Pipelining:
Instruction-Level Parallelism

Instruction Level Parallelism is a method of overlapping the execution of multiple instructions. The major benefit of any form of parallel processing is a reduction in the clock rate dependency. In contrast, serial processing techniques are wholly dependant upon clock rate. Since clock rate is ultimately limited in the physical world, parallel processing techniques are requisite for further advancement in CPU technology.

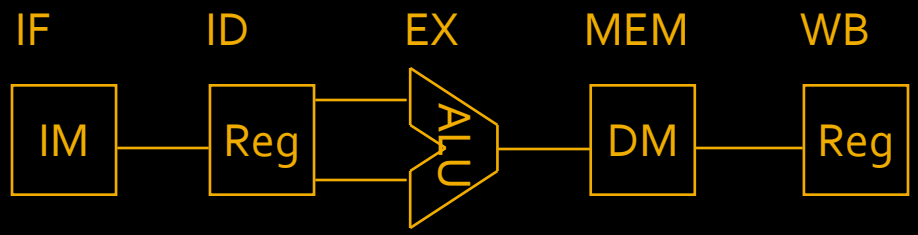
Single Cycle implementation is an example of serial processing techniques where each instruction is processed sequentially.
Pipelining is an advancement on these older designs in which instruction level parallel processing is achieved.

Review: MIPS Instructions

MIPS instruction steps are classically defined as follows:

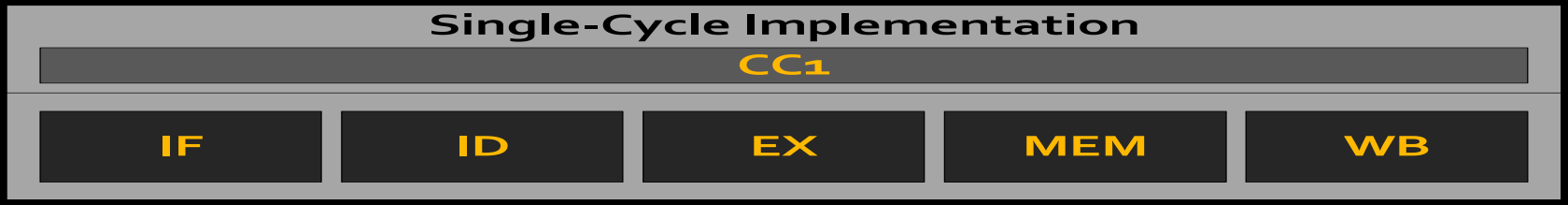
1. IF: Fetch instruction from memory.
2. ID: Read registers while simultaneously decoding the instruction.
3. EX: Execute the operation or calculate an address.
4. MEM: Access an operand in data memory.
5. WB: Write the result back into a register.

These steps are associated with modular functional units found in the data path such as the IM (Instruction Memory), Reg (Register File), ALU (Arithmetic Logic Unit) and DM (Data Memory). As such they can be graphically represented in the following manner.



Review: Single Cycle Implementation

The clock cycle in a Single-Cycle implementation is defined by the amount of time required to process the largest possible instruction.



This implementation is inefficient because simple instructions (branch equals) will take just as much time to process as a complex instruction (load/store).

The performance of a Single Cycle implementation is entirely based on the clock rate and the relative speeds of the individual hardware components.

Review: Single Cycle Performance

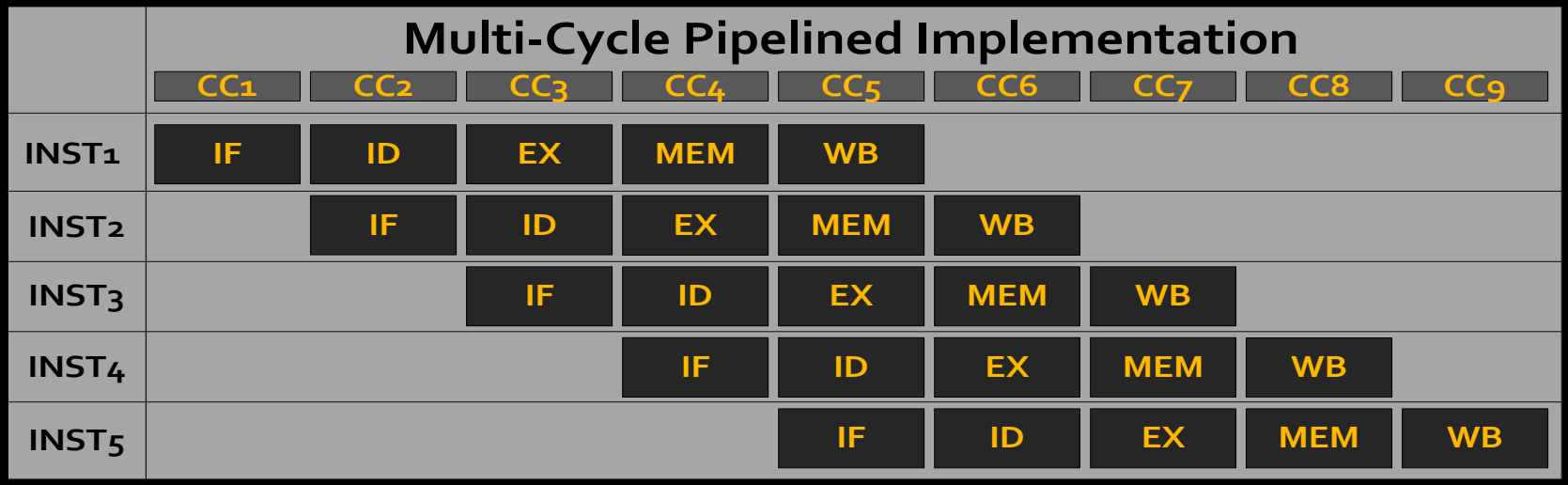
Instruction	IF	ID	EX	MEM	WB	Total
Load (lw)	200ps	100ps	200ps	200ps	100ps	800ps
Store (sw)	200ps	100ps	200ps	200ps		700ps
R-Type (add)	200ps	100ps	200ps		100ps	600ps
Branch(Beq)	200ps	100ps	200ps			500ps

(Instruction/Data) Memory access = 200ps
ALU use: 200ps
Register File access: 100ps

In a Single-Cycle implementation of the above instruction set the duration of a clock cycle would be approximately 800 pico-seconds ($8 * 10^{-12}$ seconds). Thus, the time required to process any 5 instructions would be 4000ps.

