Chapter 5 Processor Performance and Rudiments of Pipelined Processor Design (Revision number 14)

In Chapter 3, we gave a basic design of a processor to implement the LC-2200 ISA. We hinted at how to achieve good performance when we talked about selecting the width of the clock cycle time and reducing the number of microstates in implementing each macro state (Fetch, Decode, and instruction-specific Execute states).

Processor design and implementation is a quantitative exercise. An architect is constantly evaluating his/her decision to include an architectural feature in terms of its impact on performance. Therefore, we will first explore the performance metrics that are of relevance in processor design. We will then look at ways to improve processor performance first in the context of the simple design we presented in Chapter 3, and then in the context of a new concept called pipelining.

First, let's introduce some metrics to understand processor performance.

5.1 Space and Time Metrics

Let's say you are building an airplane. You want to ferry some number of passengers every time, so you have to provide adequate space inside the aircraft for accommodating the passengers, their baggage, and food for them en route. In addition, you want to engineer the aircraft to take the passengers from point A to point B in a certain amount of time. The number of passengers to be accommodated in the plane has a bearing on the time it takes to ferry them, since the horsepower of the engine to fly the plane will depend on the total weight to be carried.

Let's see how this analogy applies to processor performance. Due to the hype about processor speed, we always think of "MHz" "GHz" and "THz" whenever we think of processor performance. These terms, of course, refer to the clock rate of the processor. We know from Chapter 3 that the clock cycle time (inverse of clock rate) is determined by estimating the worst-case delay in carrying out the datapath actions in a single clock cycle.

Let us understand why processor speed is not the only determinant of performance. Let us say you wrote a program "foo" and are running it on a processor. The two performance metrics that you would be interested in are, how much memory foo occupies (**space metric**) and how long does foo take to run (**time metric**). *Memory footprint* quantifies the space metric, while *execution time* quantifies the time metric. We define the former as the space occupied by a given program and the latter as the running time of the program. Let us relate these metrics to what we have seen about processor design so far.

The instruction-set architecture of the processor has a bearing on the memory footprint of the program. First, let us understand the relationship between the two metrics. There is a belief that the smaller the footprint the better the execution time. The conventional

wisdom in the 70's was that this premise is true and led to the design of *Complex Instruction Set Computer* (CISC) architectures. The criterion in CISC architecture was to make the datapath and control unit do more work for each instruction fetched from memory. This gave rise to a set of instructions, in which different instructions took different number of microstates to complete execution depending on their complexity. For example, an *add* instruction will take less number of microstates to execute, while a *multiply* instruction will take more. Similarly, an *add* instruction that uses register operands will take less microstates to execute while the same instruction that uses memory to operands will take more. With instructions that are more complex it is conceivable that the intended program logic can be contained in fewer instructions leading to a smaller memory footprint.

Common sense reasoning and advances in computer technology weakened the premise regarding the connection between memory footprint and the execution time.

- This is where the airplane analogy breaks down. The passengers in the analogy are akin to instructions in memory. All the passengers in the airplane need to be ferried to the destination, and therefore contribute to the weight of the airplane, which in turn determines the flying time to the destination. However, not all instructions that comprise a program are necessarily executed. For example, it is well known that in many production programs a significant percentage of the program deals with error conditions that may arise during program execution. You know from your own programming experience that one of the good software engineering practices is to check for return codes on systems calls. The error-handling portion of these checks in these programs may rarely be executed. Just to drive home that point, as a pathological example, consider a program that consists of a million instructions. Suppose there is a tight loop in the program with just 10 instructions where 99% of the execution time is spent; then the size of the program does not matter.
- Second, advances in processor implementation techniques (principally the idea of pipelining instruction execution, which we will discuss in this chapter) blurred the advantage of a complex instruction over a sequence of simple instructions that accomplish the same end result.
- Third, with the shift in programming in assembly language to programming in high-level languages, the utility of an instruction-set was measured by how useful it is to the compiler writer.
- Fourth, with the advent of cache memories, which we will cover in detail in Chapter 9, trips to the main memory from the processor is reduced, thus weakening one of the fundamental premise with CISC that execution time is minimized by fetching a complex instruction as opposed to a set of simple instructions.

These arguments gave rise to the *Reduced Instruction Set Computer (RISC)* Architecture in the late 70's and early 80's. While there was a lot of debate on CISC *vs.* RISC in the 80's, such arguments have become largely irrelevant today. The reality is both sides have shifted to include good features from the other camp. As we will see shortly, when we discuss pipelining, the ISA is not as important as ensuring that the processor maintains a

throughput of one instruction every clock cycle. For example, the dominant ISA today is Intel x86, a CISC architecture, but implemented with the RISC philosophy as far as implementation goes. Another influential ISA is MIPS, which is a RISC architecture but includes instructions traditionally found in CISC architectures.

We will have more detailed discussion on memory footprint itself in a later chapter that discusses memory hierarchy. At this point, let us simply observe that there is not a strong correlation between the memory footprint of a program and its execution time.

Let us try to understand what determines the execution time of the program. The number of instructions executed by the processor for this program is one component of the execution time. The second component is the number of microstates needed for executing each instruction. Since each microstate executes in one CPU clock cycle, the execution time for each instruction is measured by the clock cycles taken for each instruction (usually referred to as *CPI – clocks per instruction*). The third component is the *clock cycle time* of the processor.

So, if *n* is the total number of instructions executed by the program, then,

$$\text{Execution time} = (\sum \text{CPI}_j) * \text{clock cycle time, where } 1 \leq j \leq n \qquad (1)$$

Sometimes, it is convenient to think of an average CPI for the instructions that are executed by the program. So, if CPI_{Avg} is the average CPI for the set of instructions executed by the program, we can express the execution time as follows:

$$\text{Execution time} = n * \text{CPI}_{\text{Avg}} * \text{clock cycle time} \qquad (2)$$

Of course, it is difficult to quantify what an average CPI is, since this really depends on the frequency of execution of the instructions. We discuss this aspect in more detail in the next section. Execution time of a program is the key determinant of processor performance. Perhaps more accurately, since the clock cycle time changes frequently, *cycle count* (i.e., the total number of clock cycles for executing a program) is a more appropriate measure of processor performance. In any event, it should be clear that the processor performance is much more than simply the processor speed. Processor speed is no doubt important but the cycle count, which is the product of the number of instructions executed and the CPI for each instruction, is an equally if not more important metric in determining the execution time of a program.

Example 1:

A processor has three classes of instructions:

A, B, C; $\text{CPI}_A = 1$; $\text{CPI}_B = 2$; $\text{CPI}_C = 5$.

A compiler produces two different but functionally equivalent code sequences for a program:

Code sequence 1 executes:

A-class instructions = 5

B-class instructions = 3

C-class instructions = 1

Code sequence 2 executes:

A-class instructions = 3

B-class instructions = 2

C-class instructions = 2

Which is faster?

Answer:

Code sequence 1 results in executing 9 instructions, and takes a total of 16 cycles

Code sequence 2 results in executing 7 instructions but takes a total of 17 cycles

Therefore, code sequence 1 is faster.

5.2 Instruction Frequency

It is useful to know how often a particular instruction occurs in programs. *Instruction frequency* is the metric for capturing this information. *Static* instruction frequency refers to the number of times a particular instruction occurs in the compiled code. *Dynamic* instruction frequency refers to the number of times a particular instruction is executed when the program is actually run. Let us understand the importance of these metrics. Static instruction frequency has impact on the memory footprint. So, if we find that a particular instruction appears a lot in a program, then we may try to optimize the amount of space it occupies in memory by clever instruction encoding techniques in the instruction format. Dynamic instruction frequency has impact on the execution time of the program. So, if we find that the dynamic frequency of an instruction is high then we may try to make enhancements to the datapath and control to ensure that the CPI taken for its execution is minimized.

Static instruction frequency has become less important in general-purpose processors since reducing the memory footprint is not as important a consideration as increasing the processor performance. In fact, techniques to support static instruction frequency such as special encoding will have detrimental effect on performance since it destroys the uniformity of the instructions, which is crucial for a pipelined processor, as we will see later in this chapter (please see Section 5.10). However, static instruction frequency may still be an important factor in embedded processors, wherein it may be necessary to optimize on the available limited memory space.

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%

5.3 Benchmarks

Let us discuss how to compare the performance of machines. One often sees marketing hype such as "processor X is 1 GHz" or "processor Y is 500 MHz". How do we know which processor is better given that execution time is not entirely determined by processor speed. **Benchmarks** are a set of programs that are representative of the workload for a processor. For example, for a processor used in a gaming console, a video game may be the benchmark program. For a processor used in a scientific application, matrix operations may be benchmark programs. Quite often, **kernels** of real programs are used as benchmarks. For example, matrix multiply may occur in several scientific applications and may be biggest component of the execution time of such applications. In that case, it makes sense to benchmark the processor on the matrix multiply routine.

The performance of the processor on such kernels is a good indicator of the expected performance of the processor on an application as a whole.
It is often the case that a set of programs constitute the benchmark. There are several possibilities for deciding how best to use these benchmark programs to evaluate processor performance:

1. Let's say you have a set of programs and all of them have to be run one after the other to completion. In that case, a summary metric that will be useful is **total execution time**, which is the cumulative total of the execution times of the individual programs.
2. Let's say you have a set of programs and it is equally likely that you would want to run them at different times, but not all at the same time. In this case, **arithmetic mean (AM)** would be a useful metric, which is simply an average of all the individual program execution times. It should be noted, however that this metric may bias the summary value towards a time-consuming benchmark program (e.g., execution times of programs: P1 = 100 secs; P2 = 1 secs; AM = 50.5 secs).
3. Let's say you have a similar situation as above, but you have an idea of the frequency with which you may run the individual programs. In this case, **weighted arithmetic mean (WAM)** would be a useful metric, which is a weighted average of the execution times of all the individual programs. This metric takes into account the relative frequency of execution of the programs in the benchmark mix (e.g., for the same programs P1 = 100 secs; P2 = 1 secs; f_{P1} = 0.1; f_{P2} = 0.9; WAM = 0.1 * 100 + 0.9 * 1 = 10.9 secs).
4. Let's say you have a situation similar to (2) above but you have no idea of the relative frequency with which you may want to run one program versus another. In this case, using arithmetic mean may give a biased view of the processor performance. Another summary metric that is useful in this situation is **geometric mean (GM)**, which is the p^{th} root of the product of p values. This metric removes the bias present in arithmetic mean (e.g. for the same programs P1 = 100 secs; P2 = 1 secs; GM = sqrt (100*1) = 10 secs).
5. **Harmonic mean (HM)** is another useful composite metric. Mathematically, it is computed by taking the arithmetic mean of the reciprocals of the values being considered, and then taking the the reciprocal of the result. This also helps the bias towards high values present in the arithmetic mean. The HM for the example we are considering (execution times of programs: P1 = 100 secs; P2 = 1 secs),

   HM = 1/(arithmetic mean of the reciprocals)
   = 1/(((1/100) + (1/1))/2) = 1.9801.

   Harmonic means are especially considered useful when the dataset consists of ratios.

If the data set consists of all equal values then all three composite metrics (AM, GM, and HM) will yield the same result. In general, AM tends to bias the result towards high values in the data set, HM tends to bias the result towards low values in the data set, and GM tends to be in between. The moral of the story is one has to be very cautious about the use of a single composite metric to judge an architecture.

Over the years, several benchmark programs have been created and used in architectural evaluations. The most widely accepted one is the SPEC benchmarks developed by an independent non-profit body called *Standard Performance Evaluation Corporation* (SPEC), whose goal is "to establish, maintain and endorse a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers[1]." SPEC benchmark consists of a set of generic applications including scientific, transaction processing, and web server that serves as the workload for general-purpose processors[2].

What makes benchmarking hard is the fact that processor performance is not just determined by the clock cycle rating of the processor. For example, the organization of the memory system and the processor-memory bus bandwidth are key determinants beyond the clock cycle time. Further, the behavior of each benchmark application poses different demands on the overall system. Therefore, when we compare two processors with comparable or even same CPU clock cycle rating, we may find that one does better on some benchmark programs while the second does better on some others. This is the reason a composite index is useful sometimes when we want to compare two processors without knowing the exact kind of workload we may want to run on them. One has to be very cautious on the over-use of metrics as has been noted in a very famous saying, "Lies, damned lies, and statistics[3]."

Example 3:

The SPECint2006 integer benchmark consists of 12 programs for quantifying the performance of processors on integer programs (as opposed to floating point arithmetic). The following table[4] shows the performance of Intel Core 2 Duo E6850 (3 GHz) processor on SPECint2006 benchmark.

Program name	Description	Time in seconds
400.perlbench	Applications in Perl	510
401.bzip2	Data compression	602
403.gcc	C Compiler	382
429.mcf	Optimization	328
445.gobmk	Game based on AI	548
456.hmmer	Gene sequencing	593
458.sjeng	Chess based on AI	679
462.libquantum	Quantum computing	422
464.h264ref	Video compression	708
471.omnetpp	Discrete event simulation	362
473.astar	Path-finding algorithm	466
483.xalancbmk	XML processing	302

[1] Source: **http://www.spec.org/**
[2] Please see **http://www.spec.org/cpu2006/publications/CPU2006benchmarks.pdf** for a description of SPEC2006 benchmark programs for measuring the integer and floating point performance of processors.
[3] Please see: **http://www.york.ac.uk/depts/maths/histstat/lies.htm**
[4] Source: **http://www.spec.org/cpu2006/results/res2007q4/cpu2006-20071112-02562.pdf**

(a) Compute the arithmetic mean and the geometric mean.

Answer:

Arithmetic mean = (510+602+…+302)/12 = **491.8 secs**
Geometric mean = $(510 * 602 * \ldots * 302)^{1/12}$ = **474.2 secs**

Note how arithmetic mean biases the result towards the larger execution times in the program mix.

(b) One intended use of the system, has the following frequencies for the 12 programs:

- 10% video compression
- 10% XML processing
- 30% path finding algorithm
- 50% all the other programs

Compute the weighted arithmetic mean for this workload

Answer:

The workload uses 9 programs equally 50% of the time.
The average execution time of these nine programs
= (510 + 602+382+328+548+593+679+422+362)/9
= 491.8 secs
Weighted arithmetic mean = (0.1 * 708 + 0.1 * 302 + 0.3 * 466 + 0.5 * 491.8)
= **486.7 secs**

One of the usefulness of metrics is that it gives a basis for comparison of machines with different architectures, implementation, and hardware specifications. However, raw numbers as reported in Example 3, makes it difficult to compare different machines. For this reason, SPEC benchmark results are expressed as ratios with respect to some reference machine. For example, if the time for a benchmark on the target machine is **x** secs, and the time on the reference machine for the same benchmark is **y** secs, then the SPECratio for this benchmark on the target machine is defined as

SPECratio
= execution time on reference machine / execution time on target machine
= y/x

SPEC organization chose Sun Microsystems's Ultra5_10 workstation with a 300-MHz SPARC processor and 256-MB of memory as a reference machine for SPEC CPU 2000 performance test results. The same is used for SPEC CPU 2006 as well.

The SPECratios for different benchmarks may be combined using one of the statistical measures (arithmetic, weighted arithmetic, geometric, or harmonic) to get a single composite metric. Since we are dealing with ratios when using SPEC benchmarks as the standard way of reporting performance, it is customary to use harmonic mean.

The nice thing about SPECratio is that it serves as a basis for comparison of machines. For example, if the mean SPECratios of two machines A and B are R_A and R_B, respectively, then we can directly make relative conclusions regarding the performance capabilities of the two machines.

5.4 Increasing the Processor Performance

To explore avenues for increasing the processor performance, a good starting point is the equation for execution time. Let us look at each term individually and understand the opportunity for improving the performance that each offers.

- **Increasing the clock speed:** The processor clock speed is determined by the worst-case delay in the datapath. We could rearrange the datapath elements such that the worst-case delay is reduced (for example bringing them closer together physically in the layout of the datapath). Further, we could reduce the number of datapath actions taken in a single clock cycle. However, such attempts at reducing the clock cycle time have an impact on the number of CPIs needed to execute the different instructions. To get any more reduction in clock cycle time beyond these ideas, the feature size of the individual datapath elements should be shrunk. This avenue of optimization requires coming up with new chip fabrication processes and device technologies that help in reducing the feature sizes.
- **Datapath organization leading to lower CPI:** In Chapter 3, the implementation uses a single bus. Such an organization limits the amount of hardware concurrency among the datapath elements. We alluded to designs using multiple buses to increase the hardware concurrency. Such designs help in reducing the CPI for each instruction. Once again any such attempt may have a negative impact on the clock cycle time, and therefore requires careful analysis. Designing the microarchitecture of a processor and optimizing the implementation to maximize the performance is a fertile area of research both in academia and in industries.

The above two bullets focus on reducing the *latency* of individual instructions so that cumulatively we end up with a lower execution time.

Another opportunity for reducing the execution time is reducing the number of instructions.

- **Reduction in the number of executed instructions:** One possibility for reducing the number of instructions executed in the program is to replace simple instructions by more complex instructions. This would reduce the number of instructions executed by the program overall. We already saw a counter example to this idea. Once again any attempt to introduce new complex instructions has to be carefully balanced against the CPI, the clock cycle time, and the dynamic instruction frequencies.

It should be clear from the above discussion that all three components of the execution time are inter-related and have to be optimized simultaneously and not in isolation.

Example 4:

An architecture has three types of instructions that have the following CPI:

Type	CPI
A	2
B	5
C	1

An architect determines that he can reduce the CPI for B to three, with no change to the CPIs of the other two instruction types, but with an increase in the clock cycle time of the processor. What is the maximum permissible increase in clock cycle time that will make this architectural change still worthwhile? Assume that all the workloads that execute on this processor use 30% of A, 10% of B, and 60% of C types of instructions.

Answer:

Let Co and Cn be the clock cycle times of the old (Mo) and new (Mn) machines, respectively. Let N be the total number of instructions executed in a program.

Execution time of Old Machine

$$ET_{Mo} = N * (F_A * CPI_{Ao} + F_B * CPI_{Bo} + F_C * CPI_{Co}) * Co$$

Where F_A, CPI_{Ao}, F_B, CPI_{Bo}, F_C, CPI_{Co} are the dynamic frequencies and CPIs of each type of instruction, respectively.

Execution time of the Old Machine

$$ET_{Mo} = N * (0.3 * 2 + 0.1 * 5 + 0.6 * 1) * Co$$
$$= N * 1.7\ Co$$

Execution time of New Machine

$$ET_{Mn} = N * (0.3 * 2 + 0.1 * 3 + 0.6 * 1) * Cn$$
$$= N * 1.5\ Cn$$

For design to be viable

$$ET_{Mn} < ET_{Mo}$$
$$N * 1.5\ Cn < N * 1.7\ Co$$
$$Cn < 1.7/1.5 * Co$$
$$Cn < 1.13\ Co$$

Max Permissible increase in clock cycle time = 13%

5.5 Speedup

Comparing the execution times of processors for the same program or for a benchmark suite is the most obvious method for understanding the performance of one processor relative to another. Similarly, we can compare the improvement in execution time before and after a proposed modification to quantify the performance improvement that can be attributed to that modification.

We define,

$$Speedup_{AoverB} = \frac{\text{Execution Time on Processor B}}{\text{Execution Time on Processor A}} \qquad (3)$$

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

Similarly,

$$Speedup_{improved} = \frac{\text{Execution Time Before Improvement}}{\text{Execution Time After Improvement}} \qquad (4)$$

Speedup due to a modification is the ratio of the execution time before improvement to that after the improvement.

