

# 5

## Alternative Approaches to Issue and Commit

In Chapter 4 we learned about the basic structure of an OOO pipeline. We further realized that a modern OOO machine is a very complex piece of hardware. To ensure performance without sacrificing on correctness, we need to add many additional hardware structures and do a lot of book keeping. The processor that was designed in Chapter 4 is very suitable for high performance implementations.

However, given that people have been designing processors for the last fifty years, there are many other designs of processors out there. Some of these techniques are for smaller embedded processors, some techniques are very power efficient at the cost of performance, and some techniques export the complexity to software. The aim of this chapter is to discuss all those *additional* techniques. Note that this chapter should be viewed as a sequel to Chapter 4. Unlike Chapter 4, this chapter discusses an assorted set of techniques, which are mostly unrelated to each other. Nevertheless, we have made a modest effort to classify these areas into the following categories:

**Support for Aggressive Speculation and Replay** Most OOO processors make guesses based on behaviors observed in the past for predicting different parameters such as the latency of memory operations – this information is used to optimistically assume that a given memory access always finds its value in the L1 cache. This is known as *speculation*. However, sometimes these guesses turn out to be wrong, then it is necessary to go back and fix the state. Some instructions, which might have potentially got wrong data, need to be *replayed*.

**Simpler Designs of OOO Pipelines** It is not necessary to have large physical register files, free lists, and separate ROB in OOO pipelines. Depending on the workloads, we can merge some of these structures, and end up with a simpler and more power efficient design.

**Software based Techniques** It is not necessary to export all the complexity to hardware. It is possible to increase the ILP of the code by applying compiler based transformations. These *software approaches* are extremely useful and are an integral part of today’s compiler tool chain. Some software based approaches require details of the underlying hardware, whereas, some others are generic.

**EPIC Processors** Most compiler based approaches are useful for generic OOO pipelines. However, there is a school of thought that advocates making the hardware significantly simpler and exporting the entire complexity to software. It is the software’s job to sequence and schedule the

instructions. Simpler hardware translates to area and power efficiency. Such EPIC (Explicitly Parallel Instruction Computing) processors often require complex compiler infrastructure with some specialized hardware support to implement the directives produced by the compiler.

### 5.0.1 Organization of this Chapter

At the outset, we shall discuss some more methods to increase the ILP in modern processors by aggressively speculating on load latencies, addresses, and values. We shall discuss some of the most common techniques in this space in Section 5.1. Recall that whenever we have speculation, there is always the possibility of an error. There is thus a need to *replay* some instructions, which might have been erroneously issued with wrong data (see Section 5.2).

Then, we shall proceed to discuss alternative designs of OOO pipelines in Section 5.3 that are less complicated. Instead of designing simpler hardware structures, we can also move some of the complexity to software. We will focus on compiler driven approaches to increase ILP in Section 5.4. As an extreme case, it is possible to keep hardware ultra-simple, and instead do all the scheduling, instruction sequencing, and renaming in software (by the compiler). We shall discuss such approaches in Section 5.5. Finally, in Section 5.6 we shall discuss the design of the Intel® Itanium® processor, which is an EPIC processor.

## 5.1 Load Speculation

### 5.1.1 Introduction

Let us start out with making the design of the processor – introduced in Chapter 4 – more complicated and more realistic. Let us begin by taking into cognizance that one of the primary sources of non-determinism in execution is the memory system. A memory access such as a load or a store is a multi-step process. We first compute the memory address, and then we check for dependences in the load-store queue (see Section 4.3). We either forward a value from a store to a load or we send the requests to the memory system. The memory system is in itself a very complicated network of components. We have a hierarchy of caches, and then finally the main memory. A request can be serviced at any level in the memory system. Thus, the duration of a memory request is fundamentally non-deterministic in nature. It is very hard to predict if, for example, a load instruction will get its value from another store instruction, or from which level in the cache hierarchy. As we shall see, instead of adopting a very conservative stance where we wait for a step to fully complete before starting the next step, it is often wiser to speculate and go forward.

*Speculation* is a very common technique in computer architecture. Almost all kinds of speculation involve the following steps: make a prediction regarding the value of an input, use it to compute the output, and forward the output to other dependent instructions. Such predictions help us execute instructions *sooner*. However, the flip side is that it is possible that sometimes these predictions might be wrong. In that case we need to locate all the instructions that might have potentially received a wrong value, and cancel them. Then these instructions need to be re-executed with the correct values.

Speculation has many advantages. It helps break dependences. For example, consider a load-use dependence, where a load instruction  $L$  supplies its value to a consumer instruction  $I$ . If we can predict the value read by instruction  $L$  reliably, and give it to instruction  $I$ , we can execute  $L$  and  $I$  in parallel. This will break dependence chains, increase ILP, and tremendously increase performance. The crux of this mechanism lies in designing an accurate predictor, and to a lesser extent, we need to also have a fast method of recovering from the ill effects of a misprediction.

**Definition 23**

Speculation is a very common technique in computer architecture where we predict something and proceed on the basis of the prediction. Consider the steps involved while predicting the input of an instruction.

1. We predict the value of an input.
2. The input is used to proceed with the execution of an instruction.
3. The instruction using the predicted input can forward its result to other instructions.
4. At a later point of time, the prediction is verified, and if we find that the prediction is incorrect, then all the influenced instructions are canceled.
5. This technique breaks dependences between instructions and thus increases the available ILP. This leads to an increased IPC.

Speculation is not limited to predicting the inputs of instructions, we can also predict the output of an instruction, or its duration.

Let us try to apply speculative techniques to load instructions. Before the astute reader asks, “Why load instructions?”, let us answer this question. Load instructions typically have non-deterministic latencies and this can cause a lot of dependent instructions to get queued in the instruction window. This is also known as the *convoy effect* because the situation is similar to a road where a car breakdown can cause a huge traffic jam. Furthermore, these convoys of instructions can be fairly long because a load can take 100s of cycles if it needs to fetch its data from main memory.

Here are the primary methods for speculation with regard to load instructions.

**Address Speculation** Based on historical values, we try to predict the address of load instructions.

If we know the address early, we can try to get forwarded values in the load-store queue, or fetch the value from the memory system in advance. This will save us valuable cycles, because we are in effect executing the load instruction and its dependent instructions early.

**Load-Store Dependence Speculation** We try to predict the dependence between loads and stores in the load-store queue. Based on predicted dependences, we can take decisions to forward values, wait for unresolved stores, or send requests to memory.

**Latency Speculation** We predict if a load hits in the L1 cache or not, and consequently the latency if it is a hit. If we predict a hit, we can wake up dependent instructions early such that they can execute as early as possible (see Section 4.2 for a detailed discussion on instruction wakeup).

**Value Prediction** Finally, we can predict the value that a load instruction is expected to read. This can then be passed on to dependent instructions.

We shall present different methods of prediction in this section, and methods to replay instructions in Section 5.2.

### 5.1.2 Address Speculation

The aim here is to design a method to predict the address of a load instruction. The only information in our arsenal is the program counter (PC) of the load. Let us discuss two fairly straightforward solutions that work in most common cases. These solutions are similar to branch predictors. In fact, we shall

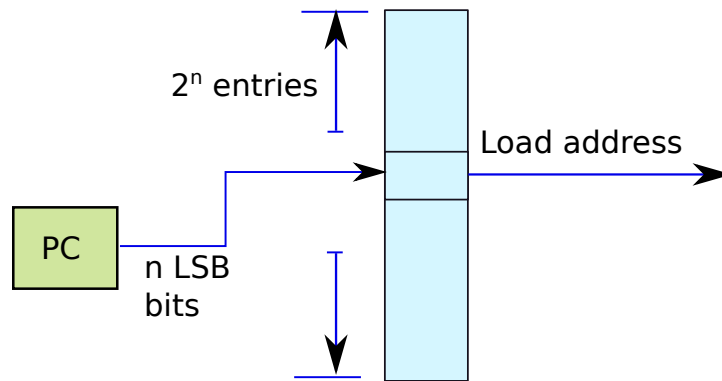


Figure 5.1: Load address predictor (based on the last computed address for this PC)

see that most predictors used to predict a host of different things are similar to our branch predictors presented in Chapter 3.

### Predict Last Address

We maintain a table that is indexed by the least significant  $n$  bits of the PC address. In each entry, we store the memory address computed by the memory instruction the last time it was invoked.

As we can see in Figure 5.1, we use  $n$  bits from the PC address to index a  $2^n$  entry table. Every time the memory address of a load is computed, we store it in this table. This is later on used for prediction.

We can use standard techniques that we had learned in Chapter 3 to increase the accuracy of this process. For example, to avoid the effects of aliasing, we can keep some bits of the PC address in each entry. We will use the entry only if these bits match with the corresponding bits of the PC address. In addition, we can use a 2-bit saturating counter. The state 00 indicates that with a strong probability the address that we read is wrong. Likewise, the state 11 indicates that with a high probability, the address is correct. This shall work in exactly the same way. Every time the prediction is correct, we increment the counter, and every time the prediction is wrong, we decrement the counter. Depending upon our appetite for risk (captured by the saturating counters), we can use the predicted address accordingly.

### Stride based Prediction

Let us now discuss a more sophisticated idea. Let us consider one of the most common scenarios where array accesses are used. One such example is a simple *for* loop used to iterate through all the elements of an array.

Consider the following piece of code (C code and equivalent assembly code).

C code

```
int sum = 0, arr[10];
for (i=0; i < 10; i++){
    sum += arr[i];
}
```

## Assembly code

```

// Let us assume that the base address of arr is in r0
mov r1, 0          // i = 0
mov r2, 0          // sum = 0

.loop: cmp r1, 10   // compare i with 10
beq .exit          // if (r1 == 10) exit
ld r3, [r0]        // load arr[i] to r3
add r2, r2, r3      // sum += arr[i]
add r0, r0, 4       // increment the memory address
add r1, r1, 1       // increment the loop index
b .loop

```

In this case the load associated with accessing the array, *arr*, is called repeatedly. Every time the array index stored in register *r1* increases by 1, the memory address gets incremented by 4 bytes (assuming the size of an integer is 4 bytes). Let's say, we want to predict the address of the single load instruction. In this case we shall perceive the address increasing by 4 every iteration. The address is thus predictable. There is a pattern, and if we are able to decipher the pattern, then we can successfully predict the address of the load for most of the iterations of the loop.

Whenever a given variable increases by a fixed value every iteration, this value is known as a *stride*. In this case we need to figure out the stride, and the fact that the memory access pattern is based on strides. Strides are a very common access pattern particularly when arrays are involved, and there are standard methods of handling them. Mathematically, we need a minimum of three iterations to identify a stride based access pattern.

We create a table with  $2^n$  entries that can be accessed using the least significant  $n$  bits of the PC. In each entry, we need to store the following information: memory address that was computed the last time the load instruction was executed ( $A$ ), the value of the stride ( $S$ ), and a bit indicating if a stride based access pattern is followed or not ( $P$ ). For a prediction, we simply predict  $A + S$  if the access pattern is based on strides.

At a later point of time, when we compute the address of this load to be  $A'$ . We need to verify that we are following a stride based access pattern. Hence, we compute  $S' = A' - A$ , and compare this with the previous stride,  $S$ , stored in the entry. If the strides match, then we can conclude that we are following a stride based access pattern; we set  $P = 1$  (stride access pattern bit set to 1). Otherwise, we set  $P = 0$ : do not make a prediction using strides. In either case, we set  $A = A'$  and  $S = S'$ .

### 5.1.3 Load-Store Dependence Speculation

Let us look at another potential source of performance improvement. In Section 4.3, we indicated that a load cannot be sent to the memory system as long as it has an unresolved (address not computed) store before it in the load-store queue. This means that the load needs to wait for such stores. Let us assume that we have ten load instructions whose address has been computed. However, there is one store instruction before them whose address is yet to be computed. In this case, the ten load instructions have to wait. Later on if we find out that the address of the store does not conflict with the addresses of any of these load instructions, we would realize that we waited in vain. This situation would represent a complete waste of time and ILP.

To handle all such situations, modern computer architectures typically *speculate*. For example, if in this case we can confidently predict if the store instruction will conflict with any of the resolved load instructions or not, then we can speculate. If the prediction is false, which means that the store is not expected to conflict, then we can send all the load instructions to the memory system before the store's address is resolved. Note that here we are making a guess. However, this is an intelligent guess because we have a mechanism to predict load-store dependences effectively. If there is a high probability of the prediction being correct, then we shall gain a lot of performance with this technique. A load will not

wait for unresolved stores before it in program order, unless our load-store dependence predictor predicts a dependence. Such aggressive speculation mechanisms are indeed extremely helpful. However, there is a flip side to every good idea. Here again, we can have the problem of occasional mispredictions.

We use the solution described in a later section (Section 5.1), where we rely on a replay mechanism that identifies the instructions that have possibly read wrong values, nullifies them, and reissues them with the correct value. Let us now proceed to describe the design of such load-store dependence predictors.

### Design of a Load-Store Dependence Predictor: Collision-based Predictor

Let us start with showing the design of a predictor that predicts if a given load collides (has the same address) with any of the preceding stores or not (see Yoaz et al. [Yoaz et al., 1999]). If a load is predicted to not collide with any of the preceding stores, then we can immediately execute the load.

The main insight here is that loads display roughly consistent and predictable behavior. This means that if a given load is non-colliding in nature, it remains so for some time. We can thus have a collision history table (CHT). The simplest design of a CHT is like a branch predictor. Here, we have a table of  $2^n$  entries, which is addressed using the last  $n$  bits of the PC. In each location, we store the prediction: 0 or 1. We can further augment this table by having a saturating counter instead of a single bit. This will create some degree of hysteresis as we had done in the case of a branch predictor and also indicate the confidence of the prediction. For example, if a given load has been historically non-colliding, then one collision should not be able to change its behavior. As we can observe, the saturating counter based predictor is a generic mechanism that can be reused in almost all cases where we need a binary prediction, as was done in this case.

The overall scheme is thus as follows.

1. When a load is either scheduled, or when its input operands are ready, we access the CHT. If we predict the load to be non-colliding, then as soon as the memory address is ready, the load instruction can be sent to the memory system.
2. However, if we predict the load to be colliding, then the load needs to wait in the LSQ till all the preceding stores are resolved.
3. Once we have computed the addresses of all the previous store instructions, we are in a position to determine if the load collided with any stores or not. We can then update the CHT accordingly with the correct value.

This mechanism is simple and works well in practice. However, it is possible to improve it even further. Let us consider a common access pattern: saving and restoring registers while calling functions. In this case, we store a register's value in memory before entering a function, and then restore its value from memory when the called function exits. Let us consider the loads that restore the state of the registers. The colliding stores for these loads are the stores that spill the registers to memory. If the behavior of the function is roughly consistent and predictable, then we roughly know the distance between a conflicting load and store in terms of regular instructions or memory instructions. For example, if there are typically 10 memory instructions between the store that saves the value of a register, and the corresponding load that restores the value, then the distance between the load-store pair is 10 memory instructions. We can make use of this fact very effectively. When the load's address is computed, we can make it wait till there are less than 10 instructions before it in the LSQ. By this time if it has not gotten a forwarded value, then the load is ready to be sent to memory. This is because the chances of it colliding with a store are very little – we have predicted with high confidence that the distance between the load and store is 10 intervening memory instructions. This condition does not hold if there are less than 10 instructions preceding the load in the LSQ (refer to Figure 5.2).

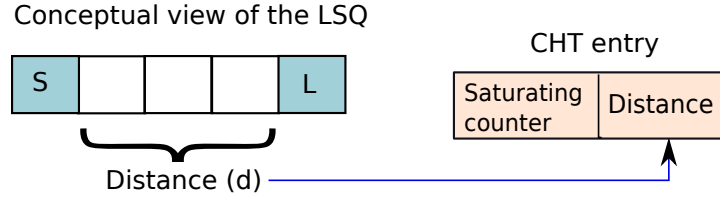


Figure 5.2: Load-store distance based prediction

We need to make a minor modification to the CHT. In each entry, we additionally store the distance between the load and store instruction. Whenever, we forward a value from a store to a load, we compute the distance between them in the LSQ (number of intervening entries), and store this in the CHT entry.

The prediction algorithm is thus as follows:

1. Either at the time of instruction dispatch, or at the time of computing the load's memory address, we access the CHT. If the load is not predictable as indicated by the saturating counter, then we make it wait till all prior stores are resolved. Subsequently, the load is sent to the memory system. Otherwise, we do the following.
2. If a load is predictable, we wait till there are less than  $N$  preceding entries in the LSQ. The value of  $N$  is stored in the CHT's entry, and denotes the predicted load-store distance.
3. If we get a forwarded value from a store, we use it and move forward.
4. Else, if there are less than  $N$  entries in the LSQ, then we send the load to the memory system.

This algorithm simply makes the load wait for some time in the hope of getting a forwarded value in the LSQ from an earlier store. However, loads do not wait forever. They wait till the number of preceding entries is below a threshold, and then the load is sent to the memory system. The main advantage of this improved scheme is that we reduce the number of replays

### Load-Store Dependence Predictor: Store Sets

Now, that we have discussed methods to find if a load collides with other stores or not, let us move one step further. Let us also focus on stores, and see if we can predict load-store pairs that are expected to collide. If we can design such a predictor, then we can do two things:

1. We can delay a load from being sent to the memory system, if the predicted store is present in the LSQ or in the instruction window and precedes the load. This will reduce the number of replays and mispredictions.
2. Once the address of a store in a predicted load-store pair is resolved, we can forward the value from the store to the load if the addresses are the same. If there are no such stores, we can send the load instruction to memory at this stage.

As compared to the previous approach that predicted just on the basis of the PCs of loads, this approach uses more information. It takes the PCs of both loads and stores, and makes the predictions on the basis of load-store pairs. Whenever, we use more information, we expect the prediction to in general be better. Let us show the design of one such predictor that uses the concept of *store sets* [Chrysos and Emer, 1998]. Note that we always assume that a given load or store will behave the same way in the near future as it has been doing in the recent past.

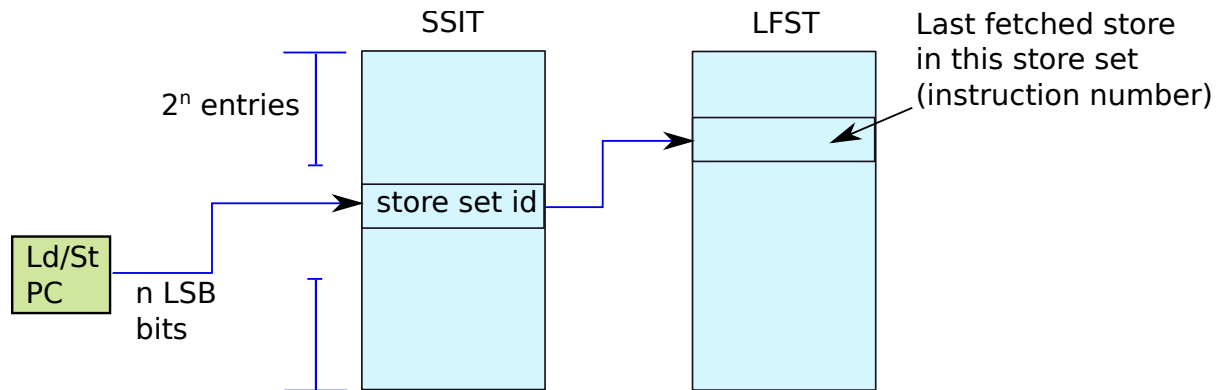


Figure 5.3: A predictor that uses store sets

For every load we associate a *store set*, which is the set of stores that have forwarded a value to the given load in the recent past. The exact mechanism is as follows. We have two tables: SSIT (Store Set Identifier Table), and LFST (Last Fetched Store Table).

The SSIT as shown in Figure 5.3 is a table that is indexed by either a load PC or a store PC. As usual, we consider the last  $n$  bits of the PC address, and access the SSIT table. Each entry of the SSIT table contains a store set identifier. It is a unique identifier that is assigned to each and every store set. If a load instruction accesses the SSIT, it reads the identifier of the store set that is associated with it. Similarly, if a store accesses the SSIT, it reads the identifier of the store set that it is a part of. There are two things to note. In both the cases (for a store and load), it is possible to read an invalid identifier. This means that the given load or store is not associated with a valid store set. Furthermore, to keep things simple, we can decide to make a store a part of only one store set. Otherwise, in each entry in the SSIT, we need to maintain multiple store set ids. This is a source of additional complexity. Note that later works [Moshovos et al., 1997, Moshovos and Sohi, 1999] did explore associating multiple store sets with a given store. However, let us explain the basic idea, where we assume that a store can only be a part of only one store set at a time.

Once we have read the id of the store set, we can use this id (if it is valid) to access the LFST. The corresponding entry in the LFST contains the instruction number of the store that was last fetched in the store set. The term *instruction number* is a unique identifier of an instruction in the pipeline. When an instruction enters the pipeline, we assign it a unique id, and it keeps this id till its retirement. Note that at any point of time, we cannot have two instructions in the pipeline with the same instruction number.

Let us now use the SSIT and LFST to create an algorithm that uses store sets. Consider the case of a load ( $L$ ) first. After it is decoded, we access the SSIT with the PC of the load. There are two cases: either we get a valid store set id, or we do not. If we get a valid store set id, then we use it to access the LFST, otherwise we do not do anything. In the LFST (indexed by the store set id), we get the instruction number of the last fetched store in the load's store set. Let us refer to this store as  $S$ . Now, this means that there is a high probability that this store might supply its value to the load. Again, at this point, we are not sure because the address of the load has not been computed. Still there is a probability, and we should be aware of that. We thus add the instruction number of the store to the load's instruction packet.

