Project 2: SuperScalar Out-of-Order Processor

This report serves to illustrate the design of a SuperScalar, out-of-order processor using Tomasulo's algorithm. It demonstrates concepts like multiple instruction issue, out-of-order execution, operand look ahead and register renaming.

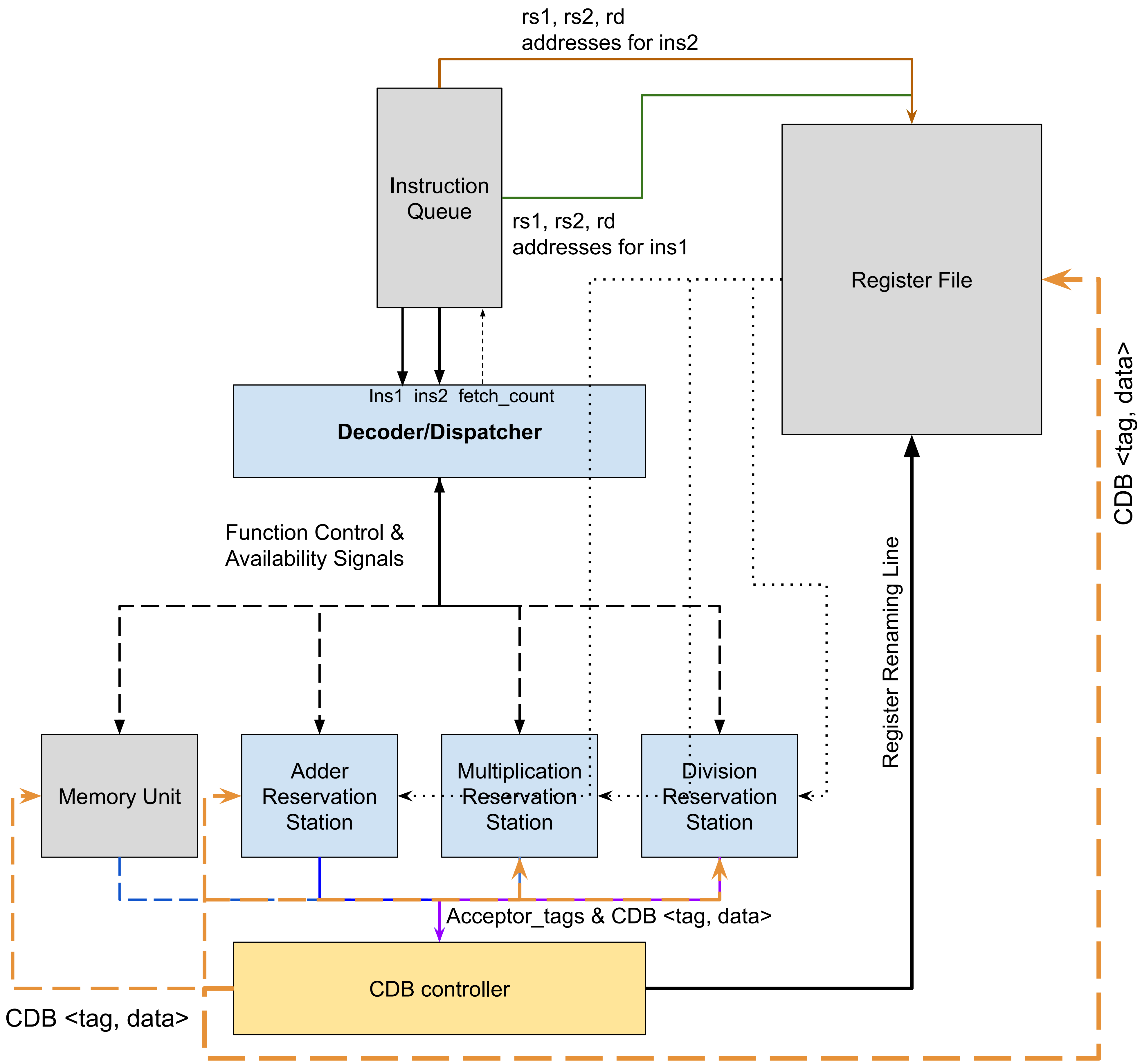
The simulation results of the implementation are shown as a means of verification.

1. Intent of Design

The project implements a Super Scalar Tomasulo's machine for the RISC-V arithmetic and memory instructions. In order to achieve a high IPC, we must make sure that the data routing does not become a bottleneck. To ensure that, I have made a 4 operand wide **Common Data Bus (CDB)**, which can route the outputs of a memory unit, and 3 functional units at once.Their resolution to the data sinks is also done in parallel and occurs in a single clock cycle. Any other way of doing this runs the risk of reducing the throughput gains from superscaling.

