4

The Issue, Execute, and Commit Stages

The fetch and decode stages that we designed in Chapter 3 can deliver a steady stream of instructions with a very high throughput. Now, we need to create a high bandwidth instruction execution engine that can execute as many instructions in parallel as possible, subject to area and power constraints. This is the area of study in this chapter. Note that for understanding the contents of this chapter, Chapter 2 is an essential prerequisite. We need to be confident in the concepts listed in Way Point 1.

Way Point 1
At this point we are supposed to be confident with the following concepts.

- *In-order pipelines: 5 stages of instruction processing, interlocks, and forwarding.*
- *Data hazards: WAW, WAR, and RAW hazards. Special case of the load-use hazard.*
- *Basic idea of out-of-order pipelines: true and false dependences.*
- *Familiarity with the following concepts: branch prediction, instruction renaming, multi-instruction issue, and the instruction window.*
- *Knowledge of precise exceptions and in-order instruction completion.*

The first task is to remove all WAR (write after read) and WAW (write after write) dependences from the sequence of instructions. As we had discussed in Chapter 2, this will increase the available parallelism in the instruction stream significantly. This process is known as *renaming*, and requires elaborate hardware support. After renaming the only dependences in the code will be RAW (read after write) dependences. A RAW dependence enforces a strict order of execution between the producer and consumer instructions (see Section 4.1).

Such dependences have the potential to reduce the ILP (instruction level parallelism) unless we take additional measures. The standard approach to dealing with such issues is to take a look at a large set of instructions together, and then find a set of instructions that can be executed in parallel. They should not have any dependences between them. To find this set of independent instructions, we need hardware

structures to track the dependences between instructions, and to find out when an instruction is ready to be executed. For an instruction to be ready, all of its input operands should be ready. It is possible that many instructions may become ready for execution in the same cycle. Given that we have a small set of execution units, we need to choose a subset of the ready instructions for execution. There are elaborate heuristics to *select* the appropriate set of instructions. This has implications in terms of the critical path of the program. Owing to their complexity, these structures are some of the most performance-critical units in the pipeline, and thus are designed very carefully (explained in Section 4.2).

Till now, we have been discussing how to handle instructions that have only register-based dependences. Handling memory instructions requires a different set of architectural structures. This is because register dependences can be figured out right after decoding the instructions; however, memory addresses are computed much later in the execution stage. Hence, most processors use an additional structure called the load-store queue (LSQ) to keep track of memory dependences. The LSQ as of today is a very sophisticated structure that enforces correctness, as well as implements many optimizations to improve performance. We shall delve into such issues in Section 4.3.

Finally, after processing all of these instructions out of order, it is necessary to create an illusion to an external observer that the instructions have actually been executing in program order. This is required to ensure precise exceptions (see Section 2.3.3) such that the program can transparently recover from faults, interrupts, and exceptions. Ensuring this in a complex system with branch prediction and out-of-order execution is fairly complex. We need to ensure that we restore the state of the program to exactly what it should have been right before the exception. This requires us to periodically take checkpoints of the state, and efficiently store them. In Section 4.4 we shall study the trade-offs between the overheads of taking periodic checkpoints and the time it takes to correctly restore the state.

4.1 Instruction Renaming

As we discussed in Section 2.3.3, instructions can have three kinds of dependences between them: WAR (write after read), WAW (write after write), and RAW (read after write). Out of these only the RAW dependence is a *true* dependence. In other words, if instruction B is dependent on the output of instruction A, B has to execute after A finishes. WAR and WAW dependences are in a sense *false* dependences because they arise due to the limited number of registers. If we had an infinite number of registers, then these dependences would not have been there. Note that in this discussion we are not talking about reads and writes to memory addresses. We shall discuss memory dependences in Section 4.3.

4.1.1 Overview of Renaming

Let us quickly recapitulate what we had learned in Section 2.3.3. Consider two pieces of code as shown in Figure 4.1. The code on the left uses regular architectural registers. We note the presence of WAR and WAW hazards, whereas the code on the right has only RAW hazards. This was made possible because we replaced architectural registers by physical registers. This process was called *renaming*.

Before proceeding further, let us clarify the terminology that we shall use. All the architectural registers start with r. We have 16 architectural registers: $r0$ to $r15$. For the time being let us assume an unlimited number of physical registers. The architectural-to-physical register mapping scheme in Figure 4.1 is as follows. Initially, architectural register ri is mapped to physical register pi. However, every time we write to architectural register ri, we assign that avatar of ri to a new physical register. The physical registers are numbered as follows in our example: $pi1, pi2, \ldots$. For example, in Line 1 we write to $r1$. We thus assign it to a new physical register $p11$. When we write to $r1$ again in Line 2, we assign it to a new physical register $p12$. On similar lines, we assign $r7$ to the physical register $p71$ in Line 4. We can think of the first digit as the number of the architectural register that the physical register is mapped to, and the second digit as the version number. Every time we write to the register,

we increment the version number. Note that this numbering is used in our running examples for the purpose of better explanation. In a real system, the numbering is done differently (see Section 4.1.5). Notwithstanding the limitations of our simple scheme, we clearly observe that the renamed code has only one dependence (between Lines 2 and 3), which is a RAW dependence.

Needless to say, during this assignment of architectural to physical registers, the correctness of the program is not affected. The producer-consumer relationships between instructions remain. The code looks like it is compiled for a machine that actually has a very large number of registers. We need to ensure that this is the case, when we discuss a more realistic implementation.

Original code

```
1 add r1, r2, r3
2 add r1, r4, r5 /* WAW dep. r1 */
3 add r6, r1, r7 /* RAW dep. r1 */
4 add r7, r8, r9 /* WAR dep. r7 */
```

Renamed code

```
add p11, p2, p3 1
add p12, p4, p5 2
add p61, p12, p7 3
add p71, p8, p9 4
```

Figure 4.1: Original and renamed code

Now, in practice, we will never have an infinite number of registers. However, let us aim for a situation, where we will never fall short of physical registers. In this case, the number of such physical registers is practically infinite.

This is a very good vision, and we would all like to have a system where the performance nullifying effects of handling WAW and WAR hazards are not there. However, before proceeding, we need to answer a basic question: "Who does the renaming?"

Let us consider what we already know about this issue. Recall that we had argued in Chapter 2 that the programmer should not be aware if the processor is in-order or out-of-order. The programmer needs to see the same view of the registers, which is the architectural register set. Any physical register has to be defined exclusively inside the processor, and has to be visible only to elements within the processor. A physical register should be an undefined concept outside the processor.

This discussion naturally answers the question, "Who does the renaming?" The answer is that the processor does it, unbeknownst to the programmer and the compiler. All WAR and WAW hazards are eliminated by the processor on its own volition, and no cooperation is required by software entities such as the compiler. This is completely internal to the processor.

Let us thus proceed to answer the next question, "How does the processor rename instructions?"

4.1.2 Renaming using Physical Registers

Let us consider a typical ISA that has 16 architectural registers. Let us assume that there are 128 physical registers that are used to do renaming. This means that a piece of code that uses architectural registers is transformed into an equivalent piece of code that uses physical registers. The end result for both the pieces of the code is exactly the same. However, as we have argued in Chapter 2, the code with physical registers runs faster because it does not have WAR and WAW hazards.

Definition 18
A physical register file *is a set of registers within a processor. Each physical register is used for the purpose of renaming. The physical registers are not visible to software or any other entity outside the processor.*

To achieve this, let us create a register file with 128 registers. A register file is defined as an array of registers, where we access the contents of a register based on its id. In this case, since there are 128 registers, each register will have a 7-bit id ($2^7 = 128$).

Way Point 2
Up till now we have covered the following concepts.

- *A processor exposes a set of architectural registers, which are visible to the programmer, compiler, and assembler. Most ISAs typically have between 8 and 32 architectural registers.*
- *For renaming we also need a set of physical registers that are completely internal to the processor. They are not visible to the programmer or the compiler.*

We want to run code, written with architectural registers in mind, to run on a processor that uses physical registers. This is where there is a need to perform renaming. Let us illustrate this by one more example. In Figure 4.2, the code in the column on the left side uses architectural registers, and the code in the column on the right side uses physical registers. The register renaming scheme is the same as that used in Figure 4.1.

Original code

```
1  mov r1, 1
2  add r1, r2, r3
3  add r4, r1, 1
4  mov r2, 5
5  add r6, r2, r8
6  mov r1, 8
7  add r9, r1, r2
```

Renamed code

```
mov p11, 1                1
add p12, p2, p3           2
add p41, p12, 1           3
mov p21, 5                4
add p61, p21, p8          5
mov p13, 8                6
add p91, p13, p21         7
```

Figure 4.2: Renaming with physical registers

4.1.3 The Rename Table

Let us now introduce the core idea of renaming. Let us *not have* an architectural register file. This means that let us not have a separate dedicated storage area for saving the architectural registers. **We shall instead designate a subset of the physical registers as architectural registers.** This mapping between physical registers and architectural registers is dynamic and will keep changing throughout the lifetime of the program. Note that this is a very important concept; hence, we would request the reader to read the next few paragraphs very attentively.

Important Point 4
The programmer and the compiler see a set of architectural registers. They are typically fewer. Most processors have anywhere between 8 and 32 architectural registers. However, in our proposed design architectural registers only exist in theory. They are a concept*. They do not have a permanent home.*

Instead, we define the concept of physical registers. We typically have 100+ physical registers in OOO processors.

The process of renaming creates a mapping between architectural registers and physical registers. This means that if we wish to read the value of a given architectural register, we shall find it in the physical register that is mapped to it. Note that this mapping is a function of time and keeps changing dynamically.

Let us look at a few example mappings in Figure 4.3.

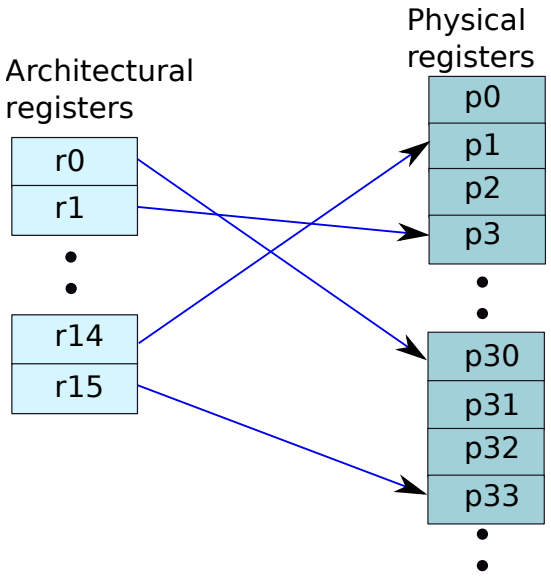


Figure 4.3: Example mapping between architectural and physical registers

We denote the architectural registers as $r0 \ldots r15$, and the physical registers as $p0 \ldots p127$. We assume that we have 128 physical registers. The need for that many physical registers is because we wish to have a lot of instructions in flight such that we can always find a set of instructions that can be issued to the execution units in parallel. The reasons for this will become clear as we read along. Even if readers at this point are not able to understand this logic, we would still urge them to read ahead.

Now, that we have 16 architectural registers and 128 physical registers, we need to create a mapping between architectural registers and physical registers. For example, a mapping would indicate that at a given point of time architectural register $r1$ is mapped to $p27$, and at a later point in time, it is mapped to $p32$, and so on. This means that if an assembly instruction wishes to read the contents of $r1$, the processor needs to read the value of its corresponding physical register. As we just described, this can be $p27$ at one point in the program and $p32$ at one more point in the program.

Figure 4.4 shows a high level overview of the mapping problem. We take an architectural register as input and the output is a physical register. Since in our running example we consider 16 architectural registers, we need 4 bits to encode architectural register ids. On similar lines, we require 7 bits to encode all the 128 physical register ids. Let us thus envision a simple 16-entry table where each entry corresponds to an architectural register (see Figure 4.5). We index the table using the 4-bit architectural register id. Each entry of the table stores a 7-bit physical register id. This is the current mapping between an architectural register and a physical register. Let us call this the rename table.

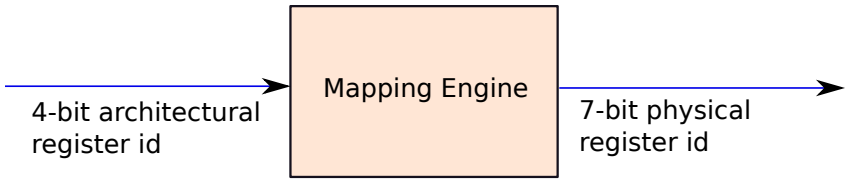


Figure 4.4: Renaming: replacing architectural register ids with physical register ids

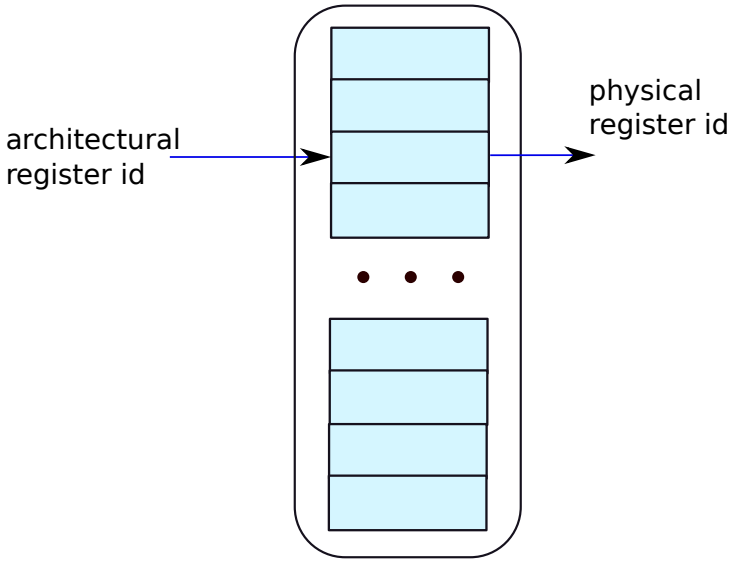


Figure 4.5: The rename table

Definition 19
A rename table *is a table in hardware that stores the mapping between architectural registers and physical registers. It is also known as the* register alias table (RAT table).

Renaming with a RAT table is very easy. We take a look at the source registers, read their corresponding physical register ids from the table, and use them for renaming. However, we need to do something extra for the destination register. Let us consider an add instruction of the form: *add* $r1, r2, r3$. Here, $r1$ is the destination register, $r2$ and $r3$ are the source registers. We need to access the rename table for the source registers $r2$ and $r3$. Subsequently, we need to replace $r2$ and $r3$ with the corresponding physical registers. However, for the destination register, $r1$, we need to follow a different approach. If we think about it, we are creating a new value (a fresh value) for $r1$. The lifetime of this new value starts after $r1$ is written, and continues till $r1$ is written the next time. Let us thus assign an unused physical register to $r1$ (elaborated in Example 2).

Example 2
Rename the following piece of code.

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

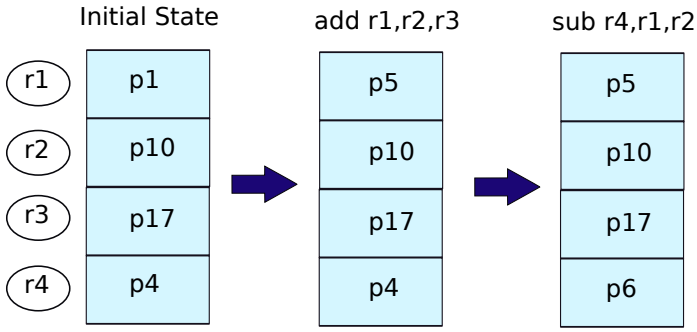
Let r2 be initially mapped to p10 and r3 to p17.

Answer: *For r1, we assign a new physical register, which was hitherto unused. Let this be p5. Along with assigning the physical register p5 to r1, we need to make an entry in the rename table such that subsequent instructions get the mapping r1 ↔ p5 from the rename table.*

The subsequent instruction sub r4, r1, r2 *needs to get the value of r1 from the physical register p5. The same mapping for r2 can be used as the previous instruction because its value has not been updated. We need to assign a new (hitherto unused) physical register to r4. Let this be p6.*

The renamed code is thus as follows:

```
add p5, p10, p17
sub p6, p5,  p10
```



Let us summarize the major steps:

- Consider the source registers (registers that will be read) in the instruction. These are architectural registers.
- Find the corresponding physical registers from the rename table.
- Now, consider the destination register, if any. It needs to be assigned an unused physical register. Assign a free physical register (algorithms to be discussed later) and update the rename table with the new mapping.

The idea of renaming sounds easy in theory; however, there are still many practical challenges that need to be solved. Let us look at some tricky corner cases. Remember that life is not always nice and round. It does have *corners* 😕, and it is often these corner cases that make life very difficult. However, in every adversity lies an opportunity, and most of the time some of the most sophisticated techniques get developed because of these corner cases. Let us look at some corner cases in the idea of renaming that we have presented up till now.

The scheme that we have discussed is fine for a simple processor that renames only one instruction per cycle. However, for a processor that renames multiple instructions per cycle, there are additional problems.

Consider the following block of code.

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

The second instruction uses the destination register of the first instruction as a source register. As a result, the renaming of the second instruction is dependent on the physical register that is assigned to the destination of the first register ($r1$). There is thus a dependence, and we need to wait for the first instruction to be fully renamed. This however will limit the amount of parallelism that we have in programs and will heavily restrict the ILP. Thus, there is a need for a better solution.

Now, think about the case where we rename four instructions together (rename width = 4). Here, there can be many dependences between the instructions. We clearly don't want to rename the instructions one after the other. There has to be a faster way of doing it.

4.1.4 Dependence Check Logic

Let us now solve the general renaming problem, where we have k instructions that need to be renamed together. Let us assume that we have enough free registers such that we can assign a free physical register to each of the destination registers.

The problem is that we can have a dependence where an earlier instruction writes to a register that is required by a later instruction as a source. In this case, the later instruction needs to know the id of the physical register that has been assigned. This is not possible to do (very easily) if they are all being renamed together. This makes the problem complex.

To solve this problem let us consider an example in Figure 4.6 with four instructions that need to be renamed together.

```
1 add r3, r1, r2
2 add r5, r3, r4
3 add r8, r6, r7
4 add r9, r8, r8
```

Figure 4.6: Example for the discussion on RAW dependences

Here, instruction *2* has a RAW dependence with instruction *1*. It is necessary to assign a physical register to $r3$ before we start renaming instruction *2*. Likewise, we have a similar dependence between instructions *3* and *4*. In the worst case, it is possible that we have dependences as follows: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, where $a \rightarrow b$ indicates a RAW dependence between instructions a and b. Here, instruction a is the producer and instruction b is the consumer.

When we have such a dependence $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$ between all four instructions, we need to rename the instructions serially (one after the other) as per the knowledge we have right now. This is clearly suboptimal, and we are not reaping any advantages of parallelism.

Let us assume that the time it takes to rename one instruction is T nanoseconds (ns). In the best case when there are no dependences between instructions, we can rename all the four instructions in T ns. However, if we have dependences between each pair of consecutive instructions, then there is a need to rename them serially, and the entire process will take $4T$ ns. We need to search for a better solution that is closer to T rather than $4T$.

It is important to first make certain key observations regarding the renaming process. Let us consider our running example once again (see Figure 4.6). We can rename instructions *1* and *3* in parallel. Let us now consider the case of instructions *2* and *4*. For instruction *2*, we do not really have to wait for the renaming of instruction *1* to finish completely. In fact an overlap exists. Let us take a look at Figure 4.7 to understand why.