Now, let us consider the case when a store ( $S$ ) gets decoded. Similar to the case of the load, we look up the address of the store in the SSIT. Recall that the SSIT has entries for both loads and stores. If the corresponding entry in the SSIT has a valid store set id, then we use it, otherwise we simply move ahead. If a valid id is found, we use it to access the LFST and write the instruction number of the current store to the entry in the LFST indexed by the store set id. This tells the LFST that the current



store ( $S$ ) is the most recent store in the store set.

There are several ways in which we can enforce a dependence. For example, we can proceed to find the instruction window entry of the store  $S$ , and then wait for  $S$  to be issued first. To find the instruction window entry we need a separate storage structure that maps the instruction number to the instruction window entry. By ensuring that there is an order in issuing the instructions, we can ensure that  $S$  is issued first, and then load  $L$  is issued. After the store is issued, we read the value it is going to write to the memory system from the register file. Since, we have predicted a load-store dependence, this value can be directly given to the load instruction. The load instruction can thus start early, albeit speculatively, and broadcast its result to instructions that are waiting for it. Another way of enforcing the dependence is to search in the LSQ for store  $S$  once load  $L$  is resolved. If  $S$  is present before  $L$  then we wait for it to get resolved before sending  $L$  to memory.

What did we gain in this process? We were able to significantly cut down on the latency of the load instruction. We did not have to send it to memory, nor wait for any value to be forwarded in the LSQ. It also allowed us to predict a dependence with a store, and directly use the value that it is going to store. Then, we were then able to wake up the consumers of the load.

To enable this mechanism in the instruction window, wakeup and broadcast mechanisms have to be modified slightly. Any store with a valid store set id, can broadcast its instruction number. Any load waiting on that instruction number can wake up and then get the value that the store is writing.

Recall that after the address for the store is computed, we do not send it to the memory system (Section 4.4), we rather wait for it to retire. This store will thus remain in the LSQ. Let us see what happens after we compute the address of the store. Since it has an entry in the LSQ, we can use it to forward values to any subsequent loads that come. Even if the address of the load is not computed, we can still speculatively forward the data if the store instruction is the latest instruction in the store set of the load. Upon retirement, we need to invalidate the LFST entry for the store, if it still points to it.

The last piece of the puzzle is the creation of an entry in the SSIT. There are several ways to do this, and there are different trade-offs associated with them. Let us discuss a simple scheme. Whenever we detect a dependence between a store  $S$  and a subsequent load  $L$  in the LSQ, we need to see if we need to create an SSIT entry or not. We first check the SSIT entry for  $S$ . If there is a valid entry then we do not do anything, otherwise we check the SSIT for  $L$ . If it is associated with a store set whose id is  $i$  then we set the store set of  $S$  to  $i$ . We write the instruction number of  $S$  to the  $i^{th}$  entry of the LFST. Otherwise, if  $L$  does not have an associated store set entry we need to create a new store set id for both  $L$  and  $S$  and then populate the LFST.

Finally, let us discuss the issue of the instruction number. We chose to have a 7 or 8 bit instruction number as opposed to identifying a store by its 64-bit PC. This was done to reduce the required storage and the resources required to broadcast the id of the store. If there are 160 entries in the ROB, then we can simply create an 8-bit counter that is incremented (modulo 256) every time a new instruction is decoded. The instruction number of an instruction will be the value of this count; we are guaranteed to never have two instructions in the pipeline with the same instruction number.

#### 5.1.4 Latency Speculation

Let us consider a simple system with a pipeline, an L1 cache, and an L2 cache. An access can either be a hit in the L1 cache or a miss. If it misses in the L1 cache, then it goes down the memory system, and it can either get its data from the L2 cache, or somewhere from deep within the memory system such as main memory or the swap space in the hard disk. The latency of a cache miss is not very predictable, and thus it is unwise to assume anything about the latency of a miss in a cache. However, we can assume that a very large percentage of our memory accesses will hit in the L1 cache. Furthermore, the latency of a hit in the L1 cache is almost always a constant. It is a very small number – typically between 1-3 cycles.

We have two options here. The first option is very conservative. Here, we wait for the load to read its data, before we wake up consumer instructions. In this case, we need to wait for the load instruction to

hit or miss in the cache. The problem here is that it is not possible to ensure back-to-back execution (see Section 4.2.4) of a load instruction, and a subsequent instruction that uses the value read by the load. Instead, once we get the status from the cache (might take 1-2 cycles), we send a wakeup signal to the consumer instructions in the instruction window. This is an inefficient process because this ensures that instructions consuming the result of load instructions are ready to be executed after a delay of several cycles.

Instead, if we assume that every load instruction hits in the L1 cache, we might do much better. This is because most programs have a L1 hit rate of roughly 85-90% or more. This means that most of the time, our loads have a deterministic latency, and thus we can wake up instructions such that they are ready just in time to consume the value read by the load (via the bypass network), and continue execution. We can therefore significantly reduce the delay between issuing a load instruction and issuing the instructions that consume its value.

This is shown in Figure 5.4. For the pipeline shown in Figure 5.4, we have been able to save 2 cycles, in other words, we have been able to execute the consumer instruction 2 cycles early.

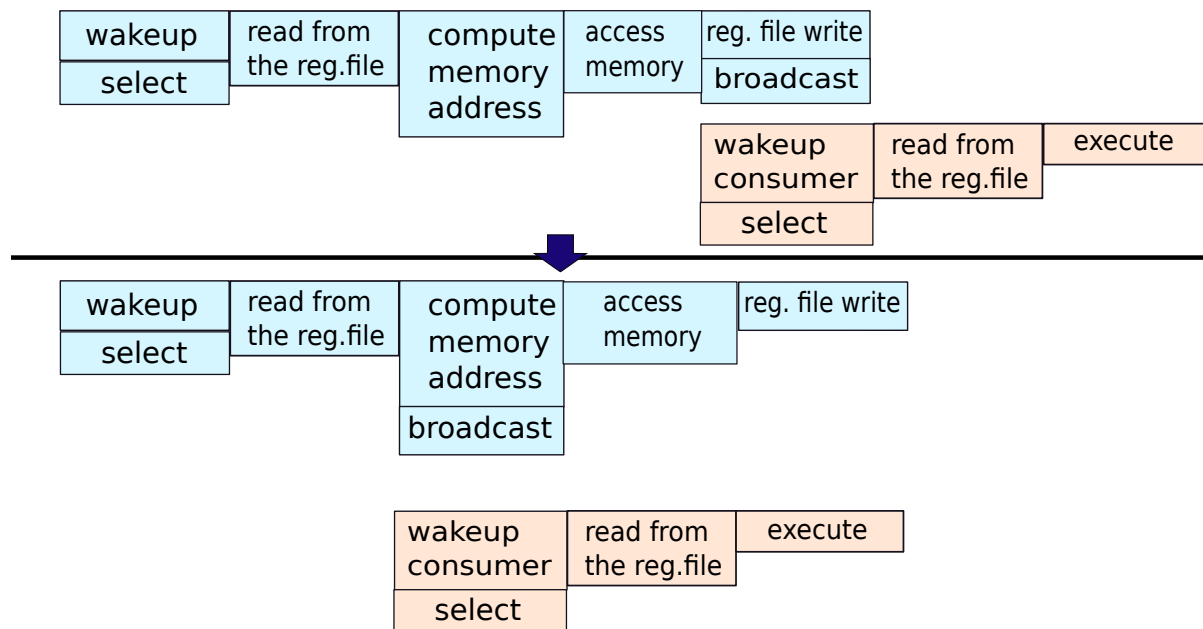


Figure 5.4: Load latency speculation

As of today, load latency speculation is a standard feature in almost all high performance OOO processors, particularly, in caches with multi-cycle hit times. We can gain a lot of performance by issuing consumer instructions early – before we determine if the load has had a hit or miss in the cache.

What about the remaining instructions (roughly 10%) that miss in the L1 cache? If we speculate on the latency of loads, we end up issuing dependent instructions before ascertaining if the load has hit in the L1 cache or not. Since these instructions are issued, they will try to pick up a value from the bypass network or the register file, and proceed. If the load has not completed, these instructions will pick up junk values and continue in the pipeline. There is a need to dynamically cancel these instructions, and reissue them once the correct value is read by the load instruction from the memory system. In this case, the load instruction will read its value from the lower levels of the memory system – L2 and beyond. This requires a replay mechanism (will be discussed in Section 5.2). Before that, let us discuss methods to optimize load latency speculation, and also discuss other forms of speculation in this space.

## Hit-Miss Predictor

Let us discuss a basic hit-miss predictor (refer to [Yoaz et al., 1999]). Designing this is easy given all the designs that we have already seen. We need to take inspiration from the branch predictors that we had designed in Chapter 3. This is again a case of binary prediction, where we need to predict a hit (1), or a miss (0).

The simplest implementation of this predictor uses a  $2^n$  entry table, where we index it using the last  $n$  bits of the PC of the load instruction. The assumption in all such cases is that the historic behavior of a load instruction will continue to be predictive of the future, at least the near future. We can either use a simple 1-bit predictor, or a predictor that uses saturating counters. All the optimizations that we used in the case of branch predictors can be used here such as storing the tags for reducing aliasing.

In general having only one hit-miss predictor for the L1 cache is considered to be sufficient. We can in theory have more predictors for other caches such as the L2 and L3 caches. However, in such cases prediction accuracies are not known to be that great, and also the latency of a cache access at such levels is not very predictable. As we shall see in Chapter 7, an L2 or L3 cache is a fairly complex entity, and does not have a fixed latency.

### 5.1.5 Value Prediction

The idea here is to predict the value that the load instruction will read. Let us first list some main sources of value predictability. Some early studies in this area were performed by Lipasti et al. [Lipasti et al., 1996].

- **Input Sets:** In most programs, the inputs exhibit a tremendous amount of predictability. Assume we have an application that processes HTML files. A lot of the values, particularly, towards the beginning and end of the file (HTML tags) are expected to be the same. Moreover, two web pages from the same organization will also have a lot of data in common.
- **Constants in the program:** Most programs rely on a lot of read-only data or data that is computed once and reused many times. These values are very predictable.
- **Base addresses:** Most of the time the base addresses of arrays, functions, and objects tend to remain the same throughout the execution of a program. When we load these addresses we can leverage the advantages of value prediction.
- **Virtual functions:** Programs in object oriented languages such as C++ often use a virtual function table that stores the addresses of the starting addresses of functions. Loads to read this table return very predictable values because this table typically does not change.
- **Register spilling:** Recall that when we run out of registers, or when we call a function, we need to write the values of some registers to memory. Their values are loaded later on. Many of these values remain constant, and are thus highly predictable.

### Value Prediction Techniques

By this time most readers would have figured out the general pattern. We create a  $2^n$  entry table that is indexed by  $n$  bits from the load instruction's PC. In this table we can store the predicted value, and then use the same optimizations that we have been using up till now. Lipasti et al. refer to this table as the LVPT (load value prediction table).

In some cases, we can leverage stride based patterns [Wang and Franklin, 1997] as we had studied in Section 5.1.2. Here, the value stored in a memory address changes by a fixed increment every time we issue the load instruction. There are fundamental reasons why such stride based patterns are more relevant for predicting memory addresses (see Section 5.1.2) as opposed to memory values. This is because most compilers try their best to put all the variables into registers. If a variable is getting

regularly incremented, most likely the updates will remain confined to the register file. The updates will reach memory when the register is spilled either because of a function call or because we run out of registers. Most of the time such writes do not have a fixed and regular pattern. Hence, if we are predicting the values of memory values, we might not see a lot of benefits with stride-based prediction.

A promising set of techniques use some compiler support [Gabbay and Mendelson, 1997]. We add code to write the values of memory addresses to a file (this method is called *profiling*). Subsequently, we inspect these files to find the predictability of values. Predictability can be of two types: last value reuse and stride based. When the last value is reused we can use the LVPT based prediction scheme that uses the last value as the current prediction, whereas for a minority of cases we observe a stride based pattern. Here, we can use a regular stride based predictor as described in Section 5.1.2.

## 5.2 Replay Mechanisms

As we saw in Section 5.1, there are many ways of speculating. The common feature of all of these methods is to speculate on the basis of predictions. Note that the predictions are never 100% accurate. It is possible to have mispredictions. In this case, we need to dynamically locate all those instructions that have possibly gotten a wrong value, and *squash* (cancel or nullify) those instructions. For example, if we falsely predict the value returned by a load instruction, then all the instructions that have consumed the wrong value have to be squashed. This set of instructions is also known as the *forward slice*. Formally, the forward slice of an instruction  $I$  is defined as all the instructions that are data dependent either directly or indirectly (via other instructions) on the value produced by instruction  $I$ . It also includes instruction  $I$ .

Consider the following dependences ( $\rightarrow$  indicates a RAW dependence):  $I_0 \rightarrow I_1$ ,  $I_1 \rightarrow I_2$  and  $I_1 \rightarrow I_3$ . In this case the forward slice of  $I_0$  contains  $I_0$ ,  $I_1$ ,  $I_2$ , and  $I_3$ . If the result produced by  $I_0$  is wrong, then its entire forward slice ( $I_0$ ,  $I_1$ ,  $I_2$ ,  $I_3$ ) also needs to be squashed.

### Definition 24

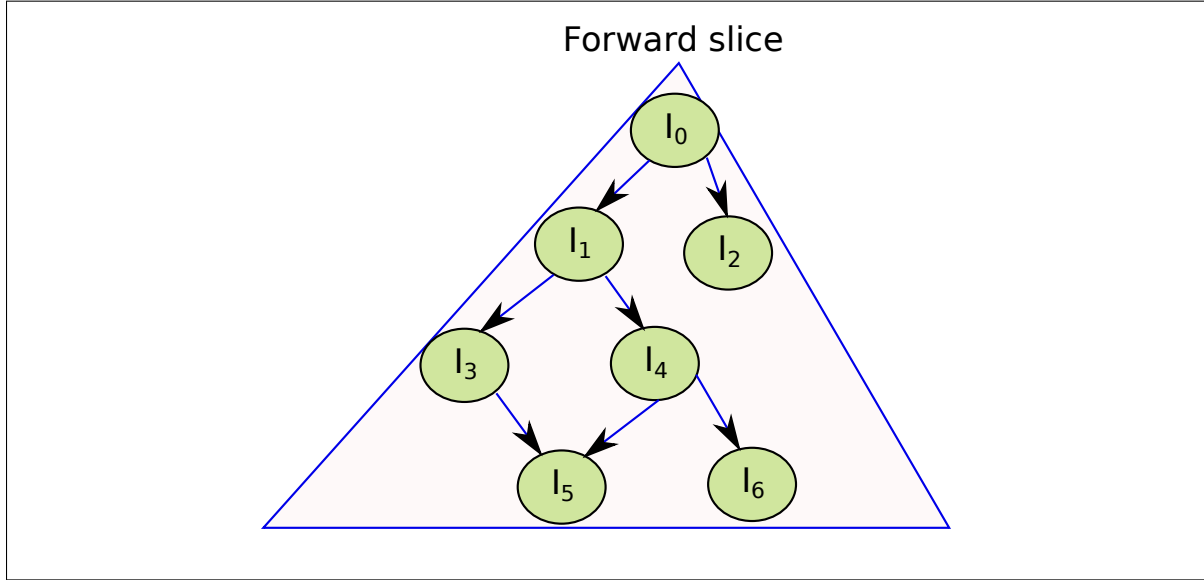
*The act of dynamically nullifying or canceling an instruction in the pipeline is known as instruction squashing or just squashing.*

What about control instructions? It is possible that the value returned by a load instruction might have influenced the direction of a branch. In this case, the branch will execute on the wrong path. However, in a modern OOO pipeline this will not happen. This is because we predict the outcome of branches at fetch time. We verify whether they were correctly predicted or not at commit time. This means that by this time all the instructions before the branch need to have fully executed, and written their results to the architectural state. All speculative loads before the branch would have completed, and their results would have been verified. We decide the outcome of a branch at commit time only on the basis of non-speculative data (data that is fully verified). Thus, there is no chance of a branch getting mispredicted because of load speculation.

Hence, we restrict our discussion to forward slices that are only created via data dependences.

### Definition 25

*The forward slice of instruction  $I_0$  is defined as the set of instructions that are data-dependent on the value produced by instruction  $I_0$  either directly or indirectly. An indirect dependence is established by a chain of direct data dependences between a pair of instructions. Refer to the following figure.*



The main aim of the *replay* mechanism is to ensure that instructions in the *forward slice* of a mis-speculated instruction are squashed and then re-executed.

### 5.2.1 Pipeline Flushing

The simplest method to fix the state of the pipeline is to flush the pipeline. Whenever we mispredict the outcome of an instruction, we mark it in the ROB. This marking is similar to marking mispredicted branches. Whenever the instruction with the mark reaches the head of the ROB, we do not allow the instruction to commit. We instead flush the pipeline, and restart execution from the instruction whose outcome was incorrectly predicted. Akin to the case of branch misprediction, we can fix the state of the pipeline and ensure that no misspeculated data gets written to the architectural state.

This method will no doubt work. It is a proven method, and does work wonderfully in the case of branch mispredictions. However, if we expect frequent mispredictions, then the overheads of this method are large. Note that flushing a pipeline is a very expensive operation. There might be 50 instructions fetched after a misspeculated instruction is fetched. All of these 50 instructions will be discarded if we flush the pipeline. This means that we will lose a lot of work that has been done. As a result, flushing the pipeline is not necessarily a good solution, even though it is very simple.

Typically, forward slices of instructions are fairly small. Flushing the pipeline for cleaning up the effects of misspeculation is metaphorically like killing a mosquito with a canon ball. We have to look for methods that can do the same with a significantly lower overhead. Let us look at a set of schemes originally proposed by Kim and Lipasti [Kim and Lipasti, 2004] in the rest of this section. Please note that we shall describe simplified adaptations.

### 5.2.2 Non-Selective Replay

This is a more efficient scheme as compared to a full pipeline flush. However, it is still not what we exactly want in the sense that it does not locate the exact forward slice and squash it. However, it is an important step in that direction. It chooses a superset of the forward slice. If we invalidate this set, then we are guaranteed to also have invalidated the forward slice. Then, we can reissue these instructions. Since we expect misspeculation events to be rare, we can afford a replay scheme with higher overheads.

Let us define the *window of vulnerability* (WV). Assume that a speculative load instruction broadcasts its tag in cycle 1. The nature of speculation can be diverse: speculating on the latency of the load or

on its value. Now, assume that we shall get to know in the  $N^{th}$  cycle if the speculation is correct or not. Thus, it is possible that any instruction that marks any of its operands as ready between cycles 1 to  $N$  can be affected by the speculated load. Since we do an early broadcast, the consumers of the load instruction might get woken up, and they might subsequently wake up their consumer instructions and so on. If in the  $N^{th}$  cycle, we realize that the speculation is wrong, the entire forward slice has to be squashed. **Since we do not explicitly keep track of the forward slice, we need to squash any instruction that has marked an operand as ready in the WV (cycles: 1 to  $N$ ).** Note that a squashed instruction may either have already been issued, or may still be in the instruction window waiting to wake up or get selected.

The hardware support that is required is as follows. Along with every operand, which is not present in the register file, we associate a counter that is initialized to 0. This counter is set to  $N$  when we receive a broadcast for the tag associated with the operand. The counter thus starts the moment we see its tag on the tag bus, and then decrements itself by 1 every cycle till it reaches 0. Now, if the counter becomes 0, and we do not receive *bad news* (notification of a misspeculation), we can successfully conclude that the operand was read correctly. This scheme works on the principle that if there is a problem, instructions in the WV (window of vulnerability) will be informed, otherwise, all is well.

However, if there is a problem (misspeculation), then all the instructions in the WV (some operand has a non-zero counter) need to be squashed, and re-executed (replayed).

Let us explain with an example. Consider the following piece of assembly code.

```

1 ld r1, [r2]
2 add r4, r1, r3
3 add r5, r6, r7
4 add r8, r9, r10

```

Assume that instruction 1, which is a load instruction, is sent speculatively to the memory system. We predict that the value that it reads to be 7. Let's say we wait for 3 cycles and at the end of 3 cycles, we find out that the prediction was wrong. Let us further assume that instructions 2, 3, and 4 have been issued during that time frame because one of their operands was marked as ready. We then need to squash these three instructions: 2, 3, and 4. Subsequently, we need to reissue instructions 1-4. This is a non-selective replay mechanism because we decided to replay all the instructions in the window of vulnerability. Only instruction 2 is dependent on the result of the load. However, we are not selective, in the sense that we do not track dependences between instructions.

Let us look at the pros and cons of this scheme. The biggest advantage is that the scheme is simple. We do not have to track dependences between instructions. However, on the other hand, it can also be inefficient, particularly if  $N$  (size of the window of vulnerability) is high. In this case, we unnecessarily have to squash many instructions, even though the size of the forward slice may be really small. Of course, to choose a given replay mechanism we need to factor in many more things like the accuracy of the predictors, the number of instructions that are actually replayed, and the size of the replay hardware (as a fraction of the size of the rest of the hardware). Let us elaborate.

## Dynamically Squashing Instructions

Whenever, we mark an operand as ready (because of a broadcast), we need to set a timer for that operand to  $N$ . The timer decrements every cycle (refer to Figure 5.5). Once the timer becomes 0, we are sure that this operand was read correctly. It is better to assume by default that things are all well, instead of the other way, because most of the time we expect the speculation to be correct.

