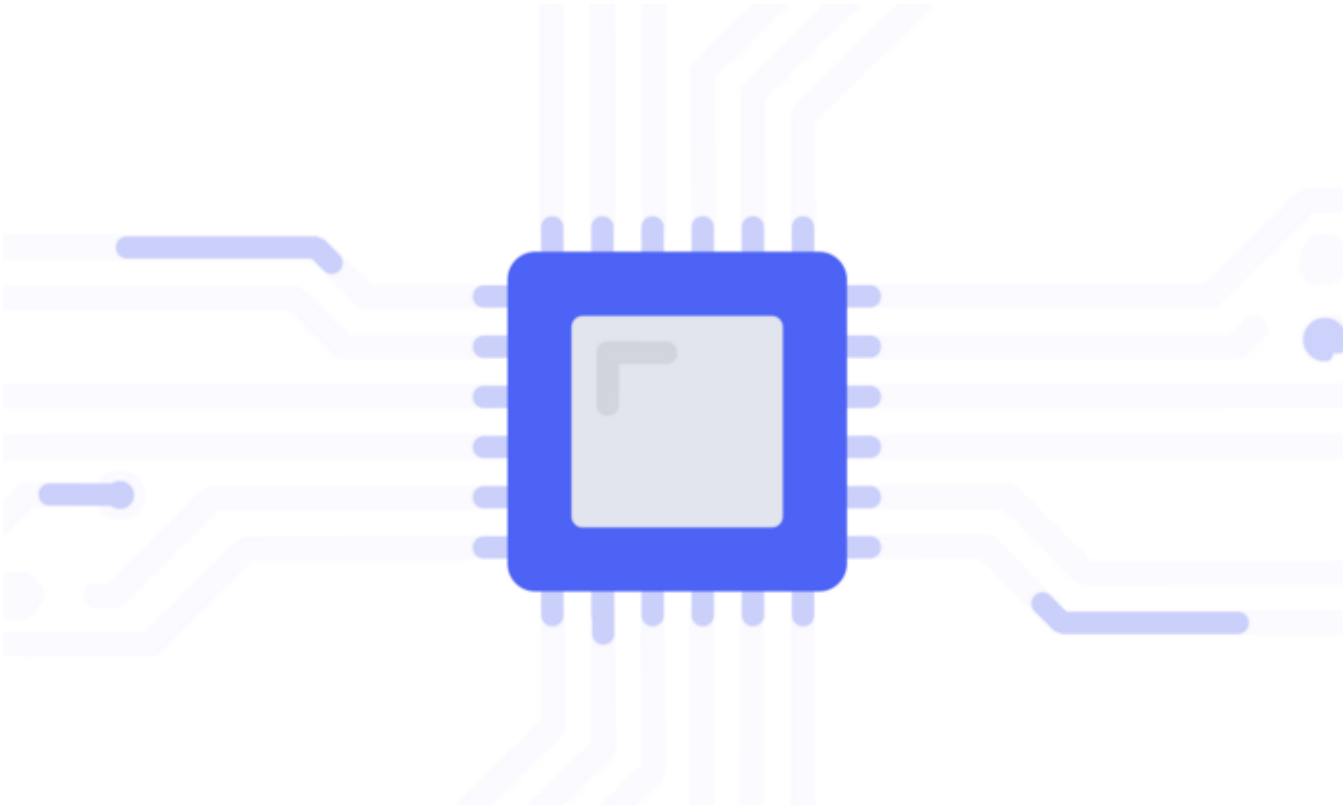
High-Performance Computer Architecture 13 | Tomasulo's Algorithm Part 1

Adam Edelweiss
Jan 28 · 16 min read ★



1. A Brief Review

For an ideal processor, we have known that the ILP can be significantly greater than 1. However, the IPC of any given processor can not be as large as ILP. To improve the performance of the IPC for a given processor, we need to handle,

- **the control dependencies** by branch prediction

- **the WAR or WAW data dependencies** (false data dependencies) by register renaming
- **the RAW data dependencies** (true data dependencies) by out-of-order execution
- **the structural dependencies** by invest in a processor that has a wider issue

2. Tomasulo's Algorithm and Out-of-order Implementation

(1) An Introduction to the Tomasulo's Algorithm

Robert Marco Tomasulo (October 31, 1934 — April 3, 2008) was a computer scientist and the inventor of the Tomasulo algorithm (1967). Tomasulo was the recipient of the 1997 Eckert–Mauchly Award "for the ingenious Tomasulo algorithm, which enabled **out-of-order execution** processors to be implemented.".

Tomasulo's Algorithm is actually one of the first techniques for out-of-order execution and now it has more than 50 years old. It was used in the IBM 360 machine which is delivered between 1965 and 1978.

IBM 360 Machine

Tomasulo's algorithm determines which instructions have inputs so that they are able to go into the next cycle and which instructions still have to wait for their inputs to be produced. It also includes a formal register renaming and it is surprisingly similar to what we actually use today as far as out-of-order execution is concerned.

(2) Tomasulo's Algorithm in History and for Today

There are actually some sorts of differences between how Tomasulo's algorithm was used originally and how it is used now.

In the olden days,

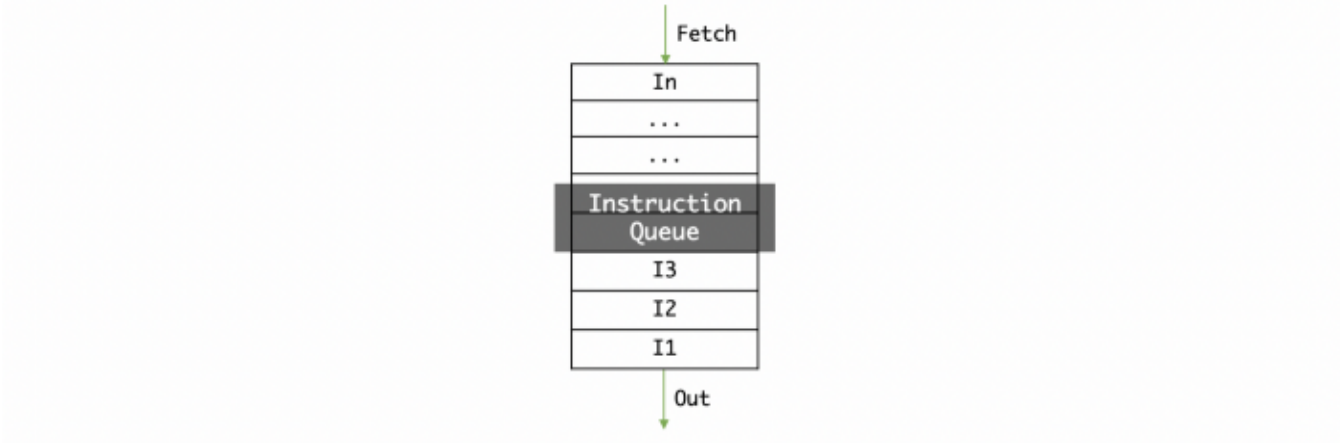
- Tomasulo's algorithm is designed only for floating-point instructions
- Tomasulo's algorithm has relatively fewer instructions in the window that they are looking at
- Exception handling for the floating points is not a big problem

For today,

- Tomasulo's algorithm is updated for even all instructions
- Tomasulo's algorithm looks at hundreds of the instructions when we are executing
- Processors include explicit support for exception handling

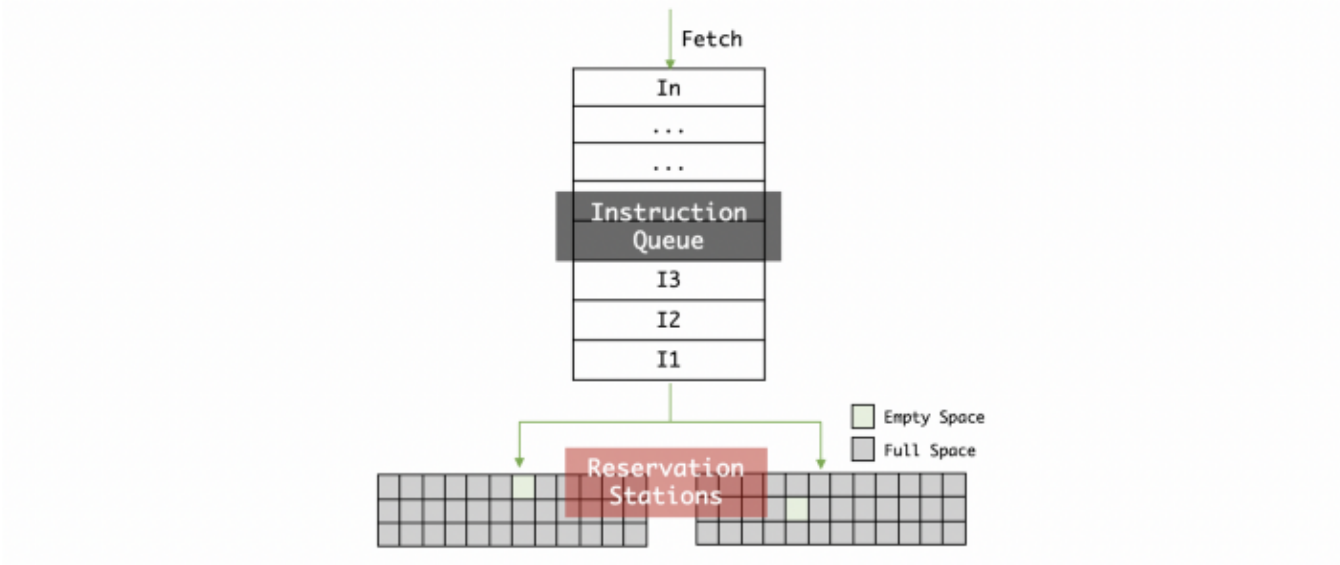
(3) Tomasulo's Algorithm: The Whole Picture

Let's begin the whole picture from the instructions queue. We have known that we need to fetch the instructions in our pipeline (fetching stage). After fetching, the instruction is actually pushed into the **instruction queue** (a FIFO queue).