Pipelining Instructions

Pipelining allows for concurrent processing of different instructions. When discussing pipelining we refer to the individual instruction steps as stages.



If the five instructions in the diagram were handled serially the collective functional units would require a total of 25 clock cycles to process. However, when a five stage pipeline is used, processing is completed in only 9 clock cycles.

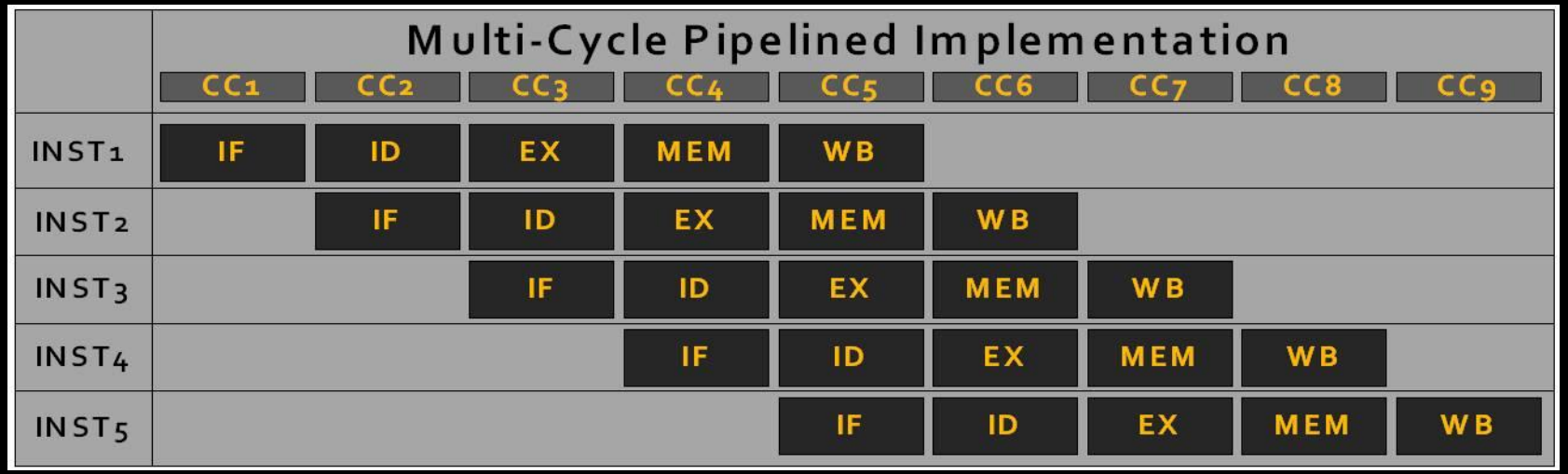
Performance: Pipelining

Instruction	IF	ID	EX	MEM	WB	Total
Load (lw)	200ps	100ps	200ps	200ps	100ps	800ps
Store (sw)	200ps	100ps	200ps	200ps		700ps
R-Type (add)	200ps	100ps	200ps		100ps	600ps
Branch(Beq)	200ps	100ps	200ps			500ps

Consider a pipelined implementation where 5 of instructions are processed. This would require a total of 9 clock cycles with a clock cycle duration of approximately 200ps yielding a processing time of 1800ps.

Performance: Pipelining

While the previous example demonstrates pipeline efficiency it does not illustrate the full potential of a pipelined CPU. With five instructions it takes five cycles for all of the instructions to be inserted into the pipe. This can be thought of as the start-up time. There is also a wind-down time where the instructions are emptied from the processor. In both instances the pipelined architecture is not as efficient as it could be since the pipe is not full.



Performance : Pipelined

Pipelined instructions cannot skip stages. For example an add instruction will wait during the mem stage, and a store instruction would wait during the write back stage, allowing other instructions to complete before proceeding.

Instruction	IF	ID	EX	MEM	WB	Total
Load (lw)	200ps	100ps	200ps	200ps	100ps	800ps
Store (sw)	200ps	100ps	200ps	200ps		700ps
R-Type (add)	200ps	100ps	200ps		100ps	600ps
Branch(Beq)	200ps	100ps	200ps			500ps

Functional units in a pipelined data path are not reused by the same instruction so it is necessary to have two adders and an ALU as well as two memory units as in single-cycle implementation.

Performance : Pipelined

Given a larger number of instructions the start-up and wind-down time become insignificant since the pipe is constantly full. Under ideal conditions the **speed-up** from pipelining is **approximately equal to the number of pipe stages** (which directly correlates to a decrease in the time-between-instructions); a five-stage pipeline is nearly five times faster. One way to measure the performance gain is to compare the time-between-instructions using the following formula.

$$\text{Time between instructions}_{\text{pipelined}} = \text{Time between instructions}_{\text{non-pipelined}} \ / \ \text{Number of stages}$$

However, the ideal situation does not take into account the following:
a) number of stages per instruction are not equal.
b) additional overhead required to manage multiple concurrent instructions.

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

MIPS Instruction Set

The MIPS instruction set was purposely designed for pipelined execution.

- Instructions are all 4 bytes (unlike x86 with 1byte to 17byte).
- Few instruction format types (R-Type/I-Type/J-Type etc.). Common fields are always located in the same place (source registers) for each instruction. This allows the second stage to begin reading the register file even before the op code is determined.
- Memory operands only appear in loads or stores in MIPS. This allows the execute stage to calculate the memory address and then access memory in the final stage.
- Operands must be aligned in memory.

