RISC-V processors with Bluespec

High Performance Processors and System
A.Y. 2020/2021

Riccardo Nannini

June 21, 2021

Tutors: Emanuele Del Sozzo,
Davide Conficconi

Professor: Marco Domenico Santambrogio

POLITECNICO
MILANO 1863

POLITECNICO MILANO 1863
NECST laboratory

Contents

1 Introduction **2**

2 RISC-V **2**
2.1 What is RISC-V 2
2.2 Instructions set goals 3
2.3 Main features 3
2.3.1 Registers 3
2.3.2 Instruction encoding 3
2.3.3 Instruction format 4
2.4 License 5

3 BSV **5**
3.1 General overview 6
3.1.1 Modules and interfaces 6
3.1.2 State 7
3.1.3 Types and polymorphism 7
3.1.4 Rules 8
3.1.5 Dynamic semantic 8

4 Processor designs **8**
4.1 Starting point 9
4.2 Two cycles 9
4.3 Four cycles 10
4.4 Two-stage pipelined 11
4.4.1 Next address prediction 12
4.5 Six stage pipelined 13
4.5.1 BHT 14
4.5.2 RAS 14

Abstract

Bluespec System Verilog (**BSV**) is a state-of-the-art Hardware Description Language. Bluespec compilation toolchain (**BSC**) has been recently released as open source [1]. The goal of the project was investigating the potentiality of said toolchain implementing different **RISC-V** processors of increasing complexity.

1 Introduction

This report covers chronologically the path that I have followed during the development of this project.

It starts from a quick overview on the *RISC-V* ISA, focused on the key ingredients that make this ISA one of the most trend topics in Computer Engineering.

It proceeds with the analysis of the *Bluespec System Verilog* language, outlining its novelties with respect to other hardware description languages, as well as its main features and capabilities.

The last section is devoted to the development of various RISC-V processor designs, describing in details the characteristics of each processor and motivating the various design choices, starting from a one cycle non pipelined processor and ending with a 6 stage pipelined one enriched with multiple branch predictors.

Everything I have produced within this project (code, scripts, this report itself) is available on GitHub `https://github.com/riccardo-nannini/BSV_RISC-V`

2 RISC-V

2.1 What is RISC-V

Since the processor designs will be based on RISC-V, knowing this ISA is fundamental.

RISC-V was born in 2010 at UC Berkley by the work of graduate students Andrew Waterman, Yunsup Lee and professors David Patterson, Krste Asanovic [2], built on the experience gained from the 4 previous major RISC ISA design from UC Berkeley (thus justifying the 'V').

As the name suggests, RISC-V belongs to the family of **RISC** ISAs. In particular, RISC instruction sets are characterized by *small* and *highly optimized* instructions that ease *pipeline* implementation, along with a large number of *general purpose registers*.

RISC-V, like many other RISC ISAs, is a **load-store architecture**; this means that, apart from some exceptions, arithmetic and logic instructions op-

erate on register and not on memory directly. Load and store instructions take care of the transit of data from memory to register and vice versa.

2.2 Instructions set goals

When the first paper that declared the birth of RISC-V [2] was released back in 2011, the authors stated some of the goals that they wanted to achieve when designing this instructions set. In particular the main objectives include, but are not limited to:

- Provide a *realistic* but *open* ISA, suitable for direct hardware implementation
- Provide a **small** but **complete** ISA, that avoids "*over-architecting*" for a particular microarchitecture style or implementation technology.
- Support highly-parallel multicore implementations, including heterogeneous multiprocessors
- Support an efficient dense *instruction encoding* with variable-length instructions (generally typical of CISC ISAs).
- **base-plus-extension**: provide a set of standard but optional extensions.

In order to achieve these goals many design choices were taken, sometimes in line with other similar ISAs, sometimes with unusual decisions.

2.3 Main features

RISC-V has 2 base variants, **RV32** and **RV64**, providing 32-bit or 64-bit user-level address spaces respectively. A 128-bit extension is available but mostly aimed at the future of warehouse scale computing.

2.3.1 Registers

RISC-V uses 31 general-purpose registers `x1-x31` with `x0` hardwired to the costant 0. Moreover, 32 64-bit registers `f0-f31` holding single or double-precision floating-point values are available.