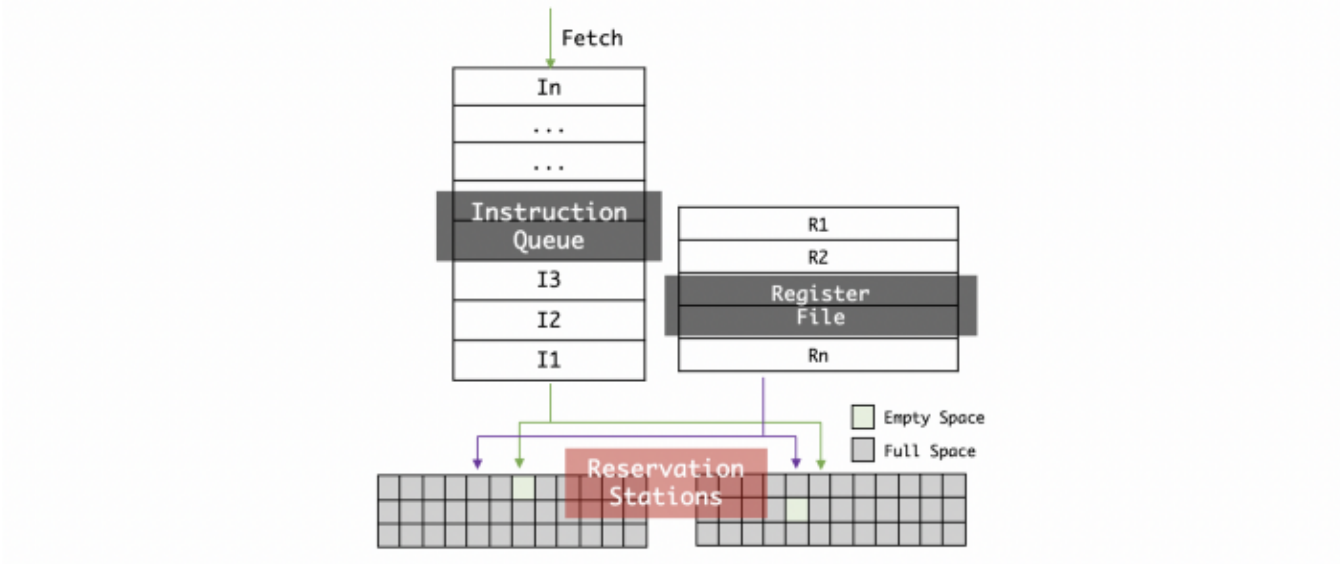


Suppose now we are having only the floating-point instructions. Then to Tomasulo's algorithm machine, we often grab the next available instruction in the order they came (a property of the queue) and then put it into one of the **reservation stations** (there are a number of reservation stations). We can only put an instruction to the reservation

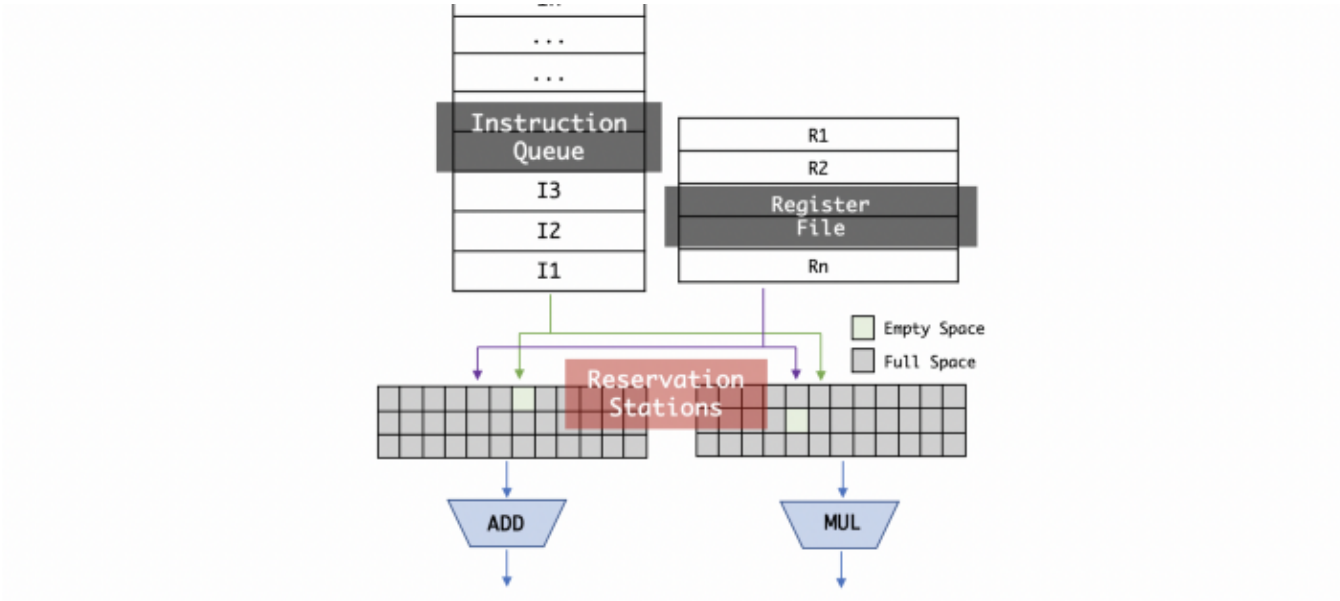
station when there is an empty space in this reservation station. The reservation stations are where instructions basically wait for the parameters to become ready.



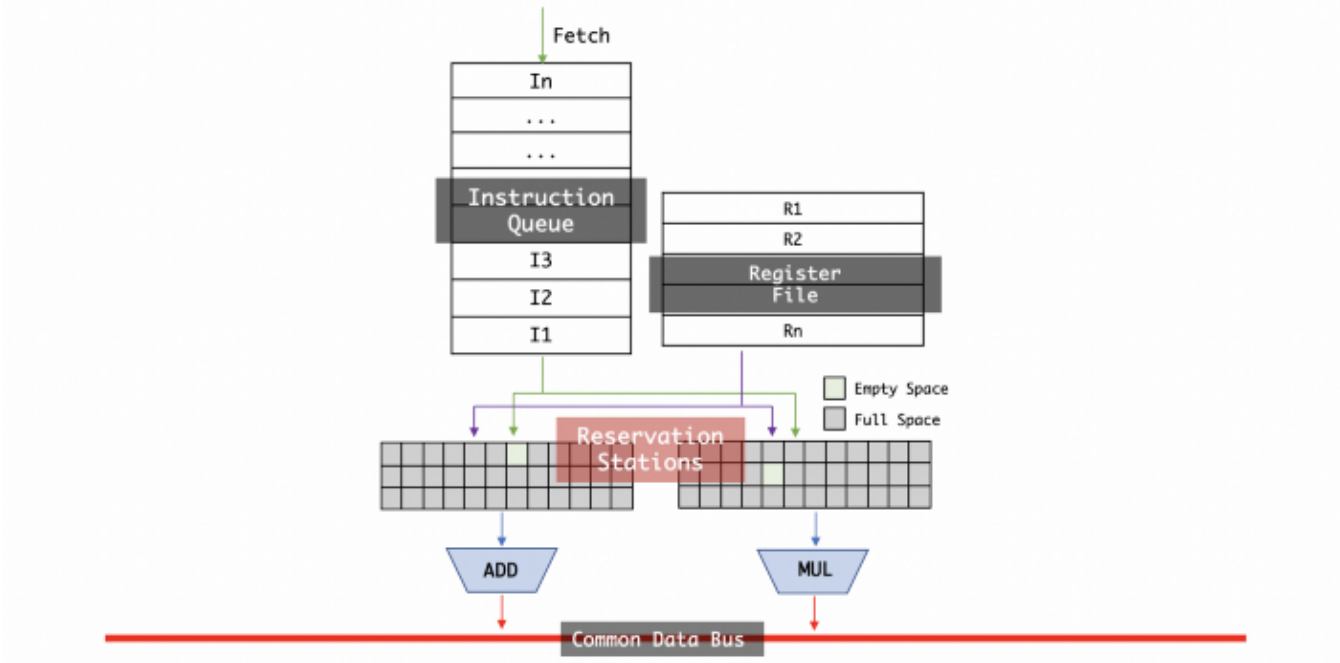
We also have a **register file** that can be used to store the floating-point numbers. A register file is an array of processor registers in a central processing unit. When an instruction is inserted into a reservation station, the values that are already in the registers are going to be simply entered into the reservation station from the register file.



