RISC-V Pipeline

Computer Architecture

Dr. Mahmoudi

May 20, 2024

Why a Single-Cycle Implementation is not Used Today

- Notice that the clock cycle must have the same length for every instruction in this single-cycle design.
- The longest possible path in the processor determines the clock cycle (load instruction).
- The overall performance of a single-cycle implementation is likely to be poor since the clock cycle is too long.
- Historically, early computers with very simple instruction sets did use this implementation technique.

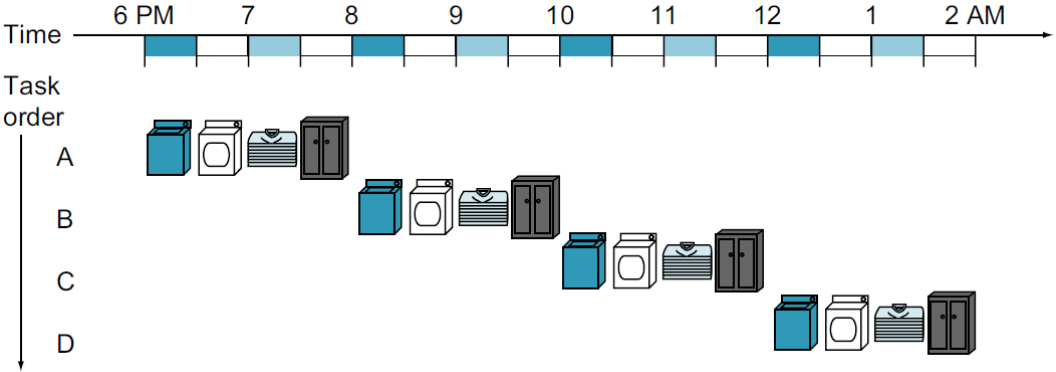
Pipelining

- **Pipelining** is an implementation technique in which multiple instructions are overlapped in execution.
- Today, pipelining is nearly universal.

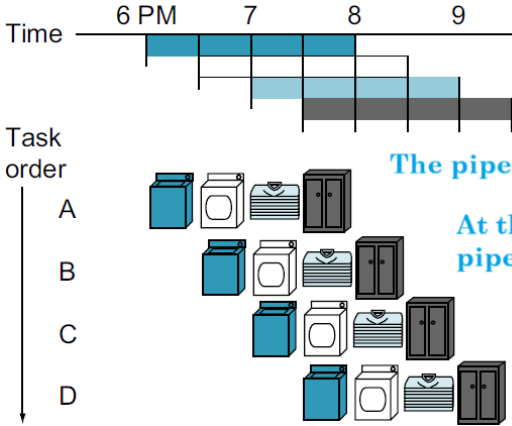
Pipelining analogy: Laundry

- The nonpipelined approach to laundry would be as follows:
  1. Place one dirty load of clothes in the washer.
  2. When the washer is finished, place the wet load in the dryer.
  3. When the dryer is finished, place the dry load on a table and fold.
  4. When folding is finished, put them away.
- When this load is done, start over with the next dirty load.

Pipelining analogy: Laundry



Pipelining analogy: Laundry



The pipelined approach takes much less time.

At this point all steps, called stages in pipelining, are operating concurrently.

As long as we have separate resources for each stage, we can pipeline the tasks.

Pipelining

- The reason why pipelining is faster is that everything is working in parallel.
- Pipelining improves the throughput of our system.
- Hence, pipelining would not decrease the time to complete one instruction, but when we can execute many instructions, the improvement in throughput decreases the total time to complete the work.
- If all the stages take about the same amount of time and there is enough work to do, then the speed-up due to pipelining is equal to the number of stages in the pipeline.
- If we had five stages then the pipelined execution is potentially five times faster than nonpipelined.

RISC-V Pipelining

- The same principles apply to processors where we pipeline instruction execution.
- RISC-V instructions classically take five steps:
  1. Fetch instruction from memory.
  2. Read registers and decode the instruction.
  3. Execute the operation or calculate an address.
  4. Access an operand in data memory (if necessary).
  5. Write the result into a register (if necessary).

Single-Cycle versus Pipelined Performance

- Compare the average time between instructions of a single-cycle implementation, in which all instructions take one clock cycle, to a pipelined implementation.
- Assume that the operation times for the major functional units in this example are:
  1. 200 ps for memory access for instructions or data
  2. 200 ps for ALU operation
  3. 100 ps for register file read or write
- In the single-cycle model, every instruction takes exactly one clock cycle, so the clock cycle must be stretched to accommodate the slowest instruction.

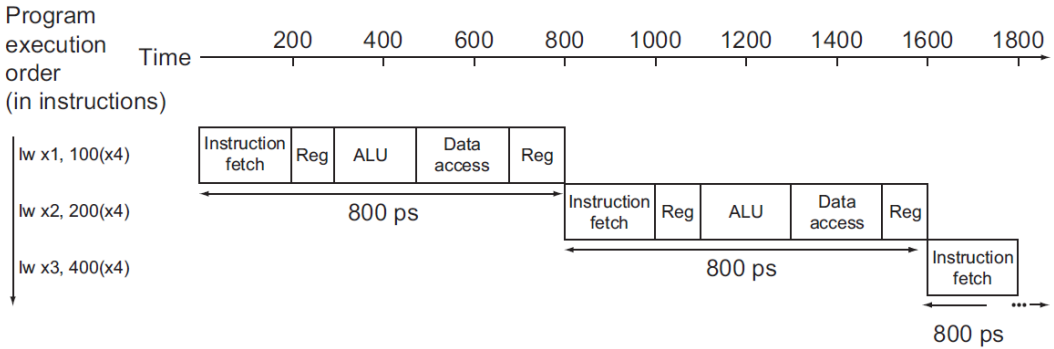
Single-Cycle versus Pipelined Performance

- The single-cycle design must allow for the slowest instruction (lw) so the time required for every instruction is 800 ps.

Instruction class	Instruction fetch	Register read	ALU operation	Data access	Register write	Total time
Load word (lw)	200 ps	100 ps	200 ps	200 ps	100 ps	800 ps
Store word (sw)	200 ps	100 ps	200 ps	200 ps		700 ps
R-format (add, sub, and, or)	200 ps	100 ps	200 ps		100 ps	600 ps
Branch (beq)	200 ps	100 ps	200 ps			500 ps

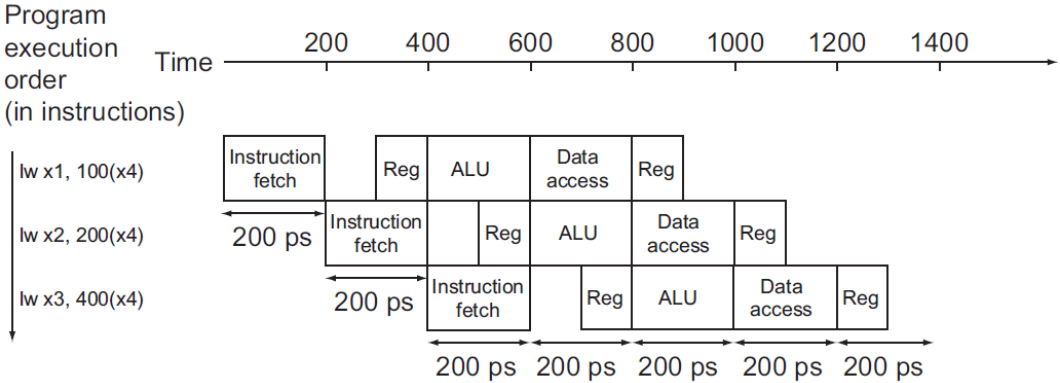
Single-Cycle versus Pipelined Performance

- The time between the first and fourth instructions in the nonpipelined design is 3 × 800 ps or 2400 ps.



