Department of Physics and Astronomy
University of Heidelberg

Bachelor Thesis in Physics
submitted by

AMANDA MATTHES

born in Hamburg (Germany) in 1998

2019

Exploiting Instruction-Level Parallelism

Hardware Structures
for a
Superscalar Out-of-Order Processor

This Bachelor Thesis has been carried out by Amanda Matthes at the
Institute of Computer Engineering (ZITI) in Heidelberg
under the supervision of
Prof. Ulrich Brüning

Abstract

Superscalar out-of-order execution has become the norm in modern processors. Yet there is little information in the literature about its implementation details. This thesis explores what new hardware structures superscalar out-of-order execution requires. It presents a design for a simple processor, implemented in SystemVerilog, that uses register renaming, reservation stations and a reorder buffer to dynamically schedule instructions.

The thesis also contains an essay that explains why power limitations have caused processor clock rates to stagnate in the noughties. This resulted in heightened pressure on computer architects to increase instruction throughput per cycle.

Abstract

Fast alle modernen Prozessoren realisieren dynamische, superskalare out-of-order execution. Trotzdem mangelt es in der Literatur an Information zu den Implementationsdetails dieser Maschinen. Diese Arbeit untersucht welche neuen Hardwarestrukturen notwendig sind um superskalare out-of-order execution zu unterstützen. Das Ergebnis dieser Arbeit ist ein einfacher, in SystemVerilog implementierter Prozessor, der diese Ideen umsetzt.

Diese Arbeit enthält außerdem ein Essay, das erforscht weshalb Prozessortaktfrequenzen um etwa 2005 aufgehört haben zu steigen. Dies legte weiter Druck auf Rechnerarchitekten den Instruktionendurchsatz pro Zyklus zu erhöhen und macht es besonders wichtig diese Optimierungsideen zu studieren.

Contents

1   Introduction

As we are reaching the limits that physics sets on the size and speed of our processor components, it is becoming increasingly important to develop sophisticated computer architectures to improve performance. A major contributor to rising instruction throughput in the last three decades has been the introduction of superscalar out-of-order execution to modern processors [1, p. 3] [2, p. 2f].

Superscalar processors increase throughput by dynamically arranging instructions to execute several of them in parallel in the same core. Out-of-order processors not only dynamically bundle instructions to be executed together, but they even allow for instructions to happen out of their original program order. This enables instructions to overtake instructions in front of them that would otherwise block the pipeline.

That is not a new idea. In fact, the original paper by Robert Tomasulo which first described an algorithm to enable superscalar out-of-order execution was published in 1967 [3]. It was used in 1966 in the large-scale System/360 Model 91, but it took over two decades for microprocessors to implement these ideas again. Since the 90s, superscalar out-of-order execution has become standard in our processors and enabled (single-core) processor performance to continue rising [2, p. 2f]. Unfortunately, the literature is often very vague about the implementation details of these machines.

The aim of my thesis project was to understand, design and simulate hardware structures that enable superscalar out-of-order execution. The result was a primitive but functioning processor, implemented in SystemVerilog and loosely based on the RISC-V ISA (see p. 45).

The following section will explain what superscalar out-of-order execution is and how it affects instruction flow and throughput. Section 4 will present my design and describe in detail how it allows for instructions to be dynamically scheduled. Section 5 will demonstrate the functionality of the design by looking at how it handles test cases. This is followed by a short discussion of the limits of superscalar execution and the dangers that out-of-order execution creates in sections 6 and 7. Section 8 outlines what work remains to be done to turn this design into a fully functioning processor and what alternative approaches exist.

Finally, this thesis also includes an essay (p. 39) which explains why it is necessary to improve the scheduling of instructions at all if a higher clock frequency could achieve the same effect (better performance) without the extra hardware complexity.

2   Background

This section gives a quick overview of the different stages of optimising a pipeline and ends with an explanation of what superscalar out-of-order execution is. This information can be found in any computer architecture textbook (e.g. [1], [2], [4]).

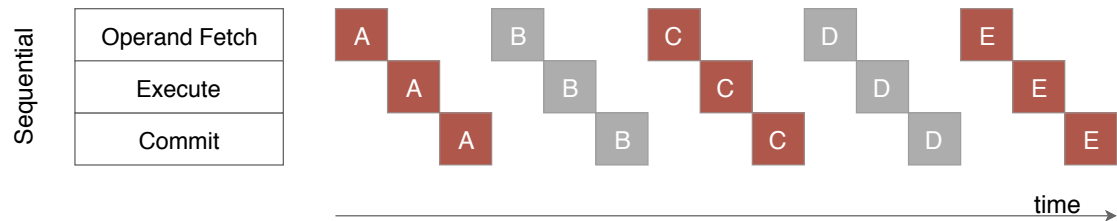## 2.1   Sequential scheduling and control dependencies

Figure 1: Sequential scheduling

In the classic von Neumann architecture, the program counter (PC) always points to the next instruction to be executed. The instruction is fetched from memory, its source operands are fetched, it is sent to the appropriate execution unit, executed and then the result is written back to memory. The PC advances to the next instruction and the same procedure starts again. Usually the PC always increases by one instruction, but jumps and branches can place the PC at any new instruction. Once an instruction is fetched from memory, the three major steps that it goes through are "operand fetch", "execute" and "commit". Figure 1 shows how instructions pass through these steps, one after the other.

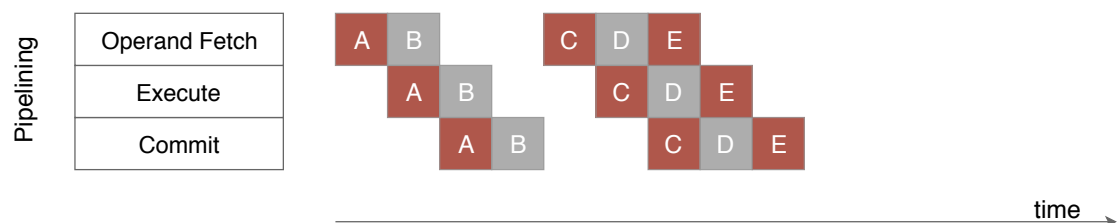## 2.2   Pipelining and data dependencies

Figure 2: Pipelining

Different parts of the CPU are responsible for the different stages that an instruction goes through. This means that at any time most of the CPU is doing nothing while it is waiting for the next instruction. The idea behind **pipelining** is to start the

next instruction one clock cycle after the first. Once A has both its source operands and starts execution, B can already start fetching its operands. When A has finished execution, B can start execution and so on.

Ideally, each of these steps can be done in one clock cycle. If a processor has a clock frequency of $f = 1\,\mathrm{GHz} = 10^9\,\mathrm{Hz}$, then one such clock cycle takes $1/f = 10^{-9}\,\mathrm{s} = 1\,\mathrm{ns}$.

The number of **instructions per cycle** (IPC) that are executed is a measure of throughput. The sequential scheduling scheme above would have an IPC count of $1/d$, where d is the **depth** of the pipeline, i.e. the number of stages that an instruction has to go through. In our case $d = 3$, so our IPC count is $1/3$. In other words, a sequential scheduling scheme completes one instruction every 3 clock cycles. With pipelining, we can achieve an IPC count of *up to one* if the pipeline is perfectly filled and there are no bubbles.

Unfortunately, this is not always possible. Consider this code (see p. 45 for how to read this):

```
1   LD    R1    0(R8)
2   LD    R2    0(R9)
3   ADD   R3    R1    R2
```

The ADD must wait for both LDs to happen before it can use its source operands. This situation is sketched in figure 2 between instructions B and C. The following instructions neatly fill the pipeline but the dependency between B and C creates a stall in the pipeline, a "bubble".

This is a **true dependency** or **flow dependency**. There is no way the ADD can happen before the LDs. Conversely, there are **false dependencies** or **name dependencies**, which do not stem from actual data flow. Consider this example, which could be the result of an unrolled loop:

```
1   LD    R3    18(R1)
2   ADD   R5    R3    R2
3   LD    R3    19(R1)
4   ADD   R6    R3    R2
```

There is a true dependency between the first ADD and the first LD, and one between the second ADD and the second LD. However, while instructions 3 and 4 are *logically* independent of instructions 1 and 2, they still have to happen strictly in sequence, because of the reuse of R3. This is a false dependency. It limits how much parallelism we can exploit from the code.

3

Name dependencies can be resolved by **register renaming**. In our example:

```
1'  LD   R3    18(R1)
2'  ADD  R5    R3    R2
3'  LD   R7    19(R1)
4'  ADD  R6    R7    R2
```

The true dependencies between 1 and 2 and between 3 and 4 remain. But now, there is no reason for the first two instructions to happen before the second two. We could swap them without violating the data flow:

```
3'  LD   R7    19(R1)
4'  ADD  R6    R7    R2
1'  LD   R3    18(R1)
2'  ADD  R5    R3    R2
```

We could even imagine them happening in parallel on independent cores:

```
1'  LD   R3    18(R1)          3'  LD   R7    19(R1)
2'  ADD  R5    R3    R2        4'  ADD  R6    R7    R2
```

This kind of parallelism between instructions that do not depend on each other is known as **instruction-level parallelism** (ILP). A major goal of CPU design is to find and exploit ILP where possible to speed up a processor without having to increase its clock speed.

The difference between true and false dependencies will become very important later when we will try to introduce such parallelism to our execution.
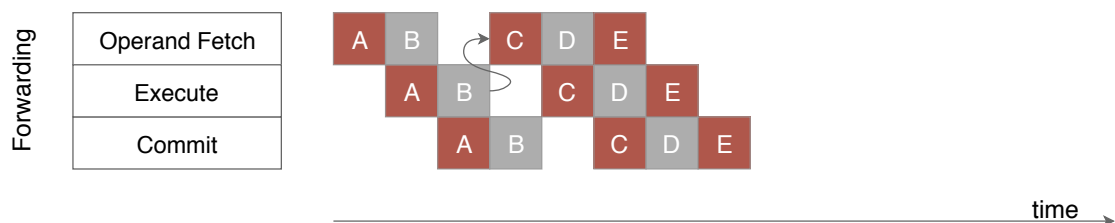
2.3   Forwarding

Figure 3: Pipelining with forwarding

In our example from figure 2 we can see that, with pipelining, instruction C has to wait for instruction B to completely finish, i.e. to commit its result by writing the

register file so that B can read it in the next cycle.

Forwarding paths in the pipeline allow instruction C to access the result of instruction B once it is calculated rather than having to wait for it to be written to the register file. We can see in figure 3 that this results in slightly tighter scheduling. In that particular example we are not achieving a big speed up, but if we have a piece of code in which many instructions depend on the previous one, forwarding paths can be a major improvement.
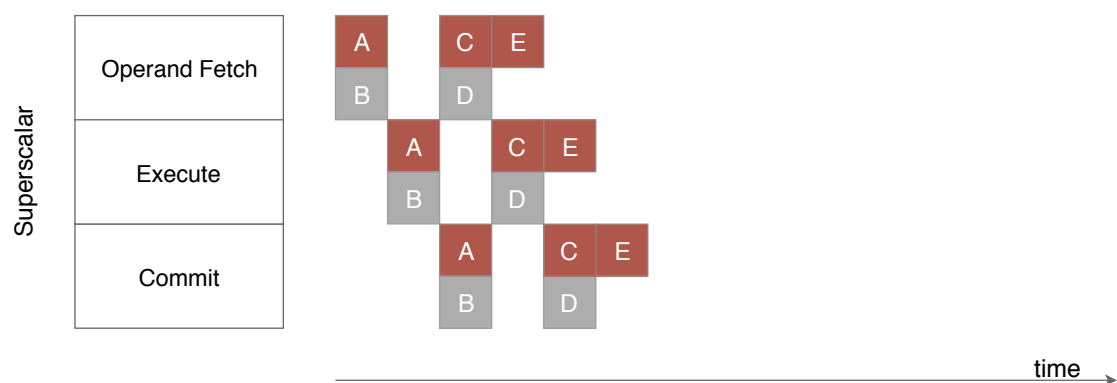
2.4   Superscalar execution

Figure 4: Superscalar execution

After optimising the IPC count by pipelining in this way, one might wonder if it is possible to optimise any further since all stages of the pipeline are being used whenever possible. However, at any point in time, we are only using one of the execution units. A processor will usually have many of these that specialise in different instruction types. One will be responsible for integer arithmetic, one for loads and stores to the memory, others will specialise in branches, floating point operations or square roots. So we might wonder if it is possible to fill these with other instructions that are ready to be executed. Consider these two instructions:

```
1   ADD   R3    R1     R2
2   LD    R5    0(R9)
```

They do not depend on each other at all. In hardware, a load instruction will be handled by the load-store unit and an integer addition by the integer unit. We can imagine these two instructions fetching their source operands, executing and committing simultaneously. This is called **superscalar execution**, as opposed to scalar execution where only one instruction is executed at a time.

With superscalar execution, we can reach IPC counts *greater than one*. If we have w execution units that can all work at the same time and all other parts of the processor can handle w instructions at once, then we can reach an IPC count of w. We call this the **width** of the pipeline. In our example (figure 4) we imagined a pipeline with $w = 2$. This maximum is only reached if the pipeline can always be filled. Figure 4 assumes that there are enough execution units to simultaneously execute A&B, C&D and D&E, respectively. In practice, this is not always the case.