Let us see what happens when we detect a misspeculation. Whenever we have such an event, we assert the kill wire (refer to Figure 5.5). Observe that the kill wire is connected to every entry of the instruction window. Here is the key idea: squash all the entries whose count for any operand is non-zero. These instructions are in the WV. For all such operands with a non-zero counter, we set their ready

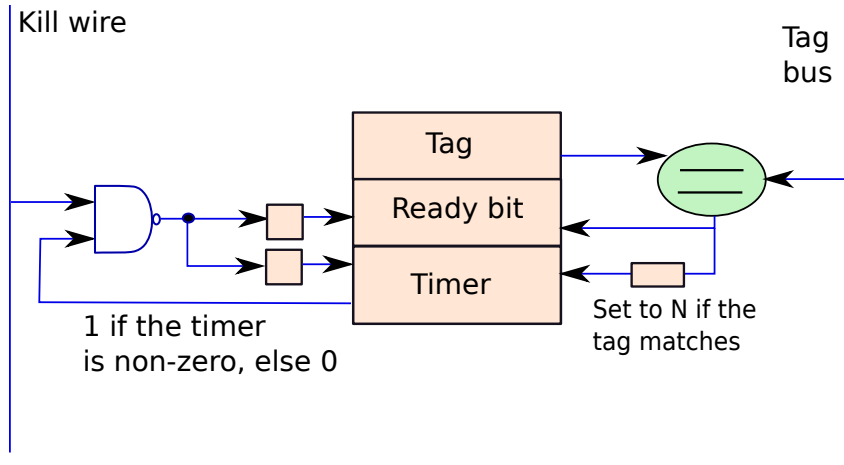


Figure 5.5: Structure of an instruction window entry with non-selective replay (only one tag bus is shown for the sake of simplicity)

bits to 0, and their counts to 0 as well. These instructions may have already been issued. In this case, they need to be replayed – re-executed with the correct values. If they have not been issued, then they need to wait till the tag is broadcasted again (corresponding to the correct value). Now, note that in Figure 5.5 we use a NAND gate that has two inputs: the kill wire and a bit that indicates if the timer’s count is zero or non-zero. If both are 1 then the output of the NAND gate is a 0. This resets the ready bit and the timer. However, if the output of the NAND gate is 1, then no action is taken because in this case either the kill wire is deasserted or the timer’s count is zero.

Let us look at the methods of re-execution, or in other words methods to replay the instructions.

### 5.2.3 Methods to Replay Instructions

Now, that we have dynamically squashed the instructions that might possibly have gotten a wrong value, it is time to re-execute or replay them. There are two ways of doing this.

#### Approach 1: Keep in the Instruction Window

The first approach is to keep instructions that have issued in the instruction window. We do not remove them after they are issued. Instead, we wait till the instructions get verified (all the predictions are correct). This means that the pipeline looks something like the one shown in Figure 5.6. Subsequently, when the instructions are *verified*, they are removed from the instruction window. Before discussing what happens when instructions need to be replayed, let us delve into the details of the verification process.

An instruction is said to be verified, when it is issued, and the counters of all of its operands reach 0. This means that even if we assert the kill wire in the future, the current instruction will stay unaffected. At this point it can be removed from the instruction window.

However, if the instruction needs to be squashed, then we need to reset its state as described in Section 5.2.2. This becomes a fresh instruction, which needs to be woken up and issued once again. Note that the second time it will not get squashed because of the same speculative instruction. This is because for the same instruction we do not speculate twice. For example, if we predict the value of a load instruction, and then the prediction turns out to be incorrect, we do not predict once again. Instead, we wait for the right value to come from the memory system. Thus, it will never happen that the speculate-replay process will continue indefinitely with no progress.

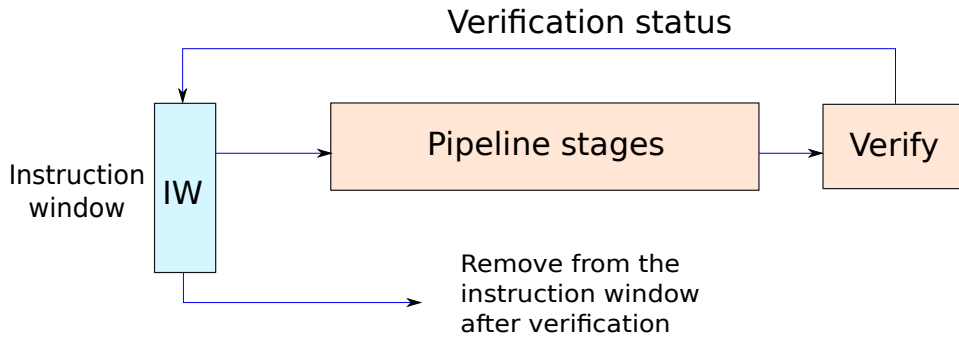


Figure 5.6: Instruction replay strategy: verify and remove

Once the right value comes from the memory system the misspeculated load instruction can re-execute, collect the right value from the memory system, and thus execute correctly. It can simultaneously broadcast the tag corresponding to its destination register to its consumer instructions that have also been squashed. They can wake up if the rest of their operands are available. In the next cycle, these consumer interactions can further wake up their consumers and so on. If the rest of the speculation is correct, this time the forward slice will execute correctly. Note that multiple misspeculations can happen concurrently, and thus a single instruction might get replayed multiple times if it is in the forward slice of multiple misspeculated instructions. However, the same instruction will suffer a misspeculation only once. We do not predict twice.

### Approach 2: Create a Separate Replay Queue

If we keep all instructions in the instruction window till they are verified, it will create an otherwise complex instruction window even more complex. Furthermore, if there is very little speculation, then this mechanism simply adds to the overheads and is counterproductive. If the instruction window needs 100 entries, and because of the replay process, if we need to extend its size to let's say 150, it will unnecessarily become very slow.

Let us thus keep the size of the instruction window the same. We instead add a separate queue called the *replay queue* as shown in Figure 5.7.

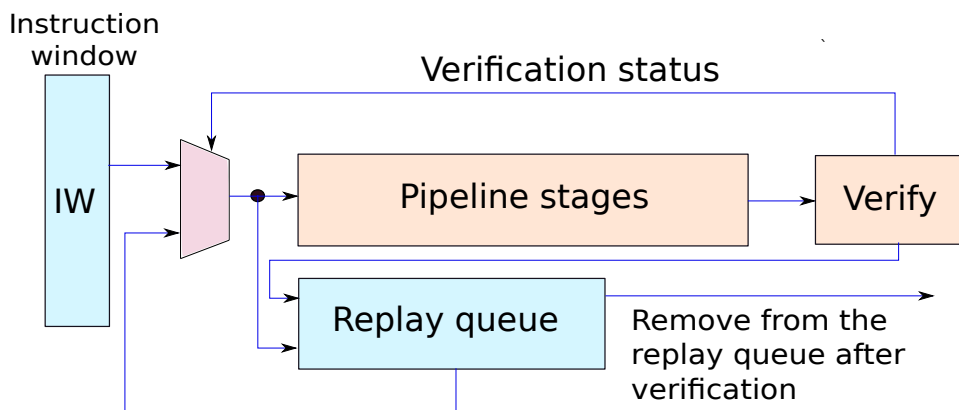


Figure 5.7: Instruction replay using the replay queue

After issuing an instruction, we move it to the replay queue. Thus, we do not need to increase the



size of the instruction window unnecessarily. An instruction remains in the replay queue till it is verified. Once it is verified it can be removed from the replay queue. The rest of the logic remains the same.

### Orphan Instructions

Let us summarize the situation that we have described till now. After we detect a misspeculation, we squash all the instructions that might have received a wrong value. We have looked at a simple method of creating this set of instructions using the non-selective replay approach. There are more sophisticated methods of creating this set, also known as the *squashed set*. Nevertheless, let us describe the basic principles that govern the correctness of the replay techniques.

If the squashed set is simply the forward slice of an instruction, then this situation is very easy to handle. We simply restart the instruction that was misspeculated with the correct value, and the entire forward slice shall get the right values through the broadcast-wakeup mechanism (over multiple cycles). Since we already guarantee that all instructions in the forward slice are squashed, we shall never miss an instruction, and the execution will be correct.

However, a problem arises when we squash a superset of the forward slice. Let us consider the following code fragment.

```

1  ld r1, 8[r2]
2  sub r4, r1, r3
3  mul r5, r6, r7

```

Assume that we try to predict the latency of instruction 1 and speculate. Instruction 2 is dependent on 1, and thus it is in its forward slice. However, instruction 3 is independent and not a part of its forward slice. Now assume that we have a misprediction while speculating on the latency of instruction 1, and after asserting the kill signal, we squash all three instructions: 1, 2, and 3. In this case, we need to replay all three instructions. Replaying instructions 1 and 2 is easy. Instruction 1 gets replayed because we had mispredicted its latency. Once the load value is available we can broadcast the tag corresponding to register *r1*. This will wake up instruction 2, and this time it will get the correct value of *r1*. However, there is nobody to wakeup instruction 3!

Instruction 3 was unfortunately squashed because it was in the window of vulnerability (WV). Its only crime was that one or more of its operands became ready within the WV of instruction 1. The producers of its operands *r6* and *r7* have long retired. We thus have a deadlock. No instruction is going to broadcast the tags corresponding to the physical registers mapped to *r6* and *r7*, and thus instruction 3 will remain in the instruction window forever. Let us call such instructions as *orphan* instructions.

We can consider re-broadcasting the tags for instruction 3. However, to do that we need to keep track of all the operands of all the instructions that have been squashed. We then need to keep track of the tags that have already been broadcast, and the ones that have not been broadcast yet. This is complicated, and requires fairly elaborate hardware. Here is a simple solution. Wait till instruction 3 reaches the head of the ROB. At that point of time all of its operands, should have gotten their correct values. This is because there will be no instruction in the pipeline that is earlier than instruction 3. All such instructions would have executed correctly, written their values to the register file, and left the pipeline. Thus, at this point if instruction 3 is still waiting for some broadcasts, we can force it to execute with the values that are currently there in the register file. The result will be correct, because the values of the operands are correct. Let us extend this idea a little further.

#### 5.2.4 Delayed Selective Replay

There are several disadvantages of non-selective replay. The forward slice can be a small portion of the squashed set. We might end up doing a lot of wasted work. Secondly, as we saw in Section 5.2.3, we can end up with orphan instructions.

Let us try to extend this scheme to solve some of these problems. Let us keep non-selective replay as the baseline scheme and make some enhancements.

### Poison Bit

The first concept that we need to introduce is the *poison bit*. The aim is to keep track of the forward slice very accurately. We augment every register file entry, the instruction packet and the bypass network with an additional bit called the poison bit. Now, as an example, let us assume that we mispredict the value of a load instruction that writes to the physical register  $p1$ . It is possible that due to the early broadcast mechanism, other instructions in the pipeline will nevertheless read  $p1$  because they have been eagerly issued. Let us thus attach a poison bit to the value stored in  $p1$  and set it to 1. This means that regardless of how we get the value of  $p1$  – via the register file or the bypass network – we always read the associated poison bit to be 1. All consuming instructions including those that have been issued because of the early broadcast mechanism will read this poison bit. If an instruction reads a source operand with its poison bit set, then it also sets the poison bit of its destination register.

This is how the poison bit propagates through the forward slice, and thus we can dynamically mark an instruction's forward slice. Keep in mind that the poison bit propagates when we read a value either from the register file or the bypass network. It is not propagated while broadcasting tags or waking up instructions.

Now, when we misspeculate an instruction, we need to do two things:

1. Set the poison bit of the instruction packet and the destination register (physical register) to 1.
2. Set the kill wire and invalidate all the instructions in the window of vulnerability (non-selective replay scheme).

### Basic Protocol

When an instruction finishes its execution, we do the following:

1. We check if the poison bit of the instruction is set. If it is, we squash the instruction by not allowing it to proceed further to the ROB. We however, set the poison bit in the physical register file for the destination register. Additionally, we also attach a poison bit along with the corresponding value on the bypass network.
2. If the poison bit is not set, then this means that this instruction has not received a speculative value. However, it is possible that this instruction might have been squashed because it is in the WV of a misspeculated instruction.
3. Let us make the instruction proceed towards the commit stage, and write its result to the register file. We also send its value on the bypass network to consumer instructions. Let us now proceed to handle such kind of corner cases.

### Understanding of the Corner Cases

Let us first outline the problems associated with a window of vulnerability based replay scheme: it is false dependences. A *false dependence* arises when an instruction in the WV is squashed even though it is independent of the misspeculated instruction. Consider two instructions  $I$  and  $J$ .  $I$  is misspeculated and  $J$  is in the WV of  $I$ , even though it is not a part of  $I$ 's forward slice. Assume that  $J$  is a safe instruction, which is not in the forward slice of any other misspeculated instruction. In this case there is a false dependence between  $I$  and  $J$ .

In the non-selective replay scheme such false dependences led to orphan instructions. In our current scheme – delayed selective replay – we have introduced a basic protocol that propagates the poison bit in the forward slice of the misspeculated instruction. Let us see what it achieves.

Assume that instruction  $J$  has been issued; it will pass through the execution units. Its poison bit will be 0. Since the instruction is safe, we need to let the replay queue know that the entry for  $J$  can be removed (similar logic for replay with the instruction window). This can be achieved by broadcasting the tag to all the elements in the replay queue. The entry for instruction  $J$  can mark itself to be free.

Now consider the tricky corner case when  $J$  has not been issued, and we have ended up invalidating one of its ready operands. It is thus an *orphan* now. Similar to the case in Section 5.2.3 (for non-selective replay) there is no instruction to wake it up.

### Dealing with Orphan Instructions

The fact that instruction  $J$  is an orphan basically means that when the kill wire for instruction  $I$  was asserted, one of  $J$ 's operands was ready and its associated timer was non-zero. This operand's ready bit got unset. Now, there is nobody to set the operand's ready bit back to 1. Let us understand this process in some more detail. The fact that  $J$ 's operand was ready means that some other instruction  $K$  must have set it to ready.

Let us now augment our design by adding a completion bus. It is similar to a tag bus that allows an instruction to broadcast its tag to all the entries in the instruction window. Now, assume that every instruction broadcasts its tag (id of its destination register) on the completion bus if it executes successfully without getting its poison bit set or getting misspeculated. This is done exactly  $N$  cycles after it has broadcasted its destination tag: there are two broadcasts in this scheme – the first is a regular broadcast for waking up consumers and the second broadcast is on the completion bus after the instruction's status is known. Assume that all the results of speculation are available within this time frame – within  $N$  cycles of broadcasting the tag (first broadcast). Finally, assume that the value of the timer associated with each source operand is set to  $N$  when the operand is woken up.

Consider the following timeline. Assume that in the  $N^{th}$  cycle, one of  $J$ 's operands was ready, its counter was non-zero, and in this cycle instruction  $I$  asserted the kill wire. Since  $J$  was not issued yet, it got orphaned. This means that  $I$  must have broadcasted the tag any time  $t$  where  $t \geq 1$  because it asserted the kill wire when  $t = N$ . Additionally, given that instruction  $K$  woke up the same operand, it must have also broadcasted its tag any time after  $t \geq 1$  because at  $t = N$  the timer of the operand was still non-zero.

Let instruction  $K$  signal its completion at any time  $t \geq N + 1$ . At this point of time, instruction  $J$  can read the tag off the completion bus and set the ready bit of the operand once again (see Figure 5.8).

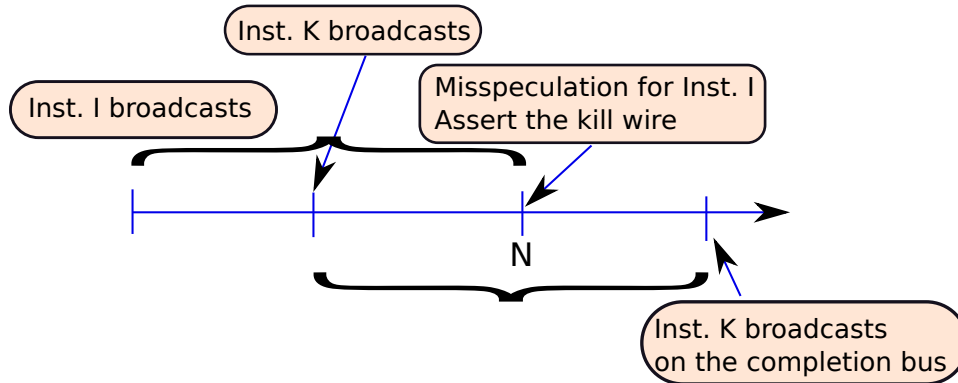


Figure 5.8: An example scenario that produces an orphan instruction

The summary of this discussion is that if an instruction has been orphaned because of a false dependence, then the instruction that had originally woken up the operand is going to again come back

in the future to rescue it. In this case instruction  $K$  rescues instruction  $J$ . The completion bus is the additional overhead of this scheme.

### 5.2.5 Token based Replay

Let us now create a system that truly buffers the forward slice and no other instruction. As we have been observing, there is a trade-off between tracking dependences and the replay complexity. The more we track dependences, the easier it is to perform replays. If we are able to precisely mark the forward slice, performing a replay is easy – there are no false dependences. In token based replay the main idea is that for a load that is predicted to miss, we mark it, and propagate the mark to instructions in the forward slice (similar to a poison bit). With these marks it is very easy to identify the instructions that are there in the forward slice of a load.

Before proceeding, let us note a common pattern found in most programs. Let us consider the latency of loads in the data cache. Most of the time in such cases, the 90/10 thumb rule is found to operate. This is a thumb rule that basically says that 90% of the misses are accounted for by 10% of the instructions and 10% of the misses are accounted for by 90% of the instructions. A word of caution is due. This is not a hard and fast rule, it is a thumb rule. This means that we typically observe similar patterns in real programs. It has to do with the way that we write programs. Since most of the code executes within loops, we have a fair amount of temporal and spatial locality. As a result, we typically do not have a lot of misses in such phases. Moreover, for regular accesses such as walking through an array, we can very easily prefetch data, and this further reduces misses. Most of the misses happen when we access irregular data structures such as linked lists and execute portions of code that are rarely accessed.

Now, given this rule, let us try to leverage it. Given a load PC, let us try to predict if it will lead to a miss. We can use the hit-miss predictor described in Section 5.1.4. We can get a good accuracy, if our code uses data structures such as arrays. Let us create two sets of load instructions:  $S_1$  and  $S_2$ . Instructions in  $S_1$  are predicted to most likely miss in the L1 data cache, and instructions in  $S_2$  are predicted to most likely hit in the L1 data cache.

At the time of decoding the instruction, we run the hit-miss predictor, and if an instruction is predicted to miss (part of  $S_1$ ), then we proceed as follows.

#### Tokens and Token Vectors

Let us add a vector of tokens to the instruction packet of each instruction, each rename table entry, and to each source operand in an instruction window entry. This is a vector of  $k$  bits, where the bit at the  $i^{th}$  position indicates the presence of token  $i$ . If the bit is 1, then it means that the given instruction or entry *holds* the token, and if the bit is 0, then it means that the token is not held. We will be using the token as a proxy for the forward slice. We have a total of  $k$  tokens.

**Token Generation:** When we predict an instruction to be in set  $S_1$  in the decode stage, we collect a free token from a token allocator. Assume we get token  $i$ . Then we set the  $i^{th}$  bit in the token vector of the instruction packet to 1. This instruction is said to be the *token head* for token  $i$ . We then proceed to the rename stage.

We add two additional fields to an entry in the rename table: *tokenId* and a token vector *tokenVec*. Let us explain with an example. Assume an instruction: *ld r1, 8[r4]*. In this case the destination register is  $r1$ . Assume it is mapped to the physical register  $p1$ . In the rename table entry of  $p1$ , we save the id of the token that the instruction owns in the field *tokenId*.

The logic for setting the field *tokenVec* is more elaborate.

#### Token Propagation:

##### Conceptual Idea

Now that we have a way to generate tokens, we need to design a method to propagate tokens along the

forward slice of an instruction. We can easily deduce that a producer needs to propagate its tokens to its consumers, and the consumers in turn need to propagate the tokens that they hold to their consumers. In this manner tokens need to propagate along the forward slice of an instruction. Note that we are using the word “tokens” in its plural form. This is because an instruction can be a part of the forward slices of numerous load instructions. It will thus hold multiple tokens – one each for each load in  $S_1$ .

Let us consider our example instruction again. It was *ld r1, 8[r4]*. In this case register *r1* was mapped to the physical register *p1*. We generated a token for this instruction and added it to its instruction packet as well as the *tokenId* field of its destination register *p1* in the rename table.

Now consider the *tokenVec* field. It is supposed to contain a list of all the forward slices that the instruction is a part of. We use a token as a proxy for a forward slice and thus with each instruction and its destination register in the rename table we maintain a vector of tokens – *tokenVec*.

### Implementation

Let us assume that the token vectors held by the source operands are  $T_1 \dots T_n$ . Let  $t'$  refer to the token generated by the instruction. If the instruction does not generate a token then  $t' = \phi$ . Then the token vector for the instruction and its destination register in the rename table is given by Equation 5.1. Let us refer to the final vector of tokens as  $T_f$ .

$$T_f = T_1 \cup \dots \cup T_n \cup t' \quad (5.1)$$

The process is shown graphically in Figure 5.9. We are essentially merging all the information – computing a union of all the token vectors. This is because now the current instruction is in the forward slice of many instructions – one forward slice per token. This computation can be done in the rename stage and then the computed token vector  $T_f$  can be used to set the token vector of the instruction and the *tokenVec* field of the rename table entry of the destination register. In this case  $T_1 \dots T_n$  correspond to the token vectors of each source register in the rename table.

