CIS501: Pipelining

Shreyas S. Shivakumar

December 2, 2019

1 Lecture Notes

1. **What are we trying to improve by introducing this optimization (pipelining)?**

   Throughput. In the *laundry example*, pipelining will only enhance performance if you have more than one load of clothes to clean.

2. **If you have 3 stages - washer, dryer and "folding robot"? How long does one load of laundry take? How long does two loads of laundry take? How long would 100 loads take?**

   1 load of laundry will take 3 units of time. 2 loads of laundry will take 4 units of time. 100 loads of laundry will take 104 units of time.

   To understand why 100 loads of laundry will take 104 units of time, understand why 3 loads of laundry will take 5 loads of time. *Intuition:* The case(s) when the number of loads of laundry is $\geq$ the number of pipeline stages.

3. **What is the processor performance equation?**

$$Execution\ Time = \frac{Seconds}{Program} = \frac{I \times S \times C}{P \times C \times I}$$

$$Execution\ Time = \frac{Instructions}{Program} \times \frac{Seconds}{Cycle} \times \frac{Cycles}{Instruction}$$

For *performance optimization*, we want to ideally minimize all three of the above terms, but sometimes they will pull against each other.

Here $\frac{Instruction}{Program}$ refers to the **Dynamic Instruction Count**. It is the instructions that the processor will actually run during the course of a program and not just the static instructions (eg: loops).

4. Briefly discuss the performance of a Single-Cycle Datapath and mention it's limitations?

Everything is done within a single clock cycle. This means that the clock cycle is chosen to be the worst-case delay through the circuit - *critical path*. Therefore, performance is limited by the slowest instruction always.

5. What are the different stages in the pipelined LC4 datapath?

There are five different stages in the LC4 datapath, called the **pipeline depth**, with one instruction in each stage in each cycle:

- Fetch (**F**) :
- Decode (**D**)
- Execute (**X**)
- Memory (**M**)
- Writeback (**W**)

6. How is the clock period selected for this pipelined datapath?

$$Clock\ Period\ =\ MAX(T_F, T_D, T_X, T_M, T_W)$$

7. What is the CPI of the pipelined architecutre?

The base CPI is 1 because instructions enter and leave every cycle, but the actual CPI ≥ 1 since the pipeline might *stall*. The individual instruction **latency** will actually **increase** due to the overhead of the pipelining process, but the benefits of additional throughput make it worth sacrificing latency for.

8. How are the different stages arranged?

The different stages are delimited by the *pipeline registers*, which are named by the stages they begin at.

- PC Register **(PC)** *(Fetch)*
- Decode Register **(D)**
- Execute Register **(X)**
- Memory Register **(M)**
- Writeback Register **(W)**

9. Pipeline Example:

Let's look at the following example of three instructions:

(a) ADD R1, R2, R3 : *where R3 is the destination register*

(b) LOAD 8(R5), R4 : *where R4 is the destination register*

(c) STORE R6, 4(R7) : *where R7+4 is the destination memory address*

Cycle 1:

In the first cycle, we do not yet know that the instruction entering the pipeline is an ADD instruction. This instruction enters the *Fetch Stage* and we identify the address of the next instruction (in *instruction memory*) using the **Program Counter (PC)**. We then go get those bits from the instruction memory.

Cycle 2:

In the second cycle, the ADD instruction enters the *Decode Stage* and proceeds to decode the instruction and read from the *Register File* to get the input values (R1 and R2). The decoding process parses the input instruction to establish all the control signals required for this instruction in the proceeding stages.

In this same cycle, the next instruction (LOAD) is in the *Fetch* stage, doing what the ADD instruction did in the previous cycle.

Cycle 3:

In the third cycle, the ADD instruction enters the *Execute Stage* and proceeds to use the **ALU** to execute an ADD operation on the two operands.

In this same cycle, the LOAD instruction enters the *Decode Stage* and the next instruction (STORE) enters the *Fetch Stage*.

Cycle 4:

In the fourth cycle, the ADD instruction goes through the *Memory Stage* even though there's nothing for it to do there. It needs to write the resulting value to register R3, which it will do in the next step. Can be thought of as a **NO-OP** stage for the ADD instruction.

The other two instructions proceed down the pipeline to *Execute Stage* and *Decode Stage* respectively.

Cycle 5:

In the fifth cycle, the ADD instruction enters the *Writeback Stage*, and then performs a write-back of the ALU result to the *Register File*, selecting the correct destination register.