2.5 Out-of-order execution
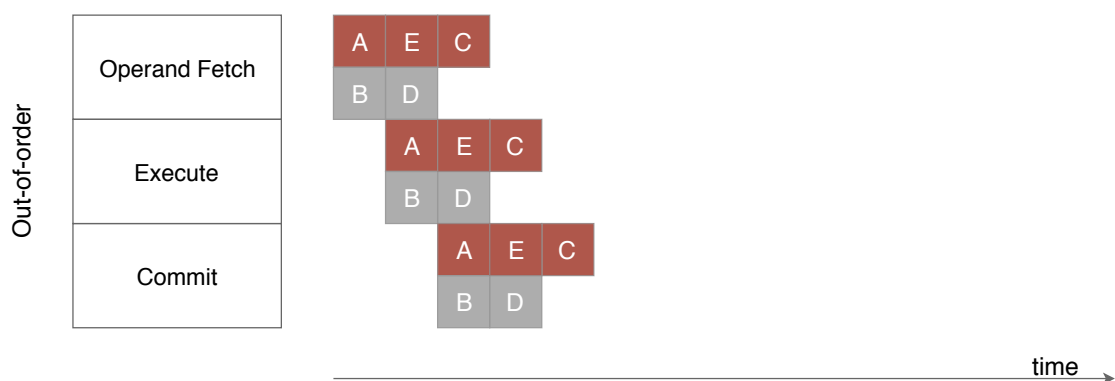
Figure 5: Out-of-order execution

We can see in figure 4 that, even with forwarding, the dependency between instructions C and B causes a bubble in the pipeline. In our simple model, where each instruction takes only one clock cycle, this may not seem like a major problem, but in reality, instruction can often take much longer. Load instructions that access memory are a good example of that. Cache misses can take over 100 cycles. If a following instruction needs that value, it has to wait for it. But there may be later instructions that do not depend on the load and could, in theory, be executed earlier. Out-of-order execution allows for this.

As long as we handle dependencies, we may execute an instruction earlier than the program order dictates, i.e. **out of order** to fill a gap in the pipeline, leading to an increase in IPC count. Figure 5 shows how, in our example, out-of-order execution can fill the gap in the pipeline between instructions B and C.

3 History and State of the Art

As mentioned in the introduction, the primary paper that first described how multiple execution units could be used simultaneously was written by Robert Tomasulo in 1967 [3]. The IBM System/360 Model 91 in 1966 used his algorithm for floating-point instructions but it took many years before superscalar execution was introduced to microprocessors.

The first one was probably the RS/6000 CPU, aka the POWER1, also developed by IBM in 1990 [5]. The time gap is at least partially due to the additional hardware that dynamic scheduling at runtime requires. In [5], where Oeler and Blasgen from the IBM Research Division present the POWER1, they write:

> The realization of the system architecture in a full superscalar organization does require substantial hardware – more than could fit on one chip in the available metal-oxide (CMOS) technology.

Technology was, however, just advanced enough at that point to fit the required hardware on *six* chips, which is what the POWER1 did. It could issue up to four instructions in one cycle and started the very successful POWER series.

It is useful to study the POWER series in the context of this thesis, because it too implements a RISC ISA and because it spans almost three decades, from the first superscalar processors to today. At the time of writing, the POWER9 is even the basis for the fastest computer in the world, Summit [6].

We can see in figure 6 that the number of instructions issued per clock cycle and the number of execution units has increased with almost every iteration. As transistors became smaller, hardware could get more complex. This allowed for wider pipelines. The POWER processors saw the last decade of exponential clock rate increase until it started stagnating after around 2005 . This increased the pressure on computer architects to design pipelines with high throughput, as they could no longer rely on an increase in clock frequency to make up for it.

The table in figure 6 also shows two other trends, increased use of multithreading and multicore designs. These exploit thread-level parallelism, which is distinct from instruction-level parallelism and outside of the scope of this thesis.

To a small degree, superscalar processors already implement a simple form of out-of-order execution since they violate the original, sequential program order. But for a processor to be declared truly out-of-order, it needs to potentially be able to swap the program order of two instructions if the data flow permits it. The POWER1

	POWER1	POWER2	POWER3	POWER4	POWER5	POWER6	POWER7	POWER8	POWER9
Release year	1990	1993	1998	2001	2004	2007	2010	2014	2017
Clock rate (MHz)	30	63	200	1300	2300	4700	5000	5000	4000
# Issues/cycle	4	4	4	5	5	7	6	10	9
# Ex. units/core	4	6	7	8	8	9	12	16	28
# Threads/core	1	1	1	1	2	2	4	8	4/8
# Cores	1	1	1	2	2	2	8	12	24/12
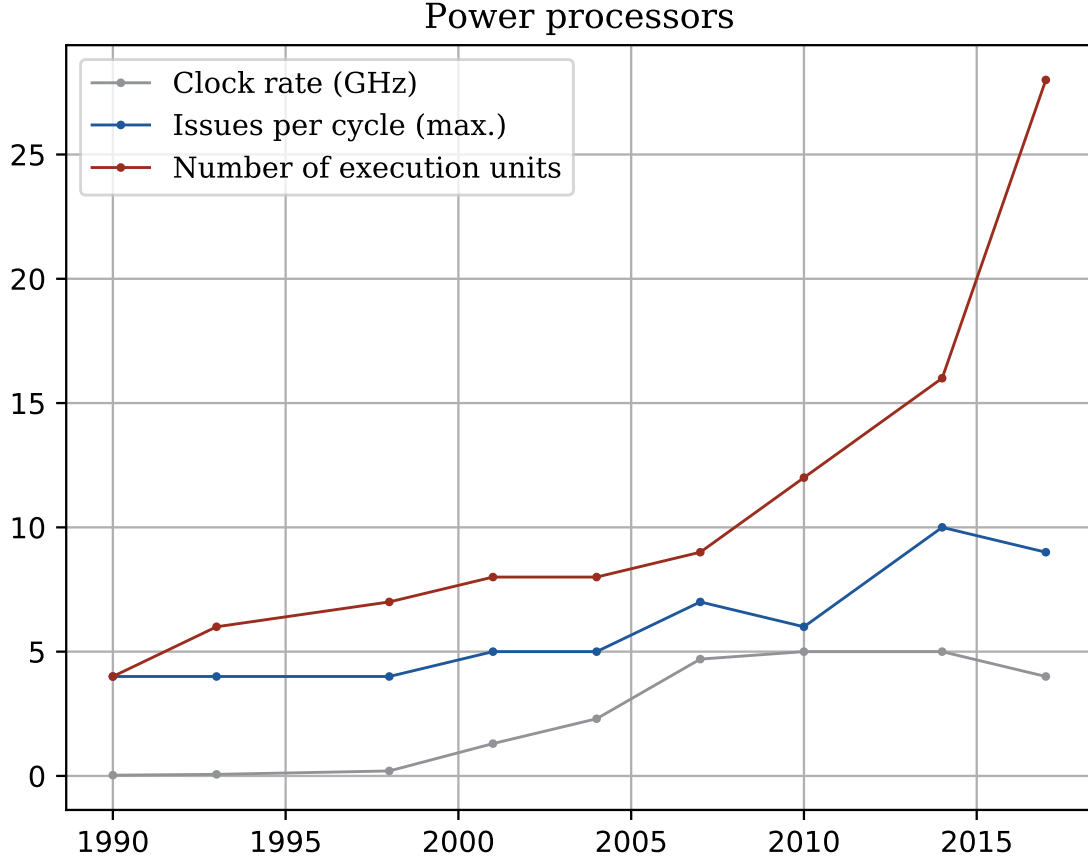
Figure 6: The POWER series. Data from [5], [7], [8], [9], [10], [11], [12], [13] and [14]

was also probably the first microprocessor to support this [5]. It dispatches instructions in groups, inside of which execution can happen out-of-order. Later iterations loosened this group restraint. The POWER9 can track up to 256 operations out-of-order in every core (24 core version) [14].

Eventually, all major processor developers introduced some form of superscalar out-of-order execution to their processors [15] [16]. It has become standard practice.

The next section will present a possible implementation of a simple superscalar out-of-order processor. It explains how new hardware structures make it possible for a processor to dynamically schedule instructions at runtime.

Figure 7: Pipeline design

4 Design

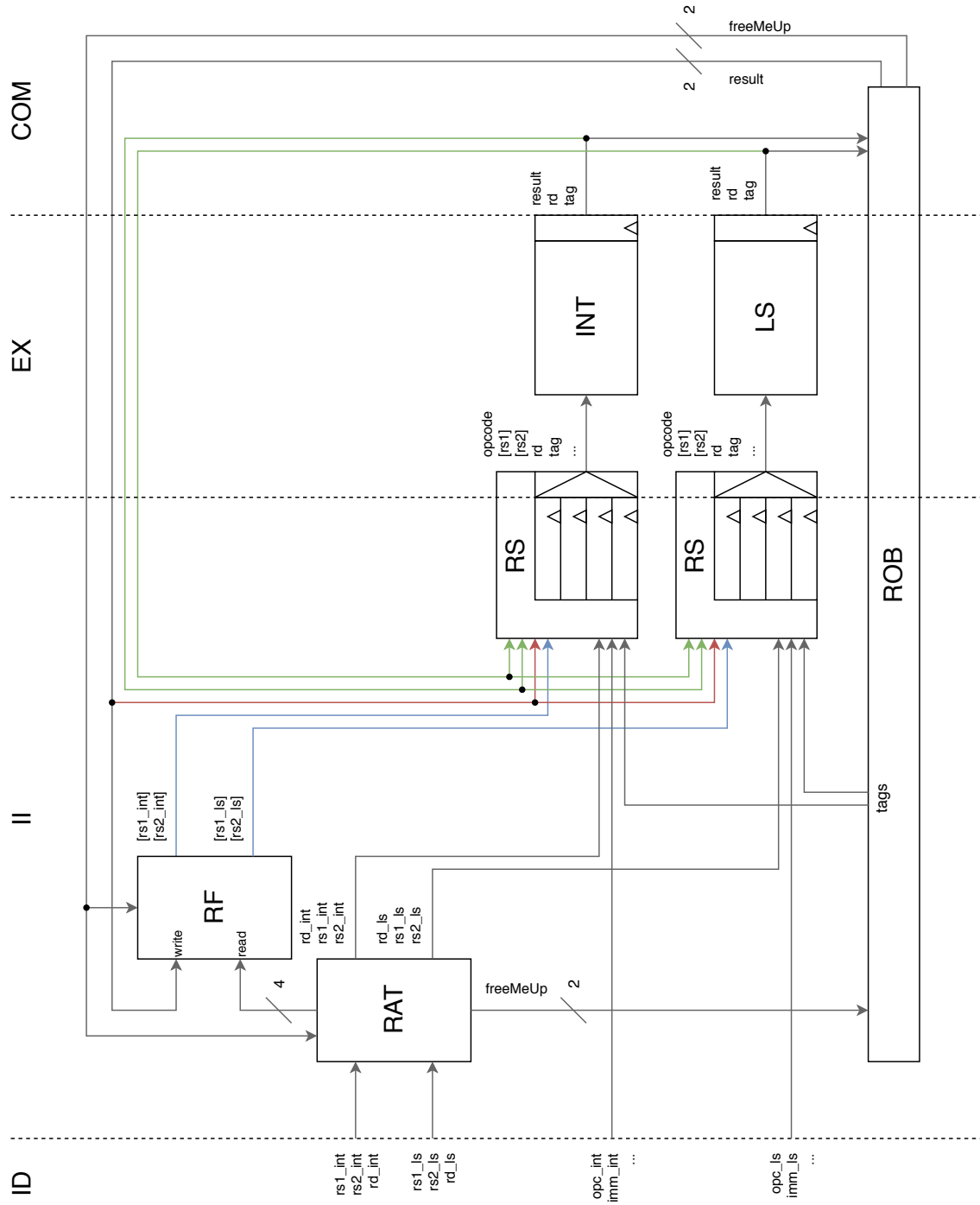Figure 7 shows a high-level diagram of the pipeline that I designed and implemented over the course of this thesis project.

To keep it simple, this pipeline only has one integer unit and one load-store unit (LS unit). The integer unit can only add two 64-bit values from registers and the LS unit can only load 64-bit values from memory. This is the functionality that is needed to run the LD and ADD instructions from the RISC-V ISA that I have introduced before. The integer unit takes one cycle for each ADD. The dummy LS unit also takes one cycle for a LD 90% of the time. But in 10% of cases, it takes ten cycles. This is supposed to simulate a cache miss. In those cases, the LS-unit blocks other LD instructions in the pipeline. LD instructions return random 64-bit values between 0 and 1000.

The general idea of the design is that instructions are sent to reservation stations (RS) where they wait until both of their source operands are available. Once they have them, they are executed, irrespective of their original program order. The results are buffered until they can be committed to the RF in-order. This is supposed to mimic an ideal dataflow engine, where there is no program order and instructions are executed as soon as they are ready to do so.

The first subsection will talk about this instruction flow step by step and then the second subsection will present the new modules in more detail.

4.1 Pipeline stages and instruction flow

Before I talk about pipeline stages, I should note that there are many different stage naming conventions. Some authors distinguish "dispatch" and "issue", others use them interchangeably. Similarly "finish", "retire" and "commit" often mean different things, but not always. I will use "issue" to refer to the stage in which instructions are moved from the ID stage to a reservation station. Instructions "finish" execution when their results are calculated. They are "committed" when their results are written to the register file.

Instruction Decode (ID)
The instruction decode stage is in charge of arranging instructions and their parts to make the next steps as easy as possible. It was not part of my thesis project to study this stage. I did, however, have to make certain assumptions about it to have

well-defined input for the following stages.

My design demands that the ID stage delivers up to two instructions every cycle, but not more than one of each type (integer, load-store). Either, both or neither can be valid in any cycle. The design also currently requires the integer instruction to come before the load instruction in program order if both are valid in one cycle.

Instruction Issue (II)

Every clock cycle the instruction issue stage receives up to two instructions from the decode stage. The register alias table (RAT) is in charge of register renaming to eliminate false dependencies (see 4.2.1 for details). Once renamed, the register file (RF) attempts to read the source operands for both instructions. If they are available, their values are stored in the respective reservation station together with the rest of the instruction (destination register, opcode, immediate etc.). Reservation stations are simply bundles of registers where instructions wait for their source operands. Every execution unit has its own reservation station. The reorder buffer (ROB, see 4.2.2) gives each instruction a tag, which is used to keep track of the program order of instructions.

