PIPELINE HAZARDS

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

Hazards

- Structural Hazards
- Data Hazards
- Control Hazards

STRUCTURAL HAZARD

❖ Structural Hazard occurs when a planned instruction cannot execute inthe proper clock cycle because the hardware does not support the combination of instructions that are set to execute.

❖ A structural hazard in the laundry room would occur if we used a washer dryer combination instead of a separate washer and dryer, or if our roommate was busy doing something else and wouldn't put clothes away. Our carefully scheduled pipeline plans would then be foiled.

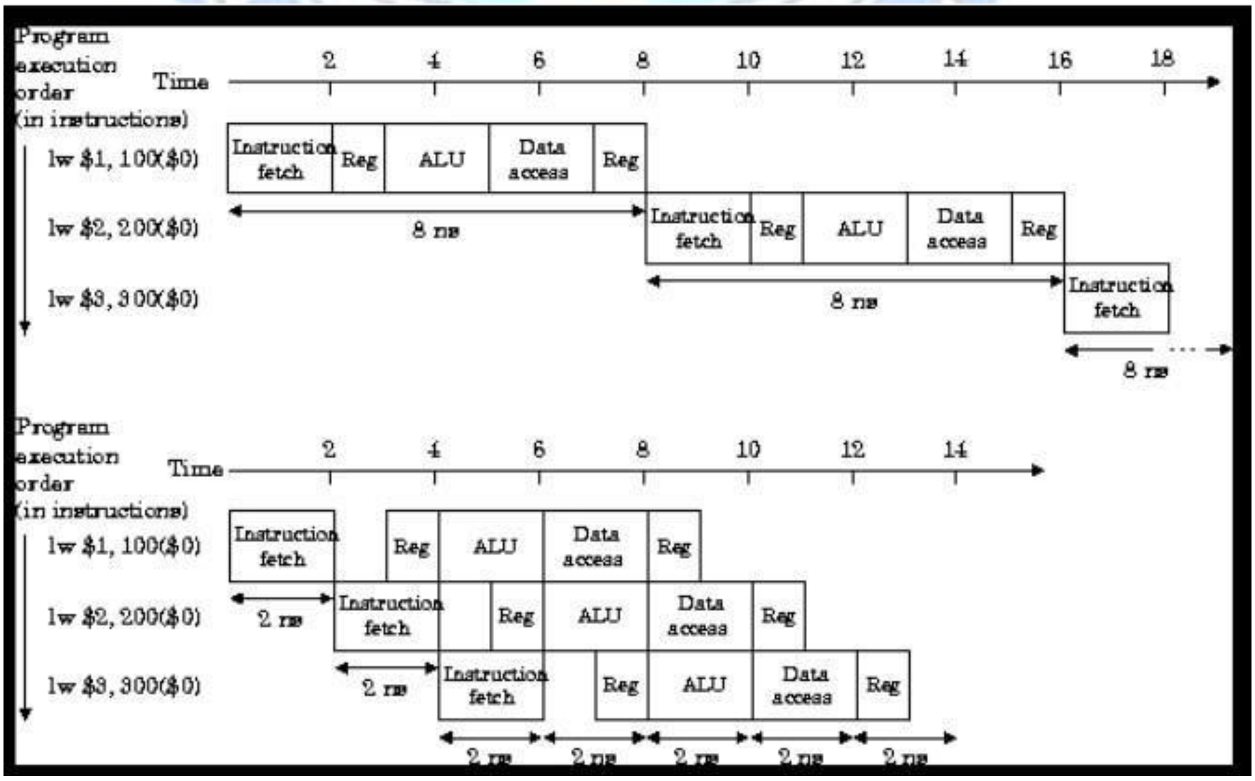


FIGURE 3.15 Single-cycle, non-pipelined execution in top versus pipelinedexecution in bottom.

As we said above, the MIPS instruction set was designed to be pipelined, making it fairly easy for designers to avoid structural hazards when designing a pipeline.Suppose, however, that we had a single memory instead of two memories. If the pipeline in Figure 3.15 had a fourth instruction, we would see that in the same clock cycle the first instruction is accessing data from memory while the fourth instruction is fetching an instruction from that same memory. Without two memories, our pipeline could have a structural hazard.

DATA HAZARDS

❖ It is 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.

❖ **Data hazards** occur when the pipeline must be stalled because one stepmust wait foranother to complete.

❖ In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline. For example, suppose we have an add instruction followed immediately by a subtract instruction that uses the sum ($s0):

add $s0, $t0, $t1 sub
$t2, $s0, $t3

❖ Without intervention, a data hazard could severely stall the pipeline. The add instruction doesn't write its result until the fifth stage, meaning that we would have to waste three clock cycles in the pipeline.

❖ To resolve the data hazard, for the code sequence above, as soon as the ALU creates the sum for the add operation, we can supply it as an input for the subtract. This is done by adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding or bypassing.**

- Figure below shows the connection to forward the value in \$s0 afterthe execution stage of the add instruction as input to the execution stage of the sub instruction.