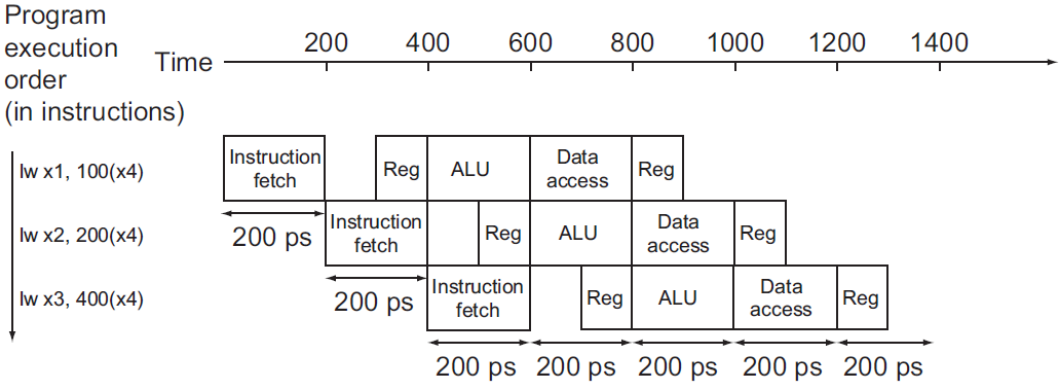
Single-Cycle versus Pipelined Performance

- All the pipeline stages take a single clock cycle, so the clock cycle must be long enough to accommodate the slowest operation (200 ps).

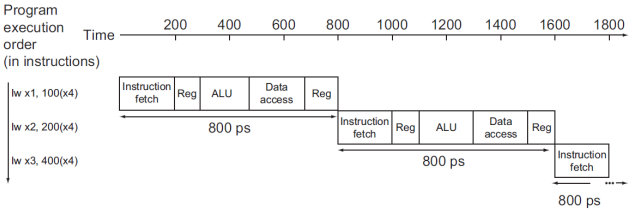


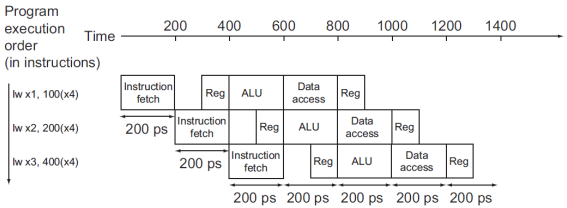
Single-Cycle versus Pipelined Performance

- Pipelining still offers a fourfold performance improvement: the time between the first and fourth instructions is 3 × 200 ps or 600 ps.



Single-Cycle versus Pipelined Performance





Single-Cycle versus Pipelined Performance

- If the stages are perfectly balanced, then the time between instructions on the pipelined processor—assuming ideal conditions—is equal to:

$$\text{Time between instructions}_{\text{pipelined}} = \frac{\text{Time between instructions}_{\text{nonpipelined}}}{\text{Number of pipe stages}} \quad (1)$$

- The formula suggests that a five-stage pipeline should offer nearly a fivefold improvement over the 800 ps nonpipelined time, or a 160 ps clock cycle.
- However, the stages may be imperfectly balanced.
- Thus, the time per instruction in the pipelined processor will exceed the minimum possible, and speed-up will be less than the number of pipeline stages.

Single-Cycle versus Pipelined Performance: Example

- What would happen if we increased the number of instructions?
- If we add 1,000,000 instructions in the previous pipelined example;
- Each instruction adds 200 ps to the total execution time.
- The total execution time would be 1,000,000 × 200 ps + 1400 ps, or 200,001,400 ps.
- In the nonpipelined example, if we add 1,000,000 instructions, each taking 800 ps, the total execution time would be 1,000,000 × 800 ps + 2400 ps, or 800,002,400 ps.

Single-Cycle versus Pipelined Performance: Example

- Under these conditions, the ratio of total execution times for real programs on nonpipelined to pipelined processors is close to the ratio of times between instructions:

$$\frac{800,002,400\ ps}{200,001,400\ ps} \approx \frac{800}{200} \approx 4.00 \tag{2}$$

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

Designing Instruction Sets for Pipelining

- RISC-V instruction set was designed for pipelined execution.
- First, all RISC-V 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.
- In an instruction set like the x86, where instructions vary from 1 byte to 15 bytes, pipelining is considerably more challenging.
- Modern implementations of the x86 architecture actually translate x86 instructions into simple operations that look like RISC-V instructions and then pipeline the simple operations rather than the native x86 instructions!

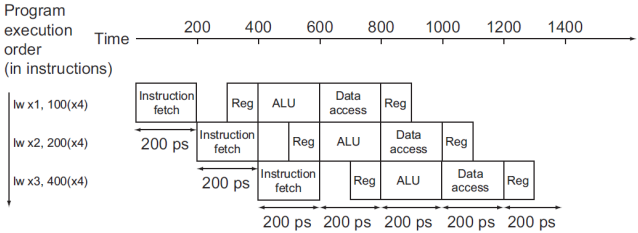
Designing Instruction Sets for Pipelining

- Second, RISC-V has just a few instruction formats, with the source and destination register fields being located in the same place in each instruction.
- Third, memory operands only appear in loads or stores in RISC-V.
- This restriction means we can use the execute stage to calculate the memory address and then access memory in the following stage.
- If we could operate on the operands in memory, as in the x86, stages 3 and 4 would expand to an address stage, memory stage, and then execute stage.

Pipeline Hazards

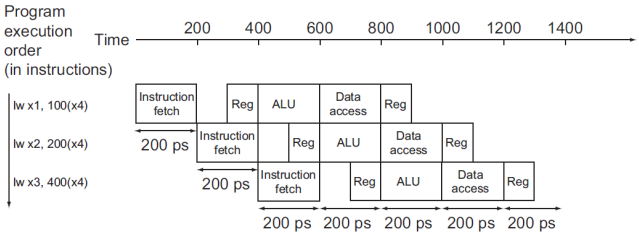
- There are situations in pipelining when the next instruction cannot execute in the following clock cycle.
- These events are called hazards, and there are three different types:
  1. Structural Hazard
  2. Data Hazard
  3. Control Hazard

Structural Hazard



- Suppose that we had a single memory instead of two memories.
- If we 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.