Instructions that did not get both of their source operands from the RF wait in their reservation station until they have received both source operands from either the forwarding paths of the execution units or the commit path (see 4.2.3 for details).

Execute (EX)

Whenever an execution unit is ready, it can start executing any one of the instructions in the reservation station that have all their source operands.

This now no longer has to happen in program order. It is possible for instructions to overtake other instructions in the reservation station if their source operands are available early enough. This prevents instructions from blocking independent instructions that are behind them in program order.

Finish and Forward

Once an instruction finishes execution, its result is written back to the reorder buffer using the tag it received in II (see 4.2.2). The result is also forwarded to both reservation stations, in case it is the source operand of a waiting instruction.

This stage cannot be easily drawn in figure 7 like the others. This is because its output register is a spot inside the ROB, which I decided to draw horizontally.

11

Commit (COM)

In this step, results that have been written back to the ROB are committed in their original program order. This is possible because instructions were tagged before they were placed into reservation stations.

Backwards flow control

There are three reasons that can cause ID to be stopped:

1. The ROB runs out of tags

2. The reservation stations are full

3. There are no more registers left for renaming (see 4.2.1)

If any one of these happens, the ID stage needs to be stalled until enough instructions have been committed to free up space for new ones.

4.2 New structures for superscalar out-of-order execution

Even before they became superscalar, processors had register files and execution units. The RAT, the ROB and the reservation stations are the new, critical additions that make superscalar out-of-order execution possible. This subsection takes a closer look at how they work.

4.2.1 The register alias table (RAT)

I introduced the general idea of register renaming in subsection 2.2 but have not yet explained how it can be implemented. There are 32 integer registers in the base RISC-V ISA, x0-x31, where x0 is hardwired to 0 [17, p. 9]. To implement register renaming effectively, the register file should have more physical registers than the ISA specifies. This version uses 64 physical registers so that they can be addressed with only one bit more. To clearly separate them, I will always use R0-R31 to refer to the architectural registers, and P0-P63 to refer to the physical registers. This is not a common naming convention but makes talking about register renaming much easier.

R1	P50
R2	P2
R3	P26
R4	P4
R5	P36
→ R6	P27

Figure 8: Possible RAT

The essential unit that handles register renaming is the register alias table (RAT), which keeps track of the mapping between physical architectural registers. It also

12

keeps a 64-bit list, that indicates which physical registers are currently being used and which ones are free. I will use the term RAT to refer both to the module that does the renaming and the actual mapping table it contains. The first few entries of that table may look like in figure 8. This would mean that R1 was mapped to P50, R2 to P2, R3 to P26 and so on. R0 will always be mapped to P0, which is always 0. The architectural register number is used to index the table. It is therefore 32 bit deep and $\log_2 64\,\text{bit} = 6\,\text{bit}$ wide.

Renaming registers

Let us re-examine the example from earlier (see subsection 2.2):

```
1   LD    R3    18(R1)
2   ADD   R5    R3    R2
3   LD    R3    19(R1)
4   ADD   R6    R3    R2
```

We noted that there are true dependencies among the first two instructions and among the last two instructions. We also noted that even though instructions 3 and 4 are logically independent of instructions 1 and 2, they still have to wait for them because of the reuse of R3.

We saw how this false dependency can be manually removed by renaming registers, but how can a CPU identify and resolve name dependencies *at runtime*?

Before the code above is decoded, the RAT will at least contain information about R1 and R2 because they will be used in this code. They may simply be mapped to P1 and P2. In that case, the RAT would look like the table marked with 0 in figure 9. Now, after instruction 1 is decoded, the RAT will replace the source operand R1 with P1. It will also give R3 a new name. In this case, P3 is the next vacant register, so we can use that. The renamed instruction will now be:

```
1'  LD    P3    18(P1)
```

So far this is not very interesting. We simply replaced R1, R2 and R3 with P1, P2 and P3.

For the ADD instruction, the RAT remaps the source registers as before. It also needs a new name for R5. The next free register is P4. Instruction 2 is renamed to:

```
2'  ADD   P4    P3    P2
```

The RAT now looks like the table marked with 2 in figure 9.

Just like with the first LD, instruction 3 has its source register R1 renamed to P1.

```
              0                      1                      2
      R1  P1                 R1  P1                 R1  P1
    → R2  P2                 R2  P2                 R2  P2
      R3                   → R3  P3                 R3  P3
      R4                     R4                     R4
      R5                     R5                   → R5  P4
      R6                     R6                     R6

              3                      4
      R1  P1                 R1  P1
      R2  P2                 R2  P2
    → R3  P5                 R3  P5
      R4                     R4
      R5  P4                 R5  P4
      R6                   → R6  P6
```
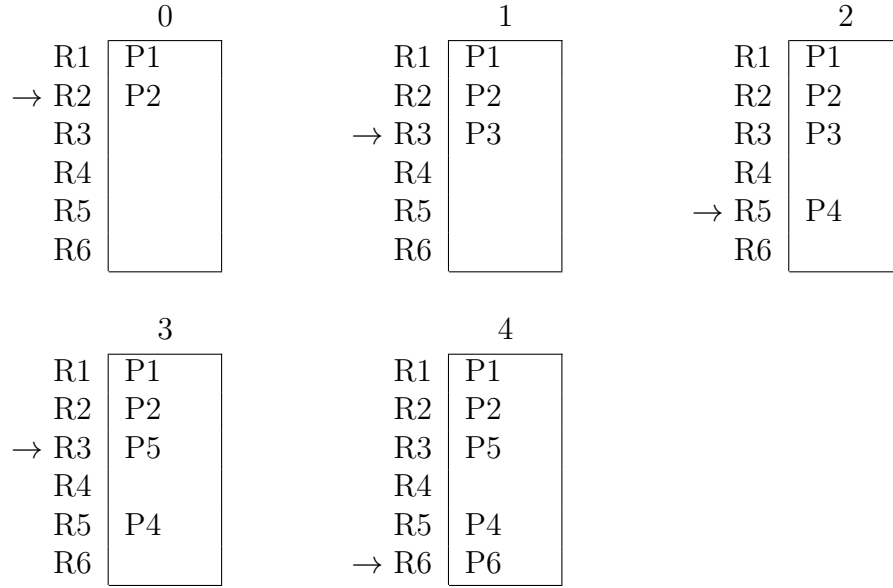
Figure 9: RAT after each instruction

However, it also gets a new, unused destination register. In our case, P5 is the next free one. The RAT updates the mapping for R3 (see figure 9) and the renamed instruction is:

```
    3'  LD    P5     19(P1)
```

Instruction 4 now uses the *new* name for R3, which is P5 instead of P3. Its destination is renamed to P6, the next free physical register:

```
    4'  ADD   P6     P5      P2
```

We can now observe that the renamed instructions 1'-4' have no false dependencies, just like the manually renamed example in 2.2.
To summarize, the procedure for renaming registers of an instruction is:

1. Replace architectural source registers with the mapped physical registers using the RAT

2. Replace the architectural destination register with a new, vacant physical register. Update the RAT

Freeing registers again

If we always pick a new, free destination physical register for each instruction, we will eventually run out of registers. We need a mechanism that frees registers again at some point. But when can a register be safely used again?

Let us revisit the code from before:

```
1   LD    R3    18(R1)
2   ADD   R5    R3    R2
3   LD    R3    19(R1)
4   ADD   R6    R3    R2
```

which after renaming became:

```
1'  LD    P3    18(P1)
2'  ADD   P4    P3    P2
3'  LD    P5    19(P1)
4'  ADD   P6    P5    P2
```

In the original code, instruction 3 overwrites R3. Later instructions will not have access to, and therefore will not depend on the old value of R3. Through renaming, the "old" value is saved in P3 and the "new" value in P5. This means that once all instructions up to instruction 3 are executed, the value in P3 will definitely never be needed again. Since we commit instructions in program order, we can safely free P3 once instruction 3 is committed. To do this, I use a so-called **free-me-up tag**. In figure 9 we can see that before instruction 3 is renamed, the RAT can check which physical register was mapped to R3 before, in this case, P3. The RAT then sets the free-me-up tag of instruction 3 to P3, before it remaps R3 to P5. Once instruction 3 is about to commit, the free-me-up tag is sent back to the RAT, so that it can update the free list.

In figure 7 we can see that the free-me-up tag is also sent back to the RF. This is because the RF keeps information on whether the value in a register is valid yet. Once we free up a register to reuse it, whatever was in this register before becomes invalid. So the RF needs to know when we repurpose a physical register.

We can think of setting the free-me-up tag as step 3 of register renaming:

3. If the architectural destination register was mapped to another physical register before, set the old one as the free-me-up tag

4.2.2 The reorder buffer (ROB)

The reorder buffer (ROB) tags instructions in program order so that they can be **committed in order** even when their actual execution order is completely different. After instructions are executed, their results are written back to the ROB, where they wait to be committed. So the ROB has three major functions:

1. Tag instructions in-order when they are placed in reservation stations

2. Save instruction results (possibly out-of-order) until they can be committed

3. Commit results in-order

These three functions need to happen simultaneously in every clock cycle. Because of how the ROB is connected to every pipeline stage, I decided to draw it as running in parallel to all the other modules in figure 7. In other depictions, it is often drawn as a buffer between the execution units and the register file.
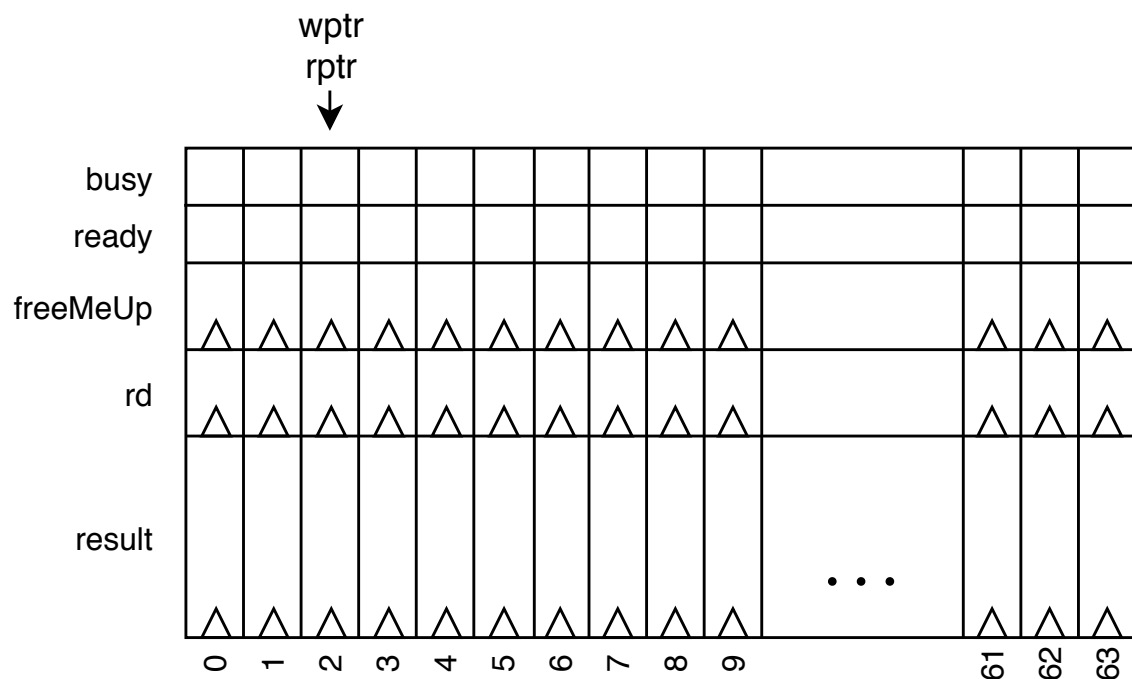

Figure 10: Empty ROB

The ROB can be thought of as an array of registers for instruction results and their destination registers (rd). It also saves the free-me-up tag of each instruction (see 4.2.1). We can draw this structure as in figure 10. The write pointer (wptr) always points to the next free spot in the ROB and doubles as the tag that is given to incoming instructions when they are issued (in-order) to a reservation station. That

