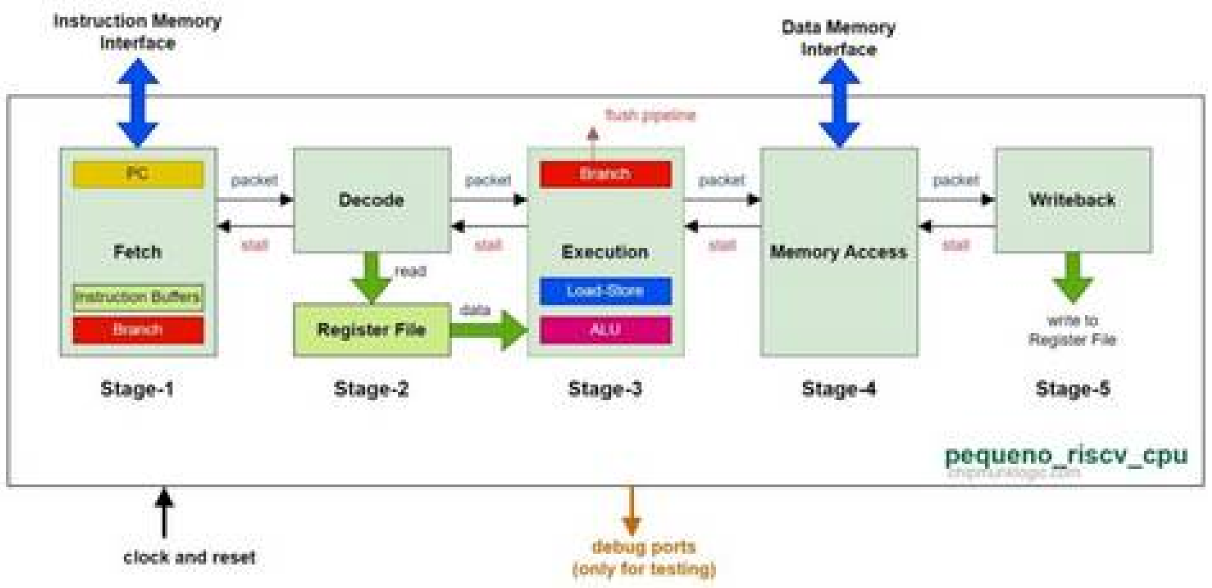
> *Pipeline Hazards are out of scope of at this point of time. We will discuss them in upcoming parts of the blog series.*

Pequeno RISC-V CPU – Architecture

Pequeno incorporates classic five-stage RISC pipeline in the architecture. We will implement strictly *in-order* pipeline. In *In-order Processors*, instructions are fetched, decoded, executed, and completed/committed in compiler-generated order. If one instruction stalls, the whole pipeline stalls.

In *Out-of-order Processors*, instructions are fetched and decoded in compiler-generated order, but execution can happen in different order. If one instruction stalls, it need not stall the subsequent instructions unless there is a dependency. Independent instructions may be allowed to pass forward. The execution may still be completed/committed in-order (that’s what happens in most CPUs today). This opens doors for implementing various architectural techniques to significantly improve the throughput and performance by cutting down clock cycles wasted on stalls and minimizing the insertion of bubbles (*What are “bubbles”!? Read on…*).

> *Out-of-order Processors are quite complex due to dynamic scheduling of instructions, but is now the de-facto pipeline architecture in today’s high-performance CPUs.*



Pequeno – CPU Architecture

The five pipeline stages are designed as independent units: **Fetch Unit** (FU)**, Decode Unit** (DU)**, Execution Unit** (EXU)**, Memory Access Unit** (MACCU)**,** and **WriteBack Unit** (WBU).

Fetch Unit (FU): Stage-1 of pipeline which interfaces with instruction memory. FU fetches instructions from the instruction memory and send to Decode Unit. FU may contain instruction buffers, initial branch logic.

Decode Unit (DU): Stage-2 of pipeline which decodes instructions from FU. Du also initiates read access on **Register File**. The packets from DU and Register File are retimed to be in sync and sent together to Execution Unit.

Execution Unit (EXU): Stage-3 of pipeline which validates and executes all decoded instructions from DU. Invalid/unsupported instructions are not allowed to move further in the pipeline. They become bubbles. **ALU** takes care of all integer arithmetic and logical instructions. **Branch Unit** takes care of jump/branch instructions. **Load-Store Unit** takes care of load/store instructions which require memory access.

Memory Access Unit (MACCU): Stage-4 of pipeline which interfaces with data memory. MACCU initiates all memory access as directed by EXU. Data memory is the addressing space which may constitute data RAM, memory-mapped IO peripherals, bridges, interconnects etc.

WriteBack Unit (WBU): Stage-5 or the final stage of pipeline. Instructions finish execution here. WBU is responsible for writing back results from EXU/MACCU (load-data) to Register File.

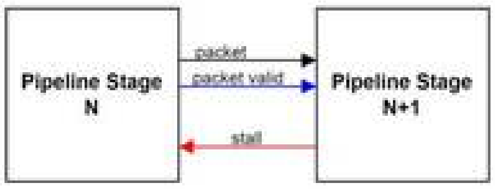
Interface between Pipeline Stages in the CPU

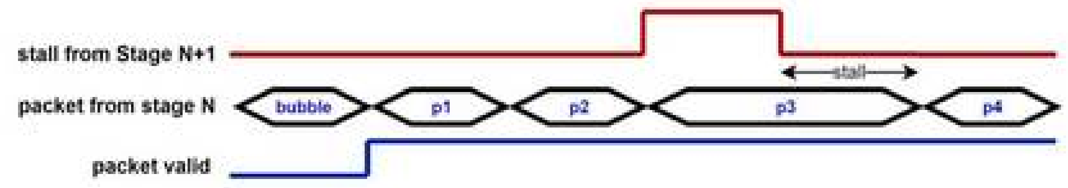
Between pipeline stages, *valid-ready handshaking* is implemented. This is not so obvious at first look. Each stage registers and sends a packet to the next stage. The packet may be instruction/control/data information to be used by next stage or by subsequent stages. The packet is validated by *valid* signal. If invalid packet, it is called a *Bubble* in the pipeline (read about *Pipeline Stalls and Bubbles* here). Bubble is nothing but "hole" in the pipeline which just moves forward through the pipeline doing nothing in effect. This is analogous to NOP instruction. But don't think they are of no use! We will see one of their uses when we discuss about *Pipeline Hazards* in upcoming parts. Following table defines a Bubble in Pequeno's instruction pipeline.

Instruction in the packet	**packet valid**	**Bubble in the pipeline?**
NOP	HIGH/LOW	YES
XXX	LOW	YES

Table: Defining Bubble in the Instruction Pipeline

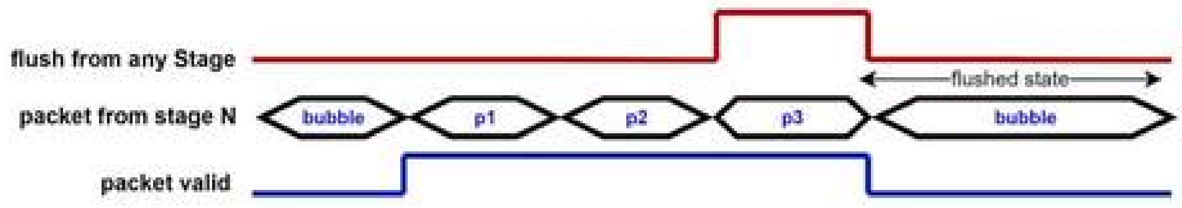
Each stage can also stall the previous stage by asserting *stall* signal. Once stalled, the stage will hold their packet until *stall* goes down. This signal is same as inverted *ready* signal. In in-order processors, stall generated at any stage acts like a global stall, as it eventually stalls the whole pipeline.





Handshaking between Pipeline Stages

The *flush* signal is used to flush the pipeline. Flushing will invalidate all packets registered by the previous stages in one go, because they are identified to be not useful anymore.

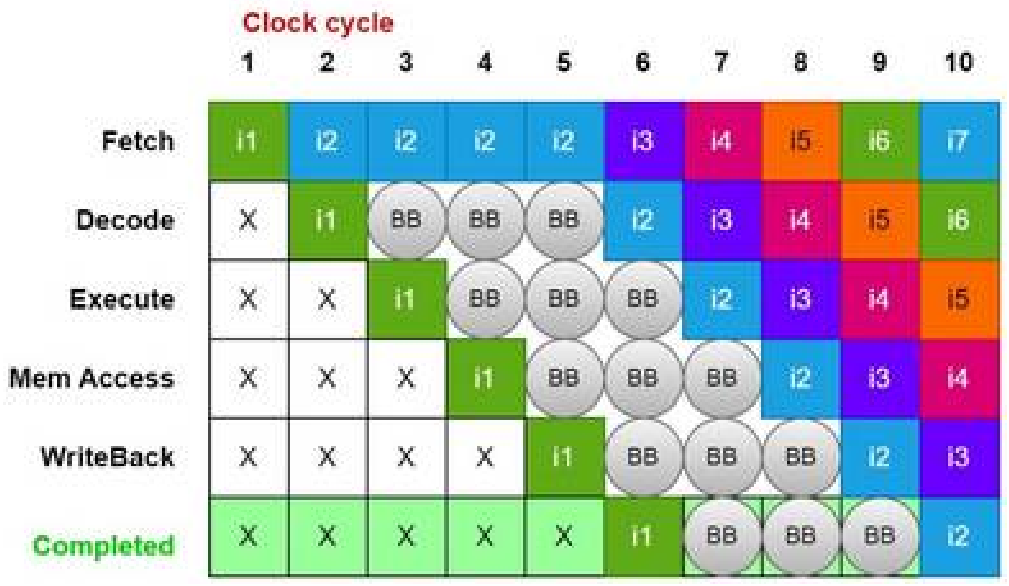


Pipeline Flush

An example is when the pipeline has fetched and decoded instructions from wrong branch after a jump/branch instruction and it is identified to be wrong only at the execution stage. Now the pipeline should be flushed and instruction has to be fetched from the correct branch!

Pipeline Hazards – Implications on Performance

Hazards in instruction pipeline of CPU are dependencies which disrupt the normal execution in the pipeline. When a hazard occurs, instruction cannot execute in the designated clock cycle as it may result in incorrect computation results or flow of control. As a result, the pipeline may be forced to stall until the instruction can be successfully executed.


Clock cycle
1
2
3
4
5
6
7
8
9
10
Fetch
Decode
Execute
Mem Access
WriteBack
Completed
i1 i2 i2 i2 i2 i3 i4 i5 i6 i7
X i1 BB BB BB i2 i3 i4 i5 i6
X X i1 BB BB BB i2 i3 i4 i5
X X X i1 BB BB BB i2 i3 i4
X X X X i1 BB BB BB i2 i3
X X X X X i1 BB BB BB i2

Pipeline Stall due to Data Dependency

In the above example, CPU performs in-order execution of instructions in the compiler-generated order. Let's assume instruction i2 has some dependency on i1, say some register has to be read by i2 , but this register is also being modified by the previous instruction i1. Hence, i2 has to wait until i1 writes back the result to register file, otherwise older data will be decoded and read from register file and used by Execute stage. To avoid this data incoherency, i2 is forced to stall by three clock cycles. *Bubbles* inserted in the pipeline represent the stall or wait states. Only when i1 is completed, i2 is decoded. Finally, i2 finishes execution in 10th clock cycle instead of 7th. Latency of three clock cycles was introduced due to the stall caused by data dependency. How does this latency affect the CPU performance?

