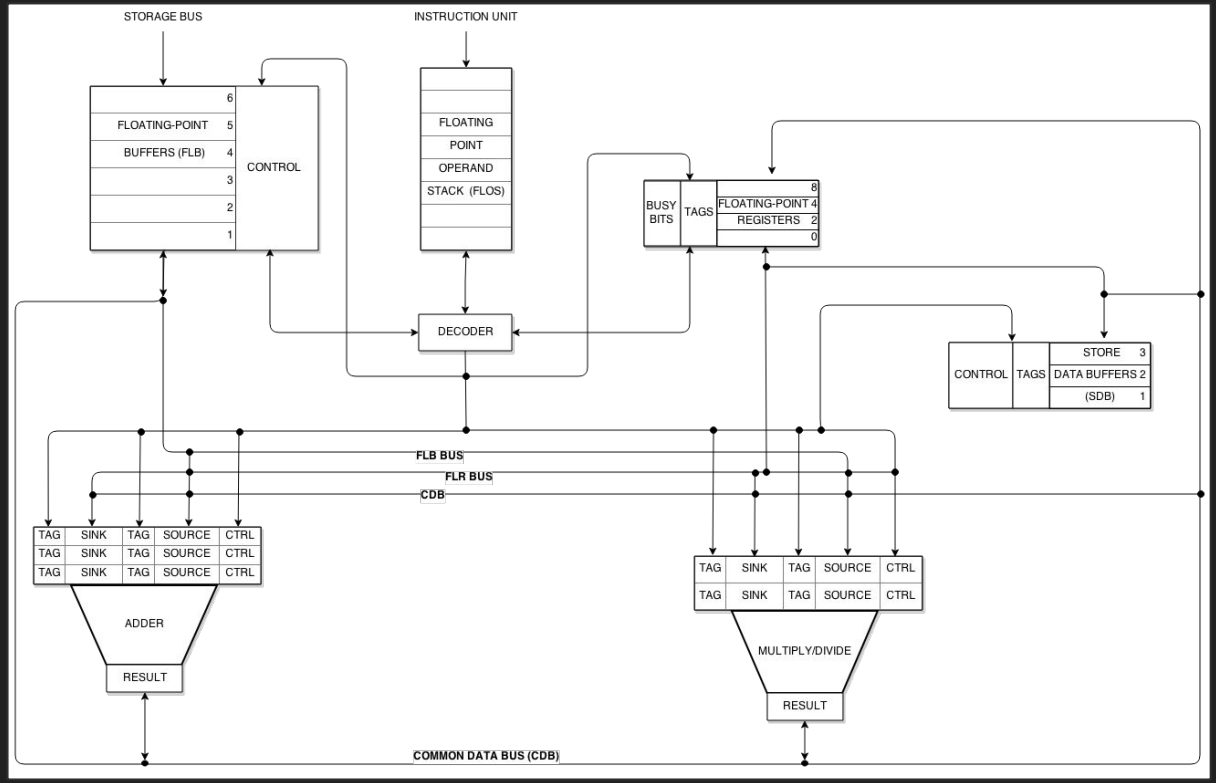
Tomasulo’s Algorithm

By Buzz Pendarvis

Overview

Tomasulo's algorithm is a hardware algorithm used to implement Out Of Order Execution (OoOExe) in high performance CPUs.

That's a lot of jargon that won't make sense unless you already know a lot about CPU design, so let's break it down.



