154B Discussion 4

January 28th, 2022

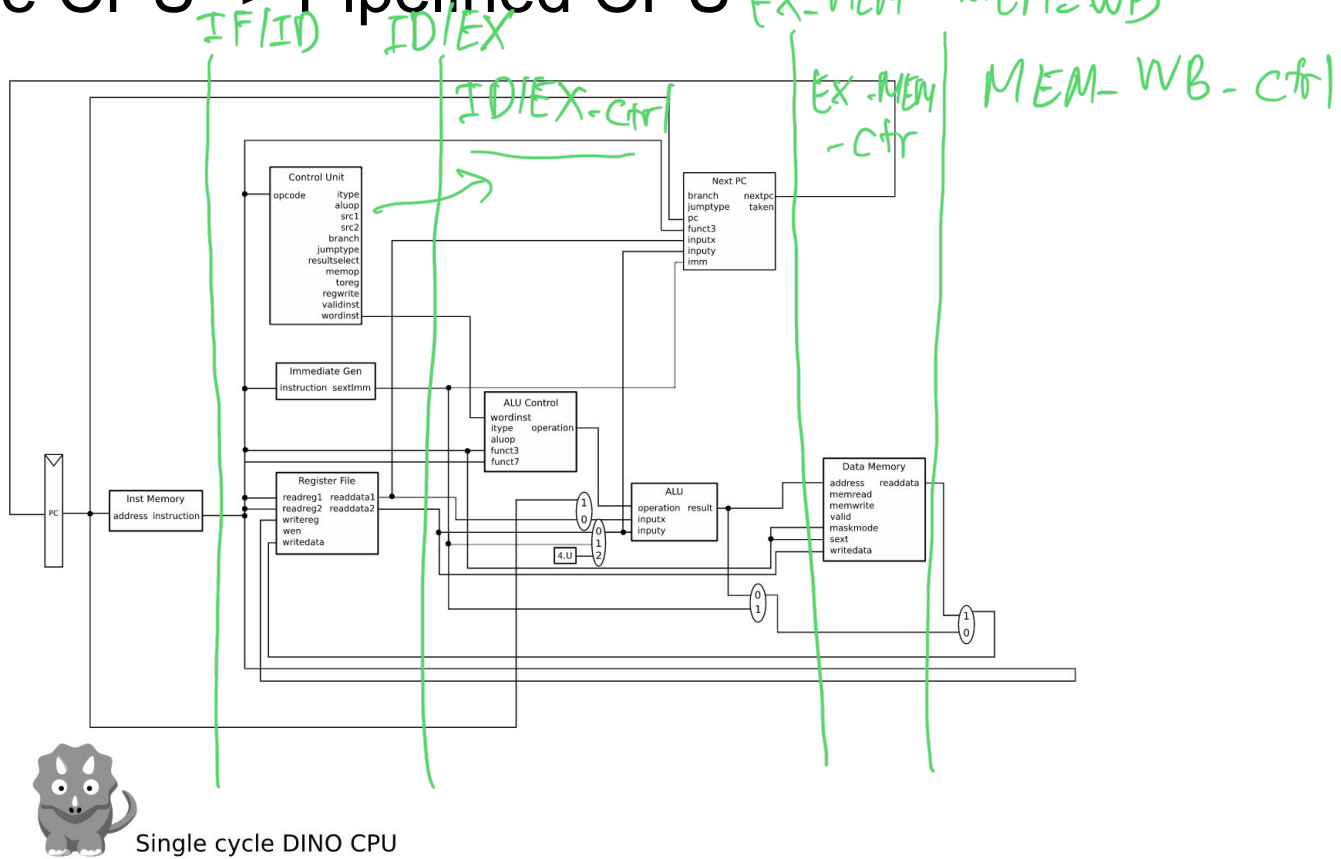
Goals

- Understanding transitioning from the single-cycle design to the 5-stage pipelined design.
- Understanding stage registers.
- Understanding how forwarding/stalling/flushing work in DINOCPU.
- Debugging pipelined CPU.

Logistics

- Assignment 3:
  - Prompt: https://jlpteaching.github.io/comparch/modules/dino%20cpu/assignment3/
  - Repo: https://github.com/jlpteaching/dinocpu-wq22
  - Due date: Feb-09 11:59PM for both part 3.1 and part 3.2.
  - The diagram was updated this morning, the writereg wire was wrong.

Single-cycle CPU -> Pipelined CPU



Single cycle DINO CPU

Stage Registers

- Implemented in: https://github.com/jlpteaching/dinocpu-wq22/blob/main/src/main/scala/pipelined/stage-register.scala
- For transferring the instruction executing context between stages.
- We use seven stage registers:
  - IF/ID
  - ID/EX
  - ID/EX_ctrl: keeping control signals used in Execute stage.
  - EX/MEM
  - EX/MEM_ctrl: keeping control signals used in Memory stage.
  - MEM/WB
  - MEM/WB_ctrl:keeping control signals used in Writeback stage.

↑ control signals, register values

(EX)

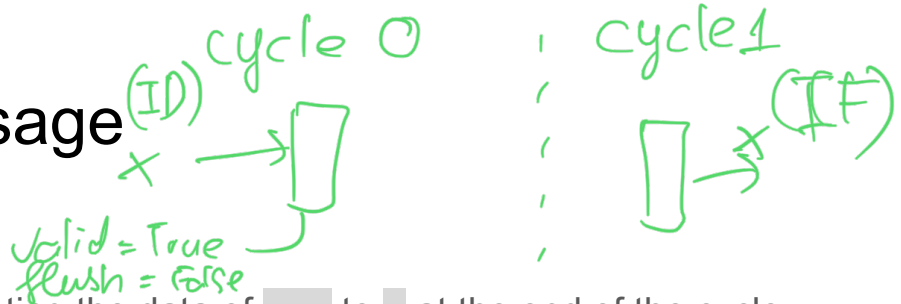
(EX, MEM)

(EX, MEM, WB)

Stage Registers: Notes

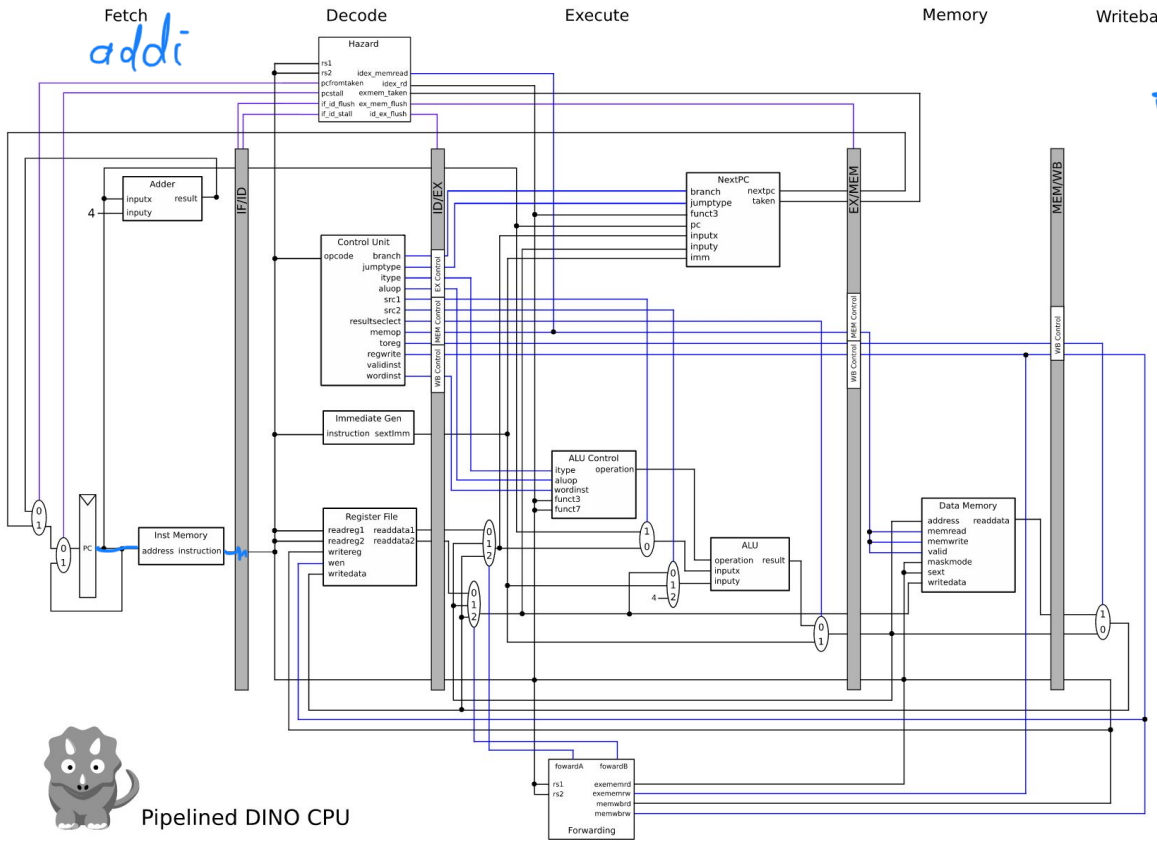
- Note: Using Stage Register is not the only way we transfer data between stages. For example, `next_pc` is determined in Memory stage and is used in Fetch stage.
- This is technically possible, but it is not recommended to bundle all signals in the stage registers.

Stage Registers: Usage



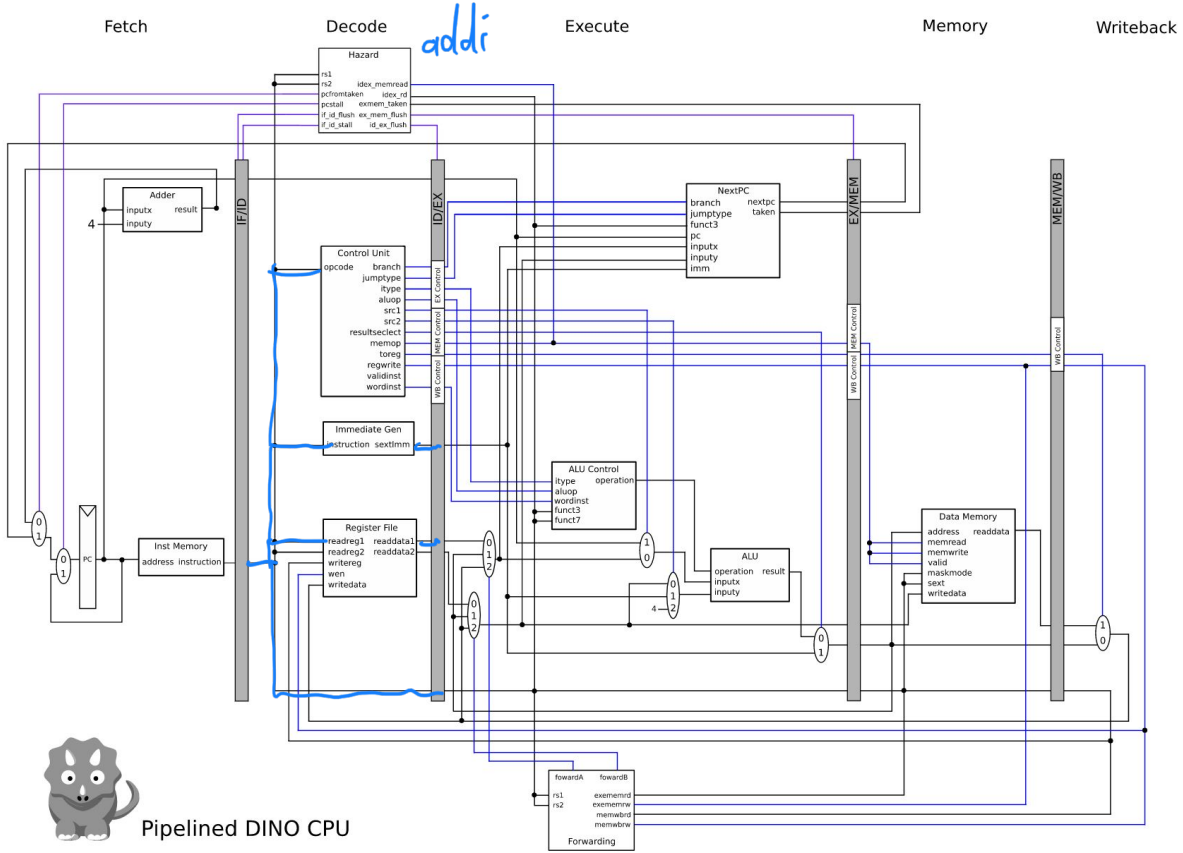
- It is a register,
  - `reg := x` means updating the data of `reg` to `x` at the end of the cycle.
  - `y := reg` means assigning data of `reg` to `y`.
- Updating stage registers,
  - E.g., `id_ex.io.in.inputx := some_signal_from_decode` (sending signal from Decode to Execute at the end of the cycle)
- Using signals from stage registers,
  - E.g., `some_io_in_execute := id_ex.io.data.inputx` (getting signal sent by Decode in the previous cycle)
- However, `id_ex.io.in.inputx` only becomes `id_ex.io.data.inputx` in the next cycle only if both conditions satisfy,
  - `id_ex.io.valid === true.B`
  - `id_ex.io.flush === false.B`

