Bachelor Project

Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Radioelectronics

Implementation of Instruction Set for RISC-V Processor

Martin Laštovka

Supervisor: doc. Ing. Jíří Jakovenko, Ph.D.
May 2022

BACHELOR‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Laštovka Martin** Personal ID number: **495621**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Radioelectronics**

Study program: **Open Electronic Systems**

II. Bachelor’s thesis details

Bachelor’s thesis title in English:

Implementation of Instruction Set for RISC-V Processor

Bachelor’s thesis title in Czech:

Implementace instrukční sady pro RISC-V procesor

Guidelines:

1) Study the RISC-V instruction set architecture. Choose an open-source implementation of RISC-V processor suitable for a low-power system
2) Describe the communication between instruction and data memory for the chosen processor implementation
3) Propose an optimization that would allow raising the maximum system clock frequency without modifying the processor's microarchitecture
4) Implement the proposed optimization, verify its functionality and measure its effect on the system's properties

Bibliography / sources:

[1] David Patterson, Andrew Waterman, The RISC-V Reader: An Open Architecture Atlas 1st edition, Berkeley: Strawberry Canyon LLC, 2018
[2] Jakub Stastny, FPGA prakticky. BEN Praha 2010
[3] David Patterson, John Hennessy, Computer Architecture: A Quantitative Approach 5th edition, Morgan Kaufmann 2011

Name and workplace of bachelor’s thesis supervisor:

doc. Ing. Jiří Jakovenko, Ph.D. Department of Microelectronics FEE

Name and workplace of second bachelor’s thesis supervisor or consultant:

Date of bachelor’s thesis assignment: **10.02.2022** Deadline for bachelor thesis submission: **20.05.2022**

Assignment valid until: **30.09.2023**

doc. Ing. Jiří Jakovenko, Ph.D. Supervisor’s signature	doc. Ing. Stanislav Vítek, Ph.D. Head of department’s signature	prof. Mgr. Petr Páta, Ph.D. Dean’s signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor’s thesis, the author must state the names of consultants and include a list of references.

. | Date of assignment receipt

Student’s signature

Acknowledgements

I would like to thank doc. Ing. Jíří Jakovenko, Ph.D. for supervising this thesis. This work would have been impossible without the aid and advice of my colleagues at ASICentrum s.r.o. Namely, I would like to thank Ing. Jakub Šťastný Ph.D. for his invaluable time. Lastly, I wish to express my gratitude to my family for emotional and financial support throughout the course of my studies.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the Methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, 20 May , 2022

Abstract

This bachelor thesis is dedicated towards optimization of a critical combinational path in a low power and low die area digital system that contains a RISC-V processor. The critical path is located in the instruction memory interface of the mentioned processor.

This thesis proposes several solution architectures; the chosen mechanism then works on the principle of speculative instruction prefetching. The first design is a simple static branching predictor. The second, more refined design is a modified branch predictor with either one-level or two-level prediction.

The second design was in different configurations tested with the CoreMark benchmark so that we could optimize the design parameters such as memory size, prediction mechanism, etc.

Our IP was verified throughout the design process on the RTL level in a simple System Verilog testbench. Next, the design was implemented in the Artix-7 xc7a100tcsg324-2 FPGA. Here, we successfully ran logic synthesis and static timing analysis. Lastly, we simulated the design at the gate-level.

Keywords: RISC-V, RI5CY, VDHL, FPGA, branch prediction, hardware cache, instruction prefetching, CoreMark, embedded system

Supervisor: doc. Ing. Jíří Jakovenko, Ph.D.

Abstrakt

Tato bakalářská práce se zabývá optimalizací kritické kombinační cesty v digitalním systému s nízkým příkonem a malou plochou, který obsahuje RISC-V procesor. Kritická cesta existuje v rámci rozhraní instrukční paměti zmíněného procesoru.

Práce podává několik návrhů řešení, vybraný mechanismus poté funguje na bázi spekulativního přednačítání instrukcí. První návrh je jednoduché statické větvení. Druhý, pokročilejší návrh byl implementován jako upravený prediktor větvení na bázi jednoúrovňové nebo dvouúrovňové predikce.

Druhý návrh byl v různých konfiguracích podroben zátěžovému testu CoreMark za účelem optimalizace parametrů jako je velikost paměti, mechanismus predikce apod.

Naše IP bylo v průběhu návrhového procesu verifikováno na RTL úrovni v jednoduchém testovacím prostředí na bázi jazyku System Verilog. Dále byl návrh implementován v Artix-7 xc7a100tcsg324-2 FPGA. Zde proběhla úspěšně logická syntéza a statická časová analýza. Na závěr byl návrh simulován na hradlové úrovni.

Klíčová slova: RISC-V, RI5CY, VDHL, FPGA, predikce větvení, hardwarová mezipaměť, přednačtení instrukcí, CoreMark, vestavěný systém

Překlad názvu: Implementace instrukční sady pro RISC-V procesor

Contents

Figures

Tables

Chapter 1

Abbreviations

- **SoC** - System on Chip
- **CPU** - Central Processing Unit
- **NVM** - Non-Volatile Memory
- **IP** - Intellectual Property
- **RAM** - Random Access Memory
- **IAS** - Instruction Address Stream
- **ISA** - Instruction Set Architecture
- **IF** - Instruction Fetch
- **ID** - Instruction Decoding
- **EX** - Execute
- **EM** - External Master
- **WB** - Write Back
- **LRU** - Least Recently Used
- **FIFO** - First In, First Out
- **FSM** - Finite State Machine
- **IPS** - Instruction Per Second
- **CP** - Critical (Combinational) Path
- **MSB** - Most Significant Bits
- **LSB** - Least Significant Bits
- **HRT** - HistoRy Table
- **PT** - Pattern Table

- **DPLL** - Digital Phase Locked Loop
- **SDF** - Standard Delay Format
- **DUT** - Design Under Test
- **ASIC** - Application-Specific Integrated Circuit

Chapter 2

Introduction

2.1 Motivation

In a digital logic system, the use of an asynchronous read Non-Volatile Memory (NVM) can easily bottleneck the system maximum clock frequency because of its long read delay. In our case, we deal with a small-sized low-power embedded System on Chip (SoC) which at its core contains a 32-bit RISC-V CPU based on the open-source RI5CY from the PULP platform [1]. This 4-stage pipeline core is used in ASICentrum s.r.o. in many low-power designs because of its small die area and good efficiency.

The asynchronous NVM is used here as the CPU instruction memory, thus creating a Critical (Combinational) Path (CP) with respect to the CPU clock. The path may be broken down into the following (see **Figure 2.1**):

1. Logic between the CPU registers and its instruction fetch address port,
2. spatial signal propagation delay,
3. NVM controller delay,
4. memory read time.

All of the delays are constant and characteristic of our design, except for spatial signal propagation delay, which is a result of the place-and-route process.

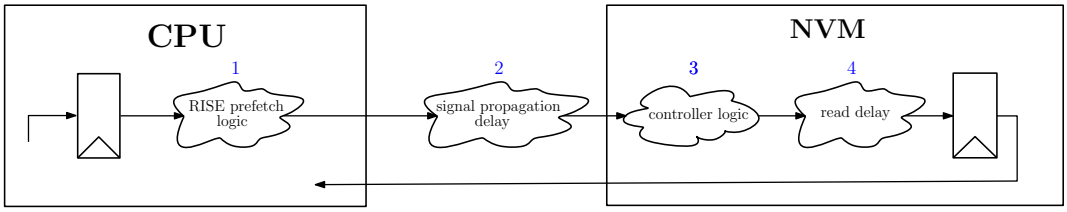

Figure 2.1: Critical combinational path

2.2 Objective

Our goal is to shorten this CP, thus removing the maximum clock frequency bottleneck and allowing the system to run faster. A major design constraint is that we can not modify the CPU microarchitecture — our solution shall be a self-contained entity. This is advantageous as it spares us from difficult integration of our solution within the CPU. Especially the verification effort would be far more intensive as it would require re-testing all of the complex CPU functionalities.

Another advantage of a self-contained solution is that it is partially or fully independent of the CPU internal structure and can therefore be easily reused. Also, we will not have to redesign our solution if the CPU microarchitecture is modified later on.

A disadvantage of this approach is that since we have only the external signals of the CPU and NVM at our disposal and we cannot influence any of the internal signals directly, our design might have to be a lot more complex than if we were to modify the CPU directly. This means higher chip area usage and more power consumed. Furthermore, our solution must comply with the system low-power and low die area design paradigm. We will have to make a trade-off between the efficiency of our solution with its power and chip area resource usage.

Chapter 3

Solution methods

3.1 Instruction fetch protocol

Our CPU inherits from RI5CY its Instruction Fetch (IF) data bus interface, which complies to the OBI (Open Bus Interface) protocol [2]. OBI is a simple, request-grant based protocol used for point-to-point on-chip communication. In our case, the protocol is used in a single master, single slave topology with many of its optional OBI signals left unused. A simplified OBI transfer is as follows:

- Address phase starts and master asserts request signal, the address is valid,
- slave responds with asserting grant signal, indicating readiness to accept the address,
- on a rising clock edge, the address phase is completed, response phase starts,
- slave asserts rvalid signal when the data are valid, the response phase is completed.

In the **Figure 3.1** we can see an example of a typical transaction compliant with our modified version of the OBI protocol. The CPU supports a RISC-V compressed instructions extension, but the instruction fetches are strictly word-aligned and half-word fetches are handled internally.

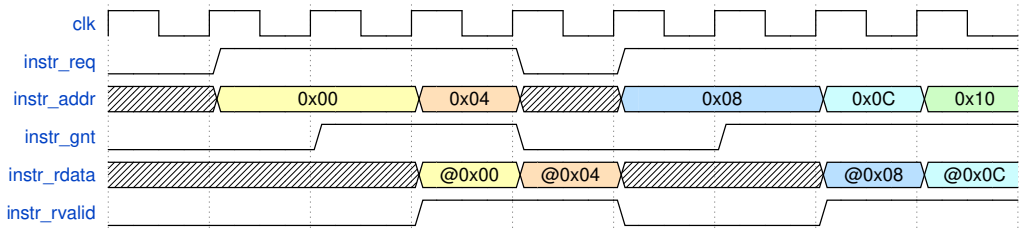

Figure 3.1: Typical bus transaction

3.2 Pipelining

A naive solution to our problem would be to pipeline the address channel (request and instruction address signals) of the instruction fetch interface simply by making the signals registered. This would then remove the instruction address logic and signal propagation delay from the CP. However, such a modification would result in two clock cycles per instruction needed — a serious decrease in instruction throughput. The effect of this modification on a typical transaction is shown in **Figure 3.2**. Such a solution would be extremely efficient in terms of power consumed per clock edge and chip area used. However, because of the dramatic decrease in code execution latency, we deem it as non-viable for our system.

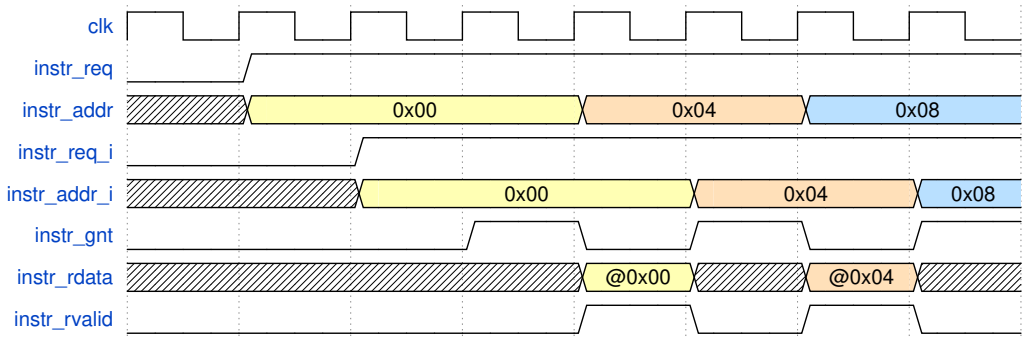

Figure 3.2: Pipelined bus transaction

3.3 Prefetcher

3.3.1 Theory of operation

A different approach is to design a custom prefetcher IP, a so-called RISE (RISc-v prEfetch), that would be placed in between the CPU and the NVM. A system-level view of this modification is shown in **Figure 3.3**

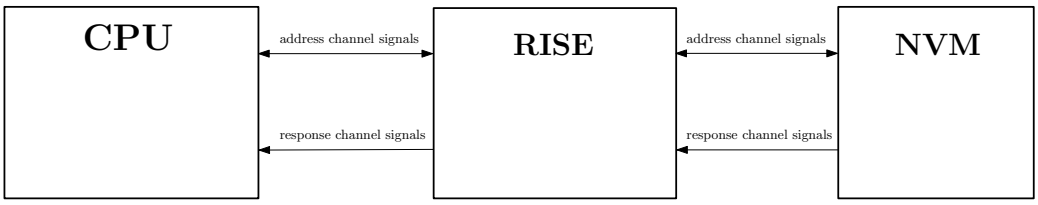

Figure 3.3: Modified system block diagram

The purpose of this digital logic block would be to speculatively fetch an instruction one clock cycle ahead and then compare it in the next cycle to the actual IF request. Its function would be simple: If the prefetched and requested instruction addresses match, then forward the instruction data to the CPU, else stall the CPU and meanwhile fetch the requested instruction.

The IP block can achieve this by mimicking the address and response channel signals coming from the NVM so that the CPU sees the RISE IP as a part of the NVM. We can then stall the CPU by simply dropping the grant and rvalid signals for a given number of cycles. The interface-level view of this mechanism is shown in **Figure 3.4**

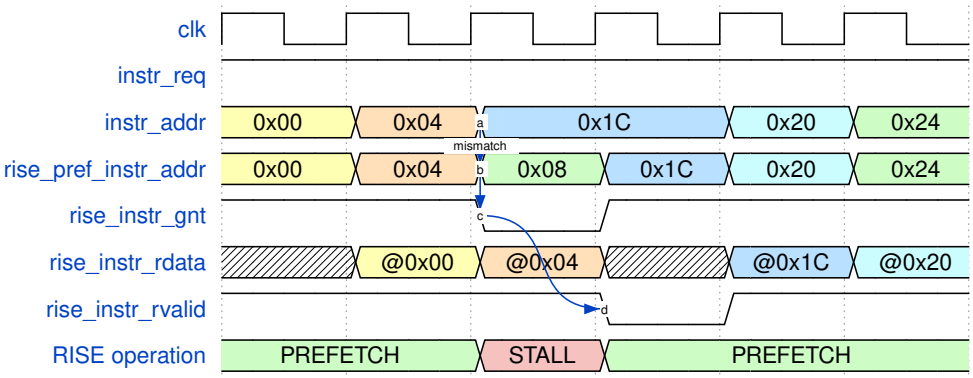

Figure 3.4: RISE IP operation

By introducing RISE into our system, we will eliminate instruction address logic and signal propagation delay (if we place the RISE sufficiently close to the NVM) from the critical path as illustrated in **Figure 3.5**. The former is determined solely by the CPU microarchitecture and therefore its combinational delay is fixed. The latter is dependent on the place-and-route process result and can vary from negligible to dominant. We have to take into account that we inadvertently introduce some additional logic into the critical path. This logic sits between the RISE's internal registers and its address port, similar to the CPU instruction address logic. Therefore, we have to design the RISE IP so that this nuisance delay is minimal.