https://commons.wikimedia.org/wiki/File:Tomasulo_Architecture.png

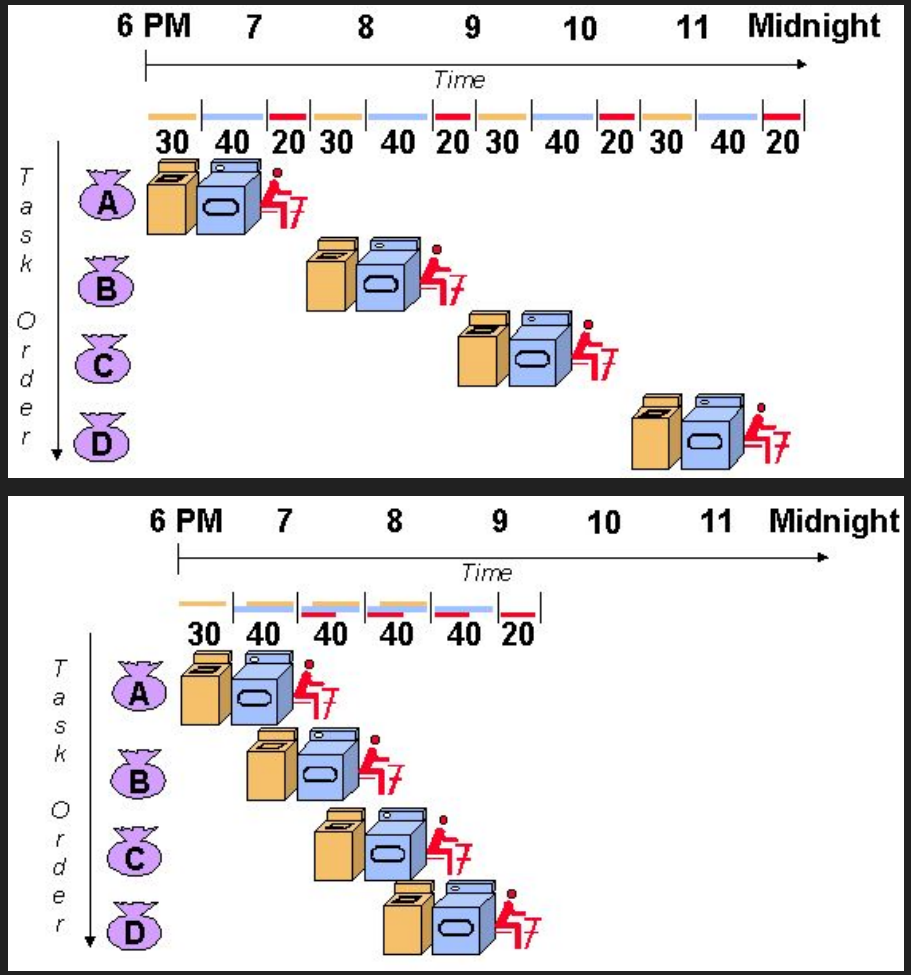
In-Order Execution

Before going into Tomasulo's algorithm, we need to understand the problem that it solves.

The goal of a high-performance CPU core is to execute one stream of instructions as quickly as possible. To achieve this, many processors utilize "instruction-level parallelism," meaning they process many instructions simultaneously.

(not to be confused with multi-threading)

To get the most instruction-level parallelism, CPU designers have to consider data dependencies, most notably…



Read After Write (RAW)

Also called a flow dependency or true dependency, RAW dependencies occur when an instruction requires the result of a previous instruction.

In the example code to the right, the second line cannot execute until the first line has finished, since the second line relies on the value of `a`.

This also means these instructions can’t run in parallel, and an in-order processor would have to stop what it’s doing and wait for the first line to finish before moving on to the second.

```
a = 9 + 10;
b = a + 2;
```

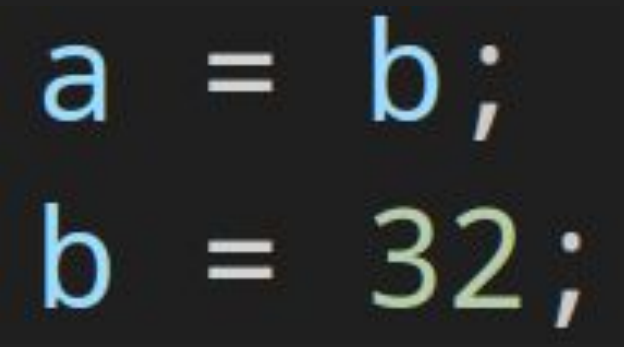
Write After Read (WAR)

Also called a false dependency or a name dependency, a WAR data dependency occurs when an instruction writes to a location whose value is required by a previous instruction.