Example 1: cycle 1



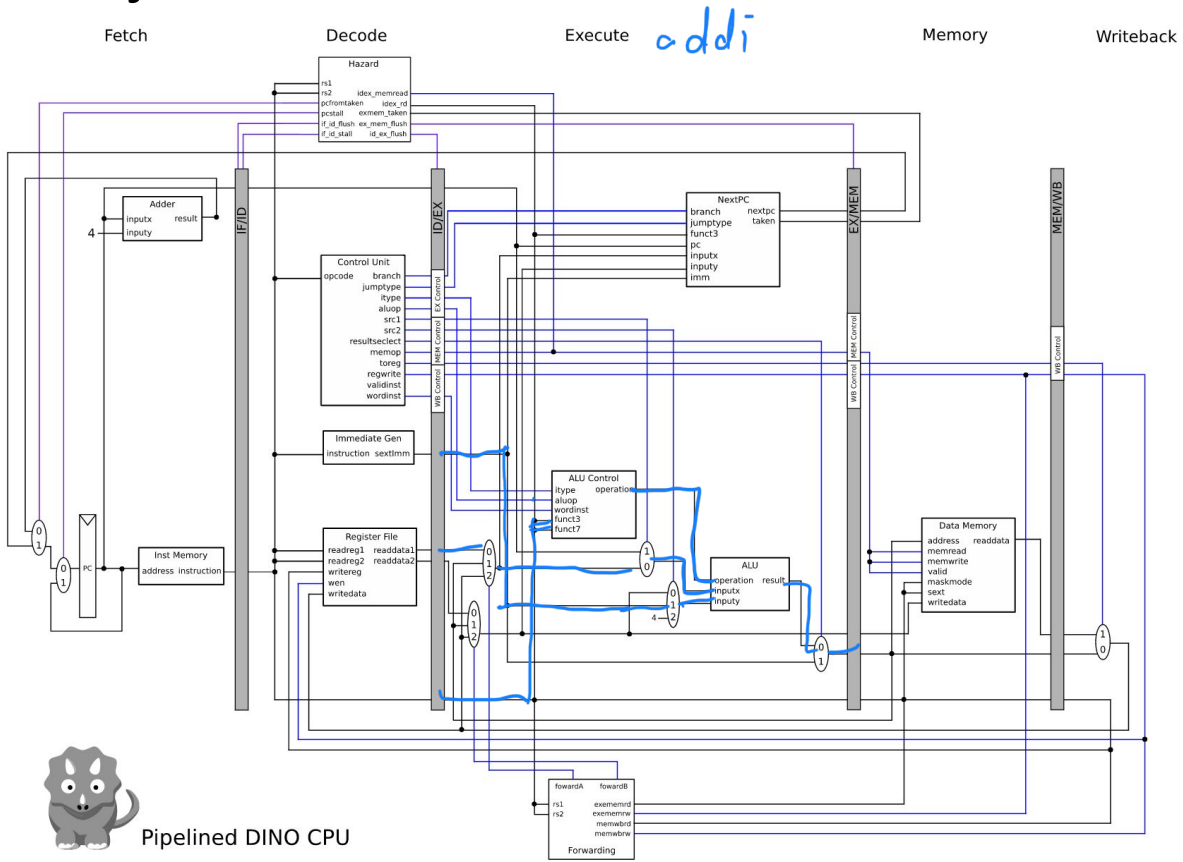
inst 1
addi rd, rs1, imm
nop
nop
nop
nop

Example 1: cycle 2

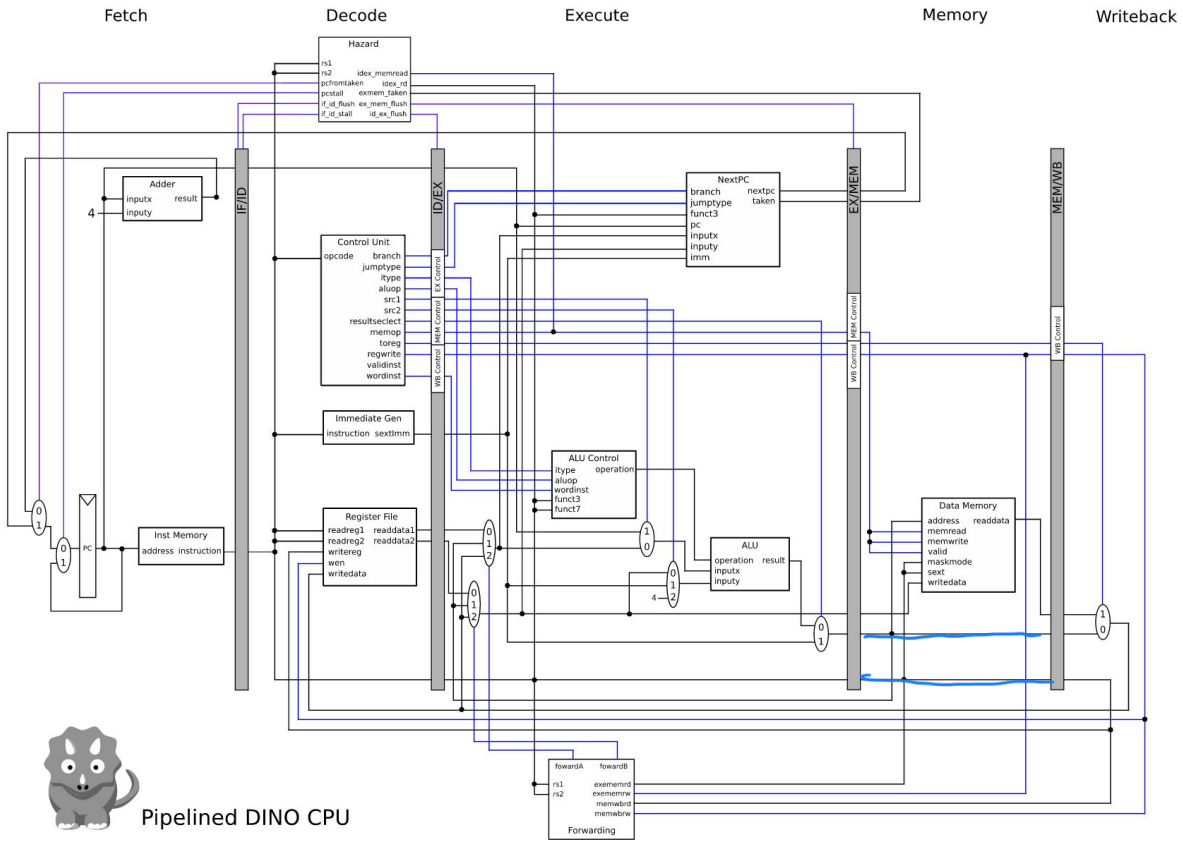


Pipelined DINO CPU

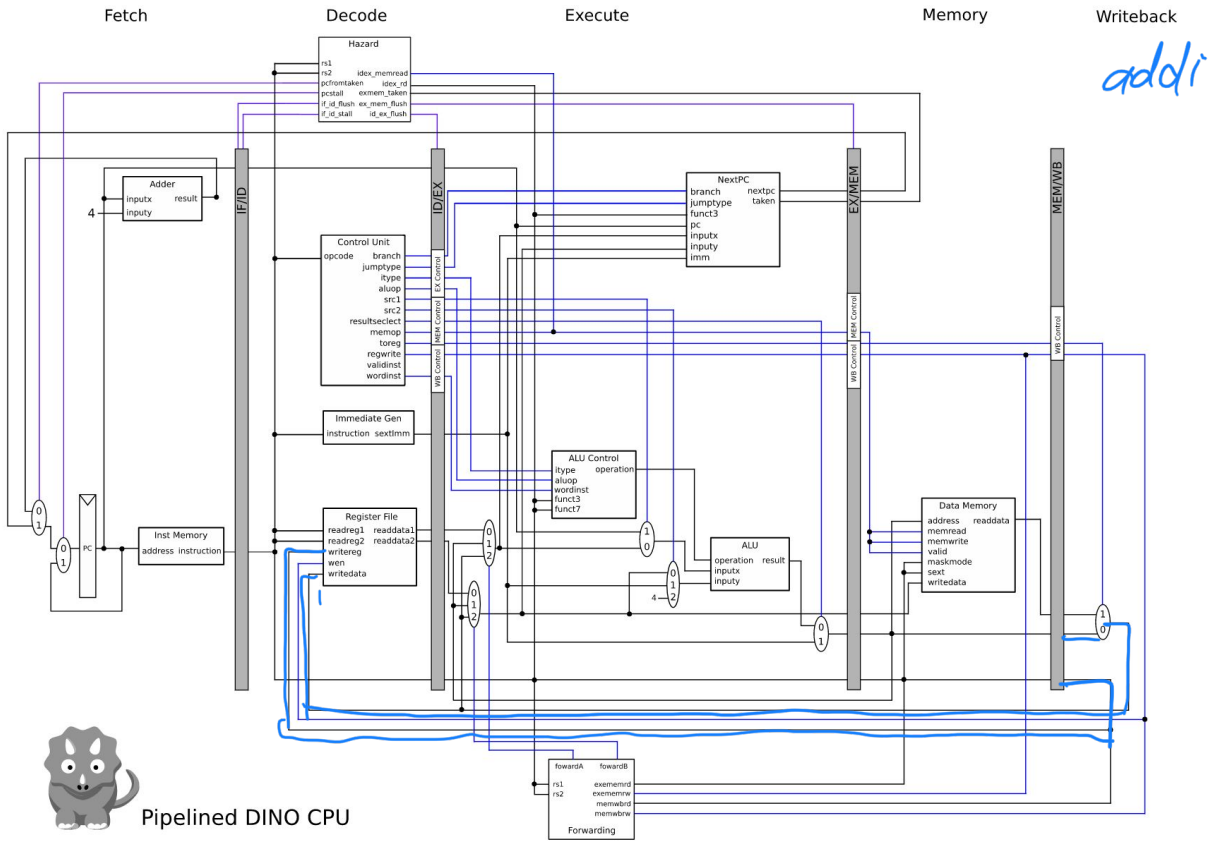
Example 1: cycle 3



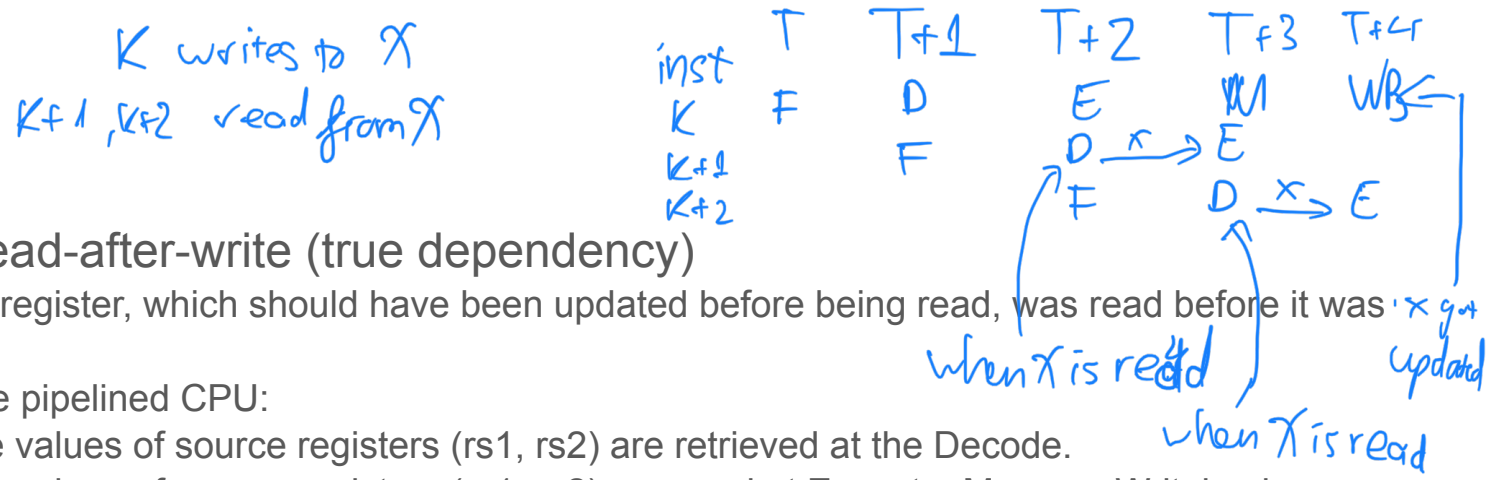
Example 1: cycle 4



Example 1: cycle 5

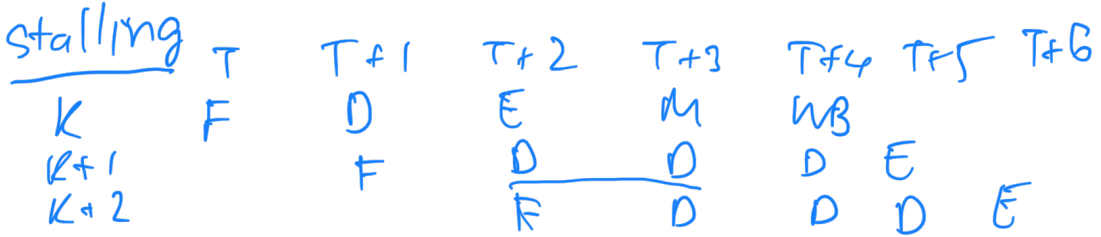


Forwarding

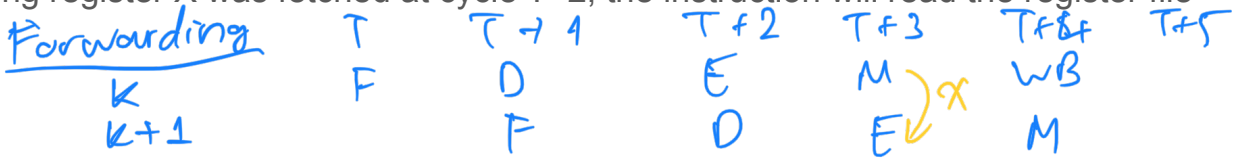


- Problem: read-after-write (true dependency)
  - A source register, which should have been updated before being read, was read before it was updated.
  - In 5-stage pipelined CPU:
    - The values of source registers (rs1, rs2) are retrieved at the Decode.
    - The values of source registers (rs1, rs2) are used at Execute, Memory, Writeback.
    - The value of destination register (rd) is updated at Writeback stage.
    - Problem occurs when, for example,
      - If instruction K writing to register X was fetched at cycle T, the register file will not be updated before cycle T+4.
      - If instruction K+1 reading register X was fetched at cycle T+1, the instruction will read the register file at T+2.
      - If instruction K+2 reading register X was fetched at cycle T+2, the instruction will read the register file at T+3.