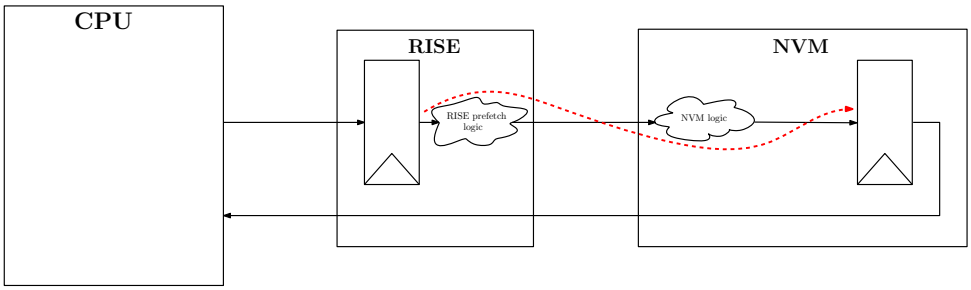

Figure 3.5: Critical path with RISE modification

Such a modification would increase the Instructions Per Second (IPS) by allowing us to raise maximum clock frequency *and not* by decreasing the number of clock cycles spent evaluating some task. If our IP could predict perfectly every IF address, we would need precisely the same number of clock cycles for a given workload. RISE therefore does not interfere with the instruction execution flow, serving instead as a sort of one-instruction lookahead prefetcher.

3.3.2 Performance analysis

Next, we wish to deduce how efficient our prefetcher IP must be to achieve a system speed-up. We view the Instruction Address Stream (IAS) purely as a series of numbers; e.g., we do not consider the underlying instructions here. Let us also define a continuous address stream as a sequence of addresses that are successively incremented by four, e.g., $4, 8, 12, ...$; otherwise, we say that the stream is discontinuous. We call the address immediately preceding the discontinuity the *source* and the following one the *destination*. Next, we define these parameters for a sufficiently long sequence of addresses:

- h — IP's hit rate: number of successfully predicted discontinuities divided by total number of discontinuities,
- k — penalty of a miss: how many clock cycles we need to fetch an instruction that we mispredicted. Shall be no less than two,
- n_b, n_s — number of discontinuous and continuous addresses respectively
- f_o, f_n — old and new system clock frequency respectively.

Then we say, that the time spent evaluating the task must be shorter after the introduction of RISE IP, i.e.:

$$\frac{1}{f_o}(n_b + n_s) > \frac{1}{f_n}(hn_b + (1-h)kn_b + n_s). \tag{3.1}$$

This is a simple inequality, linear in h, and after some simple manipulation we get following:

$$h > \frac{k - \frac{f_n}{f_o}}{k-1} + \frac{n_s}{n_b}\frac{1 - \frac{f_n}{f_o}}{k-1} \tag{3.2}$$

This gives us a lower bound for h if our goal is an execution speed up. Now, $\frac{n_s}{n_b}$ is dependent on the software that is running on the CPU, and we therefore cannot influence it. Examining **3.2**, it is clear that the higher the ratio $\frac{n_s}{n_b}$, the lower is our demand on the hit rate h — this is logical as the best discontinuity is the one that does not exist, as we cannot mispredict it.

It is also apparent, that as $h \longrightarrow 1$ the maximum system speed up approaches $\frac{f_n}{f_o}$. This places a fundamental limit on the system IPS gain. In **Figure 3.6** the different values of the lower bound are shown for h as a function of the ratio $\frac{f_n}{f_o}$ with $\frac{n_s}{n_b} = \frac{10}{3}$. A significant increase in lower bound h between $k = 2$ and $k > 2$ is apparent. This implies much higher requirements for predictor performance if $k > 2$ and we therefore wish to avoid this and keep $k = 2$.

The ratio $\frac{f_n}{f_o}$ is ultimately dependent on the place-and-route result. Still, we have to be careful as we could decrease this value by introducing too much of RISE prefetch logic into the critical path (as discussed in **Section 3.3.1**).

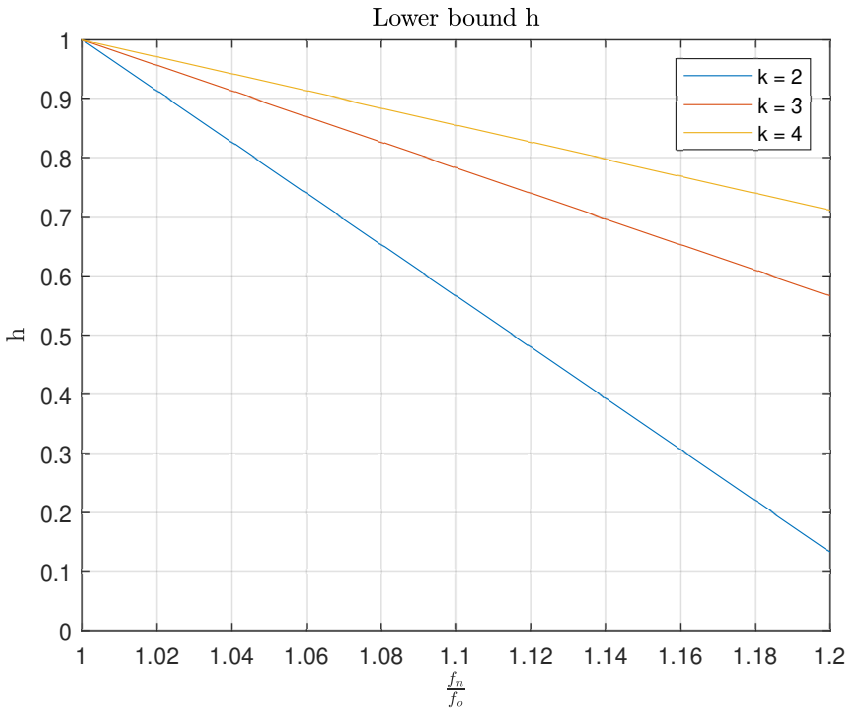

Figure 3.6: Different values for h, $\frac{n_s}{n_b} = \frac{10}{3}$

3.3.3 Instruction execution flow

In a typical CPU execution task, we can expect the majority of instructions to be spatially local, i.e., their addresses are sequentially incremented by a fixed constant (in our case by 4). These instructions do the actual computations — arithmetic and logic operations, storing and loading main memory data, etc. The rest are branching instructions, which command the CPU to jump to program counter relative (in the case of RISC-V Instruction Set Architecture (ISA)) instruction addresses, thus beginning a different execution sequence. These branching instructions can be further divided into unconditional, which always jump, and conditional, where the branch outcome depends on the CPU state (its internal registers).

We can expect a similar behavior at the IF interface — the instruction address stream is mostly continuous with a few discontinuities caused by branching instructions. However, we will not see a one-to-one mapping between the assembler-level and IF-level instruction addresses flow. We have to consider the underlying CPU instruction pipelining. This behavior is shown in **Figure 3.7**: first, a branching instruction is fetched (enters the IF stage of CPU pipeline). Some number of cycles later, the branching instruction outcome is resolved in the execution stage. If taken, the branch destination address is fetched, else we continue to fetch sequentially [3]. Thus, the corresponding discontinuity in the IAS will occur some number of cycles later, after the branching instruction passes through the RISE IP. The number of additional instructions being fetched between the fetching and resolving of

the branching instruction depends on the specific instruction. This delay is also dependent on the current CPU state, e.g., how full is the instructions prefetching buffer? Have we fetched a compressed instruction or not? Is the branching instruction execution dependent on some other instruction already in the pipeline? Etc.

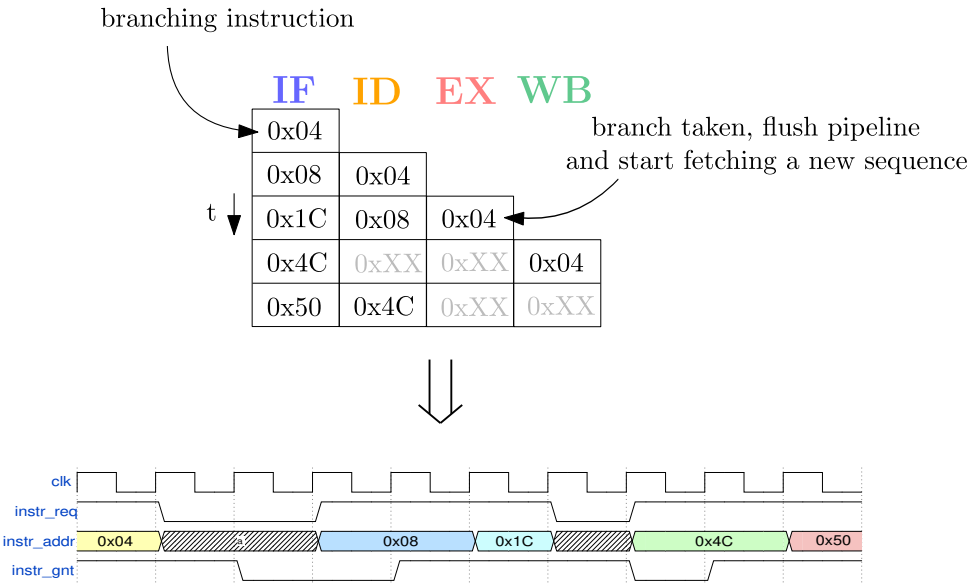

Figure 3.7: Effects of pipelining for a 4-stage pipelined CPU

In general, a single branching instruction causes two discontinuities in the IAS — when the instruction enters the IF stage, the CPU branch predictor decides to either continue fetching sequentially or it branches. Some number of cycles later, the branch is resolved, and depending on the outcome, another discontinuity can occur. All the possible scenarios are illustrated in **Figure 3.8**

Our CPU, fortunately, has a static branch prediction, in our case meaning that conditional branches are always *not taken*. The IAS will be therefore significantly less discontinuous (no discontinuities induced by CPU prediction) and we expect that this will make our efforts easier, as explained in **Section 3.3.2**. We will use IAS discontinuities and the underlying branching instructions somewhat interchangeably below in this text. It is important to remember that although in our case one branching instruction will usually induce one IAS discontinuity, the source address will not correspond to the branching instruction address.

3.3.4 Static Prefetching

The most straightforward prefetching mechanism is a static one — it always fetches the next sequential instruction. In a sufficiently long program with a diverse range of branch patterns, we can expect a hit rate of around 60% [4]. Even though this performance may not be sufficient for overall system speed up as derived in **Section 3.3.2**, we can use this scheme as a comparison to

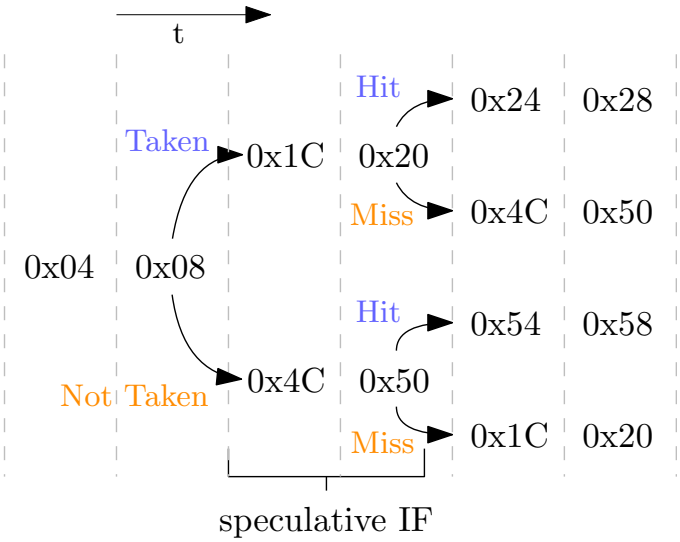

Figure 3.8: All possible branching

a more sophisticated one. We also expect this solution to have a low power consumption and a low die area usage. This design will also be very easy to verify thanks to its low complexity.

3.3.5 Dynamic Prefetching w/o ID

The prefetching mechanism we propose is as follows: (See **Figure 3.9**):

- A table indexed by the instruction address contains in each line a branch address and branch prediction data,
- at the start of the execution, all instruction addresses are assumed to be continuous and we simply fetch the next sequential address,
- if a discontinuity occurs somewhere during the task execution, we record the destination address in the table line indexed by the source address and initialize prediction data,
- if we then encounter the source address later on, we can now estimate based on the prediction data if we are to fetch the destination or sequential address,
- based on the branch outcome, we update the prediction data to hopefully increase the probability of a successful future prediction.

It can also happen that the destination address corresponding to a source address changes, i.e., we observe a discontinuity but with a different destination address than we have recorded in our table. This can happen when the clock cycle delay between the instruction being fetched and induced IAS discontinuity changes. Also, we may observe this effect when two compressed branching instructions share the same instruction address. We solve this

by reinitializing the table line with the new destination address and new prediction data.

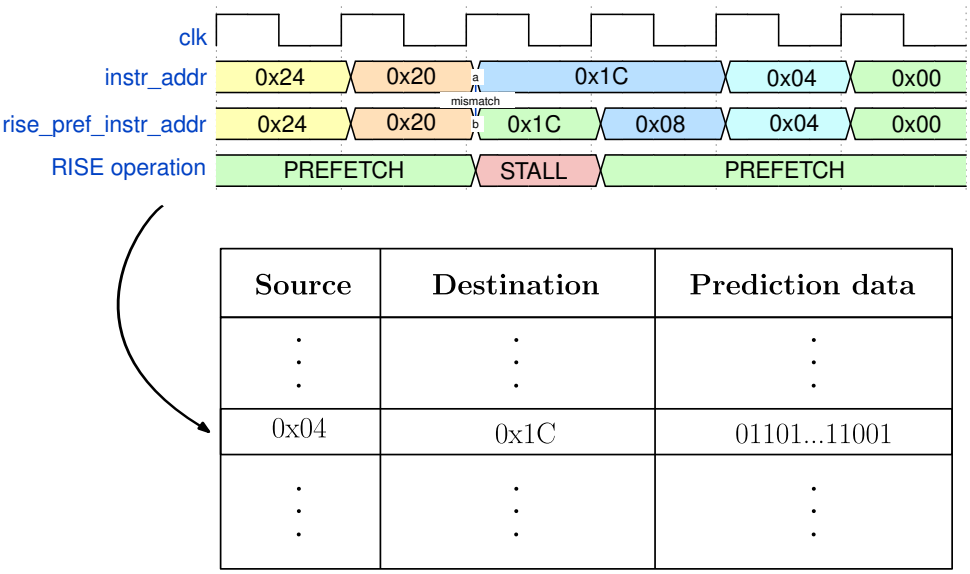

Figure 3.9: Table based prefetching mechanism

We will use the algorithms used in a classical dynamic predictor for the discontinuity prediction. Considering our low die area and low power consumption requirements, we will focus on simple one-level and two-level prediction schemes utilizing a saturating counter. This will be discussed below in **Section 4.3.2**.