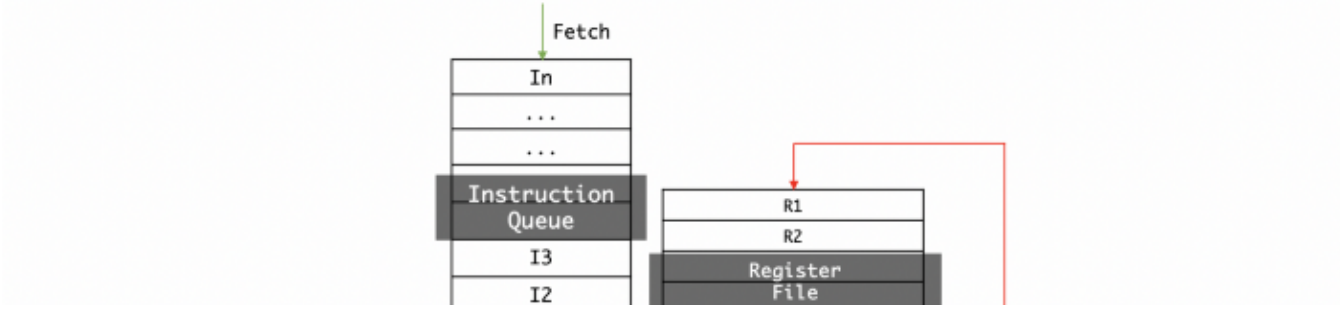
Once an instruction in the reservation is ready to execute, it goes to an execution unit. In fact, we can have different types of execution units, for example, a unit for adder `ADD` or a unit for multiplier `MUL` .

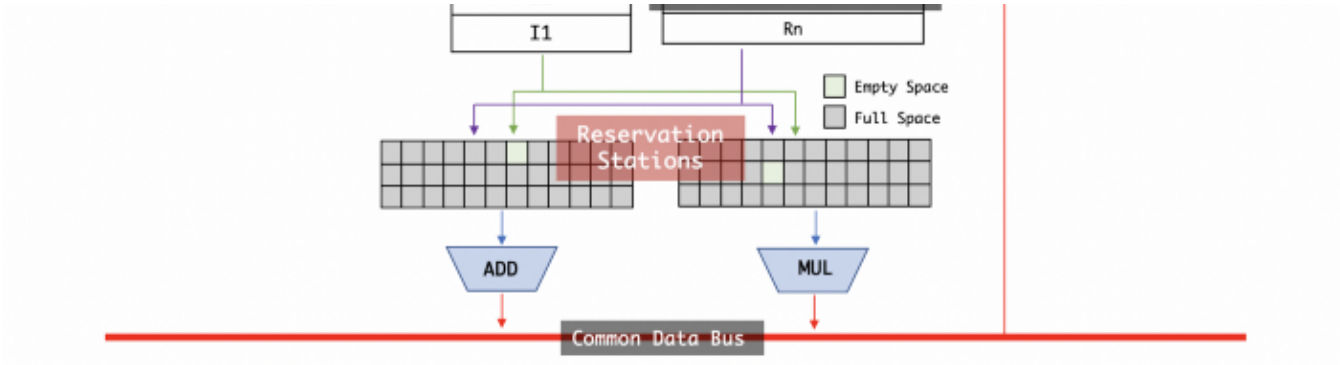


Once such a unit has produced a result, the result will be broadcast on a bus. This bus is called the **common data bus (CDB)**.

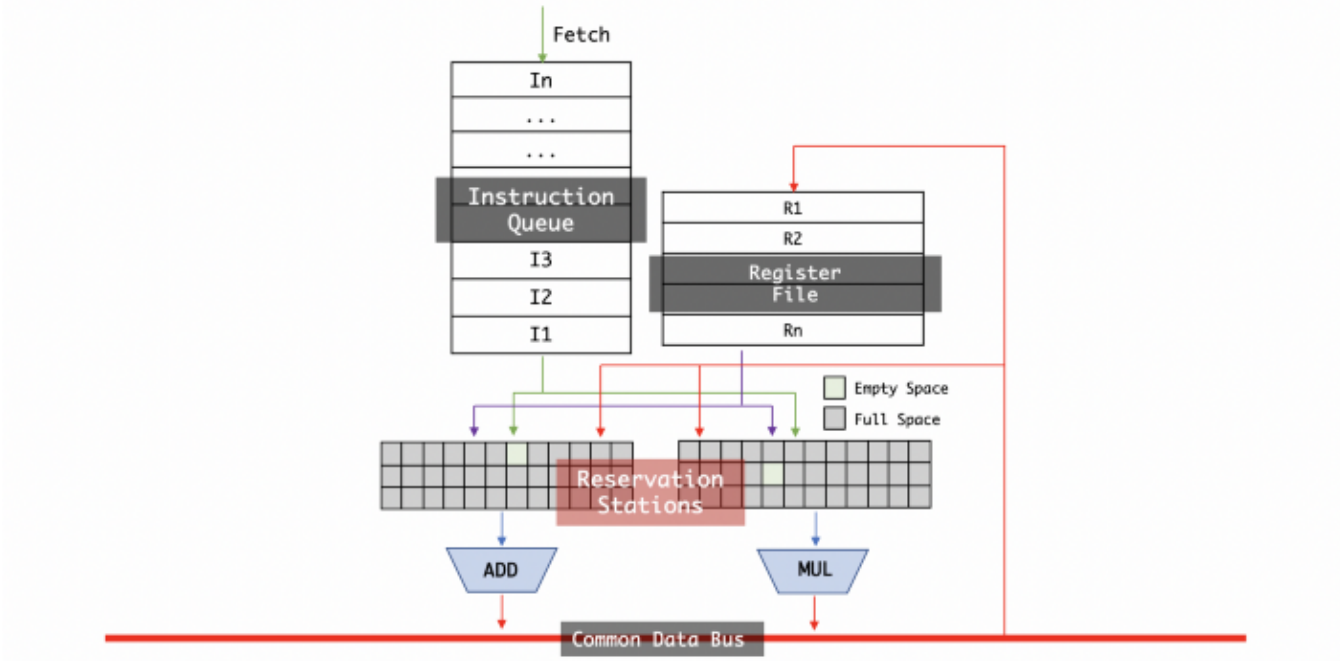


Of course, the results will **go to the register file** so all the results that are output would be available in the register file. So the instructions that **newly** come in can directly grab them the register file.

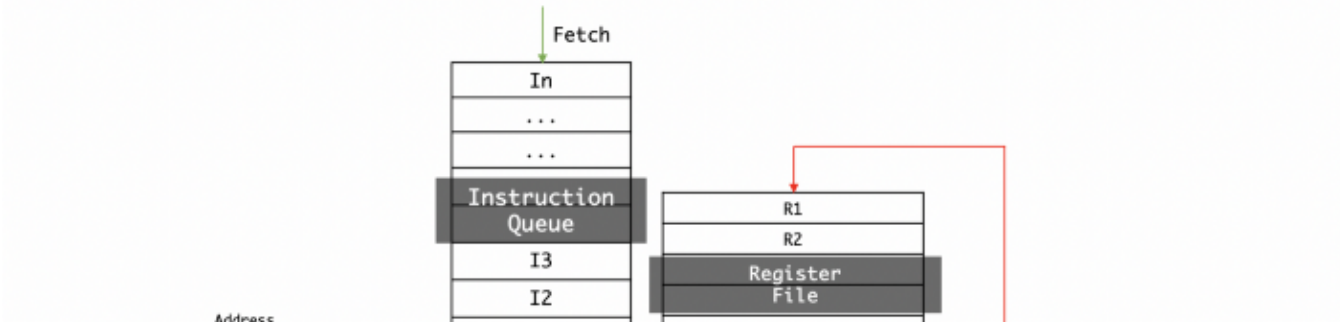


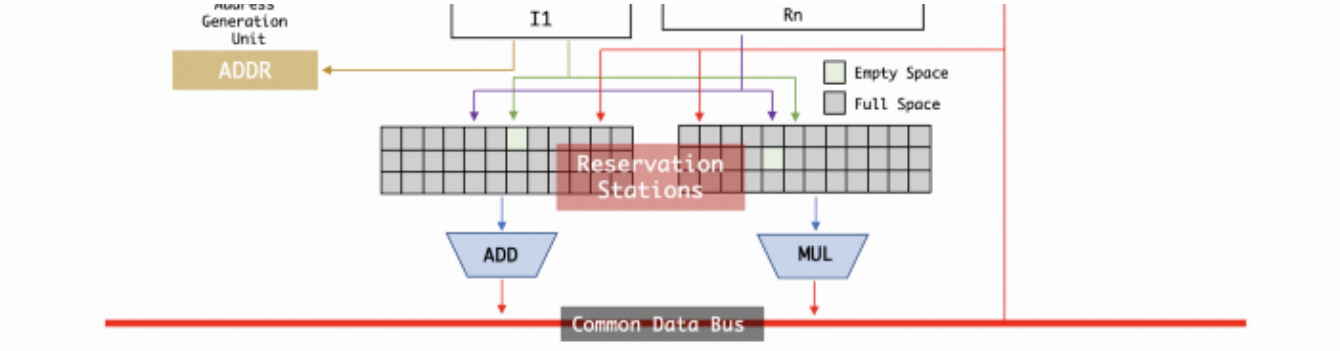


These results will also be **broadcast to the reservation station** because there are some instructions in the reservation station still waiting for the values that they need to be produced. As these values are produced, these instructions can move forward to execute.

