**Appendix C**

What Is Pipelining?

Pipelining is an implementation technique whereby multiple instructions are overlapped in execution; it takes advantage of parallelism that exists among the actions needed to execute an instruction.

The throughput of an instruction pipeline is determined by how often an instruction exits the pipeline.

The time required between moving an instruction one step down the pipeline is a processor cycle.

Because all stages proceed at the same time, the length of a processor cycle is determined by the time required for the slowest pipe stage.

Pipelining yields a reduction in the average execution time per instruction. If the starting point is a processor that takes multiple clock cycles per instruction, then pipelining reduces the CPI

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

Purpose of latches?

1)To ensure that instructions in the pipeline do not attempt to use the hardware resources at the same time, 2) we must also ensure that instructions in different stages of the pipeline do not interfere with one another. This separation is done by introducing pipeline registers between successive stages of the pipeline, so that at the end of a clock cycle all the results from a given stage are stored into a register that is used as the input to the next stage on the next clock cycle.3)And also stored intermediated result from one stage to another where the source and destination are not directly not adjacent.

**Basic Performance Issues in Pipelining**

Pipelining increases the CPU instruction throughput - the number of instructions completed per unit of time. But it does not reduce the execution time of an individual instruction. In fact, it usually slightly increases the execution time of each instruction due to overhead in the pipeline control.

The increase in instruction throughput means that a program runs faster and has lower total execution time.

limitations on practical depth of a pipeline arise from:

* Pipeline latency. The fact that the execution time of each instruction does not decrease puts limitations on pipeline depth.
* Imbalance among pipeline stages. Imbalance among the pipe stages reduces performance since the clock can run no faster than the time needed for the slowest pipeline stage;
* Pipeline overhead. Pipeline overhead arises from the combination of pipeline register delay (setup time plus propagation delay) and clock skew (Clock skew is the maximum delay between the time when the clock arrives at any two registers).

Hazards

There are situations, called hazards, that prevent the next instruction in the instruction stream from executing during its designated clock cycle. Hazards reduce the performance from the ideal speedup gained by pipelining. There are three classes of hazards:

1.Structural hazards arise from resource conflicts when the hardware cannot support all possible combinations of instructions simultaneously in overlapped execution.

2. Data hazards (data result of previous instruction is not ready yet) arise when an instruction depends on the results of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

3. Control hazards arise from the pipelining of branches and other instructions that change the PC.

**Minimizing Data Hazard Stalls by Forwarding**

Forwarding means=If the result can be moved from the pipeline register where the add stores it to where the sub needs it, then the need for a stall can be avoided.

forwarding works as follows:

1)The ALU result from both the EX/MEM and MEM/WB pipeline registers is always fed back to the ALU inputs.

2. If the forwarding hardware detects that the previous ALU operation has written the register corresponding to a source for the current ALU operation, control logic selects the forwarded result as the ALU input rather than the value read from the register file

**Data Hazards Requiring Stalls**

The load instruction has a delay or latency that cannot be eliminated by forwarding alone. Instead, we need to add hardware, called a pipeline interlock, to preserve the correct execution pattern.

In general, a pipeline interlock detects a hazard and stalls the pipeline until the hazard is cleared

In this case, the interlock stalls the pipeline, beginning with the instruction that wants to use the data until the source instruction produces it.

**Conversion of latches in pipelined processor**

To control this simple pipeline we need only determine how to set the control for the four multiplexers in the data path of Figure C.19. The two multiplexers in the ALU stage are set depending on the instruction type, which is dictated by the IR field of the ID/EX register. The top ALU input multiplexer is set by whether the instruction is a branch or not, and the bottom multiplexer is set by whether the instruction is a register-register ALU operation or any other type of operation. The multiplexer in the IF stage chooses whether to use the value of the incremented PC or the value of the EX/MEM.ALUOutput (the branch target) to write into the PC. This multiplexer is controlled by the field EX/MEM.cond. The fourth multiplexer is controlled by whether the instruction in the WB stage is a load or an ALU operation. In addition to these four multiplexers, there is one additional multiplexer needed that is not drawn in Figure C.19, but whose existence is clear from looking at the WB stage of an ALU operation. The destination register field is in one of two different places depending on the instruction type (register-register ALU versus either ALU immediate or load). Thus, we will need a multiplexer to choose the correct portion of the IR in the MEM/WB register to specify the register destination field, assuming the instruction writes a register.

