2 Out-of-order Pipelines

Processing an instruction is the most important task that modern processors perform. Hence, it is very important for us to understand the life cycle of an instruction from the time that it is fetched to the time that it is removed from the processor after completing its execution. As we shall see in this chapter, processing an instruction efficiently is a very complicated task, and we need very elaborate hardware structures to achieve this task.

We shall start our interesting journey in Section 2.1 where we shall understand the design of a simple conventional processor. The design of such a processor is typically taught in most first-level undergraduate courses. It is called an in-order processor because all the instructions are processed in program order. This processor is unfortunately very inefficient and as we shall see in Section 2.2, its performance can be significantly enhanced by processing more instructions simultaneously, and by executing them in an order that is different from their program order. Such kind of processors are known as out-of-order (OOO) processors (described in Section 2.3).

Designing an OOO processor that can process multiple instructions per cycle is a very difficult task. First, we need to handle branch instructions and try to predict their outcome (taken or not-taken) before they are executed. This will ensure that we are able to fill our processor with a large number of instructions. By buffering so many instructions within the processor we increase the likelihood of finding a set of mutually independent instructions whose operands are ready. These instructions can then be executed in parallel. We can thus increase the instruction execution throughput, and quickly execute large programs.

However, there are significant correctness issues that crop up. For example, we need to ensure that if an interrupt from a hardware device arrives, we can pause the execution of the program, handle the interrupt, and later on resume the execution of the program from exactly the same point. This requires us to maintain the state of the program in dedicated structures. This process of pause and recovery can be achieved seamlessly and without the explicit knowledge of the program. Finally, handling load and store operations is tricky because they are two-part operations: the first part computes the address, and the second part executes the memory instruction.

We shall not discuss all the techniques in this chapter. They will be discussed in the subsequent chapters. In this chapter, our aim is to introduce the issues, provide a broad overview of OOO pipelines, discuss the complexities, and motivate the reader to read the next few chapters. We do presume some background in instruction set architectures, assembly language programming, and the design of a simple processing unit (elaborated further in Section 2.1).

It is important for the reader to brush up her fundamentals on traditional in-order pipelines first. Then only she can understand the issues and nuances of more advanced designs. Hence, we discuss this topic first.

2.1 Overview of In-Order Pipelines

Let us start this section with a disclaimer. We expect the reader to be broadly familiar with basic computer architecture. The reader can refer to the entry level textbook on computer organization and architecture by the author [Sarangi, 2015] for a deeper treatment of these topics. This section is only meant to provide a very cursory overview of processor design and pipelining.

Specifically, here are the list of topics that the reader should be familiar with.

- Instruction Set Architecture
  - RISC and CISC instruction sets
  - Memory, branch, and ALU instructions
  - Loads and stores to memory
  - Instruction set encoding
- Basics of Processor Design
  - Fetch, decode, execute, memory access and write-back stages
  - Notion of selecting data from multiple sources using multiplexers.
  - Basics of pipelining and the notion of stalls
- Memory system
  - Notion of caches: instruction and data cache.
  - Idea of a memory hierarchy

This section is meant to give the readers a brief overview of pipelining, in specific, conventional in-order pipelines. Even though this section is meant to be self contained; however, it does assume that the reader is broadly familiar with instruction set architectures and at least the design of a non-pipelined processor.

2.1.1 Processor Design

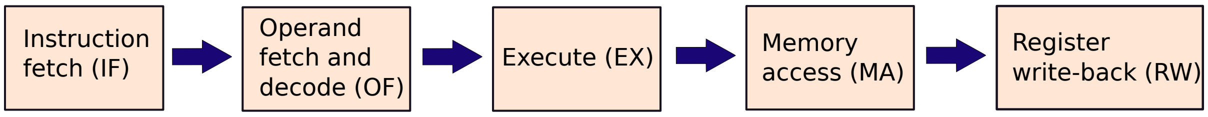


Figure 2.1: Five stages in a processor

Every basic processor consists of five stages as shown in Figure 2.1: instruction fetch (IF), decode and operand fetch (OF), execute (EX), memory access (MA), and register write-back (RW). We basically divide a processor's work into five logical stages such that the overlap between two stages in terms of functionality is negligible. Five is not a sacrosanct number. In alternative implementations of a processor, we can divide it into let's say three, four or six stages. The number of stages by itself is not important as long as the division of work between stages is done equitably, and with as little overlap as possible. We describe a representative example in this section.

Instruction Fetch Stage (IF)

In this stage, we fetch the contents of an instruction from instruction memory (often this is the instruction cache). Every instruction is uniquely indexed by the PC (program counter), which is a memory address.

In the fetch stage, we primarily do the following:

1. Given the PC, we fetch the instruction from memory. In most processors, this memory is a cache, known as the instruction cache (i-cache).

2. We compute the PC of the next instruction. This can either be the current PC plus the size of the instruction (4 or 8 bytes in most RISC processors) or if the instruction is a branch, the next PC can be the address of the branch's target.

3. If the instruction is a function return, then the PC is the value of the return address. The return address is typically stored in a register, or a location in memory.

The instruction fetch stage (IF) per se does not have a non-deterministic delay, and is meant to be simple in such in-order processors. Of course, complexities can get introduced if we assume a non-ideal memory. Recall that a cache contains a subset of all memory addresses. If we have a cache hit – we find an address in the cache – then the access is very fast. However, if we do not find the address in the cache leading to a cache miss, then it might take a long time to fetch the contents of the instruction. The IF stage needs to be stalled (kept idle) during that time.

Decode and Operand Fetch Stage (OF)

In this stage we decode the instruction. This means that we cleanly separate the different components of the instruction: opcode (operation code), ids of registers, and embedded constants.

Based on the instruction's opcode, we generate control signals to control the rest of the elements in the pipeline such that the instruction executes properly. Some explanation is due here. The part of the processor that generates all the control signals (single or multi-bit) is known as the *control logic*. The part of the processor's pipeline that creates, propagates, and uses the control signals is known as the *control path*. The rest of the processor that is involved with the storage of data, memory accesses, and computation is referred to as the *data path*.

After the ids of the registers are known, we access the register file (array of registers), and read the values of the source registers. There are some instructions such as the *return* instruction that often have an implicit operand – the return address register. This instruction is used to return from a function. Note that while calling a function we save the address of the next instruction. This is known as the *return address*. Most RISC processors save the return address in a dedicated return address register. Whereas, CISC processors store them on the stack. For a RISC ISA, we need to read the value of the return address from the register file while processing a return instruction in the operand fetch stage. Thus, the return address register is the implicit operand, whose value needs to be read.

Finally, some instructions embed constants in them. These constants can either be values that are to be used in arithmetic instructions or offsets from base registers (in load and store instructions), or offsets from the PC (in branches). In this stage, we need to extract these constants (known as *immediates*) and expand them to 32 or 64 bit values.

Thus, to summarize, in this stage we (1) generate all the control signals, (2) read values from the register file, and (3) and extract the constants embedded in the instruction. These values along with the contents of the instruction form the *instruction packet*. The instruction packet is forwarded to the subsequent stage.

Execute Stage (EX)

This stage has three major functions:

1. Perform the computation such as addition, subtraction, or multiplication.
2. If the instruction is a memory instruction (load or store), then compute the memory address. This is an addition operation. We add the offset to the contents of a register that contains the *base address* (base-offset addressing mode).
3. For a branch instruction compute the branch target. Most branches are relative. This means that the offset (embedded as a constant in the instruction) needs to be added to the PC to get the address of the branch target. This addition is performed in this stage. Subsequently, the branch target is sent to the fetch stage.

Memory Access Stage (MA)

Note that a memory access instruction is a two-stage operation. In the first stage, we need to compute the memory address by adding the offset to the contents of a base register. This was achieved in the EX (execute) stage. The next task is to access the data memory and perform the load or store.

1. A load operation has two arguments: memory address and destination register id. A dedicated circuit accesses the memory with the given address, retrieves 4 bytes or 8 bytes (depending upon the architecture), and stores it in the destination register.
2. A store operation does not have any destination registers. It has two arguments: memory address and source register id. A dedicated circuit takes the contents from the register, and stores it in memory at the given memory address.

For a memory instruction, we shall use the term *PC address* to indicate the address of the instruction in memory. To fetch the instruction, we need to set the program counter to the PC address. In comparison, the *memory address* is the address computed by the memory instruction in the EX stage. This instruction is sent to the memory system. In Section 7.2, we shall introduce the concept of address translation where the virtual address generated by the pipeline is translated into a physical address that is sent to the memory system. We shall use the term *memory address* to refer to either address – the connotation will be clear from the context.

Register Write-back Stage (RW)

This is the last stage of a typical 5-stage pipeline. In this stage, we write the final result – computed by the ALU or loaded from memory – to a register in the register file. If an instruction does not write to a register then nothing is done in this stage.

Additionally, we need to handle some special cases as well. Almost all ISAs have a *call* instruction, which jumps to the first line of a function, and simultaneously writes the return address to a register or a memory location. The latter depends on the instruction set. In instruction sets such as ARM® v7 the return address is saved in a dedicated register and in the Intel® x86 architecture the return address is saved in memory (top of the stack). Let us assume the former case where the return address is saved in a register. In this case, the return address will be written to the dedicated return address register in the RW stage.

Summary

A simple processor consists of these five stages. Here, a stage is being defined as a dedicated part of the processor's logic that has a very specialized function. Note that depending upon the instruction set, we can have more stages, or we might even remove or fuse a couple of stages.

The simplest type of such processors finish all of this work in a single clock cycle. The clock cycle begins with the instruction fetch stage getting activated, and ends when the results of the instruction

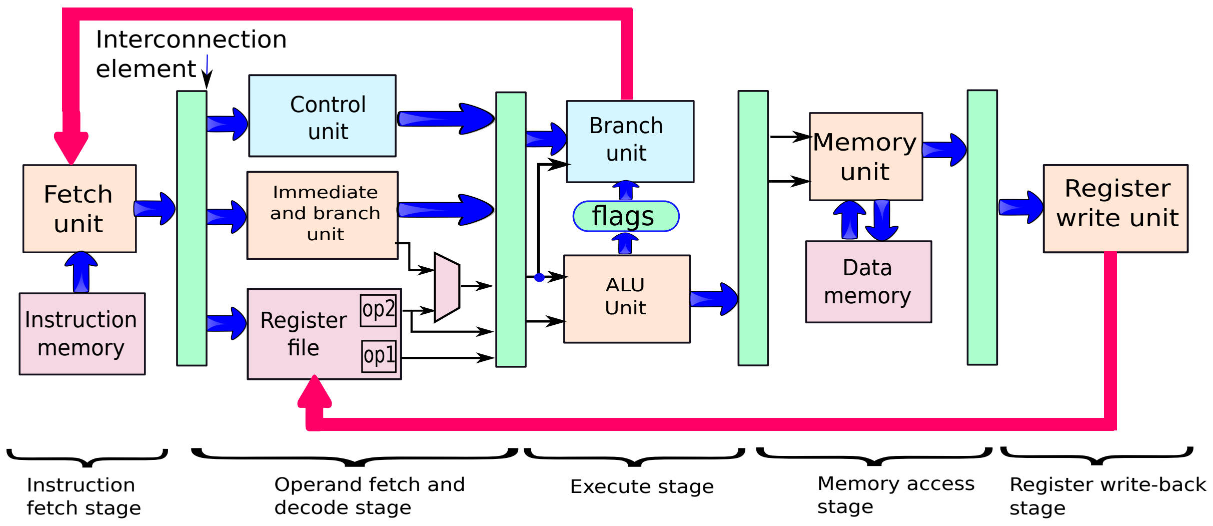


Figure 2.2: A simple RISC processor (adapted from [Sarangi, 2015]). Note the five stages in the design. The design has been purposefully simplified and a lot of the complex aspects of the processor have not been shown. Note the vertical rectangular boxes between stages. These are interconnection elements, where signals and values from the left stage move to components in the right stage.

have been written to the register file. We shall refer to such a processor as a *single-cycle processor* henceforth. Let us summarize our discussion by taking a look at Figure 2.2.

Figure 2.2 shows the design of a single-cycle RISC processor. The fetch stage reads in the instruction from the instruction memory. Subsequently, this is sent to the decode stage via an interconnection element, which in this case is essentially a set of copper wires and multiplexers that connects a set of sources in the left stage to a set of destinations in the right stage (vertical rectangle in the figure). In the second stage (decode and operand fetch), we compute the control signals, the values of immediates (constants) and branch offsets, and read the values of registers. For the second operand in a typical RISC instruction, there is a choice between an immediate and the value read from a register file. Hence, we need to add a multiplexer in this stage. For a much more detailed discussion the reader can refer to [Sarangi, 2015].