Forwarding



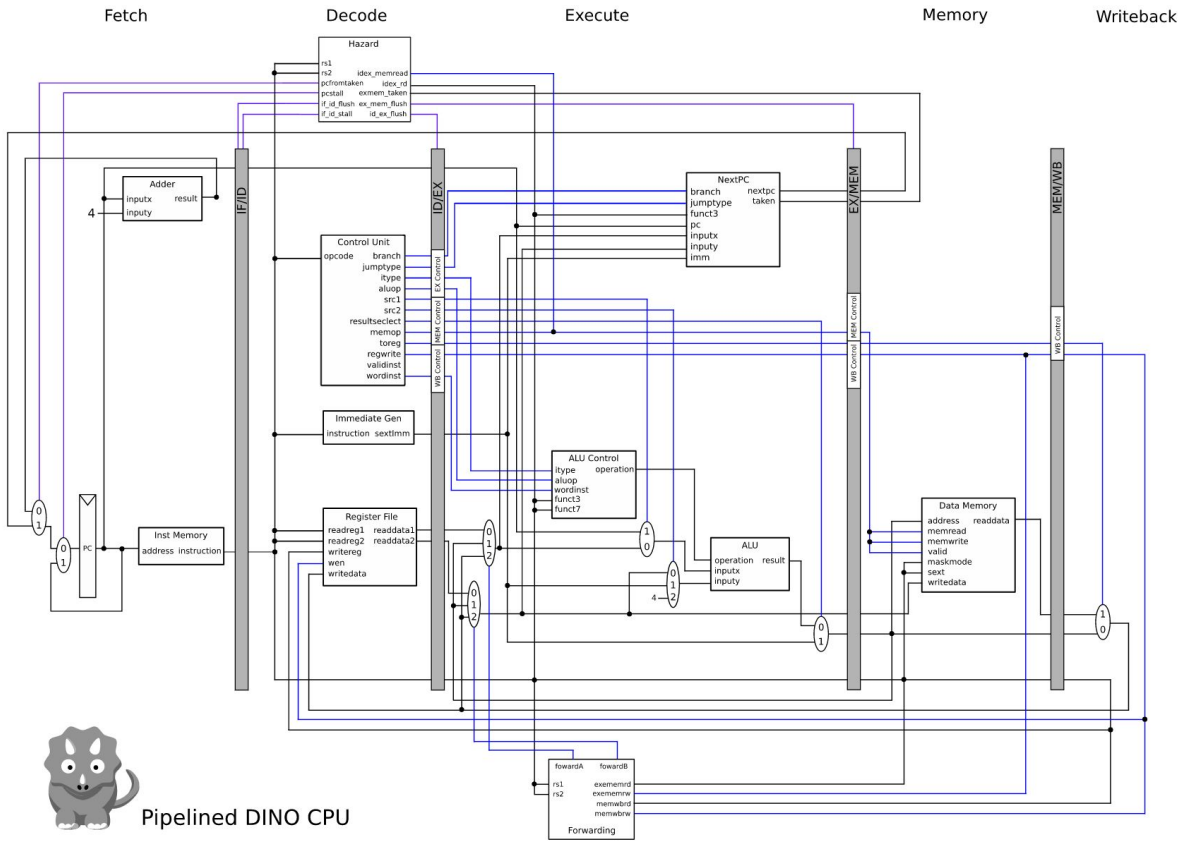
- Problem: read-after-write (true dependency)
  - If instruction K writing to register X was fetched at cycle T, the register file will not be updated before cycle T+4.
  - If instruction K+1 reading register X was fetched at cycle T+1, the instruction will read the register file at T+2.
  - If instruction K+2 reading register X was fetched at cycle T+2, the instruction will read the register file at T+3.
- Possible solutions:
  - Doing nothing
    - Fewer wires, the CPU will be kept busy, but functionally incorrect :(
  - Stalling: don't move the instruction K+1 out of the Fetch stage until T+4.
    - Functionally correct, but the CPU won't do anything for 2 cycles :(
  - Forwarding: wiring the result from Execute stage (ALU's result or immediate) of instructions at Memory stage and Writeback stage back to the instruction at Execute stage if necessary.
    - Also functionally correct, the CPU is kept busy, but more wirings and logic can be complex :(



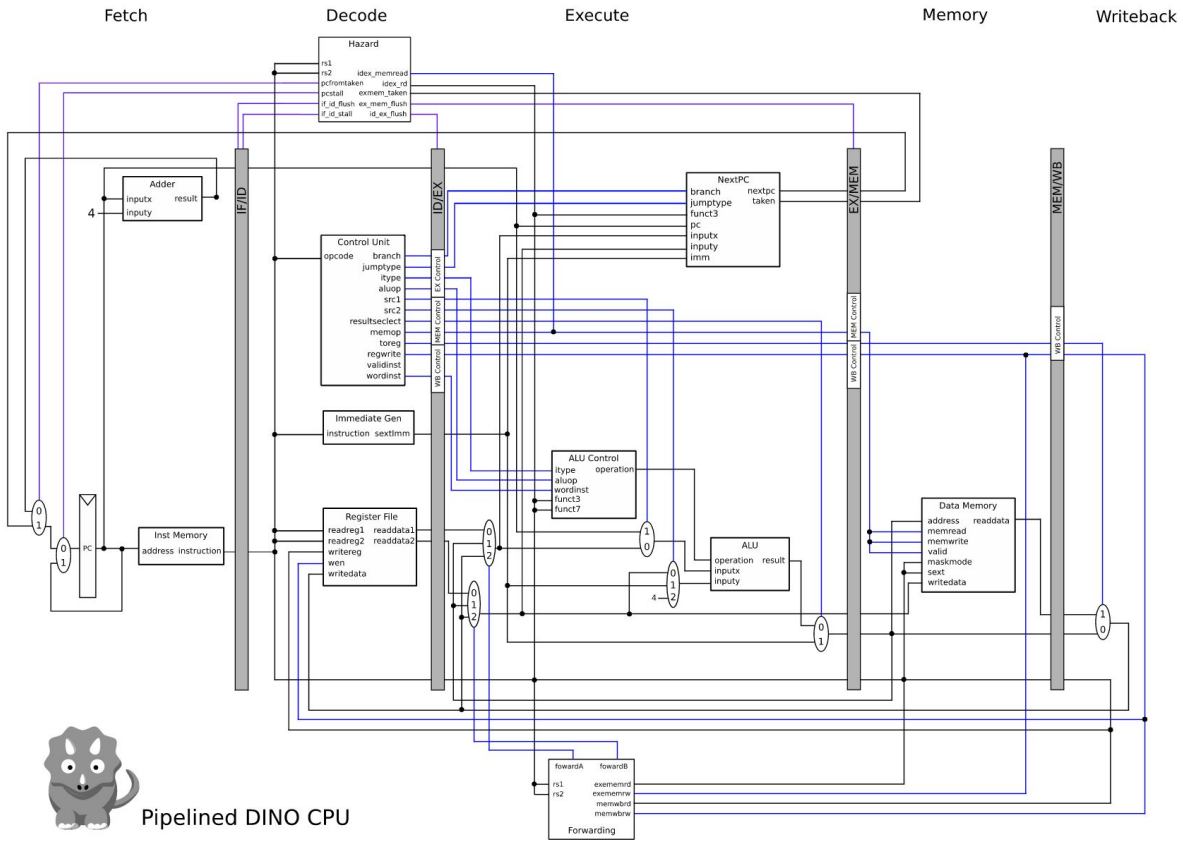
Forwarding

- Problem: read-after-write (true dependency)
  - A source register, which should have been updated before being read, was read before it was updated.
  - In 5-stage pipelined CPU:
    - The values of source registers (rs1, rs2) are retrieved at the Decode.
    - The values of source registers (rs1, rs2) are used at Execute, Memory, Writeback.
    - The value of destination register (rd) is updated at Writeback stage.
    - Problem occurs when, for example,
      - If instruction K writing to register X was fetched at cycle T, the register file will not be updated before cycle T+4.
      - If instruction K+1 reading register X was fetched at cycle T+1, the instruction will read the register file at T+2.
      - If instruction K+2 reading register X was fetched at cycle T+2, the instruction will read the register file at T+3.
- Solution:
  - ~~Doing nothing~~
    - ~~Fewer wires, the CPU will be kept busy, but functionally incorrect~~
  - ~~Stalling: don't move the instruction K+1 out of the Fetch stage until T+4.~~
    - ~~Functionally correct, but the CPU won't do anything for 2 cycles~~
  - Forwarding: wiring the result from Execute stage (ALU's result or immediate) of instructions at Memory stage and Writeback stage back to the instruction at Execute stage if necessary.
    - Also functionally correct, the CPU is kept busy, but more wirings and logic can be complex
- Why forwarding?
  - Being functionally correct is required for general purpose CPUs.
  - Cycle count matters.
  - Transistors are abundant.

