10
Principles of Pipelining

10.1 A Pipelined Processor

Let us quickly review where, we are.

Way Point 7

1. *We have designed a processor with five main stages – IF, OF, EX, MA, and RW.*
2. *We have designed a detailed data path and control path for the hardwired implementation of our processor.*
3. *We introduced a microprogram based implementation of our processor in Section 9.4, and we designed a detailed data path and control path for it.*

Now, our aim is to make our processor fast and efficient. For this we focus on the hardwired implementation of our processor. We exclude microprogrammed processors from our discussion because we are explicitly looking for high performance, and flexibility/ reconfigurability are not important criteria for us in this section. Let us begin by pointing out some problems with the design of the hardwired processor as presented in Section 9.2.

10.1.1 The Notion of Pipelining

Issues with a Single-Cycle Processor

We assumed that our hardwired processor presented in Section 9.2 takes a single cycle to fetch, execute, and write the results of an instruction to either the register file or memory. At an electrical level, this is achieved by signals flowing from the fetch unit to ultimately the register writeback unit via other units. It takes time for electrical signals to propagate from one unit to the other.

For example, it takes some time to fetch an instruction from the instruction memory. Then it takes time to read values from the register file, and to compute the results with the ALU. Memory access, and writing the results back to the register file, are also fairly time taking operations. We need to wait for all of these individual sub-operations to complete, before we can begin processing the next instruction. In other words, this means that there is a significant amount of idleness in our circuit. When the operand fetch unit is doing its job, all other units are idle. Likewise, when the ALUs are active, all the other units are inactive. If we assume that each of the five stages (IF,OF,EX,MA,RW) takes the same amount of time, then at any instant, about 80% of our circuit is idle! This represents a waste in computational power, and idling resources is definitely not a good idea.

If we can find a method to keep all the units of a chip busy, then we can increase the rate at which we execute instructions.

10.1.2 Overview of Pipelining

Let us try to find an analogy to the problem of idleness in a simple single-cycle processor as we just discussed. Let us go back to our original example of the car factory. If we assume, that we start making a car, after the previous car has been completely manufactured, then we have a similar problem. When we are assembling the engine of a car, the paint shop is idle. Likewise, when we are painting a car, the engine shop is idle. Clearly, car factories cannot operate this way. They thus typically overlap the manufacturing stages of different cars. For example, when car A is in the paint shop, car B is in the engine shop. Subsequently, these cars move to the next stage of manufacturing and another new car enters the assembly line.

We can do something very similar here. When one instruction is in the EX stage, the next instruction can be in the OF stage, and the subsequent instruction can be in the IF stage. In fact, if we have 5 stages in our processor, where we simplistically assume that each stage roughly takes the same amount of time, we can assume that we have 5 instructions simultaneously being processed at the same time. Each instruction undergoes processing in a different unit of the processor. Similar to a car in an assembly line, an instruction moves from stage to stage in the processor. This strategy ensures that we do not have any idle units in our processor because all the different units in a processor are busy at any point in time.

In this scheme, the life cycle of an instruction is as follows. It enters the IF stage in cycle n, enters the OF stage in cycle $n+1$, EX stage in cycle $n+2$, MA stage in cycle $n+3$, and finally it finishes its execution in the RW stage in cycle $n+4$. This strategy is known as *pipelining*, and a processor that implements pipelining is known as a *pipelined* processor. The sequence of five stages (IF, OF, EX, MA, RW) conceptually laid out one after the other is known as the *pipeline* (similar to the car assembly line). Figure 10.1 shows the organization of a pipelined data path.

In Figure 10.1, we have divided the data path into five stages, where each stage processes a separate instruction. In the next cycle, each instruction passes on to the next stage as shown in the figure.

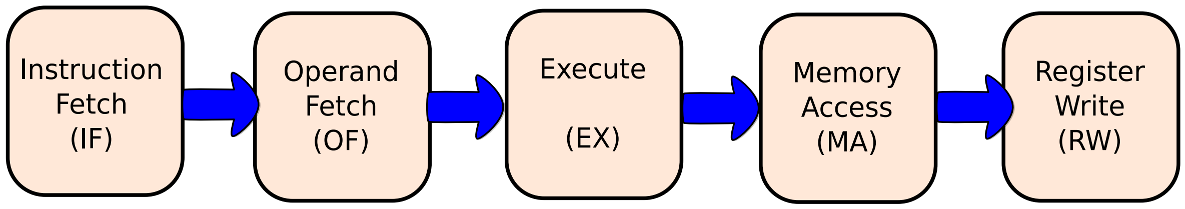


Figure 10.1: A pipelined data path

Definition 65
The notion of dividing a processor into a set of stages where the stages are ordered one after the other, and simultaneously process a set of instructions by assigning an instruction to each stage, is known as pipelining*. The implicit assumption here is that it takes the same amount of time for each stage to complete its work. After this time quanta is over, each instruction moves to the subsequent stage.*

The conceptual layout of stages where one stage is laid out after the other is known as a pipeline*, and a processor that incorporates pipelining is known as a* pipelined *processor.*

10.1.3 Performance Benefits

Let us quantify the expected benefit in terms of performance of a pipelined processor. We shall take a deeper look into performance issues in Section 10.9. Here, we shall look at this topic briefly. Let us assume that it takes τ nanoseconds for an instruction to travel from the IF to RW stage of the pipeline in the worst case. The minimum value of the clock cycle is thus limited to τ nanoseconds for the case of a single cycle pipeline. This is because in every clock cycle we need to ensure that an instruction executes completely. Alternatively, this mean that every τ nanoseconds, we finish the execution of an instruction.

Now, let us consider the case of a pipelined processor. Here, we have been assuming that the stages are balanced. This means that it takes the same amount of time to execute each stage. Most of the time, processor designers try to achieve this goal to the maximum extent that is possible. We can thus divide τ by 5, and conclude that it takes $\tau/5$ nanoseconds to execute each stage. We can thus set the cycle time to $\tau/5$. After the end of a cycle, the instructions in each stage of the pipeline proceed to the next stage. The instruction in the RW stage moves out of the pipeline and finished its execution. Simultaneously, a new instruction enters the IF stage. This is graphically shown in Figure 10.2.

In the n^{th} cycle, we have five instructions (1-5) occupying the five stages of the pipeline. In the $(n+1)^{th}$ cycle each instruction progresses by 1 stage, and instruction 6 enters the pipeline. This pattern continues.

The noteworthy point is that we are finishing the execution of a new instruction, every $\tau/5$ nanoseconds. As compared to a single-cycle processor that finishes the execution of a new instruction every τ nanoseconds, the instruction throughput is 5 times higher for a pipelined

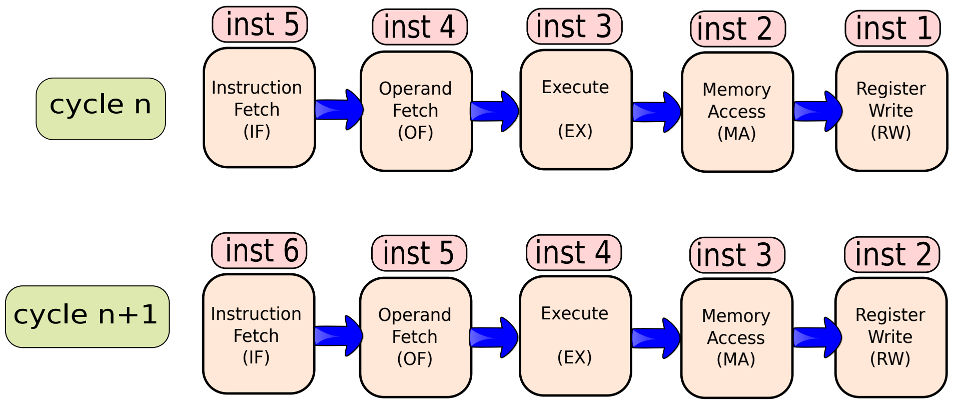


Figure 10.2: Instructions in the pipeline

processor. In a span of 1000 nanoseconds, a single cycle processor completes $1000/\tau$ instructions, whereas a pipelined processor completes $5000/\tau$ instructions, and is thus 5 times more efficient. Therefore, we observe a five-fold advantage with pipelining.

If we can obtain a five-fold advantage with a 5-stage pipeline, then by the same logic we should be able to obtain a 100-fold advantage with a 100-stage pipeline. In fact, we can keep on increasing the number of stages till a stage just contains one transistor. However, this is not the case, and there are fundamental limitations to the performance of a pipelined processor, as we shall show in the subsequent sections. It is not possible to arbitrarily increase the performance of a processor by increasing the number of pipeline stages. In fact, after a certain point, adding more stages is counterproductive.

10.2 Design of a Simple Pipeline

Let us now design a simple pipeline. Our main aim in this section is to split the data path of the single-cycle processor into five stages and ensure that five instructions can be processed concurrently (one instruction in each stage). We need to also ensure the seamless movement of instructions between the pipeline stages. Note that the problem of designing a pipeline in general is very complex, and we will explore some major nuances in the next few sections. For this section, let us not consider any dependences between instructions, or consider any correctness issues. We shall look at these issues in detail in Section 10.4. Let us reiterate that at the moment, we want to design a simple pipeline that needs to have the capability to process five instructions simultaneously, and ensure that they move to the subsequent stage every new cycle.

10.2.1 Splitting the Data Path

We have five distinct units in our data path, and all instructions traverse the units in the same order. These units are instruction fetch (IF), operand fetch (OF), execute (EX), memory access (MA), and the register write (RW) units. A layout of these five units in a pipelined fashion has already been shown in Figure 10.1. Let us now discuss the issue of splitting a data path in some more detail.

The reader needs to note that pipelining is a general concept, and any circuit can in principle be split into multiple parts and pipelined. There are however some rules that need to be followed. All the subparts of the circuit must preferably be distinct entities that have as few connections between them as possible. This is true in the case of our data path. All our units are distinct entities. The second is that all kinds of data must flow through the units in the same order, and lastly the work done by each unit should roughly be the same. This minimizes the idleness in the circuit. In our case, we have tried to follow all these rules. The reader needs to note that the *div* and *mod* operations are exceptions to this rule. They are in general, significantly slower, than add or multiply operations. They thus increase the maximum delay of the EX stage, and the pipeline consequently becomes unbalanced. Hence, most simple pipelined processors either refrain from implementing these instructions, or have specialized logic to deal with them. We shall show one solution for this problem in Section 10.6 that proposes to stall a pipeline till a division operation completes. Let us nevertheless continue to assume that all our pipeline stages are balanced.

10.2.2 Timing

Now, we need to design a method that ensures that instructions seamlessly proceed to the subsequent pipeline stage. We need a global mechanism that ensures that all the instructions proceed to the next stages simultaneously. We already have this global mechanism built in, and is nothing else, but the *clock*. We can have a protocol that for example, ensures that at the falling edge of the clock, all the instructions proceed to the subsequent stages.

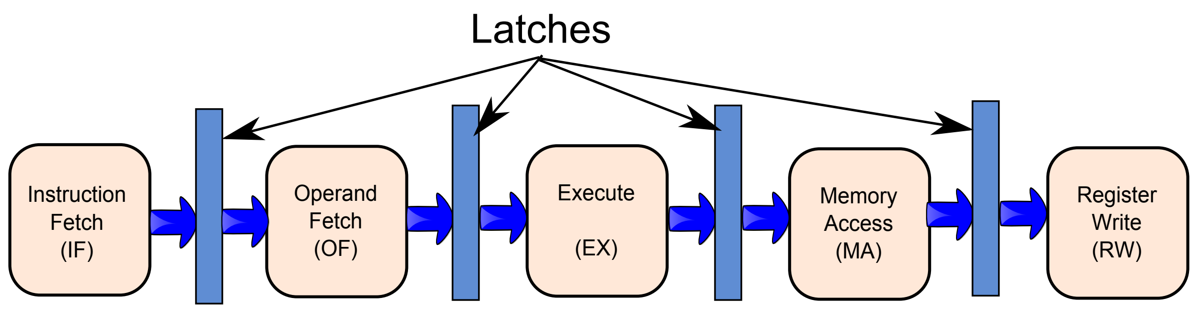


Figure 10.3: A pipelined data path with registers

Figure 10.3 shows a simple method to achieve this. We insert a register between two consecutive pipeline stages. Since we have five pipeline stages in our data path, we insert 4 registers. The four registers are named after their locations – IF-OF, OF-EX, EX-MA, and MA-RW. Each of these registers are called pipeline registers or pipeline latches. The reader needs to note that in this case, a *latch*, is actually referring to an edge triggered register. We shall use the terms interchangeably. All the pipeline registers are connected to a common clock, and read-write data at the same time.

Definition 66
A pipeline register *or a* pipeline latch *is a register that is added between two consecutive pipeline stages. All the registers are connected to the common clock, and help in seamlessly transferring instructions between pipeline stages.*

Let us explain with an example. When an instruction enters the pipeline, it enters the IF unit. At the end of the first cycle, it gets saved in the IF-OF register. At the beginning of the second cycle, it enters the OF stage, and then again at the end of the second cycle, it gets latched into the OF-EX register. This pattern continues till the instruction leaves the pipeline. The pipeline registers essentially transfer their inputs to the outputs at the end of a cycle (negative edge of the clock). Then the logic of the pipeline stage processes the instruction, and at the end of the cycle, the instruction gets transferred to the register of the subsequent pipeline stage. In this manner, an instruction hops between stages till it reaches the end of the pipeline.

10.2.3 The Instruction Packet

Let us now proceed to design our data path with pipeline stages and registers in some more detail. Up till now we have been maintaining that the instruction needs to be transferred between registers. Let us elaborate on the term "instruction". We actually mean an *instruction packet* here, which contains all the details regarding the instruction, along with all of its intermediate results and the control signals that it may require.

We need to create such an elaborate instruction packet because there are multiple instructions in the processor at the same time. We need to ensure that there is no overlap between the information required to process two different instructions. A clean way of designing this is to confine all the information required to process an instruction in a packet, and transfer the packet between pipeline registers every cycle. This mechanism also ensures that all the intermediate state required to process an instruction is removed after it leaves the pipeline.

What should an instruction packet contain? It needs to contain at the least, the PC and the contents of the instruction. It should also contain all the operands and control signals that are required by subsequent stages. The amount of information that needs to be stored in the instruction packet reduces as the instruction proceeds towards the last stage. For the sake of uniformity, we assume that all the pipeline registers have the same size, and are sized to hold the entire instruction packet. Some fields might not be used. However, this is a negligible overhead. Let us now proceed to design the data path of the pipeline. We shall use exactly the same design as we had used for the single-cycle processor. The only difference is that we add a register after each pipeline stage, other than the last stage, RW. Secondly, we add connections to transfer data in and out of the pipeline registers. Let us quickly take a look at each of the pipeline stages in our pipelined data path.

10.3 Pipeline Stages

10.3.1 IF Stage

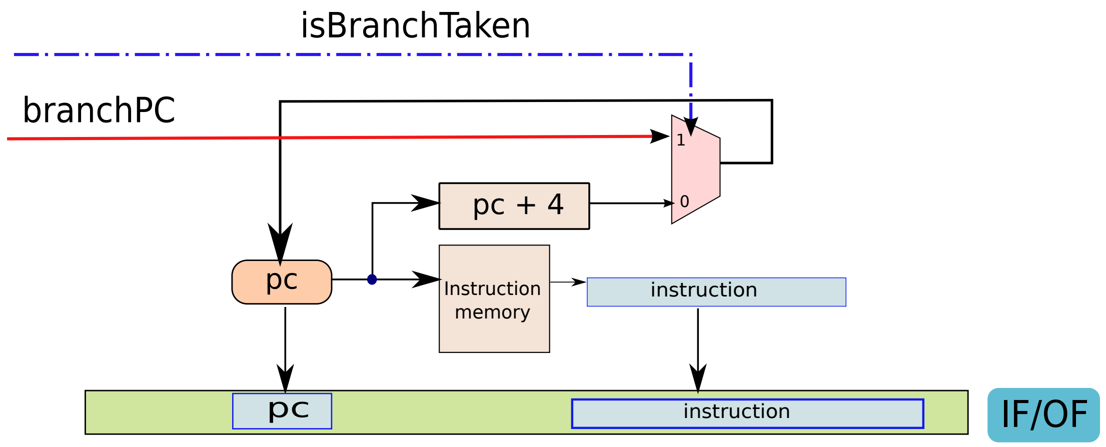


Figure 10.4: The IF stage in a pipelined processor

Figure 10.4 shows the IF stage augmented with a pipeline register. We save the value of the PC, and the contents of the instruction in the pipeline register. This is all the information that we need to carry forward to the subsequent stages of the pipeline. Other than this small change, we do not need to make any other change in this part of the data path.

10.3.2 OF Stage

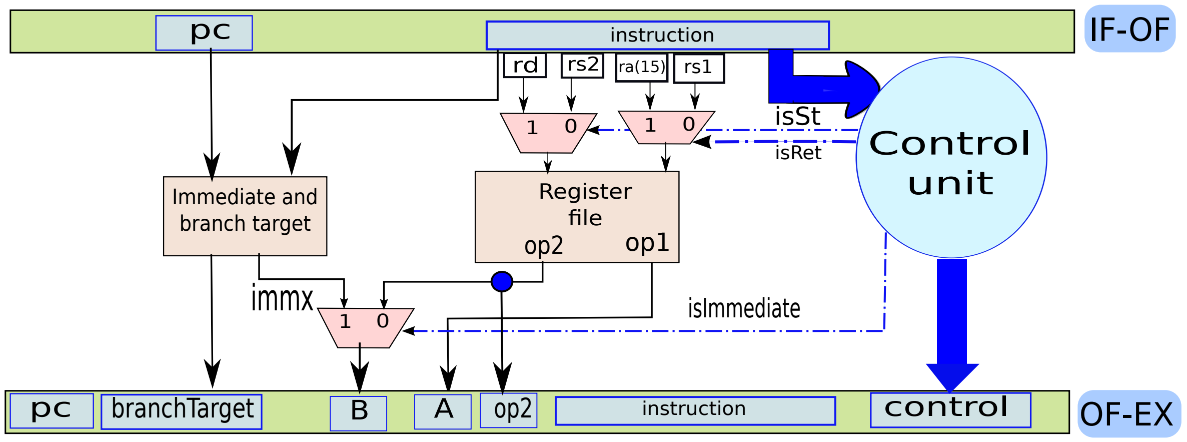


Figure 10.5: The OF stage in a pipelined processor

Figure 10.5 shows the design of the operand fetch stage. The only extra additions are the connections to the two pipeline registers IF-OF, and OF-EX. The stage starts out by extracting the fields rd (bits 23-26), $rs1$ (bits 19-22), $rs2$ (bits 15-18), and the immediate (bits 1-18) from the instruction. These are sent to the register file, the immediate and branch units. Additionally, we send the contents of the instruction to the control unit that generates all the control signals. Three of the control signals namely *isRet*, *isImmediate*, and *isSt* are used immediately. The rest of the control signals are for controlling multiplexers in subsequent stages of the pipeline. Hence, it is necessary for us to save them in the OF-EX pipeline register such that they can traverse the pipeline along with the instruction, and control the actions of different units accordingly. Therefore, we allocate some space within the instruction packet, and store all the control signals generated by the control unit (refer to the field *control* in Figure 10.5).

We need to carry all the intermediate results generated by the OF stage. In specific, the OF stage generates the $branchTarget$, both the inputs for the ALU (A, and B), and the value to be written to memory for a store instruction ($op2$). Thus, we allocate four fields in the instruction packet, and the OF-EX pipeline register to store this information as shown in Figure 10.5. Let us recall that the aim of designing the instruction packet was to have all the information required to process an instruction at one place. In accordance with this philosophy we have saved all the details of the instruction including its address, contents, intermediate results, and control signals in our pipeline registers.

10.3.3 EX Stage

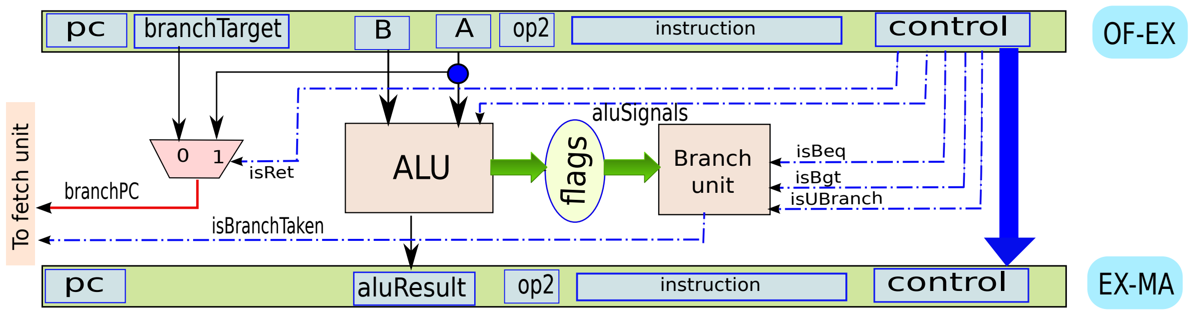


Figure 10.6: The EX stage in a pipelined processor

Let us now take a look at the EX stage in Figure 10.6. The ALU receives its inputs(A and B) from the OF-EX pipeline register. The results generated by this stage are the *aluResult* (result of the ALU operation), the final branch target, and the branch outcome. The branch outcome is 1, if the branch is taken, otherwise it is 0. The result of the ALU operation is added to the instruction packet, and saved in the EX-MA register. The EX-MA register also contains the rest of the fields of the instruction packet namely the PC, *instruction* (contents of the instruction), *control* signals, and the second operand read from the register file ($op2$).

For computing the final branch target, we need to choose between the branch target computed in the OF stage and the value of the return address register (possibly stored in A). The result of the choice is the final branch target(*branchPC*), and this is sent to the fetch unit. The branch unit computes the value of the signal, *isBranchTaken*. If it is 1, then the instruction is a branch, and it is taken. Otherwise, the fetch unit needs to use the default option of fetching the next PC.

10.3.4 MA Stage

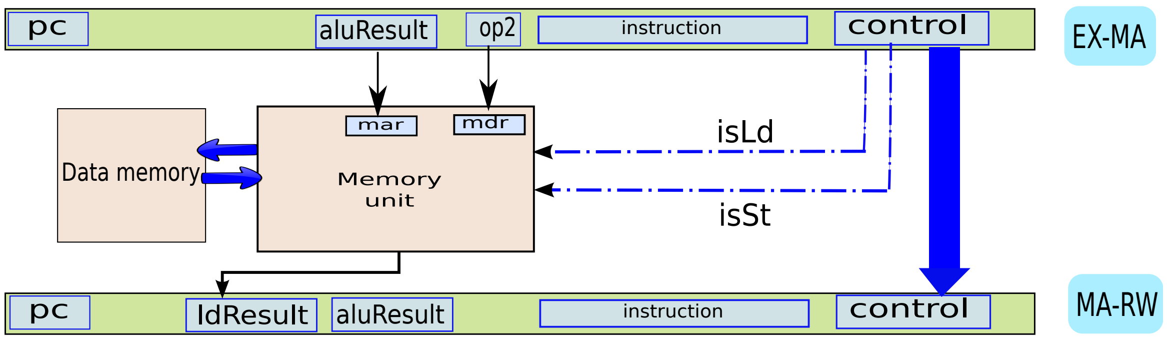


Figure 10.7: The MA stage in a pipelined processor

The MA stage is shown in Figure 10.7. The only operand that the load instruction uses is the result of the ALU, which contains the effective memory address. This is saved in the *aluResult* field of the EX-MA register. The data to be stored resides in the *rd* register. This value was read from the register file in the OF stage, and was stored in the *op*2 field of the instruction packet. In this stage, the *op*2 field is connected to the MDR (memory data register) register. The relevant control signals – *isLd* and *isSt* – are also a part of the instruction packet, and they are routed to the memory unit.

The only output of this stage is the result of the load instruction. This is saved in the *ldResult* field of the MA-RW register.

10.3.5 RW Stage

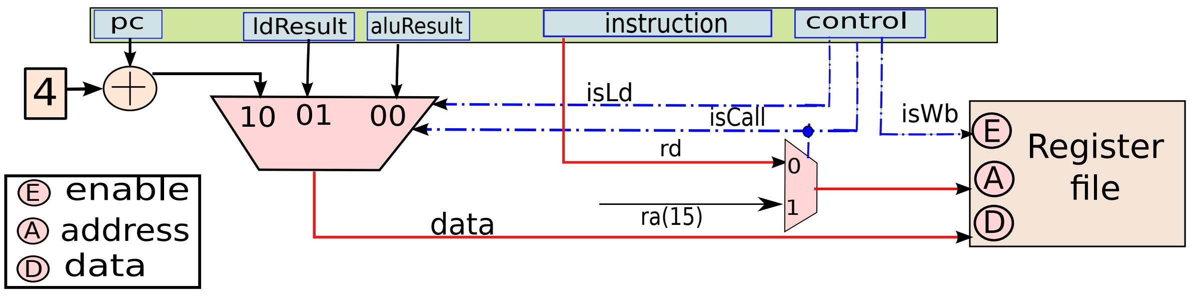


Figure 10.8: The RW stage in a pipelined processor

We need to lastly take a look at the RW stage in Figure 10.8. The inputs that it requires from the previous stages are the values of the ALU and load operations stored in the *aluResult* and *ldResult* fields respectively. These inputs along with the default next PC (current PC + 4) are connected to a multiplexer that chooses the value to be written back. The rest of the circuit is the same as that of the single-cycle processor. Note that there is no pipeline register at the end of the RW stage because it is the last stage in the pipeline.

10.3.6 Putting it All Together

Let us now summarize our discussion regarding the simple pipeline by showing our data path with the pipeline registers in Figure 10.9. The figure is undoubtedly complex. However, the reader has seen all the parts of this figure before and thus should not have a significant amount of difficulty in putting the different parts together. Nonetheless, we should note that the design of our processor has already become fairly complex, and the size of our diagrams have already reached one page !!! We do not want to introduce more complicated diagrams. The reader should note that up till now our aim was to introduce the entire circuit. However, we shall now introduce some degree of abstraction such that we can introduce more complicated features into our processor. Henceforth, we shall broadly concentrate on the logic of the pipeline, and not talk about the implementation in detail. We shall leave the implementation of the exact circuit as an exercise for the reader.

Figure 10.10 shows an abstraction of our pipeline data path. This figure prominently contains block diagrams for the different units and shows the four pipeline registers. We shall use this diagram as the baseline for our discussion on advanced pipelines. Recall that the first register operand can either be the $rs1$ field of the instruction, or it can be the return address register in the case of a *ret* instruction. A multiplexer to choose between ra and $rs1$ is a part of our baseline pipeline design, and for the sake of simplicity, we do not show it in the diagram. We assume that it is a part of the register file unit. Similarly, the multiplexer to choose the second register operand (between rd and $rs2$) is also assumed to be a part of the register file unit, and is thus not shown in the diagram. We only show the multiplexer that chooses the second operand (register or immediate).

10.4 Pipeline Hazards

In our simple pipeline discussed in Section 10.2, we were not concerned with correctness issues in the pipeline. We were simply concerned with designing the pipeline, and having the capability to process five instructions at the same time. Now, we want to take a look at correctness issues. Let us start out by introducing the pipeline diagram, which will prove to be a very useful tool in our analyses.

10.4.1 The Pipeline Diagram

We typically use a pipeline diagram to study the behavior of a pipeline. It shows the relationships between instructions, clock cycles, and the different stages of the pipeline. It can be used to study the nature of dependences across different instructions, and their execution in the pipeline.