**Implementing the Control for the RISC V Pipeline**

The process of letting an instruction move from the instruction decode stage (ID) into the execution stage (EX) of this pipeline is usually called instruction issue; an instruction that has made this step is said to have issued.

For the RISC V integer pipeline

1)All the data hazards can be checked during the ID phase of the pipeline. If a data hazard exists, the instruction is stalled before it is issued. Likewise, we can determine what forwarding will be needed during ID and set the appropriate controls then.

2)Alternatively, we can detect the hazard or forwarding at the beginning of a clock cycle that uses an operand (EX and MEM for this pipeline).

The interlock for a read after write (RAW) hazard with the source coming from a load instruction (called a load interlock) can be implemented by a check in ID.

While the implementation of forwarding paths to the ALU inputs can be done during EX.

1) Solution: If there is a RAW hazard with the source instruction being a load, the load instruction will be in the EX stage when an instruction that needs the load data will be in the ID stage. Once a hazard has been detected, the control unit must insert the pipeline stall and prevent the instructions in the IF and ID stages from advancing. Thus we need only change the control portion of the ID/EX pipeline register to all 0s, which happens to be a no-op.

The key observation needed to implement the forwarding logic is that the pipeline registers contain both the data to be forwarded as well as the source and destination register fields. All forwarding logically happens from the ALU or data memory output to the ALU input, the data memory input, or the zero detection unit.

**Dealing with Branches in the Pipeline**

In RISC V, conditional branches depend on comparing two register values, which we assume occurs during the EX cycle, and uses the ALU for this function.

We will need to also compute the branch target address. Because testing the branch condition and determining the next PC will determine what the branch penalty is, we would like to compute both the possible PCs and choose the correct PC before the end of the EX cycle.

We can do this by adding a separate adder that computes the branch target address during ID. Because the instruction is not yet decoded, we will be computing a possible target as if every instruction were a branch. This is likely faster than computing the target and evaluating the condition both in EX, but does use slightly more energy.

**C.4 What Makes Pipelining Hard to Implement?**

Exceptions: When the instruction order is changed in unexpected ways.

**Dealing with Exceptions** Overlapped execution of instructions make exception handling more difficult to implement.

**Types of Instruction (**rest of thing from slides**)**

The difficult task is implementing interrupts occurring within instructions where the instruction must be resumed.

If a pipeline provides the ability for the processor to

* handle the exception,
* save the state,
* and restart without affecting the execution of the program,

the pipeline or processor is said to be **restart able**.

**Stopping and Restarting Execution**

As in un pipelined implementations, the most difficult exceptions have two properties: (1) they occur within instructions (that is, in the middle of the instruction execution corresponding to EX or MEM pipe stages), and (2) they must be restart able

When an exception occurs, the pipeline control can take the following steps to save the pipeline state safely:

1. Force a trap instruction into the pipeline on the next IF.

2. Until the trap is taken, turn off all writes for the faulting instruction and for all instructions that follow in the pipeline; this can be done by placing zeros into the pipeline latches of all instructions in the pipeline, starting with the instruction that generates the exception, but not those that precede that instruction. This prevents any state changes for instructions that will not be completed before the exception is handled.

3. After the exception-handling routine in the operating system receives control, it immediately saves the PC of the faulting instruction. This value will be used to return from the exception later.

A pipeline is said to have **precise exception**, if the pipeline can be stopped so that the instructions before the faulting instruction are completed and those after it can be restarted from scratch.

**Simpler for integer instructions but is complex to maintain for floating point instructions**

**Two modes of operation**

One mode has precise exceptions and the other (fast or performance mode) does not. Of course, the precise exception mode is slower, since it allows less overlap among floating-point instructions.