In the example to the right, these two lines can’t run in parallel because the second line destroys the data required by the first line.

This is a false dependency because, in every case of this dependency, it can be eliminated by simply renaming the variable on the second line.

CPUs have limited registers, though, so register renaming in software isn’t very practical.

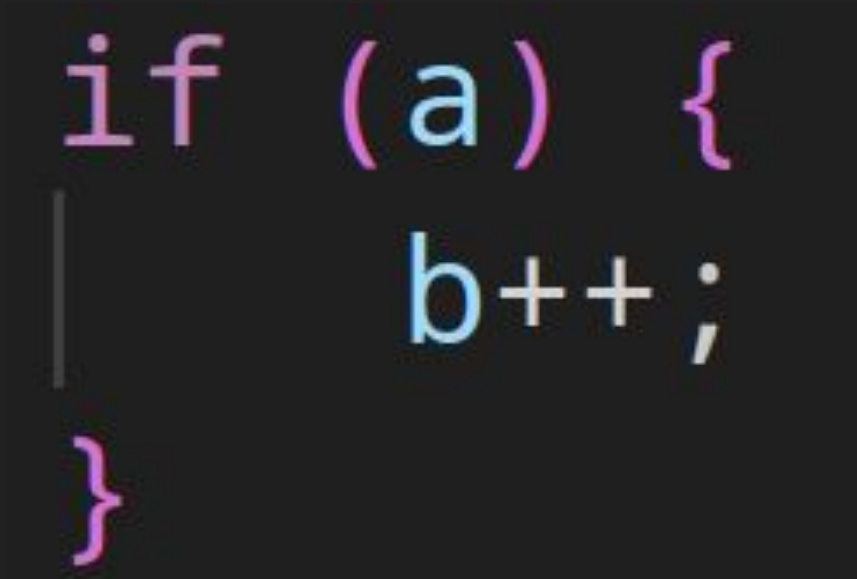


Control Dependency

A control dependency occurs when the CPU encounters a branch. The majority of in-order processors can’t keep fetching and executing instructions until the branch has been resolved.

In the example to the right, the CPU can’t determine if it should execute the second line until the condition on the first line is calculated.

This leads to wasted cycles in the majority of in-order processors.



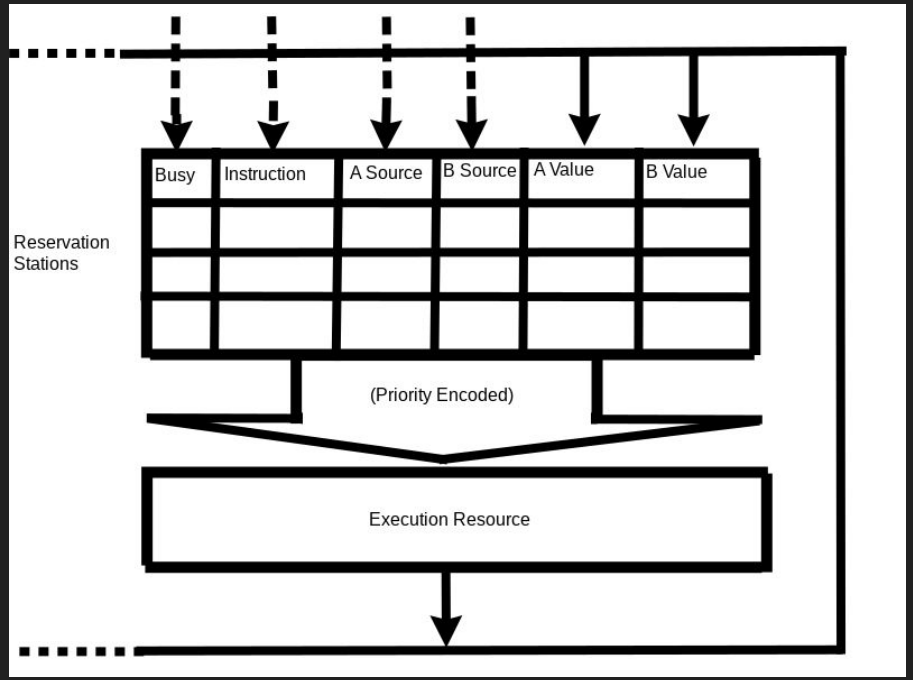
How does Tomasulo’s algorithm work?