In the next stage, which is the execute stage, we perform arithmetic and logical computations, and we also compute the outcome of branch instructions. Conditional branch instructions in modern instruction sets are often evaluated on the basis of the outcome of the last compare instruction. The result of the last compare instruction is stored in a dedicated *flags* register. In addition, for processing return instructions, it is necessary to send the value of the return address read from the register file to the branch unit.

Subsequently, we forward the instruction to the memory access stage, where we perform a load or a store (if required). The last stage is the register write-back stage where we write to the registers (if we have to). Here, we need to handle the special case of the function *call* instruction, where we write the value of the next PC (current PC + $\langle$*size of the instruction*$\rangle$) to the return address register in the register file.

Finally, note the presence of two backward paths: one from the branch unit to the instruction fetch unit, and the other from the register write-back unit to the register file.

Important Point 1
A clock cycle *is a very basic concept in the world of digital electronics. A clock produces a periodic signal. It is typically a square wave; however, it need not be. We can think of a clock as a signal, where a particular event of interest happens periodically. The duration between two such consecutive events is defined as a* clock period*. The reciprocal of the clock period is defined as the* clock frequency*. The unit of clock frequency is Hz, where 1 Hz means that an event of interest happens once every second.*

A clock is a very useful feature in a digital circuit. It gives all the elements a time base. The computation of sub-circuits starts at the beginning of a clock cycle, and finishes by the end of the clock cycle. This helps us have a uniform notion of time across an electronic circuit (such as a processor). Finally, in most circuits the results computed in a clock cycle are stored in a storage element such as a flip-flop by the end of the cycle. The results are visible to another part of the circuit at the beginning of the next clock cycle.

2.1.2 Notion of Pipelining

Now that we know all about a single-cycle processor, we are ready to take a look at its intricacies and explore all of its nooks and crannies.

Consider the single-cycle processor as described in Section 2.1.1. It has five stages, and instructions pass from stage to stage. When the IF stage is active, the rest of the stages are inactive. Similarly, when the MA (memory access) stage is active, the rest of the stages are inactive. If we naively assume that each stage takes up 20% of the area of the overall processor, then at any point of time 80% of the processor is inactive. This clearly does not represent a situation where processing an instruction is happening in a very efficient manner. In an efficient system, all the parts of the processor are expected to be used at any point of time.

However, since one instruction has to move through the stages, the only way to increase efficiency is to process multiple instructions at the same time. When one instruction is in the RW stage, one more can be in the EX stage, and yet one more instruction might be getting fetched from instruction memory. This is the crux of the idea called *pipelining*.

Consider the following snippet of assembly code:

```
1 add r1, r2, r3
2 sub r4, r5, r6
3 mul r7, r8, r8
4 mov r9, r10
5 add r11, r12, r13
```

Here, we have five instructions. There are no dependences between them. Let us send the first instruction (instruction *1*) to the processor first. When it reaches the second stage (operand fetch (OF)), the IF stage will become free. At this point of time, let us use the IF stage instead of allowing it to remain idle. We can fetch instruction *2* when instruction *1* is in the second stage (OF stage). On similar lines, when instruction *2* reaches the second stage (OF) and instruction *1* reaches the third stage (EX), we can start fetching the third instruction (instruction *3*). Figure 2.3 graphically depicts the progress of instructions through the pipeline. Such diagrams are known as *pipeline diagrams.*

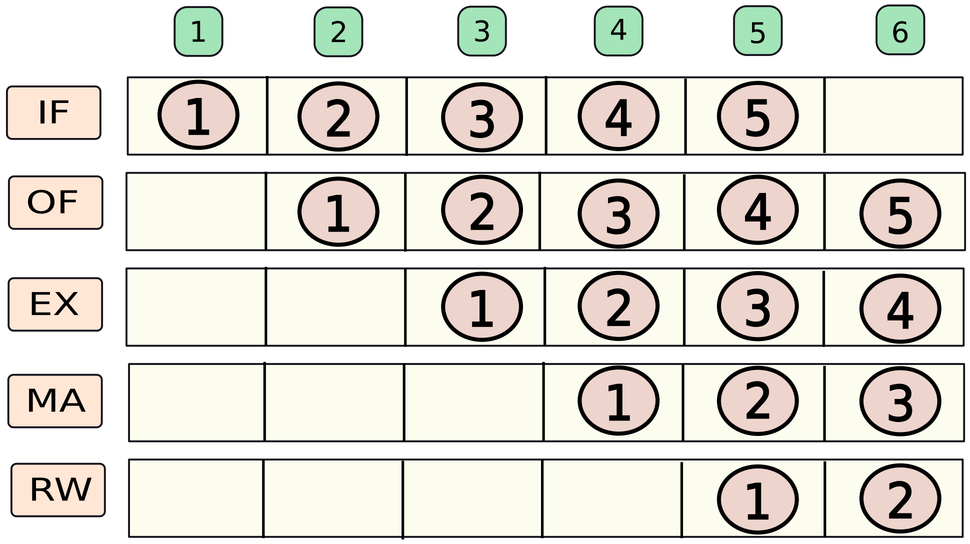


Figure 2.3: Sequence of actions in a pipeline

Implementing a Pipelined Processor

Pipelining in principle as shown in Figure 2.3 appears promising, where we can keep all parts of the processor busy. This should translate to higher efficiency. Sadly, at this point of time we are not in a position to evaluate exactly how much we gain in terms of efficiency.

Let us instead digress a little bit, and understand how to implement pipelining. Let us first consider a single-cycle processor (as described in Section 2.1.1), where the entire processing of an instruction finishes within a single processor cycle. We need to first demarcate different regions of the processor, where each region has a unique function, and it overlaps as little as possible with other regions. We have already done that in Section 2.1.1 by defining five stages within a processor, where an instruction passes through the stages sequentially. Now, we need to ensure that an instruction can pass seamlessly from stage to stage. Imagine a person who changes his hotel every day. On day 1 he is in Hotel 1, and on day 2 he is in Hotel 2, and so on. Since he fully checks out, he needs to carry all his belongings with him, and move from one hotel to the other. The case with instructions is similar. In this case, an instruction needs to move from stage to stage with all its control signals (signals to control the data path) and temporary values.

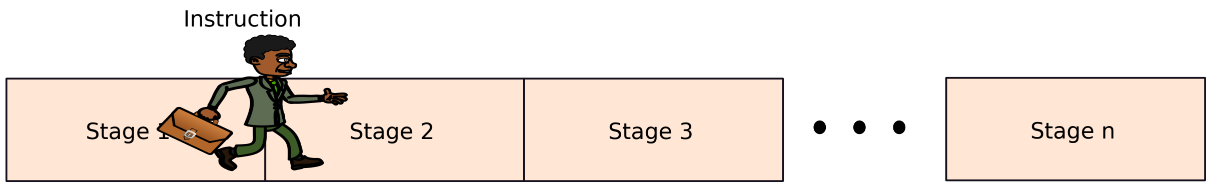


Figure 2.4: Instruction passing from stage to stage along with the instruction packet

An instruction along with its control signals and temporary values is referred to as the *instruction packet*. The instruction packet is like a traveler's briefcase that contains all the necessary information that an instruction requires to execute (see Figure 2.4).

Let us create a hardware mechanism for the instruction packet to move between stages. After every processor stage, we add a latch. A latch is a small memory that stores the instruction packet[1]. It can be made of flip-flops (refer to standard textbooks on digital logic such as [Taub and Schilling, 1977, Lin, 2011]). Each flip-flop typically reads in new data at the downward edge of the clock (when the clock transitions from 1 to 0 as shown in Figure 2.5). Subsequently, the data is stored inside the flip-flop and is visible as an input to the next stage.

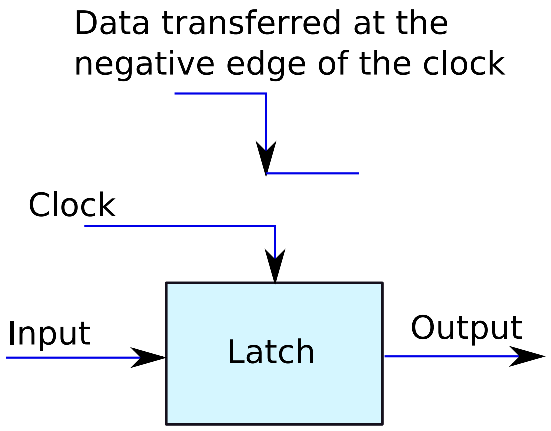


Figure 2.5: A simple clocked latch

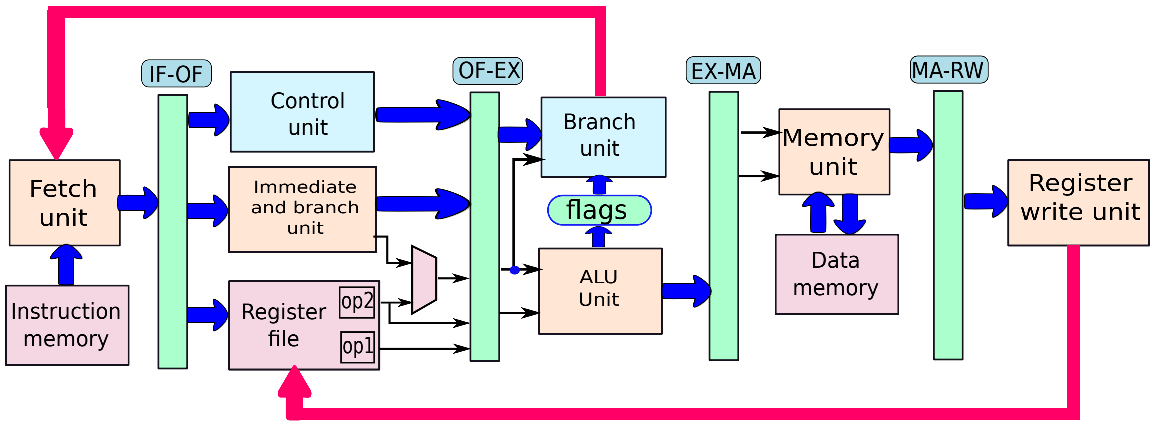


Figure 2.6: 5-stage pipeline with latches between stages

Let us refer to Figure 2.6, where we show the five stages of a typical pipeline. Note that there is a latch (also known as a pipeline register) between successive stages; we just replaced the interconnect element that was present in a single-cycle processor with a latch. The job of the latch is to buffer (or store) the instruction packet. Each of the stages is synchronized by a clock signal. It is provided as an input to each of the pipeline latches. When the clock signal has a negative edge ($1 \rightarrow 0$), the outputs of stage i are stored in the latch at the end of the stage. They are subsequently visible as inputs to the next stage (stage $(i + 1)$).

[1]A latch typically refers to a level triggered memory element. However, in this case it refers to an edge triggered memory element.

It is thus possible to ensure that we have five instructions being processed at the same time in the processor. Each instruction occupies a different stage in the pipeline.

2.1.3 Interlocks

Data Hazards

Sadly, the picture is not all that rosy. We can have dependences between instructions. Consider the following piece of code.

```
add r1, r2, r3
add r4, r1, r3
```

Here, there is a dependence between instructions *1* and *2*. Instruction *1* is writing to register $r1$ and instruction *2* is reading from it. This is called a read-after-write (RAW) dependence. There are problems executing this code.

Consider the fact that when instruction *2* is in the second stage (reading its operands), instruction *1* is in the third stage (execute stage). Since instruction *1* has not written the updated value of $r1$ to the register file, instruction *2* will get an outdated value for register $r1$. Thus, the execution will be **incorrect**. This means that unless we do something then we will have an incorrect execution. This situation is called a data hazard. Refer to Figure 2.7 for a graphical explanation. Recall that such types of diagrams are known as *pipeline diagrams*, where the rows represent pipeline stages, and the columns represent cycles.

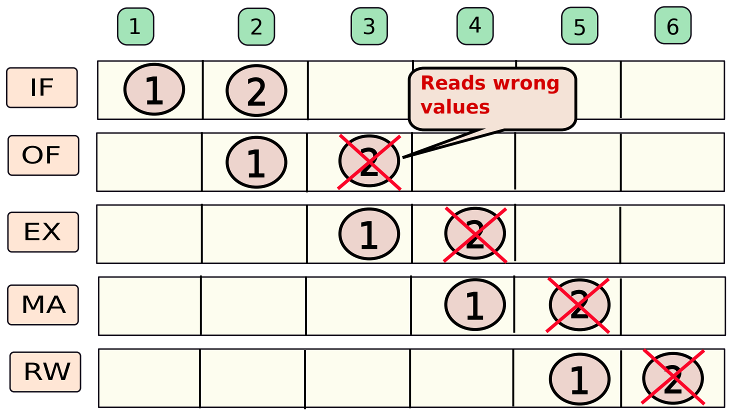


Figure 2.7: Graphical view of a data hazard

Definition 2
A data hazard *is defined as a risk of incorrect execution due to a data dependence not being respected.*

The only way for us to ensure correct execution in this case where there are producer-consumer dependences between instructions is as follows. The first instruction, which is the producer instruction (instruction *1*) can proceed as usual in the pipeline. The behavior of the second instruction (instruction *2*), which is a consumer needs to be changed such that it gets the correct values of its source registers.