3.3.6 Dynamic Prefetching Mechanism With ID

In this mechanism, we decode the prefetched instruction to check whether it is an unconditional or conditional branch. If we have an unconditional branch, we compute the target address as
$target\ address = branch\ address + 2 \cdot imm$, where the imm is the immediate field in the instruction data. Next, we store the target address in a buffer and after the IF-EX delay (discussed in **Section 3.3.3**), we present it to the CPU. In the case of a conditional branch, we do the same, but we additionally decide if the branch is to be taken or not. Branch prediction would be done in the same fashion as we would with the mechanism w\o instruction prefetching, i.e., a table-based dynamic approach. If the instruction is neither an unconditional nor a conditional branch, we fetch sequentially.

To figure out the IF-EX delay, we can either deduce it by looking at some internal signals of the CPU, or we take some sort of a statistical approach. For example, we could record this delay in our table, hoping that it does not change much for a given instruction.

3.3.7 Mechanism comparison

The static mechanism is undoubtedly the one that will be the most efficient in terms of power consumed per clock edge and die area used. However, its poor hit rate can lead to an overall system slow down. We will design this mechanism first to keep it as a benchmark, comparing it to other, more sophisticated methods.

In the dynamic approach w\o instruction decoding (ID) there is, as the name implies, no decoding of the underlying instructions and we treat it purely as a problem of predicting values with strong spatial locality. This way, we do not have to consider all the difficulties mentioned in **Section 3.3.3** — especially the problem of determining the delay between the branch instruction being fetched and resolved. However, this way, we lose the included information about the target address, which can be reconstructed from the immediate field of the branching instruction and its own address. Instead, we have to save the destination address in the table, which means a significant increase in the table resource usage. Another disadvantage of this approach is the possibly less efficient table usage — if the IF-EX delay varies for a given instruction in time, more than one discontinuity can be induced. For example, for a branching instruction with an address 0x04, in time, we can observe these sequences: 0x04, 0x08, 0x1C, 0x4C, ... or 0x04, 0x08, 0x1C, 0x20, 0x4C, ...

This will induce two entries in our table, one 0x1C - 0x4C and the other 0x20 - 0x4C, even though both are caused by the same instruction.

Without instruction decoding, we also can not tell which instructions are branching and which are not. This means we will mispredict all discontinuities induced by unconditional branches at least once. The induced discontinuities will also occupy space in our table even though we could have predicted them perfectly every time if we had decoded the underlying instructions.

We can also be sure that our design will be self-contained with this approach, dependent only on the CPU IF protocol (even ISA independent).

The approach using instruction decoding will be dependent on the CPU microarchitecture (number of stages of pipeline, internal signals, ISA dependent) but it is the best candidate for the most efficient prefetcher in terms of prediction hit rate. The corresponding table will also be much smaller and more efficiently used than in the case of no instruction decoding. Lastly, in terms of design complexity, we shall rank these approaches in the same order in which they were presented.

The advantages and disadvantages of the proposed mechanisms are summarized in **Table 3.1**

	Static	Dynamic w\o ID	Dynamic with ID
ISA independent	Yes	Yes	No
Die area	Low	Highest	Higher
Power per cycle	Low	Higher	Higher
Design complexity	Low	Higher	Highest
Hit rate	Low	Higher	Highest
Table size	X	Higher	Lower
Table use efficiency	X	Lower	Higher

Table 3.1: Comparison of different prefetcher mechanisms

Chapter 4

Implementation

Based on the discussion in the previous section, we have decided to implement the RISE IP prefetcher solution both with the static and dynamic mechanisms (no ID). The corresponding hardware will be described in the VHDL hardware description language. The pipelining solution will not be implemented, as its negative impact on the CPU IPS makes this approach non-viable for our intents and purposes.

When designing the dynamic prefetching mechanism, we will implement the table as a hardware cache with parameterized properties (size, degree of associativity). For different sets of parameter configurations, we will then evaluate the cache performance, choosing the most optimal setting with respect to die area, power consumed, and predictor hit rate.

As for the prediction mechanism itself, we will implement a one-level and a two-level 4-state saturating counter scheme.

If our design is to succeed in speeding up our system, we have to ensure that we do not waste any cycles while prefetching the CPU instructions. We have a maximum budget of one cycle per successful prefetch and ideally no more than two cycles for an unsuccessful one. We thus have to make sure that our design will not go through any unnecessary states. Furthermore, when considering the table-based dynamic approach, we will not be able to simultaneously update and read from our table and therefore we will have to choose which of these operations to prioritize.

4.1 System Level Design

The RISE IP prefetcher has to support these features:

- Compliant to the CPU IF interface (OBI protocol),
- speculative prefetching of the CPU instructions,
- seamless integration into an existing RISC-V based SoC.

4.2 Block Level Design

The basic block level design common to all mechanisms is shown in **Figure 4.1**:

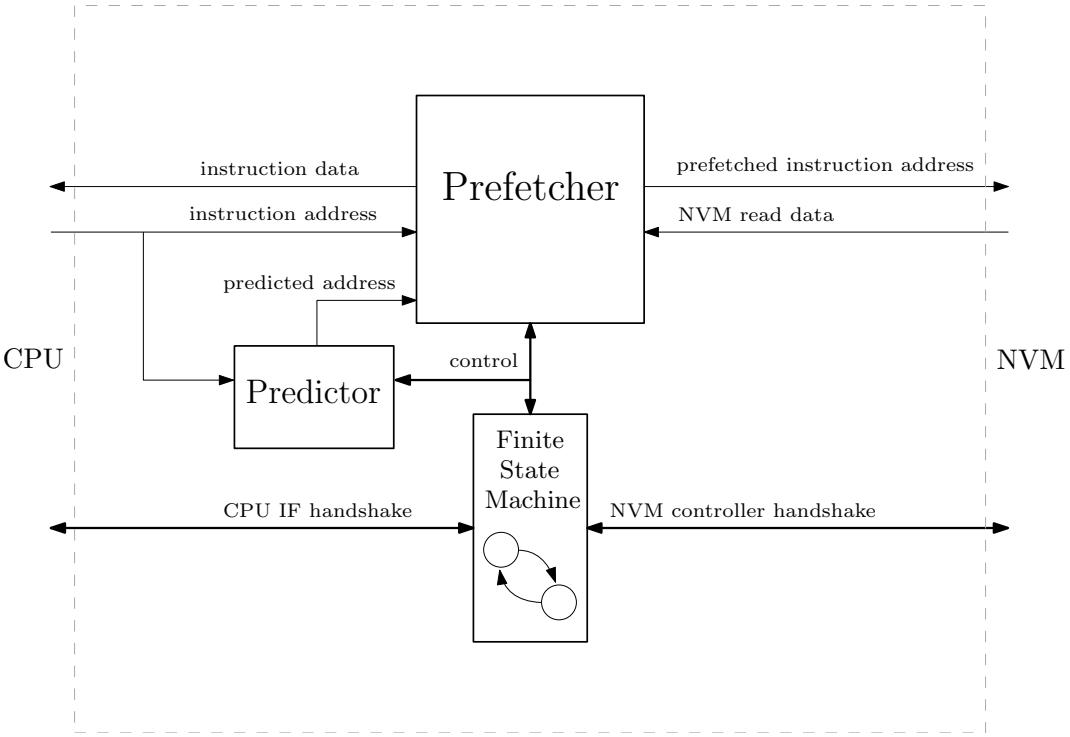

Figure 4.1: Block diagram of RISE IP

The Prefetcher block is responsible for providing the NVM with a prefetched instruction address based on the CPU requested instruction address. It also signals to the FSM if the prefetched and requested addresses match. The Predictor block provides the Prefetcher with the predicted address. For the static mechanism, this block is void, as we always fetch sequentially. The Finite State Machine (FSM) provides control signals to all other RISE IP blocks and is also responsible for OBI-compliant communication with CPU and NVM. All RISE IP blocks share the same clock with the CPU and the NVM controller.

4.3 RTL design

4.3.1 Static mechanism

The RTL structure of the Prefetcher block, together with the NVM abstraction, is shown in the **Figure 4.2**. We essentially choose between three options for the prefetched address:

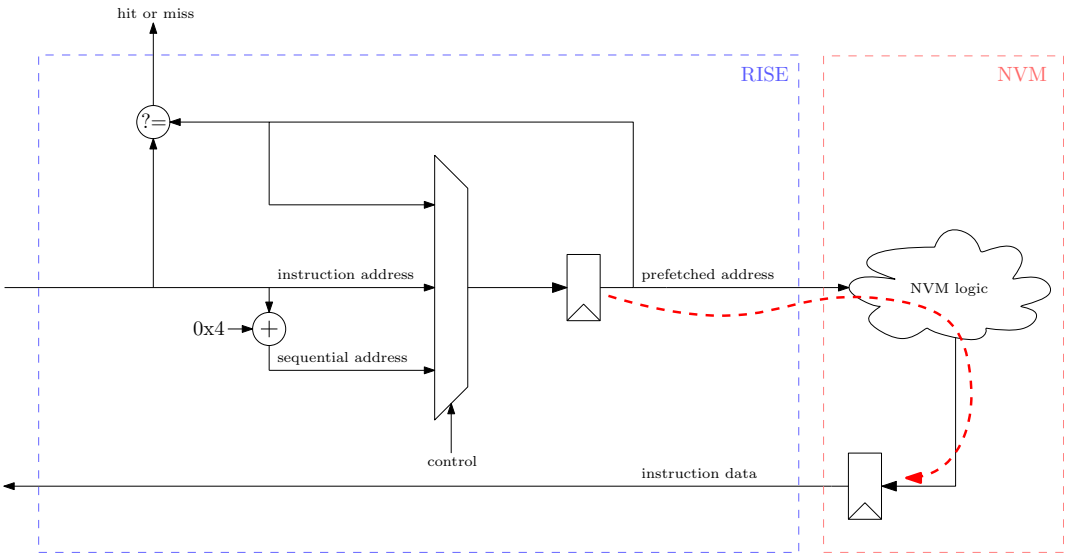

Figure 4.2: RTL design of the Prefetcher block with a highlighted CP

1. The current CPU requested instruction address — we have mispredicted and must recover, or the RISE has been just initialized,
2. the sequential address — RISE is prefetching,
3. the last prefetched address — RISE is waiting for the request signal from the CPU or grant signal from the NVM.

In the case of a miss, we have to drop the grant signal to the CPU in the *same cycle* to signal to the CPU that its request has not been granted. We have chosen to implement our FSM as a Mealy-type state machine to achieve this. Its state transition diagram is described in **Figure 4.3**.

In **Figure 4.4** and **Figure 4.5** we have shown typical scenarios of RISE operation.

As is apparent from the **Figure 4.2** a feature of this design is a minimal combinational nuisance delay (as discussed in **Section 3.3.1**) because the prefetched instruction address is registered.

4.3.2 Dynamic mechanism w\o ID

The prefetcher block of the dynamic mechanism with no ID will be the same as in the static approach, except that the multiplexer now has an additional input — the predicted address. We can thus be sure that the nuisance delay will be minimal, as the prefetched address is registered. However, we wish to implement most of the predictor table memory as a synchronous-read, synchronous-write RAM, and therefore we can not provide the predicted address stored in the RAM in the same cycle in which the CPU requested instruction address (which serves as RAM address) arrives in the RISE IP. To solve this, we use the immediately sequential address to read the table.

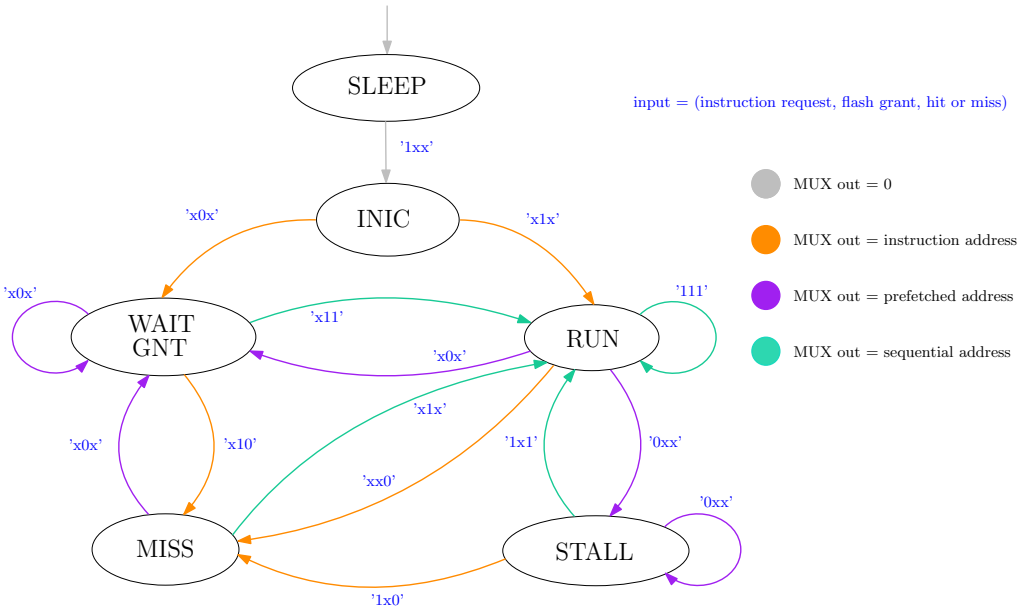

Figure 4.3: FSM state transition

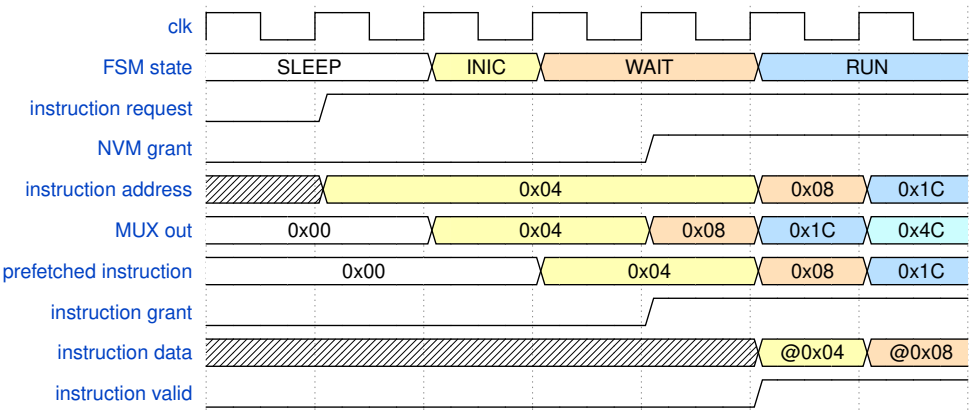

Figure 4.4: RISE initialization

Therefore, for a given instruction, we will have the table data ready almost immediately at the time the address arrives in RISE because the table access control signals have been figured out in the previous cycle. This only works provided that the current requested address is the immediately sequential in relation to the previous one. If it is not, then a discontinuity occurred and we will not have the correct data to make a prediction in this cycle. This should not be detrimental to the prefetcher performance as we expect these immediately following discontinuities to be rare.