Pipeline Hazards

On occasion there are problems where one instruction cannot execute in the following clock cycle. These situations are known as hazards and are separate into three categories.

Structural Hazard: The instruction set for a pipelined implementation must support the sequential nature and separation of functional units (based upon hardware components) inherent to the implementation. Since MIPS architecture is designed for pipelining it is relatively easy for designers to avoid structural hazards.

Data Hazard: A data hazard occurs when the pipeline must be stalled because one stage must wait for another stage to complete (instruction dependencies). An example of this would be two sequential mathematical instructions which share data. The standard solution is to include extra hardware to facilitate dependant instructions using methods known as forwarding or bypassing. This additional hardware helps to prevent a pipeline stall or load-use data hazard in which the data being loaded by a load instruction has not yet become available when needed by a second instruction.

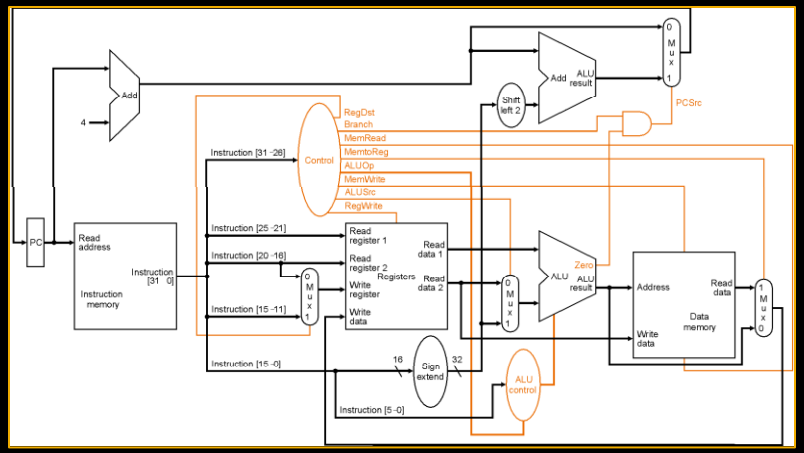
Pipeline Hazards

Control Hazard: A control hazard occurs when a decision needs to be made by one instruction while other instructions are executing. One possible method of handling a control hazard created by a branch instruction would be to stall the pipeline until the next step is known. Another possible solution is to predict the outcome and fill the pipeline based on the outcome of that prediction.

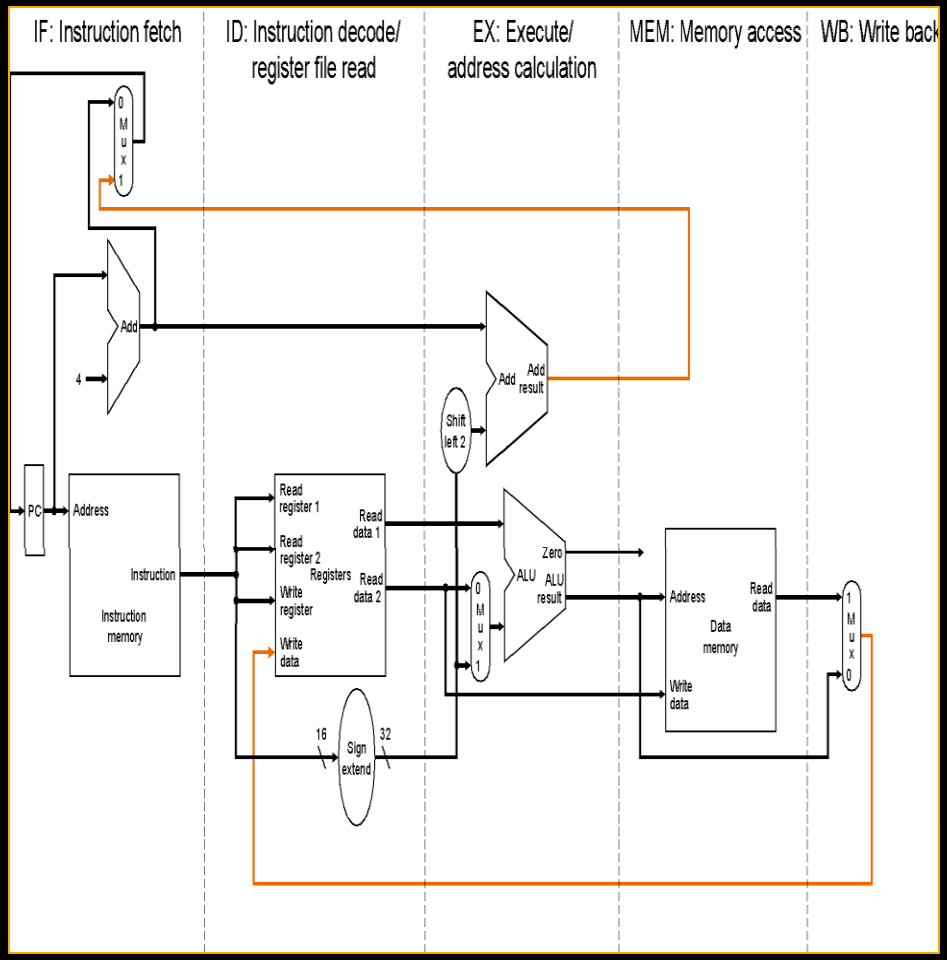
Branch Prediction: Branch prediction allows the pipeline to continue moving even when a decision is pending. If the prediction is correct, instructions are processed in advance. If the prediction is incorrect then the current instructions must be flushed or disregarded and alternate instructions are loaded. In a short pipeline (MIPS architecture) this is not much of a performance loss in event of an incorrect prediction. However, in a longer pipeline (found in some x86 implementations such as the Pentium4) this can significantly effect the processing speed of a CPU. Delayed Branch Prediction found in MIPS architecture executes the next sequential instruction (which is not affected by the Branch) before processing a branch command.