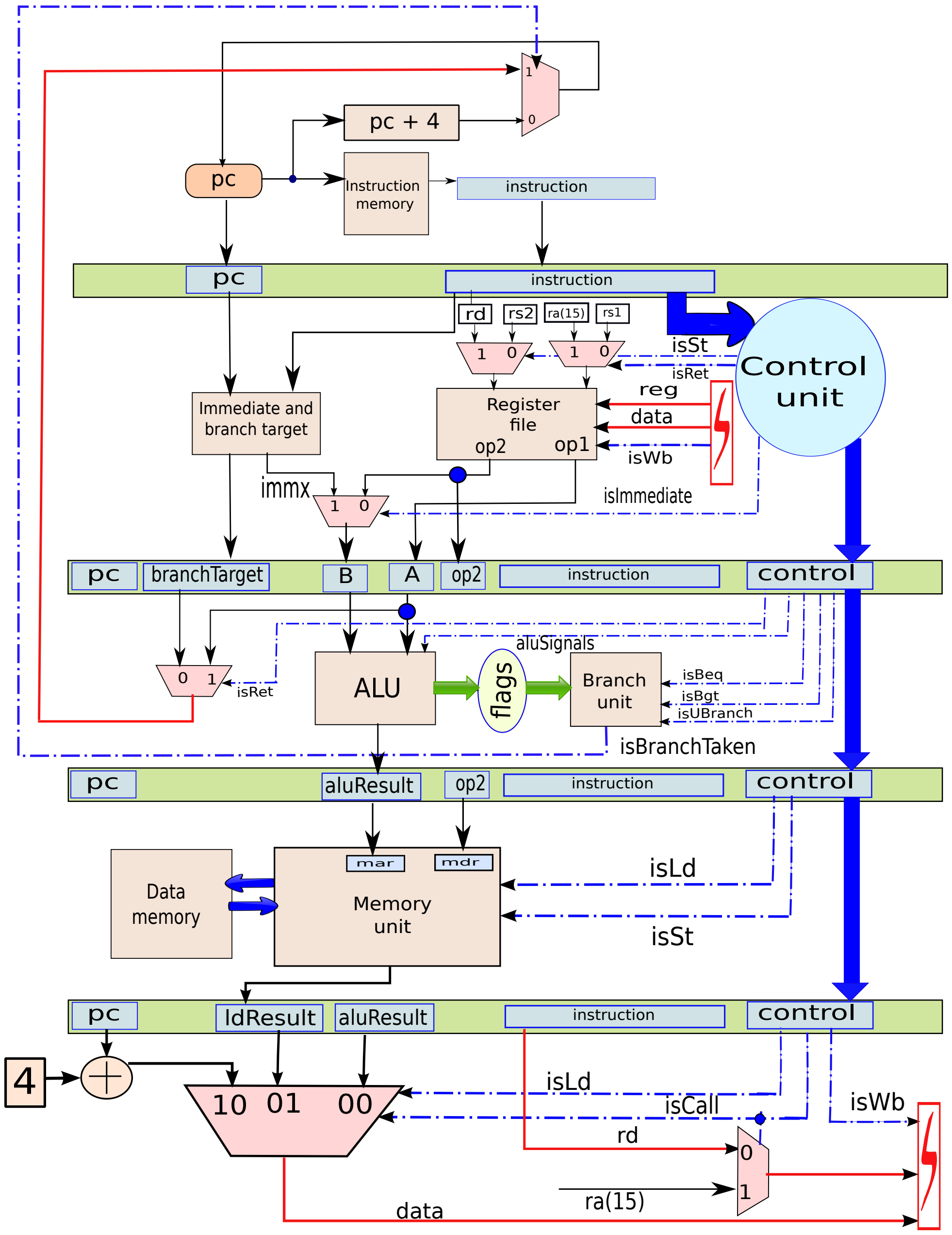


Figure 10.9: Pipelined data path

Figure 10.11 shows a pipeline diagram for three instructions as they proceed through the pipeline. Each row of the diagram corresponds to each pipeline stage. The columns correspond to clock cycles. In our sample code, we have three instructions that do not have any dependences between each other. We name these instructions – [1], [2], and [3] respectively. The earliest

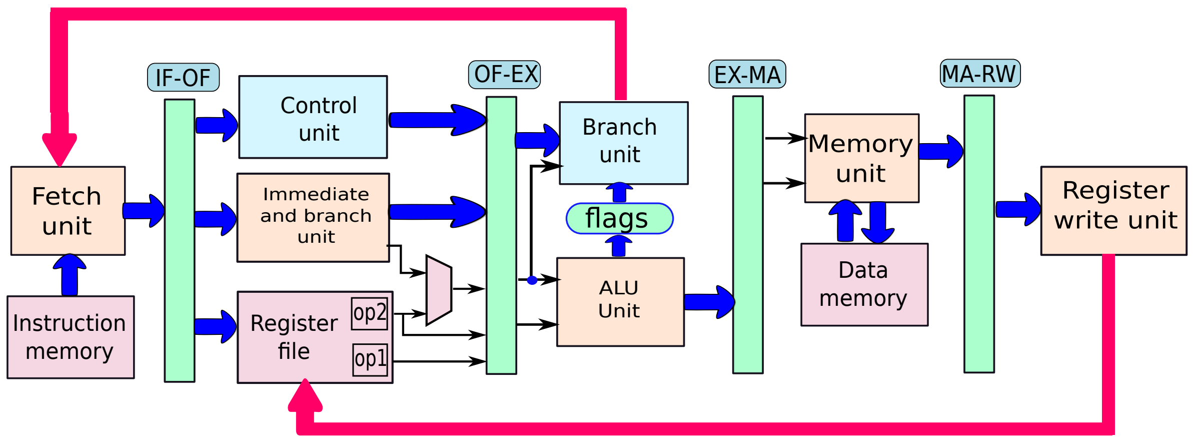


Figure 10.10: An abstraction of the pipelined data path

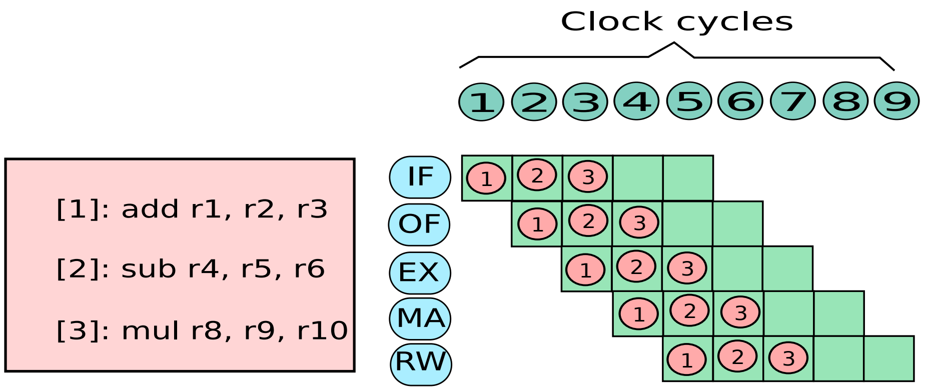


Figure 10.11: The pipeline diagram

instruction, [1] enters the IF stage of the pipeline in the first cycle, and leaves the pipeline in the fifth cycle. Similarly, the second instruction, [2], enters the IF stage of the pipeline in the second cycle, and leaves the pipeline in the sixth cycle. Each of these instructions progresses to the subsequent stage of the pipeline in each cycle. The trace of each instruction in the pipeline diagram is a diagonal that is oriented towards the bottom-right. Note that this scenario will get fairly complicated after we consider dependences across instructions.

Here, are the rules to construct a pipeline diagram.

1. Construct a grid of cells, which has five rows, and N columns, where N is the total number of clock cycles that we wish to consider. Each of the five rows corresponds to a pipeline stage.

2. If an instruction ([k]) enters the pipeline in cycle m, then we add an entry corresponding to [k] in the m^{th} column of the first row.

3. In the $(m+1)^{th}$ cycle, the instruction can either stay in the same stage (because the pipeline might be stalled, described later), or can move to the next row (OF stage). We add a corresponding entry in the grid cell.

4. Similarly, the instruction moves from the IF stage to the RW stage in sequence. It never moves backwards. However, it can stay in the same stage across consecutive cycles.

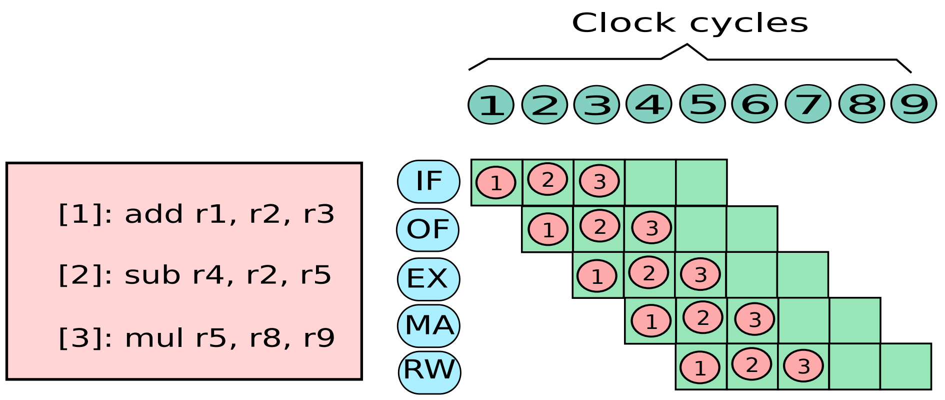
5. We cannot have two entries in a cell.

6. We finally remove the instruction from the pipeline diagram after it leaves the RW stage.

Example 137
Build a pipeline diagram for the following code snippet. Assume that the first instruction enters the pipeline in cycle 1.

```
[1]: add r1, r2, r3
[2]: sub r4, r2, r5
[3]: mul r5, r8, r9
```

Answer:



10.4.2 Data Hazards

Let us consider the following code snippet.

```
[1]: add r1, r2, r3
[2]: sub r3, r1, r4
```

Here, the *add* instruction is producing the value for register, $r1$, and the *sub* instruction is using it as a source operand. Let us now construct a pipeline diagram for just these instructions as shown in Figure 10.12.

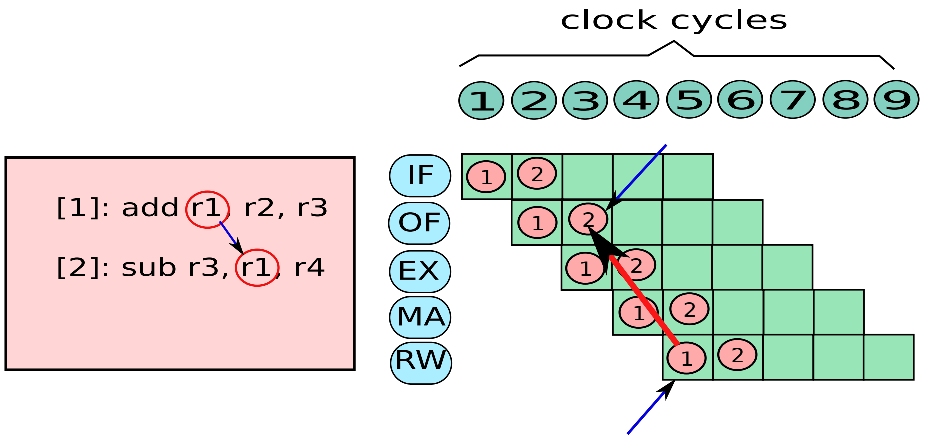


Figure 10.12: Pipeline diagram showing a RAW hazard

There is a problem. Instruction 1 writes the value of $r1$ in the fifth cycle, and instruction 2 needs to read its value in the third cycle. This is clearly not possible. We have added an arrow between the relevant pipeline stages of both the instructions to indicate that there is a dependency. Since the arrow is towards the left (backwards in time), we cannot execute this code sequence in a pipeline. This is known as a *data hazard*. A hazard is defined as the possibility of erroneous execution of an instruction in a pipeline. This specific case is classified as a *data hazard*, where it is possible that instruction 2 might get the wrong data unless adequate steps are taken.

Definition 67
A hazard is defined as the possibility of erroneous execution of an instruction in a pipeline. A data hazard *represents the possibility of erroneous execution because of the unavailability of correct data.*

This specific type of data hazard is known as a RAW (read after write) hazard. Here the subtract instruction is trying to read $r1$, which needs to be written by the add instruction. In this case, a read succeeds a write.

Note that this is not the only kind of data hazard. The two other types of data hazards are WAW (write after write), and WAR (write after read) hazards. These hazards are not an issue in our pipeline because we never change the order of instructions. A preceding instruction is always ahead of a succeeding instruction in the pipeline. This is an example of an *in-order pipeline*. In comparison, modern processors have out-of-order pipelines that execute instructions in different orders.

Definition 68
In an in-order *pipeline (such as ours), a preceding instruction is always ahead of a succeeding instruction in the pipeline. Modern processors use* out-of-order *pipelines that break this rule; it is thus possible for later instructions to execute before earlier instructions.*

Let us take a look at the following assembly code snippet.

```
[1]: add r1, r2, r3
[2]: sub r1, r4, r3
```

Here, instructions 1 and 2 are writing to register $r1$. In an in-order pipeline $r1$ will be written in the correct order, and thus there is no WAW hazard. However, in an out-of-order pipeline we run the risk of finishing instruction 2 before instruction 1, and thus $r1$ can end up with the wrong value. This is an example of a WAW hazard. The reader should note that modern processors ensure that $r1$ does not get the wrong value by using a technique known as *register renaming* (see Section 10.11.4).

Let us give an example of a potential WAR hazard.

```
[1]: add r1, r2, r3
[2]: add r2, r5, r6
```

Here, instruction 2 is trying to write to $r2$, and instruction 1 has $r2$ as a source operand. If instruction 2 executes first, then instruction 1 risks getting the wrong value of $r2$. In practice this does not happen in modern processors because of schemes such as register renaming. The reader needs to understand that a hazard is a theoretical risk of something wrong happening. It is not a real risk because adequate steps are taken to ensure that programs are not executed incorrectly.

In this book, we will mostly focus on RAW hazards, because WAW and WAR hazards are relevant only for modern out-of-order processors. Let us outline the nature of the solution. To avoid a RAW hazard it is necessary to ensure that the pipeline is aware of the fact that it contains a pair of instructions, where one instruction writes to a register, and another instruction that comes later in program order reads from the same register. It needs to ensure that the consumer instruction correctly receives the value of the operand (in this case, register) from the producer instruction. We shall look at solutions in both hardware and software.

10.4.3 Control Hazards

Let us now look at another type of hazards that arise when we have branch instructions in the pipeline. Let us consider the following code snippet.

```
[1]: beq .foo
[2]: mov r1, 4
[3]: add r2, r4, r3
...
...
```

```
.foo:
[100]: add r4, r1, r2
```

Let us show the pipeline diagram for the first three instructions in Figure 10.13.

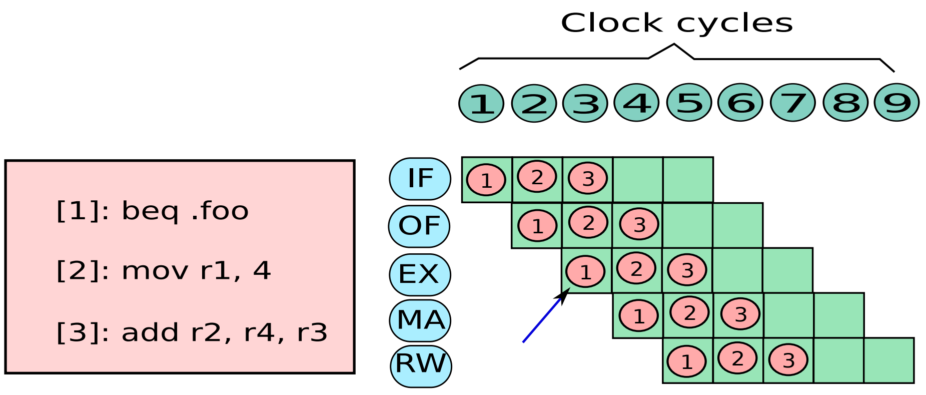


Figure 10.13: Pipeline diagram

Here, the outcome of the branch is decided in cycle 3, and is communicated to the fetch unit. The fetch unit starts fetching the correct instruction from cycle 4. Now, if the branch is taken, then instructions 2, and 3, should not be executed. Sadly, there is no way of knowing in cycles 2 and 3, about the outcome of the branch. Hence, these instructions will be fetched, and will be a part of the pipeline. If the branch is taken, then there is a possibility that instructions 2 and 3 might corrupt the state of the program, and consequently introduce an error. Instructions 2 and 3, are known as *instructions on the wrong path*. This scenario is known as a *control hazard*.

Definition 69
The instructions that would have been executed if the branch instruction 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.*

Definition 70
A control hazard *represents the possibility of erroneous execution in a pipeline because instructions in the* wrong path *of a branch can possibly get executed and save their results in memory, or in the register file.*

To avoid a control hazard, it is necessary to identify instructions on the wrong path, and ensure that their results do not get committed to the register file, and memory. There should be a way to nullify such instructions, or avoid them altogether.

10.4.4 Structural Hazards

Definition 71
A structural hazard *refers to the possibility of instructions not being able to execute because of resource constraints. For example, they can arise when multiple instructions try to access a functional unit in the same cycle, and due to capacity limitations, the unit cannot allow all the interested instructions to proceed. In this case, a few of the instructions in the conflict need to stall their execution.*

Structural hazards do not arise in the *SimpleRisc* pipeline. However, for the sake of completeness, we should still study them. They arise when different instructions try to access the same resource, and the resource cannot allow all of them to access it in the same cycle. Let us give an example. Let us suppose that we had an *add* instruction that could read one operand from memory. It could have the following form:

```
add  r1, r2, 10[r3]
```

Here, we have one register source operand, $r2$, and a memory source operand, $10[r3]$. Let us further assume that our pipeline reads the value of the memory operand in the OF stage. Let us now look at a potentially conflicting situation.

```
[1]: st r4, 20[r5]
[2]: sub r8, r9, r10
[3]: add r1, r2, 10[r3]
```

Note that there are no control and data hazards here. Let us nonetheless, consider a point in the pipeline diagram when the store instruction is in the MA stage. At this point instruction 2 is in the EX stage, and instruction 3 is in the OF stage. Note that in this cycle, both instructions 1 and 3 need to access the memory unit. However, if we assume that the memory unit can only service one request per cycle, then clearly there is a conflicting situation. One of the instructions needs to stall its execution. This situation is an example of a *structural hazard.*

We claim that in our *SimpleRisc* pipeline there are no structural hazards. In other words, we never have a situation in which multiple instructions across different pipeline stages wish to access the same unit, and that unit does not have the capacity to service all the requests. This statement can be proved by considering that the only units that are accessed by multiple stages are the fetch unit, and the register file. The fetch unit is accessed by an instruction in the IF stage, and by branch instructions in the EX stage. It is designed to handle both the requests. Likewise, the register file is accessed by instructions in the OF stage, and RW stage. Our register file has two read ports, and one write port. It can thus handle both the requests in the same cycle.

Let us thus focus on trying to eliminate RAW and control hazards.

10.5 Solutions in Software

10.5.1 RAW Hazards

Now, let us find a way of avoiding a RAW hazard. Let us look at our example again.

```
[1]: add r1, r2, r3
[2]: sub r3, r1, r4
```

Instruction 2 requires the value of $r1$ in the OF stage. However, at that point of time, instruction 1 is in the EX stage, and it would not have written back the value of $r1$ to the register file. Thus, instruction 2 cannot be allowed to proceed in the pipeline. Let us propose a naive software solution to this problem. A smart compiler can analyze the code sequence and realize that a RAW hazard exists. It can introduce *nop* instructions between these instructions to remove any RAW hazards. Let us consider the following code sequence

```
[1]: add r1, r2, r3
[2]: nop
[3]: nop
[4]: nop
[5]: sub r3, r1, r4
```

Here, when the *sub* instruction reaches the OF stage, the *add* instruction would have written its value and left the pipeline. Thus, the *sub* instruction will get the correct value. Note that adding *nop* instructions is a costly solution, because we are essentially wasting computational power. In this example, we have basically wasted 3 cycles by adding *nop* instructions. However, if we consider a longer sequence of code, then the compiler can possibly reorder the instructions such that we can minimize the number of *nop* instructions. The basic aim of any compiler intervention needs to be that there have to be a minimum of 3 instructions between a producer and consumer instruction. Let us consider Example 138.

Example 138
Reorder the following code snippet, and add a sufficient number of nop instructions to make it execute correctly on a SimpleRisc pipeline.

```
add r1, r2, r3
add r4, r1, 3
add r8, r5, r6
add r9, r8, r5
add r10, r11, r12
add r13, r10, 2
```

Answer:

```
add r1, r2, r3
add r8, r5, r6
add r10, r11, r12
nop
add r4, r1, 3
add r9, r8, r5
add r13, r10, 2
```

We need to appreciate two important points here. The first is the power of the *nop* instruction, and the next is the power of the compiler. The compiler is a vital tool in ensuring the correctness of the program, and also improving its performance. In this case, we want to reorder code in such a way that we introduce the minimum number of *nop* instructions.

10.5.2 Control Hazards

Let us now try to use the same set of techniques to solve the issue of control hazards. If we take a look at the pipeline diagram again, then we can conclude that there need to be a minimum of two instructions between the branch instruction and the instruction at the branch target. This is because, we get both the branch outcome, and the branch target at the end of the EX stage. At this point of time there are two more instructions in the pipeline. These instructions have been fetched when the branch instruction was in the OF, and EX stages respectively. They might potentially be on the wrong path. After the branch target, and outcome have been determined in the EX stage, we can proceed to fetch the correct instruction in the IF stage.

Now, let us consider these two instructions that were fetched, when we were not sure of the branch outcome. If the PC of the branch is equal to p_1, then their addresses are $p_1 + 4$, and $p_1 + 8$ respectively. They are not on the wrong path if the branch is not taken. However, if the branch is taken, then these instructions need to be discarded from the pipeline since they are on the wrong path. For doing this, let us look at a simple solution in software.

Let us consider a scheme where the hardware assumes that the two instructions immediately after a branch instruction are not on the wrong path. The positions of these two instructions are known as the *delay slots*. Trivially, we can ensure that the instructions in the delay slots do not introduce errors, by inserting two *nop* instructions after a branch. However, we will not gain any extra performance by doing this. We can instead find two instructions that execute before the branch instruction, and move them to the two delay slots immediately after the branch.

Note that we cannot arbitrarily move instructions to the delay slots. We cannot violate any data dependence constraints, and we need to also avoid RAW hazards. Secondly, we cannot move any compare instructions into the delay slots. If appropriate instructions are not available, then we can always fall back to the trivial solution and insert *nop* instructions. It is also possible that we may find just one instruction that we can reorder, then we just need to insert one *nop* instruction after the branch instruction. The method of delayed branches is a very potent method in reducing the number of *nop* instructions that need to be added to avoid control hazards.

The reader should convince herself that to support this simple software scheme, we do not need to make any changes in hardware. The pipelined data path shown in Figure 10.9 already supports this scheme. In our simple pipelined data path, the two instructions fetched after the branch have their PCs equal to $p_1 + 4$, and $p_1 + 8$ respectively (p_1 is the PC of the branch instruction). Since the compiler ensures that these instructions are always on the correct path irrespective of the outcome of the branch, we do not commit an error by fetching them. After the outcome of the branch has been determined, the next instruction that is fetched either has a PC equal to $p_1 + 12$ if the branch is not taken, or the PC is equal to the branch target if the branch is taken. Thus, in both the cases, the correct instruction is fetched after the outcome of the branch is determined, and we can conclude that our software solution executes programs correctly on the pipelined version of our processor.

To summarize, the crux of our software technique is the notion of the *delay slot*. We need two delay slots after a branch because we are not sure about the two subsequent instructions. They might be on the wrong path. However, using a smart compiler we can manage to move instructions that get executed irrespective of the outcome of the branch to the delay slots. We can thus avoid placing *nop* instructions in the delay slots, and consequently increase performance. Such a branch instruction is known as a *delayed branch*.

Definition 72
A branch instruction is known as a delayed branch *if the processor assumes that all the succeeding instructions that are fetched before its outcome is determined, are guaranteed to be on the correct path. If the processor fetches n instructions between the time that a branch instruction has been fetched, and its outcome has been determined, then we say that we have n delay slots. The compiler needs to ensure that instructions on the correct path occupy the delay slots, and no additional control or RAW hazards are introduced. The compiler can also trivially introduce nop instructions in the delay slots.*

Now, let us consider a set of examples.

Example 139
Reorder the following piece of assembly code to correctly run on a pipelined SimpleRisc processor with delayed branches. Assume two delay slots per branch instruction.

```
add r1, r2, r3
add r4, r5, r6
b .foo
add r8, r9, r10
```

Answer:

```
b .foo
add r1, r2, r3
add r4, r5, r6
add r8, r9, r10
```

10.6 Pipeline with Interlocks

Up till now, we have only looked at software solutions for eliminating RAW and control hazards. However, compiler approaches, are not very generic. Programmers can always write assembly code manually, and try to run it on the processor. In this case, the likelihood of an error is high, because programmers might not have reordered their code properly to remove hazards. Secondly, there is an issue of portability. A piece of assembly code written for one pipeline, might not run on another pipeline that follows the same ISA. This is because it might have a different number of delay slots, or different number of stages. One of our main aims of introducing assembly programs gets defeated, if our assembly programs are not portable across different machines that use the same ISA.

Hence, let us try to design solutions at the hardware level. The hardware should ensure that irrespective of the assembly program, it is run correctly. The output should always match that produced by a single cycle processor. To design such a processor, we need to ensure that an instruction never receives wrong data, and wrong path instructions are not executed. This can be done by ensuring that the following conditions hold.

- Condition: `Data-Lock` : We cannot allow an instruction to leave the OF stage unless it has received the correct data from the register file. This means that we need to effectively *stall* the IF and OF stages and let the rest of the stages execute till the instruction in the OF stage can safely read its operands. During this time, the instruction that passes from the OF to the EX stage needs to be a *nop* instruction.
- Condition: `Branch-Lock` : We never execute instructions on the wrong path. We either stall the processor till the outcome is known, or use techniques to ensure that instructions on the wrong path are not able to commit their changes to the memory, or registers.

Definition 73
In a hardware implementation of a pipeline, it is sometimes necessary to stop a new instruction from entering a pipeline stage, till a certain condition ceases to hold. The notion of stopping a pipeline stage from accepting and processing new data, is known as a pipeline stall*, or a* pipeline interlock*. Its primary purpose is to ensure the correctness of program execution.*

If we ensure that both the `Data-Lock` and `Branch-Lock` conditions hold, then our pipeline will execute instructions correctly. Note that both the conditions dictate that possibly some stages of the pipeline needs to be stalled for some time. These stalls are also known as *pipeline interlocks*. In other words, by keeping our pipeline idle for some time, we can avoid executing instructions that might potentially lead to an erroneous execution. Let us now quickly compare the pure software and hardware schemes in Table 10.1, and see what are the pros and cons of implementing the entire logic of the pipeline in hardware. Note that in the software solution we try to reorder code, and subsequently insert the minimum number of *nop* instructions to nullify the effect of hazards. In comparison, in the hardware solution, we dynamically stall parts of the pipeline to avoid executing instructions on the wrong path, or with wrong values of operands. Stalling the pipeline is tantamount to keeping some stages idle, and inserting *nop* instructions in other stages as we shall see later in this section.