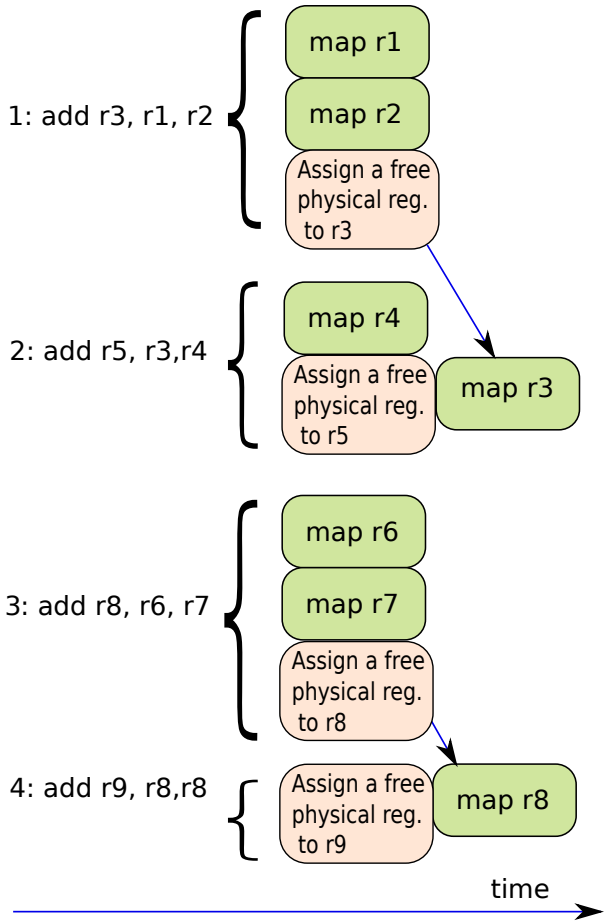


Figure 4.7: Flow of actions for simultaneously renaming four instructions

For instruction *1*, we can read the rename table for registers $r1$ and $r2$ in parallel. At the same time, we can also start reading the rename table for register $r4$, which is a source register for instruction *2*. The dependence exists for register $r3$. We need to wait for instruction *1* to assign a physical register to $r3$. This process can also be initiated in parallel as shown in Figure 4.7. Once we have assigned a physical register, this value can be forwarded to instruction *2*. There is per se no need for instruction *2* to access the rename table to get a mapping for $r3$. Akin to forwarding in pipelining, it can directly get the mapping from the hardware that is processing instruction *1*.

We thus observe that it is possible to perform a lot of actions in parallel while renaming. For example, instruction *2* does not have to wait for instruction *1*'s renaming to completely finish. In this case, instruction *2* simply needs to wait for a physical register to be assigned to *1*'s destination register.

We can have many more such cases, where for example in a 4-instruction bundle instruction *3* has RAW dependences with both instructions *1* and *2*. In that case the nature of actions will be different. We will have to wait for both instructions *1* and *2* to assign physical registers to their destination registers. Subsequently, instruction *3* can quickly use the values that have been forwarded to it by instructions *1* and *2*.

We can clearly see that the space of possibilities is very large. However, our goal is very clear – reduce the time required for renaming as much as possible.

For this we need to use a trick from our bag of architectural tricks. The specific technique that we

shall use involves doing extra work that might be discarded later. However, we nevertheless need to do the additional (redundant) work because we might not be in a position to know if we need to do the additional work or not.

Using Redundant Work to Solve the Problem

Let us look at two of the instructions that we have been considering for renaming once again. We shall use the following piece of code as a running example.

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

We did outline a solution in Figure 4.7, where we try to create an overlap between the process of assigning physical registers, and accessing the rename table. It is not fully practical. This is because there is an assumption in this figure that we are already aware of the RAW dependences between the instructions. This is not the case; hence, we need to create a practical implementation that is conceptually similar to the flow of actions proposed in Figure 4.7.

Consider the following line of reasoning. The physical register assignment for $r3$ will be produced by the renaming process of instruction *1*. However, at the outset we have no way of knowing if at all there is a dependence between instructions *1* and *2*. The process of finding whether there is a dependence or not takes time, and during that time we would like to do useful work. It is possible that there might be a dependence, or it is alternatively possible that there is no dependence. There is no way of knowing without finding out, and that takes time.

We thus propose an alternative method of operation keeping our 2-instruction example in mind. Let us read the mappings for all the source registers from the rename table together. The source registers are $r1$, $r2$, $r3$, and $r4$. We will get valid mappings for three registers $r1$, $r2$, and $r4$. We will however not get a valid mapping for $r3$ because it is simultaneously updated by instruction *1*. Herein, lies the issue.

Let us simultaneously start a process of finding dependences between the instructions. We need to compare the source registers of instruction *2* with the destination register of instruction *1*. In this case, we compare 3 (from $r3$) with the numbers 3 (from $r3$) and 4 (from $r4$). There are two possibilities. Either there is no match, or there is a match. The former case is very easy to handle. It basically means that all the mappings from the rename table that we are simultaneously reading are all correct. However, the latter case is tricky. It means that some of the mappings that we are simultaneously reading are not correct. Let us just note down those mappings. In the case of this example, the mapping for $r3$ that is read from the rename table is not correct.

To ensure that we do not waste a lot of time, let us in parallel start a process to assign new mappings to the destinations of instructions *1* and *2*. This means that while we are reading the mappings of the sources from the rename table, we are simultaneously assigning new physical registers to the destinations. The latter is an independent activity. Let us assign $r3$ to the physical register $p22$.

Once, we have figured out the dependences between instructions, we are in a position to know that the mapping for $r3$ must come from the physical register assignment unit. It needs to be $p22$ in this case, and not the mapping that is contained in the rename table. We thus have a very simple problem on our hands. We have two options to choose from: a mapping from the previous instruction, and a mapping from the rename table. We can quickly choose between these options and get the final mapping between the architectural registers and the corresponding physical registers.

The timing is shown in Figure 4.8. We create an overlap between three actions: reading the rename table, assigning an unused physical register to an architectural register, and computing RAW dependences.

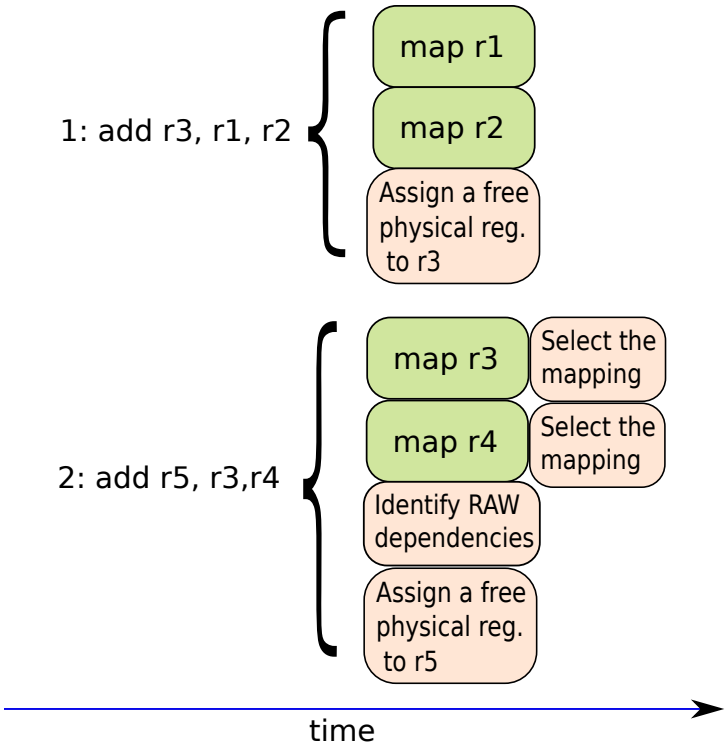


Figure 4.8: Flow of actions in a practical renaming system that tracks dependences

Final Solution for a 2-Issue Processor

Let us now implement this in hardware.

The first piece of hardware that we need is a circuit to choose between two options: mapping read from the rename table and the physical register assigned to the destination of a previous instruction. We need to choose one of the options based on whether there is a RAW dependence or not. The hardware structure that achieves exactly this is a **multiplexer** (see Figure 4.9).

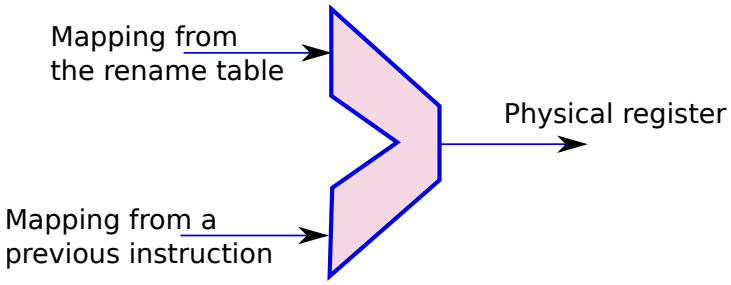


Figure 4.9: Multiplexer to choose between two options for the purpose of renaming

In Figure 4.9 we show a multiplexer with two inputs, one output (final register mapping), and one bit for selecting the input that is based on the comparison of the source register id and the previous instruction's destination register id. Let us now use this multiplexer to design the renaming stage. The logic is now complete because it traces RAW dependences between instructions that are being

simultaneously renamed. It is referred to as the dependence check logic.

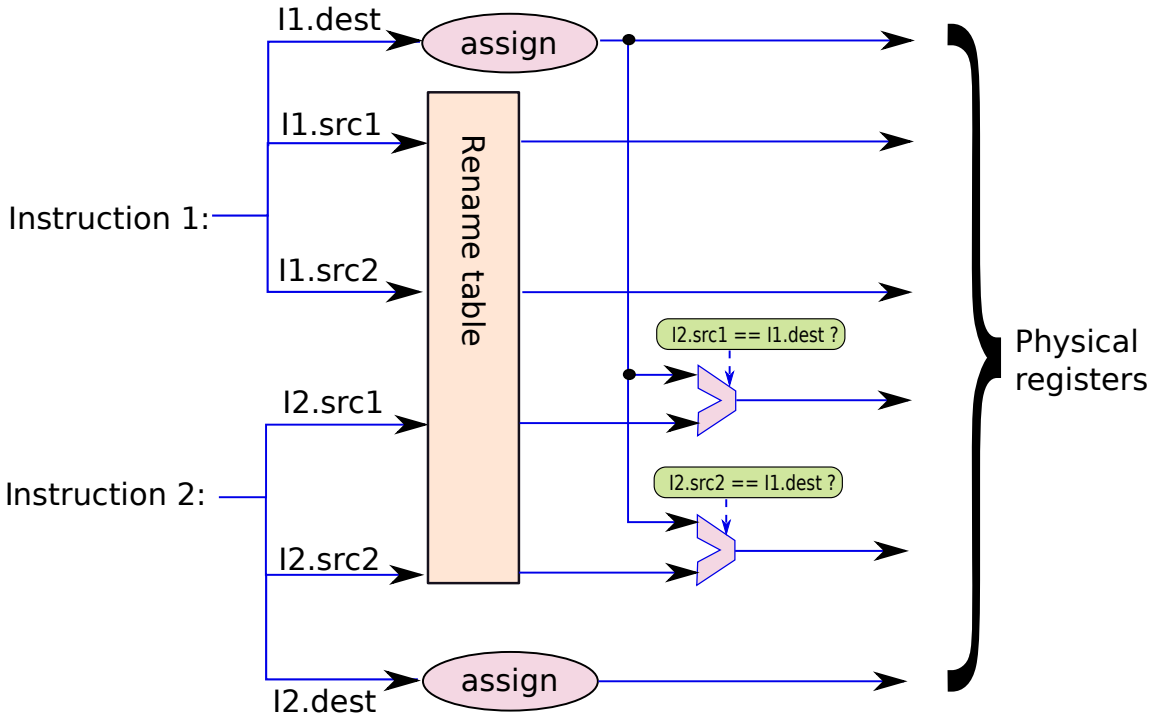


Figure 4.10: The rename stage with dependence check logic

Here is a textual description of our solution (also refer to Figure 4.10).

1: Parallel Activity: starts at t=0 Read the mappings of all the source registers.

2: Parallel Activity: starts at t=0 Assign physical registers to all the destination registers.

3: Parallel Activity: starts at t=0 Find RAW dependences between all the instructions.

4: Final Activity Once when activities (1), (2), and (3), are over $\rightarrow$ for each source register choose the right mapping with the help of a multiplexer.

We leave the process of extending our solution to a processor with a larger rename width as an exercise for the reader. All that we need is a wider multiplexer that takes in more inputs. Assume that we are renaming a set of instructions in parallel, and we need to choose the mapping for a source operand of instruction k. We need to consider the mapping provided by the rename table and the physical registers assigned to the destinations of the previous $k-1$ instructions. Thus, we require a k-input multiplexer. Additionally, we need the logic to compute the control signals for these multiplexers. For a k-input multiplexer we need $\lceil log_2(k) \rceil$ bits.

4.1.5 The Free List

The only part that is remaining is how to assign a *free* physical register to an instruction's destination register. For this purpose we use a structure called a *free list*.

Definition 20
A free list *is a hardware structure that maintains a list of physical registers that are currently free, and can be assigned to architectural registers.*

Let us think of the free list as a black box. It takes in a request for a physical register, and returns the id of a free physical register. Similarly, we can also return a physical register to the free list. The quintessential way of designing a free list is by using a circular queue. The circular queue stores a list of registers as shown in Figure 4.11.

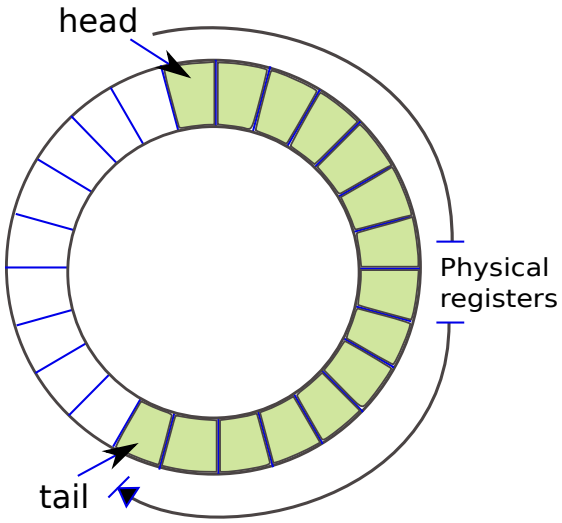


Figure 4.11: A circular queue of physical registers

A circular queue is an array of entries, where each entry contains the id of a physical register. When we add more entries, they wrap around the array and start getting added from the beginning. We maintain two pointers – *head* and *tail* – in hardware. Whenever we add a set of entries to the circular queue we add it to the tail and *increment* the *tail* pointer. Similarly, when we remove entries, we remove them from the side of the head. We also increment the *head* pointer.

The increment operation on the *head* pointer is $head = (head + 1)\%SIZE$. Here, $SIZE$ is the size of the queue. The reason we perform a '%' (remainder or modulo) operation is because the queue is supposed to wrap around (notion of a circular queue). Similarly, the corresponding operation for the tail is $tail = (tail + 1)\%SIZE$.

If the reader at this point is having difficulty, and finding it hard to understand the notion of a circular queue, then she can consult any of the classic texts on basic data structures and algorithms such as the book by Cormen et al. [Cormen et al., 2009].

To check if the queue is empty or not, we maintain a simple count of the number of entries currently present in the queue. When we add entries we increment the count, and when we remove entries we decrement the count. If the count becomes zero we can infer emptiness.

Such circular queues are very common structures, which find use in many architectural components. The free list is one such example, where we can keep track of unused physical registers, and assign them to architectural registers as and when required. When a free physical register is required we dequeue an entry from the free list, and similarly when we need to return a register we can add (enqueue) it to the

free list. The benefits of using a circular queue are its simplicity and the ease of adding and deleting an entry.

There is one open question remaining.

Question 1
When do we add physical registers to the free list?

Unfortunately, we will have to wait till we discuss the process of committing instructions (see Section 4.4) to find the answer to this question. Till that point let us assume that we are never short of physical registers, and we always find enough physical registers to satisfy our requirements.

Now, the set of renamed instructions, which don't have any false dependences need to be sent to the execution units.

4.2 Instruction Dispatch, Wakeup, and Select

Now that we have a stream of renamed instructions, we are sure of the following:

- The stream does not have any WAW and WAR dependences; we only have genuine RAW dependences.
- All the instructions access physical registers.

Now, the task at hand is to first provide a place to temporarily buffer the instructions. In this buffer, we need to find w instructions per cycle to be sent to the execution units. Here, the number w is known as the *issue width*. It is typically a number between 1 and 6.

Note that a renamed instruction might not find its input operands immediately. The instruction that is producing the value of the input operand might be in the temporary buffer awaiting execution. In this case the instruction needs to wait. Similarly, many other instructions would be waiting. However, we will have some instructions whose operands are ready. We can then *issue* them to the execution units. The aim is to choose as many instructions as we can – subject to the issue width – and then issue them to the execution units. Let us define three terms here namely *instruction window*, *dispatch,* and *issue* (refer to Figure 4.12 and Definition 21). In addition, let the term *scheduling* encompass the process of dispatching and issuing an instruction.

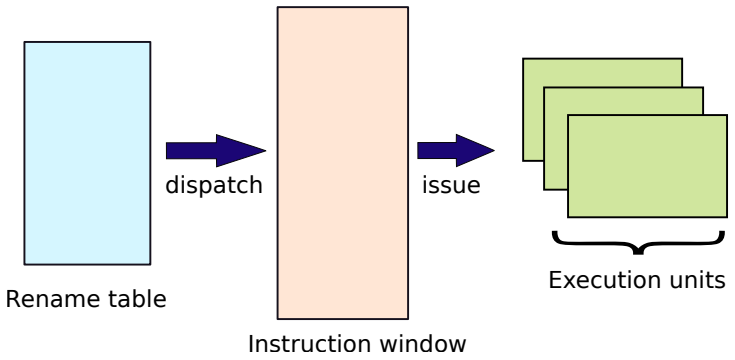


Figure 4.12: Instruction, dispatch, and issue

Definition 21

- *An* instruction window *is a storage structure that temporarily buffers instructions after they are renamed. Instructions wait in the instruction window till their source operands are ready, and the execution unit is free.*
- *The process of sending instructions from the rename table to the instruction window is known as instruction* dispatch.
- *The process of sending instructions from the instruction window to the execution units is known as* instruction issue.
- *The entire process of dispatching the instruction, temporarily buffering it, and then issuing the instruction to the execution units is known as* scheduling.

4.2.1 Instruction Window

Before doing any further processing, let us temporarily buffer the instructions in a queue called the *instruction window*. Most hardware implementations of queues use circular queues as we had described in Section 4.1.5. Recall that such queues use an underlying array of entries; this is implemented in hardware. We have a head pointer, a tail pointer, and a count of the number of entries. A pointer in this case is an index in the underlying array.

Let us outline the need for having an instruction window or having a queue in general between pipeline stages. A queue is used to buffer instructions and provide a sense of *rate control*. Assume that at a given point of time, we are renaming four instructions per cycle, and we are able to execute only two instructions per cycle. This situation can happen if we have a lot of RAW dependences between the instructions, and we do not find enough instructions to execute every cycle.

In such cases, it is nice to have a queue that can absorb the excess instructions, albeit temporarily. Later on, when we can execute more instructions in parallel, the instructions can come from the instruction window. In other words, a queue can buffer instructions till they are consumed and thus try to reduce the mismatch between the rate of production and the rate of consumption. It can absorb spikes in excess production or excess consumption.

The other advantage is that if we are looking at a large set of instructions at a given point of time, the probability of finding more instructions that have all their inputs ready is higher. We should thus have a fairly large instruction window such that we can find a lot of instructions to send to the execution units in parallel.

If the issue width is w, then the best possible situation is when we can issue w instructions per cycle. However, most of the time, we will not find enough instructions primarily because we will be constrained by dependences.

Our aim is that in the instruction window we should be able to find all the instructions that can be issued simultaneously. This has to be an efficient process in terms of both performance and power. Figure 4.13 shows the pipeline of the processor that we have described till this point.

Structure of an Entry

Let us look at the structure of an entry in the instruction window. Its list of fields is shown in Table 4.1. We consider a 64-bit processor with 16 architectural, and 128 physical registers. Note that we are not showing all the fields that are typically associated with an instruction. For example, there are other fields

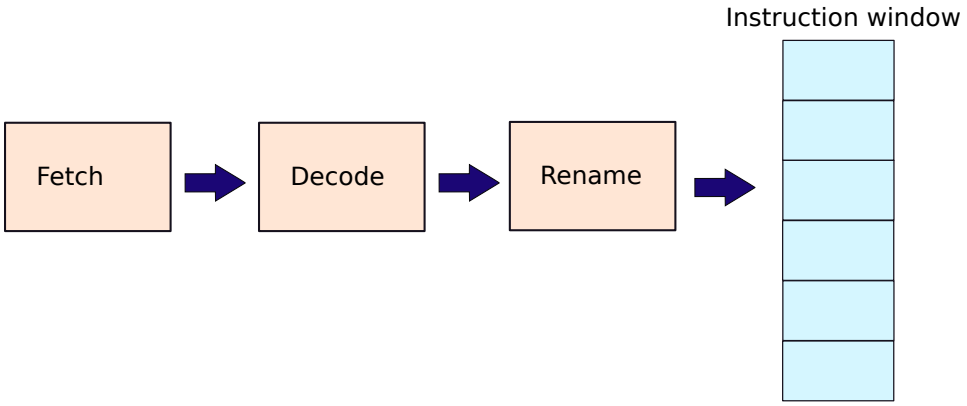


Figure 4.13: The instruction window

in the instruction packet that will be used later on such as the opcode, id of the destination register, and some control signals for controlling execution units. We have two options: either we can keep this information in the instruction window entry, or keep it in a separate location and ensure that the two parts of the instruction packet move together in the pipeline. A shortcoming of the former strategy is that it makes each entry in the instruction window very large, and a shortcoming of the latter scheme is that it makes the design of the pipeline more complicated. Designers typically make such difficult choices after detailed simulation based studies.