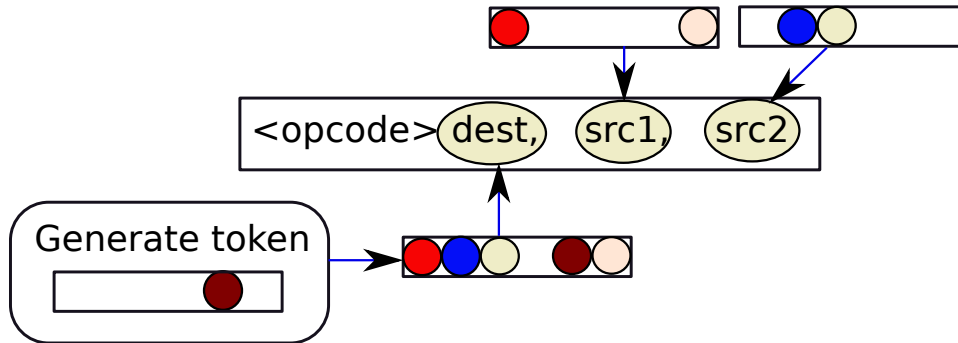


Figure 5.9: Token propagation (a colored circle represents a token)

Subsequently, the instruction enters the instruction window. Let us keep two token vectors in each instruction window entry – one for each source operand. We can read these token vectors in the rename stage and populate the corresponding fields in the instruction window entry in the dispatch stage. If we have a replay queue then its entries will also be augmented with this information.

### Verifying and Squashing Instructions

Let us continue our discussion. Note that we are only discussing instructions in set  $S_1$  (predicted to suffer from a misspeculation).

Depending upon the type of speculation, we will have different methods of verifying the speculation. For example, if we are speculating on the latency of the load, then once a load completes, we can check

if it took extra (more than its predicted value) cycles or not. If we are trying to predict a load-store dependence, we can always check whether this dependence exists or not, once the addresses of the corresponding loads and stores have been resolved. We can thus conclude that at a future point of time, we can expect a Boolean answer from the verification logic: True (speculation is correct) or False (speculation is false).

**Speculation is Correct:** In this case, we need to broadcast the token id that the load *owns* to all the entries in the rename table, the instruction window, and replay queue (if it exists). We need to set the bit corresponding to the token id in the token vectors to 0. This basically means that the respective token is being freed and removed from the system.

Broadcasting a token id to the entries in the rename table and instruction window requires some changes to the hardware. We need to create a new bus called the *token bus* that is connected to each entry. Furthermore, it is very well possible that multiple tokens might need to be released in each cycle. The simplest solution is to augment each entry with an AND gate. In each cycle we compute a logical AND operation between the *tokenVec* of each entry and the value that is broadcast on the token bus. Let us assume that the token bus is as wide as the number of tokens in circulation. It transmits a mask that we shall refer to as *tokenMask*, and a single bit that indicates if we are freeing a token, or initiating a squash (referred to as the *squashBit*). Assume that we can have 8 tokens (numbered 1 to 8) in circulation and tokens 1, 2, and 4 are getting released. In this case, we will set the mask as 00001011 (counting from the right starting from 1). The logic is that if the  $i^{th}$  token is being released we set the  $i^{th}$  bit to 1, otherwise we set it to 0. After an AND operation between the bitwise complement of the token mask with the token vector *tokenVec*, the  $i^{th}$  bit in *tokenVec* will become 0. The respective token will thus get released.

We thus compute:

$$tokenVec = tokenVec \wedge \overline{tokenMask} \quad (5.2)$$

In this case the *squashBit* = 0 – the speculation is correct.

**Speculation is Incorrect:** In this case, we need to initiate a replay. The load that has had a misspeculation needs to be a token head (because it is a part of set  $S_1$ ). Let it be the owner of token with id  $j$ . We need to broadcast  $j$  to all the entries in the instruction window and replay queue. We can use the same token bus mechanism with the squash bit set to 1. We can also support replays due to multiple misspeculations. Let us explain with an example.

Assume that in a system with 8 tokens, we have misspeculations for tokens 3 and 5. With *squashBit* = 1, we transmit the following *tokenMask*: 00010100. To find out if a given operand needs to be invalidated or not, we need to find if any of the tokens associated with the operand correspond to misspeculated instructions or not. This is possible by computing the result of the following equation.

$$result = tokenVec \wedge tokenMask \quad (5.3)$$

In this case, if any bit of *result* is equal to 1, then it means that the given operand is a part of the forward slice of a squashed instruction. We thus need to squash that instruction. In this case, all that needs to be done is that we need to reset the ready bits and reissue these instructions when the operand becomes available again (similar to non-speculative replay).

**Avoiding Orphan Instructions:** It is easy to avoid orphan instructions in this scheme. We only invalidate those operands and instructions that are part of a misspeculated forward slice. We do not have false dependences in this scheme.

### Instructions in Set $S_2$

Let us now consider instructions in set  $S_2$ . Instructions in this set are not expected to suffer from a misspeculation. Note that these instructions do not generate any tokens.

If they do not suffer from a misspeculation, then there is no problem. We continue as is. However, if they have a misspeculation, then we use a sledge hammer like approach as we had proposed in Section 5.2.1. We simply wait till the instruction reaches the head of the ROB, and then we flush the pipeline. This solution has a high overhead, yet is simple.

## 5.3 Simpler OOO Processor without a Register File

Let us now simplify things. We have invested a lot of silicon real estate in creating a high performance processor. However, since this chapter is about alternative designs, let us look at OOO processors that are less complex, and not necessarily very inefficient. Let us look at one of the most popular alternative designs that uses the ROB also as the physical register file (PRF)<sup>1</sup>. It leads to a simpler implementation. We need not have the paraphernalia associated with a physical register file. Even recovering from a branch misprediction is also much easier.

Let us first try to motivate this design by first taking a look at some aspects of the pipeline presented in Chapter 2, which are amenable to simplification. Here were the big sources of complexity.

**Physical Register File** To support large instruction window sizes, and remove all hazards, we needed large physical register files. To keep track of physical register file allocation it was necessary to have additional structures such as a free list. In addition, the logic for freeing a physical register is non-trivial. We need to wait for another instruction writing to the same architectural register to commit.

**Maintaining Precise State** Maintaining the correct architectural state (see Section 4.4) was difficult. We had to introduce many schemes to remember the mapping between physical registers and architectural registers at various points in the program. This was a slow and complicated mechanism. We would love to at least make this part simpler.

**Recovery from Branch Mispredictions** This point extends the previous point, where we discussed the complexity of maintaining architectural state. Recovery from a branch misprediction is also fairly involved in the scheme with physical register files. We need to restore a checkpoint, which can be a set of saved register values, or a set of mappings between architectural registers and physical registers. This process requires time and additional hardware.

Let us try to propose a new scheme that avoids the physical register file altogether and makes it easy to recover from branch mispredictions.

### 5.3.1 Overview of the Design

#### Main Insights

Let us ask the basic question: Why did we use a physical register file in the first place? We used it to avoid WAR and WAW hazards. This is not the only way of avoiding such hazards. Here is an alternative idea.

Instead of using the temporary storage provided by the physical register file, let us instead use an ROB entry to additionally play the role of a physical register. We can thus achieve both our aims with the same structure. It is true that this will increase the number of ROB accesses and make it a bottleneck. Let us entertain such considerations later. Let us first present the design.

<sup>1</sup>The terms *physical register file* and *PRF* will be used interchangeably

## Design of the Pipeline

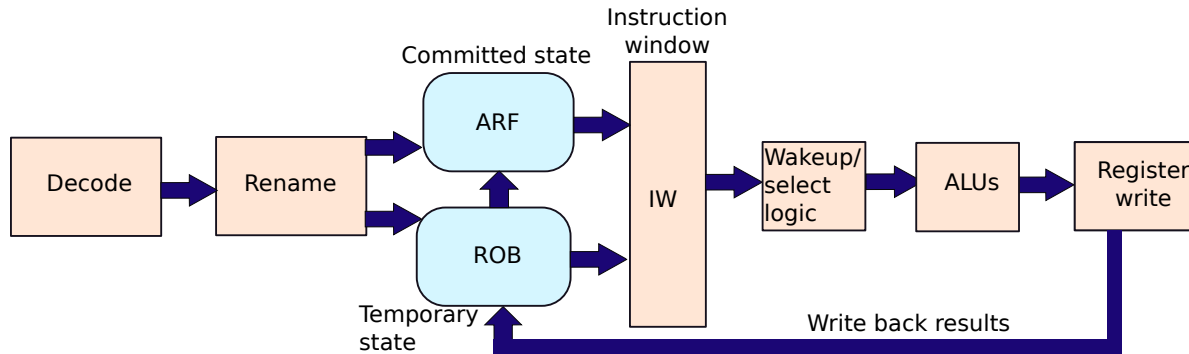


Figure 5.10: Overview of a simplified pipeline

Figure 5.10 provides an overview of our simplified pipeline. Instead of having a large physical register file, we have a smaller architectural register file (ARF). The number of entries in the ARF is the same as the number of architectural registers. Furthermore, the ARF also contains the precise architectural (committed) state. There is thus no need to create periodic checkpoints and restore them if there is a need. The ARF contains the committed state. The crucial assumption here is that committed values stay in the ARF and temporary (not committed) values reside in the ROB. Henceforth, we shall refer to the pipeline introduced in Chapter 4 as the PRF based pipeline, and the pipeline introduced in this section as the ARF based pipeline.

The crux of the idea is to change the renaming stage. Given an architectural register, the renaming stage needs to point out a location at which the value is available. Since there is no physical register file, the renaming stage in this case points to either the architectural register file (ARF) or the ROB (Reorder buffer). If the value that we want to read is a part of the committed state, then we need to read it from the ARF. However, if the instruction that has written the value to an architectural register has not committed yet, then we need to read its value from the ROB. The rename table points to the right location.

It is to be noted that once an instruction retires, we need to update the committed state. We write to the destination register in the ARF upon instruction retirement. The rest of the pipeline is roughly the same. The key difference is that in the PRF based design we read the register values after issuing the instruction. In this case, we read the operand values either from the ARF or ROB before the instruction enters the instruction window (IW).

### 5.3.2 Detailed Design

#### Rename Stage

Given that we have seen the overview of the design in Figure 5.10, let us look at the rename stage in some more detail. The structure of each entry in the rename table is as follows.

As before, the number of rows in the rename table is equal to the number of architectural registers. Let us assume that the architectural register that we want to rename is  $r1$ . The value of this architectural register can either be present in the ROB or ARF if it has already been computed. We thus add a bit in each entry of the rename table called *inARF*, which indicates if the value for the register  $r1$  is in the ARF (*inARF* = 1) or in the ROB (*inARF* = 0). If the value is present in the ARF, then we can directly read the architectural register in the next cycle.

However, if the value is not present in the ARF, then we need to read the subsequent field *robEntry*. This is the number of the ROB entry that contains the value of the register. It contains the current value



of the architectural register  $r1$ . Note that we are augmenting every ROB entry to contain additional data – result produced by the instruction.

### Operand Read Stage

The major difference in this pipeline is that we read the values first and carry the values along with us in the pipeline. This is quite unlike the PRF based design, where we read the values from the register file or the bypass network just before execution. Here we try to read the values right after renaming the instruction and carry it through the processes of dispatch and issue. This is no doubt a source of inefficiency because a 7-bit PRF tag (assuming a 128-entry physical register file) is far easier to carry along as compared to a fat 64-bit value. However, if this is a 16-bit processor, where every value is 16 bits wide, then this idea does not look that bad.

Now, if the rename stage indicates that the value can be found in the ARF, we read the ARF, otherwise, we access the ROB.

Let us consider the case when the rename table indicates that the value of the operand (register  $r1$ ) is present in the ROB. Each entry of the ROB contains the following fields: *avlbl* and *val*. *avlbl* is a 1-bit Boolean field that indicates if the instruction corresponding to the ROB entry has produced its result or not. The field *val* contains the result of the instruction after the completion of its execution.

There are two cases that we need to consider. If *avlbl* = 1, then it means that the ROB entry pointed to by the field *robEntry* in the rename table contains the value of  $r1$ ; however if *avlbl* = 0 then this means that the value is not ready yet. Instead, we need to wait in the instruction window for the value of  $r1$  to be generated. Recall that we had relied on a broadcast-wakeup based mechanism in the PRF based pipeline. In that case we were broadcasting the PRF register id. This was also referred to as the *tag*. In this case, the tag is the index of the ROB entry that produces the value for  $r1$ , which is *robEntry*. We have to thus wait for the id of the ROB entry along with the value of the operand to be broadcasted. Subsequently, we can proceed for execution, if we have the values of the rest of the operands.

### Dispatch Stage

In this stage, the instruction enters the instruction window. The only difference in this case is that an instruction enters the instruction window along with the values of operands. The structure of an entry in the instruction window is shown in Table 5.1. We assume that we have 128 entries in the ROB.

Field	Description	Width (in bits)
valid	validity of the entry	1
ready	instruction is ready to be executed	1
<b>First source operand</b>		
ready1	value is present	1
tag1	tag of the first source register	7
val1	value of the first operand	64
<b>Second source operand</b>		
ready2	value is present	1
tag2	tag of the second source register	7
val2	value of the second operand	64
<b>Destination</b>		
isregd	destination is a register	1
robTag	ROB entry of instruction	7

Table 5.1: List of fields in an instruction window entry

We have two additional fields in an instruction window entry: *val1* and *val2*. These are the values of the two source operands. If each value is 64 bits, then we are adding 128 bits to each instruction window entry. This is expensive; however, carrying the values along simplifies things to a great deal. It reduces dependences. Instead of waiting to read other structures such as the register file, we can directly execute the instruction if all the values have been read earlier, or can be obtained through the bypass network.

The bypass network also needs to change. The tag in this case is the index of the ROB entry that produces the value of the operand. The rest of the tag matching logic and wakeup logic remain the same. There is one more significant change as well. Along with broadcasting the tag, we also need to broadcast the value of the operand. Previously, this was not required because we could always read the value from the register file. This kept the bypass network relatively lean, and did not add a lot of wires and buffers. However, now we do not get a chance to read the register file after the instruction enters the instruction window. Hence, we need to broadcast the value also.

After an instruction picks up all of its operand values, it wakes up, is selected, and proceeds for execution.

### Speculative Broadcast

Recall that in Section 4.2.4, we were broadcasting the tag in advance to ensure that instructions did not have to unnecessarily wait longer. They could pick up the values of operands on the way (from the bypass network). This enabled back-to-back execution.

Here also we can do the same. We can issue instructions *early* as long as they are guaranteed to get their operand values later in the pipeline. This aspect of the pipeline does not change.

### Write-back Stage

After an instruction has executed, it is time to write its value back. We need to write this value to its ROB entry. The ROB entry buffers this value till the instruction is committed. We need to set *avlbl* = 1 in the ROB entry. This indicates that the value has been computed and can be read. Recall that by default we set *inARF* to be 0 while creating an entry in the rename table; this means that by default we access the ROB unless instructed otherwise.

### Commit Stage

This is the last stage. In this stage, we need to update the architectural state. For each instruction that is being committed, we write its result (stored in its ROB entry) to the ARF. This thus updates the architectural state.

Simultaneously, we need to update the *inARF* bit in the rename table. However, this is easier said than done. Let us explain with an example. Assume a committing instruction *I* of the form *add r1, r2, r3*. In this case, the destination register is *r1*, which contains the result of the instruction. Let us first assume that after this instruction was renamed, no other instruction has passed through the rename stage with *r1* as the destination register. In this case once instruction *I* commits, the value of *r1* needs to be transferred from the ROB to the ARF, and in addition the rename table needs to be updated. We need to set the *inARF* bit to 1 because now the value can be found in the ARF.

Let us now consider the other situation. In this situation, just after instruction *I* gets renamed, another instruction *I'* passes through the pipeline. It also writes to *r1*. In this case, the rename table entry for *r1* gets updated – it points to the ROB entry for *I'*. When instruction *I* commits, it cannot set the *inARF* bit to 1 for *r1* in the rename table. This is because the latest value will be produced by instruction *I'* or a later instruction.

Let us thus summarize the logic. We need to compare the *robEntry* field of the rename table's entry for *r1* with the id of the ROB entry for instruction *I*. If they are equal, then it means that no instruction that also writes to *r1* has been renamed after *I*. We can thus set *inARF* in the rename table entry to 1. However, if they are not equal, then we can infer that there is another instruction in the pipeline that

overwrites *r1*, and thus *inARF* needs to remain 0. To summarize, an additional comparator is required here. It needs to compare the *robEntry* field in the rename table entry with the id of the ROB entry of the committing instruction.

#### Historical Note 1

*It is not necessary to have one unified instruction window. Indeed, one of the earliest proposals for designing an OOO processor by Robert Tomasulo in 1967 envisioned multiple instruction windows. The rest of the design was conceptually similar to the ARF based OOO processor. He proposed mini-instruction windows attached to each functional unit. Such instruction windows were known as reservation stations. Each entry in a reservation station was similar to an entry in the instruction window in our ARF based processor. All the reservation stations were connected to a common data bus (CDB) where the tags and values were broadcasted. The reservation stations compared the tags and upon a match buffered the value.*

*Modern designs often have different instruction windows for different classes of instructions. For example, if the integer and floating point register set is completely disjoint, then separate instruction windows can be used for each class of functional units. We can also have such a separation between regular integer/floating point instructions, and vector based SIMD instructions (instructions that perform arithmetic instructions on multiple operands at a time). Having such a separation is desirable because small instruction windows are faster and more power efficient. However, having a unified window is also sometimes advantageous because it helps balance out the unevenness of the load between different classes of functional units.*

### 5.3.3 Comparison

Let us now compare the PRF and ARF based designs by listing out their pros and cons (refer to Table 5.2). Note that *IW* refers to the instruction window in the table.

Attribute	PRF based design	ARF based design
Values reside in only a single location	Only in the PRF	Multiple locations: ARF, IW, ROB
Size of an entry in the IW	small	large (contains operand values)
Restoring state after a misprediction	hard	very easy
Need for a free list	yes	no
Write ports in the ROB	decode_width	decode_width + issue_width
Read ports in the ROB	commit_width	commit_width + 2 * issue_width

Table 5.2: Comparison between the ARF and PRF based designs

The main drawbacks of the ARF based design are the size and complexity of the ROB. We write to the ROB twice – while creating an entry in the decode stage and while writing the results of an instruction. In the worst case, the number of ports required for writing the results can be equal to the issue width.

We can always optimize this by populating the ROB entry lazily. We can write most of the contents to the ROB entry at the time of writing the results. The ROB entry itself can be split across two

sub-arrays of memory cells (will be described in more detail in Chapter 7). Since the ROB also acts as a register file, in the worst case, we would need to read  $\langle 2 \times \text{issue\_width} \rangle$  number of operands. In addition, we need to read the ROB while committing instructions. The number of simultaneous reads (for committing instructions) is equal to the commit width.

Note that even in the PRF based design we needed to update the ROB after the successful execution of each instruction. We needed to update a bit in each ROB entry to indicate that the instruction has completed successfully. To optimize, we can have a separate structure to store these bits. Moreover, since the size of each entry is a single bit, we can even have an array of flip-flops instead of an SRAM array. Hence, we have not added this source of overheads to Table 5.2.

Given that the ROB can be a bottleneck, and we need comparators with each rename table entry, this design will have significant performance overheads. It is simple, yet not expected to be as efficient as the PRF based designs particularly for large 64-bit server processors.

## 5.4 Compiler Based Techniques

We have looked at a lot of hardware starting from the basic in-order processor to an aggressive out-of-order processor with value prediction and token based replay. Adding such sophisticated features is no doubt a good thing. They ensure significant gains in performance. However, there is a flip side to everything. Adding extra hardware always has negative implications in terms of power consumption and complexity.

We did take a sharp turn towards simplicity in Section 5.3. We dismantled the physical register file, added more bits to the ROB, and simplified the processor. However, such processors have fairly large instruction window entries, a complex multi-ported ROB, and they move fat 64-bit values around. These are not particularly performance enhancing optimizations.

These are examples, where simplicity can give you some advantages, yet take some other advantages away. Let us now discuss a set of compiler based approaches that need not necessarily have a negative effect on performance.

Let us look at the compiler – something that we have been ignoring up till now. The role of a compiler is not just limited to converting a high level program to machine code, it is actually much more than that. Modern compilers spend most of their time in optimizing code. This does increase the compilation time; however, at the same time it makes the code run efficiently on modern processors. The IPC increases, and in a lot of cases, the number of instructions also decreases. The only trade-off is the size, complexity, and time of execution of the compiler. These are not significant issues as of today.

Given the amount of physical memory we have, we don't mind if a compiler takes a couple of megabytes of more memory to do a good job in compilation. The same maxim is true for complexity and execution time. We need to run the compiler only once to produce a binary; however, we run the binary many times. Hence, the compilation speed per se does not matter. For example, at the moment, your author is typing this book using the *gvim* editor. The editor is really fast and allows the authors to achieve many complex tasks very quickly. It does not matter if it took 2 minutes to compile the source code of the editor or 2 hours.

We are definitely in a position to afford a very complex compiler that might possibly take a very long time to compile a given program. However, at the end, the compiler must do a very good job in creating a binary that has a lot of ILP, and in reducing the number and complexity of instructions. Let us look at some of the basic techniques that compilers use to produce efficient code.

### 5.4.1 Data Flow Optimizations

The main idea here is to optimize the process of data propagation between instructions, and see that we are able to reduce the number of instructions as well as their complexity. Recall that different instructions have different latencies, or in other words they take different amounts of time to execute. As a result, we

are better off having simpler instructions such as add and subtract instead of more complex instructions such as multiply and divide.

### Constant Folding

This is one of the simplest optimizations in our arsenal, yet is extremely effective. Consider the following piece of C code.

```
int a = 4 + 6;  
int b = a * 2;  
int c = b * b;
```

A naive compiler will first add 4 and 6, then store the result in *a*, and then execute the rest of the statements in order. Is this required? The answer is absolutely not. A smart compiler can figure out at compile time that the value of *a* is a constant, and this constant is equal to 10. Similarly, it can also figure out that the values of *b* and *c* are also constants, and we can directly compute their values and update the registers that correspond to them. This saves us a lot of computation, and also decreases the number of instructions, which directly leads to an increase in performance.