There are also two special registers: the *program counter* `pc` holds the address of the current instruction while the *floating point status register* `fsr` contains the operating mode and exception status of the floating-point unit.

2.3.2 Instruction encoding

In addition to standard fixed-length 32-bit instructions, RISC-V supports extensions with **variable-length** ones too, useful in reducing code size specially for specific domains like embedded systems.

Figure 1 shows the instruction length encoding convention. 32-bit instructions have the 2 least significant bits always set to `1` while the 16-bit ones must have said bits in any combination but `11`. Instructions longer than 32-bits have additional lower bits set to `1`.

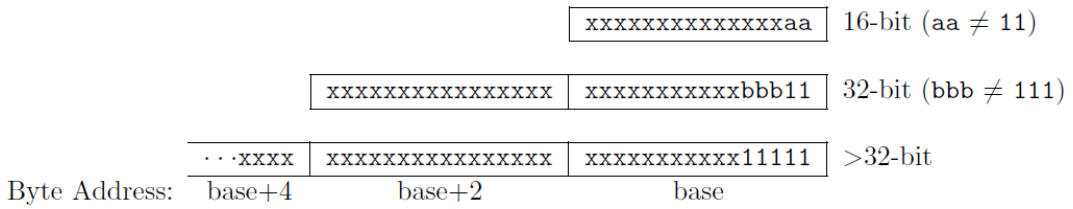

Figure 1: RISC-V instruction encoding

2.3.3 Instruction format

Six basic instruction formats are present in the base version of the ISA as shown in Figure 2.

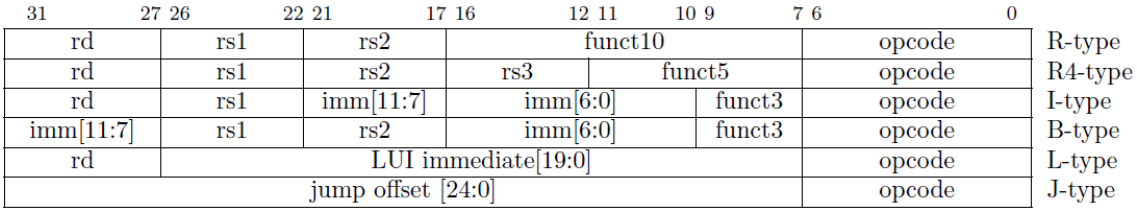

Figure 2: RISC-V instruction formats

It is worth noting a particular design choice made by the authors. Since, as they state, the *decoding* of register specifiers is usually on the **critical path** in implementations, they decided to keep all register specifiers in the same position across every instruction type. This comes with the tradeoff of having some instructions with an immediate value that has to be split in two parts.

R-Type instructions are for operations with two source registers (*rs1* and *rs2*) while **R4-Type** format is for special three source register floating point instructions. Moreover, **I-Type** format is for operations between a source register (*rs1*) and an 12-bit sign-extended immediate, while **B-Type** instructions are designed for conditional branches (where the branch type is specified in *funct3*). The LUI (*Load upper immediate*) operation, generally used to build 32-bit constants, is the typical instruction specified by the **L-Type**, meanwhile **J-Type** instructions encode a 25-bit target address as a PC-relative offset.

Describing in details the specification of every instruction is way out of the focus of this report and is already done with great details in the original paper [2].

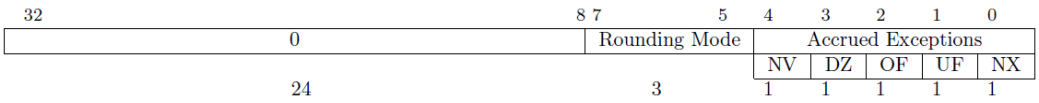

Figure 3: Floating-point status register

In the end, just a mention to the *fsr*, a special register containing the status of floating-point operation, shown in Figure 3.

It makes possible to specify the rounding mode with a specific encoding as well as reading the status of the previous floating-point operation in the flag bits.

2.4 License

Until here the main features of RISC-V were summarized. Despite being a really well designed standard, this by itself does not explain the popularity of this ISA.

