# Lecture Notes: Out-of-Order Processors

Rajeev Balasubramonian

October 13, 2007

Most modern high-performance processors today employ out-of-order execution. These notes cover the design of a microarchitecture style that is most commonly used (MIPS R10k, Alpha 21264, Pentium4, etc.).

## 1 Temporary Storage in ROB

We'll start by discussing a design style that is slightly different, but easier to understand.

As shown in Figure 1, instructions are fetched and placed in the instruction fetch queue (IFQ). These are the original assembly instructions found in the program executable and they refer to the (say) 32 registers defined in the architecture. These are referred to as the program's *architectural registers* and they have a separate special storage: *the architectural register file* (shown in the right top corner of the figure). If the first fetched instruction were to be interrupted or raise an exception, the architectural register file stores the results of all instructions until that point – this is the "view" that needs to be shown to the programmer, or this is the state that needs to be saved away and restored so we can return back to this program.

The instructions in the IFQ have not yet begun to execute and we plan to execute them in an out-of-order fashion while still preserving data dependence constraints. Since instructions may finish in an arbitrary order, we can't allow them to modify the contents of the architectural register file as they finish (else, it will be difficult to provide a clean view if there's a bad event (branch mispredict, exception, interrupt)). Hence, every instruction that enters the pipeline is provided a temporary register where it can save its result. These temporary results are eventually written into the architectural register file in program order. Thus, the contents of the architectural register file change as if we are executing one instruction at a time in the original program order (when in fact the processor is executing instructions in an arbitrary order in an effort to improve performance). This feature makes it possible to easily recover from a bad event. It is the *Reorder Buffer (ROB)* that facilitates the above process. The ROB keeps track of the program order in which instructions entered the pipeline and for each of these instructions, it maintains a temporary register storage. In the example shown in Figure 1, we are assuming a ROB with only six entries, with the temporary register storage being named T1-T6.

The instructions in the IFQ go through a process of decode and rename and get simultaneously placed in the ROB and the *issue queue (IQ)*. The instruction's result is to be placed in its temporary storage (T1). Hence, the first instruction ($R1 \leftarrow R1 + R2$) is renamed to $T1 \leftarrow R1 + R2$. A table in the Rename unit (the Speculative Register Map Table) keeps track of the current name (mapping) for every register (the current name for R1 is now T1). When the next instruction comes along ($R2 \leftarrow R1 + R3$), the R1 it refers to is actually T1 and its result is written into T2 ($T2 \leftarrow T1 + R3$) (and the current mapping for R2 is now T2). These renamed instructions are placed in the ROB and in the IQ. Note that branches and stores do not produce new register results. These instructions are provided a slot in the ROB, but they end up not using the corresponding temporary storage since they do not produce a register result (to reduce complications, this temporary storage is not allocated to other instructions).

This process of renaming has an important effect: it eliminates write-after-read (WAR) and write-after-write (WAW) hazards. In the original program, note that there is a WAR hazard with register R2 between the first two instructions: the first instruction reads R2 and the second instruction writes into R2. This WAR

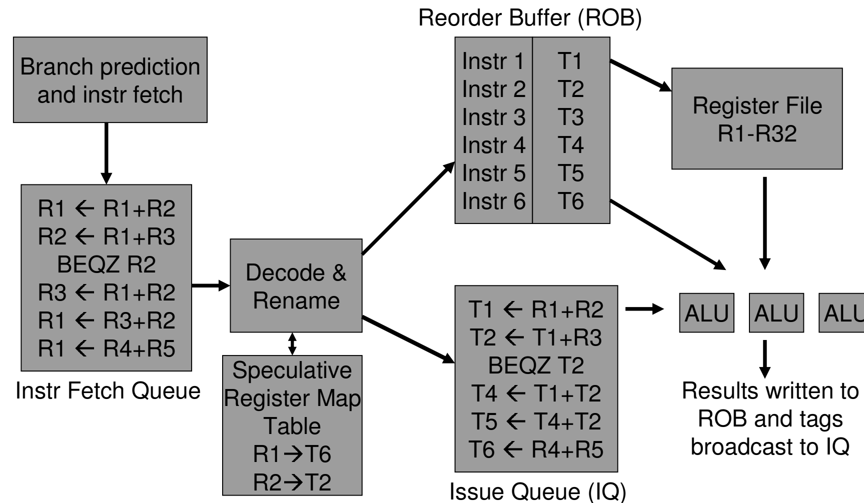



Figure 1: Example with temporary storage in the ROB.