### Strength Reduction

Now that we have folded away our constants, let us look at operators. We need to understand that different arithmetic operations have different latencies. In particular multiplication and division are slow, with division being the slowest. It is best to replace such instructions with faster variants wherever possible. The faster instructions are add, subtract, and shift (left/right) instructions. Recall that shifting a value to the left by  $k$  places, is equivalent to multiplying it by  $2^k$ . Similarly, shifting a value to the right by  $k$  places, is equivalent to dividing it by  $2^k$ .

Let us now consider an example.

```
int b = a * 8;  
int d = c / 4;  
int e = b * 12;
```

We have two multiplication operations and one division operation here. These are expensive operations in terms of both power and time. Hence, it is highly advisable to replace these operations with simpler variants, if we have an option. In this case we do, because we can leverage the fact that the numbers 8, 4, and 12 are either powers of 2, or can be expressed as a sum of powers of 2. We can thus use shift operations here. Let us rewrite the code snippet to produce a more optimized variant.

```
int b = a << 3;  
int d = c >> 2;  
int e = b << 2 + b << 3;
```

In this case, we have used far simpler shift operations, which can often be implemented in a single cycle and are far more power efficient.

What we see here is that we were able to replace multiply and divide operators with equivalent shift left and shift right operators. This *strength reduction* operation will lead to performance gains because of the lower latency of the shift instructions.

## Common Subexpression Elimination

Consider the following snippet of C code.

```
int c = (a + b) * 10;
int d = (a + b) * (a + b);
```

An unoptimized compiler will generate code to perform all the additions and multiplications. However, this is not required. We can alternately transform this code to a more optimized version.

```
int t1 = a + b;
int c = t1 * 10;
int d = t1 * t1;
```

Instead of 3 additions and 2 multiplications, we are now doing 1 addition and 2 multiplications. We have definitely saved on 2 additions by a simple trick. Instead of computing the *common subexpression*  $a + b$  again and again, we have computed it just once and saved it in a local variable  $t1$ , which can be mapped to a register. Subsequently, we use  $t1$  to act as a substitute for  $a + b$  in all subsequent instructions.

Compilers use this technique to reduce the number of instructions wherever possible. They try to compute the values of subexpressions before hand, and then they save it in registers. These values are then used over and over again in subsequent instructions. Since we decrease the number of instructions, we have a definite performance gain.

## Dead Code Elimination

Consider the following program.

```
int main (){
    int a=0, b=1, c;
    int vals[4];

    printf ("Hello World\n");
    c = a + b;
    vals[1] = c;
}
```

Is there a need to perform the last addition,  $c = a + b$  and then set  $vals[1]$  to  $c$ ? There is no statement that is using the value of  $c$  and the array  $vals$ . These values are not influencing the output of the program, which is what most users care about. Unless, we explicitly want to run these instructions to measure the performance of the program with these instructions, in an overwhelming majority of the cases, we do not need these instructions.

We can thus label such instructions as *dead code*. This is code that does not have any purpose, and does not influence the output. Most compilers these days are fairly good at identifying and removing dead code. Other than the obvious advantage of reducing the number of instructions, another major advantage is that we can efficiently pack the useful instructions into instruction cache (i-cache) blocks. There is no wastage of space in the i-cache. Note that we do not want to waste valuable i-cache bandwidth in fetching instruction bytes that are not required.

**Definition 26**

*Lines of code that do not influence the final output are referred to as dead code.*

**Silent Stores**

Let us now increase the degree of sophistication. Consider the following piece of code.

```
int arr[5], a, b, c;

arr[1] = 3;
a = 29;
b = a * arr[0];
arr[1] = 3;          /* Not required */
printf ("%d \n", (arr[1] + b));
```

Consider the second store to `arr[1]`. It is not required. It writes exactly the same value as the first store, even though it is not really dead code. However, the second store is a silent store, because it has no effect. It writes a value to a memory location, which is already present there. In that sense, it does not write a new value. Hence, we can happily get rid of the second store instruction to `arr[1]`. This is called *silent store elimination*.

**Definition 27**

*Assume that a memory location at a given point of time contains the value  $v$ . If at that point of time, a store writes value  $v$  to that memory location, then it is called a silent store.*

Such kind of data flow analyses can become increasingly sophisticated, and we can find a lot of redundancy in the program, which can be successfully eliminated. To understand how exactly these mechanisms work, the reader needs to take an advanced course on compilers.

**5.4.2 Loop Optimizations**

Most of the programs that we write use loops, and also most of the execution of a program is within loops. These are the repetitive structures that most often take up more than 90% of the execution time. Thus, optimizing loops is essential. Even if we make a small change in the code of the loop, the benefits have the potential to multiply.

**Loop Invariant based Code Motion**

A variable or property that does not change across the iterations of a loop is known as a *loop invariant*. Assume that in every iteration we set the same variable to the same value, then that variable is a loop invariant. There is in fact no reason for it to be repeatedly updated to the same value within the body of the loop. The update statement can be moved outside the loop. Let us explain with an example.

```
for (i=0; i<N; i++){  
    val = 5;  
    A[i] = val;  
}
```

There is no reason for the variable *val* to be updated within the loop. This update instruction can very well be moved by a smart compiler to a point before the loop. We will save a lot of dynamic instructions ( $N$  instructions in  $N$  iterations) by making this change. The code after moving the loop invariant to before the loop looks like this:

```
val = 5;  
for (i=0; i<N; i++){  
    A[i] = val;  
}
```

This is a much faster implementation because we reduce the number of dynamic instructions.

**Definition 28**

*A variable or property that does not change across the iterations of a loop is known as a loop invariant.*

**Induction Variable-based Optimization**

Let us now climb up the ladder of complexity. Consider the following *for* loop.

```
for (i=0; i<N; i++){  
    j = 6*i;  
    A[i] = B[j] + C[j];  
}
```

This piece of code uses a loop variable *i* that gets incremented every cycle. However, let us concentrate our attention on the variable *j*. It is a multiple of *i*, and sadly we need to perform a multiplication to set *j* once every iteration. Is it possible to remove this multiplication? It turns out that the answer is yes. Consider the following piece of optimized code.

```
j = 0;  
for (i=0; i<N; i++){  
    A[i] = B[j] + C[j];  
    j = j + 6;  
}
```

The important observation that we need to make is that in every iteration we are incrementing the value of *j* by 6 because *i* is getting incremented by 1. We can thus replace a multiplication by an addition. In every iteration, we increment *j* by 6, which is mathematically the same thing. However, when we translate this to gains in performance, it can be significant, particularly if there is a large difference in the latencies of add and multiply instructions. For example, if a multiplication requires 4 cycles, and



an addition requires 1 cycle, we can execute 4 times as many addition instructions as multiplication instructions in the same time.

Such analyses can be extended to nested loops with multiple induction variables and multiple constraints. We need to understand that most modern compilers are today constrained by the amount of information that is available to them at the time of compilation. This is because we do not know the values (or range of values) of all variables at compile time.

### Loop Fusion

Let us now look at a slightly more complicated optimization. Consider the following piece of code. In the next few examples we shall show assembly code written in the SimpleRisc assembly language (described in Appendix A).

#### C code

```
for (i=0; i<N; i++) /* Loop 1 */
    A[i] = 0;

for (i=0; i<N; i++) /* Loop 2 */
    B[i] = 0;
```

#### Assembly code

```
/* r0 and r1 contain the base addresses of A and B
   i is mapped to r2
   N is contained in r3 */

mov r10, 0          /* store 0 in r10 */
mov r2, 0           /* i = 0 */
.loop1:
    cmp r2, r3      /* compare i with N */
    beq .exit1      /* go to exit1 if i==N */
    lsl r4, r2, 2    /* r4 = i * 4, size of int is 4 bytes */
    add r5, r0, r4   /* address of A[i] */
    st r10, [r5]     /* A[i] = 0 */
    add r2, r2, 1    /* i = i + 1 */
    b .loop1        /* Jump to the beginning of loop1 */

.exit1:
mov r2, 0           /* i = 0 */
.loop2:
    cmp r2, r3      /* compare i with N */
    beq .exit2      /* go to exit2 if i==N */
    lsl r4, r2, 2    /* r4 = i * 4, size of int is 4 bytes */
    add r5, r1, r4   /* address of B[i] */
    st r10, [r5]     /* B[i] = 0 */
    add r2, r2, 1    /* i = i + 1 */
    b .loop2        /* Jump to the beginning of loop2 */

.exit2:
```

This is a very standard piece of code where we initialize arrays. This pattern of writing code is also very common among programmers, particularly novice programmers. However, there are sub-optimal decisions in this code. We have two loops: *loop1* and *loop2*.

Notice that the only difference in the bodies of these loops is that we are updating different arrays. Otherwise, the code is identical. Instead of executing so many extra instructions, we can fuse loops 1

and 2, and create a larger loop. This will ensure that we execute as few extra instructions as possible. Most of the code to update the loop variable  $i$  and to check for loop termination can be shared. Let us thus try to rewrite the code.

#### C code

```
for (i=0; i<N; i++){ /* Loop 1 */
    A[i] = 0;
    B[i] = 0;
}
```

#### Assembly code

```
/* r0 and r1 contain the base addresses of A and B
   i is mapped to r2
   N is contained in r3 */

mov r10, 0
mov r2, 0 /* i = 0 */
.loop1:
    cmp r2, r3 /* compare i with N */
    beq .exit1 /* go to exit1 if i==N */
    lsl r4, r2, 2 /* r4 = i * 4, size of int is 4 bytes */
    add r5, r0, r4 /* address of A[i] */
    st r10, [r5] /* A[i] = 0 */

    add r5, r1, r4 /* address of B[i] */
    st r10, [r5] /* B[i] = 0 */

    add r2, r2, 1 /* i = i + 1 */
    b .loop1 /* Jump to the beginning of loop1 */

.exit1:
```

Let us see what we have achieved. In our original code, the body of loops 1 and 2 had 7 instructions each. Since each loop has  $N$  iterations, we shall execute  $7N$  instructions per loop. In total, we shall execute  $14N$  instructions.

However, now we execute far fewer instructions. Our loop body has just 9 instructions. We thus execute only  $9N$  instructions. For large  $N$ , we save 36% of instructions, which can lead to a commensurate gain in performance. Note that we are ignoring instructions that initialize variables and instructions in the last iteration of the loop where we exit the loop (small constants).

## Loop Unrolling

This is by far one of the most popular optimizations in this area. It has a very wide scope of applicability and is supported by almost all modern compilers.

The idea is as follows. Consider a loop with  $N$  iterations. We have multiple branch statements in the body of a loop. There are several ill effects of having these branch instructions. The first is that these are extra instructions in their own right. Executing them in the pipeline requires time. Additionally, they take up slots in the instruction window and ROB. If we can to a certain extent get rid of these additional branch instructions, we will be able to decrease the number of dynamic instructions appreciably.

The second effect is that every branch needs to be predicted and there is a finite chance of a misprediction. The number of mispredictions increases with the number of branches. A misprediction is very expensive in terms of time because we need to flush the pipeline. Hence, eliminating as many branches as possible is a good strategy.

To motivate the discussion, let us consider the following piece of C code.

#### C code

```
for (i=0; i<10; i++){
    sum = sum + i;
}
```

#### Assembly code

```
mov r0, 0      /* sum = 0 */
mov r1, 0      /* i = 0 */

.loop:
cmp r1, 10
beq .exit      /* if (i == 10) exit */
add r0, r0, r1 /* sum = sum + i */
add r1, r1, 1  /* i = i + 1 */
b .loop        /* next iteration */

.exit:
```

In this code, in each loop iteration we have 5 assembly instructions. Three of them are only for maintaining proper control flow within the loop and only the add instruction is for the data flow. It is pretty much the only instruction that is doing any real work.

This seems to be a rather inefficient use of the instructions in a typical loop. Most of the instructions in the body of the loop are just for ensuring that the loop's control flow is correct. However, we are not doing a lot of work inside the loop. Only 1 out of the 5 instructions is doing the useful work. Given that we already know that the number of iterations in the loop (i.e., 10), we should make an effort to do more useful work.

Hence, let us try to reduce the number of branches, and increase the amount of useful work done per iteration. Let us *unroll* the loop. This basically means that we need to fuse multiple iterations of a loop into a single iteration. Let us show the equivalent C code and assembly code.

#### C code

```
for (i=0; i<10; i+=2){
    sum = sum + i + (i+1);
}
```

#### Assembly code

```
mov r0, 0      /* sum = 0 */
mov r1, 0      /* i = 0 */

.loop:
cmp r1, 10
beq .exit      /* if (i == 10) exit */

add r0, r0, r1 /* sum = sum + i */
add r1, r1, 1  /* i = i + 1 */
add r0, r0, r1 /* sum = sum + i */

add r1, r1, 1  /* i = i + 1 */
b .loop        /* next iteration */

.exit:
```

We have basically fused two consecutive iterations into one single iteration. Let us now work out the math in terms of the number of instructions. To keep the maths elegant let us only count the 10 successful iterations, and not the one in which we don't enter the body of the loop because the comparison is successful. Before unrolling we executed 50 instructions (5 instructions in the body of the loop). However, now we execute 35 instructions because we have 5 iterations, and there are 7 instructions in the body of the loop. There is thus a savings of 30%, which is significant by all means.

Can we unroll further? Well, yes. We can fuse 4 or 8 iterations into one. However, this does not mean that we can unroll indefinitely. Otherwise, we will replace the entire loop with one large piece of unrolled code. There definitely are limits to unrolling. If we unroll too much, the code size will become very large, it will not fit in the instruction cache (small instruction memory), and we will simply have too many cache misses. If we have multiple programs resident in memory then we might also run out of memory. However, within limits, unrolling is a very effective technique.

### 5.4.3 Software Pipelining

Let us now come to one of the most difficult optimizations. Note that till now, we have not concerned ourselves with the details of the pipeline; however, this optimization is very important in the context of modern pipelines. It takes into account instruction latencies and also the nature of hazards in a pipeline.

The key insight used in software pipelining is as follows. Loop iterations often execute inefficiently because of slow instructions. For example, if we have a load and a consumer instruction that uses its value, we have the possibility of a load-use hazard (see Section 2.1.4). Particularly, with a slow cache, waiting for a load instruction to return its value is a very expensive proposition. We would love to do useful work in the time being. However, we often might not have enough instructions in the loop iteration to execute between a load and its consumer. The key insight is to bring instructions from other iterations to fill this void. This will ensure that we waste as few cycles as possible. In addition, it is possible to optimize this technique to enable greater ILP, and make it possible to parallelize code with loops on multi-issue processors.

Let us consider the following piece of code.

#### C code

```
int A[300], B[300];
...
for(i=0; i<300; i++){
    A[i] = B[i];
}
```

#### Assembly code

```
1  /* Assume the base address of A is in r0
2     and B is in r1 */
3  mov r2, 0          /* i = 0 */
4  mov r10, 0         /* offset = 0 */
5
6  .loop:
7      cmp r2, 300     /* termination check */
8      beq .exit
9
10     add r3, r1, r10 /* r3 = addr(B) + offset */
11     ld r5, 0[r3]    /* r5 = B[i] */
12
13     add r4, r0, r10 /* r4 = addr(A) + offset */
14     st r5, 0[r4]    /* A[i] = r5 (= B[i]) */
15
```

```

16     add r2, r2, 1    /* i = i + 1 */
17     lsl r10, r2, 2   /* offset = i * 4 */
18
19     b .loop
20
21 .exit:

```

We show the code of a simple loop that does an element wise copy from array  $B$  to array  $A$ . Note that we introduced the *offset* variable stored in  $r10$  for the sake of readability. We could have incremented  $r2$  by 4 in each iteration instead.

Other than the statements that manage the loop and compare the loop variable  $i$  with 300, we can divide the statements into three blocks.

Block	Lines	Statements	Role
$L$ (load)	10 - 11	add r3, r1, r10 ld r5, 0[r3]	Read $B[i]$
$S$ (store)	13 - 14	add r4, r0, r10 st r5, 0[r4]	Write to $A[i]$
$I$ (increment)	16 - 17	add r2, r2, 1 lsl r10, r2, 2	Increment $i$

There are dependences between these blocks:  $L \rightarrow S \xrightarrow{a} I$ . Here “ $\rightarrow$ ” denotes a RAW dependence and  $\xrightarrow{a}$  denotes an anti or WAR dependence. Let us now introduce some new notation here. Let block  $L^k$  denote the load block in iteration  $k$ . Let the superscript represent the iteration number (starting from 0).

For such codes, we have a problem. In an in-order machine if we have a slow L1 cache that takes 3 cycles, we unnecessarily have to stall for one cycle between the load and store instructions. If we have an even slower L1 cache, we need to stall for more cycles. In an OOO machine, things can be slightly better. An OOO machine can automatically resolve WAR dependences. The only real dependence is the chain of increments to the loop variable. Let us make a very important observation here. **If we know the value of the loop variable for each iteration before hand, we can execute the iterations in parallel.** It turns out we can do something in the compiler to expose more instruction level parallelism.

Now, let us write all of these blocks in a linear (column wise fashion), and try to find a pattern. See Figure 5.11.

Let us explain Figure 5.11 very carefully. We first show 5 iterations of the loop. In each iteration, we show the value of the loop iterator,  $i$ , which increases from 0 to 4. In each iteration, we execute the three blocks of statements:  $L$ ,  $S$ , and  $I$ . In the typical scheme of things, we first execute iteration 0, then iteration 1, and so on.

Let us change the order of execution of instructions. Let us execute instructions in the following order:  $L^0 \rightarrow S^0 \rightarrow L^1 \rightarrow I^0 \rightarrow S^1 \rightarrow L^2 \rightarrow I^1 \dots$ . This method of execution is referred to as *software pipelining*. We will very soon explain its connection with real pipelining as we have studied in Chapter 2. Let us understand the way we are proposing to execute these statements. Refer to Figure 5.12, where we show a graphical view of the order of execution.

Let us first convince ourselves that we execute exactly the same set of blocks in Figure 5.12 and Figure 5.11. It is just that the order of evaluation is different. Now, how is this related to pipelining? Let us look at the diagram deeply. In a normal execution we proceed column-wise (iteration by iteration). However, in this case we proceed row-wise. This is the crux of the idea.

Consider iteration 2. The diagram looks as if iteration 2 is going from one stage (row) to the next. Let us see why. First we execute  $L^2$ . Then we move to the next row, which we can consider to be a stage. In this stage we execute  $S^2$ . Then we move to the next row (or stage), where we execute  $I^2$ . Given that an iteration is logically seen to move between rows, we draw an analogy with pipelining, and refer to such coding styles as *software pipelining*. We treat each row in the figure as a pipeline stage.

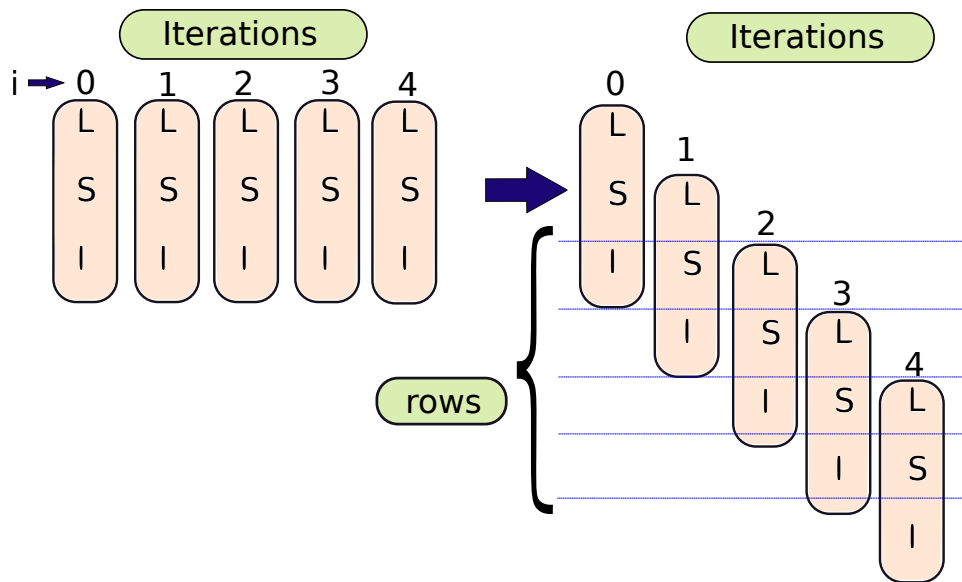


Figure 5.11: Overview of software pipelining

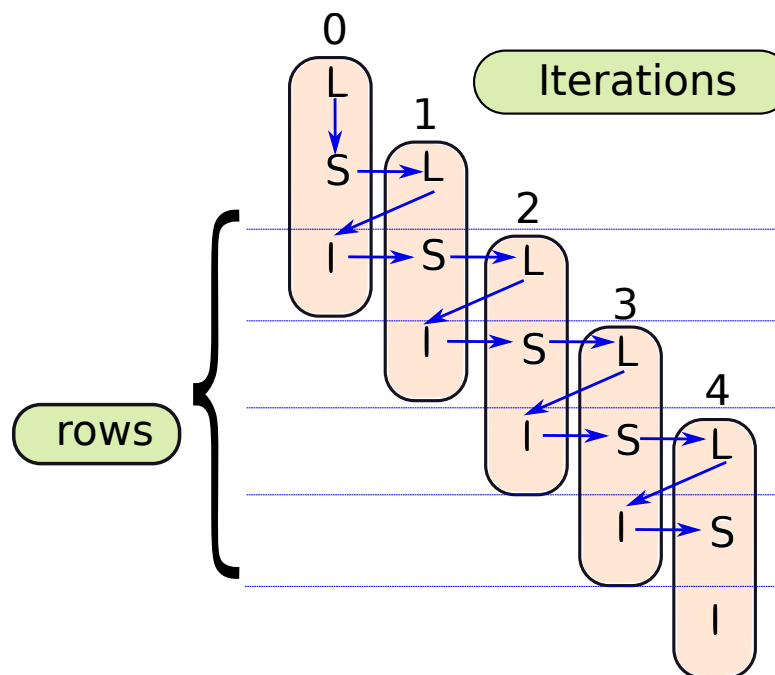


Figure 5.12: Flow of execution

Let us now look at correctness issues. Recall that there is a single loop variable  $i$  and the variable offset (stored in  $r10$ ) that is derived from  $i$ . If we are executing  $I^0$ , then  $S^1$ , and then  $L^2$ , we have a problem.  $I^0$  will increment  $i$  from 0 to 1.  $S^1$  will see the right value of  $i$  and *offset*. However,  $L^2$  needs to see  $i = 2$  and *offset* = 8; it will however see  $i = 1$  and *offset* = 4. This is a *loop-carried dependence*,

where one iteration of a loop is dependent on the values (in this case it is  $i$  and  $offset$ ) computed by another iteration of the loop. Thus, this execution style will be incorrect, unless we do something.