To support this, the register file has 4 write ports, and each reservation station listens for the required tag on CDB.

Block diagram of the whole system is on the next page.



rs1, rs2, rd
addresses for ins2
Instruction Queue
Register File
rs1, rs2, rd
addresses for ins1
Ins1 ins2 fetch_count
Decoder/Dispatcher
CDB <tag, data>
Function Control &
Availability Signals
Register Renaming Line
Memory Unit
Adder Reservation Station
Multiplication Reservation Station
Division Reservation Station
Acceptor_tags & CDB <tag, data>
CDB controller
CDB <tag, data>

2. Features implemented

Baseline :

1. **Dispatch unit capable of handling two instructions per cycle.**
2. **RISC-V ISA instructions: Add, mul, load, store, nop and div.**
3. **Support for three functional units: Adder (4 cycles), multiplier (6 cycles), divider (14 cycles).**
4. **RISC-V ISA decoding:** As a proof, I assembled the assembly code via a RISC-V assembler, and pasted the results as verilog for the instruction queue. Details shown in the test section.
5. **Instructions pre-stored in the instruction queue, with simulation ending when instructions run out.**
6. **In-order instruction dispatch, with stalling of the second dispatch unit if the first stalls:** Of course.

Target :

1. **Efficient handling of the instruction queue, including the capability to shift one or two instructions based on decoding.** Working shown in testbench.
2. **Out-of-Order Tomasulo's algorithm**: Implemented a wide Common Data Bus (CDB).
3. **Incorporation of hazard detection and resolution mechanisms to manage dependencies and stalls effectively:**

   I have managed following important hazards and stall conditions:

- **SuperScalar instruction destination conflicts** (details in test section)
- **Same Cycle CDB resolution** (details in test section)
- **Reservation station unavailable stall**

Reach:

1. **Enhancements to the dispatch unit for improved throughput:** Added a 4 <tag, data> pair wide CDB, that can handle 4 resolutions in a single cycle. Also added a **division** functional unit.
2. **ISA extended to support division.** I chose division because its operand ordering is very similar to the multiplication, and hence can be simply decoded. Floating point operand ordering is a bit of a mess.

3. Test Cases and Simulation Results

To test the operation of my RISC-V out-of-order SuperScalar machine, I have tried to come up with the following comprehensive test which demonstrates all the essential design goals of this project, and then goes on to show that the design can also handle hazards, corner cases and race conditions like:

- **SuperScalar instruction destination conflicts:** *Resolution of cases where two instructions instructions have interdependencies, e.g., instruction 2 uses the result of instruction 1 as an operand. The superscalar design postpones the second instruction for the next cycle in this case.*
- **Recursive Register Renaming:** *Renaming a pre-renamed register without fault*
- **Same cycle CDB resolution and incoming tag race condition conflict:** *Suppose a corner case where an incoming register tag for an ALU has its resolution in a tag/data pair that is on the CDB in exactly the same cycle. Correct resolution is to override the incoming tag and accept CDB data as input*

Test Program in C:

```
extern long int* a;
extern long int* w;

long int weighted_average() {
    long int x = (w[0] * a[0] + w[1] * a[1] + w[2] * a[2]);
    long int y = x / (w[0] + w[1] + w[2]);
    return y;
}
```

The above program performs a **weighted average** operation, i.e.;

$$y = \frac{w_0 * a_0 + w_1 * a_1 + w_2 * a_2}{w_0 + w_1 + w_2}$$

Using the online Godbolt compiler with RISC-V option (in addition to some subsequent common sense), one can achieve the following RISC-V assembly program:

Test Program in ASM:

```
lw      x1, 11(x0)      # w0
lw      x2, 15(x0)      # w1
lw      x3, 22(x0)      # w2
lw      x4, 45(x0)      # a1
lw      x5, 62(x0)      # a2
lw      x6, 79(x0)      # a3

mul     x10, x1, x4   # w0 * a0
mul     x11, x2, x5   # w1 * a1
```

```
add     x16, x10, x11 # (w0 * a0) + (w1 * a1)
mul     x12, x3, x6   # w2 * a2
add     x16, x16, x12 # (w0 * a0) + (w1 * a1) + (w2 * a2)

add     x17, x1, x2   # w0 + w1
add     x17, x17, x3  # w0 + w1 + w2

div     x16, x16, x17 # (w0 * a0 + w1 * a1 + w2 * a2) / (w0 + w1 + w2)
```

Test Program Machine Code as a Verilog Snippet:

Using the online compiler from https://riscvasm.lucasteske.dev/#, one can generate a RISC-V compatible machine code from the above assembly code. The machine code looks like as follows:

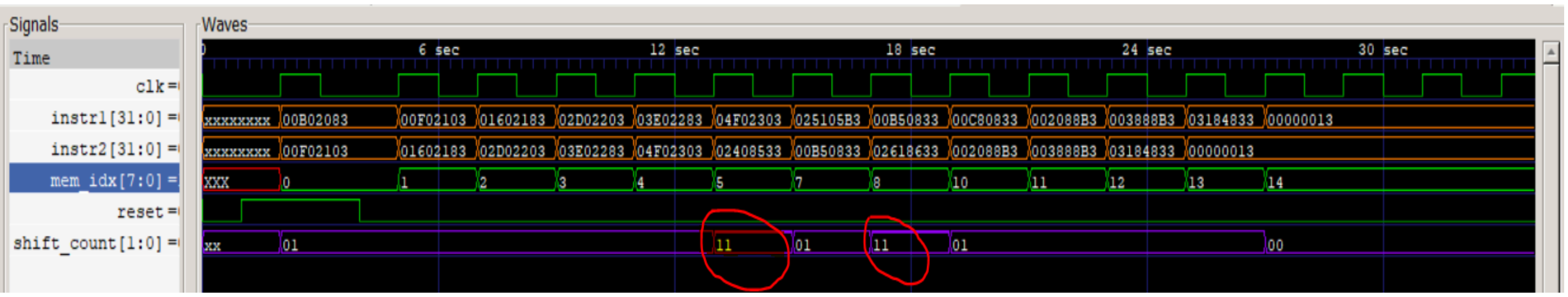
```
instr_mem[8'h0] = 32'h00b02083;         // lw           x1, 11(x0)
instr_mem[8'h1] = 32'h00f02103;         // lw           x2, 15(x0)
instr_mem[8'h2] = 32'h01602183;         // lw           x3, 22(x0)
instr_mem[8'h3] = 32'h02d02203;         // lw           x4, 45(x0)
instr_mem[8'h4] = 32'h03e02283;         // lw           x5, 62(x0)
instr_mem[8'h5] = 32'h04f02303;         // lw           x6, 79(x0)

instr_mem[8'h6] = 32'h02408533;         // mul          x10, x1, x4
instr_mem[8'h7] = 32'h025105b3;         // mul          x11, x2, x5
instr_mem[8'h8] = 32'h00b50833;         // add          x16, x10, x11
instr_mem[8'h9] = 32'h02618633;         // mul          x12, x3, x6
instr_mem[8'ha] = 32'h00c80833;         // add          x16, x16, x12

instr_mem[8'hb] = 32'h002088b3;         // add          x17, x1, x2
instr_mem[8'hc] = 32'h003888b3;         // add          x17, x17, x3

instr_mem[8'hd] = 32'h03184833;         // div          x16, x16, x17
```

I also wrote a python script (`asm_converter.py`) that generates verilog snippets from machine code. The decoder written for this project (`dispatch_and_decode_unit`) is according to the RISC-V spec, and seems to handle these instructions correctly.

Execution Traces:

Due to the out-of-order nature of the processor, I have opted to show small snippets of traces from the perspective of different modules inside the processor. I'll mark important timestamps in the waveform for reference across traces of different modules.