Field	Description	Width (in bits)
valid	validity of the entry	1
ready	instruction is ready to be executed	1
First source operand		
isreg1	register or immediate	1
ready1	value is present in the register file	1
rs1	id of the first source register	7
imm1	32-bit immediate	32
Second source operand		
isreg2	register or immediate	1
ready2	value is present in the register file	1
rs2	id of the second source register	7
imm2	32-bit immediate	32
Destination		
isregd	destination is a register	1
rd	destination register id	7

Table 4.1: List of fields in an instruction window entry

Let us now explain the fields in Table 4.1. Note that we consider instructions with two source operands and one destination operand (refer to Appendix A for the details of the ISA). A source operand can be a register or can be an immediate value calculated in an earlier stage of the pipeline (typically decode). There of course can be many other kinds of instructions such as branch instructions that need not have any source operands, or compare instructions that do not have any destination operands. Extending the current approach to handle such instructions is fairly trivial and is left as an exercise for the reader. Let us now focus on the broad concepts. The first row is self explanatory. The *valid* bit indicates if a

given entry is valid or empty. Let us discuss the rest of the rows. We propose a *ready* bit that indicates whether the instruction is ready to be executed or not. If all the operands are either immediates or can be found in the register file, then we set the *ready* bit to 1. Otherwise, we set the *ready* bit to 0, and wait for the operands to be ready.

Going back to Table 4.1, let us consider the set of rows labeled as "First source operand". The field $isreg1$ indicates if the first source operand is a register or an immediate. If this field is 1, then the first source operand is a register. Next, we have the field $ready1$, which indicates if the operand is ready. If the operand is an immediate, then $ready = 1$, otherwise it indicates if the operand can be found in the register file or not. If the operand is a register, then the id of the register is stored in the field $rs1$. Finally, the field $imm1$ contains the 32-bit value of the immediate if the operand is an immediate. Note that these are all physical registers. We are not considering architectural registers here.

We use a similar terminology for the second source operand. The corresponding fields are $isreg2$, $ready2$, $rs2$, and $imm2$.

Finally, let us consider the last set of rows that correspond to the destination register. The field $isregd$ indicates if we have a destination register or not, and the field rd is the id of the destination register. Note that some instructions notably the store instruction do not write to a register, or in other words, do not have a destination that is a register.

Let us now discuss the rules for populating an entry in the instruction window. After we decode an instruction we are aware of its type, and its nature of operands. We are also aware of which operand is a register, and which operand is not. All of this information is a part of the instruction packet that moves from stage to stage. Hence, while creating an instruction window entry, all this information is available. Here, we are making an assumption that physical register ids – obtained post renaming – are also a part of the instruction packet.

The $ready1$ and $ready2$ Bits

Let us explain the logic for setting the fields $ready1$ and $ready2$. Recall that these fields indicate whether the operand is ready or not. If the operand is an immediate, then the logic for setting these flags is obvious. Since we get the value of an immediate at the time of decoding an instruction, the operand is definitely ready. Hence, let us not consider this trivial case. Let us instead consider the more difficult case where the operand is a register.

In this case, there are a lot of possibilities. After the instruction enters the instruction window, it needs to know whether it needs to wait for a source register's value to be produced by an earlier instruction, or the instruction can be issued to the functional units. It will then read the value of the source register along the way from the physical register file. It is important for us to be armed with this information at the time of entering the instruction window such that we know whether we need to wait, or we can proceed.

Thus, setting the values of the fields $ready1$ and $ready2$ is important, and at the moment appears to be non-trivial. The problem at hand is to make a determination before entering the instruction window to the value of these fields. The previous stage is the renaming stage. We clearly need to make a determination in this stage. Let us thus propose a small modification to the rename table.

Let us reconsider the structure of a rename table entry (see Figure 4.14). Let us add an additional bit to the mapping that indicates whether the value of a given register (at that point of time) exists in the physical register file or not. If it does not exist, then it means that the value is going to be produced by an earlier instruction in the pipeline. Let us call this bit the *available* bit, and represent it with the mnemonic *avlbl*. The *avlbl* bit thus indicates the availability of a source operand in an instruction, which is being renamed.

Assume that an instruction i is being renamed in the 10^{th} cycle, then the *avlbl* bit for the source operand will indicate if the instruction expects to find the values of the source registers in the register file or not. This typically refers to the status of the source registers at the beginning of the 10^{th} cycle. Once instruction i reads its corresponding *avlbl* bit it takes this information along with it to the next

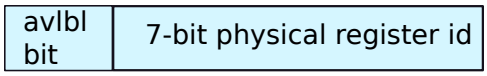


Figure 4.14: A rename table entry with the *avlbl* bit

.

stage.

If the *avlbl* bit is 0 for a physical register px, then instruction i needs to wait in the instruction window for the value of px to get produced. This value will be produced by another instruction that is currently in the pipeline. However, if the *avlbl* bit is 1, then the value for px is ready. It can be read from the physical register file, or it can be forwarded by another earlier instruction in the pipeline. Now the question that we need to answer is, "When do we set the *avlbl* bit?" This is set when the producer instruction writes the value of px to the register file. At the same time the producer can set the *avlbl* bit of px in the rename table.

It is true that adding a bit does solve the problem. However, it brings along a lot of complexity along with it. Assume that we are renaming and issuing 4 instructions per cycle. If we assume that each instruction has two sources and one destination, then our rename table requires 8 read ports (reading two sources) and 4 write ports (creating a mapping by reading the free list). Now, we have further burdened the rename table with the load of maintaining *avlbl* bits. We need 4 extra write ports such that we can update the available bits.

A structure with 8 read ports and 8 write ports is to say the least very complicated and difficult to design. However, we should get some relief from the fact that each entry is only a single bit. Since each entry is a single bit, we can design fast structures for achieving this. Sadly, at this point we are not in a position to understand the design issues associated with memory structures. We shall look at such issues in Chapter 7. Some of the broad approaches that we shall discuss in Chapter 7 include dividing a large array into several smaller sub-arrays. Each such sub-array will have a much lower number of read/write ports. Furthermore, we can also use simple flip-flops instead of expensive SRAM (static RAM) arrays. By a combination of such approaches we can design a fast structure that can be used to store the *avlbl* bits.

Important Point 5

- *The ready field in each instruction window entry is related to ready*1 *and ready*2 *as follows: ready* $= ready1 \wedge ready2$*. It is not strictly required because it can be inferred from ready*1 *and ready*2*. We have added it for the sake of simplicity.*
- *An operand can also be considered to be* ready *if we can read its value from the forwarding paths. This is a minor point and will be revisited later. The point to note is that when we say that an operand is "ready", its value is either available via a forwarding path or can be read from the register file.*

4.2.2 Broadcast and Wakeup

Now that we have enqueued all the renamed instructions in the instruction window, it is time to execute them. However, they may not be ready yet. Some of these instructions might be waiting for their source operands to be ready as we discussed in Section 4.2.1. Once the source operands are ready, the instructions can be sent for execution.

The problem that we wish to solve in this section is to create a mechanism to track and resolve dependences between instructions. There has to be some mechanism by which a producer instruction can let all its consumer instructions know that the value of its destination register is ready. It basically needs to *broadcast* the produced value to the consuming instructions. Furthermore, for this mechanism to work, each consumer instruction needs to wait for its operands to become ready. Once, all of its operands are ready, it is said to *wakeup* [1] Let us thus propose an architecture for the broadcast and wakeup mechanism, where a producer broadcasts information regarding its completion, and consumer instructions use this information to wakeup. We shall first discuss the architecture for broadcast (see Figure 4.15).

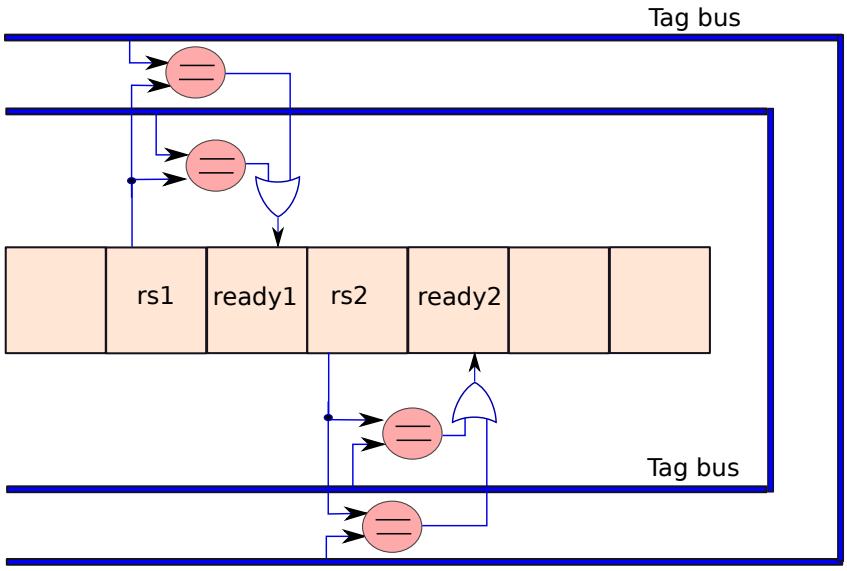


Figure 4.15: Instruction window with additional logic for tag broadcast and comparison

Each entry in the instruction window is connected to at least one set of copper wires called a *tag bus*. A tag bus is used to broadcast the id of the physical register whose value has been written to the register file. We alternatively refer to this id as a *tag*. The producer instruction broadcasts its tag (id of its destination register) on a tag bus: this is connected to both the source operands 1 and 2. If we have multiple instructions executing per cycle, then we need multiple tag buses. An instruction writes the id of its destination register to its corresponding tag bus. Figure 4.15 shows two tag buses: one for each producer instruction.

Let us elaborate. Consider the following code snippet.

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

Instruction *1* is the producer instruction. It produces the value for register $r1$. Assume that register $r1$ is mapped to physical register $p17$. Then 17 is the value of the tag that gets broadcasted on the corresponding tag bus. Each entry in the instruction window is connected to all the tag buses. For each source operand, we check to find out if it is equal to any of the broadcasted tags. This is done with the help of comparators and an OR gate as shown in Figure 4.15. If there is a match, then we get to know that the corresponding source operand is ready. For example, in the current scenario, instruction *2* is

[1] Note that we shall use the term "wakeup" rather than the regular English words "wake up" or "wake-up".

the consumer instruction. It waits for the value of register $r1$ (mapped to $p17$) to be produced and to be subsequently broadcast on a tag bus. Once there is a match in the comparator for the first source operand of instruction *2*, we can set the corresponding ready bit to 1. This means that we can proceed to read the value corresponding to architectural register $r1$ (physical register $p17$) from the register file.

Every producer instruction broadcasts its destination tag (if it has one) on one of the tag buses. This allows consumer instructions to see the broadcast and subsequently wakeup. We can further augment this mechanism to send the broadcast to the rename table. This will update the available bit (*avlbl*).

The process of waking up is simple. Once we observe the tag on a tag bus, we mark the corresponding operand as ready, and if all the operands are ready, we proceed to execute the instruction. This is in itself a multi-step process. We first need to set the *ready* bit in the instruction window's entry to indicate that the instruction is ready for execution. It is possible that multiple instructions might be ready for execution. For example, it is possible that five add instructions are ready; however, we have only two adders. In this case we need to choose two among the five instructions for execution. This process is called *instruction selection*.

4.2.3 Instruction Select

We can conceptually think of two kinds of valid instructions in the instruction window. One set of instructions have all their operands ready, and the other set of instructions are waiting for their operands to be ready. Let us consider the former set. Note that in any processor we cannot simultaneously execute all the instructions which are in the *woken up* state. It is theoretically possible that one instruction wakes up 100 other instructions. All of these 100 instructions will become immediately ready to execute. Given the fact that we have a limited number of functional units, we need to choose a subset of instructions that can execute simultaneously.

For example, if we have two adders and one multiplier, and have five instructions that are ready, we need to find two add instructions and one multiply instruction out of the five ready instructions. This involves a certain amount of decision-making. The aim is to maximize the ILP.

Let us design a basic select unit that selects only one instruction (see Figure 4.16). Consider a theoretical version of the problem where we have n entries in the window. Each entry is connected to the select unit. If the instruction in an entry is ready (woken up), then it sets its *request* line to 1. However, if it is not ready, then its request line remains 0. Any subset of these n entries can have their request lines set to 1. The job of the select unit is to choose one of these n entries, which has a ready instruction. Once that entry is chosen (or selected), the select unit sets its grant line to 1. This lets the entry know that it is ready to be sent for execution.

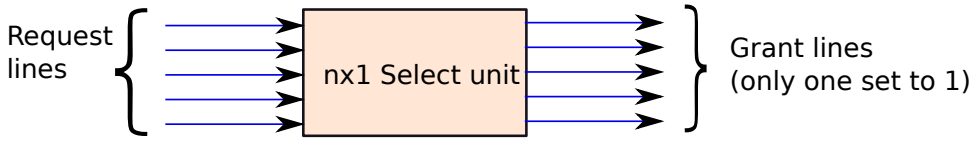


Figure 4.16: A basic select unit

Figure 4.16 basically shows a black box that takes in n 1-bit inputs (request lines), does some processing (unbeknownst to us at the moment), and then sets one of the grant lines. Let us elaborate.

Tree Based Select Unit

Let us consider one of the simplest designs for the select unit that has a tree-based shape. Please refer to Figure 4.17. Such a select unit is an $n \times 1$ select unit because we are choosing at most one out of n inputs.

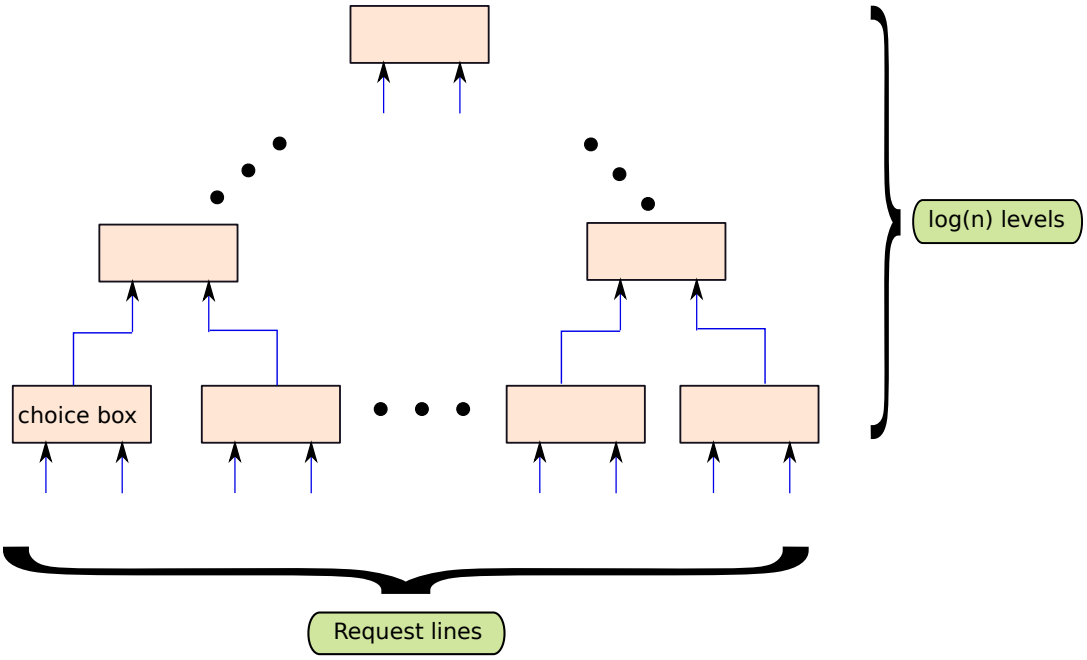


Figure 4.17: A tree based select unit

In this example, we have n request lines. At the lowest level (the leaves of the tree) successive pairs of consecutive request lines are routed to a set of $n/2$ elements. Each of these elements is a small select unit in its own right. Each such element has two inputs, and out of these, it needs to choose at most one. It is thus a 2×1 select unit. Instead of using the cryptic term, 2×1 select unit, we shall refer to these elements as *choice boxes*.

If we again take a look at Figure 4.17, we can make out that the choice boxes are organized in layers. The first layer of $n/2$ choice boxes choose a maximum of $n/2$ inputs for the next layer. In the next layer, we have $n/4$ choice boxes. They again choose at most half of the inputs as possible selections, which are forwarded to the next layer, and so on. Let us now delve into a choice box.

A choice box has two inputs (two request lines) as shown in Figure 4.18. Let us name the inputs i_0 and i_1. There are four possibilities. Either both of them are 1 (both interested), or one of them is 1 (two such possibilities), or none of them are 1. For the first case where both the inputs are 1, we need to make a choice. Let us at the moment choose one of the inputs arbitrarily. We shall discuss the policies for selection later. It is important to remember the choice. For this purpose, we can have a small state element (*choice*) inside each choice box such as a latch that remembers which input was chosen. For example, if we choose input i_0, then we store 0 in the latch, else we store 1. The choice box also has two grant lines corresponding to each input: g_0 and g_1.

Now, for the other two cases where only one input is asserted (set to 1), we choose that input. Subsequently, we set the output request line *result* to 1 thus indicating that the choice box has an input that is asserted. The output request line is an input to the next layer of choice boxes. If none of the inputs are asserted, then we set *result* to 0. This indicates to the next layer that there are no requests to be made.

Note that in every layer the number of choice boxes decreases by a factor of two (like a binary tree [Cormen et al., 2009]). We thus have a total of $log_2(n)$ levels. The final layer (the root node) has a single choice box. It chooses between its inputs and in a sense makes the final choice by asserting the corresponding grant signal for the chosen input. This information needs to propagate back to the

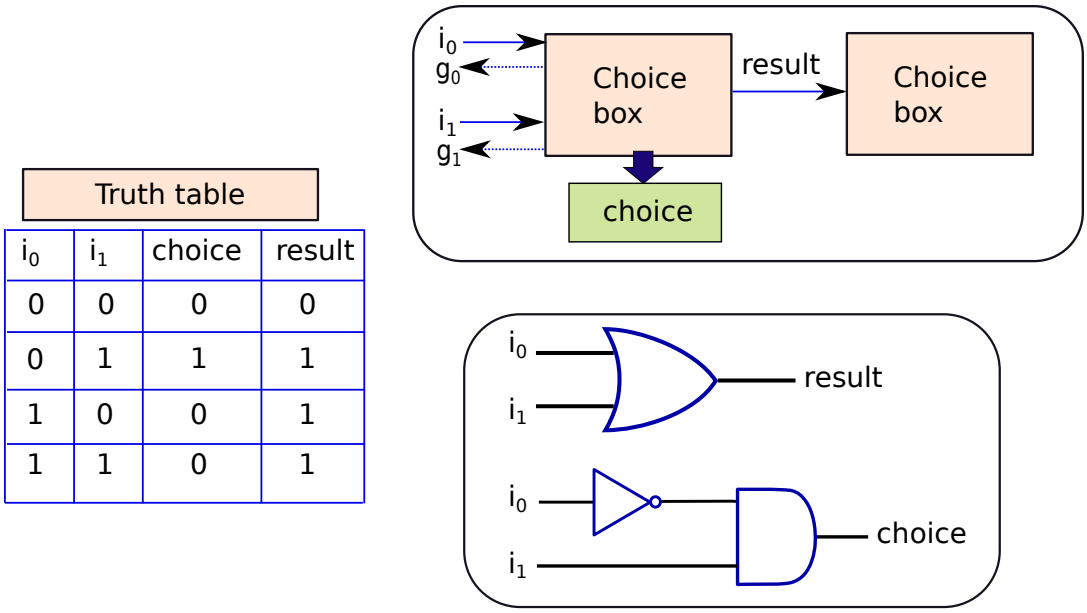


Figure 4.18: A choice box (preference given to i_0)

original entry. The reverse path is followed. In each choice box along the way we set the appropriate grant signal.