An alternative approach is to insert a second multiplexing stage in the prefetcher block, which would choose between the first stage and the predicted address. This would, however, mean some added nuisance delay in the CP. The RTL design of such a block is shown in **Figure 4.6**. The predictor table provides the predicted destination address based on the sequential

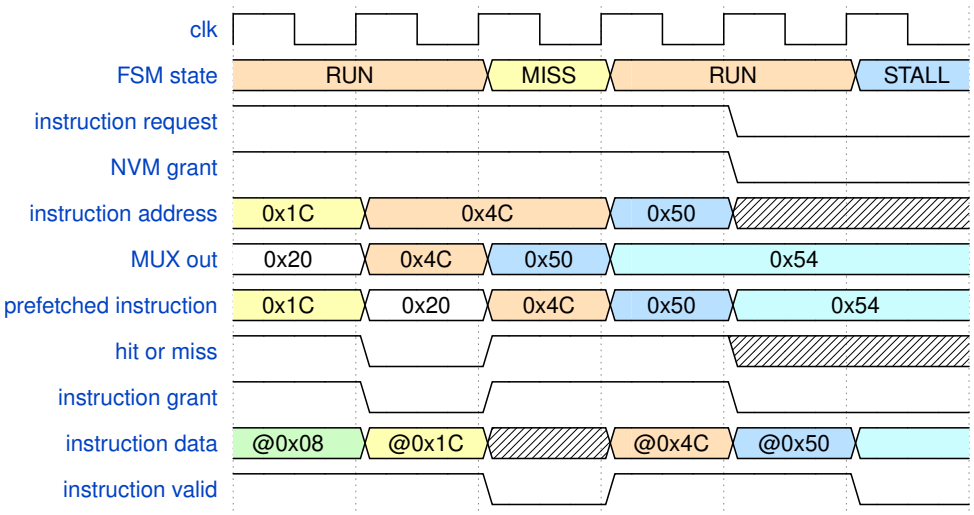

Figure 4.5: RISE miss and stall function

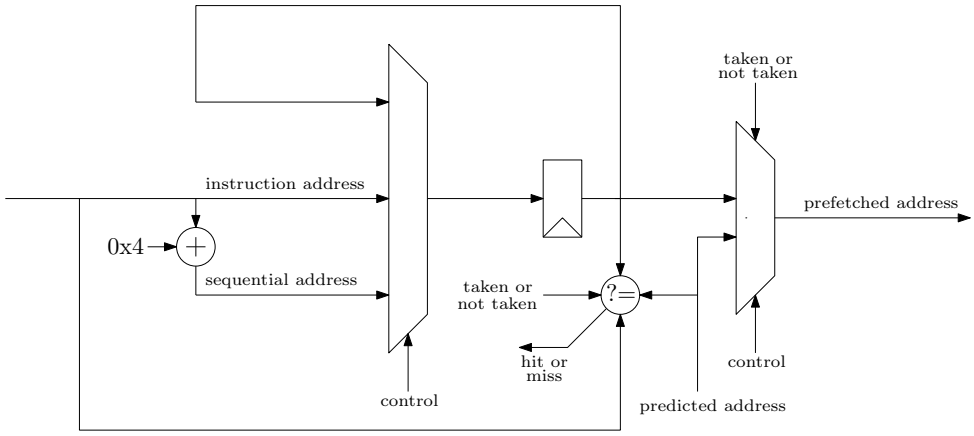

Figure 4.6: RTL design of the alternative Prefetcher block

address, while the predictor FSM decides if the branch is to be taken or not. Then, when the branch is resolved, the predictor FSM computes the updated prediction data as a function of the current data and branch being hit or not.

For the FSM, we use the same Mealy-type state machine as in the static approach, the state transitions being the same. We only have to modify the control signals so that we can distinguish if the table line's data are to be initialized, updated, or reinitialized in the case of a miss.

The block diagram view of the predictor block is shown in **Figure 4.7**.

Cache theory

When designing the table for our predictor, we have to deal with the problem of storage size — we cannot map every single instruction to a unique table line as that would require an immense memory size. We can take advantage of the fact that the instructions exhibit a temporal and spatial locality. Temporal

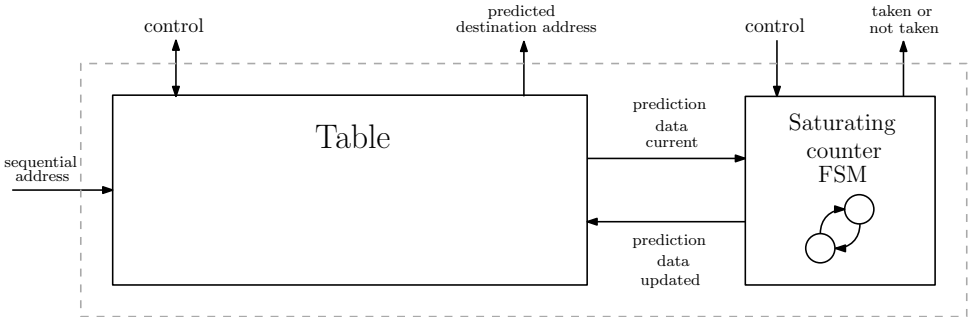

Figure 4.7: Block diagram of the predictor block

locality means locality in time — if a branch instruction is fetched, it is highly likely to be fetched in the near future again. Consider this pseudocode snippet:

```
for i in 1 to N do
    for j in 1 to M do
        <some calculation>
    end for
end for
```

The compiler will map these two for loops to two branching instructions, each fetched repeatedly in a short span of time when the CPU starts executing this task. Spatial locality means that these two instructions will be close in the instruction address space, i.e., the difference between their addresses will be small relative to the address space scope.

Therefore, we will implement our table as a hardware cache, a small amount of memory that temporarily holds data with a high probability of being referenced in the near future. The goal of this design is to:

- Maximize the probability of the cache storing the data being referenced next,
- minimize the size of the cache.

A typical CPU cache that has one-cycle associative search is shown in **Figure 4.8**. The atomic unit of data is called a cache line. The address space is partitioned into S number of sets (S is also alternatively called the degree of cache associativity), each containing N number of lines, which is the smallest unit of data in the cache. The set size is $N = 2^m$ so that each set can be implemented as a RAM addressed by the set index. The number of sets is arbitrary. To identify the correct data in the cache, we append to every line a tag index so that the mapping $address \mapsto \{tag\ index, set\ index\}$ is unique (address bits are partitioned to tag and set index), which is then compared to the tag index. A typical read operation is as follows:

1. Read S cache lines, addressing with the set index,
2. compare the tags of the read lines with the tag index,
3. if a hit, then forward the correct line to the CPU, if miss allocate a new line with the correct data from the main memory, then forward it to CPU.

The cache starts out empty and is gradually filled. As the address space is limited, later in the future, we have to decide which lines we keep and which we overwrite. The algorithm in charge of this is called a replacement policy and several options are available such as First In, First Out (FIFO), Least Recently Used (LRU), or random replacement [6].

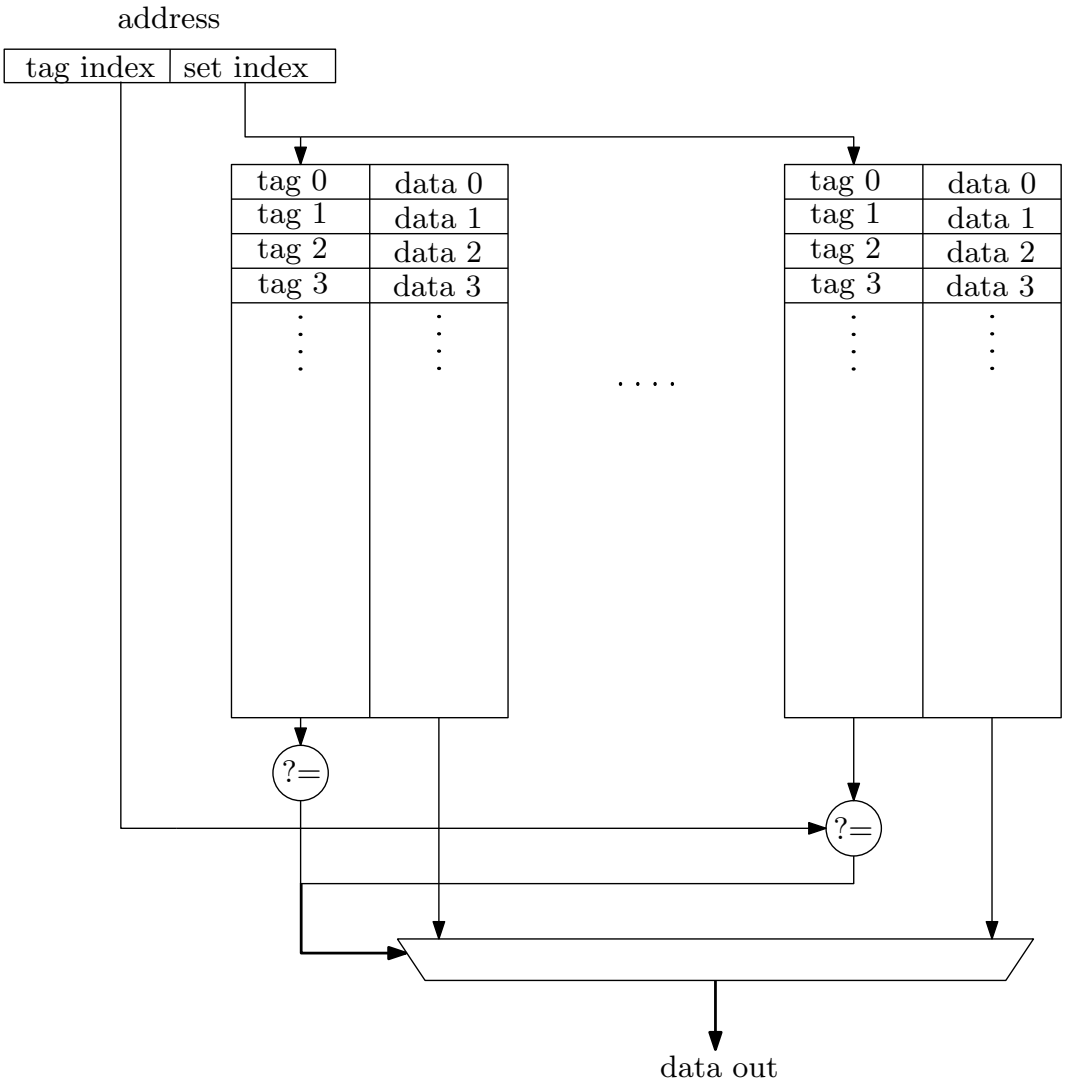

Figure 4.8: CPU cache structure

As [6] notes, for a given memory budget of $B = S \cdot N$ lines, as N increases, the probability of finding the correct line decreases, as does the cache's

complexity. The first can be seen clearly from the fact that the $address \mapsto set\ index$ is a many-to-one mapping. The latter is also evident: we need fewer comparators and less multiplexing logic.

The nature of the cache in relation to N and S can be best understood by looking at its limits: if we set $S = 1$ we have a direct-mapped cache, the least complex, the smallest hit probability. For $N = 1$ we get a fully associative cache, the most complex, the best hit probability. In between is a set-associative cache, providing a compromise between the two corner cases.

Cache implementation

When implementing the cache design, we will have to tweak it to suit our needs. First of all, our cache has to have a maximum of one clock cycle latency when doing a read operation (write operation can be longer, but we prefer it not to). A parallel tag comparison is therefore needed, implying a need for S comparators. We could also solve this by pipelining our cache, partitioning its operation into stages. We will not implement our table this way as it would make the design needlessly complex and it would result in a higher k value as discussed in **Section 3.3.2**.

Second, the output of our cache has to be directly the output of a synchronous logic in order to minimize the nuisance delay as described earlier in this section.

Third, the cache design shall comply with the low die area and low power paradigm. One compromise we can do is to make the $address \mapsto tag\ index$ also a many-to-one mapping, therefore making the tags smaller and saving some space.

This is possible because we do not mind if we occasionally fetch the incorrect cache line, as it will simply lead to a lower chance (on average) of a prediction hit. Whereas in a CPU cache, it is imperative that we fetch the correct line, else we feed the CPU the incorrect data or address value, possibly leading to a critical failure.

However, we have to be careful, as lowering the tag bit length could lead to a higher dynamic memory power consumption. Consider this: if we have a very small tag bit length and all the cache lines have been initialized, then on every fetch, there is a high probability of a cache hit, resulting in the table memory being read. We therefore have to optimize between die area (tag bits) and power consumption (table being read).

Fourth, we need to choose a replacement policy with an implementation that has a good balance between power and die area used versus efficiency. The most efficient replacement policy would be a one that replaces the line that will not be accessed for the longest time relative to all other lines, a so-called clairvoyant replacement algorithm [7].

Now we will discuss the shortcomings of our cache memory. In a traditional CPU cache, an important metric of cache performance is its hit rate — how often do we find the data we need in the cache relative to all cache fetches. A miss results in the cache having to fetch the data from the main memory,

possibly stalling the CPU execution. Our cache, however, will not store the full tag value of the address because of memory size considerations. We will thus observe the phenomenon of colliding addresses. This happens when we read a cache line onto which a different address was mapped in the past — thus, two or more instructions share the same prediction data, leading to a lower prediction hit probability.

Initially, when the CPU starts executing, the cache will perform poorly simply by not having any data available yet. Then, when the cache is filled, so long as the CPU task is executed roughly sequentially (discontinuities in IAS, but small in comparison to address space scope) we will see a slow but steady rise in the number of collisions as the addresses wrap around the tag and set index scope. If the task execution makes a big leap in the address space, we will see a sudden spike in the number of collisions before the prediction data is recomputed. Therefore the cache will yield the worst results if the task execution jumps often in the address space, not giving the prediction data the time to settle.

In **Figure 4.9** we can see our RTL implementation of the direct mapped cache ($S = 1$, $N = 2^m$, where m is the bit length of set index)

To satisfy the read latency of one cycle, we first read an asynchronous-read memory which tells us if the corresponding cache line has been activated and also the tag is compared. Then, based on the comparison result, FSM control signals, and cache line being set or not, we provide the corresponding enable signal level to a RAM which will read the predicted destination address and the prediction bits on the next rising clock edge. The asynchronous-read memory also has to be reset at the start of the RISE IP operation; we will therefore construct it from D-type flip-flops. Such storage is expensive in terms of transistors per bit [8], but since this memory is very small, we can afford this.

When we write to the cache when we want to either

- Initialize a new line,
- reinitialize a new line,
- update prediction data.

All of these operations occur in the next cycle after a prefetch miss. Therefore, we have to use a registered (delayed) instruction address to address the cache.