location in the ROB is then marked as busy. The read pointer (rptr) always points to the oldest instruction. This is the head of the commit queue. Figure 11 shows a ROB with six busy slots before any of those instructions finished execution. The free-me-up tag and the destination register are known at this point.
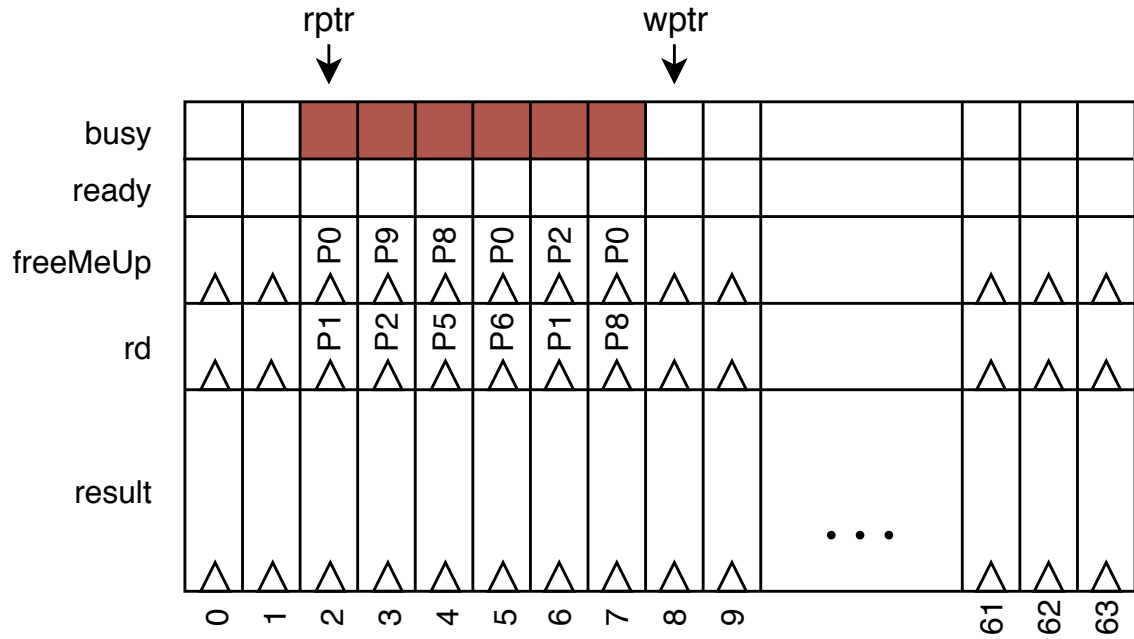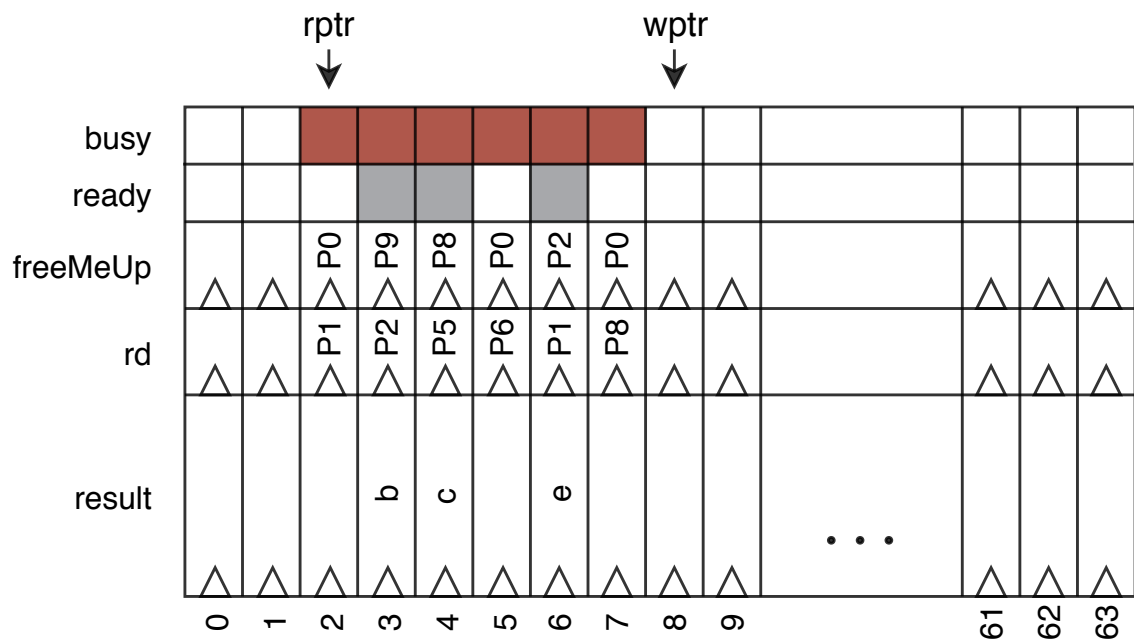

Figure 11: Partially busy ROB

Figure 12: ROB with some entries that are ready to commit

When an instruction has finished execution, its result is written to the ROB by matching the tag that the instruction got in II. Reservation stations allow for instructions to be executed out-of-order, so the ROB needs to be able to write any result register in each cycle. Figure 12 illustrates one possible situation, where three of the six instructions have finished execution and written their results to the ROB.
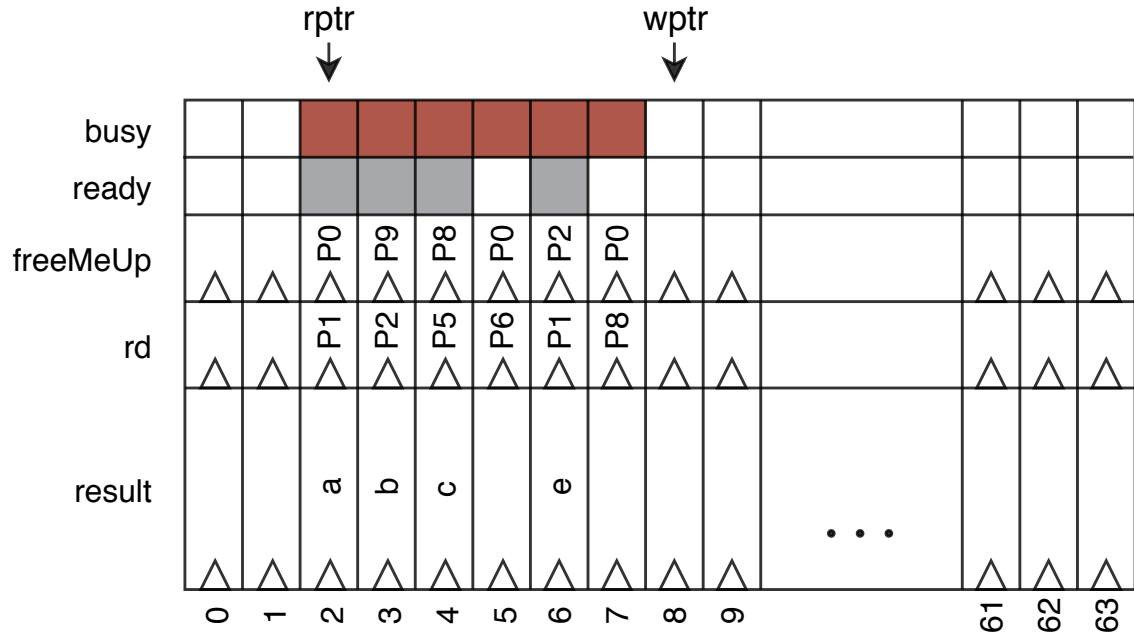

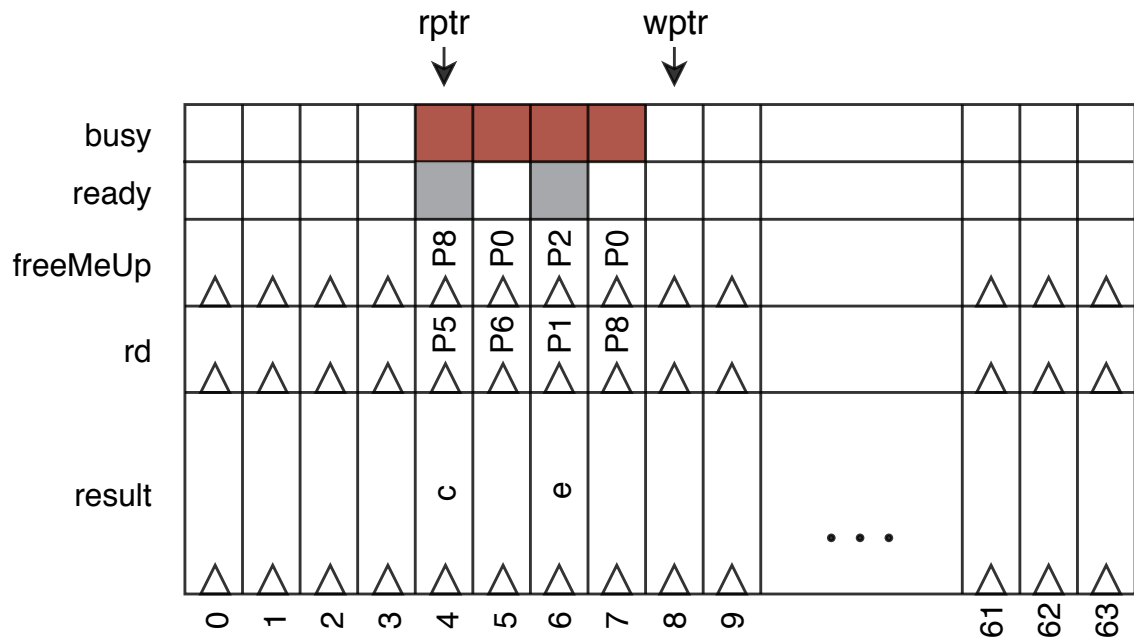Figure 13: ROB that is about to commit

Figure 14: ROB after commit

In this case, no results can be committed yet because the instruction at rptr is not ready to commit yet. Once the two instructions at rptr and rptr+1 are ready to commit, like in figure 13, their results are written to the register file. The rptr advances two spots and the two locations are marked as not busy and not ready. In our example, the resulting ROB may look like in figure 14.

In practice, the ROB will tag new instructions, collect results and commit ready instructions every clock cycle, rather than in sequence like in this example.

Other tags and values in the ROBs

One can easily see that the busy tag is not actually necessary. It is directly determined by the location of wptr and rptr. However, it can make the ROB easier to understand and debug. Similarly, my SystemVerilog implementation also saves the opcode of each instruction. This makes waveforms, in particular, much easier to read (see section 5) and these redundant values can easily be removed in later stages of development.

4.2.3   The reservation stations

In this design, both reservation stations can keep up to four instructions. Each slot contains space for the following information:

- The register address of the source operands rs1 and rs2

- The value of the source operands rs1 and rs2

- A valid tag for each source operand that indicates if it has been received yet

- Other instruction parts that come directly from ID (opcode, immediate, etc.)

- The tag from the ROB

- A free tag that indicates if the slot is being used

An instruction is ready for execution once it has both source operands, that is when both valid tags are set. There are three ways an instruction can receive its source operands:

1. From the RF:
   If a source operand has already been written back to the RF by the time an instruction is issued to a reservation station, the value is copied into the reservation station in the same cycle as the instruction. This is the blue path in figure 7.

2. From the forwarding paths:

    The reservation stations listen in on the results paths from execution units to the ROB. Every clock cycle, the destination register of a new result is checked against all the source registers of the waiting instructions. These are the green paths in figure 7.

3. From the commit path:

    The reservation stations also check against destination registers of instructions that are being written back to the RF. This has to happen both for waiting instructions and for instructions just being issued. In figure 7, this is the red path.

The four slots are numbered and new instructions are placed into the first free one. If several instructions are ready for execution, the first ready one is chosen for execution.

This page has been intentionally left blank

5   Results

The design presented in section 4 was implemented in around 1300 lines of IEEE 1800-2009 SystemVerilog code. This includes several test cases implemented as tasks that can be called in the top module. The code was developed and tested using the Incisive Enterprise Simulator from Cadence Design Systems.

During the course of this thesis, I also wrote a python script that takes SystemVerilog code for a module and creates the skeleton code for a corresponding test bench. It instantiates the module, a clock and all the regs and wires that are necessary to connect the test bench to the module. It can be found at `github.com/amanda-matthes/` `Testbench-Generator-for-SystemVerilog-Modules` and may be useful for further development.

To expedite testing, I also created several .tcl.svcf command scripts for SimVision. These can be sourced to automatically arrange and colour waveforms in useful ways. They also include definitions of mnemonics. Mnemonics translate the original binary waveforms into text, where appropriate. This means that instead of the 7-bit opcode 0110011, the waveform will read "ADD" and instead of the architectural register address 10110, the waveform will read "R22", etc. Used mnemonics include:

- R0-R31 instead of 00000-11111 for architectural registers

- P0-P63 instead of 000000-111111 for physical registers

- ADD and LD instead of the opcodes 0110011 and 0000011 [17, p. 104f]

- BUSY, FREE, VALID, RESET, FULL and STOP instead of 0 and 1, where appropriate

In this section, I will demonstrate the core functions of my design by looking at how the simulation handles certain inputs. To do this, this section includes waveforms from SimVision of selected signals for each example. They make use of the above-mentioned mnemonics.

5.1 Superscalar execution

I first want to show that my design can, in fact, achieve superscalar execution, with IPC = 2. To do this, I fed the pipeline one ADD and one LD instruction at the same time.

Figure 15 shows selected waveforms that demonstrate how both instructions pass the pipeline stages at the same time.

The first signal is the clock, which I set to a period of 20 ns. The next four signals are the inputs. For this demonstration, the exact instructions are uninteresting. It is, however, necessary that they are of different types so that they can be executed in parallel by different execution units. I coloured the LD signals blue to distinguish them from the ADD signals in red. To ensure that the input was valid at the positive clock edge, I always manipulated it at the negative clock edge.

The next four signals show the first entry of the reservation station of the integer and the load-store unit, respectively (INT RS and LS RS). At the next positive clock edge (70 ns), we can see that these are loaded with the instructions. I ensured that all source operands were available in the RF so that both instructions could load them in the same cycle and start execution in the next.

At the next positive clock edge (90 ns), the reservation stations are freed again and the functional units start execution. Because this is a simulation at the register-transfer level (RTL), we do not see an execution delay and the output is valid immediately. In this case, there was a cache hit which means that the result of the load was available in the next clock cycle.

We can see that the ROB saved the instructions at the same time as the reservation stations (70 ns). Once they finish execution (110 ns), their results are written to the ROB. This also flags those ROB entries as "ready to commit". In the next clock cycle (starting at 130 ns) those results are written back to the RF. The ROB entries are freed again and one cycle later, the results are available in the RF.