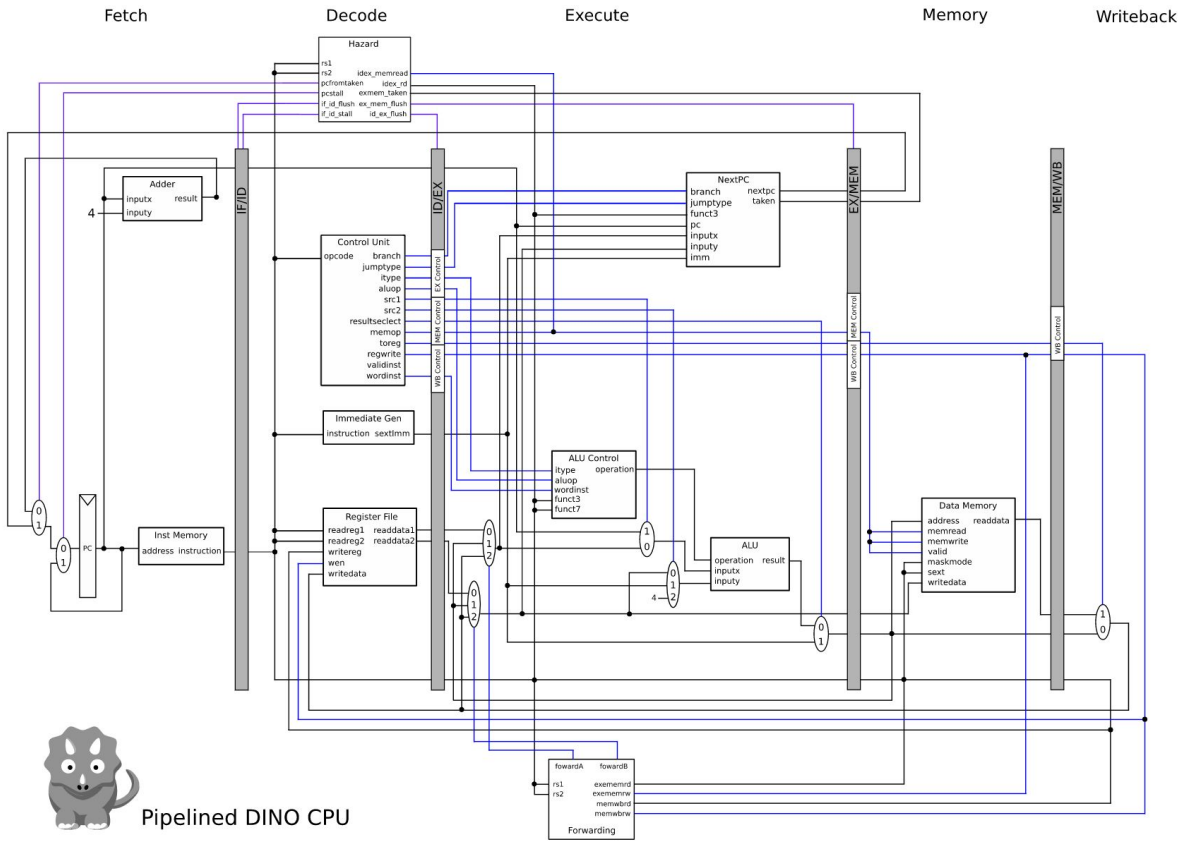
Example 2: cycle 1



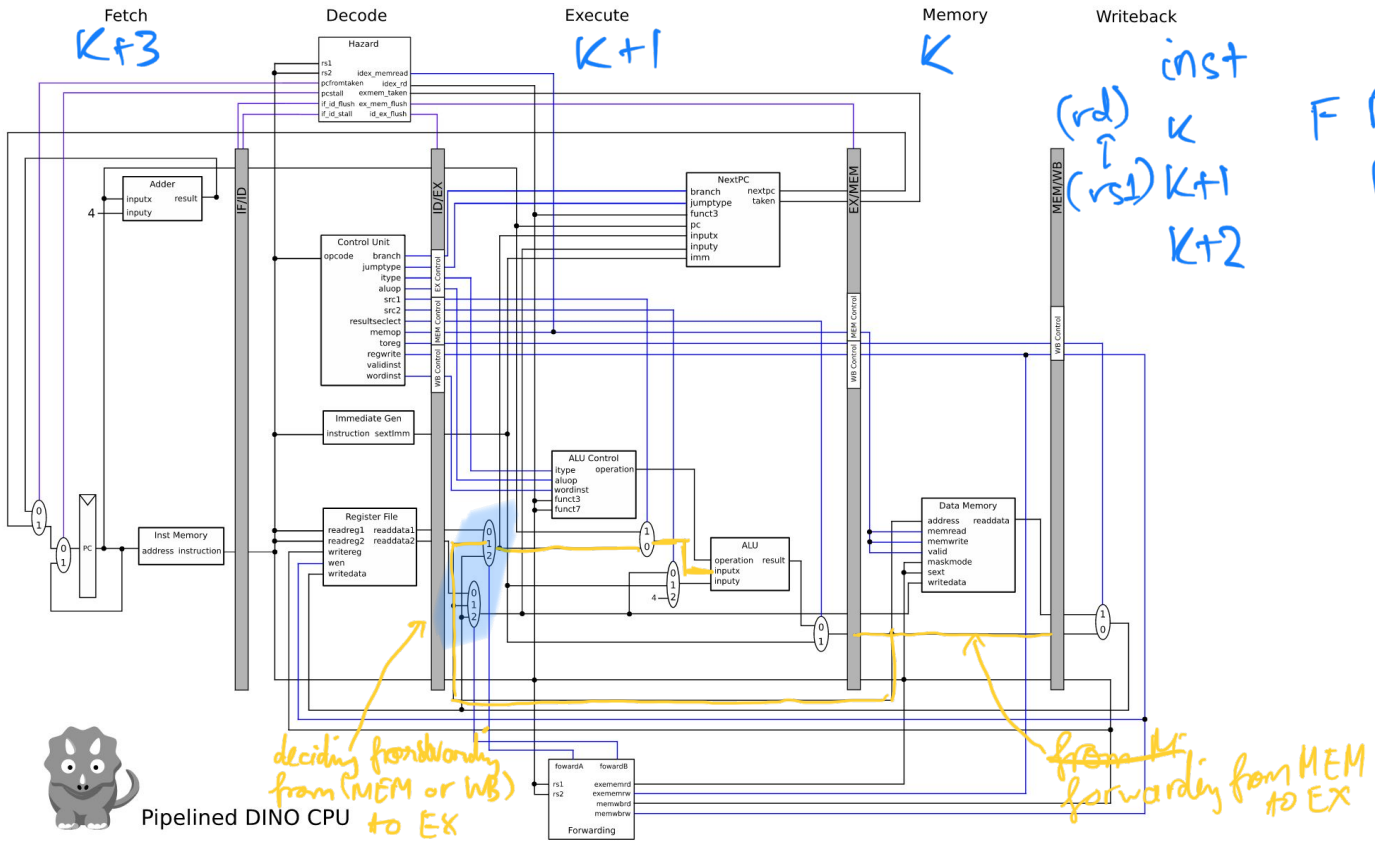
Example 2: cycle 2



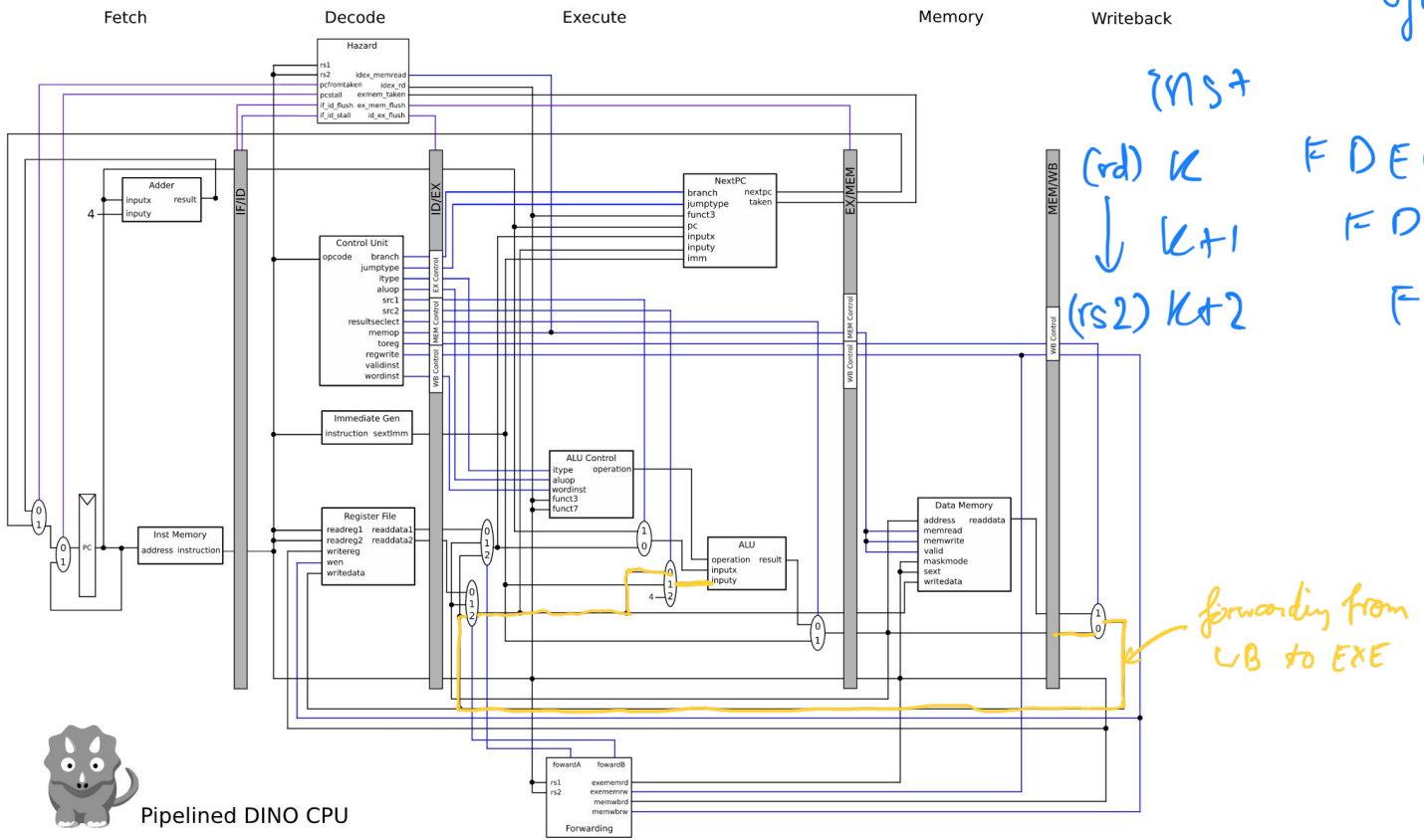
Example 2: cycle 3



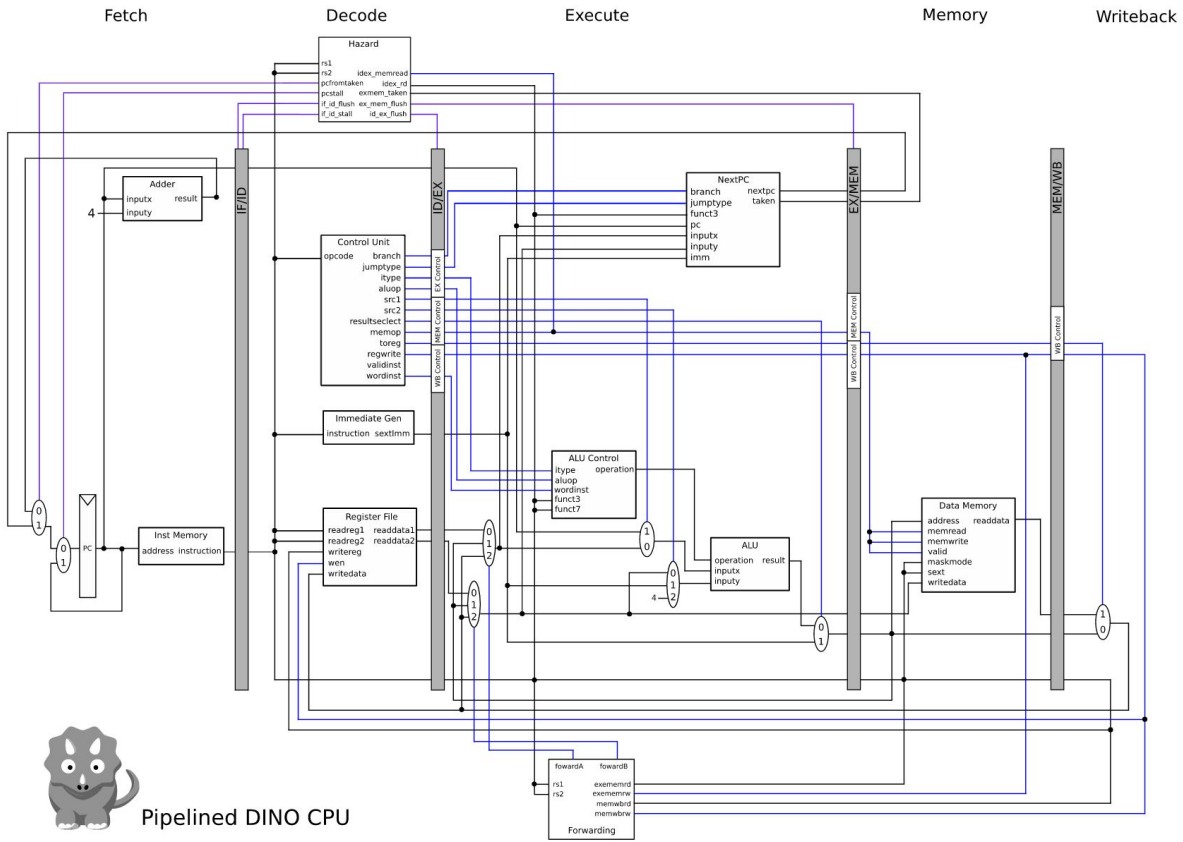
Example 2: cycle 4



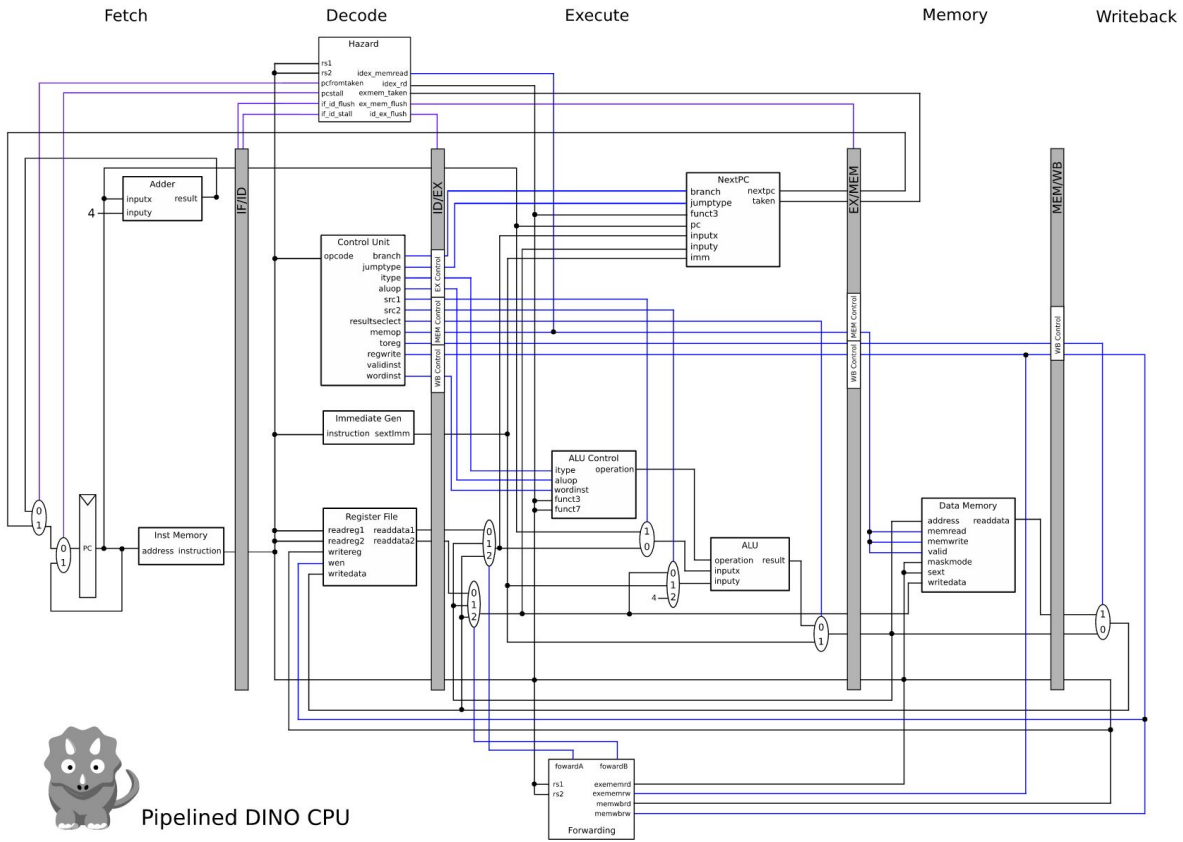
Example 2: cycle 5



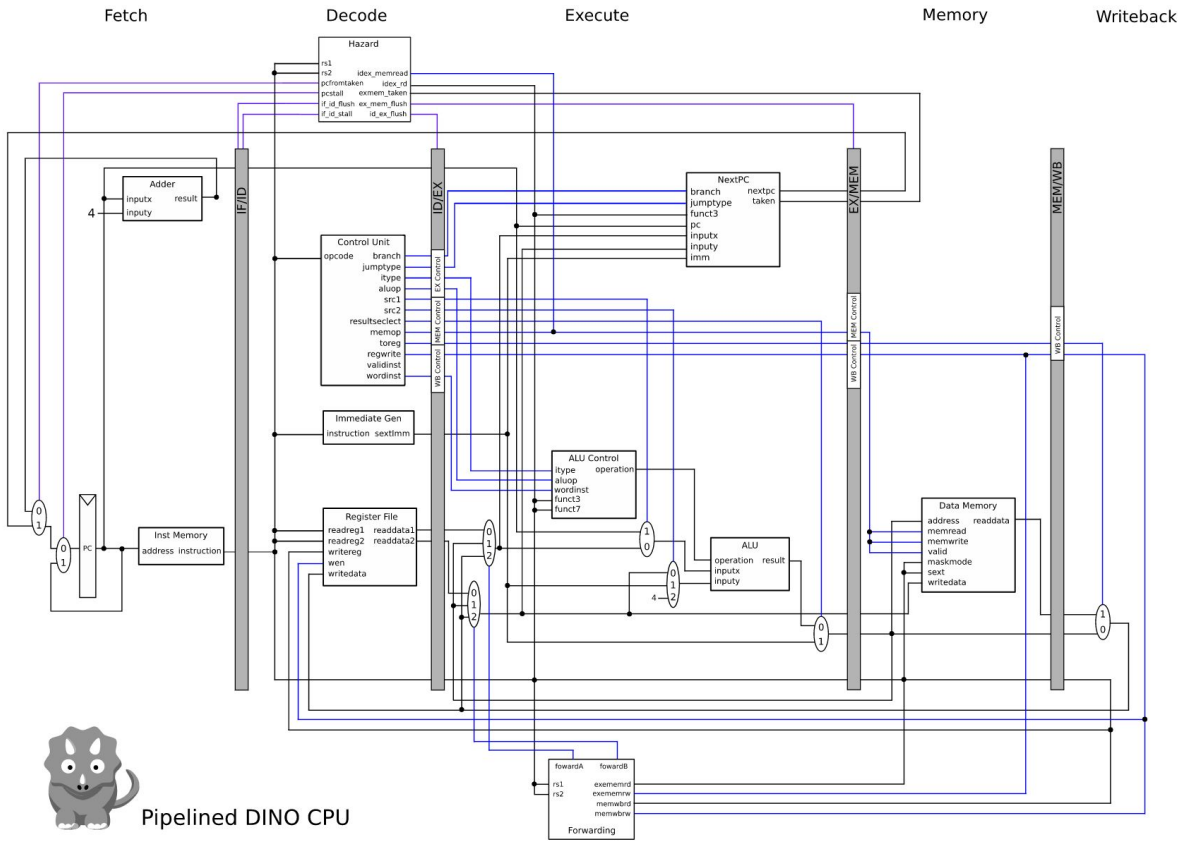
Example 2: cycle 6



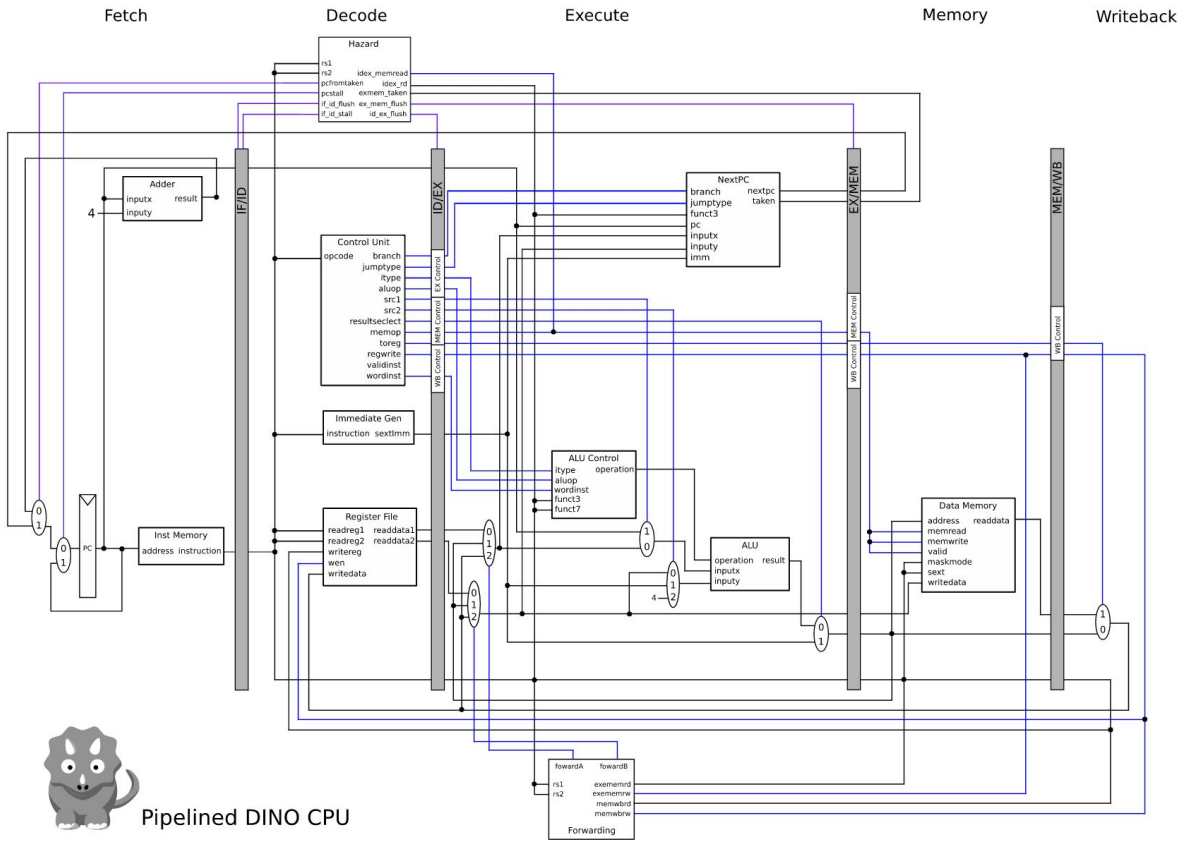
Example 3: cycle 1



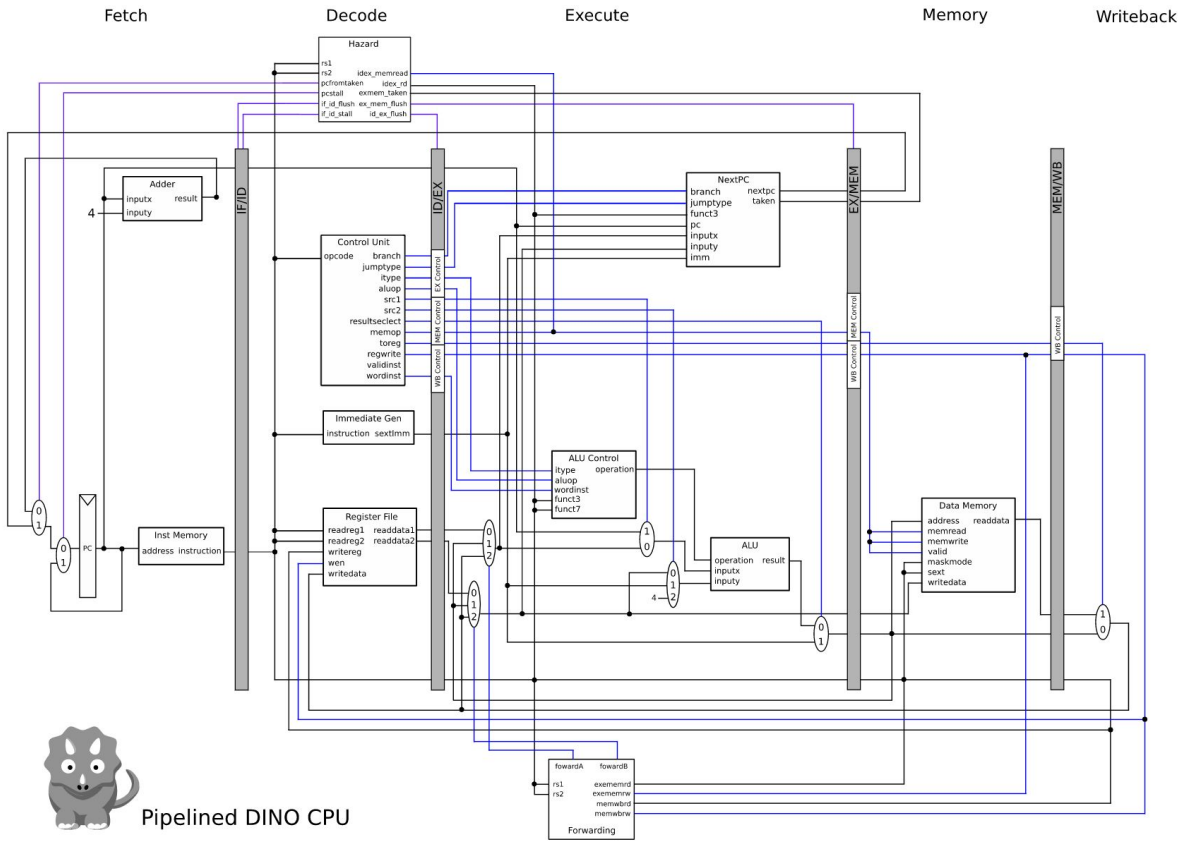
Example 3: cycle 2



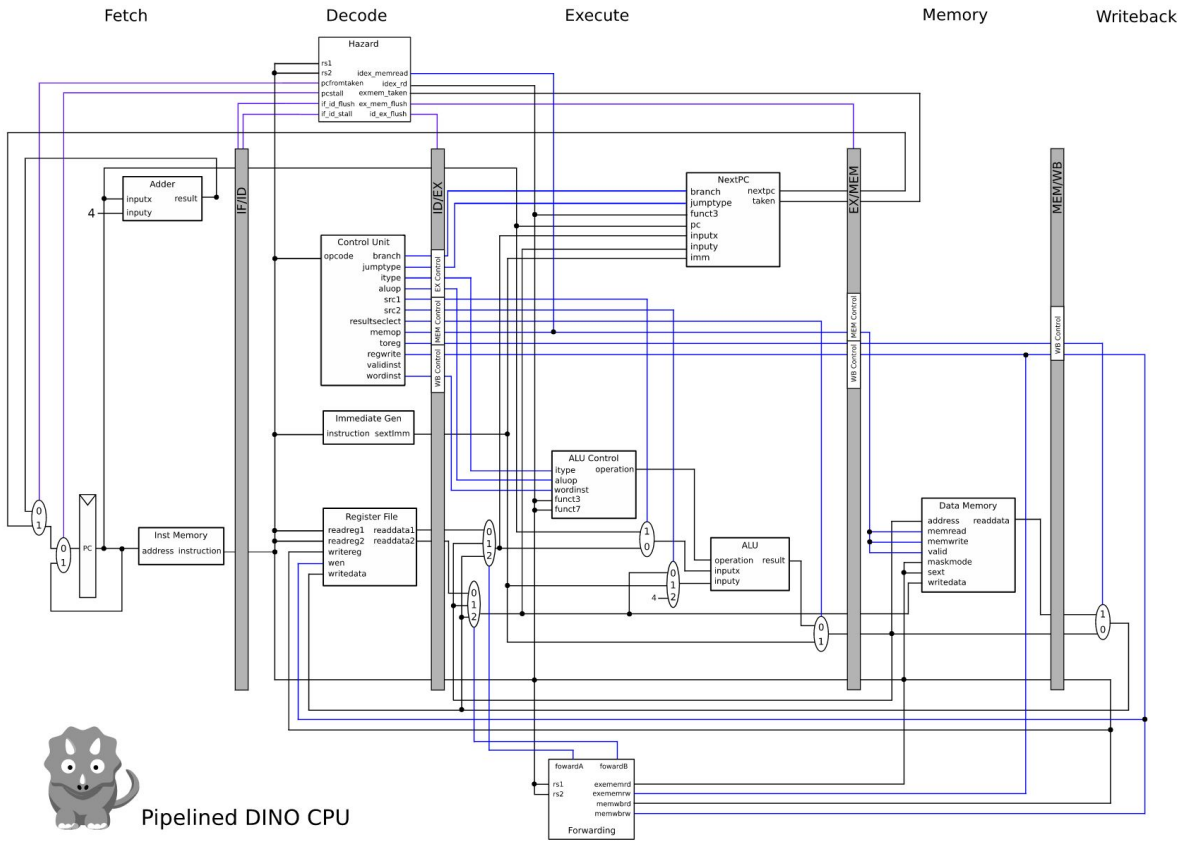
Example 3: cycle 3



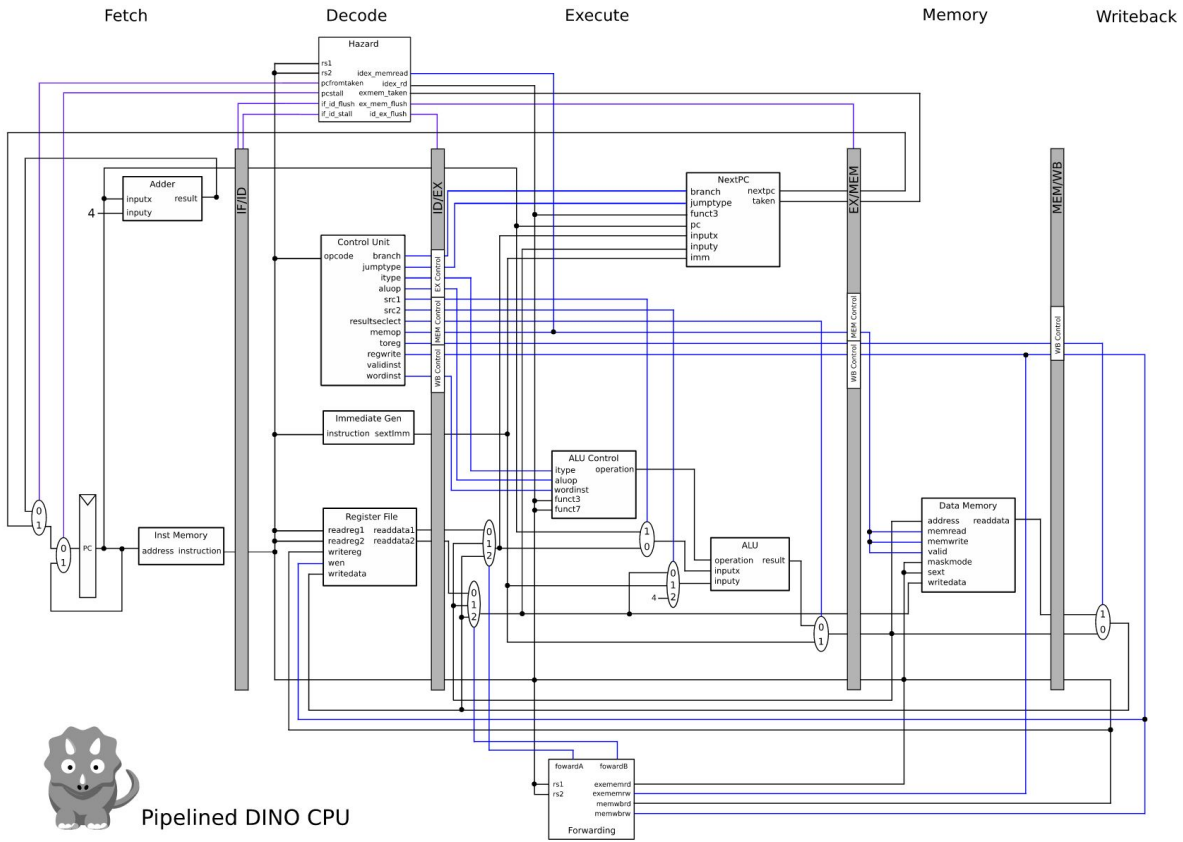
Example 3: cycle 4



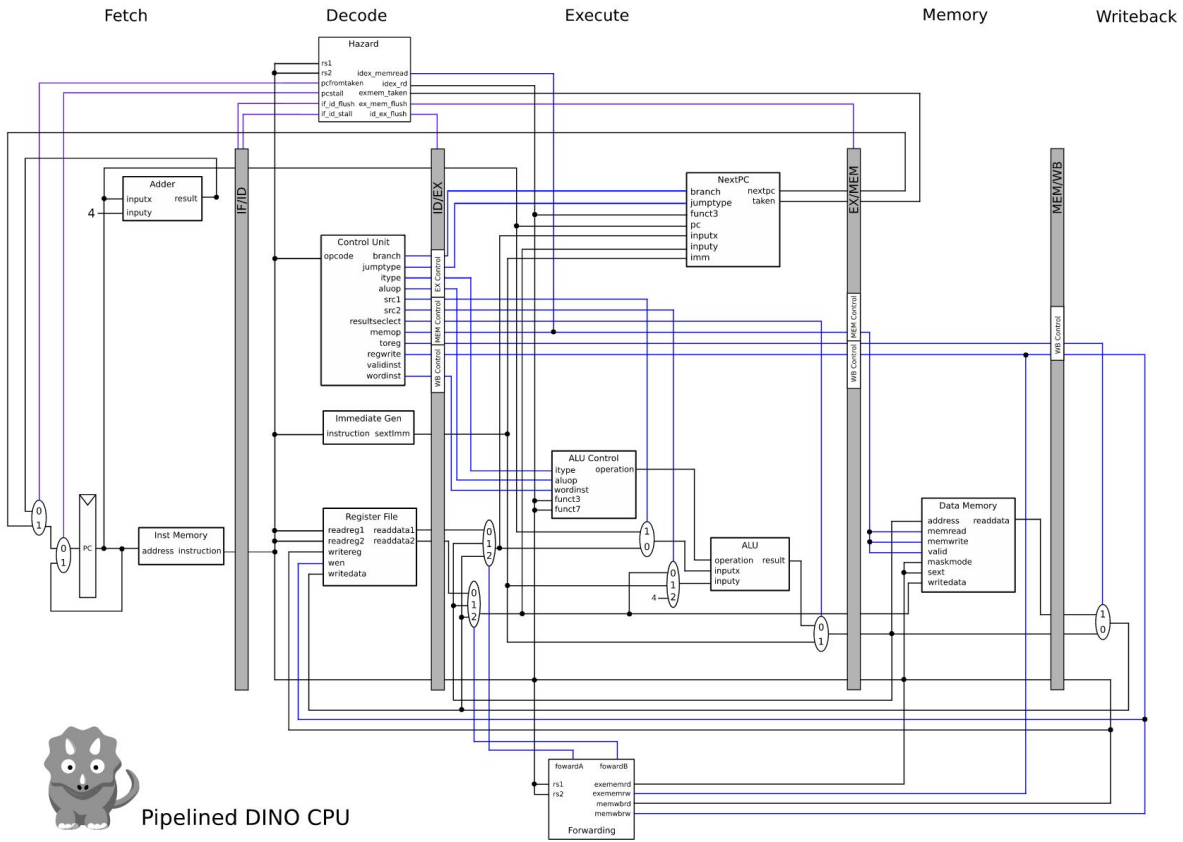
Example 3: cycle 5



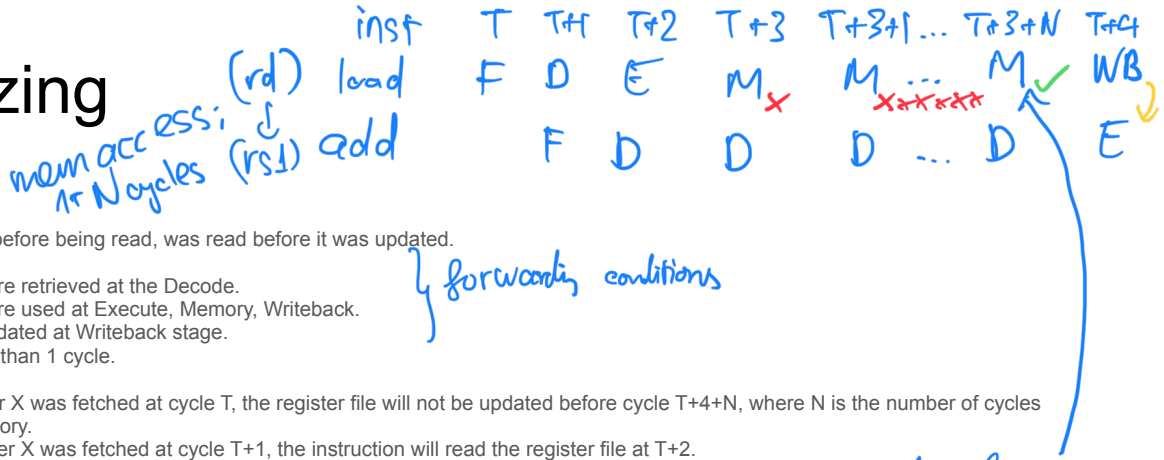
Example 3: cycle 6



Example 3: cycle 7