Example 5:
Given below are the CPIs of instructions in an architecture:

Instruction	CPI
Add	2
Shift	3
Others	2 (average of all instructions including Add and Shift)

Profiling the performance of a programs, an architect realizes that the sequence ADD followed by SHIFT appears in 20% of the dynamic frequency of the program. He designs a new instruction, which is an ADD/SHIFT combo that has a CPI of 4.

What is the speedup of the program with all {ADD, SHIFT} replaced by the new combo instruction?

Answer:
Let N be the number of instructions in the original program.

Execution time of the original program
= N * frequency of ADD/SHIFT * (2+3) + N * frequency of others * 2
= N * 0.2 * 5 + N * 0.8 * 2 = 2.6 N
With the combo instruction replacing {ADD, SHIFT}, the number of instructions in the new program shrinks to 0.9 N. In the new program, the frequency of the combo instruction is 1/9 and the other instructions are 8/9.

Execution time of the new program
= (0.9 N) * frequency of combo * 4 + (0.9 N) * frequency of others * 2
= (0.9 N) * (1/9) * 4 + (0.9 N) * (8/9) * 2
= 2 N

Speedup of program = old execution time/new execution time
= (2.6 N) / (2 N)
= 1.3

Another useful metric is the performance improvement due to a modification:

$$\text{Improvement in execution time} = \frac{\text{old execution time} - \text{new execution time}}{\text{old execution time}} \quad (5)$$

Example 6:
A given program takes 1000 instructions to execute with an average CPI of 3 and a clock cycle time of 2 ns. An architect is considering two options. (1) She can reduce the average CPI of instructions by 25% while increasing the clock cycle time by 10%; or (2) She can reduce the clock cycle time by 20% while increasing the average CPI of instructions by 15%.

(a) You are the manager deciding which option to pursue. Give the reasoning behind your decision.

Answer:
Let E0, E1, and E2, denote the execution times with base machine, first option and second option respectively.

E0 = 1000 * 3 * 2 ns
E1 = 1000 * (3 * 0.75) * 2 (1.1) ns = 0.825 E0
E2 = 1000 * (3 * 1.15) * 2 (0.8) ns = 0.920 E0

Option 1 is better since it results in lesser execution time than 2.

(b) What is the improvement in execution time of the option you chose compared to the original design?

Answer:
Improvement of option 1 relative to base = (E0 – E1)/E0
= (E0 – 0.825 E0)/E0
= 0.175
17.5 % improvement

Another way of understanding the effect of an improvement is to take into perspective the extent of impact that change has on the execution time. For example, the change may have an impact only on a portion of the execution time. A law associated with Gene Amdahl, a pioneer in parallel computing can be adapted for capturing this notion:

Amdahl's law:

$$\text{Time}_{after} = \text{Time}_{unaffected} + \text{Time}_{affected}/x \quad (6)$$

In the above equation, $Time_{after}$ the total execution time as a result of a change is the sum of the execution time that is unaffected by the change ($Time_{unaffected}$), and the ratio of the affected time ($Time_{affected}$) to the extent of the improvement (denoted by x). Basically, 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.

Name	Notation	Units	Comment
Memory footprint	-	Bytes	Total space occupied by the program in memory
Execution time	$(\sum CPI_j)$ * clock cycle time, where $1 \leq j \leq n$	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
Harmonic mean	$1/(\ (\ (1/E_1)+(1/E_2)+\ldots+(1/E_p)\ )/p\ )$	Seconds	Arithmetic mean of the reciprocals of the execution times of the constituent *p* benchmark programs
Static instruction frequency		%	Occurrence of instruction *i* in compiled code
Dynamic instruction frequency		%	Occurrence of instruction *i* in executed code
Speedup (M_A over M_B)	E_B/E_A	Number	Speedup of Machine A over B
Speedup (improvement)	E_{Before}/E_{After}	Number	Speedup After improvement
Improvement in Exec time	$(E_{old}-E_{new})/E_{old}$	Number	New Vs. old
Amdahl's law	$Time_{after} = Time_{unaffected} + Time_{affected}/x$	Seconds	*x* is amount of improvement

Table 5.1: Summary of Performance Metrics

Table 5.1 summarizes all the processor-related performance metrics that we have discussed so far.

Example 7:
A processor spends 20% of its time on add instructions. An engineer proposes to improve the *add* instruction by 4 times. What is the speedup achieved because of the modification?

Answer:
The improvement only applies for the Add instruction, so 80% of the execution time is unaffected by the improvement.
Original normalized execution time = 1

New execution time = (time spent in add instruction/4) + remaining execution time
= 0.2/4 + 0.8
= 0.85
Speedup = Execution time before improvement/Execution time after improvement
= 1/0.85 = 1.18

5.6 Increasing the Throughput of the Processor

Thus far, we focused on techniques to reduce the latency of individual instructions to improve processor performance. A radically different approach to improving processor performance is not to focus on the *latency* for individual instructions (i.e., the CPI metric) but on *throughput*. That is, the number of instructions executed by the processor per unit time. Latency answers the question, how may clock cycles does the processor take to execute an individual instruction (i.e., CPI). On the other hand, throughput answers the question, how many instructions does the processor execute in each clock cycle (i.e., *IPC* or *instructions per clock cycle*). The concept called *pipelining* is the focus of the rest of this chapter.

5.7 Introduction to Pipelining

Welcome to Bill's Sandwich shop! He has a huge selection of breads, condiments, cheese, meats, and veggies to choose from in his shop, all neatly organized into individual stations. When Bill was starting out in business, he was a one-man team. He took the orders, went through the various stations, and made the sandwich to the specs. Now his business has grown. He has 5 employees one for each of the five stations involved with the sandwich assembly: taking the order, bread and condiments selection, cheese selection, meat selection, and veggies selection. The following figure shows the sandwich assembly process.

Station 1 (place order)	station II (select bread)	station III (cheese)	station IV (meat)	station V (veggies)
New (5^{th} order)	4^{th} order	3^{rd} order	2^{nd} order	1^{st} order

Each station is working on a different order; while the last station (station V) is working on the very first order, the first station is taking a new order for a sandwich. Each station after doing "its thing" for the sandwich passes the partially assembled sandwich to the next station along with the order. Bill is a clever manager. He decided against dedicating an employee to any *one* customer. That would require each employee to have his/her own stash of *all* the ingredients needed to make a sandwich and increase the inventory of raw materials unnecessarily since each customer may want only a subset of the ingredients. Instead, he carefully chose the work to be done in each station to be roughly the same, so that no employee is going to be twiddling his thumbs. Of course, if a particular sandwich order does not need a specific ingredient (say cheese) then the corresponding station simply passes on the partially assembled sandwich on to the next station. Nevertheless, most of the time (especially during peak time) all his employees are kept busy rolling out sandwiches in rapid succession.

Bill has quintupled the rate at which he can serve his customers.

5.8 Towards an instruction processing assembly line

You can see where we are going with this…in the simple implementation of LC-2200, the FSM executes one instruction at a time taking it all the way through the FETCH, DECODE, EXECUTE macro states before turning its attention to the next instruction. The problem with that approach is that the datapath resources are under-utilized. This is because, for each macro state, not all the resources are needed. Let us not worry about the specific datapath we used in implementing the LC-2200 ISA in Chapter 3. Regardless of the details of the datapath, we know that any implementation of LC-2200 ISA would need the following datapath resources: Memory, PC, ALU, Register file, IR, and sign extender. Figure 5.1 shows the hardware resources of the datapath that are in use for each of the macro states for a couple of instructions:

Macro State	Datapath Resources in Use				
FETCH	IR	ALU	PC	MEM	
DECODE	IR				
EXECUTE (ADD)	IR	ALU	Reg-file		
EXECUTE (LW)	IR	ALU	Reg-file	MEM	Sign extender

Figure 5.1: Datapath resources in use for different macro states

We can immediately make the following two observations:

1. The IR is in use in every macro state. This is not surprising since IR contains the instruction, and parts of the IR are used in different macro states of its execution. IR is equivalent to the "order" being passed from station to station in the sandwich assembly line.

2. At the macro level, it can be seen that a different amount of work is being done in the three-macro states. Each state of the FSM is equivalent to a station in the sandwich assembly line.

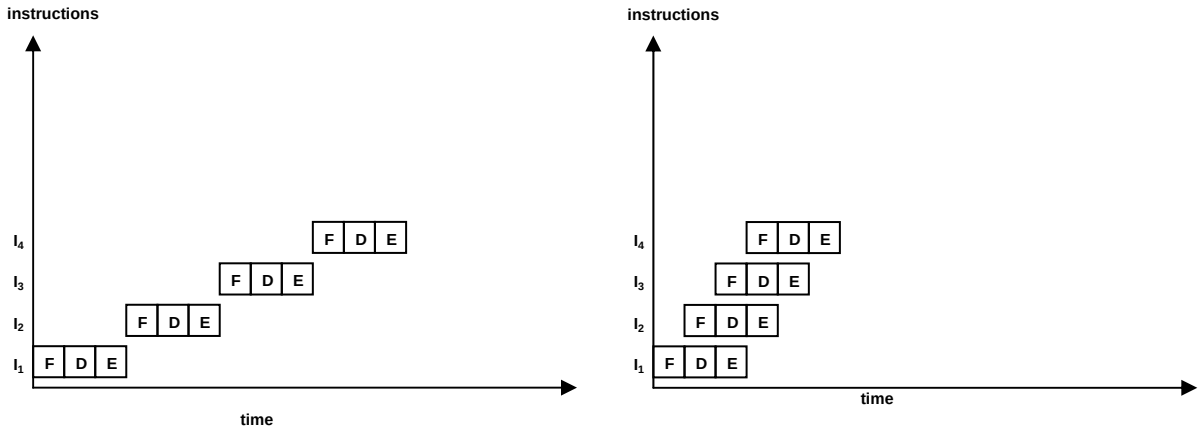
What we want to do is to use all the hardware resources in the datapath all the time if possible. In our sandwich assembly line, we kept all the stations busy by assembling multiple sandwiches simultaneously. We will try to apply the sandwich assembly line idea to instruction execution in the processor.

A program is a sequence of instructions as shown in Figure 5.2:

```
I1:   Ld R1, MEM [1000];  R1 <- Memory at location 1000
I2:   Ld R2, MEM [2000];  R2 <- Memory at location 2000
I3:   Add R3, R5, R4;     R3 <- R4 + R5
I4:   Nand R6, R7, R8;    R6 <- R7 NAND R8
I5:   St R9, MEM [3000];  R9 -> Memory at location 3000
I6:   .....
I7:   .....
I8
I9
I10
I11
I12
I13
```

Figure 5.2: A Program is a sequence of instructions

With the simple implementation of the FSM, the time line for instruction execution in the processor will look as shown in Figure 5.3-(a):



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

Figure 5.3: Execution Timeline

A new instruction processing starts only after the previous one is fully completed. Using our sandwich assembly line idea, to maximize the utilization of the datapath resources, we should have multiple instructions in execution in our instruction *pipeline* as shown in Figure 5.3-(b). A question that arises immediately is if this is even possible. In a sandwich assembly line, assembling your sandwich is completely independent of what your predecessor or your successor in the line wants for his/her sandwich. Are the successive instructions in a program similarly independent of one another? "No" is the immediate answer that comes to mind since the program is sequential. However, look at the sequence of instructions shown in Figure 5.2. Even though it is a sequential program, one can see that instructions I_1 through I_5 happen to be independent of one another. That is, the execution of one instruction does not depend on the results of the execution of the previous instruction in this sequence. We will be quick to point out this happy state of affairs is not the norm, and we will deal with such dependencies in a processor pipeline in a later section (please see Section 5.13). For the moment, it is convenient to think that the instructions in sequence are independent of one another to illustrate the possibility of adopting the sandwich assembly line idea for designing the processor pipeline.

Notice that in the pipeline execution timeline shown in Figure 5.3-(b), we are starting the processing of a new instruction as soon as the previous instruction moves to the next macro state. If this were possible, we can triple the throughput of instructions processed. The key observation is that the datapath resources are akin to individual ingredients in the sandwich assembly line, and the macro states are akin to the employees at specific stations of the sandwich assembly line.

5.9 Problems with a simple-minded instruction pipeline

Notice several problems with the above simple-minded application of the assembly line idea to the instruction pipeline.

1. The different stages often need the same datapath resources (e.g. ALU, IR).
2. The amount of work done in the different stages is not the same. For example, compare the work done in DECODE and EXECUTE-LW states in Figure 5.1. The former is simply a combinational function to determine the type of instruction and the resources needed for it. On the other hand, the latter involves address arithmetic, memory access, and writing to a register file. In general, the work done in the EXECUTE state will far outweigh the work done in the other stages.

The first point implies that we have resource contention among the stages. This is often referred to as a *structural hazard*, and is a result of the limitations in the datapath such as having a single IR, a single ALU, and a single bus to connect the datapath elements. In the sandwich assembly line the individual order (equivalent of the contents of IR) is on a piece of paper that is passed from station to station; and the partially assembled sandwich (equivalent of an instruction in partial execution) is passed directly from one station to the next (i.e., there is no central "bus" as in our simple datapath). We can use similar ideas to fix this problem with our instruction pipeline. For example, if we add an *extra* ALU, an *extra* IR, and an *extra* MEM to the datapath dedicated to the FETCH stage then that stage will become independent of the other stages. An extra ALU is understandable

since both the FETCH and EXECUTE stages need to use it, but we need to understand what it means to have an extra IR. Just as the order is passed from station to station in the sandwich assembly line, we are passing the contents of the IR from the FETCH to the DECODE stage and so on to keep the stages independent of one another. Similarly, we split the total memory into an I-MEM for instruction memory and D-MEM for data memory to make the FETCH stage independent of the EXECUTION stage. As should be evident, instructions of the program come from the I-MEM and the data structures that the program manipulates come from the D-MEM. This is a reasonable split since (a) it is a good programming practice to keep the two separate, and (b) most modern processors (e.g., memory segmentation in Intel x86 architecture) separate the memory space into distinct regions to ensure inadvertent modifications to instruction memory by a program.

The second point implies that the time needed for each stage of the instruction pipeline is different. Recall, that Bill carefully engineered his sandwich assembly line to ensure that each employee did the same amount of work in each station. The reason is the slowest member of the pipeline limits the throughput of the assembly line. Therefore, for our instruction assembly line to be as efficient, we should break up the instruction processing so that each stage does roughly the same amount of work.

5.10 Fixing the problems with the instruction pipeline

Let us look at the DECODE stage. This stage does the least amount of work in the current setup. To spread the work more evenly, we have to assign more work to this stage. We have a dilemma here. We cannot really do anything until we know what the instruction is which is the focus of this stage. However, we can do something opportunistically so long as the semantics of the actual instruction is not affected.

We expect most instructions to use register contents. Therefore, we can go ahead and read the register contents from the register file without actually knowing what the instruction is. In the worst case, we may end up not using the values read from the registers. However, to do this we need to know which registers to read. This is where the instruction format chosen during instruction-set design becomes crucial. If you go back (Chapter 2) and look at the instructions that use registers in LC-2200 (ADD, NAND, BEQ, LW, SW, and JAL) you will find the source register specifiers for arithmetic/logic operations or for address calculations occupy always the *same* bit positions in the instruction format. We can exploit this fact and opportunistically read the registers as we are decoding what the actual instruction is. In a similar vein, we can break up the EXECUTE stage which potentially does a lot of work into smaller stages.

Using such reasoning, let us subdivide the processing of an instruction into the following five functional components or stages:

- **IF:** This stage 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:** This stage decodes the instruction and reads the register files to pull out two source operands (more than what may actually be needed depending on the

instruction). To enable this functionality, the register file has to be *dual-ported*, meaning that two register addresses can be given simultaneously and two register contents can be read out in the same clock cycle. We will call such a register file dual-ported register file (DPRF). Since this stage contains the logic for decoding the instruction as well as reading the register file, we have given it a hybrid name.

- **EX:** This is the stage that does all the arithmetic and/or logic operations that are needed for processing the instruction. We will see shortly (Section 5.11) that one ALU may not suffice in the EX stage to cater to the needs of all the instructions.
- **MEM:** This stage either reads from or writes to the D-MEM for LW and SW instructions, respectively. Instructions that do not have a memory operand will not need the operations performed in this stage.
- **WB:** This stage writes the appropriate destination register (Rx) if the instruction requires it. Instructions in LC-2200 that require writing to a destination register include arithmetic and logic operations as well as load.

Pictorially the passage of an instruction through the pipeline is shown in Figure 5.4.

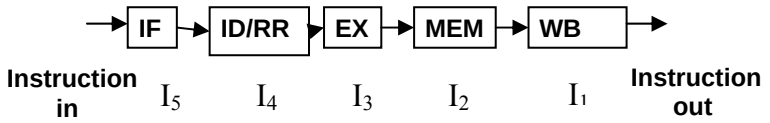


Figure 5.4: Passage of instructions through the pipeline

Similar to the sandwich assembly line, we expect every instruction to pass through each of these stages in the course of the processing. At any point of time, five instructions are in execution in the pipeline: when Instruction I_1 is in the WB stage, instruction I_5 is in the IF stage. This is a synchronous pipeline in the sense that on each clock pulse the partial results of the instruction execution is passed on to the next stage. The inherent assumption is that the clock pulse is wide enough to allow the slowest member of the pipeline to complete its function within the clock cycle time.

Every stage works on the partial results generated in the *previous clock cycle* by the preceding stage. Not every instruction needs every stage. For example, an ADD instruction does not need the MEM stage. However, this simply means that the MEM stage does nothing for one clock cycle when it gets the partial results of an ADD instruction (one can see the analogy to the sandwich assembly line). One might think that this is an inefficient design if you look at the passage of specific instructions. For example, this design adds one more cycle to the processing of an ADD instruction.

However, the goal of the pipelined design is to increase the throughput of instruction processing and *not* reduce the latency for each instruction. To return to our sandwich assembly line example, the analogous criterion in that example was to keep the customer line moving. The number of customers served per unit time in that example is analogous to the number of instructions processed per unit time in the pipelined processor.

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. Such buffering is essential to give the independence for each stage. Figure 5.5 shows the instruction pipeline with the buffers added between the stages. *Pipeline register* is the term commonly used to refer to the buffer between the stages. We will use pipeline register and buffer interchangeably in this chapter. In our sandwich assembly line example, the partially assembled sandwich serves as the buffer and gives independence and autonomy to each stage.

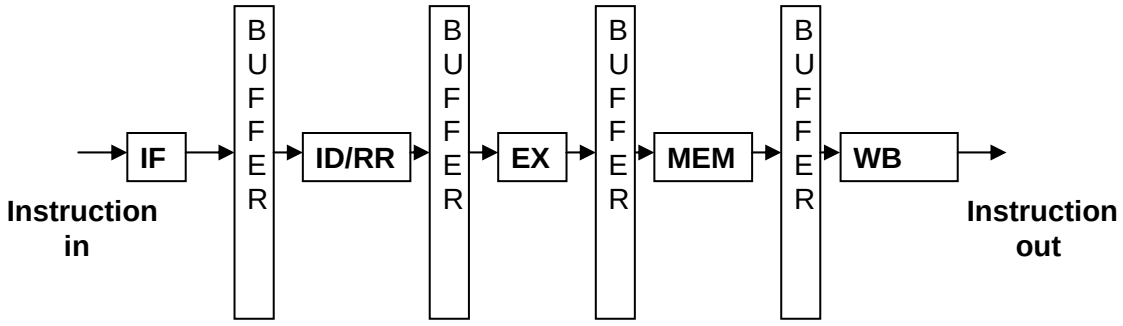


Figure 5.5: Instruction pipeline with buffers between stages

5.11 Datapath elements for the instruction pipeline

The next step is to decide the datapath elements needed for each stage of the instruction pipeline and the contents of the buffers between the stages to provide the isolation between them.