This shows that, in theory, this design can execute two instructions every clock cycle if we can schedule one ADD and one LD in every cycle and there are no cache misses.
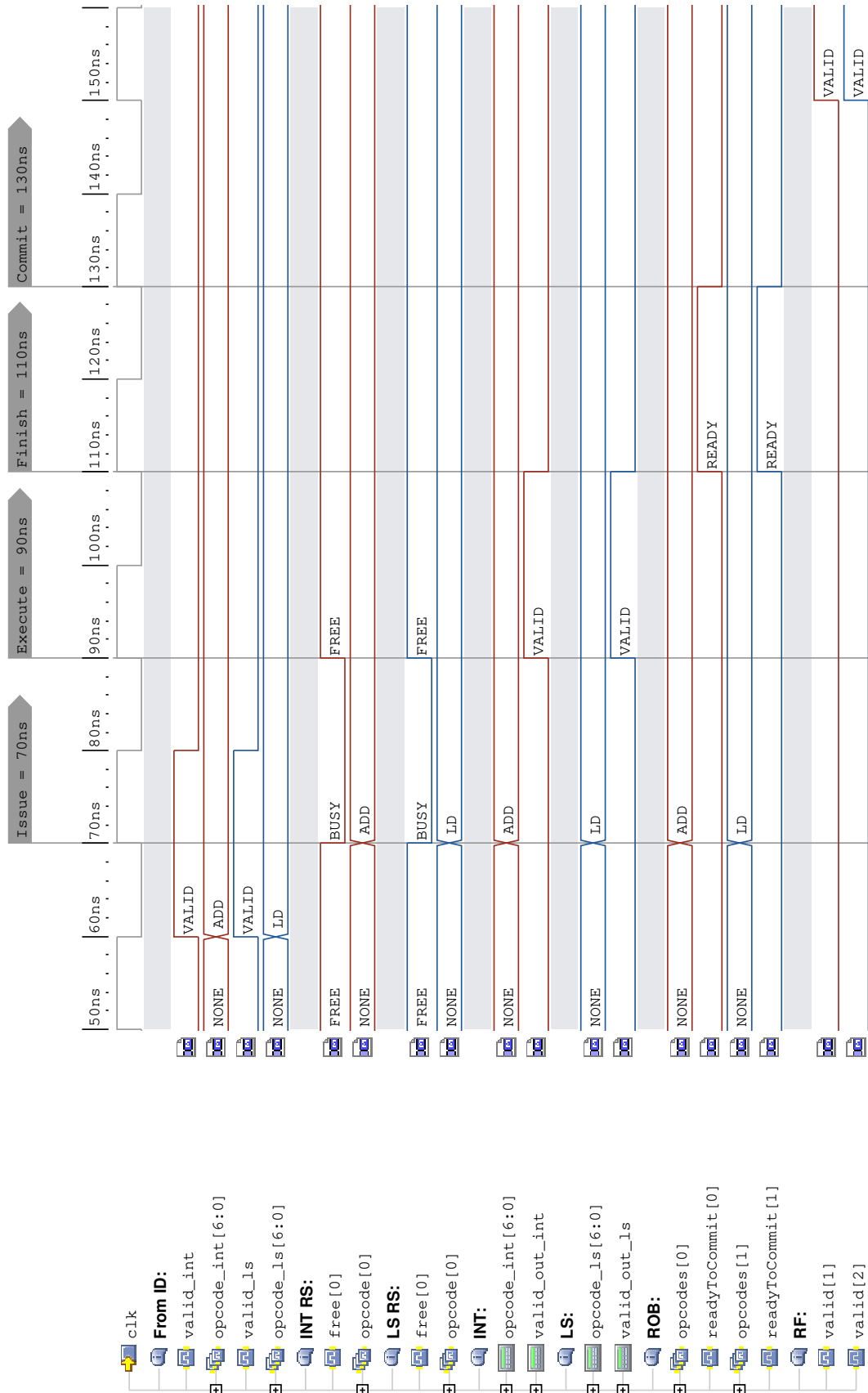
Figure 15: Superscalar execution

5.2 Register renaming

I now want to demonstrate register renaming. In section 4.2.1, I showed how this code:

```
1   LD    R3    18(R1)
2   ADD   R5    R3    R2
3   LD    R3    19(R1)
4   ADD   R6    R3    R2
```

can be renamed to this code:

```
1'  LD    P3    18(P1)
2'  ADD   P4    P3    P2
3'  LD    P5    19(P1)
4'  ADD   P6    P5    P2
```

to resolve false dependencies using two simple rules. I want to demonstrate that this is in fact what the design does.

To execute the code above, we first need to load the registers R1 and R2. I did this with:

```
LD    R1    0(R0)
LD    R2    0(R0)
```

The source register and the immediate are not important since loads return random values anyway, so I set them to zero.

As before, I always changed signals at the negative clock edge to ensure that they were stable at the next positive one. I always applied only one instruction every clock cycle, so that we could see the RAT after each instruction renaming. The result is shown in figure 16.

The red waveforms show the integer input which consists of the important instruction parts and the corresponding valid signal. Our code has two integer instructions. Instruction 2 is issued at 130 ns and instruction 4 at 170 ns. The first one is quite easy to read in the waveforms, but the second one seems to be missing some parts. This is simply because the opcode and both source registers have stayed the same and the waveform viewer only lets us see change.

Similarly, the blue waveforms show the load-store input. First, R1 and R2 are loaded, then R3 and, after the first ADD, R3 again, but this time with a different immediate.

The last six lines show the first few entries of the RAT. We can compare this with figure 9 and find that the simulation behaves exactly as predicted.
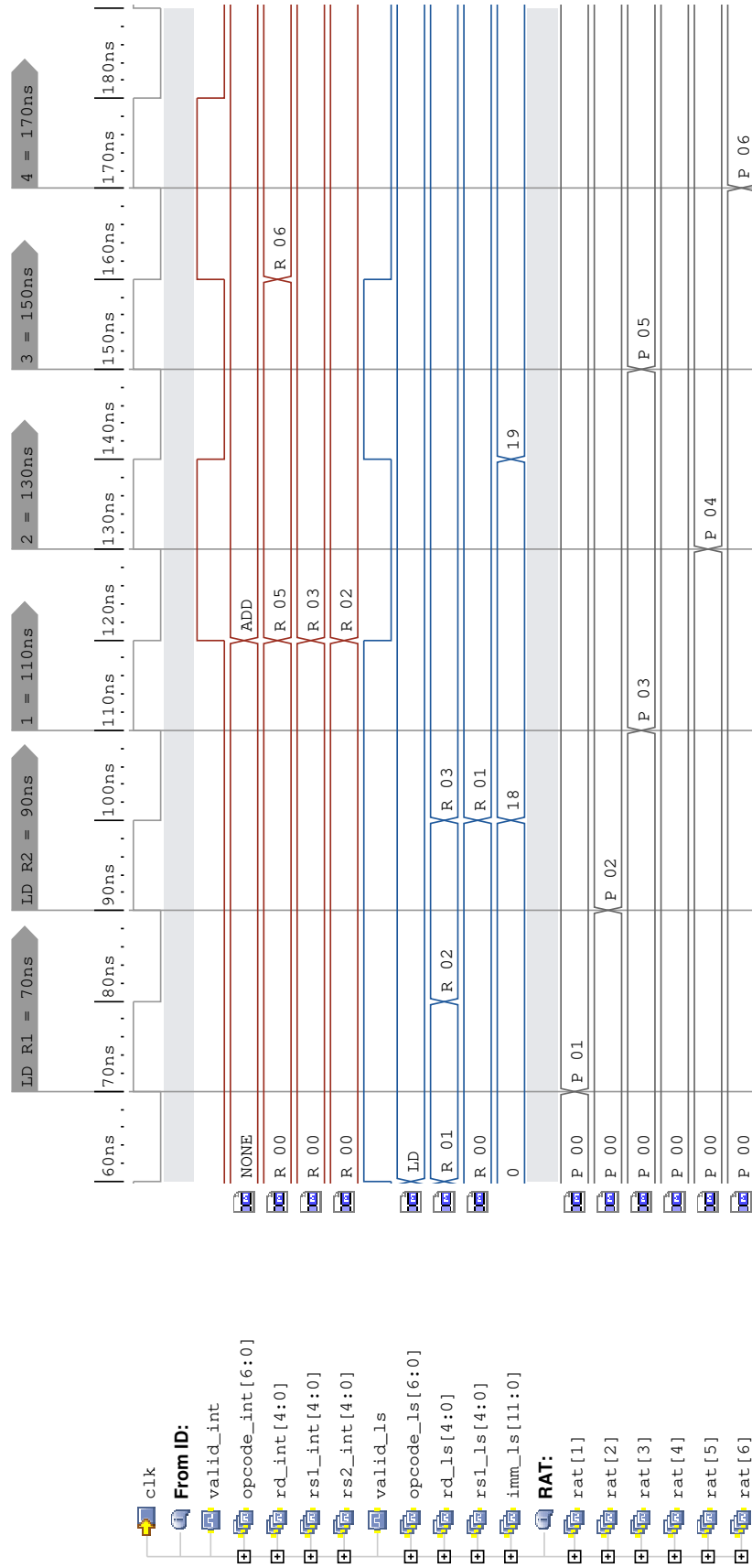
Figure 16: Register renaming

5.3 Waiting for source operands and forwarding

I now want to show how an instruction waits in its reservation station until it receives both its source operands. To do this, I executed the following instructions:

```
1   LD    R1     0(R0)
2   LD    R2     0(R0)
3   ADD   R3     R1     R2
```

This code loads R1 and R2 with random values and then adds them together. Figure 17 shows the result.

The first few signals simply show the input from the ID stage. As before, I use red for signals associated with integer operations and blue for load-store signals. The next section of waveforms shows what the LS unit outputs. We can see that the first LD returns the value 73 in the clock cycle starting at 90 ns, before the ADD instruction is even issued. This result is written back to the ROB at 110 ns and then at 130 ns is committed to the RF. These are the next three signals.

The last few signals show what happens inside the integer reservation station. While the first load is writing its result to the ROB, the ADD instruction is sent to the reservation station. In this case, R1 and R2 were simply renamed to P1 and P2 (see 4.2.1). The register file had not been written yet and so neither rs1 nor rs2 is valid at this point. But when the result of the first LD is committed, it is also forwarded back to the reservation station (red path in figure 7). The 73 is copied into the reservation station (rs1_content) and the valid tag is set (rs1_valid).

The second LD is issued one cycle after the first which means that it starts execution when the first LD finishes it, at 110 ns. But this time there was a cache miss. Recall that I declared those to take ten cycles, that is $10 \times 20 \, \text{ns} = 200 \, \text{ns}$. At 310 ns the LS unit returns the value 428, which this time can be directly forwarded to the reservation station (green path in figure 7). This means that the ADD instruction can start execution at 330 ns, before the value of P2 has even been committed.

The next subsection demonstrates how placing instructions into reservation stations like this enables out-of-order execution.
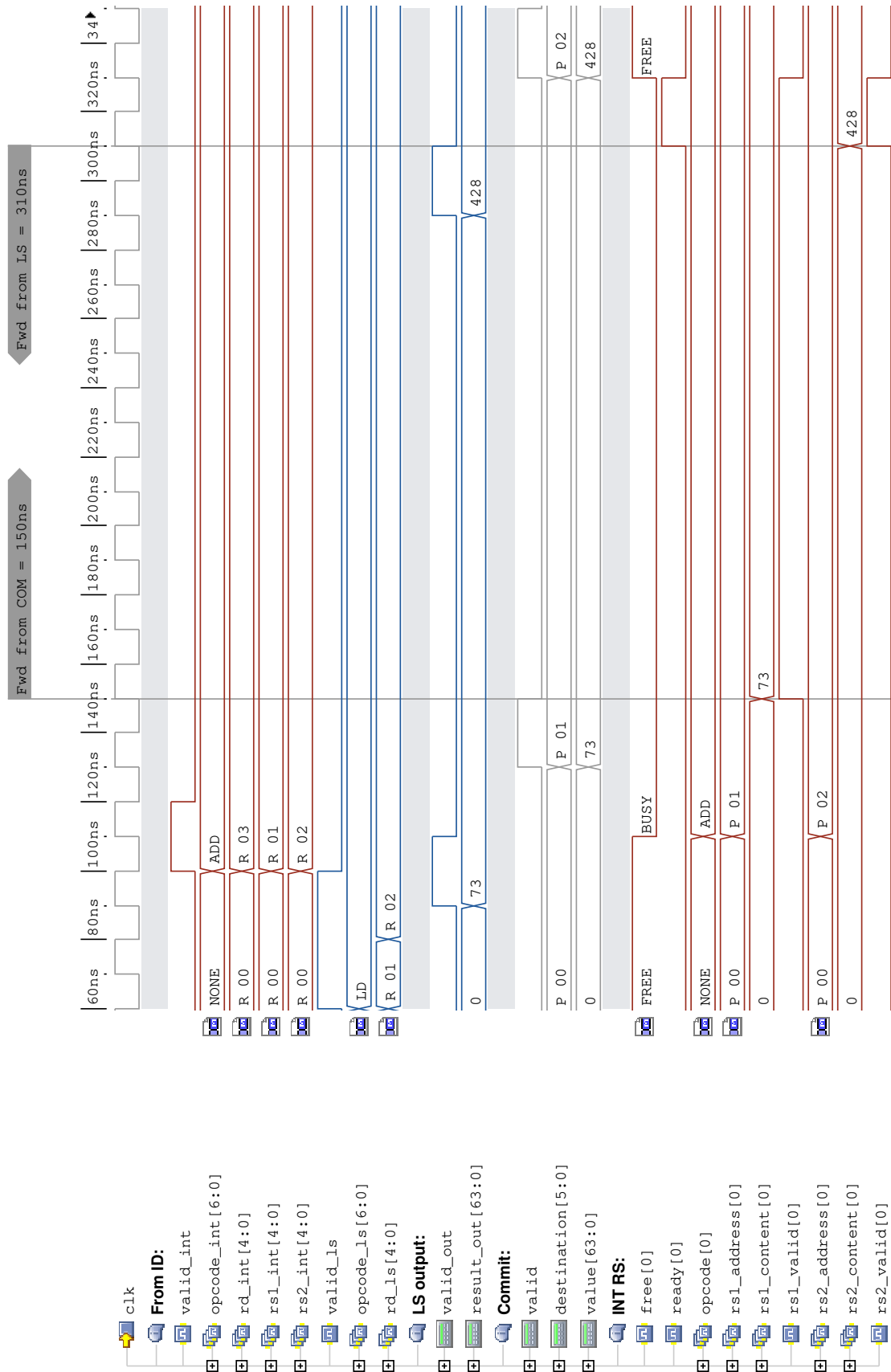
Figure 17: Waiting for source operands in reservation stations

28

5.4 Out-of-order execution

Building on the example from the last subsection, I now want to show how an instruction can overtake a stalling instruction in front of it.