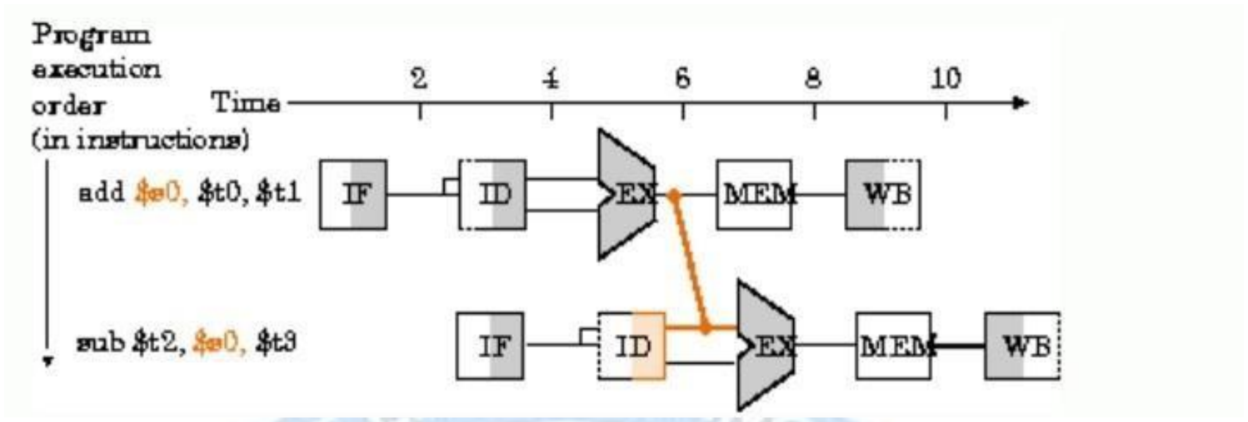
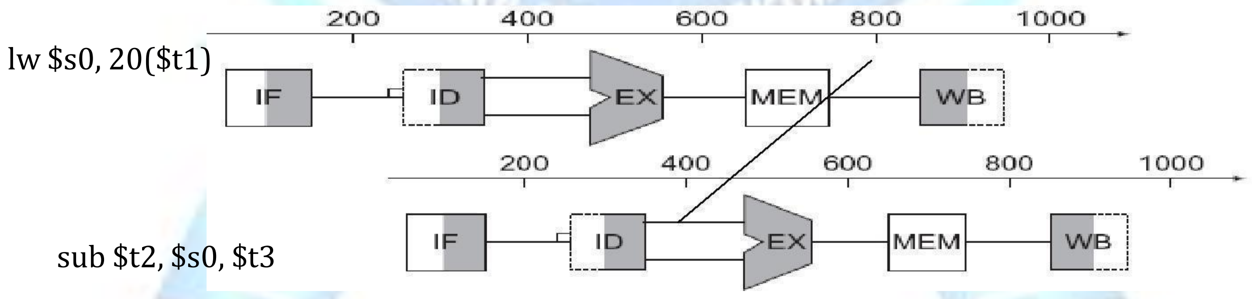
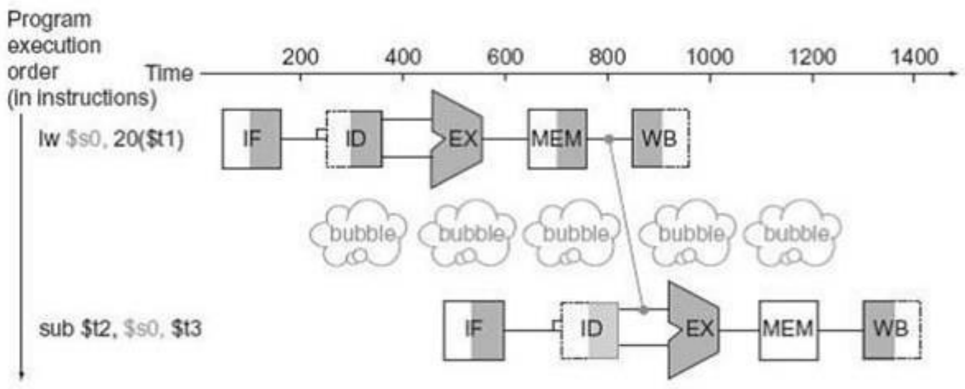


Fig 3.16: **Graphical representation of forwarding**

- Forwarding paths are valid only if the destination stage is later in time than the source stage. For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backward in time.
- Forwarding cannot prevent all pipeline stalls, suppose the first instruction was a load of $s0 instead of an add, So desired data would be available only after the fourth stage of the first instruction in the dependence, which is too late for the input of the third stage of sub instruction.
-



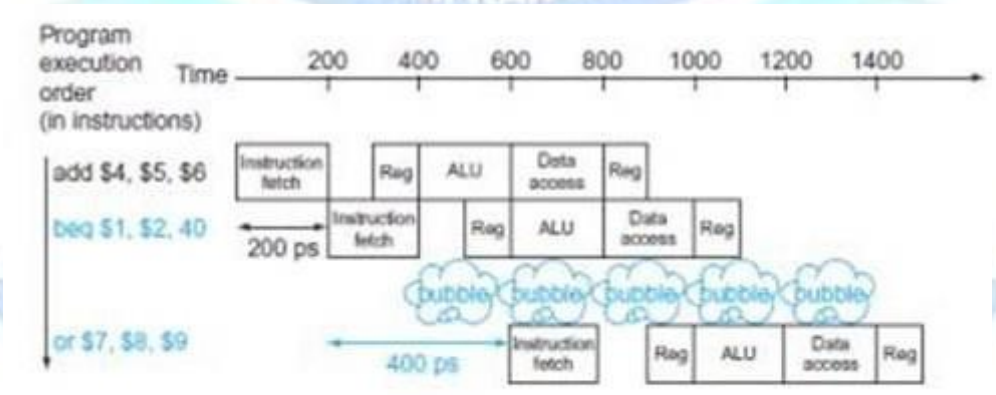
Even with forwarding, we would have to stall one stage for a load-use datahazard, this figure shows below an important pipeline concept, officially called a pipeline stall, but often given the nickname bubble.


Program execution order (in instructions)
Time
200
400
600
800
1000
1200
1400
lw $s0, 20($t1)
IF
ID
EX
MEM
WB
bubble
bubble
bubble
bubble
bubble
sub $t2, $s0, $t3
IF
ID
EX
MEM
WB

A stall even with forwarding when an R-format instruction following a loadtries to use the data.

<u>CONTROL HAZARDS</u>

- ❖ It is also called as branch hazard. When the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that wasfetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.
- ❖ A control hazard, arising from the need to make a decision based on theresults of one instruction while others are executing.
- ❖ Even with this extra hardware, the pipeline involving conditional brancheswould look like figure 3.18. The **lw** instruction, executed if the branch fails,is stalled one extra 200 ps clock cycle before starting.
- ❖ The equivalent decision task in a computer is the branch instruction. Notice that we must begin fetching the instruction following the branch onthe very next clock cycle. Nevertheless, the pipeline cannot possibly know what the next instruction should be, since it *only just received* the branch instruction from memory.
- ❖ One possible solution is to stall immediately after we fetch a branch, waiting until the pipeline determines the outcome of the branch and knows what instruction address to fetch from. Let's assume that we put in enough extra hardware so that we can test registers, calculate the branch address, and update the PC during the second stage of the pipeline.



Pipeline showing stalling on every conditional branch as solution tocontrol hazards.

<u>BRANCH PREDICTION</u>

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.

Static branch prediction

A more sophisticated version of **branch prediction** would have some branches predicted as taken and some as untaken. In the case of programming, at the bottom of loops are branches that jump back to the top of the loop. Since they are likely to be taken and they branch backward, we could always predict taken for branches that jump to an earlier address.

Dynamic branch prediction

Dynamic hardware predictors, in stark contrast, make their guesses depending on thebehavior of each branch and may change predictions for a branch over thelife of a program. Following our analogy, in dynamic prediction a person would look at how dirty the uniform was and guess at the formula, adjusting the next **prediction** depending on the success of recent guesses. One popular approachto dynamic prediction of branches is keeping a history for each branch as taken or untaken, and then using the recent past behavior to predict the future.