What happened if multiple exception occurs in same clock cycle?

If exception occur in-order;

This case can be handled by dealing with example only the data page fault and then restarting the execution. The second exception will reoccur (but not the first, if the software is correct), and when the second exception occurs it can be handled independently.

If exceptions may occur out of order;

The pipeline cannot simply handle an exception when it occurs in time, because that will lead to exceptions occurring out of the un pipelined order. Instead, the hardware posts all exceptions caused by a given instruction in a status vector associated with that instruction.

The exception status vector is carried along as the instruction goes down the pipeline.

Once an exception indication is set in the exception status vector, any control signal that may cause a data value to be written is turned off (this includes both register writes and memory writes). Because a store can cause an exception during MEM.

When an instruction enters WB (or is about to leave MEM), the exception status vector is checked.

If any exceptions are posted, they are handled in the order in which they would occur in time on an un pipelined processor—the exception corresponding to the earliest instruction is handled first. This guarantees that all exceptions will be seen on instruction I before any are seen on I + 1.

Of course, any action taken in earlier pipe stages on behalf of instruction I may be invalid, but because writes to the register file and memory were disabled, no state could have been changed.

**Instruction Set Complications**

No RISC V instruction has more than one result, and our RISC V pipeline writes that result only at the end of an instruction’s execution.

When an instruction is guaranteed to complete, it is called committed. In the RISC V integer pipeline, all instructions are committed when they reach the end of the MEM stage (or beginning of WB) and no instruction updates the state before that stage.

But Some processors have instructions that change the state in the middle of the instruction execution, before the instruction and its predecessors are guaranteed to complete.

Thus, to maintain a precise exception model, most processors with such instructions have the ability to back out any state changes made before the instruction is committed.

If an exception occurs, the processor uses this ability to reset the state of the processor to its value before the interrupted instruction started.

A related source of difficulties arises from instructions that update memory state during execution, such as the string copy operations and if memory state changes it cannot be undo.

To make it possible to interrupt and restart these instructions, the instructions are defined to use the general-purpose registers as working registers. Thus, the state of the partially completed instruction is always in the registers, which are saved on an exception and restored after the exception, allowing the instruction to continue.

A different set of difficulties arises from odd bits of state that may create additional pipeline hazards or may require extra hardware to save and restore.

Condition codes are a good example of this. Many processors set the condition codes implicitly as part of the instruction. This approach has advantages, because condition codes decouple the evaluation of the condition from the actual branch.

But However, implicitly set condition codes can cause difficulties in scheduling any pipeline delays between setting the condition code and the branch.

A final thorny area in pipelining is multicycle operations

These instructions do differ radically in the number of clock cycles they will require, from as low as one up to hundreds of clock cycles. s. These instructions also require different numbers of data memory accesses, from zero to possibly hundreds. The data hazards are very complex and occur both between and within instructions.

The simple solution of making all instructions execute for the same number of clock cycles is unacceptable because it introduces an enormous number of hazards and bypass conditions and makes an immensely long pipeline.

Solution: They pipeline the microinstruction execution; a microinstruction is a simple instruction used in sequences to implement a more complex instruction set.

**C.5 Extending the RISC V Integer Pipeline to Handle Multicycle Operations**

This section concentrates on the basic approach and the design alternatives, closing with some performance measurements of a RISC V floating-point pipeline.

It is impractical to require that all RISC V FP operations complete in 1 clock cycle, or even in 2. 2. Doing so would mean accepting a slow clock or using enormous amounts of logic in the FP units, or both.

Instead, the FP pipeline will allow for a longer latency for operations.

Pipelined FP instructions as same integer instructions, with two important changes.