Structural Hazard



- Without two memories, our pipeline could have a structural hazard.
- **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 Hazards

- Suppose we have an add instruction followed immediately by a subtract instruction that uses that sum (x19):

```
add  x19, x0, x1
sub  x2, x19, x3
```

- 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.
- Data hazards occur when the pipeline must be stalled because one step must wait for another to complete.

Data Hazards

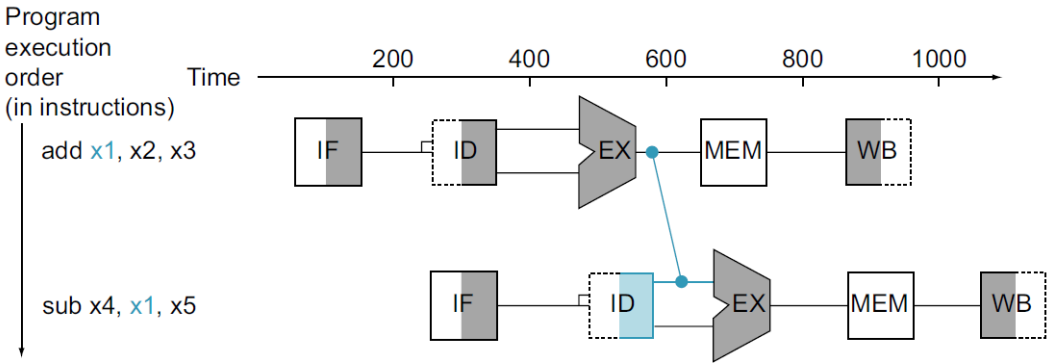
- In a computer pipeline, data hazards arise from the dependence of one instruction on an earlier one that is still in the pipeline.
- **Data hazard** Also called a **pipeline data hazard**. When a planned instruction cannot execute in the proper clock cycle because data that are needed to execute the instruction are not yet available.
- Without intervention, a data hazard could severely stall the pipeline.

ensia

Data Hazards: forwarding or bypassing

- The primary solution is based on the observation that we don't need to wait for the instruction to complete before trying to resolve the data hazard.
- For the code sequence above, as soon as the ALU creates the sum for the add, we can supply it as an input for the subtract.
- Adding extra hardware to retrieve the missing item early from the internal resources is called **forwarding** or **bypassing**.
- **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.

Data Hazards: forwarding or bypassing



Data Hazards: forwarding or bypassing

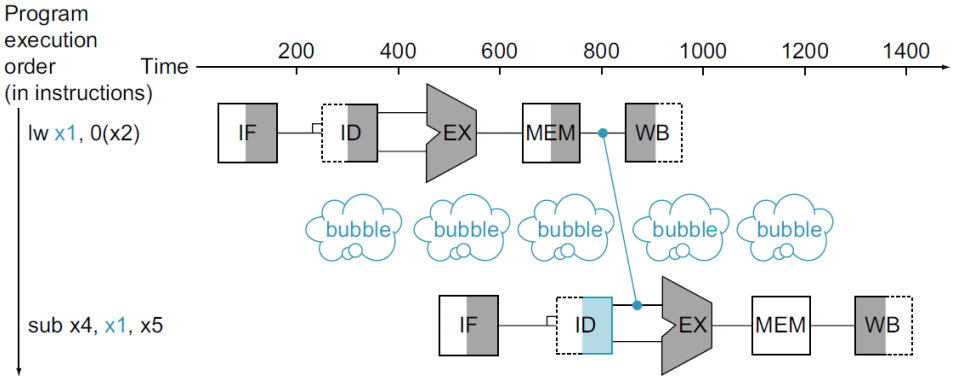
- Forwarding paths are valid only if the destination stage is later in time than the source stage.
- For example, there cannot be a valid forwarding path from the output of the memory access stage in the first instruction to the input of the execution stage of the following, since that would mean going backward in time.
- Forwarding works very well, but cannot prevent all pipeline stalls.

Data Hazards: forwarding or bypassing

- For example, suppose the first instruction was a load of x1 instead of an add.
- The 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.
- Hence, even with forwarding, we would have to stall one stage for a load-use data hazard.

Data Hazards: forwarding or bypassing

- This pipeline concept is officially called a **pipeline stall**, but it is often given the nickname **bubble**.



Data Hazards: forwarding or bypassing

- Forwarding yields another insight into the RISC-V architecture, in addition to the three mentioned above.
- Each RISC-V instruction writes at most one result and does this in the last stage of the pipeline.
- Forwarding is harder if there are multiple results to forward per instruction or if there is a need to write a result early on in instruction execution.
- The name forwarding comes from the idea that the result is passed forward from an earlier instruction to a later instruction.
- Bypassing comes from passing the result around the register file to the desired unit.

Data Hazards: Reordering Code to Avoid Pipeline Stalls

- **Where are the hazards?**

```
a = b + e;
c = b + f;
```

```
lw x1, 0(x31) # Load b
lw x2, 8(x31) # Load e
add x3, x1, x2 # b + e
sw x3, 24(x31) # Store a
lw x4, 16(x31) # Load f
add x5, x1, x4 # b + f
sw x5, 32(x31) # Store c
```

Data Hazards: Reordering Code to Avoid Pipeline Stalls

- Both add instructions have a hazard because of their respective dependence on the previous lw instruction.

```
a = b + e;
c = b + f;
```

```
lw x1, 0(x31) # Load b
lw x2, 8(x31) # Load e
add x3, x1, x2 # b + e
sw x3, 24(x31) # Store a
lw x4, 16(x31) # Load f
add x5, x1, x4 # b + f
sw x5, 32(x31) # Store c
```

Data Hazards: Reordering Code to Avoid Pipeline Stalls

- Moving up the third lw instruction to become the third instruction eliminates both hazards.
- On a pipelined processor with forwarding, the reordered sequence will complete in two fewer cycles than the original version.

```
lw    x1, 0(x31)
lw    x2, 4(x31)
lw    x4, 8(x31)
add   x3, x1, x2
sw    x3, 12(x31)
add   x5, x1, x4
sw    x5, 16(x31)
```

Control Hazards

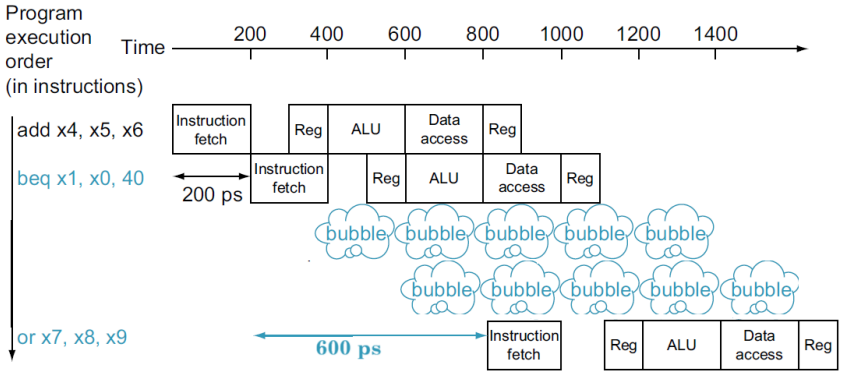
- **Control hazard** Also called **branch hazard**: when the proper instruction cannot execute in the proper pipeline clock cycle because the instruction that was fetched is not the one that is needed; that is, the flow of instruction addresses is not what the pipeline expected.
- It arises from the need to make a decision based on the results of one instruction while others are executing.
- The equivalent decision task in a computer is the conditional branch instruction.