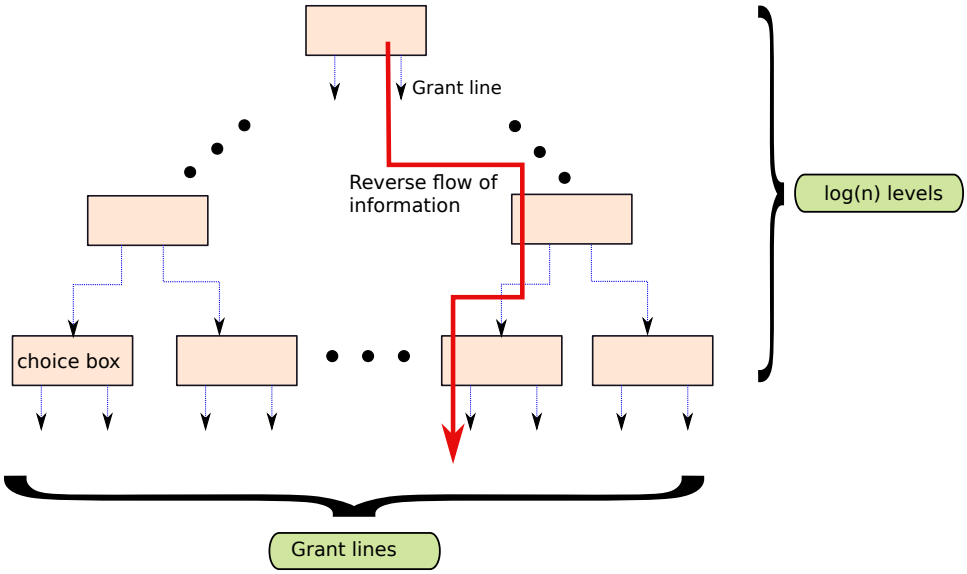


Figure 4.19: The path of grant requests

Figure 4.19 shows the path that is taken by the grant signal for one particular example. In this case, when a choice box finds that its input grant signal (coming from the root) is asserted, it finds out which input it had chosen, and asserts the corresponding grant signal. For example, if a choice box had chosen input i_1, and it subsequently finds that its input grant signal (coming from the root) is asserted, it sets

the grant line associated with input i_1 to 1.

In this manner the grant signal propagates to the selected entry in the instruction window. Once an entry receives an asserted grant signal, it knows that it has been selected and should immediately proceed for execution.

$n \times m$ Select Units

We have discussed the design of an $n \times 1$ select unit. Let us now discuss the design of general $n \times m$ select units. Note that here m is typically not a very large number. After all, it is limited by the number of functional units. m is typically 2 or 3.

We have several options for designing an $n \times 2$ select unit.

Option 1: The first option is easier but slower. Here we cascade two $n \times 1$ select units. We first select one of the inputs. Then we de-assert (set to 0) its input request line, and proceed to select a request out of the rest of the requests using the second select unit. The schematic is shown in Figure 4.20.

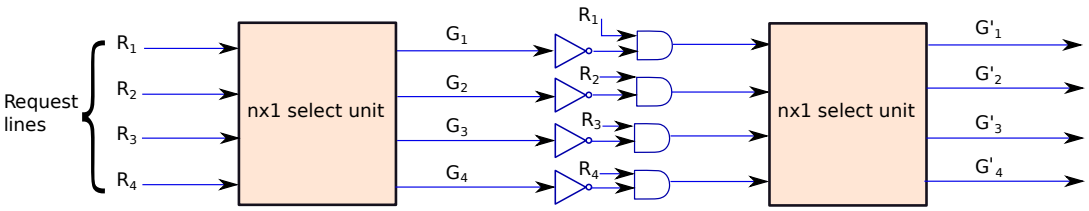


Figure 4.20: Two cascaded $n \times 1$ select units

With this design we take twice the time as a normal $n \times 1$ select unit. However, the design is simple, easy to create and understand. Note that there are issues with scalability. Designing an $n \times 3$ select unit on similar lines will be fairly slow.

Option 2: Let us now look at a slightly more direct approach. Let us modify a simple $n \times 1$ select unit to actually choose two instructions. We shall make the modification in each choice box. Each choice box now will have two 2-bit inputs (or request lines). Each input line will indicate the number of requests that have been selected in the subtree rooted at the choice box. This number can be either 0, 1, or 2. It is now possible that a choice box might be presented with four requests. Out of these, it needs to choose at most two and propagate this information towards the root of the tree. Finally, the root node will choose two requests and let the corresponding choice boxes know. This information will flow back towards the instruction window entry.

Refer to Figure 4.21 for a high level view.

Option 3: The select unit in Option 2 is complicated. There is no need to further underscore the fact that complicated units are also slow units. Let us instead divide the entries in the instruction window into disjoint sets. Each set can have an associated select unit. For example, we can divide the entries into two sets: entries at odd indices of the instruction window array and entries at even entries. We can have one $n \times 1$ select unit for each set. This strategy will ensure that we will never select more than two instructions and both the select units can act in parallel. However, the flip side is that if two entries at even locations are ready, and there are no entries at odd locations that are ready, we will only be able to select just one entry. This will lead to idleness and a consequent loss in performance. In spite of such concerns, having select entries work on disjoint portions of the instruction window is deemed to be a reasonably good solution primarily because of its simplicity.

Option 4: We can do slightly better. Let us have two select units, where each select unit is connected to all the instruction window entries. For each choice box let us refer to one of the inputs as *left* (i_0) and

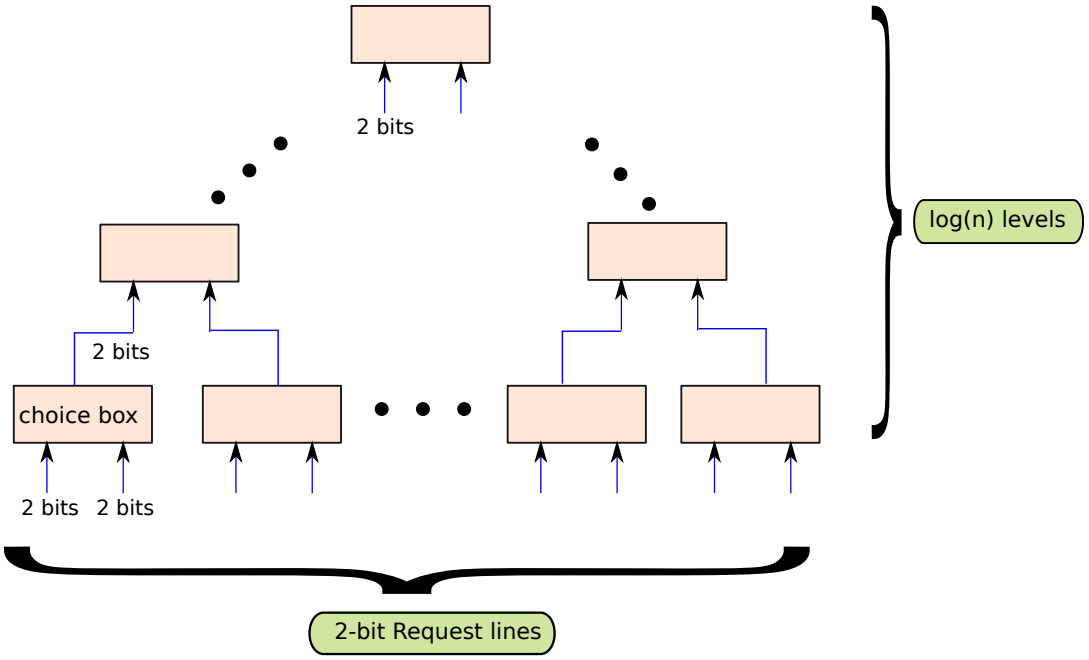


Figure 4.21: Non-cascaded design of an $n \times 2$ select unit (the grant lines are not shown)

the other one as *right* (i_1). Let us force the choice boxes in one select unit to always give a preference to their left inputs whenever there is a choice. Let us similarly force the choice boxes in the other select unit to always give a preference to their right inputs (whenever there is a choice). Let us now prove that it is never possible that the same input is chosen by both the select units when we have at least two requests – two instruction window entries that are ready.

Assume that we order the requests in a linear order from left to right. Consider any two requests R_0 and R_1. Furthermore, assume that in this order R_1 is to the right of R_0 (without loss of generality).

Consider the select unit where each choice box always prefers its right input. It is not possible for this select unit to choose R_0. It will either choose R_1 or some other request that is to right of R_1. Similarly, we can prove that the select unit where each choice box always prefers its left input chooses either R_0 or some other request that is to the left of R_0. Hence, we prove that both the select units can never choose the same request.

Important Point 6
There is a trade-off between Options 3 and 4. For the solution in Option 3, we connect half the entries in the instruction window to the first select unit and the rest half to the other select unit. However, in the more efficient solution (Option 4), we connect all the entries to both the select units. There is thus an increase in efficiency at the cost of doubling the number of connections. This is one more example of a general maxim: there is always a trade-off between efficiency and the number of resources.

Selecting Instructions for Different Types of Functional Units

The traditional solution for this problem is to have different select units for each class of instructions. For example, if we have adders and multipliers as the only functional units, then we can have one select unit for adders, and one more for multipliers. The problem of selecting instructions for these two classes of units is independent.

This approach is not scalable if we slightly complicate the situation. We can have functional units that can process instructions belonging to many different classes, or we might have a lot of classes of functional units. In this case, there is a need to complicate the select logic, where we need to send additional bits along with each request. We basically need to annotate each request with the type of the instruction. The choice boxes have to analyze the types of the requests, and make appropriate choices. This will slow down the select unit. Hence, it is wise to have a few classes of instructions and a set of functional units that can process only one class of instructions. Again, there are complex trade-offs in this case between the complexity of the ISA and the complexity of the select unit.

4.2.4 Early Broadcast

Let us begin by asking, "When should we broadcast?". A naive answer would be once we have finished executing the producer instruction. While writing to the register file, we can in parallel broadcast the tag. Unfortunately, this will lead to an extremely inefficient implementation.

Let us consider the following piece of code.

```
1 add r1, r2, r3
2 add r4, r1, r5
3 add r6, r4, r7
```

Instructions *1*, *2*, and *3* have RAW dependences between them. We have a RAW dependence between instructions *1* and *2* because register $r1$ is written to by instruction *1* and instruction *2* reads it. Similarly, there is a RAW dependence between instructions *2* and *3* (via register $r4$).

Now, let us see what happens if these instructions pass through our pipeline. Assume that instruction *1* wakes up in cycle 1. It is possible that multiple instructions might wake up in cycle 1, and we need to *select* which instructions shall proceed to execution. This will take one more cycle (cycle 2). Now in cycle 3, instruction *1* will proceed to read its operands from the register file, and in cycle 4 it will move to the execution units. Assuming it takes 1 cycle to execute the instruction, we will broadcast the tag in cycle 5, and in cycle 6 the consumer instruction will wake up. Along with broadcasting the tag, we can write the results to the register file. The chain of events is shown in Figure 4.22.

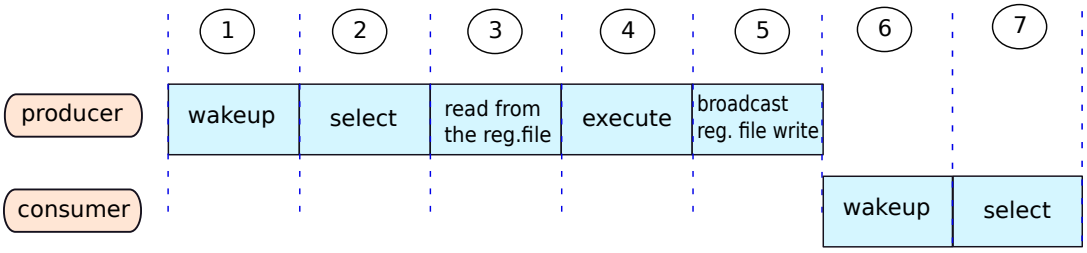


Figure 4.22: Chain of events between the execution of a producer and a consumer instruction

The main issue is that we have a delay of 5 cycles between executing instructions *1* and *2*. In other words, if instruction *1* executes in cycle 4, then instruction *2* will execute in cycle 9. Let us say that we

are able to find a lot of independent instructions between cycles 4 and 9, then there is no problem: our pipeline will always be full of instructions.

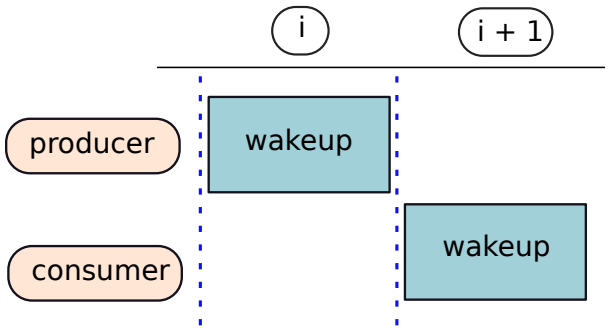
However, this need not be the case all the time. Sometimes we might not find enough independent instructions. In this case, the pipeline will not have any work to do, and our performance will dip. In fact the situation is far worse than a simple in-order pipeline where for such execution sequences we would not have stalled. Clearly, we are not getting any benefit out of an OOO pipeline.

To ensure that we are able to get some gains out of an OOO pipeline we need to ensure that such instructions with a RAW dependence can execute in consecutive cycles. This will at least ensure that we are doing as well as an in-order pipeline. The additional benefits of OOO pipelines will accrue when we find enough independent instructions to fill up the rest of the issue slots. Nevertheless, waiting for 5 cycles to issue a consumer instruction seems to be a very bad idea. Let us aim for 1 cycle, which is the minimum (same as an in-order pipeline).

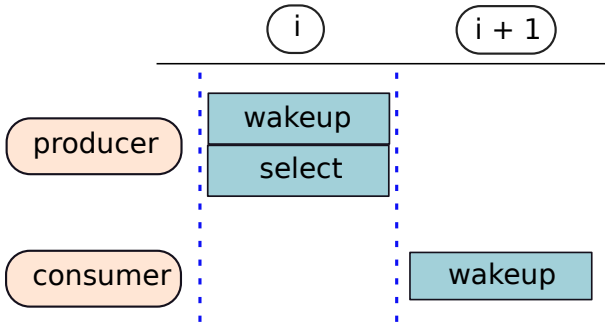
Let us thus summarize our new found objective. It is to execute instructions with a RAW dependence in consecutive cycles. Let us only confine our attention to regular arithmetic instructions, and keep memory instructions out of this discussion for the time being. Such kind of execution, known as back-to-back execution, is very beneficial and will guarantee us some degree of minimum performance, even in programs with very little instruction level parallelism (ILP) (see Definition 8).

To ensure back-to-back execution, we need to take a very deep look at three actions namely broadcast, wakeup, and select. If a producer instruction wakes up in cycle i, then the consumer instruction has to wake up in cycle $i + 1$. Before getting overly concerned with the exact mechanism, let us start drawing some diagrams to explain the process. We shall then add some meat to the bones by working on the mechanism.

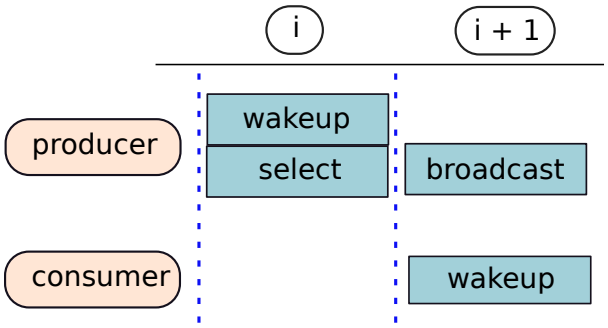
At the moment, this is what we need to ensure:



Now, given the fact that we need to perform a select operation, after an instruction wakes up, we have two options. Assume that we have slightly optimized our wakeup and select procedures such that they fit in a single cycle. In this case, the instruction can get selected in the same cycle. Assume it does get selected. We thus have arrived at the following pipeline diagram.



Once an instruction is selected, it knows that it is on its way to execution. There are no more roadblocks. It can proceed to the subsequent stages. In cycle $i+1$, it can broadcast the tag corresponding to its destination register. This is an early broadcast because we are broadcasting the tag before the producer instruction has computed its result and written it to the register file. We subsequently expect the consumer instruction to pick up the broadcasted tag (in the same cycle), and proceed through its wakeup and select stages. This is thus the final pipeline diagram:

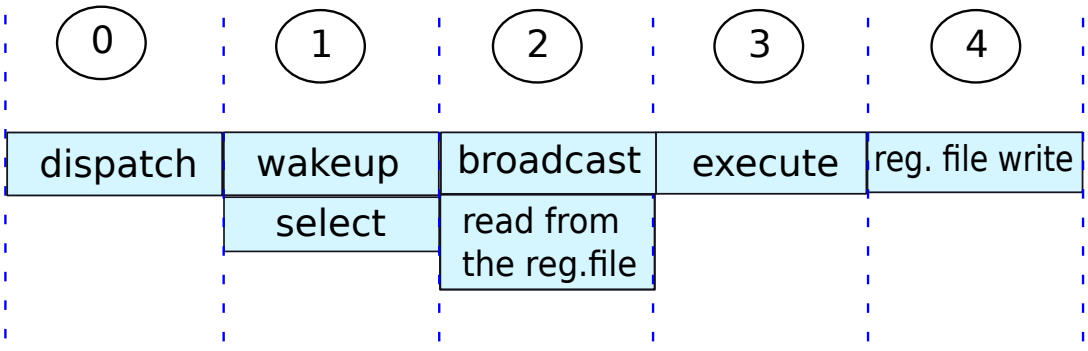


To summarize, for ensuring back-to-back execution, we have had to make significant changes to our design.

1. We have overlapped the broadcast of the producer instruction with the wake-up/select operations of consumer instructions. This requires us to ensure that these are very fast operations.

2. We do an *early broadcast*. This means that before the result of the producer is ready, we wake up the consumer instructions. They believe the producer, and proceed through the wakeup/select stages. The producer is expected to forward (or bypass) its result to the consumer instructions such that they can execute correctly. This is similar to classic forwarding in in-order processors, where the result of the producer is sent to the consumer. The consumer chooses between the value read from the register file, and the forwarded value using a multiplexer (see Section 2.1.4). **It is important to understand the forwarding technique in in-order processors before reading this section. We use exactly the same logic here**. In OOO processors forwarding is typically called *bypassing*.

3. The OOO pipeline from the dispatch stage to the register file write stage is thus as follows:

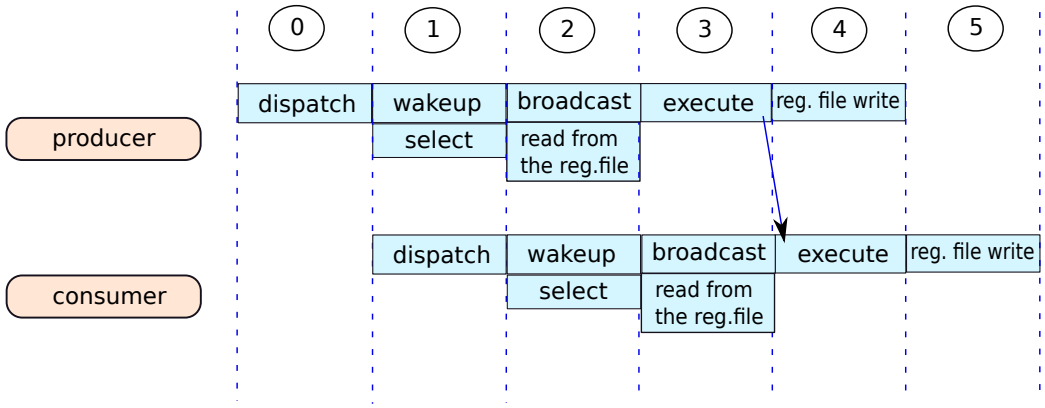


4. A question naturally arises: Is there a correctness issue in performing early broadcast? The reader should first try to answer this question on her own. The answer is given in Point 7.

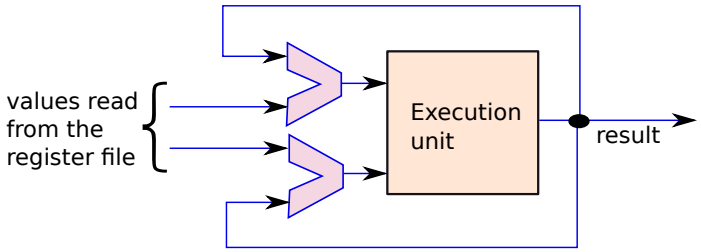
Important Point 7

We are doing an early broadcast. This means that we are broadcasting the tag before the register has been written to. Is there a correctness issue?

Let us try to use the same logic that we used while discussing forwarding in in-order processors in Section 2.1.4. Let us look at the pipeline diagrams for a producer and consumer instruction issued back-to-back.