Architecture Comparison

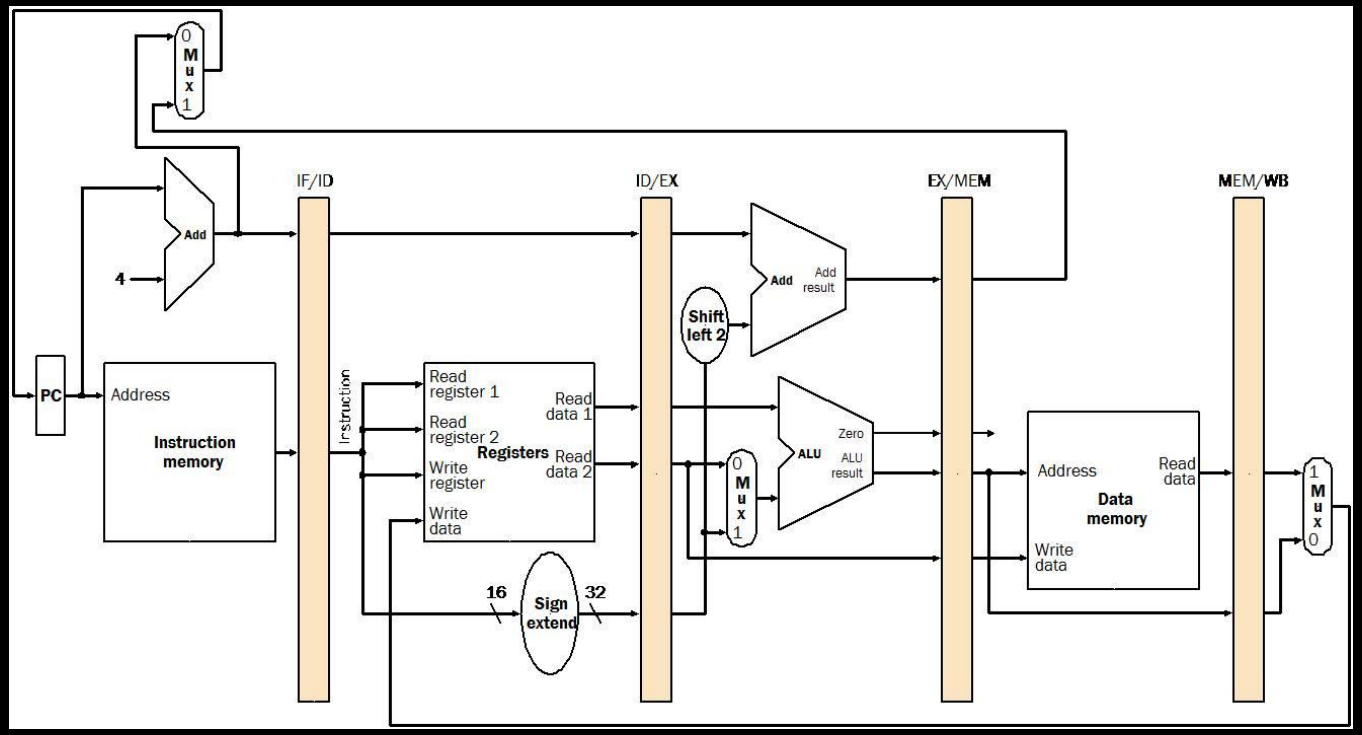
Single Cycle Datapath



Pipelined Datapath (Simplified)



Pipelined Datapath – Pipeline Registers



IF/ID ID/EX EX/Mem Mem/WB

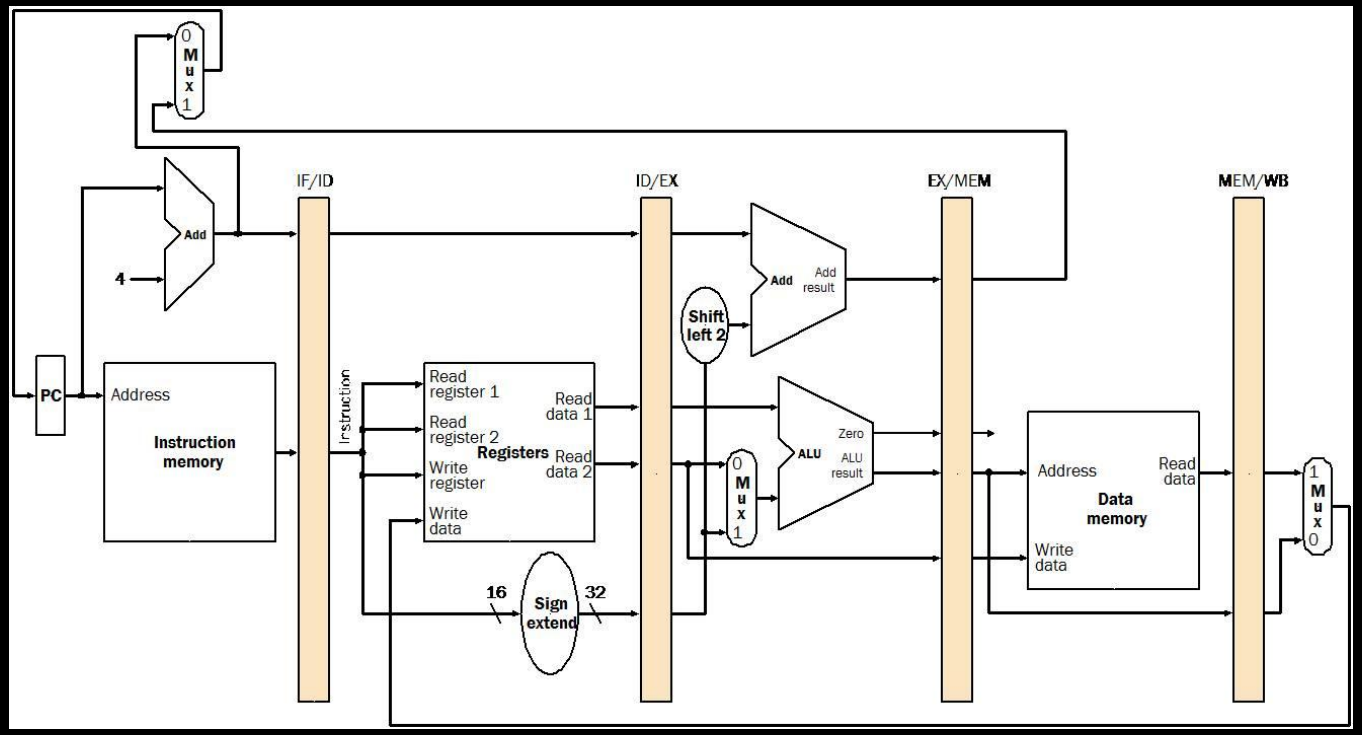
Note that there is no register following the write-back stage.