We ideally expect our CPU to work at full throughput i.e., CPI = 1. But when pipeline stalls, it reduces the throughput/performance of the CPU because CPI increases. For non-ideal CPU:

Clock cycles Per Instruction=1+Stall cycles per instruction.
$\Longrightarrow$CPInon-ideal=CPIideal+Stall cycles per instruction

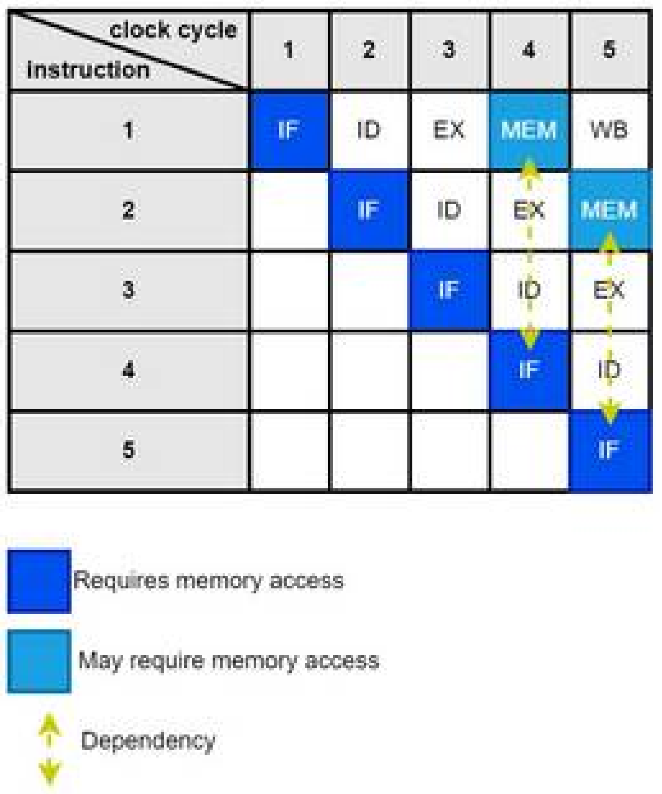
Pipeline Hazards – Types & Mitigation

There are different ways in which a hazard can occur in the pipeline. Pipeline hazards can be classified into three types:

- Structural Hazards
- Control Hazards
- Data Hazards

Structural Hazards

Structural Hazards occur due to hardware resource conflict. i.e., when two stages of the pipeline want to access the same resource. For eg: two instructions require access to memory in the same clock cycle.



clock cycle / instruction	1	2	3	4	5
1	IF	ID	EX	MEM	WB
2		IF	ID	EX	MEM
3			IF	ID	EX
4				IF	ID
5					IF

■ Requires memory access

▨ May require memory access

↕ Dependency

Structural Hazard due to Memory Access Conflict

In the above example, CPU has a single memory for both instruction and data. Fetch stage accesses the memory every clock cycle to fetch next instruction. Hence, instructions at Fetch stage and Memory Access stage may get conflicted if an earlier instruction at Memory Access stage also needs memory access. This will force the CPU to add stall cycles and Fetch stage has to wait until the resource (memory) is relinquished by the instruction in Memory Access stage.

Mitigating Structural Hazards

Some ways to mitigate structural hazards are:

- Stall the pipeline until the resource is available.
- Duplicate resources so that there will be no conflict at all.
- Pipeline resources so that two instructions will be at different pipeline stages of the resource.

Let's analyze different scenarios which may cause structural hazard in the pipeline of Pequeno and how to resolve it. We have no intention to use stalling as an option to mitigate the structural hazards!

Scenario	**Solution**
1) Memory access conflict b/w Fetch and Memory Access stages	✅ Separate instruction and data memory access paths. Use instruction and data caches with a single physical memory or physically separate instruction and data memories.
2) Register File access conflict b/w Decode and WriteBack stages	✅ Separate ports for read and write. Register File to be designed with two read ports and one write port. WB stage uses write port. ID stage uses read ports. Both of them may read/write to Register File in the same clock cycle without conflicts.
3) ALU conflict b/w different stages and	✅ Exclusive ALU in Execute stage. If any other stage requires to do operations like address computation (for eg: IF needs to do PC

Execute Stage	increment every cycle), use their own local ALU.

Table: Mitigating Mechanisms for Structural Hazards

In Pequeno’s architecture, we implement the above three solutions to mitigate all kinds of structural hazards.

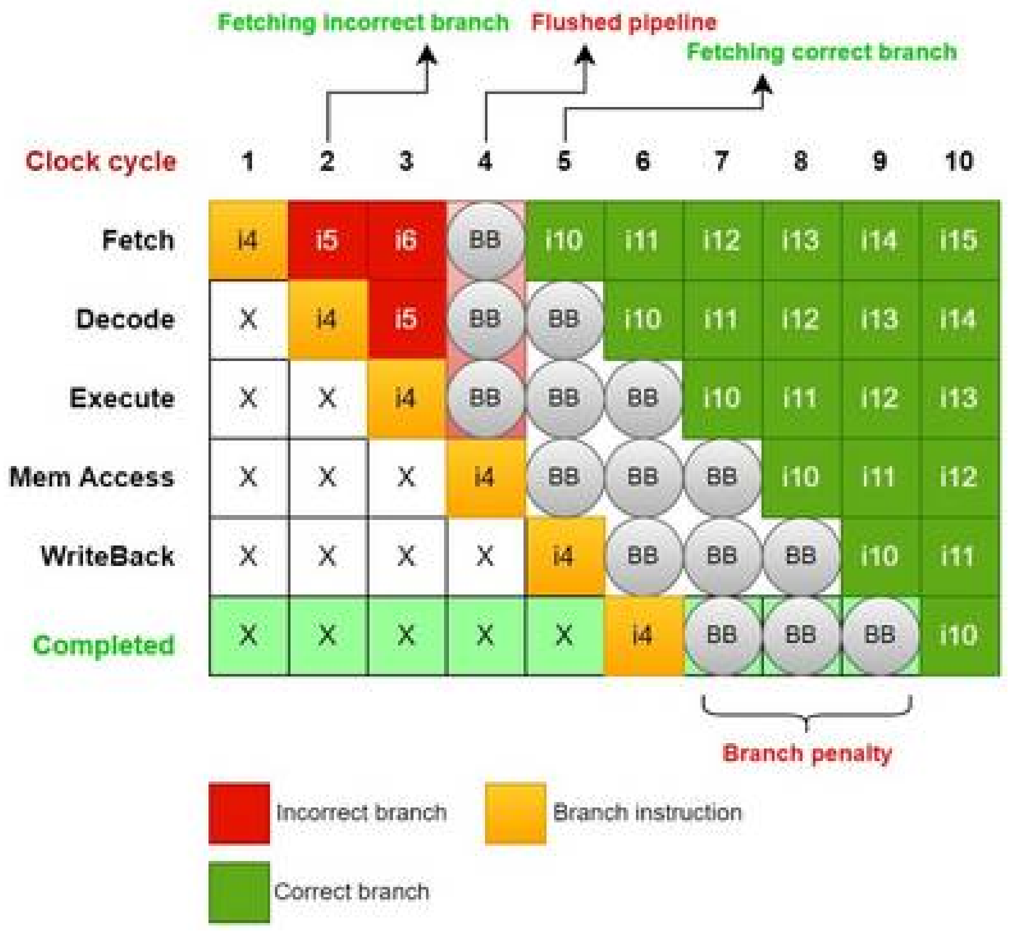
Control Hazards

Control Hazards are caused by Jump/Branch instructions. Jump/Branch instructions are the flow control instructions in the ISA of CPU. When the control reaches a Jump/Branch instruction, CPU has to make a decide whether to take the branch or not. One of the following actions should be taken.

- Fetch the next instruction at PC+4 (*branch not taken*) OR
- Fetch the instruction at the branch target address (*branch taken*).

Whether the decision was right or wrong is figured only at the Execute stage where the result of the branch instruction is computed. Depending on whether branch taken or not taken, *branch address* (address to which CPU should branch out) is resolved. If the decision made earlier was wrong, all the instructions in the pipeline fetched and decoded until that clock cycle should be discarded. Because those instructions should not be executed at all! This is done by flushing the pipeline and fetching the instruction at branch address in the next clock cycle. Flushing invalidate the instructions and convert them into NOPs or bubbles. This incurs significant amount of clock cycles as penalty. This is called *branch penalty*. Control hazards thus have the worst effect on the performance of CPU.

```
ASM program with potential control hazard
mvi x1, 0x100      # i1
mvi x2, 0x200      # i2
mvi x3, 0x000      # i3
bne x1, x2, EXIT   # i4: EXIT branch to be taken because x1 != x2
mvi x3, 0x001      # i5: This instruction shouldn't be executed!
...
EXIT:
...                # i10: This instruction should be executed after i4
```


Fetching incorrect branch
Flushed pipeline
Fetching correct branch
Clock cycle 1 2 3 4 5 6 7 8 9 10
Fetch i4 i5 i6 BB i10 i11 i12 i13 i14 i15
Decode X i4 i5 BB BB i10 i11 i12 i13 i14
Execute X X i4 BB BB BB i10 i11 i12 i13
Mem Access X X X i4 BB BB BB i10 i11 i12
WriteBack X X X X i4 BB BB BB i10 i11
Completed X X X X X i4 BB BB BB i10
Branch penalty
Incorrect branch
Branch instruction
Correct branch

Control Hazard and Flushing

In the above example, i10 completes the execution in 10th clock cycle, but it should have completed the execution in 7th clock cycle. A penalty of three clock cycles was incurred because wrong branch (i5) was taken. Flush had to be done in the pipeline when this was identified by Execute stage in 4th clock cycle. How this will impact the CPU performance?

If a program running in above CPU has 30% branch instructions, CPI will become:

$\text{CPI}=(\text{CPIideal}\times 0.70)+(3\times 0.30)\approx 2$

The CPU performance cuts down by 50%!

Mitigating Control Hazards

To mitigate control hazards, we can employ some strategies in the architecture…

- Simply stall the pipeline if the instruction is identified as branch. This decoding logic can be implemented in Fetch stage itself. Once the branch instruction is executed and branch address is resolved, next instruction can be fetched and the pipeline can resume.
- Add a dedicated branch logic in Fetch stage like Branch Prediction.

The essence of branch prediction is: we employ some kind of a prediction logic in Fetch stage that will guess whether the branch should be taken or not. In the next clock cycle, the guessed instruction is fetched. This will be fetched either from PC+4 (predicts *branch not taken*) or branch target address (predicts *branch taken*). Now there are two possibilities:

- If the prediction is found to be correct at Execute stage, do nothing, pipeline can continue processing.
- If the prediction is found to be wrong, flush the pipeline, fetch the correct instruction from the branch address resolved by Execute stage. This incurs branch penalty.

As you can see, branch prediction can still incur branch penalty if it makes incorrect prediction. Design goal should be to reduce the probability of making incorrect prediction. The CPU performance heavily depends on how “good” is the prediction algorithm. Complex techniques like Dynamic Branch Prediction keep the history of instructions to predict correctly at orders of 80−90% probability.