- In the IF stage we need a PC, an ALU, and I-MEM. The output of this stage is the instruction fetched from the memory; therefore, the pipeline register between IF and ID/RR stages should contain the instruction.

- In the ID/RR stage, we need the DPRF. The 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). These constitute the contents of the pipeline register between ID/RR and EX stages.

- The 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. This depends on the instruction-set repertoire.

  In our case (LC-2200), the only instruction that required more than one arithmetic operation is the BEQ instruction. BEQ requires one ALU to do the comparison (A == B) and another to do the effective address computation (PC + signed-offset). Therefore, we need two ALU's for the EX stage.

BEQ instruction also brings out another requirement. The address arithmetic requires the value of the PC that corresponds to the BEQ instruction (Incidentally, the value of PC is required for another instruction as well, namely, JALR). Therefore, the value of PC should also be communicated from stage to stage (in addition to the other things) through the pipeline.
The output of the EX stage is the result of the arithmetic operations carried out and therefore instruction specific. The contents of the pipeline register between EX and MEM stage depends on such specifics. For example, if the instruction is an ADD instruction, then the pipeline register will contain the result of the ADD, the opcode, and the destination register specifier (Rx). We can see that the PC value is not required after the EX stage for any of the instructions. Working out the details of the pipeline register contents for the other instructions is left as an exercise to the reader.

- The MEM stage requires the D-MEM. If the instruction processed by this stage in a clock cycle is not LW or SW, then the contents of the input buffer is simply copied to the output buffer at the end of the clock cycle. 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. With a little of reflection, it is easy to see that no action is needed in the WB stage if the opcode is SW.

- The WB stage requires the register file (DPRF). This stage is relevant only for the instructions that write to a destination register (such as LW, ADD, and NAND). This poses an interesting dilemma. In every clock cycle we know every stage is working on a different instruction. Therefore, with reference to Figure 5.4, at the same time that WB is working on I_1, ID/RR is working on I_4. Both these stages need to access the DPRF simultaneously on behalf of different instructions (for example: I_1 may be ADD R1, R3, R4 and I_4 may be NAND R5, R6, R7). Fortunately, WB is writing to a register while ID/RR is reading registers. Therefore, there is no conflict in terms of the logic operation being performed by these two stages, and both can proceed simultaneously as long as the same register is not being read and written to in a given cycle. It is a semantic conflict when the same register is being read and written to in the same cycle (for example consider I_1 as ADD R1, R3, R4 and I_4 as ADD R4, R1, R6). We will address such semantic conflicts shortly (Section 5.13.2).

Pictorially, we show the new organization of the resources for the various stages in Figure 5.6. Note that DPRF in ID/RR and WB stages refer to the same logic element in the datapath. In Figure 5.6, both the stages include DPRF just for clarity of the resources needed in those stages.

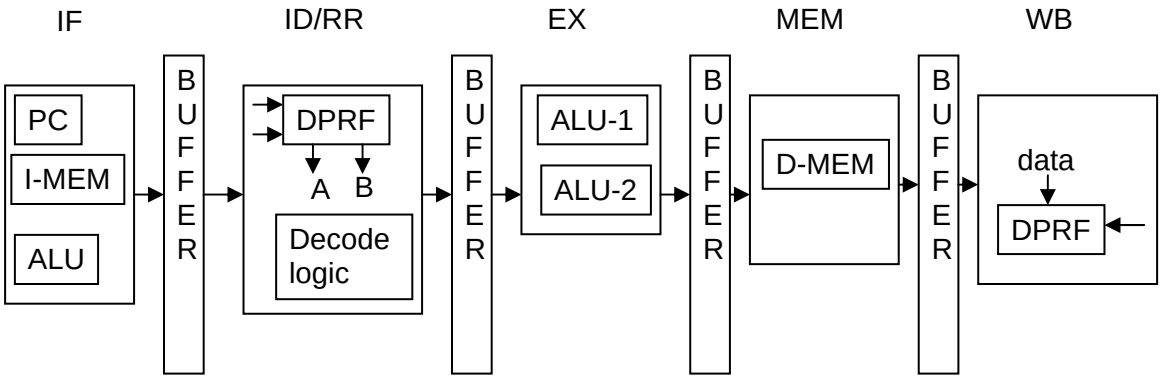


Figure 5.6: Organization of hardware resources for various stages

Some things to note with our pipelined processor design. In the steady state, there are 5 instructions in different stages of processing in the pipeline. We know in the simple design of LC-2200, the FSM transitioned through microstates such as **ifetch1, ifetch2,** etc**.** In a given clock cycle, the processor was in *exactly* one state. In a pipelined implementation, the processor is simultaneously in all the states represented by the stages of the pipelined design. Every instruction takes 5 clock cycles to execute. Each instruction enters the pipeline at the IF stage and is *retired* (i.e., completed successfully) from the WB stage. In the ideal situation, the pipelined processor retires one new instruction in every clock cycle. Thus, the effective CPI of the pipelined processor is 1. Inspecting Figure 5.6, one might guess that MEM stage could be the slowest. However, the answer to this question is more involved. We discuss considerations in modern processors with respect to reducing the clock cycle time in Section 5.15.

The performance of pipelined processor is crucially dependent on the memory system. Memory access time is the most dominant latency in a pipelined processor. To hide this latency, processors employ caches. Recall the toolbox and tool tray analogy from Chapter 2. We mentioned that registers serve the role of the tool tray in that we *explicitly* bring the memory values the processor needs into the registers using the load instruction. Similarly, caches serve as *implicit* tool trays. In other words, whenever the processor brings something from the memory (instruction or data), it is implicitly placed into a high-speed storage inside the processor called caches. By creating an implicit copy of the memory location in the cache, the processor can subsequently re-use values in these memory locations without making trips to the memory. We will discuss caches in much more detail in a later chapter (please see Chapter 9) together with their effect on pipelined processor implementation. For the purpose of this discussion on pipelined processor implementation, we will simply say that caches allow hiding the memory latency and make pipelined processor implementation viable. In spite of the caches and high-speed registers, the combinational logic delays (ALU, multiplexers, decoders, etc.) are significantly smaller than access to caches and general-purpose registers. In the interest of making sure that all stages of the pipeline have roughly the same latency, modern processor implementation comprises much more than five stages. For example, access to storage elements (cache memory, register file) may take multiple cycles in the processor pipeline. We will discuss such issues in a later section (please see Section 5.15). Since,

this is the first introduction to pipelined implementation of a processor, we will keep the discussion simple.

5.12 Pipeline-conscious architecture and implementation

The key points to note in designing a pipeline-conscious architecture are:

- **Need for a symmetric instruction format:** This has to do with ensuring that the locations of certain fields in the instruction (e.g. register specifiers, size and position of offset, etc.) remain unchanged independent of the instruction. As we saw, this was a key property we exploited in the ID/RR stage for LC-2200.
- **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.

Figure 5.6a shows the full datapath for a pipelined LC-2200 implementation.

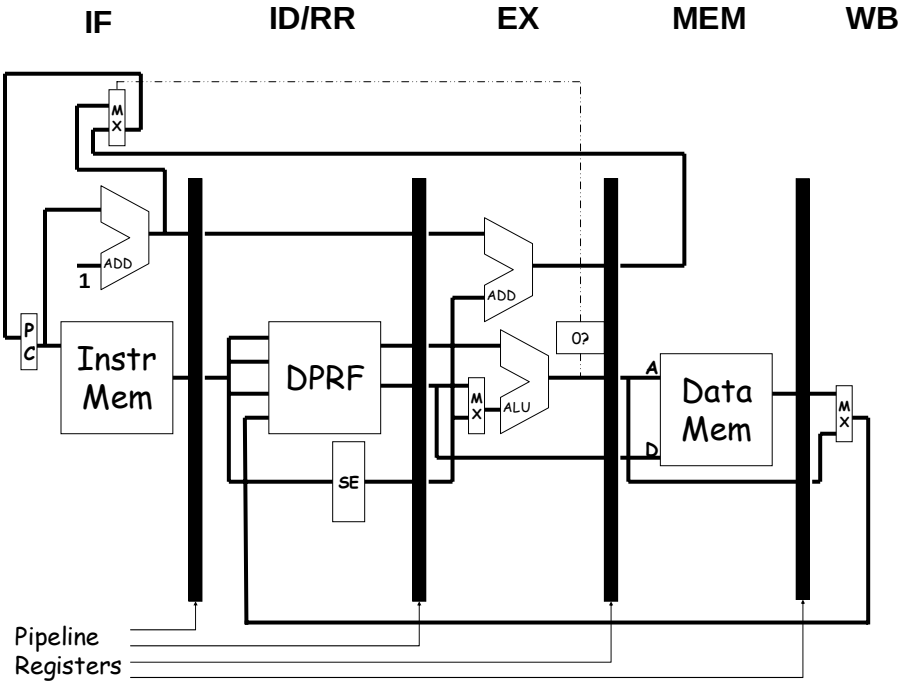


Figure 5.6a: Datapath for a pipelined LC 2200

5.12.1 Anatomy of an instruction passage through the pipeline

In this subsection, we will trace the passage of an instruction through the 5-stage pipeline. We will denote the buffers between the stages with unique names as shown in Figure 5.6b.

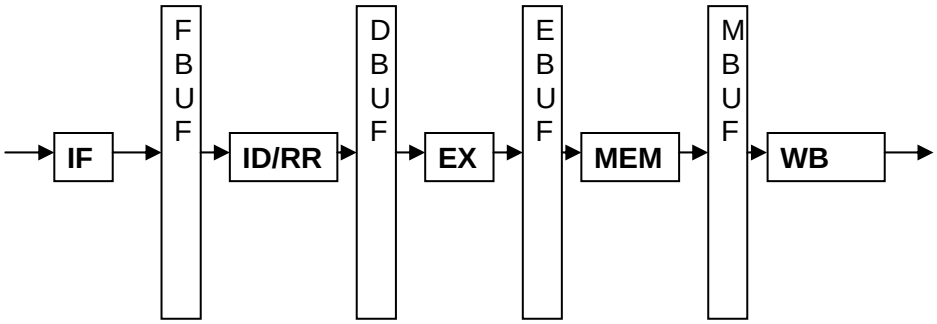


Figure 5.6b: Pipeline registers with unique names

Table 5.2 summarizes the function of each of the pipeline buffers between the stages.

Name	Output of Stage	Contents
FBUF	IF	Primarily contains instruction read from memory
DBUF	ID/RR	Decoded IR and values read from register file
EBUF	EX	Primarily contains result of ALU operation plus other parts of the instruction depending on the instruction specifics
MBUF	MEM	Same as EBUF if instruction is not LW or SW; If instruction is LW, then buffer contains the contents of memory location read

Table 5.2: Pipeline Buffers

Let us consider the passage of the Add instruction that has the following syntax and format:

Add Rx, Ry, Rz; Rx <- Ry + Rz

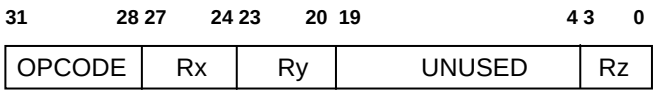


Figure 5.6c: Syntax and format of LC 2200 Add instruction

Each stage performs the actions summarized below to contribute towards the execution of the Add instruction:

IF stage (cycle 1):

```
I-MEM[PC] -> FBUF     // The instruction at memory address given by PC is
                         fetched and placed in FBUF (which is essentially
                         the IR); the contents of FBUF after this action will
                         be as shown in Figure 5.6c
PC + 1 -> PC          // Increment PC
```

ID/RR stage (cycle 2):

```
DPRF[ FBUF[Ry] ] -> DBUF [A];          // read Ry into DBUF[A]
DPRF[ FBUF[Rz] ] -> DBUF [B];          // read Rz into DBUF[B]
FBUF[OPCODE]    -> DBUF[OPCODE];       // copy opcode from FBUF to
                                          DBUF
FBUF[Rx]        -> DBUF[Rx];           // copy Rx register specifier from
                                          FBUF to DBUF
```

EX stage (cycle 3):

```
DBUF[A] + DBUF [B] -> EBUF[Result];    // perform addition
DBUF[OPCODE]    -> EBUF[OPCODE];       // copy opcode from DBUF to
                                          EBUF
DBUF[Rx]        -> EBUF[Rx];           // copy Rx register specifier
                                          from DBUF to EBUF
```

MEM stage (cycle 4):

```
DBUF -> MBUF;       // The MEM stage has nothing to contribute towards
                       the execution of the Add instruction; so simply copy
                       the DBUF to MBUF
```

WB stage (cycle 5):

```
MBUF[Result] -> DPRF [ MBUF[Rx] ];  // write back the result of the
                                       addition into the register
                                       specified by Rx
```

Example 8:

Considering only the Add instruction, quantify the sizes of the various buffers between the stages of the above pipeline.

Answer:

Size of FBUF (same as the size of an instruction in LC-2200) = **32 bits**

Size of DBUF:

Size of contents of Ry register in DBUF[A] = 32 bits
Size of contents of Rz register in DBUF[B] = 32 bits
Size of opcode in DBUF[opcode] = 4 bits
Size of Rx register specifier in DBUF[Rx] = 4 bits
Total size (sum of the above fields) = **72 bits**

Size of EBUF:
Size of result of addition in EBUF[result] = 32 bits
Size of opcode in EBUF[opcode] = 4 bits
Size of Rx register specifier in EBUF[Rx] = 4 bits
Total size (sum of the above fields) = **40 bits**

Size of MBUF (same as EBUF) = **40 bits**

5.12.2 Design of the Pipeline Registers

While we looked at the passage of a single instruction through the pipeline in the previous section, it should be clear that each stage of the pipeline is working on a different instruction in each clock cycle. The reader should work out the actions taken by each stage of the pipeline for the different instructions in LC-2200 (see problems at the end of this chapter). This exercise is similar to the design of the FSM control sequences that we undertook in Chapter 3 for the sequential implementation of LC-2200. Once such a design is complete, then the size of the pipeline register for each stage can be determined as the maximum required for the passage of any instruction. The pipeline register at the output of the ID/RR stage will have the maximal content since we do not yet know what the instruction is. The interpretation of the contents of a pipeline register will depend on the stage and the opcode of the instruction that is being acted upon by that stage.

A generic layout of the pipeline register is as shown in Figure 5.6d. Opcode will always occupy the same position in each pipeline register. In each clock cycle, each stage interprets the remaining fields of the pipeline register at its input based on the opcode and takes the appropriate datapath actions (similar to what we detailed for the Add instruction in Section 5.12.1) .

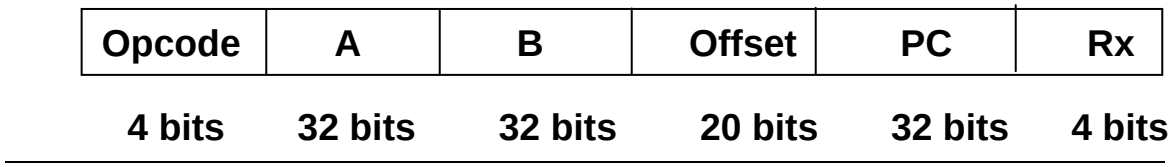
Figure 5.6d: A generic layout of a pipeline register

Example 9:
Design the DBUF pipeline register for LC-2200. Do not attempt to optimize the design by overloading the different fields of this register.

Answer:
DBUF has the following fields:

Opcode (needed for all instructions)	4 bits
A (needed for R-type)	32 bits
B (needed for R-type)	32 bits
Offset (needed for I-type and J-type)	20 bits
PC value (needed for BEQ)	32 bits
Rx specifier (needed for R-, I-, and J-type)	4 bits

Layout of the DBUF pipeline register:



5.12.3 Implementation of the stages

In a sense, the design and implementation of a pipeline processor may be simpler than a non-pipelined processor. This is because the pipelined implementation modularizes the design. This modularity leads to the same design advantages that accrue in writing a large software system as a composite of several smaller modules. The layout and interpretation of the pipeline registers are analogous to well-defined interfaces between components of a large software system. Once we complete the layout and interpretation of the pipeline registers, we can get down to the datapath actions needed for each stage in complete isolation of the other stages. Further, since the datapath actions of each stage happen in one clock cycle, the design of each stage is purely combinational. At the beginning of each clock cycle, each stage interprets the input pipeline register, carries out the datapath actions using the combinational logic for this stage, and writes the result of the datapath action into its output pipeline register.

Example 10:
Design and implement the datapath for the ID/RR stage of the pipeline to implement the LC-2200 instruction set. You can use any available logic design tool to do this exercise.

Answer:
Figures 5.6, 5.6a, and 5.6b are the starting points for solving this problem. The datapath elements have to be laid out. Using the format of the instruction the register files have to be accessed and put into the appropriate fields of DBUF. The offset and the opcode fields of the instruction have to be copied from FBUF to DBUF. The PC value has to be put into the appropriate field of DBUF. Completing this example is left as an exercise for the reader.

5.13 Hazards

Though the sandwich assembly line serves as a good analogy for the pipelined processor, there are several twists in the instruction pipeline that complicate its design. These issues are summed up as *pipeline hazards*. Specifically, there are three types of hazards: **structural, data, control**.

As we will see shortly, the effect of all these hazards is the same, namely, **reduce** the pipeline efficiency. In other words, the pipeline will execute less than one instruction every clock cycle. Recall, however, that the pipeline is synchronous. That is, in each clock cycle every stage is working on an instruction that has been placed on the pipeline register by the preceding stage. Just like air bubbles in a water pipe, if a stage is not ready to send a valid instruction to the next stage, it should place the equivalent of an "air bubble", a dummy instruction that does nothing, in the pipeline register. We refer to this as a *NOP (No-Operation)* instruction.

We will include such a NOP instruction in the repertoire of the processor. In the subsequent discussions on hazards, we will see how the hardware automatically generates such NOPs when warranted by the hazard encountered. However, this need not be the case always. By exposing the structure of the pipeline to the software, we can make the system software, namely, the compiler, responsible for including such NOP instructions in the source code itself. We will return to this possibility later (see Section 5.13.4).

Another way of understanding the effect of bubbles in the pipeline is that the average CPI of instructions goes above 1. It is important to add a cautionary note to the use of the CPI metric. Recall that the execution time of a program in clock cycles is the product of CPI and the number of instructions executed. Thus, CPI by itself does not tell the whole story of how good an architecture or the implementation is. In reality, the compiler and the architecture in close partnership determine the program execution time. For example, unoptimized code that a compiler generates may have a lower CPI than the optimized code. However, the execution time may be much more than the optimized code. The reason is the optimization phase of the compiler may have gotten rid of a number of useless instructions in the program thus reducing the total number of instructions executed. But this may have come at the cost of increasing the three kinds of hazards and hence the average CPI of the instructions in the optimized code. Yet, the net effect of the optimized code may be a reduction in program execution time.