PIPELINED DATAPATH

Figure 3.19 shows the single-cycle datapath from with the pipeline stages identified. The division of an instruction into five stages means a five-stage pipeline, which in turn means that up to five instructions will be in execution during any single clock cycle. Thus, we mustseparate the datapath into five pieces, with each piece named corresponding to a stage of instruction execution:

1. IF: Instruction fetch
2. ID: Instruction decode and register fi le read
3. EX: Execution or address calculation
4. MEM: Data memory access
5. WB: Write back

- In Figure 3.19, these five components correspond roughly to the way the datapath is drawn; instructions and data move generally from left to right through the five stages as they complete execution.

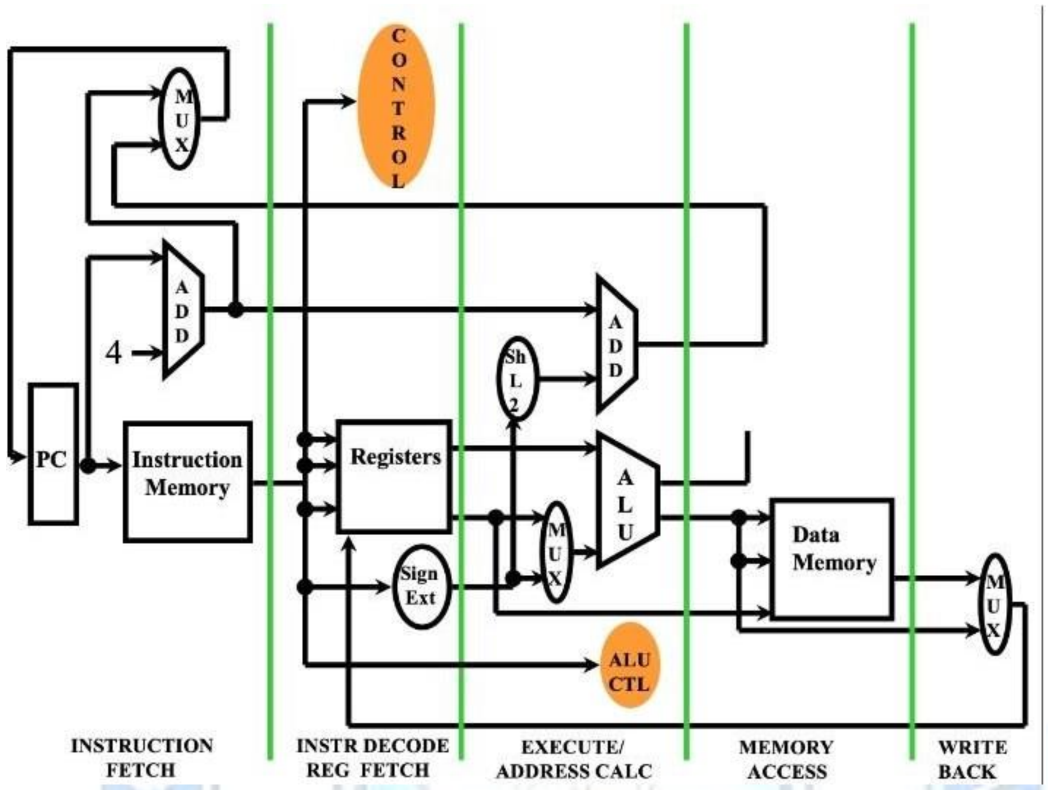
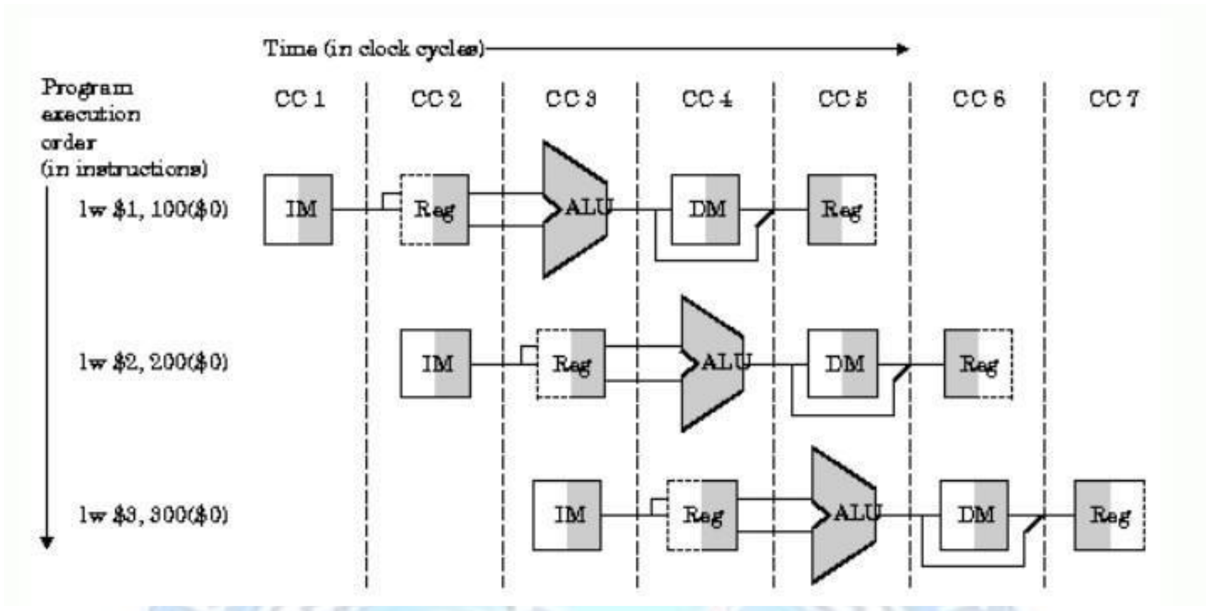