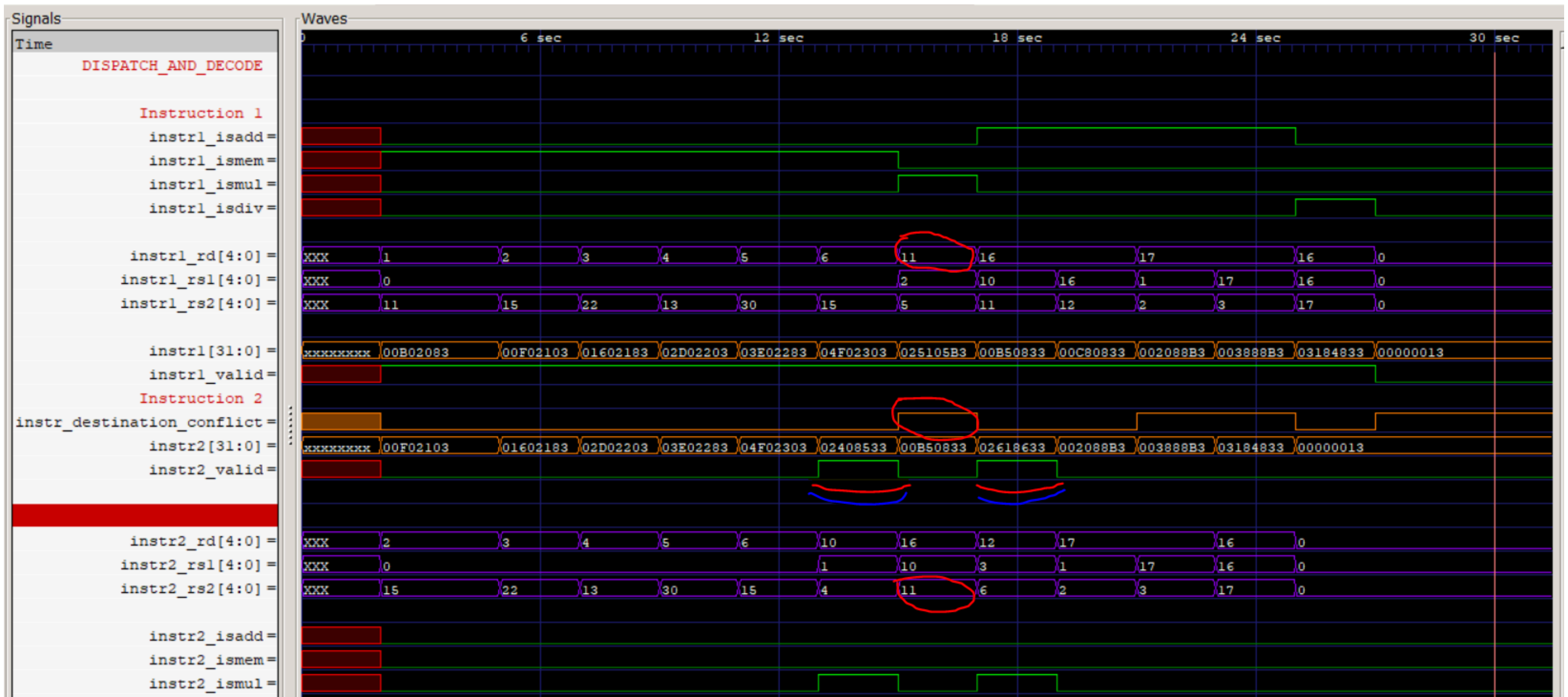
1. Instruction Queue Trace


Signals
Waves
Time
clk
instr1[31:0]
instr2[31:0]
mem_idx[7:0]
reset
shift_count[1:0]
6 sec
12 sec
18 sec
24 sec
30 sec
xxxxxxxx 00B02083 00F02103 01602183 02D02203 03E02283 04F02303 025105B3 00B50833 00C80833 002088B3 003888B3 03184833 00000013
xxxxxxxx 00F02103 01602183 02D02203 03E02283 04F02303 02408533 00B50833 02618633 002088B3 003888B3 03184833 00000013
XXX 0 1 2 3 4 5 7 8 10 11 12 13 14
xx 01 11 01 11 01 00

Our instruction queue has 14 instructions in total. At every clock cycle, a max of 2 instructions can be issued (superscalar issue). However, this specific program has significant interdependence between every two consecutive instructions, which hinders superscalar issue. This is why the `mem_idx` signal increments by only 1 for most cycles. In these cases, the instr2 is invalid; it becomes instr1 in the next cycle.

For two cycles, the `shift_count` is **2'b11**. For these cycles, superscalar issue is possible, and `mem_idx` increases by 2 to reflect that.

2. Instruction Decoder and Dispatcher Trace.


Signals
Waves
Time
DISPATCH_AND_DECODE
Instruction 1
instr1_isadd
instr1_ismem
instr1_ismul
instr1_isdiv
instr1_rd[4:0]
instr1_rs1[4:0]
instr1_rs2[4:0]
instr1[31:0]
instr1_valid
Instruction 2
instr_destination_conflict
instr2[31:0]
instr2_valid
instr2_rd[4:0]
instr2_rs1[4:0]
instr2_rs2[4:0]
instr2_isadd
instr2_ismem
instr2_ismul
6 sec
12 sec
18 sec
24 sec
30 sec
XXX 1 2 3 4 5 6 11 16 17 16 0
XXX 0 2 10 16 1 17 16 0
XXX 11 15 22 13 30 15 5 11 12 2 3 17 0
xxxxxxxx 00B02083 00F02103 01602183 02D02203 03E02283 04F02303 025105B3 00B50833 00C80833 002088B3 003888B3 03184833 00000013
xxxxxxxx 00F02103 01602183 02D02203 03E02283 04F02303 02408533 00B50833 02618633 002088B3 003888B3 03184833 00000013
XXX 2 3 4 5 6 10 16 12 17 16 0
XXX 0 1 10 3 1 17 16 0
XXX 15 22 13 30 15 4 11 6 2 3 17 0

This is the trace for the instruction decoder and dispatcher. It shows the register addresses targeted by each instruction (rs1, rs2, rd). Cases with superscalar issue are highlighted in **blue and red**. Here, both instructions are valid.