5.13.1 Structural hazard

We already alluded to the structural hazard. This comes about primarily due to the limitations in the hardware resources available for concurrent operation of the different stages. For example, a single data bus in the non-pipelined version is a structural hazard for a pipelined implementation. Similarly, a single ALU is another structural hazard. There are two solutions to this problem: live with it or fix it. If the hazard is likely to occur only occasionally (for some particular combinations of instructions passing through the pipeline simultaneously) then it may be prudent not to waste additional hardware resources to fix the problem. As an example, let us assume that our pointy-haired manager has told us we can use only one ALU in the EX unit. So every time we

encounter the BEQ instruction, we spend two clock cycles in the EX unit to carry out the two arithmetic operations that are needed, one for the comparison and the other for the address computation. Of course, the preceding and succeeding stages should be made aware that the EX stage will occasionally take two cycles to do its operation. Therefore, the EX unit should tell the stages preceding it (namely, IF and ID/RR) not to send a new instruction in the next clock cycle. Basically there is a *feedback line* that each of the preceding stages looks at to determine if they should just "pause" in a given cycle or do something useful. If a stage decides to "pause" in a clock cycle it just does not change anything in the output buffer.

The succeeding stages need to be handled differently from the preceding stages. Specifically, the EX stage will write a NOP opcode in its output buffer for the opcode field. NOP instruction is a convenient way of making the processor execute a "dummy" instruction that has no impact on the actual program under execution. Essentially, via the NOP instruction, we have introduced a "bubble" in the pipeline between the BEQ and the instruction that preceded it.
The passage of a BEQ instruction is pictorially shown in a series of timing diagrams in Figure 5.7. We pick up the action from cycle 2, when the BEQ instruction is in the ID/RR stage. A value of "STAY" (equal to binary 1) on the feedback line tells the preceding stage to remain in the same instruction and not to send a new instruction at the end of this cycle.

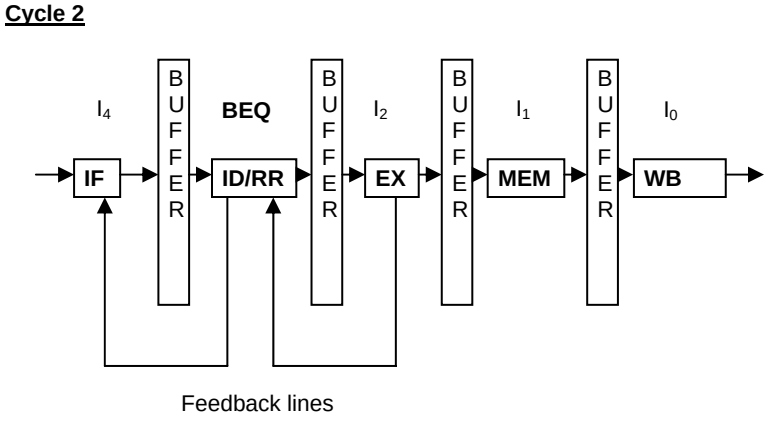


Figure 5.7 (a)

Cycle 3

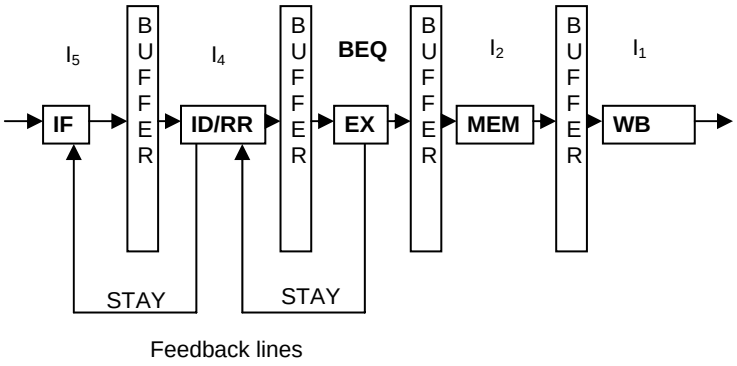


Figure 5.7 (b)

Cycle 4

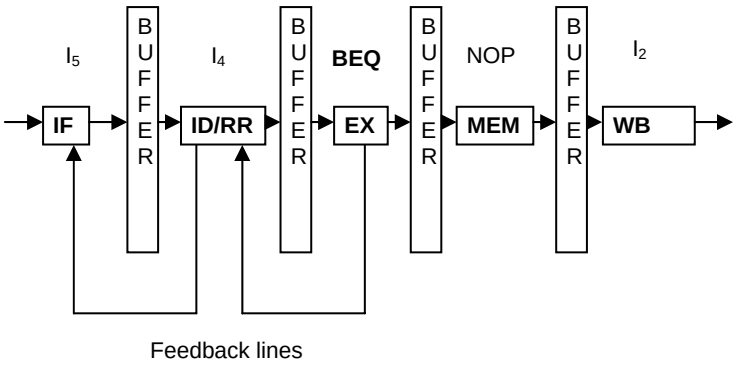


Figure 5.7 (c)

Cycle 5

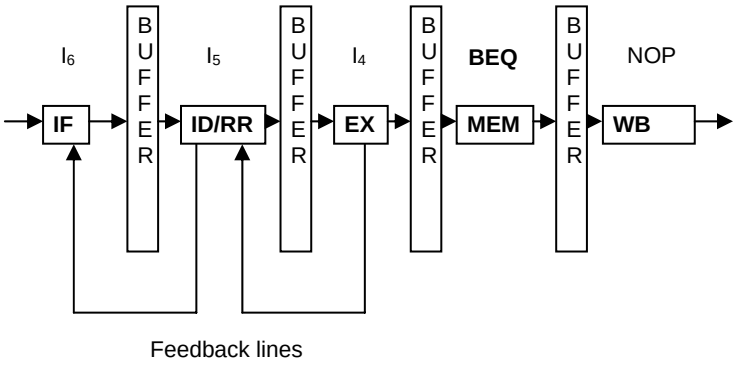


Figure 5.7 (d)

Figure 5.7: Illustration of structural Hazard

Essentially, such a "bubble" in the pipeline brings down the pipeline efficiency since the effective CPI goes above 1 due to the bubble.

If on the other hand, it is determined that such an inefficiency is unacceptable then we can fix the structural hazard by throwing hardware at the problem. This is what we did in our earlier discussion when we added an extra ALU in the EX stage to overcome this structural hazard.

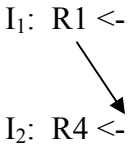
A quick note on terminology:

- The pipeline is *stalled* when an instruction cannot proceed to the next stage.
- The result of such a stall is to introduce a *bubble* in the pipeline.
- A NOP instruction is the manifestation of the bubble in the pipeline. Essentially a stage executing a NOP instruction does nothing for one cycle. Its output buffer remains unchanged from the previous cycle.

You may notice that we use stalls, bubbles, and NOPs interchangeably in the textbook. They all mean the same thing.

5.13.2 Data Hazard

Consider the following two instructions occurring in the execution order of a program:

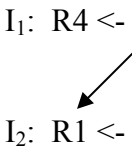


(7)

Figure 5.8a: RAW hazard

I_1 reads the values in registers R2 and R3 and writes the result of the addition into register R1. The next instruction I_2 reads the values in registers R1 (written into by the previous instruction I_1) and R5 and writes the result of the addition into register R4. The situation presents a *data hazard* since there is a dependency between the two instructions. In particular, this kind of hazard is called a *Read After Write (RAW)* data hazard. Of course, the two instructions need not occur strictly next to each other. So long as there is a data dependency between any two instructions in the execution order of the program it amounts to a data hazard.

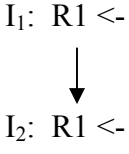
There are two other kinds of data hazards.



(8)

Figure 5.8b: WAR hazard

The situation presented above is called a *Write After Read (WAR)* data hazard. I_2 is writing a new value to a register while I_1 is reading the old value in the same register.



(9)

Figure 5.8c: WAW hazard

The situation presented above is called a *Write After Write (WAW)* data hazard. I_2 is writing a new value to a register that is also the target of a previous instruction.

These data hazards pose no problem if the instructions execute one at a time in program order, as would happen in a non-pipelined processor. However, they could lead to problems in a pipelined processor as explained in the next paragraph. It turns out that for the simple pipeline we are considering, WAR and WAW hazards do not pose much of a problem, and we will discuss solutions to deal with them in Section 5.13.2.4. First let us deal with the RAW hazard.

5.13.2.1 RAW Hazard

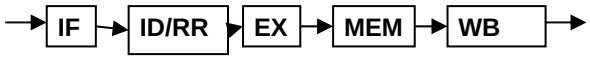
Let us trace the flow of instructions represented by (7) (Figure 5.8(a)) through the pipeline.

I_2 I_1

When I_1 is in the EX stage of the pipeline, I_2 is in ID/RR stage and is about to read the values in registers R1 and R5. This is problematic since I_1 has not computed the new value of R1 yet. In fact, I_1 will write the newly computed value into R1 only in the WB stage of the pipeline. If I_2 is allowed to read the value in R1 as shown in the above picture, it will read to an erroneous execution of the program. We refer to this situation as *semantic inconsistency*, when the intent of the programmer is different from the actual execution. Such a problem would never occur in a non-pipelined implementation since the processor executes one instruction at a time.
The problem may not be as severe if the two instructions are not following one another. For example, consider:

I_1: R1 <- R2 + R3
I_x: R8 <- R6 + R7
I_2: R4 <- R1 + R5

In this case, the pipeline looks as follows:

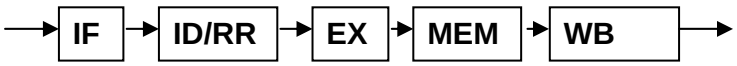


I_2 I_x I_1

I_1 has completed execution. However, the new value for R1 will be written into the register only when I_1 reaches the WB stage. Thus if there is a RAW hazard for any of the subsequent three instructions following I_1 it will result in a semantic inconsistency.

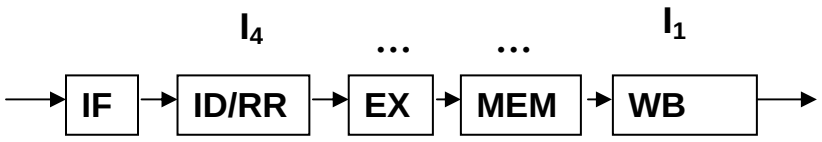
Example 11:
Consider

I_1: ld R1, MEM
<2 unrelated instructions>
I_4: R4 <- R1 + R5



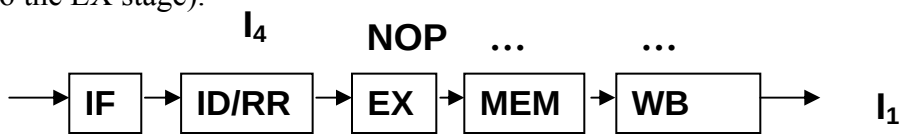
I_4 and I_1 are separated by two unrelated instructions in the pipeline as shown above. How many bubbles will result in the pipeline due to the above execution?

Answer:
The state of the pipeline when I_4 gets to ID/RR stage is shown below:



I_1 will write the value into R1 only at the end of the current cycle. Therefore, I_4 cannot read the register in this cycle and get the correct value.

Thus there is a 1 cycle delay leading to one bubble (NOP instruction passed down from the ID/RR stage to the EX stage).



Example 12:

I_1: R1 <- R2 + R3
I_2: R4 <- R4 + R3
I_3: R5 <- R5 + R3
I_4: R6 <- R1 + R6

As shown above, the sequence of instructions I_1 through I_4 are just about to enter the 5-stage pipeline.
(a)
In the table below, show the passage of these instructions through the pipeline until all 4 instructions have been completed and retired from the pipeline; "retired" from the pipeline means that the instruction is not in any of the 5-stages of the pipeline.

Answer:

Cycle Number	IF	ID/RR	EX	MEM	WB
1	I_1	-	-	-	-
2	I_2	I_1	-	-	-
3	I_3	I_2	I_1	-	-
4	I_4	I_3	I_2	I_1	-
5	-	I_4	I_3	I_2	I_1
6	-	I_4	NOP	I_3	I_2
7	-	-	I_4	NOP	I_3
8	-	-	-	I_4	NOP
9	-	-	-	-	I_4

(b) Assuming that the program contains just these 4 instructions, what is the average CPI achieved for the above execution?

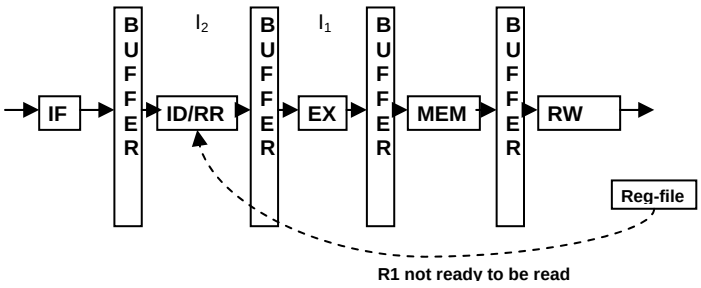
Answer:

Four instructions are retired from the pipeline in 9 cycles. Therefore the average CPI experienced by these instructions:

Avg CPI = 9/4 = 2.25

5.13.2.2 Solving the RAW Data Hazard Problem: Data Forwarding

A simple solution to this problem is similar to the handling of the structural hazard. We simply stall the instruction that causes the RAW hazard in the ID/RR stage until the register value is available. In the case of (7) shown in Figure 5.8(a), the ID/RR stage holds I_2 for 3 cycles until I_1 retires from the pipeline. For those three cycles, bubbles (in the form of NOP instructions manufactured by the ID/RR stage) are sent down the pipeline. For the same reason, the preceding stage (IF) is told to stay on the same instruction and not fetch new instructions.



(a)

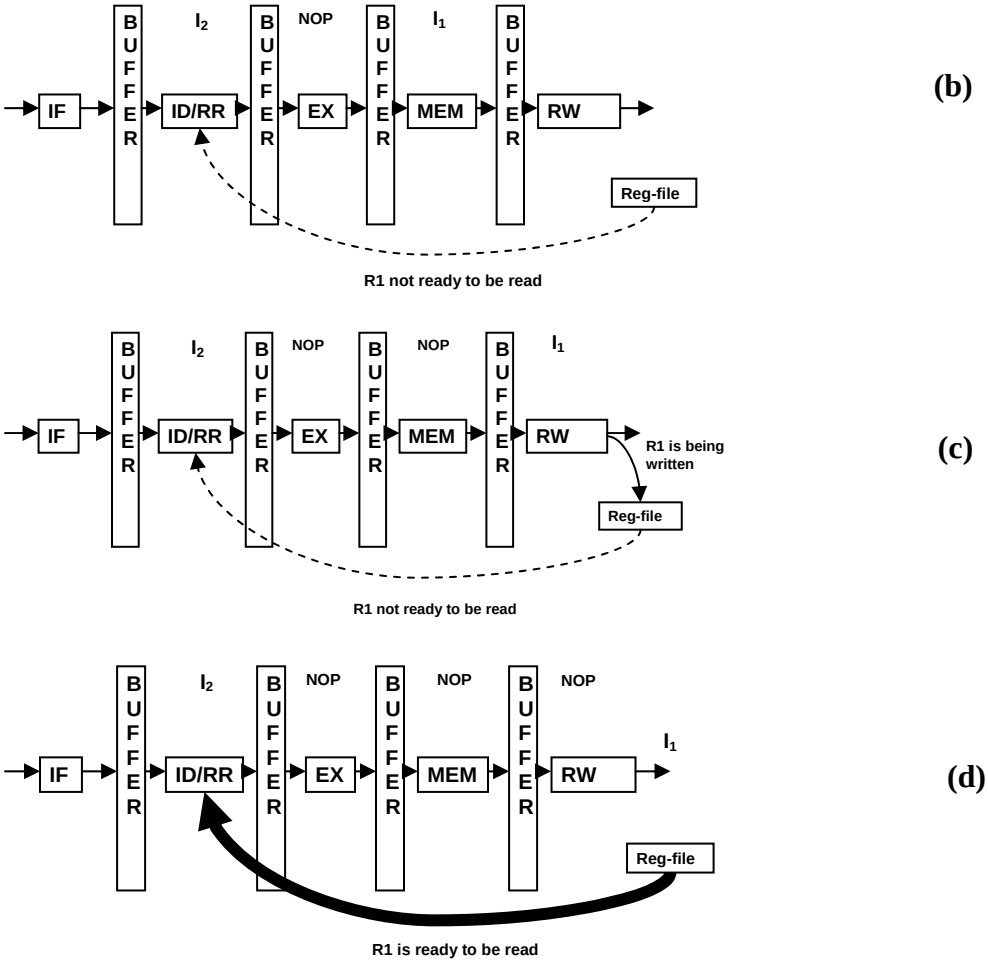
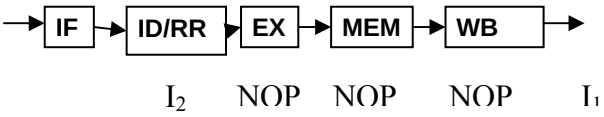


Figure 5.9: RAW hazard due to (7) shown in Figure 5.8a

The series of pictures shown in Figure 5.9 illustrate the stalling of the pipeline due to the data hazard posed by (7) (Figure 5.8a).

In the clock cycle following the picture shown below, normal pipeline execution will start with the IF stage starting to fetch new instructions.



The ID/RR needs some hardware help to know that it has to stall. The hardware for detecting this is very simple. In the register file shown below (Figure 5.10), the **B** bit is the register **busy** bit, and there is one bit per processor register. The ID/RR stage determines the destination register for an instruction and sets the appropriate B bit for that register in the DPRF. I_1 would have set the B bit for register R1. It is the responsibility of the WB stage to clear this bit when it writes the new value into the register file. Therefore, when I_2 gets to ID/RR stage it will find B bit set for R1 and hence stall until the busy condition goes away (3 cycles later).

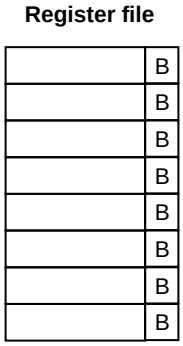


Figure 5.10: Register file with busy bits for each register

Let us see how we can get rid of the stalls induced by the RAW hazard. It may not be possible to get rid of stalls altogether, but the number of stalls can certainly be minimized with a little bit of hardware complexity. The original algorithm, which bears the name of its inventor, Tomasulo, made its appearance in the IBM 360/91 processor in the 60's. The idea is quite simple. In general, 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[5].

With respect to our simple pipeline, the only stage that reads a register is the ID/RR stage. Let us examine what we need to do to enable this data forwarding. We add a bit called RP (read pending) to each of the registers in the register file (see Figure 5.11).

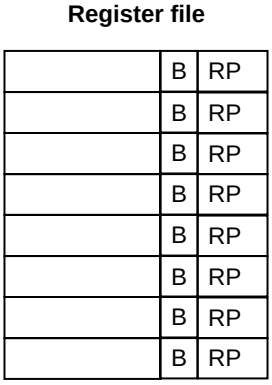


Figure 5.11: Register file with busy and read-pending bits for each register

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. The hardware complexity comes from having to run wires from these stages back to the ID/RR stage. One might wonder why not let

[5] The solution we describe for RAW hazard for our simple pipeline is inspired by the Tomasulo algorithm but it is nowhere near the generality of that original algorithm.

these stages simply write to the register file if the RP bit is set. In principle, they could; however, as we said earlier, writing to the register file and clearing the B bit is the responsibility of the WB stage of the pipeline. Allowing any stage to write to the register file will increase the hardware complexity. Besides, this increased hardware complexity does not result in any performance advantage since the instruction (in the ID/RR stage) that may need the data is getting it via the forwarding mechanism.