### Definition 29

*A loop-carried dependence is defined as a dependence between two statements within a loop, where the latter statement depends on a value that has been computed by the former statement in a previous iteration of the loop.*

To solve this problem, let us create three loop variables, instead of one (i.e.,  $i$ ). Let us save them in registers  $r6$ ,  $r7$ , and  $r8$ . Let us assign  $r6$  (initialized to 0) to iteration 0,  $r7$  (initialized to 1) to iteration 1, and  $r8$  (initialized to 2) to iteration 2. Let their corresponding offsets be stored in the registers  $r10$ ,  $r11$ , and  $r12$ . Now, there is no problem. There is no dependence between the instructions in the row that contains  $I^0$ ,  $S^1$ , and  $L^2$ . Let us now move to the next row. It contains  $I^1$ ,  $S^2$ , and  $L^3$ .  $I^1$  and  $S^2$  have their versions of the loop variable – in registers  $r7$  and  $r8$  respectively. What about iteration 3? Let us assume that it uses the same loop variable as iteration 0. Since iteration 0 is over,  $L^3$  can use its loop variable, which is stored in register  $r6$ . This means that at this point  $r6$  should contain 3 and the corresponding offset should be 12. We can indeed ensure this by modifying  $I^0$ . Instead of adding 1 to the loop variable stored in  $r6$ , it needs to add 3. In other words, we use three different loop variables stored in registers  $r6$ ,  $r7$ , and  $r8$  and corresponding offsets in registers  $r10$ ,  $r11$ , and  $r12$ . When we increment the loop variable in the last block ( $I$ ), instead of adding 1, we add 3. This value is then used by a subsequent iteration. For example, iteration 3 uses the loop variable of iteration 0, iteration 4 uses the loop variable of iteration 1, and so on. Note that in all cases the corresponding offsets get computed correctly because we just perform a left shift on the loop variable to compute the offset. Figure 5.13 shows this graphically. Note that since the loop variable and offset are intertwined we will not mention both of them all the time. Whenever, we mention loop variables, it should be inferred that we are also referring to the corresponding offset in the same context.

The three loop variables for iterations 0, 1, and 2 are stored in registers  $r6$ ,  $r7$ , and  $r8$  respectively. When the first block of iteration 3 runs, it needs to see the loop variable equal to 3. It uses  $r6$ , which has already been incremented by block  $I^0$  to 3. Likewise, iteration 4, needs to see the loop variable (stored in  $r7$ ) equal to 4. Block  $I^1$  increments  $r7$  (initialized to 1) by 3 to contain 4.

Before writing the code, we need to understand that we have one more constraint. All the iterations of a loop need to have the same code. We thus cannot make each row of Figure 5.13 an iteration of the loop. It does not have the same content. Let us thus unroll the loop and fuse three iterations into one. The content of each fused iteration (three rows) is as follows (also see Figure 5.14):

```
/* First Row */
add r6, r6, 3      /* I0 */
lsl r10, r6, 2

add r4, r0, r11    /* S1 */
st r5, 0[r4]

add r3, r1, r12    /* L2 */
ld r5, 0[r3]

/* Second Row */
add r7, r7, 3      /* I1 */
lsl r11, r7, 2

add r4, r0, r12    /* S2 */
```

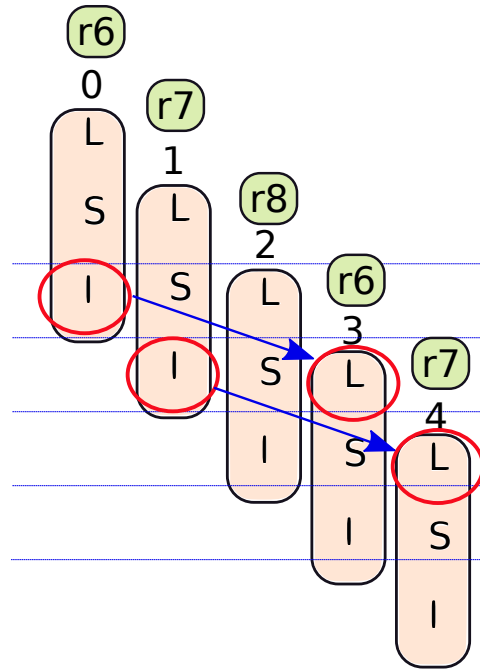


Figure 5.13: Updating the loop variables and corresponding offsets

```

st r5, 0[r4]

add r3, r1, r10 /* L3 */
ld r5, 0[r3]

/* Third Row */
add r8, r8, 3 /* I2 */
lsl r12, r8, 2

add r4, r0, r10 /* S3 */
st r5, 0[r4]

add r3, r1, r11 /* L4 */
ld r5, 0[r3]

```

Please take a look at the code in great detail and try to appreciate the fact that we are simply executing one row after the other. There is a dependence between instructions in the same iteration (same column); however, there is no dependence across instructions in the same row because we use three separate loop variables in registers  $r6$ ,  $r7$ , and  $r8$  respectively. Furthermore, in the same iteration  $r5$  contains the value that is loaded, and subsequently stored in a different array. Between  $L^k$  and  $S^k$ ,  $r5$  is not overwritten by instructions from another iteration.

Let us discuss correctness by focusing on a row that has three entries. The ideal sequence of execution is  $L^k \rightarrow S^k \rightarrow I^k$ . However, now we execute  $L^k \rightarrow I^{k-1} \rightarrow S^k \rightarrow L^{k+1} \rightarrow I^k$ . We are basically executing extra instructions from other iterations in between two blocks of instructions in an iteration. This does not cause an issue because there are no dependences between  $L^k$  and  $I^{k-1}$ , or  $I^{k-1}$  and  $S^k$ . Similarly, the sequence  $S^k \rightarrow L^{k+1} \rightarrow I^k$  does not violate any dependences primarily because we use different loop variables for different iterations. Since no dependences are violated, there is no difference between an



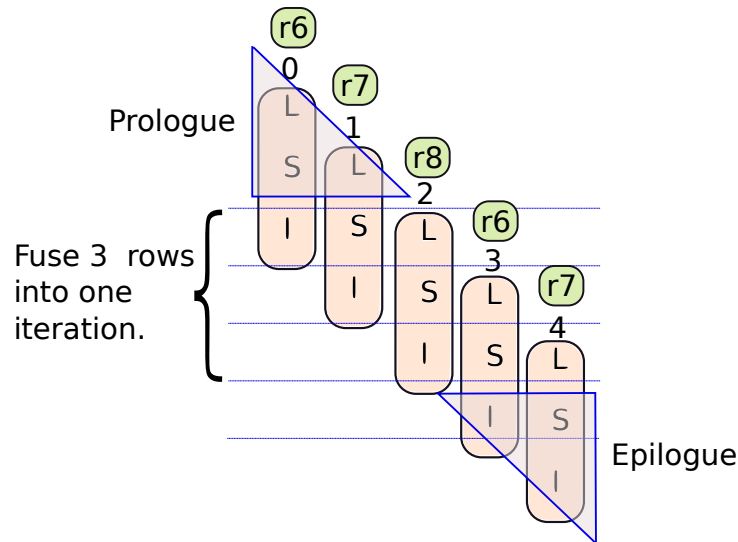


Figure 5.14: Fusing three iterations into one

execution without software pipelining and an execution with software pipelining. They are identical as far as correctness is concerned.

Let us now consider some corner cases. If a loop has a lot of iterations, we can unroll it by a factor of 3, and execute it in this fashion. However, there are some instructions that are not a part of this code. Look at the top of Figure 5.14. Instructions  $L^0$ ,  $S^0$ , and  $L^1$  are not a part of the fused loop. They need to be executed before the main loop starts. This piece of code is known as the prologue. Similarly, instructions  $I^3$ ,  $S^4$ , and  $I^4$  are a part of the epilogue that needs to execute separately, and in the correct sequence. In terms of correctness nothing changes; however in terms of overhead this is a minor overhead because we do not get the benefits of software pipelining for these codes. Nevertheless, when the number of iterations is large, this overhead is negligible. There is a rich theory of software pipelining to cater to the general case where we can have all kinds of dependences between instructions. The reader should refer to the work of Bob Rau [Rau, 1994] and Monica Lam [Lam, 1988].

We can do slightly better if we have more registers. There is a dependence between the  $L$  and  $S$  blocks even between different iterations because they use the same register  $r5$ . This precludes us from executing  $S^1$  and  $L^2$  in parallel. If we have a multi-issue in-order pipeline then we would want to execute  $S^1$  and  $L^2$  simultaneously. Let us try to do some renaming in software to take care of such issues. Similar to hardware renaming, the natural solution in this case will be to create three versions of  $r5$  (one for each iteration).

Let us give ourselves some more room by considering a system that has 32 registers instead of the 16 that we have. For the  $0^{th}$  iteration let us use  $r20$ , and for the  $1^{st}$  and  $2^{nd}$  iterations let us use  $r21$  and  $r22$  respectively. The code thus looks as follows.

```
/* First Row */
add r6, r6, 3 /* I0 */
lsl r10, r6, 2

add r4, r0, r11 /* S1 */
st r21, 0[r4]
```

```

add r3, r1, r12 /* L2 */
ld  r22, 0[r3]

/* Second Row */
add r7, r7, 3    /* I1 */
lsl r11, r7, 2

add r4, r0, r12 /* S2 */
st  r22, 0[r4]

add r3, r1, r10 /* L3 */
ld  r20, 0[r3]

/* Third Row */
add r8, r8, 3    /* I2 */
lsl r12, r8, 2

add r4, r0, r10 /* S3 */
st  r20, 0[r4]

add r3, r1, r11 /* L4 */
ld  r21, 0[r3]

```

Now, there are no dependences between the *S*, *I*, and *L* blocks. They can be executed in parallel. However, we cannot arbitrarily keep on doing this for larger loops because we will run out of registers. There is thus a trade-off between the number of registers and the degree of software pipelining. As we increase the number of registers, there are fewer dependences and higher is the ILP.

### Advantages of Software Pipelining:

Let us list out the advantages:

1. Between a load and its use there are 3 instructions, as opposed to 1 earlier. Thus, we can tolerate a slower L1 cache, and we do not have to introduce any pipeline bubbles in an in-order pipeline. This can be further increased by increasing the level of software pipelining (create rows of 6 instructions for example). Note that this is one of the biggest advantages of software pipelining in in-order machines. Instead of compulsorily introducing stalls in the case of load-use hazards, we can insert instructions from other loop iterations. This crucial insight allows us to get rid of the penalty associated with load-use hazards almost entirely.
2. By using more registers we can make the three blocks, *I*, *S*, and *L*, independent of each other. They can be executed in parallel on a machine that allows us to issue more than one instruction per cycle.
3. Assume that the original loop had  $N$  iterations. There was a chain of  $N$  RAW dependences between consecutive updates to the loop variable ( $0 \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow (N-1)$ ). In the case of software pipelining, this chain of dependences (critical path) got compressed to roughly a third:  $0 \rightarrow 3 \rightarrow 6 \dots \dots [N/3]$ . Shorter dependence chains imply higher ILP, because it means that we can issue more instructions in parallel.

### Software Pipelining Without Loop Unrolling

Note that loop unrolling is not a necessary feature of software pipelining. Even though most of the time both the optimizations are used together, we can have software pipelining that does not use loop unrolling. The main reason that we used loop unrolling is because when we brought together instructions

from three different iterations, we needed three separate loop variables. We kept them in three separate registers.

Instead of keeping different variables for each iteration, we can instead **exploit the relationship between them**. Let us explain with the same example. To make things simple, let us modify the original C code such that each line corresponds to a block of 1-2 assembly instructions in a typical RISC ISA. In each block we never have two memory accesses.

Original C code

```
int A[300], B[300];
for(i=0; i<300; i++){
    A[i] = B[i];
}
```

Simpler C code

```
int A[300], B[300];
int i = 0;
.loop: if (i<300){
    t = B[i]; /* L */
    A[i] = t; /* S */
    i++;     /* I */
    goto .loop;
}
```

In a typical software pipelined execution the ordering is  $I^0 \rightarrow S^1 \rightarrow L^2 \rightarrow I^1 \rightarrow S^2 \rightarrow L^3$ . Let us try to change the code such that we overlap instructions of different iterations, but we do not unroll the loop. Let us just write the body of the loop.

Body of the loop

```
i++;          /* I */
A[i] = t;     /* S */
t = B[i+1];   /* L */

goto .loop;
```

Let us analyze the sequence of operations when  $i = 10$ :  $\boxed{i=11} \rightarrow \boxed{A[11] = t} \rightarrow \boxed{t = B[12]} \rightarrow \boxed{i = 12} \rightarrow \boxed{A[12] = t} \rightarrow \boxed{t = B[13]} \rightarrow \dots$ . Consider the operation  $A[12] = B[12]$ . In between them the only operation is  $i = i + 1$ , which sets  $i$  to 12. However, this does not induce any problems in correctness because the  $I$  block does not modify  $t$ . Similarly, for the loop variable,  $i$ , we increment it by 1 every cycle, and thus this is also correct. The only disadvantage is that we need an additional increment operation to compute the address  $B[i + 1]$ . We need to add 1 to  $i$ , and then add that to the register that contains the base address of  $B$ .

This code snippet is short and tricky. As we have more instructions, forming such loops can get very complicated. There is essentially a trade-off between keeping loop variables in registers (earlier approach), and using an arithmetic relationship between them to compute the array indices (this approach).

## Alternative Method of Doing Software Pipelining

Consider the following example that uses an alternative method. We have deliberately written the C code in a way such that each line corresponds to roughly 1-2 lines of assembly code. There is only one memory operation per line.

C code

```
int A[300], B[300];
for(i=0; i<300; i++){
    t1 = B[i];
    t2 = t1 * 5;
    A[i] = t2;
}
```

SW pipelined version

```
int A[300], B[300];
for(i=0; i<300; i+= 3){
    t1 = B[i];
    t2 = B[i+1];
    t3 = B[i+2];

    t11 = t1 * 5;
    t12 = t2 * 5;
    t13 = t3 * 5;

    A[i]   = t11;
    A[i+1] = t12;
    A[i+2] = t13;
}
```

In this piece of code, we unroll the loop by a factor of 3, and then mix the instructions from the three iterations.

## 5.5 EPIC Processors

Can we do renaming at the level of the compiler and then schedule instructions in software akin to an OOO processor? If the compiler does renaming, it needs to be aware of the details of the hardware at a level that is lower than the instruction set. This further means that a compiled program (a binary) will not be able to run on a different machine (with a slightly different internal organization). Indeed, there are processors of this type. They are known as EPIC machines (Explicitly Parallel Instruction Computing). A classic example of an EPIC processor was the Intel® Itanium® processor [Sharangpani and Arora, 2000]. It required sophisticated compilers to compile regular C code to Itanium compatible binaries. The compilers were aware of the details of the architecture.

Without going into the specific details of the Itanium architecture, let us highlight some advantages and disadvantages of such architectures. The advantage is based on the maxim – keep the hardware simple and move the complexity to software. This means that we don’t need to have circuitry for taking care of stalls and our entire scheduling mechanism – dispatch, broadcast, wakeup, and select. Simpler hardware implies faster and more power efficient hardware.

### 5.5.1 Pros and Cons of EPIC and VLIW Processors

Even though this appears to be a worthy goal, there are problems. The first is that the compiler is not always aware of the dependences in a piece of code. Typical code is littered with *if* statements, *for* loops, and function calls. As a result, predicting all the dependences in advance and creating optimized code is difficult. Let us elaborate. Most EPIC processors follow the VLIW (Very Long Instruction Word) philosophy, where multiple instructions are packed together in a single *bundle* and the bundle is stored as a sequence of contiguous memory words<sup>2</sup>. The advantage of this approach is that we can fetch a bundle of instructions in one go and execute all the constituent instructions in parallel. Conceptually, this is similar to multi-issue in-order pipelines that we studied in Section 2.2.2. For now let us consider EPIC processors to be an advanced avatar of VLIW processors. We will explain the exact differences later.

Instructions in a bundle are fetched, decoded, and executed together (in parallel). However, creating such a bundle is difficult if there are branch instructions and memory instructions. Assume that we create a bundle of four instructions and the third instruction is a branch. Now at runtime if the branch is taken,

<sup>2</sup>A memory word is the minimum number of bytes that we can read or write in one go; it is typically 4 or 8 bytes.

then some instructions in the bundle will become invalid. It is thus necessary to have a mechanism to mark instructions as invalid, and either kill them or let them pass through the pipeline in the invalid state. In the latter case, when they are allowed to pass through the pipeline, such instructions will not be able to write to memory, the register file, or forward (bypass) values. This is called *predicated execution*. It simplifies the pipeline. We can just let instructions in the wrong path flow through the pipeline along with the correct instructions. The invalid instructions will not be processed by the functional units.

Now, let us consider the case of having multiple memory instructions in a bundle. For most memory instructions, we do not have an idea about their addresses at compile time. The addresses are computed at run time. There is always a possibility of a memory dependence (reads/writes to the same address) between different instructions in a bundle, or between instructions across bundles. We need elaborate hardware to take care of memory dependences, forward values between instructions if necessary, and also break a bundle if it is not possible to execute instructions together. Let us consider the following two-instruction bundle.

```
1  st r1, 8[r2]
2  ld r3, 8[r4]
```

In this case if instructions 1 and 2 have the same address then we cannot execute the load and store together. The store has to happen first, and the load later. The only issue is that it is not possible to figure out such dependences at compile time because we don't know the values that *r2* and *r4* will take. However, at runtime such issues will emerge, and dealing with each and every corner case requires extra hardware and extra power.

Along with performance and correctness issues, there are issues regarding the portability of the processor. By exposing details – beyond the instruction set – to the compiler, we are unnecessarily constraining the usability of compiled code. Code compiled for one processor might not work on another processor of the same family. Even if additional measures are taken to ensure mutual compatibility, there might be performance issues. As a result, the industry has by and large not adopted this solution. They have instead tried to do renaming and scheduling in hardware. Of course, this increases the complexity of the hardware, introduces concomitant power issues, and makes it hard to design a processor. Nevertheless, at least as of 2020 for general purpose programs, the benefits outweigh the costs.

Even though EPIC and VLIW processors are not used in modern laptops, desktops, and servers, they are still useful in certain situations. For example, such processors are still very common in the embedded domain particularly in digital signal processors [Eyre and Bier, 2000] and multimedia processors [Rathnam and Slavenburg, 1996]. In such cases, the code is fairly predictable, and thus it is possible to come up with good designs. Furthermore, reducing the power consumption of hardware is an important goal, and thus the EPIC and VLIW paradigms naturally fit in.

Before proceeding further, let us describe the difference between the terms VLIW and EPIC. They are often confused. Note that till now we have pretty much only mentioned that EPIC processors are modern avatars of VLIW processors: they have many features in common such as packing multiple independent instructions in a long memory word.

### 5.5.2 Difference between VLIW and EPIC Processors

The idea of moving all the complexity to software has been a very captivating idea for a long time. Way back in the mid-eighties two important startups in this space came up: Cydrome and Multiflow. The main aim of these startups was to create a VLIW processor with very sophisticated compilers. There are some natural reasons for this thought to have come up. By the mid-eighties, software had already attained a certain level of sophistication, the field of compilers had matured, and modern languages such as C and Pascal had arrived. However, hardware was slow. Intel's 386 processor had a  $1\mu\text{m}$  feature size and could at best run at roughly 30 MHz. Coupled with a very low amount of memory, the

hardware of those days was orders of magnitude slower than today's smartphones. Hence, increasing their performance using sophisticated compilers seemed to be a very worthy idea.

As we saw with software pipelining (Section 5.4.3), it is indeed possible to improve the available ILP significantly using compiler based techniques. Hence, early programmers logically extended the micro-programming paradigm. In this paradigm, we create a very long encoding of an instruction such that it need not be decoded, and furthermore micro-code can directly control the behavior of different hardware units. With micro-programming, we need to know the details of the hardware including the interfaces of all of its components like the ALU and register file. We can create custom instructions by being able to directly program these components. Exposing such low-level hardware details is unthinkable as of today. It would be a very serious security risk.

However, in the good old days, this was considered acceptable. In continuation of this trend, the VLIW community proposed compilers that create large instruction words, which are similar to micro-programs, and have good visibility into the hardware. Components of the instruction's word (binary encoding) direct different functional units to perform different tasks. From packing a set of micro-instructions in a single memory word, this paradigm gradually evolved to co-locating multiple RISC instructions in the same group of memory words (referred to as a bundle). This entire bundle was sent down the pipeline. The obvious benefit was higher ILP, and the obvious shortcoming was the behavior of branches and memory instructions in a bundle. Compilers were conservative and introduced nops (dummy instructions that don't do anything) to avoid stalls and interlocks. This strategy is alright for DSPs (digital signal processors) because their control flow and data flow are both predictable to a large extent.