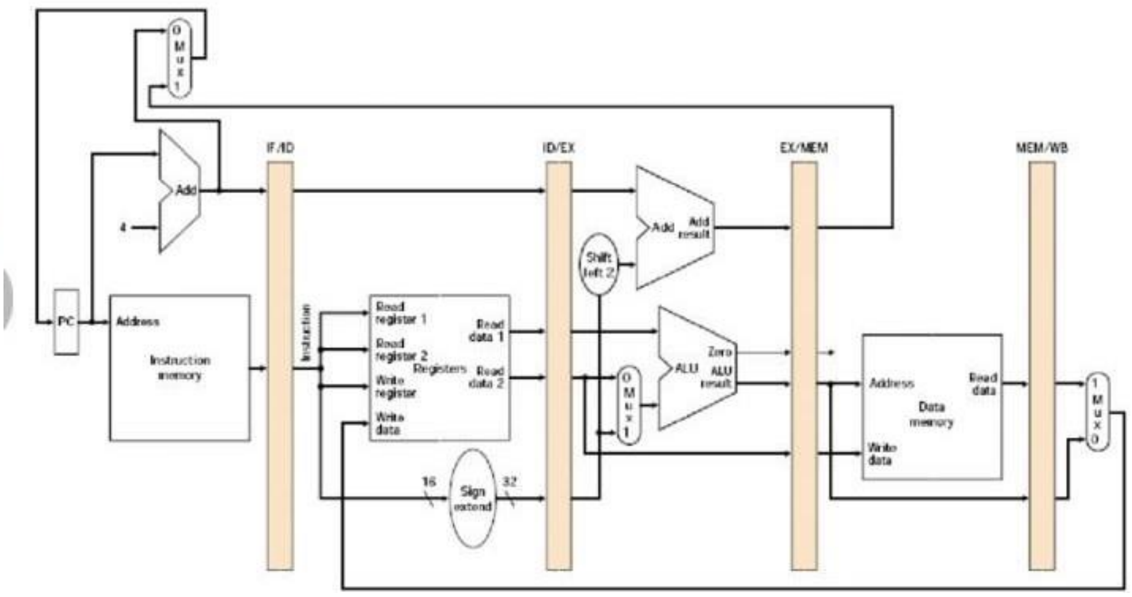
Fig 3.19: The single-cycle datapath

- ❖ Returning to our laundry analogy, clothes get cleaner, drier, and more organized as they move through the line, and they never move backward. There are, however, two exceptions to this left -to-right flow of instructions:
  - ■ The write-back stage, which places the result back into the register fi le inthe middle of the datapath
  - ■ The selection of the next value of the PC, choosing between the incremented PC and the branch address from the MEM stage
- ❖ Data flowing from right to left does not affect the current instruction; these reverse data movements influence only later instructions in the pipeline.Note that the fi rst right-to-left flow of data can lead to data hazards and the second leads to control hazards.
- ❖ One way to show what happens in pipelined execution is to pretend that eachinstruction has its own datapath, and then to place these datapaths on a timeline to show their relationship. Figure 3.20 shows the execution of the instructions in Figure 4.27 by displaying their private datapaths on a common timeline. Instead, we add registers to hold data so that portions of a single datapath can be shared during instruction execution.



Instructions being executed using the single-cycle datapath in Figure3.19, assuming pipelined execution.

- For example, as Figure 4.34 shows, the instruction memory is used during only one of the five stages of an instruction, allowing it to be shared byfollowing instructions during the other four stages.
- To retain the value of an individual instruction for its other four stages, the value read from instruction memory must be saved in a register. Returning toour laundry analogy, we might have a basket between each pair of stages to hold the clothes for the next step.



The pipelined version of the data path

- Figure 3.21 shows the pipelined data path with the pipeline registers highlighted. All instructions advance during each clock cycle from one pipeline register to the next. The registers are named for the two stages separated by that register. For example, the pipeline register between the IF and ID stages is called IF/ID. Notice that there is no pipeline register at the end of the write-back stage.

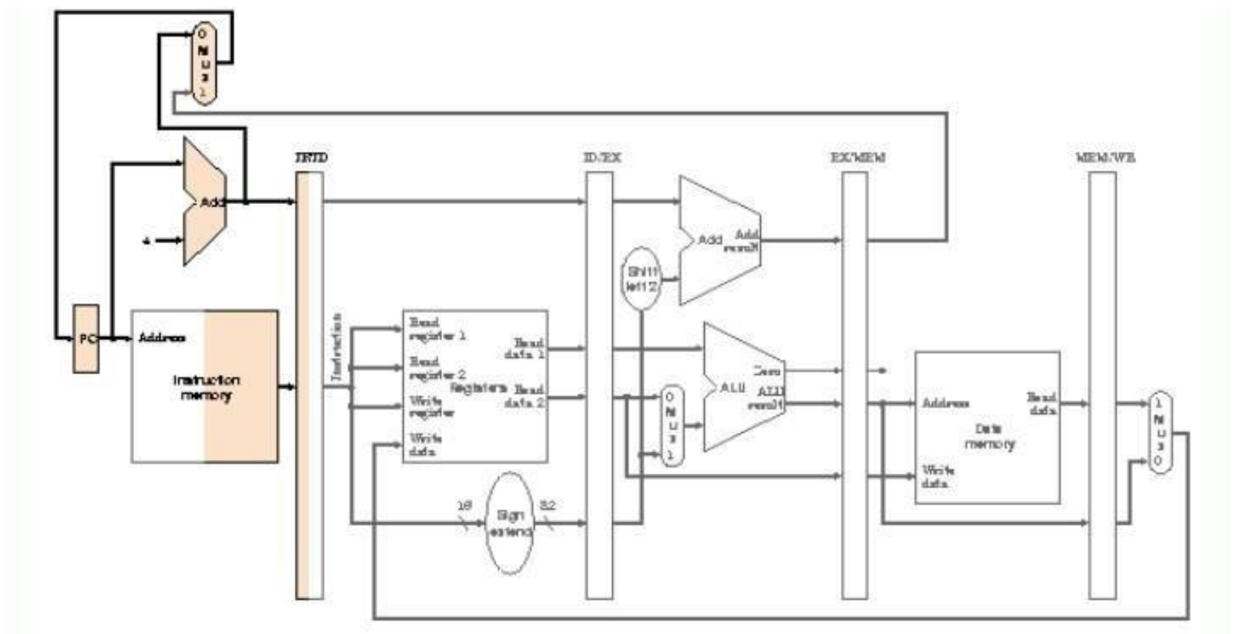
  aAll instructions must update some state in the processor—the register file,memory, or the PC.

