Chapter 5: Processor Performance and Pipelining

Space and Time Metrics

`Execution time = (∑ CPIj) * clock cycle time`, where `1 ≤ j ≤ n`

- CPI = `clocks per instruction`
- Sometimes, it also helps to think about this in terms of average clocks per cycle
- Alternatively `Execution time = n * CPIAvg * clock cycle time`

Since clock cycle time changes frequently, cycle count is a more apt measure of performance

- Key takeaway is processor performance ≠ speed
- Cycle count (product of the number of instructions executed and the CPI for each instruction)

Instruction Frequency

- We want to know how often a particular instruction occurs in programs
- **Static instruction frequency** = number of times a particular instruction occurs in the compiled code
  - Impact on memory footprint
  - Try to optimize amount of space it occupies in memory by clever instruction encoding
- **Dynamic instruction frequency** = # of times a particular instruction is actually executed when run
  - Try to make enhancements to datapath to minimize CPI taken for its instruction

Example 2:
Consider the program shown below that consists of 1000 instructions.

I_1:
I_2:
..
..

..
I_{10}:
I_{11}: ADD
I_{12}:
I_{13}:
I_{14}: COND BR I_{10}

} loop

..
..
I_{1000}:

ADD instruction occurs exactly once in the program as shown. Instructions I_{10}-I_{14} constitute a loop that gets executed 800 times. All other instructions execute exactly once.

Example 2:
Consider the program shown below that consists of 1000 instructions.

I_1:
I_2:
..
..

..
I_{10}:
I_{11}: ADD
I_{12}:
I_{13}:
I_{14}: COND BR I_{10}
} loop
..
..
I_{1000}:

ADD instruction occurs exactly once in the program as shown. Instructions I_{10}-I_{14} constitute a loop that gets executed 800 times. All other instructions execute exactly once.

(a) What is the static frequency of ADD instruction?

Answer:

The memory footprint of the program is 1000 instructions. Out of these 1000 instructions, Add occurs exactly once.

Hence the static frequency of Add instruction = 1/1000 * 100 = 0.1%

(b) What is the dynamic frequency of ADD instruction?

Answer:

Total number of instructions executed = loop execution + other instruction execution
= (800 * 5) + (1000-5) * 1
= 4995

Add is executed once every time the loop is executed.
So, the number of Add instructions executed = 800
Dynamic frequency of Add instruction
= (number of Add instructions executed/total number of instructions executed) * 100
= (800 / (995+4000)) * 100 = 16%

Benchmarks

- **Benchmarks** are a set of programs that are representatiave of a workload for a processor
- **Kernels** of real programs are often used as benchmarks
  - Matrix Multiplication for scientific applications

Metrics for measuring performance

Execution time	$(\sum CPI_j)$ * clock cycle time, where $1 \le j \le n$	Seconds	Running time of the program that executes *n* instructions
Arithmetic mean	$(E_1+E_2+\ldots+E_p)/p$	Seconds	Average of execution times of constituent *p* benchmark programs
Weighted Arithmetic mean	$(f_1*E_1+f_2*E_2+\ldots+f_p*E_p)$	Seconds	Weighted average of execution times of constituent *p* benchmark programs
Geometric mean	p^{th} root $(E_1*E_2*\ldots*E_{p)}$	Seconds	p^{th} root of the product of execution times of *p* programs that constitute the benchmark

This chart is also missing

1. Total execution time = cumulative total of the execution times of the individual programs
2. Harmonic Mean = 1/arithmetic mean of reciprocals (e.g. `1/[(1/E1+1/E2+...+1/Ep)/p]`)

- Arithmetic mean tends to bias result towards high values
- Harmonic mean tends to bias towards low values
- Geometric mean tends to be somewhere in the middle

One widely accepted benchmark is called the *Standard Performance Evaluation Corporation (SPEC)*

- Metrics are often compared to reference machine
- `SPECratio = execution time on reference machine / execution time on target machine`

Increasing the Processor Performance

1. Increasing the clock speed
2. Datapath organization leading to a lower CPI

Clock speed is determined by worst-case delay in datapath

- Arrange elements such that delay is reduced (e.g. physically closer)
- Reduce # of datapath actions taken in a single clock cycle

- Feature size of the individual datapath elements should be shrunk
- Requires coming up with new chip fabrication processes and device technologies that help in reducing the feature size
- May impact CPI

Multiple buses can help to increase hardware concurrency

- Reduce CPI for each instruction
- May impact clock cycle time

Alternatively, we can also **reduce the number of executed instructions**

- Replace simple instructions with complex ones
- Has to be balanced against CPI, clock cycle time, and dynamic instruction frequencies