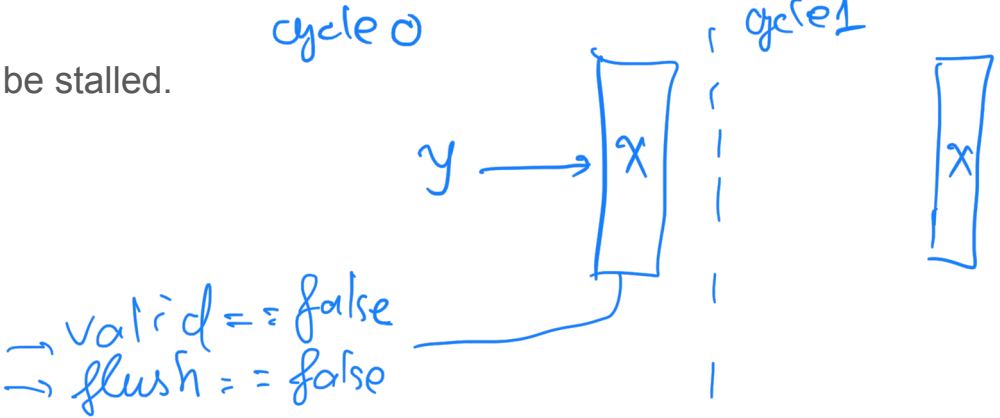
Stalling/Bubbling/Freezing



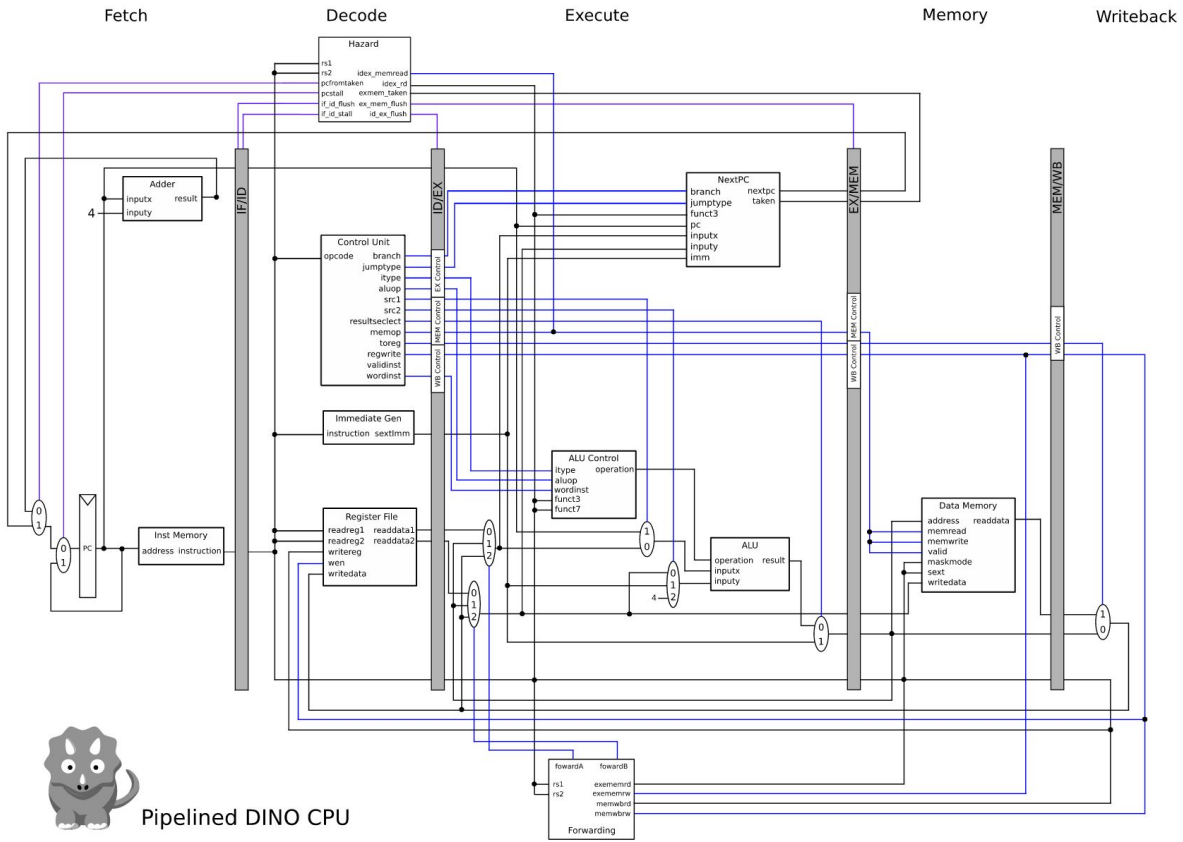
- Problem: read-after-write (true dependency)
  - A source register, which should have been updated before being read, was read before it was updated.
  - In 5-stage pipelined CPU:
    - The values of source registers (rs1, rs2) are retrieved at the Decode.
    - The values of source registers (rs1, rs2) are used at Execute, Memory, Writeback.
    - The value of destination register (rd) is updated at Writeback stage.
    - New condition: Memory stage takes more than 1 cycle.
    - Problem occurs when, for example,
      - If instruction K writing to register X was fetched at cycle T, the register file will not be updated before cycle T+4+N, where N is the number of cycles required to load data from memory.