Attribute	Software	Hardware (with interlocks)
Portability	Limited to a specific processor	Programs can be run on any processor irrespective of the nature of the pipeline
Branches	Possible to have no performance penalty, by using delay slots	Need to stall the pipeline for 2 cycles in our design
RAW hazards	Possible to eliminate them through code scheduling	Need to stall the pipeline
Performance	Highly dependent on the nature of the program	The basic version of a pipeline with interlocks is expected to be slower than the version that relies on software

Table 10.1: Comparison between software and hardware approaches for ensuring the correctness of a pipeline

We observe that the efficacy of the software solution is highly dependent on the nature of the program. It is possible to reorder the instructions in some programs to completely hide the deleterious effects of RAW hazards and branches. However, in some programs we might not find enough instructions that can be reordered. We would be thus compelled to insert a lot of *nop* instructions, and this would reduce our performance. In comparison, a pure hardware scheme, which obeys the `Data-Lock` and `Branch-Lock` conditions stalls the pipeline whenever it detects an instruction that might execute erroneously. It is a generic approach, which is slower than a pure software solution.

Now, it is possible to combine the hardware and software solutions to reorder code to make it "pipeline friendly" as much as possible, and then run it on a pipeline with interlocks. Note that in this approach, no guarantees of correctness are made by the compiler. It simply spaces producer and consumer instructions as far apart as possible, and takes advantage of delayed branches if they are supported. This reduces the number of times we need to stall the pipeline, and ensures the best of both worlds. Before proceeding to design a pipeline with interlocks, let us study the nature of interlocks with the help of pipeline diagrams.

10.6.1 A Conceptual Look at a Pipeline with Interlocks

Data Hazards

Let us now draw the pipeline diagram of a pipeline with interlocks. Let us consider the following code snippet.

```
[1]: add r1, r2, r3
[2]: sub r4, r1, r2
```

Here, instruction [1] writes to register $r1$ and instruction [2] reads from $r1$. Clearly, there is a RAW dependence. To ensure the `Data-Lock` condition, we need to ensure that instruction [2] leaves the OF stage only when it has read the value of $r1$ written by instruction [1]. This is possible only in cycle 6 (refer to the pipeline diagram in Figure 10.14). However, instruction [2] reaches the OF stage in cycle 3. If there had been no hazard, then it would have ideally proceeded to the EX stage in cycle 4. Since we have an interlock, instruction [2] needs to stay in the OF stage in cycles 4,5 and 6 also. The question is, "what does the EX stage do when it is not processing a valid instruction in cycles 4, 5 and 6?" Similarly, the MA stage does not process any valid instruction in cycles 5, 6 and 7. We need to have a way to disable pipeline stages, such that we do not perform redundant work. The standard approach is to insert *nop* instructions into stages, if we want to effectively disable them.

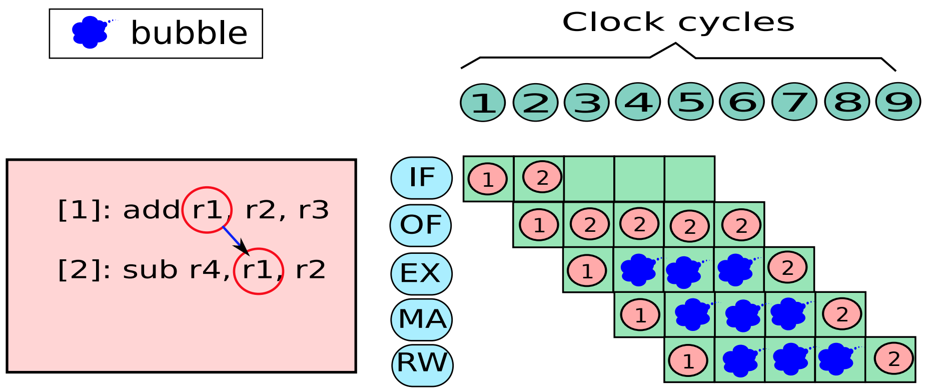


Figure 10.14: A pipeline diagram with bubbles

Let us refer to Figure 10.14 again. At the end of cycle 3, we know that we need to introduce an interlock. Hence, in cycle 4, instruction [2] remains in the OF stage, and we insert a *nop* instruction into the EX stage. This *nop* instructions moves to the MA stage in cycle 5, and RW stage in cycle 6. This *nop* instruction is called a *pipeline bubble*. A bubble is a *nop* instruction that is dynamically inserted by the interlock hardware. It moves through the pipeline stages akin to normal instructions. Similarly, in cycles 5 and 6 also, we need to insert a pipeline bubble. Finally, in cycle 7, instruction [2] is free to proceed to the EX, and subsequent stages. A bubble by definition does not do anything, and thus none of the control signals are turned on when a stage encounters a bubble. The other subtle point to note here is that we cannot read and write to the same register in the same cycle. We need to give preference to the write because it is an earlier instruction, and the read needs to stall for one cycle.

There are two ways to implement a bubble. The first is that we can have a separate bubble bit in the instruction packet. Whenever, the bit is 1, the instruction will be construed to be a bubble. The second is that we can change the opcode of the instruction to that of a *nop*, and replace all of its control signals by 0s. The latter approach is more invasive, but can eliminate redundant work in the circuit completely. In the former approach, the control signals will be on, and units that are activated by them, will remain operational. The hardware needs to ensure that a bubble is not able to make changes to registers or memory.

Definition 74
A pipeline bubble *is a nop instruction that is inserted dynamically in a pipeline register by the interlock hardware. A bubble propagates through the pipeline in the same way as normal instructions.*

We can thus conclude that it is possible to avoid data hazards, by dynamically inserting bubbles in the pipeline. Let us quickly take a look at the issue of slow instructions such as the *div* and *mod* instructions. It is highly likely that in most pipelines these instructions will take n ($n > 1$) cycles to execute in the EX stage. In each of the n cycles, the ALU completes a part of the processing of the *div* or *mod* instructions. Each such cycle is known as a *T State*. Typically, one stage has 1 T State; however, the EX stage for a slow instruction has many T states. Hence, to correctly implement slow instructions, we need to stall the IF and OF stages for $(n - 1)$ cycles till the operations complete.

For the sake of simplicity, we shall not discuss this issue further. Instead, we shall move on with the simplistic assumption that all our pipeline stages are balanced, and take 1 cycle to complete their operation.

Control Hazards

Now, let us look at control hazards. Let us start out by considering the following code snippet.

```
[1]: beq .foo
[2]: add r1, r2, r3
[3]: sub r4, r5, r6
....
....
.foo:
[4]: add r8, r9, r10
```

Instead of using a delayed branch, we can insert bubbles in the pipeline if the branch is taken. Otherwise, we do not need to do anything. Let us assume that the branch is taken. The pipeline diagram for this case is shown in Figure 10.15.

In this case, the outcome of the branch condition of instruction [1] is decided in cycle 3. At this point, instructions [2] and [3] are already in the pipeline (in the IF and OF stages, respectively). Since the branch condition evaluates to *taken*, we need to cancel instructions [2] and [3], otherwise they will be executed erroneously. We thus, convert them to bubbles as shown in Figure 10.15. Instructions[2] and [3] are converted to bubbles in cycle 4. Secondly, we fetch from the correct branch target (.foo) in cycle 4, and thus instruction [4] enters the pipeline. Both of our bubbles proceed through all the pipeline stages, and finally leave the pipeline in cycles 6 and 7 respectively.

We can thus ensure both the conditions (`Data-Lock` and `Branch-Lock`) by dynamically introducing bubbles in the pipeline. Now, let us look at these approaches in some more detail.

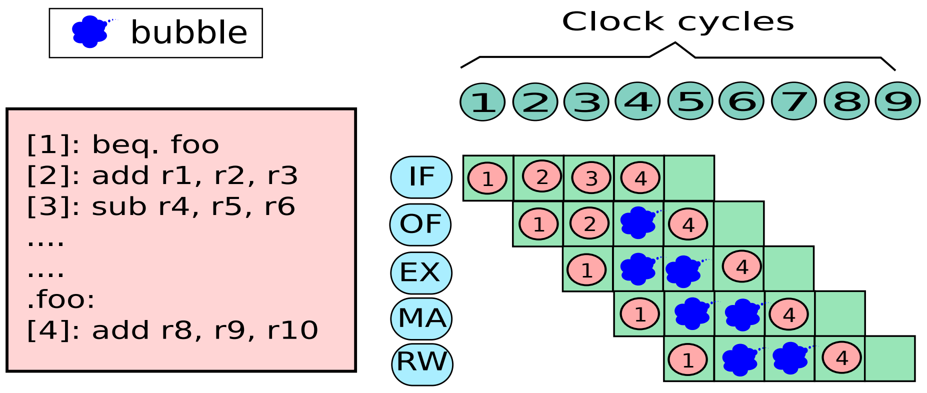


Figure 10.15: Pipeline diagram for a control hazard with bubbles

10.6.2 Ensuring the `Data-Lock` Condition

To ensure the `Data-Lock` condition we need to ensure that there is no conflict between the instruction in the OF stage, and any instruction in the subsequent stages. A *conflict* is defined as a situation that can cause a RAW hazard. In other words, a conflict exists if an instruction in a subsequent stage writes to a register that is read by the instruction in the OF stage. There are thus two pieces of hardware that we require to implement the `Data-Lock` condition. The first is to check if a conflict exists, and the second is to ensure that the pipeline gets stalled.

Let us first look at the conflict detection hardware. The conflict detection hardware needs to compare the contents of the instruction in the OF stage with the contents of each of the instructions in the other three stages namely EX, MA, and RW. If there is a conflict with any of these instructions, we can declare a conflict. Let us thus focus on the logic of detecting a conflict. We leave the design of the exact circuit as an exercise for the reader. Let us outline the brief pseudo-code of a conflict detection circuit. Let the instruction in the OF stage be [A], and an instruction in a subsequent stage be [B]. The algorithm to detect a conflict is shown as Algorithm 5.

Implementing Algorithm 5 in hardware is straightforward. The reader can draw a simple circuit and implement this algorithm. All we need is a set of logic gates and multiplexers. Most hardware designers typically write the description of a circuit similar to Algorithm 5 in a hardware description language such as Verilog or VHDL, and rely on smart compilers to convert the description to an actual circuit. Hence, we shall refrain from showing detailed implementations of circuits henceforth, and just show the pseudo code.

We need three conflict detectors (OF $\leftrightarrow$ EX, OF $\leftrightarrow$ MA, OF $\leftrightarrow$ RW). If there are no conflicts, then the instruction is free to proceed to the EX stage. However, if there is at least one conflict, we need to stall the IF and OF stages. Once an instruction passes the OF stage, it is guaranteed to have all of its source operands.

Stalling the Pipeline:
Let us now look at stalling the pipeline. We essentially need to ensure that till there is a conflict no new instruction enters the IF and OF stages. This can be trivially ensured by disabling the write functionality of the PC and the IF-OF pipeline register. They thus cannot accept new data on a clock edge, and thus will continue to hold their previous values.

Secondly, we also need to insert bubbles in the pipeline. For example, the instruction that passes from the OF to the EX stage needs to be an invalid instruction, or alternatively a bubble. This can be ensured by passing a *nop* instruction. Hence, the circuit for ensuring the `Data-Lock` condition is straightforward. We need a conflict detector that is connected to the PC, and the IF-OF register. Till there is a conflict, these two registers are disabled, and cannot accept new data. We force the instruction in the OF-EX register to contain a *nop*. The augmented circuit diagram of the pipeline is shown in Figure 10.16.

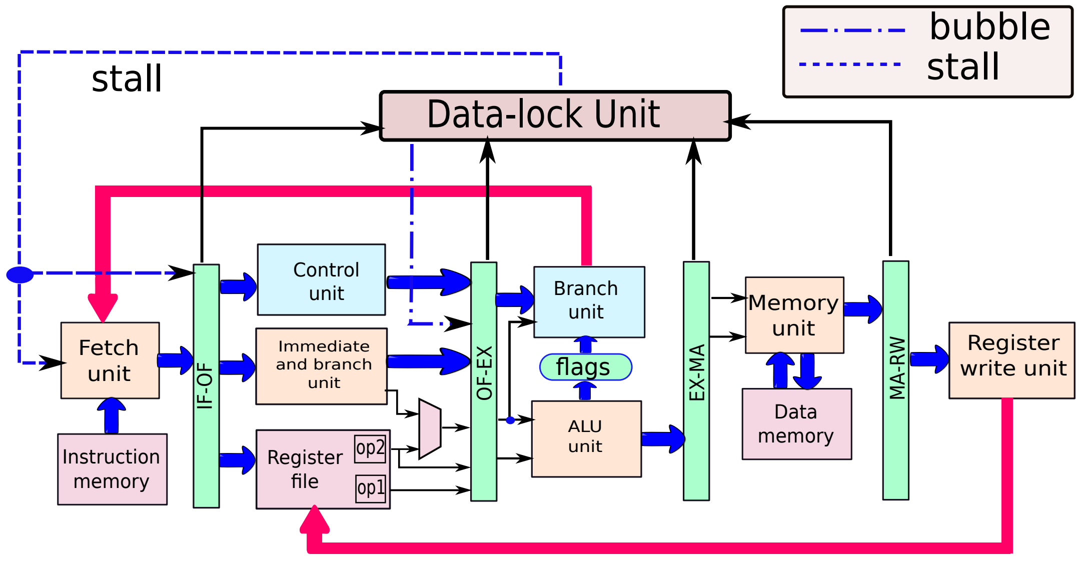


Figure 10.16: Data path of a pipeline with interlocks (implements the `Data-Lock` condition)

Algorithm 5: Algorithm to detect conflicts between instructions

```
   Data: Instructions: [A] and [B]
   Result: Conflict exists (true), no conflict (false)
 1 if [A].opcode ∈ (nop,b,beq,bgt,call) then
       /* Does not read from any register */
 2     return false
 3 end
 4 if [B].opcode ∈ (nop, cmp, st, b, beq, bgt, ret) then
       /* Does not write to any register */
 5     return false
 6 end
   /* Set the sources */
 7 src1 ← [A].rs1
 8 src2 ← [A].rs2
 9 if [A].opcode = st then
10     src2 ← [A].rd
11 end
12 if [A].opcode = ret then
13     src1 ← ra
14 end
   /* Set the destination */
15 dest ← [B].rd
16 if [B].opcode = call then
17     dest ← ra
18 end
   /* Check if the first operand exists */
19 hasSrc1 ← true
20 if [A].opcode ∈ (not,mov) then
21     hasSrc1 ← false
22 end
   /* Check the second operand to see if it is a register */
23 hasSrc2 ← true
24 if [A].opcode ∉ (st) then
25     if [A].I = 1 then
26         hasSrc2 ← false
27     end
28 end
   /* Detect conflicts */
29 if (hasSrc1 = true) and (src1 = dest) then
30     return true
31 end
32 else if (hasSrc2 = true) and (src2 = dest) then
33     return true
34 end
35 return false
```

10.6.3 Ensuring the `Branch-Lock` condition

Let us now assume that we have a branch instruction in the pipeline (*b*, *beq*, *bgt*, *call*, *ret*). If we have delay slots, then our data path is the same as that shown in Figure 10.16. We do not need to do any changes, because the entire complexity of execution has been offloaded to software. However, exposing the pipeline to software has its pros and cons as discussed in Table 10.1. If we add more stages in the pipeline, then existing executables might cease to work. To avoid this let us design a pipeline that does not expose delay slots to software.

We have two design options here. The first is that we can assume that a branch is not taken till the outcome is decided. We can proceed to fetch the two instructions after a branch and process them. Once, the outcome of the branch is decided in the EX stage, we can take an appropriate action based on the outcome. If the branch is not taken, then the instructions fetched after the branch instruction, are on the correct path, and nothing more needs to be done. However, if the branch is taken, then it is necessary to cancel those two instructions, and replace them with pipeline bubbles (*nop* instructions).

The second option is to stall the pipeline till the outcome of the branch is decided, irrespective of the outcome. Clearly, the performance of this design is less than the first alternative that assumes that branches are not taken. For example, if a branch is not taken 30% of the time, then with the first design, we do useful work 30% of the time. However, with the second option, we never do any useful work in the 2 cycles after a branch instruction is fetched. Hence, let us go with the first design in the interest of performance. We cancel the two instructions after the branch only if the branch is taken. We call this approach *predict not taken*, because we are effectively predicting the branch to be *not taken*. Later on if this prediction is found to be wrong, then can cancel the instructions on the wrong path.

Important Point 14
If the PC of a branch instruction is equal to p, then we choose to fetch the instructions at $p + 4$, and $p + 8$ over the next two cycles. If the branch is not taken, then we resume execution. However, if the branch is taken, then we cancel these two instructions, and convert them to pipeline bubbles.

We do not need to make any significant changes to the data path. We need a small branch hazard unit that takes an input from the EX stage. If the branch is taken, then in the next cycle it converts the instructions in the IF-OF and OF-EX stages to pipeline bubbles. The augmented data path with the branch interlock unit is shown in Figure 10.17.

10.7 Pipeline with Forwarding

10.7.1 Basic Concepts

We have now implemented a pipeline with interlocks. Interlocks ensure that a pipeline executes correctly irrespective of the nature of dependences across instructions. For the `Data-Lock` condition we proposed to add interlocks in the pipeline that do not allow an instruction to

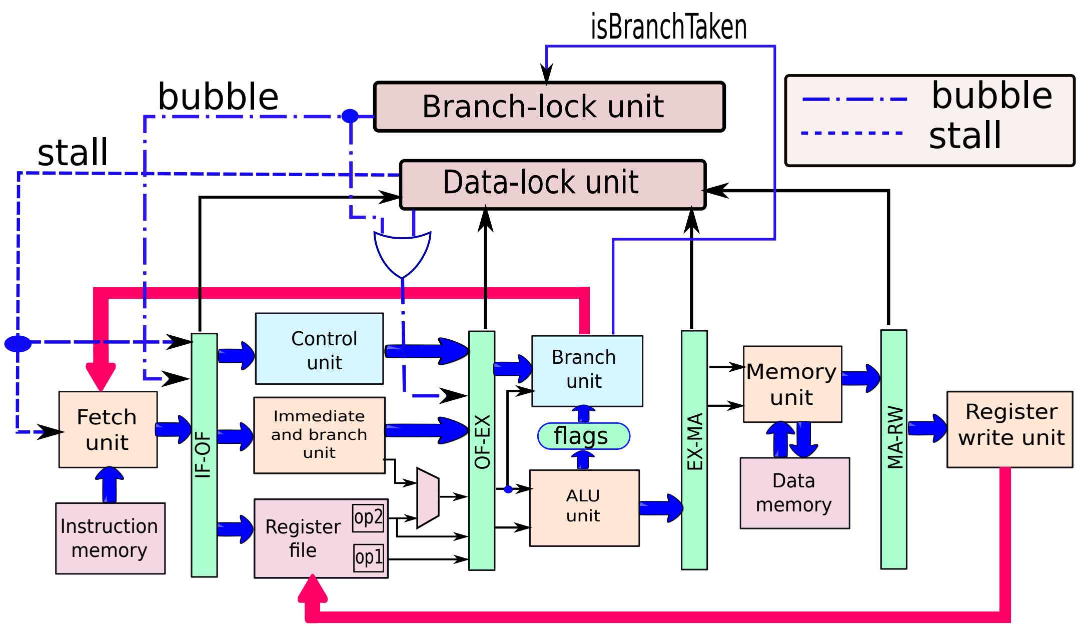


Figure 10.17: Data path of a pipeline with interlocks (implements both the `Data-Lock` and `Branch-Lock` conditions)

leave the operand fetch stage until the correct values are available in the register file. However, we shall see in this section that we do not need to add interlocks always. In fact, in a lot of instances, the correct data is already present in pipeline registers, albeit not in the register file. We can design a method to properly pass data from the internal pipeline registers to the appropriate functional unit. Let us consider a small example by considering this *SimpleRisc* code snippet.

```
[1]: add r1, r2, r3
[2]: sub r4, r1, r2
```

Let us take a look at the pipeline diagram with just these two instructions in Figure 10.18. Figure 10.18(a) shows the pipeline diagram with interlocks. Figure 10.18(b) shows a pipeline diagram without interlocks and bubbles. Let us now try to argue that we do not need to insert a bubble between the instructions.

Let us take a deeper look at Figure 10.18(b). Instruction 1 produces its result at the end of the EX stage, or alternatively at the end of cycle 3, and writes to the register file in cycle 5. Instruction 2 needs the value of $r1$ in the register file at the beginning of cycle 3. This is clearly not possible, and thus we had proposed to add pipeline interlocks to resolve this issue. However, let us try an alternative solution instead. Let us allow the instructions to execute. Then in cycle 3, [2] will get the wrong value. We allow it to proceed to the EX stage in cycle 4. At this point of time, instruction [1] is in the MA stage, and its instruction packet contains the correct value of $r1$. This value of $r1$ was computed in the previous cycle, and is present in the *aluResult* field of the instruction packet. [1]'s instruction packet is in the EX-MA register in

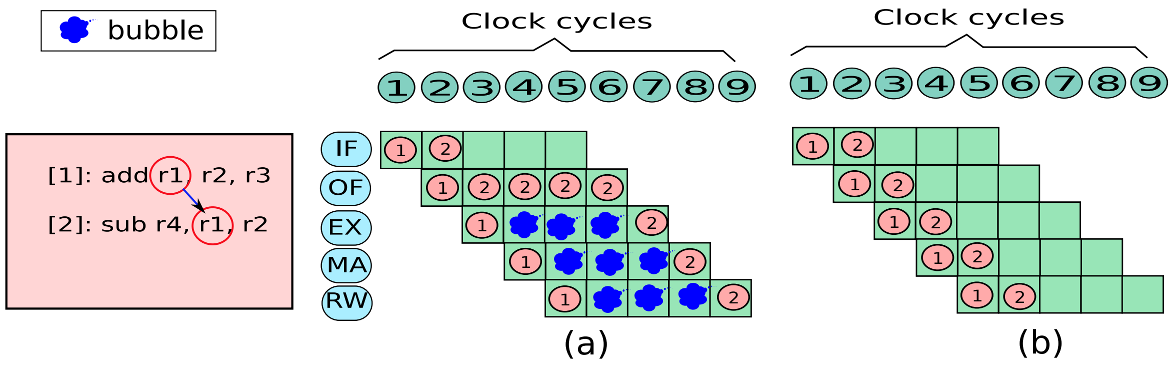


Figure 10.18: (a) Pipeline diagram with interlocks and bubbles (b) Pipeline diagram without bubbles

cycle 4. Now, if we add a connection between the *aluResult* field of the EX-MA register and an input of the ALU, then we can successfully transfer the correct value of $r1$ to the ALU. There will be no error in our computation, because the operands to the ALU are correct, and thus the result of the ALU operation will also be computed correctly. Figure 10.19 shows the result of our actions in the pipeline diagram. We add a line from the MA stage of instruction [1] to the EX stage of instruction [2]. Since the arrow does not go backwards in time, it is possible to **forward** the data (value of $r1$) from one stage to the other.

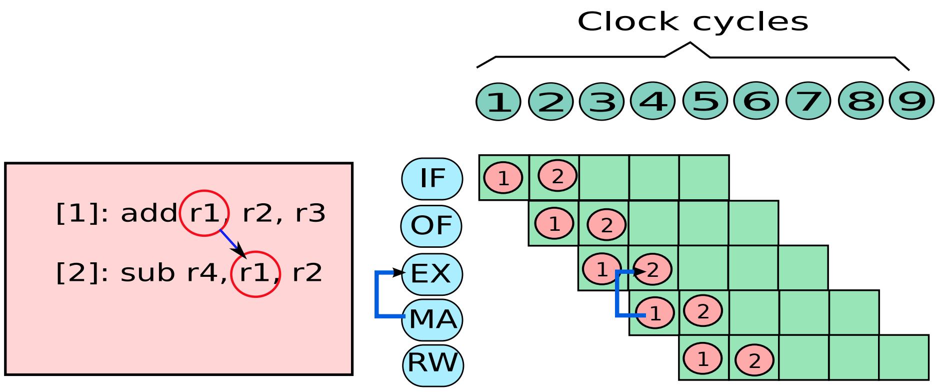


Figure 10.19: Example of forwarding in a pipeline

Definition 75
Forwarding *is a method to transfer values of operands between instructions in different pipeline stages through direct connections between the stages. We do not use the register file*

for transferring the values of operands across instructions. Forwarding allows us to avoid costly pipeline interlocks.

We have just looked at an extremely powerful technique for avoiding stalls in pipelines. This technique is known as *forwarding*. Essentially, we allow the values of operands to flow between instructions by directly transferring them across stages. We do not use the register file to transfer values across instructions. The notion of forwarding has allowed us to execute instructions [1] and [2] back to back (in consecutive cycles). We do not need to add any stall cycles. Hence, it is not necessary to reorder code, or insert *nop*s.

Before, we proceed to the implementation of forwarding, let us discuss forwarding conceptually using pipeline diagrams. To forward the value of $r1$ between instructions [1] and [2], we added a connection between the MA stage and the EX stage. We showed this connection in Figure 10.19 by drawing an arrow between the corresponding stages of instructions [1] and [2]. The direction of this arrow was vertically upwards. Since it did not go backwards in time, we concluded that it is possible to forward the value. Otherwise, it would not have been possible.

Let us now try to answer a general question. Can we forward values between all pairs of instructions? Note that these need not be consecutive instructions. Even if there is one instruction between a producer and a consumer ALU instruction, we still need to forward values. Let us now try to think of all possible forwarding paths between stages in a pipeline.

10.7.2 Forwarding Paths in a Pipeline

Let us discuss the basic tenets of forwarding that we shall broadly aim to follow.

1. We add a forwarding path between a later stage and an earlier stage.
2. We forward a value as late as possible in the pipeline. For example, if a given value is not required in a given stage, and it is possible to get the value in a later stage from the producer instruction, then we wait to get the forwarded value in the later stage.

Note that both of these basic tenets do not affect the correctness of programs. They simply allow us to eliminate redundant forwarding paths. Let us now systematically look at all the forwarding paths that we require in our pipeline.

RW → MA : Let us consider the MA stage. It needs a forwarding path from the RW stage. Let us consider the code snippet shown in Figure 10.20 Here, instruction [2] needs the value of $r1$ in the MA stage (cycle 5), and instruction [1] fetches the value of $r1$ from memory by the end of cycle 4. Thus, it can forward its value to instruction [2] in cycle 5.
RW → EX : The code snippet shown in Figure 10.21 shows a load instruction that fetches the value of register $r1$ by the end of cycle 4, and a subsequent ALU instruction that requires the value of $r1$ in cycle 5. It is possible to forward the value because we are not going backwards in time.
MA → EX : The code snippet shown in Figure 10.22 shows an ALU instruction that computes the value of register $r1$ by the end of cycle 3, and a consecutive ALU instruction that requires

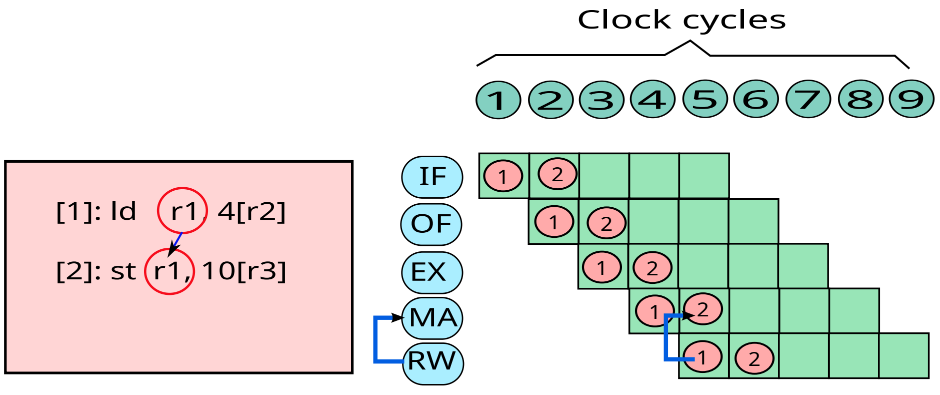


Figure 10.20: RW → MA forwarding

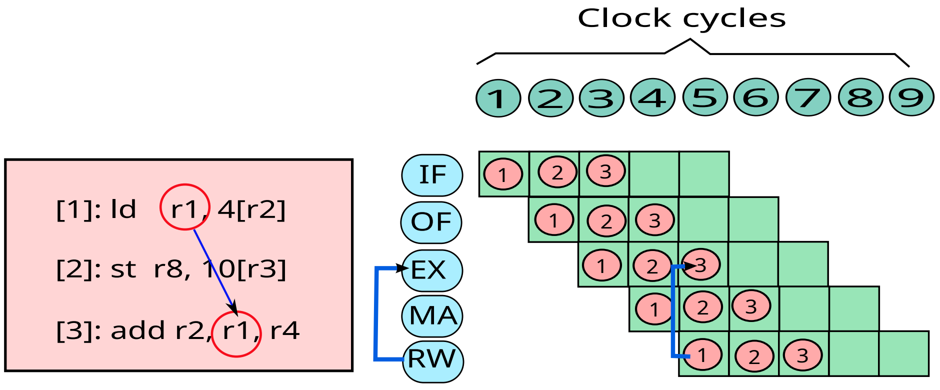


Figure 10.21: RW → EX forwarding

the value of $r1$ in cycle 4. In this case also, it is possible to forward the data by adding an interconnection (forwarding path) between the MA and EX stages.