In **red circles** we have a case that shows the handling of an *instruction destination hazard*. Here, the destination register of instruction 1 (rd = x11) is used by instruction 2 as a source (rs2 = x11). This raises a destination conflict, and stalls the superscalar issue.

3. CDB Controller and Arbitrator

This is probably the most important trace to demonstrate the working of the processor. Since all the outputs travel on the CDB (Common Data Bus), the results of our test program can be seen step-by-step as it traverses between different units.

First let's review our program:

```
extern long int* a;
extern long int* w;

long int weighted_average() {
    long int x = (w[0] * a[0] + w[1] * a[1] + w[2] * a[2]);
    long int y = x / (w[0] + w[1] + w[2]);
    return y;
}
```

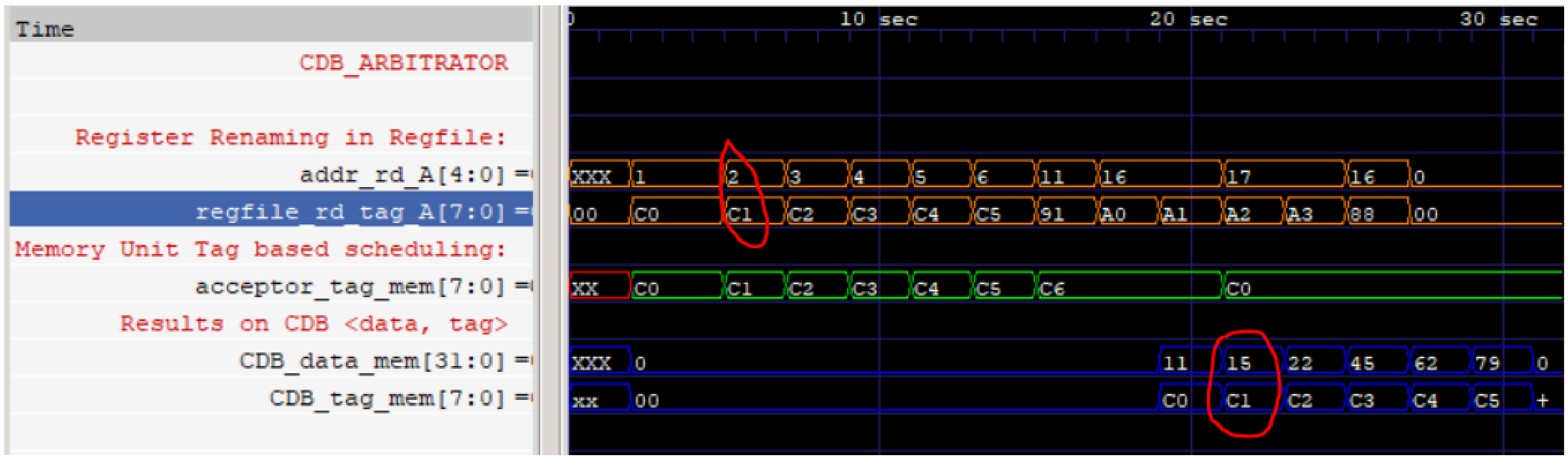
Here, in this weighted average, **w[0:3] = {11, 15, 22}** and **a[0:3] = {45, 62, 79}.**

Load Operations:

I have rigged the memory such that the read output data always equals the input address. So the following load instructions will correctly populate their destination registers with input addresses in operand 2.

```
lw      x1, 11(x0)      # w0
lw      x2, 15(x0)      # w1
lw      x3, 22(x0)      # w2
lw      x4, 45(x0)      # a1
lw      x5, 62(x0)      # a2
lw      x6, 79(x0)      # a3
```

This operation is shown in the following section of the trace.


Time
0
10 sec
20 sec
30 sec
CDB_ARBITRATOR
Register Renaming in Regfile:
addr_rd_A[4:0]
XXX 1 2 3 4 5 6 11 16 17 16 0
regfile_rd_tag_A[7:0]
00 C0 C1 C2 C3 C4 C5 91 A0 A1 A2 A3 88 00
Memory Unit Tag based scheduling:
acceptor_tag_mem[7:0]
XX C0 C1 C2 C3 C4 C5 C6 C0
Results on CDB <data, tag>
CDB_data_mem[31:0]
XXX 0 11 15 22 45 62 79 0
CDB_tag_mem[7:0]
xx 00 C0 C1 C2 C3 C4 C5 +

This section of the trace shows memory loads. The tag ID of memory station - which is a reservation station for memory ops - starts with **C**, i.e **C0, C1, C2** etc.