Let us revisit the code from before:

```
1   LD    R1     0(R0)
2   LD    R2     0(R0)
3   ADD   R3     R1     R2
```

We have seen how the ADD instruction waits in the first slot of the integer reservation station for both of its source operands.

I am now introducing a second ADD instruction, two cycles after the first:

```
    ADD   R4     R1     R1
```

This one only needs the result of the first LD to execute.

Figure 18 shows the first two slots of the integer reservation station. I set up the simulation so that the two LD instructions would behave as before. That is, return the same values and have one cache hit and one cache miss. This time, I used the colours red and blue to distinguish signals associated with the two source operands. We can see that the second ADD is loaded into the reservation station just as the first LD commits its result (73) at 150 ns. The value is forwarded to both ADD instructions. As before, the second LD causes a cache miss and so rs2 of the first ADD takes another 10 cycles to become available. The second ADD instruction, however, only needs the first result. It can start execution at 170 ns, many cycles before the first ADD, even though it is behind it in program order. This is out-of-order execution.

The second ADD still needs to wait in the ROB for the first one to finish before it can commit in-order to the RF. But its result can be forwarded to waiting instructions many cycles before that. Furthermore, we could imagine up to nine more ADD instructions to execute while the first one still waits for its source operand. This is how out-of-order execution can speed up computers.
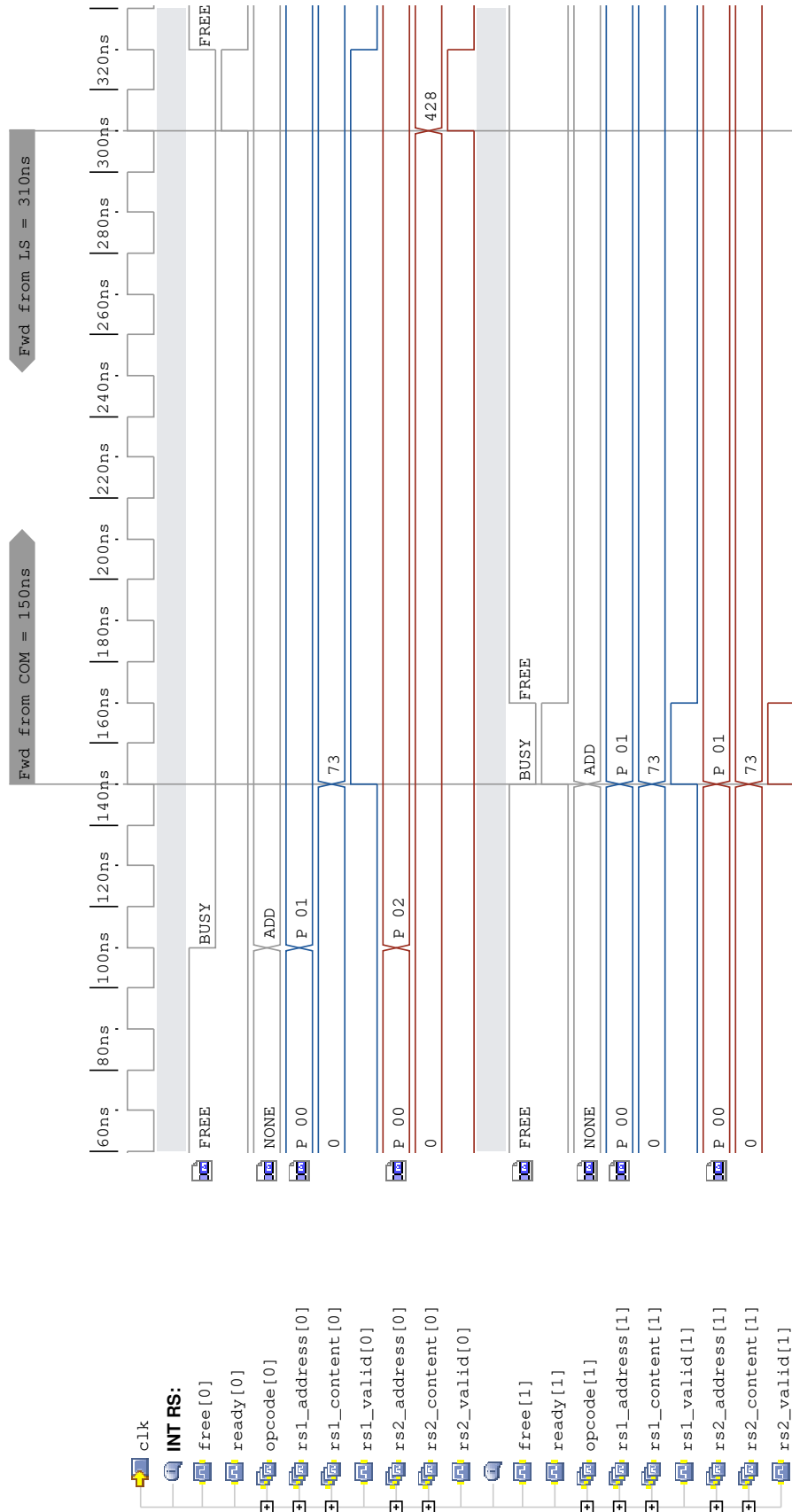
Figure 18: Instruction overtaking stalling instruction inside reservation station

6 The limits of superscalar execution

My design is a very simple prototype that can at most issue/commit two instructions per clock cycle. But we can easily imagine how this design could be scaled up to many more execution units, each with their own reservation station. There are two main reasons for why this is only useful to a certain degree.

The first point is that every piece of code inherently only has a few instructions that can be executed at any point. This is the data-flow limit. Since 1970, researches have made many different educated guesses on the limit of ILP in code, ranging from 2 to 50 [1, p. 24]. This value is heavily application dependent and was generally underestimated before the introduction of speculative execution using branch prediction. A basic block is a piece of code with no branches. Without branch prediction, ILP could only be exploited inside these small basic blocks. Better branch prediction means that the amount of code across which ILP can be used increases. But even with advanced branch predictors, there will always be a hard limit (see section 8 for how data value speculation could help).

The second problem is that at some point, the hardware will become so complex that the potential speed-up is negated. Critical points include:

- **The IF stage**
  The number of instructions that can be fetched in a cycle is limited by the size and architecture of the instruction cache and its cache lines. Misalignment and control dependencies also limit this stage.

- **The RF**
  A broad pipeline requires a heavily multi-ported RF. This greatly affects area and power consumption [18, p. 4].

- **The forwarding logic**
  The complexity of the forwarding paths grows both with the number of execution units and the number of slots in the reservation stations.

- **The ROB**
  A broader pipeline means that the ROB needs to be able to tag and commit more instructions. The bigger problem is that it also needs to potentially save more results every clock cycle. This is a random access (see section 4.2.2) and faces the same restrictions as the RF.

Because of these issues, manufacturers in the mid-noughties started focusing on thread-level parallelism [2, p. 245f]. But figure 6 shows that exploiting ILP has continued to be important, even as the number of cores and threads increased.

7   The dangers of out-of-order speculative execution: Meltdown and Spectre

The separation of the architectural state and the micro-architectural (physical) state of a machine has opened up new security risks. Most notably, 2018 saw the disclosure of **Meltdown** and **Spectre**. These vulnerabilities allow attackers to access all memory in a system.

If an instruction accesses memory that it is not allowed to read, an exception is raised. However, the exception is only raised when the instruction is committed. Until then, the value can be used by **transient instructions**. These are instructions that will not be committed because of the exception. They cannot affect the correctness of the program and therefore used to be regarded as harmless. However, while these transient instructions cannot affect the architectural state of the machine, i.e. the register file, they *can* affect the micro-architectural state of for example the cache. An attacker can use the illegally accessed value to calculate a memory address and load that cache-line into the cache. By timing the access time to that cache-line, the attacker can reconstruct the address and thereby the secret value. This is Meltdown. [19]

Spectre makes use of **speculative execution**. If there is a conditional branch in the instruction flow, modern processors will speculatively execute following instructions until the condition is evaluated. If the branch direction was mispredicted, the instructions are discarded. If not, the processor just saved time. Out-of-order execution can make speculative execution more prominent because the outcome of a conditional branch instruction may depend on preceding instructions that have not been completed yet. The branch predictor can be trained to predict a certain outcome. If a branch is taken 100 times in a row, it will likely be taken again. An attacker can exploit this by hiding instructions that use secret information behind a branch that is not taken. The branch predictor can be trained to mispredict this branch and the processor will be tricked to execute these instructions. Similarly to Meltdown, there now is a time window in which the micro-architectural state can be used to leak secret information. [20]

The underlying reason for these vulnerabilities is the discrepancy between the architectural and the physical state of modern processors. Computer architects now need to carefully consider the visibility of the micro-architectural state.

8   Conclusion and Outlook

I have presented one possible implementation of a simple, superscalar out-of-order processor. I demonstrated the major advantages and challenges that these techniques bring.

8.1   Further work

The aim of this thesis was to design hardware structures to allow for superscalar out-of-order execution. I did not attempt to build a complete RISC-V processor. There is still work left to do to achieve a fully functioning CPU. This subsection lists some of this necessary work as well as other improvements that can be made to this design.

Support for other instructions
The processor is currently only able to support ADD and LD instructions from the RISC-V instruction set. Adding support for the rest will change the output registers of the ID stage. The processor will also need to handle the immediate value which replaces one of the source operands. This version simply drags it along, since it does not affect the output. Support for floating-point instructions will require a second RF and more renaming logic.

Speculative execution
My design assumes a linear instruction flow without branches and control dependencies that stem from them. Branch prediction with speculative execution has become the norm in processors and provides a significant speed-up. It will eventually need to be incorporated.

Exceptions
Exceptions can be caused by things like dividing by zero, over- or underflows and page-faults during memory access. One major reason for having a ROB to ensure in-order commits is to have **precise exceptions**. This means that when an instruction causes an exception, the *architectural* state of the RF can be saved. That means that all preceding instructions have committed their results and all following instructions have not. The ROB will need to keep some sort of exception tag to support this.
To handle exceptions and interrupts, the processor will also need to be able to save and restore information in the RAT.

Dependencies in the store queue

The RAT ensures that dependencies between registers are respected. Once the processor is capable of proper memory access, the LS unit will need to ensure that it also respects dependencies between memory locations.

Simultaneous multithreading (SMT)

SMT allows for multiple threads to run in parallel on the same core, i.e. using the same execution units. To clearly separate threads, each requires their own register file and ROB among other resources that will depend on implementation details. [1, p. 584-599]

Tighter scheduling

If a reservation station is empty and the corresponding execution unit is idle, there is no reason for the instruction to waste a clock cycle in the reservation station. One could introduce some sort of skipping mechanism. Similarly, instructions that have finished execution sometimes spend one cycle in the ROB, when they could be committed directly because they are the oldest instruction in the pipeline.

Loosen demands on ID

This prototype has very strict demands on the ID stage. It requires one integer and one load-store instruction. In particular, it needs the integer instruction to be before the load-store instruction in program order. Future versions may want to allow for the opposite order or for two instructions of the same type to be issued at once. It is not clear if the additional hardware required for this is worth the increase in throughput.

Dimensioning the ROB, the RF and the reservation stations

In my example, with only two execution units and a cache miss penalty of only ten cycles, the RF and ROB are grossly oversized. It is very hard to construct realistic examples where more than ten registers or tags are being used at any point. However, if we increase our cache miss penalty to a much more realistic 100 cycles, we can imagine 100 ADD instructions to execute and then wait in the ROB for a LD to finish.

It is also not clear how many instructions a reservation station should be able to hold. If a reservation station is full, the pipeline needs to stall. So from this perspective, larger reservation stations are useful. However, more instructions in the reservation stations means more complex compare logic in the forwarding paths.

The optimal size may also depend on the execution unit it is servicing.

Once the design incorporates more execution units with more variable delay, it will be useful to analyse how many ROB and RF entries are really necessary.

The perfect mix and number of execution units

To achieve maximal usage of all execution units, their number and mix should mirror the mix of instruction types in the code that the processor will be expected to run. There is no use in having a lot of integer units if the vast majority of instructions will be floating point calculations.

The instruction mix is heavily application dependent and will change during and across threads. It can, therefore, be useful to have more execution units than the pipeline width can support at any time. This allows for the mix of execution units in use to change dynamically at runtime.

However, more execution units are not always better. More execution units means more forwarding paths and more compare logic in the reservation stations. The load-store unit is particularly hard to duplicate because of memory synchronisation problems. [1, p. 205f]

Execution order

If multiple instructions in a reservation station are ready to execute, the one in the slot with the lowest number is picked. It could be useful to instead execute in program order if there is such a conflict. This is because other instructions are more likely to depend on instructions that have been in the pipeline for longer.

Placement of the pipeline stages

It is not clear if the II stage in this design is perhaps too long. It might be useful to do the register renaming in the ID stage.

Data speculation

Similar to branch prediction, researchers in the 90's have proposed value prediction as a way to exceed the data-flow limit [1, p. 261f]. Just like processors can speculate on which branch will be taken, they could speculate on the output of a calculation or a load. These ideas have recently been revisited by other researchers, however, they do not seem to be implemented by any current processors [21].