Revisiting the RAW hazard shown in Figure 5.8(a), data forwarding completely eliminates any bubbles as shown in Figure 5.12.

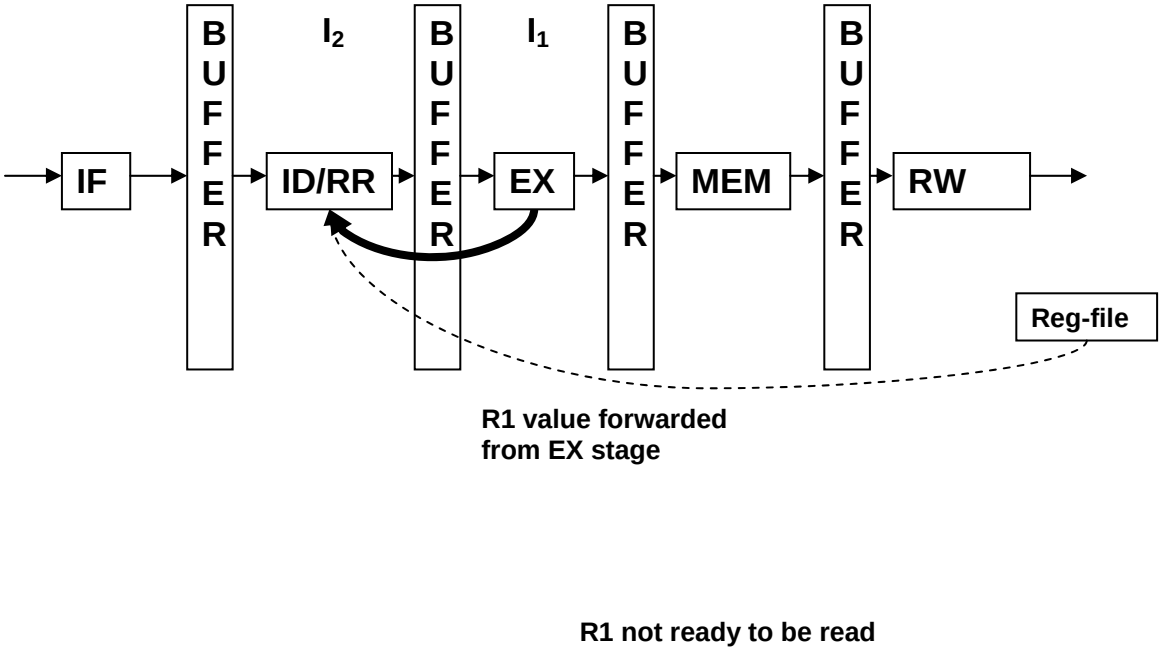


Figure 5.12: Data forwarding[6] to solve RAW hazard shown in Figure 5.8a

Consider the following stream of instructions:

I_1
I_2
I_3
I_4

If in the above sequence of instructions, $I_1$is an arithmetic/logic instruction that generates a new value for a register, and any of the subsequent three instructions (I_2, I_3, or I_4) need the same value, then this forwarding method will eliminate any stalls due to the RAW hazard. Table 5.3 summarizes the number of bubbles introduced in the pipeline with and without data forwarding for RAW hazard shown in Figure 5.8a, as a function of the

[6] "Forwarding" seems counter-intuitive given the arrow is going backwards! I_1 is ahead of I_2 in the execution order of the program and it "forwards" the value it computed to I_2.

number of unrelated instructions separating I_1 and I_2 in the execution order of the program.

Number of unrelated instructions Between I_1 and I_2	Number of bubbles Without forwarding	Number of bubbles With forwarding
0	3	0
1	2	0
2	1	0
3 or more	0	0

Table 5.3: Bubbles created in pipeline due to RAW Hazard shown in Figure 5.8a

---

Example 13:

I_1: R1 <- R2 + R3
I_2: R4 <- R4 + R3
I_3: R5 <- R5 + R3
I_4: R6 <- R1 + R6

This is the same sequence of instructions as in Example 12. Assuming data forwarding, show the passage of instructions in the chart below. What the average CPI experienced by these instructions?

Answer:
As we see in Table 5.3, with data forwarding, there will be no more bubbles in the pipeline due to RAW hazard posed by arithmetic and logic instructions since there is data forwarding from every stage to the preceding stage. Thus, as can be seen in the chart below, there are no more bubbles in the pipeline for the above sequence.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	I_1	-	-	-	-
2	I_2	I_1	-	-	-
3	I_3	I_2	I_1	-	-
4	I_4	I_3	I_2	I_1	-
5	-	I_4	I_3	I_2	I_1
6	-	-	I_4	I_3	I_2
7	-	-	-	I_4	I_3
8	-	-	-	-	I_4

Average CPI = 8/4 = **2**.

5.13.2.3 Dealing with RAW Data Hazard introduced by Load instructions

Load instructions introduce data hazard as well. Consider the following sequence:

I_1: Load R1, 0(R2)

↘

I_2: Add R4, R1, R4

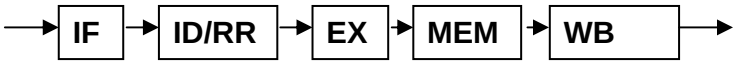
(4)

In this case, the new value for R1 is not available until the MEM stage, Therefore, even with forwarding a 1 cycle stall is inevitable if the RAW hazard occurs in the immediate next instruction as shown above in (4). The following exercise goes into the details of bubbles experienced due to a load instruction in the pipeline.

Example 14:

Consider

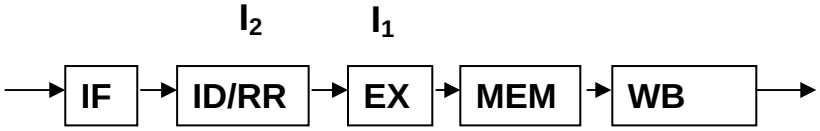
I_1: ld R1, MEM
I_2: R4 <- R1 + R5



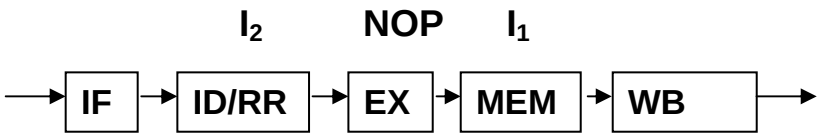
(a)
I_2 immediately follows I_1 as shown above. Assuming there is register forwarding from each stage to the ID/RR stage, how many bubbles will result with the above execution?
(b)
How many bubbles will result with the above execution if there was no register forwarding?

Answer:

(a)
The state of the pipeline when I_2 gets to ID/RR stage is shown below:



I_1 will have a value for R1 only at the end of the MEM cycle. Thus there is a 1 cycle delay leading to one bubble (NOP instruction passed down from the ID/RR stage to the EX stage) as shown below.



The MEM stage will simultaneously write to its output buffer (MBUF) and forward the value it read from the memory for R1 to the ID/RR stage, so that the ID/RR stage can use this value (for I_2) to write into the pipeline register (DBUF) at the output of the ID/RR stage. Thus, there is no further delay in the pipeline. This is despite the fact that I_1 writes the value into R1 only at the end of WB stage. The chart below shows the progression of the two instructions through the pipeline.

Cycle Number	**IF**	**ID/RR**	**EX**	**MEM**	**WB**
1	**I_1**	-	-	-	-
2	**I_2**	**I_1**	-	-	-
3	-	**I_2**	**I_1**	-	-
4	-	**I_2**	**NOP**	**I_1**	-
5	-	-	**I_2**	**NOP**	**I_1**
6	-	-	-	**I_2**	**NOP**
7	-	-	-	-	**I_2**

So, total number of bubbles for this execution with forwarding = **1**.

(b)
Without the forwarding, I_2 cannot read R1 until I_1 has written the value into R1. Writing into R1 happens at the end of the WB stage. Note that the value written into R1 is available for reading only in the following cycle. The chart below shows the progression of the two instructions through the pipeline.

Cycle Number	**IF**	**ID/RR**	**EX**	**MEM**	**WB**
1	**I_1**	-	-	-	-
2	**I_2**	**I_1**	-	-	-
3	**I_3**	**I_2**	**I_1**	-	-
4	**I_4**	**I_2**	**NOP**	**I_1**	-
5	-	**I_2**	**NOP**	**NOP**	**I_1**
6	-	**I_2**	**NOP**	**NOP**	**NOP**
7	-	-	**I_2**	**NOP**	**NOP**
8	-	-	-	**I_2**	**NOP**
9	-	-	-	-	**I_2**

So, total number of bubbles for this execution without forwarding = **3**.

Table 5.4 summarizes the number of bubbles introduced in the pipeline with and without data forwarding for RAW hazard due to Load instruction, as a function of the number of unrelated instructions separating I_1 and I_2 in the execution order of the program.

Number of unrelated instructions Between I_1 and I_2	**Number of bubbles Without forwarding**	**Number of bubbles With forwarding**
0	3	1
1	2	0
2	1	0
3 or more	0	0

Table 5.4: Bubbles created in pipeline due to Load instruction induced RAW Hazard

5.13.2.4 Other types of Data Hazards

Other types of hazards, WAR and WAW pose their own set of problems as well for the pipelined processor. However, these problems are much less severe compared to the RAW problem in terms of affecting pipeline processor performance. For example, WAR is not a problem at all 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. For the WAW problem, a simple solution would be to 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. This instruction will be able to proceed once the preceding instruction that set the B bit clears it in the WB stage.

Let us understand the source of a WAW hazard. WAW essentially means that a register value that was written by one instruction is being over-written by a subsequent instruction with no intervening read of that register. One can safely conclude that the first write was a useless one to start with. After all, if there were a read of the register after the first write, then the ensuing RAW hazard would have over-shadowed the WAW hazard. This begs the question as to why a compiler would generate code with a WAW hazard. There are several possible answers to this question. We will defer discussing this issue until later (please see Section 5.15.4). Suffice it to say at this point that WAW hazards could occur in practice, and it is the responsibility of the hardware to deal with them.

5.13.3 Control Hazard

This hazard refers to breaks in the sequential execution of a program due to branch instructions. Studies on the dynamic frequency of instructions in benchmark programs show that conditional branch instructions occur once in every 3 or 4 instructions. Branches cause disruption to the normal flow of control and are detrimental to pipelined

processor performance. The problem is especially acute with conditional branches since the outcome of the branch is typically not known until much later in the pipeline.

Let us assume that the stream of instructions coming into the pipeline in program order is as follows:

```
BEQ
ADD
NAND
LW
…
…
```

Cycle	IF*	ID/RR	EX	MEM	WB
1	BEQ				
2	ADD	BEQ			
3	ADD+	NOP	BEQ		
4	NAND	ADD	NOP	BEQ	
5	LW	NAND	ADD	NOP	BEQ

***** We do not actually know what the instruction is in the Fetch Stage

+ 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.

Figure 5.13: Conservative Approach to Handling Branches

One conservative way of handling branches is to stop new instructions from entering the pipeline when the decode stage encounters a branch instruction. Once the branch is resolved, normal pipeline execution can resume either along the sequential path of control or along the target of the branch. Figure 5.13 shows the flow of instructions for such a conservative arrangement. For BEQ instruction, we know the outcome of the branch at the end of the EX cycle. If the *branch is not taken* (i.e. we continue along the sequential path) then the IF stage already has the right instruction fetched from memory and ready to be passed on to the ID/RR stage. Therefore, we stall the pipeline for 1 cycle and continue with the ADD instruction as shown in Figure 5.13. However, if the *branch is taken*, then we have to start fetching the instruction from the newly computed address of the PC which is clocked into PC only at the end of the EX cycle for the BEQ instruction. So, in cycle 4 we will start fetching the correct next instruction from the target of the branch. Hence, there will be a 2-cycle stall if the branch is taken with the above scheme.

Example 15:

Given the following sequence of instructions:

```
            BEQ     L1
            ADD
            LW
            ….
L1          NAND
            SW
```

The hardware uses a conservative approach to handling branches.

(a) Assuming that the branch is not taken, show the passage of instructions through the pipeline by filling the chart below until three instructions have successfully retired from the pipeline. What is the observed CPI for these three instructions?

(b) Assuming that the branch is taken, show the passage of instructions through the pipeline by filling the chart below until three instructions have successfully retired from the pipeline. What is the observed CPI for these three instructions?

Answer :

(a) The time chart for the above sequence when the *branch is not taken* is shown below. Note that ADD instruction is stalled in IF stage for 1 cycle until BEQ instruction is resolved at the end of the EX stage.

Cycle Number	**IF**	**ID/RR**	**EX**	**MEM**	**WB**
1	**BEQ**	**-**	**-**	**-**	**-**
2	**ADD**	**BEQ**	**-**	**-**	**-**
3	**ADD**	**NOP**	**BEQ**	**-**	**-**
4	**LW**	**ADD**	**NOP**	**BEQ**	**-**
5	**-**	**LW**	**ADD**	**NOP**	**BEQ**
6	**-**	**-**	**LW**	**ADD**	**NOP**
7			**-**	**LW**	**ADD**
8				**-**	**LW**

The average CPI for these three instructions = 8/3 = **2.666**.

(b) The time chart for the above sequence when the *branch is taken* is shown below. Note that ADD instruction in the IF stage has to be converted into a NOP in cycle 4 since the branch is taken. A new fetch has to be instantiated from the target of the branch in cycle 4, thus resulting in a 2 cycle stall of the pipeline.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	ADD	NOP	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ
6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

The average CPI for the three instructions = 9/3 = **3**.

Of course, we can do even better if we are ready to throw some hardware at the problem. In fact, there are several different approaches to solving hiccups in pipelines due to branches. This has been a fertile area of research in the computer architecture community. Especially since modern processors may have deep pipelines (with more than 20 stages), it is extremely important to have a speedy resolution of branches to ensure high performance.

In this chapter, we will present a small window into the range of possibilities for solving this problem. You will have to wait to take a senior level course in computer architecture to get a more detailed treatment of this subject.

5.13.3.1 Dealing with branches in the pipelined processor

1. **Delayed branch:** The idea here is to assume that the instruction following the branch executes irrespective of the outcome of the branch. This simplifies the hardware since there is no need to terminate the instruction that immediately follows the branch instruction. The responsibility of ensuring the semantic correctness underlying this assumption shifts to the compiler from the hardware. The default is to stash a NOP instruction in software (i.e. in the memory image of the program) following a branch. You may ask what the use of this approach is. The answer is a smart compiler will be able to do program analysis and find a useful instruction that does not affect the program semantics instead of the NOP. The instruction slot that immediately follows the branch is called a **delay slot.** Correspondingly, this technique is called **delayed branch.**

   The code fragments shown in Figure 5.14 and 5.15 show how a compiler may find useful instructions to stick into delay slots. In this example, every time there is

branch instruction, the instruction that immediately follows the branch is executed regardless of the outcome of the branch. If the branch is unsuccessful, the pipeline continues without a hiccup. If the branch is successful, then (as in the case of branch misprediction) the instructions following the branch except the immediate next one are terminated.

```
;     Code before optimization to fill branch delay slots
;     Add 7 to each element of a ten-element array
;     whose address is in a0
addi  t1, a0, 40              ; When a0=t1 we are done
loop: beq   a0, t1, done
      nop                     ; branch delay slot       [1]
      lw    t0, 0(a0)                                   [2]
      addi  t0, t0, 7
      sw    t0, 0(a0)
      addi  a0, a0, 4                                   [3]
      beq   zero, zero, loop
      nop                     ; branch delay slot       [4]
done: halt
```

Figure 5.14: Delayed branch: delay slots with NOPs

```
;     Code after optimization to fill branch delay slots
;     Add 7 to each element of a ten-element array
;     whose address is in a0
addi  t1, a0, 40              ; When a0=t1 we are done
loop: beq   a0, t1, done
      lw    t0, 0(a0)         ; branch delay slot       [2]
      addi  t0, t0, 7
      sw    t0, 0(a0)
      beq   zero, zero, loop
      addi  a0, a0, 4         ; branch delay slot       [3]
done: halt
```

Figure 5.15: Delayed branch: NOPs in delay slots replaced with useful instructions

With reference to Figure 5.14, the compiler knows that the instructions labeled [1] and [4] are always executed. Therefore, the compiler initially sticks NOP instructions as placeholders in those slots. During optimization phase, the compiler recognizes that instruction [2] is benign[7] from the point of view of the program semantics, since it simply loads a value into a temporary register. Therefore, the compiler replaces the NOP instruction [1] by the load instruction [2] in Figure 5.15. Similarly, the last

[7] This is for illustration of the concept of delayed branch. Strictly speaking, executing the load instruction after the loop has terminated may access a memory location that is non-existent.

branch instruction in the loop is an unconditional branch, which is independent of the preceding add instruction [3]. Therefore, the compiler replaces the NOP instruction [4] by the add instruction [3] in Figure 5.15.

Some machines may use even multiple delay slots to increase pipeline efficiency. The greater the number of delay slots the lesser the number of instructions that need terminating upon a successful branch. However, this increases the burden on the compiler to find useful instructions to fill in the delay slot, which may not always be possible.

Delayed branch seems like a reasonable idea especially for shallow pipelines (less than 10 stages) since it simplifies the hardware. However, there are several issues with this idea. The most glaring problem is that it exposes the details of the microarchitecture to the compiler writer thus making a compiler not just ISA-specific but processor-implementation specific. Either it necessitates re-writing parts of the compiler for each generation of the processor or it limits evolution of the microarchitecture for reasons of backward compatibility. Further, modern processors use deep pipelining. For example, Intel Pentium line of processors has pipeline stages in excess of 20. While the pipelines have gotten deeper, the sizes of basic blocks in the program that determine the frequency of branches have not changed. If anything, the branches have become more frequent with object-oriented programming. Therefore, delayed branch has fallen out of favor in modern processor implementation.

2. **Branch prediction:** The idea here is to assume the outcome of the branch to be one way and start letting instructions into the pipeline even upon detecting a branch. For example, for the same sequence shown in Figure 5.13, let us predict that the outcome is negative (i.e., the sequential path is the winner). Figure 5.16 shows the flow of instructions in the pipeline with this prediction. As soon as the outcome is known (when BEQ is in the EX stage), the result is fed back to the preceding stages. Figure 5.16 shows the happy state when the outcome is as predicted. The pipeline can continue without a hiccup.

Cycle	IF*	ID/RR	EX	MEM	WB
1	BEQ				
2	ADD	BEQ			
3	NAND	ADD	BEQ		
4	LW	NAND	ADD	BEQ	
5	...	LW	NAND	ADD	BEQ

* We do not actually know what the instruction is in the Fetch Stage

Figure 5.16: Branch Prediction

Of course, there are chances of misprediction and we need to be able to recover from such mispredictions. Therefore, we need a hardware capability to terminate the

instructions that are in partial execution in the preceding stages of the pipeline, and start fetching from the alternate path. This termination capability is often referred to as *flushing*. Another feedback line labeled "flush", shown pictorially in Figure 5.17, implements this hardware capability. Upon receiving this flush signal, both the IF and ID/RR stages abandon the partial execution of the instructions they are currently dealing with and start sending "bubbles" down the pipeline. There will be 2 bubbles (i.e., 2-cycle stall) corresponding to the ADD and NAND instructions in Figure 5.17 before normal execution can begin.