Speedup

$$Speedup_{AoverB} = \frac{\text{Execution Time on Processor B}}{\text{Execution Time on Processor A}}$$

- Speedup of processor A over processor B is the ratio of execution time on processor B to the execution time on processor A

$$Speedup_{improved} = \frac{\text{Execution Time Before Improvement}}{\text{Execution Time After Improvement}}$$

- Speedup due ot a modification

Amdahl's law suggests an engineering approach to improving processor performance, namely, spend resources on critical instructions that will have the maximum impact on the execution time

```
Time_after = Time_unaffected + Time_affected/x
```

where `x` = amount of improvement

Introduction to Pipelining

Sandwich shop example

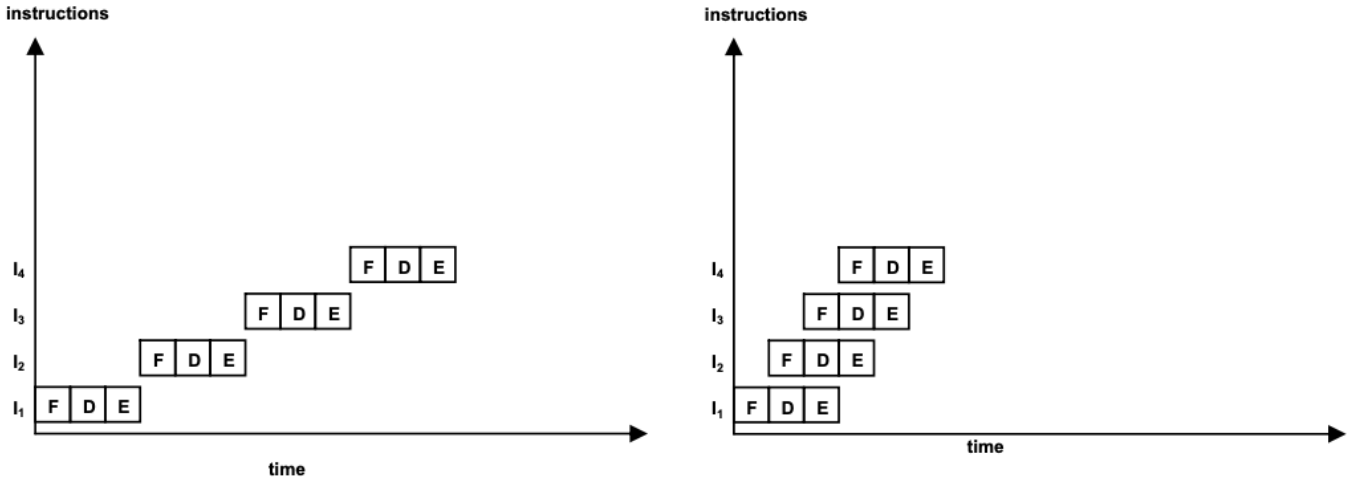
Place Order	Select Bread	Cheese	Meat	Veggies
Station 1	Station 2	Station 3	Station 4	Station 5

Instead of dedicating each station to a single customer (requiring you to stash all ingredients needed to make a sandwich at each station), you pass the sandwich along the line

Towards an instruction processing assembly line

Right now, we are doing FETCH -> DECODE -> EXECUTE

- Not all resources are being used at each stage
- Only IR must be used every macro state
- Approximately equal work being done in each state



(a): Non-pipelined **(b): Pipelined**

Problems With a Simple-minded Instruction Pipeline

1. Different stages often need the same datapath resources (ALU, IR)
2. Amount of work done in the different stages is not the same (e.g. sometimes it's just a combinational function, other times we need to read/write to memory)

To fix this problem, we propose a 5-stage pipeline

1. IF
2. ID/RR
3. EX

4. MEM
5. WB

IF: fetches the instruction pointed to by the PC from I-MEM and places it into IR; it also increments the current PC in readiness for fetching the next instruction