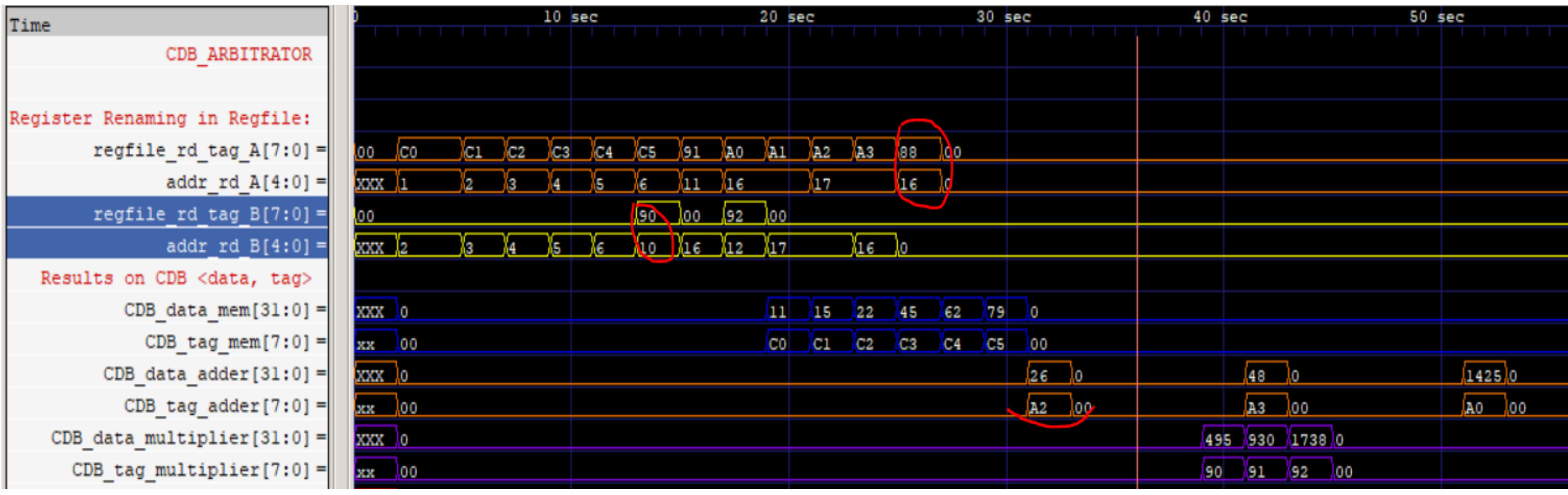
In orange, the register file's first six registers are invalidated/renamed and allotted memory tags C0 through C5. For instance, **rd2** is renamed, and waits for tag **C1** on CDB. This is resolved later on the CDB (shown in blue trace), when **C1** arrives with data **15**.

Arithmetic Operations:

The instructions we have to execute are as follows:

```
mul     x10, x1, x4   # w0 * a0
mul     x11, x2, x5   # w1 * a1
add     x16, x10, x11 # (w0 * a0) + (w1 * a1)
mul     x12, x3, x6   # w2 * a2
add     x16, x16, x12 # (w0 * a0) + (w1 * a1) + (w2 * a2)

# These two ADDs will be executed before above instructions
# Because of out-of-order processing.
add     x17, x1, x2   # w0 + w1
add     x17, x17, x3  # w0 + w1 + w2

div     x16, x16, x17 # (w0 * a0 + w1 * a1 + w2 * a2) / (w0 + w1 + w2)
```

Following is the trace on CDB:


Time
0
10 sec
20 sec
30 sec
40 sec
50 sec
CDB_ARBITRATOR
Register Renaming in Regfile:
regfile_rd_tag_A[7:0] = 00 C0 C1 C2 C3 C4 C5 91 A0 A1 A2 A3 88 00
addr_rd_A[4:0] = XXX 1 2 3 4 5 6 11 16 17 16 0
regfile_rd_tag_B[7:0] = 00 90 00 92 00
addr_rd_B[4:0] = XXX 2 3 4 5 6 10 16 12 17 16 0
Results on CDB <data, tag>
CDB_data_mem[31:0] = XXX 0 11 15 22 45 62 79 0
CDB_tag_mem[7:0] = xx 00 C0 C1 C2 C3 C4 C5 00
CDB_data_adder[31:0] = XXX 0 26 0 48 0 1425 0
CDB_tag_adder[7:0] = xx 00 A2 00 A3 00 A0 00
CDB_data_multiplier[31:0] = XXX 0 495 930 1738 0
CDB_tag_multiplier[7:0] = xx 00 90 91 92 00