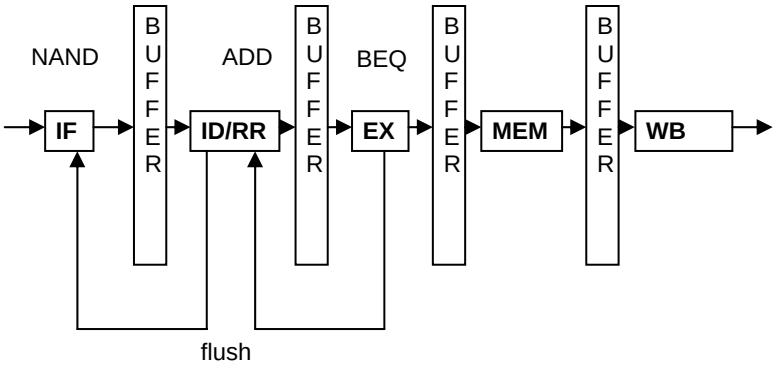


Figure 5.17: Pipeline with flush control lines

Example 16:

Given the following sequence of instructions:

```
        BEQ   L1
        ADD
        LW
        ….
L1      NAND
        SW
```

The hardware uses branch prediction (branch will not be taken).

(a) Assuming that the prediction turns out to be true, show the passage of these instructions through the pipeline by filling the chart below until three instructions have successfully retired from the pipeline. What is the observed CPI for these three instructions?

(b) Assuming that the prediction turns out to be false, show the passage of these instructions through the pipeline by filling the chart below until three instructions have successfully retired from the pipeline. What is the observed CPI for these three instructions?

Answer:

(a) The time chart for the above sequence when the prediction is true is shown below. The branch prediction logic starts feeding in the instructions from the sequential path into the pipeline and they all complete successfully.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	LW	ADD	BEQ	-	-
4	-	LW	ADD	BEQ	-
5	-	-	LW	ADD	BEQ
6	-	-	-	LW	ADD
7	-	-	-	-	LW

At the end of 7 cycles, 3 instructions complete,
yielding an average CPI of 7/3 = **2.333**

(b) The time chart for the above sequence when the prediction turns out to be false is shown below. Note that ADD and LW instructions are flushed in their respective stages as soon as BEQ determines that the outcome of the branch has been mispredicted. This is why in cycle 4, ADD and LW have been replaced by NOPs in the successive stages. The PC is set at the end of the EX cycle by the BEQ instruction (cycle 3) in readiness to start fetching the correct instruction from the target of the branch in cycle 4.

Cycle Number	IF	ID/RR	EX	MEM	WB
1	BEQ	-	-	-	-
2	ADD	BEQ	-	-	-
3	LW	ADD	BEQ	-	-
4	NAND	NOP	NOP	BEQ	-
5	SW	NAND	NOP	NOP	BEQ
6	-	SW	NAND	NOP	NOP
7	-	-	SW	NAND	NOP
8			-	SW	NAND
9				-	SW

The average CPI for the three instructions = 9/3 =**3**.
Note that this average CPI is the same as what we obtained earlier with no branch prediction and the branch is taken.

You may be wondering how one can predict the outcome of a branch since on the surface the odds seem evenly split between taking and not taking. However, programs have a lot more structure that aids prediction. For example, consider a loop. This usually consists of a series of instructions; at the end of the loop, a conditional branch takes control back to the top of the loop or out of the loop. One can immediately see that the outcome of this conditional branch is heavily biased towards going back to the top of the loop. Thus, the branch prediction technique relies on such structural properties of programs.

There has been considerable research invested in understanding the properties of branches that occur in programs and designing prediction schemes to support them. As we already mentioned, loops and conditional statements are the high-level constructs that lead to conditional branch instructions. 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.

Branch prediction is in the purview of the compiler based on program analysis. The ISA has to support the compiler by providing a mechanism for conveying the prediction to the hardware. To this end, modern processors include two versions of many if not all branch instructions, the difference being a prediction on whether the branch will be taken or not. The compiler will choose the version of a given branch instruction that best meets its need.

3. **Branch prediction with branch target buffer:** This builds on branch prediction that we mentioned earlier, It uses a hardware device called **Branch Target Buffer (BTB)**[8] to improve the prediction of branches. Every entry of this table contains three fields as shown in Figure 5.18.

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

Figure 5.18: An entry in the Branch Target Buffer

This table records the history of the branches encountered for a particular program during execution. The table may have some small number of entries (say 100). Every time a branch instruction is encountered, the table is looked up by the hardware. Let us assume the PC value of this branch instruction is not present in the

[8] We already introduced the idea of caches earlier in this Chapter (see Section 5.11). In Chapter 9, we discuss caches in detail. BTB is essentially a cache of branch target addresses matched to addresses of branch instructions. The hardware needed to realize a BTB is similar to what would be found in a cache.

table (this is the first encounter). In that case, once the branch outcome is determined, a table entry with the address of this branch instruction, the target address of the branch, and the direction of the branch (taken/not taken) is created. Next time the same branch instruction is encountered this history information helps in predicting the outcome of the branch. The BTB is looked up in the IF stage. Thus, if the lookup is successful (i.e., the table contains the address of the branch instruction currently being fetched and sent down the pipeline), then the IF stage starts fetching the target of the branch immediately. In this case, there will be no bubbles in the pipeline due to the branch instruction. Of course, there could be mispredictions. Typically, such mispredictions occur due to a branch instruction that is executed in either the first or the last iteration of a loop. The flush line in the datapath takes care of such mispredictions. We can make the history mechanism more robust by providing more than one bit of history.

5.13.3.2 Summary of dealing with branches in a pipelined processor

Implementation detail of an ISA is commonly referred to as the *microarchitecture* of the processor. Dealing with branches in the microarchitecture is a key to achieving high performance, especially with the increased depth of the pipelines in modern processors. We already mentioned that the frequency of branches in compiled code could be as high as 1 in every 3 or 4 instructions. Thus, with a pipeline that has 20 stages, the instructions in partial execution in the various stages of the pipe are unlikely to be sequential instructions. Therefore, taking early decisions on the possible outcome of branches so that the pipeline can be filled with useful instructions is a key to achieving high performance in deeply pipelined processors. We will discuss the state-of-the-art in pipelined processor design in a later section (see Section 5.15). Table 5.5 summarizes the various techniques we have discussed in this chapter for dealing with branches and processors that have used these techniques.

Name	Pros	Cons	Use cases
Stall the pipeline	Simple strategy, no hardware needed for flushing instructions	Loss of performance	Early pipelined machines such as IBM 360 series
Branch Prediction (branch not taken)	Results in good performance with small additional hardware since the instruction is anyhow being fetched from the sequential path already in IF stage	Needs ability to flush instructions in partial execution in the pipeline	Most modern processors such as Intel Pentium, AMD Athlon, and PowerPC use this technique; typically they also employ sophisticated branch target buffers; MIPS R4000 uses a combination 1-delay slot plus a 2-cycle branch-not-taken prediction

Branch Prediction (branch taken)	Results in good performance but requires slightly more elaborate hardware design	Since the new PC value that points to the target of the branch is not available until the branch instruction is in EX stage, this technique requires more elaborate hardware assist to be practical	-
Delayed Branch	No need for any additional hardware for either stalling or flushing instructions; It involves the compiler by exposing the pipeline delay slots and takes its help to achieve good performance	With increase in depth of pipelines of modern processors, it becomes increasingly difficult to fill the delay slots by the compiler; limits microarchitecture evolution due to backward compatibility restrictions; it makes the compiler not just ISA-specific but implementation specific	Older RISC architectures such as MIPS, PA-RISC, SPARC

Table 5.5: Summary of Techniques for Handling Branches

5.13.4 Summary of Hazards

We discussed structural, data, and control hazards and the most basic mechanisms for dealing with them in the microarchitecture. We presented mostly hardware assists for detecting and overcoming such hazards. When hazards are detected and enforced by the hardware, it is often referred to in the literature as *hardware interlocks*. However, it is perfectly feasible to shift this burden to the software, namely, the compiler. The trick is to expose the details of the microarchitecture to the compiler such that the compiler writer can ensure that either (a) these hazards are eliminated in the code as part of program optimization, or (b) by inserting NOPs into the code explicitly to overcome such hazards. For example, with the 5-stage pipeline that we have been discussing, if the compiler always ensures that a register being written to is not used by at least the following three instructions, then there will be no RAW hazard. This can be done by either placing other useful instructions between the definition of a value and its subsequent use, or in the absence of such useful instructions, by placing explicit NOP instructions. The advantage of shifting the burden to the compiler is that the hardware is simplified, since it has neither to do hazard detection nor incorporate techniques such as data forwarding. Early versions of the MIPS architecture eliminated hardware interlocks and depended on the compiler to resolve all such hazards. However, the problem is it is not always possible to know such dependencies at compile time. The problem gets worse with deeper pipelines. Therefore, all modern processors use hardware interlocks to eliminate hazards. However, in the spirit of the partnership between hardware and

software, the chip vendors publish the details of the microarchitecture to aid the compiler writer in using such details in writing efficient optimizers.
Table 5.6 summarizes the LC-2200 instructions that can cause potential stalls in the pipeline, and the extent to which such stalls can be overcome with hardware solutions.

Instruction	Type of Hazard	Potential Stalls	With Data forwarding	With branch prediction (branch not taken)
ADD, NAND	Data	0, 1, 2, or 3	0	Not Applicable
LW	Data	0, 1, 2, or 3	0 or 1	Not Applicable
BEQ	Control	1 or 2	Not Applicable	0 (success) or 2 (mispredict)

Table 5.6: Summary of Hazards in LC-2200

5.14 Dealing with program discontinuities in a pipelined processor

Earlier when we discussed interrupts, we mentioned that we wait for the FSM to be in a clean state to entertain interrupts. In a non-pipelined processor, instruction completion offers such a clean state. In a pipelined processor since several instructions are in flight (i.e., under partial execution) at any point of time, it is difficult to define such a clean state. This complicates dealing with interrupts and other sources of program discontinuities such as exceptions and traps.

There is one of two possibilities for dealing with interrupts. One possibility is as soon as an external interrupt arrives, the processor can:
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 as we discussed in Chapter 4 and do the needful (See Example 17).

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. The other possibility is to **flush** the pipeline. As soon as an interrupt arrives, send a signal to all the stages to abandon their respective instructions. This allows the process to switch to the interrupt state immediately. Of course, in this case the 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. There are subtle issues with implementing such a scheme to ensure that no permanent program state has been changed (e.g., register values) by any instruction that was in partial execution in any of the pipeline stages.

In reality, processors do not use either of these two extremes. Interrupts are caught by a specific stage of the pipe. The program execution will restart at the instruction at that stage. The stages ahead are allowed to complete while the preceding stages are flushed. In each of these cases, one interesting question that comes up is what should happen implicitly in hardware to support interrupts in a pipelined processor. The pipeline registers are part of the internal state of the processor. Each of the above approaches gets the processor to a clean state, so we have to worry only about saving a single value of the PC for the correct resumption of the interrupted program. The value of the PC to be saved depends on the approach chosen.

If we want to be simplistic and drain the pipe before going to the INT macro state (see Chapter 4), then it is sufficient to freeze the PC value pointing to the next instruction in program order in the IF stage. Upon an interrupt, the hardware communicates this PC value to the INT macro state for saving.

Example 17:

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. (An English description is sufficient.)

Answer:

1. Allow instructions already in the pipeline (useful instructions) to complete execution
2. Stop fetching new instructions
3. Start sending NOPs from the fetch stage into the pipeline
4. Once all the useful instructions have completed execution, record the address where the program needs to be resumed in the PC (this will be memory address of last completed useful instruction + 1);
5. The next three steps are interrupt state actions that we covered in Chapter 4.
6. Go to INT state; sent INTA; receive vector; disable interrupts
7. Save current mode in system stack; change mode to kernel mode;
8. Store PC in $k0; Retrieve handler address using vector; Load PC; resume pipeline execution

---

Flushing the instructions (either fully or partially) would require carrying the PC value of each instruction in the pipeline register since we do not know when an external interrupt may occur[9]. Actually, carrying the PC value with each instruction is a necessity in a pipelined processor since any instruction may cause an exception/trap. We need the PC value of the trapping instruction to enable program resumption at the point of trap/exception. Upon an interrupt, the hardware communicates the PC value of the first incomplete instruction (i.e., the instruction from which the program has to be resumed) to the INT macro state for saving. For example, if the interrupt is caught by the EX stage of

[9] Recall that in the discussion of the pipeline registers earlier, we mentioned that only the passage of the BEQ instruction needs the PC value to be carried with it.

the pipe, then the current instructions in the MEM and WB stages are allowed to complete, and the PC value corresponding to the instruction in the EX stage of the pipeline is the point of program resumption.

Example 18:

A pipelined implementation of LC-2200, allows interrupts to be caught in the EX stage of the 5-stage pipeline draining the instructions preceding it, and flushing the ones after it. Assume that there is register forwarding to address RAW hazards.

Consider the following program:

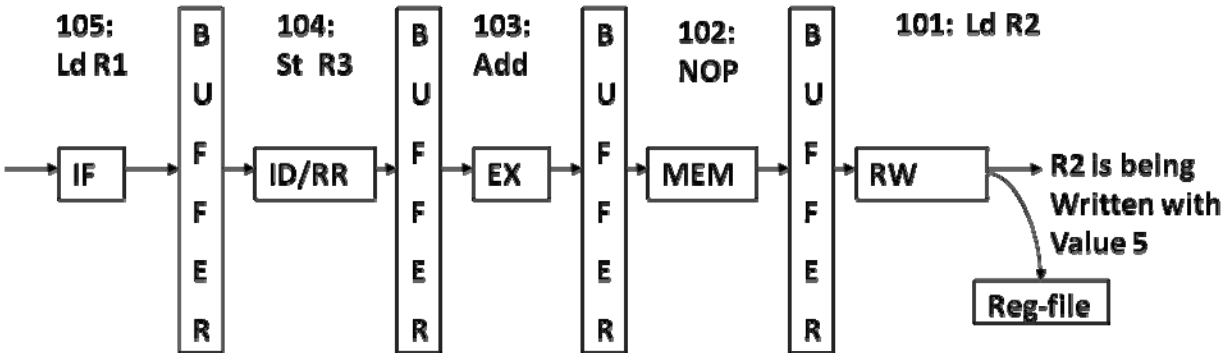
```
Address
100    Ld     R1, MEM[2000];      /* assume MEM[2000] contains value 2 */
101    Ld     R2, MEM[2002];      /* assume MEM[2002] contains value 5 */
102    NOP                ;      /* NOP for Ld R2 */
103    Add    R3, R1, R2;         /* addition R3 <- R1+R2 */
104    St     R3, MEM[2004];      /* Store R3 into MEM[2004]
105    Ld     R1, MEM[2006];      /* assume MEM[2006] contains value 3 */
```

An interrupt occurs when the NOP (at address 102) is in the MEM stage of the pipeline. Answer the following.

(a) What are the values in R1, R2, R3, MEM[2000], MEM[2002], MEM[2004], and MEM[2006], when the INT state is entered?
(b) What is the PC value passed to the INT state?

Answer:

When the interrupt occurs the state of the pipeline is as shown below:



Therefore, instructions ahead of the EX stage (namely, MEM and RW) will complete, and the ones in the ID/RR and IF stages will be flushed. The program will restart execution at address `103: Add R3, R1, R2`

(a)

R1 = 2; R2 = 5;
R3 = unknown (since Add instruction has been aborted due to the interrupt)
MEM[2000] = 2; MEM[2002] = 5;
MEM[2004] = unknown; (since Add instruction has been aborted due to the interrupt)
MEM[2006] = 3

(b)

The PC value passed to the INT state is the address of the Add instruction, namely, 103, which is where the program will resume after the interrupt service.

The above discussion is with respect to the program discontinuity being an external interrupt. With traps and exceptions, we have no choice but to complete (i.e., drain) the preceding instructions, flush the succeeding instructions, and go to the INT state passing the PC value of the instruction that caused the trap/exception.

5.15 Advanced topics in processor design

Pipelined processor design had its beginnings in the era of high-performance mainframes and vector processors of the 60's and 70's. Many of the concepts invented in that era are still relevant in modern processor design. In this section, we will review some advanced concepts in processor design including the state-of-the-art in pipelined processor design.

5.15.1 Instruction Level Parallelism

The beauty of pipelined processing is that it does not conceptually change the *sequential programming model*. That is, as far as the programmer is concerned, there is no perceived difference between the simple implementation of the processor presented in Chapter 3 and the pipelined implementation in this chapter. The instructions of the program *appear to execute* in exactly the same order as written by the programmer. The order in which instructions appear in the original program is called *program order*. Pipelined processor shrinks the execution time of the program 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. *Instruction Level Parallelism (ILP)* is the name given to this potential overlap that exists among instructions. ILP is a property of the program, and is a type of parallelism that is often referred to as *implicit parallelism*, since the original program is sequential. In Chapter 12, we will discuss techniques for developing explicitly parallel programs, and the architectural and operating systems support for the same. Pipelined processor exploits ILP to achieve performance gains for sequential programs. The reader can immediately see that ILP is limited by hazards. Particularly, control hazards are the bane of ILP exploitation. *Basic block* is a term used to define the string of instructions in a program separated by branches (see Figure 5.19). With reference to Figure 5.19, the amount of ILP available for the first basic block is 4, and that for the second basic block is 3. The actual

parallelism exploitable by the pipelined processor is limited further due to other kinds of hazards (data and structural) that we discussed in this chapter.

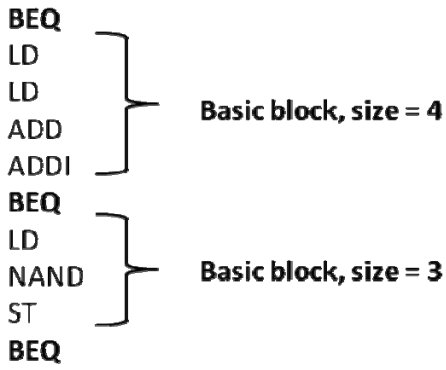


Figure 5.19: Basic Blocks and ILP

We have only scratched the surface of the deep technical issues in processor design and implementation. For example, we presented very basic mechanisms for avoiding conflicts in shared resource requirements (such as register file and ALU) across stages of the pipeline, so that the available ILP becomes fully exploitable by the processor implementation. Further, we also discussed simple mechanisms for overcoming control hazards so that ILP across multiple basic blocks becomes available for exploitation by the processor. Multiple-issue processors that have become industry-standard pose a number of challenges to the architect.

5.15.2 Deeper pipelines

The depth of pipelines in modern processors is much more than 5. For example, Intel Pentium 4 has in excess of 20 pipeline stages. However, since the frequency of branches in compiled code can be quite high, the size of typical basic blocks may be quite small (in the range of 3 to 7). Therefore, it is imperative to invent clever techniques to exploit ILP across basic blocks to make pipelined implementation worthwhile. The field of microarchitecture, the myriad ways to improve the performance of the processor, is both fascinating and ever evolving. In Chapter 3, we mentioned *multiple issue* processors, with Superscalar and VLIW being specific instances of such processors. *Instruction issue* is the act of sending an instruction for execution through the processor pipeline. In the simple 5-stage pipeline we have considered thus far, exactly one instruction is issued in every clock cycle. As the name suggests, multiple issue processors *issue* multiple instructions in the same clock cycle. As a first order of approximation, let us assume that the hardware and/or the compiler would have ensured that the set of instructions being issued in the same clock cycle do not have any of the hazards we discussed earlier, so that they can execute independently. Correspondingly, the processors have *multiple decode* units, and *multiple functional units* (e.g., integer arithmetic unit, floating point arithmetic unit, load/store unit, etc.) to cater to the different needs of the instructions being issued in the same clock cycle (see Figure 5.20). The fetch unit would bring in multiple

instructions from memory to take advantage of the multiple decode units. A decode unit should be able to dispatch the decoded instruction to any of the functional units.