We now discuss some power and die area optimizations used in this design. First of all, we can expect that the range of most jumps is fairly small compared to the address space scope and thus the source and destination address will be relatively close to each other in the address space. We can then save some of our memory budget by not saving the full destination address but only a given number of lower order bits. When we read the clipped destination address based on some source address, we attempt to reconstruct the original destination address by concatenating the stored value with the complementary higher order bits of the source address. This will inevitably

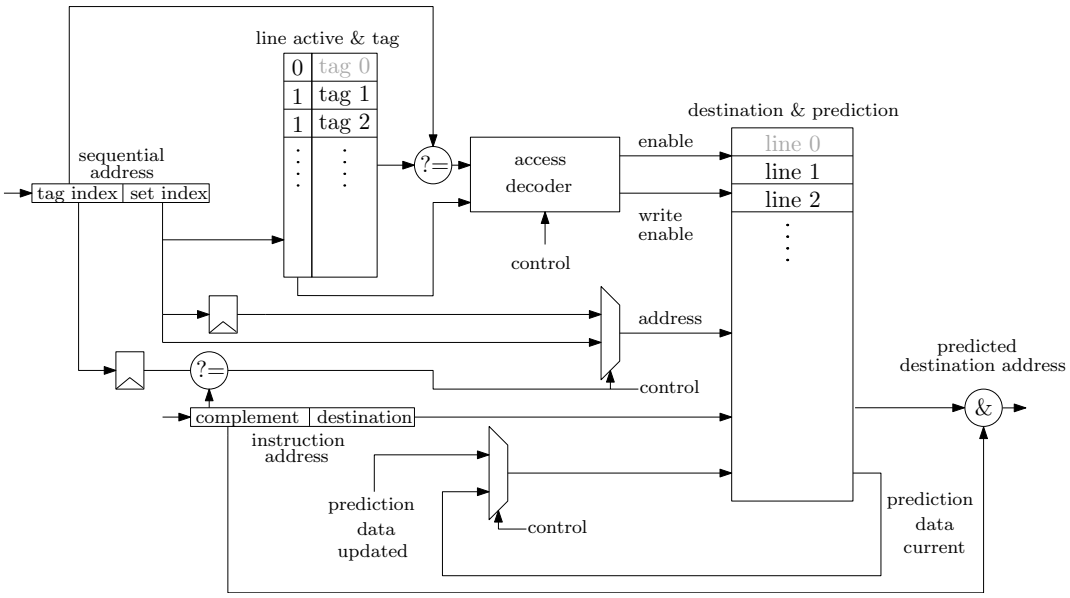

Figure 4.9: Table RTL structure, direct mapped cache

lead to some discontinuities being unpredictable — their destination and source address are too far apart in the address space. The relative number of such discontinuities and their impact on the predictor performance is highly dependent on the workload and we will try to optimize the clipped destination length based on simulation results. To prevent these unpredictable discontinuities from polluting the cache, we always compare the source and destination the bits complementary to the clipped least significant ones before an initialization, to prevent them from being written into the cache.

We can also save some power by partitioning the RAM into two — one containing the destination address and the other the prediction data. We expect that the intent of updating a cache line will mostly be to update the prediction data. Since an SRAM dynamic power consumption is proportional to its size, we would waste energy by overwriting the SRAM line by mostly the same bits (destination address bits).

Since we are using a saturated counter branch prediction (described in 4.3.2), we expect that for some cache lines, the updated prediction data will often match the current data. These updates are redundant — they consume extra power and provide no additional information. Since we prioritize updates over reads, they also degrade the predictor performance by needlessly occupying the cache. We get rid of them by comparing the updated and current prediction data; if they match, we do not update the cache.

The line active table is itself a sort of a power optimization because it lowers cold start power consumption, limiting the cache accesses only to addresses that have been already set. For a sufficiently long task however, this effect should vanish.

In **Figure 4.10** we see the implementation of the full associative cache. We use the FIFO replacement policy implemented by a mod S counter. This

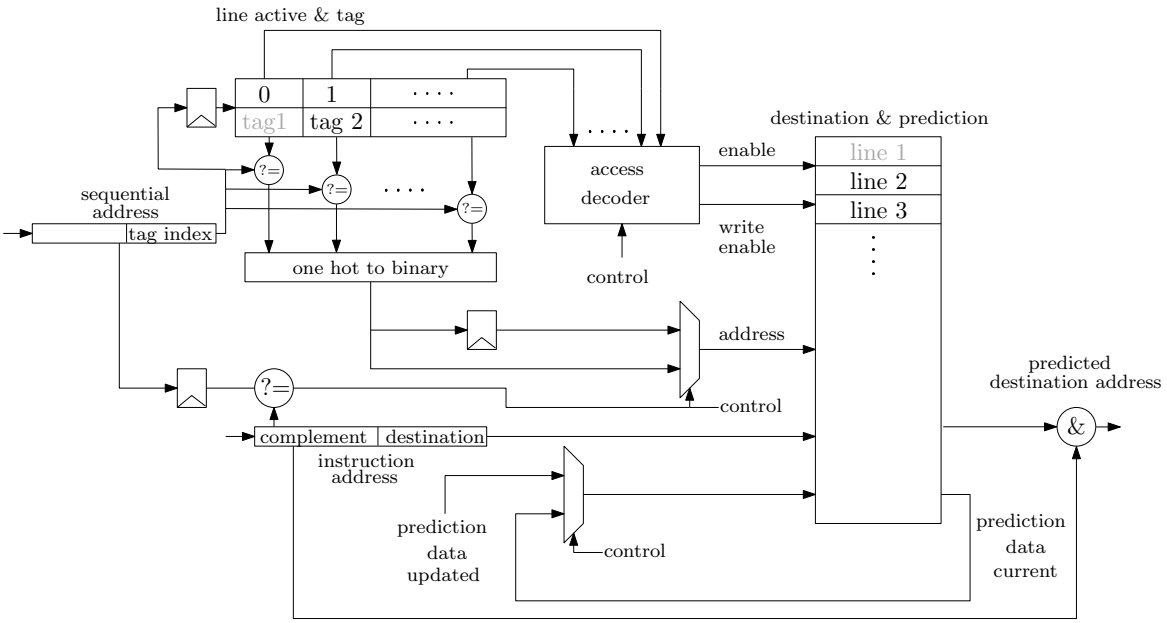

Figure 4.10: Table RTL structure, full associative cache

means that we always replace the line that has been added the least recently (not to confuse with Least Recently Used replacement policy). We choose this policy because it is easy to implement, costing us only $\log_2 S$ bits. We expect it to perform well (performance being defined in **Section 3.3.2**) if the task execution is sufficiently sequential.

In **Figure 3.10** there is the n-way associative cache. For the replacement policy, we choose a random replacement policy.

From here, a pattern emerges for a general cache — first, we read out S tag values addressed by the set index, then make S comparisons with the fetched tag index. Then convert the S bit, one hot encoded comparison result to binary, which yields the hit set number. Then concatenate this with the set index to obtain the line address.

Prediction mechanism

Before designing the prediction mechanism itself, we have to consider the typical behavior of the underlying branching instructions. We categorize the branching patterns into three categories:

- Loop branches,
- repeating pattern branches,
- pseudo-random branches.

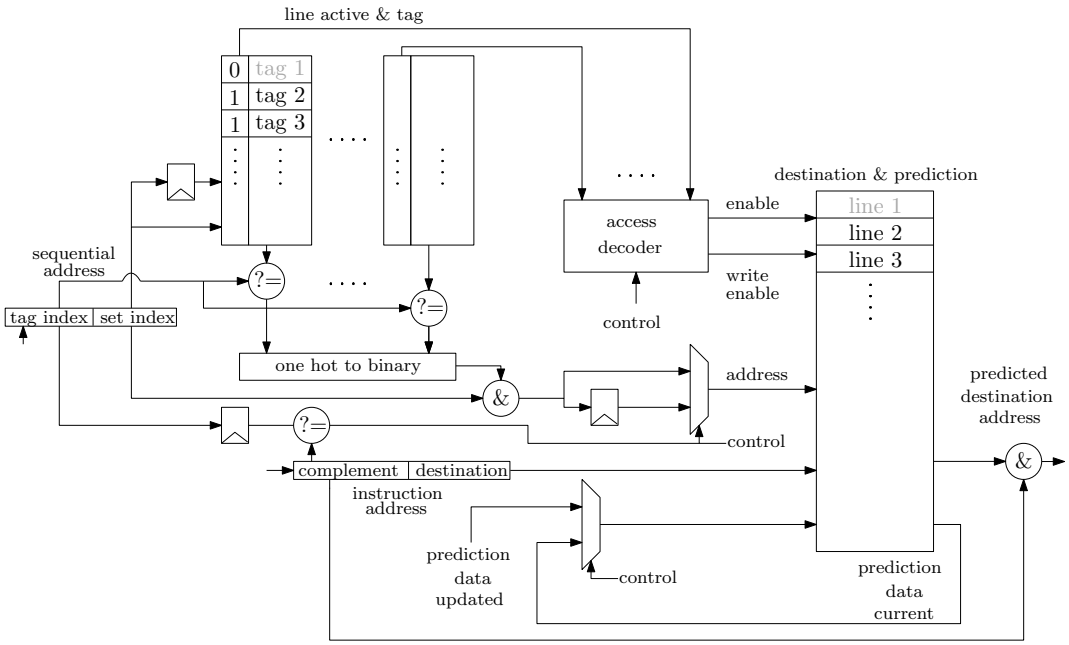

Figure 4.11: Table RTL structure, n-way associative

We have already encountered an example of a loop branch:

```
for i in 1 to N do
    for j in 1 to M do
        <some calculation>
    end for
end for
```

The branch pattern of the nested instruction is (T — Taken, NT — Not Taken) $\{T, T, T, ..., NT, T, T, ...\}$. Looping behavior is nicely predictable and also very common; we should therefore pay attention to the way our proposed branch prediction mechanism handles them.

The repeating pattern branches, as the name implies, exhibit a repeating pattern, that is *sufficiently short* in length. A loop branch is therefore a special case of a repeating pattern. Here, every third branch is not taken: $\{T, T, NT, T, T, NT, T, T, ...\}$ Such a behavior arises for example when executing short loops or when branching instructions are somehow coupled:

```
for i in 1 to N do                                  ▷ N » 4
    if i mod 4 == 0 then
        <some calculation>
    end if
    <some calculation>
end for
```

Predicting these branches will be much more difficult, as the prediction scheme has to somehow adapt to the branching pattern. We also expect the prediction to be more and more difficult as the pattern length increases.

The pseudo-random branches either exhibit a pattern that is impractically long or the branch outcome is determined by some random or pseudo-random data (such as user input). When evaluating these branches, the performance of our predictor will be highly dependent on the ratio of branch outcomes and also on the conditional probability $P(T|NT)$ and $P(NT|T)$, e.g., we expect different performance for branch histories $\{T, T, T, T, NT, NT, NT, NT\}$ and $\{T, NT, NT, T, T, NT, T, NT\}$ — same ratio of branch outcomes, different conditional probabilities.

An elementary prediction mechanism is the so-called saturating counter. It is a finite state machine that based on the current state determines if the branch is to be taken or not. The next state is then determined when the branch is resolved. We assume that when a branch is resolved, there is a high probability that the outcome will repeat when the branch is evaluated again in the future. The simplest two-state predictor is in **Figure 4.12**. Its prediction saturates after only one Taken or Not Taken evaluated branch outcome.

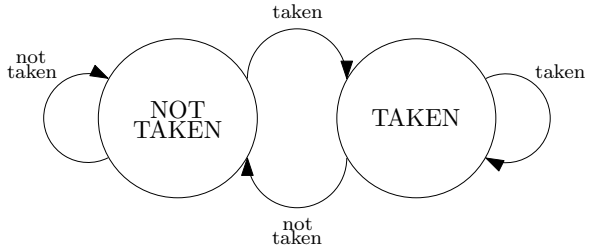

Figure 4.12: Two-state saturating counter

A four-state predictor is shown in **Figure 4.13**. It assumes one of 4 states — Strongly Not Taken (SNT), Weakly Not Taken (WNT), Weakly Taken (WT) and Strongly Taken (ST), the default state is the WT. Other state transition schemes are possible, but [9] showed empirically that the one presented performs the best for a wide range of branching patterns.

Here, if the predictor is saturated, we must deviate twice to switch the prediction outcome. This is beneficial when evaluating a nested loop branch — with the four-state counter, we make only one misprediction (on the closing loop) rather than two (on the closing loop and then on the opening one) with the two-state counter.

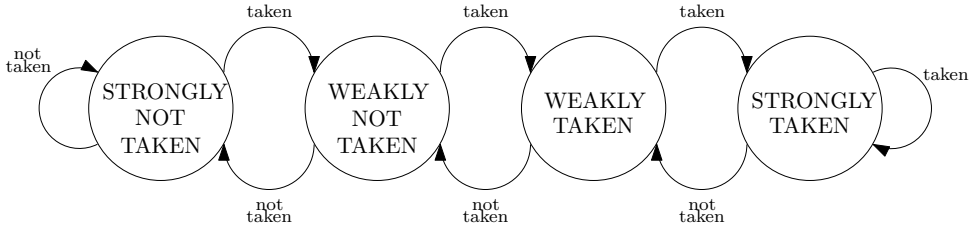

Figure 4.13: Four-state saturating counter

In general, when it comes to sensitivity to changes in branch patterns, the four-state counter is a good compromise between two-state predictor (too sensitive) and more-state predictors (not sensitive enough). The saturating counter predictor performance significantly degrades when presented with branch patterns with high probabilities $P(T|NT)$ and $P(NT|T)$. For example a pattern $\{T, NT, T, NT, T, NT, ...\}$ will cause a saturated/unsaturated four-state predictor to have a 50% and 0% hit rate respectively. This type of predictor performs well when evaluating a task containing long loops, or more generally branches, where one outcome is statistically much more likely. Advantages of this scheme are easy implementation, relatively low die area, and power costs.

To improve on the idea of saturating counters, we can make our predictor take into account the branch history that we have observed so far. This scheme is then called two-level prediction — the first level is some kind of memory that utilizes the branch histories of the currently executed task. The second level is a table of predictors that we index by combining the cache address and the branch histories. The mechanism that we discussed previously is then called a one-level prediction. The motivation is to be able to better predict short repetitive patterns that the previously described 4-state saturating counter struggles with.

In its simplest form, this history can be either global, meaning that as the task executes, we keep a shared record of previous branch outcomes, or it is local and we therefore have the same record, but for every branch individually.

Now we will discuss the case of the local branch history mechanism.

Paper [9] defines the local history as a branch HistoRy Table (HRT) — a memory that contains the branch histories of every cache line. One such history keeps track of the last M branch outcomes for that particular cache line. That means that the history can assume one of 2^M patterns. This history is then used as an index to select one of 2^M 4-state saturating counters stored in a global branch history Pattern Table (PT), see **Figure 4.14** . This scheme allocates different branch predictors to different branch patterns, thus allowing us to predict (after the predictor values settle) any repeating pattern with a period less than $M + 1$. If the pattern period length is greater than $M + 1$ but less than 2^M, then we can also predict it correctly, provided that all the sub-sequences are unique [10].

In **Figure 4.14** we illustrate the fact that the $history_i \mapsto predictor_k$ is a many-to-one mapping — multiple matching histories (cache lines) share one predictor. These collisions are similar to cache collisions discussed in **Section 4.3.2** and can be detrimental to the performance of our predictor.