Tomasulo’s algorithm elegantly improves performance in all three of these cases.

RAW dependencies are mostly unavoidable, so OoOExe CPUs use the otherwise wasted time to calculate the results of later instructions that don’t depend on unavailable data.

Tomasulo’s algorithm achieves this with “Reservation Stations”, which store instructions with their parameters, and execute them only after their parameters are available.

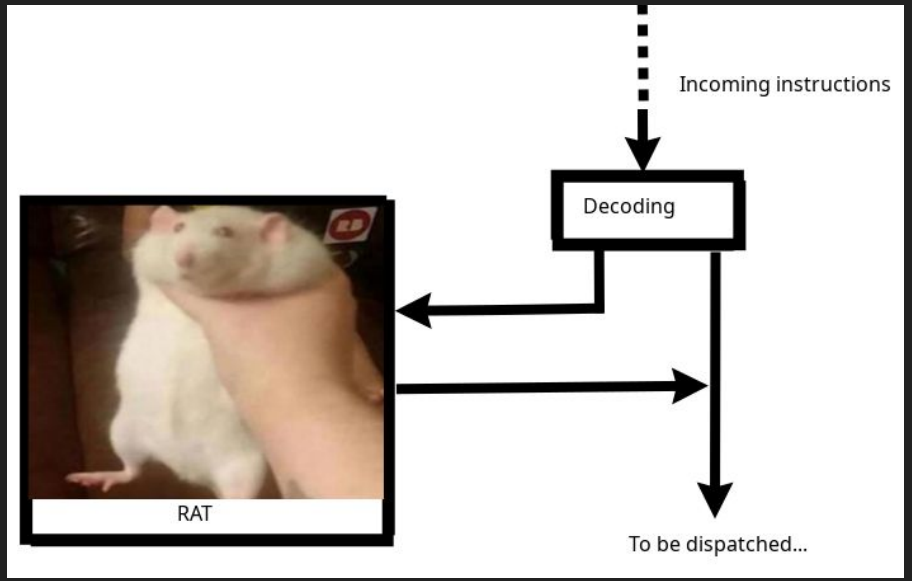
Reservation stations all listen on a “Common Data Bus” for calculated values that they depend on.



How does Tomasulo’s algorithm work?

WAR dependencies are completely eliminated through register renaming. A Register Allocation/Alias Table (RAT) keeps track of where dispatched instructions should look for the most recent copy of the data.

The RAT will remember if a register’s value was retired to the actual register file and is thus already available, or it remembers which reservation station / ROB entry holds the instruction that will calculate its new value.

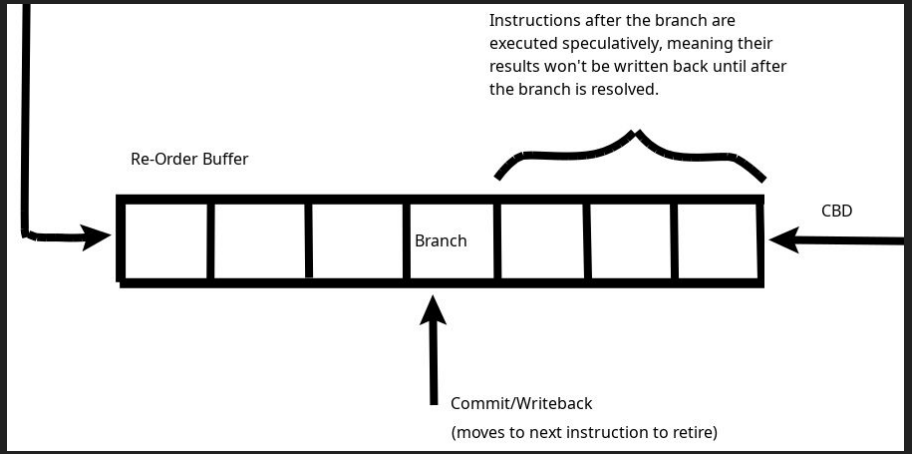


How does Tomasulo’s algorithm work?