Pipeline registers are added to the datapath to hold data so that portions of the datapath can be shared during instruction execution.

The pipeline registers separate each pipeline stage.

They are labeled by the stages they separate;

Pipelined Datapath – Pipeline Registers



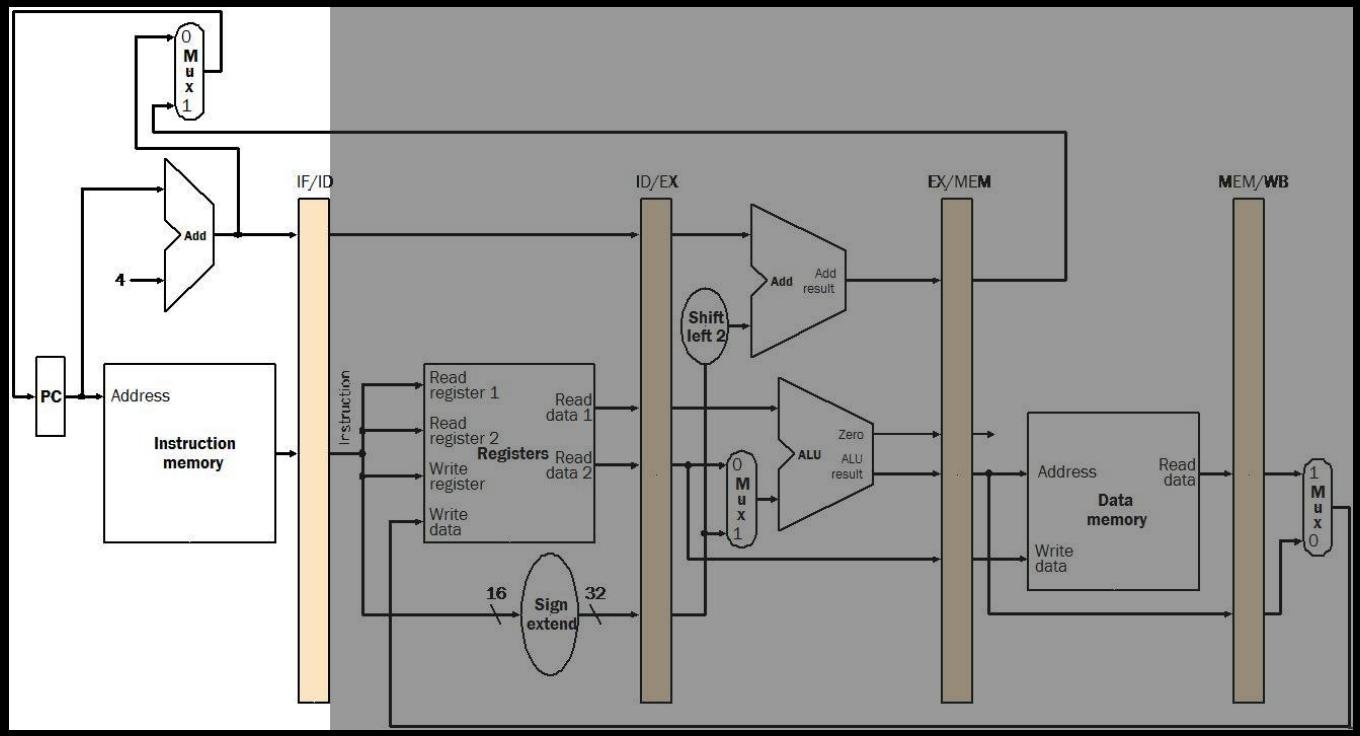
64 bits 128 bits 97 bits 64 bits

The registers must be wide enough to store all of the data corresponding to the lines that go through them.

For example, the IF/ID register must be 64 bits wide because it must hold both the 32 bit instruction fetched from memory and the incremented 32 bit PC address.

Load
Pipelined Datapath – Instruction Fetch

IF



The instruction is read from memory using the address in the PC and is placed in the IF/ID pipeline register.