To mitigate control hazards in Pequeno, we will implement a simple branch prediction logic. More details to be revealed in the upcoming blog where we design Fetch Unit.

Scenario	Solution
1) Control Hazards due to Jump/Branch instructions	✅ Add a simple branch prediction logic in Fetch stage.

Table: Mitigating Mechanisms for Control Hazards

Data Hazards

Data Hazards occur when an instruction’s execution has data dependency on the results of some previous instruction that is still being processed in the pipeline. Let’s visit the three types of data hazards with examples to understand the concept better.

WAW (Write-After-Write)

Suppose an instruction i1 writes result to a register x. The next instruction i2 also writes result to the same register. Any subsequent instructions in the program order should be reading the result of i2 at x. Otherwise, data integrity is lost. This type of data dependency is called *output dependency* which may lead to WAW data hazards.

```
ASM program with potential WAW hazard
add x1, x2, x3  # i1; x1 = x2 + x3
add x1, x4, x5  # i2; x1 = x4 + x5
.
.
add x5, x1, x2  # This instruction should be reading the result of i2 @x1
```

WAR (Write-After-Read)

Suppose an instruction i1 reads a register x. The next instruction i2 writes result to the same register. Here, i1 should read the older value of x, not the result of i2. If i2 writes result to x before it is read by i1, it results in data hazard. This type of data dependency is called *anti-dependency* which may lead to WAR data hazards.

```
ASM program with potential WAR hazard
add x1, x2, x3  # i1; x1 = x2 + x3 -- should be reading the older value @x2
add x2, x4, x5  # i2; x2 = x4 + x5 -- new value written @x2
```

RAW (Read-After-Write)

Suppose an instruction i1 writes result to a register x. The next instruction i2 reads the same register. Here, i2 should read the value written by i1 to the register x, not the older value. This type of data dependency is called *true dependency* which may lead to RAW data hazards.

```
ASM program with potential RAW hazard
add x1, x2, x3  # i1; x1 = x2 + x3
add x5, x1, x4  # i2; x5 = x1 + x4 -- should be reading the value written by i1
@x1
```

This is the most common and predominant type of data hazard in pipelined CPUs.

Mitigating Data Hazards

To mitigate data hazards in in-order CPUs, we can employ a few techniques:

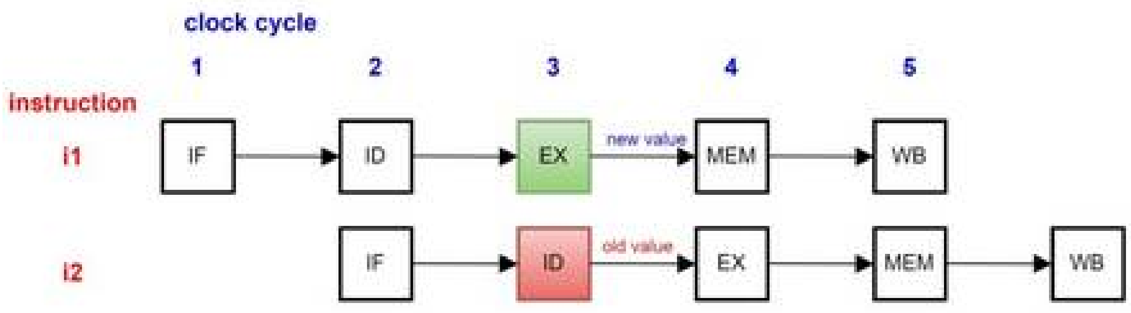
- Stall the pipeline on detecting data dependency (*refer to the first figure*). Decode stage can wait until execution the previous instruction is completed.
- *Compile Re-scheduling*: Compiler re-arranges the code to avoid data hazards by scheduling it for later. The intention here is to avoid stalling without affecting the integrity of flow of control in the program, but this may not be possible to do always. Compiler can also insert NOP instructions in between two instructions that have data dependency. But this will incur stalls, and hence is a performance hit.

```
Compiler Re-scheduling
add x1, x2, x3  # i1 being decoded
# Compiler inserts NOP or any other non-dependent instructions
# Compiler inserts 3 NOPs here because 3 stages ahead of Decode in the
pipeline...
NOP             # NOP being decoded; i1 in Execute stage
NOP             # NOP being decoded; i1 in Memory Access stage
NOP             # NOP being decoded; i1 in WriteBack stage
add x5, x1, x4  # i2 being decoded; by this time i1 would have completed
execution
                # and hence reads correct value @x1
```

- *Data/Operand Forwarding*: This is the prominent architectural solution in in-order CPUs to mitigate RAW data hazards. Let's analyze the CPU pipeline to understand the idea behind this technique.

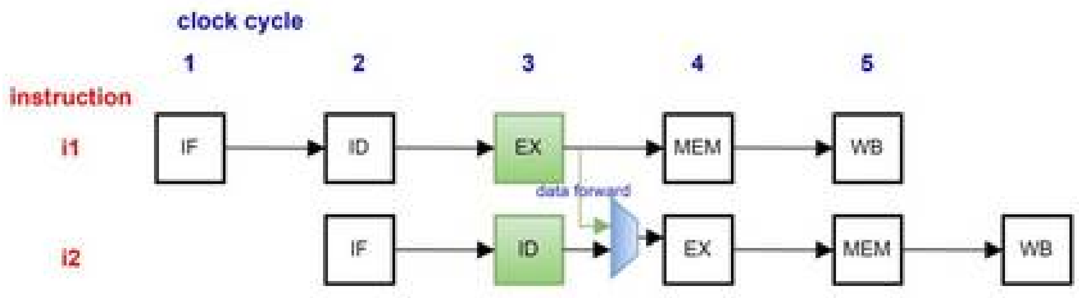
Data/Operand Forwarding

Assume two adjacent instructions i1 and i2 have RAW data dependency between them as both are accessing a register x. The CPU should stall instruction i2 until i1 writes back the result to the register x. If the CPU has no stalling mechanism, the older value is read by i2 from x at Decode stage in 3rd clock cycle. In the 4th clock cycle, i2 gets executed with wrong value of x!



RAW Data Hazard in action!

If you look closely at the pipeline, we already have the result of i1 available in the 3rd clock cycle. It is not written back to register file of course, but still the result is available at the output of Execute stage. So, if we are somehow able to detect the data dependency and then “forward” this data to Execute stage input, then the next instruction can use the forwarded data instead of the data from Decode stage. Thus data hazard is mitigated! The idea looks like this:



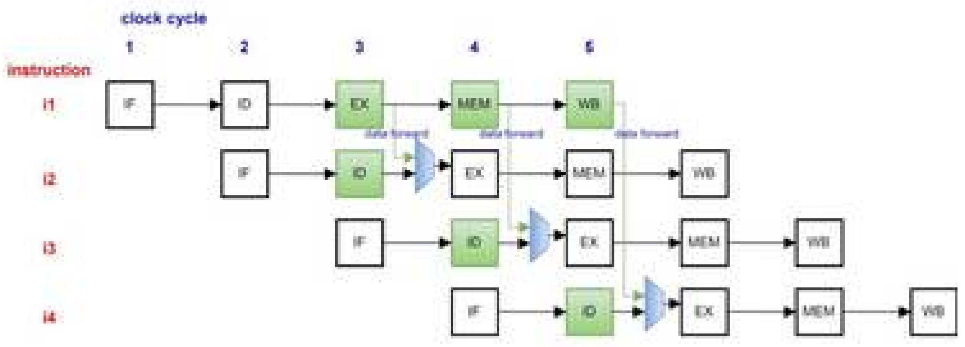
Data Forwarding

This is called *Data/Operand Forwarding or Data/Operand Bypassing* . We forward the data in forward direction in time, so that trailing dependent instructions in the pipeline can access this bypassed data for execution at Execute stage.

The idea can be extended across different stages. In a 5-stage pipeline executing instructions in order i1,i2,....in , data dependency can be there between:

- i1 and i2 – Need to bypass between Execute and Decode stage outputs.
- i1 and i3 – Need to bypass between Memory Access and Decode stage outputs.
- i1 and i4 – Need to bypass between WriteBack and Decode stage outputs.

Architectural solution to mitigate RAW data hazards originating at any stage of the pipeline would look like:



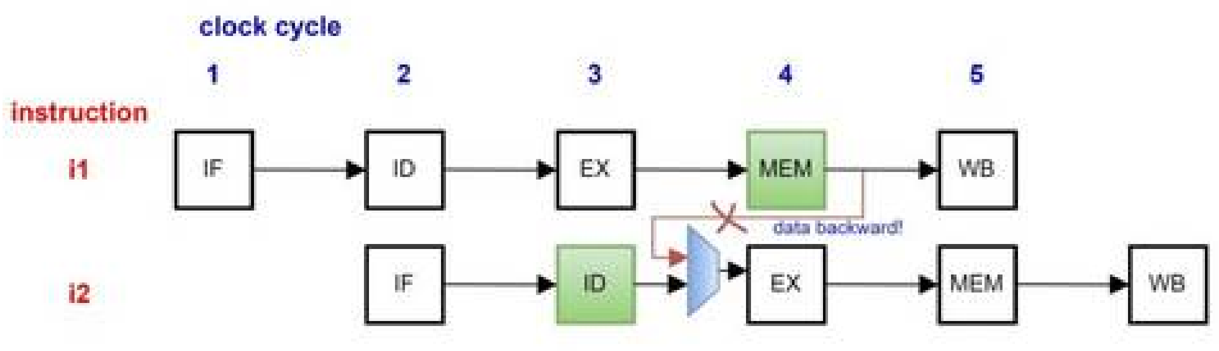
Data Forwarding to mitigate RAW Hazards in 5-stage Pipeline

Pipeline Interlock

Consider the following scenario:

```
Pipeline Interlock Scenario
lw x1, x2, 0xABC  # i1: Load data from data at addr = [x2 + 0xABC] --> x1
add x3, x1, x1    # i2: x3 = x1 + x1 ; x1 should contain value loaded from
memory
```

There is a data dependency between two adjacent instructions i1 and i2 where the first instruction is a Load. This is a special case of data hazard. Here, we cannot execute i2 until data is loaded to x1. So, the question is can we still mitigate this data hazard with data forwarding? The load data will be available only at Memory Access stage of i1, and this has to be forwarded to Decode stage of i2 to prevent the hazard. This requirement would look like:

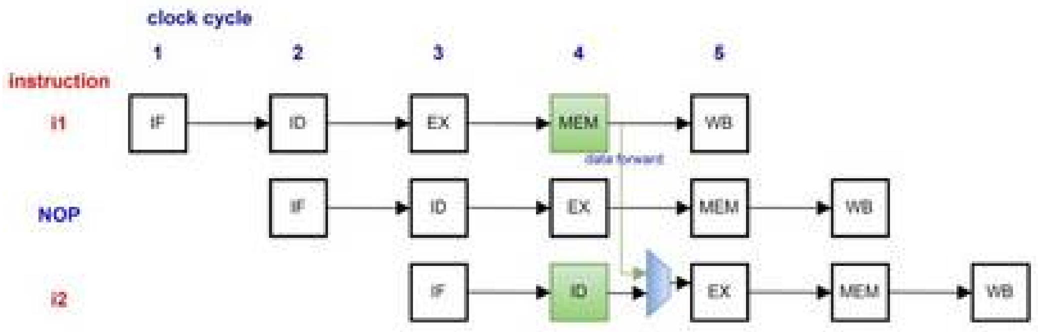


Pipeline Interlock

Say, the load data is available at 4th cycle at Memory Access stage, you need to “forward” this data to 3rd cycle to the Decode stage output of i2 (Why 3rd cycle? Because in 4th cycle i2 would have completed execution in Execute stage already!) . Essentially, you are trying to forward present data to the past, which impossible unless your CPU time travel! This is not data forwarding but “data backwarding” 😀 ….