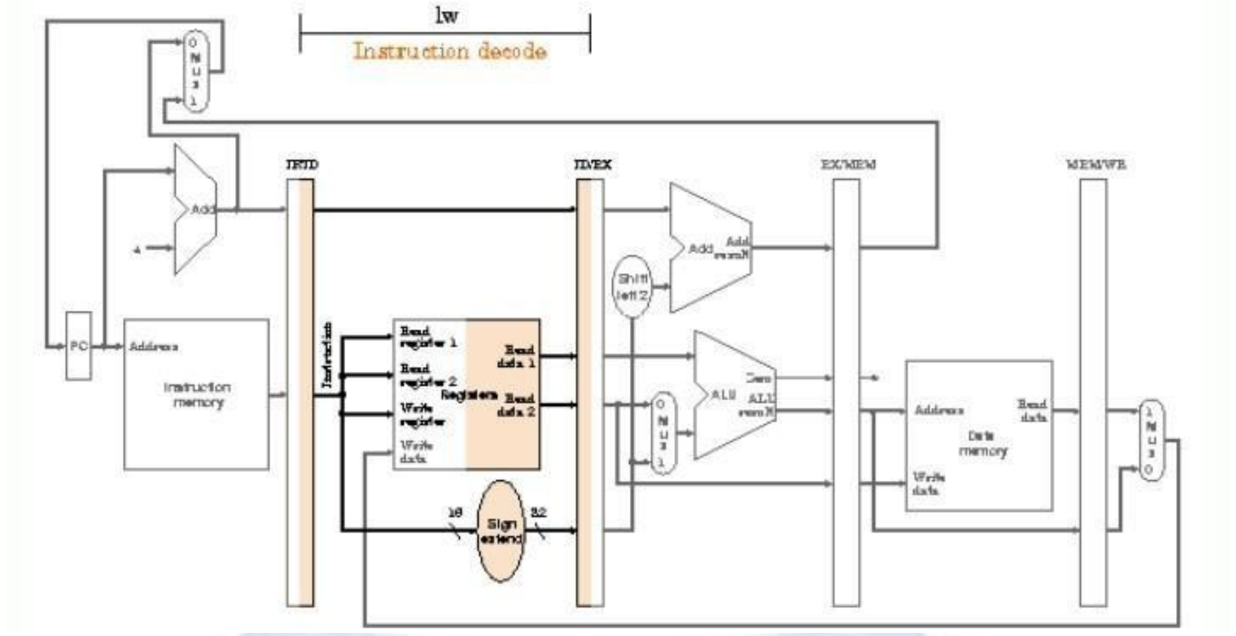
EXECUTION OF *load* INSTRUCTION IN A PIPELINED DATAPATH

Figures 3.22 through 3.24, show the active portions of the datapath highlighted as a load instruction goes through the five stages of pipelined execution. The fivestages are the following:

1. ***Instruction fetch:*** The top portion of Figure 3.22 shows the instruction being read from memory using the address in the PC and then being placed in the IF/ID pipeline register. The PC address is incremented by 4 and then written back into the PC to be ready for the next clock cycle.

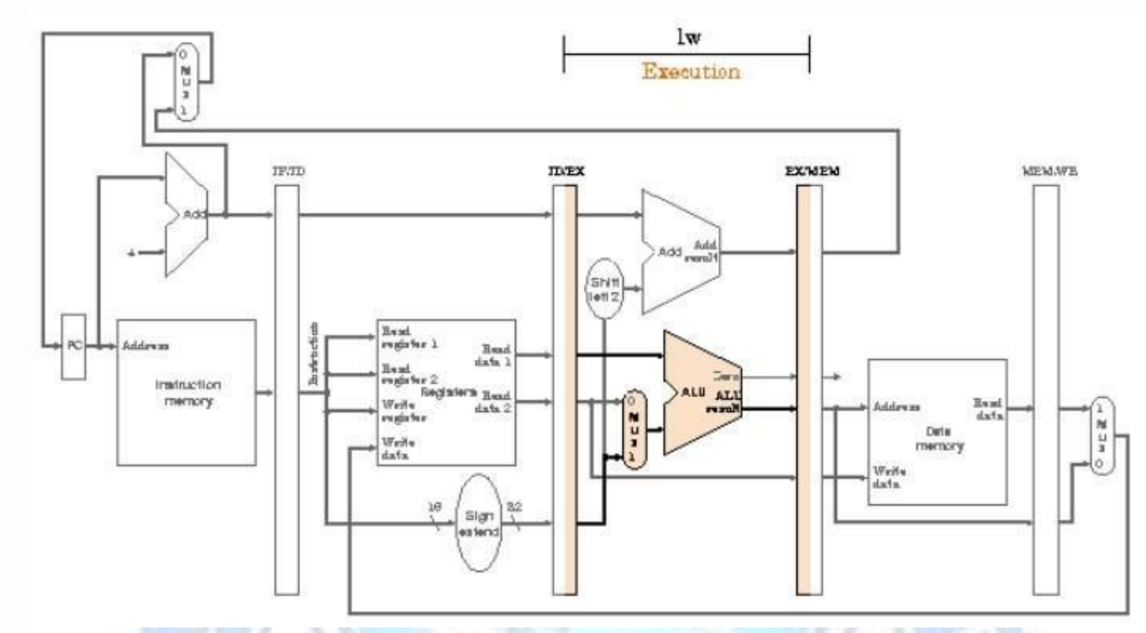
Instruction decode and register file read: The bottom portion of Figure 3.22 shows theinstruction portion of the IF/ID pipeline register supplying the 16-bitimmediate field, whichis sign-extended to 32 bits, and the register numbers to read the two registers. All three values are stored in the ID/EX pipeline register,along with the incremented PC address.





IF and ID: First and second pipe stages of an instruction, with the activeportions of the data path in highlighted.

3. Execute or address calculation: Figure 3.23 shows that the load instruction reads the contents of register 1 and the sign-extended immediate from the ID/EX pipeline register and adds them using the ALU. That sum is placed in the EX/MEM pipeline register.



The third pipe stage of a load instruction, highlighting the portions ofthe data path in used in this pipe stage.

4. **Memory access:** The top portion of Figure 3.24 shows the loadinstruction reading thedata memory using the address from the EX/MEMpipeline register and loading the data into the MEM/WB pipeline register. **5.Write-back:** The bottom portion of Figure 3.24 shows the final step: reading the data from the MEM/WB pipeline register and writing it into theregister file in the middle of the figure. This walk-through of the load instruction shows that any information needed in a later pipe stage must bepassed to that stage via a pipeline register.

