**A Basic MIPS Implementation**

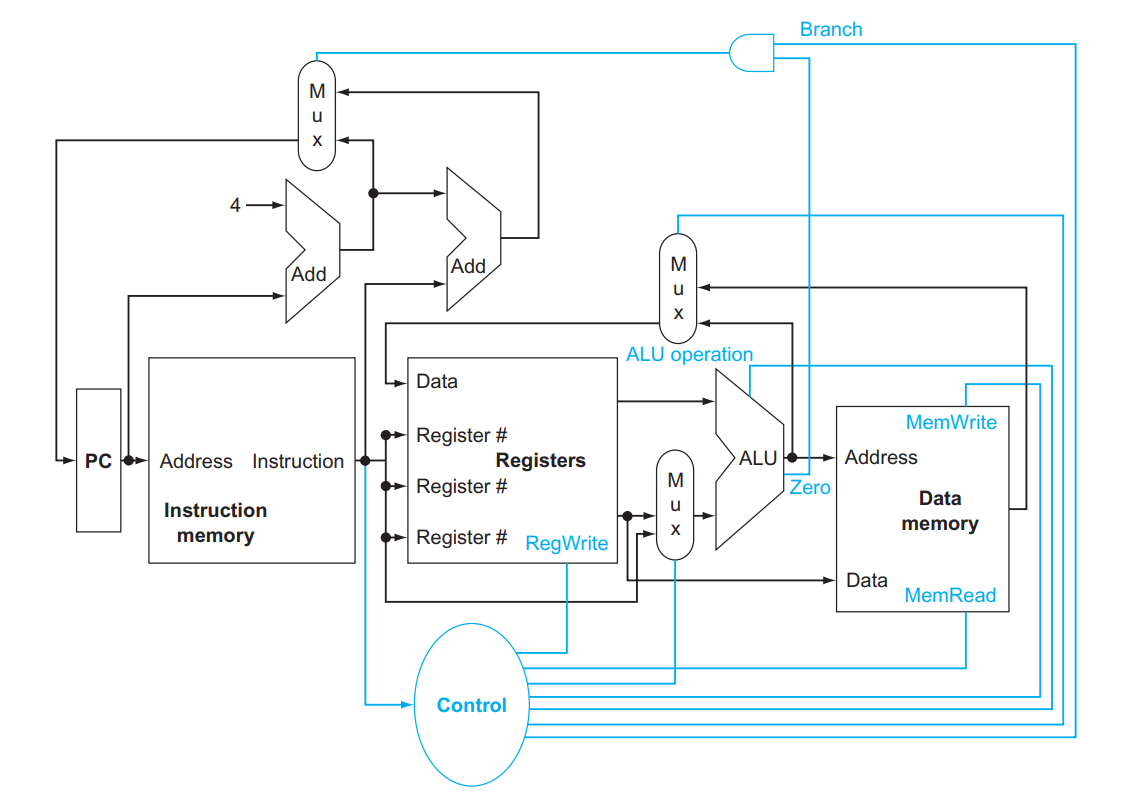
We will be examining an implementation that includes a subset of the core MIPS instruction set

1. The memory-reference instructions load word (lw) and store word (sw)
2. The arithmetic-logical instructions add, sub, AND, OR, and slt
3. The instructions branch equal (beq) and jump (j), which we add last.

For every instruction, the first two steps are identical:

1. **IF -** Send the program counter (PC) to the memory that contains the code and fetch the instruction from that memory.
2. **ID -** Read one or two registers, using fields of the instruction to select the registers to read. For the load word instruction, we need to read only one register, but most other instructions require reading two registers.

After these two steps, the actions required to complete the instruction depend on the instruction class.