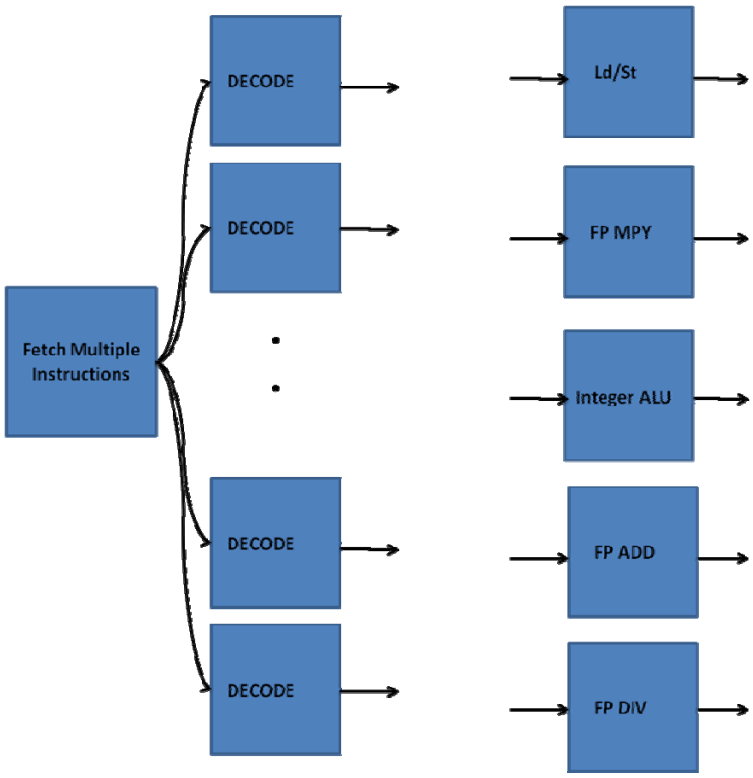


Figure 5.20: Multiple Issue processor pipeline

The effective throughput of such processors approaches the product of the depth of the pipelining and the degree of superscalarity of the processor. In other words, modern processors not only have deep pipelines but also have several such pipelines executing in parallel. Deep pipelining and superscalarity introduce a number of interesting challenges, to both the architect and the developers of system software (particularly compiler writers).

You may be wondering about the necessity for deep pipelining. There are several reasons:

- **Relative increase in time to access storage:** In Chapter 3, we introduced the components of delays in a datapath. As feature sizes of transistors on the chip keep shrinking, wire delays (i.e., the cost of ferrying bits among the datapath elements) rather than logic operations become the limiting factor for deciding the clock cycle time. However, the time to pull values out of registers and caches (which we discuss in detail in Chapter 9) is still much greater than either the logic delays or wire delays. Thus, 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:** In Chapter 3, we introduced the concept of microprogrammed implementation of instructions. While this technique has its pitfalls, pipelined processors may still use it for implementing some selected complex instructions in the ISA. 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. Typically, the microarchitecture includes specialized functional units for performing floating-point add, multiply, and division that are distinct from the integer ALU. The EX unit of the simple 5-stage pipeline gets replaced by this collection of functional units (see Figure 5.20). There may be an additional stage to help schedule these multiple functional units.
- **Dedicated floating-point pipelines:** Floating-point instructions require more time to execute compared to their integer counter-parts. With a simple 5-stage pipeline, the slowest functional unit will dictate the clock cycle time. Therefore, it is natural to pipeline the functional units themselves so that we end with a structure as shown in Figure 5.21, with different pipeline depth for different functional units. This differential pipelining facilitates supporting multiple outstanding long latency operations (such as floating-point ADD) without stalling the pipeline due to structural hazards.

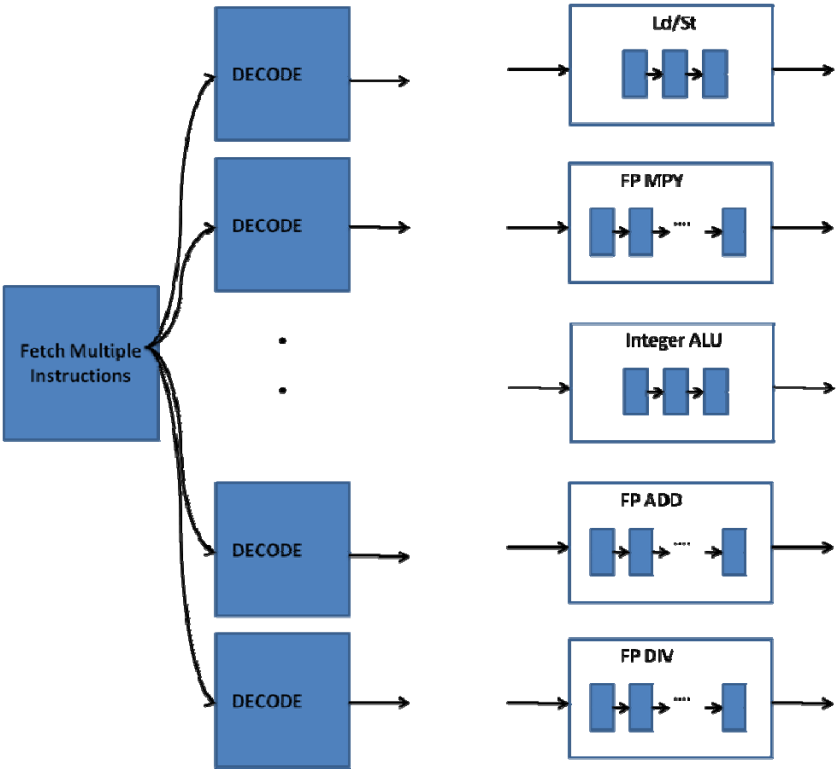


Figure 5.21: Different depths of pipelining for different functional units

- **Out of order execution and Re-Order Buffer:** Recall that the pipelined processor should preserve the appearance of sequential execution of the program (i.e., the program order) despite the exploitation of ILP. With the potential for different instructions taking different paths through the pipeline, there is a complication in maintaining this appearance of sequentiality. For this reason, 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. Essentially, ROB ensures that the effect of an instruction completion (e.g., writing to an architecture-visible register, reading from or writing to memory, etc.) is not *committed* until all its predecessor instructions in program order have been committed. It does this by recording the program order of instructions at issue time and buffering addresses and data that need to be transmitted to the architecture-visible registers and memory. Reading from a register in the presence of a ROB, respects the RAW dependency as already discussed in Section 5.13.2.2.
- **Register renaming:** To overcome data hazards, 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*[10]. Essentially, register renaming is a one-level indirection between the architectural register named in an instruction and the actual physical register to be used by this instruction. We will discuss this in more detail later in this section after introducing Tomasulo algorithm.
- **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. The idea is to execute instructions in different basic blocks without waiting for the resolution of the branches and have mechanisms to *undo* the effects of the instructions that were executed incorrectly due to the speculation, once the branches are resolved. Both ROB and/or register renaming (in hardware) help hardware-based speculation since the information in the physical registers or ROB used in the speculation can be discarded later when it is determined that the effects of a speculated instruction should not be committed.

[10] Note that register renaming could be done by the compiler as well upon detection of data hazards as part of program optimization, in which case such an extra stage would not be needed in the pipeline.

5.15.3 Revisiting program discontinuities in the presence of out-of-order processing

In Section 5.14, we discussed simple mechanisms for dealing with interrupts in a pipelined processor. Let's see the ramifications of handling interrupts in the presence of out-of-order processing. Several early processors such as CDC 6600 and IBM 360/91 used out-of-order execution of instructions to overcome pipeline stalls due to data hazards and structural hazards. The basic idea is to issue instructions in order, but let the instructions start their executions as soon as their source operands are available. This out-of-order execution coupled with the fact that different instructions incur different execution time latencies leads to instructions potentially completing execution out of order and retiring from the pipeline. That is, they not only complete execution but also update processor state (architecture-visible registers and/or memory). Is this a problem? It would appear that there should not be since these early pipelined processors did respect the program order at issue time, did honor the data dependencies among instructions, and never executed any instructions speculatively. External interrupts do not pose a problem with out-of-order processing, since we could adopt a very simple solution, namely, stop issuing new instructions and allow all the issued instructions to complete before taking the interrupt.

Exceptions and traps, however, pose a problem since some instructions that are subsequent to the one causing the exception could have completed their execution. This situation is defined as *imprecise exception*, to signify that the processor state, at the time the exception happens, is not the same as would have happened in a purely sequential execution of the program. These early pipelined processors restored precise exception state either by software (i.e., in the exception handling routines), or by hardware techniques for early detection of exceptions in long latency operations (such as floating-point instructions).

As we saw in the previous section, modern processors retire the instructions in program order despite the out-of-order execution. This automatically eliminates the possibility of imprecise exception. Potential exceptions are buffered in the re-order buffer and will manifest in strictly program order.

Detailed discussion of interrupts in a pipelined processor is outside the scope of this book. The interested reader may want to refer to advanced textbooks on computer architecture[11].

5.15.4 Managing shared resources

With multiple functional units, managing the shared resources (such as register files) becomes even more challenging. One technique, popularized by CDC 6600, is *scoreboard*, a mechanism for dealing with the different kinds of data hazards we discussed earlier in this chapter. The basic idea is to have a central facility, namely a

[11] Hennessy and Patterson, "Computer Architecture: A Quantitative Approach," Morgan Kaufman publishers.

scoreboard that records the resources in use by an instruction at the time it enters the pipeline. A decision to either progress through the pipeline or stall an instruction depends on the current resource needs of that instruction. For example, if there is a RAW hazard then the second instruction that needs the register value is stalled until the scoreboard indicates that the value is available (generated by a preceding instruction). Thus, the scoreboard keeps track of the resource needs of all the instructions in flight through the pipeline.

Robert Tomasulo of IBM came up with a very clever algorithm (named after him), which is a distributed solution to the same resource sharing and allocation problem in a pipelined processor. This solution was first used in IBM 360/91, one of the earliest computers (in the 60's) to incorporate principles of pipelining in its implementation. The basic idea is to associate storage (in the form of local registers) with each functional unit. At the time of instruction issue, the needed register values are transferred to these local registers[12] thus avoiding the WAR hazard. If the register values are unavailable (due to RAW hazard), then the local registers remember the unit from which they should expect to get the value. Upon completing an instruction, a functional unit sends the new register value on a *common data bus (CDB)* to the register file. Other functional units (there could be more than one) that are waiting for this value grab it from the bus and start executing their respective instructions. Since the register file also works like any other functional unit, it remembers the peer unit that is generating a value for a given register thus avoiding the WAW hazard. In this manner, this distributed solution avoids all the potential data hazards that we have discussed in this chapter.

The core of the idea in Tomasulo algorithm is the use of local registers that act as surrogates to the architecture-visible registers. Modern processors use this idea by consolidating all the distributed storage used in Tomasulo algorithm into one large physical register file on the chip. Register renaming technique that we mentioned earlier, creates a dynamic mapping between an architecture-visible register and a physical register at the time of instruction issue. For example, if register R1 is the architecture-visible source register for a store instruction, and the actual physical register assigned is say, P12, then the value of R1 is transferred into P12 in the register renaming stage. Thus, the register renaming stage of the pipeline is responsible for dynamic allocation of physical registers to the instructions. Essentially, register renaming removes the WAR and WAW data hazards. We already discussed in Section 5.13.2.2 how data forwarding solves the RAW hazard in a pipelined processor. Thus, register renaming combined with data forwarding addresses all the data hazards in modern processors. The pipeline stage that is responsible for register renaming keeps track of which physical registers are in use at any point of time and when they get freed (not unlike the scoreboarding technique of CDC 6600) upon the retirement of an instruction.

The role of the re-order buffer (ROB) is to ensure that instructions retire in program order. The role of register renaming is to remove data hazards as well as to support hardware-based speculative execution. Some processors eliminate the ROB altogether

[12] These local registers in Tomasulo algorithm perform the same function as the large physical register file in modern processors to support register renaming.

and incorporate its primary functionality (namely, retiring instructions in program order) into the register renaming mechanism itself.

Let us revisit WAW hazard. We saw that the techniques (such as scoreboarding and Tomasulo algorithm) used in the early pipelined machines, and the techniques (such as register renaming and ROB) now being used in modern processors eliminates WAW, despite the out-of-order execution of instructions. Does speculative execution lead to WAW hazard? The answer is no, since, despite the speculation, the instructions are retired in program order. Any incorrect write to a register due to a mispredicted branch would have been removed from the ROB without being committed to any architecture-visible register.

Despite multiple issue processors, speculation, and out-of-order processing, the original question that we raised in Section 5.13.2.4, still remains, namely, why would a compiler generate code with a WAW hazard. The answer lies in the fact that a compiler may have generated the first write to fill a delay slot (if the microarchitecture was using delayed branches) and the program took an unexpected branch wherein this first write was irrelevant. However, the out of order nature of the pipeline could mean that the second useful write may finish before the irrelevant first write. More generally, WAW hazard may manifest due to unexpected code sequences. Another example, is due to the interaction between the currently executing program and a trap handler. Let us say, an instruction, which would have written to a particular register (first write) if the execution was normal, traps for some reason. As part of handling the trap, the handler writes to the same register (second write in program order) and resumes the original program at the same instruction. The instruction continues and completes writing to the register (i.e., the first write in program order, which is now an irrelevant one due to the trap handling). If the writes do not occur in program order then this first write would overwrite the second write by the trap handler. It is the hardware's responsibility to detect such hazards and eliminate them.

5.15.5 Power Consumption

Another interesting dimension in processor design is worrying about the power dissipation. Even as it is, current GHz microprocessors dissipate a lot of power leading to significant engineering challenges in keeping such systems cool. As the processing power continues to increase, the energy consumption does too. The challenge for the architect is the design of techniques to keep the power consumption low while aspiring for higher performance.

The ability to pack more transistors on a single piece of silicon is a blessing while posing significant challenges to the architect. First of all, with the increase in density of transistors on the chip, all the delays (recall the width of the clock pulse discussion from Chapter 3) have shrunk. This includes the time to perform logic operations, wire delays, and access times to registers. This poses a challenge to the architect in the following way. In principle, the chip can be clocked at a higher rate since the delays have gone down. However, cranking up the clock increases power consumption. Figure 5.22 shows

the growth in power consumption with increasing clock rates for several popular processors. You can see the high correlation between clock cycle time and the power consumption. A 3.6 GHz Pentium 4 'E' 560 processor consumes 98 watts of power. In the chart, you will see some processors with lower clock rating incurring higher power consumption (e.g. A64-4000). The reason for this is due to the fact the power consumption also depends on other on-chip resources such as caches, the design of which we will see in a later chapter.

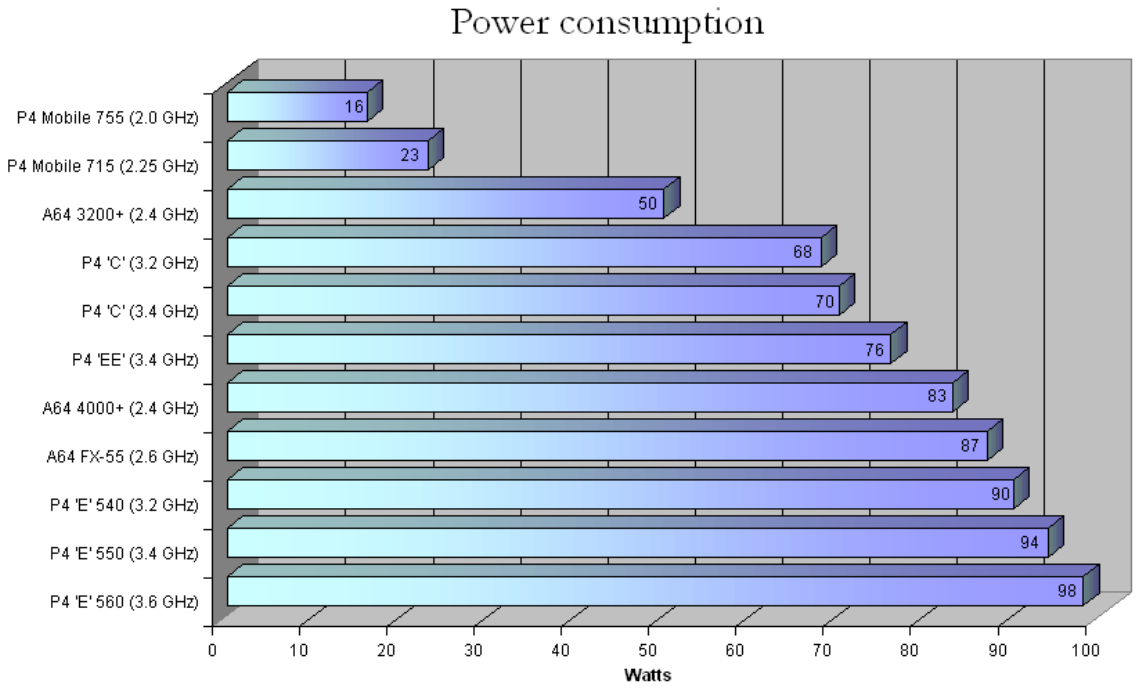


Figure 5.22: CPU Power Consumption[13]

5.15.6 Multi-core Processor Design

The reality is, as technology keeps improving, if the processor is clocked at the rate that is possible for that technology, pretty soon we will end up with a laptop whose power consumption equals that of a nuclear reactor! Of course, the solution is not to stop making chips with higher density and higher clock rates. Architects are turning to another avenue to increase performance of the processor, without increasing the clock rate, namely, *multiprocessing*. This new technology has already hit the market with the name *multi-core* (Intel Core 2 Duo, AMD Opteron quad-core, etc.). Basically, each chip has multiple processors on it and by dividing the processing to be done on to these multiple processors the system throughput, i.e., performance is increased. The architecture and hardware details underlying multi-core processors (or *chip multiprocessors* as they are often referred to) are beyond the scope of this textbook.

[13] Source: **http://en.wikipedia.org/wiki/CPU_power_dissipation**. Please note that we are not attesting to the veracity of these exact numbers but the charts give a visual ballpark of CPU power dissipation.

However, multi-core technology builds on the basic principles of parallel processing that have been around as long as computer science has been around. We will discuss the hardware and software issues surrounding multiprocessors and parallel programming in much more detail in a later chapter.

5.15.7 Intel Core[14] Microarchitecture: An example pipeline

It is instructive to understand the pipeline structure of modern processors. Intel Pentium 4 Willamette and Galatin use a 20 stage pipeline, while Intel Pentium Prescott and Irwindale use a 31 stage pipeline. One of the chief differentiators between Intel and AMD product lines (though both support the same x86 ISA) is the depth of the pipeline. While Intel has taken the approach of supporting deeper pipelines for larger instruction throughput, AMD has taken the approach of a relatively shallower (14-stage) pipeline.