      - If instruction K+1 reading register X was fetched at cycle T+1, the instruction will read the register file at T+2.
      - If instruction K+2 reading register X was fetched at cycle T+2, the instruction will read the register file at T+3.
- Solution:
  - Doing nothing
    - The CPU will be kept busy, but functionally incorrect.
  - Stalling: don't move the instruction K+1 (and later instructions) out of the Fetch stage until T+N.
    - Functionally correct, but the CPU won't do anything for 2 cycles
  - Forwarding from memory stage to execute stage: technically possible if Memory latency is 0 cycles; impossible otherwise.
- Why NOT forwarding from memory stage to execute stage?
  - Memory accesses are expensive (a typical cycle is a few nanoseconds, a typical memory access costs microseconds)
    - Nanoseconds vs microseconds: https://www.youtube.com/watch?v=9eyFDBPk4Yw
  - Being functionally correct is required for general purpose CPUs.
  - Not all instructions require memory access.
  - CPU frequency matters.
- Why stalling?
  - Forwarding is not possible.

How to stall pipelined CPU?

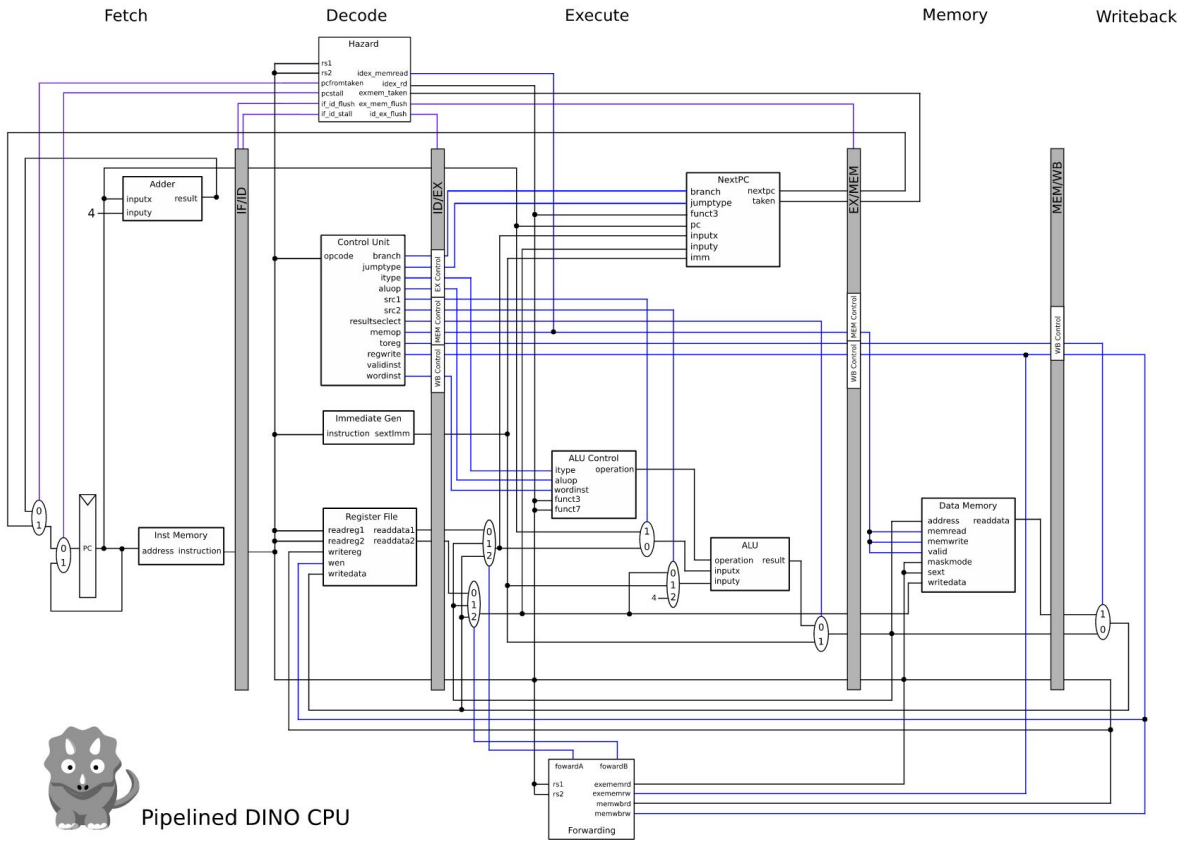
- When a stage register `.io.valid === false.B` and `.io.flush === false.B`, the registers won't be updated.
- The stalling logic is in `src/main/scala/components/hazard.scala` (also the place to implement flushing logic).
- Notes:
  - When the hazard is detected matters, it affects when the CPU makes a decision on stalling on the next cycle.
  - Not all stage registers should be stalled.