**Combinational Element**: An operational element, such as an AND gate or an ALU.

**State Element**: A memory element, such as a register or a memory.

**Clocking Methodology**: The approach used to determine when data is valid and stable relative to the clock.

**Edge-Triggered Clocking**: A clocking scheme in which all state changes occur on a clock edge.

**Control signal**: A signal used for multiplexor selection or for directing the operation of a functional unit.

**Data signal**: Contains information that is operated on by a functional unit.

**Asserted**: The signal is logically high or true.

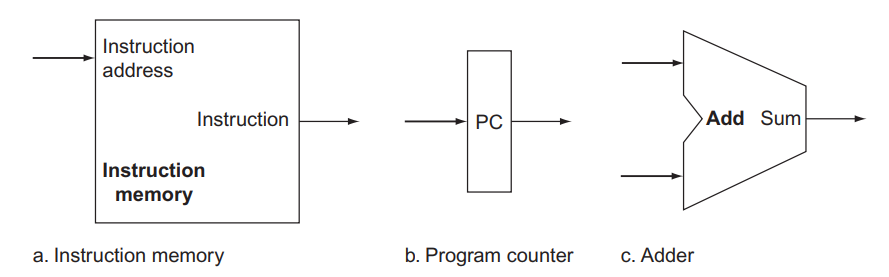
**De-asserted**: The signal is logically low or false.

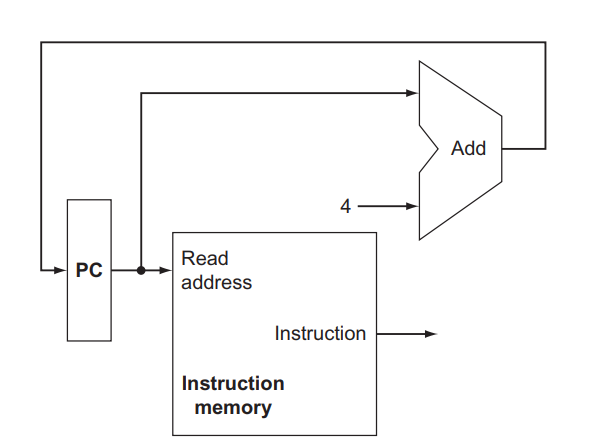
**Datapath element**: A unit used to operate on or hold data within a processor. In the MIPS implementation, the data-path elements include the instruction and data memories, the register file, the ALU, and adders.

**Memory unit**: To store the instructions of a program and supply instructions given an address.

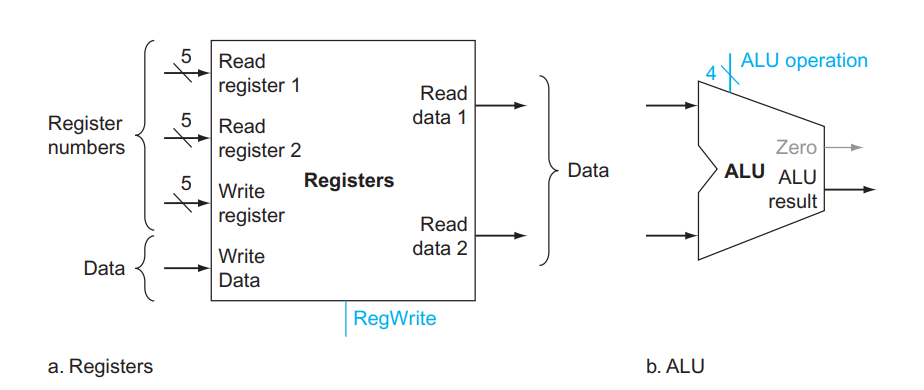
**Program counter (PC)**: The register containing the address of the instruction in the program being executed.

**Adder**: To increment the PC to the address of the next instruction.

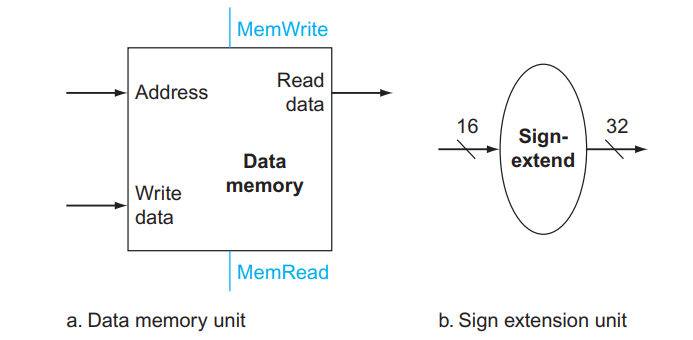




**Register file**: A state element that consists of a set of registers that can be read and written by supplying a register number to be accessed.