RW → OF : Typically the OF stage does not need forwarding paths because it does not have any functional units. Hence, it does not need to use a value immediately. We can thus forward the value later according to tenet 2. However, the only exception is forwarding from the RW stage. We cannot forward the value later because the instruction will not be there in the pipeline. Hence, it is necessary to add a forwarding path from the RW to the OF stage. An example of a code snippet that requires $RW \rightarrow OF$ forwarding is shown in Figure 10.23. Instruction [1] produces the value of $r1$ by reading its value from memory by the end of cycle 4. It then writes the value of $r1$ to the register file in cycle 5. Meanwhile, instruction [4] tries to read the value of $r1$ in the OF stage in cycle 5. Unfortunately, there is a conflict here. Hence, we propose to resolve the conflict by adding a forwarding path between the RW and OF stages. Thus, we prohibit instruction [4] from reading the register file for the value of $r1$. Instead,

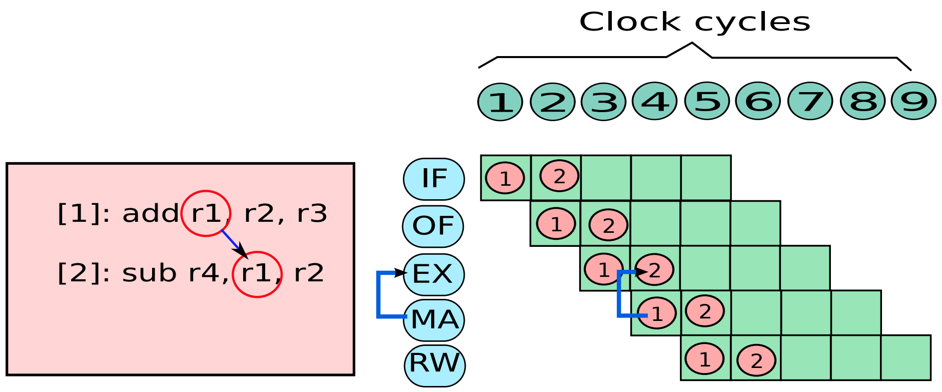


Figure 10.22: MA → EX forwarding

instruction [4] gets the value of $r1$ from instruction [1] using the $RW \rightarrow OF$ forwarding path.

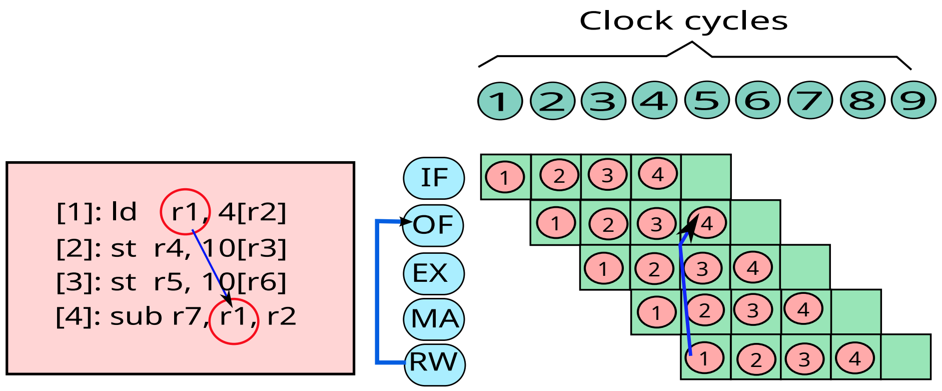


Figure 10.23: RW → OF forwarding

Important Point 15
Forwarding from the RW to the OF stage is a very tricky operation. This is because the instruction in the RW stage is writing to the register file, and the instruction in the OF stage is also reading from the register file. If the value of the register is the same in these two instructions, then it is typically not possible to perform both the operations (read and write) in the same cycle. This is because reading and writing to the same SRAM cell can lead to incorrect operation of the circuit, and it is hard to ensure correctness. Consequently, it is a standard practice to allow the write from RW to go through, and cancel the register

read operation issued by the instruction in the OF stage. Thus, this read operation does not go to the register file. Instead, the instruction in the OF stage gets the value of the register through the forwarding path. This strategy ensures that we do not have any remote chances of leaving data in an inconsistent state in the register file. The instruction in the OF stage also gets the right value of the operands.

It is not necessary to add the following forwarding paths: MA → OF, and EX → OF. This is because, we can use the following forwarding paths (RW → EX), and (MA → EX) instead. In accordance with tenet 2, we need to avoid redundant forwarding paths. Hence, we do not add the forwarding paths to the OF stage from the MA and EX stages. We do not add forwarding paths to the IF stage because at this stage, we have not decoded the instruction, and thus we do not know about its operands.

10.7.3 Data Hazards with Forwarding

Question 7
Has forwarding completely eliminated data hazards?

Let us now answer this question. Let us consider ALU instructions. They produce their result in the EX stage, and they are ready to forward in the MA stage. Any succeeding consumer instruction will need the value of an operand produced by the preceding ALU instruction at the earliest in its EX stage. At this point, we can effect a successful forwarding because the value of the operand is already available in the MA stage. Any subsequent instruction can always get the value using any of the available forwarding paths or from the register file if the producer instruction has left the pipeline. The reader should be convinced that if the producer instruction is an ALU instruction, then it is always possible to forward the result of the ALU operation to a consumer instruction. To prove this fact, the reader needs to consider all possible combinations of instructions, and find out if it is possible to forward the input operands to the consumer instruction.

The only other instruction that produces a register value explicitly is the load instruction. Recall that the store instruction does not write to any register. Let us look at the load instruction. The load instruction produces its value at the end of the MA stage. It is thus ready to forward its value in the RW stage. Let us now consider a code snippet and its pipeline diagram in Figure 10.24.

Instruction [1] is a load instruction that writes to register r1, and instruction [2] is an ALU instruction that uses register $r1$ as a source operand. The load instruction is ready to forward at the beginning of cycle 5. Sadly, the ALU instruction needs the value of $r1$ at the beginning of cycle 4. We thus need to draw an arrow in the pipeline diagram that flows backwards in time. Hence, we can conclude that in this case forwarding is not possible.

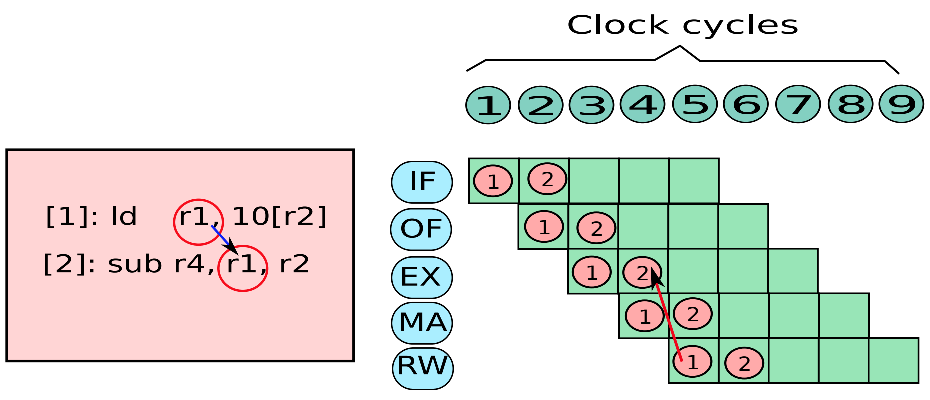


Figure 10.24: The load-use hazard

Definition 76
Load-Use Hazard A load-use *hazard is a situation where a load instruction supplies the loaded value to an immediately succeeding instruction that needs the value in its EX stage. A pipeline even with forwarding needs to insert a single stall cycle after the load instruction.*

This is the only case in which we need to introduce a stall cycle in our pipeline. This situation is known as a *load-use* hazard, where a load instruction supplies the loaded value to an immediately succeeding instruction that needs the value in its EX stage. The standard method of eliminating load-use hazards is by allowing the pipeline to insert a bubble, or by using the compiler to either reorder instructions or insert a *nop* instruction.

Thus, we can conclude that a pipeline with forwarding does need interlocks, albeit rarely. The only special condition is a load-use hazard.

Note that if there is a store instruction after a load instruction that stores the loaded value, then we do not need to insert a stall cycle. This is because the store instruction needs the value in its MA stage. At this point of time the load instruction is in the RW stage, and it is possible to forward the value.

10.7.4 Implementation of a Pipeline with Forwarding

Now, let us come to the most important part of our discussion. Let us design a pipeline with forwarding. We shall first design the data path, and then briefly look at the control path. To implement a data path that supports forwarding, we need to make minor changes to our pipeline stages. These changes will allow the functional units to use their default input values, as well as outputs of subsequent stages in the pipeline. The basic idea is to use a multiplexer before every input to a functional unit. The role of this multiplexer is to select the right input. Let us now look at each of the pipeline stages. Note that we do not need to make any changes to the IF stage because it does not send or receive any forwarded value.

OF Stage with Forwarding

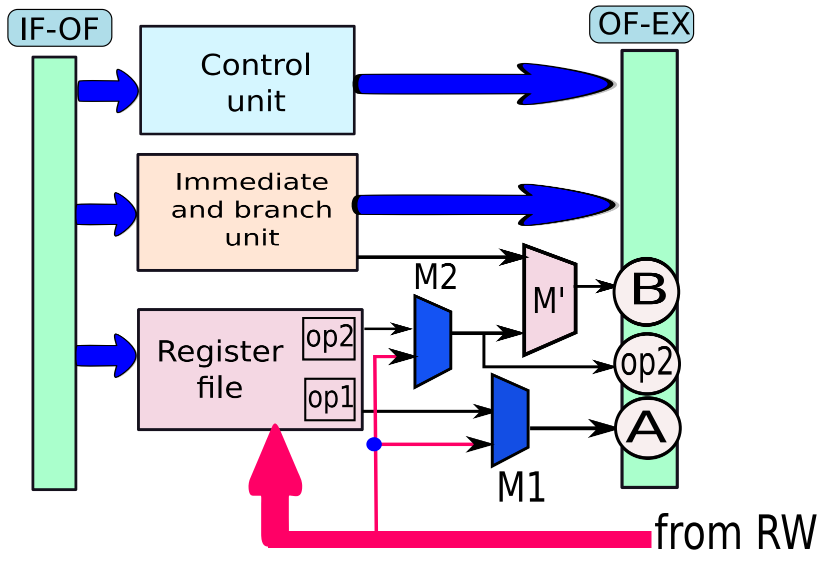


Figure 10.25: OF stage with forwarding

The OF stage with support for forwarding is shown in Figure 10.25. The multiplexers in our baseline pipeline without forwarding are colored with a lighter color. Whereas, the additional multiplexers added to enable forwarding are colored with a darker color. We shall use this convention for the rest of our discussion on forwarding. Let us focus on the two new multiplexers in the OF stage.

We only show those multiplexers that are relevant to our discussion on forwarding. We need to choose between the first operand read from the register file, and the value forwarded from the RW stage. We thus add a multiplexer($M1$) to help us choose between these two inputs. Likewise, we need to choose between the second operand read from the register file, and the value forwarded from the RW stage. To implement forwarding, we add a multiplexer ($M2$) to make a choice between the value fetched from the register file, and the value forwarded from the RW stage (see Figure 10.25). Multiplexer (M'), which is a part of our baseline design chooses between the second register operand and the immediate computed from the contents of the instruction. Recall that the three fields in the instruction packet that save the results of the OF stage are as follows. A saves the value of the first register operand, $op2$ saves the value of the second register operand (rd register in case of a store), and B saves the value of the second operand of the instruction (register or immediate). Recall that we had decided to read all the values that might be required by any instruction in the interest of time. For example, the *not* instruction does not require the first register operand. Nevertheless, we still read it because we do not have enough time to take a decision about whether to read or not read the register operands.

EX Stage with Forwarding

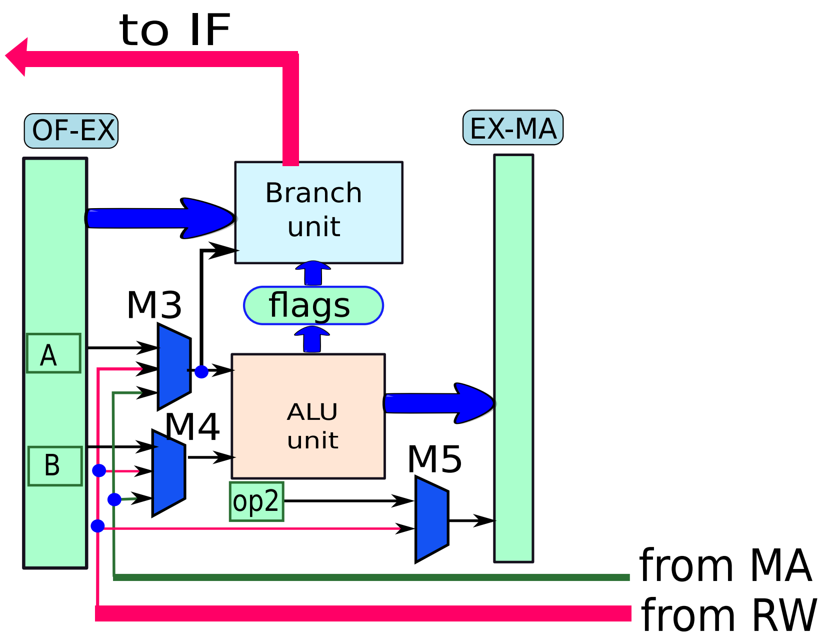


Figure 10.26: EX stage with forwarding

Figure 10.26 shows the modified EX stage. The three inputs that the EX stage gets from the OF stage are *A* (first ALU operand), *B* (second ALU operand), and *op*2 (second register operand). For *A* and *B*, we add two multiplexers, $M3$, and $M4$, to choose between the values computed in the OF stage, and the values forwarded from the MA and RW stages respectively. For the *op*2 field, which possibly contains the store value, we do not need $MA \rightarrow$ EX forwarding. This is because the store value is required in the MA stage, and thus we can use $RW \rightarrow MA$ forwarding. This observation allows us to reduce one forwarding path. Hence, multiplexer $M5$ has two inputs (default and the value forwarded from the *RW* stage).

MA Stage with Forwarding

Figure 10.27 shows the MA stage with additional support for forwarding. The memory address is computed in the EX stage, and saved in the *aluResult* field of the instruction packet. The memory unit directly uses this value for the address. However, in the case of a store, the value that needs to be stored (*op*2) can possibly be forwarded from the RW stage. We thus add multiplexer $M6$, which chooses between the *op*2 field in the instruction packet and the value forwarded from the RW stage. The rest of the circuit remains the same.

RW Stage with Forwarding

Finally, Figure 10.28 shows the RW stage. Since this is the last stage, it does not use any forwarded value. However, it sends the value that it writes to the register file to the MA, EX,

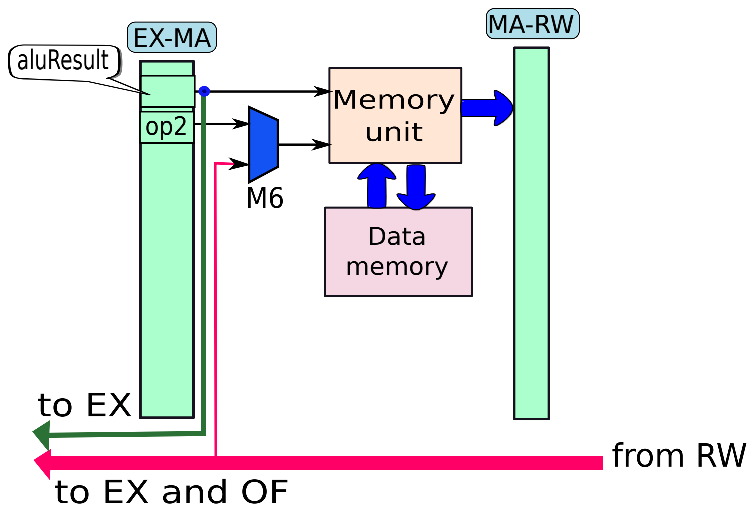


Figure 10.27: MA stage with forwarding

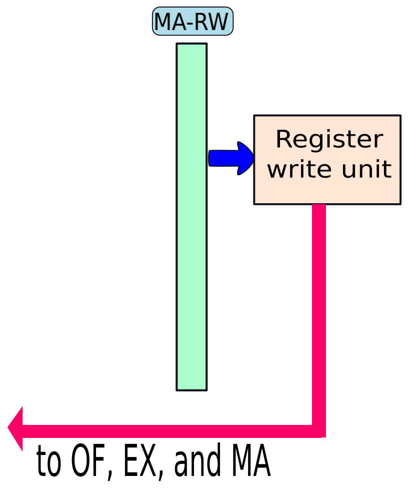


Figure 10.28: RW stage with forwarding

and OF stages, respectively.

Putting it All Together

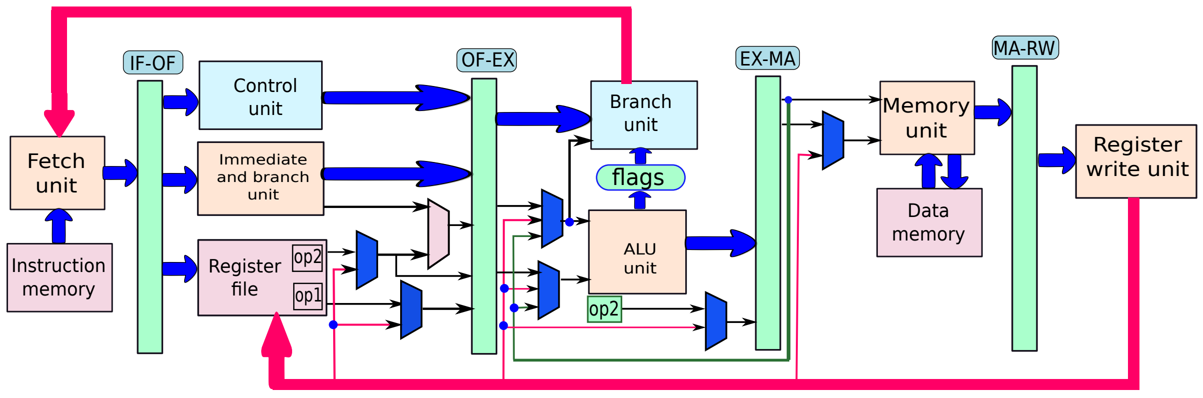


Figure 10.29: Pipelined data path with forwarding (abridged diagram)

Figure 10.29 puts all the pieces together and shows the pipeline with support for forwarding. To summarize, we need to add 6 multiplexers, and make some extra interconnections between units to pass the forwarded values. We envision a dedicated *forwarding unit* that computes the control signals for the multiplexers (not shown in the diagram). Other than these small changes, no other major change needs to be done to the data path.

We have been using an abridged diagram (similar to Figure 10.29) in our discussions on forwarding. The reader needs to note that the actual circuit has become fairly complicated now.

Along with the augmentations to the data path, we need to also add a dedicated forwarding unit to generate the control signals for the multiplexers. A detailed picture of the pipeline is shown in Figure 10.30.

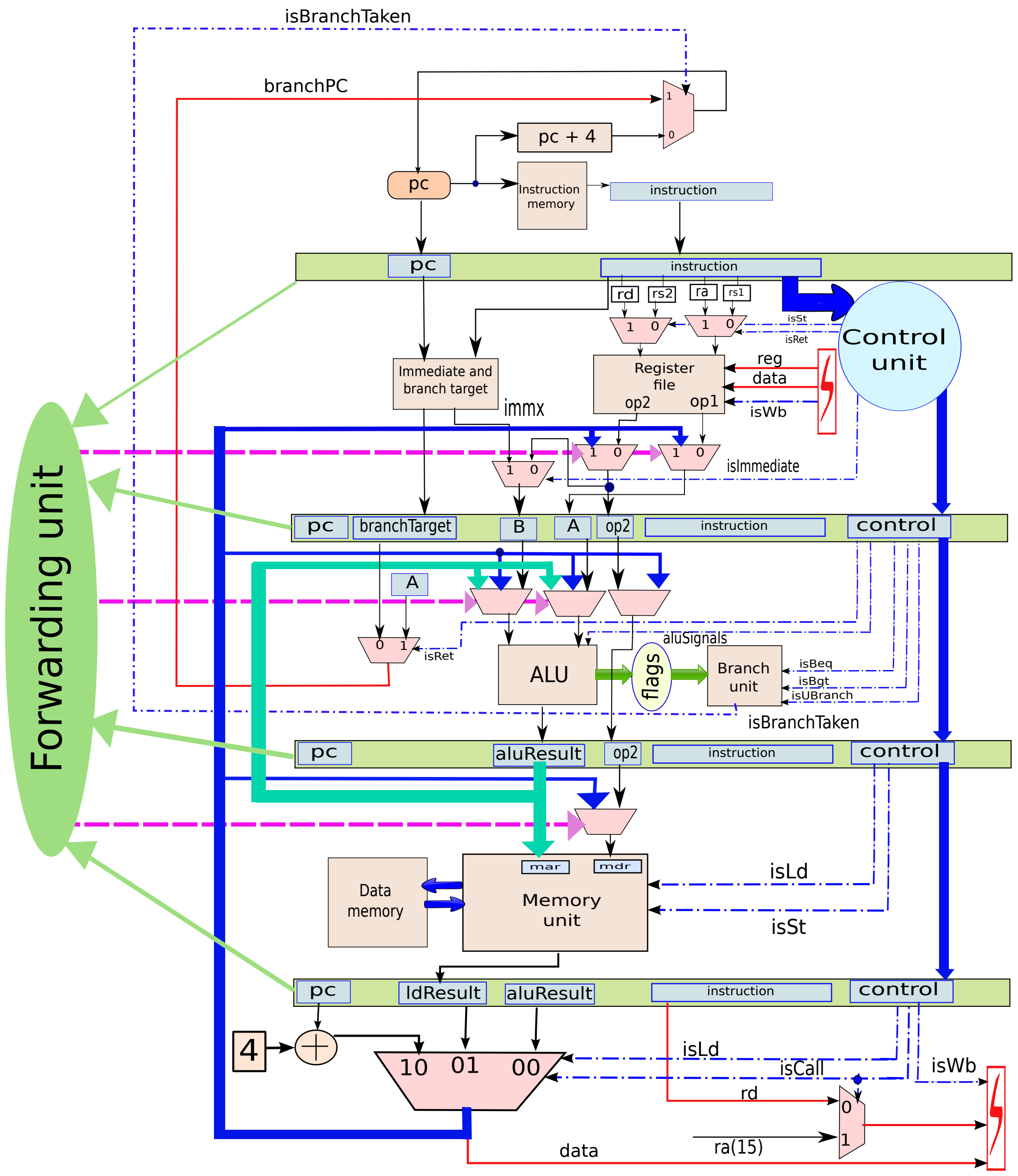


Figure 10.30: Pipelined data path with forwarding

Let us now add the interlock logic to our pipeline. We need the interlock logic for both the `Data-Lock` and `Branch-Lock` conditions. Note that now we have successfully handled all

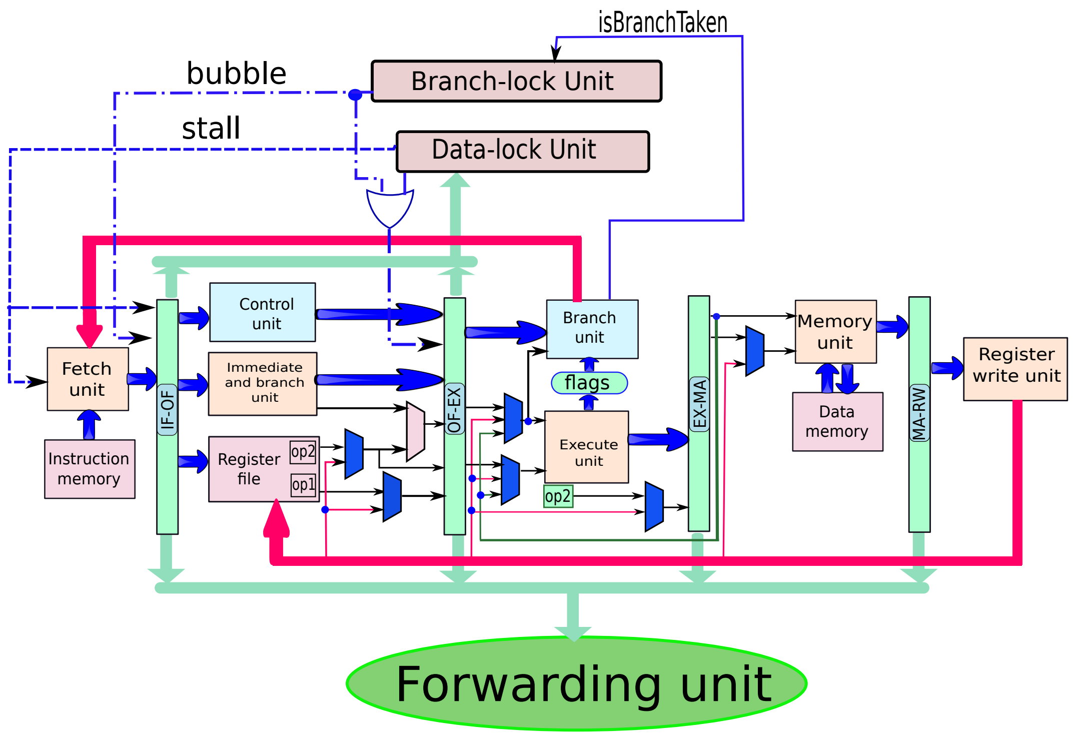


Figure 10.31: Pipelined processor with forwarding

RAW hazards other than the load-use hazard. In the case of a load-use hazard, we need to stall for only 1 cycle. This significantly simplifies our `Data-Lock` circuit. If there is a load instruction in the EX stage, then we need to check if there is a RAW data dependence between the load instruction, and the instruction in the OF stage. The only RAW hazard that we do not need to consider here is a load-store dependence, where the load writes to a register that contains the store value. We do not need to stall because we can forward the value to be stored from the RW to the MA stage. For all other data dependences, we need to stall the pipeline by 1 cycle by introducing a bubble. This will take care of the load-use hazard. The circuit for ensuring the `Branch-Lock` condition remains the same. Here also, we need to inspect the instruction in the EX stage, and if it is a taken branch, we need to invalidate the instructions in the IF and OF stages. Lastly, the reader should note that interlocks always take precedence over forwarding.