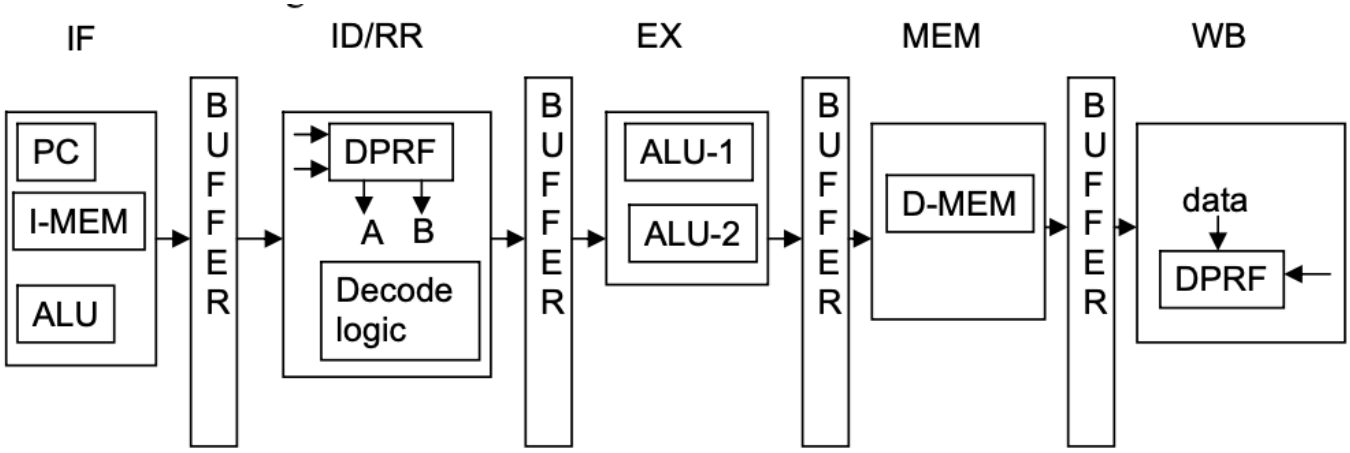
ID/RR: decodes the instruction and reads the register files to pull out two source operands. To enable this functionality, the register file has to be **dual-ported**

EX: does all the arithmetic and/or logic operations

MEM: reads/writes to the D-MEM for LW and SW instructions. Instructions that do not have a memory operand will not need operations in this stage

WB: writes the appropriate destination register (`Rx`) if required

Since each stage is working on a different instruction, once a stage has completed its function it has to place the results of its function in a well-known place for the next stage to pick it up in the next clock cycle. This is called buffering the output of a stage



Passage of instructions through the pipeline

Every stage works on the partial results generated in the *previous clock cycle* by the preceding stage and not every instruction needs every stage

- This design increases latency but also increases throughput

Since each stage is working on a different instruction, once a stage has completed it has to place the results in a well-known place for the next stage to pick it up. This is called ***buffering*** the output of a stage

Datapath Elements For The Instruction Pipeline

1. **IF Stage**: Need PC, ALU, and I-MEM. Pipeline register between IF and ID/RR stages should contain the instruction
2. **ID/RR Stage**: Need DPRF. Output of this stage are the contents of the registers read from the register file (call them A and B); and the results of decoding the instruction (opcode of the instruction, and offset if needed by the instruction).
3. **EX Stage** performs any needed arithmetic for an instruction. Since this is the only stage that performs all the arithmetic for an instruction execution, we need to determine the worst-case resource need for this stage. For BEQ, we need one ALU to do the comparison (`A == B`) and another one to do the computation (`PC + Offset`)
4. **MEM Stage**: Need D-MEM. For LW instruction, the output buffer of the stage contains the D-MEM contents read, the opcode, and the destination register specifier (`Rx`); while for SW instruction the output buffer contains the opcode
5. **WB Stage**: Needs DPRF. We know every stage is working on a different instruction, but since we are only reading in the previous stages, it's okay to write (kinda).

Pipeline Conscious Architecture and Implementation

The key things to note are

- **Need for a symmetric instruction format**: This has to do with ensuring that the locations of certain fields in the instruction
  - Need to ensure equal amount of work in each stage: This property ensures that the clock cycle time is optimal since the slowest member of the pipeline determines it

Hazards

Recall that the pipeline is synchronous. If a stage isn't ready to send a valid instruction to the next one, we send a NOP, kinda like placing an air bubble through a water pipe.