In fact, one of the main factor of his success is that RISC-V is provided under a **free open source** license that does not require fees to use.

This is quite unusual in the instruction set architectures world, that for a great majority are *proprietary* due to historical or business reasons. Companies like ARM, IBM and Intel have patents on their ISAs which prevent others from using them without licenses.

Negotiations for said licenses can take 6-24 months and can cost up to $1M-$10M, ruling out academia and other with small volumes. An ARM license, for instance, doesn't even let you design an ARM core; you just get to you *their* designs [3].

Some open-source ISA already existed but were not as popular and well designed as RISC-V is.

The birth of this standard enabled a **real free open market** of processor designs leading to:

- greater innovation via **free-market competition**
- shared open core designs
- processors becoming **affordable** for more devices

3 BSV

The hardware description language used to implement the RISC-V processors within the project is **Bluespec System Verilog** (BSV).

Belonging to Bluespec Inc, co-founded in 2003 by MIT professor Arvind, the BSV language is aimed at providing hardware designers used to Verilog, VHDL or System Verilog, a **high level** language for hardware design and synthesis.

Based on a synthesizable subset of SystemVerilog, BSV can significantly improve the hardware design process with some key innovations:

- Hardware behavior is expressed with **Rules**. Rules are powerful concepts for achieving correct concurrency and eliminating race conditions.
- **Polymorphism** makes possible to write more general code and reuse designs and glue them together in a flexible way.
- Provides **formal semantics**, enabling formal verification and formal design-by-refinement. This is due to the fact that BSV rules are based on *Term Rewriting Systems*, a clean formalism supported by decades of theoretical research in the computer science community.

3.1 General overview

This section is devoted to the main features of BSV in order to give the reader a general overview of the language and understand its strength. The complete reference guide (also used for this quick introduction) can be found at [4].

3.1.1 Modules and interfaces

In order to understand BSV, it is useful start with the notion of **modules** and **interfaces**, the core of the language. Modules and interfaces are what, in the end, is turned into actual hardware.

An interface describes what a module 'offers' to the outside. On the other hand, a module 'implements' its interface as it prefers.

In BSV, an interface consists of *methods* that encapsulate the possible transactions that clients can perform.

```
1  interface Fifo#(int n); //n is the size of the fifo
2
3    method Action enqueue(Bool x);
4    method Action dequeue;
5    method Bit#(8) first;
6
7  endinterface
8
9  module mkFifo( Fifo#(n) );
10
11   //... implementation
12
13 endmodule
```

Code snippet 1 Example of module and interface

When it comes to modules, however, we need to distinguish between a module *definition* and a module *instantiation*. A module definition is like a class in object oriented programming while a module instantation can be considered as an object of said class.

A module consists of three things:

- a **state**
- **rules** that operate on that state
- an **interface** to the outside world

3.1.2 State

In Verilog and SystemVerilog, one simply declares a variable, and a synthesis tool "infers" how those variable actually map into state element in hardware; a variable may map into a bus, a latch, a flip-flop or even nothing.

BSV eliminates this ambiguity and places the state control directly in the hand of the designer. BSV *never* implies state; variables are just convenient names for intermediate values.

State components are declared (and instantiated) after the module declaration like in snippet 2

```
module mkFifo( Fifo#(n) );

    Vector#(n, Reg#(Bit#(8))) data <- replicateM(mkRegU());
    Reg#(Bit#(8))         value_reg <- mkReg(0);

    //...

endmodule
```

Code snippet 2 Example of state declaration

3.1.3 Types and polymorphism

Every variable and every expression in BSV has a *type*. BSV supports standard *simple type* like in snippet 3

```
int
Bool
String
```

Code snippet 3 Simple types

or *parametrized* types (either standard or user-defined)in the form of `X#(`t_1`, ..., `t_n`)`

```
Typle2#(int, Bool)     //pair of items, an int and a Bool
List#(Bool)            //list of booleans
Vector#(n, int)        //vector of n integers
Bit#(16)               //16-bit wide bit-vector
Int#(29)               //signed integer, 29 bits wide
Vector#(16, List#(Int#(29)))  //vector of 16 Lists of 29 bits
    wide signed integers
```

Code snippet 4 Parametrized types

In snippet 5 it is possible to observe that parametrized types utilize **polymorphism**, making code way more reusable.