10.7.5 Forwarding Conditions

After designing the data path for supporting forwarding, let us design the control path. The only extra addition to the control path is the *forwarding unit*. This unit computes the values of the signals to control the *forwarding* multiplexers. Let us now discuss the design of the forwarding unit.

The Forwarding Unit

As shown in Figure 10.31 the forwarding unit receives inputs from all the four pipeline registers. They provide the contents of the instructions resident in the OF, EX, MA, and RW stages respectively. Based on the contents of the instructions, the forwarding unit computes the values of the control signals.

Salient Points

Let us now consider the four forwarding paths in our architecture – $RW \rightarrow OF$, $RW \rightarrow EX$, $MA \rightarrow EX$, and $RW \rightarrow MA$. We note that the distance between the producer and consumer stages for these four paths are 3, 2, 1, and 1 respectively. Alternatively, we can say that instruction number i, can get its inputs from instructions $i-1$, $i-2$, and $i-3$. The reader needs to note that there are two forwarding paths between adjacent stages (distance equal to 1).

Forwarding Paths with Distance Equal to 1

These forwarding paths are $MA \rightarrow EX$, and $RW \rightarrow MA$. We actually need both these forwarding paths. The reason is as follows. The $MA \rightarrow EX$ path is required for forwarding results between consecutive ALU instructions. The $RW \rightarrow MA$ path is required when the value of the input is generated in the MA stage, and it is also required in the MA stage. The only instruction that generates a value in the MA stage is the load instruction, and the only instruction that requires register operands in the MA stage, is the store instruction. Thus, we need to use the $RW \rightarrow MA$ forwarding path between a load instruction, and an immediately succeeding store instruction, when there is a register dependence. The following code snippet gives an example.

```
ld r1, 10[r2]
st r1, 20[r4]
```

Note that sometimes we might have a choice of forwarding paths ($MA \rightarrow EX$, or $RW \rightarrow MA$). The following code snippet shows an example.

```
[1]: add r1, r2, r3
[2]: st r1, 20[r4]
```

Here, instruction [1] is ready to forward the value of $r1$ when it reaches the MA stage. However, instruction [2] requires the value of $r1$ when instruction [1] reaches the RW stage. We can thus use either forwarding path ($MA \rightarrow EX$, or $RW \rightarrow MA$). Let us choose to use $RW \rightarrow MA$ forwarding in this case (also see Section 10.7.4). This optimization allows us to reduce a forwarding path between MA to EX for $op2$. This is also in accordance with tenet 2 mentioned in Section 10.7.2 that says that we should forward as late as possible.

Case of the *mov* Instruction

The other special case arises for the *mov* instruction. Since the EX stage does not produce its output value, we can theoretically use $RW \rightarrow MA$ forwarding for it. Ideally, if the consumer instruction in a load-use hazard, is a *mov* instruction, we should not have the necessity to stall the pipeline. However, for the purpose of simplicity, let us choose to treat a *mov* instruction as a regular ALU instruction, and choose to disregard any optimizations in this case.

Conflicts with Multiple Instructions

Let us look at our four forwarding paths: $RW \rightarrow OF$, $RW \rightarrow EX$, $MA \rightarrow EX$, and $RW \rightarrow MA$, again. We notice that the EX stage gets forwarded inputs from two stages – MA and RW. It is possible that the instruction in the EX stage has a conflict (RAW register dependence) with the instructions in both the MA and RW stages for the same input. In this case, we need to choose the input from the MA stage because it is an earlier instruction. Let us show an example.

```
[1]:add r1, r2, r3
[2]:sub r1, r4, r5
[3]:mul r8, r9, r1
```

In this case, when instruction [3] is in the EX stage, instruction [2] is in the MA stage, and instruction [1] is in the RW stage. The second source operand (value of register $r1$) needs to be forwarded. We need to get the value from the MA stage because instruction [2] will overwrite the value written by instruction [1]. We can design a simple circuit to give a higher priority to the MA stage than the RW stage while forwarding results to the EX stage. We leave this as an exercise for the interested reader.

Algorithms for Forwarding Conditions

We show the pseudo codes for the forwarding conditions. We need to first detect if a conflict exists for the first operand, which is typically the $rs1$ field of the instruction packet. In the case of a *ret* instruction, the first operand is the ra (return address) register. If a conflict exists, then we can potentially forward a value. For reasons of brevity, we do not show the code that disregards the case of forwarding if one of the instructions is a pipeline bubble.

Algorithm 6 shows the algorithm for detecting a conflict on the first operand. We first rule out the trivial cases in which instruction [A] does not read from any register, and [B] does not write to any register. Then, we set the first operand. It is equal to the $rs1$ field in the instruction packet. The only exception is the *ret* instruction whose first operand is the ra register. Similarly, the destination operand is always rd, with the *call* instruction being the only exception. Its destination operand is the return address register, ra. Then we detect a conflict in Line 15, and we return true if a conflict (RAW dependence) exists, otherwise we return false. We can use the output of Algorithm 6 to set the input of the forwarding multiplexers for the first operand.

Algorithm 7 shows the pseudo code of the algorithm for detecting conflicts for the second operand. We first rule out the trivial cases, in which [A] does not read any register and [B] does not write to any register. Then, we need to see if the second operand of [A] is an immediate. In this case, forwarding is not required. The second operand is typically equal to the $rs2$ field of the instruction packet. However, in the case of a store instruction, it is equal to the rd field of the instruction packet. Similarly, we find the destination register of instruction [B], and take care of the special case of the *call* instruction. We finally detect a conflict in Line 20. Note that we do not consider the load-use hazard, or `Branch-Lock` conditions in the forwarding logic, because we always assume that interlocks have higher priority over forwarding. Secondly, whenever we do not have a forwarding path, the forwarding conditions do not apply. Finally,

Algorithm 6: Conflict on the first operand (rs1/ra)

```
   Data: Instructions: [A] and [B] (possible forwarding: [B] $\rightarrow$ [A])
   Result: Conflict exists on rs1/ra (true), no conflict (false)
 1 if [A].opcode $\in$ (nop,b,beq,bgt,call,not,mov) then
       /* Does not read from the rs1 register */
 2     return false
 3 end
 4 if [B].opcode $\in$ (nop, cmp, st, b, beq, bgt, ret) then
       /* Does not write to any register */
 5     return false
 6 end
   /* Set the sources */
 7 src1 $\leftarrow$ $[A].rs1$
 8 if [A].opcode = ret then
 9     src1 $\leftarrow$ $ra$
10 end
   /* Set the destination */
11 dest $\leftarrow$ $[B].rd$
12 if [B].opcode = call then
13     dest $\leftarrow$ $ra$
14 end
   /* Detect conflicts */
15 if $src1 = dest$ then
16     return true
17 end
18 return false
```

Algorithm 7: Conflict on the second operand (rs2/rd)

```
Data: Instructions: [A] and [B] (possible forwarding: [B] → [A])
Result: Conflict exists on second operand (rs2/rd) (true), no conflict (false)
1  if [A].opcode ∈ (nop,b,beq,bgt,call) then
       /* Does not read from any register */
2      return false
3  end
4  if [B].opcode ∈ (nop, cmp, st, b, beq, bgt, ret) then
       /* Does not write to any register */
5      return false
6  end
   /* Check the second operand to see if it is a register */
7  if [A].opcode ∉ ( st) then
8      if [A].I = 1 then
9          return false
10     end
11 end
   /* Set the sources */
12 src2 ← [A].rs2
13 if [A].opcode = st then
14     src2 ← [A].rd
15 end
   /* Set the destination */
16 dest ← [B].rd
17 if [B].opcode = call then
18     dest ← ra
19 end
   /* Detect conflicts */
20 if src2 = dest then
21     return true
22 end
23 return false
```

in the case of multiple conflicting instructions, the forwarding unit needs to ensure that the correct value is forwarded.

Special Case of Forwarding from the Call Instruction

Let us consider the following code snippet.

```
call .function
..
...
.function:
ret
```

Here, we call a function and immediately return. In this case, the *call* instruction will still be in the pipeline, when the *ret* instruction enters the pipeline. Recall that the *call* instruction writes to register ra and the *ret* instruction reads from register ra. Moreover, the *call* instruction computes the value of ra, and writes it to the register file in the RW stage. We shall prove that this does not cause any correctness issues.

A *call* instruction is a taken branch. This means that when it enters the EX stage, the `Branch-Lock` circuitry will detect that it is a taken branch, and convert the instructions in the IF and OF stages to bubbles. Any instruction that requires the value of the ra register will at least be three stages behind the *call* instruction. This means that when the *call* instruction will reach the RW stage, the next valid instruction in the pipeline will be in the OF stage. If this is a *ret* instruction, or any other instruction that needs the value of the ra register, then it can simply get its value through the $RW \rightarrow OF$ forwarding path. Hence, the special case of forwarding from the call instruction is handled correctly.

10.8 Support for Interrupts/ Exceptions*

The process of building our pipelined processor is almost done. We just need to put the final piece together. We have up till now focused on building a fast pipeline that has interlocks for correctness and has forwarding for enhancing performance. We shall now discuss the interaction of our processor with external devices such as I/O devices and with specialized programs such as the operating system. The operating system is a master program that controls the behavior of other programs, the processor, and I/O devices. The standard mechanism for supporting the operating system, and other I/O devices, is through a mechanism called an *interrupt*. We shall have ample opportunities to discuss interrupts in Chapter 13. In this section, we discuss the implementation of an interrupt from the point of view of a pipeline.

10.8.1 Interrupts

The main idea of an interrupt is as follows. Assume that we click a key on a keyboard. The keyboard records the ASCII code of the clicked key, and then sends a message to the processor with the code of the key that was clicked. This message is known as an *interrupt*. After the processor receives an interrupt, it stops the execution of the currently executing program and jumps to a dedicated program known as the *interrupt handler*. The interrupt handler reads the value sent by the I/O device (in this case, the keyboard), and sends it to the program that handles the display device (monitor/ laptop screen). This program shows the character typed. For example, if the user clicks the character, 'a', then monitor ultimately shows an 'a' on the screen through a sequence of steps.

Definition 77
An interrupt is a signal sent by an I/O device to the processor. An interrupt is typically used to draw the attention of the processor to new inputs, or changes in the status of an I/O device. For example, if we click a key on a keyboard, then a new interrupt is generated and sent to the processor. Upon receiving an interrupt, the processor stops the execution of the currently executing program, and jumps to an interrupt handler routine*. This routine* processes *the interrupt, by reading the values sent by the I/O device, and performing any other action if required.*

10.8.2 Exceptions

Interrupts are not the only kind of events sent to the processor. Sometimes some actions of the program can generate interrupts. For example, if a program accesses an illegal memory address, then it is necessary to take corrective action. The memory system typically sends an interrupt to the processor. The processor in turn invokes the interrupt handler routine, which in turn calls dedicated modules in the operating system. Note that in this book, we shall use the terms interrupt handler and exception handler interchangeably. These modules either take some kind of corrective action, or terminate the program. Such kind of interrupts that are generated as a result of actions of the executing program are called *exceptions*. Readers familiar with languages such as Java can relate to the concept of exceptions. For example, in Java if we access an illegal array index such as -1, an exception is generated, and the processor jumps to a pre-specified location to take corrective action.

Definition 78
An exception *is a special event that is generated when the executing program typically performs an erroneous action, and it becomes necessary to take corrective action.*

10.8.3 Precise Exceptions

Let us now discuss how we need to handle interrupts and exceptions. The processor needs to clearly stop what it is currently doing, and jump to the interrupt handling routine. After handling the interrupt, and performing the desired action, it needs to come back and start from exactly the same point in the program, at which it had stopped. Let us now define the notion of a *precise exception*. The term "precise exception" is also used in the case of interrupts. We can think of it as a generic term for all kinds of interrupts and exceptions.

Definition of Precise Exceptions

At any point of time, a program will typically have multiple instructions in the pipeline with different PCs. When the processor encounters an interrupt, it needs to branch to the starting location of the interrupt handler. To facilitate this process, it can have an interrupt handler table. This table typically stores a list of interrupt types, and the starting PCs of their interrupt handlers. The processor uses this table to branch to the appropriate interrupt handler. After finishing the processing of the interrupt handler, it needs to come back to exactly the same point in the original program. In other words, the original program should not be aware of the fact that another program such as the interrupt handler executed in the middle. This entire process needs to be orchestrated very carefully.

Let us elaborate. Assume that a program, P, is executing on the processor. Let us record all its dynamic instructions that leave the pipeline after successfully completing their execution, and number them $I_1, I_2, \ldots I_n$. A *dynamic instruction* is the instance of an instruction created by the processor. For example, if a loop has 5 instructions, and executes 100 times, then we have 500 dynamic instructions. Furthermore, an instruction *completes* its execution when it finishes its job and updates the state of the processor (registers or memory). A store instruction completes in the MA stage, and instructions with a destination register complete in the RW stage. All other instructions, are assumed to complete in the MA stage. The *nop* instruction is excluded from this discussion. Let I_k be the last instruction in P that completes its execution before the first instruction in the interrupt handler completes its execution. We wish to ensure that at the time that I_k leaves the pipeline, all the instructions in P before I_k have completed their execution and left the pipeline, and no instruction in P after I_k has completed or will complete its execution before the program resumes. Let the set of completed instructions at this point of time (when I_k leaves the pipeline) be $\mathcal{C}$. Formally, we have:

$$I_j \in \mathcal{C} \Leftrightarrow (j \leq k) \tag{10.1}$$

An interrupt or exception implemented in this manner is said to be *precise*.

Definition 79
An interrupt or exception is precise *if the following conditions are met:*

Condition 1: *Let I_k be the last dynamic instruction in the original program, P, that completes its execution before the first instruction in the interrupt handler completes its execution. Let I_k leave the pipeline at time, τ. At τ, all instructions I_j $(j < k)$ have also completed their execution.*

Condition 2: *No instruction after I_k in P completes its execution before all the instructions in the interrupt handler complete, and the program resumes execution.*

Condition 3: *After the interrupt handler finishes, we can seamlessly start executing all the instructions starting from I_k (if it has not completed successfully) or I_{k+1}.*

When the interrupt handler returns, it needs to start executing instruction, I_{k+1}. For some special types of interrupts/ exceptions it might be required to re-execute I_k. Secondly, the register state (values of all the registers) needs to be restored before the original program, P, starts executing again. We can thus ensure that a processor can seamlessly switch to an interrupt handler and back without violating the correctness of the program.

Marking Instructions

Let us now discuss how to implement precise exceptions. Let us look at the three conditions in Definition 79 in more detail.

When an interrupt arrives, we can at the most have 5 instructions in the pipeline. We can designate one of these instructions as the last instruction before the interrupt handler executes such that the three conditions outlined in Definition 79 are satisfied. Now, we cannot designate the instruction in the RW stage as the last instruction (I_k) because the instruction in the MA stage might be a store instruction. In the current cycle it will complete its execution, and thus condition 2 will get violated. However, we are free to designate instructions in any of the four other stages as the last instruction. Let us decide to mark the instruction in the MA stage as the last instruction.

Now, let us look at exceptions. Exceptions are typically caused by the erroneous execution of instructions. For example, in the IF stage we might fetch from an illegal address, try to perform an illegal arithmetic operation in the EX stage, or write to a non-existent address in the MA stage. In these situations it is necessary to take corrective action. The processor needs to invoke a dedicated exception handler. For example, a very common type of exception is a page fault as we shall discuss in Chapter 11. A page fault occurs when we try to read or write a memory address in a 4 KB block of memory for the first time. In this case, the operating system needs to read the 4 KB block from the hard disk and copy it to memory. The faulting instruction executes again, and it succeeds the second time. In this case, we need to re-execute the exception causing instruction I_k, and needless to say we need to implement a *precise* exception. To properly take core of exceptions, the first step is to mark an instruction, immediately after it causes an exception. For example, if we try to fetch from an uninitialized or illegal address we mark the instruction in the IF stage.

Making a Marked Instruction Proceed to the End of the Pipeline

Now, that we have marked instructions, we need to ensure two conditions. The first is that all the instructions before the *marked* instruction need to complete. The second is that all the instructions after the *marked* instruction should not be allowed to write to the register file, or the main memory. We should ideally not allow any writes to the $flags$ register also. However, it is difficult to implement this functionality, because we are typically aware of interrupts at the end of the clock cycle. We shall devise an ingenious solution to handle updates to the $flags$ register later.

For implementing a precise exception, we need to add an *exception* unit to our pipeline. Its role is to process interrupts and exceptions. Once an instruction is marked, it needs to let the exception unit know. Secondly, we envision a small circuit that sends a code identifying the exception/ interrupt to the exception unit. Subsequently, the exception unit needs to wait for the marked instruction to reach the end of the pipeline such that all the instructions before it complete their execution. Instructions fetched after the marked instruction need to be converted into bubbles. This needs to be done to ensure that instructions after a marked instruction do not complete. Once, the marked instruction reaches the end of the pipeline, the exception unit can load the PC with the starting address of the interrupt handler. The interrupt or exception handler can then begin execution. This mechanism ensures that asynchronous events such as interrupts and exceptions remain precise. Now, we have a mechanism to seamlessly transition to executing interrupt handlers. Sadly, we still do not have a mechanism to come back to exactly the same point in the original program, because we have not remembered the point at which we had left.

10.8.4 Saving and Restoring Program State

Let us define the term *program state* as the state of all the registers and memory elements associated with the program. In specific, the program state, comprises the contents of the register file, PC, $flags$ register, and main memory.

Definition 80
The term program state *is defined as the state of all the registers and memory elements associated with the program. In specific, the program state, comprises the contents of the register file, PC, $flags$ register, and main memory.*

We need to find effective means of saving and restoring the state of the executing program. Let us start by stating that we do not need a method to save and restore the state of main memory because the assumption is that the interrupt handler uses a different region of main memory. We shall discuss methods to enforce a separation of memory regions between programs in Chapter 11. Nonetheless, the bottom line is that there is no unintended overlap of the memory regions of the executing program and the interrupt handler. In the case of exceptions, the interrupt handler might access some parts of the memory space of the program such that it can add some data that the program requires. One such example of exceptions is a page fault. We will have ample opportunities to discuss page faults in Chapter 11.

Hence, we need to explicitly take care of the PC, the $flags$ register, and the set of registers. The state of all of these entities is known as the *context* of a program. Hence, our problem is to successfully save and retrieve the context of a program upon an interrupt.

Definition 81
The context *of a program refers to the values of the PC, the $flags$ register, and the values contained in all the registers.*

The *oldPC* Register

Let us add an NPC field for the next PC in the instruction packet. By default, it is equal to $PC + 4$. However, for branch instructions that are taken, the NPC field contains the branch target. We envision a small circuit in the EX stage that adds the branch target, or $PC + 4$ to the NPC field of the instruction packet. Recall that the instruction packet gets passed from one stage to the next in a pipeline. Once a marked instruction reaches the RW stage, the exception unit looks up a small internal table indexed by the interrupt/ exception code. For some types of interrupts such as I/O events, we need to return to the next PC ($PC + 4$ or the branch target). This value is stored in the NPC field of the MA-RW pipeline register. However, for some types of exceptions such as page faults, it is necessary to re-execute the faulting instruction once again. A page fault happens because a certain memory location is not loaded with its data. The interrupt handler (for a page fault) needs to load the data of the memory location by fetching values from the hard disk, and then re-execute the instruction. In this case, we need to return to the PC of the marked instruction. In either case, the exception unit transfers the correct return address to an internal *oldPC* register, and then starts fetching instructions for the interrupt handler.

Spilling General Purpose Registers

We need a mechanism to save and restore registers akin to spilling and restoring registers as in the case of function calls. However, there is an important difference in the case of interrupt handlers. Interrupt handlers have their own stacks that are resident in their private memory regions. To use the stack pointer of an interrupt handler, we need to load its value into *sp*. This step will overwrite the previous value, which is the value of the stack pointer of the program. Hence, to avoid losing the value of the stack pointer of the program, we add another register called *oldSP*. The interrupt handler first transfers the contents of *sp* to *oldSP*. Subsequently, it loads *sp* with the value of its stack pointer and then spills all the registers excluding *sp* to its stack. At the end of this sequence of steps, it transfers the contents of *oldSP* to the stack.

The *oldFlags* Register

The only part of the program state that we have not saved up till now is the $flags$ register. Let us assume that the $flags$ register is a 32-bit register. Its lower 2 bits contain the values, $flags.E$ and $flags.GT$ respectively. Moreover, let us add a $flags$ field to the instruction packet. Instructions other than the *cmp* instruction write the contents of the $flags$ register to the $flags$ field in the instruction packet, in the EX stage. The *cmp* instruction writes the updated value of the $flags$ register to the $flags$ field in the EX stage and moves to the subsequent stages. When a marked instruction reaches the RW stage, the exception unit extracts the contents of the $flags$ field in the instruction packet, and saves it in the *oldFlags* register. The *oldFlags* register is a special register that is visible to the ISA, and helps store the last value of the $flags$ register that a valid instruction in the program had seen.

Saving and Restoring Program State

For saving the program state, the interrupt handler contains assembly routines to save the general purpose registers (excluding *sp*) and the *oldSP*, *oldFlags*, and *oldPC* registers. We save all of these values in the stack of the interrupt handler. Likewise, we can restore program state in almost the reverse order. We restore the value of *oldPC*, the $flags$ register, the general purpose registers, and the stack pointer. As the last step, we need to transfer the contents of *oldPC* to PC such that we can resume executing the original program.

Privileged Instructions

We have added the following special registers namely *oldPC*, *oldSP*, *oldFlags* and *flags*. Note that we had the *flags* register before also. However, it was not accessible as a register. Next, we add a special category of instructions called *privileged instructions* that are only accessible to specialized programs such as operating systems, and interrupt handlers. The first privileged instruction that we introduce is *movz*. It transfers values between regular registers and the special registers (*oldPC*, *oldSP*, *oldFlags*, and *flags*).

The other privileged instruction that we introduce in this section is *retz*. It reads the value of *oldPC*, and transfers its contents to PC. In other words, we jump to the location contained in *oldPC*. We do not allow instructions to directly transfer the values of special registers to and from memory, because we have to create privileged versions of both load and store instructions. We wish to avoid creating two additional instructions.

Definition 82
A privileged instruction *is a special instruction that has access to the internals of the processor. It is typically meant to be used only by operating system programs such as the kernel (core of the operating system), device drivers (programs to interact with I/O devices), and interrupt handlers.*

To implement the *movz* instruction, we add a new instruction opcode. Recall that we introduced only 21 instructions in the *SimpleRisc* instruction set. We can afford to have 11 more instructions in the ISA. *movz* uses the same register format based encoding as the *mov* instruction. However, it sees a different view of registers. The registers visible to privileged instructions, and their identifiers are shown in Table 10.2.

Register	Encoding
$r0$	0000
oldPC	0001
oldSP	0010
flags	0011
oldFlags	0100
sp	1110

Table 10.2: View of registers for privileged instructions

Privileged instructions use a different register encoding. They can only see the four special registers, $r0$, and *sp*. We need to make a small modification to the OF and RW stages to implement the *movz* instruction. The first is that we need to have a circuit in the OF stage to quickly find out if the opcode of an instruction is *movz*. We can use a fast circuit similar to the one that we use to find out if an instruction is a store. Then, we can choose the right set of register inputs from either the normal register file, or from one of the privileged registers using multiplexers. Similarly, in the RW stage, we can choose to either write the value in the normal register file, or in one of the special registers, again, with the help of additional multiplexers. For the sake of brevity, we do not show the circuit. We leave implementing *movz* as an exercise for the reader. We can implement *retz* on the same lines as the *ret* instruction. The only difference is that instead of getting the return value from the *ra* register, we get it from the *oldPC* register. Note that we will also require forwarding and interlock logic that takes special registers into account. The pseudocode of the forwarding and interlock logic needs to be updated.

Let us summarize the discussion in terms of two new concepts that we have learned. The first is the notion of *privileged instructions.* These instructions are typically used by interrupt handlers, and other modules of the operating systems. They have more visibility into the internals of the processor. Since they are very powerful, it is not a good idea to give programmers the ability to invoke them. They might corrupt system state, and introduce viruses. Hence, most systems typically disallow the usage of privileged instructions by normal programs. Most processors have a register that contains the *current privilege level* (CPL). It is typically 1 for user programs, and 0 for operating system programs such as interrupt handlers. There is a privilege level change, when we switch to processing an interrupt handler (1 to 0), and when we execute the *retz* instruction to return to a user program (0 to 1). Whenever, we execute a privileged instruction, the processor checks the CPL register, and if the program is not allowed to execute the instruction, then an exception is flagged. The operating system typically terminates the program, since it may be a virus.

Definition 83
Most processors have a register that contains the current privilege level *(CPL). It is typically 1 for user programs, and 0 for operating system programs such as interrupt handlers. We are allowed to execute privileged instructions, only when the CPL is equal to 0.*

The second important concept is the notion of different register views for different instructions, or different pieces of code. This concept is known as a *register window*, and was pioneered by the Sun Ultrasparc processors. The Sun processors used different register windows for different functions. This allowed the compiler to avoid costly register spills. Here, we use register windows to separate the set of registers that can be accessed by user programs and the interrupt handlers. The interrupt handlers can see all the special registers and two regular registers ($r0$ and sp).

Definition 84
A register window *is defined as the set of registers that a particular instruction or function can access. For example, in our case, privileged instructions can access only six registers,*

out of which four are special registers. In comparison, regular instructions have a register window that contains all the 16 general purpose registers, but no special register.

10.8.5 *SimpleRisc* Assembly Code of an Interrupt Handler

Let us now quickly conclude our discussion by showing the assembly code of an interrupt handler. The code for saving the context is shown in Figure 10.32, and the code for restoring the context and returning to the user program is shown in Figure 10.33. We assume that the stack pointer for the interrupt handler starts at : 0x FF FF FF FC.

10.8.6 Processor with Support for Exceptions

Figure 10.34 shows an abridged diagram of the data path with support for exceptions. We have added an exception unit that takes inputs from all the pipeline registers. Whenever, an instruction detects an exception, or an interrupt is detected, the exception unit is notified. The exception unit proceeds to *mark* an instruction as the last instruction. It waits till the marked instruction leaves the pipeline, and concurrently converts all the instructions fetched after the marked instruction to bubbles. Finally, when the marked instruction reaches the RW stage, the exception unit stores the PC, or NPC (next PC) value in the *oldPC* register. It also saves the $flags$ field in the instruction packet to the *oldFlags* register. We add four registers namely *oldPC*, *oldSP*, *oldFlags*, and $flags$. The ALU immediately updates the $flags$ register if it processes a *cmp* instruction. The RW stage can also write to the $flags$ register. These four registers are bundled with the regular register file. We call the new structure as the *register unit* (shown in Figure 10.34). We do not show the multiplexers to choose between the inputs from the register file, and the special registers. We assume that the multiplexers are embedded inside the register unit.

10.9 Performance Metrics

10.9.1 The Performance Equation

Let us now discuss the performance of our pipelined processor. We need to first define the meaning of "performance" in the context of processors. Most of the time, when we look up the specifications of a laptop or smartphone, we are inundated with a lot of terms such as the clock frequency, RAM, and hard disk size. Sadly, none of these terms are directly indicative of the performance of a processor. The reason that the performance is never explicitly mentioned on the label of a computer, is because the term "performance" is rather vague. The term *performance of a processor* is always with respect to a given program or set of programs. This is because processors perform differently with respect to different programs.