Let us now explore this phenomenon. We expect that the longer the history length M, the fewer collisions we will observe (with the number of lines B kept constant). However, we cannot hope to avoid them altogether — if we approximate that the probability distribution of history values for any given cache line is uniform (histories are independent), then the probability that not all histories in a given cycle are unique is

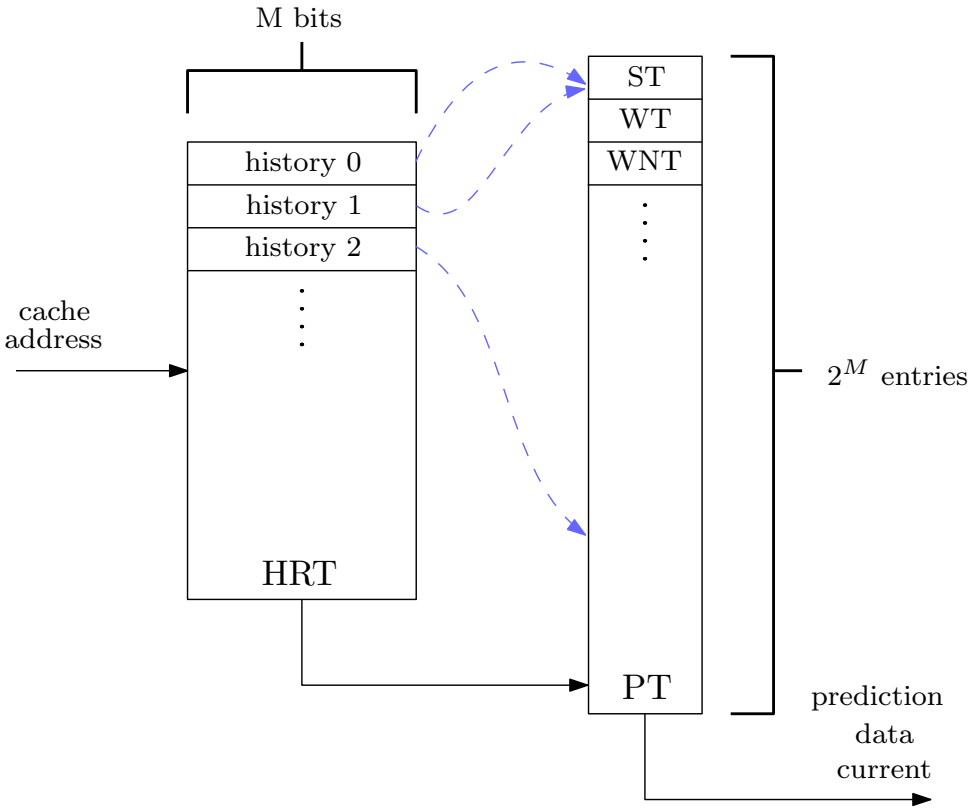

Figure 4.14: Local two-level prediction scheme

$$P(not\ all\ unique) = 1 - P(all\ unique) = 1 - \frac{2^M!}{(2^M - B)!}\frac{1}{(2^M)^B}, 2^M > B \tag{4.1}$$

Thus for any cache of a reasonable size for our intents and purposes $(64 < B < 1024, 2 < M < 10)$ the probability $P(not\ all\ unique) \longrightarrow 1$ as the term $\frac{1}{(2^M)^B}$ (all possible outcomes) dominates the $\frac{2^M!}{(2^M-B)!}$ (all favourable outcomes).

We can gain further insight into how $MandB$ influence the number of collisions. First, let us define a collision as an event when a PT entry is updated that is shared between two histories in HRT (we can have multiple collisions per one PT entry update). The expected value of collisions when a random cache line is updated is equal to

$$E[X_{count}] = \sum_{k=0}^{B} x_k P(x_k) = 0 \cdot (\frac{2^M - 1}{2^M})^B + 1 \cdot \binom{B}{1} \cdot (\frac{2^M - 1}{2^M})^{B-1}(\frac{1}{2^M}) + \dots \tag{4.2}$$

This corresponds to the mean of the binomial distribution (provided that $E[X_{count}] >> 1$) where B is the number of trials and $\frac{1}{2^M}$ is the probability

of success. Therefore $E[X_{count}] = \frac{N}{2^M}$. This model thus makes a simple prediction — for a twice as large PT, we get half as many collisions. Although this is only an approximation (the HRT values are generally dependent and their distribution is not uniform), it hints at something — for small values of M we expect a large number of collisions that will degrade the performance significantly and as we make M larger and larger, the benefit (with respect to $E[X_{count}]$) of plateaus.

We will now discuss the case of using the global branch history. The HRT now reduces to a single history shared between all the branches (we should probably call it HR now, but we keep the name for consistency reasons). The PT, however, is now local — it is indexed by concatenating the cache address and the HRT, see **Figure 4.15**.

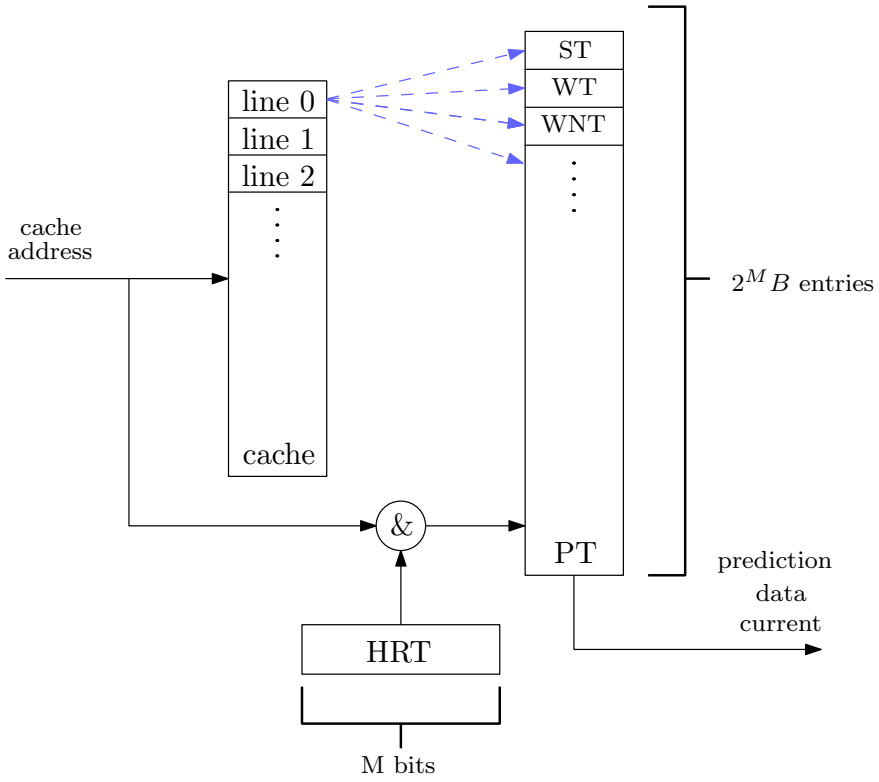

Figure 4.15: Global two-level prediction scheme

We thus allocate to every cache line 2^M 4-state predictors. We therefore do not experience the problem of collisions as we did in the local scheme. We can also now successfully predict coupled patterns because their dependency is now captured in the HRT. This works, provided that the branch history length is long enough and the underlying branches are temporally local. However, the branch history can therefore be also polluted with branching patterns that are easy to predict anyways.

Consider this snippet:

```
for i in 1 to N do                                    ▷ N » 4
   if i mod 4 == 0 then
      <some calculation>
   end if
   <some calculation>
end for
```

The for loop pattern is $\{T, T, T, T, T, ...\}$, the if statement pattern is $\{NT, NT, NT, T, NT, NT, ...\}$. The HRT will then record this sequence $\{T, NT, T, NT, T, NT, T, T, T, NT...\}$ and we would therefore need $M = 7$ to perfectly predict the *if* statement branch, while the local prediction scheme would need only $M = 3$, provided that no collisions occur.

For the RISE IP we have decided to implement the one-level scheme and the two-level global history scheme. For the one-level scheme we do not show the implementation, as it is only a simple Moore FSM. The two-level scheme RTL implementation of the PT and HRT is shown in **Figure 4.16**.

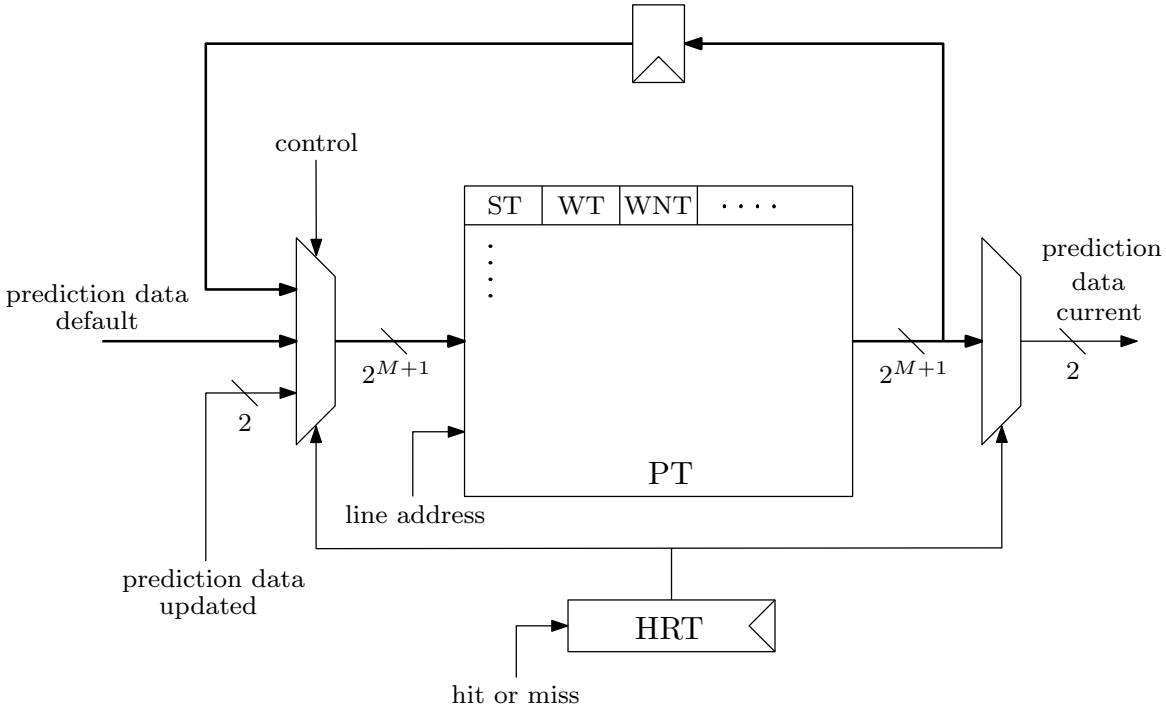

Figure 4.16: Global two-level prediction RTL

Instead of concatenating the line address and the HRT, we keep all the predictors corresponding to a single cache line as a single RAM word and then use HRT to multiplex the desired predictor for w/r operations. We do this so that we can initialize all of the predictors in a single cycle. The HRT is implemented as a simple shift register. Otherwise, the RISE cache and predictor FSM look the same.

The two-level predictors described were local HRT, global PT (lHRT-gPT) and global HRT, local PT (gHRT-lPT). There also exist other schemes — a local HRT and a local PT (lHRT-lPR) and global HRT and a global PT (gHRT-gPT) [11]. In high-performance computing, these basic schemes are combined to create even better-performing predictors, a well-known example are the *gshare* and *gselect* predictors [12]. Because these more advanced schemes have high die area usage and high power consumption, we will not discuss them in this text further.

4.4 System Integration

Finally, when the standalone IP is designed then comes the task of integrating it in the system. In **Figure 4.17** we show the the preliminary solution.

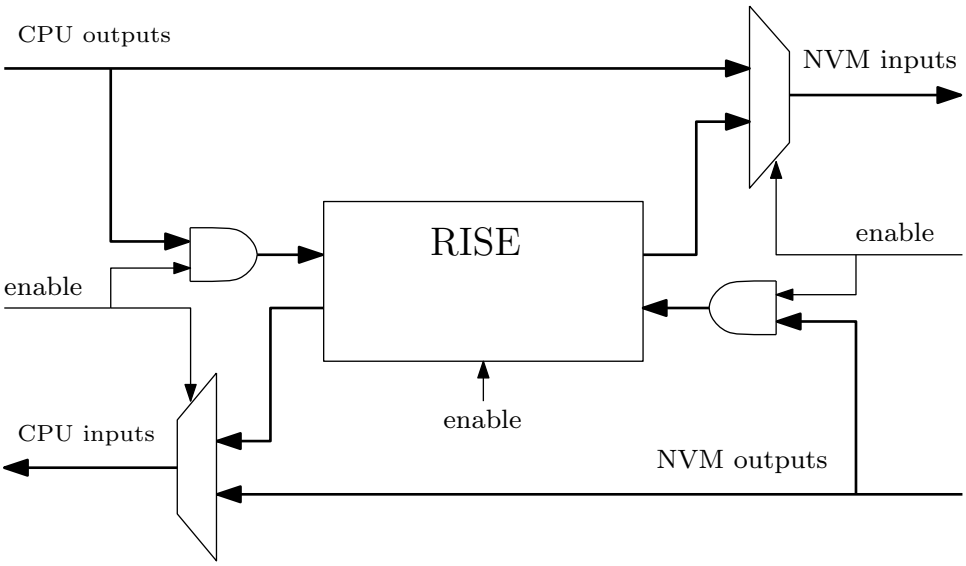

Figure 4.17: Integration of RISE into the SoC

The RISE IP can be disabled or enabled by multiplexing the input and output signals of the CPU and the NVM. In addition, we AND all the RISE input signals with the registered enable signal to reduce signal transitions at the inputs of RISE logic, thus reducing the power consumption when the IP is disabled. Right now, the IP can be disabled or enabled only at the power-up of the system. In the future, to allow greater flexibility, we propose that RISE be dynamically enabled or disabled programmatically by the CPU. This would require designing a RISE slave interface compliant to the internal bus protocol of the CPU, similar to other peripherals. This would be useful when the SoC clock frequency changes on the fly as there is no need for RISE to be enabled when $\frac{1}{f_{clock}} = T_{clock} < T_{crit}$

Chapter 5

Parameter Optimization

The described branch prediction mechanisms are mostly heuristic and not much can be said about the predictor performance prior to executing some benchmark software. The same applies for the different combinations of the table parameters. In the previous sections, we have tried to make some educated guesses about the effects of the different RISE parameters on its prediction performance. However, to at least provide an estimate of the optimal values for these parameters with respect to hit rate, die area, and power consumption, we will have to benchmark our IP using different configurations and choose the optimum based on the benchmark results.

As for the code itself, we will use a modified version of the CoreMark benchmark that is intended for testing CPUs used in embedded systems [13]. CoreMark comprises of simple algorithms such as matrix multiplication, list sorting, and state machine transitions. At the end of its operation, it also performs a cyclic redundancy check on the algorithm data to validate the results. With the data structures and algorithms used, CoreMark provides a workload quite common in the space of embedded CPUs. Nonetheless, the benchmark results are only an *estimate* of the RISE performance when presented with the intended software. We use these results primarily to compare the different parameter configurations and we should not be overly pessimistic or optimistic about the yielded absolute values of the performance statistics.