Here, we execute a producer-consumer instruction pair back-to-back, and we do an early broadcast. There is no problem in correctness, because the consumer instruction can always get the correct values from the producer. In this case, the consumer needs the producer's results at the beginning of cycle 4. The producer's results are ready by the end of cycle 3, and thus it can forward the results to the consumer. Similar to classic forwarding in in-order pipelines we require multiplexers.



We know from our study of in-order pipelines that such kind of forwarding (also known as bypassing) can be done seamlessly (see [Sarangi, 2015] and Section 2.1.4) in most cases. All that we need to ensure is that when we need the data at the beginning of the execute stage, it is available somewhere in the pipeline. As long as we can ensure this, early broadcast will not introduce any correctness issues.

Let us now comment about the efficiency of this process. It is true that this method has enabled back-to-back execution, and thus we are guaranteed to at least get the same IPC as an in-order processor for codes that have a lot of such dependences. However, such optimizations come at a cost, and the cost is that we need to perform the broadcast-wakeup-select operations very quickly – all within one cycle (see Figure 4.23).

This might not be possible all the time, particularly when the instruction window has a large size. There are wire delays involved, and the wake-up/select operations can take more than one clock cycle particularly in high frequency processors. Hence, it might be a wise idea to forego the notion of back-

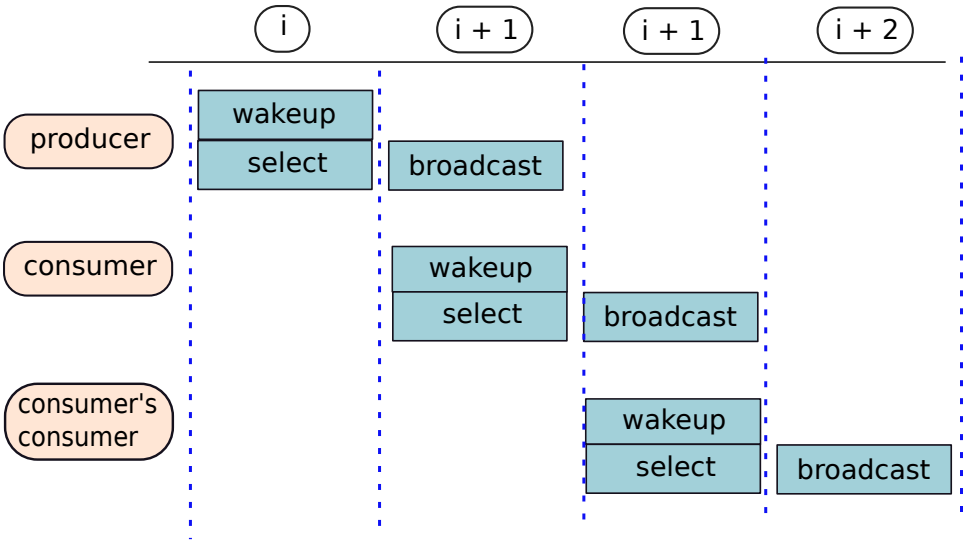


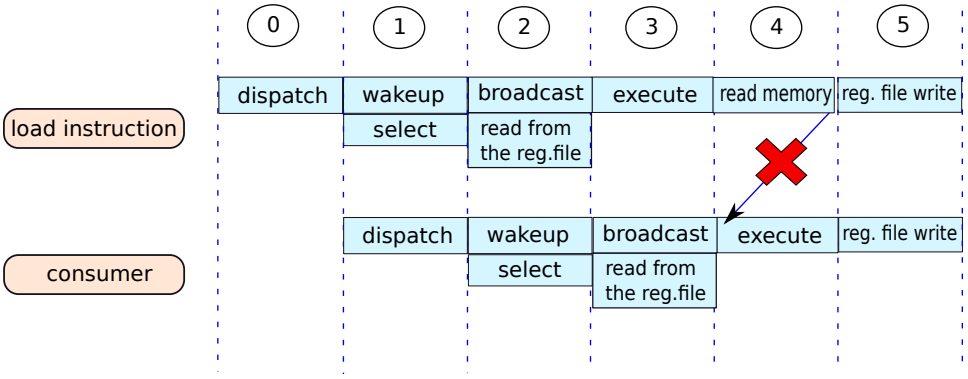
Figure 4.23: Back-to-back execution in an OOO pipeline

to-back execution if we desire a very high frequency processor. We will definitely lose IPC in codes with a lot of dependences; however for most general purpose programs we will always be able to find enough independent instructions to execute every cycle. There can be a net gain in performance because of the high frequency. Such kind of decisions illustrate the trade-off in designing high performance processors, where we cannot get high IPC and frequency at the same time. Again this also depends on the type of programs that we expect to run. If we expect that programs will have high ILP, then back-to-back execution is not a necessity, otherwise it is.

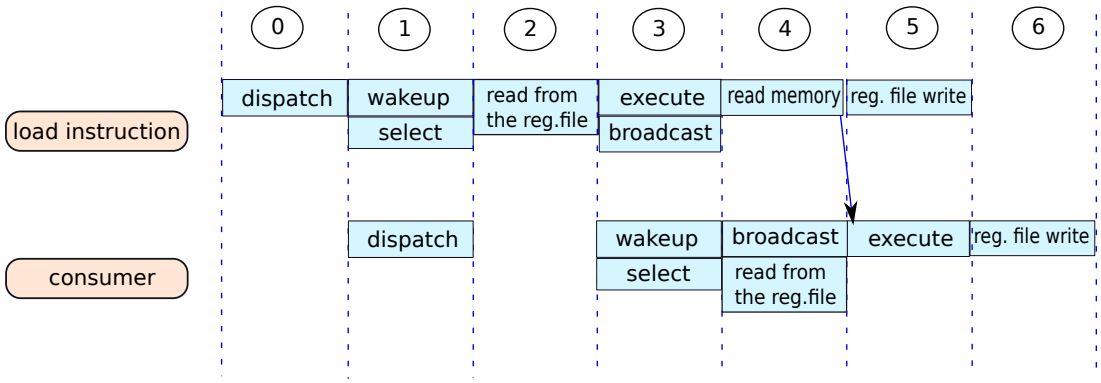
4.2.5 Tricky Issues with Early Broadcast

The Load-use Hazard

As we have discussed in Point 7, early broadcast per se is not an issue as long as it is possible to bypass values from one stage to the other. Of course, there are exceptions as we had discussed in Section 2.1.4 such as the load-use hazard, where if the producer instruction is a load, back-to-back execution is not possible. In the case of OOO processors, we have the same problem. The execute stage computes the address, and then we need to perform a memory access; hence, bypassing is not possible. Let us visualize this.



We add an extra stage after the execute stage to access memory. We make the simplistic assumption that a memory access takes one cycle. Then we try to forward the data. As we show in the diagram this is not possible. We are moving backwards in time. We need the results at the beginning of the 4^{th} cycle, whereas they are produced at the end of the 4^{th} cycle. Thus, back-to-back execution is not possible. However, we can still broadcast the tag early, and get some benefits.



Instead of broadcasting the tag right after instruction select, let us instead broadcast the tag in the 3^{rd} cycle as shown above. We thus broadcast one cycle later, and the consumer instruction needs to stall for an additional cycle. The rest of the processing for the consumer instruction remains the same. In this case, we do not have any correctness issues owing to the fact that the consumer instruction needs the result in the beginning of the 5^{th} cycle, and it gets it. This is because the load returns with its value at the end of the 4^{th} cycle.

To summarize, the method to handle a load-use hazard is to broadcast two cycles after selection. Let us generalize this. Let us club the execute and memory access stages into one large execute stage. In the current example this stage takes 2 cycles (1 for computing the address, and 1 for accessing memory). Assume it takes k cycles ($k \geq 1$). We claim that we need to do a broadcast k cycles after selecting the instruction to ensure that all the consumer instructions get the result exactly on time.

Let us prove this. Assume that the producer instruction wakes up and gets selected in cycle 1. It will then proceed to read the values of its register operands in the next cycle. Since the execution takes k cycles, the execution will finish in cycle $k+2$. Consider the next instruction. The worst case is that it is a consumer instruction. The earliest that it can execute is cycle $k+3$ (one cycle after the producer). Calculating backwards the consumer needs to wake up (and get selected) in cycle $k+1$ (one register read stage in the middle). This means that the producer needs to broadcast in cycle $k+1$, and that is the earliest. Recall that the producer had woken up in cycle 1, and we just proved that the earliest it can broadcast the tag is cycle $k+1$ (k cycles later). This proves our claim.

This means that for each class of instructions, we have different times at which we need to broadcast their tags. If an add instruction takes 1 cycle, then we can broadcast the tag immediately after the instruction gets selected (in the next cycle). However, if we have a slow divide or memory access operation, then we need to wait for k cycles. This is typically achieved by using a timer for each selected instruction that counts down from k to 0.

Setting the *avlbl* bit with Early Broadcast

We had discussed the *avlbl* bit in each entry of the rename table (see Figure 4.14). We had said in Section 4.2.1 that this bit indicates if a given source operand can be found in the register file or not. This needs to be revised in the light of our current discussion.

The *avlbl* bit should indicate if it is possible to get the value of the operand from either the register file or the bypass network (network of wires and multiplexers to transfer forwarded values). If it is so, then we can mark the operand to be ready, and the instruction can be issued if the rest of the operands

are also ready. To summarize, when we say that an operand is available, it means that its value is present somewhere in the pipeline.

Regarding when we should set the *avlbl* bit, the answer should be obvious to us now. It should be set when the tag is being broadcasted on the tag buses. This is when the consuming instructions also get to know that the value corresponding to the tag is available (either from the register file or the bypass network). Along with doing this we can just forward the tags to the rename table, and set the appropriate *avlbl* bits.

Missing a Broadcast

Consider the following sequence of operations. For instruction I, we read the rename table in cycle 1. We find that physical register $p1$ (one of the operands) is not available. Then in cycle 2 we dispatch this instruction to the instruction window. In cycle 2, the producer instruction (for $p1$) broadcasts the tag on the tag buses. If instruction I misses this broadcast because it is being simultaneously written to the instruction window, then there is a problem. The instruction window entry of I will continue to wait for the broadcast for $p1$, and this will never happen, because we have already broadcast the value in cycle 2.

Let us look at several ways to fix this problem.

- We write to the instruction window in the first half of the clock cycle, and we broadcast the tags in the second half of the clock cycle. This means that the instruction that is being dispatched (written to the instruction window) will not miss a broadcast. By the time that the tag is broadcast, the dispatched instruction is ready to wake up its operands. This is an easy solution. However, it is very inefficient. We are artificially reducing the time that we have for a broadcast and the subsequent wakeup. To accommodate this we need to elongate the duration of a clock cycle, which is not desirable.

- The other option is to store all the tags that were sent in a given cycle in a small buffer. We can compare these tags with the operands of the dispatched instructions in parallel. The *ready* bits for the operands can then be written later – either at the end of the current cycle or at the beginning of the next cycle – to the instruction window entries. This makes the circuit design complex in the sense that we need to create a separate structure to store the ready bits; however, in cases like this such complexities are inevitable.

- One more approach is to broadcast the tags that were missed once again. This will double the number of tag buses in the worst case. This can be done intelligently by broadcasting only those tags that have been genuinely missed and are needed to set the appropriate operands to the ready state. In the worst case, we need to double the number of tag buses, which is not desirable. In most cases, we can consider an average case; however, this depends on the benchmark, and is hard to predict.

4.3 The Load-Store Queue (LSQ)

We have up till now been discussing the execution of regular arithmetic instructions. Now we have acceptable solutions for renaming, selecting, and issuing instructions. Next, we need to turn our attention towards memory instructions. Sadly, we cannot use the mechanisms we have developed to process arithmetic instructions in the case of memory instructions. We need additional hardware. The most important structure that we add is known as the load-store queue (LSQ). Trivially speaking, it contains a list of load and store operations – arranged in FIFO (first-in first-out) order based on when they were fetched. Note that we shall use the terms "load-store queue" and "LSQ" interchangeably.

Let us motivate the need for an LSQ. Consider two memory instructions: *I1* and *I2*. We can have many kinds of dependences between them. Some of these dependences have been covered in previous

sections, and some are new. We have considered dependences via registers in the previous sections. However, now let us consider dependences via memory. We shall have such kind of dependences, when the addresses of these instructions are the same.

4.3.1 Memory Dependences

Let us consider four separate cases. Let *I1* and *I2* be memory instructions that access the same address. Assume that *I1* precedes *I2* in program order ($I1 \rightarrow I2$). Finally, assume that there are no intervening memory instructions between *I1* and *I2* that access the same address.

load $\rightarrow$ *load* dependence: In this case a load instruction (*I1*) is followed by another load instruction (*I2*) from the same address: *I1* is the earlier instruction and *I2* is the later instruction. We can reorder *I1* and *I2*. Of course, we shall have an issue when we consider multiprocessors (explained in Chapter 9); however, for single processors there is no issue. Furthermore, to reduce memory traffic, we need not send two instructions to memory. We can just send one instruction. For example, we can send *I1* to memory first. Then we can forward the value ready by *I1* to instruction *I2*. This will halve the memory traffic.

load $\rightarrow$ *store* dependence: In this case a load instruction (*I1*) is followed by another store instruction (*I2*) – both access the same address. We cannot reorder *I1* and *I2*. If we do that, then the load will read the wrong value. It will read the value written by the store *I2*, which is wrong. As a result these instructions need to execute in program order. Recall that this is a classic WAR dependence in memory.

store $\rightarrow$ *load* dependence: In this case a store instruction (*I1*) is followed by another load instruction (*I2*). This is a classic RAW dependence in memory, where the store instruction is the producer and the load instruction is the consumer. Here again, it is not possible to reorder the instructions. Otherwise, the load instruction (*I2*) will get an older value, which is wrong.

store $\rightarrow$ *store* dependence: In this case a store instruction (*I1*) is followed by another store instruction (*I2*) to the same address (a WAW dependence). Akin to a similar case with registers, we cannot reorder the memory accesses.

The summary of all of this discussion is that there can be RAW, WAR, and WAW dependences between memory instructions the same way we have dependences between instructions with register operands. However, the sad part is that we cannot use the same techniques that we used to get rid of WAR and WAW hazards for registers in this case. For registers, we used renaming, and built an elaborate mechanism centered around the rename table. However, renaming memory is far more expensive. We can have thousands of memory locations, and maintaining such large rename tables is practically not feasible. Moreover, the memory address is not a 4-bit quantity, it is instead a 32-bit or 64-bit quantity. Thus, a memory renaming table will require millions of entries, and thus it is not practical.

As a result, whenever we have a dependence of any form that involves a write (WAR, RAW, or WAW), we need to ensure that the memory requests are sent to the memory system *in program order*.

This is not all. We sadly have more bad news. Register dependences are clearly visible after decoding the instruction. We know about the nature of dependences by taking a look at the ids of the source and destination registers. However, processing memory instructions is a multi-step process. The first step is, of course, to read the values of the source registers. The second step is to compute the address by adding the contents of the base address register and the offset. Only after address computation do we get to know the address of a memory instruction. This address needs to be subsequently used to find dependences between memory instructions. Unlike decisions that are taken right after the decode stage (such as register dependences), and that also in program order, the addresses of memory instructions are generated out of order, and need to be handled in the order in which they are generated. This out-of-order generation of memory addresses complicates the problem of managing and tracking memory

dependences significantly.

Let us first try to solve this problem from a conceptual point of view. Subsequently, we shall propose a practical realization of our method.

4.3.2 Conceptually Handling Loads and Stores

It should be noted that a memory instruction has two execution steps. The first step is to compute the address, and the next is the memory access itself. Insofar as the former step is concerned, it is a regular arithmetic addition. The memory instruction can happily pass through the dispatch and issue stages for this specific part of its processing. Now, consider the second execution step where we need to handle memory dependences.

Let us imagine a large conceptual first-in first-out queue that contains all the load and store instructions that are currently in the pipeline; let this be our LSQ (load-store queue). Whenever, we decode a memory instruction we create an entry for it in the LSQ. At this point, we do not have any knowledge of its memory address. This will be computed after the corresponding instruction is issued, and we add the contents of the base register to the offset (embedded in the instruction). Once, the address is generated, we can write it to the corresponding entry in the LSQ.

This is when the real processing starts. Every entry in the LSQ is marked *not ready* by default. However, once its address is generated it is ready to be executed – sent to the memory system. As we shall see in Section 4.4, stores cannot be sent to the memory system as soon as their address is computed. Otherwise, we will not be able to guarantee precise exceptions (see Section 2.3.3). Hence, stores are sent when the instruction is ready to be removed from the pipeline. As we shall see in Section 4.4, this is the point where the instruction is the oldest in the pipeline.

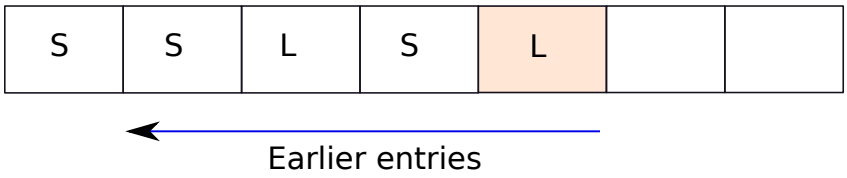


Figure 4.24: Example of a sequence of loads and stores (arranged in a queue as per the order in which they entered the pipeline).

Loads

Let us look at an entry right after its memory address is computed. We are ready to go to the memory system, if we are not violating any memory dependence as defined in Section 4.3.1. However, whether we do have dependences or not needs to be ascertained first.

Let us look at Figure 4.24 in detail. Assume that the shaded (or colored) box in the load queue (load A) just got its address computed. Let us consider all the stores before it. There is a possibility of a $store \rightarrow load$ dependence. We need to read all the store entries before it and find if there is a store with the same address or with an unresolved (not computed) address. This means that in Figure 4.24, we need to search all the stores before the load – proceed in a leftward direction (towards earlier entries). There are three possible scenarios. Let us consider them in decreasing order of priority.

1. **While proceeding towards earlier entries we encounter a store with an unresolved address.** This means that in theory the address could be the same as the load. We cannot take the risk of ignoring this store instruction. If we ignore it and go ahead, it is possible that we might load the wrong data if the address of the store comes out to be the same as the address of load

A. Hence, we need to wait for the address of that store to be resolved (computed). The process terminates here, and we do not check for the rest of the scenarios.

2. **Assume that we encounter a store to the same address before encountering an unresolved store.** In this case, we can forward the value of the store to the load instruction. This is known as *load-store forwarding* or *forwarding in the LSQ*. In this case, the load can take the value of the store and proceed. Note that the store in consideration has not written its value to the memory system yet. However, since we know the value that it is going to write, we can happily let dependent load instructions proceed. This is similar to forwarding in an in-order pipeline.

3. **None of the above:** This means that we keep searching for previous store instructions; however, we do not encounter a store with an unresolved address or a store instruction with the same address as the load instruction. In this case, there is no reason to wait or forward from an earlier store. We can let the load instruction access the memory system and read its value from there.

To summarize, we need to search all the store entries before the load instruction A (in Figure 4.24). We need to keep searching till we find the first (latest before the load) store instruction that either stores to the same address or has an unresolved address. In the former case there is a $store \rightarrow load$ dependence and in the latter case there is a possibility of a dependence, and since we do not know, we need to wait. If there is a dependence, then we can directly forward the store's value to the load instruction. We are guaranteed to have the value in the LSQ because in our assumed RISC ISA, store instructions read the value, which is to be stored, from a register. We further assume that register file reads happen before the address of a store is computed. Thus, if a store's address is resolved, its value should also be present in the corresponding LSQ entry. With such forwarding, the load can continue its execution. This method effectively increases the IPC because it releases the load instruction as soon as possible and allows it to carry on with its execution. There will be fewer stalls in the future.

If there is no waiting or forwarding, then it means that we have searched all the earlier store entries, and there is no possibility of a memory dependence. The load instruction can be sent to the memory system.

Stores

Let us see what happens in the case of store instructions. Note that here there are two distinct points of time. The first point of time is when the store instruction is decoded, and we create an entry for it in the LSQ. The second point of time is when we finish computing the address of the store instruction and update the address in the LSQ entry. We are assuming that at this point we know the value to be stored (contents of some register) as well. This is because we read the contents of the register that contains the base address and the contents of the register that holds the store value at the same time. Given that we have finished computing the address, the store value must also be present with us.

Now that we know the value that needs to be stored, and the address, a naive reader would think that we are ready to send the store to the memory system. However, as we shall see in Section 4.4, because of several reasons centered around correctness we can only send a store to the memory system when the instruction is being removed from the pipeline, and it is the oldest instruction in the pipeline. As long as we have earlier (older) instructions in the pipeline, we cannot send the store to memory. Since we do not have earlier instructions when the store instruction is sent to memory, it is guaranteed to be at the head of the LSQ, and there will be no memory dependences that can stop us from sending the store to memory.