Structural Hazard

Limitations in the hardware resources available for concurrent operation of the different stages

- Only one ALU
  - BEQ requires two cycles now, we send one NOP
- Only one bus

Data Hazard

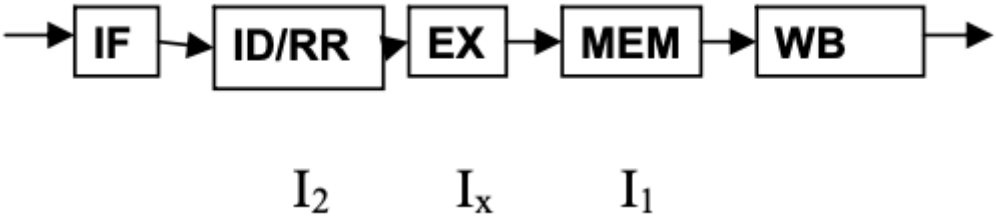
1. Read After Write (RAW)

2. Write After Read (WAR)
3. Write After Write (WAW)

Read After Write

```
I1: R1 <- R2 + R3
Ix: R8 <- R6 + R7
I2: R4 <- R1 + R5
```

In this example, the pipeline will look like this



The new value for R1 will only be registered when I1 reaches the WB stage, so I2 has to wait.

Solution: Data Forwarding

A stage that generates a new value for a register looks around to see if any other stage is awaiting this new value. If so, it will forward the value to the stages that need it

- Add a read-pending bit

Register file

	B	RP
	B	RP
	B	RP
	B	RP
	B	RP
	B	RP
	B	RP
	B	RP

When an instruction hits the ID/RR stage, if the B bit is set for a register it wants to read, then it sets the RP bit for the same register. If any of the EX, MEM, or WB stages sees that the RP bit is set for a register for which it is generating a new value then it supplies the generated value to the ID/RR stage.

Data forwarding completely eliminates the RAW problem for add

LD Instruction RAW

```
LD R1, 0(R2)
ADD R4, R1, R4
```

In this case, the new value for R1 is not available until the MEM stage, so even with forwarding, a 1 cycle stall is inevitable

Write After Read

- Not even a problem since the instruction that needs to read the data has already copied the register value into the pipeline buffer when it was in ID/RR stage

Write After Write

```
i1. R2 <- R4 + R7
i2. R2 <- R1 + R3
```

- Just want to make sure R2 has the result of i2 and not i1
- Stall an instruction that needs to write to a register in the ID/RR stage if it finds the B bit set for the register
- Instruction will be able to proceed once the preceding instruction that set the B bit clears it in the WB stage
- Doesn't happen in vanilla pipeline where register writes are in order

Control Hazard

- Breaks in the sequential execution of a program due to branch instructions

Example

```
BEQ
ADD
NAND
```

Cycle	IF	ID/RR	EX	MEM	WB
1	BEQ				
2	ADD	BEQ			
3	ADD+	NOP	BEQ		

ADD instruction is stalled in the IF stage until the BEQ is resolved

The above schedule assumes that the branch was unsuccessful allowing instructions in the sequential path to enter the IF stage once BEQ is resolved. There will be one more cycle delay to access the non-sequential path, if the branch was successful.

Dealing With Branches in Pipelined Processor

1. Delayed Branch (just keep NOPing)
2. Branch Prediction (assume one outcome)
3. Branch Prediction With Target Buffer (BTB)

Each entry of the BTB looks like this:

Address of branch instruction	Taken/Not taken	Address of Target of Branch Instruction

Once a branch is taken, we update the table with this entry. Next time the same branch instruction is encountered this history information helps in predicting the outcome of the branch

One strategy for branch prediction is to predict the branch is likely to be taken if the **target address is lower than the current PC value**. The flip side of this statement is that if the target address is higher than the current PC value, then the prediction is that the branch will not be taken. The motivation behind this strategy is that loops usually involve a backward branch to the top of the loop (i.e., to a lower address), and there is a higher likelihood of this happening since loops are executed multiple times. On the other hand, forward branches are usually associated with conditional statements, and they are less likely to be taken.

Dealing With Program Discontinuities

One method is to

1. Stop sending new instructions into the pipeline (i.e., the logic in the IF stage starts manufacturing NOP instructions to send down the pipeline)

2. Wait until the instructions that are in partial execution complete their execution (i.e. drain the pipe).
3. Go to the interrupt state

The downside to draining the pipe is that the response time to external interrupts can be quite slow especially with deep pipelining, as is the case with modern processors

Another option is to **flush** the pipeline

- Send a signal to all the stages to abandon their respective instructions
- Memory address of the last completed instruction will be used to record in the PC, the address where the program needs to be resumed after interrupt servicing

In reality, most modern processors do not use either of these extremes. Interrupts are caught by a specific stage of the pipe (anything above of it)

Example
Enumerate the steps taken in hardware from the time an interrupt occurs to the time the processor starts executing the handler code in a pipelined processor. Assume the pipeline is drained upon an interrupt.