**Sign-extend**: A data-path item used to increase the size of a data item by replicating the high-order sign bit of the original data item in the highorder bits of the larger, destination data item.

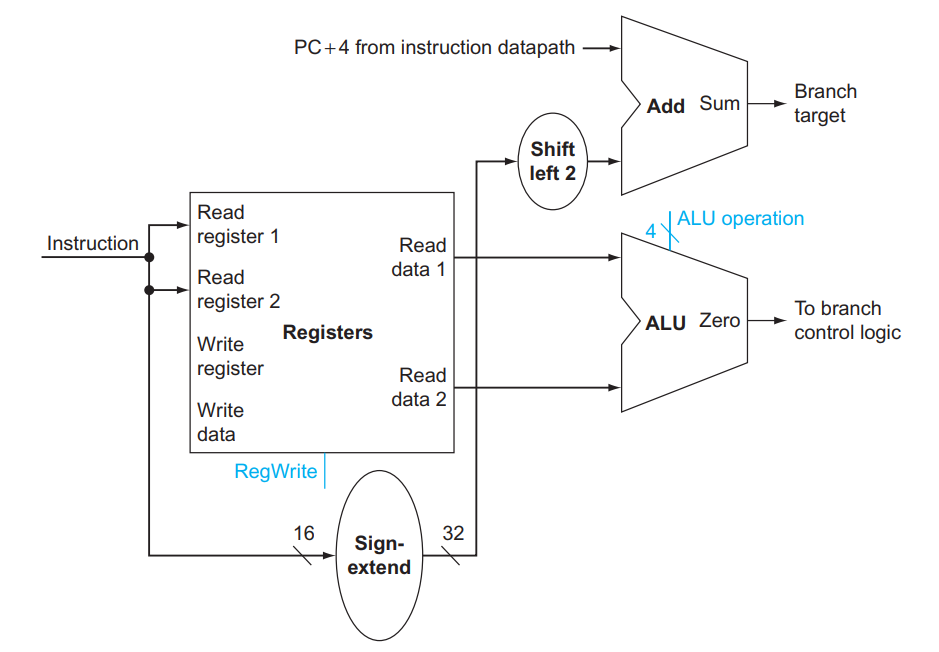


**Branch Target Address**: The address specified in a branch, which becomes the new program counter (PC) if the branch is taken. In the MIPS architecture the branch target is given by the sum of the offset field of the instruction and the address of the instruction following the branch.

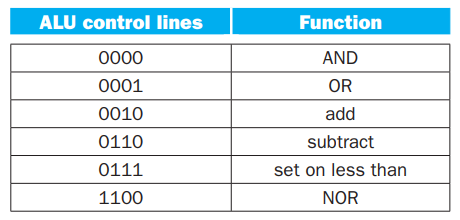
**Branch Taken**: A branch where the branch condition is satisfied and the program counter (PC) becomes the branch target. All unconditional jumps are taken branches.

**Branch Not Taken** or (untaken branch): A branch where the branch condition is false and the program counter (PC) becomes the address of the instruction that sequentially follows the branch.

**Branch**: A type of branch where the instruction immediately following the branch is always executed, independent of whether the branch condition is true or false.