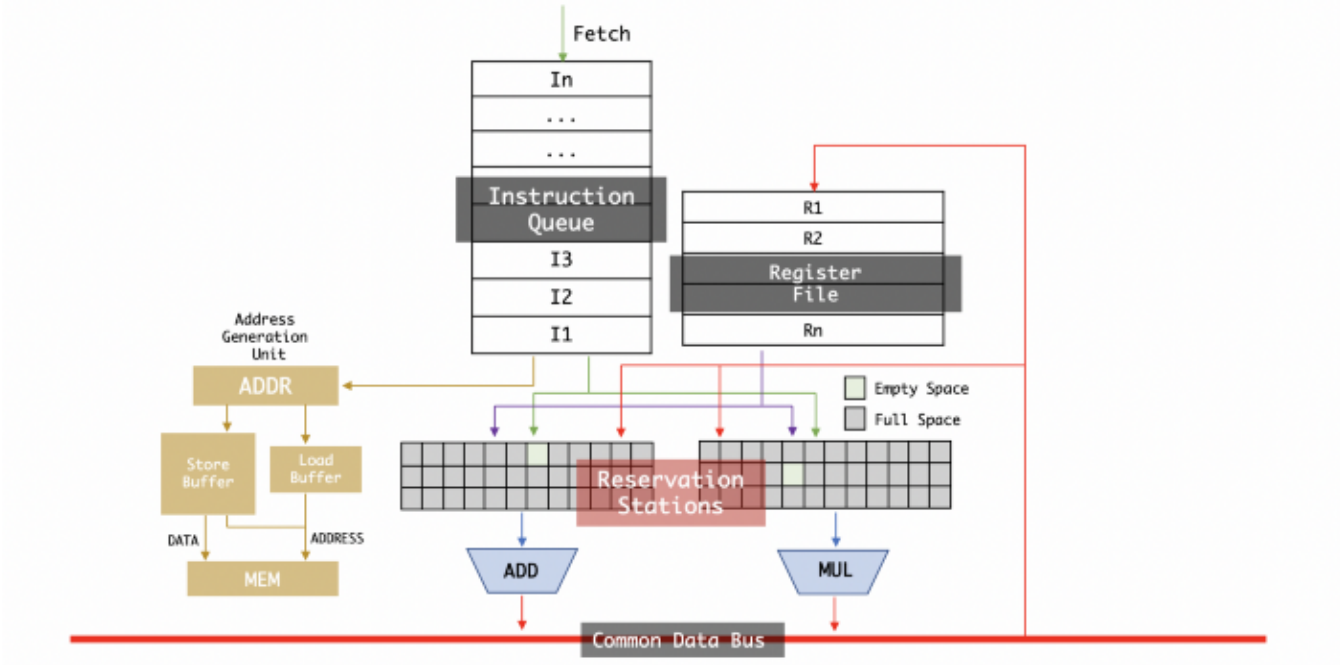


Finally, if the instruction is not an arithmetic instruction, instead if it is a **load** (load from memory to the floating-point register file) or a **store** instruction (store from the floating-point register file to the memory), then the instruction will go to the **address generation unit**. The address generation unit is an integer operation unit so it won't go through the floating-point ALU.

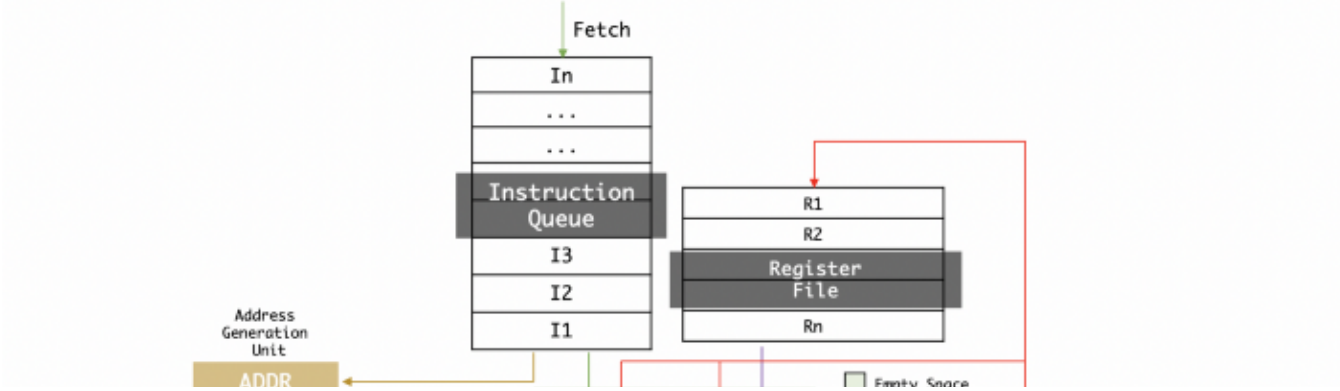


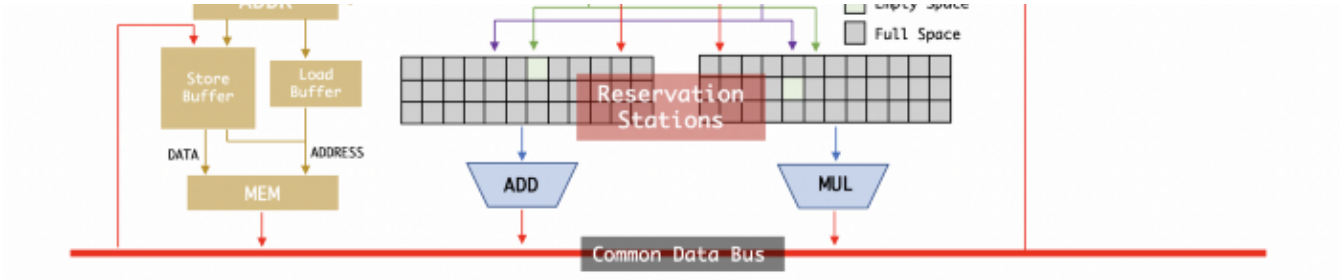


After we compute the address we would like to load/store, then we insert our instruction into a **load buffer** or a **store buffer**. So the instructions will be queued up before they going to the memory. The load buffer only provides a data address to memory, while the store buffer provides both the data and the address to the memory.

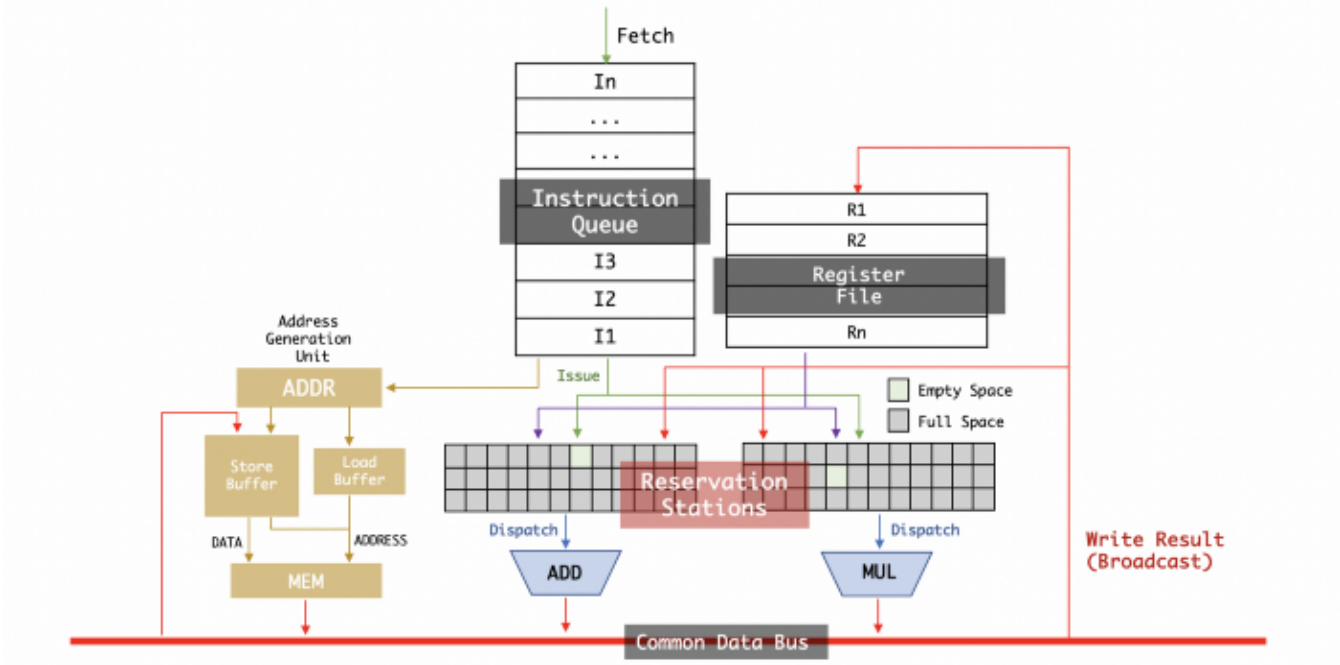


When a load comes back from the memory, its value will be broadcast on the CDB and goes to an appropriate register in the register file. Also, the values that are broadcasted on the bus are going to go to the store buffer so the store instruction can get their value when they are available.