The other two instructions proceed down the pipeline to *Memory Stage* and *Execute Stage* respectively.

Cycle 6 & Cycle 7:

LOAD and STORE pass through the last two stages of the pipeline over the these two cycles.

10. Pipeline Performance:

In a single-cycle design, if the clock period is **50ns** and CPI = 1, then your effective performance is **50ns/insn**.

In the 5-stage pipeline above, you could naively say the performance is now $\frac{50ns}{5}$ = **10ns**, but this is not the case for the following reasons:

- Not all stages are uniform in the amount of time they will take.
- Pipeline registers add delay to the system
- There are more datapaths in pipelined sytems (bypasses)

Therefore longer (deeper) pipelines show diminishing clock frequency gains. However, in our example, we can assume that our clock period = **10ns** + overheads = **12ns**. And let's say that our CPI = 1 + pipeline penalty = **1.5**. Our effective performance will then be **12ns** $\times$ **1.5 = 18ns/insn**.

11. Dependencies and Hazards

A key challenge in pipelining is the identification and handling of dependencies across multiple different instructions.

Dependence: It is a relationship between any two instructions.

- Data Dependence: If two instructions use the same storage locations (i.e either the same memory address or register)
- Control Dependence: One instruction affects whether another instruction executes at all (i.e such as in a branch instruction)

Hazard: A hazard is a result of the existence of a dependence leading to the possibility of wrong instruction order. The effects of a hazard should not be externally visible. **Stalls** are used to maintain this order by holding a younger instruction in the same stage and propagating an empty instruction (**no-op**) instead. Hazards are undesirable because the stalls that are introduce result in reduction in performance.

12. Data Hazards

Ignoring *control hazards* for now, let us look at Data Hazards, i.e when two instructions use the same storage locations (memory or register).

Example 1:

(a) ADD R1, R2 $\rightarrow$ **R3**

(b) ADD **R3**, R5 $\rightarrow$ R6

Example 2:

(a) ADD R1, R2 $\rightarrow$ **R3**

(b) LOAD 8(**R3**) $\rightarrow$ R4

(c) ADDI 1, **R3** $\rightarrow$ R6

(d) STORE 8, R7 $\rightarrow$ **R3**

At **Cycle 4** of this example, ADD is writing it's result to R3. LOAD has already read R3 two cycles ago, which means that the value that it currently has is outdated! ADDI also read R3 one cycle ago which means that it's value is outdated too. STORE is currently in the process of reading R3 but it is

unclear whether it has the updated value of R3 or not depending on how the *Register File* was implemented (i.e Read-after-Write or Write-after-Read).

13. What are some initial ways we can try fix this problem?

We can fix this problem in two ways:

(a) **Software Interlocks:**

The compiler will identify and put two independent instructions between the two instructions that have a dependence. If no independent instructions exist to re-order between them, a *no-op* is placed.

- For software interlocks, the CPI is still technically 1 but there are now more instructions. For *example*: 20% of instructions require the insertion of 1 **no-op**. 5% of instructions require the insertion of 2 **no-ops**.
- The total number of instructions then goes up to $1+0.20\times1+0.05\times2$ = 1.3 resulting in a 30% slowdown.
- Another problem with software interlocks is the dependence on specific architecture design. These optimizations are pipeline dependent and make it difficult to ensure backward compatibility (e.g if you were to move towards a 7-stage pipeline, this would not work anymore).

(b) **Hardware Interlocks:**

Here, the processor detects the data hazards and fixes them. We are not concerned anymore with compatibility issues.

We can *detect data hazards* by comparing the input register names of the instruction in the **Decode Stage** with output register names of instructions in the **Execute Stage** and the **Memory Stage**.

$$\begin{aligned} stall = &(Decode.IR.src1 == Execute.IR.dest) \;||\\ &(Decode.IR.src2 == Execute.IR.dest) \;||\\ &(Decode.IR.src1 == Memory.IR.dest) \;||\\ &(Decode.IR.src2 == Memory.IR.dest) \end{aligned}$$

If any of these conditions hold *True*, then insert a **no-op** into the pipeline at the next stage. This maneuver is called a **stall** or a **bubble**.

For hardware interlocks, the CPI can be calculated as above. Here 20% of instructions require 1 cycle stall and 5% of instructions require a 2% cycle stall. The effective CPI is still the same as with *software interlocks*, except here the instructions stay at 1 and the CPI increases by 30%.