Data Forwarding can be done only in forward direction in time.

This type of data hazard is called *Pipeline Interlock*. The only way to get around this by stalling the pipeline by one clock cycle by inserting a bubble when this data dependency is detected.



NOP aka Bubble is inserted between i1 and i2. This delays i2 by one cycle, thus data forwarding can now forward load data from Memory Access stage to Decode stage output.

So far, we saw only how to mitigate RAW data hazards. So, what about WAW and WAR hazards? Well, RISC-V architectures are inherently resistant to WAW and WAR hazards for in-order pipeline implementations!

- All writeback to registers happens in the order of issuing. Data written back is always overwritten by later instruction which writes to the same register. So, WAW hazard never happens!

- Writeback is the last stage of pipeline. By the time writeback happens, read would have already completed execution with older data successfully. So WAR hazard never happens!

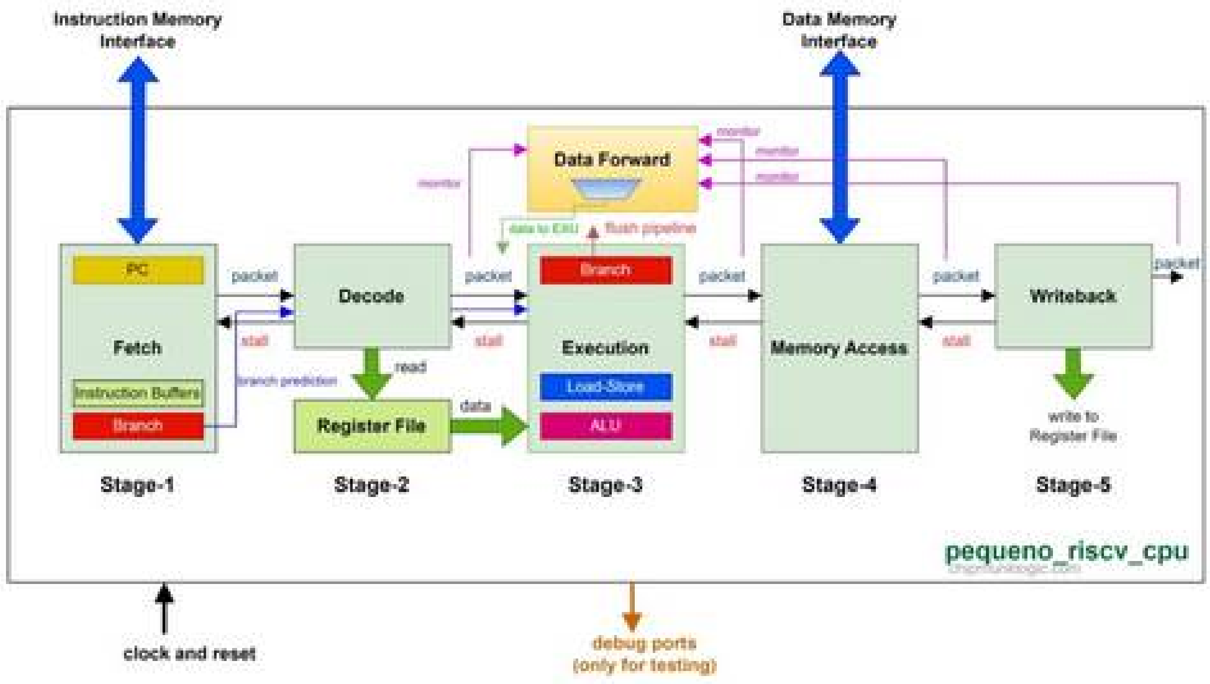
To mitigate RAW data hazards in Pequeno, we will implement data forwarding with pipeline interlock detection feature hardware. More details to be revealed in the upcoming blog where we design Data Forwarding logic.

Scenario	**Solution**
1) WAW/WAR Data Hazards	✅ Inherently mitigated by the in-order pipeline architecture .
2) RAW Data Hazards	✅ Data forwarding + pipeline interlock detection hardware

Table: Mitigating Mechanisms for Data Hazards

Modified Architecture of Pequeno

We have understood and analyzed potential various pipeline hazards that could fail instruction execution by the existing CPU architecture. We also devised solutions and mechanisms to mitigate them. Let’s incorporate the necessary micro-architectures and draft the final architecture of Pequeno RISC-V CPU which is devoid of all kinds of pipeline hazards!



Pequeno – Modified CPU Architecture to mitigate all Pipeline Hazards

Fetch Unit

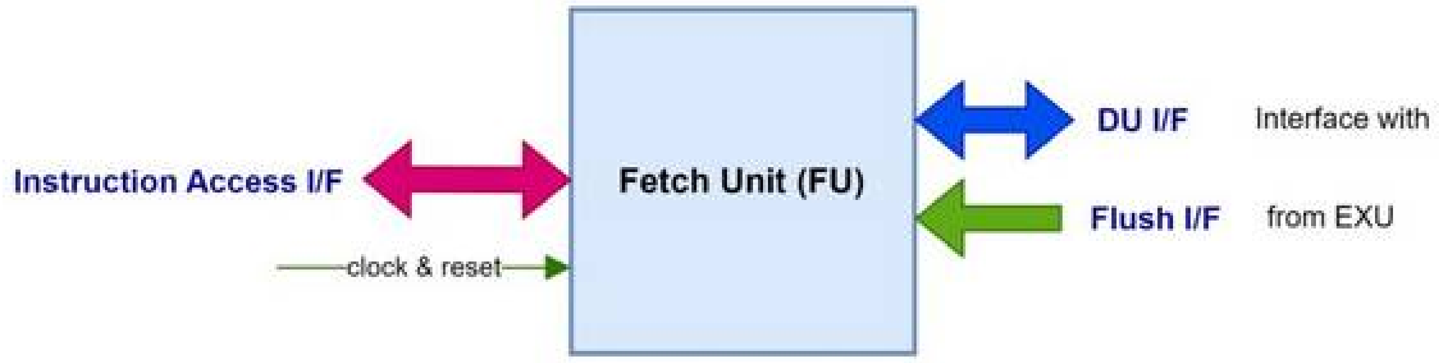
Fetch Unit is the Stage-1 of the CPU pipeline which interfaces with instruction memory. FU fetches instructions from the instruction memory and sends the fetched instruction to **Decode Unit (DU)**. As discussed in the modified architecture of Pequeno in Part-3, FU accommodates a branch prediction logic and flush support.

Interfaces

Let’s define the interfaces for Fetch Unit.

Instruction Access Interface	To access instruction memory/cache
DU Interface	To send the fetched instruction, control/data to Decode Unit
Flush Interface	To flush FU externally

Table: Fetch Unit – Interfaces



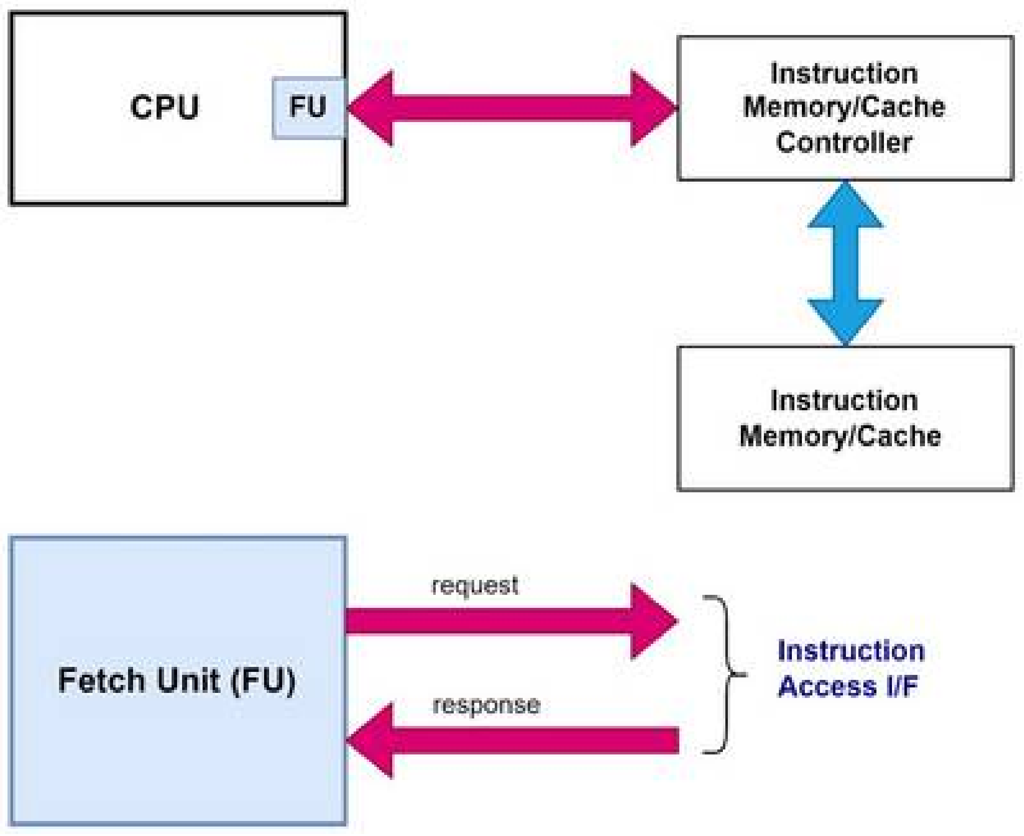
Fetch Unit – Interfaces

Instruction Access Interface

The core functionality of FU in the CPU is instruction access. **Instruction Access I/F** is used for that purpose. Instructions are stored in instruction memory (RAM) during execution. Modern CPUs fetch instructions from cache memory rather than directly from the instruction memory. *Instruction Cache* (in computer architecture terms, this is called a *Primary Cache* or *L1 Cache*) is located closer to CPU and facilitates faster instruction access by caching/storing frequently accessed instructions and pre-fetching a larger chunk of instructions in the vicinity. Thus, there is no need of continuously accessing the slower main memory (RAM). Hence, most of the instructions are accessed fast, directly from the cache.