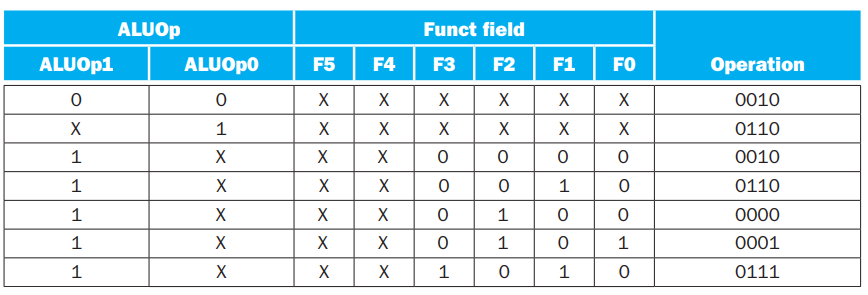


**ALU Control**

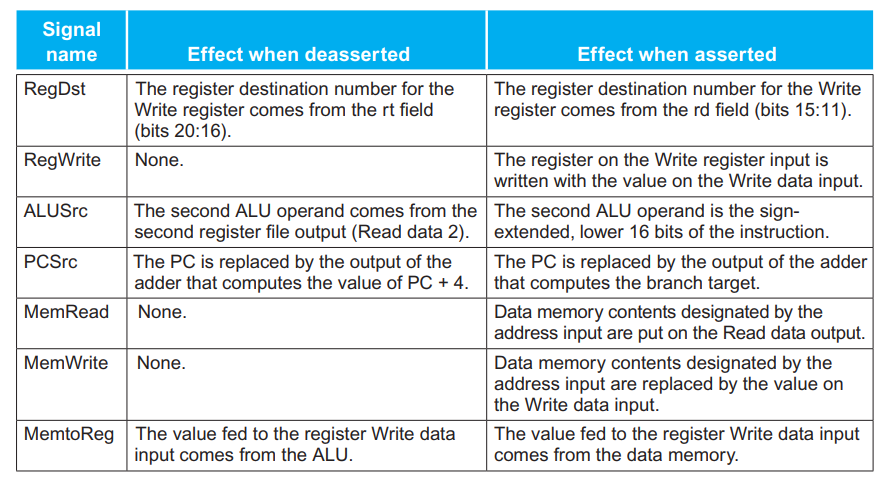


For branch equal, the ALU must perform a subtraction.

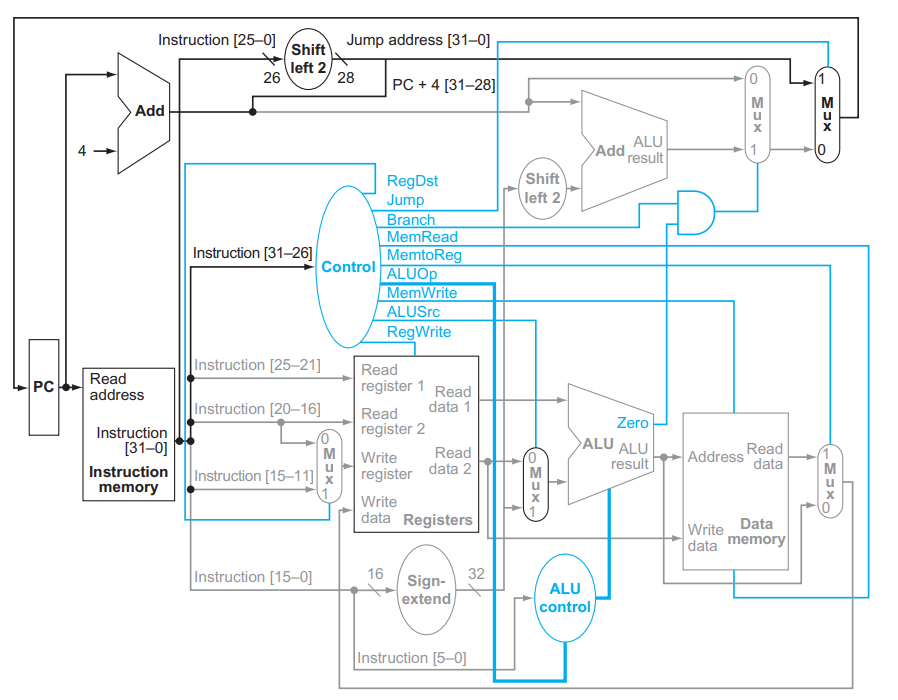
**Main Control Unit**



**Effect of Control Lines** (7 here + 2 from ALUop)

****





**Pipelining**: An implementation technique in which multiple instructions are overlapped in execution, much like an assembly line. All steps—called stages in pipelining—are operating concurrently. As long as we have separate resources for each stage, we can pipeline the tasks. The reason pipelining is faster for many loads is that everything is working in parallel, so more loads are finished per hour.

MIPS instructions classically take five steps:

1. **IF** Fetch instruction from memory.

2. **ID** Read registers while decoding the instruction. The regular format of MIPS instructions allows reading and decoding to occur simultaneously.

3. **EX** Execute the operation or calculate an address.

4. **MEM** Access an operand in data memory.

5. **WB** Write the result into a register.