It is worth mentioning also the type **Action** that denotes any expression that acts on the *state* of the circuit, fundamental in order to produce useful hardware.

3.1.4 Rules

Rules specity the internal behavior of modules. A rule may have a *guard* (boolean expression) that indicates when the rule can execute (fire), and a *body*, of type Action, that describes what the rule does.

```
1 module mkCounter(Empty);
2   Reg#(Bit#(8)) value <- mkReg(0);
3
4   rule increment (value >= 0); //guard
5     value <= value + 1;        //body
6   endrule
7 endmodule
```

Code snippet 5 Autoincrementing couter with a single rule

The body of a rule can contain, with an ad-hoc semantic, the basic constructs typical of software programming languages such as if-else, switch, loop etc.

3.1.5 Dynamic semantic

The simplest way to understand the dynamic semantic of BSV is thinking that, at each step, the system picks any *one* enabled rule and fires it.

Since only one rule is executed at each step, we have only to look at each rule in **isolation**, without worrying about the interactions with other rules simultaneously. Each rule can be viewed as an **atomic state transition**.

The BSC compiler maps BSV into efficient parallel clocked synchronous hardware. In particular, the mapping allows multiple rules to be executed at each clock cycle. The compiler is able do to this by producing a **role-control circuit** which takes all the enabled conditions and possible data outputs (actions) and feeds them into a **scheduler** circuit. It is the scheduler duty, at each clock cycle, to select a subset of conflict-free rules to fire.

4 Processor designs

This section is devoted to the processor designs created within this project.

After studying the RISC-V ISA and getting to know the BSV language, it was the point to get my hands dirty and creating a variety of processor designs.

Every design, among other things like BSV working examples I have made as an exercise, is available in the GitHub repository `https://github.com/riccardo-nannini/BSV_RISC-V`.

4.1 Starting point

I was provided with an initial setup comprising a processor infrastructure (Figure 4) based on Connectal [5], some test benches, and an initial simple one cycle implementation of a RISC-V processor.

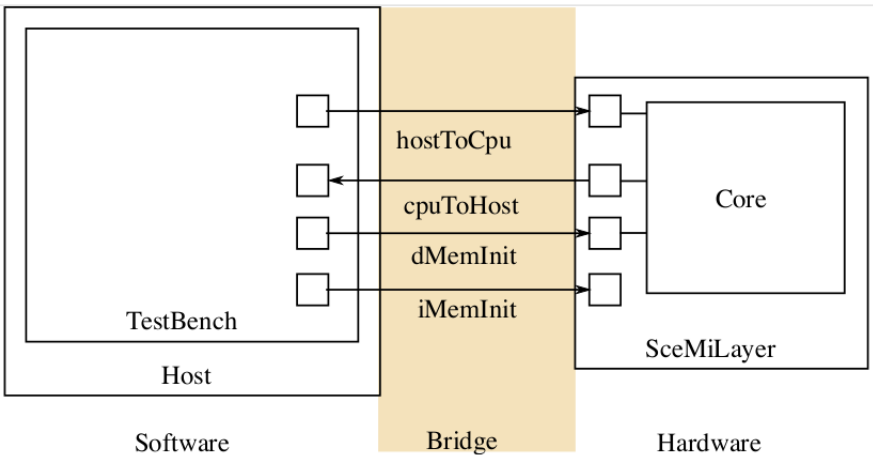

Figure 4: Processor infrastructure with control signals

Figure 5 shows the structure of the initial one cycle processor.

Starting from here, I developed more realistic processors of increasing complexity.