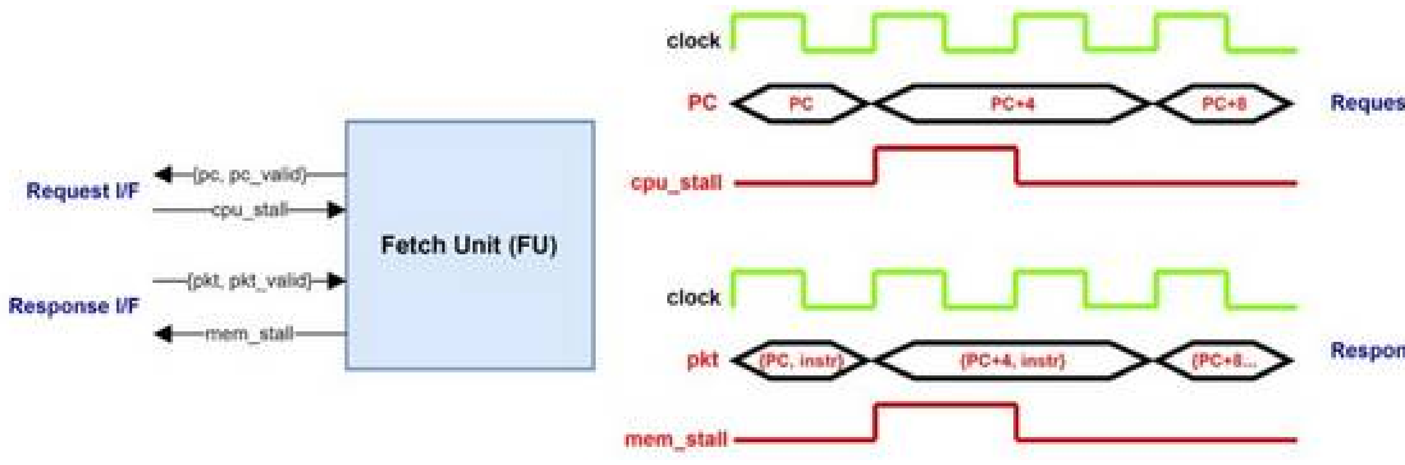
```
Caches are complex designs in computer architecture. Read more about caches
here.
```

CPU doesn’t directly access the interface with an instruction cache/memory. There will be a cache/memory controller in between to control the memory access between them.



Fetch Unit – Instruction Fetch

It would be a good idea to define a standard interface so that any standard instruction memory/cache (IMEM) can be plugged easily to our CPU with minimal or no glue logic. Let's define two interfaces for instruction access. **Request I/F** handles requests from FU to instruction memory. **Response I/F** handles the responses from instruction memory to FU. We will define a simple valid-ready based Request & Response I/Fs for FU, as this is easy to translate to bus protocols like APB, AXI, if required.



Fetch Unit – Request & Response I/F

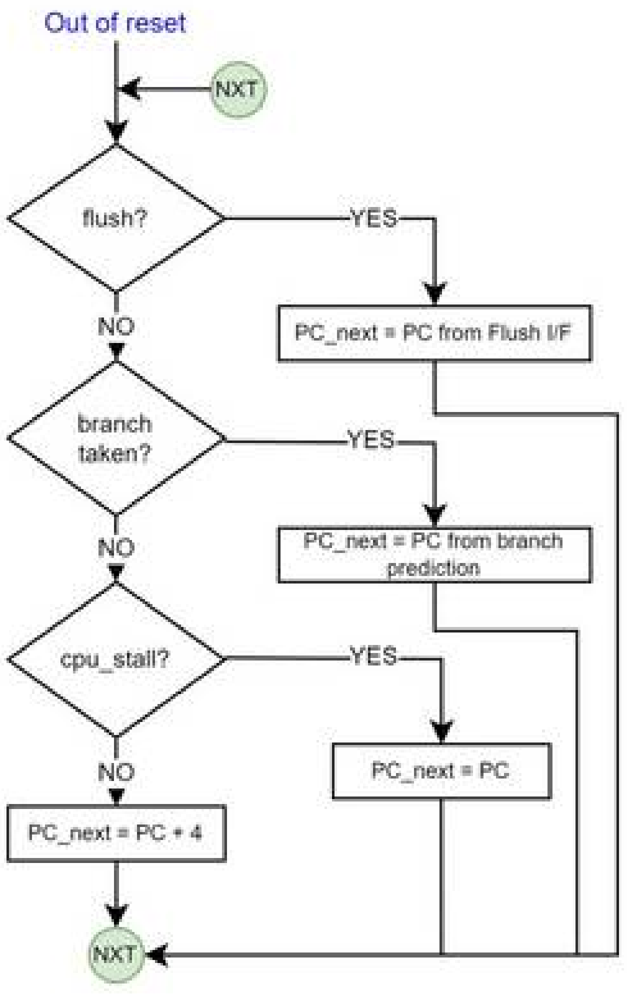
Instruction access requires the address of instruction in the memory. Address to be requested via Request I/F is simply the PC generated by FU. Rather than *ready*, we will use *stall* signal terminology at FU interfaces, which is the inverted version of *ready* in behavior. Cache controllers usually have a stall signal to stall requests from processor. This signal is represented by *cpu_stall*. The response from memory is the fetched instruction received via Response I/F. Along with the fetched instruction, the response should also include the corresponding PC. The PC serves as the ID to identify the request to which the response has been received. Or in other words, it indicates the address of the fetched instruction. This is a vital information which will be required by next stages of the CPU pipeline (*How? We will see it soon!*). Therefore, the fetched instruction and its PC constitute the response packet to FU. CPU may also need to stall responses from instruction memory at times when the internal pipeline is stalled. This signal is represented by *mem_stall*.

At this point, let's define **instruction packet** in our CPU pipeline = {instruction, PC}

PC Generation Logic

At the heart of FU is the PC generation logic which controls Request I/F. Since we are designing a 32-bit CPU, PC should be generated in increments of four. This logic once comes out of reset, generates PC every clock cycle. The on-reset value of PC can be hard-coded. This is the address from which the instructions are fetched and executed by CPU after coming out of reset i.e., the very first instruction's address in the memory. PC generation is free-running logic stalled only by c*pu_stall*.

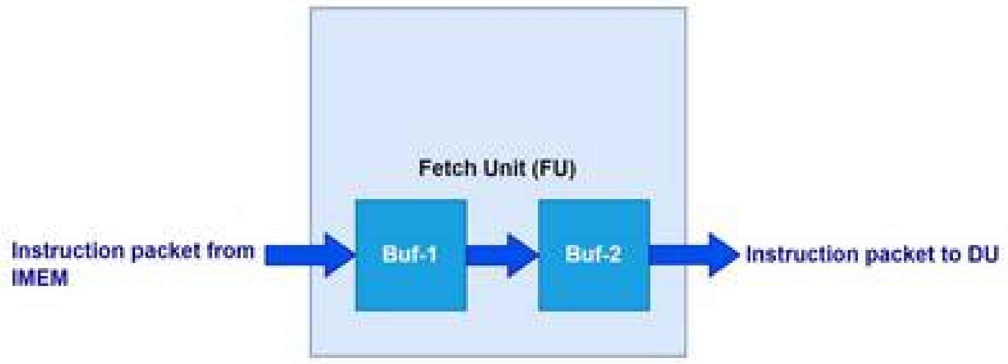
Free-running PC can be bypassed by Flush I/F and internal branch prediction logic. The PC generation algorithm is implemented as:



Fetch Unit – PC Generation Logic

Instruction Buffers

There are two back-to-back instruction buffers inside FU. **Buffer-1** buffers the fetched instruction from instruction memory. Buffer-1 has direct access to Response I/F. **Buffer-2** buffers the instruction from Buffer-1 and then sent it to DU via DU I/F. These two buffers form the internal instruction pipeline in FU.

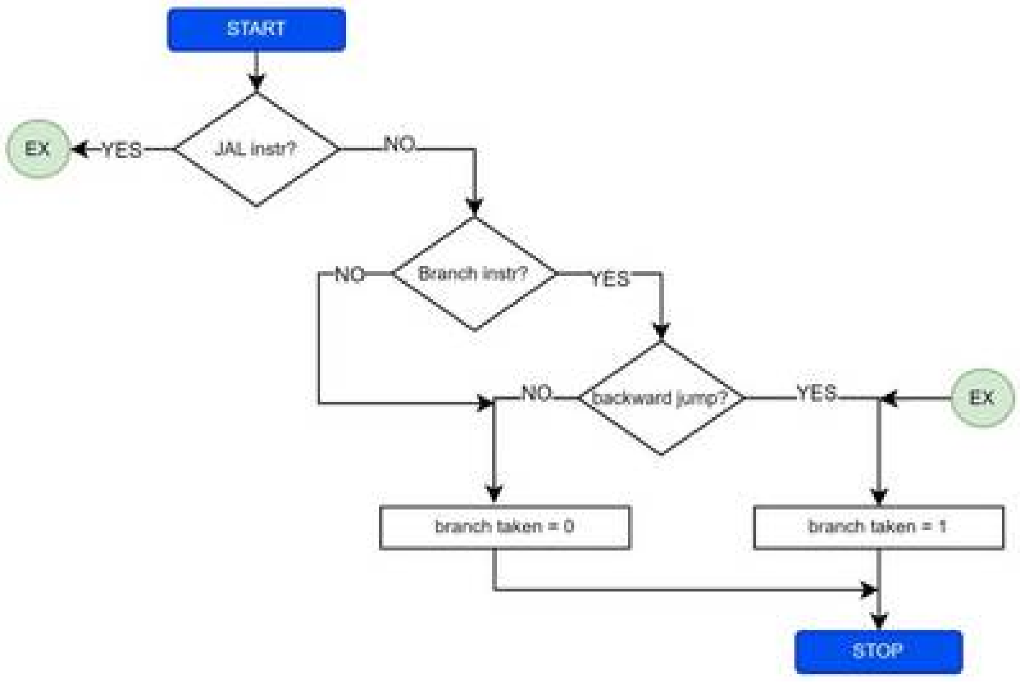


Fetch Unit – Instruction Buffers

Branch Prediction Logic

As discussed in the previous blog, we have to add a branch prediction logic in FU to mitigate control hazards. We will implement a simple and static branch prediction algorithm. Major aspects of the algorithm are:

- Unconditional jumps are always taken.
- If Branch instruction, take the branch if it's a **backward jump**. Because the chances are:
    - This instruction could be part of the loop exit check of some *do-while loop*. There is a higher probability to be correct if we take the branch in this case.
- If Branch instruction, do not take it if it's a **forward jump**. Because the chances are:
    - This instruction could be part of the loop entry check of some *for loop* or *while loop*. There is a higher probability to be correct if we do not take the branch and continue with the next instruction.
    - This instruction could be part of some *if-else* statement. In this case, we always assume that *if* condition is *true* and continue with the next instruction. This bargain theoretically has the probability of 50% to be correct.



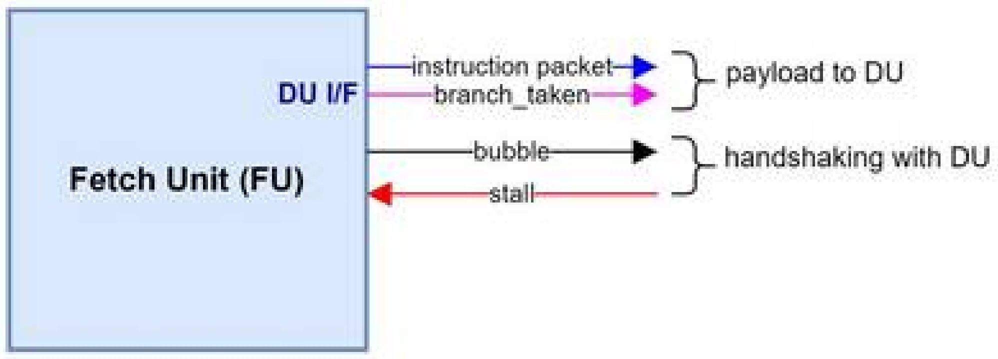
Fetch Unit – Branch Prediction Logic

```
You may want to check pseudo-assembly code for: if-else, for loop, while loop,
do-while loop. I used ChatGPT to generate pseudo-assembly code and reach the
conclusions for branch prediction!
```

Buffer-1 instruction packet is monitored and analyzed by **Branch Prediction Logic** and generates the branch prediction signal: *branch_taken*. The branch prediction signal is then registered and piped forward in synchronization with the instruction packet sent to DU. Branch prediction signal is sent to DU via DU I/F.