1. In the OF (operand fetch) stage, check for data dependences.

2. If there is a producer instruction in a later stage in the pipeline, then there is a data hazard. Stall the consumer instruction in the OF stage. Do not allow it to proceed.

3. This means that some pipeline stages will remain unused.

4. Finally, after the producer instruction writes to its destination register in its write-back(RW) stage, we can allow the consumer instruction to proceed.

This process will ensure that all data dependences are respected, and data hazards do not lead to incorrect execution. This process is shown in the pipeline diagram in Figure 2.8. We can see that between cycles 3 and 5, there is no activity in the operand fetch(OF) stage. Instruction *2* simply waits there for the operand to be ready. Sadly, later stages of the pipeline expect instructions from the earlier stages. In this case, we need to inject dummy instructions in the pipeline. Such dummy instructions are known as *pipeline bubbles*. A bubble basically refers to an instruction that does not perform any operation. We observe that in Figure 2.8, we insert 3 pipeline bubbles in the EX stage (cycles 4-6), and they subsequently propagate down the pipeline.

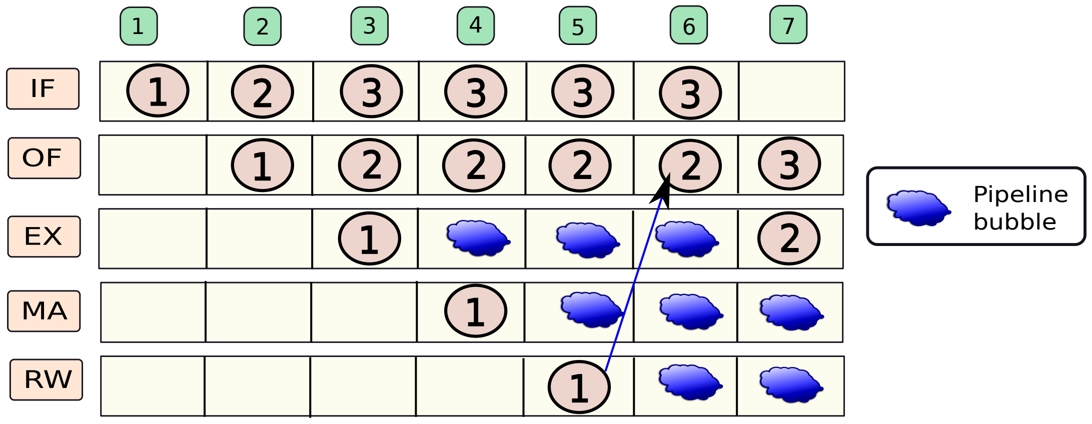


Figure 2.8: Pipeline diagram of a stall (RAW dependence between instructions 1 and 2)

Now assume that there is a RAW dependence between instructions *1* and instruction *3*, and there are no other dependences. In this case the dependence is not between consecutive instructions, but it is between a set of two instructions that have an instruction between them. In this case also, instruction *3* has to wait for instruction *1* to complete its RW (register write-back) stage. However, the time that instruction *3* needs to stall is lower. It is 2 cycles as compared to 3 cycles. Figure 2.9 shows this situation.

Now, it is true that we have avoided data hazards by this technique, which involves stalling the consumer instruction. This method is also known as a *data interlock*. Here, every consumer instruction waits for its producer instruction to produce its value, and write it to the register file before proceeding past the OF (operand fetch) stage. However, this comes at a price, and the price is *efficiency*. Let us see why.

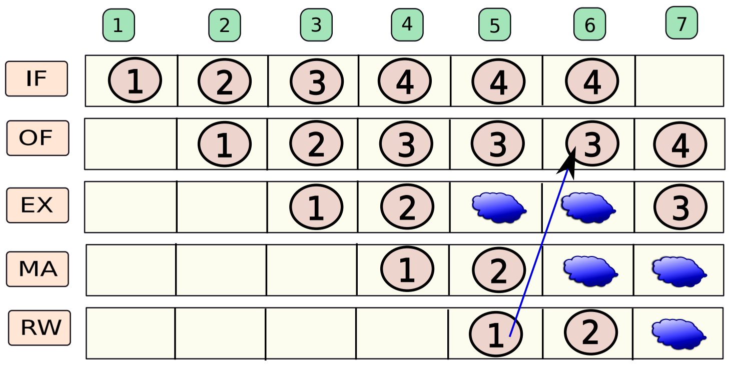


Figure 2.9: Pipeline diagram of a stall (RAW dependence between instructions 1 and 3)

Definition 3
A data interlock *is a method of stalling a consumer instruction in the OF stage till its producer instruction writes the value of the source operand to the register file. While the instruction is stalled, certain pipeline stages are idle (are not processing an active instruction). We can instead say that inactive stages execute invalid (or empty) instructions, which are popularly known as* pipeline bubbles*.*

Consider a person standing at the end of the last stage of the pipeline – the write-back stage (RW). Ideally, she will observe one instruction getting successfully executed every clock cycle. Thus, the CPI (clock cycles per instruction) will be 1. However, the moment we factor in dependences, we would have to stall the pipeline.

We thus have the following equation:

$$CPI = 1 + stall\_rate \times stall\_penalty \qquad (2.1)$$

Here, the ideal CPI is 1 (also the CPI of a single-cycle processor). The term *stall_rate* denotes the probability of a stall per instruction, and the *stall_penalty* is measured in cycles. For example, it is 3 cycles if there is a RAW dependence between consecutive instructions. We see that as we have more dependences, the CPI will increase because of the increased number of stalls. An increase in the CPI basically means that it takes more time to process an instruction, and the processor thus effectively gets slower. This is a bad thing, and should be avoided. However, in the interest of correctness, this needs to be done unless we come up with a better idea.

Control Hazards

Sadly, data hazards are not the only kind of hazards. We also have control hazards, which arise because of specific complications while handling branch instructions.

Consider the following piece of code.

```
beq .foo
add r1, r1, r3
add r4, r5, r6
...
...
.foo:
```

Here, the first instruction (instruction *1*) is a branch instruction. It is a conditional branch instruction, which is dependent on a previous comparison. If that comparison resulted in an equality, then we jump to the label (*.foo*). The next two instructions are regular arithmetic instructions without any dependences between them. Let us assume that the branch instruction, *beq*, is taken. Then, the point to note is that because of the branch instruction, we will not execute instructions *2* and *3*. We will instead branch to the label .foo. In this case, these two instructions are said to be on the *wrong path*.

Definition 4

- Wrong path instructions *are defined as the set of instructions after a branch in program order, which are not executed if the branch is taken. The correct path is defined on similar lines.*
- *A* control hazard *is a situation where there is a risk of executing instructions on the wrong path.*

The question is, "When do we know if instructions *2* and *3* are on the correct path or the wrong path?" We will only know that when instruction *1*, which is the branch, will reach stage 3 – the execute stage. Here, we will consider the result of the last comparison, and see if that resulted in an equality. If yes, then we need to jump to the branch target *.foo*. This means that in the next clock cycle, we will start fetching instructions from the instruction address corresponding to *.foo*.

However, in pipeline stages 1 and 2 (IF and OF), we have two instructions, which are unfortunately on the wrong path. They have been fetched, and unless something is done, they might very well execute and write their results to either the data memory or the register file. This will corrupt the program's state and lead to incorrect execution, which is definitely not desirable. Hence, when we realize that instruction *1* is a taken branch, we can automatically infer that instructions in the two previous stages (instructions *2* and *3*) are on the wrong path. They need to be canceled. This means that we can set a bit in the instruction packet, which indicates that the instruction packet and the contained instruction is henceforth invalid. We thus convert instructions on the wrong path to pipeline bubbles. This is shown in Figure 2.10. Note that we need to retrospectively convert two instructions to bubbles, when we realize that they are on the wrong path.

We can alternatively say that we have incurred a stall of two cycles when we encounter a taken branch. We can infer a stall because we do not do any useful work in the two cycles after a taken branch. This is called a *branch interlock* where we are basically stalling the pipeline upon encountering a taken branch.

Definition 5
A branch interlock *is a method to stall a pipeline for several cycles to ensure that instructions on the correct path are fetched from instruction memory.*

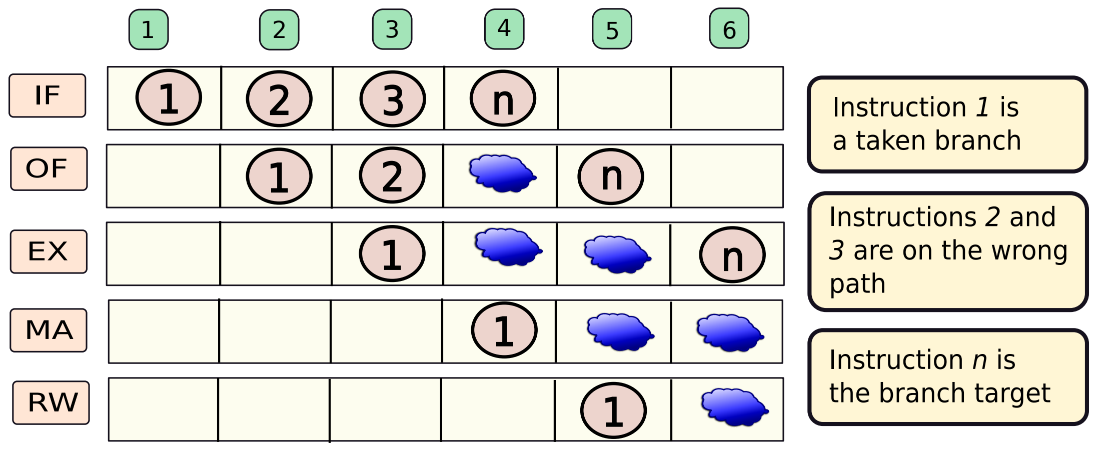

1
2
3
4
5
6
IF
OF
EX
MA
RW
1
2
3
n
1
2
n
1
n
1
1
Instruction *1* is a taken branch
Instructions *2* and *3* are on the wrong path
Instruction *n* is the branch target

Figure 2.10: Nullifying two instructions on the wrong path

Thus, we can conclude that there are two ways of stalling processors: data interlocks and branch interlocks. Both of them increase the CPI as per Equation 2.1. There exists a trade-off between correctness and performance. To achieve correctness, we had to sacrifice performance.

Delayed Branches

Note that there are many ways of reducing the CPI while preserving correctness. Some of the approaches include (but are not limited to) branch prediction and delayed branches. We shall discuss branch prediction in great detail in Chapter 3. Let us take a very brief look at delayed branches here. Since we stand to lose the two successive cycles after fetching a taken branch, let us instead try to utilize these slots. Let us refer to these time slots as *delay slots*. The idea is to bring two instructions before the branch (which are on the correct path), and place them in the delay slots. This has to be done by the compiler while generating the machine code for the program. The caveat is that these instructions should not determine the direction of the branch instruction and should preferably not have any dependences between them.

Example 1
Reorder the following code snippet assuming that the hardware supports delayed branches. Assume two delay slots.

```
add r1, r2, r3
sub r4, r2, r3
beq .foo
...
...
.foo:
```

Answer:

```
beq .foo
add r1, r2, r3
sub r4, r2, r3
...
...
.foo:
```

Now, after fetching the branch instruction, let us execute the instructions in the delay slots as normal instructions. Since these instructions would have been executed anyway irrespective of the direction of the branch, we have not hampered correctness in any way. Once the direction of the branch is determined in the third cycle (EX stage), we can proceed to fetch instructions from the correct path. This approach does not require any stall cycles, because irrespective of the direction of the branch, we do not execute instructions on the wrong path. Please refer to Example 1 for an example of code reordering in the presence of delayed branches.

For a more extensive introduction to delayed branches, please refer to the book by Sarangi et al. [Sarangi, 2015]. However, the pitfalls of this approach lie in the fact that the compiler has to be aware of the details of the hardware, and this approach will constrain binaries to only work on only one kind of processors. For example, if a given processor has four delay slots, then such binaries will not work on them. This is not desirable. Let us search for a better method.

2.1.4 Forwarding

Data interlocks are inefficient mechanisms when it comes to eliminating the risk of incorrect execution due to data hazards. We stall the pipeline and insert bubbles. In the worst case when there is a data dependence between consecutive instructions, we need to stall for at the most 3 cycles. This represents wasted work. The important question that we need to answer is, "Are the stalls necessary?"