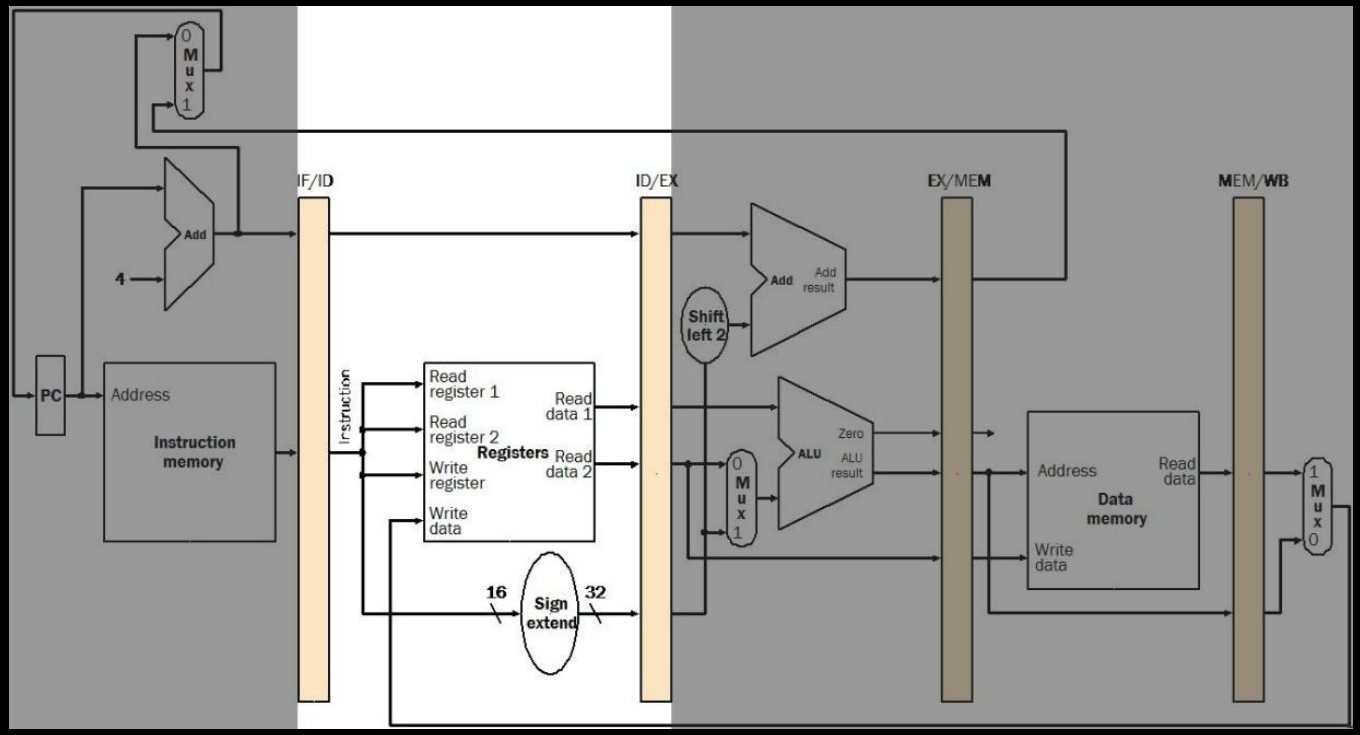
The PC address is incremented by 4 and then written back to the PC to be ready for the next clock cycle.

This incremented address is also saved in the IF/ID pipeline register in case it is needed later for an instruction, such as **beq**.

The computer cannot know which type of instruction is being fetched, so it must prepare for any instruction, passing potentially needed information down the pipeline.

Load
Pipelined Datapath – Instruction Decode

ID



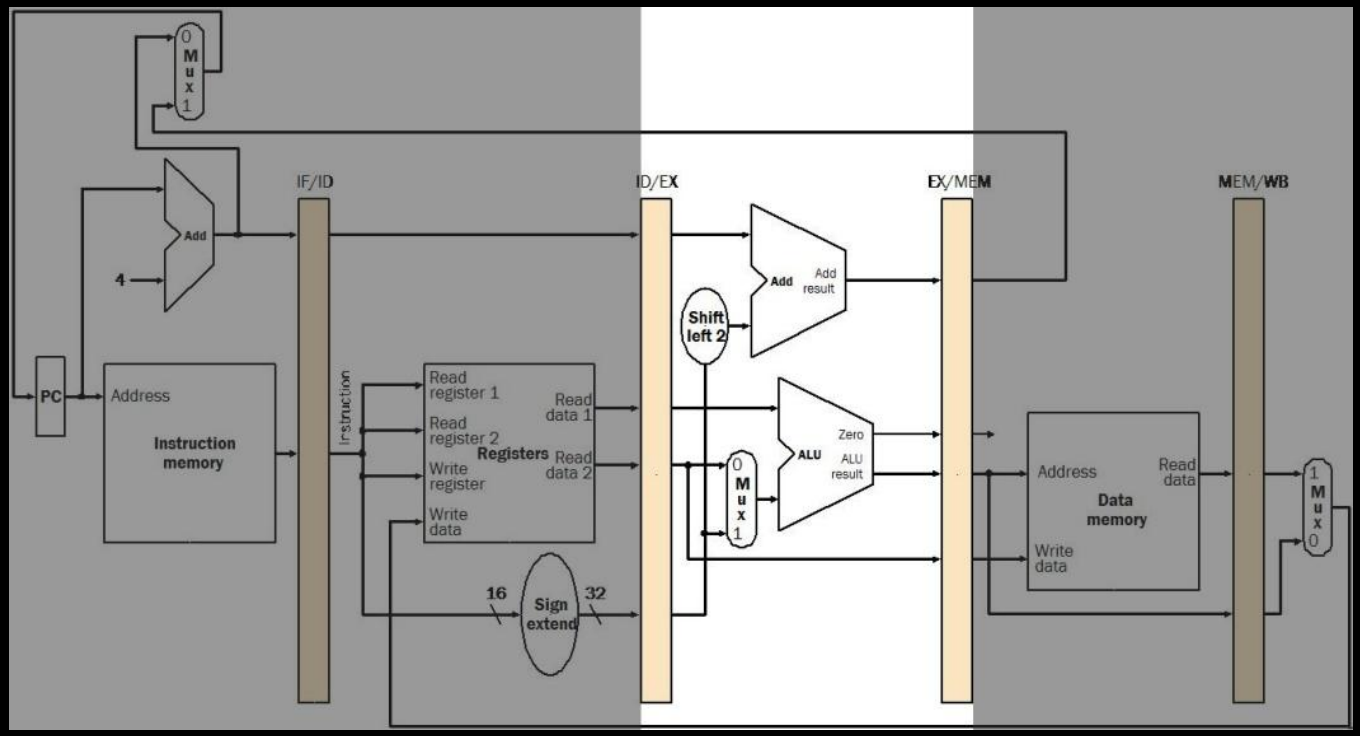
The instruction portion of the IF/ID pipeline register supplying the 16 bit immediate field, which is sign-extended to 32 bits , and the register numbers to read the two registers.

All three values are stored in the ID/EX pipeline register, along with the incremented PC address.

Everything that that might be needed by an instruction by a later clock cycle is once again transferred.