Nevertheless, handling stores is not that simple. Initially, when an entry is created for a store at the time of decoding an instruction, the store's address is unresolved. Let us refer to this situation with a question mark (?) in our figures. Once its address is computed, the store's address is resolved. We shall use a tick mark to indicate this situation. Even though we cannot send the store to memory immediately after computing its address, we still have some work to do. Given that we have a store with a resolved

address, new dependences will be created. The store can forward its value to newer loads as we can see in Figure 4.25. In this figure all the entries that are shaded (or colored) have the same address.

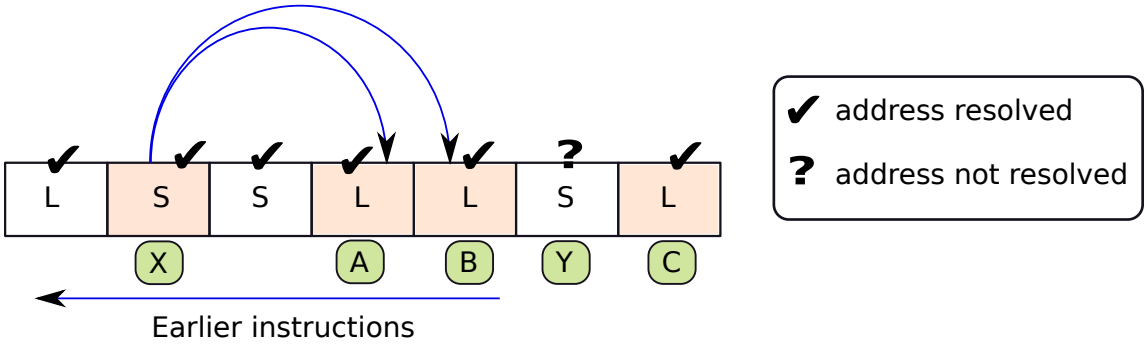


Figure 4.25: Forwarding in the LSQ (we cannot forward to C because of Y)

When store X in Figure 4.25 is resolved, suddenly loads A and B, which have the same address are eligible to get the forwarded data. They can take the forwarded value and continue their execution. However, load C is not eligible to get the forwarded value because it is preceded by the store instruction Y. Y's address has not been resolved, and it is possible that its address might be the same as X's and C's address. In that case, C should get the forwarded value from Y and not X. Since we do not know, load C needs to wait.

The algorithm is thus as follows:

- Search later entries. If we encounter a store to the same address or if a store is unresolved, then stop.
- Otherwise, if there is a load with the same address, forward the value, and then keep scanning later entries.

4.3.3 Design of the LSQ

Most common designs of the LSQ have a separate load queue and a store queue as shown in Figure 4.26. The queues by themselves are internally designed as circular queues, which as we have been seeing is almost always the case. Entries are added to the bottom of the corresponding queue (load or store), and they gradually move up. The logic for having two separate queues is efficiency. We typically need to find all the earlier stores, or later loads. Having smaller queues helps speed up the process.

Let us summarize the discussion that we had in Section 4.3.2 regarding the search operations that need to be performed (see Table 4.2) along with the conditions for terminating the search. Whenever we resolve the address of a load, we search all the stores before it till we find a store to the same address or a store with an unresolved address. In the former case we forward the value of the store to the load.

Whenever, we resolve the address of a store instruction we search all the loads and stores after it. We terminate the process when we encounter a store with the same address or an unresolved address. If we encounter a load to the same address before termination, then we forward the value of the store to the load, and mark the load instruction as ready.

The task now is to design basic hardware mechanisms to implement the logic in Table 4.2. Here are the basic primitives that we need to implement.

1. Search entries before or after a given entry.
2. Find entries with the same address.

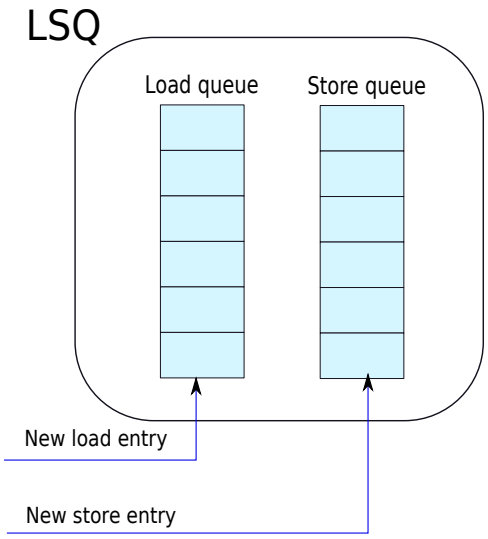


Figure 4.26: Design of the LSQ

Type	Search direction	Action: Condition
Loads	Search all stores before it	1. Terminate and forward value: Store to the same address 2. Terminate: Store with an unresolved address
Stores	Search all loads and stores after it	1. Terminate: Store to the same address 2. Terminate: Store with an unresolved address 3. Forward value: Load from the same address

Table 4.2: Rules for processing load and store entries after their address is resolved

3. Find the first (latest) entry before a given entry that satisfies a certain condition, or the first (earliest) entry after a given entry that satisfies a given condition.

Let us add the following fields to each entry (see Figure 4.27). Along with the address we create two additional fields: *valid* and *resvd*. *valid* indicates if the entry is valid (as the name suggests) and *resvd* indicates if the address has been resolved or not. If $resvd = 1$ then it means that the address has been computed (resolved).

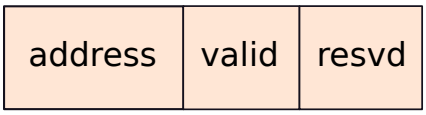


Figure 4.27: An entry in the LSQ

Finding Earlier or Later Loads and Stores

The basic question is, "Given a load or store how do we find the loads and stores before or after it ?" Since the load and store queues are implemented as circular queues, we have a *head* and a *tail* pointer associated with each queue. Every time we enqueue an entry, we increment the *tail* pointer (modulo the

size of the queue). Similarly, every time we dequeue an entry, we increment the *head* pointer (modulo the size of the queue).

Let us create an abstract version of the problem. Let us consider an array v, which we shall use as a circular queue with a *head* and a *tail* pointer. Additionally, we have a *size* field that indicates the number of entries in the queue. If $size = 0$, then it means that the queue is empty, and no processing needs to be done. Let us proceed with the assumption that $size \neq 0$. If $head = tail$, then it means that the queue has just one entry. If $tail < head$, then it means that the queue has wrapped around the end of the array. Here, we are not considering the possibility of overflows: more entries than the maximum size of the queue. Finally, note that we do not look at the *head* and *tail* pointers to decide whether a queue is empty or not, we simply look at the *size* field.

To record the relative ordering of loads and stores, we need to create a mechanism. We shall describe the mechanism for loads in this section. There is an analogous mechanism for stores. For each load entry we record the value of the *tail* pointer of the store queue when the load instruction was entered into the load queue. Recording the *tail* pointer of the store queue will help us find all the stores that came after the load, and the stores that were in the pipeline present before it.

Given an index j, we need to find all the entries that are either before it (towards *head*) or after it (towards *tail*). The assumption is of course that j is a valid index in the circular queue. The most trivial solution is to start at j and walk the array sequentially. This is a slow operation and is proportional to the size of the array in the average case. This is not acceptable. We need a parallel implementation.

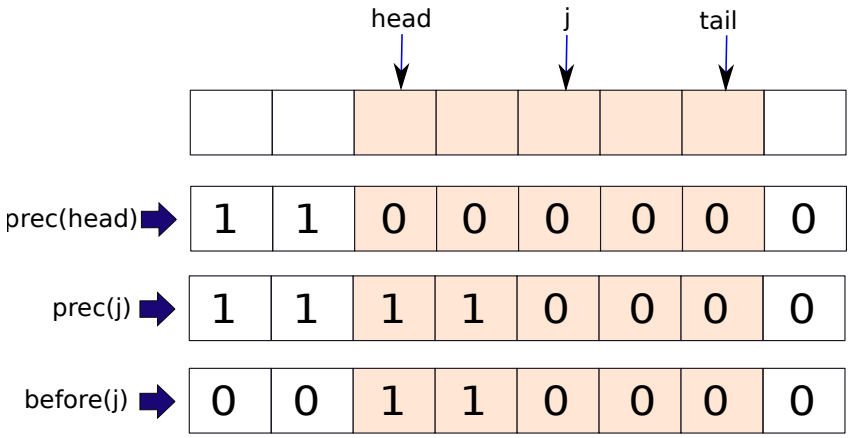
Let us first fix the format of the output. Since the load queue or store queue typically have less than 64 entries, we can use this fact to significantly speed up operations by using some extra space. Let the output be a bit vector, where the number of bits is equal to the number of entries in the relevant queue (array v). A bit is 1, if the entry satisfies the predicate (before or after), otherwise it is 0. To create an efficient hardware implementation, we can store these bit vectors in registers. Additionally, let us assume that we have the following bit vectors available in registers: *valid* and *resvd*. The *valid* register contains all the *valid* bits of entries in the relevant queue. On similar lines we have a *resvd* register that contains one bit for each entry in the queue: 1 if the address is resolved and 0 if it is not.

Let us now show an example of how to compute a bit vector that contains a 1 for all the valid queue entries that are *before* a given j. Let the total number of entries in v be N. Let us first create a small N-entry array, where for a given index i (starts from 0) we store its unary representation using N bits. For example, when $N = 8$, we store 00000111 for 3. Basically, for a given number i, we store an N-bit number where the least significant i digits are 1 and the rest are 0. If $i = 17$, we store a number whose least significant 17 bits are 1, and the remaining $N - 17$ bits are 0. We can alternatively say that for each i we store $2^i - 1$. Such small arrays that store the precomputed results of functions (unary representations in this case) are known as lookup tables. These are really fast. In this case, let us represent this lookup table operation by the function *prec*.

First, consider the case where $j \geq head$ (no wraparound). The bit vector representing the elements before j – represented as $before(j)$ – is given as follows:

$$before(j) = \overline{prec(head)} \wedge prec(j) \tag{4.1}$$

This can also be viewed graphically. The first term captures all those entries that do not precede the *head* and the second term captures all those entries that precede entry number j. Note that in this diagram the least significant bit is the leftmost position and the most significant bit is the rightmost position (we go from left to right unlike the conventional way: right to left).



An analogous equation for the case when there is a wraparound ($j < head$) is as follows.

$$before(j) = \overline{prec(head)} \vee prec(j) \tag{4.2}$$

The first part, $\overline{prec(head)}$, captures all the entries from $head$ till the last $(N-1)^{th}$ entry of the array. The second part, $prec(j)$, comprises all the entries that are between the 0^{th} and $(j-1)^{th}$ indices in the array. The proof of this case is left as an exercise for the reader.

Equations 4.1 and 4.2 are very easy to compute. They require simple logical operations and the bits can be processed in parallel. We do not have to sequentially scan any array. If we want to find all the resolved entries that are before a given index j, then we just need to compute $before(j) \wedge resvd$.

The corresponding equations for the function $after$ are as follows. The reader is encouraged to verify their correctness.

Case $j \leq tail$:

$$after(j) = \overline{prec(j)} \wedge \overline{map(j)} \wedge (prec(tail) \vee map(tail)) \tag{4.3}$$

Here, we use a function $map(i)$, which computes an N-bit bit vector where the i^{th} bit is 1, and the rest of the bits are 0. Here, the first two terms set all the bits in the range $[j+1, \ldots, N-1]$ to 1. The last term $(prec(tail) \vee map(tail))$ computes a bit vector that has 1s at all the positions in the range $[0, \ldots, tail]$. The intersection gives us the correct result.

Case $j > tail$:

$$after(j) = (\overline{prec(j)} \vee prec(tail) \vee map(tail)) \wedge \overline{map(j)} \tag{4.4}$$

Note: In both the cases we are computing a logical AND operation with $\overline{map(j)}$ because we want to remove the j^{th} entry from the result of the $after$ function. The rest of the proof is straightforward.

Finding Entries with the Same Address

To solve this problem, we need to implement the queues using a content addressable (CAM) array. We shall learn more about CAM arrays in Chapter 7. In such arrays we can access an entry both by its index and by its content. For example, if we designate the content to be the memory address, then we can search for all the entries that contain a matching address. The output will be an N-bit bit vector, where a value of 1 in the k^{th} position indicates that the address matches with the k^{th} entry in the array.

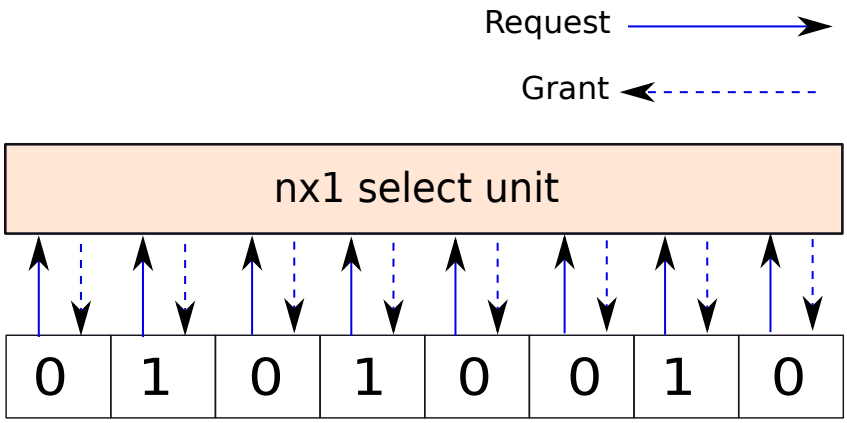
For small arrays we can get this bit vector in less than a cycle, and this can then be used to compute functions of the form: find the loads after a given store with a matching address.

Finding the Earliest or Latest Entries Satisfying a Given Property

Now that after we have computed the $before$ and $after$ functions, we can find the set of all the loads and stores before or after a given entry satisfying a certain property. For example, we can find all the stores after a certain store whose address is unresolved. We will get a bit vector as an output. As we saw in Table 4.2, it is often necessary to find the first entry before or after an entry that satisfies a given property. For example, for a load, we need to find the latest store before the load that has a matching or unresolved address.

Let us generalize this problem, and propose a solution. We have a bit vector where the entries of interest are set to 1, and the rest are set to 0. This bit vector has been computed after a series of logical operations. Each entry with a 1 satisfies a given property. Now, if $tail \geq head$, then there is no wraparound, and we are fine. However, if $tail < head$, then we have a wraparound. In this case, let us create a modified bit vector that contains all the bits from $head$ to $tail$ in sequence. This needs a set of bit shift and copy operations that can be performed in less than a clock cycle. At the end of this operation our modified bit vector, v', will not have a wraparound.

We thus have a bit vector with a set of 0s and 1s. The aim is to find either the position of the leftmost or rightmost 1. We have seen such problems before. This is similar to an $n \times 1$ select operation, as we have seen in Section 4.2.3. We program the choice boxes as follows. If we want to choose the leftmost bit that is a 1, then we need to program the choice boxes to always choose their left input if there is a choice (when both inputs are 1). We can argue by mathematical induction that ultimately the leftmost 1 will be chosen as the output of the select unit. On similar lines we can find the position of the rightmost 1. Let us visualize this.



The request lines carry 1 if the corresponding bit is 1. Only one of the grant lines is set to 1, and this corresponds to the entry that is the leftmost or rightmost 1. We can thus very easily find the earliest or latest entries using a modification of the classic select unit that we have studied before.

Putting it all Together

The crucial insight that we can derive from this design is that it is often very expensive to process data one entry at a time. There is a need for parallel processing, and this requires the use of novel storage structures and algorithms. We often end up doing extra work; however, in high performance processors it is essential to do this additional work such that results can be computed in less than 1-2 cycles.

4.4 Instruction Commit

This is the last stage in an instruction's life. This stage is known as instruction commit, and is even known as the *instruction retirement* stage. The instruction is supposed to *logically complete* in this stage,

and then subsequently leave the pipeline. The issue of an instruction logically completing needs to be further explained. It is a difficult concept and will require several pages of text before readers can fully appreciate what exactly this term means. It would be wise to go over Section 2.3.3 on precise exceptions once again before reading this section.

4.4.1 Notion of Precise Exceptions and In-order Commit

Let us take a second look at this topic (refer to Section 2.3.3). A precise exception was defined as a mechanism where all the instructions before (in program order) the faulting instruction execute completely, and none of the instructions after the faulty instruction appear to execute. We argued that having precise exceptions is vitally important for modern out-of-order processors. Otherwise, programs will appear to execute incorrectly, and we will not be able to reason about them.

A common example that is given in this regard is the case of an instruction that accesses a piece of data that is not in memory (a page fault defined in Section 7.2). The CPU and the operating system do the following in response: (1) store the *execution state (context)* of the program and load the operating system, (2) execute a small function called an interrupt handler (part of the OS) that will bring the data into memory, (3) load the execution state of the original program back, and finally, (4) re-execute the faulting instruction. This process is shown in Figure 4.28. The interrupt handler loads the data into memory and then the OS restarts the original program. The original program re-executes the faulting instruction once again. This time it does not encounter a fault because its data is in memory. The entire process needs to happen seamlessly without the original program perceiving the interruption.

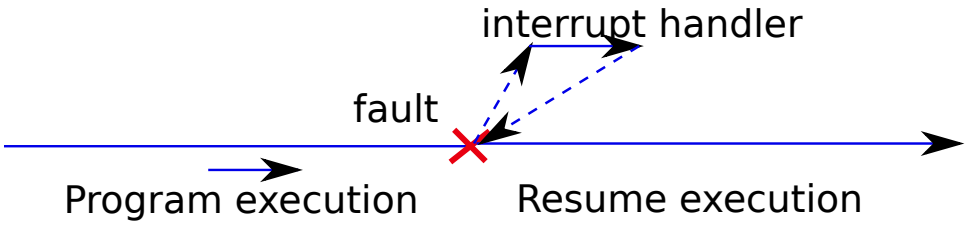


Figure 4.28: Life cycle of a faulting instruction

Let us look at the finer points. We are making an implicit assumption that before the faulting instruction all the instructions have fully executed – computed all their results, and written the results to the register file or memory. We are also assuming that no instruction after the faulting instruction has written its final result to the register file or memory. Basically, no permanent or visible changes have been made by instructions after the faulting instruction. We have in a sense *cleanly split* the execution of the original program, somehow stored its context (or execution state), executed other programs, and then restarted the same program magically from the same point. Such exceptions or interruptions in program execution are known as *precise exceptions*.

We further argued in Section 2.3.3 that to an outsider, a program should appear to execute in exactly the same way as if it was running on an in-order processor or a single-cycle processor. In such processors, precise exceptions are guaranteed because instructions write their results to the register file or main memory in program order. We can thus stop the program at any point fairly easily, flush the pipeline, and safely restart from either the faulting instruction (if there is a need) or from the instruction that appears after the faulting instruction in program order – depending upon the nature of the fault. For an out-of-order pipeline, to ensure similar behavior, it will take more work.

Let us complicate the situation by adding a little more complexity. Till an instruction completes, we are not really sure if that instruction has any faults or not. It is possible that it might access an illegal address, perform some illegal arithmetic operation, have an illegal opcode, or do something else that is not allowed. Since we do not know sufficiently in advance which instruction will have a fault,

it is a good idea to assume that any instruction might have a fault. This means that at any point in the execution of an instruction we might encounter an error – we still need to ensure that the notion of precise exceptions is maintained.

The crux of our discussion is thus as follows: an OOO processor should appear to be executing instructions in program order to an outsider. This idea can be visualized better in Figure 4.29.

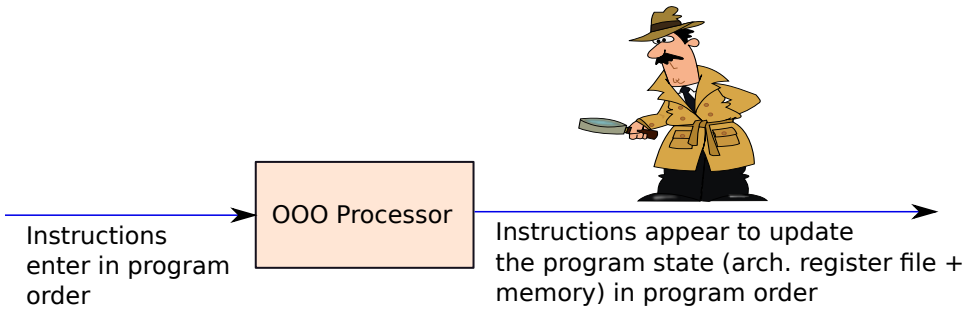


Figure 4.29: View of instruction execution from the point of view of an outsider

If we create a system where instructions are read in program order (this is already the case), and to an outsider sitting outside the processor, the instructions appear to complete also in program order, then we can achieve all our goals. This is shown in Figure 4.29. Within the processor, which we treat as a black box in the figure, instructions can compute their results and even write results to temporary storage out of order. However, for a hypothetical entity sitting just outside the processor, all instructions need to appear to make permanent changes to the register file and memory in program order.