We ran the benchmark as a waveform simulation. Alternatively, we could run the design on a physical FPGA. That would enable us to benchmark with more test vectors in a fraction of the time. However, the simulated benchmark can be automated with a simple script and also the run statistics outputting is much easier, so we chose the former.

While we were benchmarking our design it became clear, that the $h = \frac{hits}{misses+hits}$ is a poor indicator of RISE performance. This is due to collisions (described in **Section 4.3.2**) which can induce false hits. This happens when a non branching instruction which maps onto a cache lines invokes a 'correct' Not Taken prediction. We have thus replaced the h with the $t_{norm} = \frac{t_{run}}{t_{ideal}}$ — simulation run time normalized to the run time of the DUT with RISE disabled. This metric is also convenient because it is

immediately apparent what the required frequency gain for overall speed up is, i.e. $\frac{f_n}{f_o} > t_{norm}$.

In **Figure 5.1** we show the benchmark results for different values of tag index size t_{size}. The used configuration was a two-way associative cache with 128 cache lines ($S = 2, N = 64$) and a clipped address size of 10 bits. We will use this basic configuration in other tests if not said otherwise. We can see that the associated run time improvement is small relative to the die area cost — the tag index bits have to be implemented as D flip flops which are high in transistor count per bit. We therefore think that for a cache with a number of sets equal to S a $t_{size} = \log_2 S$ is sufficient (accuracy-wise).

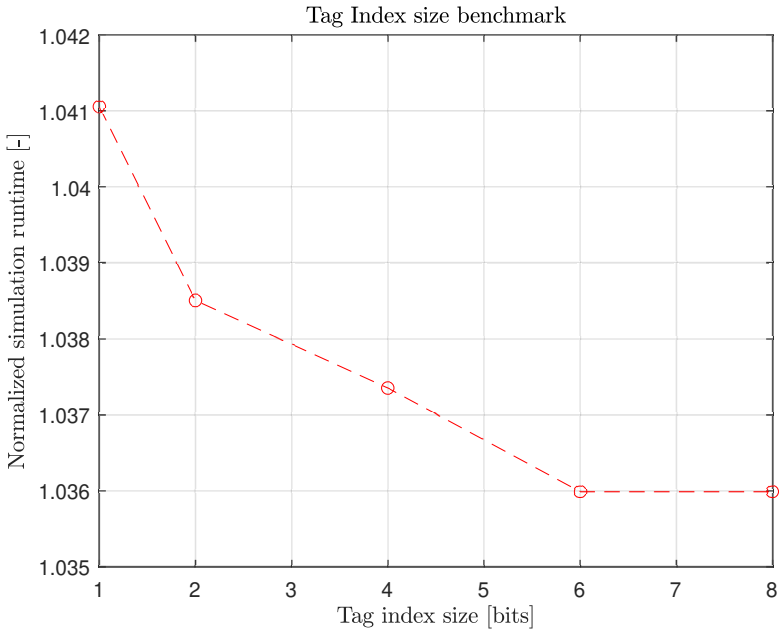

Figure 5.1: Benchmark results for different tag index sizes

Figure 5.2 shows the effects of different clipped destination address size c_d. We see a noticeable increase in performance for $c_d > 8$. Generally, we think that the c_d should be at least 10 bits, which saves us a third of the destination address memory.

Next, we tested different cache types, specifically direct-mapped, full associative, two-way associative and four-way associative cache. All had $c_d = 10$ and $t_{size} = \log_2 S$. Surprisingly, as a function of total cache lines, the different cache performances were nearly identical. We think that this is because the number of cache lines was fairly small in comparison to large-scale predictors used in modern CPUs. Most RISE misses were then caused by the inherently limited capabilities of the one-level predictor and not because of collisions or cache misses. In **Figure 5.3** we show the benchmark results only for the direct-mapped cache. For comparison, we also show the performance of the full-sized (every instruction address maps to a unique line) prefetcher. We think that for a number of lines smaller than 2^9 the direct-mapped cache is

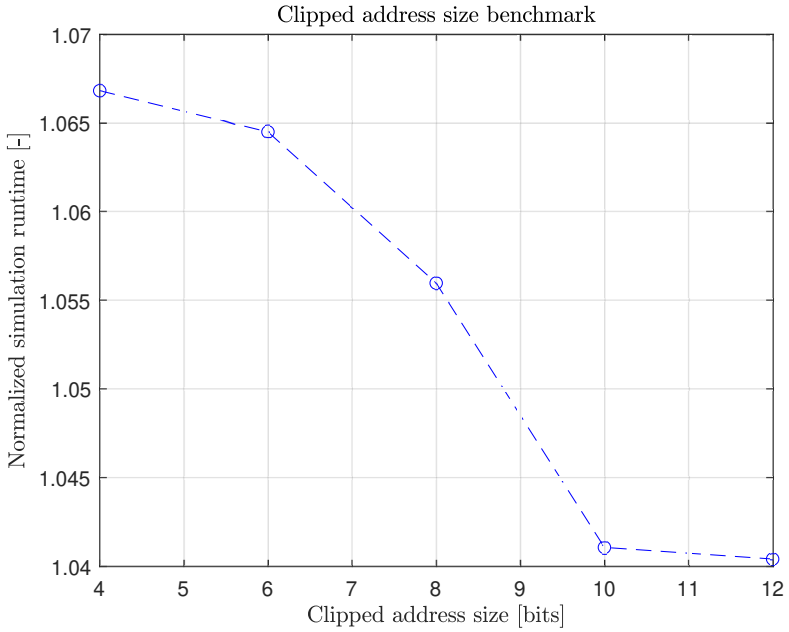

Figure 5.2: Benchmark results for different clipped address sizes

optimal, as it requires the least amount of logic and consumes the least power per clock edge while performing nearly just as well as other types.

Since the RISE main memory is the IP's largest block of logic, its power consumption will be a strong factor when determining the total power consumption. We benchmarked our IP, using different cache sizes and tag index sizes t_{size}, and we counted the number of main memory accesses multiplied by the size of the memory (SRAM dynamic power consumption is proportional to its size). The result is a number R_i that should be proportional to the dynamic energy consumed by the SRAM. In **Figure 5.4** we show these results.

We use $R_{norm} = \frac{R_i}{R_{max}}$ because the purpose of this benchmark is only to compare different configurations and not give any meaningful absolute values. We can see from the results that, as predicted in **Section 4.3.2**, the higher t_{size} leads to a lower number of RAM accesses. However, the added benefit of adding more tag index bits largely disappears for $t_{size} > 2$ (for $t_{size} = 4$ and $t_{size} = 8$ the data were so similar that we displayed them as one curve). If we were to use a cache of a smaller size, then choosing $1 < t_{size} < 4$ should be beneficial in terms of power consumption, even if it does not improve accuracy that much.

One must not forget that these results are only illustrative and to obtain a reliable estimate of the power consumption of different configurations, we would have to use a specific tool, such as the Xilinx Power Estimator. This is out of the scope of this thesis.

We have also tested the two-level prediction (same configuration as in tag index testing), and in **Figure 5.5** it is shown that this scheme performs

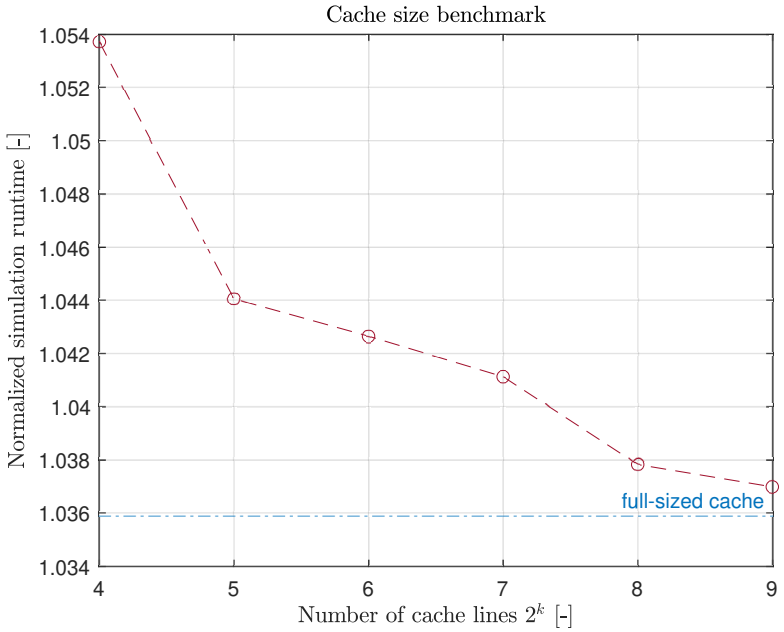

Figure 5.3: Benchmark results for various cache sizes

very poorly indeed. The larger we make the branch history, the less accurate the predictor is — the exact opposite of what we are trying to achieve. Our explanation is that a large history register will contain a lot of noise information, making it difficult to select the correct predictor for a given branch instruction. The one-level scheme therefore decisively beats the two-level scheme with global history.

Based on these benchmark results, we estimate that the cache collisions and cache misses influence the RISE performance only slightly — tag index of size $\log_2 S$ seems sufficient, number of cache lines is satisfactory in the 32-128 range and all cache types perform very similarly. Most of the prefetcher misses are caused by the limitations of the one-level prediction scheme. The ratio of misses inherent to the dynamic prefetching without ID (initializations, reinitializations) is very much dependent on the workload being run and is usually in the $5\% - 20\%$ range. This was measured by comparing the CoreMark benchmark and our internal benchmark. We therefore think that if a higher speed up is required, the dynamic prefetcher with ID is needed.

We also benchmarked the static prefetcher with resulting $t_{norm} = 1.082$. As predicted, this makes the static approach inferior (accuracy-wise) to the dynamic one in most configurations.

The $\frac{f_n}{f_o}$ is estimated to be around 10% and we can thus say that most dynamic prefetcher configurations should result in an overall system speed up. However, the uncertainty of this value is considerable and if we wish to obtain a better estimate we would have to run an Application-Specific Integrated Circuit (ASIC) synthesis of our SoC which is out of the scope of this work.

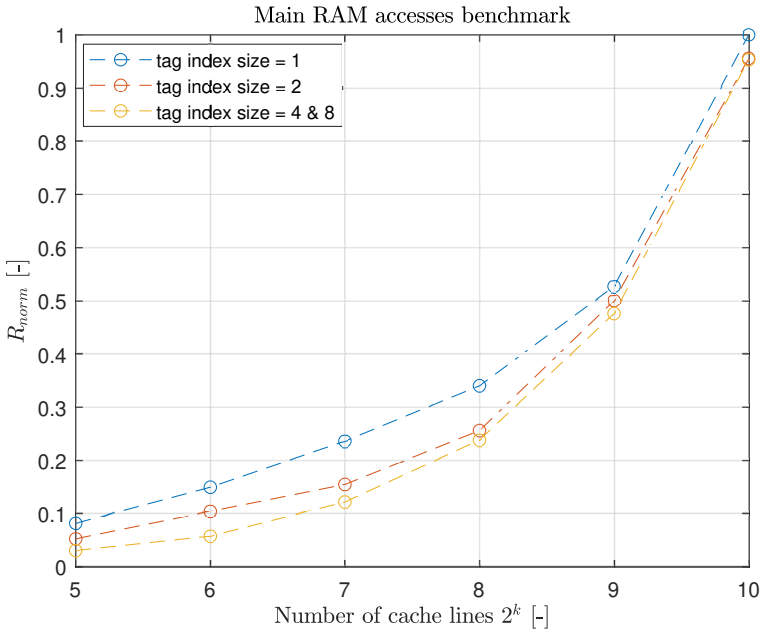

Figure 5.4: Benchmark results RAM accesses

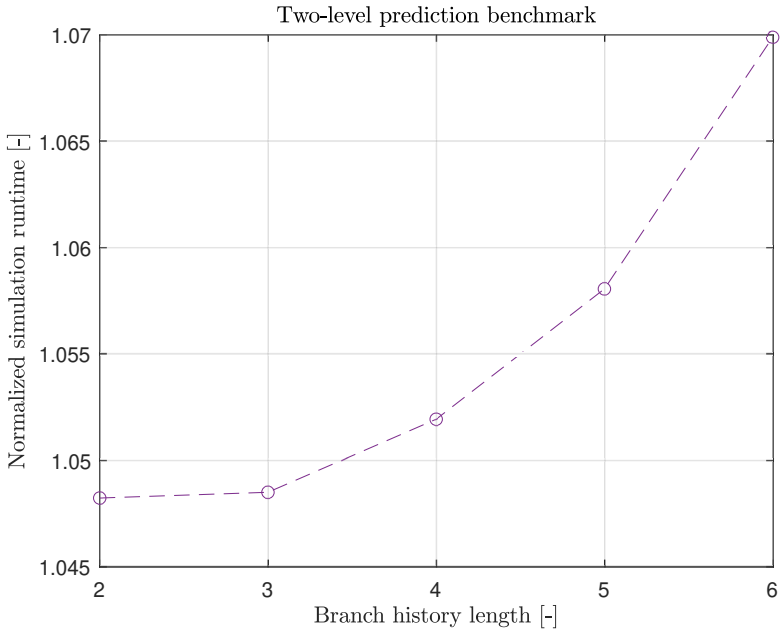

Figure 5.5: Benchmark results two-level prediction

Chapter 6

Verification

6.1 System Verilog testbench

We have verified the RISE IP throughout the design process using a simple System Verilog testbench. The Design Under Test (DUT) was comprised of the RI5CY based CPU, RISE IP, code flash, and data RAM. The purpose of this test was to verify the compliance of our design to these requirements:

1. The CPU shall receive all instructions that it requests in the correct order.
2. All of the IF transactions shall be compliant to the OBI bus protocol.
3. The RISE IP shall correctly execute the given prediction mechanism.
4. The RISE IP cache shall correctly execute the following operations: line update, line read, line initialize, and line reinitialize.
5. Line update (non-redundant) shall have higher priority than line read.

The testbench thus implements a following simple testcase:

1. The External Master (EM) resets the DUT and disables RISE,
2. CPU runs a simple benchmark code — matrix multiplication, Fast Fourier transform, and insert sorting. The CPU then outputs a single result value (XOR of all the mathematical operations result) to the EM,
3. EM resets the DUT and enables RISE,
4. same benchmark runs again, the result value is read,
5. compare the result values.

Example of the simulation waveform is shown in **Figure 6.1**.