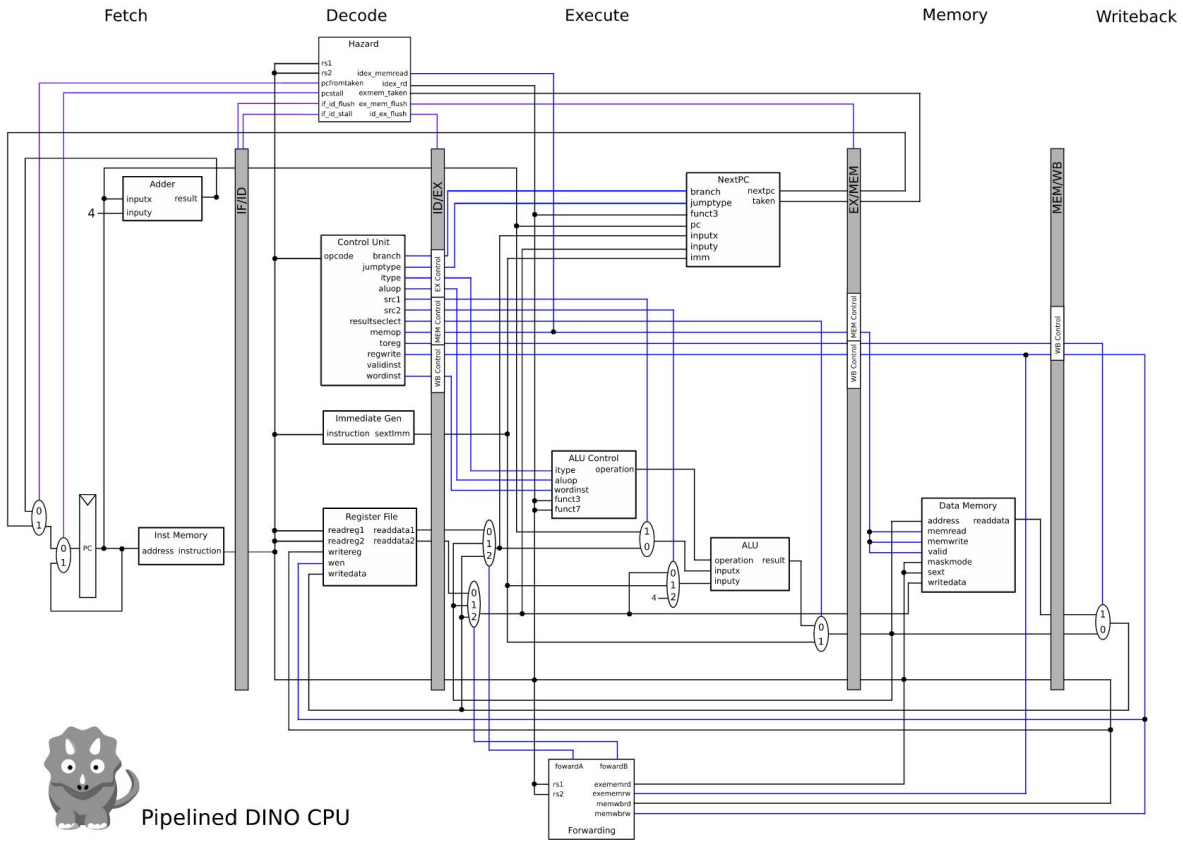
Example 4: cycle 1



Example 4: cycle 2

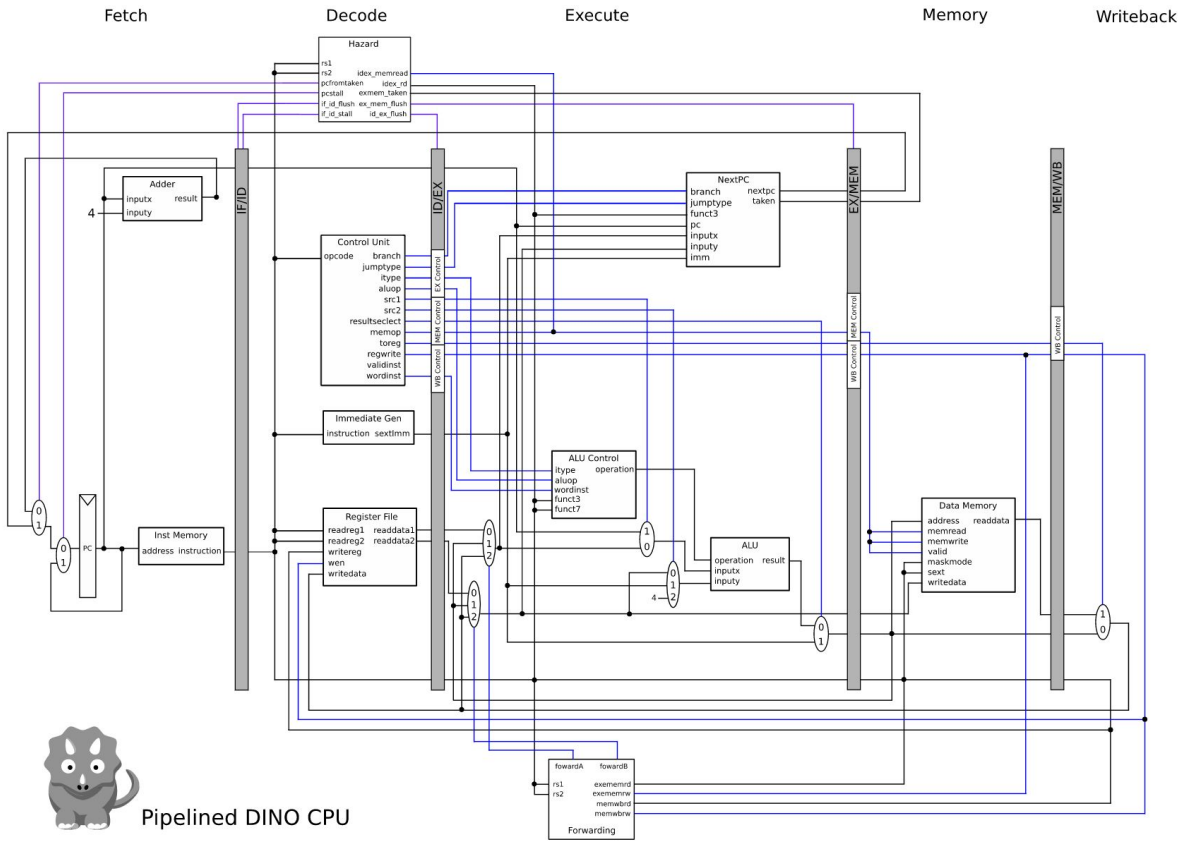


Example 4: cycle 3

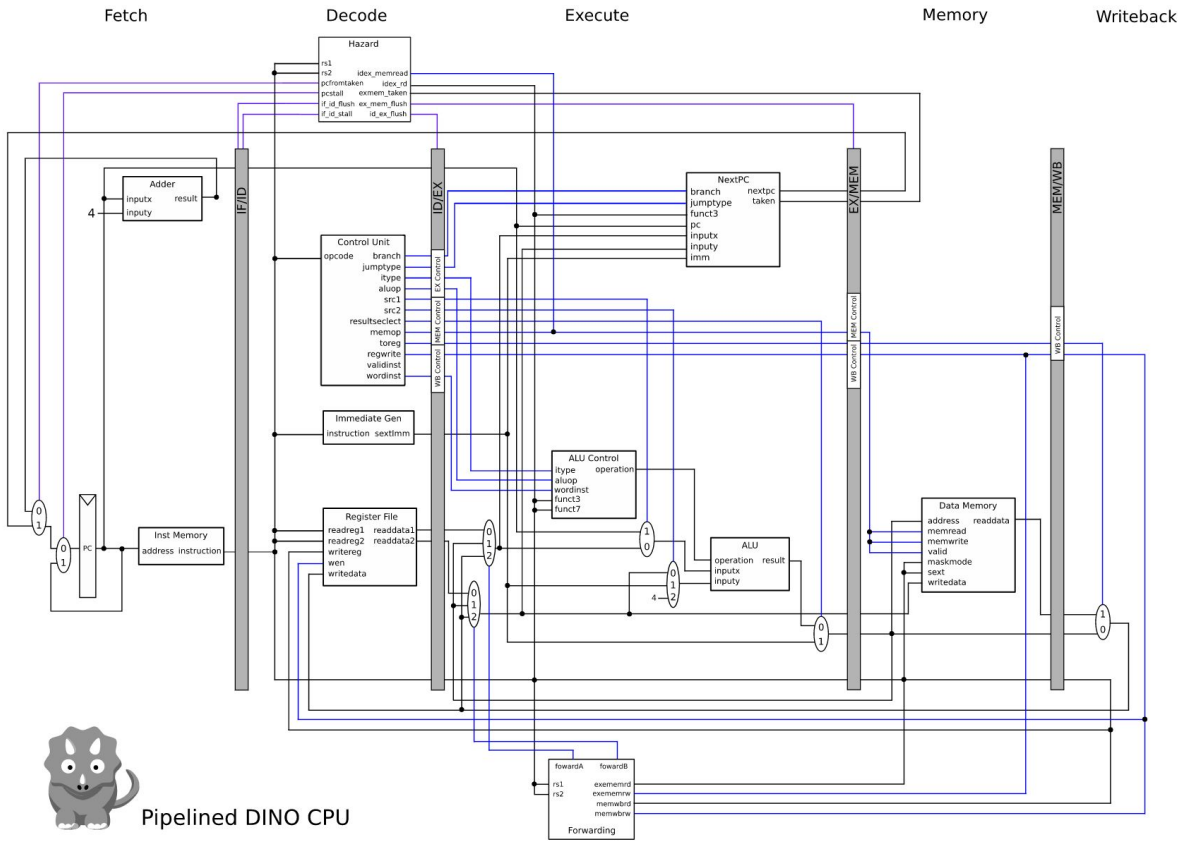


Pipelined DINO CPU

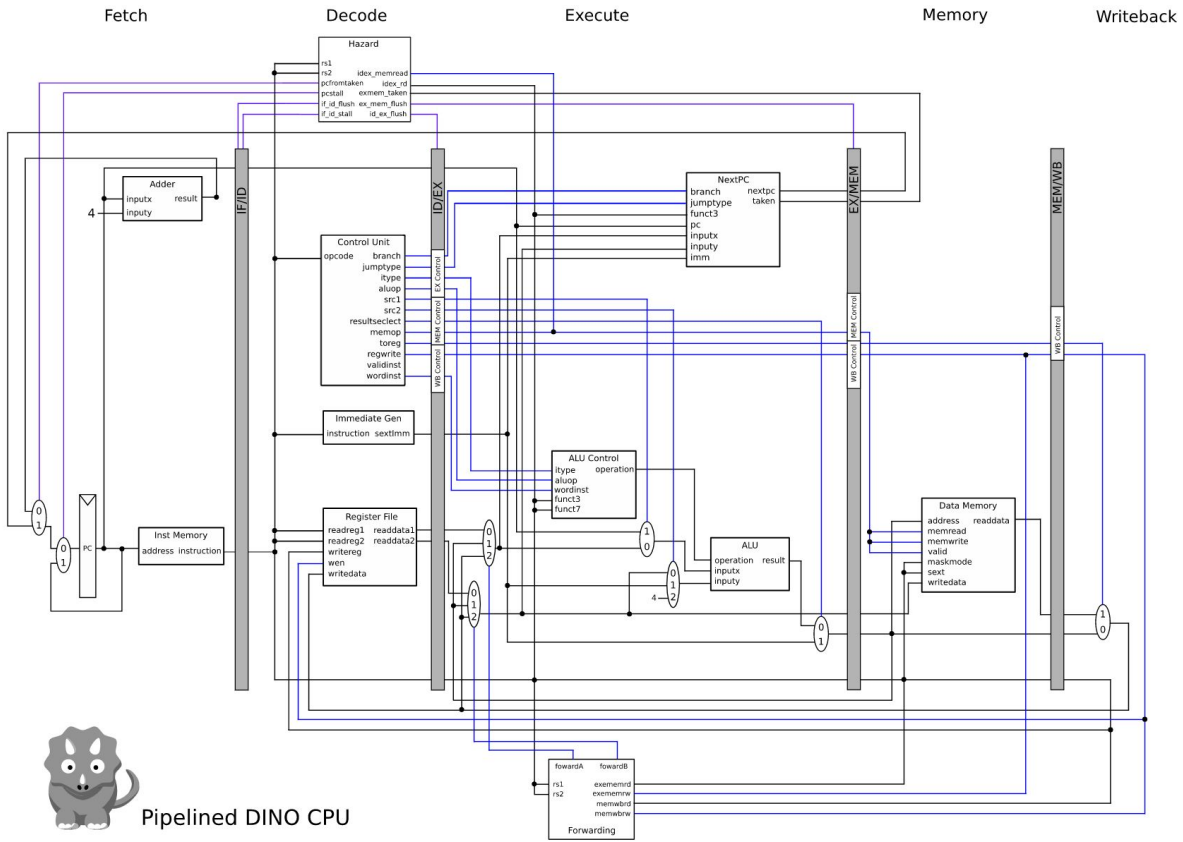
Example 4: cycle 4



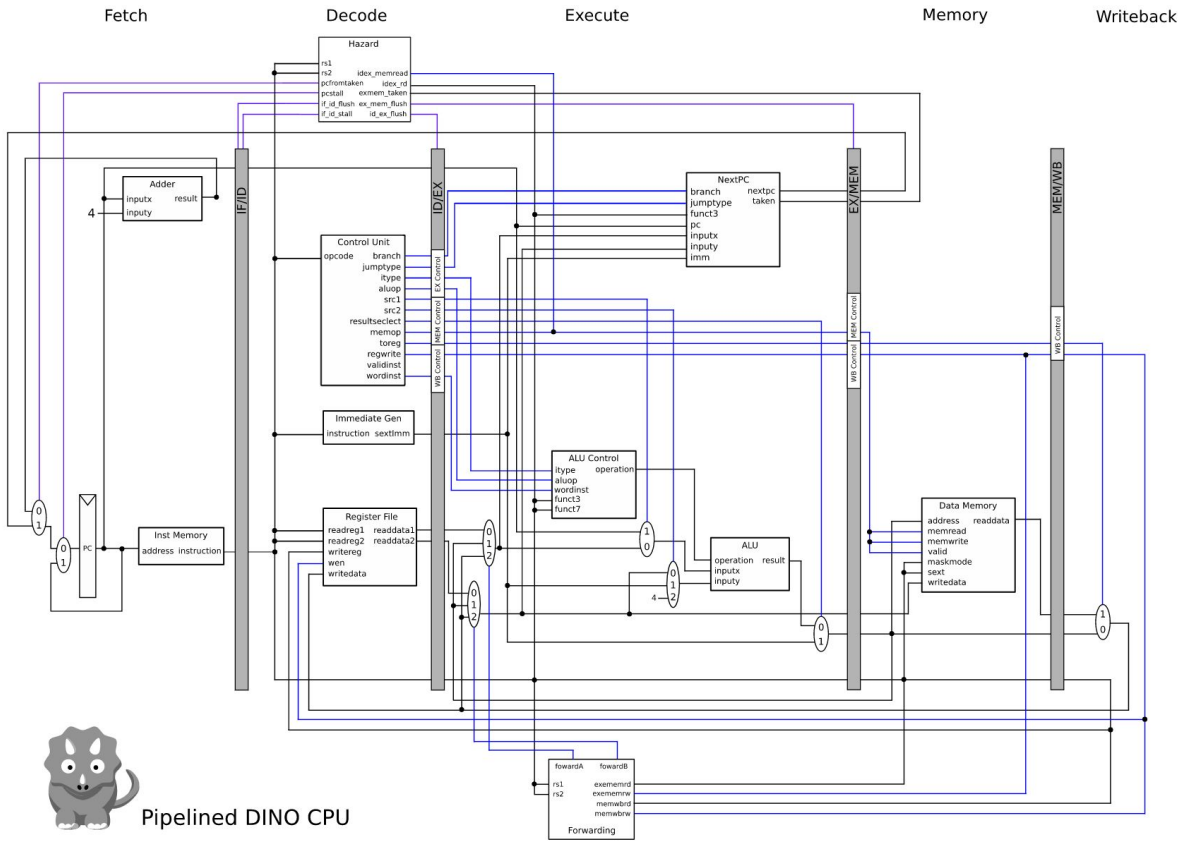
Example 4: cycle 5



Example 4: cycle 6



Example 4: cycle 7



Branch/Jump Instructions

- Branch instructions
  - Typically, if the branch condition is true, the next PC won’t be PC + 4 (branch taken).
  - And if the branch condition is false, the next PC is PC + 4 (branch is not taken).
- Jump instructions
  - The next PC will definitely not be PC + 4.

Branch/Jump Instructions

- Problem:
  - The condition of the branch (thus the next PC) is not known until the Execute stage.
  - The next PC from the jump instruction is not known until the Execute stage.
- Possible solutions:
  - Stalling:
    - Stall the pipeline until the branch/jump is resolved.
    - Cost: 3 cycles.
  - Speculative execute a branch/jump:
    - Keep putting more instructions into the pipeline as if the branch is not taken or the jump does not occur.
    - By the time the next PC is resolved, 3 more instructions (in Fetch, Decode, Execute) are already in the pipeline.
    - Have to remove the incorrect instructions from the pipelined (i.e., flushing).
    - Cost:
      - Branch mispredictions: 3 cycles
      - Branch correct predictions: 0 cycles
      - Jumps: 3 cycles
      - So, the average cost cannot be more than 3 cycles
- Why speculative execution?
  - Average cost is always better than stalling.
  - In a real system, the pipeline is typically longer, so the cost is much higher; but modern branch predictor is typically very accurate (much higher than 90%).

How to flush incorrect instructions out of the pipeline?

- When a stage register `.io.flush === true.B`, the registers will be erased.
- Notes:
  - Not all stage registers will be flushed.
  - `jal` and `jalr` still need the writeback stage.

Debugging pipelined CPU

- PC outputted by the single stepper is that of the instruction in the Fetch stage.
- The code is not executed sequentially.
- There are 5 instructions in the pipelined at the same time; instruction PC can be used to determine which instruction is in which stage for debugging (i.e. printf, step debugger).
- Full applications (fibonacci, naturalsum, multiplier, divider) correct traces are provided,
  - https://jlpteaching.github.io/comparch/modules/dino%20cpu/assignment3/#testing-your-hazard-detection-unit