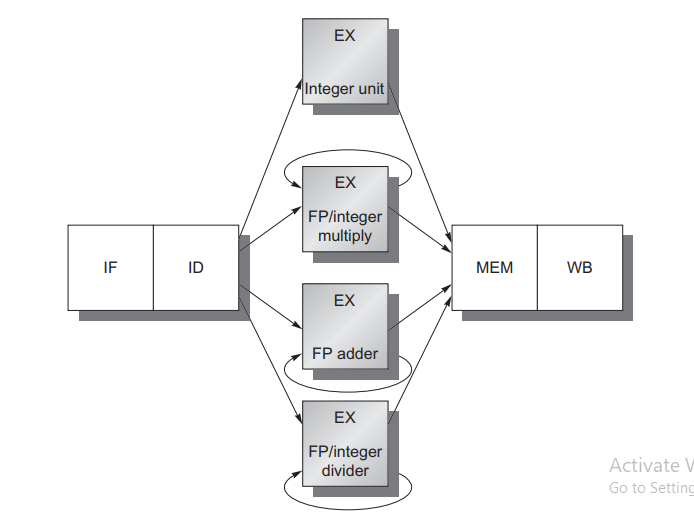
First, the EX cycle may be repeated as many times as needed to complete the operation—the number of repetitions can vary for different operations.

Second, there may be multiple FP functional units.

A stall will occur if the instruction to be issued will cause either a structural hazard for the functional unit it uses or a data hazard.

For this section, let’s assume that there are four separate functional units in our RISC V implementation:

IF Functional Unit are not Pipelined

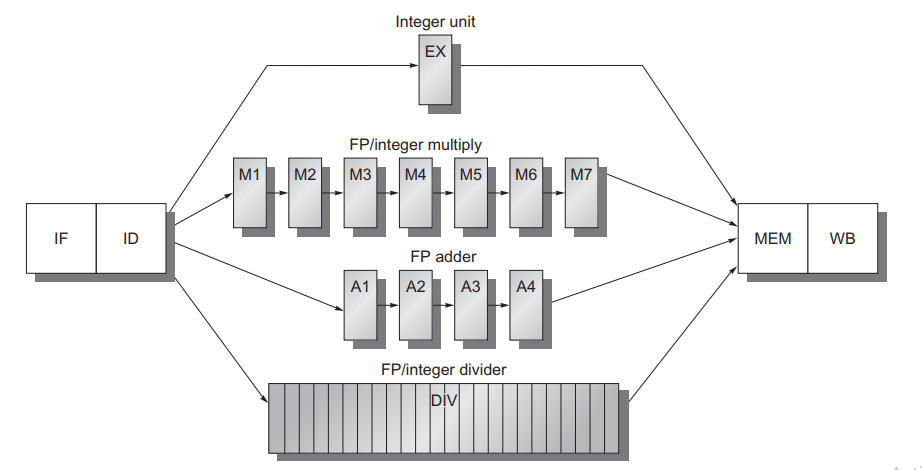


Because EX is not pipelined, no other instruction using that functional unit may issue until the previous instruction leaves EX. Moreover, if an instruction cannot proceed to the EX stage, the entire pipeline behind that instruction will be stalled.

We define **latency** the same way we defined it earlier: the number of intervening cycles between an instruction that produces a result and an instruction that uses the result.

The **initiation or repeat interval** is the number of cycles that must elapse between issuing two operations of a given type.

IF Functional Unit with Pipelined



The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete

Problem with this technique

Naturally, the longer latency of the FP operations increases the frequency of RAW hazards and resultant stalls, as we will see later in this section.

**Hazards and Forwarding in Longer Latency Pipelines**

1. Because the divide unit is not fully pipelined, structural hazards can occur. These will need to be detected and issuing instructions will need to be stalled.

2. Because the instructions have varying running times, the number of register writes required in a cycle can be larger than 1.

3. Write after write (WAW) hazards are possible, because instructions no longer reach WB in order. Note that write after read (WAR) hazards are not possible, because the register reads always occur in ID.

4. Instructions can complete in a different order than they were issued, causing problems with exceptions; we deal with this in the next subsection.

5. Because of longer latency of operations, stalls for RAW hazards will be more frequent.

Now look at the problems arising from writes, described as (2) and (3) in the earlier list.

If we assume that the FP register file has one write port, sequences of FP operations, as well as an FP load together with FP operations, can cause conflicts for the register write port.