hazard can prevent the second instruction from executing before the first (else, the first instruction may end up reading the wrong value of R2). This WAR hazard is no longer there in the renamed version of the code. The second instruction writes its result into T2: the use of a different name has helped avoid the WAR hazard. Similarly, the fifth and sixth instructions have a WAW hazard with R1 in the original program (disallowing us to re-order the execution of these two instructions). This hazard disappears in the renamed version because we use different names (T5 and T6) for the register result. WAW and WAR hazards are not true dependences – they are *name dependences* – we are avoiding them by simply providing more names (T1-T6) for register results. By avoiding these hazards, we are providing more opportunities to re-order instructions and get higher performance.

Once instructions are renamed and placed in the issue queue, it is the task of the issue queue to determine which instructions are ready to execute in the next cycle. For each instruction, the issue queue keeps track of whether the input operands for the instruction are available or not. In the example in the figure, let's say that all six instructions were simultaneously renamed and placed in the issue queue in cycle 0. The issue queue knows that registers R1, R2, R4, R5 are available and sitting in the architectural register file. So the first and sixth instructions have both their operands ready and can begin execution. The other instructions all depend on at least one of the temporary register values that have not yet been produced. When the first and sixth instructions finish their execution, they write their results into T1 and T6. They also broadcast the availability of T1 and T6 to the issue queue. The second instruction in the issue queue already knew that R3 was available. It now realizes that T1 is also available: it then leaves the issue queue and starts executing in the next cycle. When it completes, it broadcasts the availability of T2 and this allows the third and fourth instruction to leave the issue queue and start executing. The issue queue is thus the "smarts" of the out-of-order processor: it figures out when an instruction has enough information to execute and it causes instruction execution to happen in an out-of-order fashion.

Note that when an instruction is issued out of the issue queue, there is no need to keep it in the issue queue any longer; so it is removed from the issue queue.

Now let's move to the ROB. When the oldest instruction in the ROB produces a valid result, that instruction can now be safely "committed". That is, we can make its result permanent since there is no earlier instruction

that can raise a misprediction/exception that may cause us to un-do the effect of this instruction. As a part of this commit process, the result in T1 is now written into the architectural register file. In this case, T1 was a temporary storage for register R1, so the value in T1 is copied into R1. The ROB also sends a message to the issue queue to clarify the name-change. If an instruction is waiting in the issue queue and needs a result from T1, it corrects the name to R1 so that it gets its input operand from the correct storage.

Now that the instruction is committed, we can re-cycle the first entry of the ROB. The ROB is organized as a circular buffer: instruction-2 now becomes the oldest instruction in the ROB and instruction-7 can now enter the pipeline and be placed in the first entry of the ROB with T1 as the temporary storage for its result.

If a bad event happens, the ROB realizes it and squashes every subsequent instruction. For example, if the third instruction (a branch) turns out to be a mispredict, the ROB simply removes instructions 4, 5, 6 from the ROB and the issue queue and instructs the fetch unit to bring in instructions from the correct path. Even though the sixth instruction may have already completed and written its result into T6, no permanent damage is done. T6 will not get written into the architectural register file unless it is the oldest instruction in the ROB, which means that every previous instruction has committed successfully.

When an exception happens, we must wait until that instruction reaches the top of the ROB. At that point, all previous instructions have committed and saved their results in the architectural register file. Instructions after the excepting instruction have only written results into the ROB and not into the architectural register file. The PC and register file are saved, so we can return to that PC after servicing the exception.