Note that this is a stronger property than precise exceptions. If we can ensure this property, precise exceptions are automatically guaranteed, because now all instructions appear to in some sense *finish* or *complete* in program order. This model is followed by almost all processors today, and as we shall shortly see, there are no significant performance penalties.

This mechanism is known as *in-order commit*, which means that instructions finish and commit (permanently write) their results in program order. Let us try to design a hardware structure that ensures in-order commit. This is known as a Reorder Buffer. The notion of committing an instruction will become gradually clear over the next few sections.

4.4.2 The Reorder Buffer (ROB)

The Reorder Buffer or ROB is a queue of instructions. Like all our structures it is also designed as a circular queue in hardware. After an instruction is decoded, we create an entry for it in the ROB. Since instructions are decoded in program order, they are also inserted into the ROB in program order. Let us create a very basic ROB entry that contains the following fields: program counter of the instruction, the next PC, the type of the instruction, and a $finished$ bit (initialized to 0).

Here, the next PC is either the branch target if the instruction is a branch or is the address of the next instruction in program order. If the instruction has completed its execution, and has computed its values, then we can set the $finished$ bit to 1. After this we need to commit (or retire) the instruction, which means that at this point a hypothetical observer sitting outside the processor should be able to conclude that the instruction has fully finished its execution in the OOO pipeline and needs to be removed. After committing an instruction (in program order) we can remove it from the ROB and all other architectural structures. The claim is that we will be able to ensure precise exceptions with the help of the ROB. We will gradually realize this.

Definition 22

- *Immediately after committing (or retiring) an instruction, a hypothetical observer sitting outside the processor can conclude that the instruction has fully finished its execution in the OOO pipeline and needs to be removed. After committing an instruction we can remove it from the ROB and all other architectural structures.*
- *We commit instructions in program order such that precise exceptions are guaranteed. Moreover, to guarantee precise exceptions we need to ensure that no instruction makes permanent changes to the memory or the architectural register file before committing. Think of the point of committing an instruction as a point of no-return for that instruction.*
- *The* commit width *is defined as the maximum number of instructions that a processor can commit per cycle.*

The process of committing an instruction is very simple (at a high level). Let's say that we want to commit (or retire) four instructions in a cycle. This is also referred to as the *commit width* (defined as the maximum number of instructions that we can commit per cycle). We take a look at the $finished$ bit of the earliest instruction (head of the ROB). If its $finished$ bit is 1, then we can commit the instruction, and remove it from the ROB. Removing an instruction from the ROB, implies that we remove it from all other structures like the load-store queue and instruction window. The instruction is deemed to be removed from the pipeline at this point.

Then we move to the next instruction in the queue, and try to commit it. We stop when we either find an instruction that is still executing ($finished$ bit set to 0), or when we have successfully committed κ instructions, where κ is the commit width. Ideally, if we are able to commit κ instructions every cycle, we have fully saturated the pipeline because the IPC will become equal to κ. However, life is never that ideal. Because of dependences and misses in memory, most processors typically have an IPC that is much lower than their commit width.

A ROB typically has anywhere between 100-200 entries in a modern OOO processor. If we aggressively fetch instructions, it is possible that the ROB might fill up. Recall that we can commit an instruction and remove it from the ROB only if all of its earlier instructions (in program order) have finished. It is thus possible for one instruction to block a lot of instructions after it. This can happen for many reasons such as a miss in the L2 or L3 cache. Since we have accepted in-order commit as the paradigm that we shall use, there is nothing that can be done in such a situation. If the ROB fills up, we should stop fetching instructions, and wait till there is space created in the ROB. This is thus a method to apply back-pressure on the decode and fetch stages such that they stop reading and fetching instructions.

We need to ensure that before an instruction commits, its results are not permanent, and after the instruction commits, its results become permanent. Along with this there are additional things that we need to do while committing an instruction such as releasing resources and some additional bookkeeping.

Thus, there are two aspects to instruction commit – releasing resources and moving computed results to some form of permanent storage. Once both of these tasks are done, the instruction can be removed from the pipeline, and simultaneously from all the structures within the processor.

Let us look at releasing resources and doing bookkeeping. Subsequently, we shall look at methods to move computed results to some form of permanent storage, and restoring state to recover from faults, interrupts, and exceptions.

4.4.3 Releasing Resources and Bookkeeping

Arithmetic and Logical Instructions

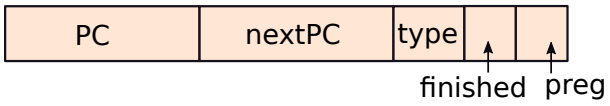
Let us consider an instruction *I1* of the form: *add r1, r2, r3*. Here, $r1$, $r2$, and $r3$ are architectural registers. Let us map these to physical registers. Assume that the instruction gets converted to *add p1, p2, p3*, where $r1$ is mapped to $p1$, $r2$ to $p2$, and $r3$ to $p3$. The question that we had asked in Section 4.1 was, "When do we release physical registers and add them to the free list?" At that point we did not have an answer; however, we are in a position to answer this question now.

Let us assume that before instruction *I1* was renamed, $r1$ was mapped to the physical register px. After instruction *I1* got renamed, $r1$ got mapped to $p1$. We need to figure out when we can add the physical register px to the free list. Before answering this question, let us keep the following points in mind.

- Assume that $r1$ was mapped to px by instruction *Ix*. At a later point, $r1$ got mapped to $p1$ by instruction *I1*.
- *I1* is overwriting the value of $r1$ written by *Ix*.
- All the instructions that use the value of px written by *Ix* are between *Ix* and *I1* in program order.
- This means that once *I1* is ready to commit, all the instructions before it in program order have committed. It further means that there is no instruction in the pipeline that is going to use the value of $r1$ written by *Ix*. This is because all such instructions are before *I1* in program order, and all of them have committed.
- When we are ready to commit *I1*, there is no instruction in the pipeline that needs the value of $r1$ written by *Ix* (via the physical register px).
- We can thus release px at this point of time.

This reasoning clearly establishes that when we are committing instruction *I1*, we can release the register px. Here "releasing" means that we can return px to the free list such that it can be assigned to another instruction.

Let us now outline what we need to do to enable this mechanism. Whenever, we are renaming an instruction such as *I1*, we are creating a mapping. In this case we are mapping the architectural register $r1$ to the physical register $p1$. We need to remember the previous mapping, which is $r1 \leftrightarrow px$. The id of the physical register px can be stored in the ROB entry for *I1*. Thus, the structure of an ROB entry is as follows.



The new field *preg* contains the id of the physical register that was previously mapped to the destination register. Once, the instruction is ready to commit, we can release the physical register *preg* and return it to the free list. This ends the life cycle of the physical register *preg*.

Important Point 8
A physical register is released (returned to the free list) after the instruction that overwrites its corresponding architectural register is ready to commit.

Branch Instructions

As we discussed in Chapter 3, we predict the direction of branches, and then we start fetching instructions from the predicted path. There is definitely a possibility of misprediction. The fact that we have mispredicted the branch will be discovered when we are executing the branch. For example, if it is a conditional branch we need to compare the value of a register with some value (typically 0). This will be done in the execute stage and the result of the comparison will indicate if the branch has been mispredicted or not.

If the branch has been predicted correctly, then there is no problem. However, if we discover that the branch has been mispredicted, then we need to treat this event as a *fault*. The instructions fetched after the mispredicted branch are on the *wrong path*. Their results should not be allowed to corrupt the program state. This is not different from an exception, where a given instruction leads to an error.

We can thus add another bit in each ROB entry called the *exception bit*. If a branch is found to be mispredicted, then we set the exception bit of its ROB entry to 1.

We proceed as usual and keep committing instructions till the mispredicted branch reaches the head of the ROB. At that point the commit logic will find out that the instruction's exception bit is set to 1. This means that all the instructions after it are on the wrong path and should not be executed. The commit logic needs to discard the branch instruction, and all the instructions after it by flushing the pipeline. In this case, flushing the pipeline means that all the structures of the pipeline are cleared. This includes the ROB, instruction window, and LSQ. We can then start execution from the mispredicted branch instruction. Since we know the direction of the branch, we need not do a prediction once again. Instead, we can use the direction of the branch to fetch the subsequent instructions and resume normal execution.

Of course, whenever there is a pipeline flush it is necessary to ensure that none of the instructions on the wrong path have written their results to permanent state. This is a separate issue and will be tackled in Section 4.4.4.

This mechanism can be used to process other events such as interrupts, exceptions, and system calls [2]. Whenever, we receive an interrupt from a device, we can mark the topmost instruction in the ROB by setting its exception bit. Then, the processor can flush the pipeline and load the interrupt handler. Similarly, if there is an exception such as a division by zero or an illegal memory access, then we can mark the instruction by setting its exception bit. Likewise, for system calls (asking the OS to intervene by suspending the current program), we can mark the instruction invoking the system call. When these instructions reach the head of the ROB, the processor will simply flush the pipeline, and then take appropriate action.

In such cases the next PC field of the ROB entry needs to be used. Recall that this field is set as either the branch target for a branch or the address of the next instruction for a non-branch instruction. We always keep track of the next PC field of the latest committed instruction. Now, let's say that the instruction at the head of the ROB has its exception bit set. Then it means that this instruction should not be committed. We thus flush the pipeline at this stage and store the "next PC" of the previously committed instruction in the context of the program.

Load Instructions

In our system, load instructions can get their value from earlier store instructions in the LSQ or can get it from the memory system. For the purpose of committing the instruction, we can treat a load instruction as a regular arithmetic or logical instruction with a destination register. Akin to arithmetic and logical instructions we remember the previous mapping of the destination register. We release the previously allocated physical register when the load instruction commits.

[2] A system call is a special instruction that allows the programmer to generate an exception. This mechanism is typically used to invoke routines within the operating system.

Store Instructions

Handling store instructions is tricky. This is because they directly make changes in the memory system – these are permanent changes. Hence, we cannot send a store to the memory system unless it is guaranteed to commit. We do not know in advance if a store instruction is guaranteed to commit or not. This will only be known when we are ready to commit a store instruction.

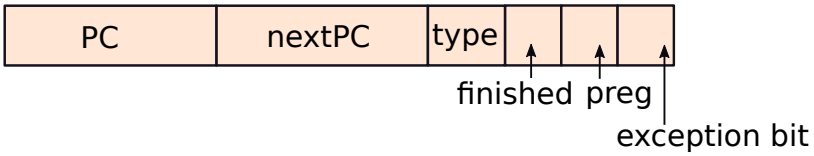
Hence, most processors send a store to memory only at commit time. Once the address of a store is resolved, they try to forward its value to load instructions that appear later in program order (see Section 4.3.2). However, they do not send the store to memory. Once the store instruction reaches the head of the ROB, it is sent to the memory system.

Let us analyze the pros and cons of this discussion. A clear disadvantage is that we keep a store instruction in the LSQ even after its address is resolved. We can in principle increase the IPC if we send the store instruction to the memory system as soon as its address is resolved. However, if we do this it will be impossible to guarantee precise exceptions.

However, the silver lining in the dark cloud is that we don't have to wait for the store instruction to finish the process of writing to the memory system. We just need to hand it over to the memory system. We shall discuss this issue in great detail in Chapter 9 and figure out when we can just hand over a store to the memory system, and when we need to wait for it to finish the write. The good news for us is that in most practical systems, we can simply hand over the store instruction to the memory system and proceed. This will not cause significant issues in performance.

Way Point 3

The final structure of each ROB entry is as follows.



With an ROB, the context or the execution state or the program state is defined as the values of all the architectural registers, the contents of the memory, the PC and the next PC.

4.4.4 Checkpointing and Restoring the Program State

We have discussed the bookkeeping aspect of the commit process. Now, we need to discuss the correctness aspect. We need to ensure that the relevant program state (next PC, registers, and memory) is updated in program order. Precise exceptions are a direct consequence of this property. However, this is easier said than done. Let us consider a branch misprediction. It is possible that we fetch tens of instructions after we mispredict a branch. It is further possible that most of these instructions go through the renaming and register write stages before we detect that the branch has been mispredicted in the execute stage. In other words, when the branch is about to be committed, we shall have a lot of instructions in the wrong path that have expressed themselves by consuming rename table entries, and by writing to physical registers. This does not augur well for our aim of ensuring that only committed instructions write to permanent state – registers and memory.

Recall that we had discussed in Section 4.4.3 that we can stop uncommitted data from going to memory by allowing store instructions to update the memory system only after they have committed. We

did not create any such mechanism for instructions with register destinations such as ALU instructions and loads; hence, we have this problem.

Let us clarify that we define the state of a program at any point of time as the state of the program after the last committed instruction. Let this be defined as the *precise state* or the *committed state*. It should be possible to pause the program, run some other program, and then restart the original program from this point.

Let us quickly recapitulate what we know and what we need to know.

1. For each instruction we record the current PC and the next PC in its ROB entry. Whenever we flush the pipeline we always keep a record of the next PC of the latest committed instruction. We resume the execution of the program at this point.

2. We ensure that only committed stores write their value to memory. This also ensures the notion of a precise state in memory. This issue will be revisited in Section 7.2. However, for the time being we can assume that the memory state remains safe in an environment where we switch between multiple programs by storing and restoring contexts.

3. We need to know the values of all the architectural registers in the precise state. Assume that the last committed instruction is instruction I. Now, assume that a single-cycle processor was executing the same program. Then the architectural state of the registers in the precise state (in our OOO processor) should be the same as that produced by executing the program till instruction I using the single-cycle processor. We can read the values of all the architectural registers, store them in the program's context, execute other programs, and then restore the original program's context. The original program needs to see exactly the same values of all the architectural registers.

Important Point 9
At any point of time, we need to only keep track of the contents of the architectural registers, the next PC, and the contents of the memory if we only consider all the committed instructions. We are assuming that none of the uncommitted instructions have even begun their execution. Since we have successfully solved the problem for the next PC and memory, we only need to create a method for architectural registers. Let us define this as the precise register state*.*

Let us look at some of the most common methods for tracking the precise register state at any point of time.

Retirement Register File (RRF)

Let us create a new structure called a retirement register file (RRF) as shown in Figure 4.30.

We maintain a small register file in the commit stage. It contains as many entries as the number of architectural registers. The moment an instruction with a register destination commits, we write its result to the corresponding architectural register in the RRF. As simple as that!

This mechanism ensures that the RRF always contains the precise state or the committed state of the program for the architectural registers. The RRF does not contain any values generated by uncommitted instructions. Let us summarize a few more salient points of the mechanism.

1. Each ROB entry needs to be augmented with the following fields: value produced by the instruction (64 bits), and the id of the destination register (4 bits)

2. Every time an instruction commits, we need to do a register write (to the RRF).

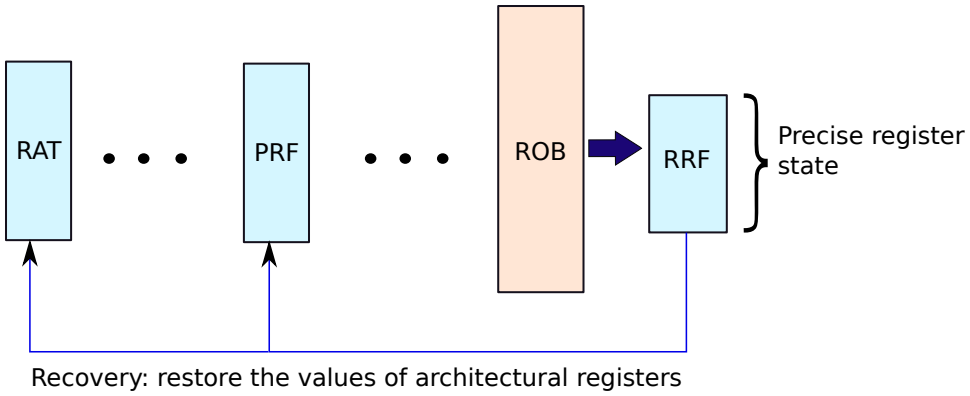


Figure 4.30: A retirement register file (RRF)

3. Restoring the state involves transferring the entire contents of the RRF to the regular register file with appropriate changes made to the rename table. Since we are initializing from a clean state, we can transfer the contents of architectural register ri (in the RRF) to physical register pi and create a corresponding mapping.

Let us try to do the same thing in another way.

Retirement Register Alias Table (RRAT)

Instead of a retirement register file, let us instead try to do something with the RAT table. Let us have an additional RAT table (rename table) in the commit stage as shown in Figure 4.31.

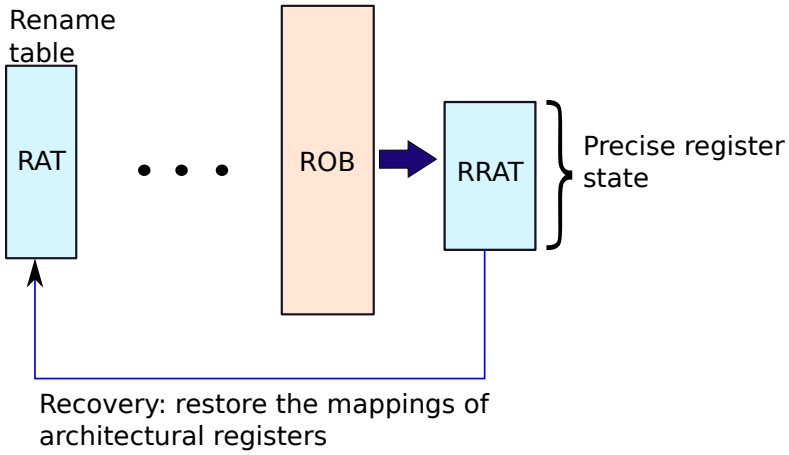


Figure 4.31: A retirement register alias table (RRAT)

In this case, each entry of the RRAT table maintains a mapping between the destination architectural register, and its mapped physical register for each committing instruction. Let us consider the instruction: *add r1, r2, r3*. Assume that the architectural register $r1$ is mapped to the physical register $p1$. When this instruction commits we add the mapping between $r1$ and $p1$. If we need to know the contents of the architectural register file in the committed state at any point of time, then we can just access the RRAT, get the corresponding physical registers, and access them. The mapping stored in the

RRAT stores the precise register state. The RRAT does not have mappings for instructions, which are not committed. Proving the correctness of this scheme is left as an exercise for the reader. Bear in mind that the physical registers pointed to by the RRAT will continue to maintain their values; they will not have been released.

The salient points of this scheme are as follows:

1. We need to maintain additional information in each ROB entry: id of the destination register and its corresponding physical register id.
2. For every instruction with a register destination we need to update the RRAT at commit time.
3. Restoring the state upon a pipeline flush involves performing N reads from the RRAT, and performing N writes to the actual RAT. Here, N is the number of architectural registers.

Nevertheless, this is a simple mechanism and has lower storage overheads than the RRF scheme: in this case, we just store the ids of the mapped physical registers instead of the full 64-bit values.

Checkpointing the RAT: SRAM Array

Instead of making changes to the commit stage, let us instead make changes to the rename stage. In this case, let us take a checkpoint of the RAT every time we encounter a branch instruction. A checkpoint is defined as a *snapshot* (a copy of all the entries, in this case it is a copy of the RAT table). Whenever, there is a branch misprediction of branch B we restore the RAT table to the checkpoint associated with it. This process is known as *recovery*. Every checkpoint captures the current mapping at that point of time. If all the instructions till instruction B are committed, then the registers mapped to the architectural registers in the checkpointed RAT table represent the committed state at that point of time. Note that subsequently, the registers will not be released because a physical register is released only when an instruction that overwrites the mapped architectural register commits. Since we restore the state when instruction B commits, we are guaranteed that no instruction after B would have committed. Thus, the physical registers containing the committed state would not have been released.

Note that other than branch mispredictions regular instructions can also suffer from faults, and we can receive hardware interrupts any time. In this case we will not have a precise checkpoint to roll back to. A naive option is to take a checkpoint of the RAT table after every instruction – this is too expensive. The other option is to roll back the state to the latest branch instruction. However, this introduces additional complexities in terms of correctness. Hence, let us proceed with the naive assumption that we are only dealing with branch mispredictions.