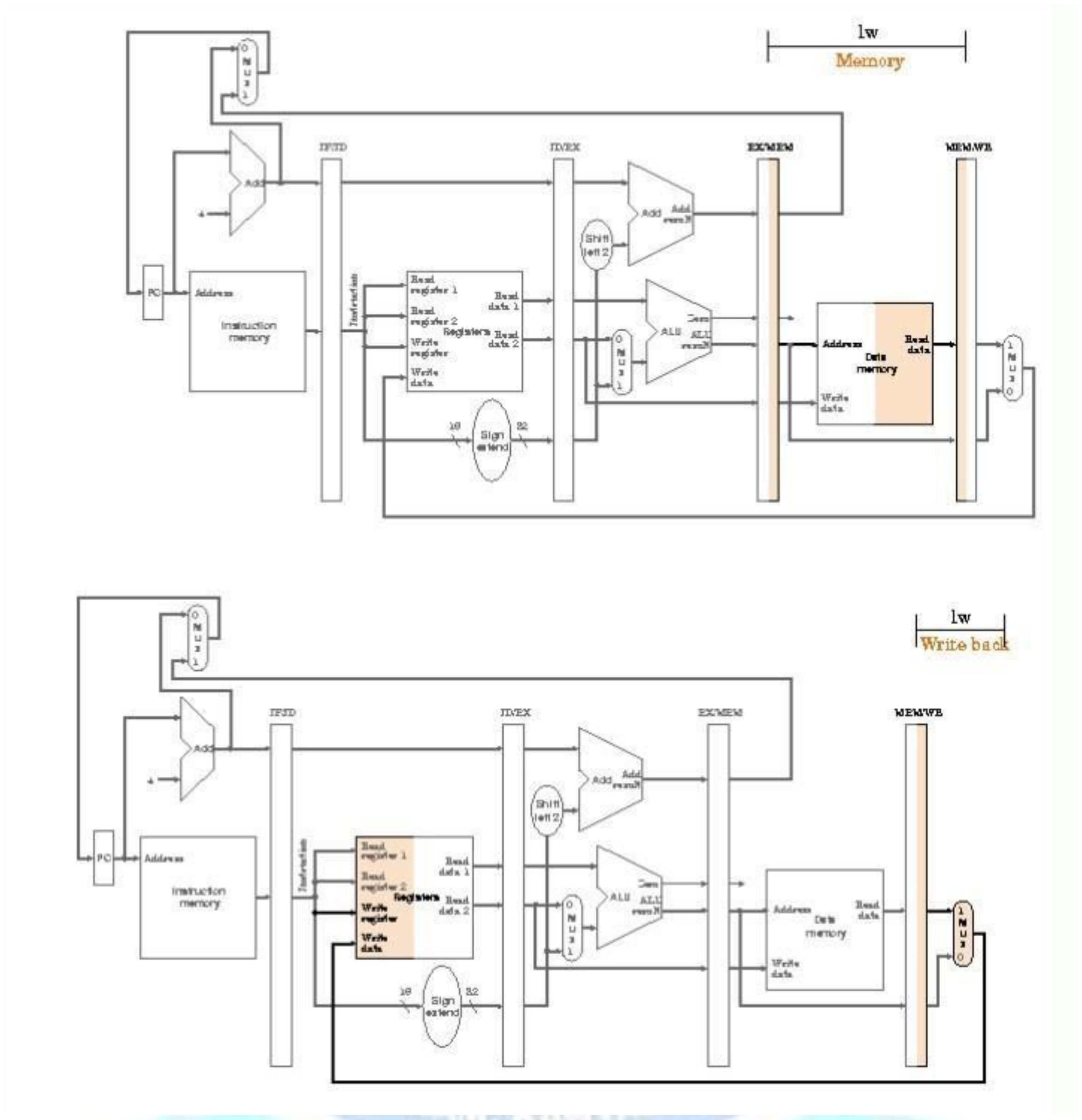
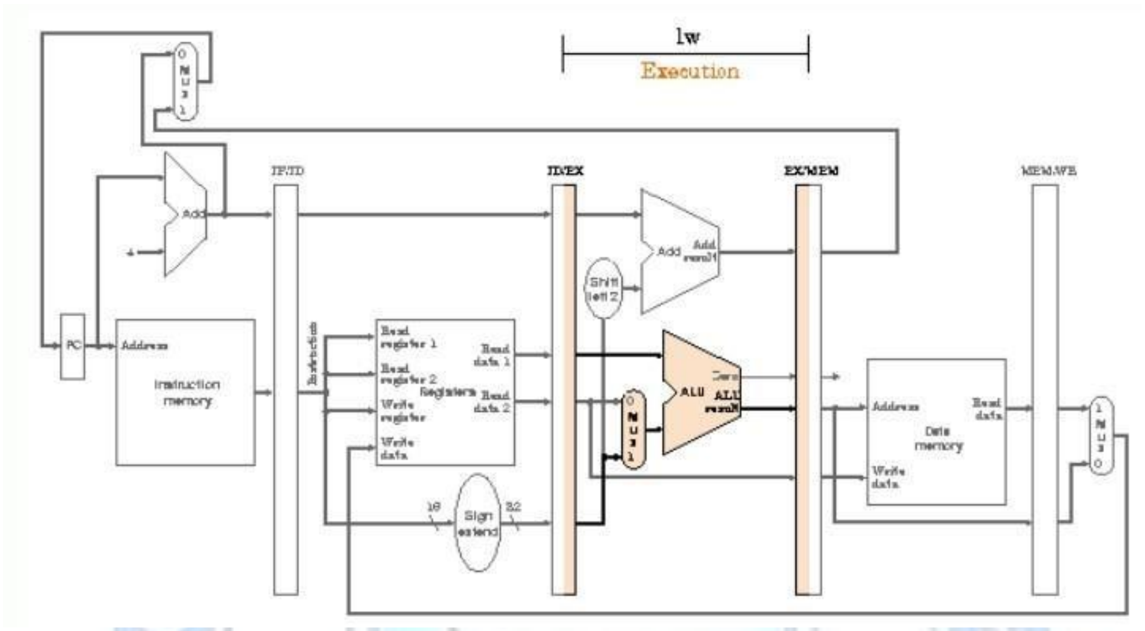


FIGURE 3.24 MEM and WB: The fourth and fifth pipe stages of a loadinstruction, highlighting the portions of the datapath in Figure 3.21used in this pipe stage.

<u>EXECUTION OF *Store* INSTRUCTION IN A PIPELINED DATAPATH</u>

Walking through a store instruction shows the similarity of instruction execution, as well as passing the information for later stages. Here are thefive pipe stages of the store instruction:

1. Instruction fetch: The instruction is read from memory using theaddress in the PC andthen is placed in the IF/ID pipeline register. This stage occurs before the instruction is identified, so the top portion of Figure 3.25 works for store as well as load.