Pipelining improves performance by increasing instruction throughput, as opposed to decreasing the execution time of an individual instruction, but instruction throughput is the important metric because real programs execute billions of instructions.

Why MIPS is designed for pipeline execution:

1. All MIPS instructions are the same length. This restriction makes it much easier to fetch instructions in the first pipeline stage and to decode them in the second stage.
2. MIPS has only a few instruction formats, with the source register fields being located in the same place in each instruction.
3. Memory operands only appear in loads or stores in MIPS.
4. Operands must be aligned in memory.

There are situations in pipelining when the next instruction cannot execute in the following clock cycle. These events are called **hazards**, and there are three different types.

**Structural Hazard**: When a planned instruction cannot execute in the proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

**Data Hazard**: Also called a pipeline data hazard. When a planned instruction cannot execute in the proper clock cycle because data that is needed to execute the instruction is not yet available.

**Forwarding**: Also called bypassing. A method of resolving a data hazard by retrieving the missing data element from internal buffers rather than waiting for it to arrive from programmer-visible registers or memory.

**Load-use Data Hazard**: A specific form of data hazard in which the data being loaded by a load instruction has not yet become available when it is needed by another instruction.

**Pipeline stall**: Also called **bubble**. A stall initiated in order to resolve a hazard.

**Control Hazard**: Also called branch hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.

Computers use prediction to handle branches. One simple approach is to predict always that branches will be untaken. When you’re right, the pipeline proceeds at full speed. Only when branches are taken does the pipeline stall.

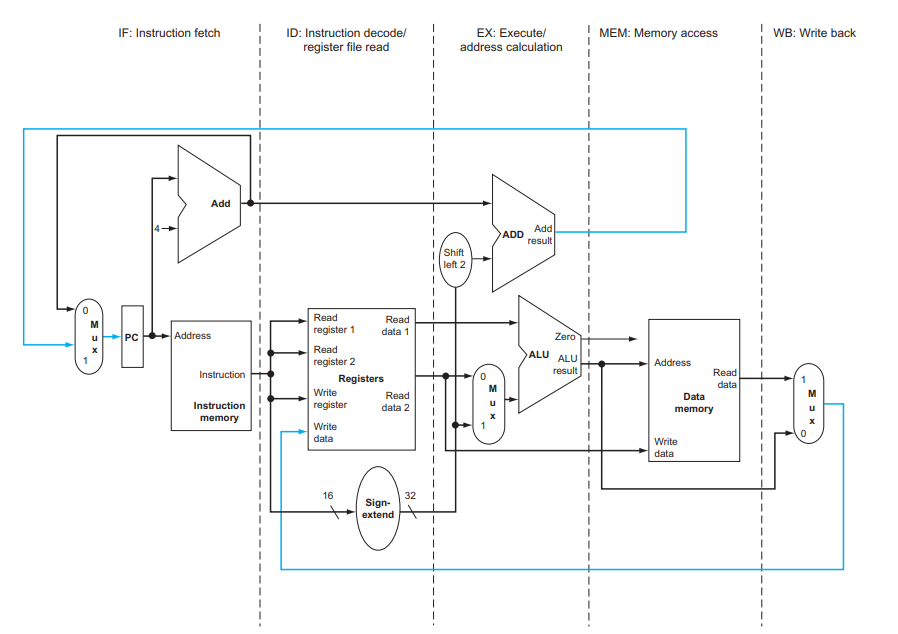
**Branch Prediction**: A method of resolving a branch hazard that assumes a given outcome for the branch and proceeds from that assumption rather than waiting to ascertain the actual outcome.

One popular approach to dynamic prediction of branches is keeping a history for each branch as taken or untaken, and then using the recent past behaviour to predict the future.

There is a third approach to the control hazard, called **delayed decision**. Called the delayed branch in computers, and mentioned above, this is the solution actually used by the MIPS architecture. The delayed branch always executes the next sequential instruction, with the branch taking place after that one instruction delay.