Control Hazards: Stall

- Notice that we must begin fetching the instruction following the branch on the following 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.
- This conservative option certainly works, but it is slow.

Control Hazards: Stall

- The instruction to be executed if the branch fails is stalled one extra 400 ps clock cycle before starting.



Control Hazards: Prediction

- A second solution to the control hazard is **Prediction**.
- Computers use prediction to handle conditional branches.
- One simple approach is to predict always that conditional branches will be untaken.
- When you’re right, the pipeline proceeds at full speed.
- Only when conditional branches are taken does the pipeline stall.

Control Hazards: Prediction

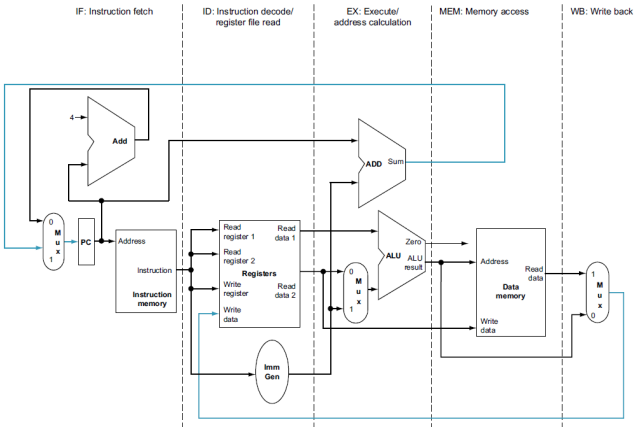
- At the bottom of loops are conditional branches that branch back to the top of the loop.
- Since they are likely to be taken and they branch backward, we could always predict taken for conditional branches that branch to an earlier address.

Control Hazards: Dynamic Prediction

- **Dynamic hardware predictors**: make their guesses depending on the behavior of each conditional branch and may change predictions for a conditional branch over the life of a program.
- One popular approach to dynamic prediction of conditional branches is keeping a history for each conditional branch as taken or untaken, and then using the recent past behavior to predict the future.
- When the guess is wrong, the pipeline control must ensure that the instructions following the wrongly guessed conditional branch have no effect and must restart the pipeline from the proper branch address.

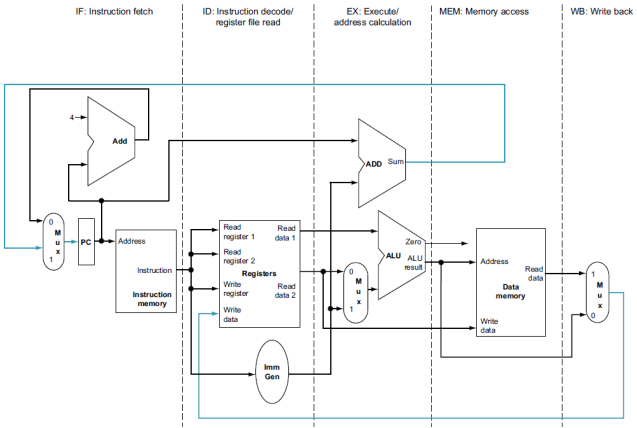
Pipelined Datapath

- Instructions and data move generally from left to right.
- Two exceptions: write-back stage and the selection of the next value of the PC.



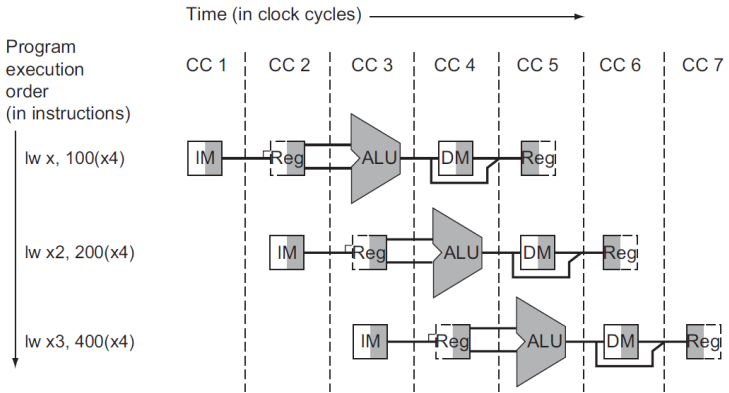
Pipelined Datapath

- Data flowing from right to left does not affect the current instruction, but later instructions in the pipeline.
- Note that the first right-to-left flow of data can lead to data hazards and the second leads to control hazards.



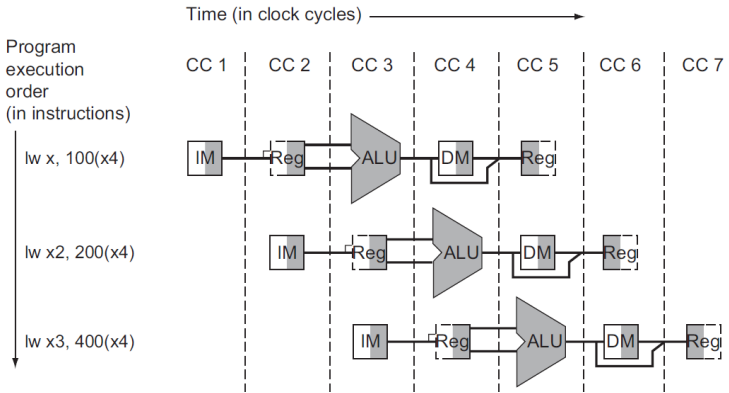
Pipelined Datapath

- For example the instruction memory is used during only one of the five stages of an instruction, allowing it to be shared by following instructions during the other four stages.



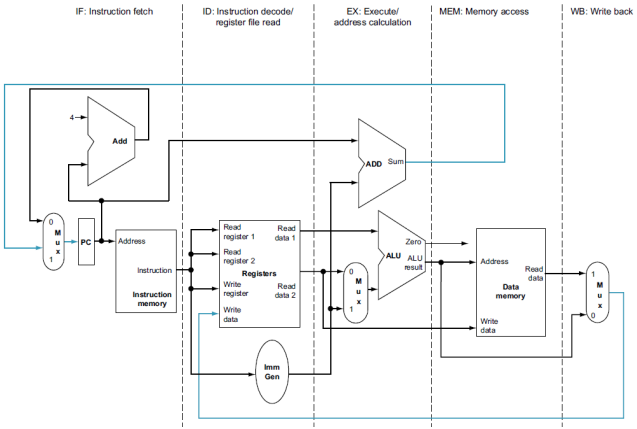
Pipelined Datapath

- To retain the value of individual instruction for its other four stages, the value read from instruction memory must be saved in a register.