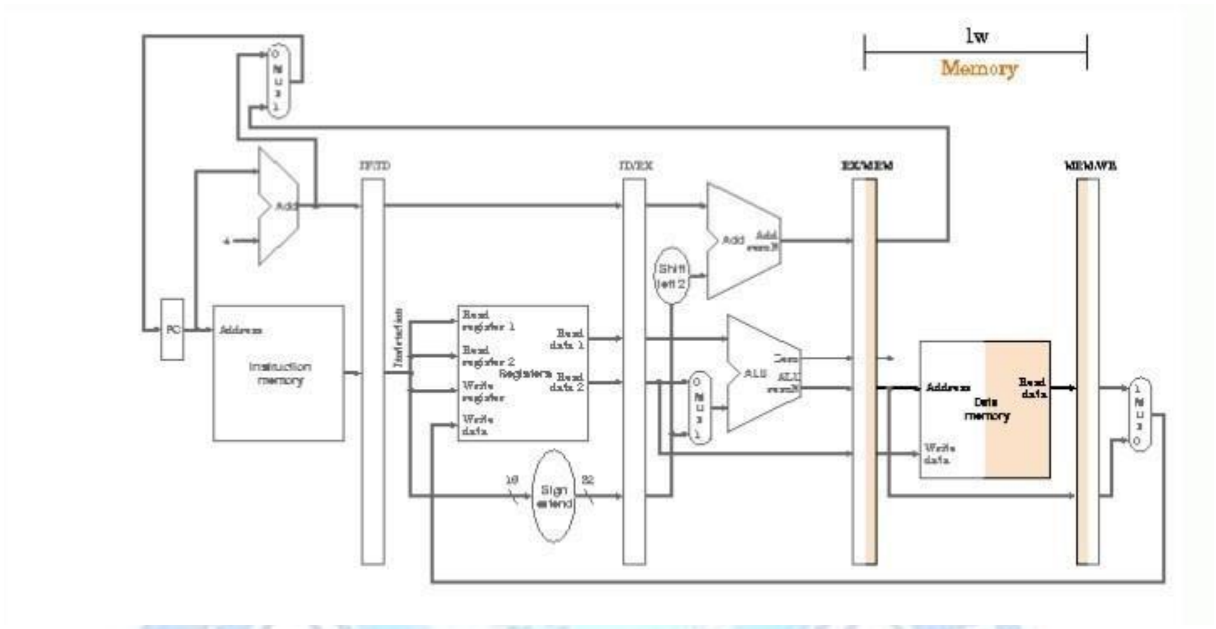


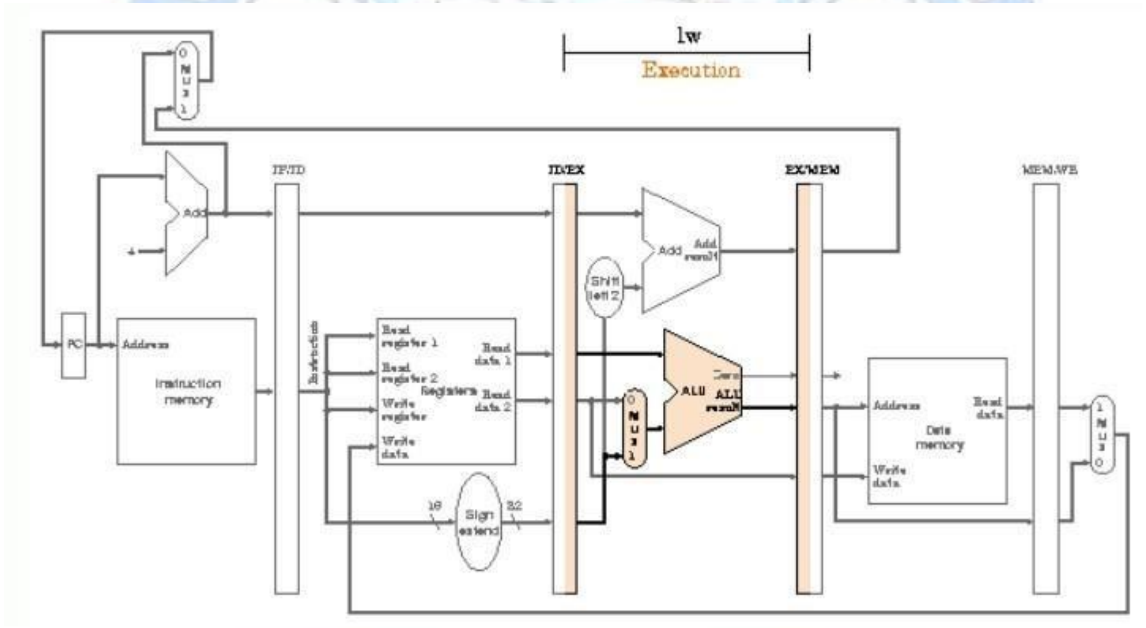
The third pipe stage of a store instruction

2. ***Instruction decode and register file read:*** The instruction in the IF/ID pipeline register supplies the register numbers for reading two registers and extends the sign of the 16-bit immediate. These three 32-bit values are all stored in the ID/EX pipeline register. The bottom portion of Figure 3.25 for load instructions also shows the operations of the second stage for stores. These first two stages are executed by all instructions, since it is too early to know the type of the instruction.

3. ***Execute and address calculation:*** Figure 3.26 shows the third step; theeffective address is placed in the EX/MEM pipeline register.

4. ***Memory access:*** The top portion of Figure 3.26 shows the data being written to memory.Note that the register containing the data to be stored was read in an earlier stage and stored in ID/EX. The only way to make the data available during the MEM stage is to place the data into the EX/MEM pipeline register in the EX stage, just as we stored the effective address intoEX/MEM.