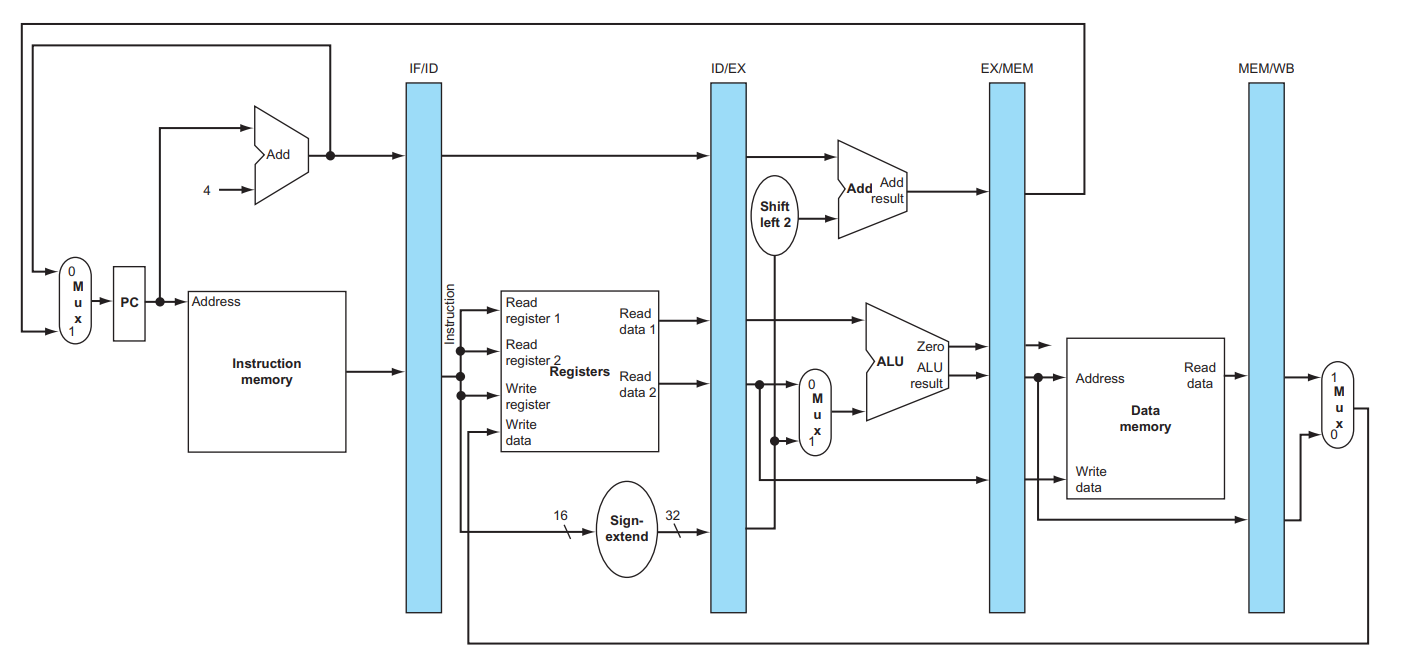
Pipelining is a technique that exploits **parallelism** among the instructions in a sequential instruction stream.

**Latency** (pipeline): The number of stages in a pipeline or the number of stages between two instructions during execution.

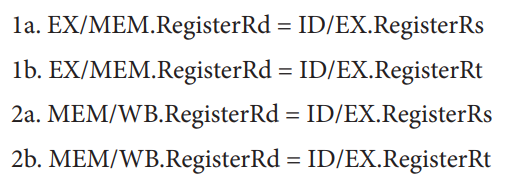


Instructions and data move generally from left to right through the five stages as they complete execution. There are, however, two exceptions to this left -to-right flow of instructions:

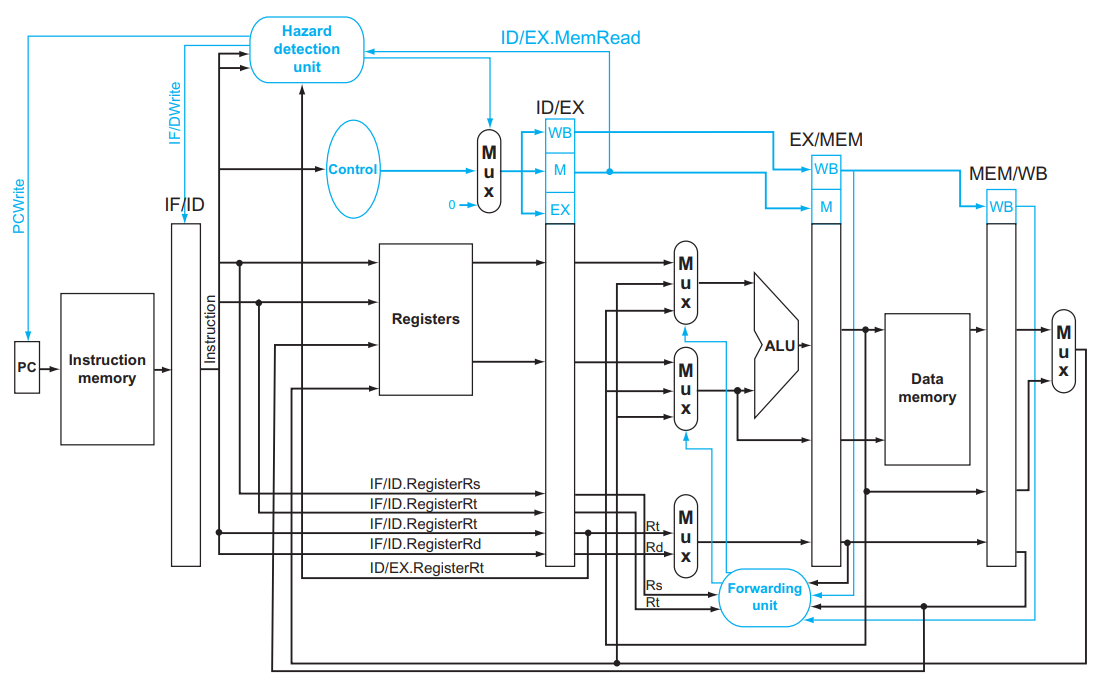
1. The write-back stage, which places the result back into the register file in the middle of the data-path.
2. The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stag.



Two pairs of hazard conditions:



**Nop**: An instruction that does no operation to change state.



**Instruction-Level Parallelism**: The parallelism among instructions.

**Multiple issue**: A scheme whereby multiple instructions are launched in one clock cycle.

**Static Multiple Issue**: An approach to implementing a multiple-issue processor where many decisions are made by the compiler before execution.

**Dynamic Multiple Issue**: An approach to implementing a multiple-issue processor where many decisions are made during execution by the processor.

**Issue slots**: The positions from which instructions could issue in a given clock cycle; by analogy, these correspond to positions at the starting blocks for a sprint.

**Speculation**: An approach whereby the compiler or processor guesses the outcome of an instruction to remove it as a dependence in executing other instructions.

**Issue Packet**: The set of instructions that issues together in one clock cycle; the packet may be determined statically by the compiler or dynamically by the processor.

**Very Long Instruction Word (VLIW)**: A style of instruction set architecture that launches many operations that are defined to be independent in a single wide instruction, typically with many separate opcode fields.

**Use Latency**: Number of clock cycles between a load instruction and an instruction that can use the result of the load without stalling the pipeline.

**Loop Unrolling**: A technique to get more performance from loops that access arrays, in which multiple copies of the loop body are made and instructions from different iterations are scheduled together.

**Register Renaming**: The renaming of registers by the compiler or hardware to remove anti-dependences.

**Anti-dependence**: Also called name dependence. An ordering forced by the reuse of a name, typically a register, rather than by a true dependence that carries a value between two instructions.

**Superscalar**: An advanced pipelining technique that enables the processor to execute more than one instruction per clock cycle by selecting them during execution.

**Dynamic Pipeline Scheduling**: Hardware support for reordering the order of instruction execution so as to avoid stalls.

**Commit Unit**: The unit in a dynamic or out-of-order execution pipeline that decides when it is safe to release the result of an operation to programmer visible registers and memory.

**Reservation Station**: A buffer within a functional unit that holds the operands and the operation.

**Reorder Buffer**: The buffer that holds results in a dynamically scheduled processor until it is safe to store the results to memory or a register.

**Out-Of-Order Execution**: A situation in pipelined execution when an instruction blocked from executing does not cause the following instructions to wait.

**In-Order Commit**: A commit in which the results of pipelined execution are written to the programmer visible state in the same order that instructions are fetched.