In the orange trace, I show the register renaming in the **superscalar in-order** issue. The renaming/CDB tag mapping is as follows:

C for memory i.e. **C0, C1, C2** etc
A for Adder Reservation Station, i.e., **A0, A1, A2** and **A3** for the 4 add instructions.
9 for Multiplication Reservation Station e.g., register **x10** is given tag **90**
8 for Divider Reservation Station e.g., register **x16** is given tag **88.**

In the **blue** trace, we see the resolution of load instructions.

Out of Order Execution

Now here is the interesting part: In orange, we have the outputs of the adders with their tags. As can be seen, the first to respond is **A2**, which was executing the **3rd** add instruction, i.e. :

```
        “add      x17, x1, x2         # w0 + w1”
```

The result is **11 + 15 = 26**. This instruction executes before the first two instructions because they are waiting for the results of multiplications. Recall that

$$y = \frac{w_0 * a_0 + w_1 * a_1 + w_2 * a_2}{w_0 + w_1 + w_2}$$

$$y = \frac{11 * 45 + 15 * 62 + 22 * 79}{11 + 15 + 22}$$

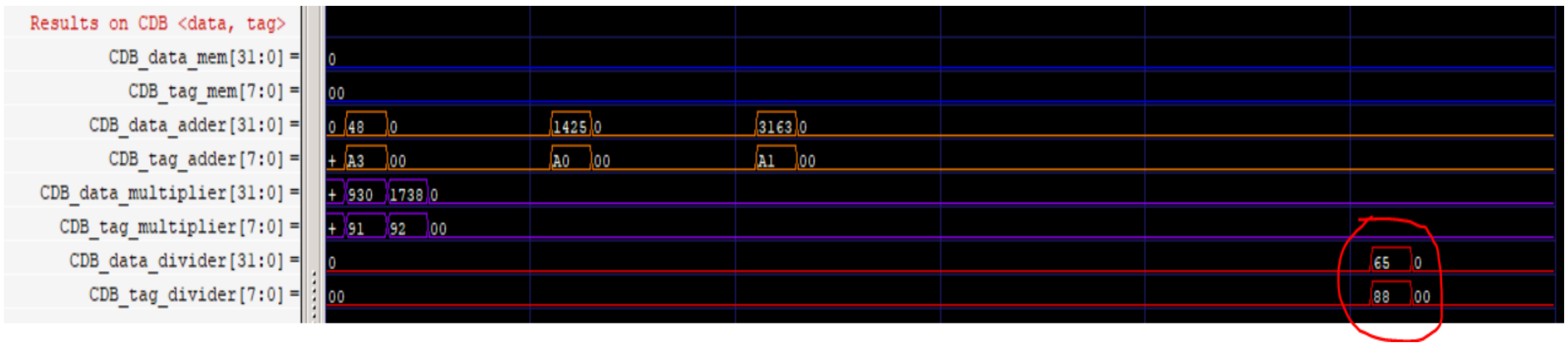
We have to calculate both the numerator and the denominator. The numerator also involves multiplications. The denominator however, can be readily calculated. That is why the **4th** instruction, which gives us the denominator "`w0 + w1 + w2`", is calculated as **11 + 15 + 22 = 48**, and appears with the **A3** tag way before the results of the additions from the numerator.

The results of the multiplications are calculated next, that are shown in the violet trace as <tag, data> pairs. That is **<90, 495>, <91, 930>, <92, 1738>**. They are output after the denominator addition due to higher latency of the multiplier. Arithmetic validation is given below:

$$y = \frac{11 * 45 + 15 * 62 + 22 * 79}{11 + 15 + 22}$$

$$y = \frac{495 + 930 + 1738}{48}$$

Finally, we have the results of the numerator, and then the division (circled in red).


Results on CDB <data, tag>
CDB_data_mem[31:0] = 0
CDB_tag_mem[7:0] = 00
CDB_data_adder[31:0] = 0 48 0 1425 0 3163 0
CDB_tag_adder[7:0] = + A3 00 A0 00 A1 00
CDB_data_multiplier[31:0] = + 930 1738 0
CDB_tag_multiplier[7:0] = + 91 92 00
CDB_data_divider[31:0] = 0 65 0
CDB_tag_divider[7:0] = 00 88 00

In orange, the numerator is added **<A0, 495 + 930 = 1425>** and
<A1, 1425 + 1738 = 3163>

In red, we have the result of the division, with quite a high latency (14 cycles). The result is **<tag, data> = <88, 65>**. So **65** is the final result of our weighted average.