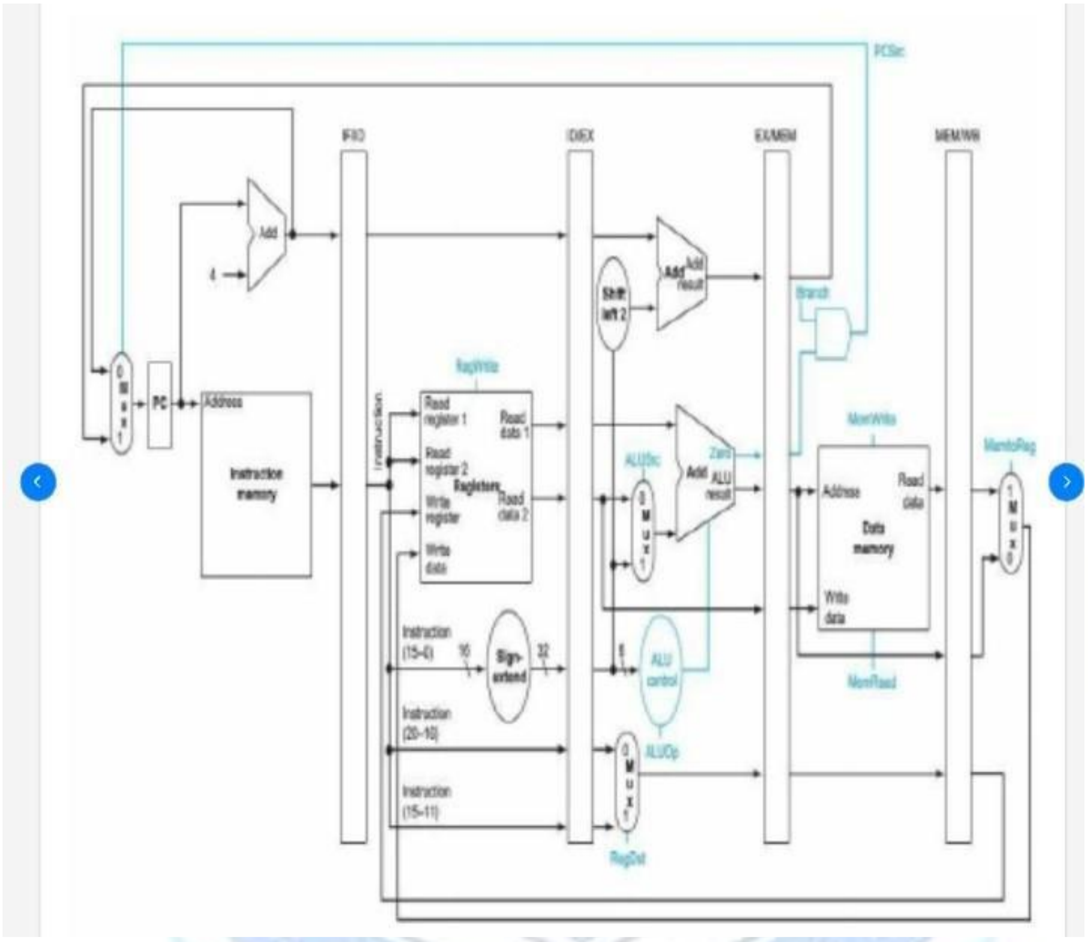
MEM and WB: The fourth and fifth pipe stages of a store instruction.

5. Write-back: The bottom portion of Figure 3.26 shows the final step ofthe store. For this instruction, nothing happens in the write-back stage.

For the store instruction we needed to pass one of the registers read in the ID stage to the MEM stage, where it is stored in memory. The data was first placed in the ID/EX pipeline register and then passed to the EX/MEM pipeline register.

PIPELINED CONTROL

Adding control to the pipelined data path is referred to as pipelined control.It is started with a simple design that views the problem through pipeline bars in between the stages. The first step is to label the control lines on theexisting data path. Figure 3.27 shows those lines.



The pipelined data path with the control signals identified.

To specify control for the pipeline, we need only set the control values during each pipeline stage. Because each control line is associated with acomponent active in only a single pipeline stage, we can divide the control lines into five groups according to the Pipe line stage.

1. ***Instruction fetch:*** The control signals to read instruction memory and towrite the PC arealways asserted, so there is nothing special to control in this pipeline stage.
2. ***Instruction decode/register file read:*** As in the previous stage, the samething happens atevery clock cycle, so there are no optional control lines to set.

3. ***Execution/address calculation:*** The signals to be set are RegDst, ALUOp, and ALUSrc(see Figures 4.48). The signals select the Result register, the ALU operation, and either Read data 2 or a sign-extended immediate for the ALU.

4. ***Memory access:*** The control lines set in this stage are Branch, MemRead, and Mem Write.The branch equal, load, and store instructions set these signals, respectively. Recall that PCSrc in Figure 4.48 selects the next sequential address unless control asserts Branch and the ALU result was 0.

Write-back: The two control lines are MemtoReg, which decides between sending theALU result or the memory value to the register file, and Reg-Write,which writes the chosen value. Since pipelining the datapath leaves the meaning of the control lines unchanged, we can use the same control values.

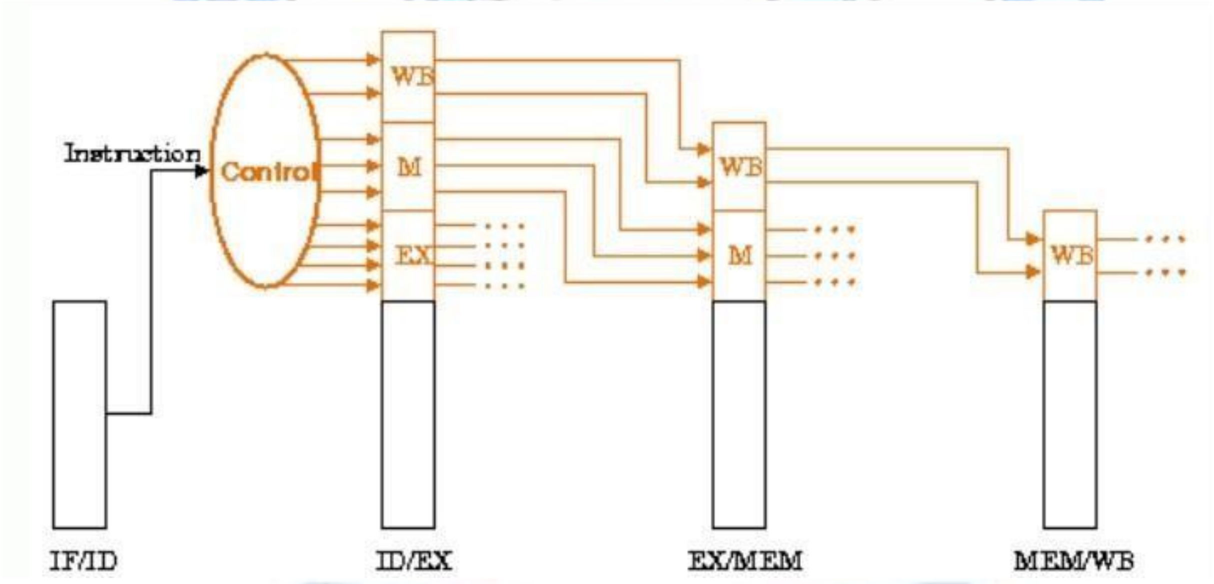


FIGURE 3.27: The control lines for the final three stages.

Implementing control means setting the nine control lines to these values in each stage for each instruction. The simplest way to do this is to extend the pipeline registers to include control information. Since the control lines start with the EX stage, we can create the control information during instruction decode. Figure 3.27 above shows that these control signals are then used in the appropriate pipeline stage as the instruction moves downthe pipeline.