The part when we send the instructions from the instruction queue to the reservation station is called an **issue**. The part when the instruction is finally sent to execution units from the reservation station is called **dispatch**. The part when the instruction is ready to broadcast its result is called **write result** (or simply **broadcast**).



This is the whole picture of Tomasulo's algorithm. Now we are going further into the details of this algorithm.

(4) Issue Phase in Tomasulo's Algorithm

- **Step 1. Take** the next instruction in program order from the instruction queue. This has to be done in order for register renaming to work correctly.
- **Step 2. Determine** where the inputs of the instructions come from. They can come from the **register file** or are they produced by some other instructions that still haven't bought us the result. Also, if we have to **wait for a result**, which is the instruction we are waiting for.

- **Step 3. Get a free/available reservation station** of the correct kind (some reservation stations are for `MUL`, while the others are for `ADD`, etc.)
- **Step 4. Put** instruction in the reservation station.
- **Step 5. Tag** the destination register so that the produced result goes to this corresponding register so that the instructions that want that register in the future will know which instruction is going to produce this value if it has already not been produced

(5) Issue: An Example

Suppose we have the following program,

```
ADD F2, F4, F1
DIV F1, F2, F3
SUB F4, F1, F2
ADD F1, F2, F3
```

and if we are given a processor with the initial register file,

```
Register File =======
F1    3.14
F2   -1.00
F3    2.72
F4    0.71
```

and register station `RS1`, `RS2`, `RS3` link to the `ADD` ALU and register station `RS4`, `RS5` link to the `MUL` ALU. The RAT is empty at the beginning and this means that we can directly grab the data from the register file. Then, initially,

```
==== Inst Queue ====
+------------------+   <- top in
|  ADD F1, F2, F3  |
|  SUB F4, F1, F2  |
|  DIV F1, F2, F3  |
|  ADD F2, F4, F1  |
+------------------+   <- bottom out

======== RAT =======
+------------------+
```

```
|   F1   |         |
+------------------+
|   F2   |         |
+------------------+
|   F3   |         |
+------------------+
|   F4   |         |
+------------------+

======== RF ========
+------------------+
|   F1   |   3.14  |
+------------------+
|   F2   |  -1.00  |
+------------------+
|   F3   |   2.72  |
+------------------+
|   F4   |   0.71  |
+------------------+

================= ADD RS ================
+---------------------------------------+
|   RS1   |         |         |         |
+---------------------------------------+
|   RS2   |         |         |         |
+---------------------------------------+
|   RS3   |         |         |         |
+---------------------------------------+

================= MUL RS ================
+---------------------------------------+
|   RS4   |         |         |         |
+---------------------------------------+
|   RS5   |         |         |         |
+---------------------------------------+
```

After issuing the 1st instruction,

```
==== Inst Queue ====
+------------------+   <- top in
|                  |
|  ADD F1, F2, F3  |
|  SUB F4, F1, F2  |
|  DIV F1, F2, F3  |
+------------------+   <- bottom out

======== RAT =======
+------------------+
|   F1   |         |
+------------------+
|   F2   |   RS1   |
+------------------+
|   F3   |         |
```

```
+------------------+
|   F4   |         |
+------------------+

======== RF ========
+------------------+
|   F1   |   3.14  |
+------------------+
|   F2   |  -1.00  |
+------------------+
|   F3   |   2.72  |
+------------------+
|   F4   |   0.71  |
+------------------+

================= ADD RS ================
+---------------------------------------+
|   RS1   |   ADD   |   0.71  |   3.14  |
+---------------------------------------+
|   RS2   |         |         |         |
+---------------------------------------+
|   RS3   |         |         |         |
+---------------------------------------+

================= MUL RS ================
+---------------------------------------+
|   RS4   |         |         |         |
+---------------------------------------+
|   RS5   |         |         |         |
+---------------------------------------+
```

After issuing the 2nd instruction,

```
==== Inst Queue ====
+------------------+   <- top in
|                  |
|                  |
|  ADD F1, F2, F3  |
|  SUB F4, F1, F2  |
+------------------+   <- bottom out

======== RAT =======
+------------------+
|   F1   |   RS4   |
+------------------+
|   F2   |   RS1   |
+------------------+
|   F3   |         |
+------------------+
|   F4   |         |
+------------------+
```

```
======== RF ========
+------------------+
|   F1   |   3.14  |
+------------------+
|   F2   |  -1.00  |
+------------------+
|   F3   |   2.72  |
+------------------+
|   F4   |   0.71  |
+------------------+

================= ADD RS ================
+---------------------------------------+
|   RS1   |   ADD   |   0.71  |   3.14  |
+---------------------------------------+
|   RS2   |         |         |         |
+---------------------------------------+
|   RS3   |         |         |         |
+---------------------------------------+

================= MUL RS ================
+---------------------------------------+
|   RS4   |   DIV   |   RS1   |   2.72  |
+---------------------------------------+
|   RS5   |         |         |         |
+---------------------------------------+
```

After issuing the 3rd instruction,

```
==== Inst Queue ====
+------------------+   <- top in
|                  |
|                  |
|                  |
|  ADD F1, F2, F3  |
+------------------+   <- bottom out

======== RAT =======
+------------------+
|   F1   |   RS4   |
+------------------+
|   F2   |   RS1   |
+------------------+
|   F3   |         |
+------------------+
|   F4   |   RS2   |
+------------------+

======== RF ========
+------------------+
|   F1   |   3.14  |
+------------------+
|   F2   |  -1.00  |
```

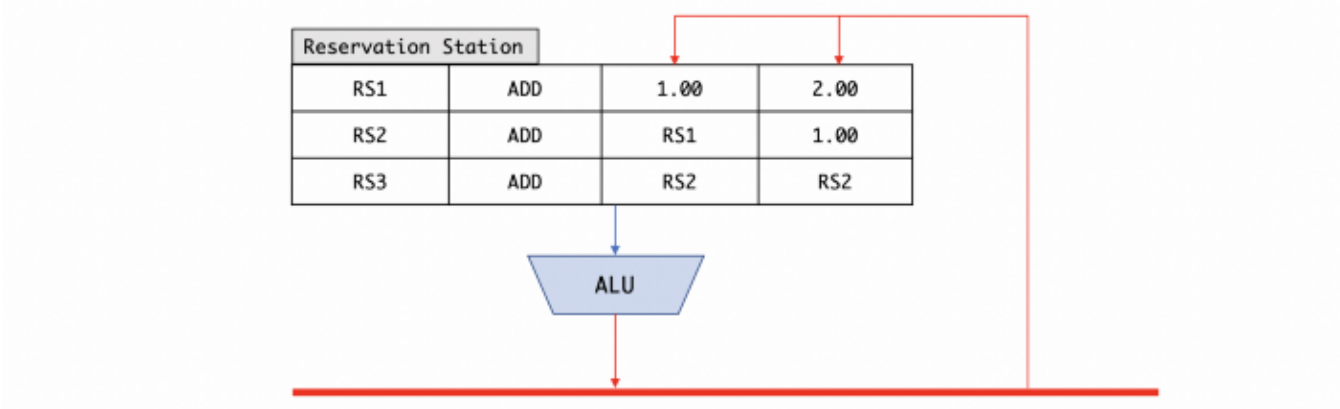
```
+------------------+
|   F3   |   2.72  |
+------------------+
|   F4   |   0.71  |
+------------------+

================= ADD RS ================
+---------------------------------------+
|   RS1   |   ADD   |   0.71  |   3.14  |
+---------------------------------------+
|   RS2   |   SUB   |   RS4   |   RS1   |
+---------------------------------------+
|   RS3   |         |         |         |
+---------------------------------------+

================= MUL RS ================
+---------------------------------------+
|   RS4   |   DIV   |   RS1   |   2.72  |
+---------------------------------------+
|   RS5   |         |         |         |
+---------------------------------------+
```

After issuing the last instruction, there will be no more instructions in the instruction queue. So the final state is,

```
==== Inst Queue ====
+------------------+   <- top in
|                  |
|                  |
|                  |
|                  |
+------------------+   <- bottom out

======== RAT =======
+------------------+
|   F1   |   RS3   |
+------------------+
|   F2   |   RS1   |
+------------------+
|   F3   |         |
+------------------+
|   F4   |   RS2   |
+------------------+

======== RF ========
+------------------+
|   F1   |   3.14  |
+------------------+
|   F2   |  -1.00  |
+------------------+
|   F3   |   2.72  |
+------------------+
```

```
|    F4    |    0.71   |
+------------------+

================ ADD RS ================
+-------------------------------------+
|    RS1    |    ADD    |    0.71   |    3.14   |
+-------------------------------------+
|    RS2    |    SUB    |    RS4    |    RS1    |
+-------------------------------------+
|    RS3    |    ADD    |    RS1    |    2.72   |
+-------------------------------------+

================ MUL RS ================
+-------------------------------------+
|    RS4    |    DIV    |    RS1    |    2.72   |
+-------------------------------------+
|    RS5    |           |           |           |
+-------------------------------------+
```

(6) Dispatch Phase in Tomasulo's Algorithm



Step 1. Find the instructions in the reservation station that can be directly executed.