Given a program, P, let us try to quantify the performance of a given processor. We say that processor A performs better than processor B, if it takes less time for P to execute P on A than on B. Thus, quantifying performance with respect to a given program is very simple. We measure the time it takes to run the program, and then compute its reciprocal. This number

Saving the context

```
/* save the stack pointer */
movz oldSP, sp
mov sp, 0x FF FC

/* spill all the registers other than sp*/
st r0, -4[sp]
st r1, -8[sp]
st r2, -12[sp]
st r3, -16[sp]
st r4, -20[sp]
st r5, -24[sp]
st r6, -28[sp]
st r7, -32[sp]
st r8, -36[sp]
st r9, -40[sp]
st r10, -44[sp]
st r11, -48[sp]
st r12, -52[sp]
st r13, -56[sp]
st r15, -60[sp]

/* save the stack pointer */
movz r0, oldSP
st r0, -64[sp]

/* save the flags register */
movz r0, oldFlags
st r0, -68[sp]

/* save the oldPC */
movz r0, oldPC
st r0, -72[sp]

/* update the stack pointer */
sub sp, sp, 72

/* code of the interrupt handler */
....
....
....
```

Figure 10.32: *SimpleRisc* assembly code for saving the context

```
Restoring the context
/* update the stack pointer */
add sp, sp, 72

/* restore the oldPC register */
ld r0, -72[sp]
movz oldPC, r0

/* restore the flags register */
ld r0, -68[sp]
movz flags, r0

/* restore all the registers other than sp*/
ld r0, -4[sp]
ld r1, -8[sp]
ld r2, -12[sp]
ld r3, -16[sp]
ld r4, -20[sp]
ld r5, -24[sp]
ld r6, -28[sp]
ld r7, -32[sp]
ld r8, -36[sp]
ld r9, -40[sp]
ld r10, -44[sp]
ld r11, -48[sp]
ld r12, -52[sp]
ld r13, -56[sp]
ld r15, -60[sp]

/* restore the stack pointer */
ld sp, -64[sp]

/* return to the program */
retz
```

Figure 10.33: *SimpleRisc* assembly code for restoring the context

can be interpreted to be proportional to the performance of the processor with respect to the program.

Let us first compute the time(τ) it takes to run program P.

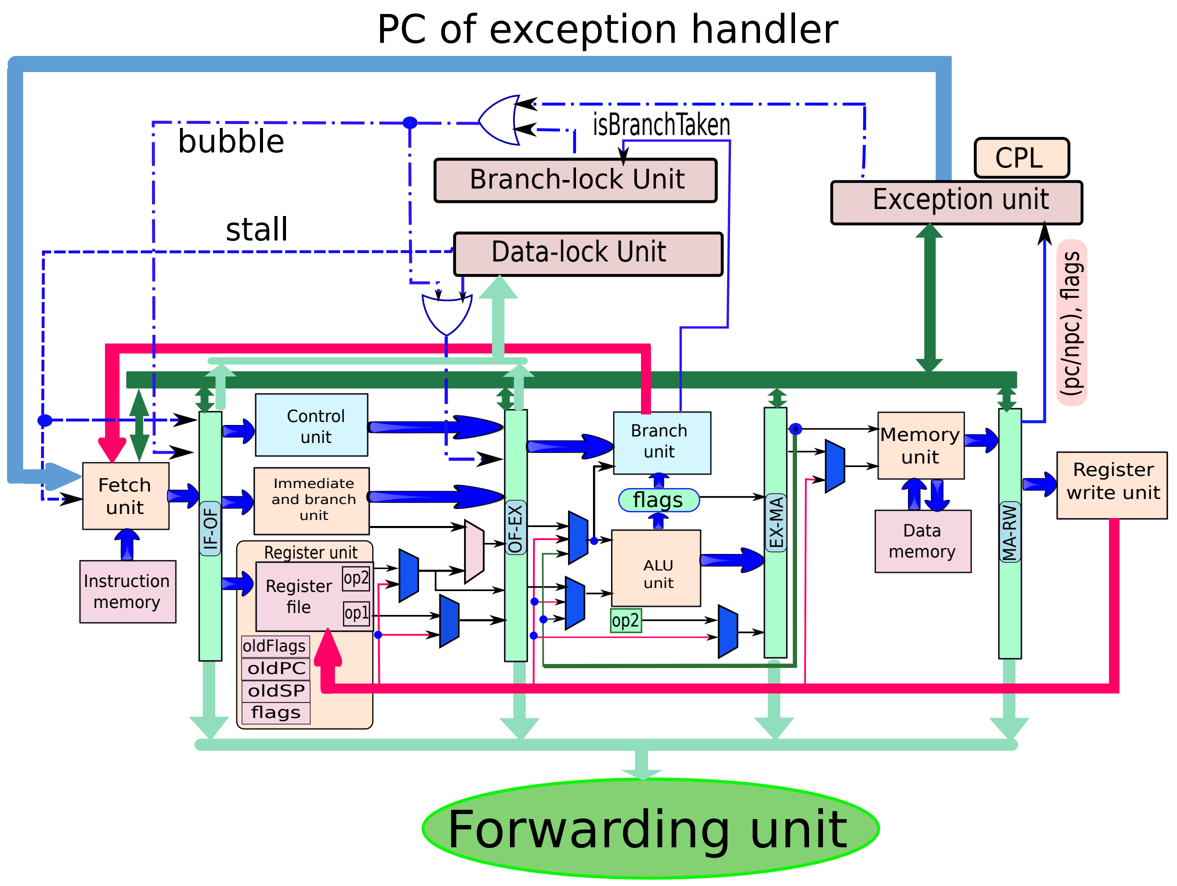


Figure 10.34: Pipelined data path with support for exceptions

$$\begin{aligned}
\tau &= \#seconds \\
&= \frac{\#seconds}{\#cycles} \times \frac{\#cycles}{\#instructions} \times (\#instructions) \\
&= \underbrace{\frac{\#seconds}{\#cycles}}_{1/f} \times \underbrace{\frac{\#cycles}{\#instructions}}_{CPI} \times (\#instructions) \\
&= \frac{CPI \times \#insts}{f}
\end{aligned} \tag{10.2}$$

The number of cycles per second is the processor's clock frequency (f). The average number of cycles per instruction is known as the *CPI*, and its inverse (number of instructions per cycle) is known as the *IPC*. The last term is the number of instructions (abbreviated to #insts). Note that this is the number of dynamic instructions, or, alternatively, the number of instructions that the processor actually executes. Note that it is NOT the number of instructions in the program's executable file.

Definition 85

Static Instruction *The binary or executable of a program contains a list of* instructions*. Each such instruction is a* static instruction*.*

Dynamic Instruction *A* dynamic instruction *is the instance of a static instruction, which is created by the processor when an instruction enters the pipeline.*

Definition 86

CPI *Cycles per instruction*

IPC *Instructions per cycle*

We can now define the performance P as a quantity that is inversely proportional to the time, τ. Equation 10.3 is known as the **Performance Equation**.

$$P \propto \frac{IPC \times f}{\#insts} \tag{10.3}$$

We can thus quickly conclude that the performance of a processor with respect to a program is proportional to the IPC, and frequency, and inversely proportional to the number of instructions.

Let us now look at the performance of a single cycle processor. Its CPI is equal to 1 for all instructions. The performance is thus proportional to $f/\#insts$. This is a rather trivial result. It says that as we increase the frequency, a single cycle processor keeps getting faster proportionally. Likewise, if we are able to reduce the number of instructions in our program by a factor of X, then the performance also increases by a factor of X. Let us consider the performance of a pipelined processor. The analysis is more complicated, and the insights are very profound.

10.9.2 Performance of an Ideal Pipelined Processor

Let us look at the three terms in the performance equation (Equation 10.3), and consider them one by one. Let us first consider the number of instructions.

Number of Instructions

The number of instructions in a program is dependent on the intelligence of the compiler. A really smart compiler can reduce instructions by choosing the right set of instructions from the

ISA, and by using smart code transformations. For example, programmers typically have some code, which can be categorized as *dead code.* This code has no effect on the final output. A smart compiler can remove all the dead code that it can find. Another source of additional instructions is the code to spill and restore registers. Compilers often perform *function inlining* for very small functions. This optimization dynamically removes such functions and pastes their code in the code of the calling function. For small functions, this is a very useful optimization since we are getting the rid of the code to spill and restore registers. There are many more compiler optimizations that help in reducing code size. The reader is referred to [Aho et al., 2006, Muchnick, 1997] for a detailed discussion on compiler design. For the rest of this section, we shall assume that the number of instructions is a constant. Let us exclusively focus on the hardware aspect.

Computing the Total Number of Cycles

Let us assume an ideal pipeline that does not need to insert any bubbles, or stalls. It will be able to complete one instruction every cycle, and thus will have a CPI of 1. Let us assume a program containing n instructions, and let the pipeline have k stages. Let us compute the total number of cycles it will take for all the n instructions to leave the pipeline.

Let the first instruction enter the pipeline in cycle 1. It leaves the pipeline in cycle k. Henceforth, one instruction will leave the pipeline every cycle. Thus, after $(n-1)$ cycles, all the instructions would have left the pipeline. The total number of cycles is therefore, $n+k-1$. The CPI is equal to:

$$CPI = \frac{n+k-1}{n} \tag{10.4}$$

Note that the CPI tends to 1, as n tends to ∞.

Relationship with the Frequency

Let the maximum amount of time that an instruction takes to finish its execution on a single cycle processor be t_{max}. This is also known as the total amount of *algorithmic work.* We are ignoring the delays of pipeline registers while computing t_{max}. Now, let us divide the data path into k pipeline stages. We need to add $k-1$ pipeline registers. Let the delay of a pipeline register be l. If we assume that all the pipeline stages are balanced (do the same amount of work, and take the same amount of time), then the time that the slowest instruction will take to finish its work in a stage is equal to $\frac{t_{max}}{k}$. The total time per stage is equal to the circuit delay and the delay of a pipeline register.

$$t_{stage} = \frac{t_{max}}{k} + l \tag{10.5}$$

Now, the minimum clock cycle time has to be equal to the delay of a pipeline stage. This is because, the assumption while designing a pipeline is that each stage takes exactly one clock cycle. We thus have the minimum clock cycle time (t_{clk}), or the maximum frequency (f) equal to:

$$t_{clk} = \frac{1}{f} = \frac{t_{max}}{k} + l \tag{10.6}$$

Performance of a Pipeline

Let us now compute the performance of this pipeline, and make a simplistic assumption that performance is equal to (f / CPI) because the number of instructions is a constant(n).

$$\begin{aligned}
P &= \frac{f}{CPI} \\
&= \frac{\frac{1}{\frac{t_{max}}{k}+l}}{\frac{n+k-1}{n}} \\
&= \frac{n}{(t_{max}/k + l) \times (n+k-1)} \\
&= \frac{n}{((n-1)t_{max}/k + (t_{max} + ln - l) + lk}
\end{aligned} \tag{10.7}$$

Let us try to maximize performance by choosing the right value of k. We have:

$$\begin{aligned}
&\frac{\partial\left((n-1)t_{max}/k + (t_{max} + ln - l) + lk\right)}{\partial k} = 0 \\
\Rightarrow &- \frac{(n-1)t_{max}}{k^2} + l = 0 \\
\Rightarrow &k = \sqrt{\frac{(n-1)t_{max}}{l}}
\end{aligned} \tag{10.8}$$

Equation 10.8 provides a theoretical estimate of the optimal number of pipeline stages as a function of the latch delay (l), the total algorithmic work (t_{max}), and the number of instructions (n). Let us gauge the trends predicted by this equation. The first is that as we increase the number of instructions, we can afford more pipeline stages. This is because the startup delay of k cycles, gets nullified when there are more instructions. Secondly, as we increase the amount of algorithmic work (t_{max}), we need a deeper pipeline. More is the number of pipeline stages, less is the amount of work we need to do per stage. We can thus have a higher frequency, and thus have a higher instruction throughput. Lastly, the optimal number of stages is inversely proportional to $\sqrt{l}$. As we increase the latch delay, we start wasting more time inserting and removing data from latches. Hence, it is necessary to adjust the number of pipeline stages with the latch delay. If the latches are very slow, we need to reduce the number of pipeline stages also such that we do not waste a lot of time in adding, and removing data from pipeline latches.

Sadly, an ideal pipeline does not exist in practice. This means that they do not have a CPI equal to $(n+k-1)/n$. Almost all programs have dependences between instructions, and thus it becomes necessary to insert bubbles in the pipeline. Inserting bubbles increases the CPI from the ideal CPI computed in Equation 10.4. Equation 10.8 provides us with interesting insights. However, the reader needs to note that it is hypothetical. It predicts that the optimal number of stages approaches infinity, for very large programs. This is unfortunately not the case in practical scenarios.

10.9.3 Performance of a Non-Ideal Pipeline

Mathematical Characterization

We need to incorporate the effect of stalls in the CPI equation. Let us assume that the number of instructions (n) is very large. Let the ideal CPI be CPI_{ideal}. In our case, $CPI_{ideal} = 1$. We have:

$$CPI = CPI_{ideal} + stall\_rate \times stall\_penalty \tag{10.9}$$

Example 140
Assume that the ideal CPI is 1. Assume that 10% of the instructions suffer a load-use hazard, and 20% of the instructions are taken branches. Find the CPI of the program.
Answer: *We need to insert 1 bubble for a load-use hazard, and 2 bubbles for a taken branch. Thus, the average number of bubbles that we need to insert per instruction is equal to: 0.1 * 1 + 0.2 * 2 = 0.5. Thus,*

$$CPI_{new} = CPI_{ideal} + 0.5 = 1 + 0.5 = 1.5$$

Example 141
Compare the performance of two programs, P_1 and P_2. Assume that the ideal CPI for both of them is 1. For P_1, 10% of the instructions have a load-use hazard, and 15% of its instructions are taken branches. For P_2, 20% of the instructions have a load-use hazard, and 5% of its instructions are taken branches.
Answer:

$$CPI_{P_1} = 1 + 0.1 * 1 + 0.15 * 2 = 1.4$$

$$CPI_{P_2} = 1 + 0.2 * 1 + 0.05 * 2 = 1.3$$

The CPI of P_2 is less than the CPI of P_1. Hence, P_2 is faster.

The final CPI is equal to the sum of the ideal CPI and number of mean stall cycles per instruction. The mean stall cycles per instruction is equal to the product of the average stall rate per instruction multiplied by the average number of bubbles that we need to insert per stall (*stall_penalty*). The *stall_rate* term is typically a function of the nature of dependences across instructions in a program. The *stall_penalty* term is also typically dependent on the design of the pipeline, and its forwarding paths. In our case, we need to stall for at most one cycle for RAW hazards, and for 2 cycles for taken branches. However, pipelines with more stages might have different behaviors. Let us now try to model this pipeline mathematically.

We assume that the *stall_rate* is only dependent on the program, and the *stall_penalty* is proportional to the number of stages in a pipeline. This assumption is again not completely correct. However, it is good enough for developing a coarse mathematical model. The reason, we assume that *stall_penalty* is proportional to the number of stages is because, we assume that we create deeper pipelines by essentially splitting the stages of our simple pipeline further. For example, we can pipeline the functional units. Let us assume that we divide each stage, into two sub-stages. Then, we need to stall for 2 cycles on a load-use hazard, and stall for 4 cycles for a taken branch.

Let us thus assume that CPI = $(n + k - 1)/n + rck$, where r and c are constants, and k is the number of pipeline stages. r is equal to the average number of stalls per instruction (*stall_rate*). We assume that the *stall_penalty* $\propto k$, or alternatively, *stall_penalty* $= ck$, where c is the constant of proportionality.

We thus have:

$$\begin{aligned} P &= \frac{f}{CPI} \\ &= \frac{\frac{1}{t_{max}/k+l}}{(n+k-1)/n + rck} \\ &= \frac{n}{((n-1)t_{max}/k + (rcnt_{max} + t_{max} + ln - l) + lk(1+rcn)} \end{aligned} \tag{10.10}$$

To maximize performance, we need to minimize the denominator. We get:

$$\begin{aligned} &\frac{\partial\left((n-1)t_{max}/k + (rcnt_{max} + t_{max} + ln - l) + lk(1+rcn)\right)}{\partial k} = 0 \\ \Rightarrow &- \frac{(n-1)t_{max}}{k^2} + l(1+rcn) = 0 \\ \Rightarrow &k = \sqrt{\frac{(n-1)t_{max}}{l(1+rcn)}} \approx \sqrt{\frac{t_{max}}{lrc}} \quad (as\ n \to \infty) \end{aligned} \tag{10.11}$$

Equation 10.11 is more realistic than Equation 10.8. It is independent of the number of instructions. The implicit assumption is that the number of instructions tends to infinity, because in most programs, we execute billions of instructions. Akin to Equation 10.8, the optimal number of pipeline stages is proportional to $\sqrt{t_{max}}$, and inversely proportional to $\sqrt{l}$. Additionally, $k \propto 1/\sqrt{rc}$. This means that as the penalty for a stall increases, or the number of stall events per instruction increase, we need to use less pipeline stages.

Let us now find the performance for the optimal number of pipeline stages. In Equation 10.10, we assume that $n \to \infty$. Thus, $(n + k - 1)/n \to 1$. Hence, we have:

$$
\begin{aligned}
P_{ideal} &= \frac{1}{(t_{max}/k + l) \times (1 + rck)} \\
&= \frac{1}{t_{max}/k + l + rct_{max} + lrck} \\
&= \frac{1}{t_{max} \times \sqrt{\left(\frac{lrc}{t_{max}}\right)} + l + rct_{max} + lrc \times \sqrt{\left(\frac{t_{max}}{lrc}\right)}} \\
&= \frac{1}{rct_{max} + 2\sqrt{lrct_{max}} + l} \\
&= \frac{1}{\left(\sqrt{rct_{max}} + \sqrt{l}\right)^2}
\end{aligned}
\tag{10.12}
$$

Implications of Equation 10.11 and Equation 10.12

Let us now study the different implications of the result regarding the optimal number of pipeline stages.

Implication 1
The crucial implication of these results is that for programs with a lot of dependences, we should use processors with a lesser number of pipeline stages. Inversely, for programs that have high IPC (fewer dependences across instructions), we should use processors that have deeper pipelines.

Implication 2
Let us compare two versions of our pipeline. One version uses interlocks for all dependences, and the other uses forwarding. For the pipeline with forwarding, the *stall_penalty* is much lower. Consequently, the value of the constant, c, is smaller in the case of the pipeline with forwarding turned on. This means that a pipeline with forwarding ideally requires more pipeline stages for optimal performance. As a general rule, we can conclude that as we increase the amount of forwarding in a pipeline, we should make it deeper.

Implication 3
The optimal number of pipeline stages is directly proportional to $\sqrt{(t_{max}/l)}$. If we have faster latches, we can support deeper pipelines. Secondly, with the progress of technology, t_{max}/l is not changing significantly [ITRS, 2011], because both logic gates, and latches are getting faster (roughly equally). Hence, the optimal number of pipeline stages for a processor has remained almost the same for at least the last 5 years.

Implication 4
As we increase l, r, c, and t_{max} the ideal performance goes down as per Equation 10.12. The latch delay can be a very sensitive parameter, especially, for processors that are designed to run workloads with few dependences. In this case, r, and c, will have relatively small values, and Equation 10.12 will be dominated by the value of the latch delay.

Example 142

Find the optimal number of pipeline stages for the following configuration. $t_{max}/l = 20$, $r = 0.2$, $c = 0.6$.

Answer: *We have:*

$$k = \sqrt{\frac{t_{max}}{lrc}} = \sqrt{20/(0.2 * 0.6)} = 12.9 \approx 13$$

Example 143

Consider two programs that have the following characteristics.

Program 1		*Program 2*	
Instruction Type	*Fraction*	*Instruction Type*	*Fraction*
loads	*0.4*	*loads*	*0.3*
branches	*0.2*	*branches*	*0.1*
ratio(taken branches)	*0.5*	*ratio(taken branches)*	*0.4*

The ideal CPI is 1 for both the programs. Let 50% of the load instructions suffer from a load-use hazard. Assume that the frequency of P_1 is 1, and the frequency of P_2 is 1.5. Here, the units of the frequency are not relevant. Compare the performance of P_1 and P_2.

Answer:

$$\begin{aligned} CPI_{new} =& CPI_{ideal} + 0.5 \times (ratio(loads)) \times 1 \\ & + ratio(branches) \times ratio(taken\,branches) \times 2 \end{aligned} \tag{10.13}$$

We thus have:

$$\begin{aligned} CPI_{P_1} &= 1 + 0.5 \times 0.4 + 0.2 \times 0.5 \times 2 = 1 + 0.2 + 0.2 = 1.4 \\ CPI_{P_2} &= 1 + 0.5 \times 0.3 + 0.1 \times 0.4 \times 2 = 1 + 0.15 + 0.08 = 1.23 \end{aligned}$$

The performance of P_1 can be expressed as f/CPI = 1 / 1.4 = 0.71 (arbitrary units). Similarly, the performance of P_2 is equal to f/CPI = 1.5/1.23 = 1.22 (arbitrary units). Hence, P_2 is faster than P_1. We shall often use the term, arbitrary units, a.u., when the choice of units is irrelevant.

10.9.4 Performance of a Suite of Programs

Most of the time, we do not measure the performance of a processor with respect to one program. We consider a set of known benchmark programs and measure the performance of our processor with respect to all the programs to get a consolidated figure. Most processor vendors typically summarize the performance of their processor with respect to the SPEC (`http://www.spec.org`) benchmarks. SPEC stands for "Standard Performance Evaluation

Corporation". They distribute suites of benchmarks for measuring, summarizing, and reporting the performance of processors, and software systems.

Computer architectures typically use the SPEC CPU benchmark suite to measure the performance of a processor. The SPEC CPU 2006 benchmarks have two types of programs – integer arithmetic benchmarks (SPECint), and floating point benchmarks (SPECfp). There are 12 SPECint benchmarks that are written in C/C++. The benchmarks contain parts of C compilers, gene sequencers, AI engines, discrete event simulators, and XML processors. On similar lines, the SPECfp suite contains 17 programs. These programs solve different problems in the domains of physics, chemistry, and biology.

Most processor vendors typically compute a SPEC score, which is representative of the performance of the processor. The recommended procedure is to take the ratio of the time taken by a benchmark on a reference processor, and the time taken by the benchmark on the given processor. The SPEC score is equal to the geometric mean of all the ratios. In computer architecture, when we report the mean relative performance (as in the case of SPEC scores), we typically use the geometric mean. For just reporting the average time of execution (absolute time), we can use the arithmetic mean.

Sometimes, instead of reporting SPEC scores, we report the average number of instructions that we execute per second, and in the case of scientific programs, the average number of floating point operations per second. These metrics give us an indication of the speed of a processor, or a system of processors. We typically use the following terms:

KIPS Kilo(10^3) instructions per second

MIPS Million(10^6) instructions per second

MFLOPS Million(10^6) floating point operations per second

GFLOPS Giga(10^9) floating point operations per second

TFLOPS Tera(10^{12}) floating point operations per second

PFLOPS Peta(10^{15}) floating point operations per second

10.9.5 Inter-Relationship between Performance, the Compiler, Architecture, and Technology

Let us now summarize our discussion by looking at the relationships between performance, compiler design, processor architecture, and manufacturing technology. Let us consider the performance equation again (see Equation 10.14) (let us assume arbitrary units for performance and replace the proportional sign by an equality).

$$P = \frac{f \times IPC}{\#insts} \tag{10.14}$$

If our final aim is to maximize performance, then we need to maximize the frequency (f), and the IPC. Simultaneously, we need to minimize the number of dynamic instructions ($\#insts$). There are three knobs that are under our control namely the processor architecture, manufacturing technology, and the compiler. Note that we loosely use the term "architecture" here. We wish to use the term "architecture" to refer to the actual organization and design of the processor. However, in literature, it is common to use the term "architecture" to refer to both the ISA, and the design of a processor. Hence, we use the same terminology here. Let us look at each of our knobs in detail.

The Compiler

By using smart compiler technology we can reduce the number of dynamic instructions, and also reduce the number of stalls. This will improve the IPC. Let us consider two examples: Examples 144 and 145. Here, we remove one stall cycle by reordering the *add* and *ld* instructions. On similar lines, compilers typically analyze hundreds of instructions, and optimally reorder them to reduce stalls as much as possible.

Example 144
Reorder the following piece of code without violating the correctness of the program to reduce stalls.

```
add r1, r2, r3
ld  r4, 10[r5]
sub r1, r4, r2
```

Answer: *We have a load-use hazard here, between the ld and sub instructions. We can reorder the code as follows.*

```
ld  r4, 10[r5]
add r1, r2, r3
sub r1, r4, r2
```

Now, we do not have any load-use hazards, and the logic of the program remains the same.

Example 145
Reorder the following piece of code without violating the correctness of the program to reduce stalls. Assume delayed branches with 2 delay slots

```
add r1, r2, r3
ld  r4, 10[r5]
sub r1, r4, r2
add r8, r9, r10
b .foo
```

Answer:

```
add r1, r2, r3
ld  r4, 10[r5]
b .foo
sub r1, r4, r2
add r8, r9, r10
```

We eliminate the load-use hazard, and optimally used the delay slots.

The Architecture

We have designed an advanced architecture in this chapter by using pipelining. Note that pipelining by itself, does not increase performance. In fact because of stalls, pipelining reduces the IPC of a program as compared to a single cycle processor. The main benefit of pipelining is that it allows us to run the processor at a higher frequency. The minimum cycle time reduces from t_{max} for a single cycle pipeline to $t_{max}/k + l$ for a k-stage pipelined machine. Since we complete the execution of a new instruction every cycle unless there are stalls, we can execute a set of instructions much faster on a pipelined machine. The instruction execution throughput is much higher.

Important Point 16
The main benefit of pipelining is that it allows us to run the processor at a higher frequency. By running the processor at a higher frequency, we can ensure a higher instruction throughput (more instructions complete their execution per second). Pipelining by itself, reduces the IPC of a program as compared to a single cycle processor, and it also increases the time it takes to process any single instruction.

Techniques such as delayed branches, and forwarding help increase the IPC of a pipelined machine. We need to focus on increasing the performance of complex pipelines through a variety of techniques. The important point to note here is that architectural techniques affect both the frequency (via the number of pipeline stages), and the IPC (via the optimizations such as forwarding and delayed branches).

Manufacturing Technology

Manufacturing technology affects the speed of transistors, and in turn the speed of combinational logic blocks, and latches. Transistors are steadily getting smaller and faster. Consequently, the total algorithmic work (t_{max}) and the latch delay (l), are also steadily reducing. Hence, it is possible to run processors at higher frequencies leading to improvements in performance (also see Equation 10.12). Manufacturing technology exclusively affects the frequency at which we can run a processor. It does not have any effect on the IPC, or the number of instructions.

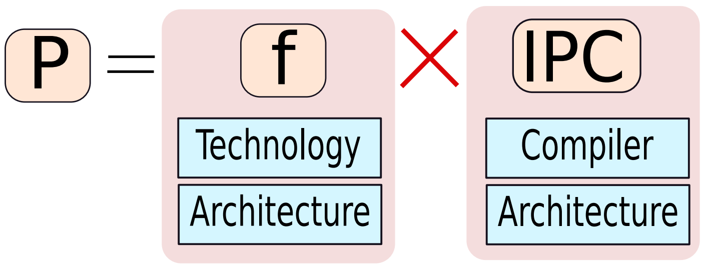


Figure 10.35: Relationship between performance, the compiler, architecture and technology

We can thus summarize our discussion in Figure 10.35.