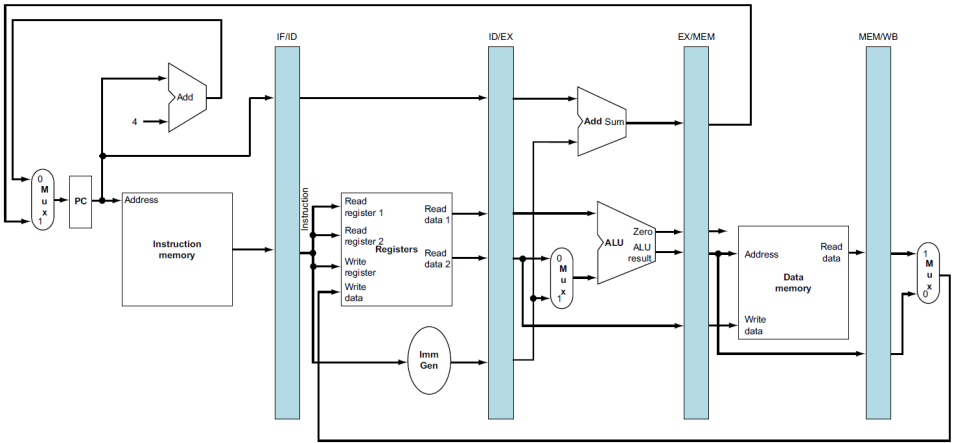


Pipelined Datapath

- Similar arguments apply to every pipeline stage, so we must place registers wherever there are dividing lines between stages in the following figure.

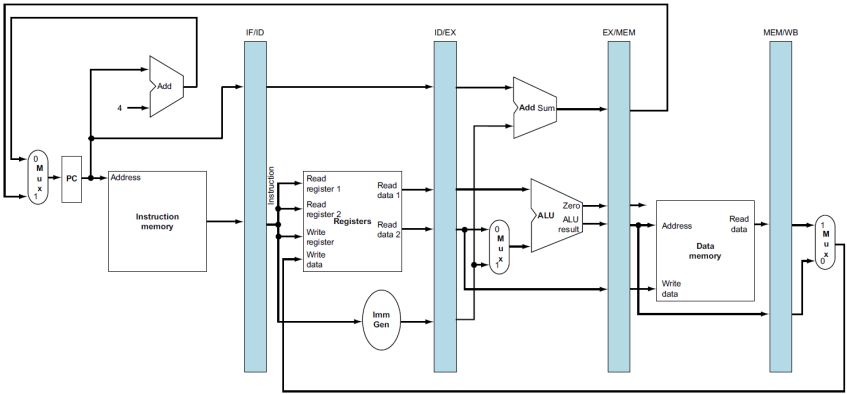


Pipelined Datapath



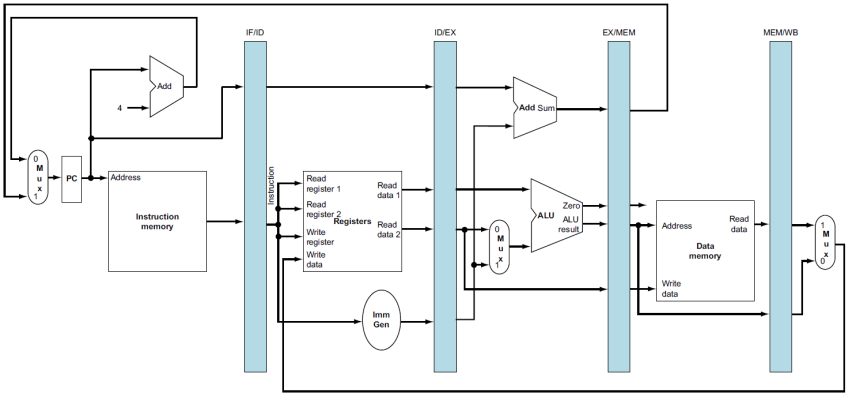
Pipelined Datapath

- All instructions advance during each clock cycle from one pipeline register to the next.



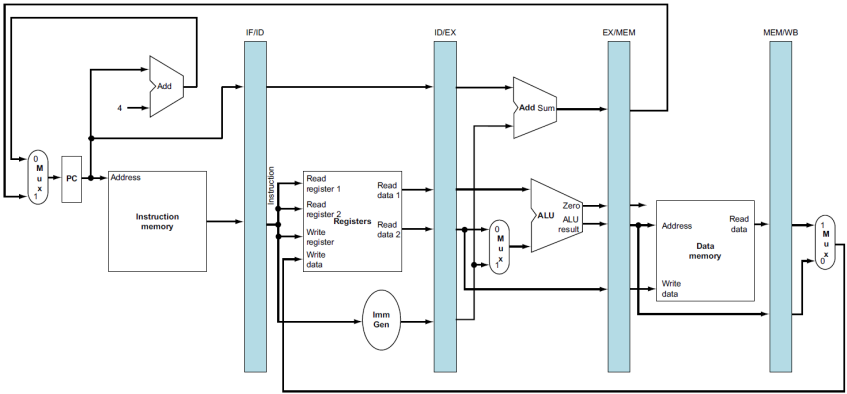
Pipelined Datapath

- 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.



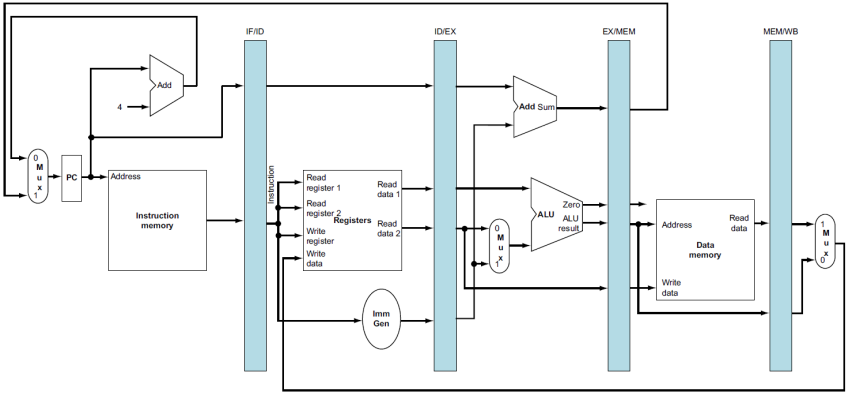
Pipelined Datapath

- Notice that there is no pipeline register at the end of the write-back stage (redundant).