14. Bypassing

For *example*: say we had an instruction [ADD R1, R2 $\rightarrow$ R3] in the *Memory Stage* and an instruction [LOAD 8(R3) $\rightarrow$ R4] in the *Execute Stage*. Since the LOAD instruction is already in the Execute Stage, this tells us that the current value of R3 that the LOAD instruction has is outdated, since the ADD instruction is just about to update it's value. However, at this very moment, nothing catastrophic has happened yet, even though we know it will in the next cycle if left unchecked.

1. MX Bypass:

What if we were to pass the updated value of R3 from the Memory Stage back to the Execute Stage, so that the LOAD instruction can now possess the latest value for R3?

This is called **Bypassing** or **Forwarding** and this specific case of forwarding information from the *Memory Stage* to the *Execute Stage* is called **MX Bypassing**.

There is indeed an additional hardware overhead of performing a bypass operation since we need extra wires and a multiplexer. But these overheads are negligible in terms of the performance gained. Instead of filling that cycle with a *no-op* we were able to make progress at the cost of minimum hardware latency.

2. WX Bypass:

Similarly, if such a dependence exists across the *Write-back Stage* and the *Execute Stage*, we can perform the same operation and forward this value to the *Execute Stage* with an additional wire and Mux.

- *Which ALU input in the Execute Stage can you bypass to?* You can add similar wires and multiplexers and make bypass operations available

to both inputs of the ALU. Both **ALUinA** and **ALUinB** bypassing is possible.

3. WM Bypass:

Another bypassing technique, although helpful in limited circumstances, is WM Bypass. For *example* if you have instructions [LOAD 8(R2) → R3] and [STORE R3 → 4(R4)]. When the LOAD instruction is in the *Write-back Stage* it writes the updated value of register R3 back to the Register File. However, the STORE instruction needs to write the value from register R3 to memory address 4($R4$). Currently, the STORE instruction contains an outdated value of R3 and hence a bypass from the *Write-back Stage* to the *Memory Stage* makes sense.

However, let us look at an **important contradicting example**:

(a) LOAD 8(R2) → R3

(b) STORE R4 → 4(R3)

This will not work, because when the LOAD instruction is in the *Write-back Stage*, and the STORE is in the *Memory Stage*, the STORE instruction would have already calculated the address in the *Execute Stage* and hence cannot recalculate the address with the updated value of R3 here. In this case, WM Bypassing will not work.

15. Bypass Logic

$$
\begin{aligned}
ALUinA_{MUX} : & \; if \; (Execute.IR.src1 == Memory.IR.dest) \implies 0 \\
& \; if \; (Execute.IR.src1 == Writeback.IR.dest) \implies 1 \\
& \; else \implies 2
\end{aligned}
$$

$$
\begin{aligned}
ALUinB_{MUX} : & \; if \; (Execute.IR.src2 == Memory.IR.dest) \implies 0 \\
& \; if \; (Execute.IR.src2 == Writeback.IR.dest) \implies 1 \\
& \; else \implies 2
\end{aligned}
$$

16. Bypass example:

Cycle	1	2	3	4	5	6	7	8
ADD R2, R3 → R1	F	D	X	M	**W↓**	-	-	-
What goes here?	-	F	D	X	**M**	W	-	-

First, for a **WM Bypass** to occur, we know that the instruction after the ADD instruction must have a dependence on register R1.

Second, this instruction cannot be writing to an address that depends on register R1. For *example* the instruction [STORE R5 → 7(R1)] is not possible.

With this in mind, a suitable instruction could be [**STORE R1 → 8(R5)**].

Cycle	1	2	3	4	5	6	7	8
ADD R2, R3 → R1	F	D	X	M	**W↓**	-	-	-
STORE R1 → 7(R5)	-	F	D	X	**M**	W	-	-

17. Load-to-Use Dependency

Have we prevented all possible data hazards with the above 3 bypassing techniques? **No**. The load-to-use case is an example where bypassing just doesn't cut it and we have to use a **Stall**.

For *example:* consider the following instructions:

(a) LOAD 8(R2) → R3

(b) ADD R2, R3 → R4

When the LOAD instruction is in the *Memory Stage* and the ADD instruction is in the *Execute Stage*, the LOAD instruction cannot forward the updated value of R3 yet since it hasn't accessed the *Data Memory* yet to perform the address calculation.