Note that the overall picture is not as simple as we describe in this section. We need to consider power and complexity issues also. Typically, implementing a pipeline beyond 20 stages is very difficult because of the increase in complexity. Secondly, most modern processors have severe power and temperature constraints. This problem is also known as the **power wall**. It is often not possible to ramp up the frequency, because we cannot afford the increase in power consumption. As a thumb rule, power increases as the cube of frequency. Hence, increasing the frequency by 10% increases the power consumption by more than 30%, which is prohibitively large. Designers are thus increasingly avoiding deeply pipelined designs that run at very high frequencies.

10.10 Power and Temperature Issues

10.10.1 Overview

Let us now briefly look at power and temperature issues. These issues have increasingly become more important over the last decade. High performance processor chips typically dissipate 60-120W of power during normal operation. If we have four chips in a server class computer, then we shall roughly dissipate 400W of power. As a general rule of thumb the rest of the components in a computer such as the main memory, hard disk, peripherals, and fans, also dissipate a similar amount of power. The total power consumption is roughly 800W. If we add additional overheads such as the non-ideal efficiency of the power supply, the display hardware, the power requirement goes up to about 1KW. Now, a typical server farm that has 100 servers will require 100 kW of power for running the computers. Additionally, it will require extra power for the cooling units such as air conditioners. Typically, to remove 1 W of heat, we require 0.5W of cooling power. Thus, the total power dissipation of our server farm is about 150 kW. In comparison, a typical home has a rated power of 6-8 kW. This means that the power dissipated by one server farm is equivalent the power used by 20-25 homes, which is significant. Note that a server farm containing 100 machines is a relatively small setup, and in practice we have much larger server farms containing thousands of machines. They require megawatts of power, which is enough for the needs of a small town.

Let us now consider really small devices such as the processors in cell phones. Here, also power consumption is an important issue because of the limited amount of battery life. All of us would love devices that have very long battery lives especially feature rich smartphones. Let us now consider even smaller devices such as small processors embedded inside the body for medical applications. We typically use small microchips in devices such as pacemakers. In such cases, we do not want to inconvenience the patient by forcing him or her to also carry heavy batteries, or recharge the batteries often. To prolong battery life, it is important to dissipate as little power as possible.

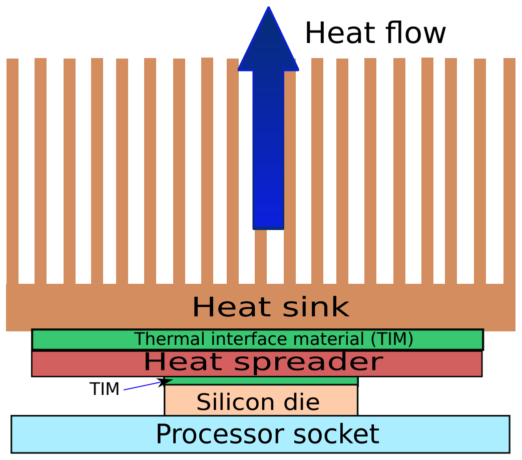


Figure 10.36: Diagram of a chip's package

Now, let us consider temperature, which is a very closely related concept. Let us take a look at the diagram of the typical package of a chip in Figure 10.36. We typically have a 200-400 mm^2 silicon die. The die refers to a rectangular block of silicon that contains the circuit of the chip. Since this small piece of silicon dissipates 60-100 W of power (equivalent to 6-10 CFL light bulbs), its temperature can rise to 200°C unless we take additional measures to cool the silicon die. We first add a 5 cm $\times$ 5 cm nickel plated copper plate on the silicon die. This is known as the *spreader* . The spreader helps in creating a homogeneous temperature profile on the die by spreading the heat, and thus eliminating hot spots. We need a spreader because all the parts of a chip do not dissipate the same amount of heat. The ALUs typically dissipate a lot of heat. However, the memory elements, are relatively cooler. Secondly, the heat dissipation depends on the nature of the program. For integer benchmarks, the floating point ALU is idle, and thus it will be much cooler. To ensure that heat properly flows from the silicon die to the spreader we typically add a thermally conducting gel known as the thermal interface material (TIM).

Most chips have a structure known as the *heat sink* placed above the spreader. It is a copper based structure that has an array of fins as shown in Figure 10.36. We add an array of fins to increase its surface area. This ensures that most of the heat generated by the processors can get dissipated to the surrounding air. In chips that are used in desktops, laptops, and servers, we have a fan mounted on the heat sink, or in the chassis of the computer that blows air over the heat sink. This ensures that hot air is dissipated away, and colder air from outside flows over the heat sink. The assembly of the spreader, heat sink, and fan help in dissipating most of the heat generated by the processor.

In spite of advanced cooling technology, processors still heat up to 60-100°C . While playing highly interactive computer games, or while running heavy number crunching applications like weather simulation, on-chip temperatures can go up to 120°C . Such temperatures are high enough to boil water, cook vegetables, and even warm a small room in winter. Instead of buying heaters, we can just run a computer!!! Note that temperature has a lot of deleterious effects. In particular, the reliability of on-chip copper wires, and transistors decreases exponentially with increasing temperature [Srinivasan et al., 2004]. Secondly, chips tend to age over time due to an effect known as NBTI (Negative Bias Temperature Instability). Ageing effectively slows down transistors. Hence, it becomes necessary to reduce the frequency of processors over time to ensure correct operation. Secondly, some power dissipation mechanisms such as leakage power are dependent on temperature. This means that as the temperature goes up the leakage component of the total power also goes up, and this further increases temperature.

Let us thus conclude that **it is very important** to reduce on-chip power and temperature in the interest of lower electricity bills, reduced cooling costs, longer battery life, higher reliability, and slower aging.

Let us now quickly review the main power dissipation mechanisms. We shall primarily focus on two mechanisms namely dynamic and leakage power. Leakage power is also known as static power.

10.10.2 Dynamic Power

Let us consider a chip's package as a closed black box. We have electrical energy flowing in, and heat coming out. Over a sufficiently long period of time, the amount of electrical energy flowing in to the chip is exactly equal to the amount of energy dissipated as heat according to the law of conservation of energy. Note that we disregard the energy spent in sending electrical signals along I/O links. In any case, this energy is negligible as compared to the power dissipation of the entire chip.

Any circuit consisting of transistors, and copper wires can be modeled as an equivalent circuit with resistors, capacitors, and inductors. Capacitors and inductors do not dissipate heat. However, resistors convert a part of the electrical energy that flows through them to heat. This is the only mechanism through which electrical energy can get converted to thermal energy in our equivalent circuit.

Let us now consider a small circuit that has a single resistor and a single capacitor as shown in Figure 10.37. The resistor represents the resistance of the wires in the circuit. The capacitor represents the equivalent capacitance of transistors in the circuit. We need to note that different parts of a circuit such as the gates of transistors have a certain potential at a given point in time. This means that the gate of a transistor is functioning as a capacitor, and hence storing charge. Similarly, the drain and source of a transistor have an equivalent drain and source capacitance. We typically do not consider equivalent inductance in a simplistic analysis, because most wires are typically short, and they do not function as inductors.

If we analyze this simple circuit, then we can conclude that the total energy required to charge the capacitor is CV^2. $\frac{1}{2}CV^2$ is dissipated by the resistor while charging the capacitor, and the remaining energy is stored in the capacitor. Now, if the capacitor gets discharged, then the remaining $\frac{1}{2}CV^2$ gets dissipated via the resistor.

Now, let us generalize this result. In a large circuit with billions of transistors, we essentially

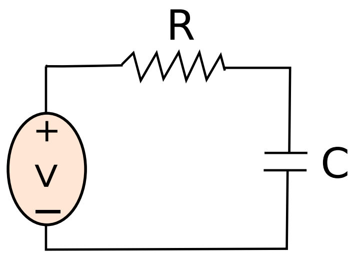


Figure 10.37: A circuit with a resistance and capacitance

have billions of subcircuits with resistive and capacitive elements. Each cycle, we can either have a transition in a bit ($0 \rightarrow 1$ or $1 \rightarrow 0$), or we might have no transitions at all. If there is a transition in the value of a bit, then either a capacitor gets charged or gets discharged. However, if there are no transitions, then there is no current flow, and thus there is no heat dissipation.

Let us assume that we have n subcircuits. Let, α_i be known as the *activity factor*. It is 1 if there is a transition, and 0 if there is no transition in subcircuit i. Let $E_1 \ldots E_n$ be the energy dissipated by all the n subcircuits. We thus have:

$$E_1 = \frac{1}{2}\alpha_1 C_1 V^2 \tag{10.15}$$

$$E_2 = \frac{1}{2}\alpha_2 C_2 V^2 \tag{10.16}$$

$$\ldots$$

$$E_n = \frac{1}{2}\alpha_n C_n V^2 \tag{10.17}$$

The total energy dissipated is equal to $\sum_{i=1}^{n} E_i$. Let us now group the small subcircuits into functional units, and assume that the capacitance values across all the subcircuits in a functional unit are roughly similar. Thus, for a given functional unit j, we can say that:

$$E_j \propto \alpha_j C_j V^2 \tag{10.18}$$

Here, C_j is a representative value of capacitance for the entire functional unit, and α_j is the activity factor for the entire functional unit. 0 represents no activity, and 1 represents 100% activity. $0 \leq \alpha_j \leq 1$. Note that we have also replaced the equality by a proportional sign because we are interested in the nature of power dissipation rather than the exact values.

We can thus express the total energy consumption of a circuit having n' functional units as:

$$E \propto \sum_{i=1}^{n'} \alpha_i C_i V^2 \tag{10.19}$$

This equation represents the energy consumed per cycle. Power is equal to energy divide by time. In this case the time is equal to the clock cycle time, or the reciprocal of the chip's frequency (f). Thus, the total power ($\mathcal{P}$) is equal to:

$$\mathcal{P} \propto \sum_{i=1}^{n'} \alpha_i C_i V^2 f \tag{10.20}$$

The power dissipated is thus proportional to the frequency, and the square of the supply voltage. Note that this power dissipation represents the resistive loss due to the transitions in the inputs and outputs. Hence, it is known as the *dynamic power*, $\mathcal{P}_{dyn}$. Thus, we have:

$$\mathcal{P}_{dyn} \propto \sum_{i=1}^{n'} \alpha_i C_i V^2 f \tag{10.21}$$

Definition 87
Dynamic power *is the cumulative power dissipated due to the transitions of inputs and outputs across all the transistors in a circuit.*

10.10.3 Leakage Power

Note that dynamic power is not the only power dissipation mechanism in processors. *Static* or *leakage* power is a major component of the power dissipation profile of high-performance processors. It accounts for roughly 20-40% of the total processor power budget.

The main insight is as follows. We have up till now been assuming that a transistor does not allow any current to flow through it when it is in the off state. There is absolutely no current flow across the terminals of a capacitor, or between the gate and the source of an NMOS transistors. All of these assumptions are not strictly correct. No structure is a perfect insulator in practice. There is a small amount of current flow across its terminals, even in the *off* state. We can have many other sources of leakage power across other interfaces that are ideally not supposed to pass current. Such sources of current are together referred to as *leakage current*, and the associated power dissipation is known as the *leakage power*.

Definition 88
Leakage current *is the minimal amount of current that flows across two terminals of a circuit element that are ideally supposed to be completely electrically isolated from each other. For example, we do not expect any current flow between the drain and the source of an NMOS transistor in the* off *state. However, a small amount of current does flow, and this is known as the sub-threshold leakage current. When leakage current flows across a resistive element, it dissipates leakage power. Leakage power is* static *in nature and is dissipated all the time irrespective of the level of activity in a circuit.*

There are different mechanisms for leakage power dissipation such as sub-threshold leakage, and gate induced drain leakage. Researchers typically use the following equation from the BSIM3 model [Cheng and Hu, 1999] for leakage power (primarily captures sub-threshold leakage):

$$\mathcal{P}_{leak} = A \times \nu_T^2 \times e^{\frac{V_{GS}-V_{th}-V_{off}}{n \times \nu_T}} \left(1 - e^{\frac{-V_{DS}}{\nu_T}}\right) \tag{10.22}$$

Variable	Definition (SI units)
A	Area dependent constant of proportionality
ν_T	Thermal voltage (kT/q)
k	Boltzmann's constant (1.38×10^{-23}) (SI units)
q	1.6×10^{-19}
T	Temperature (in Kelvins)
V_{GS}	Voltage between the gate and source
V_{th}	Threshold voltage. It is also dependent on temperature. $\frac{\partial V_{th}}{\partial T} = -2.5mV/K$
V_{off}	Offset voltage
n	Sub-threshold swing coefficient
V_{DS}	Voltage between the drain and source

Table 10.3: Definition of variables in Equation 10.22

Table 10.3 defines the variables used in Equation 10.22. Note that the leakage power is dependent on temperature via the variable $\nu_T = kT/q$. To show the temperature dependence, we can simplify Equation 10.22 to obtain Equation 10.23.

$$\mathcal{P}_{leak} \propto T^2 \times e^{A/T} \times \left(1 - e^{B/T}\right) \tag{10.23}$$

In Equation 10.23, A and B are constants, and can be derived from Equation 10.22. Around 10 years ago (as of 2002), when the transistor threshold voltages used to be higher (around 500 mV), leakage power was exponentially dependent on temperature. Hence, a small increase in temperature would translate to a large increase in leakage power. However, nowadays, the threshold voltages are between 100-150 mV. Consequently, the relationship between temperature and leakage has become approximately linear [Sarangi et al., 2014].

The important point to note here is that leakage power is dissipated all the time by all the transistors in a circuit. The amount of leakage current might be very small; but when we consider the cumulative effect of billions of transistors, the total amount of leakage power dissipation is sizable, and can even become a large fraction of the dynamic power. Consequently, designers try to control temperature to keep leakage power under control.

Hence, the total power, $\mathcal{P}_{tot}$, is given by:

$$\mathcal{P}_{tot} = \mathcal{P}_{dyn} + \mathcal{P}_{leak} \tag{10.24}$$

10.10.4 Modeling Temperature*

Modeling the temperature on a chip is a fairly complex problem, and requires a fair amount of background in thermodynamics and heat transfer. Let us state a basic result here, and move on.

Let us divide the area of a silicon die into a grid. Let us number the grid points $1 \ldots m$. Let the power vector $\mathcal{P}_{tot}$ represent the total power dissipated by each grid point. Similarly, let the temperature of each grid point be represented by the vector T. Power and temperature are typically related by the following linear equation for a large number of grid points.

$$T - T_{amb} = \Delta T = A \times \mathcal{P}_{tot} \tag{10.25}$$

T_{amb} is known as the *ambient* temperature – it is the temperature of the surrounding air. A is an $m \times m$ matrix, and is also known as the thermal resistance matrix. According to Equation 10.25 the change in temperature (ΔT), and the power consumption are linearly related to each other.

Note that $\mathcal{P}_{tot} = \mathcal{P}_{dyn} + \mathcal{P}_{leak}$, and $\mathcal{P}_{leak}$ is a function of temperature. Hence, Equations 10.24, and 10.25 form a feedback loop. We thus need to assume an initial value of temperature, compute the leakage power, estimate the new temperature, compute the leakage power, and keep iterating till the values converge.

10.10.5 The ED^2 Metric

Now, let us try to integrate performance, and energy into one model. The performance of a program is given by the performance equation (Equation 10.3). Let us simplistically assume that the time a program takes, or its delay (D) is inversely proportional to the frequency. Again, this is not strictly correct because the IPC is dependent on the frequency. We cannot appreciate the relationship between IPC and frequency right now, because we do not have adequate background. However, we shall touch this topic in Section 11.3, and see that there are components to the IPC that are frequency dependent such as the latency of main memory. In any case, let us move ahead with the approximation that $D \propto 1/f$.

Let us compare two processor designs for the same program. One design dissipates E_1 Joules for the execution of the entire program, and it takes D_1 units of time. The second design dissipates E_2 Joules, and takes D_2 units of time. How do we say, which design is better? It is possible that the second design is slightly faster but dissipates 3 times more energy per cycle. There has to be a common metric.

To derive a common metric, we need to either make the performance the same ($D_1 = D_2$), and then compare the energy, or make the energy the same ($E_1 = E_2$), and compare the performance. To ensure that $D_1 = D_2$ we need to either speed up one design or slowdown the other one. To achieve this, we can use a standard technique called dynamic voltage-frequency scaling (DVFS).

According to the DVFS technique, to scale up the frequency by a factor of κ_1, we scale the voltage by a factor of κ_2. Typically, we assume that $\kappa_1 = \kappa_2$. For example, to double the frequency, we double the voltage also. Note that with a higher frequency and consequent lower clock cycle time, we need to ensure that signals can rise and fall quickly. To ensure quicker signal transition, we increase the voltage such that it takes a lesser amount of time for a signal to rise and fall by ΔV volts. This fact can be proved by considering the basic capacitor charging and discharging equations. From our point of view, we need to appreciate the fact that the voltage and frequency need to be scaled together.

Definition 89
DVFS is a technique that is used to adjust the voltage and frequency of a processor at run time. If we scale the frequency by a factor of κ_1, then we need to scale the voltage by a factor of κ_2. In most cases, we assume that $\kappa_1 = \kappa_2$.

Now, let us try to equalize the execution time of designs 1 and 2, and compare the energy. We have made the following assumptions: $D \propto 1/f$, and $f \propto V$. Thus, $D \propto 1/V$. To make the delays equal we need to scale the delay of design 2 by D_1/D_2, or alternatively we need to scale its voltage and frequency by D_2/D_1. After equalizing the delay, let the energy dissipation of design 2 be E_2'. Since $E \propto \alpha V^2$, we have:

$$\begin{aligned} E_2' &= E_2 \times \frac{V_1^2}{V_2^2} \\ &= E_2 \times \frac{f_1^2}{f_2^2} \\ &= E_2 \times \frac{D_2^2}{D_1^2} \end{aligned} \tag{10.26}$$

Now, let us compare E_1 and E_2'.

$$\begin{aligned} &E_2' <=> E_1 \\ \Leftrightarrow &E_2 \times \frac{D_2^2}{D_1^2} <=> E_1 \\ \Leftrightarrow &E_2 D_2^2 <=> E_1 D_1^2 \end{aligned} \tag{10.27}$$

In this case, we observe that comparing E_2' and E_1 is tantamount to comparing $E_2D_2^2$, and $E_1D_1^2$. Since $E \propto V^2 (\propto 1/D^2)$, $ED^2 = \kappa$. Here, κ is a constant that arises out of the different constants of proportionality. It is thus a property that is independent of the voltage and frequency of the system. It is related to the activity factor, and the capacitance of the circuits, and is inherent to the design. Consequently, the ED^2 metric is used as an effective baseline metric to compare two designs.

Designers aim to reduce the ED^2 metric of a design as much as possible. This ensures that irrespective of the DVFS settings, a design with a lower value of ED^2 is a much better design than other designs that have a higher ED^2 metric. Note that a lot of performance enhancing schemes do not prove to be effective because they do not show any benefit with regard to the ED^2 metric. They do increase performance, but also disproportionately increase the energy dissipation. Likewise, a lot of power reduction schemes are impractical because they increase the delay, and the ED^2 metric increases. Consequently, whenever we need to jointly optimize energy/power and performance we use the ED^2 metric to evaluate candidate designs.

10.11 Advanced Techniques*

Way Point 8

- *We designed a complete single cycle processor for the SimpleRisc instruction set in Section 9.1. This processor had a hardwired control unit.*
- *We designed a more flexible variant of our SimpleRisc processor using a micro-programmed control unit. This required a bus based data path along with a new set of microinstructions, and microassembly based code snippets for each program instruction.*
- *We observed that our processors could be significantly sped up by pipelining. However, a pipelined processor suffers from hazards that can be significantly eliminated by a combination of software techniques, pipeline interlocks, and forwarding.*

In this section, we shall take a brief look at advanced techniques for implementing processors. Note that this section is by no means self contained, and its primary purpose is to give the reader pointers for additional study. We shall cover a few of the broad paradigms for substantially increasing performance. These techniques are adopted by state-of-the-art processors.

Modern processors typically execute multiple instructions in the same cycle using very deep pipelines (12-20 stages), and employ advanced techniques to eliminate hazards in the pipeline. Let us look at some common approaches.

10.11.1 Branch Prediction

Let us start with the IF stage, and see how we can make it better. If we have a taken branch in the pipeline then the IF stage in particular needs to stall for 2 cycles in our pipeline, and then needs to start fetching from the branch target. As we add more pipeline stages, the *branch penalty* increases from 2 cycles to more than 20 cycles. This makes branch instructions extremely expensive, and they are known to severely limit performance. Hence, it is necessary to avoid pipeline stalls even for taken branches.

What if, it is possible to predict the direction of branches, and also predict the branch target? In this case, the fetch unit can immediately start fetching from the predicted branch target. If the prediction is found to be wrong at a later point of time, then all the instructions after the mispredicted branch instruction need to be canceled, and discarded from the pipeline. Such instructions are also known as *speculative* instructions.

Definition 90
Modern processors typically execute large sets of instructions on the basis of predictions. For

example, they predict the direction of branches, and accordingly fetch instructions starting from the predicted branch target. The prediction is verified later when the branch instruction is executed. If the prediction is found to be wrong, then all the instructions that were incorrectly fetched or executed are discarded from the pipeline. These instructions are known as speculative *instructions. Conversely, instructions that were fetched and executed correctly, or whose predictions have been verified are called* non-speculative *instructions.*

Note that it is extremely essential to prohibit speculative instructions from making changes to the register file or writing to the memory system. Thus, we need to wait for instructions to become non-speculative before we allow them to make permanent changes. Second, we also do not allow them to leave the pipeline before they become non-speculative. However, if there is a need to discard speculative instructions, then modern pipelines adopt a simpler mechanism. Instead of selectively converting speculative instructions into pipeline bubbles as we have done in our simple pipeline, modern processors typically remove all the instructions that were fetched after the mispredicted branch instruction. This is a simple mechanism that works very well in practice. It is known as a *pipeline flush*.

Definition 91
Modern processors typically adopt a simple approach of discarding all speculative instructions from a pipeline. They completely finish the execution of all instructions till the mispredicted instruction, and then clean up the entire pipeline, effectively removing all the instructions that were fetched after the mispredicted instruction. This mechanism is known as a pipeline flush*.*

Main Challenges

Let us now outline the main challenges in branch prediction.

1. We need to first find out in the fetch stage if an instruction is a branch, and if it is a branch, we need to find the address of the branch target.

2. Next, we need to predict the expected direction of the branch.

3. It is necessary to monitor the result of a predicted instruction. If there is a misprediction, then we need to perform a pipeline flush at a later point of time such that we can effectively remove all the speculative instructions.

Detecting a misprediction in the case of a branch is fairly straightforward. We add the prediction to the instruction packet, and verify the prediction with the actual outcome. If they are different, then we schedule a pipeline flush. The main challenge is to predict the target of a branch instruction, and its outcome.

Branch Target Buffer

Modern processors use a simple hardware structure called a *branch target buffer* (BTB). It is a simple memory array that saves the program counter of the last N branch instructions, and their targets (N typically varies from 128 to 8192). There is a high likelihood of finding a match, because programs typically exhibit some degree of *locality.* This means that they tend to execute the same piece of code repeatedly over a period of time such as loops. Hence, entries in the BTB tend to get repeatedly reused in a small window of time. If there is a match, then we can also automatically infer that the instruction is a branch.

2-bit Saturating Counter based Branch Predictor

It is much more difficult to effectively predict the direction of a branch. However, we can exploit a pattern here. Most branches in a program typically are found in loops, or in *if* statements where both the directions are not equally likely. In fact, one direction is far more likely than the other. For example, branches in loops are most of the time taken. Sometimes, we have *if* statements that are only evaluated if a certain exceptional condition is true. Most of the time, the branches associated with these *if* statements are not taken. Similarly, for most programs, designers have observed that almost all the branch instructions follow certain patterns. They either have a strong bias towards one direction, or can be predicted on the basis of past history, or can be predicted on the basis of the behavior of other branches. There is of course no theoretical proof of this statement. This is just an observation made by processor designers, and they consequently design predictors to take advantage of such patterns in programs.

We shall discuss a simple 2-bit branch predictor in this book. Let us assume that we have a branch prediction table that assigns a 2-bit value to each branch in the table, as shown in Figure 10.38. If this value is 00, or 01, then we predict that the branch is not taken. If it is equal to 10, or 11, then we predict that the branch is taken. Moreover, every time the branch is taken, we increment the associated counter by 1, and every time, the branch is not taken we decrement the counter by 1. To avoid overflows, we do not increment 11 by 1 to produce 00, and we do not decrement 00 to produce 11. We follow the rules of saturating arithmetic that state that (in binary): (11 + 1 = 11), and (00 - 1 = 00). This 2-bit value is known as a 2-bit saturating counter. The state diagram for the 2-bit counter is shown in Figure 10.39.

There are two basic operations for predicting a branch – prediction, and training. To *predict* a branch, we look up the value of its program counter in the branch prediction table. In specific, we use the last n bits of the address of the *pc* to access a 2^n entry branch predictor table. We read the value of the 2-bit saturating counter, and predict the branch on the basis of its value. When, we have the real outcome of the branch available, we *train* our predictor by incrementing or decrementing the value of our counter using saturating arithmetic (as per Table 10.39).

Let us now see why this predictor works. Let us consider a simple piece of C code, and its equivalent *SimpleRisc* code.

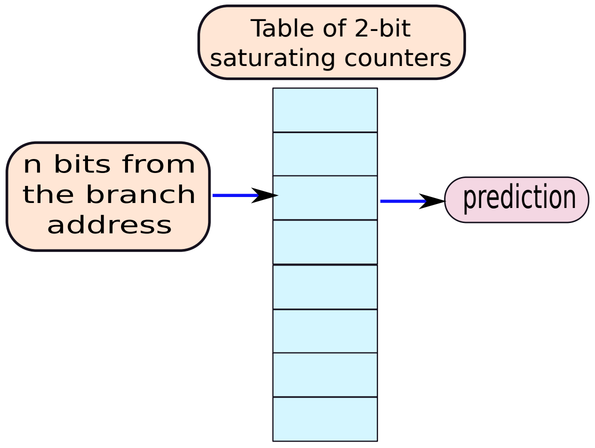


Table of 2-bit saturating counters
n bits from the branch address
prediction

Figure 10.38: A branch prediction table

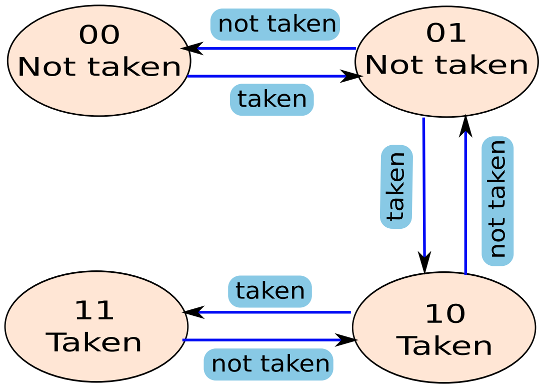


Figure 10.39: 2-bit saturating counter

C

```c
void main(){
        foo();
        ...
        foo();
}

int foo() {
    int i, sum = 0
    for(i=0; i < 10; i++) {
        sum = sum + i;
    }
    return sum;
}
```

SimpleRisc

```
.main:
        call .foo
        ...
        call .foo

.foo:
        mov r0, 0             /* sum = 0                    */
        mov r1, 0             /* i = 0                      */
.loop:
        add r0, r0, r1        /* sum = sum + i              */
        add r1, r1, 1         /* i = i + 1                  */
        cmp r1, 10            /* compare i with 10          */
        bgt .loop            /* if(r1 > 10) jump to .loop */
        ret
```

Let us take a look at the branch in the loop statement (Line 13). For all the iterations other than the last one, the branch is taken. If we start our predictor in the state 10, then the first time, the branch is predicted correctly (taken). The counter gets incremented and becomes equal to 11. For each of the subsequent iterations, the branch is predicted correctly (taken). However, in the last iteration, it needs to be predicted as not taken. Here, there is a misprediction. The 2-bit counter thus gets decremented, and gets set to 10. Let us now consider the case when we invoke the function foo again. The value of the 2-bit counter is 10, and the branch (Line 13) is correctly predicted as taken.