Load
Pipelined Datapath – Execute

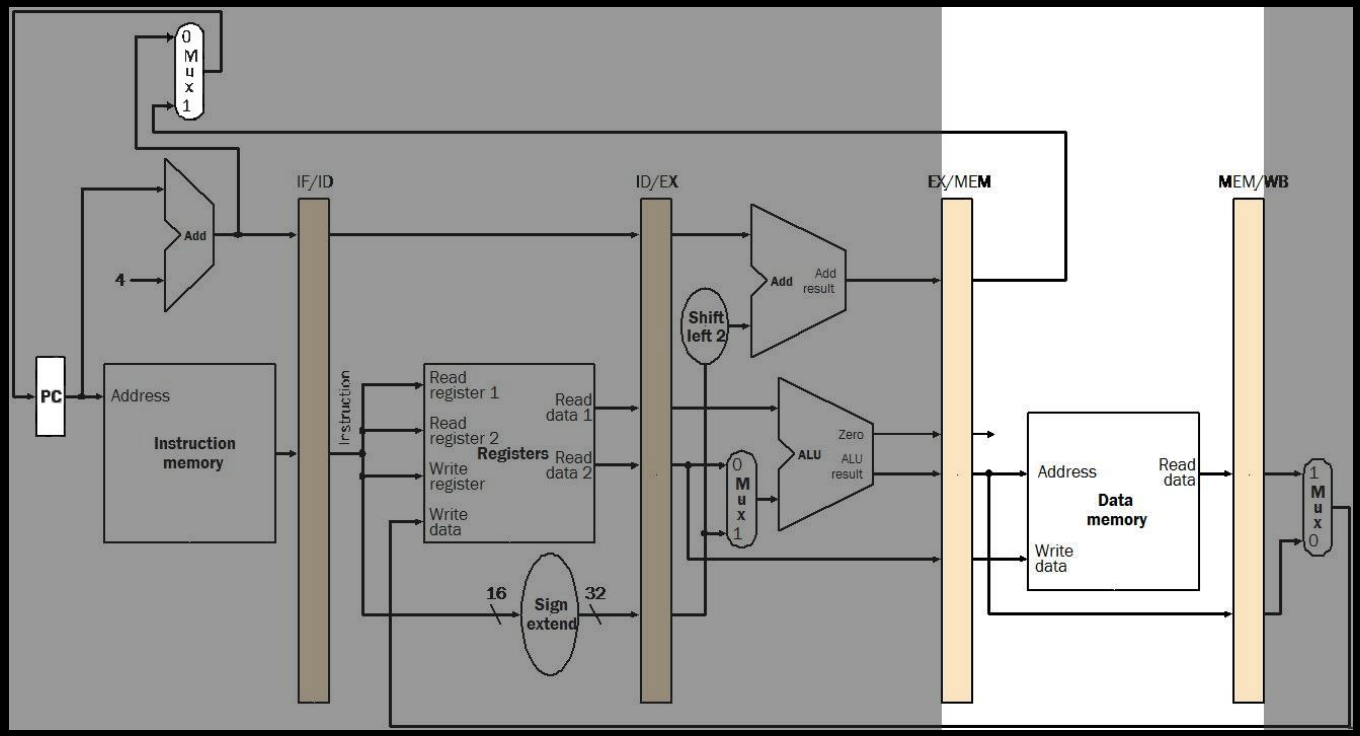
EX



The load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them to the ALU.

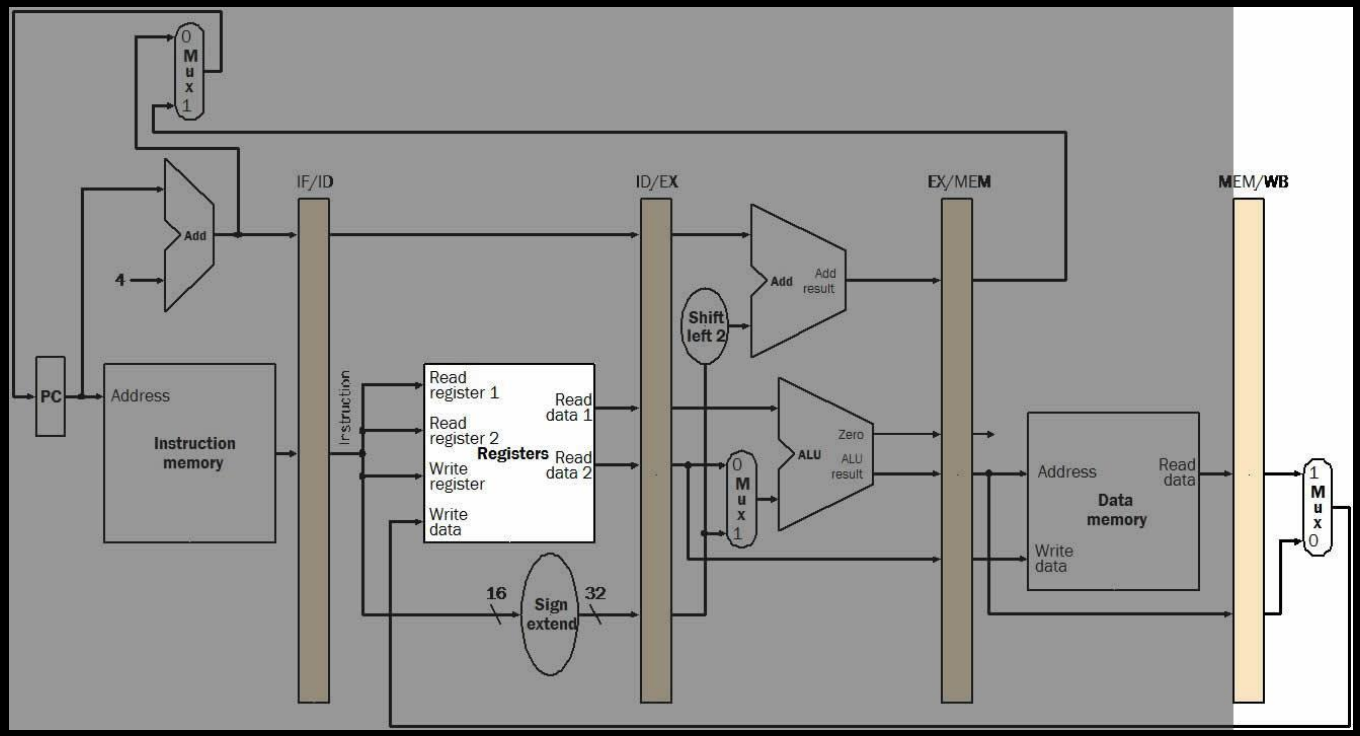
That sum is placed in the EX/MEM pipeline register.