DU Interface

This is the primary interface between Fetch Unit and Decode Unit to send the payload. The payload includes the **fetched instruction** and **branch prediction information**.



DU Interface to send payload

Since this is the interface between two pipeline stages of the CPU, valid-ready I/F is implemented. Following signals constitute the DU I/F:

instruction packet	{instruction, PC} to DU
branch_taken	Branch prediction signal to DU

bubble	Inverted version of *valid* to DU
stall	Inverted version of *ready* from DU

Table: Decode Unit Instruction I/F

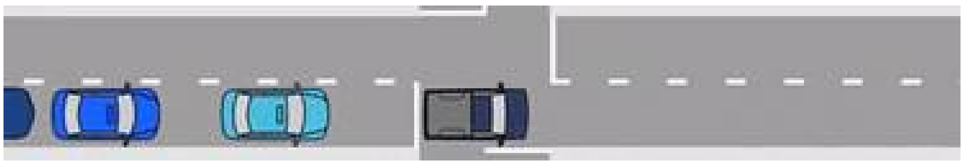
```
Refer to Part-2 to refresh the discussion about the valid-ready I/F designed
between the pipeline stages of Pequeno!
```

Pipeline Stall and Flush in Pequeno

In previous blogs, we discussed the concept and importance of stall and flush in CPU pipeline. We also discussed various scenarios in Pequeno architecture when it would be required to stall or flush. Therefore, appropriate stall and flush logic have to be incorporated in every pipeline stage of the CPU. It is important to identify the conditions at which stall or flush needs to be generated in a stage. And also what part of logic in the stage needs to be stalled and flushed.

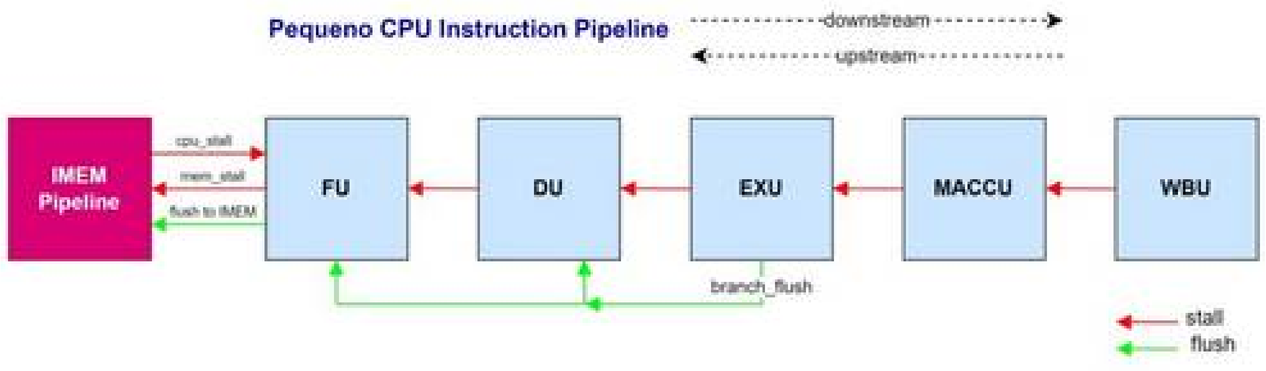
Some initial thoughts before implementing stall and flush logic:

- A pipeline stage may be stalled externally or by internally generated conditions.
- A pipeline stage may be flushed externally or by internally generated conditions.
- There is no centralized stall or flush generation logic in Pequeno. Every stage may have its own stall and flush generation logic.
- A stage can be stalled only by the next stage in the pipeline. The stall from any stage trickles up the pipeline eventually and stalls the entire pipeline in the upstream.



Stall in pipelines is analogous to ripple effect seen in traffic

- A stage can be flushed by any of the stages in the downstream pipeline. This is called a pipeline flush, because the whole pipeline in the upstream needs to be flushed simultaneously. In Pequeno, branch miss in **Execution Unit (EXU)** is the only scenario where a pipeline flush is required.



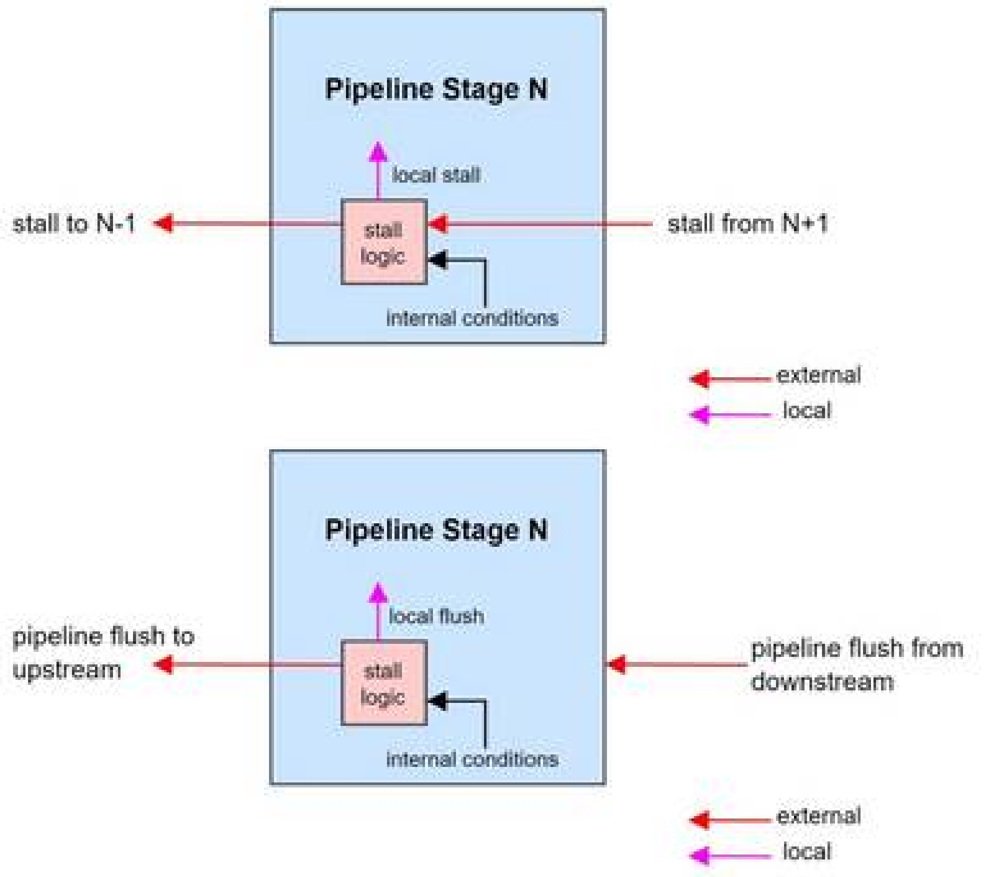
Stall and Flush Network in Pequeno

```
Refer to Part-2 to revisit stall and flush behavior in the CPU pipeline.
```

Stall logic contains the logic to generate local and external stall. **Flush logic** contains the logic to generate local and pipeline flush.

Local stall is generated internally and used locally to stall the operation of the stage. **External stall** is generated internally and sent externally to the next stage in the upstream pipeline. Local and external stalls are generated based on internal conditions and external stall from the next stage in the downstream pipeline.

Local flush is the flush which is generated internally and used locally to flush the stage. **External flush** or **Pipeline flush** is generated internally and sent externally to the upstream pipeline. This flushes all stages in the upstream simultaneously. Local and external flushes are generated based on internal conditions.



Local and External Stall/Flush in Pipeline Stages

Stall Logic

Only DU can externally stall the operation of FU. When DU asserts stall, FU's internal instruction pipeline (Buffer-1 –>Buffer-2) should be stalled immediately, and it should also assert *mem_stall* to IMEM as FU cannot accept anymore packets from IMEM. Depending on the pipeline/buffering depth in the IMEM, PC Generation Logic may also gets eventually stalled by *cpu_stall* from IMEM as no more requests may be accepted by IMEM. There are no internal conditions in FU that generates local stall.

Flush Logic

Only EXU can externally flush FU. EXU initiates *branch_flush* in the CPU instruction pipeline with the address of the next instruction to be fetched after flushing the pipeline (*branch_pc*). FU has provided **Flush I/F** so that external flush can be accepted.

Buffer-1, Buffer-2, PC Generation Logic in FU are flushed by *branch_flush*. The signal *branch_taken* from Branch Prediction Logic also acts like a local flush to Buffer-1, PC Generation Logic. If the branch is taken:

- Next instruction should be fetched from the PC of branch prediction. Therefore, PC Generation Logic should be flushed and next PC should be = *branch_pc*.
- Next instruction at Buffer-1 should be flushed and invalidated i.e., NOP/bubble is inserted.