The temporary storage T1-T6 are also known as rename registers. The instructions in the pipeline are referred to as speculative instructions because we don't yet know if they will commit successfully. The ROB keeps track of all these speculative instructions and stores their results in rename registers. It makes these results permanent in the register file through the in-order commit process. Typically, a ROB has many more entries than just six. For many of our subsequent examples, we will assume that the ROB has 32 entries, allowing 32 speculative instructions to enter the pipeline and improving our chances of finding ready instructions to execute in every cycle.

## 2 Design Details of the Issue Process

Let's take a step back and look at the "issue" process in the 5-stage in-order processor with full bypasing. Actually, to simplify the discussion, we will assume the following variant of the pipeline:

```
Instr Fetch & Decode  :  Reg-Read  :  ALU  :  Data-memory  :  Reg-Write
```

In the first stage, an instruction is fetched and we also decode the instruction to figure out if a stall is required. If a stall is required, the PC is not changed and we repeatedly keep fetching this instruction until we figure out that the instruction can now proceed through the pipeline and it will get its appropriate input operands either from the register file or from the bypass network. Thus, it is the decode unit in the first stage that is responsible for the task of "issue". Once the "issue" happens, the instruction goes through the other stages until a value is finally written in the register file in the fifth stage.

As an example, assume there are two back-to-back add operations where the second one depends on the first:

```
R1 <- ...
.. <- R1
```

The first instruction proceeds without any stalls (say). The second instruction is following one cycle behind. The decode unit determines that the first instruction is a single-cycle operation (that is, it takes only 1 cycle after Reg-read to produce a result), and the second instruction need not be stalled. Thus, when the second instruction leaves its issue (decode) stage, the first instruction hasn't even begun to execute – but we are issuing it because we know it will eventually receive its input operands correctly.

Similarly, in the out-of-order processor, the issue queue need not wait for an earlier instruction to produce its result before broadcasting the availability of the result to the instructions in the issue queue. If it waited so long to wake up the dependent, there would be a gap of 4-5 cycles between back-to-back add operations (much more than even the simple in-order case). Hence, as soon as an instruction leaves the issue queue, we estimate the latency of that instruction – for example, an integer-add may be a single-cycle operation, an FP-add may be a four-cycle operation, a load may be a three-cycle operation, etc. Accordingly, we broadcast the availability of that result to the issue queue in either the next cycle (for the integer-add), or four cycles later (for the FP-add), or three cycles later (for the load), etc. Even though the integer-add is deemed a single-cycle operation, it produces a result a few cycles after it leaves the issue queue (this is determined by the depth of the pipeline and how long it takes to read a value out of the register file).

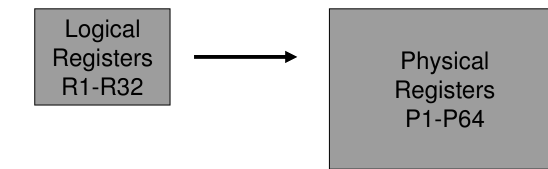
As a concrete example, see how the two back-to-back instructions may go through an out-of-order pipeline that has a width[1] of two. We also assume here that a register-read takes two cycles:

```
Cycle 1: Br-predictor updates the value of PC
Cycle 2: Instr-1 and instr-2 are fetched from that PC and placed in the IFQ
Cycle 3: Instr-1 and instr-2 are decoded/renamed and placed in the ROB/IQ
Cycle 4: The IQ checks for dependences and determines that Instr-1 can issue
Cycle 5: Instr-1 leaves the issue queue.
         The issue queue broadcasts the availability of instr-1's result.
         The IQ determines that Instr-2 can issue in the next cycle.
Cycle 6: Instr-1 goes through the first half of the register read process.
         Instr-2 leaves the issue queue.
         The issue queue broadcasts the availability of instr-2's result.
Cycle 7: Instr-1 goes through the second half of the register read process.
         Instr-2 goes through the first half of the register read process.
Cycle 8: Instr-1 performs its add.
         Instr-2 goes through the second half of the register read process.
Cycle 9: Instr-2 performs its add (receives the result of Instr-1 from bypass).
         Instr-1 sends signal to the ROB.
Cycle 10: Instr-2 sends signal to the ROB.
          Instr-1 writes its result into ROB and updates various flags.
Cycle 11: Instr-2 writes its result into ROB and updates various flags.
          Instr-1 is top of the ROB, so it begins copying its result into
           the architectural register file. It sends signal to the issue
           queue to indicate the name-change.
Cycle 12: Instr-1 completes its copy. It is now committed.
          Instr-2 is now top of the ROB, so it begins copying its result into
           the architectural register file. It sends signal to the issue
           queue to indicate the name-change.
Cycle 13: Instr-2 completes its copy. It is now committed.
```