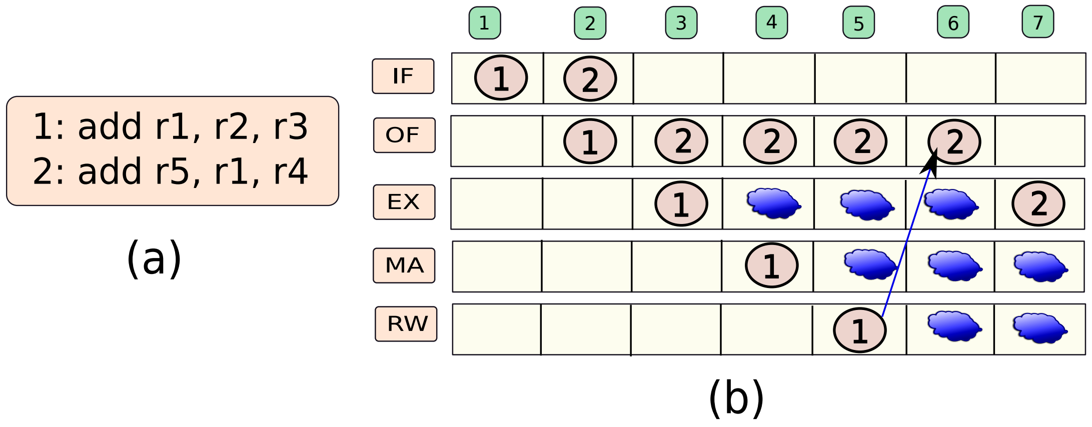


Figure 2.11: Example of a stall

Let us consider the piece of code in Figure 2.11(a), and its associated pipeline diagram in Figure 2.11(b). There is clearly a read-after-write (RAW) dependence between instructions *1* and *2* via register $r1$. With traditional data interlocks we need to stall instruction *2* till instruction *1* gets past the last stage (write-back (RW) stage). However, do we really need this delay?

Pay greater attention to Figure 2.11. Now, answer the following questions:

1. When does instruction *1* produce its destination value ($r1$)?
2. When does instruction *2* need the value of $r1$?

The answers are as follows.

1. Instruction *1* produces its result at the end of cycle 3 when it finishes executing its EX stage.
2. Instruction *2* needs the value of $r1$ at the beginning of cycle 4 – the beginning of its execution in the EX stage.

The important point to note is as follows: instruction *2* needs the value of $r1$ just after instruction *1* produces it. In real terms there are no data hazards. The result is available; it is albeit not present in the register file. That is not a big issue. Here is how we can solve the problem.

1. We can store the value of $r1$ that instruction *1* produces in its instruction packet.
2. This value will be present in the instruction packet at the beginning of cycle 4. At this point, instruction *1* will be in the MA stage (fourth stage).
3. We can **forward** the value of register $r1$ from the MA stage to the EX stage where instruction *2* needs it.
4. The value will arrive just in time for instruction *2* to execute correctly.
5. There will thus be no data hazards, and the code will execute correctly. Furthermore, there is no need to stall the pipeline. We have thus **eliminated stalls**.

This method is known as *forwarding*, or *bypassing*, where we pass results between pipeline stages.

Definition 6
A technique to pass results between pipeline stages to eliminate data hazards is known as forwarding *or* bypassing*.*

Forwarding Paths

To enable forwarding, we need to add a connection (forwarding path) between the beginning of the MA stage and the EX stage. This means that at the beginning of the EX stage, we need to choose between the operand that we have read from the register file in the previous stage (OF), and the forwarded value coming from the MA stage. We need to add a circuit that helps us choose between the two inputs. This is typically a multiplexer that helps us choose between the inputs (refer to Figure 2.12).

In Figure 2.12, we have two inputs, and we need to choose the correct input based on whether we are forwarding inputs or not. We thus need a separate forwarding unit in the chip whose job is to compute the control signals for all the *forwarding multiplexers* in the processor. These control signals are used to choose between the value read in the OF stage and the forwarded value.

Figure 2.13 shows the modified pipeline diagram with forwarding. Note the arrow between the stage that forwards the value and the stage that uses it.

We have taken a look at only one example of forwarding, where we forward from the MA stage to the EX stage. However, this is not the only example of forwarding. We can have many more forwarding paths. For a deeper explanation of forwarding paths please refer to the textbook by Sarangi et al. [Sarangi, 2015]. We shall take a look at this issue very superficially in this book. A lot of concepts in this section

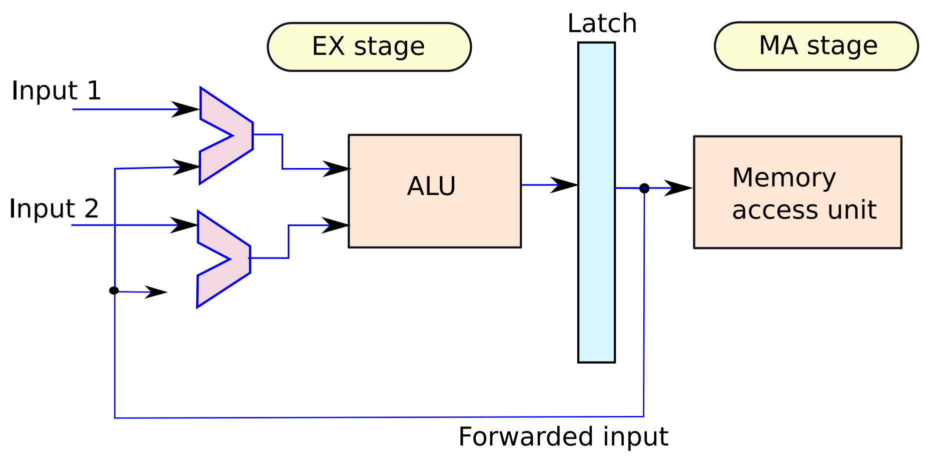


Figure 2.12: Circuit used to forward values from the MA stage to the EX stage

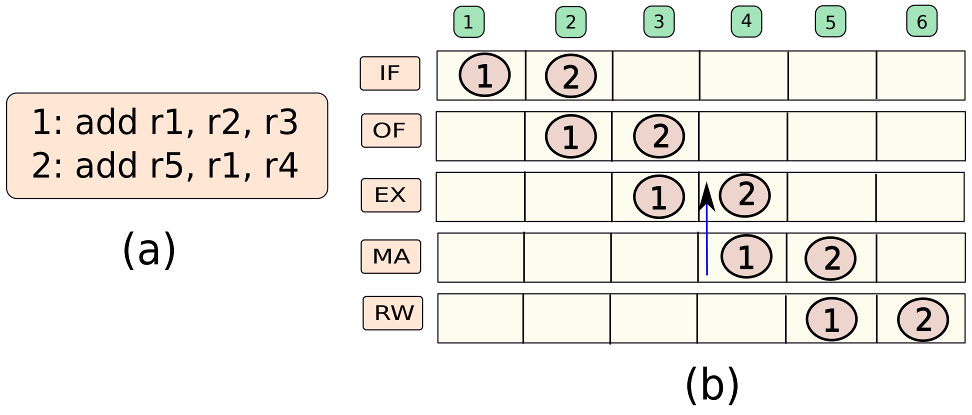


Figure 2.13: Modified pipeline diagram with forwarding

will be presented without proofs. The assumption is that readers will do their due diligence in picking up the background material from textbooks that look at in-order pipelines in detail.

Naively, we need forwarding paths between every pair of stages; however, this is overkill. Not all the paths are used. Let us settle on the following principles, while deciding on the forwarding paths.

Forwarding Principle 1 We forward from a later stage to an earlier stage.

Forwarding Principle 2 We forward as late as possible. This means that if we can delay forwarding by one or a few more cycles, then we do it. Note that we never compromise on correctness.

Now if we work out the details of the forwarding paths, we arrive at the following four forwarding paths:

Forwarding path	Example of code

RW → MA	`ld r1, 8[r2]`
	`st r1, 8[r3]`

RW → EX	`ld r1, 8[r2]`
	`sub r5, r6, r7`
	`add r3, r2, r1`

RW → OF	`ld r1, 8[r2]`
	`sub r5, r6, r7`
	`sub r8, r9, r10`
	`add r3, r2, r1`

MA → EX	`add r1, r2, r3`
	`sub r5, r1, r4`

We need to quickly understand, why these are the forwarding paths that we require, and no additional forwarding paths are needed. The forwarding paths need to always be from a later stage to an earlier stage (Forwarding Principle 1). Second, it makes no sense to forward from one stage to itself. Finally, it also makes no sense to forward a value to the IF stage. This is because in the IF stage we have not decoded the instruction, and we are thus not aware of its contents. This leaves us with the following forwarding paths: $RW \rightarrow OF$, $RW \rightarrow EX$, $RW \rightarrow MA$, $MA \rightarrow EX$, $MA \rightarrow OF$, $EX \rightarrow OF$.

Now, consider the forwarding path $MA \rightarrow OF$. This is not required because we do not have an immediate need for any value in the OF stage. In accordance with *Forwarding Principle 2*, where we forward as late as possible, we can instead use the $RW \rightarrow EX$ forwarding path. We can argue on similar lines that we do not need to add the forwarding path $EX \rightarrow OF$. We can use the $MA \rightarrow EX$ path instead.

We are thus left with four forwarding paths, which are required: $RW \rightarrow OF$, $RW \rightarrow EX$, $RW \rightarrow MA$, and $MA \rightarrow EX$.

For each input in these stages, we need to have a multiplexer. In each multiplexer, we have several input terminals, where we choose between the default value (from the previous stage), and values forwarded from the forwarding paths. Figure 2.14 shows the augmented structure of the pipeline after adding forwarding paths.

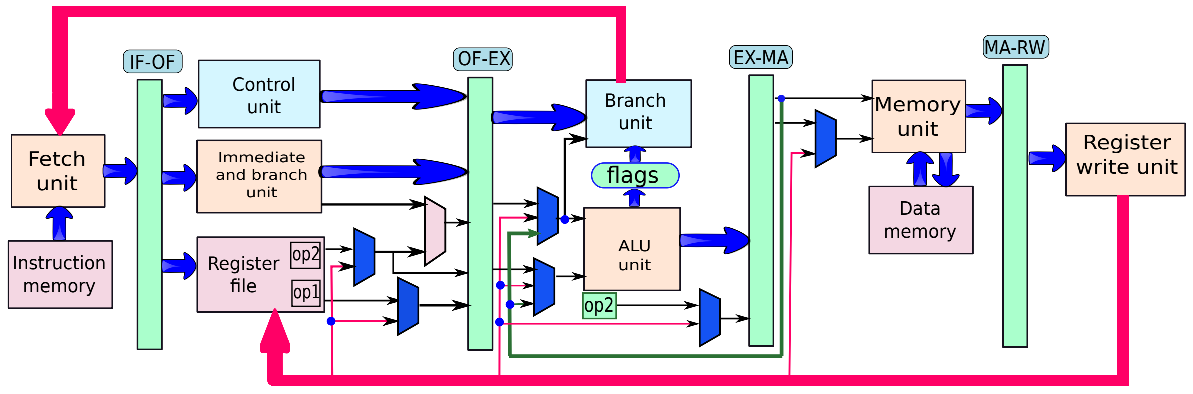


Figure 2.14: The pipeline with forwarding paths

Unfortunately, we cannot remove all RAW hazards using forwarding. There is one special case in a 5-stage pipeline that cannot be handled with forwarding. It is called a load-use hazard. Consider Figure 2.15.

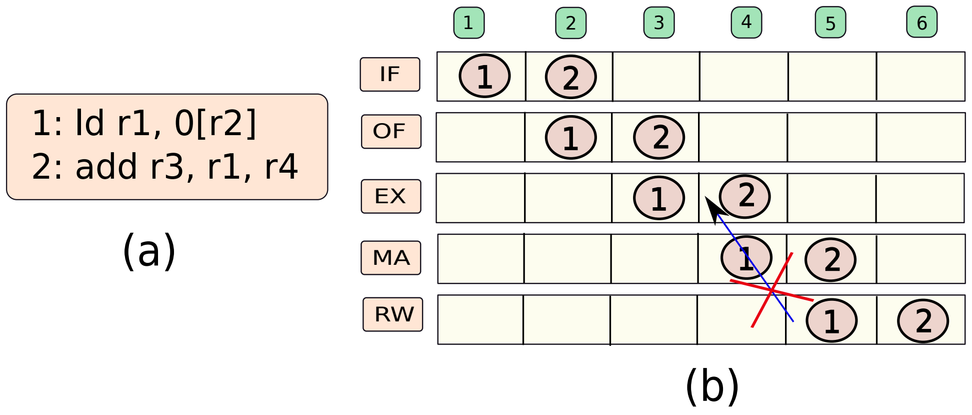


Figure 2.15: Pipeline diagram for a load-use hazard

Here we load a value and put it in register $r1$. Then this value is used by the subsequent add instruction. We can see in the corresponding pipeline diagram (see Figure 2.15(b)) that we produce the value of the load at the end of the 4^{th} cycle, and it is available for forwarding at the beginning of the 5^{th} cycle. However, the add instruction needs the value of $r1$ at the beginning of the 4^{th} cycle. Thus, it is not possible to forward the value from the RW to the EX stage. This is the only case in a typical 5-stage pipeline where forwarding is not possible. The only way to solve this issue is by adding a one-cycle delay (one bubble) between instruction *1* and instruction *2*. This is shown in Figure 2.16.