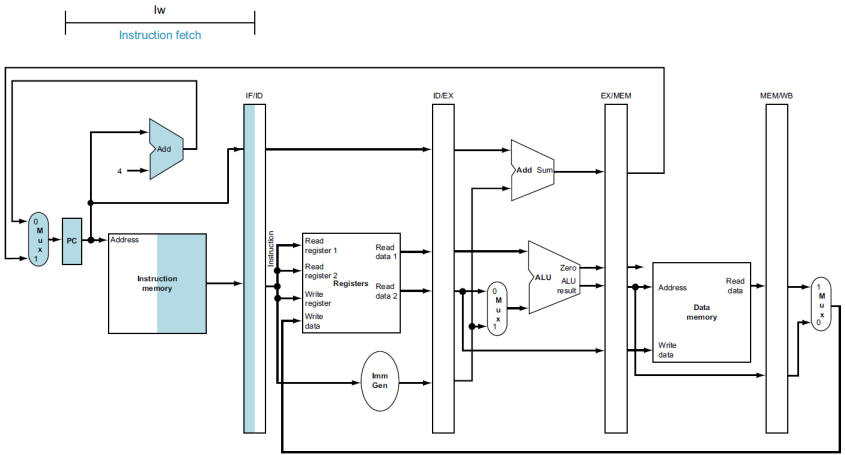
Pipelined Datapath

- We highlight the right half of registers or memory when they are being read and highlight the left half when they are being written.



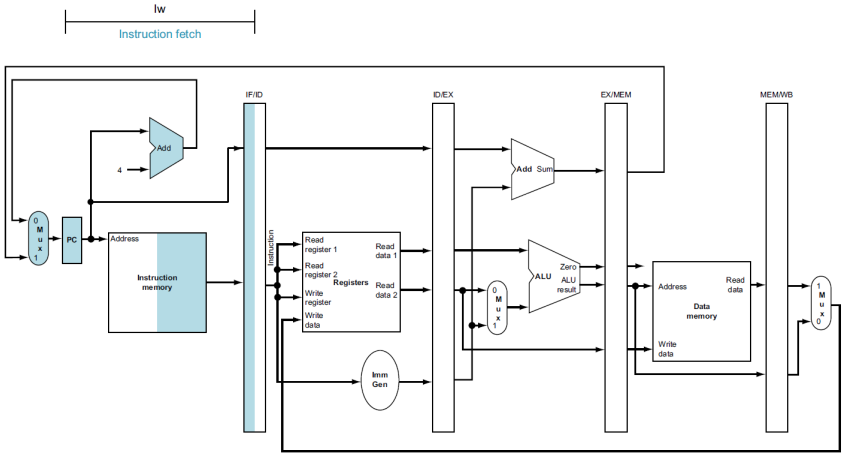
Pipelined Datapath: lw: *Instruction fetch*

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



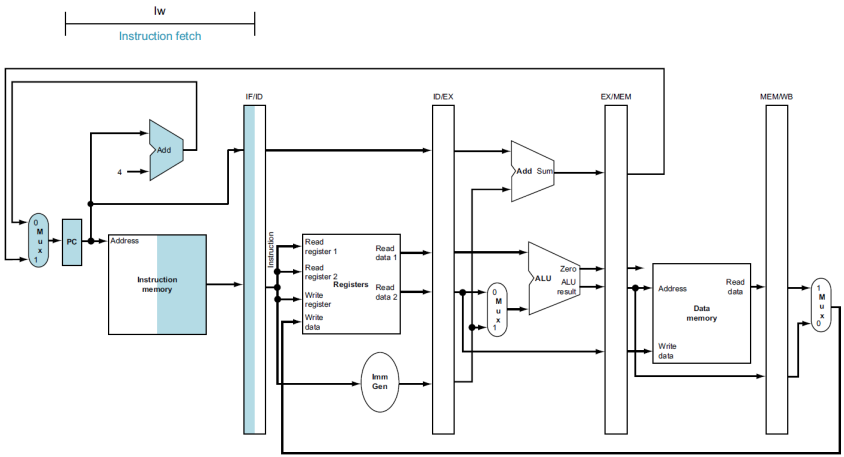
Pipelined Datapath: lw: *Instruction fetch*

- The PC address is incre-mented by 4 and then written back into the PC to be ready for the next clock cycle.



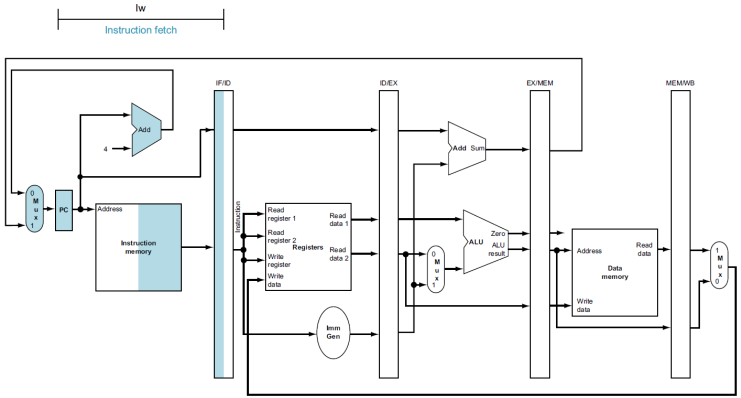
Pipelined Datapath: lw: *Instruction fetch*

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



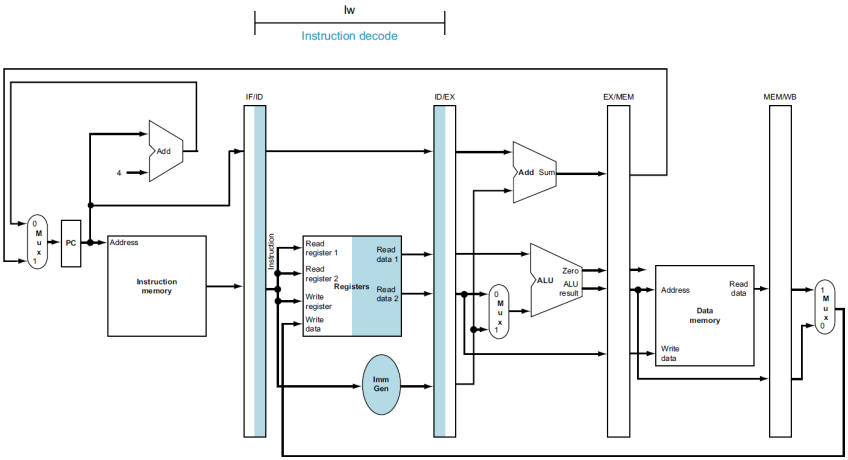
Pipelined Datapath: lw: *Instruction fetch*

- 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.



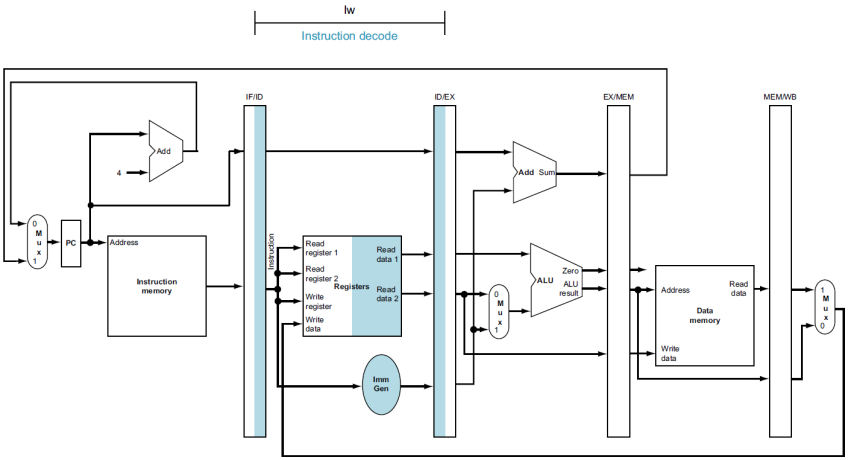
Pipelined Datapath: lw: *Instruction decode*

- The instruction portion of the IF/ID pipeline register supplies the immediate field, which is sign-extended to 32 bits.