**Figure C.**33

Solution: the processor must serialize the instruction completion.

This single register port represents a structural hazard. We could increase the number of write ports to solve this, but that solution may be unattractive because the additional write ports would be used only rarely. This is because the maximum steady-state number of write ports needed is 1. Instead, we choose to detect and enforce access to the write port as a structural hazard.

1)Track the use of write port in the ID stage and stall an instruction before it issues

Implemented with a shift register.

Indicates when already issued instructions will use the register file.

Maintains the property of stalling instructions only in the ID stage

Cost is the shift register and the write conflict logic

2)Stall the conflicting instruction when it tries to enter either MEM or WB stage

Give priority to the unit with the longest latency

Advantage - Detect the conflict at MEM stage where it is easy to see

Disadvantage - Complicates pipeline control as stalls can now arise from two places.

**Our other problem is the possibility of WAW hazards**

f.mul f1, f2, f3

.

.

f.add f1, f4, f5

yahan aega waw hazard q k add pehle complete hoga aur pehle write krdega phir usko mult overwrite krdega.

Solution:

1. **Delay the issue of load instruction until the ADD.D enters MEM stage (bad wale ko rok lo).**
2. **Stamp out the result of ADD.D by detecting the hazard and changing the control so that ADD.D does not write its result (store in r0).**

It is difficult to detect this hazard.

* **Use a simpler solution** **If an instruction in ID wants to write the same register as an instruction already issued, do not issue the instruction.**

Three checks that must be performed in the ID stage before an instruction issue can take place

1. Check for structural hazards:

* Wait until the required functional unit is not busy
* Make sure the register write port is available when it will be needed

ii. Check for a RAW data hazard:

* Wait until the source registers are not listed as pending destinations in a pipeline register

iii. Check for a WAW data hazard:

Determine if any instruction in A1, A2, …, A4; D; M1,.., M7 has the same destination register as this instruction.do not issue that instruction.

**Maintaining Precise Exceptions**

fdiv.d f0,f2,f4

fadd.d f10,f10,f8

fsub.d f12,f12,f14

This is called out-of-order completion and is common in pipelines with long-running operations

This problem arises because instructions are completing in a different order than they were issued.

why is out-of-order completion a problem?

Answer: Because of imprecise exception, instruction issue earlier may trigger exception when the succeeding instruction completed.

There are four possible approaches to dealing with outof-order completion.

1)The first is to ignore the problem and settle for imprecise exceptions. where certain classes of exceptions were not allowed or were handled by the hardware without stopping the pipeline.

Some recent processors have solved this problem by introducing two modes of execution: a fast and precise mode.

2) A second approach is to buffer the results of an operation until all the operations that were issued earlier are complete.

becomes expensive when the difference in running times among operations is large, because the number of results to buffer can become large.

**Two viable variations**

***History file*: Keep track of original values of registers**

***Future file*: Keeps the newer values of registers Update later from the future file.**

**Speculation,** a method of executing instructions before we know the outcome of previous branches.

3) A third technique in use is to allow the exceptions to become somewhat imprecise, but to keep enough information so that the trap-handling routines can create a precise sequence for the exception.

**4)A hybrid scheme:** Allows the instruction issue to continue only if it is certain that all instructions before the issuing instruction will complete without causing an exception. May require to stall the CPU to maintain precise exceptions.

**Performance of a RISC-V Floating Point Pipeline**

* Number of stalls increase due to greater number of hazards
* There are four classes of stalls
  + - FP result stalls
    - FP compare stalls
    - Load and branch delay
    - Floating point structural delays

Compiler tries to schedule both load and FP delays. The total number of stalls per instruction varies from 0.65 to 1.21

**Putting It All Together: The MIPS R4000 Pipeline**

This deeper pipeline allows it to achieve higher clock rates by decomposing the five-stage integer pipeline into eight stages. Because cache access is particularly time critical, the extra pipeline stages come from decomposing the memory access. This type of deeper pipelining is sometimes called super pipelining.