We need to therefore **identify** this situation early on and prevent the ADD instruction from advancing beyond the *Decode Stage* by inserting a **stall/bubble** into the *Execute Stage*.

This condition can be identified as follows:

$$
\begin{aligned}
Stall = (&Execute.IR.operation == LOAD)\ \&\& \\
&(\ (Decode.IR.src1 == Execute.IR.dest)\ || \\
&\quad ((Decode.IR.src2 == Execute.IR.dest)\&\& \\
&\quad\ (Decode.IR.op! = STORE)) \\
&)
\end{aligned}
$$

Once this condition is detected as above, a stall or bubble can be inserted. In the following cycle, a regular **WX Bypass** can be used.

Performance: Assuming that 50% of loads are followed by dependent instructions (load-to-use), then this will require 1 stall cycle (an insertion of a no-op).

The resulting performance will then be CPI = 1 + (1 × 0.20 × 0.50).

We can also include additional optimization by re-arranging instructions as seen with *software interlocks*, but this time we do it for performance and not for correctness.

18. Structural Hazards:

A structural hazard is when two instructions are trying to use a piece of the circuit at the exact same time. For *example:* two instructions trying to use a write port at the same time.

This can be avoided under the following conditions:

- Each instruction uses every structure in the circuit exactly once
- Each instruction does this for at most one cycle
- All instructions travel through all stages of the circuit

19. Structural Hazards and Multiple-Cycle Operations

Another example of a **Structural Hazard** occurring is when we have operations that don't follow the assumption that all instructions are going to go through all the different stages of the pipeline.

Multi-cycle Multiply:

For *example:* If you have a multiply instruction, it is a high latency operation - *Execute Stage* gets really large because of this. We can then put it through it's own separate pipeline. We can then run the original part of the pipeline faster, but multiply operations will take 4 clock cycles. It has it's own register and lives as it's own unit. We can also make it skip the *Memory Stage* and go straight to the *Write-back Stage*.

Let's look at some of the implications of this approach:

Case 1: *What happens during a regular data dependence condition?*

Cycle	1	2	3	4	5	6	7	8	9
MUL R3, R5 → R4	F	D	P0	P1	P2	P3	**W↓**	-	-
ADDI R4, 1 → R6	-	F	D	d^*	d^*	d^*	**X**	M	W

A sequence of instructions such as the one above will result in 3 stalls!

The stall logic to identify this condition is:

$$\begin{aligned} Stall = &(PreviousStallLogic)|| \\ &(Decode.IR.src1 == PO.IR.dest)||(Decode.IR.src2 == PO.IR.dest)|| \\ &(Decode.IR.src1 == P1.IR.dest)||(Decode.IR.src2 == P1.IR.dest)|| \\ &(Decode.IR.src1 == P2.IR.dest)||(Decode.IR.src2 == P2.IR.dest) \end{aligned}$$

Case 2: *What if two instructions are trying to write to the register file in the same cycle?*

$$\begin{aligned} Stall = &(PreviousStallLogic) \ || \\ &(Decode.IR.dest == "valid") \ \&\& \\ &(Decode.IR.operation \neq MULT) \ \&\& \\ &(P1.IR.dest == "valid") \end{aligned}$$

Here, being *valid* indicates that it is indeed writing to a register since not all instructions must write to a register. You could also avoid this problem by introducing an additional write port, but that requires additional hardware and comes with it's own additional overhead and latency.

Case 3: *Since multiplies take 4 cycles to complete, what if the cycle that comes after it finishes before it?*

Cycle	1	2	3	4	5	6	7	8	9
MUL R3, R5 $\rightarrow$ R4	F	D	P0	P1	P2	P3	**W**	-	-
ADDI R1, 1 $\rightarrow$ R4	-	F	D	X	M	**W**	-	-	-
...	-	-	-	-	-	-	-	-	-
...	-	-	-	-	-	-	-	-	-
ADD R4, R6 $\rightarrow$ R8	-	-	-	-	F	D	X	M	W

The value of $R4$ that the ADD instruction will recieve is most likely to be incorrect because of the mis-ordering.