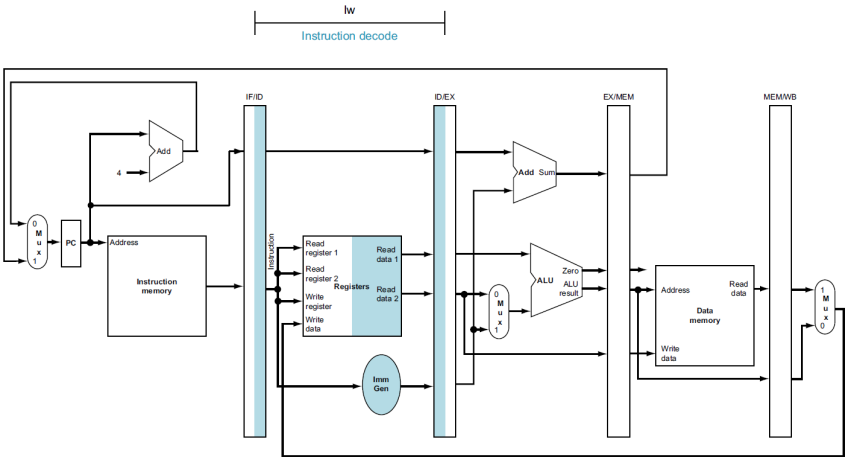
Pipelined Datapath: lw: *Instruction decode*

- It supplies also the register numbers to read the two registers.



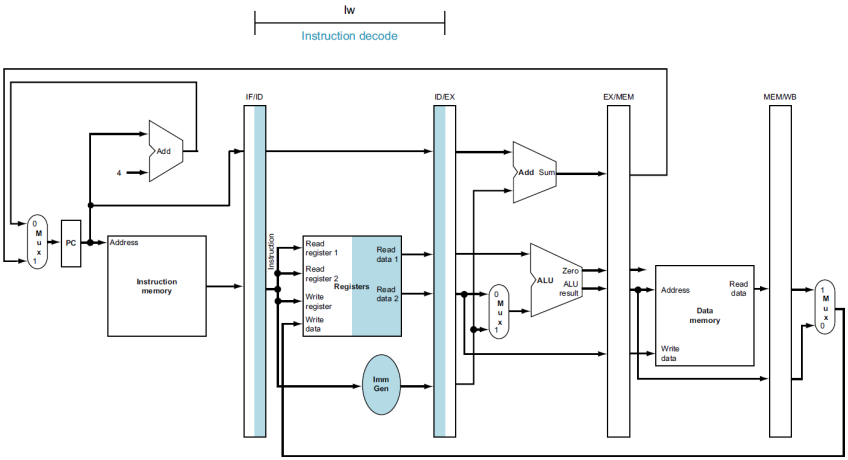
Pipelined Datapath: lw: *Instruction decode*

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



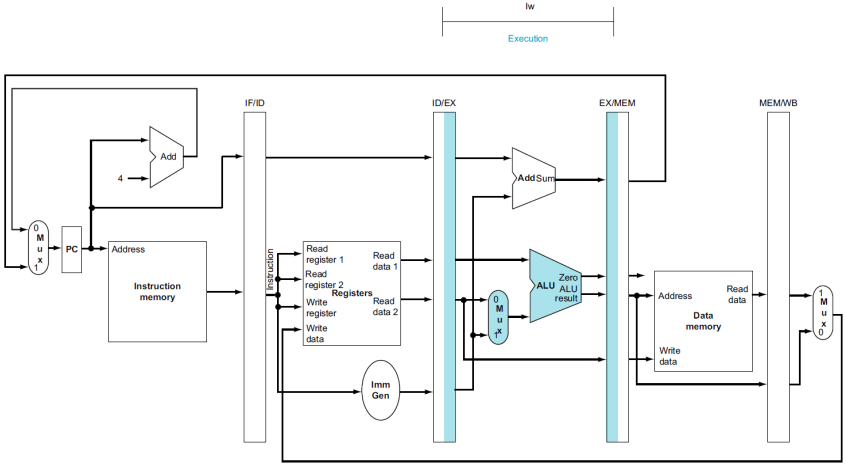
Pipelined Datapath: lw: *Instruction decode*

- We again transfer everything that might be needed by any instruction during a later clock cycle.



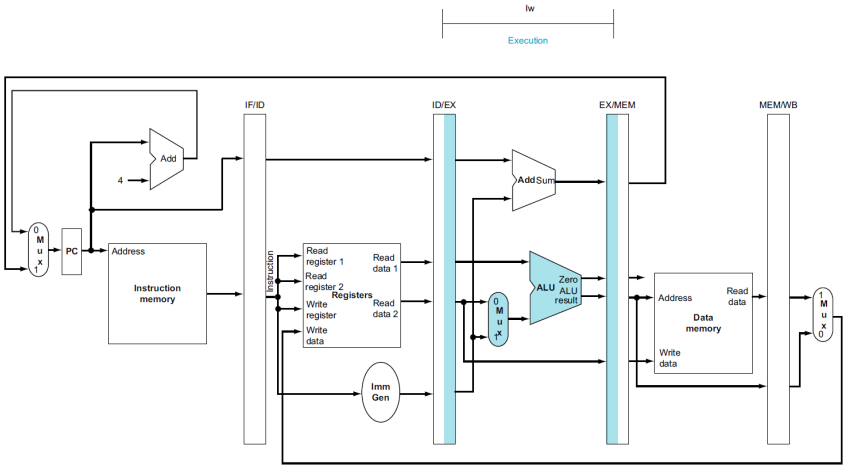
Pipelined Datapath: lw: *Execute*

- lw reads the contents of a register and the immediate from the ID/EX pipeline register and adds them using the ALU.