[1]"Width" refers to the number of instructions that can be simultaneously processed in a pipeline stage in a cycle. A width of four implies that four instructions can be simultaneously fetched in a cycle, four instructions can be simultaneously decoded in a cycle, ..renamed.. issued...committed, etc.

## Managing Register Names

Temporary values are stored in the register file and not the ROB




At the start, R1-R32 can be found in P1-P32
Instructions stop entering the pipeline when P64 is assigned

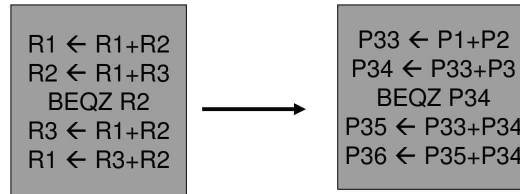



Figure 2: Managing register names.

# 3 Eliminating Register Copy

The one salient dis-advantage of the above implementation is that results must be copied from the ROB to the architectural register file, requiring that we also update the names used by instructions in the issue queue. This is eliminated with the following design.

Instead of maintaining two separate storage structures (32 temporary rename registers in the ROB and 32 architectural registers), we now maintain a single large storage structure with 64 registers. The rename registers and architectural registers are now inter-mingled and two map tables are required to keep track of the designation of each register. The single large storage is referred to as the physical register file.

An architectural map table keeps track of where we can find the 32 architectural registers of the program (note that the architectural registers are also referred to as *logical registers*). For example, logical register R1 may be currently found in physical register P17, R2 may be mapped to P39, R3 to P5, and so on. To simplify the discussion, we will assume that at the start of the program (or upon resumption after a context-switch), R1 is mapped to P1, R2 to P2, ..., R32 to P32. P33-P64 are initially un-used.

When the first instruction is renamed, its result must be written to a temporary rename register. That register is selected out of the free pool of registers (P33-P64). As shown in Figure 2, the original program instruction $R1 \leftarrow R1 + R2$ is renamed as $P33 \leftarrow P1 + P2$. Just as before, we need a speculative map table to keep track of the latest current mapping of a logical register. This map table is updated to indicate that R1 is currently mapped to P33, so subsequent instructions that read R1 must be renamed to read P33.

Finally, when the first instruction commits, it must make the result in P33 permanent. This is done by simply updating the architectural map table to indicate that the latest committed value of R1 can now be found in P33 and not in P1. Since we no longer have any need for P1, we can return P1 back to the free pool of physical registers. Thus, the process of commit does not require us to copy register values, and since there is no name-change for a result, we also need not update the names used by instructions in the issue queue. The use of a single unified register file also makes it easier to design the wiring in the processor between ALUs and register storage. The only significant addition to the previous design is the use of a new architectural