We thus observe that our 2-bit counter scheme, adds a tiny amount of hysteresis (or past history) to the prediction scheme. If a branch has historically been taking one direction, then one anomaly, does not change the prediction. This pattern is very useful for loops, as we have seen in this simple example. The direction of the branch instruction in the last iteration of a loop is always different. However, the next time we enter a loop, the branch is predicted correctly, as we have seen in this example. Note that this is only one pattern. There are many more types of patterns that modern branch predictors exploit.

10.11.2 Multiple Issue In-Order Pipeline

In our simple pipeline, we executed only one instruction per cycle. However, this is not a strict necessity. We can design a processor such as the original Intel Pentium that had two parallel pipelines. This processor could execute two instructions simultaneously in one cycle. These pipeline have extra functional units such that instructions in both the pipelines can be executed without any significant structural hazards. This strategy increases the IPC. However, it also makes the processor more complex. Such a processor is said to contain a *multiple issue* in-order pipeline, because we can issue multiple instructions to the execution units in the same cycle. A processor, which can execute multiple instructions per cycle is also known as a *superscalar* processor.

Secondly, this processor is known as an in-order processor, because it executes instructions in program order. The *program order* is the order of execution of dynamic instances of instructions as they appear in the program. For example, a single cycle processor, or our pipelined processor, executes instructions in program order.

Definition 92
A processor that can execute multiple instructions per cycle is known as a superscalar processor*.*

Definition 93
An in-order processor executes instructions in program order. The program order *is defined*

as the order of dynamic instances of instructions that is the same as that is perceived if we execute each instruction of the program sequentially.

Now, we need to look for dependences and potential hazards across both the pipelines. Secondly, the forwarding logic is also far more complex, because results can be forwarded from either pipeline. The original Pentium processor released by Intel had two pipelines namely the U pipe and the V pipe. The U pipe could execute any instruction, whereas the V pipe was limited to only simple instructions. Instructions were fetched as 2-instruction bundles. The earlier instruction in the bundle was sent to the U pipe, and the later instruction was sent to the V pipe. This strategy allowed the parallel execution of those instructions.

Let us try to conceptually design a simple processor on the lines of the original Pentium processor with two pipelines – U and V. We envisage a combined instruction and operand fetch unit that forms *2-instruction* bundles, and dispatches them to both the pipelines for execution simultaneously. However, if the instructions do not satisfy some constraints, then this unit forms a 1-instruction bundle, and sends it to the U pipeline. Whenever, we form such bundles, we can broadly adhere to some generic rules. We should avoid having two instructions that have a RAW dependence. In this case, the pipeline will stall.

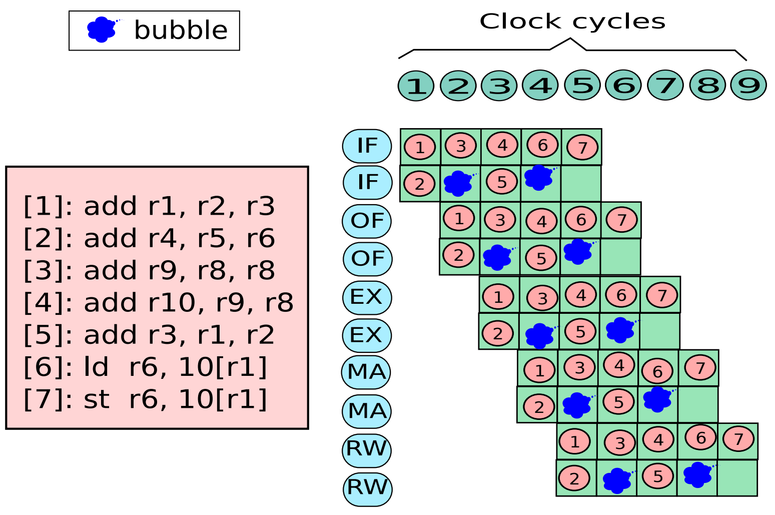
Secondly, we need to be particularly careful about memory instructions because dependences across them cannot be discovered till the end of the EX stage. Let us assume that the first instruction in a bundle is a store instruction, and the second instruction is a load instruction, and they happen to access the same memory address. We need to detect this case, at the end of the EX stage, and forward the value from the store to the load. For the reverse case, when the first instruction is a load instruction, and the second is a store to the same address, we need to stall the store instruction till the load completes. If both the instructions in a bundle store to the same address, then the earlier instruction is redundant, and can be converted into a *nop*. We thus need to design a processor that adheres to these rules, and has a complex interlock and forwarding logic.

Let us show a simple example.

Example 146
Draw a pipeline diagram for the following SimpleRisc assembly code assuming a 2 issue in-order pipeline.

```
[1]: add r1, r2, r3
[2]: add r4, r5, r6
[3]: add r9, r8, r8
[4]: add r10, r9, r8
[5]: add r3, r1, r2
[6]: ld  r6, 10[r1]
[7]: st r6, 10[r1]
```

Answer: *Here, the pipeline diagram contains two entries for each stage, because two instructions can be in a stage at the same time. We start out by observing that we can execute instructions [1] and [2] in parallel. However, we cannot execute instructions [3] and [4] in parallel. This is because instruction [3] writes to* r9*, and instruction [4] has* r9 *as a source operand. We cannot execute both the instructions in the same cycle, because the value of* r9 *is produced in the EX stage, and is also required in the EX stage. We thus insert a bubble. We proceed to execute [4], and [5] in parallel. We can use forwarding to get the value of* r9 *in the case of instruction [4]. Lastly, we observe that we cannot execute instructions [6] and [7] in parallel. They access the same memory address. The load needs to complete before the store starts. We thus insert another bubble.*



10.11.3 EPIC and VLIW Processors

Now, instead of preparing bundles in hardware, we can prepare them in software. The compiler has far more visibility into the code, and can perform extensive analyses to create multi-instruction bundles. The Itanium® processor designed by Intel and HP was a very iconic processor, which was based on similar principles.

Let us first start out by defining the terms – EPIC and VLIW.

Definition 94
VLIW → *Very Long Instruction Word: Compilers create bundles of instructions that do not have dependences between them. The hardware executes the instructions in each bundle in parallel. The complete onus of correctness is on the compiler.* EPIC → *Explicitly Parallel Instruction Computing: This paradigm extends VLIW computing. However, in this case the hardware ensures that the execution is correct regardless of the code generated*

by the compiler.

EPIC/VLIW processors require very smart compilers to analyze programs and create bundles of instructions. For example, if a processor has 4 pipelines, then each bundle contains 4 instructions. The compilers create bundles such that there are no dependences across instructions in a bundle. The broader aim of designing EPIC/VLIW processors is to move all the complexity to software. Compilers arrange the bundles in a way such that we can minimize the amount of interlock, forwarding, and instruction handling logic required in the processor.

However, in hindsight, such processors failed to deliver on their promise because the hardware could not be made as simple as the designers had originally planned for. A high performance processor still needed a fair amount of complexity in hardware, and required some sophisticated architectural features. These features increased the complexity and power consumption of hardware.

10.11.4 Out-of-Order Pipelines

We have up till now been considering primarily in-order pipelines. These pipelines execute instructions in the order that they appear in the program. This is not strictly necessary. Let us consider the following code snippet.

```
[1]: add r1, r2, r3
[2]: add r4, r1, r1
[3]: add r5, r4, r2
[4]: mul r6, r5, r2
[5]: div r8, r9, r10
[6]: sub r11, r12, r13
```

Here, we are constrained to execute instructions 1 to 4 in sequence because of data dependences. However, we can execute instructions, 5 and 6 in parallel, because they are not dependent on instructions 1-4. We will not be sacrificing on correctness if we execute instructions 5 and 6 *out-of-order*. For example, if we can issue two instructions in one cycle, then we can issue (1,5) together, then (2,6), and finally, instructions 3, and 4. In this case, we can execute the sequence of 6 instructions in 4 cycles by executing 2 instructions for the first two cycles. Recall that such a processor that can potentially execute multiple instructions per cycle is known as a superscalar processor (see Definition 92).

Definition 95
A processor that can execute instructions in an order that is not consistent with their program order *is known as an* out-of-order*(OOO) processor.*

An out-of-order(OOO) processor fetches instructions in-order. After the fetch stage, it proceeds to decode the instructions. Most real world instructions require more than one cycle for decoding. These instructions are simultaneously added to a queue called the reorder buffer (ROB) in program order. After decoding the instruction, we need to perform a step called *register renaming.* The broad idea is as follows. Since we are executing instructions out of order, we can have WAR and WAW hazards. Let us consider the following code snippet.

```
[1]: add r1, r2, r3
[2]: sub r4, r1, r2
[3]: add r1, r5, r6
[4]: add r9, r1, r7
```

If we execute instructions [3] and [4] before instruction [1], then we have a potential WAW hazard. This is because instruction [1] might overwrite the value of $r1$ written by instruction [3]. This will lead to an incorrect execution. Thus, we try to rename the registers such that these hazards can be removed. Most modern processors define a set of architectural registers, which are the same as the registers exposed to software (assembly programs). Additionally, they have a set of physical registers that are only visible internally. The renaming stage converts architectural register names to physical register names. This is done to remove WAR and WAW hazards. The only hazards that remain at this stage are RAW hazards, which indicate a genuine data dependency. The code snippet will thus look as follows after renaming. Let us assume that the physical registers range from $p1 \ldots p128$.

```
[1]: add p1, p2, p3   /* p1 contains r1 */
[2]: sub p4, p1, p2
[3]: add p100, p5, p6 /* r1 is now begin saved in p100 */
[4]: add p9, p100, p7
```

We have removed the WAW hazard by mapping r1 in instruction 3, to $p100$. The only dependences that exist are RAW dependences between instructions [1] $\rightarrow$ [2], and [3] $\rightarrow$ [4]. The instructions after renaming enter an instruction window. Note that up till now instructions have been proceeding in-order.

The instruction window or instruction queue typically contains 64-128 entries (refer to Figure 10.40). For each instruction, it monitors its source operands. Whenever all the source operands of an instruction are ready, the instruction is ready to be *issued* to its corresponding functional unit. It is not necessary for instructions to access the physical register file all the time. They can also get values from forwarding paths. After the instructions finish their execution, they broadcast the value of their result to the waiting instructions in the instruction window. Instructions waiting for the result, mark their corresponding source operand as ready. This process is known as *instruction wakeup.* Now, it is possible that multiple instructions are ready in the same cycle. To avoid structural hazards, an *instruction select* unit chooses a set of instructions for execution.

We need another structure for load and store instructions known as the *load-store* queue. It saves the list of loads and stores in program order. It allows loads to get their values through an internal forwarding mechanism if there is an earlier store to the same address.

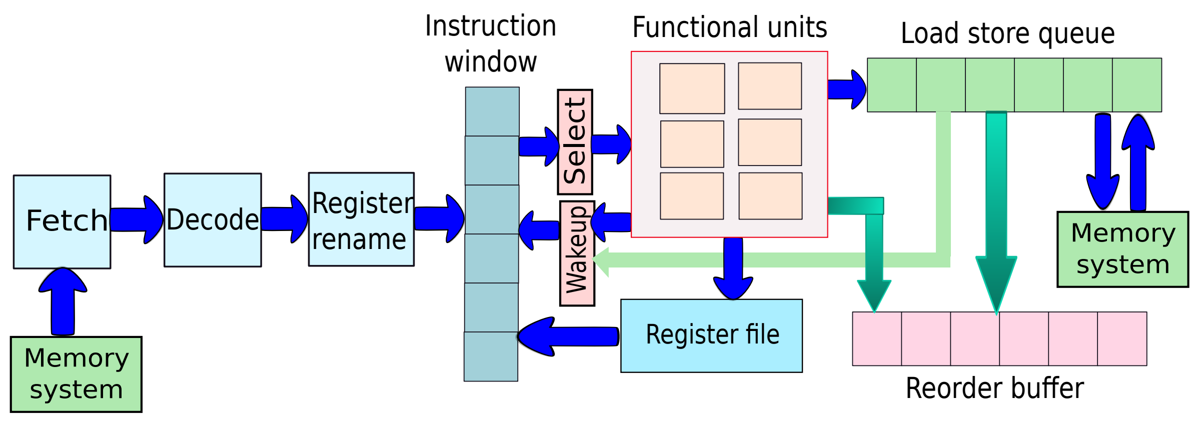


Figure 10.40: An out-of-order pipeline

After an instruction finishes its execution, we mark its entry in the reorder buffer. Instructions leave the reorder buffer in program order. If an instruction does not finish quickly for some reason, then all the instructions after it in the reorder buffer need to stall. Recall that instruction entries in the reorder buffer are ordered in program order. Instructions need to leave the reorder buffer in program order such that we can ensure precise exceptions.

To summarize, the main advantage of an out-of-order processor(OOO) is that it can execute instructions that do not have any RAW dependences between them, in parallel. Most programs typically have such sets of instructions at most points of time. This property is known as instruction level parallelism (abbreviated as *ILP*). Modern OOO processors are designed to exploit as much of ILP as possible.

Definition 96
Typically, most programs have multiple instructions in a pipeline that can be executed in parallel. This is because they do not have any RAW dependences between them. Modern superscalar processors exploit this fact to increase their IPC by executing multiple instructions in the same cycle. This property of a program is known as instruction level parallelism (abbreviated as ILP*).*

10.12 Summary and Further Reading

10.12.1 Summary

Summary 10

1. *We observe that large parts of our basic SimpleRisc processor are idle while processing an instruction. For example, the IF stage is idle, when the instruction is in the MA stage.*
2. *We thus propose the notion of "pipelining". Here, we execute 5 instructions simultaneously (1 in each stage). At the negative edge of the clock, all the instructions proceed to the next stages simultaneously, the instruction in the RW stage completes its execution, and a new instruction enters the IF stage.*
3. *To design a pipeline we split the data path into five parts (1 stage per part), and add pipeline registers between subsequent stages. A pipeline register stores the instruction packet (instruction contents, control signals, source operands and intermediate results).*
4. *Each pipeline stage reads operands for its functional units from its corresponding pipeline register at the beginning of a clock cycle. It processes them, and writes the results to the pipeline register between the given stage and its adjacent stage, before the end of the clock cycle.*
5. *We can have RAW hazards, and control hazards in our pipeline because we cannot ascertain data dependences and branch outcomes before fetching subsequent instructions.*
6. *We can avoid RAW, and control hazards using pure software solutions. We can introduce nop instructions between producer and consumer instructions, and after branch instructions. Alternatively, we can reorder instructions to minimize the addition of nop instructions, and use delayed branching.*
7. *In the absence of software solutions, we can use pipeline interlocks to avoid hazards by stalling and canceling instructions.*
8. *An efficient method of minimizing stall cycles is* forwarding.
   (a) *If a later stage contains the value of an operand, then we can forward the value from the producer stage to the consumer stage. We can thus bypass the register file.*
   (b) *This allows us to avoid hazards because a consumer instruction can quickly get its operands from other pipeline stages.*

(c) To detect dependences, and implement forwarding we propose a dedicated forwarding unit. Furthermore, it is necessary to augment, every functional unit with multiplexers to choose between the default inputs, and forwarded inputs. Forwarding eliminates all the data hazards, other than the load-use hazard.

9. *Modern processors have interrupts and exceptions that require us to save the state of a program, and branch to an interrupt handler. We need to implement precise exceptions such that we can return to the exact same point at which we had stopped the execution of our original program.*
10. *Performance of a processor with respect to a program is defined to be proportional to the inverse of the time required to execute the program.*
11. *The performance equation is as follows:*

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

IPC (instructions per cycle), f(frequency), #insts (number of dynamic instructions)

12. *The performance of a processor is dependent on the manufacturing technology, architecture, and compiler optimizations. In specific, a pipelined processor has higher performance as compared to a single cycle processor, because it allows us to increase the frequency roughly as many times as the number of stages. There is a consequent loss in IPC, and wastage of time due to the latch delay. Hence, it is necessary to choose an optimal pipelining strategy.*
13. *The clock frequency is limited by power and temperature constraints.*

    (a) There are two power dissipation mechanisms in modern processors namely dynamic power and leakage power. Dynamic power is dissipated due to the switching activity in circuits. It is proportional to $\alpha CV^2 f$, where α is the activity factor, C is the lumped circuit capacitance, V is the supply voltage, and f is the frequency.

    (b) Leakage power or static power is dissipated due to the flow of current through the terminals of a transistor, when it is in the off *state. Leakage power is a superlinear function of the current temperature.*

    (c) Power and temperature for different points on a chip are typically related by a set of linear equations.

    (d) Dynamic voltage-frequency scaling is a technique to dynamically modify the voltage and frequency of a processor. We typically assume that the frequency is proportional to voltage.

    (e) We use the ED^2 metric to simultaneously compare the power and performance of competing processor designs.

14. *Some advanced techniques for speeding up a processor are branch prediction, superscalar execution, EPIC/VLIW processors, and out-of-order pipelines.*

10.12.2 Further Reading

The design of high performance pipelines is a prime focus of computer architecture researchers. Researchers mostly look at optimizing performance of pipelines and simultaneously reducing power consumption. The reader can start out with textbooks on advanced computer architecture [Hennessy and Patterson, 2012, Hwang, 2003, Baer, 2010, Sima et al., 1997, Culler et al., 1998]. After getting a basic understanding of the techniques underlying advanced processors such as out-of-order and superscalar execution, the reader should be able to graduate to reading research papers. The first step in this journey should be the book titled, "Readings in Computer Architecture" [Hill et al., 1999]. This book comprises a set of foundational research papers in different areas of computer architecture. Subsequently, the reader can move on to reading research papers for getting a deeper understanding of state-of-the-art techniques in processor design.

The reader should start with some basic references in the design of out-of-order processors [Brown et al., 2001, Smith and Sohi, 1995, Hwu and Patt, 1987]. After getting a basic understanding, she can move on to read papers that propose important optimizations such as [Brown et al., 2001, Petric et al., 2005, Akkary et al., 2003]. For a thorough understanding of branch prediction schemes and fetch optimization, the reader should definitely look at the work of Yeh and Patt [Yeh and Patt, 1991, Yeh and Patt, 1992, Yeh and Patt, 1993], and the patent on Pentium 4 trace caches [Krick et al., 2000].

Simultaneously, the reader can also look at papers describing the complete architecture of processors such as the Intel Pentium 4 [Boggs et al., 2004], Intel ATOM [Halfhill, 2008], Intel Sandybridge [Gwennap, 2010], AMD Opteron [Keltcher et al., 2003], and IBM Power 7 [Ware et al., 2010]. Finally, readers can find descriptions of state-of-the-art processors in the periodical, "Microprocessor Report", along with emerging trends in the processor industry.

Exercises

Pipeline Stages

Ex. 1 — Show the design of the IF, OF, EX, MA, and RW pipeline stages. Explain their functionality in detail.

Ex. 2 — Why do we need to store the *op*2 field in the instruction packet? Where is it used?

Ex. 3 — Why is it necessary to have the *control* field in the instruction packet?

Ex. 4 — Why do we require latches in a pipeline? Why are edge sensitive latches preferred?

Ex. 5 — Why is it necessary to split the work in a data path evenly across the pipeline stages?

* **Ex. 6** — We know that in an edge sensitive latch, the input signal has to be stable for t_{hold} units of time after the negative edge. Let us consider a pipeline stage between latches L_1 and L_2. Suppose the output of L_1 is ready immediately after the negative edge, and almost

instantaneously reaches the input of L_2. In this case, we violate the hold time constraint at L_2. How can this situation be avoided?

Pipeline Design

Ex. 7 — Enumerate the rules for constructing a pipeline diagram.

Ex. 8 — Describe the different types of hazards in a pipeline.

Ex. 9 — In the *SimpleRisc* pipeline, why don't we have structural hazards?

Ex. 10 — Why does a branch have two delay slots in the *SimpleRisc* pipeline?

Ex. 11 — What are the `Data-Lock` and `Branch-Lock` conditions?

Ex. 12 — Write pseudo-code for detecting and handling the `Branch-Lock` condition? (without delayed branches)

Ex. 13 — What is delayed branching?

* **Ex. 14** — Let us consider two designs: D_1 and D_2. D_1 uses a software-based approach for hazards, and assumes delayed branching. D_2 uses interlocks, and assumes that a branch is not taken till the outcome is decided. Intuitively, which design is faster?

Ex. 15 — Assume that 20% of the dynamic instructions executed on a computer are branch instructions. We use delayed branching with one delay slot. Estimate the CPI, if the compiler is able to fill 85% of the delay slots. Assume that the base CPI is 1.5. In the base case, we do not use any delay slot. Instead, we stall the pipeline for the total number of delay slots.

Ex. 16 — Describe the role of the forwarding multiplexers in each stage of the pipeline.

Ex. 17 — Why do we not require a forwarding path from MA to EX for the $op2$ field?

Ex. 18 — Answer the following questions.

i) What are the six possible forwarding paths in our *SimpleRisc* processor?

ii) Which four forwarding paths, are required, and why? (Give examples to support your answer).

Ex. 19 — Assume that we have an instruction immediately after a call instruction that reads ra. We claim that this instruction will get the correct value of ra in a pipeline with forwarding. Is this true? Prove your answer.

Ex. 20 — Reorder the following code snippet to minimize the execution time for the following configurations:

1. We use software techniques, and have 2 delay slots.
2. We use interlocks, and predict not taken.
3. We use forwarding, and predict not taken.

```
add r1, r2, r3
sub r4, r1, r1
mul r8, r9, r10
cmp r8, r9
beq .foo
```

Ex. 21 — Reorder the following code snippet to minimize execution time for the following configurations:

1. We use software techniques, and have 2 delay slots.
2. We use interlocks, and predict not taken.
3. We use forwarding, and predict not taken.

```
add r4, r3, r3
st  r3, 10[r4]
ld  r2, 10[r4]
mul r8, r9, r10
div r8, r9, r10
add r4, r2, r6
```

Ex. 22 — Answer the following:

```
add r1, r2, r3
sub r4, r1, r6
ld  r5, 10[r4]
add r6, r5, r5
sub r8, r8, r9
mul r10, r10, r11
cmp r8, r10
beq .label
add r5, r6, r8
st  r3, 20[r5]
ld  r6, 20[r5]
ld  r7, 20[r6]
lsl r7, r7, r10
```

i) Assuming a traditional *SimpleRisc* pipeline, how many cycles will this code take to execute in a pipeline with just interlocks? Assume that time starts when the first instruction reaches the RW stage. This means that if we had just one instruction, then it would have taken exactly 1 cycle to execute (Not 5). Moreover, assume that the branch is not taken. [Assumptions: No forwarding, No delayed branches, No reordering]

ii) Now, compute the number of cycles with forwarding (no delayed branches, no reordering).

iii) Compute the minimum number of cycles when we have forwarding, and we allow instruction reordering. We do not have delayed branches, and in the reordered code, the branch instruction cannot be one of the last three instructions.

iv) Compute the minimum number of cycles when we have forwarding, allow instruction reordering, and have delayed branches. Here, again, we are not allowed to have the branch instruction as one of the last three instructions in the reordered code.

** **Ex. 23** — We have assumed up till now that each memory access requires one cycle. Now, let us assume that each memory access takes two cycles. How will you modify the data path and the control path of the *SimpleRisc* processor in this case.

** **Ex. 24** — Assume you have a pipeline that contains a value predictor for memory. If there is a miss in the L2 cache, then we try to predict the value and supply it to the processor. Later this value is compared with the value obtained from memory. If the value matches, then we are fine, else we need to initiate a process of recovery in the processor and discard all the wrong computation. Design a scheme to do this effectively.

Performance and Power Modeling

Ex. 25 — If we increase the average CPI (Cycles per Instruction) by 5%, decrease the instruction count by 20% and double the clock rate, what is the expected speedup, if any, and why?

Ex. 26 — What should be the ideal number of pipeline stages (x) for a processor with $CPI = (1 + 0.2x)$ and clock cycle time $t_{clk} = (1 + 50/x)$?

Ex. 27 — What is the relationship between dependences in a program, and the optimal number of pipeline stages it requires?

Ex. 28 — Is a 4 GHz machine faster than a 2 GHz machine? Justify your answer.

Ex. 29 — How do the manufacturing technology, compiler, and architecture determine the performance of a processor?

Ex. 30 — Define dynamic power and leakage power.

* **Ex. 31** — We claim that if we increase the frequency, the leakage power increases. Justify this statement.

Ex. 32 — What is the justification of the ED^2 metric?

* **Ex. 33** — How do power and temperature considerations limit the number of pipeline stages? Explain your answer in detail. Consider all the relationships between power, temperature, activity, IPC, and frequency that we have introduced in this chapter.

* **Ex. 34** — Define the term DVFS.

** **Ex. 35** — Assume that we wish to estimate the temperature at different points of a processor. We know the dynamic power of different components, and the leakage power as a function of temperature. Furthermore, we divide the surface of the die into a grid as explained in Section 10.10.4. How do we use this information to arrive at a steady state value of temperature for all the grid points?

Interrupts and Exceptions

Ex. 36 — What are precise exceptions? How does hardware ensure that every exception is a precise exception?

Ex. 37 — Why do we need the *movz* and *retz* instructions?

Ex. 38 — List the additional registers that we add to a pipeline to support interrupts and exceptions.

Ex. 39 — What is the role of the CPL register? How do we set and reset it?

Ex. 40 — How do we locate the correct interrupt handler? What is the structure and role of an interrupt handler?

Ex. 41 — Why do we need the registers $oldPC$, and $oldSP$?

Ex. 42 — Why do we need to add a $flags$ field to the instruction packet? How do we use the $oldFlags$ register?

* **Ex. 43** — Consider a hypothetical situation where a write back to a register may generate an exception (register-fault exception). Propose a mechanism to handle this exception *precisely*.

* **Ex. 44** — Define the concept of register windows. How can we use register windows to speed up the implementation of functions?

Advanced Topics

Ex. 45 — Can you intuitively say why most of the branches in programs are predictable?

Ex. 46 — Is the following code sequence amenable to branch prediction. Why or why not?

```
int status=flip_random_unbiased_coin();
if (status==Head)
        print(\head");
else
        print(\tail");
```

Ex. 47 — We need to design a 2-issue inorder pipeline that accepts a bundle of two instructions every cycle. These bundles are created by the compiler.

(a) Given the different instruction types, design an algorithm that tells the compiler the different constraints in designing a bundle. For example, you might decide that you don't want to have two instructions in a bundle if they are of certain types, or have certain operands.

(b) To implement a two issue pipeline, what kind of additional functionality will you need in the MEM stage?

Ex. 48 — Describe the main insight behind out-of-order pipelines? What are their major structures?

Design Problems

Ex. 49 — Implement a basic pipelined processor with interlocks using Logisim (refer to the design problems in Chapter 9).

Ex. 50 — Implement a basic pipelined processor in a hardware description language such as Verilog or VHDL. Try to add forwarding paths and interrupt processing logic.

Ex. 51 — Learn the language SystemC. It is used to model hardware at a high level. Implement the *SimpleRisc* pipeline in SystemC.