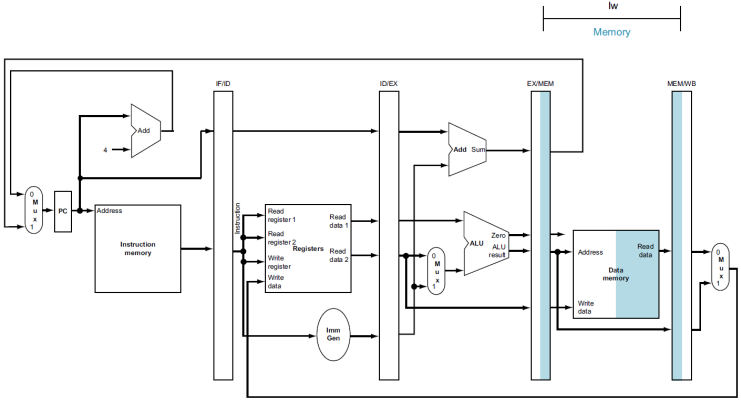
Pipelined Datapath: lw: *Execute*

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



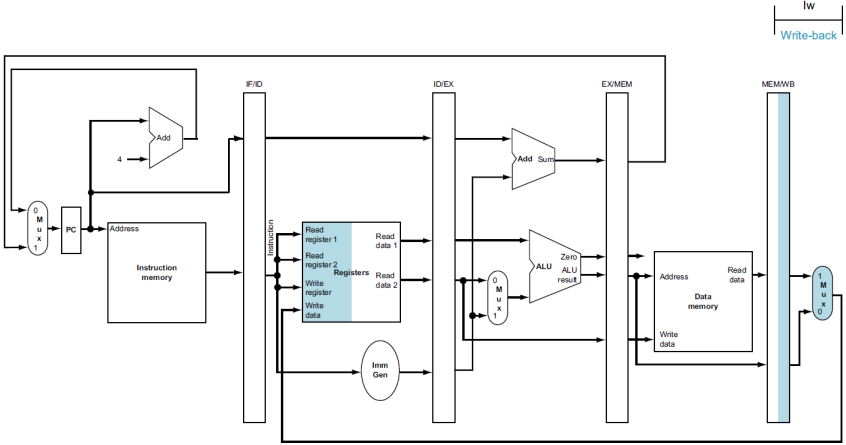
Pipelined Datapath: lw: *Memory access*

- lw reads the data memory using the address from the EX/MEM pipeline register and loading the data into the MEM/WB pipeline register.



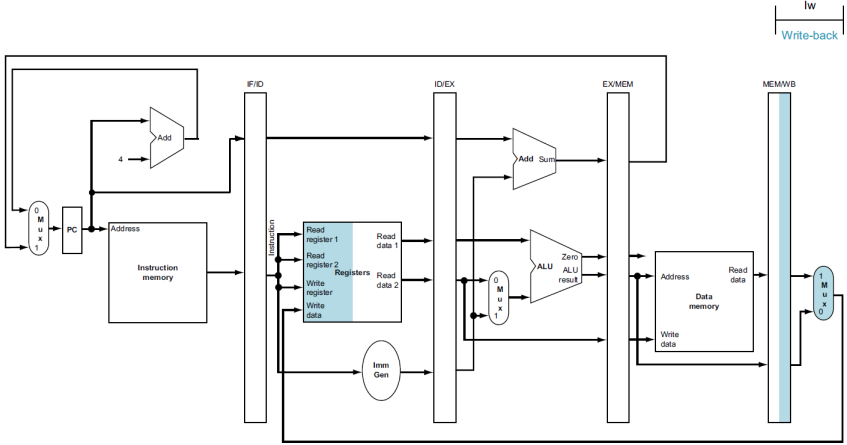
Pipelined Datapath: lw: *Write-back*

- lw reads the data from the MEM/WB pipeline register and writes it into the register file.



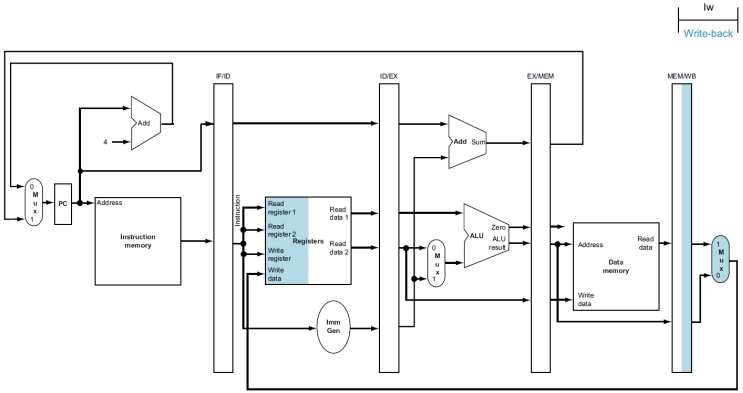
Pipelined Datapath: lw bug

- **Which register is changed in the final stage of the load?**
- **Which instruction supplies the write register number?**



Pipelined Datapath: lw bug

- The instruction in the IF/ID pipeline register supplies the write register number, yet this instruction occurs considerably after the load instruction!

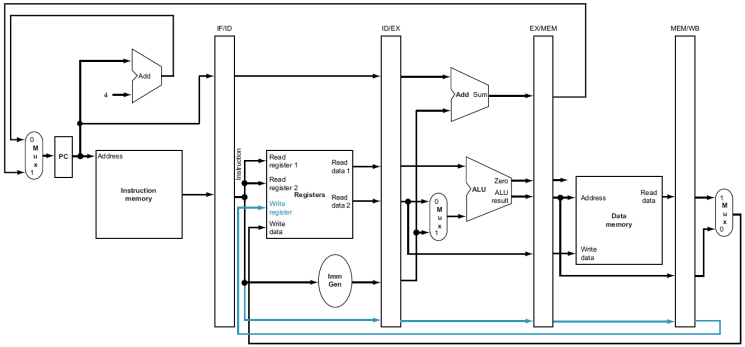


Pipelined Datapath: lw bug fix

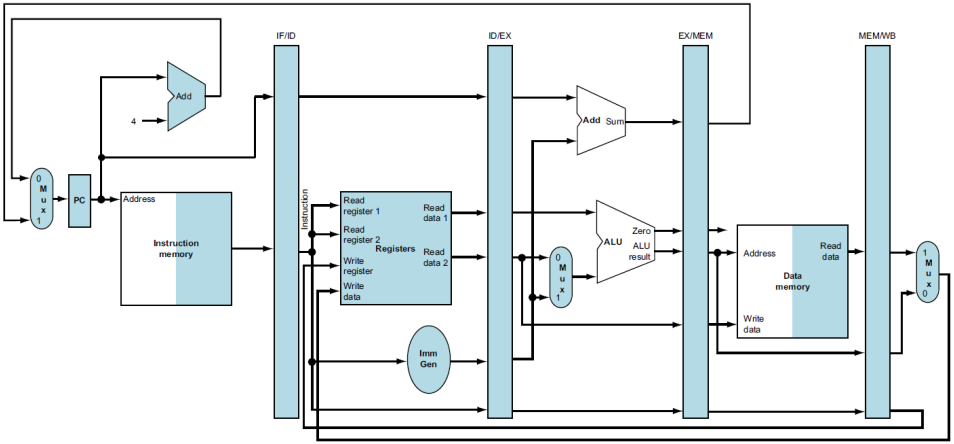
- Hence, we need to preserve the destination register number in the load instruction.
- Load must pass the register number from the ID/EX through EX/MEM to the MEM/WB pipeline register for use in the WB stage.

Pipelined Datapath: lw bug fix

- The correct version of the datapath, passing the write register number first to the ID/EX register, then to the EX/MEM register, and finally to the MEM/WB register.



Pipelined Datapath: lw summary



ensia

Thank you!