8.2 Alternative design choices

There also are other variants of ideas that I have implemented. Further analysis is required to decide if one is inherently superior or if the choice is application dependent. Such choices include:

Centralized vs distributed reservation stations

Rather than having one reservation station per execution unit, one can imagine one big centralized reservation station where all instruction types wait for their source operands. It would in most cases be more filled than the average reservation station and so would allow for more instructions to be "in flight" at any point. This could increase the amount of ILP that can be exploited. However, a centralised reservation station will also need complicated selection logic to service all execution units. [1, p. 201-203]

Alternative register renaming schemes

There are other ways to implement register renaming and to organise the register file. Other approaches might use both a rename register file (RRF) and an architectural one (ARF) which mirrors the RRF to reflect the non-speculative architectural state [1, p. 239]. Or they might only need one physical RF of the size of the architectural RF but a more complicated scheme of virtual rename registers [22]. In Tomasulo's original paper [3], names for reservation stations were used as register aliases. So that instead of two source registers an instruction can have two source reservation stations. One can also use the position in the ROB for this purpose [2, 186f]. Unfortunately, there often are no sources that describe the implementation details of these ideas, which makes it very difficult to compare their advantages and problems.

It is also usually not clear how register renaming is implemented in today's machines, so I cannot tell how close my solution is to the industry standard if there is one. It is, however, very similar to the way that the Berkeley Out-of-Order Machine (BOOM) implements register renaming [23, p. 25].

Dynamic vs static scheduling

One might wonder why the compiler has to pretend that there are only 32 (architectural) registers when in reality we have 64. One could just use a larger architectural register file in the first place and let the compiler remove false data dependencies and schedule instructions to be executed in parallel.

This kind of static scheduling at compile time is the approach that **very long in-**

struction word (VLIW) ISAs take. Each instruction contains multiple operations that can be executed concurrently. This can greatly decrease hardware complexity, as there is no more need for things like register alias tables or reservation stations. However, there are other problems that arise. The compiler cannot anticipate all dependencies, as they can be caused by external input at run time or cache misses which can cost many clock cycles. A small architectural RF with register renaming in hardware also has the advantage that it requires fewer register address bits in the instruction format. It also allows for the same code to be executed on many different machines with different sized RFs as long as they implement the same ISA. This can be particularly useful for backwards compatibility. [2, p. 192-196]

APPENDIX

Why you still cannot buy a 10 GHz laptop in 2019

The average runtime of an instruction is given by

$$
\begin{aligned}
\text{Runtime} &= \frac{\text{Cycles}}{\text{Instruction}} \times \frac{\text{Time}}{\text{Cycle}} \\
&= \frac{1}{\text{IPC} \times \text{clock rate}}
\end{aligned}
\tag{1}
$$

where IPC stands for "instructions per cycle".

Making a processor superscalar is an attempt to reduce runtime by increasing IPC count. However, we can see that it would be just as effective to increase the clock rate. It might seem easier to keep the simple linear pipeline and make our clock twice as fast than to design a complicated pipeline that can handle two instructions at once. Especially now that we have seen how much this increases hardware complexity, we might ask, why do hardware designers bother?

To answer this question, this section contains several figures plotting various processor characteristics against their release year, with data from the Stanford CPU database [24]. We should note that in recent years it has been becoming harder to account for things like thermal design power (TDP), supply voltage (V_{dd}) and frequency due to dynamic voltage and frequency scaling (DVFS). Intel, for example, offers a turbo mode, configurable TDP (cTDP) and a low-power mode [25, p. 87-94]. These allow a processor to exceed its standard performance for short amounts of time, or to trade in performance for a longer battery life. I tried to account for this in figure 21 by plotting a maximum (grey) and a minimum (red) voltage that the manufacturer provided. Such values are unfortunately not always available or representative of the actual average value in real-world applications, especially when it comes to frequency and power [26]. There also may be some selection bias in the data for manufacturers that make their specifications more available. While the Stanford CPU database is very valuable, the above-mentioned issues should be kept in mind when using this data.

Figure 19 shows how clock rate has changed over the last few decades. Up to about 2004, we see an exponential increase (note the logarithmic scale). This can mostly be attributed to shrinking feature size. If component size drops exponentially, and propagation speed stays constant then path delay drops exponentially. This means that frequencies can rise exponentially ($f = \frac{1}{T}$).
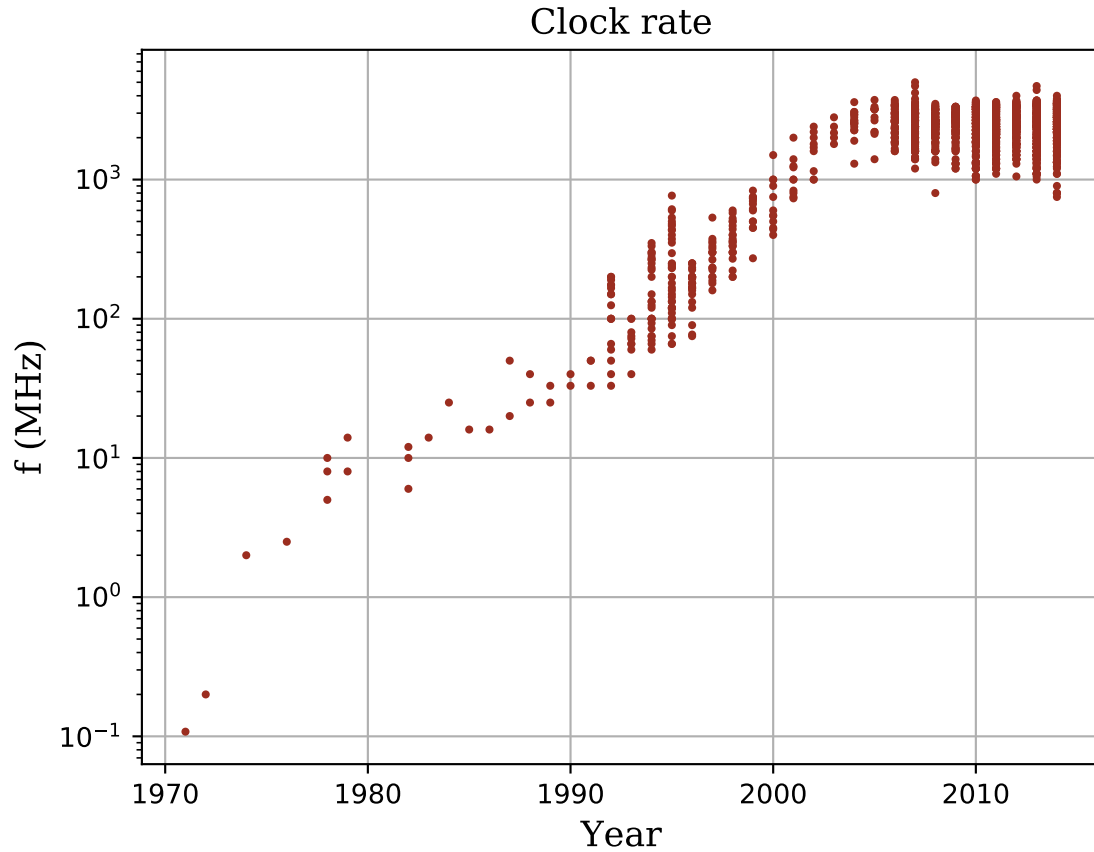
Figure 19: Clock rate over the last 50 years. Drawn with data from [24]

However, since the mid-noughties, clock rate has plateaued at around $3\,\mathrm{GHz}$. This was not an anticipated stagnation. Even in 2005, textbooks still expected this exponential increase to continue. See for example [1, p. 3], where Lipasti and Shen wrote:

> *By 2010, we can expect microprocessors to contain more than* 1 *billion transistors with clocking frequencies greater than* $10\,\mathrm{GHz}$.

While the prediction about transistor count was quite accurate (the POWER7, released in 2010, for example, had around 1.2×10^9 transistors [27, p. 44]), people still dream about $10\,\mathrm{GHz}$ processors in 2019. To explain this plateau, we need to understand how power consumption in CMOS logic works.

The main source of power dissipation is dynamic switching power. Switching power is dissipated wherever parts of the circuitry are being charged/discharged. We can model this as a kind of capacitor which is charged over a resistor. The resistor corresponds to the resistance of all the wires and transistors. This is where power is dissipated in the form of heat.

We are now going to take a closer look at this charging process which is sketched
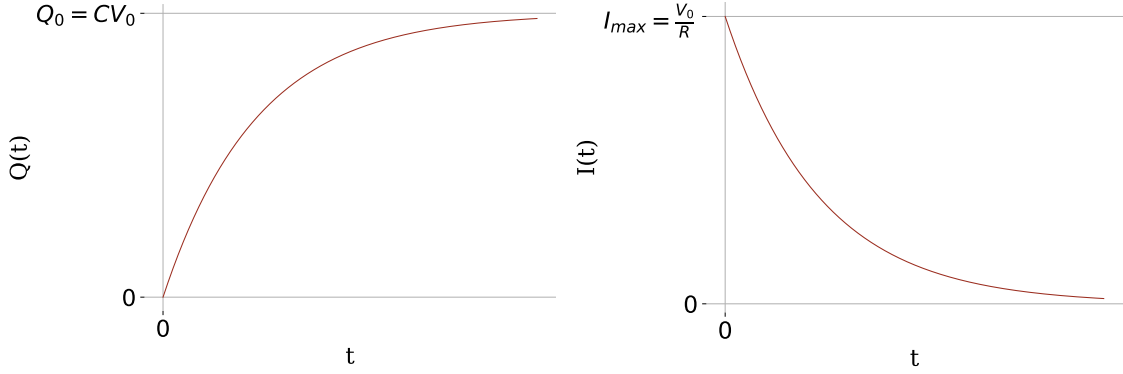
Figure 20: Charge and current for loading a capacitor over a resistor

in figure 20. If we turn on the voltage source at $t = 0$, the charge on the capacitor $Q(t)$ will start to rise to the full $Q_0 = CV_0$, where V_0 is the supply voltage and C is the capacity of the capacitor. Figure 20 also shows the current which flows through the resistor. It is simply the derivative of the first curve $I(t) = \frac{d}{dt}Q(t)$. This current determines how much power is dissipated in the resistor.

Every charge transfer dQ onto the capacitor comes with a transfer in energy, given by

$$dE = V\,dQ \tag{2}$$

Capacity C is defined as charge Q per voltage V at any point in time, so we can write this as

$$dE = \frac{Q}{C}dQ \tag{3}$$

For a full charge to Q_0 we find[1]:

$$\Delta E = \int_0^{Q_0} \frac{Q}{C}dQ = \frac{1}{2}\frac{Q_0^2}{C} = \frac{1}{2}CV_0^2 = \frac{1}{2}Q_0V_0 \tag{4}$$

However, the voltage source supplies the full Q_0V_0. So the other $\frac{1}{2}Q_0V_0$ are dissipated in the resistance in the form of heat. If we also include the heat dissipated in the discharge, then the total energy lost per "clock cycle" is

$$\frac{\Delta E}{cycle} = \alpha Q_0 V_0 = \alpha CV_0^2 \tag{5}$$

where I have included an activity factor α. This accounts for the fact that not every transistor is switching every clock cycle.

[1]In practice, we will not reach that voltage, however the calculations are similar if we only require a charge of $0.95\,Q_0$.

Now, power is $P = \frac{\Delta E}{\Delta t}$ and frequency is $f = \frac{cycles}{\Delta t}$, so

$$P = \frac{\Delta E}{\Delta t} = \frac{\Delta E}{cycle} \times \frac{cycles}{\Delta t} = \alpha C V_0^2 f \tag{6}$$

The capacitance of each transistor has historically scaled down with CMOS dimensions [28] but because of the increase in transistor numbers, we should expect the total capacitance to stay constant.

Equation 6 tells us that power dissipation scales quadratically with voltage. To keep power low, voltage needs to be kept low. Figure 21 shows that supply voltage has indeed decreased over the last two decades. The first plateau at 5 V simply is due to convention before a decrease in power became necessary [29]. The thermal voltage
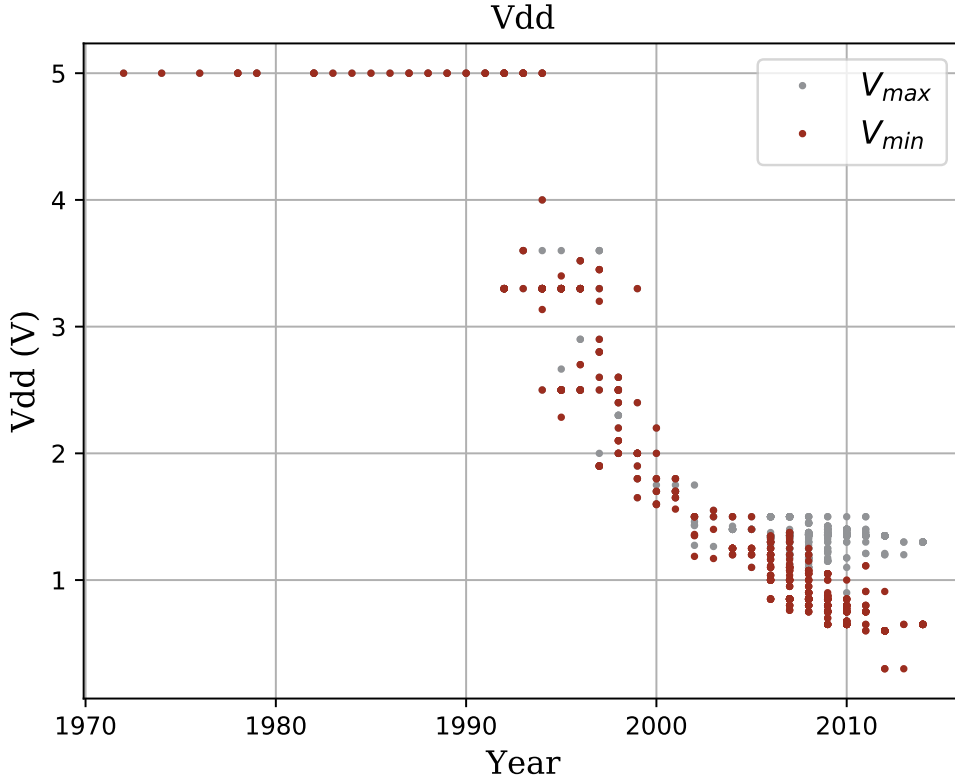

Figure 21: V_{dd} over the last 50 years. Drawn with data from [24].