However, for running general purpose programs, we need to make certain modifications to the basic VLIW design – it will be too inefficient otherwise. We thus arrived at EPIC processors, which are safe by design. This means that even if there are dependences within instructions in a bundle, or across two bundles, the processor handles them using a combination of stalls, speculation, and interlocks. VLIW processors unlike EPIC processors are not necessarily safe and correct by design. EPIC processors thus provide a virtual interface to programs, and internally also do a lot of virtualization and translation. This ensures that a given program compiled for another EPIC processor with the same ISA but a different version, still runs correctly.

### Definition 30

- *In a VLIW processor we create bundles of instructions that can either be regular RISC instructions or micro-instructions. The entire bundle of instructions is issued to the pipeline as an atomic unit, and then parallel execution units execute the constituent instructions. For programs that have a lot of ILP such as digital signal processing routines, this approach is very beneficial because we can achieve very high ILP. However, VLIW programs often rely on the compiler for correctness, and typically have limited portability.*
- *EPIC processors are modern versions of VLIW processors, which are correct by design. In other words, it is not possible to incorrectly execute a program. The hardware assures correctness sometimes at the cost of performance. In addition, programs compiled for one EPIC machine can often execute on other machines of the same processor family that have a different internal organization. The designers of EPIC machines provide a virtual interface to software such that it is easy for compilers to generate code. The hardware internally tries to execute the code as efficiently as possible without compromising on correctness.*

## 5.6 Design of the Intel Itanium Processor

Let us now discuss the reference design of an EPIC processor – the Intel Itanium Processor. Intel and HP<sup>®</sup> together decided to work on a new EPIC processor and released the Itanium in 2001. Note that designing a VLIW processor would not have been the best thing to do because we never want to compromise on correctness. We need to ensure that irrespective of the compiler, the code always works correctly. Hence, creating an EPIC processor that works correctly in the face of nondeterminism due to branch misprediction, cache misses, and interrupts/exceptions, is the best way to go forward. Subsequently, Itanium 2 was released in 2002. However, since our aim is to describe the core concepts underlying an EPIC architecture, we shall mostly describe the architecture of the basic Itanium processor. We source most of the details from the paper by Sharangpani and Arora [Sharangpani and Arora, 2000]

Note that we take some creative liberties in this section particularly for the assembly code. Instead of showing proper Itanium assembly code, we have shown equivalent code in *SimpleRisc* for the sake of readability, and easier explanation. In addition, we try to generalize the architecture and describe multiple competing mechanisms wherever they exist.

### 5.6.1 Overview of the Constraints

Let us now start with some basic questions with regard to an EPIC architecture. What does a compiler do for us? It can find the dependences between instructions, and schedule them if it is aware of the latencies of execution units. We still need a branch predictor to sustain a high fetch bandwidth. Compilers cannot predict branches with 100% accuracy because the outcomes of branches depend on values obtained at run time. We need the decode unit as well because instructions have to be parsed, checked for errors, and converted into an internal representation that is easy to process. In fact, we conceptually need the rename stage as well. This is required because we still need to get rid of WAR and WAW hazards.

The unit that we can get rid of is the scheduler: instruction window, wakeup, broadcast, and select. Since we are supposed to know about the instructions in the pipeline (at compile time), we can schedule them such that all RAW dependences are taken care of. However, this is easier said than done because we have to very accurately take structural hazards into account.

Moreover, we do need the register file access stage and definitely the execution units. Figure 5.15 shows an overview of the Itanium architecture. It is necessary to keep referring to this figure throughout this section.

### 5.6.2 Fetch Stage

The design philosophy of the Itanium processor was to create a very high performance server processor. In line with this philosophy, the designers created a processor that could fetch up to six instructions per cycle. This requires two pieces of sophisticated hardware: a high bandwidth i-cache, and a very accurate branch predictor.

The key to having a high fetch bandwidth is to have an i-cache that supports a high throughput. The Itanium thus has a 16 KB 4-way set associative i-cache that can provide 32 bytes per cycle. In these 32 bytes, we can fit six instructions, which are grouped into bundles of three instructions each. There is a possibility of a rate mismatch between the fetch engine and subsequent stages of the pipeline. In such a case, it is advisable to have a buffer to store instructions that have been fetched, yet are not able to enter the pipeline. Itanium uses a *decoupling buffer* that can store up to 8 such bundles. Note that a *bundle* is an important concept in an EPIC architecture. We typically treat a bundle as an indivisible unit. The compiler creates bundles very carefully. In addition, the three instructions in a bundle should not have any mutual dependences between them.

Subsequently, a highly accurate branch predictor is required. This is because the branch misprediction penalty is nine cycles in Itanium. Given that the compiler has a significant involvement in such

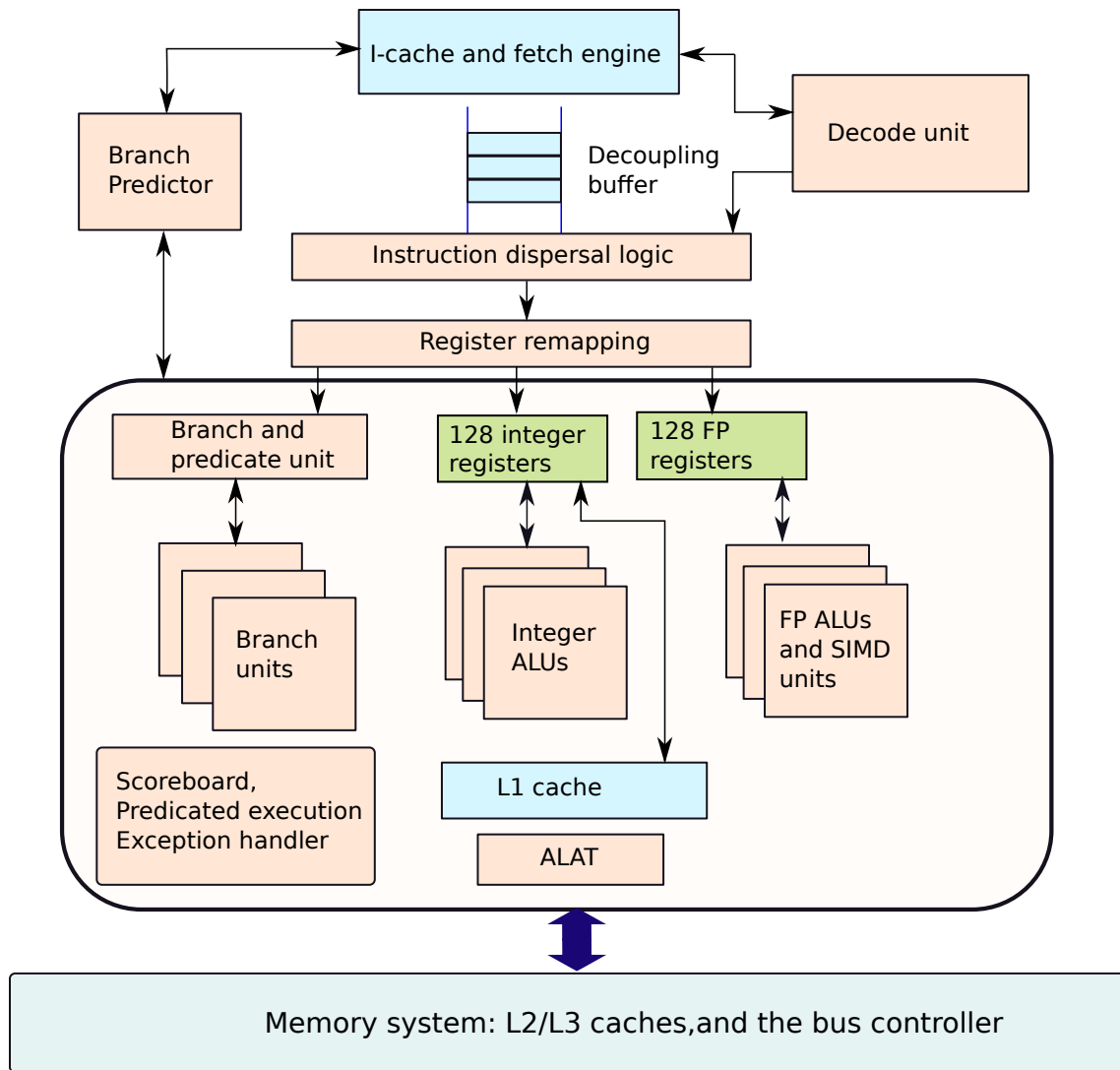


Figure 5.15: The Itanium processor (©[2000]IEEE. Reprinted, with permission, from [Sharangpani and Arora, 2000]).

architectures, we can send a lot of compile time information to the branch predictors. Let us briefly outline the various strategies followed by the designers of the Itanium processor.

**Compiler Directed** Since the compiler plays a very active role in such architectures, we spend much more time analyzing the nature of loops particularly in numerically intensive code. In such codes, the behavior of loops is often very predictable. We have four special registers called target address registers (TARs). It is the compiler's job to enter the branch targets into these registers via special instructions called *hints*. The hints contain the PC of the branch and the target address. Whenever the next program counter matches the PC contained in any of the TAR registers, we automatically predict taken, and extract the branch target from the corresponding TAR register. Thus, the entire process is very fast, and very energy efficient as well. We can easily accomplish this series of tasks within one cycle. How can the compiler be so sure? In most scientific codes, we exactly know the



value of the loop index and the target addresses; hence, the compiler can easily analyze such loops and let the hardware know about the loop continuation and termination information.

**Traditional Branch Prediction** Not all branches are that well behaved. We thus do need high performance branch predictors. Itanium has a large PAp branch predictor that can predict branches very well.

**Multi-way Branches** Note that Itanium instructions are stored in 3-instruction bundles. Typically, compilers ensure that the last instruction in a bundle is a branch. This means that just in case it is mispredicted, the earlier instructions in the bundle are still on the correct path. However, this is not always possible. Just in case there are multiple branches within a bundle, we need a method to handle this situation. Note that in this case, if we consider a bundle as a whole, it is a multi-way branch statement. It has many possible targets depending on the behavior of the branches contained within it. Such a set of branches is also referred to as a *multi-way branch*.

We need to predict the first taken branch within a bundle. This means that if a bundle has three instructions, we have four possible choices for the first instruction that succeeds the current bundle: default, target of the first instruction, target of the second instruction, or the target of the third instruction. Itanium uses a history based predictor for each bundle. This predicts the first instruction that is most likely a taken branch. Once we predict this instruction, we can get its target from the branch target cache.

Itanium has other traditional modules such as the return address stack. They were already discussed in Chapter 3.

**Loop Exit Prediction** After we decode the instructions, we get to know of the opcodes of all the instructions including the branches within a bundle. At this stage Itanium uses a perfect loop-exit predictor, which can override earlier predictions. We need to initialize this predictor with the iteration count of the loop. The compiler marks the loop-branch (branch that takes us to the beginning of the loop) with a special instruction. The loop exit predictor keeps decrementing the loop count, every time we encounter this instruction. It can thus figure out the last iteration, and we can avoid mispredicting the last branch in a loop. This is not a very effective optimization for large loops (large number of iterations). However, for small loops, this is a very good optimization; it avoids a lot of mispredictions.

The last stage of the fetch unit also has the role of processing software initiated prefetch instructions. Itanium's compiler plays a fairly aggressive role in prefetching instructions.

### 5.6.3 Instruction Dispersal Stage

After fetching all our instructions, we keep the instruction bundles in the decoupling buffer. Having such buffers reduces the mismatch between the rate of fetching instructions and the rate of executing them.

Let us now look at instruction dispersal or instruction delivery – providing instructions to the execution units. Our explicit aim was to avoid the use of schedulers. Hence, we need to find a way to avoid data and structural hazards without using expensive hardware such as an instruction window, wakeup, select and broadcast hardware. We disperse (synonym of dispatch in the current context) two bundles of instructions (6 instructions) at a time via the issue ports. The Itanium processor has 9 issue ports: 2 for memory, 2 for integer, 2 for floating point, and 3 for branch instructions. Within each bundle, we disperse the instructions from the earlier to the later. For dispersing instructions, we need to respect both data and control dependences. Let us elaborate on how these are handled.

#### Data Hazards

Ideally, the compiler should find three instructions that are absolutely independent and place them in a bundle. However, this is not always possible. In such cases, we do have the option of putting

*nop* instructions in the bundle; however, here again there are associated performance penalties because of wasted issue slots. Hence, it is sometimes wiser to have instructions within a bundle with data dependences between them – we get more performance than using *nops*. There are two features in the IA-64 ISA (Itanium’s ISA) that make this easier.

1. It is possible to have a compare instruction and a conditional branch that is dependent on its outcome in the same bundle. Itanium can internally forward the result of the comparison to the branch.
2. In the worst case, it is necessary to use *stop bits* in instructions. Let us consider the instructions in the order from the earliest instruction to the latest. Some instructions will have their stop bits set to 1, and the rest of the instructions will have their stop bits set to 0. The instructions between two instructions with their stop bits set to 1, are independent of each other. As a result, we do not need sophisticated hardware to check and mark dependences between instructions. Instructions between two stop bits are also referred to as an *instruction group*. Within an instruction group we have parallelism, and the instructions can be issued simultaneously. Instructions that are not marked by the compiler as independent, need to execute in program order.

### Structural Hazards

Instead of using sophisticated decoding logic, Itanium has a very simple way of figuring out the resource requirements of instructions. It uses a 4-bit template field in each bundle. This indicates the type of instructions in a bundle: M (memory), I (integer), F (floating point) and B (branch). With these 8 bits (4 bits for each bundle), the processor can very quickly find the resource requirements of all the instructions, and schedule the issue ports accordingly.

#### 5.6.4 Register Remapping Stage

We unfortunately do need register remapping (sophisticated form of renaming) here also. It would have been the best if we could have avoided this stage since with advanced compiler analyses, it is possible to analyze the structure of dependences in a program very well. However, as we shall see, Itanium has extended the concept of register renaming and in fact the term that is used in the paper by Sharangpani and Arora [Sharangpani and Arora, 2000] is “register remapping”, which is much more than renaming.

If we think about it, we would still need renaming in some form. This is because we need to get rid of false dependences. One approach is that the compiler is aware of the physical register file, and directly assigns architectural registers to physical registers while generating the binary itself. This approach limits portability and introduces dependences between instructions using the same physical register. This is because the compiler cannot predict with 100% accuracy, and moreover there are several sources of nondeterminism in the execution of modern programs: branch misprediction, and misses in the memory system. Let us look at what Itanium does.

### Virtual Registers

Itanium solves this problem by using virtual registers. The software assumes that the hardware has a multitude of virtual registers, and thus the software simply maps variables to virtual registers. This keeps the software simple and also the code remains portable. The hardware maps the virtual registers to physical registers. The Itanium architecture has a large 128-entry register file. These 128 entries are partitioned into two sets [Settle et al., 2003]: 32 *static* registers that are visible to all functions and 96 *stacked* registers that have limited visibility.

Specifically, Itanium optimizes for two kinds of scenarios: argument passing to function calls and software pipelining. When we are making function calls, we often need to write the values of registers to memory. This is because the called function may overwrite the registers. Hence, it is a good idea to store

the values of the registers in memory, and later restore them once the function returns. Assume that the function *foo* is calling the function *bar*. Now, there are two schemes: caller saved and callee-saved. In the *caller-saved* scheme, the code in the function *foo* is assigned the responsibility of saving and later restoring the registers that might get overwritten. If we hand over this responsibility to the function *bar*, where it needs to save and restore the registers that might get overwritten, then we have the *callee-saved* scheme. Both of these schemes are expensive in terms of memory reads and writes.

Itanium solves this problem by allocating a different set of virtual registers to each function. This ensures that there is no possibility of different functions overwriting each other's registers unless there is an explicit intent to do so. We sometimes deliberately create an overlap between the register sets, when we want to pass arguments and return values between functions. If there is an overlap between the virtual register sets used by the caller (*foo*) and callee (*bar*) functions, then we can pass arguments and return values via virtual registers.

Let us explain with an example. In Figure 5.16, we show the example of a function call. Function *foo* calls the function *bar*. In Itanium it is possible to specify the virtual registers that contain the input arguments (*in*), the local variables (*local*), and the values to be sent to callee functions (*out*). In this case, let us assume that for the function *foo*, virtual registers 32 and 33 contain the input arguments, registers 34-39 contain the local variables, and register 40 contains the value that needs to be an input argument to the function *bar*. For the function *bar* we can create a different mapping. For example, we can assume that register 32 contains the input argument, and registers 33-36 contain the local variables. In this case, there is a need to map register 40 of function *foo* to register 32 of the function *bar* to pass the argument. This can be done by the hardware very easily. We just need to map these virtual registers to the same physical registers. Then unbeknownst to the functions *foo* and *bar*, arguments can be passed very easily between the functions. The need for saving and restoring registers is not there because the registers that are used by different function invocations are different. We only create an overlap in the register sets while passing parameters, otherwise, because the register sets are separate there is no need to spill registers to memory. This decreases the number of loads and stores.

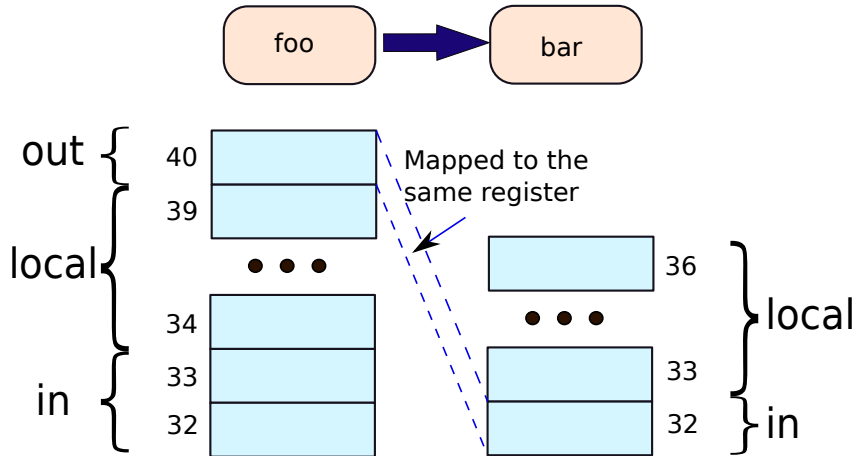


Figure 5.16: Using virtual registers for parameter passing

### Register Stack Frame

We allocate a register stack frame for each function based on a special instruction added by the compiler called the *alloc* instruction. In the stack frame we store three kinds of registers: *in*, *local*, and *out*. There is a fourth optional kind called *rot*, which we shall discuss shortly. However, let us consider the first three

kinds of registers first. The *in* and *local* registers are preserved across function calls. The *out* registers are meant to be accessed by callee functions and thus do not need to be preserved.

Once a function returns, the register stack frame for that function is destroyed. This is similar to a regular stack in programs that is stored in memory. Recall that a conventional stack stores the arguments and local variables of a function. It is destroyed once the function returns. Something very similar needs to be done here. We need to automatically destroy the stack frame after a return.

To summarize, we need to create a stack frame based on the number of virtual registers that the function requires (specified via the *alloc* instruction) when the function is invoked, and then destroy it once the function returns. This process of automatically managing registers across function calls reduces the work of the compiler significantly, and also reduces the unnecessary memory accesses that were happening because of saving and restoring registers.

### Communicating Return Values

The last piece of the register remapping stage is communicating return values. The return value is typically communicated via a static register: *r8*. The main advantage of doing this is *tail recursion elimination*. Consider the following piece of code for the binary search routine.

```
int bin_search(int arr[], int left, int right, int val){
    /* exit conditions */
    int mid;
    if (right < left) return -1;

    mid = (left + right) / 2;
    if(val == arr[mid])
        return mid;

    /* recursive conditions */
    if(val < arr[mid])
        return bin_search(arr, left, mid - 1, val);
    else
        return bin_search(arr, mid + 1, right, val);
}

int main(){
    ...
    result = bin_search ( ... );
next:
    printf("%d", result);
    ...
}
```

Here, we show a traditional binary search routine. All the parts of the code that are not relevant have been replaced with three dots (...). Let us consider the sequence of function calls. *main* calls *bin\_search*, which is called recursively over and over again. The final answer is computed in the last call to *bin\_search*, and then this answer is propagated to *main* via a sequence of return calls. This pattern is known as *tail recursion*, where the statement that produces the result is the last statement in the function. One way of optimizing such patterns is to store the final answer at a known location, and return directly to the label that is after the call to *bin\_search* in the *main* function (label *next* in the code). This will help us eliminate the overheads of tens of return calls. Most compilers are able to recognize such patterns very easily, and they directly replace a sequence of returns with a direct jump to the line after the first call to the recursive function (label *next* in this case). In such cases, it makes

sense to store the return value in a fixed place that is outside the register stack. This is exactly what is done, and that's why we save the return value in a static register.

### Support for Software Pipelining

Itanium has special support for software pipelining. Recall that in Section 5.4.3 we needed different sets of registers for different iterations of a software pipelined loop. Specifically, to store the loop variable we needed different registers, otherwise, there would have been a correctness issue. In fact one of the major limiting factors while doing software pipelining is that we run out of registers. The designers of the Itanium processor have very nicely solved this issue. They have created a rotating register set for storing the loop variable. There is a method to keep track of the iteration of a loop. For every iteration, the hardware automatically assigns new loop variables in a set of virtual registers. This ensures that we can seamlessly implement software pipelining without bothering about manually assigning different loop variables to different iterations. This also significantly simplifies the process of code generation.