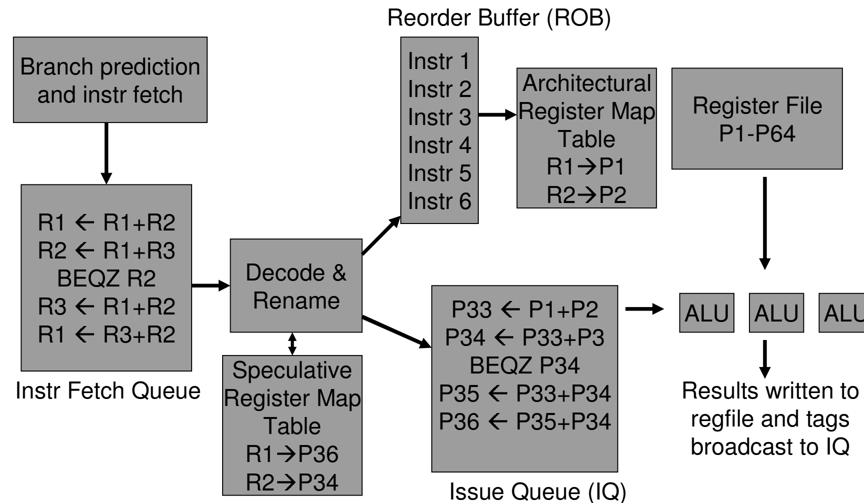



Figure 3: The microarchitecture with a unified physical register file.

map table to keep track of the latest committed value for each architectural register (this change is indicated in Figure 3).

Now let's see when P33 gets released back in the free pool. In the example in Figure 2, the fifth instruction also writes a result into logical register R1. In the renamed version, the result is written into physical register P36. Until this instruction commits, we can not release P33. Because if an instruction before the fifth instruction raises an exception, we must indicate P33 as the latest committed value of R1. Similarly, if the third instruction (the branch) is a mispredict, we may have to go the other way where an instruction may want to read the result in R1, which refers to the value in P33. Hence, the result in P33 is valuable until the fifth instruction commits. At that point, we know that P36 is now the latest committed value of R1 and no subsequent instruction will ever want to read the result in P33. Stated formally, a physical register is released back in the free pool only after the next instruction to write to the same logical register commits.

Just as before, the number of speculative instructions that can enter the pipeline is a function of the number of rename registers. If the physcial register file has 80 registers and 32 of these are designated as the committed architectural register values, that leaves 48 rename registers for speculative instructions. This means that the ROB can bring in 48 instructions and all of these instructions will be guaranteed a rename register to hold on to their temporary result. In reality, the ROB can typically bring in more than 48 instructions because some of these instructions will be stores or branches that need not be assigned a rename register. Thus, size of ROB needs to be a little more than the number of physical registers minus the number of architectural registers.

## 4 Handling Loads and Stores

So far, all of our examples have dealt with simple instructions that only deal with registers. When determining if an instruction can issue, so far we have only had to check for register dependences and it is the issue queue that performs this task. Now we'll consider examples with loads and stores as well – since these instructions also access memory, it is not enough to just check for register dependences; we must also check

Out-of-Order Loads/Stores

| Ld | R1 ← [R2] |
|---|---|
| Ld | R3 ← 8+[R4] |
| St | R5 → 4+[R6] |
| Ld | R7 ← [R8] |
| Ld | R9←8+[R10] |
| St | R11 → [R12] |
| Ld | R13 ← 4+[R14] |
| Ld | R15←[R16] |

| Ld | 0x abcdef |
|---|---|
| Ld | |
| St | |
| Ld | |
| Ld | 0x abcdef |
| St | 0x abcd00 |
| Ld | 0x abc000 |
| Ld | 0x abcd00 |

Figure 4: Example contents in the LSQ.

for memory dependences before issuing them out-of-order.

When an instruction is renamed, it is typically placed in the issue queue and ROB. When a load/store instruction is renamed, it is placed in the ROB and *load/store queue (LSQ)*. As shown in the left of Figure 4, a load instruction uses registers to compute an effective address, then brings the data from that address in memory into a register. The store similarly uses registers to compute an effective address, then transfers data from a register into that address in memory. Hence, loads and stores must first wait for register dependences to be resolved in order to compute their respective effective addresses. In that regard, part of the LSQ is very similar to the issue queue. Each load/store instruction is waiting for a register value to be produced – when it receives the broadcast regarding its availability, the effective address computation part of the load/store is issued.

The effective address is computed on an ALU, and the effective address is then written into the entry in the LSQ (this is unlike other instructions that write the ALU result into the register file). The diagram on the right of Figure 4 shows the contents of the LSQ after a few cycles: corresponding to each Ld/St entry is the effective address that has been computed. Some of the instructions don't yet have an effective address computed because those instructions are still waiting for their register operands to be produced. Now let's see the conditions under which a load can proceed and read data from cache/memory.