1. Allow instructions already in the pipeline to complete execution
2. Stop fetching new instructions
3. Start sending NOPs
4. Once all useful instructions have finished, record address of where the program needs to be resumed in the PC (last completed useful instruction + 1)
5. Go to INT state, sent INTA, recive vector, disable interrupts
6. Save current mode in system stack; change mode to kernel
7. Store PC in `$k0`, retrieve handler via IVT, load PC, resume pipeline execution

---

Flushing the instructions (whether partially or fully) requires you to carry the PC value each instruction in the pipeline register

- Recall previously that we only needed to do this with the BEQ state

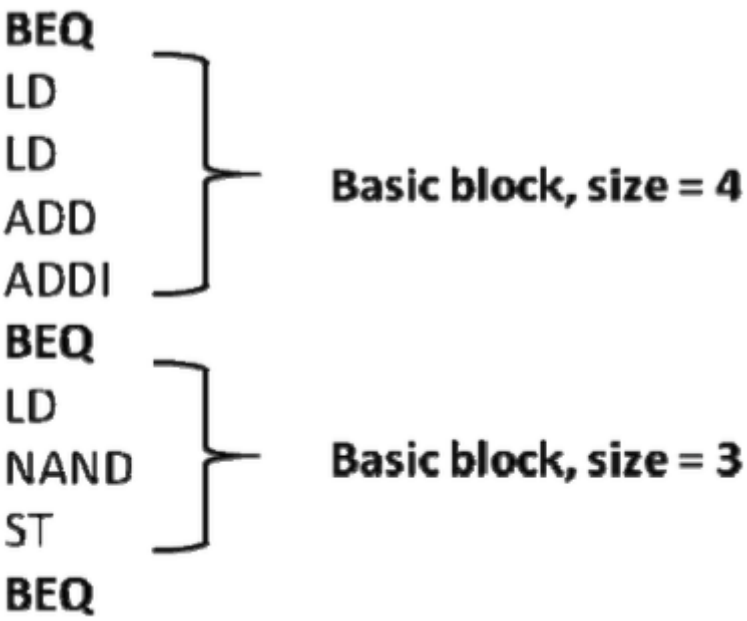
Instruction Level Parallelism (ILP)

Pipelined processor shrinks the execution time by recognizing that **adjacent instructions** of the program are **independent of each other** and therefore their

executions may be overlapped in time with one another

- Works very well for sequential programs

Basic block is a term used to define the string of instructions in a program separated by branches



Basic Block and ILP

Deeper Pipelines

Modern processors have really deep pipelines (Intel can often have 15+ stages!!)

- Frequency of branches in compiled code is pretty high
  - Means size of basic blocks is small

Modern processors have more than just a really deep pipeline – they have many pipelines running in parallel

Why do we need deeper pipelines?

- **Relative increase in time to access storage**: with faster clock cycle times, it may become necessary to allow multiple cycles for reading values out of caches. This increases the complexity to the unit that expects a value from the cache for performing its task, and increases the overall complexity of the processor. Breaking up the need for multiple cycles to carry out a specific operation into multiple stages is one way of dealing with the complexity
- **Microcode ROM access**: Accessing the microcode ROM could add a stage to the depth of the pipeline

- **Multiple functional Units**: Modern processors include both integer and floating-point arithmetic operations in their ISA. The EX unit of the simple 5-stage pipeline gets replaced by this collection of functional units
- **Dedicated floating-point pipelines**: Floating point operations take more time than their counterparts. This differential pipelining facilitates supporting multiple outstanding long latency operations (such as floating-point ADD) without stalling the pipeline due to structural hazards
- **Out of order execution and Re-Order Buffer**: modern processors distinguish between issue order and completion order. The fetch unit issues the instructions in order. However, the instructions may execute and complete out of order, due to the different depths of the different pipelines and other reasons (such as awaiting operands for executing the instruction). It turns out that this is fine so long as the instructions are retired from the processor in program order. In other words, an instruction is not retired from the processor even though it has completed execution until all the instructions preceding it in program order have also completed execution. To ensure this property, modern processors add additional logic that may be another stage in the pipeline. Specifically, the pipeline includes a **Re-Order Buffer** (ROB) whose job it is to retire the completed instructions in program order
- **Register Renaming**: modern processors have several more physical registers than the architecture-visible general-purpose registers. The processor may use an additional stage to detect resource conflicts and take actions to disambiguate the usage of registers using a technique referred to as register renaming
- **Hardware-based Speculation**: To overcome control hazards and fully exploit the multiple issue capability, many modern processors use hardware-based speculation, an idea that extends branch prediction