A family of Intel processors including Intel Core 2 Duo, Intel Core 2 Quad, Intel Xeon, etc., uses a common "core" microarchitecture shown in Figure 5.23. This is a much-simplified diagram to show the basic functionalities of a modern processor pipeline. The interested reader is referred to the Intel architecture manuals available freely on the Intel web site[15]. At a high-level the microarchitecture has a *front-end*, an *execution core*, and a *back-end*. The role of the front-end is to fetch instruction streams *in-order* from memory, with four decoders to supply the decoded instructions (also referred to as *microops*) to the execution core. The front-end is comprised of the fetch and pre-decode unit, the instruction queue, the decoders, and microcode ROM. The middle section of the pipeline is an *out-of-order* execution core that can issue up to *six* microops every clock cycle as soon as the sources for the microops are ready (i.e., there are no RAW hazards) and the corresponding execution units are available (i.e., there are no structural hazards). This middle section incorporates register renaming, re-order buffer, reservation station, and an instruction scheduler. The back-end is responsible for retiring the executed instructions in *program order*, and for updating the programmer visible architecture-registers. The functionalities of the different functional units that feature in the pipeline microarchitecture of Intel Core are as follows:

- **Instruction Fetch and PreDecode:** This unit is responsible for two things: fetching the instructions that are most likely to be executed and pre-decoding the instructions recognizing variable length instructions (since x86 architecture supports it). Pre-decode helps the instruction fetcher to recognize a branch instruction way before the outcome of the branch will be decided. An elaborate branch prediction unit (BPU) is part of this stage that helps fetching the most likely instruction stream. BPU has dedicated hardware for predicting the outcome of different types of branch instructions (conditional, direct, indirect, and call and return). The pre-decode unit can write up to six instructions every clock cycle into the instruction queue.
- **Instruction Queue:** The instruction queue takes the place of an instruction register (IR) in a simple 5-stage pipeline. Since it houses many more instructions (has a depth of 18 instructions), the instruction queue can hold a snippet of the original program (e.g., a small loop) to accelerate the execution in the pipelined processor. Further, it

[14] Intel Core is a registered trademark of Intel Corporation.
[15] Intel website: http://www.intel.com/products/processor/manuals/index.htm

can also help in saving power since the rest of the front-end (namely, the instruction fetch unit) can be shut down during the execution of the loop.

- **Decode and Microcode ROM:** This unit contains four decoders, and hence can decode up to 4 instructions in the instruction queue in every clock cycle. Depending on the instruction, the decode unit may expand it into several microops with the help of the microcode ROM. The microcode ROM can emit three microops every cycle. The microcode ROM thus facilitates implementing complex instructions without slowing down the pipeline for the simple instructions. The decoders also support *macro-fusion*, fusing together two instructions into a single microop.
- **Register Renaming/Allocation:** This unit is responsible for allocating physical registers to the architectural registers named in the microop. It keeps the mapping thus created between the architectural registers and the actual physical registers in the microarchitecture. It supports hardware-based speculation and removes WAR and WAW hazards.
- **Re-Order Buffer:** This unit has 96 entries and is responsible for registering the original program order of the microops for later reconciliation. It holds the microops in various stages of execution. There can be up to 96 microops in flight (i.e., in various stages of execution) given the size of the ROB.
- **Scheduler:** This unit is responsible for scheduling the microops on the functional units. It includes a **reservation station** that queues all the microops until their respective sources are ready and the corresponding execution unit is available. It can schedule up to six microops every clock cycle subject to their readiness for execution.
- **Functional Units:** As the name suggests these are the units that carry out the microops. They include execution units with single-cycle latency (such as integer add), pipelined execution units for frequently used longer latency microops, pipelined floating-point units, and memory load/store units.
- **Retirement Unit:** This represents the back-end of the microarchitecture and uses the re-order buffer to retire the microops in program order. It also updates the architectural states in program order, and manages the ordering of exceptions and traps that may arise during instruction execution. It also communicates with the reservation station to indicate the availability of sources that microops may be waiting on.

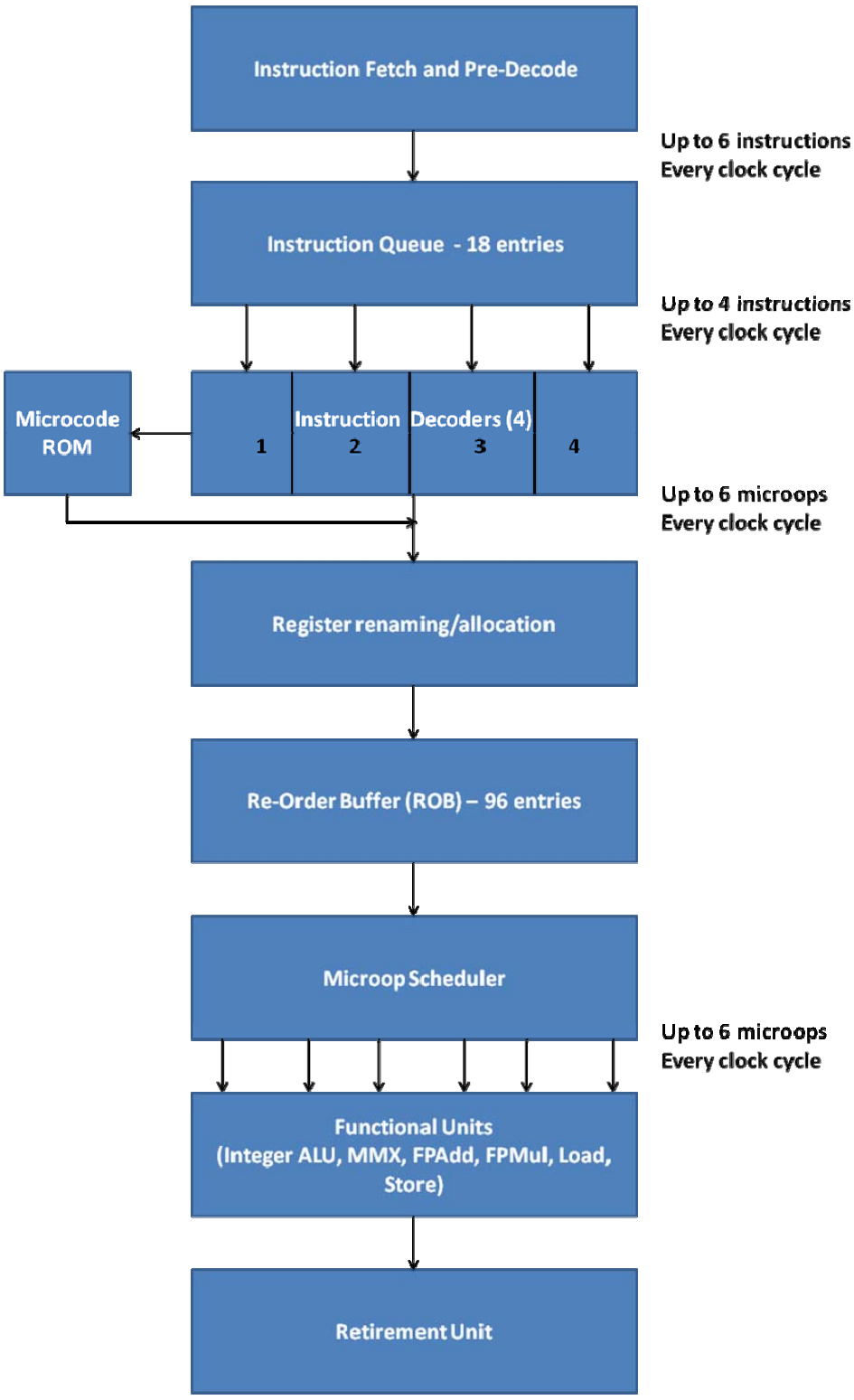


Figure 5.23: Intel Core Microarchitecture Pipeline Functionality

5.16 Historical Perspective

Most of us have a tendency to take things for granted depending on where we are (in space and time), be it running water in faucets 24x7, high speed cars, or high-performance computers in the palm of our hand. Of course, many of us realize quickly we have come a long way in a fairly short period of time in computer technology due to the amazing pace of breakthrough development in chip integration.

It is instructive to look back at the road we have traversed. Not so long ago, pipelined processor design was reserved for high-end computers of the day. Researchers at IBM pioneered fundamental work in pipelined processor design in the 60's, which resulted in the design of systems such as the IBM 360 series and later IBM 370 series of high-end computer systems. Such systems dubbed *mainframes*, perhaps due to their packaging in large metal chassis, were targeted mainly towards business applications. Gene Amdahl, who was the chief architect of the IBM 360 series and whose name is immortalized in Amdahl's law, discovered the principles of pipelining in his PhD dissertation at UW-Madison in 1952 in the WISC computer[16]. Seymour Cray, a pioneer in high-performance computers, founded Cray Research, which ushered in the age of *vector supercomputers* with the Cray series starting with Cray-1. Prior to founding Cray Research, Seymour Cray was the chief architect at Control Data Corporation, which was the leader in high-performance computing in the 60's with offerings such as the CDC 6600 (widely considered as the first commercial supercomputer) and shortly thereafter, the CDC 7600.

Simultaneous with the development of high-end computers, there was much interest in the development of *minicomputers*, DEC's PDP series leading the way with PDP-8, followed by PDP-11, and later on the VAX series. Such computers were initially targeted towards the scientific and engineering community. Low cost was the primary focus as opposed to high performance, and hence these processors were designed without employing pipelining techniques.

As we observed in Chapter 3, the advent of the killer micros in 80's coupled with pioneering research in compiler technology and RISC architectures, paved the way for instruction pipelines to become the implementation norm for processor design except for very low-end embedded processors. Today, even game consoles in the hands of toddlers use processors that employ principles of pipelining.

Just as a point of clarification of terminologies, supercomputers were targeted towards solving computationally challenging problems that arise in science and engineering (they were called *grand challenge problems*, a research program started by DARPA[17] to stimulate breakthrough computing technologies), while mainframes were targeted towards technical applications (business, finance, etc.). These days, it is normal to call such high-end computers as *servers* that are comprised of a collection of computers interconnected by high-speed network (also known as *clusters*). Such servers cater to

[16] Source: **http://en.wikipedia.org/wiki/Wisconsin_Integrally_Synchronized_Computer**
[17] DARPA stands for a Federal Government entity called *Defense Advanced Research Projects Agency*

both scientific applications (e.g., IBM's BlueGene massively parallel architecture), as well as technical applications (e.g., IBM z series). The processor building block used in such servers are quite similar and adhere to the principles of pipelining that we discussed in this chapter.

5.17 Review Questions

1. True or false: For a given workload and a given instruction-set architecture, reducing the CPI (clocks per instruction) of all the instructions will always improve the performance of the processor.

2. An architecture has three types of instructions that have the following CPI:

Type	CPI
A	2
B	5
C	3

   An architect determines that he can reduce the CPI for B by some clever architectural trick, with no change to the CPIs of the other two instruction types. However, she determines that this change will increase the clock cycle time by 15%. What is the maximum permissible CPI of B (round it up to the nearest integer) that will make this change still worthwhile? Assume that all the workloads that execute on this processor use 40% of A, 10% of B, and 50% of C types of instructions.

3. What would be the execution time for a program containing 2,000,000 instructions if the processor clock was running at 8 MHz and each instruction takes 4 clock cycles?

4. A smart architect re-implements a given instruction-set architecture, halving the CPI for 50% of the instructions, while increasing the clock cycle time of the processor by 10%. How much faster is the new implementation compared to the original? Assume that the usage of all instructions are equally likely in determining the execution time of any program for the purposes of this problem.

5. A certain change is being considered in the non-pipelined (multi-cycle) MIPS CPU regarding the implementation of the ALU instructions. This change will enable one to perform an arithmetic operation and write the result into the register file all in one clock cycle. However, doing so will increase the clock cycle time of the CPU. Specifically, the original CPU operates on a 500 MHz clock, but the new design will only execute on a 400 MHz clock.

   Will this change improve, or degrade performance? How many times faster (or slower) will the new design be compared to the original design? Assume instructions are executed with the following frequency:

Instruction	Frequency
LW	25%
SW	15%
ALU	45%
BEQ	10%
JMP	5%

Compute the CPI of both the original and the new CPU. Show your work in coming up with your answer.

Cycles per Instruction

Instruction	CPI
LW	5
SW	4
ALU	4
BEQ	3
JMP	3

6. Given the CPI of various instruction classes

Class	CPI
R-type	2
I-type	10
J-type	3
S-type	4

And instruction frequency as shown:

Class	Program 1	Program 2
R	3	10
I	3	1
J	5	2
S	2	3

Which code will execute faster and why?

7. What is the difference between static and dynamic instruction frequency?

8. Given

Instruction	CPI
Add	2
Shift	3
Others	2 (average for all instructions including Add and Shift)
Add/Shift	3

If the sequence ADD followed by SHIFT appears in 20% of the dynamic frequency of a program, what is the percentage improvement in the execution time of the program with all {ADD, SHIFT} replaced by the new instruction?

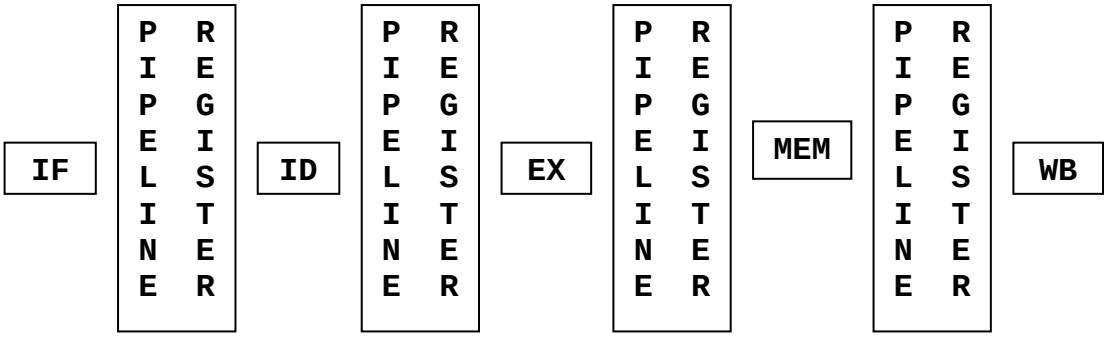
9. Compare and contrast structural, data and control hazards. How are the potential negative effects on pipeline performance mitigated?

10. How can a read after write (RAW) hazard be minimized or eliminated?

11. What is a branch target buffer and how is it used?

12. Why is a second ALU needed in the Execute stage of the pipeline?

13. In a processor with a five stage pipeline as discussed in the class and shown in the picture below (with buffers between the stages), explain the problem posed by a branch instruction. Present a solution.



14. Regardless of whether we use a conservative approach or branch prediction ("branch not taken"), explain why there is always a 2-cycle delay if the branch is taken (i.e., 2 NOPs injected into the pipeline) before normal execution can resume in the 5-stage pipeline used in Section 5.13.3.

15. With reference to Figure 5.6a, identify and explain the role of the datapath elements that deal with the BEQ instruction. Explain in detail what exactly happens cycle by cycle with respect to this datapath during the passage of a BEQ instruction. Assume a conservative approach to handling the control hazard. Your answer should include both the cases of branch taken and branch not taken.

16. A smart engineer decides to reduce the 2-cycle "branch taken" penalty in the 5-stage pipeline down to 1. Her idea is to directly use the branch target address computed in the EX cycle to fetch the instruction (note that the approach presented in Section 5.13.3 requires the target address to be saved in PC first).

    a) Show the modification to the datapath in Figure 5.6a to implement this idea [hint: you have to simultaneously feed the target address to the PC and the Instruction memory if the branch is taken].
    b) While this reduces the bubbles in the pipeline to 1 for branch taken, it may not be a good idea. Why? [hint: consider cycle time effects.]

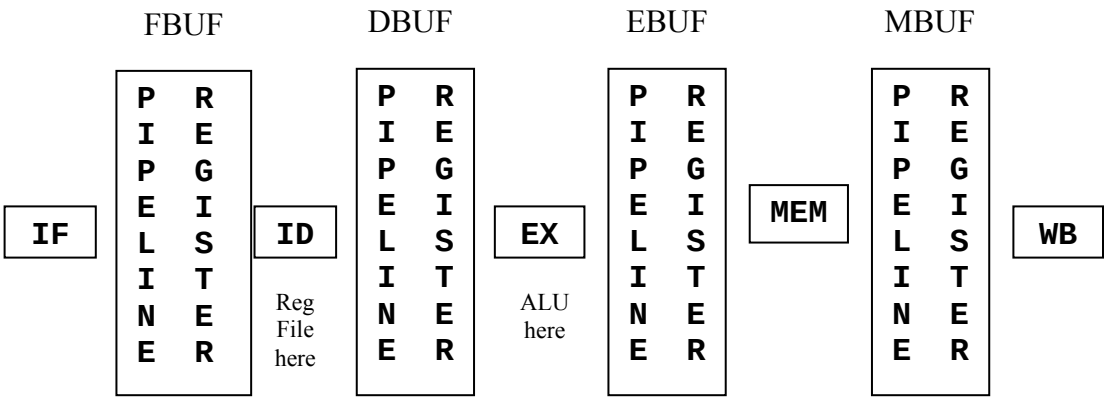
17. In a pipelined processor where each instruction could be broken up into 5 stages and where each stage takes 1 ns what is the best we could hope to do in terms of average time to execute 1,000,000,000 instructions?

18. Using the 5-stage pipeline shown in Figure 5.6c answer the following two questions:
    a. Show the actions (similar to Section 5.12.1) in each stage of the pipeline for BEQ instruction of LC-2200.
    b. Considering only the BEQ instruction, compute the sizes of the FBUF, DBUF, EBUF, and MBUF.

19. Repeat problem 7 for the SW instruction of LC-2200.

20. Repeat problem 7 for JALR instruction of LC-2200.

21. You are given the pipelined datapath for a processor as shown below.



A load-word instruction has the following 32-bit format:

OPCode	A Reg	B Reg	Offset
8 bits	4 bits	4 bits	16 bits

The semantic of this instruction is

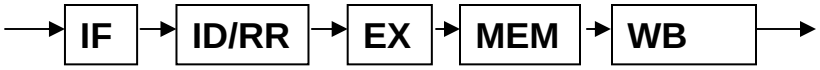
B <- Memory[A + offset]

Register B gets the data at the memory address given by the effective address generated by adding the contents of Register A to the 16-bit offset in the instruction. All data and addresses are 32-bit quantities.

Show what is needed in the pipeline registers (FBUF, DBUF, EBUF, MBUF) between the stages of the pipeline for the passage of this instruction. Clearly show the layout of each buffer. Show the width of each field in the buffer. You do not have to concern yourself about the format or the requirements of other instructions in the architecture for this problem.

22. Consider

I_1: R1 <- R2 + R3
I_2: R4 <- R1 + R5



If I2 is immediately following I1 in the pipeline with no forwarding, how many bubbles (i.e. NOPs) will result in the above execution? Explain your answer.

23. Consider the following program fragment:

Address	instruction
1000	add
1001	nand
1002	lw
1003	add
1004	nand
1005	add
1006	sw
1007	lw
1008	Add

Assume that there are no hazards in the above set of instructions. Currently the IF stage is about to fetch instruction at 1004.

(a) Show the state of the 5-stage pipeline
(b) Assuming we use the "drain" approach to dealing with interrupts, how many cycles will elapse before we enter the INT macro state? What is the value of PC that will be stored in the INT macro state into $k0?
(c) Assuming the "flush" approach to dealing with interrupts, how many cycles will elapse before we enter the INT macro state? What is the value of PC that will be stored in the INT macro state into $k0?