In the example shown on the right in Figure 4, the first load has its address available: it can issue in the next cycle, read the contents of that address from cache/memory and store those contents into its destination register. The second and third loads can obviously not issue in the next cycle because they don't yet know the address they must access. The fourth load knows the address it must access, but cannot proceed just yet. Note that there is an earlier store instruction that has not yet computed its address. If this address turns out to be 0x abcdef, then the fourth load needs to return the value written by that store and not the current contents in cache/memory. Hence, there is a possible memory dependence here. Until we know the store address, we can't tell for sure if this load depends on the earlier store or not. For now, we'll take the conservative approach and assume the worst-case scenario – we will not issue the fourth load in this cycle and wait until the earlier store produces its effective address. The same argument applies to the fifth load as well. For the sixth load, we can see that its address (0x abcd00) is the same as the earlier second store. We therefore know

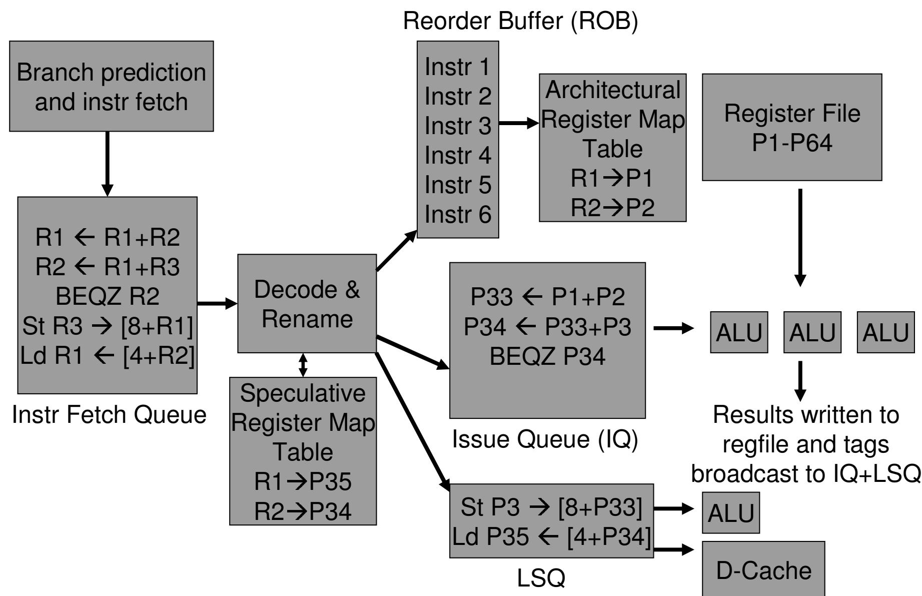



Figure 5: The final microarchitecture that resembles the Alpha 21264: includes LSQ, unified physical register file.

that there is a memory dependence between the two and the load must return the value that is written by the store and not the current contents of cache/memory (this is true even if the first store happens to produce the same effective address of 0x abcd00). The second store is writing the contents of R11 into memory: if R11 is available, the sixth load can issue in the next cycle: all it has to do is copy the contents of R11 into R15.

To summarize, the effective address computation of a load/store can issue as soon as the input register operands are made available. The cache/memory access part of a load can issue as soon as we resolve its memory dependences. The load can access the cache/memory as soon as it knows that there are no prior stores that refer to the same address. If there is a prior store with the same effective address, the result of that store is bypassed to the load (assuming there are no other unresolved store addresses between the store and the load).

When does the store write its register contents into cache/memory? Note that when this write happens, the old value in cache/memory is lost forever. Hence, we must not perform this write until we are sure that this store will not be squashed. The write into cache/memory can be performed only when the store instruction reaches the top of the ROB and is ready to commit.

Figure 5 shows the final processor design with the LSQ included as well. Again, recall that stores and branches are not assigned new physical registers as they do not produce new register results.