Consider a simple rename table, where we have N entries (N is the number of architectural registers). This can be constructed using simple static RAM (SRAM) cells (see Chapter 7 and [Sarangi, 2015]). Note that an SRAM cell is a memory cell consisting of 6 transistors. Each SRAM cell stores a single bit, and SRAM cells are organized as a matrix of cells (in rows and columns).

In this case, we can organize the SRAM array as follows. We organize each entry as a circular queue, where we insert new entries at one end and remove older entries at the other end. We always maintain two pointers: *head* and *tail* to the first and last entries respectively. Given that we are dealing with real hardware here, we can place a limit on the number of entries in the queue, and this should roughly be equal to the maximum number of branches we expect to have in the pipeline at any point of time. Figure 4.32 shows the design.

Whenever, we encounter a branch instruction, we create a new copy of the up-to-date entries of the RAT table and push them into the queues associated with the entries. This means that for a given entry, let's say architectural register $r4$, we read its latest mapping (let's say $r4 \leftrightarrow p10$), and insert $p10$ into the queue associated with the entry for $r4$. Subsequent instructions (after the branch) are free to update the mapping of $r4$. Now, let us assume that this branch (instruction B) gets mispredicted. We wait till instruction B commits. At that point, the checkpoint for each entry associated with B such as $r4 \leftrightarrow p10$ is guaranteed to be at the head of the respective queue. This is because no instruction older

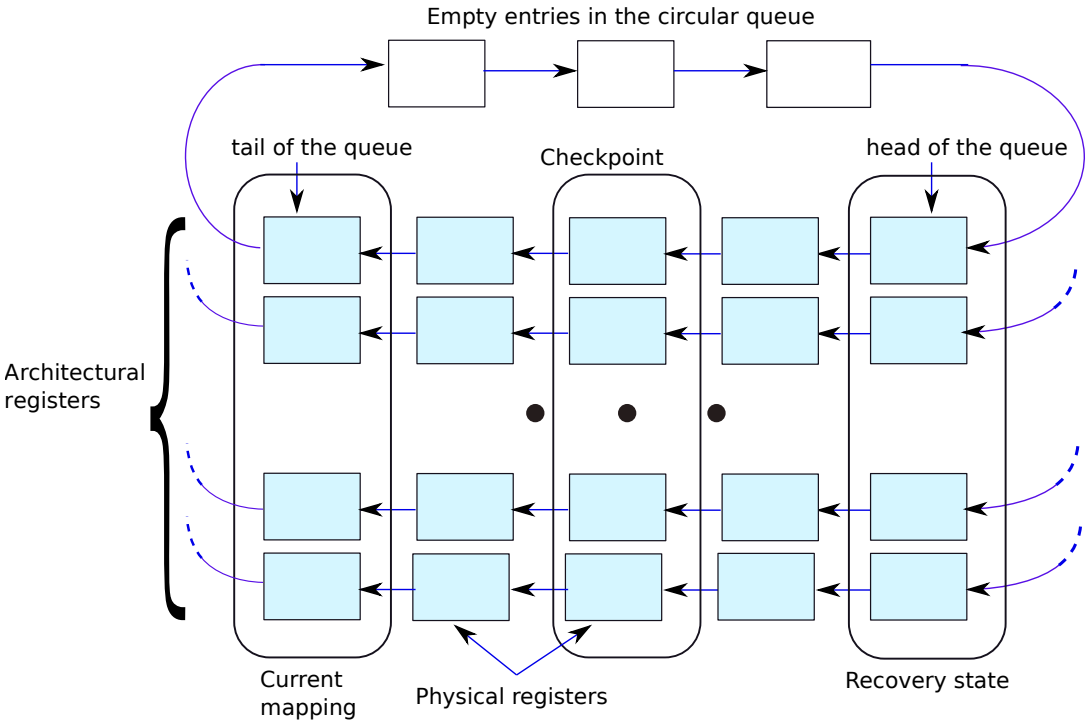


Figure 4.32: Checkpointing the RAT array with an SRAM-based implementation

than B will be in the pipeline, and thus its checkpoint will also not be there. This result can also be deduced from the FIFO (first in first out) property of each queue.

At this point to restore the checkpoint, we just need to discard the rest of the entries, which is very easy to do. We simply set the *head* pointer equal to the *tail* pointer – the rest of the entries get discarded on their own. This is how we can very easily restore a checkpoint.

The shortcomings of this scheme are as follows:

1. Every entry of the rename table needs to be organized as a circular queue.
2. We need to take a checkpoint (create a copy of the latest entry in each row) on every branch instruction. This means that we need to copy 7 bits (assuming there are 128 physical registers) between the tail of the queue and the memory cells that store the current mapping.

Checkpointing the RAT: CAM Array

A major problem with the previous solution was that we needed to move 7 bits between locations for creating a checkpoint. Secondly, if there are N entries in each circular queue, we need $7N$ bits of storage in each row of the SRAM array. Can we reduce this further such that the process of checkpoint creation becomes very easy?

Let us design the rename table (RAT table) in another way. Instead of using an SRAM array, let us use a CAM array (see Chapter 7 and [Sarangi, 2015]). A CAM (content-addressable memory) as discussed in Section 4.3 can be addressed in two ways: by the row index, and by the content in each row. We can design the RAT table as a CAM as follows. The content is the architectural register and an additional bit that indicates if the mapping is valid or not. Let us consider a scenario with 16 architectural registers and 128 physical registers. Instead of a traditional 16-entry table, let us have an

128-entry table. The contents in each row comprise a 4-bit architectural register id and 1 bit (valid bit). Note that at any point of time, only 16 entries will have their valid bits set. These 16 entries are mapped to each of the 16 architectural registers. For example, if we need to find the mapping corresponding to register $r4$, we create a 5-bit bit field: 4 bits from $r4$ that are 0100, and 1 as the valid bit. The bit field is thus 01001. We then look up the CAM for a row with contents that match 01001. Only one row should match this value, and that row contains the current mapping for architectural register $r4$. Let this be row number 37 in the 128-entry RAT table. We can automatically infer that the physical register that is mapped to $r4$ is $p37$. We can use a simple Boolean encoder in this process.

Note that we made many statements in the previous paragraph without proof. The reader is invited to prove them. For example, why are we claiming that only 16 out of 128 entries will have their valid bit set to 1?

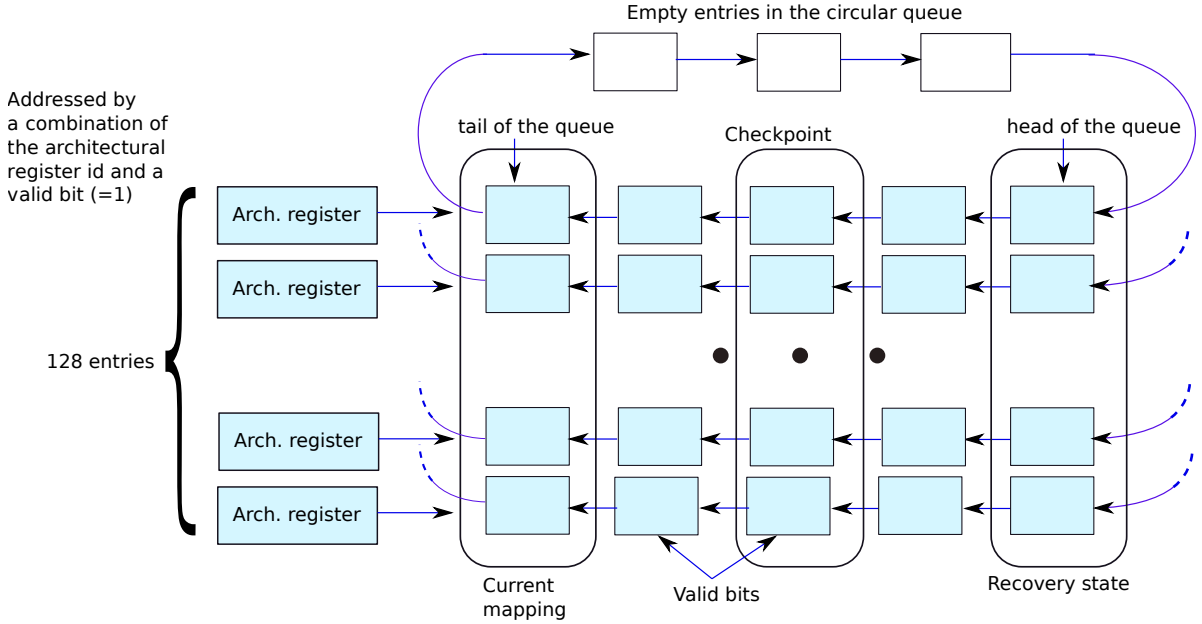


Figure 4.33: Checkpointing the RAT array with a CAM-based implementation

The important take-home point is that an array of valid bits contains the current mapping from architectural registers to physical registers. Now assume that $r4$ is mapped to $p37$. One more instruction comes by later that updates $r4$. It will be mapped to another physical register – let's say $p10$. The only change that needs to be done is that the valid bit for the 37^{th} entry needs to be unset (set to 0) and for the 10^{th} entry we need to set the contents to 4 (for $r4$), and then set the valid bit to 1.

Our design is similar to the checkpointing scheme with an SRAM array. In every row, we have a small circular queue that in this case stores a single bit per entry (valid bit). Before every branch, we take a checkpoint of the current mapping (128 valid bits). We do this by inserting the current valid bit of each row into its circular queue. Whenever, a mispredicted branch commits we remove its checkpoint from the head of the queue. To restore a checkpoint, and restart from that point we just restore the state of the RAT table to what it was before renaming the branch instruction. The head of each queue contains this state, and thus akin to the scheme with an SRAM array we can restore the checkpoint by making the entries at the head of the queues act as the current mapping. Note that in this scheme only the valid bits are a part of the queue (not 7-bit physical register ids). Refer to Figure 4.33.

Let us convince ourselves of one more fact. Consider the time at which we are restoring the checkpoint. At that point of time we wish to say that the architectural state is contained within a set of physical

registers. Now, the entries corresponding to those registers will still be mapped to the same architectural registers, albeit their valid bits may not be 1 anymore. This is because we might have encountered a subsequent instruction that writes to the same architectural register. However, the mapping will still be there because to release the mapping, a later instruction that writes to the same architectural register needs to commit. Since the branch that owns this checkpoint has not committed, those later instructions would also have not committed – they would be after the branch. Hence, the mappings between physical and architectural registers would still be there. We just need to restore the set of valid bits. Recall that at any point of time only 16 out of 128 entries will be set to 1. The rest will be set to 0. This is because we shall always have a one-to-one mapping between the architectural registers and physical registers.

A clear advantage of this scheme is that instead of moving around 7 bits, we move just 1 bit. This means that taking a checkpoint is far easier, and also the overhead of storing checkpoints is much lower (7 times lower if we have 128 physical registers). However, there are other problems. Let us quickly review the shortcomings.

1. A CAM is far slower than an SRAM array of equivalent size. In this case, the CAM is expected to have many more rows than the equivalent SRAM based design. Considering these factors, a regular access to the RAT table will have a much higher latency.

2. A CAM also consumes more power. This needs to be taken into account when we opt for such designs.

3. We have the same issue of "taking checkpoints only at branches" as we had with the design that used SRAM arrays. If we want to take more checkpoints we need to increase the size of the circular queues. Otherwise, we lose the ability to recover at arbitrary points within the program, unless we do some additional bookkeeping.

Summary of Register Checkpointing Schemes

RRF	
+	Easy to implement. Transferring the checkpoint to the register file is easy.
-	Extra register writes every cycle. More power.
-	Need to store the result and destination register in the ROB. More space.
RRAT	
+	Requires less space in the ROB than the RRF.
-	Involves a few writes almost every cycle. More power.
-	Each entry of the ROB needs to be augmented with the id of the destination register and its associated physical register.
SRAM based RAT	
+	Activity only on a branch.
+	Checkpoint restoration is easier than RRF and RRAT based schemes.
-	Each row of the RAT is wider.
-	For every branch instruction we need to add an additional entry into the circular queue. It is hard to checkpoint and restore the state at arbitrary points in the program without additional bookkeeping.
CAM based RAT	
+	Checkpoint creation is very easy. Insert only 1 bit.
-	The CAM per se is a slower structure than an SRAM array. It has a higher latency, and consumes much more power.

4.5 Summary and Further Reading

4.5.1 Summary

Summary 3

1. *To remove WAW and WAR dependences in out-of-order (OOO) pipelines, it is necessary to perform instruction renaming.*
2. *The classical method to perform renaming assigns architectural registers to physical registers. We store a mapping between architectural registers and physical registers in a rename table (RAT table).*
3. *It is possible that there might be RAW dependences between different instructions that we are trying to simultaneously rename. We need to have dependence check logic, and forward the ids of assigned physical registers between instructions in this set.*
4. *The list of free (unassigned) physical registers is kept in a structure called a free list. When a physical register is released we add it to the free list.*
5. *A physical register is released when a subsequent instruction that writes to the same architectural register (that the physical register is mapped to) exits the pipeline.*
6. *The process of adding renamed instructions to the instruction window is known as instruction dispatch. Instructions wait in the instruction window till their operands are ready, and the functional units that they need are available.*
7. *Every instruction whose operand is ready waits for the tag to be broadcast on a tag bus. The tag is the id of the physical register assigned to the operand.*
8. *Once an instruction sees the tag on a tag bus, the operand is deemed to be ready. Once all the operands are ready, the instruction wakes up, and is ready to execute.*
9. *Since multiple instructions can be ready at the same time, we need to select a subset of instructions that can begin execution in a given cycle. The select unit is typically structured like a tree. The requests are the leaves, and in every level we discard some requests either based on priority, or randomly.*
10. *To ensure back-to-back execution (dependent instructions executing in consecutive cycles) it is necessary to execute the wakeup and select operations in the same cycle. In addition, we need to broadcast early (much before the instruction actually executes). In specific, if the instruction takes k cycles to execute, then we need to broadcast the tag k cycles after the instruction gets selected. This will ensure that instructions with RAW dependences can execute back to back.*
11. *To track the dependences between loads and stores, there is a need to create a separate structure called a load-store queue (LSQ). We create entries in this queue at the time of decoding an instruction. When we compute the address (in the execute stage), we update the respective LSQ entry.*
12. *At that point of time, the load instruction searches for earlier store instructions. If it finds a store instruction with a matching address, then it uses the value that it is going to store. Otherwise, if it finds a store with an unresolved address, then it waits.*

13. *Similarly, a store instruction searches for later load instructions until it encounters a store instruction that is either to the same address or is unresolved. If any of these load instructions have a matching address, then it forwards the value.*

14. *The LSQ is implemented as two separate queues: one load queue and one store queue. It uses parallel Boolean operations to speed up its operation.*

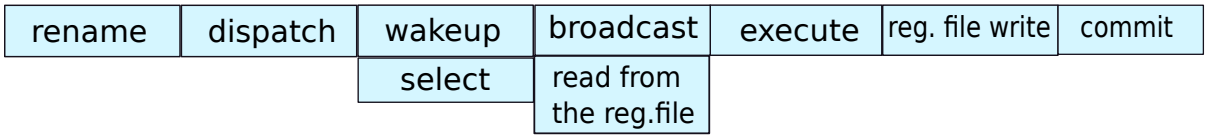
15. *We use a reorder buffer (ROB) to queue all the instructions that are active in the pipeline. An instruction commits (retires) when it reaches the head of the ROB. At that point of time it stores to memory, and its effects are said to be visible to the external world. The process of committing needs to be in program order to guarantee precise exceptions.*

16. *Whenever we mispredict a branch, or encounter an exception, we mark the instruction and wait for it to reach the head of the ROB. Once it does so we flush the pipeline, handle the exceptional event, and then restart the program from the same point.*

17. *To restart a program we need to store a checkpoint of the pipeline state – state of the architectural registers, and the next PC.*

18. *The state of the architectural registers can be stored in a retirement register file, or their mappings can be stored in an RRAT (retirement RAT). In addition, it is possible to achieve the same objective by storing checkpoints of the rename table at different points of interest such as right before a branch.*

The final pipeline from the rename to commit stages looks as follows.



4.5.2 Further Reading

Some of the earliest papers in the area of out-of-order execution were published by a few groups in the University of Wisconsin and the University of Illinois. A few of these landmark papers are [Hwu and Patt, 1987, Smith and Sohi, 1995]. A more detailed circuit level analysis can be found in [Palacharla et al., 1997]. After these research ideas, many processor vendors started building OOO processors. They published the details of their design. Some influential papers for industrial designs are descriptions of the Alpha 21264 [Leibholz and Razdan, 1997], Intel Pentium 4 [Hinton et al., 2001] and AMD Opteron [Keltcher et al., 2003] processors.

Let us now look at some promising ideas in the research community. [Brown et al., 2001] discuss instruction scheduling without the select operation, [Petric et al., 2005] perform standard compiler optimizations at the rename stage, and [Akkary et al., 2003] propose to create very large instruction windows.

To learn more about optimized LSQs, the reader should read the paper by Park et al. [Park et al., 2003]. There are two papers that we would like to recommend regarding releasing pipeline resources early: [Martínez et al., 2002] and [Ergin et al., 2004].

Exercises

Ex. 1 — Design the dependence check logic for a processor with a rename width of 4 (can rename 4 instructions per cycle).

Ex. 2 — Describe in detail how to set the *avlbl* bit for each entry in the rename table, and how to use it in the pipeline.

Ex. 3 — Why is the free list typically designed as a circular queue?

Ex. 4 — How do we free entries in the instruction window? Design an efficient scheme.

Ex. 5 — Assume that we want to create a scheme where we try to allocate physical registers uniformly. How can we modify the free list to support this feature?

Ex. 6 — Describe the wakeup mechanism in detail, particularly, when we are broadcasting multiple tags every cycle.

Ex. 7 — Do we need bypass and dependence check logic to access the register file? If yes, then provide an implementation.

Ex. 8 — Why do we need to broadcast the tags twice?

* **Ex. 9** — How do we perform an early broadcast if the execution duration is not predictable? Can we do better if we have a bound on the maximum number of cycles we require to execute an instruction?

* **Ex. 10** — We want to design a high performance OOO processor that has separate pipeline stages for the wakeup and select operations. Can you suggest modifications to the pipeline with physical registers? Your answer should address the following issues/points:

- What is the advantage of having separate stages for wakeup and select?
- What complications will it introduce to the simple design discussed in this chapter?
- When do we broadcast?
- What are the other changes that should be done to the rest of the stages in the pipeline?
- How do we take care of the issue of double broadcasts?

* **Ex. 11** — Let us design an OOO processor with a speculative select logic. In a regular OOO processor, an instruction might not necessarily get selected immediately after it wakes up. Assume that there is one adder, and three add instructions wake up at the same time. Only one of them will be immediately selected. The rest of the instructions need to wait.
Now let us speculate on this. We assume that the moment an instruction wakes up, it is eligible to be subsequently selected without any delays. It can thus go ahead and wake up consumer instructions.
Design a scheme that has such a speculative select mechanism. Your answer should address the following issues (points).

- How do we realize the fact that we have speculatively selected more instructions than the number of functional units? This will lead to structural hazards unless corrected.
- How do we handle such situations?
- What do we do with instructions that have been speculatively selected?
- How do we reduce the number of misspeculations?

** **Ex. 12** — It is very frequently the case that we have single-shot instructions. An instruction i : $r1 \leftarrow r2 + r3$ is a single-shot instruction if there is only one instruction j that reads the value that i writes to its destination register $r1$, and after j executes, the value in $r1$ is not required. However, we cannot deallocate this register till a subsequent instruction that writes to $r1$ commits. This approach decreases the number of available physical registers in a pipeline because we have many such *short-lived* registers.
Can we speculate? Can we speculatively release a register before it should be actually released? How will this mechanism work? Explain in detail.

** **Ex. 13** — Assume we have an OOO processor with a PRF (physical register file). Given that we have 128 physical registers, what is the maximum possible size of the instruction window? In such processors, it is typical to have a large ROB. For example, the ROB in this case (with 128 physical registers) can be sized to contain 160 entries. Why is this the case?

* **Ex. 14** — In an in-order processor, the compare (*cmp*) instruction is used to compare the values in two registers. The result is saved in a $flags$ register that is not accessible to software. Subsequent branch instructions use the value of the $flags$ register to compute their decision. Will the same mechanism work in an OOO pipeline? If not, then how do we augment it to support this feature of the ISA?

Design Problems

Ex. 15 — Understand the wakeup, select, and broadcast logic in the Tejas simulator ™.

Ex. 16 — Extend the simulator to make the delays of the wakeup, select, and broadcast stages configurable. They need not be done in the same cycle, and back-to-back execution is not a necessity.