Simultaneously, we have also embedded PSL and VHDL assertions into this testbench which help us to verify the requirements. One of them checks whether the CPU requested instruction address matches the one that the

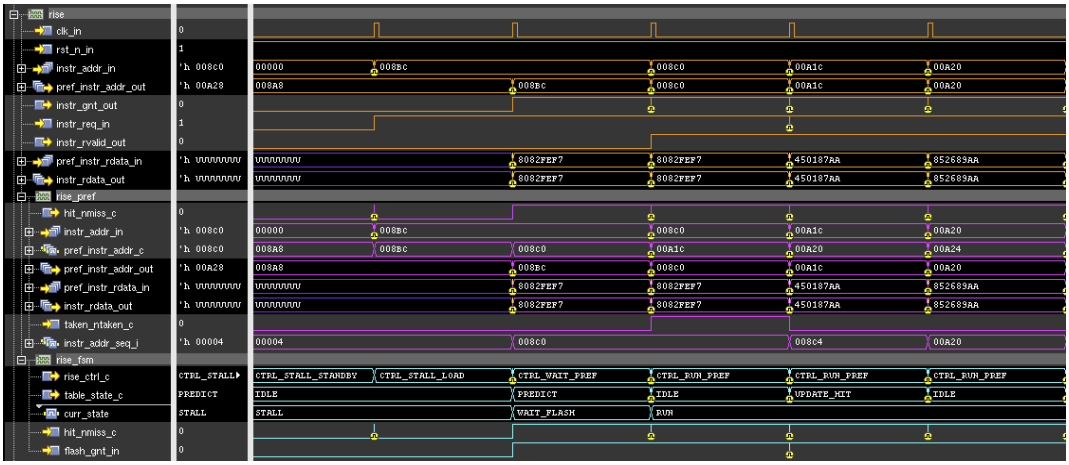

Figure 6.1: RTL simulation waveform

RISE IP fetches from the code memory — if not, then the whole simulation is aborted.

The described test only verifies the first requirement. This is a necessary but not sufficient condition for a proper RISE function. Consider the dynamic prefetching mechanism — even if a design error causes RISE to have a subpar hit rate, then the described test returns no errors. During the test, the CPU runs a moderately long code with non-trivial branching patterns. We were therefore able to visually spot from the simulated waveform an error in the predictor design because it would cause a noticeable degradation of predictor performance. However, some corner case errors influence the hit rate only slightly and we cannot rely on visual checks to detect them. We have therefore integrated additional PSL and VHDL assertions into the RISE IP to monitor the proper function of the table cache and prediction FSM. All of the design errors were fixed and the RISE IP is working properly on the RTL level.

6.2 Physical design verification

After the RISE RTL had been verified, we moved ahead in the verification process and started doing synthesis and physical implementation. We chose the Artix-7 xc7a100tcsg324-2 FPGA. The integration of our IP and the resulting DUT is shown in **Figure 6.2**. The CPU communicates with the peripherals using the AHB lite protocol via the Memory Crossbar. The JTAG is connected to the CPU for debugging purposes. The Clock Reset Manager provides the clock and reset signals to all other blocks based on the main system clock. It also allows us to change the clock frequency on the fly. We also have a set of timers and GPIO registers. External hardware interrupts are handled by the Interrupt ReQuest (IRQ) controller.

We generated the system clock using a Digital Phased Locked Loop (DPLL) hardwired into the FPGA fabric. The input to the DPLL is the board main

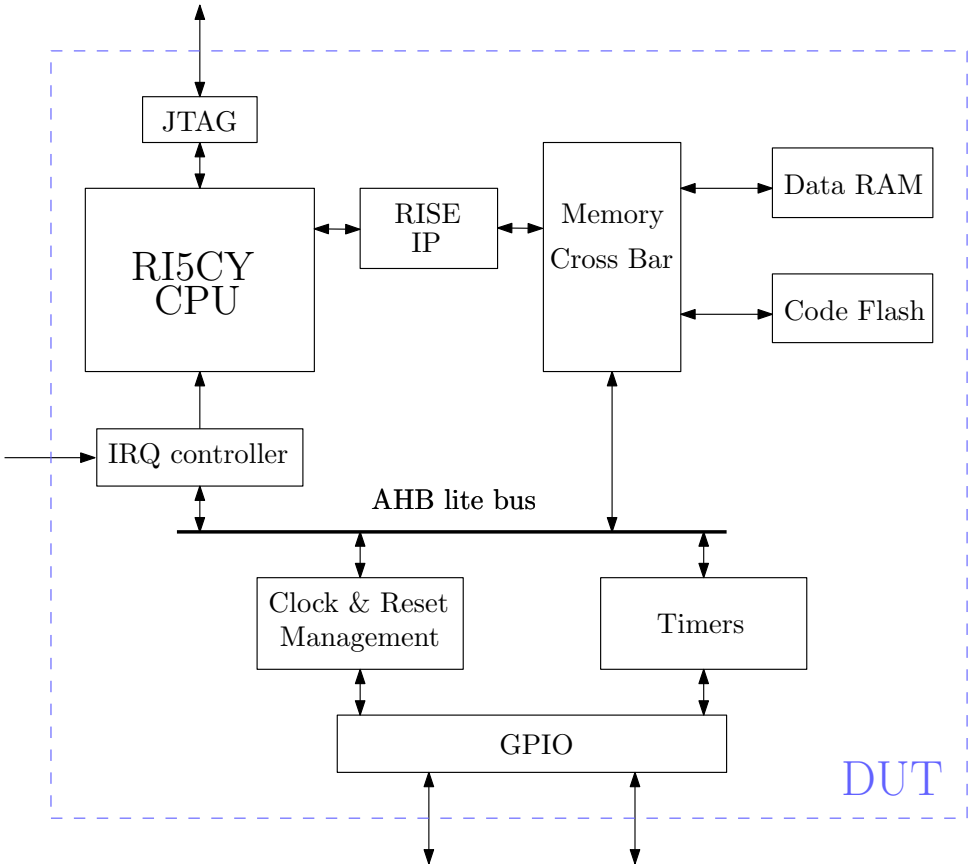

Figure 6.2: Block diagram of the DUT

clock operating at 100MHz and the output is a 12MHz clock which was then used as the system clock. The clock sources were specified in the timing constraints

In **Figure 6.3** we show the results of the timing analysis after the design implementation.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.000 ns	Worst Hold Slack (WHS):	0.053 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	23044	Total Number of Endpoints:	23044	Total Number of Endpoints:	7889

All user specified timing constraints are met.

Figure 6.3: Results of timing analysis

Physical constraints were written to specify the FPGA pad mapping, reset signal, Rise enable signal, and I/O standards.

The synthesis and implementation was run for three cache configurations: 4-way associative ($S = 4, N = 32$), full associative ($S = 64, N = 1$) and direct mapped ($S = 1, N = 256$) respectively. All configurations have 6-bit wide

tag index, use one level prediction and save 10 lower order bits of the source address. The FPGA resource utilization is reported in **Table 6.1**.

Apart from the absolute number of LUTs and Register slices, we also show the slice ratio relative to the CPU so that the trade-off between CPU speed up and additional logic is better apparent.

We also list the size of the main predictor RAM which stores the source address and prediction bits.

Resource	$S = 4, N = 32$	$S = 64, N = 1$	$S = 1, N = 256$
Slice LUTs [-]	681	483	1032
Slice LUTs ratio [%]	6.7	4.7	10.2
Slice Registers [-]	999	544	1934
Slice Registers ratio [%]	28.0	15.1	53.9
main RAM [bits]	1536	768	3072

Table 6.1: Comparison of resource usage for different cache configurations

In Vivado, using the provided SIMPRIM primitive cells, we have generated a Verilog wrapper of our design and an SDF file (Standard Delay Format) which specifies the delay of the primitives. We then used these two generated files to run a gate-level simulation of our design. The example waveform is shown in **Figure 6.4**. The same simple test 6.1 was run successfully and the waveform was reviewed visually.

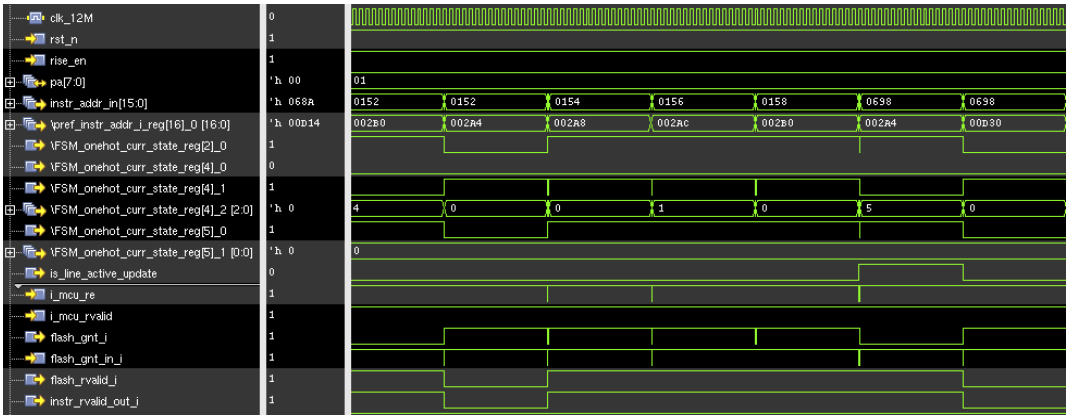

Figure 6.4: Gate level simulation waveform

Chapter 7

Conclusions

The main goal of this thesis was to design an optimization of a CP in a low-power, low die area digital system containing a RISC-V CPU based on the open-source RI5CY core. The CP is part of the IF interface connecting the CPU and the instruction memory.

Number of possible power and die area aware design architectures for our RISE IP were proposed and compared. Among them the prefetcher solution was chosen for implementation, specifically the static mechanism and the dynamic mechanism without ID.

When implementing the latter, we had chosen to design the table memory as a cache and we studied extensively the possible effects of its parameters on the prefetcher performance. We have provided a similar study on the one-level and two-level branch prediction mechanisms.

The two prefetcher solutions were described in VHDL language with emphasis on making as many of the design parameters reconfigurable, so that they could be then optimized based on benchmark results. This approach was needed, because the prediction algorithms are heuristic in nature and no satisfactory estimate of the prefetcher performance could be made based on the theoretical analysis alone.

We benchmarked our IP in different configurations using the CoreMark software to obtain an estimate of the optimal parameter set. In summary:

- Cache collisions and cache misses degrade the performance only slightly — cache with 2^4 lines performed quite similarly to a 2^{15}-line cache,
- a more noticeable effect had the number of saved destination address bits with 10 bits being the minimum recommended length,
- the one-level scheme outperforms the two-level scheme (with global history length) easily,
- we assume that the main memory power consumption is reduced by using more tag index bits but for a large number of cache lines and tag index bits, the added benefit vanishes,
- as expected, the dynamic prefetcher without ID outperforms the static prefetcher in just about any configuration,

- the dynamic prefetcher with ID could be $5\% - 20\%$ more accurate that the one without ID.

Based on the simulation results, we estimate that the use of the dynamic prefetcher without ID (in most configurations) will result in an overall system speed-up of around 5%. The exact performance gain is not given in this thesis as it would require doing ASIC synthesis and place-and-route of our SoC.

In the future, we propose designing the dynamic prefetcher with ID. We could also benchmark our system with a workload more similar to the software that is intended to be run on the SoC to obtain more accurate data.

Our IP was verified with a simple System Verilog testbench to ensure basic functionality. When the design process had finished and the IP microarchitecture had become stable, we implemented the prefetcher in an Artix-7 xc7a100tcsg324-2 FPGA. Also, as a part of the physical design verification, we ran a gate-level simulation of our design.

The thesis goals were accomplished and the RISE IP is ready for further development as a part of the Integrated Circuit design efforts in ASICentrum s.r.o. The next goal is to integrate the IP into an ASIC. Prior to that, we would desire a more thorough verification of our design, possibly by exploring the code coverage or by using formal verification methods. ASIC synthesis of our IP would also be needed.

The future use-cases of the RISE IP are not limited to our RI5CY based CPU. If we were to redesign the interfacing of the IP with the CPU and the NVM so that it is compliant to a different bus protocol, we could use our design for any other CPU which suffers from the same CP problem.

More generally, in any master-slave topology where we have a CP as part of the master-to-slave addressing (as illustrated in **Figure 2.1**), we could use the RISE IP to optimize the CP as long as the addresses exhibit the spatial locality as discussed in **Section 3.3.3**. However, to ensure that the performance that results in overall speed-up (discussed in **Section 3.3.2**) is achieved, we would have to consider the characteristics of the address stream specific to that system.

Bibliography

[1] TRABER, Andreas, Michael GAUTSCHI a Pasquale Davide SCHIAVONE. RI5CY: User Manual [online]. In: ETH Zurich and University of Bologna, 2019 [cit. 2022-05-09]. Available from: https://www.pulp-platform.org/docs/ri5cy_user_manual.pdf

[2] Open Bus Protocol: OBI 1 [online]. Silicon Labs, 2020 [cit. 2022-05-09]. Available from: https://github.com/openhwgroup/core-v-docs/blob/master/cores/obi/OBI-v1.4.pdf

[3] HENNESSY, John L. a David A. PATTERSON. Computer architecture: A Quantitative Approach. 5th edition. Waltham: Morgan Kaufmann, c2012, p. 677-683. ISBN 978012-3838728.

[4] LEE, Alan Jay a Johnny K.F. SMITH. Branch Prediction Strategies and Branch Target Buffer Design. Computer. 1984, 17(1), 6-22. ISSN 0018-9162. Available from: doi:10.1109/MC.1984.1658927

[5] PATTERSON, David A. a John L. HENNESSY. Computer organization and design: the hardware/software interface. 4th rev. ed. Waltham: Morgan Kaufmann, c2012, p. 374-378. Morgan Kaufmann series in computer architecture and design. ISBN 978-0123747501.

[6] SMITH, Alan Jay. Cache Memories. ACM Computing Surveys. 1601 Broadway, Times Square, New York City, United States: Association for Computing Machinery, 1982, 1982(14), 473–530. ISSN 0360-0300.

[7] DOUGLAS, Jones. 22C:116, Lecture Notes [online]. University of Iowa Department of Computer Science, 6.2.1995 [cit. 2022-05-09]. Available from: http://homepage.cs.uiowa.edu/ jones/opsys/fall95/notes/old/0206.html

[8] HARRIS, David Money a Sarah L. HARRIS. Digital design and computer architecture. 2nd ed. Waltham: Morgan Kaufmann, c2013, p. 266-267 ISBN 978-0123944245.

[9] YEH, Tse-Yu a Yale N. PATT. Two-level adaptive training branch prediction. In: MICRO 24: Proceedings of the 24th annual international symposium on Microarchitecture. 1601 Broadway, Times Square, New

York City, United States: Association for Computing Machinery, 1991, p. 51-61. ISBN 978-0-89791-460-4. ISSN 0360-0300.

[10] FOG, Agner. The microarchitecture of Intel, AMD, and VIA CPUs: An optimization guide for assembly programmers and compiler makers [online]. In: Technical University of Denmark, 17.8.2021, p. 36-37 [cit. 2022-05-09]. Available from: https://www.agner.org/optimize/microarchitecture.pdf

[11] YEH, Tse-Yu a Yale N. PATT. Alternative implementations of two-level adaptive branch prediction. MICRO 24: Proceedings of the 19th annual international symposium on Computer architecture. 2. 1601 Broadway, Times Square, New York City, United States: Association for Computing Machinery, 1992, 1992(20), 124-134. ISSN 0163-5964.

[12] MCFARLING, Scott. Combining Branch Predictors. 250 University Avenue Palo Alto, California 94301 USA: Western Research Laboratory, 1993.

[13] CoreMark: MCU benchmark [online]. Embedded Microprocessor Benchmark Consortium, 2021 [cit. 2022-05-09]. Available from: https://github.com/eembc/coremark/blob/main/README.md