Step 2. Execute the instruction and **broadcast** the returned value of this instruction to all the two operands.

Step 3. Replace the waiting operands with the result if the broadcasted value matches the value we want to achieve (for example, the broadcasted value is RS1 and we are waiting for the RS1).

Step 4. Loop step 1 to step 3 until all the instructions are dispatched

(7) Dispatch: An Example

Let's continue with the example of issuing. Now, after the 1st dispatching,

```
================ ADD RS ================
+--------------------------------------+
|   RS1   |   ADD   |         |         |       // RS1 = 3.85
+--------------------------------------+
|   RS2   |   SUB   |   RS4   |   3.85  |
+--------------------------------------+
|   RS3   |   ADD   |   3.85  |   2.72  |
+--------------------------------------+

================ MUL RS ================
+--------------------------------------+
|   RS4   |   DIV   |   3.85  |   2.72  |
+--------------------------------------+
|   RS5   |         |         |         |
+--------------------------------------+
```

After the 2nd dispatching,

```
================ ADD RS ================
+--------------------------------------+
|   RS1   |   ADD   |         |         |
+--------------------------------------+
|   RS2   |   SUB   |   1.90  |   3.85  |
+--------------------------------------+
|   RS3   |   ADD   |         |         |         // RS3 = 3.85
+--------------------------------------+

================ MUL RS ================
+--------------------------------------+
|   RS4   |   DIV   |         |         |         // RS4 = 1.90
+--------------------------------------+
|   RS5   |         |         |         |
+--------------------------------------+
```

After the 3rd dispatching,

```
================ ADD RS ================
+--------------------------------------+
|   RS1   |   ADD   |         |         |
+--------------------------------------+
|   RS2   |   SUB   |         |         |          // RS2 = -1.95
+--------------------------------------+
|   RS3   |   ADD   |         |         |
+--------------------------------------+

================ MUL RS ================
+--------------------------------------+
|   RS4   |   DIV   |         |         |
```

```
+---------------------------------------+
|   RS5   |         |         |         |
+---------------------------------------+
```

(8) Dispatch More than 1 Ready

There's another situation of dispatching that we have to consider. Suppose we have the following register stations for an ADD execution unit,

```
================ ADD RS ================
+---------------------------------------+
|   RS1   |   ADD   |   0.71  |   3.14  |
+---------------------------------------+
|   RS2   |   SUB   |   1.90  |   RS1   |
+---------------------------------------+
|   RS3   |   ADD   |   RS1   |   2.72  |
+---------------------------------------+
```

Then after the first dispatch, we can have 2 instructions `RS2` and `RS3` ready for execution.

```
================ ADD RS ================
+---------------------------------------+
|   RS1   |   ADD   |         |         |     // RS1 = 3.85
+---------------------------------------+
|   RS2   |   SUB   |   1.90  |   3.85  |
+---------------------------------------+
|   RS3   |   ADD   |   3.85  |   2.72  |
+---------------------------------------+
```

But meanwhile, because we have only one ALU for these instructions and the ALU can only execute 1 instruction per cycle. So we have to decide which one to dispatch next.

Normally, we want to choose the one that leads to the best performance, which means that if many following instructions depending on `RS2` and only a few instructions depending on `RS3`, then we have to choose `RS2` to be executed first for better performance. However, this can not be possible for a processor because the processor does a really bad job in predicting future instructions.

There are actually some heuristics even though we can not do a perfect job of this. One choice is to make the **oldest first**. So whichever instruction that has been sitting in the

register station longer will be allowed to go first. This is because, for an older instruction with all other things being equal, it is more likely that more instructions are by now waiting for it.

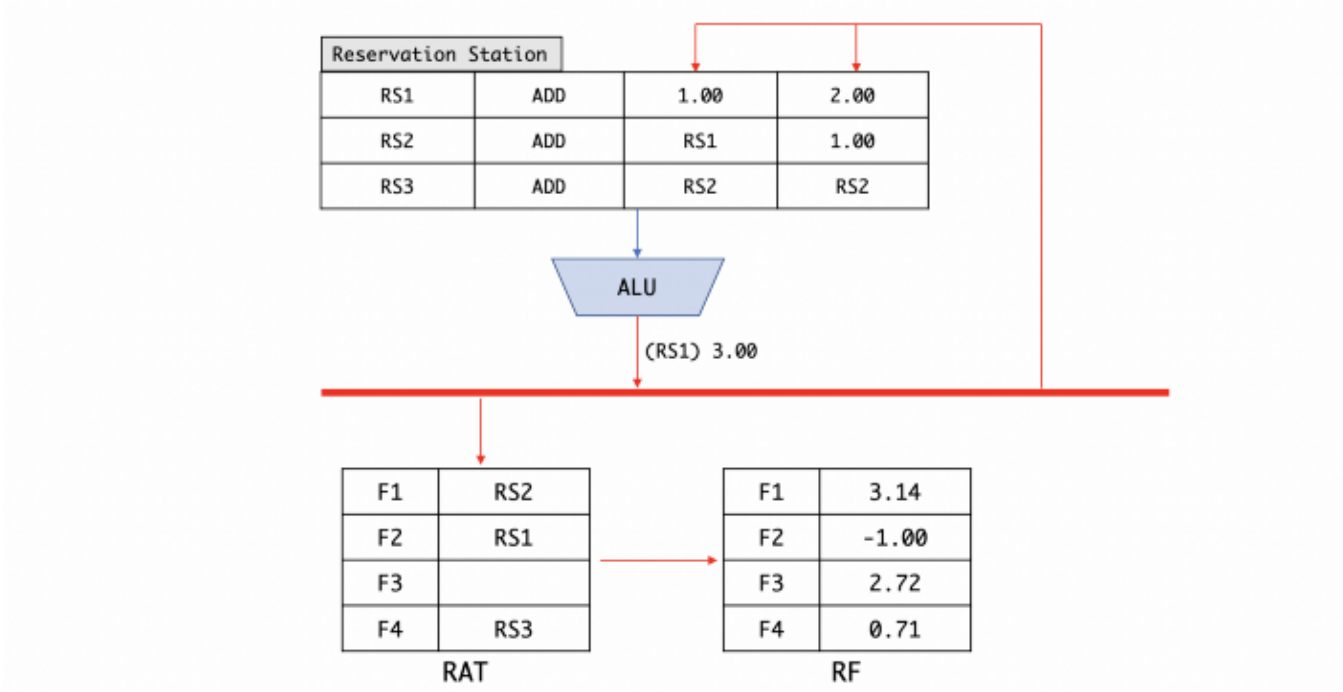
Another one is typically is to make the **most dependencies first**, where we will check how many instructions in the register station needs one of these and then use the one. This means that we are going to execute the one that can free up most of the other instructions. The downside of this approach is that it requires us to search for a lot of stuff and this can be power-hungry.

The last heuristic is to make a **random** execution. Normally, we will choose the **oldest first**, because it is a compromise between the most dependencies first (cares about effectiveness but no power consumption) and random (cares about the power consumption but no effectiveness).