Control dependencies can’t be solved outright, but their significance can be reduced via speculative execution.

Upon reaching a branch instruction, the CPU guesses where it should branch to and continues execution there. If it realizes that it mispredicted, it can simply throw away the results of the speculatively executed instructions.

While this technically isn’t part of Tomasulo’s original algorithm, it’s frequently added via the inclusion of a Re-Order Buffer (ROB) which keeps track of the original order of instructions and which instructions are being executed speculatively in a circular queue.



Demo - C++ emulator

I didn’t implement a fully functional processor, but I implemented enough to demonstrate basic instruction reordering.

In this emulator, execution units take 4 cycles to complete, while everything else only takes one.

On the top, you see the program i fed the computer.

On the bottom, you see the order in which the instructions were actually fed through the execution units and with what parameters. (it doesn’t show their destinations, sorry)

Note that in the program, the subtraction occurs right after the addition, but the subtraction actually ends up being calculated after the

I hope to finish this in the future.

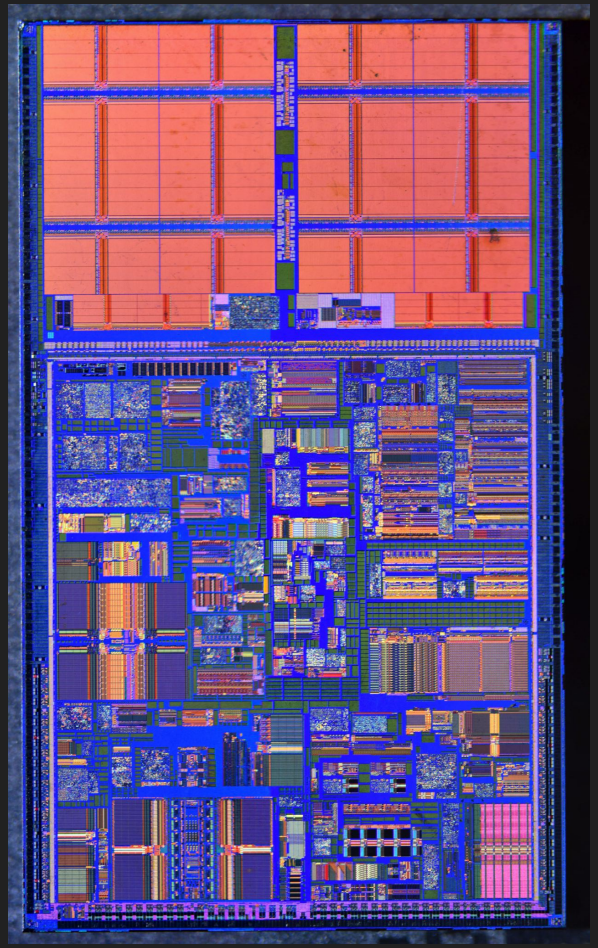
```
cpu.memory[0] = 0b0100001000001001; // LDI r2, 9
cpu.memory[1] = 0b0100001100001010; // LDI r3, 10
cpu.memory[2] = 0b0000000100110010; // ADD r1, r2, r3
cpu.memory[3] = 0b0001000100110001; // SUB r1, r1, r3
cpu.memory[4] = 0b0100010100000000; // LDI r5, 0
```

```
LDI 9
LDI 10
ADD 9 + 10
LDI 0
SUB 19 - 10
```

Relevance

Most every high-performance processor, such as those in all of your desktops, laptops, and even phones, make use of some form of Tomasulo's Algorithm. Typically, these processors also make use of superscalar execution, which is dispatching multiple instructions simultaneously, but even these use and build on Tomasulo's method for instruction reordering and register renaming.

This is a primary example of how advanced discrete mathematics affects your everyday life, as it affects the performance of most every digital system you interact with.



That’s it :)