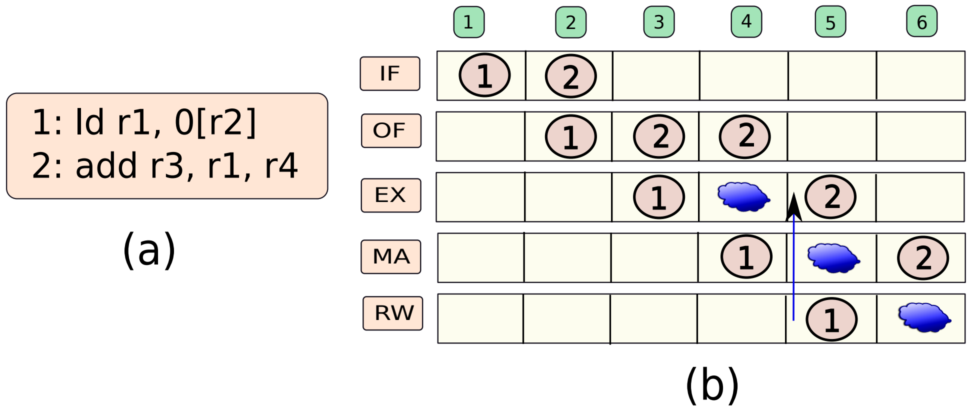


Figure 2.16: Pipeline diagram for a load-use hazard (we add a pipeline bubble here)

By using forwarding we have reduced the number of stall cycles to zero in most cases. Other than the case of load-use hazards, we do not need to stall the pipeline. If we use delayed branching, we can also eliminate the need for stalling for 2 cycles after a taken branch. However, as we discussed in Section 2.1.3 delayed branches reduce the portability of the code, and it may be difficult to find instructions that can be put in the delay slots. Our aim was to ideally reduce the CPI to 1 as far as possible. We have proposed a lot of fundamental mechanisms in this section; however, we are still short of this goal. Furthermore, in modern processors, we would like to also reduce the CPI to a number below 1 by issuing multiple instructions in parallel. In the subsequent sections, we shall discuss such designs.

2.2 Performance Considerations

2.2.1 The Performance Equation

The performance (P) of a program is defined as a quantity that is inversely proportional to the time that the program takes to execute. The units of performance are arbitrary (do not matter). We can always compare the relative performance of two programs as a ratio. In fact that is how the performance of programs and suites of programs in the popularly used SPEC benchmarks [Henning, 2006] is evaluated.

Let us assume that the constant of proportionality is 1 for ease of explanation. Now, let us derive the performance equation from first principles. Assume that a program takes $\#secs$ to execute (in seconds), and it has $\#insts$ dynamic instructions. Note that the term *dynamic instructions* refers to the actual number of instructions the processor executes. This is not the same as the term *static instructions*, which is the number of instructions in a program's binary.

$$\begin{aligned} P &= \frac{1}{\#secs} \\ &= \underbrace{\frac{\#insts}{\#cycles}}_{IPC} \times \underbrace{\frac{\#cycles}{\#secs}}_{f} \times \frac{1}{\#insts} \\ P &= \frac{IPC \times f}{\#insts} \end{aligned} \tag{2.2}$$

The rest of the terms are defined as follows. IPC refers to the average number of instructions the processor executes per cycle, and f is the clock frequency.

The implications of the performance equation are very profound. The IPC is determined by the architecture. The more forwarding we have in a pipeline, better (higher) is the IPC. Additionally, it is also determined by the way the compiler organizes the code. Consider the following assembly code snippet.

Unoptimized code

```
ld r1, 4[r10]
add r3, r1, r2
ld r5, 4[r11]
add r7, r6, r5
```

This code is clearly not optimal. There are two load-use hazards between the first and second statements, and the third and fourth statements respectively. There will thus be two stalls in a typical 5-stage pipeline. However, it is possible for the compiler to reorganize the code such that there are no load-use hazards.

Optimized code

```
ld r1, 4[r10]
ld r5, 4[r11]
add r3, r1, r2
add r7, r6, r5
```

This code snippet does not have any load-use hazards and will thus not have any stalls in a pipeline with forwarding. Thus, the compiler and the hardware have an important role in determining the IPC.

Let us come to the next variable in the performance equation: number of instructions ($\#insts$). This is predominantly determined by the compiler. Better compilers can generate shorter code sequences for the same snippet of high-level code.

The frequency, f, is determined by two factors namely the technology and the computer architecture. If we use smaller and more power efficient transistors, then we can run the circuit at a higher frequency. Note that power consumption is roughly proportional to the cube of the frequency, and thus using power efficient transistors are critical for running a chip at high frequencies. There are many other aspects to the *technology* aspect, which we shall study in later chapters. For example, it is possible to reduce the power consumption of a processor without actually using more power efficient transistors. We can use tricks at the level of the processor architecture.

The frequency is also very intimately related to the design of the processor. Before elaborating further, let us debunk a common myth associated with pipelined processors. The most common myth with regard to pipelining in a conventional pipelined processor is that since we can process more instructions at the same time, the performance is higher. This is **NOT CORRECT**, and neither is it according to the performance equation.

Let us compare the execution of the same program on two processors: one has a 5-stage pipeline, and the other does not have pipelining (single-cycle processor). Since they run the same program, the number of instructions is the same. The IPC of the single cycle processor is 1. The IPC of the pipelined processor is at the most 1, and is often less than 1. This is because of stalls in the pipeline. While most RAW hazards are avoidable, stalls associated with control hazards (on the wrong branch path) and load-use hazards are not avoidable and lead to wasted processor cycles. Thus, the IPC of the pipelined processor in all likelihood is less than 1. Now, if both the processors have the same frequency, then the performance of the single-cycle processor is more than that of the processor with pipelining. Where is the advantage of pipelining then?

The answer is that if we keep the frequency the same, there is no advantage of pipelining. The advantage of pipelining comes from the fact that we divide the processor into five smaller sub-processors (stages), where each sub-processor takes a lesser amount of time to complete its work. We can thus reduce the clock cycle period, where each clock period corresponds to the maximum time that it takes each stage to complete its work. If a pipeline with k stages is balanced (each stage takes roughly the same amount of time), then we can more or less reduce the clock period by a factor of k. This corresponds to a k times increase in the frequency(f). It is this increase in the frequency that allows us to realize the gains in pipelining. A mere implementation of pipelining does not give us the benefits, it is rather the opportunity to increase the frequency, which is more important. Keep in mind that the process of increasing the frequency has its limits: power consumption and pipeline latch delay.

Important Point 2
The gains in pipelining come from the fact that we can increase the clock frequency significantly.

Most beginners miss this point completely, and incorrectly assume that just processing multiple instructions in parallel (albeit in different stages) is good enough. This is far from the truth since neither the IPC nor the frequency increase. However, splitting the entire processor's circuit into numerous sub-circuits helps us have a much faster clock.

Sadly, this approach does not take us very far. We cannot arbitrarily increase the frequency by creating more and more pipeline stages. We will very soon have diminishing returns because the latch delay (delay in the pipeline registers) will start to dominate the timing, and secondly we will greatly increase the number of stalled cycles. Additionally, there is a cubic dependence between frequency and power. Given the fact that power and temperature are extremely important issues today, we cannot expect to increase on-chip frequency (beyond the current levels of 3 to 3.5 GHz). Because of this reason the frequency has remained roughly static since 2005.

2.2.2 Multi-issue In-order Pipelines

Given the fact that we cannot increase the frequency any further due to power and temperature constraints, and it is also not possible to reduce the number of dynamic instructions significantly, we need to look at methods to increase the IPC. Sadly, after introducing pipelining, the IPC became less than 1 for most programs (due to the presence of stalls).

We can increase the IPC by processing more than one instruction per stage simultaneously. Here we focus on in-order pipelines where instructions are fetched and processed in program order. Note that the term *program order* refers to the order of dynamic instructions that a single-cycle processor will see.

Consider the following code sequence.

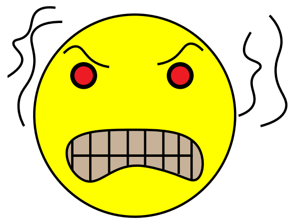
```
1 add r1, r2, r3
2 sub r4, r5, r6
3 mul r7, r8, r9
4 div r10, r11, r11
```

All the four instructions are independent of each other. We can treat instructions *1* and *2* as a single bundle and send them together in the pipeline. We just need to duplicate the resources. Instead of one decode unit, we need to have two. Similarly, we can duplicate the execute and memory access units. The second bundle (instructions: *3* and *4*) has no conflicts with the first bundle. Both the instructions in the second bundle can be sent through the pipeline in the next cycle after the instructions in the first bundle are sent.

However, we shall seldom be that fortunate. Note that hardware designers need to be wary of Murphy's law. This law states that if something can go wrong, it will definitely go wrong.

Trivia 1
Murphy's law states that if something can go wrong, it will go wrong. It is not a scientific law, it is rather a pessimistic view of the world. For engineers, it means that they should always consider the worst possible case irrespective of its chance of occurrence while designing a system.

Example of Murphy's Law: *Imagine you fall down and break your leg, then you try to call the hospital, the phone is dead, then you knock your neighbor's door, nobody answers, then you somehow crawl to the street, no car stops for you, then you finally hop into your car and try to drive with the leg that is working, you get caught in a traffic jam, finally, when you reach the hospital, you find that it has closed down, and in its place a kids' amusement park has come up !!!*



Now, what can go wrong? We can have RAW hazards within instructions in a bundle, we can have RAW hazards across instructions from different bundles, we can have control hazards, and we can have two instructions in the same bundle accessing the same memory address. All of these issues will make the design of such a pipeline very difficult. Ensuring correctness will require a lot of additional circuitry. Always remember that additional circuitry implies more area, more power, and often more latency. The reason that additional circuitry slows down the execution of a processor is two-fold. First, it increases the length of the critical path. This means that signals need to travel for a longer duration – through more wires and transistors – to reach the end of a pipeline stage. Second, it increases the *routing* and *placement* overhead.

Let us briefly explain these terms. Circuit designers typically design their circuits in a hardware description language such as Verilog or VHDL. These tools are known as EDA (electronic design and automation) tools. Their job is to arrange transistors and wires in a circuit. The reasons for doing so are to reduce the area, decrease the signal propagation latency (alternatively the clock period), and reduce power consumption. Two of their main tasks are placement and routing. The process of *placement* refers to the proper arrangement of blocks of transistors in a circuit to increase its efficiency. For example, we should place the decode unit close to the execute unit in a processor, and the fetch unit should be far away from the memory access unit. If these constraints are not respected, then signals will take a long time to traverse along the wires, and the clock period needs to be unnecessarily increased. The process of *routing* arranges the wires in a circuit. A large processor with billions of transistors has billions of small copper wires also. No two wires can intersect. Additionally, we need to ensure that signals reach their destination as soon as possible. As a result, we need to reduce the length of wires as much as possible. This is a very complicated process.

Definition 7

Placement *It is the process of arranging electronic components in a chip to produce an efficient circuit. The criteria of* efficiency *can vary but typically focuses on reducing the area and the clock period.*

Routing *It is the process of laying out the wires in a circuit to make it more efficient.*

Now assume that we introduce a new piece of circuitry in a processor. We need to place it somewhere close to the other circuits that will use it. This will cause some amount of displacement of the other components, and most likely the placement tool will need to make compromises to accommodate this new circuit. The new piece of circuitry will also have its own wires, and this will complicate the routing process further. As a result, signals will take longer to reach their destination in the vicinity of the new circuit. Such effects can lead to a slowing down of the frequency, and an increase in the chip area. Hence, as far as possible, we try to avoid introducing new circuits in a high performance processor unless the gains outweigh the costs associated with placement and routing.

Adding a new feature clearly has issues in terms of complexity, routing, and placement. Let us gauge the expected benefits of such a scheme. Consider an in-order pipeline that fetches and executes multiple instructions at the same time. First, the onus of producing good code falls on the compiler. If the compiler produces code where instruction i is dependent on instruction $i+1$, then both of them cannot be a part of the same bundle, and we need to stall the pipeline. It is in general not a good idea to make the compiler do all the work because it becomes way too dependent on the architecture. Additionally, a program that runs well on one architecture will not run well on another architecture. For example, consider a processor A that can process two instructions at a time, and a processor B that can process three instructions at the same time. A program optimized for processor A might perform very poorly on processor B.