### Overflows

Let us now consider the case when we run out of registers. Recall that we only have 128 registers, and if we call a lot of functions, or have large loops, we will clearly run out of registers. The only option that we have is to store the registers in memory, and later on restore them. This is known as *register spilling*. Itanium thankfully has an automatic mechanism for doing this.

#### Definition 31

*The process of saving registers in memory, when we run out of registers, is known as spilling (or register spilling). These spilled registers are later on restored when they are required.*

Itanium has a dedicated Register Stack Engine (RSE). It keeps track of the number of registers we are using, and whenever there is an overflow it comes into action. It silently spills registers at the bottom of the register stack to a dedicated region in memory. When these registers are required, they are restored from memory back again. The programmer and compiler are blissfully unaware of this process. Unbeknownst to them, the RSE performs the task of saving and restoring registers. There is a performance penalty though. While this is happening, the pipeline is stalled for a couple of cycles, and this interferes with the execution of the current program.

## 5.6.5 High Performance Execution Engine

Ensuring high performance in such processors is the joint responsibility of the hardware, compiler, and the programmer. In EPIC processors, the compiler has a disproportionate role. Now, to create a high performance execution engine, the first and foremost requirement is to have a large array of functional units. At the same time, we need to ensure that control and data dependences do not get violated.

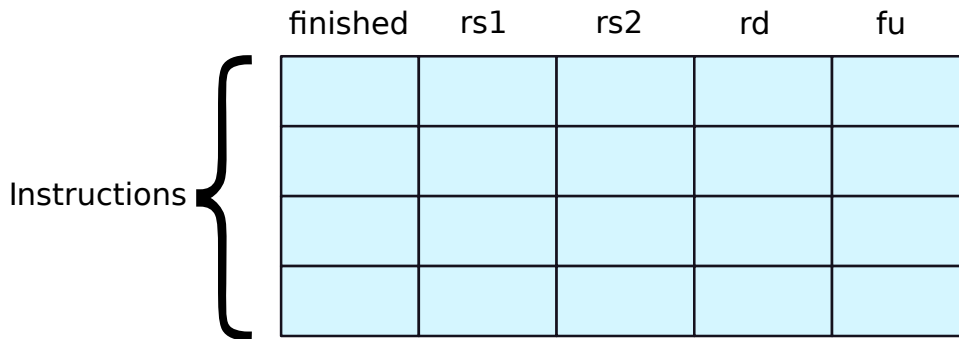
The Itanium processor uses a scoreboard based strategy to enforce dependences. Scoreboarding is a very old technique and has been around since the sixties. The first known use of scoreboarding was by the CDC 6600 machine way back in 1965. Let us look at different aspects of a modern scoreboard.

### Scoreboarding

Scoreboarding [Thornton, 2000, Budde et al., 1990] is a technique that stalls instruction execution till it is guaranteed that the instructions will get the correct values for their operands. It explicitly takes WAW, WAR, and RAW hazards into account, and ensures that correctness is never compromised with.

The exact design of the Itanium Scoreboard is not available in the public domain. Let us thus try to create our own scoreboard.

Let us create a matrix (table in hardware), where the rows are the instructions listed in program order (see Figure 5.17). The columns are *finished* (single bit), source register 1 (*rs1*), source register 2 (*rs2*), the destination register (*rd*), and the functional unit number (*fu*).



	finished	rs1	rs2	rd	fu
Instructions {					

Figure 5.17: A simple scoreboard

Using this matrix, here is how we detect different hazards. We assume that this table is a content addressable array (refer to Chapter 7) that can be addressed by the content of a specific field of an entry. Furthermore, we use the same method to detect earlier instructions as in the load-store queue.

Let us introduce the terminology. For an instruction  $I$ , let the fields  $I.finished$ ,  $I.rs1$ ,  $I.rs2$ ,  $I.rd$ , and  $I.fu$  indicate the status of the instruction, ids of the source registers (1 and 2), id of the destination register, and the id of the functional unit respectively. Given a table entry  $E$ , let us use the same terminology for it as well. For example, the destination register of the instruction associated with the entry  $E$  is  $E.rd$ .

Now, given an instruction  $I$ , we need to create custom logic to ensure the following conditions are never violated.

**WAW Hazards:** Check all the earlier entries in the table. For each earlier entry  $E$ , the following expression should evaluate to false:  $(E.finished = 0) \wedge (E.rd = I.rd)$ . Otherwise, there is a potential WAW hazard.

**WAR Hazards** Similar to the earlier case, for each earlier entry  $E$ , the following expression should be false:  $(E.finished = 0) \wedge ((E.rs1 = I.rd) \vee (E.rs2 = I.rd))$ .

**RAW Hazards** Here is the corresponding expression that should always evaluate to false:  $(E.finished = 0) \wedge ((E.rd = I.rs1) \vee (E.rd = I.rs2))$ .

**Structural Hazards** The corresponding expression is  $(E.finished = 0) \wedge (E.fu = I.fu)$ .

The basic insight is that an unfinished earlier instruction can potentially conflict with the current instruction. Evaluating these expressions is not difficult. We need to access the matrix using different keys – destination register, source register, and functional unit number. Then we evaluate the aforementioned conditions, and if any of these conditions is true, then we stall the current instruction.

We can slightly optimize the scoreboard by avoiding costly CAM accesses for detecting RAW and WAW hazards. We can keep an array called *dest* that is indexed by the register id. Instruction  $I$  sets  $dest[rd] = I$  after getting decoded. This array is thus accessed in program order. Before issuing an instruction we read the arrays and get the instruction ids that will generate values for the source registers, and the latest instruction that writes to the destination register. An instruction thus depends on at the most three other instructions: two instructions that write to the source registers, and one that

writes to the destination register. We wait for all of them to get finished. We thus automatically take care of RAW and WAW hazards in this process.

### Important Point 10

*Why do we need to use a CAM array for detecting WAR hazards? Why can't we use the same trick that we used for detecting WAW and RAW hazards?*

**Answer:** We need to write to a register in program order. Hence, there is a strict order between all the instructions that write to the same register. Thus, it suffices to remember just one instruction for let's say source 1 of instruction  $I$ . This will be the instruction that generates the value for source register 1 ( $rs1$ ); it is also the most recent instruction that writes to  $rs1$  (out of all the instructions fetched before  $I$ ). We stored this information in the dest array, and thus we created a very efficient technique to detect RAW and WAW hazards. However, this mechanism cannot be used for WAR hazards.

For a WAR hazard, we need to find if there is any instruction in the pipeline fetched before the current instruction  $I$  that is unfinished, and reads from the register  $I.rd$ . This information is not stored in the dest array. Thus, it cannot be used in this case. Instead, we need to access every single entry in the CAM array and check if there is a WAR hazard.

### Predication

By now, we know that branch prediction is a very sensitive operation. Even a very tiny increase in the misprediction rate can severely degrade the performance. As a result a lot of processor companies spend a disproportionate amount of time designing branch predictors, and this is often one of their biggest trade secrets. Let us look at mechanisms that do not use branch prediction at all.

Consider a piece of code with a lot of loops, where the number of iterations is known in advance. Itanium has elaborate loop counters that ensure that we shall never have mispredictions in such loops. Now, consider (hypothetically) that we have a small piece of code within a very well behaved loop, which looks like this:

```
if(rand()%2 == 0)
    x = y;
else
    x = z;
```

We generate a random number. If it is even, we set  $x$  equal to  $y$ , and if it is odd, we set  $x$  equal to  $z$ . This branch is genuinely hard to predict because it is based on a random number, and thus finding a pattern is very difficult. In traditional OOO hardware, we still have to predict the direction, and statistically we will mispredict 50% of the time. Every misprediction will lead to a pipeline flush, which is a massive performance penalty. Furthermore, this is very unfair to the rest of the code, which is very well behaved. Maybe such kind of code is embedded inside a library, which is not visible to the programmer. In this case, the programmer in most cases will not even know the reason for poor performance.

We clearly need to do something to handle such cases. Flushing the pipeline is like burning down the house to kill just one mosquito inside it! Let us instead work on creating a different paradigm.

Here is the idea. Let us fully or partially execute a few more instructions in the pipeline than required. If by executing a few additional instructions, we can avoid a costly pipeline flush, which will cost us more than 20-30 cycles, it is well worth the effort. Also, the IPC for most programs is not equal to the issue

width. For example, in the case of Itanium, the IPC will not be 6 most of the time because of limited ILP. We can thus find additional fetch, decode, and issue slots to send in a few more instructions for a large fraction of the time. To explain the idea, let us look at the corresponding *SimpleRisc* assembly code of an Itanium like EPIC processor.

```

1  /* mappings: x <-> r1, y <-> r2, z <-> r3 */
2
3  mod r0, r0, 2    /* assume r0 contains the output of rand(),
4                   compute the remainder when dividing it by 2 */
5
6  cmp r0, 0        /* compare */
7  beq .even
8  mov r1, r3       /* odd case */
9  b. exit
10 .even:
11 mov r1, r2       /* even case */
12
13 .exit:

```

From a cursory examination of this code, it does not look like that we can do anything. However, let us now introduce some additional hardware to open up an avenue of opportunity. Assume that the compare instruction sets two bits *po* and *pe* called the *predicate bits* corresponding to the result of the comparison. If *pe* = 1 we execute the even path (Line 11), and if *po* = 1, we execute the odd path (Lines 8 and 9).

Furthermore, let us augment each instruction in the even and odd paths with the predicate bits that need to be set to 1 for it to execute. The code thus looks as follows:

```

1  /* mappings: x <-> r1, y <-> r2, z <-> r3 */
2
3  mod r0, r0, 2    /* assume r0 contains the output of rand()
4                   compute the remainder when dividing it by 2 */
5
6  po,pe = cmp r0, 0 /* compare and set the predicates */
7  [po] mov r1, r3   /* odd case */
8  [pe] mov r1, r2   /* even case */

```

In Line 6, we set the predicate bits *po* and *pe*. Then, we use these bits for the subsequent instructions. We expect the compare instruction to be ordered before the execution of the instructions that use the predicates generated by it. Subsequently, in Line 7 and Line 8 we execute the instructions in the odd path and even path respectively.

How is this different? Note that we do not have any branch instructions. All the conditional and unconditional branch instructions have been removed. If there are no branch instructions, it implies that there are no mispredictions. The code is linear albeit for the fact that we have predicates. We fetch, decode, and issue the predicated move instructions as regular instructions. We even allow both of them to read the register file. However, the key difference lies in the execution stage. Instead of executing an instruction on the wrong path (predicate bit is 0) we nullify it. In no case, should we allow an instruction on the wrong path to update the register file or write to memory.

Thus, predication is a simple mechanism. It removes branches, and basically converts control dependences into data dependences. We simply need to read the values of the corresponding predicate registers and figure out if the instruction should execute and write back its result or not. For an instruction to do so, it needs to be on the correct branch path, and this will only happen if the values of all the predicates that the instruction depends on are 1. We thus need to compute a logical AND of all the predicate bits. In line with this argument, Itanium adds a high throughput 64-entry predicate register file, where each entry is 1 bit.



Predication is sometimes a very good mechanism, and can do wonders to the performance, particularly for cases that we have looked at. However, note that there are costs associated with this mechanism as well. It increases the number of instructions that need to be fetched, decoded, renamed, and issued. This decreases the effective throughput because a lot of these instructions might potentially be on the wrong path. Furthermore, we need a mechanism of generating and keeping track of predicates. This requires good compiler support.

### 5.6.6 Support for Aggressive Speculation

In Section 5.1, we introduced the notion of aggressive load speculation in OOO processors. Recall that we executed load instructions much before we were aware of their addresses and dependences. We claimed that if we could design accurate predictors, this approach would lead to large performance gains, and indeed this is so in most cases. Let us try to create a similar mechanism for EPIC processors.

Itanium has the notion of *load boosting*. Here, a load and a subset of its forward slice (dependent instructions) can be placed (boosted) at a point that is much before their actual position in the code; this is done by the compiler. This will increase the number of instructions between the load and the instructions that use its value. This means that even if the load misses in the L1 cache, and we need to go to the L2 cache or beyond, most of this delay will get hidden. By the time we encounter the use of the load, its value will mostly likely be in the L1 cache.

This mechanism can very effectively reduce the stalls associated with a read miss. However, there are several correctness issues. Assume that the load instruction encounters an exception. Maybe it accesses an illegal address. To maintain the notion of precise exceptions we need to remember this, and flag the exception only when we reach the original position of the load in the code.

Second, there might be stores between the original position of the load and the hoisted position that have the same address. In this case we will have a RAW dependence violation. Thus, we need to keep a record of all the stores between the two positions, and check for an address match. If we find such a match, the latest such store needs to forward its value to the load.

The summary of this entire discussion is that since Itanium does not have a load-store queue, and if we are still desirous of performing load dependence speculation, then we need to implement the functionality of the LSQ using a combined software-hardware technique. Let us elaborate.

Itanium defines a hardware structure called an Advanced Load Address Table (ALAT). It contains the addresses of all the loads that have been boosted. EPIC processors such as Itanium need to define two instructions for loads: one for a normal load and one for a boosted load. Whenever the hardware encounters an instruction for a boosted load, it enters the load address into the ALAT. Subsequently, each store checks the entries of the ALAT for a match. If there is a match, then we can infer that there is a dependence violation. Thus, we mark the ALAT entry as invalid.

At the original point of the load in the code, Itanium embeds a load check (ld.c) instruction. This checks the validity of the load in the ALAT. If the load is still valid, then it means that the speculation was successful, and nothing needs to be done. However, if the load is invalid, then we need to get the data from the latest store with a matching address. We thus need to reissue the load. This is exactly what is done. In such cases, the chances of getting the store data in the L1 cache itself is very high given the recency of the update. If the load had encountered an exception, then that also can be recorded in the ALAT, and the exception can be handled when the load check instruction is issued.

## 5.7 Summary and Further Reading

### 5.7.1 Summary

#### Summary 4

1. *To further increase the performance of an out-of-order machine, we can perform four types of aggressive speculation.*
  - (a) *Try to predict the address of a load instruction (address speculation).*
  - (b) *Try to predict dependences between load-store instruction pairs (dependence speculation).*
  - (c) *Try to predict the latency of a load instruction (latency speculation).*
  - (d) *Try to predict the value returned by a load instruction (value speculation).*
2. *The standard approaches to make a prediction are:*
  - (a) *Predict the last value if the prediction has a high confidence, otherwise do not predict at all.*
  - (b) *If the value increases by a fixed increment every time it is predicted, then it follows a stride based access pattern. If we detect such an access pattern, then the next prediction is equal to the current value plus the stride.*
3. *Whenever there is a misspeculation (misprediction), we need to replay the instructions that have received wrong values.*
4. *There are three methods of performing a replay:*
  - (a) *In non-selective replay we squash (kill or nullify) all instructions that have read an operand within  $W$  cycles of the faulting instruction being issued. Here,  $W$  is the duration of the window of vulnerability.*
  - (b) *In delayed selective replay, we associate a poison bit with all the values that are computed by the forward slice of the misspeculated load instruction. For all the instructions that have a poison bit set for one of their operands, we squash them, otherwise we let the instruction successfully complete. In this scheme, dealing with orphan instructions is tricky.*
  - (c) *This is solved by the token based replay scheme, where we associate one token for each speculated load. At the cost of additional hardware complexity, this is the most elegant out of our three schemes.*
5. *It is possible to design a fundamentally simpler OOO pipeline by avoiding the physical register file altogether. Instead, we can use the ROB to store uncommitted values. In this pipeline we do not need to store checkpoints. It is stored within the architectural register file, which is updated at commit time. The disadvantage of this pipeline is that we need to store values at multiple locations, and the ROB becomes very large and slow.*
6. *Instead of putting the onus on the hardware to increase performance, we can do a lot of analyses at the level of the compiler such that the branch prediction performance and register usage improves.*

7. *The most complicated optimization in this space is software pipelining. Here, we create an overlap between instructions of different loop iterations, and execute them in a manner such that it is not necessary to stall the pipeline for multi-cycle RAW dependences.*
8. *The epitome of compiler assisted execution is an EPIC processor. It relies on the compiler for generating and scheduling code. This keeps the hardware simple and power efficient.*
9. *The Intel Itanium processor is a classic EPIC processor that moves most of the work to the compiler. Some of its prominent features include compiler directed branch prediction, virtual registers with register windows for functions, hardware support for software pipelining, predicated execution, and support for latency speculation.*

### 5.7.2 Further Reading

For aggressive speculation, the reader can consult some highly cited papers in this area: survey of load speculation [Calder and Reinman, 2000], speculative memory cloaking and dynamic speculation [Moshovos and Sohi, 1999, Moshovos et al., 1997], and selective value prediction [Calder et al., 1999].

Over the years, researchers have proposed many novel methods for implementing processors; one of the most notable examples is the Transmeta Crusoe processor [Klaiber et al., 2000], whose core engine is a VLIW processor. It has a software layer that converts x86 instructions to the native VLIW instructions.

In the area of compiler optimization, the best books are by Aho and Ulmann [Aho and Ullman, 1977, Aho, 2003], and the book on advanced compiler techniques by Muchnick [Muchnick et al., 1997]. For software pipelining, the two classic papers by Rau [Rau, 1994] and Lam [Lam, 1988] provide a very good introduction. Specifically, the paper by Bob Rau proposes modulo scheduling, which is one of the most efficient methods for software pipelining.

Rau's paper [Rau, 1993] on dynamic scheduling in VLIW processors tries to add features of regular pipelined processors such as scoreboard and interlocks to VLIW processors. For a practical perspective, the paper on the iWarp processor [Peterson et al., 1991] is a good reference.

For EPIC processors, the best references are the papers [Sharangpani and Arora, 2000, Settle et al., 2003, McNairy and Soltis, 2003] on the design of the Itanium and Itanium 2 processors.

## Exercises

**Ex. 1** — Does aggressive speculation increase the IPC all the time?

\* **Ex. 2** — In load latency speculation, how do we know that we have predicted the latency correctly or not. In which stage is this logic required?

**Ex. 3** — Design a stride predictor with saturating counters that has some hysteresis. This means that if just one access does not fit the stride based pattern, we have a means of ignoring it.

**Ex. 4** — Extend the design of the predictor that uses store sets such that one store can be associated with multiple store sets. What are the pros and cons of doing so?

**Ex. 5** — Why are values predictable in programs?

**Ex. 6** — How will you use a profiling based approach to improve the value prediction hit rates in programs? In a profiling based approach, we first do a dry run of the benchmark, and collect some run

time information. This information is subsequently used to improve the performance of the regular run of the benchmark.

**Ex. 7** — Assume a program, where we have many variables whose value alternates between two integer values across read operations. How do we design a value predictor to take care of this case?

**Ex. 8** — Compare the advantages and disadvantages of the three replay schemes: non-selective replay, deferred selective replay, and token based replay.

**Ex. 9** — What are the trade-offs between keeping instructions in the instruction window versus keeping them in a separate replay queue?

**Ex. 10** — How do we deal with orphan instructions in the non-selective and delayed selective replay schemes?

**Ex. 11** — Why do we need a kill wire when we already have the mechanism of poison bits in the delayed selective replay scheme?

**Ex. 12** — Design an efficient scheme to separate instructions into predictable and non-predictable sets for the token based replay scheme. Use insights from the chapter on branch predictors.

\* **Ex. 13** — The replay schemes are all about collecting the forward slice. Let us consider the backward slice, which is defined as the set of instructions that determine the value of the destination(result) of an instruction. It consists of the instruction itself, the producers of its source operands, the producers of the source operands of those instructions, and so on. Consider an example.

```
1: add r1, r2, r3
2: sub r4, r5, r6
3: add r7, r1, r0
4: ld r8, 4[r7]
5: add r9, r8, r10
6: add r10, r4, r3
```

The backward slice of instruction 5 comprises instructions 5, 4, 3, and 1. The backward slice of instruction 6 comprises instructions 6 and 2.

Let's say that we need to compute the backward slice of a given instruction in a window of the last  $\kappa$  instructions. Suggest an efficient method to do this given  $\kappa$ . This approach should be fully hardware based.

\* **Ex. 14** — Can a backward slice be defined in terms of forward slices?

**Ex. 15** — What is the problem in accessing registers after the instruction is dispatched in the ARF based design?

**Ex. 16** — In programs with high ILP, is a scheme with a unified instruction window or a scheme with reservation stations expected to perform better? What about for programs with low ILP?

**Ex. 17** — Consider the ARF based design. How many read and write ports do we need in the ROB? Provide an efficient implementation of the ROB.

\* **Ex. 18** — Outline a scheme to perform strength reduction in hardware. Note that the first task is to identify those instructions where a multiplication or division operation can be replaced with a sequence of shift operations.

\* **Ex. 19** — How do we create a loop detector in hardware with possible compiler support? What can it be used for?

\* **Ex. 20** — In software pipelining, is the degree of loop unrolling related to the latency of operations?

**Ex. 21** — What is the function of the ALAT?

\*\* **Ex. 22** — Provide the outline of a compiler algorithm to insert stop bits.

**Ex. 23** — How does Itanium avoid structural hazards?

**Ex. 24** — What is the advantage of tail recursion elimination?

**Ex. 25** — Is scoreboarding an efficient technique? Should it be used in regular OOO pipelines?

\*\* **Ex. 26** — Under what conditions can a load be hoisted in Itanium?

## Design Problems

**Ex. 27** — Design a stride predictor using Logisim or Verilog/VHDL. Create a circuit to predict if the access pattern is based on strides, calculate the stride, and use it for different types of prediction.

**Ex. 28** — Implement the replay based techniques (deferred selective and token based) in the Tejas architectural simulator. Compare and analyze the results.

**Ex. 29** — Implement a load-store dependence predictor in the Tejas simulator.