Load
Pipelined Datapath – Memory Access



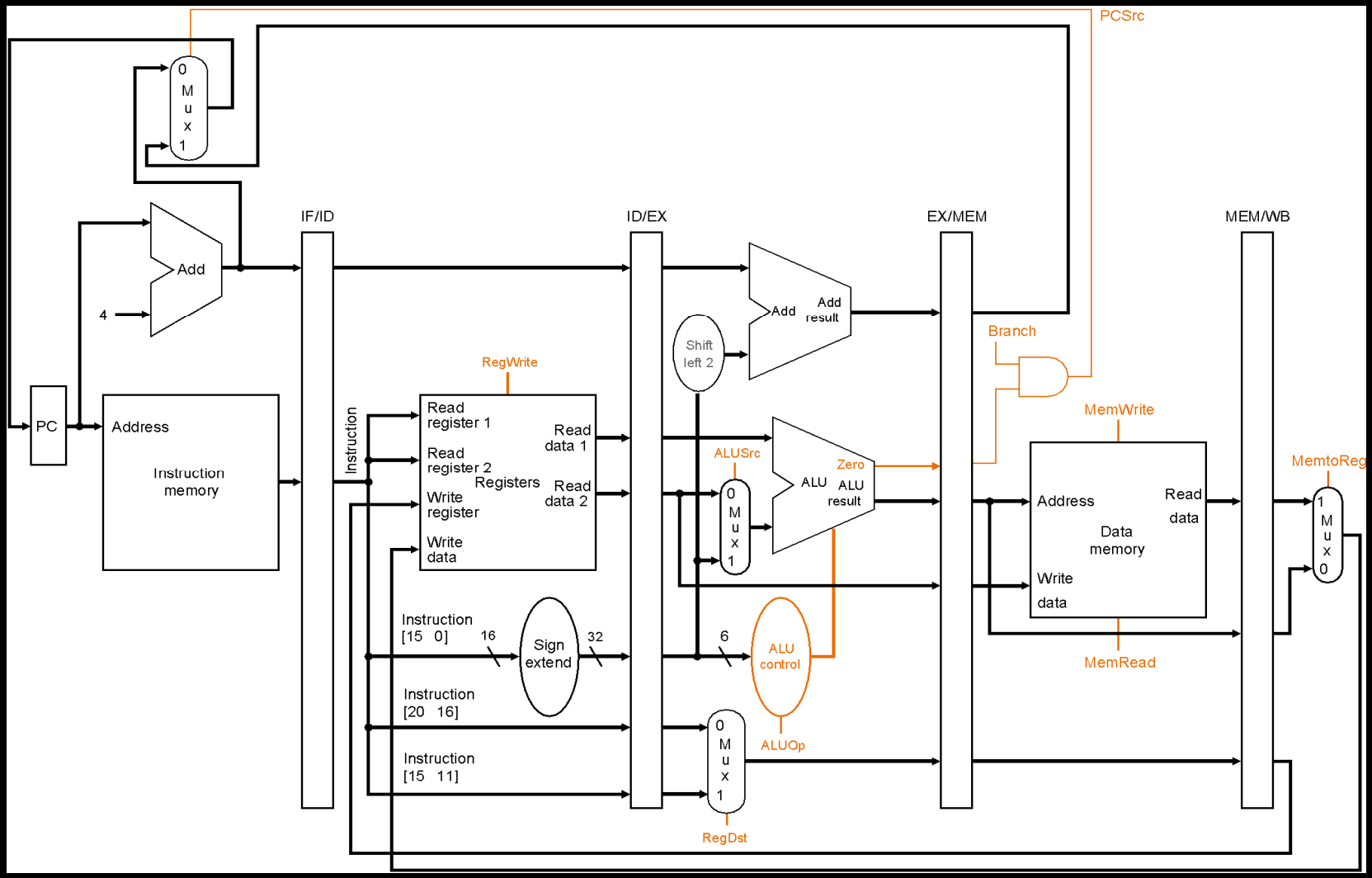
The load instruction reads the data memory using the address from the EX/MEM pipeline register and loads the data into the MEM/WB pipeline register.

Load
Pipelined Datapath – Write Back



During the final step, it reads the data from the MEM/WB pipeline register and writes it into the register file in the middle of the diagram.

The Pipeline in Execution, with controls



Pipeline Forwarding Unit

Forwarding Unit :

The data needed by an ‘and’ or an ‘or’ instruction is needed at the beginning of the Ex stage, but it may not be stored in a readable register. Thus the forwarding unit will provide this information in advance of the initial instructions write-back stage.

The forwarding unit controls a pair of multiplexors that allow the data to bypass the registers and make it available as soon as possible.

Pipeline Hazard Detection Unit

Hazard Detection Unit (HDU):

The purpose of the hazard detection unit is to initiate stalls when there exists a data dependency between two sequential instructions. For example, given a load instruction that is in the execute stage, the HDU will automatically check the destination register and compare it to either operand (source registers) of the following dependant instruction. If the register is the same, one or more stalls are inserted into the pipeline until the load instruction has completed the load function. The stall or bubble is implemented by inserting No-Op instructions, which is the setting of all nine control lines to zero which in the EX, Mem, and WB stages creates a do-nothing instruction.

The final datapath