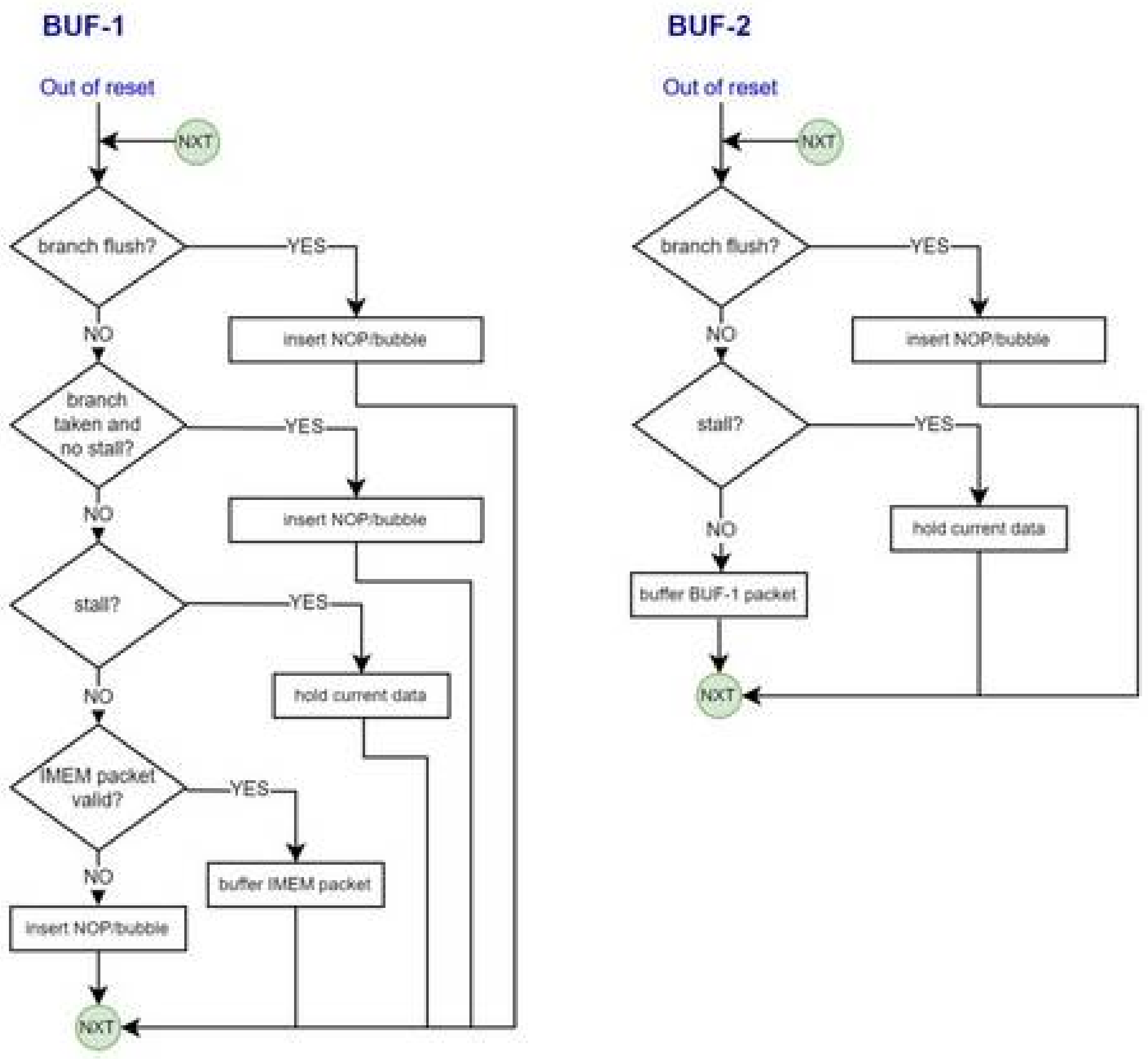


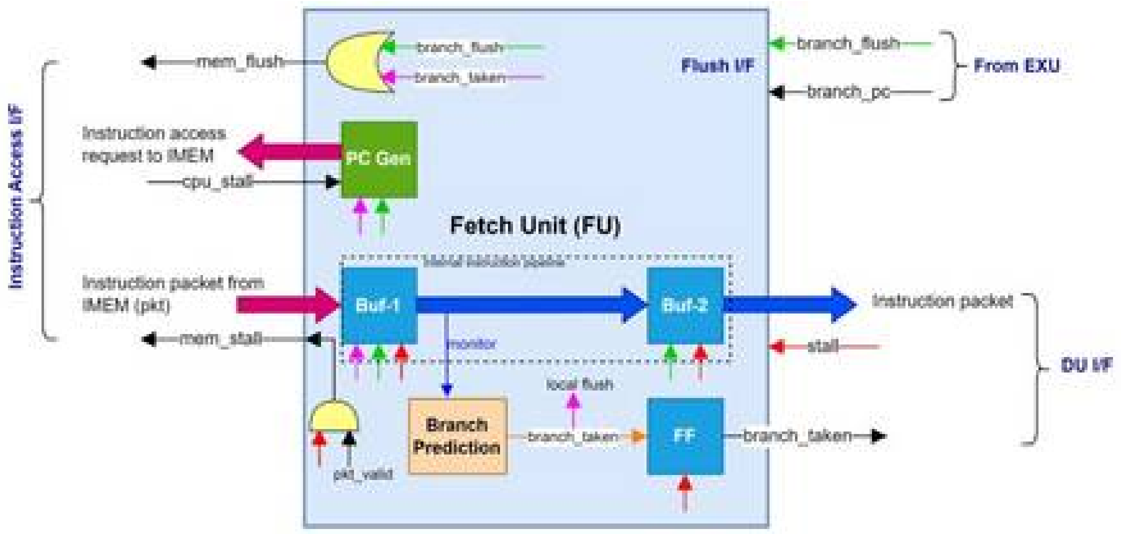
Figure: Buffer-1 and Buffer-2 functionality

Wonder why Buffer-2 is not flushed by *branch_taken*? Because the branch instruction (which is responsible for the flush generation) from Buffer-1 should be buffered at Buffer-2 in the next clock cycle, and allowed to move forward in the pipeline for execution. This instruction shouldn't be flushed off!

Instruction memory pipeline should also be flushed appropriately. IMEM flush *mem_flush* is generated from *branch_flush* and *branch_taken*.

Architecture

Let's integrate all the micro-architectures we designed so far to complete the architecture of Fetch Unit.



Fetch Unit – Architecture

Decode Unit

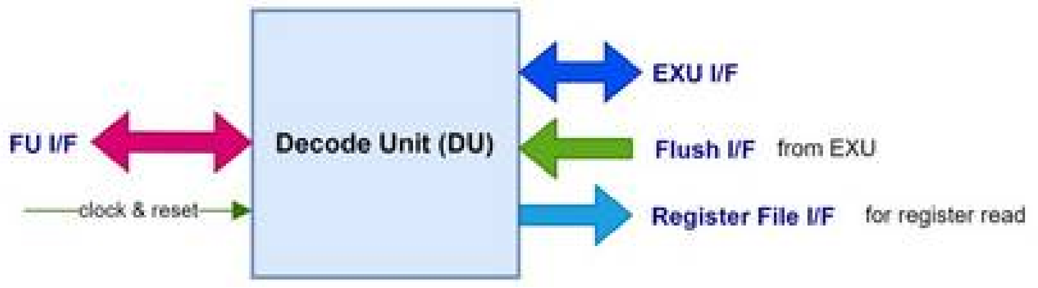
Decode Unit (DU) is the Stage-2 of the CPU pipeline which decodes the instructions from Fetch Unit (FU), and send them to Execution Unit (EXU). It is also responsible for decoding the register addresses and sending them to Register File for register read operation.

Interfaces

Let’s define the interfaces for Decode Unit.

FU Interface	To receive instruction, control/data from Fetch Unit
Register File Interface	To access the source registers (*rs0*, *rs1*) for register read operation
EXU Interface	To send the decoded instruction, control/data to Execution Unit
Flush Interface	To flush DU externally

Table: Decode Unit – Interfaces



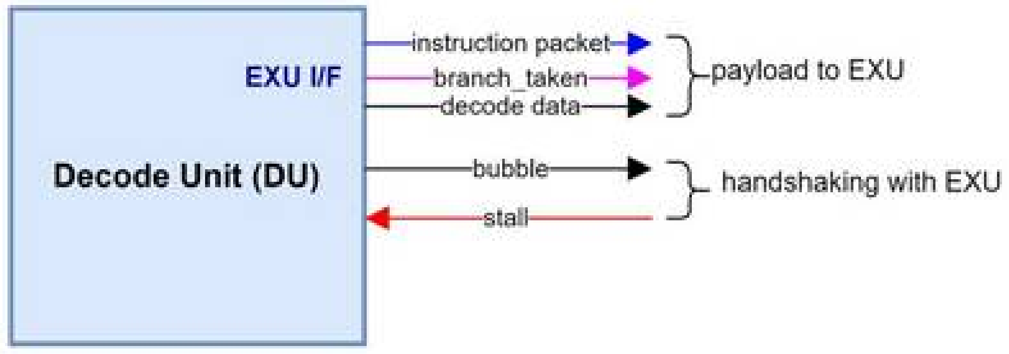
Decode Unit – Interfaces

FU Interface

This is the primary interface between Fetch Unit and Decode Unit to receive the payload. The payload includes the fetched instruction and branch prediction information. This interface was already discussed in the previous part.

EXU Interface

This is the primary interface between Decode Unit and Execution Unit to send the payload. The payload includes the **decoded instruction**, **branch prediction information**, and **decode data**.



EXU Interface to send payload

Following are the instruction and branch prediction signals that constitute the EXU I/F:

instruction packet	{instruction, PC} to EXU
branch_taken	Branch prediction signal to EXU; simply piped forward: FU->DU->EXU
bubble	Inverted version of *valid* to EXU
stall	Inverted version of *ready* from EXU

Instruction packet and branch prediction signals to EXU

Decode data are vital information decoded by DU from the fetched instruction and sent to EXU. Let's gather what information would be required by EXU for the execution of an instruction.

1. **Opcode**, **funct3, funct7**: to identify the operation to be performed by EXU on the operands.
2. **Operands**: depending on the opcode, the operands can be register data (*rs0*, *rs1*), register address for writeback (*rdt*), or 12-bit/20-bit immediate values.
3. **Instruction type**: to identify which operands/immediate values have to processed.

The decoding can be tricky. If you have correctly understood the ISA and the instruction structuring, patterns can be identified for different types of instructions. Identifying patterns helps to design the decoding logic in DU.

Following information are decoded and sent to EXU via EXU I/F.

opcode	Instruction opcode. opcode = instruction[6:0]

rs0, rs1, **rdt**	Source registers0/1, Destination register.
	rs0 = instruction[19:15]
	rs1 = instruction[24:20]
	rdt = instruction[11:7]

funct3/funct7	funct3 = instruction[14:12]
	funct7 = instruction[31:25]

is_<r/i/s/b/u/ j>_type	Instruction type. 1) R-type –> (opcode == 0x33) 2) I-type –> (opcode == 0x67) or (opcode == 0x03) or (opcode == 0x13) 3) S-type –> (opcode == 0x23) 4) B-type –> (opcode == 0x63)

	5) U-type –> (opcode == 0x37) or (opcode == 0x17) 6) J-type –> (opcode == 0x6F)
alu_opcode[3:0]	ALU opcode. Instructions which require the use of ALU are categorized as ALU instructions. They are: 1) R-type instructions 2) I-type instructions 3) U-type instructions LUI & AUIPC instructions require adding operation, hence considered as ALU instructions. R-type: alu_opcode = {funct3, funct7[5]} I-type : alu_opcode = {funct3, funct7[5]} // SLLI/SRLI/SRAI instructions = {funct3, 1'b0} U-type: alu_opcode = 4'b0000
<i/s/b/u/ j>_type_imm	Immediate value. 1) I-type imm[11:0] = instruction[31:20] 2) S-type imm[11:0] = {instruction[31:25], instruction[11:7]} 3) B-type imm[11:0] = {instruction[31], instruction[7], instruction[30:25], instruction[11:8]} 4) U-type imm[19:0] = instruction[31:12] 5) J-type imm[19:0] = {instruction[31], instruction[19:12], instruction[20], instruction[30:21]}

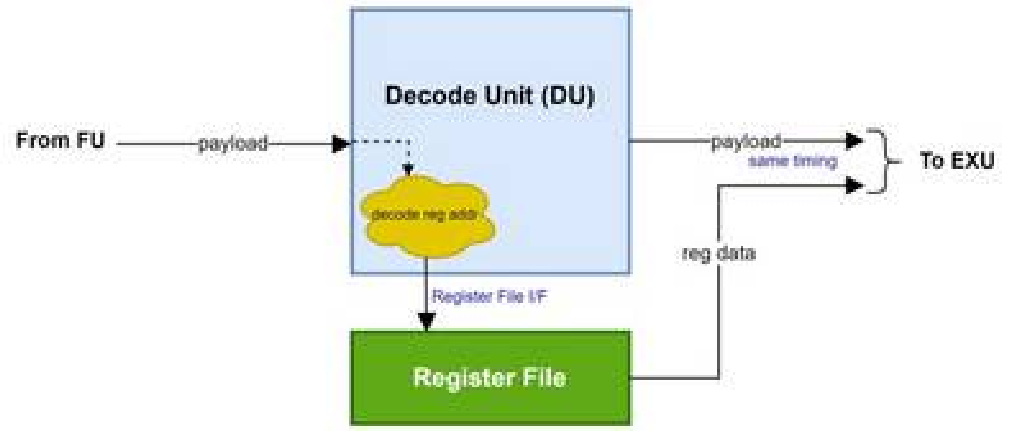
Decode data to EXU

EXU will use this information to de-mux the data to appropriate execution sub-units and execute the instruction.

```
Refer to Part-1 to refresh the ISA and understand the reasoning behind the
decoding logic used by Decode Unit.
```

Register File Interface

For R-type instructions, source registers *rs1*, *rs2*, have to be decoded and read. The data read from the registers are the operands. All the general purpose user registers are present in **Register File** outside DU. Register File Interface is used by DU to send *rs0*, *rs1* addresses to Register File for register access. Along with the payload, the data read from the Register File should also be sent to EXU in the same clock cycle.