The more important question is, "How frequently will we find code sequences that can be rearranged to work optimally on such a multi-issue in-order processor?" Here the term, "multi-issue", means that we *issue* multiple instructions to the execution units simultaneously. The answer is that this will typically not happen. Most of the code that we write uses the results of the immediately previous statements, and thus we expect a lot of dependences across consecutive instructions to be present. As a result, the gains are expected to be limited. Other than a few processors such as the Intel® Pentium® processor, most of the other commercial processors have not adopted this approach.

2.3 Overview of Out-of-order Pipelines

2.3.1 Motivation

Given the fact that multi-issue in-order pipelines have limited benefits, we need to think of a new way of increasing IPC. To get inspiration for an idea, let us consider the following snippet of code.

```
add r1, r2, r3
sub r4, r1, r5
mul r6, r7, r8
div r9, r6, r7
```

Here the instructions have a dependence between them. Instruction *2* is dependent on the result of *1*, and *4* is dependent on the result of *3*. If the operator, →, depicts the fact that the event on the left-hand side needs to happen before the event on the right-hand side, we have, *1* → *2* and *3* → *4*.

The first solution is to rely on the compiler to re-organize the code. However, as discussed in Section 2.2.2, this is not a very good idea because it limits the generality of compilers, and makes programs very dependent on the type of processor that they are running on. We need a generic mechanism that can execute such code sequences without compiler support. The other issue with compilers is that they are great for pieces of code that they can analyze; however, a lot of code is dependent on run time parameters and cannot be analyzed at compile time. For example, if we have a lot of code involving memory accesses, we will not know the values of the addresses till the code actually executes. It is very difficult for the compiler to optimize the code beforehand. We thus need a generic mechanism in hardware to optimize the execution sequence of dynamic instructions such that it is possible to execute as many instructions as possible in parallel. This is referred to as *instruction level parallelism* (ILP).

Definition 8
The term instruction level parallelism *(ILP) is a measure of the number of instructions that can be executed simultaneously. For two instructions to be executed simultaneously, they should not have any dependences between them.*

Let us take a second look at this piece of code and find out the degree of ILP. The dependences are *1* → *2*, and *3* → *4*. We can process instructions *1* and *3* together, and then process *2* and *4* together. In this case no dependence will be violated, and we can thus execute 2 instructions simultaneously without any issues. Figure 2.17 shows a conceptual view of such an execution. Note that this execution does not follow program order. The execution is *out of order*.

The main challenge is to automatically identify such sets of independent instructions in large instruction sequences, and execute them without causing any correctness issues. Note that the execution shown in Figure 2.17 has two distinguishing features.

1. We are processing multiple instructions in parallel.

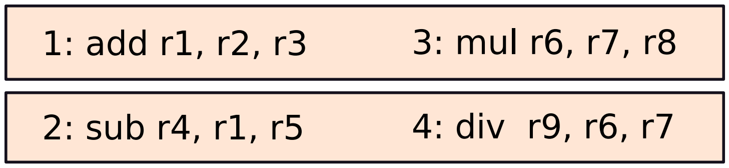


Figure 2.17: Parallel execution of instructions out of order

2. The instructions are not executing in program order. In this case instruction *3* is executing before instruction *2*. Instructions are executing *out of order*.

Let us take this opportunity to define two new terms: *superscalar* execution, and *out-of-order* execution. A superscalar processor fetches and executes multiple instructions simultaneously. In this case, we are exactly doing this. We are fetching and executing two instructions simultaneously. Now, there can be two kinds of superscalar processors. We can either execute instructions in program order (in-order processing as described in Section 2.2.2), or execute them in a different order. The latter approach is known as out-of-order processing, and a processor, which executes instructions out of order is known as an out-of-order processor. We shall often use the term OOO as an abbreviation for out-of-order.

Definition 9

Superscalar processor *A superscalar processor fetches and executes multiple instructions simultaneously.*

Out-of-order processor *An out-of-order processor executes instructions in an order that might not be the same as the program order. Note that data dependences are never violated. If instruction B is dependent on the output of instruction A, then an out-of-order processor will always execute B after A.*

Most high performance modern processors are out-of-order processors primarily because of their potential to increase the IPC. Let us consider our running example once again. We have the following dependences: $1 \rightarrow 2$ and $3 \rightarrow 4$. If the instruction sequence from instructions *1* to *4* is executed by an in-order processor that can process two instructions simultaneously, the IPC will still be less than 2 because instructions *1*,*2* and instructions *3*,*4* have dependences between them. However, we can always execute them out of order and get an IPC of 2 as shown in Figure 2.17. If the frequency remains the same, we can improve the performance of the processor by a factor of 2.

2.3.2 Program Order versus Data Dependence Order

An in-order processor executes instructions in program order. This approach limits the ILP and thus leads to lower performance. We often cannot find enough independent instructions to execute in parallel. Even if we have a very smart compiler that can reorder code very efficiently, it is still very hard to extract high levels of ILP because a large part of the behavior of programs is determined at run time.

It is much better to execute instructions out of order. Does it mean that we can arbitrarily execute instructions in parallel. The answer is, NO. If instruction B is dependent on the result of instruction A, then we still have to respect the order $A \rightarrow B$. This is known as the *data dependence order*. Let us formally define it.

Definition 10
The data dependence order *is defined as an ordering of instructions where instruction A must appear before B, if B is dependent on the results generated by instruction A.*

Note that the data dependence order is a transitive relationship. This means that $A \rightarrow B$ and $B \rightarrow C$ implies $A \rightarrow C$.

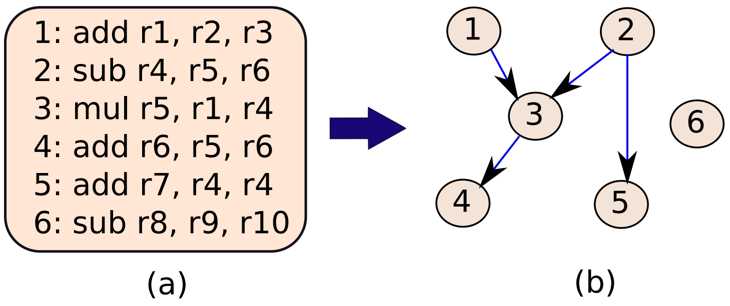
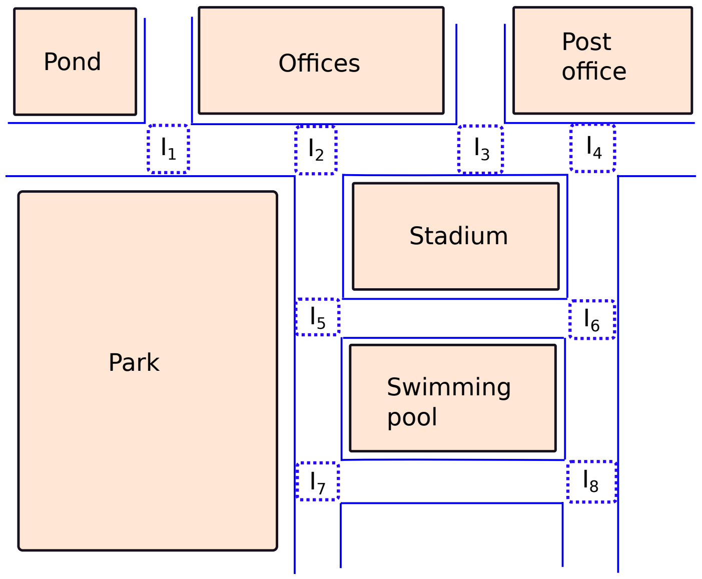

```
1: add r1, r2, r3
2: sub r4, r5, r6
3: mul r5, r1, r4
4: add r6, r5, r6
5: add r7, r4, r4
6: sub r8, r9, r10
```

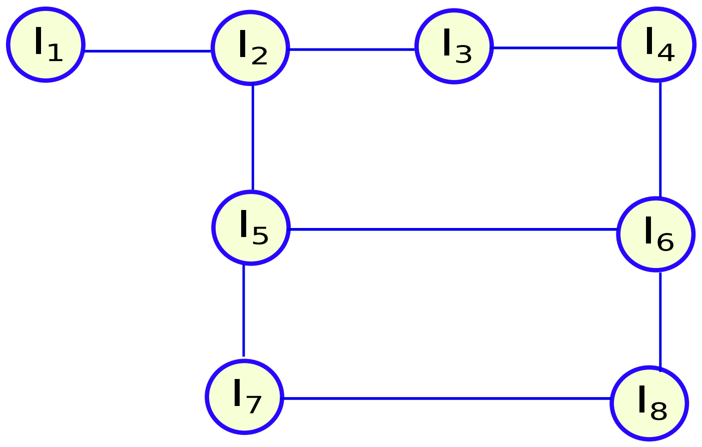
Figure 2.18: Code sequence modeled as a directed acyclic graph (DAG)

Definition 11

A graph *is a data structure in Computer Science. A data structure is a method of conveniently representing complex data. Consider the map of a city as shown below. Here the intersections are shown as rectangles with dotted boundaries. The intersections are numbered: $I_1 \dots I_8$.*



The main problem with such complicated scenarios is that they cannot be processed by a program. They need to be converted to a simpler format. We thus represent the map of a city as a graph. A *graph as shown below contains a set of nodes (or vertices), which in this case are the intersections. The intersections are connected by edges, where each edge is a segment of the road that connects the intersections. An edge can be annotated with additional information that indicates the length of the segment of the road, or other attributes.*



The advantage of modeling such a scenario as a graph is that this data structure can easily be analyzed by a program. For example, we can find some interesting properties in the graph such as the shortest path between two points or the existence of cycles. We shall see in later chapters that a lot of problems can be modeled as graphs. The graphs can then be very intuitively analyzed to provide important insights.

A directed acyclic graph or a DAG is a special type of graph where the edges are directed. They are like a city with one-way streets.

Let us explain the difference between the program order and the data dependence order. Consider Figure 2.18(a) and (b). Figure 2.18(a) shows a set of instructions arranged in program order. The arrows in Figure 2.18(b) indicate the dependences where the destination of the arrow needs to execute after the source. The same relationships are visualized in Figure 2.18(b) as a directed acyclic graph or DAG (also see Definition 11). In the DAG (Figure 2.18(b)) two instructions can be executed simultaneously only if there is no dependence between them. There is a dependence in the DAG between instructions A and B only if there is a path from A to B.

We can clearly see that in an out-of-order machine, it is much easier to find two instructions that can be executed simultaneously as compared to an in-order machine where we are restricted to strictly follow program order.

2.3.3 Basics of an Out-of-order Machine

Let us now try to build an out-of-order machine from scratch. What should the first stage be? Without doubt, in the first stage we need to fetch instructions from memory. To be specific, we fetch instructions from the instruction cache, which is the highest level in the memory hierarchy. Recall that if there is a miss in the instruction cache, then we need to access the lower levels of the memory hierarchy such as the L2 and L3 caches. After fetching the instruction, we need to decode it.

The process of fetching and decoding instructions needs to be done in program order because at this point we do not know the details of the instruction. As a result, we have no idea about the data dependence order at this point.

We have till now discussed two main stages: fetch and decode (see Figure 2.19). Note that for increasing performance we can further pipeline these stages.

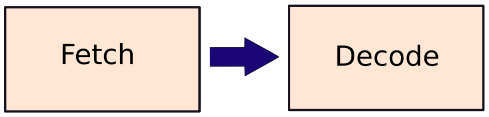
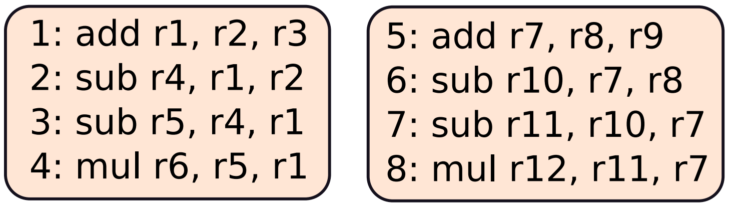


Figure 2.19: The front-end of the OOO pipeline (fetch and decode stages)

Once we get the instructions decoded we need to proceed to find dependences across instructions and find the sets of instructions that we can execute simultaneously. Let us consider an example, and try to prove that the more instructions we can simultaneously look at, the more ILP we shall find.



```
1: add r1, r2, r3
2: sub r4, r1, r2
3: sub r5, r4, r1
4: mul r6, r5, r1
```

```
5: add r7, r8, r9
6: sub r10, r7, r8
7: sub r11, r10, r7
8: mul r12, r11, r7
```

Figure 2.20: Two sets of 4 instructions with dependences between them

Consider Figure 2.20. Here, if we consider the first set of 4 instructions, the maximum amount of ILP is 1. There is a chain of dependence. However, if we consider all 8 instructions, we find that we can execute two instructions simultaneously: *1* and *5*, *2* and *6*, *3* and *7*, *4* and *8*. The larger is this pool of instructions, the higher is the expected ILP.