$\frac{kT}{e} \approx 26\,\mathrm{mV}$ (at 300 K) gives an ultimate lower limit on the operating voltage [30]. Lower operating voltages also increase leakage currents [31] [32]. This has resulted in V_{dd} plateauing at around 1 V (with some manufacturers allowing for lower values in low-power modes). The change in voltage is, however, rather minor compared to the change in the order of magnitude of the clock frequency in figure 19. So we would expect power to scale very similarly to clock frequency, as $P \propto f$, according
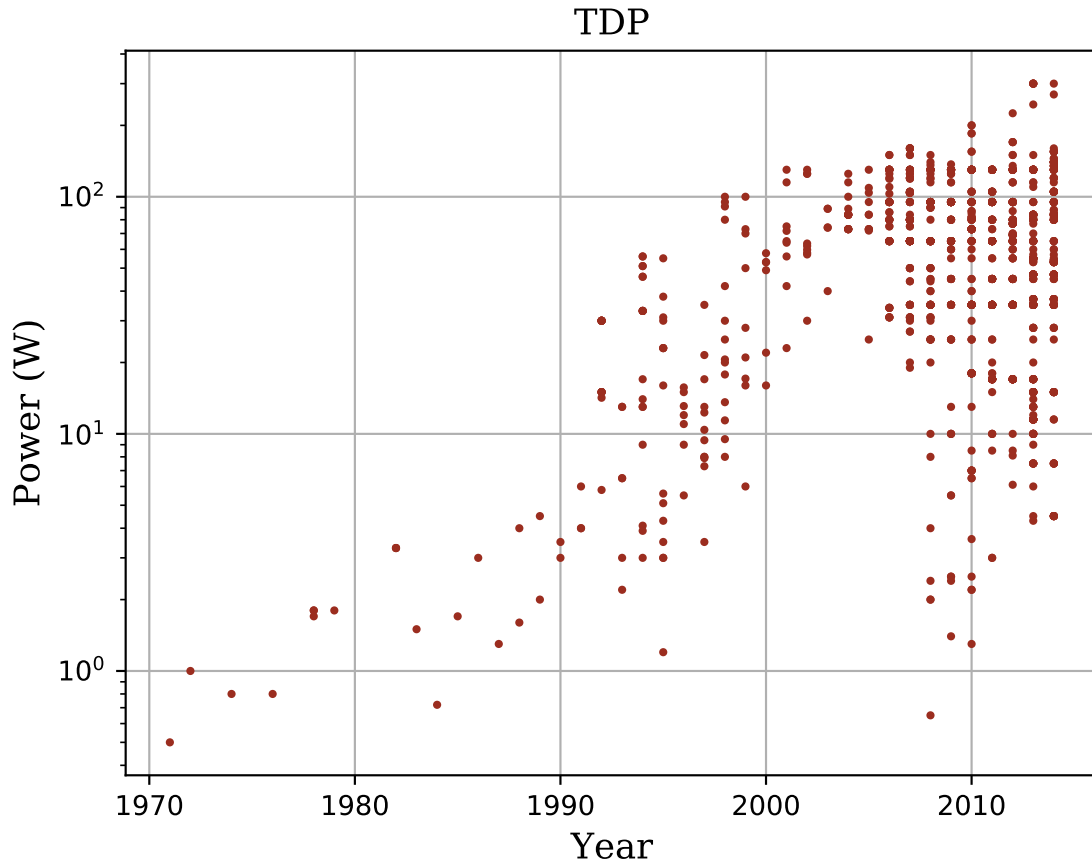
Figure 22: TDP over the last 50 years. Drawn with data from [24]

to equation 6.And indeed, figure 22 shows that this is the case. It plots thermal design power which is the maximum amount of power that a processor is designed to dissipate. More power means that the system needs to have a more powerful cooling system. In [31], Horowitz claims that the limit to air cooling for personal computers is around 100 W. The data from the Stanford CPU DB seems to support that estimate. As figure 22 demonstrates, the upper limit for thermal design power (TDP) has stagnated and it did so at around the same time as clock rate (compare to figure 19).

Figure 22 also shows that since then, manufacturers have started offering a wide array of products from low-power to high-power devices. Many researchers argue that this trend in application-specific optimisation has allowed and will continue to allow scaling of performance even if CMOS scaling laws are coming to an end [31] [33] [34] [35].

There are two other sources of power, short-circuit power and static power. Short-circuit power is dissipated every clock cycle when both nMOS and pMOS transistors

are briefly on and there is a short circuit between V_{dd} and GND. Static power is dissipated due to leakage currents in transistors. The latter is becoming a greater problem as transistors get smaller. But the similarity between the change in clock rate and the change in power suggests that it can be neglected in this discussion.

This estimation explains why we have seen a plateau in clock speed over the last decade. Higher clock speeds will require a more sophisticated cooling method or transistor design.

Alternative approaches to computing

These limits could also potentially be overcome by pursuing more exotic approaches to computing, like asynchronous or neuromorphic logic. However, the investment barrier might be too high for such radical changes. In [31] Horowitz states that he is "sure there are better technologies out there" but that we might be stuck with CMOS for computing because of two big reasons. Firstly, we have invested enormously in the physical manufacturing process of CMOS technologies and any competition would have to have a huge inherent advantage to compete with it. Secondly, CMOS VLSI has greatly influenced our design abstractions. A new technology would require a completely new design flow and new tools to compete.

Single-core performance is still rising, even though clock speeds have stagnated [29]. But when it eventually reaches its limits and all inherent parallelism is exploited, other technologies might have a chance to shine. Until then, researchers looking into ideas like neuromorphic, asynchronous or quantum logic need to develop tools and design abstractions to make their technologies viable alternatives.

Assembly code in this thesis

This thesis occasionally contains assembly code snippets. I restricted myself to ADD and LD instructions from the RV64I Base Integer Instruction Set.
RISC-V is a load-store architecture, which means that memory is only accessed by load and store instructions that move data between registers and memory. Calculations can only be performed on registers.
A LD instruction

```
LD    rd    imm(rs1)
```

loads a 64-bit value from memory and saves it in rd. The effective byte address is calculated by adding the immediate imm to the content of rs1.
An ADD instruction has the form

```
ADD   rd    rs1    rs2
```

where rs1, rs2 and rd are registers. It adds the contents of the source registers rs1 and rs2 and saves the result in the destination register rd.

[17, p. 5, p. 18f, p. 31]

About RISC-V

RISC-V is an open-source ISA. It was developed at the University of California, Berkeley for research and education purposes. RISC stands for reduced instruction set computer. The idea behind it is that few simple instructions are easier to work with than many complex instructions.
RISC-V gives researchers in academia the opportunity to study and develop actual hardware implementations without having to pay royalties for commercial ISAs.

[36] [17, p. 1-3]

List of Figures

List of Acronyms

BOOM   Berkeley Out-of-Order Machine

CMOS   Complementary metal-oxide-semiconductor

cTDP   Configurable thermal design power

DVFS   Dynamic voltage and frequency scaling

ID   Instruction decode (stage)

II   Instruction issue (stage)

ILP   Instruction level parallelism

INT   Integer (unit)

IPC   Instructions per cycle

ISA   Instruction set architecture

LS   Load-store (unit)

PC   Program counter

RAT   Register alias table

RF   Register file

RISC   Reduced instruction set computer

ROB   Reorder buffer

RS   Reservation station

RTL   Register-transfer level

SMT   Simultaneous multithreading

TDP   Thermal design power

VLIW   Very long instruction word (architecture)

References

[1] Shen, J. P., & Lipasti, M. H. (2013). *Modern processor design: fundamentals of superscalar processors*. Waveland Press.

[2] Hennessy, J. L., & Patterson, D. A. (2011). *Computer architecture: a quantitative approach*. Elsevier.

[3] Tomasulo, R. M. (1967). An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development, 11(1)*, 25-33.

[4] Silc, J., Robic, B., & Ungerer, T. (2012). *Processor architecture: from dataflow to superscalar and beyond*. Springer Science & Business Media.

[5] Oehler, R. R., & Blasgen, M. W. (1991). IBM RISC System/6000: Architecture and performance. *IEEE Micro, 11(3)*, 14-17.

[6] TOP500 reports. Retrieved from `top500.org` on 08.06.2019.

[7] White, S. W., & Dhawan, S. (1994). POWER2: Next generation of the RISC System/6000 family. *IBM Journal of Research and Development, 38(5)*, 493-502.

[8] O'Connell, F. P., & White, S. W. (2000). Power3: The next generation of PowerPC processors. *IBM Journal of Research and Development, 44(6)*, 873-884.

[9] Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., & Sinharoy, B. (2002). POWER4 system microarchitecture. *IBM Journal of Research and Development, 46(1)*, 5-25.

[10] Kalla, R., Sinharoy, B., & Tendler, J. M. (2004). IBM Power5 chip: A dual-core multithreaded processor. *IEEE micro, (2)*, 40-47.

[11] Le, H. Q., Starke, W. J., Fields, J. S., O'Connell, F. P., Nguyen, D. Q., Ronchetti, B. J., ... & Vaden, M. T. (2007). Ibm power6 microarchitecture. *IBM Journal of Research and Development, 51(6)*, 639-662.

[12] Sinharoy, B., Kalla, R., Starke, W. J., Le, H. Q., Cargnoni, R., Van Norstrand, J. A., ... & Nguyen, D. Q. (2011). IBM POWER7 multicore server processor. *IBM Journal of Research and Development, 55(3)*, 1:1-1:8.

[13] Sinharoy, B., Van Norstrand, J. A., Eickemeyer, R. J., Le, H. Q., Leenstra, J., Nguyen, D. Q., ... & Levitan, D. (2015). IBM POWER8 processor core microarchitecture. *IBM Journal of Research and Development, 59(1)*, 2:1-2:21.

[14] Sadasivam, S. K., Thompto, B. W., Kalla, R., & Starke, W. J. (2017). IBM Power9 processor architecture. *IEEE Micro, 37(2)*, 40-51.

[15] Intel Corporation. (2011). *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Volume 1. Order Number: 253665-039US.

[16] Advanced Micro Devices (2017). *Software Optimization Guide for AMD Family 17h Processors*. Revision 3.00. Publication No. 55723.

[17] Waterman, A. & Asanovic, K. (2017). *The RISC-V Instruction Set Manual, Volme I: User-Level ISA*. Document Version 2.2.

[18] Mittal, S. (2017). *A survey of techniques for designing and managing CPU register file. Concurrency and Computation: Practice and Experience*, 29(4), e3906.

[19] Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Mangard, S., ... & Hamburg, M. (2018). Meltdown. *arXiv preprint arXiv:1801.01207*.

[20] Kocher, P., Genkin, D., Gruss, D., Haas, W., Hamburg, M., Lipp, M., ... & Yarom, Y. (2018). Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*.

[21] Fu, C. Y., Jennings, M. D., Larin, S. Y., & Conte, T. M. (1998). Value speculation scheduling for high performance processors. *ACM SIGOPS Operating Systems Review*, 32(5), 262-271.

[22] Sharafeddine, M., Akkary, H., & Carmean, D. (2013, February). Virtual register renaming. In *International Conference on Architecture of Computing Systems* (pp. 86-97). Springer, Berlin, Heidelberg.

[23] Celio, C., Zhao, J., Gonzalez, A., & Korpan, B. (2019). RISCV-BOOM Documentation.

[24] Stanford VLSI Group. CPU DB. Retrieved from `cpudb.stanford.edu` on 20.04.2019.

[25] Intel Corporation. (2019). *8th and 9th Generation Intel® Core™ Processor Families Datasheet*. Volume 1 of 2, Revision 003.

[26] Huck, S. (2011). *Measuring Processor Power. TDP vs. ACP.* Revision 1.1. Available at `intel.com/content/dam/doc/white-paper/resources-xeon-measuring-processor-power-paper.pdf`

[27] Anselmi, G., Blanchard, B., Cho, Y., Hales, C. & Quezada, M. (2010). *IBM Power 770 and 780 Technical Overview and Introduction.* Available at `www.redbooks.ibm.com/redpapers/pdfs/redp4639.pdf`

[28] Dennard, R. H., Gaensslen, F. H., Rideout, V. L., Bassous, E., & LeBlanc, A. R. (1974). Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits, 9(5)*, 256-268.

[29] Danowitz, A., Kelley, K., Mao, J., Stevenson, J. P., & Horowitz, M. (2012). CPU DB: recording microprocessor history. *Communications of the ACM, 55(4)*, 55-63.

[30] Taur, Y. (2002). CMOS design near the limit of scaling. *IBM Journal of Research and Development, 46(2.3)*, 213-222.

[31] Horowitz, M. (2014, February). Computing's energy problem (and what we can do about it). In *2014 IEEE international solid-state circuits conference digest of technical papers (ISSCC)* (pp. 10-14). IEEE.

[32] Taur, Y., & Nowak, E. J. (1997, December). CMOS devices below $0.1\mu m$: how high will performance go?. In *International Electron Devices Meeting. IEDM Technical Digest* (pp. 215-218). IEEE.

[33] Frank, D. J., Dennard, R. H., Nowak, E., Solomon, P. M., Taur, Y., & Wong, H. S. P. (2001). Device scaling limits of Si MOSFETs and their application dependencies. *Proceedings of the IEEE, 89(3)*, 259-288.

[34] Nowak, E. J. (2002). Maintaining the benefits of CMOS scaling when scaling bogs down. *IBM Journal of Research and Development, 46(2.3)*, 169-180.

[35] Frank, D. J. (2002). Power-constrained CMOS scaling limits. *IBM Journal of Research and Development, 46(2.3)*, 235-244.

[36] Webpage of the RISC-V Foundation. Retrieved from `riscv.org/risc-v-foundation` on 05.06.2019.

Erklärung

Ich versichere, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Heidelberg, den 10.06.2019

Amanda Matthes