Decode Unit and Register File interaction with EXU

Register File requires one cycle to read a register. DU takes one cycle to register the payload to be sent to EXU. The source register addresses are hence decoded directly from FU instruction packet by combinatorial logic. This ensures that the timing of 1) Payload from DU to EXU and 2) Data from Register File to EXU are synchronized.

Stall Logic

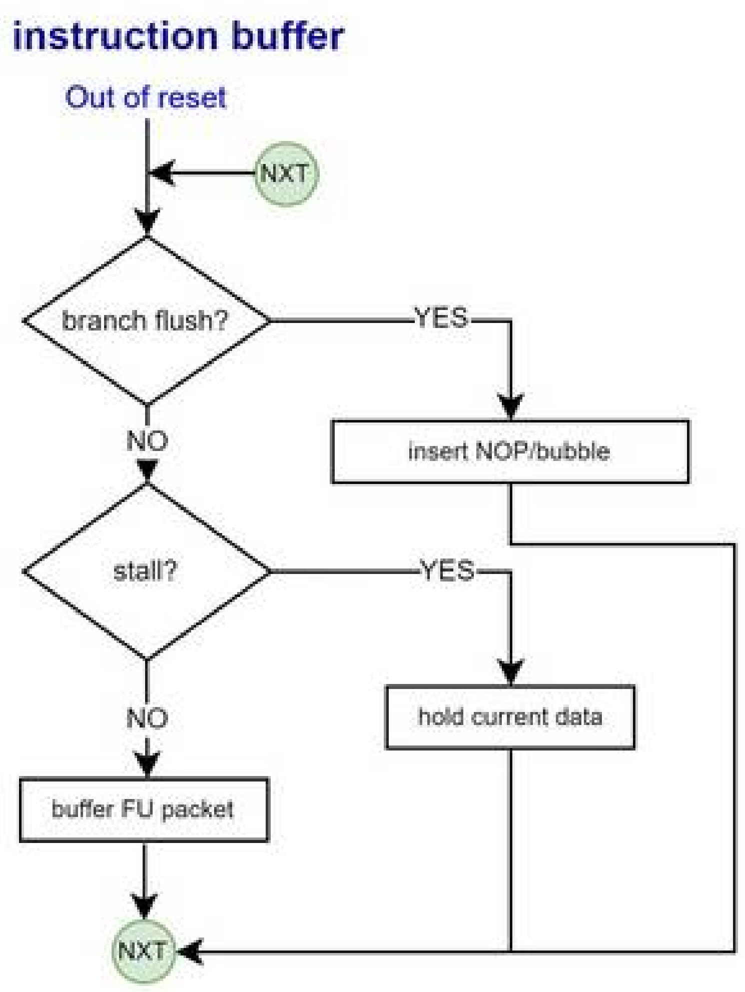
Only EXU can externally stall the operation of DU. When EXU asserts stall, DU's internal instruction pipeline should be stalled immediately, and it should also assert stall to FU as it cannot accept anymore packets from FU. Register File should be stalled together with DU for synchronized operation as both of them are at the same stage of the 5-stage pipeline of the CPU. Hence, DU feeds forward the external stall from EXU to Register File. There are no internal conditions in DU that generates local stall.

Flush Logic

Only EXU can externally flush FU. EXU initiates *branch_flush* in the CPU instruction pipeline with the address of the next instruction to be fetched after flushing the pipeline (*branch_pc*). DU has provided **Flush I/F** so that external flush can be accepted.

The internal pipeline is flushed by *branch_flush*. The *branch_flush* from EXU should immediately invalidate the DU instruction to EXU with 0 cycle delay. This is to avoid potential control hazard in EXU in the next clock cycle.

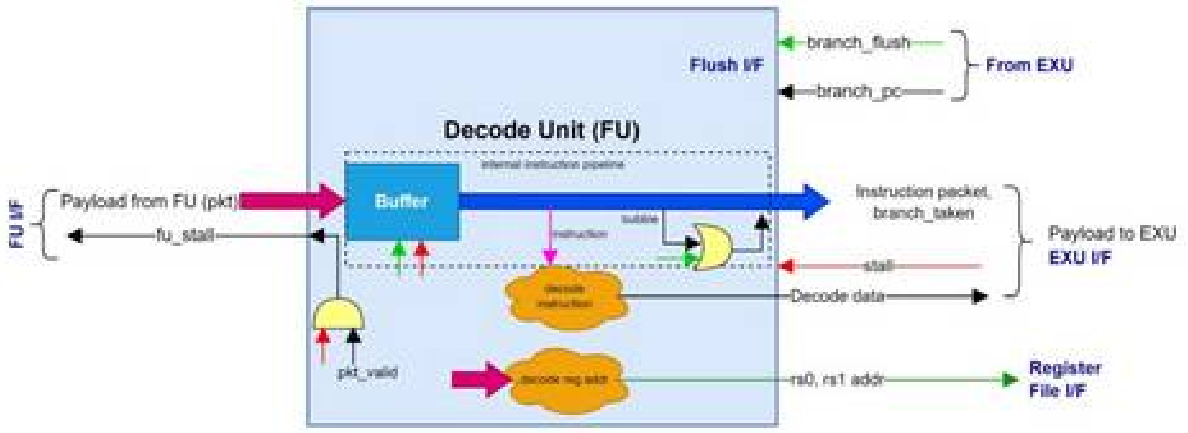
```
In the design of Fetch Unit, we didn't invalidate the FU instruction to DU with
0 cycle delay on receiving branch_flush. This is because the DU will also be in
flush in the next clock cycle, hence no control hazard can happen in DU. So, it
is not necessary to invalidate the FU instruction. The same idea applies to the
instruction from IMEM to FU.
```



The above flow chart represents how the instruction packet and branch prediction data from FU are buffered in DU in the instruction pipeline. Only single stage of buffering is used in DU.

Architecture

Let’s integrate all the micro-architectures we designed so far to complete the architecture of Decode Unit.



Decode Unit – Architecture

Register File

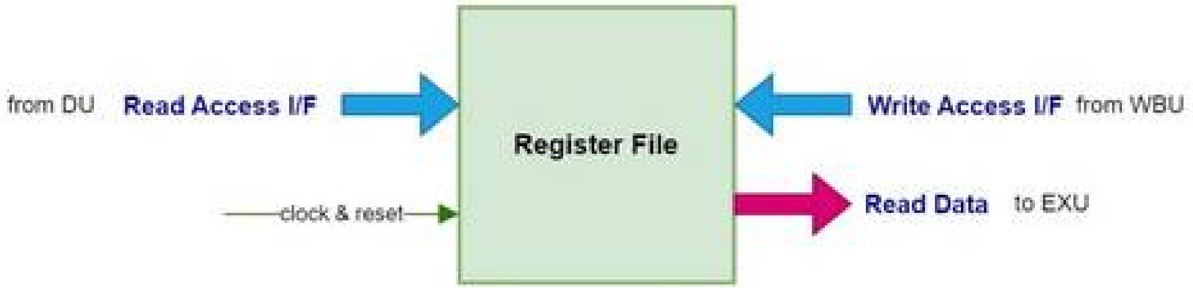
In RISC-V CPUs, the *register file* is a critical component that consists of a set of general purpose registers used for data storage during execution. Pequeno CPU has 32 general-purpose registers of size 32-bit (*x0*–*x31*).

- The register *x0* is called *zero register*. It is hard wired to a constant value 0, providing a useful default value which can be used with other instructions. Say, you want to initialize another register to 0. It is as simple as *mv x1, x0*.
- *x1-x31* are general purpose registers to hold intermediate data, addresses, and results of arithmetic or logic operations.

Interfaces

In the CPU's architecture, which we designed in Part-2, Register File requires two access interfaces.

Read Access Interface	For read access from Decode Unit (DU) The read data is sent to Execution Unit (EXU)
Write Access Interface	For writeback from WriteBack Unit (WBU)



Register File – Interfaces

Read Access Interface

This interface is used to read the register at the address sent by DU. Some instructions for e.g., *ADD*, require two source register operands *rs1*, *rs2*. Therefore, two read ports are required at Read Access I/F, to read two registers simultaneously. The read access should be single-cycle access so that the read data is sent to EXU along with the payload of DU in the same clock cycle. The read data and the DU payload are thus synchronized in the pipeline.

Write Access Interface

This interface is used to writeback the result of execution to the register at the address sent by the WBU. Only one destination register, *rdt*, is written at the end of execution. Hence, one write port is sufficient. The write access should be single-cycle access.

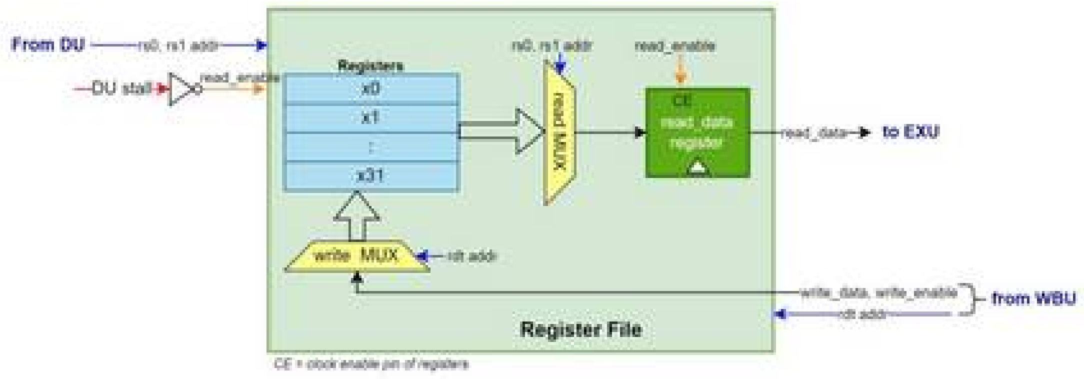
Stall and Flush Logic

Since DU and Register File are required to be in sync at the same stage of pipeline, they should stall together always (Why? Check the block diagram from the last part!). For example, if DU gets stalled, Register File should not pump out the read data to EXU, as it will corrupt the pipeline. Register File should also stall in this scenario. This is ensured by generating the *read_enable* to Register File by inverting the stall signal to DU. When stall is active, *read_enable* is driven low and the previous data is retained at the read data output, effectively stalling the Register File operation.

Since Register File doesn’t send any instruction packet to EXU, it doesn’t require any flushing logic. Flushing logic needs to be taken care only inside the DU.

Architecture

To summarize, Register File is designed with **two independent read ports** and **one write port**. Both read and write accesses are single-cycle. The read data is registered. The final architecture would look like:



Register File – Architecture