Now to create such a pool of instructions, do we fetch 8 instructions in one go? This is not possible because our fetch bandwidth (number of instructions that can be fetched simultaneously) is limited. We can at best fetch 4 or 8 instructions at once. A promising solution is to maintain a pool of instructions after the decode stage. Instructions that pass through the decode stage can be added to the pool. We can dynamically keep scanning this pool and then try to find instructions whose operands are ready and do not have dependences between each other. If the pool is large enough, there are high chances that we will always find more than one instruction to execute per cycle, and we can thus sustain an IPC of more than 1.

Our pipeline at the moment can be visualized as shown in Figure 2.21. Instructions pass through the fetch and decode stages. They then enter an instruction pool. We then choose sets of ready instructions without dependences between each other to execute in parallel. Note that a *ready* instruction is one whose operands have been computed and are available. The instructions then pass to the execution units, and finally write back their results. It is possible that an instruction might wait in this pool for a long time. It will wait till all the instructions that produce its source operands complete their execution. Alternatively, it is also possible that an instruction leaves the pool immediately because all of its inputs are available.

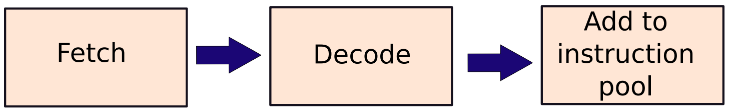


Figure 2.21: A part of the OOO pipeline with three stages

This instruction pool is referred to as the *instruction window* in computer architecture parlance. We shall refrain from formally defining this term now. We will have a lot of opportunity to look into the design of the instruction window later.

Ideally, the instruction window should be as large as possible. In this case, we can maximize the ILP because the larger is the instruction window higher are the chances of finding sets of ready instructions that do not have dependences between each other. But how large can the instruction window be? In modern processors, the instruction window contains somewhere between 64-256 entries. Note that it cannot be very small because the ILP needs to maximized, and it cannot be very large because in that case it will become a very slow structure and will consume a lot of power.

As a memory based structure such as the instruction window increases in size, it gets slower, and consumes more power. An instruction window is typically made of SRAM memory cells, and thus we cannot make it arbitrarily large.

Branch Prediction

Assume an instruction window with 128 entries (1 entry per instruction). We would ideally want to find as many instructions as possible to execute in parallel from this window. Thus, it is best if this window is close to being full almost all the time. Sadly, branch instructions have the potential of spoiling the party. In most programs, branch instructions are fairly frequent, and in most programs 1 in 5 instructions are branches. A window of 100+ instructions will have at least 20 branches.

What do we do about them? In an in-order pipeline we adopted several strategies. The simplest strategy was to stall the pipeline till the outcome of the branch was known. This is not possible in an out-of-order pipeline. It might take more than 10-20 cycles (at least) to know the outcome of a branch. We will never be able to fill our window if we adopt this strategy. The second strategy was to assume that the branch was not taken and proceed. If this assumption (not taken prediction) was found to be wrong, then we canceled or nullified the instructions that were on the wrong path (see Definition 12 for a more generic definition). This is also not possible in our setup because the order of execution need not be consistent with the program order. Hence, in a sequence of 20 branches, we will have a lot of mispredictions, and we will end up canceling a lot of instructions because they will be on the wrong path. Note that the moment we have a misprediction, the rest of the instructions are pretty much useless.

Definition 12
Instructions that would have been executed if a branch had an outcome that is different from its real outcome, are said to be on the wrong path. *For example, instructions succeeding a branch instruction in the program are on the wrong path if the branch is taken. Valid instructions are said to be on the* correct path.

To avoid this, we need a very accurate method of predicting branches. If we can accurately predict the outcome of **all** the branch instructions in the window, and the targets of the taken branches in the window, then we can ensure that it has a lot of instructions that are on the correct path most of the time. This will help in maximizing the ILP.

Let us get a quick idea of what the branch prediction accuracy should be. Assume that one in five instructions is a branch. To keep things simple, let the probability of misprediction of any given branch be p, and let us assume that the branch outcomes are independent of each other. If we consider n instructions, then the number of branches is $n/5$. The probability, P_n, of mispredicting at least one branch in a sequence of n instructions is shown in Equation 2.3. It can be derived as follows. The probability of predicting any given branch instruction correctly is $(1-p)$; hence, the probability of predicting all $n/5$ branches correctly is $(1-p)^(n/5)$. The probability of at least a single misprediction is thus 1 minus this number.

$$P_n = 1 - (1-p)^{n/5} \tag{2.3}$$

A single misprediction in a large instruction window is rather lethal because we may end up flushing (removing from the pipeline) a large number of instructions. A lot of work might get wasted. Hence, we are interested in the probability P_n. Alternatively, we can interpret the probability $1 - P_n$ as the probability of all the $n/5$ branches being predicted correctly.

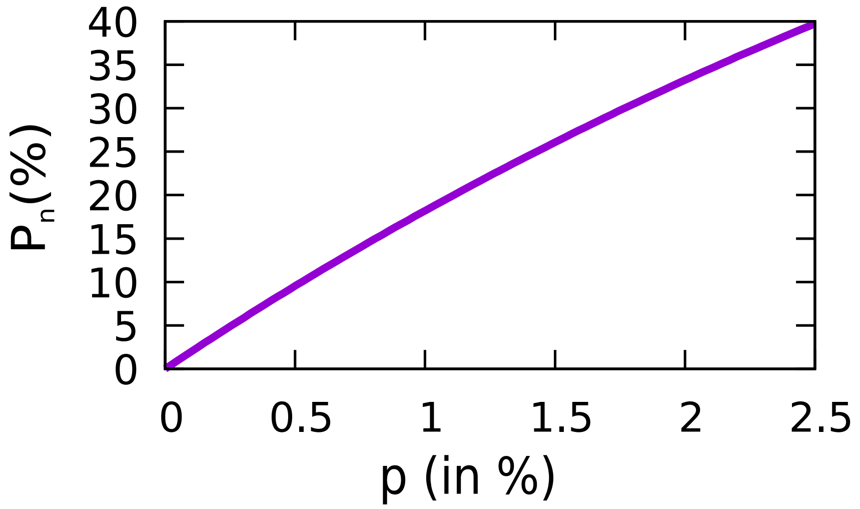


Figure 2.22: P_n versus p ($n = 100$)

Consider the results shown in Figure 2.22 for $n = 100$. We observe that even if the probability of a correct prediction is 99% ($p = 1\%$), there is a 17% chance that we shall have at least one branch misprediction in a sequence of 100 instructions. In that case, we will not be able to fill the instruction window. Even if the branch prediction rate (per branch) is as good as 99.5% (p=0.5%) we still have a roughly 7% probability of having at least a single misprediction. This gives us an idea of the kind of accuracies we need to ensure for having a window full of 100 instructions to be completely free of mispredicted branches. The per-branch prediction accuracy should be more than 99%. Creating a branch predictor that is 99% accurate is a very challenging task. We shall delve into this topic in Chapter 3.

Managing Anti and Output Dependences

The kind of dependences that we have been considering up till now are called real dependences, flow dependences, or RAW (read after write) dependences. This is because if we have, $A \rightarrow B$, then B cannot execute before A because it needs the results that A has generated.

Let us now consider some other classes of dependences that also need to be considered. Consider the following code snippet.

```
1 add r5, r1, r6
2 add r1, r2, r3
```

In this case instruction *2* cannot execute before instruction *1* because it will get the wrong value of $r1$. In the first instruction we are reading from $r1$, and in the second instruction we are writing to $r1$.

We have a write after read (WAR) dependence here. This is also called an *anti* dependence. We will use the $\xrightarrow{a}$ arrow to represent it. We thus have, *1* $\xrightarrow{a}$ *2*.

We can have another dependence of the following form.

```
add r1, r2, r3
add r1, r4, r5
```

In this case both the instructions are writing to the register $r1$. Here also instruction *2* cannot execute before instruction *1*. If this happens we will end up writing the wrong value to register $r1$. This type of dependence is known as an output dependence. It is alternatively called a write after write (WAW) dependence. We will use the $\xrightarrow{o}$ arrow to represent it. We have, *1* $\xrightarrow{o}$ *2*.

Definition 13
We have three kinds of data dependences between instructions in a program. Assume instruction B is after instruction A in program order.

- *A RAW dependence between instructions A and B means that A writes a value to a register that B reads from. It is represented as* $A \rightarrow B$.
- *A WAR dependence between instructions A and B means that B writes to a register that A reads from. It is represented as* $A \xrightarrow{a} B$.
- *A WAW dependence between instructions A and B means that both the instructions A and B write to the same register r, without any intervening writes to r. It is represented as* $A \xrightarrow{o} B$.

Register Renaming

It should be noted that WAW and WAR dependences are not real dependences. They are simply there because we have a finite number of registers.

Consider the following code snippet.

```
add r1, r2, r3
add r2, r5, r6
```

In this case, we cannot reorder instructions *1* and *2* because instruction *2* writes to $r2$, which is a source register for instruction *1*. This WAR hazard is only arising because we are using $r2$ as the destination register for instruction *2*. If we instead use another register in the place of $r2$ (not used before), we shall not have a WAR hazard.

Now, in practice the number of registers is limited; this number is also much lower than the number of entries in the instruction window. Thus, there will always be cases where we need to reuse registers, and we shall consequently have a lot of WAR and WAW hazards. Let us propose a solution to take care of this problem.

Let us call our traditional registers $(r1, r2, \ldots)$ as *architectural registers*. Let us also define a new set of registers $(p1, p2, \ldots)$ as *physical registers*. Let the physical registers be completely internal to the processor and be invisible to the compiler or the assembly language programmer. Furthermore, let us have many more physical registers than architectural registers. If we can map the architectural registers in an instruction to a set of physical registers, then we can possibly get rid of WAR and WAW hazards. This process is known as *renaming*. It will be explained in detail in Section 4.1. Right now, the important point to remember is that the process of renaming modifies an instruction such that its source and destination registers get converted from regular architectural registers to physical registers.

We need to ensure that the logic of the program does not change, if we convert our architectural registers into physical registers. Let us explain with an example. Assume that we already know that the data for the architectural registers $r1$ and $r2$ is saved in the physical registers $p1$ and $p2$ respectively. We can then take a piece of assembly code and convert it into equivalent assembly code that uses physical registers as follows.

With arch. registers

```
1 add r3, r1, r2
2 add r4, r3, 2
3 sub r3, r1, 1
```

With physical registers

```
1 add p3, p1, p2
2 add p4, p3, 2
3 sub p5, p1, 1
```

Let us consider the statements in order. In the first instruction we already know that $r1$ is mapped to $p1$ and $r2$ is mapped to $p2$. Let us use the $\leftrightarrow$ operator to indicate a mapping. We thus have $r1 \leftrightarrow p1$ and $r2 \leftrightarrow p2$. We map the architectural register $r3$ to $p3$. We thus convert the instruction *add r3, r1, r2* to *add p3, p1, p2*. For the second instruction, we already know that $r3$ is mapped to $p3$. We need to create a new mapping for $r4$. Let us map $r4$ to $p4$. We thus convert the instruction *add r4, r3, 2* to *add p4, p3, 2*. By using the same mapping for $r3$ ($r3 \leftrightarrow p3$), we reaffirm the fact that there is a RAW dependence between the first and second instructions. Recall that we are not allowed to change the logic of the program by creating such mappings. The only reason for mapping architectural registers to physical registers is to remove WAR and WAW dependences.

This is exactly what we have here. We have a WAW dependence between instructions *1* and *3*, and a WAR dependence between instructions *2* and *3*. We want to execute instructions *1* and *3*, or *2* and *3* in parallel if possible. This can be done by creating a new avatar for $r3$ in instruction *3*. Let us map it to a new physical register, $p5$. For the input operands of instruction *3*, we can use the earlier mappings. The only register input operand is $r1$, which has been mapped to $p1$. Thus *sub r3, r1, 1* becomes *sub p5, p1, 1*.

Now, we need to convince ourselves that the program with physical registers actually works, and does not change the logic of the original program with architectural registers. Consider the original program. The first instruction writes its result to $r3$ and then the second instruction needs to get this value. We have achieved this by using the same mapping for $r3$ in instructions *1* and *2* ($r3 \leftrightarrow p3$). Instruction *3* overwrites the value of $r3$. Any subsequent instruction will read the value of $r3$ written by instruction *3* or a newer value. Since instruction *3* does not read any value written by *1* and *2*, it does not have any RAW dependences, and thus it should be possible to execute it in parallel. However, there is a risk of it polluting the value of $r3$ if it executes and writes back its result before *1* and *2*. We thus create a new mapping for $r3$ ($r3 \leftrightarrow p5$). Now, there are no WAR or WAW dependences between instruction *3* and the previous instructions (*1* and *2*). We request the reader to look at both the code sequences thoroughly and convince herself that the renamed code (after register mapping) is correct.