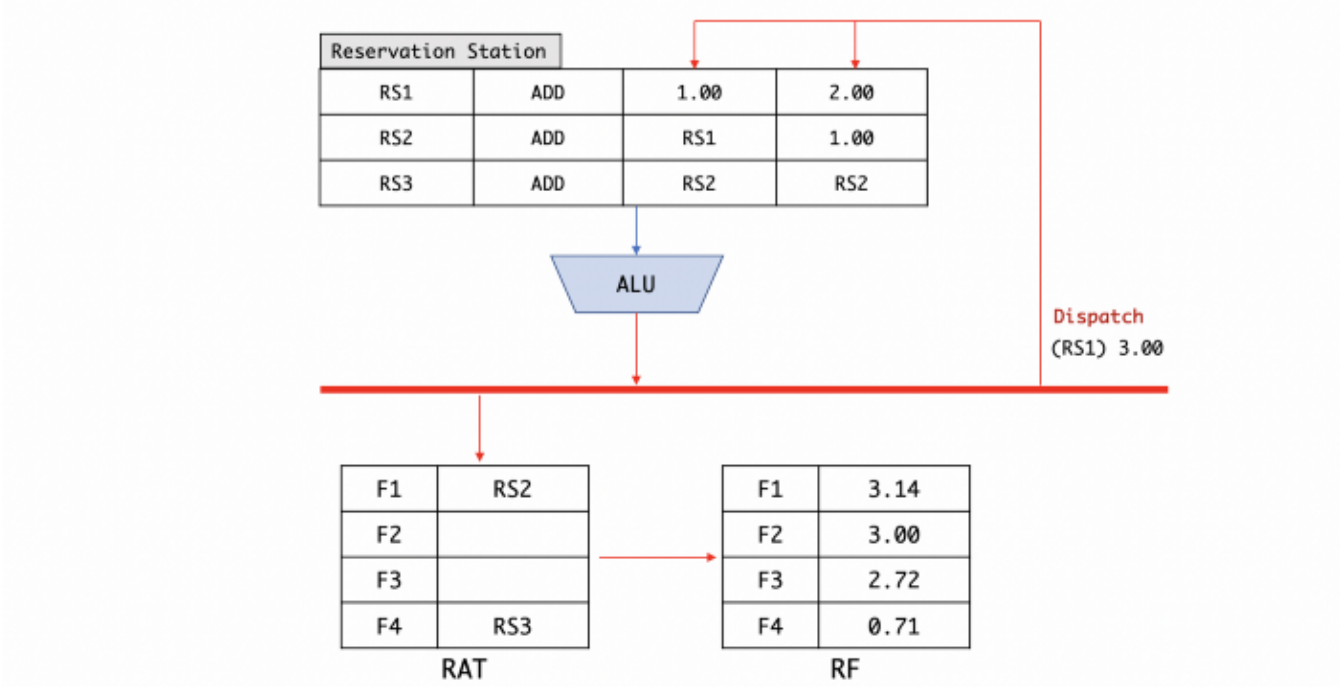
(9) Write Result (Broadcast) Phase in Tomasulo's Algorithm

So what will happen if we successfully executed an instruction in the ALU. Then possibly we will get a result of this instruction attached with a tag (i.e. `RS1` , `RS2` , etc.). Thus,

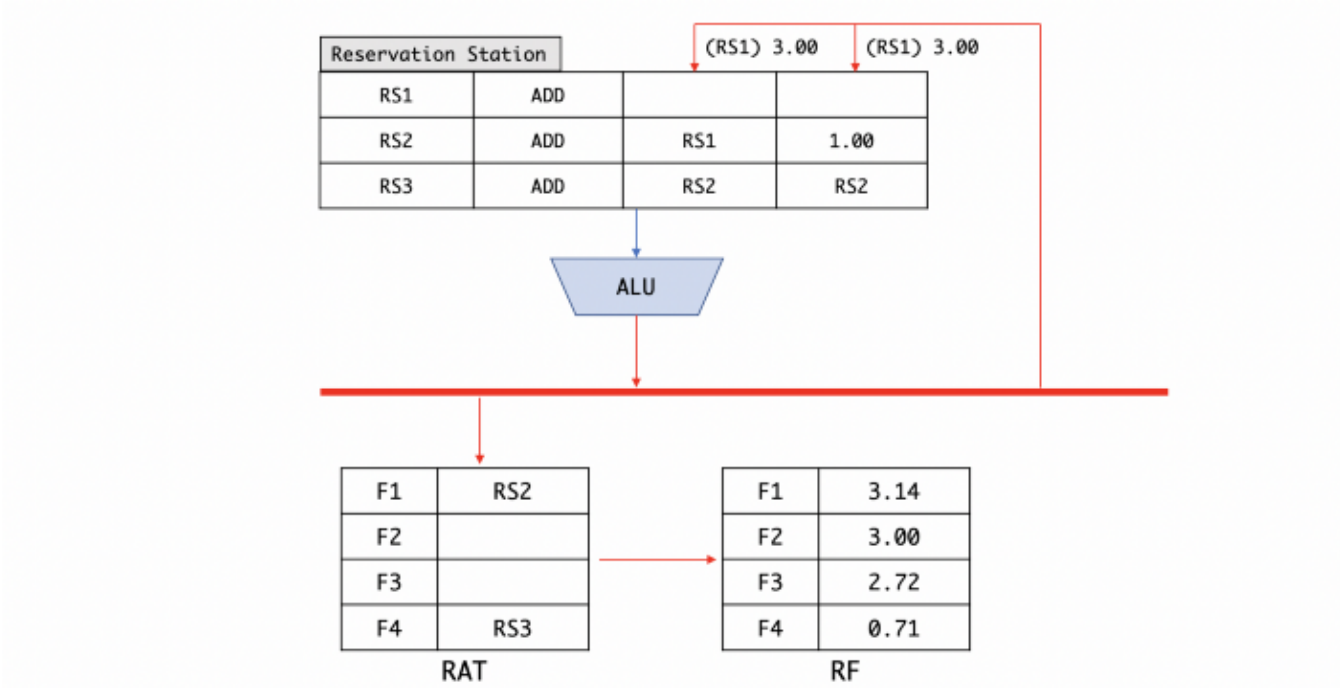
Step 1. Put the **tag** and the **result** on the **bus** (common data bus or CDB) so it can be broadcasted to all the other parts.



Step 2. Write to **register file (RF)** based on the **register address table** (RAT). And then **update the RAT** by changing the corresponding entry to empty in order to make it point to the register file.

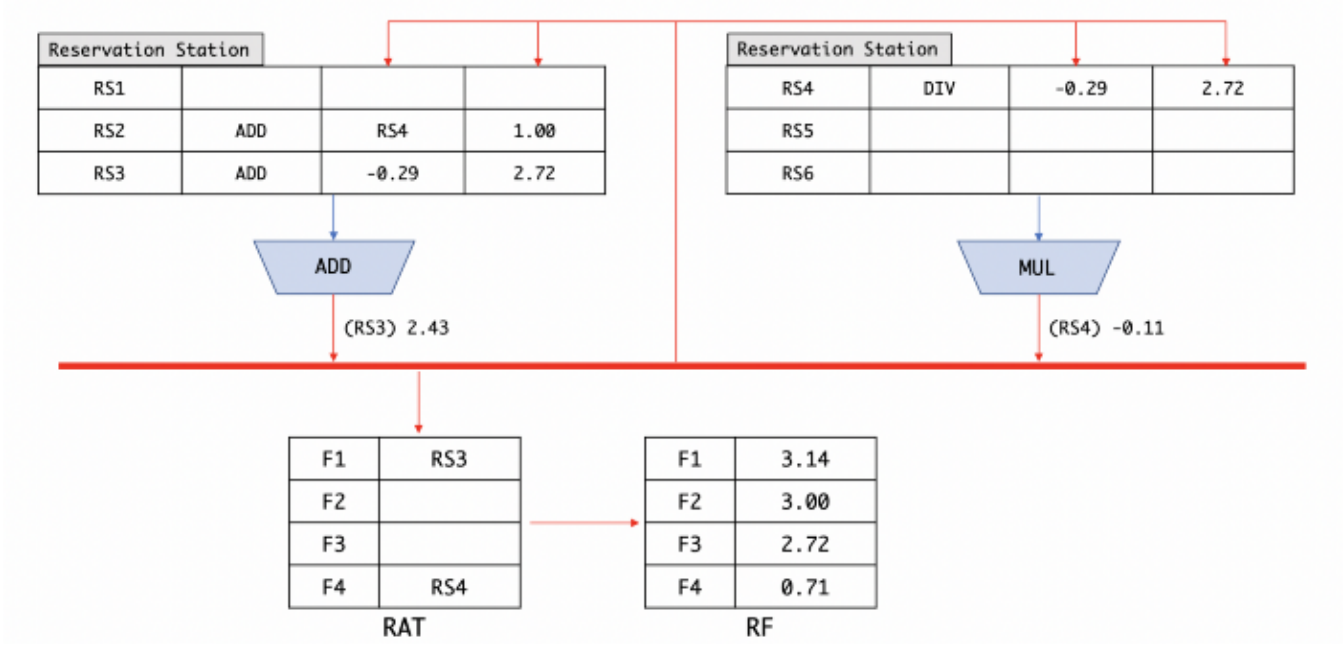


Step 3. Free the register station (RS).