We can identify this condition as follows:

$$\begin{aligned} Stall = (&PreviousStallLogic) \;||\\ &((Decode.IR.dest == P0.IR.dest) \;||\\ &(Decode.IR.dest == P1.IR.dest)) \end{aligned}$$

Summary:

Multi-cycle operations complicate pipeline logic, but there are definite performance gains to be had. It is an *area-efficient* way to add throughput. However, in some cases such as in int/FP divide, it's difficult to do and may just not be worth it, in which case we should just accept the structural hazard and stall for a few cycles.

2 Textbook Notes

1. Pipelining Paradox:

The time from placing a single instruction into the pipeline till it's finished is **not** faster for pipelining. The reason pipelining **is faster** is because when many instructions go through the pipeline, instructions in different stages of the pipeline can work in parallel and more instructions can be finished in a given unit of time. *Throughput* is what we improve, not *latency*. But *throughput* is important because computers execute billions of instructions!

2. Performance:

Under ideal conditions and with a large number of instructions, the speed-up from pipelining is approximately equal to the number of pipeline stages. For *example* a five-stage pipeline is nearly five times faster.

There are however **start-up** and **wind-down** effects that lower performance when the number of tasks is not large compared to the number of stages. For *example* a 5 stage pipeline with 2 instructions isn't a huge improvement in performance.

Additionally, dependence problems and stall cycles further reduce performance indicating that the speed-up is usually less than the number of pipeline stages.

$$\text{speedup}_{pipeline} \leq \text{number of pipeline stage}$$

$$\text{speedup}_{pipeline} = \#\text{stages} - (\text{start-up \& wind-down} + \text{dependencies})$$

3. Conditions that make pipelining easier:

(a) LC4 instructions are all the same length

(b) LC4 instructions have only a few unique formats

(c) LC4 instructions have only LOAD and STORE memory instructions

(d) LC4 instructions have operands aligned in memory (only one access required)

4. **Does bypassing solve all dependence problems?**

   No, sometimes we just have to take the *stall* and move on. An *example* is the **Load-to-Use** condition.

5. **What are the consequences of allowing JUMP, BRANCH and ALU instructions to take fewer stages than the five required?**

   This would create more problems than it's worth. Additionally, ALU instructions require to go through till the fifth stage since they write back to the destination register. Essentially, we should be looking to optimize our pipeline to require shorter cycles, even if that means splitting the pipeline into more stages.

6. **What happens when a register is read and written in the same clock cycle?**

   One way to resolve this problem is to design the register file to enforce *Read-after-Write (RAW)* which means that the write is performed in the first half of the clock cycle and read is in the second half of the clock cycle.

   Alternatively, we could also have *Write-after-Read (WAR)* and derive our correctness guarantees accordingly.

7. **Bypass all instructions all the time (!?)**

   Some instructions do not write registers, so attempting to add bypassing for all instructions isn't accurate. However, an easily fix for this problem is to check if the *RegisterWrite* signal is **active** for that particular instruction and decide to bypass accordingly.

8. **MX Bypass**

   If the destination register of the instruction in the X→**M** stage is the same register as any one of the source registers of the instruction in the D→**X** stage, **and** if a *RegisterWrite* signal is enabled, then a **MX Bypass** is required.

Cycle	1	2	3	4	5	6	7	8	9
SUB R1, R3 → *R2*	F	D	X	**M↓**	W	-	-	-	-
AND *R2*, R5 → R12	-	F	D	**X**	M	W	-	-	-

9. WX Bypass

If the destination register of the instruction in the M→**W** stage is the same register as any one of the source registers of the instruction in the D→**X** stage, **and** if a *RegisterWrite* signal is enabled, then a **WX Bypass** is required.

Cycle	1	2	3	4	5	6	7	8	9
SUB R1, R3 → *R2*	F	D	X	**M↓**	**W↓↓**	-	-	-	-
AND *R2*, R5 → R12	-	F	D	**X**	M	W	-	-	-
OR R6, *R2* → R13	-	-	F	D	**X**	M	W	-	-

10. WM Bypass

This is a more specific bypass mechanism when you have a LOAD instruction immediately followed by a STORE instruction. This is usually seen in memory-to-memory copies.

11. Load-to-Use

There is a case where bypassing cannot save the day. This occurs when a load instruction is followed by an instruction that tries to read from the same register that the load instruction writes to.

To detect this condition: check to see if the instruction is a load instruction and check if the destination register field of the load in the *Execute Stage* matches either of the source registers in the *Decode Stage*. Once this condition is detected, a **bubble** is inserted to stall the pipeline for one cycle.