The dependences in the original program were $1 \rightarrow 2, 1 \xrightarrow{o} 3$, and $2 \xrightarrow{a} 3$. After renaming (mapping registers) the only dependence is $1 \rightarrow 2$. As we see, only the real (RAW) dependences are left in the renamed code. All WAW and WAR dependences have been removed! This increases the ILP, and can translate to a higher IPC if we start executing multiple instructions in the same cycle. Note that if we execute instructions *1* and *3* together in the first cycle, and then execute instruction *2*, we would be executing instructions **out of order**. Since we have removed WAR and WAW dependences, the execution will be correct.

Renaming can be done either in hardware or by the compiler. The latter is not a preferred option because in this case the compiler needs to be aware of the number of physical registers a processor has. We do not want to expose such internal details to the compiler. This is because across processor versions the number of internal physical registers can keep changing. We want to fix the ISA and then make as many changes as we want within the processor unbeknownst to the programmer and the compiler. As a result, almost all processors have a renaming unit in hardware that maps architectural registers to physical registers. The modified code does not have WAR and WAW dependences and thus has more ILP.

At this point our pipeline looks as shown in Figure 2.23.

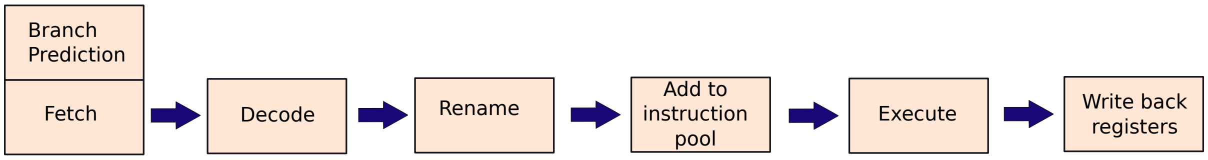


Figure 2.23: The OOO pipeline (as explained till now)

Ensuring Precise Exceptions

After renaming, we can execute multiple instructions out of order, and thus ensure a higher IPC. However, we shall have many problems in correctness particularly if we have an interrupt or exception.

Let us revisit this problem in the context of a traditional single-issue in-order pipeline. Now, recall that we define an *interrupt* as an external event such as a message from an I/O device, and an *exception* as an internal event such as an instruction dividing a number by 0. Typically, during the execution of a long program the processor receives thousands or even millions of interrupts. It is necessary to stop the execution of the program, handle the interrupt/exception, and then come back and restart the program. We have several options here. For some types of events, we can restart the program at the instruction that immediately succeeds the last instruction whose results were written back, in the case of a page fault we can re-execute the faulting instruction, or we can restart the program at a different point. Regardless of the nature of the event, the correctness of the execution needs to be guaranteed.

Let the latest instruction (in program order) to write back its results to either the memory or register file be instruction I. Let us say that an instruction *completes* when it writes back its results to either the memory or the register file. To ensure correctness we need to ensure that the following conditions are met:

1. All the instructions before I in program order need to complete before the interrupt/exception handler begins to execute.

2. No instruction after I in program order should have completed at this point.

Let us intuitively explain the meaning of this definition. Assume that a user is looking at the execution of a program from outside. She does not care about the details of the processor. It can be a single-cycle processor, or a complex pipelined out-of-order processor. All that she wants to see is that the processor can stop a program $\mathcal{P}$ at any instruction in response to an event. It can then handle the event, run other programs, and then start $\mathcal{P}$ exactly at the point at which it had left it. Such an exception is called a *precise exception*. Note that this definition treats exceptions and interrupts interchangeably.

All processors support precise exceptions. Unless we have precise exceptions there will be serious violations in correctness. Consider this piece of code:

```
mov r4, 1
div r7, r5, r6
sub r4, r4, 1
```

Assume that after instruction *2*, the processor stops the program ($\mathcal{P}$) and proceeds to handle an interrupt. When we come back we expect $r4$ to be 1, and instruction *3* to have not executed. Assume that precise exceptions are not supported, and the processor has already executed instruction *3* when $\mathcal{P}$ returns. We will incorrectly execute instruction *3* twice.

Given the fact that interrupts can happen any time, we need to ensure that the processor can suspend the execution of the current program very quickly, and jump to an interrupt handler. The interrupts/exceptions thus **have to be precise**.

To ensure precise exceptions in simple in-order pipelines, we allow the instructions in the memory access and write-back stages to complete. Then, we nullify the rest of the instructions in the pipeline. This gives us a clean point at which we can restart the program's execution. Given that we are executing instructions in program order, creating such a boundary between complete and incomplete instructions is very easy.

However, ensuring precise exceptions in an out-of-order pipeline is an entirely different issue. In this example, if we complete instructions *1* and *3* before completing instruction *2*, then we will not be able to ensure precise exceptions if an interrupt arrives before *2* is completed. In fact the whole idea of out-of-order machines is to issue instructions in an order that is possibly not consistent with the program order. It appears that the goal of ensuring precise exceptions and out-of-order execution are completely at odds with each other. We can only achieve one at the expense of the other. This situation is clearly undesirable; however, at this point we do not have a method to ensure both.

We need to think of an innovative solution to solve this problem without sacrificing the benefits of out-of-order execution. We need to somehow ensure that instructions write their results to the register file or the memory system in program order. This process is known as *committing* an instruction. We need a dedicated commit engine in hardware that ensures precise exceptions. It should appear to the outside world that instructions are committing in program order, even though they might be executing and computing their results out of order.

Let us take a look at the current shape of our out-of-order pipeline in Figure 2.24. Henceforth, we shall mostly use the acronym OOO to refer to an out-of-order pipeline.

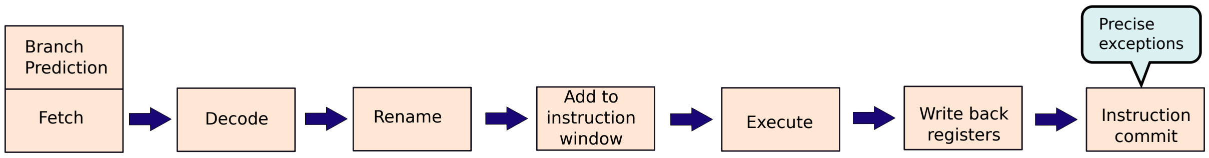


Figure 2.24: The OOO pipeline

We shall look at each of the stages in detail in the following chapters. The plan for the rest of the chapters in this area is as follows.

1. In Chapter 3, we shall look at branch prediction, and the fetch/decode logic.

2. In Chapter 4, we shall describe the rename unit, and the process of issuing instructions to execution units from the instruction window.

3. In Chapter 5, we shall describe alternative implementations of OOO pipelines.

2.4 Summary and Further Reading

2.4.1 Summary

Summary 1

1. *A traditional 5-stage in-order pipeline processes instructions in program order. A later instruction never overtakes an earlier instruction in any stage of the pipeline.*
2. *It consists of 5 stages: Instruction Fetch (IF), Operand Fetch and Decode (OF), Execute (EX), Memory Access (MA), and Register Write-back (RW).*
3. *In-order pipelines primarily suffer from two kinds of stalls: stalls due to data hazards, and stalls due to control hazards. The need for such stalls are required (also called pipeline interlocks) to ensure that instructions read the correct values for their operands.*
4. *Almost all stalls can be eliminated by using forwarding techniques in such pipelines. Here, we pass results from a later stage to an earlier stage such that stalls are not required. The only exception is the load-use hazard, where we need to stall the pipeline for one cycle.*
5. *Control hazards typically require us to nullify the instructions in the IF and OF stages. Otherwise, we will end up executing instructions on the wrong path. One way to completely eliminate the need for doing this is by using the delayed branch mechanism where we assume that the two instructions after the branch instruction are on the correct path. This is ensured by the compiler that takes independent instructions before the branch and puts them just after the branch. The other way is to predict the direction of the branch.*
6. *The main problem with in-order pipelines is that their IPC (instructions per cycle) is less than 1 (due to stalls). Nevertheless, the primary advantage of pipelining is that we can increase the frequency because of the lesser amount of work we need to do per clock period as compared to a single-cycle pipeline.*
7. *The relationship between performance(P), IPC, frequency(f), and the number of instructions is given by the performance equation.*

$$P \propto \frac{IPC \times f}{\#insts}$$

8. *Due to limitations imposed by power consumption, it is very hard to increase the frequency further. We need to increase the IPC to get higher performance. Even though a multi-issue in-order pipeline can increase the IPC to a number above 1, it has limitations in terms of complexity. Additionally, it suffers from the* convoy effect*, where one slow instruction can delay all the instructions after it.*
9. *The only other option is to create an out-of-order(OOO) pipeline that can execute instructions in an order that is not consistent with the program order. We can additionally fetch and execute multiple instructions per cycle to further increase the IPC. This is known as* superscalar *execution.*
10. *Instructions in such OOO processors suffer from three kinds of data hazards: RAW (read after write), WAW (write after write), and WAR (write after read). These hazards prevent us from reordering instructions unless additional steps are taken.*

11. *The key steps in an OOO processor are as follows:*
    (a) *We need an extremely accurate branch predictor such that we can create large sequences of instructions that are on the correct path. This is a part of the fetch stage. Subsequently, we decode the instructions and send them to the rename unit.*
    (b) *We rename instructions by replacing architectural registers with physical registers such that we can get rid of WAR and WAW hazards in the code.*
    (c) *The only dependences that we have between renamed instructions are RAW dependences.*
    (d) *We store a large set(50-100) of renamed instructions in the instruction window. In each cycle we try to find a set of instructions that are ready (operands are available in the register file), and are mutually independent. These instructions are sent for execution.*
    (e) *Finally, we write back the results, and remove the instruction from the pipeline (commit).*
12. *We need to ensure that precise exceptions are guaranteed, which means that it is possible to pause a running program at any point, run another program, and again resume it at the same point seamlessly.*

2.4.2 Further Reading

For readers who are unfamiliar with the details of in-order pipelines, and would like to read further, we would like to recommend the book by Sarangi [Sarangi, 2015]. Subsequently, it is necessary to get a perspective of commercial processors that use in-order pipelines. Readers should refer to Saini [Saini, 1993] and Alpert et al. [Alpert and Avnon, 1993] for understanding the design of the Intel® Pentium® processor. After appreciating classical architectures, readers can move on to read about contemporary in-order ARM processors ARM® Cortex®-M3 [Yiu, 2009] and ARM® Cortex®-A8 [Williamson, 2007], and the Intel Atom® processor [Halfhill, 2008].

Exercises

Ex. 1 — Draw a pipeline diagram for the following code sequence. Assume we only have interlocks and no forwarding.

```
add r1, r2, r3
add r4, r1, r6
add r5, r6, r7
add r7, r8, r9
ld  r2, 8[r1]
sub r1, r2, r2
```

Ex. 2 — Solve Exercise 1 assuming a pipeline that has forwarding enabled.

Ex. 3 — What is the performance equation? On the basis of that can we explain why a pipelined processor is faster than an equivalent non-pipelined processor? Justify your answer.

Ex. 4 — Draw a pipeline diagram for the following code sequence. Assume we have interlocks, no forwarding, and no support for delayed branches. Assume that the branch is taken.

```
add r1, r2, r3
add r7, r8, r9
beq .loop
...
.loop:
sub r5, r6, r7
mul r8, r8, r1
```

Ex. 5 — Solve Exercise 4 for the case when we have delayed branches. Rest of the assumptions remain the same.

* **Ex. 6** — Derive a set of conditions for creating instruction bundles in multi-issue in-order pipelines.

Ex. 7 — What are the advantages of OOO pipelines over in-order pipelines?

Ex. 8 — What is a superior mode of execution: execution in program order or in data dependence order? In a data dependence order, instruction B needs to execute after instruction A, if it reads a value that A produces. There are no other restrictions.

Ex. 9 — How does renaming remove WAR and WAW dependences? If we had a very large number of registers, would renaming still be required?

Ex. 10 — Why cannot RAW dependences be removed with renaming?

Ex. 11 — Why are precise exceptions required? Are they required in processors where we shall never have any interrupts or any other exceptional conditions where the intervention of another program is necessary?

Ex. 12 — In the case of an ISA where there are instructions that access the memory in the OF stage, what are the forwarding paths and dependences? Can we remove all dependences using forwarding?

Ex. 13 — Why is there a need for pipeline registers? Why can't we simply forward the instruction packet to the next stage and process it.

Ex. 14 — What will happen if we try to create a 1000-stage pipeline? Is it a good idea?