(10) Broadcast More than 1 Result

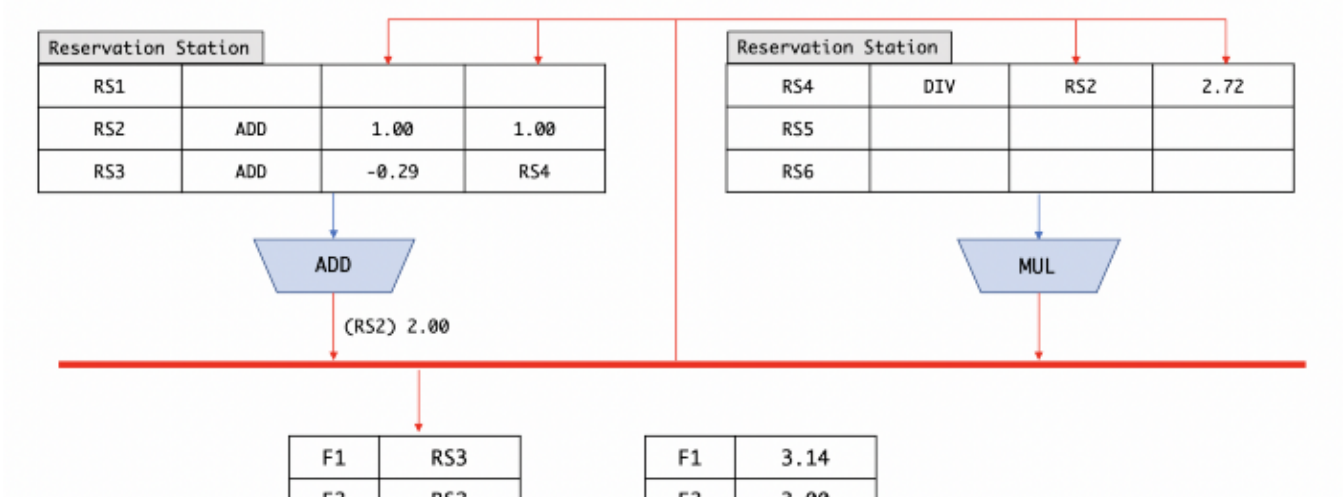
This can happen if we have several ALUs. Let's say that we have two ALUs, one is for `ADD` and the other is for `DIV`.

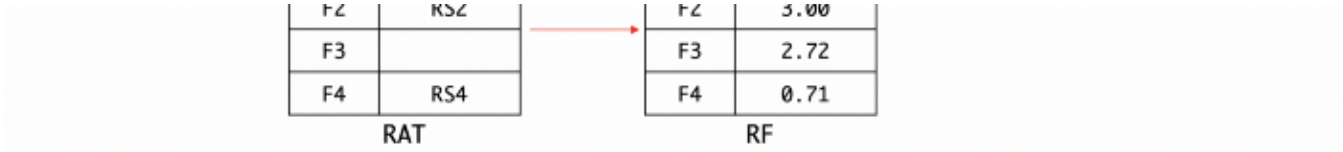


So now we have two tag-result pairs waiting for broadcasting. One is `(RS3) 2.43` and the other is `(RS4) -0.11`. So which one will we broadcast in the first place? One common **heuristic** is that if one of the units is known to be slower than the other, then we will first broadcast the slower one. Let's say the multiplication or division operations are usually slower than the addition or subtraction operations which is usually the case, then what we are going to do is given the priority to the slower unit. This is because the slower unit dispatches in the first because it takes longer to execute and this also means there can usually be more instructions waiting for the result of this unit. Thus, the result we are going to broadcast in the first place is `(RS4) -0.11` and then `(RS3) 2.43`.

(11) Broadcast Stale Result

So before we begin, let's see what's a stale result. At the beginning point, suppose we have,

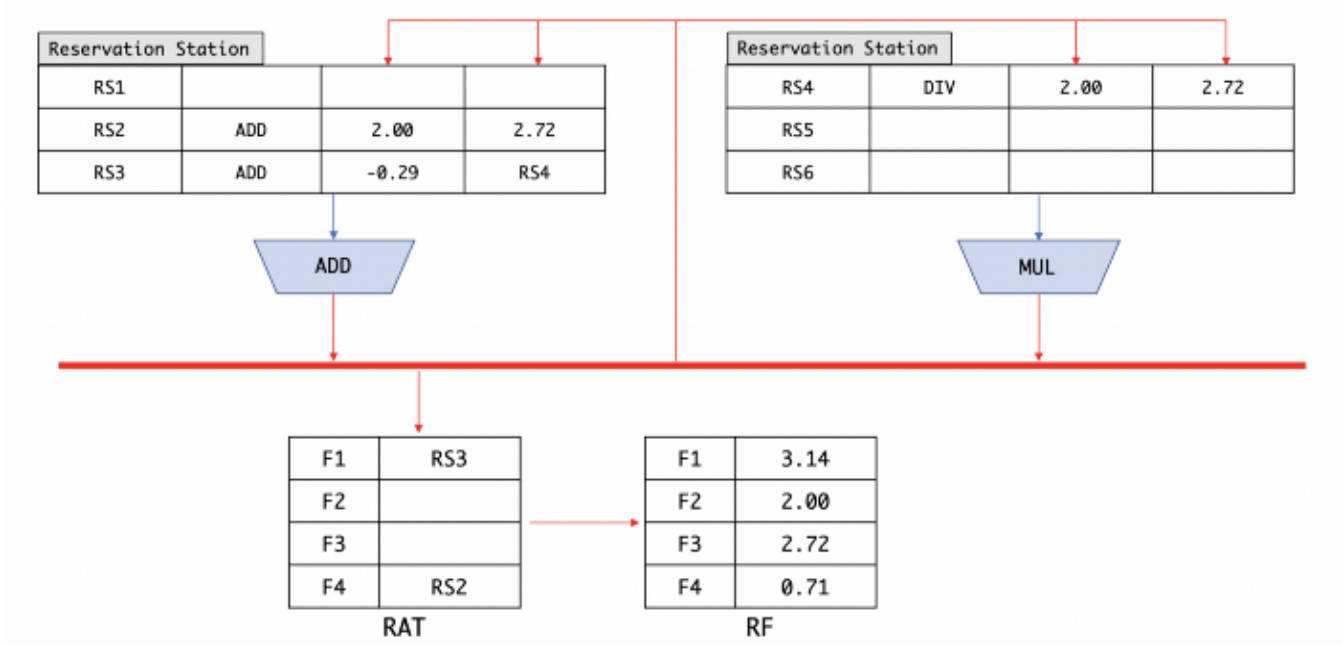




Then `(RS2) 2.00` is going to be broadcasted to all the other parts. Meanwhile, we are going to issue the next instruction (suppose it is issued to `RS2`), which is,

```
ADD F4, F2, F3
```

After issuing this instruction, the entry value of F4 in the RAT will be replaced by the new `RS2`,



After dispatching, we have,

Based on the **slower broadcast heuristic**, we have to broadcast `RS4` in the first place. But it is quite strange that we haven't got a tag `RS4` in the RAT because it has already been replaced by `ADD F4, F2, F3`. So we call the `(RS4) 0.74` a **stale result**.

But our question is that what will happen if we broadcast this result.

For filling the reservation stations, it is going to be done normally. So the operands with a `RS4` will be replaced by our broadcasted result.

However, for the register file and RAT, we actually do nothing. We don't have to update this result to the RF and RAT not only because it does not exist in the RAT, but also because that the instructions come later will not use this value. Instead, when they refer to `F4`, they are actually calling `RS2`. The instructions that need the results of `RS4` have already been pushed into the reservation stations and after this broadcasting, the operand `RS4` will all be replaced by the result. So `RS4` will have no other usages in this case.

(12) Review of Tomasulo's Algorithm

For one instruction,

- Take this instruction from the instruction queue

- Issue this instruction to the register station
- Rename the operands by RAT
- If empty, get the register value from RF
- Capture the broadcasted result from the other executed instruction
- Check all the operands that we need is existing
- Dispatch the instruction to the execution unit
- After the execution, write the result to the common data bus
- Broadcast the result to the reservation station
- Free the reservation station corresponding to the current instruction
- Broadcast the result to the RAT
- If no corresponding tag in the RAT, abandon this result and finish
- If there's a corresponding tag, update the value in RF with respect to RAT
- Then empty this tag in the RAT and finish

Note that these are what happens for 1 instruction. However, in practice, during a particular cycle, some instructions might be issued, some instructions are waiting to capture their operands, some instructions are trying to be dispatched, and finally, some instructions are trying to write their results.

(13) Multiple Phases for Tomasulo's Algorithm

Because all of these things can happen every cycle, there are some interesting things that we need to consider,

- Can we dispatch an instruction immediately after issuing it if it doesn't need to capture any results? The answer is typically **no** because we have to check which one to dispatch because there can be many instructions waiting for dispatching (see **dispatch more than 1 ready**). However, it can be possible but we are not going to discuss it now.
- Can we dispatch an instruction immediately after it captures the value it needs from the broadcasted result? The answer is typically **no** because also we have to

check which one to dispatch and this must happen in the next cycle. But again, it can be possible but we are not going to discuss it now.

- Can we update the RAT when there is an issuing and a writing result happen simultaneously? We have known that we can update the RAT during the renaming (part of issuing) and we can also update the RAT when we broadcast a result. If these things happen simultaneously, then it is possible that RAT will be written twice in turn. The answer is typically yes for this problem, but we have to keep in mind that we have to do result writing first and then issuing. This is because we want to keep the issuing result in the RAT, not the value after the result writing so that the later instructions will rename with the newly issued tag instead of the old one.

3. What's Next?

In this section, we have discussed how does Tomasulo's Algorithm deal with the floating-point instructions. In the next section, we are going to continue with Tomasulo's algorithm and see what will happen for load & store instructions.

Computer Science Computer Architecture Tomasulo Out Of Order