Arithmetic verification is below:

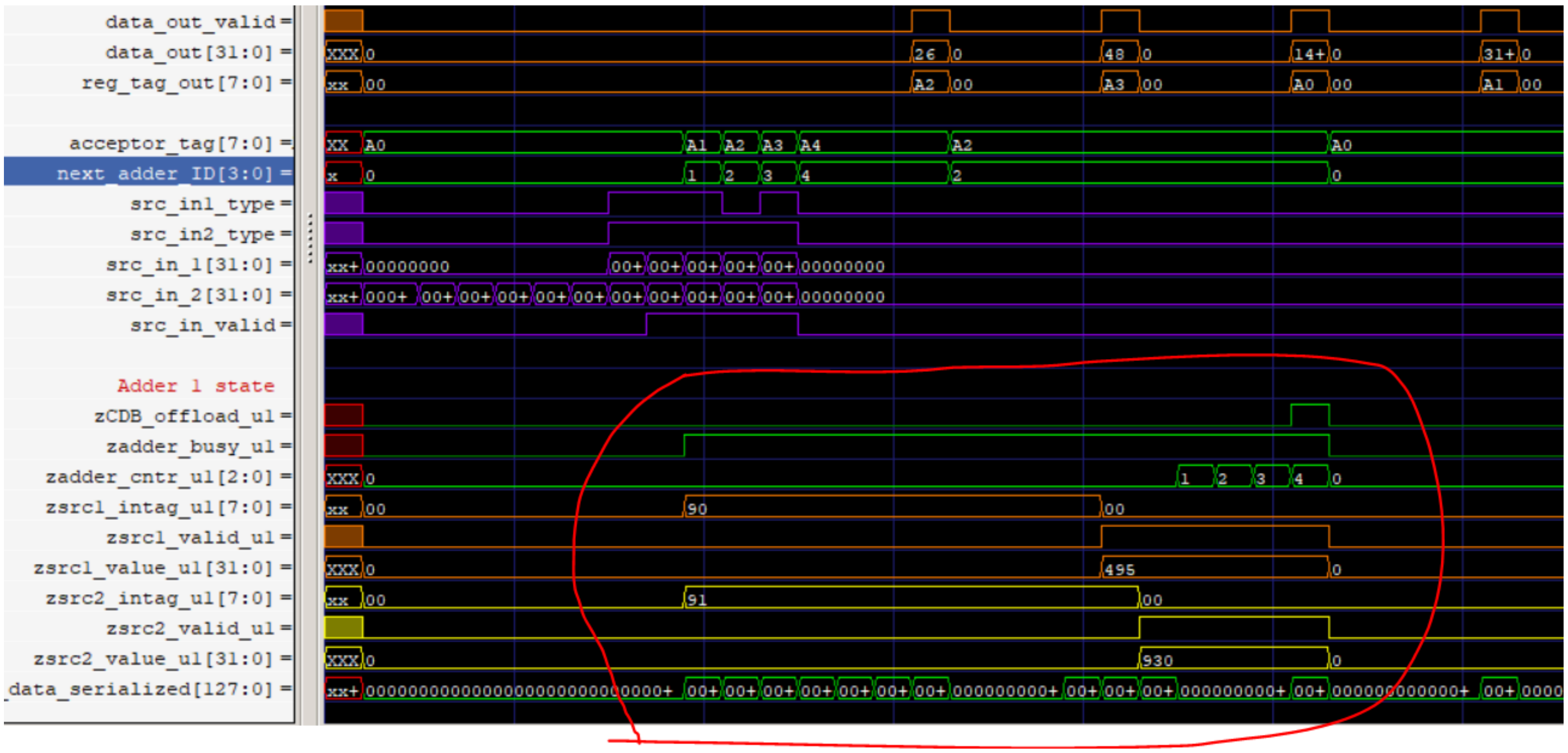
$$y = \frac{495 + 930 + 1738}{48}$$

$$y = \frac{3163}{48}$$

$$y = floor(65.895) = 65$$

4. Reservation Station Trace

In the interest of brevity, I will quickly go over the *addition reservation station* trace. Traces of other reservation stations are also very similar:



Circled in red is the operation of the first adder unit. As can be seen, the tags **90** and **91** are waiting at the start. These tags are the results of the first two multiplications. They get resolved to 495 and 930 respectively, and then the adder begins its operation. After 4 clock cycle latency, the output is given out, i.e. **1425.**