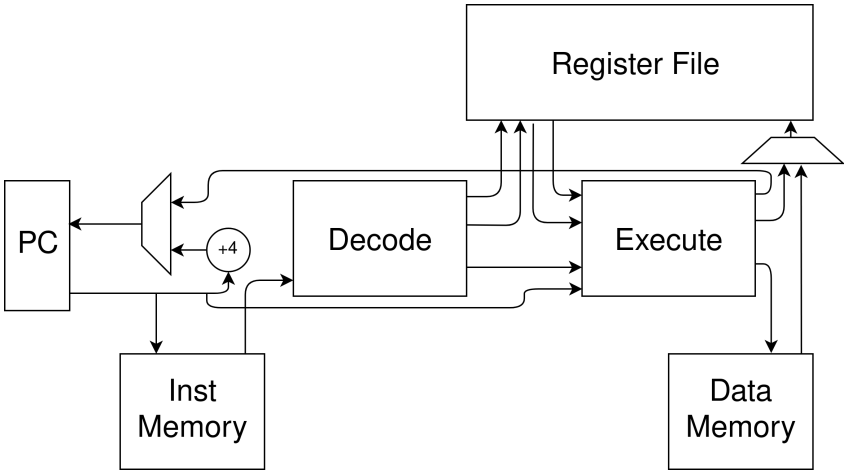

Figure 5: One cycle processor architecture

4.2 Two cycles

The one cycle processor is able to operate in a single cycle because it has **separate** instruction and data memory; my first design was aimed at creating a more realistic architecture, based on the von Neumann model, with a single memory accessible only once per clock cycle.

This indeed creates a structural hazard, solved by splitting the instruction execution in **two cycles**: one for fetching the instruction and one for decoding/executing the instruction and access the memory.

In order to achieve this, an intermediate register (`f2d`) has to hold the data that a state has to pass to the following one (in this case the fetched instruction).

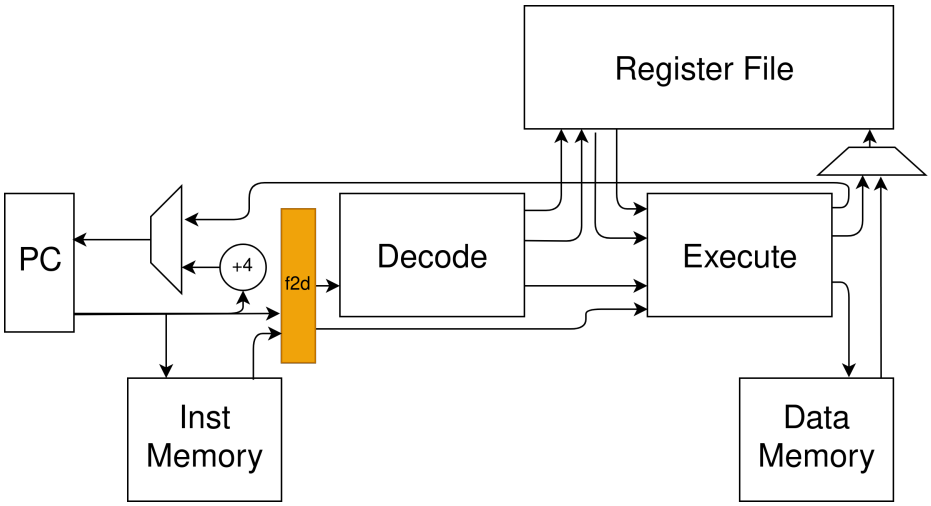

Figure 6: Two cycles processor architecture

```
typedef enum {
  Fetch, Execute
} State;

Reg#(State) state <- mkRegU;

rule doFetch(state == Fetch);
  f2d <= mem.req(pc);   //fetch instruction
  state <= Execute;     //pass to the execute state
endrule

rule doExecute(state == Execute);
  let instruction = f2d;
  //... decode, execute, memory and writeback
  state <= Fetch;
endrule
```

Code snippet 6 Two cycle processor simplified code

4.3 Four cycles

Another unrealistic assumption I had to get rid of was that, since now, memory access was treated as *combinational*. The processor had only to set some input lines to the memory and receive an answer in the same cycle. Most real memories have reads with **longer latencies**: first you set the address bits and then the result is ready in the next clock cycle.

This structural hazard is avoided by further splitting the processor into **4 cycles**:

- the *instruction fetch* stage sets the address lines on the memory to read the current instruction.
- the *instruction decode* stage reads the instruction from memory, decodes it and reads the registers.
- the *execute* stage performs ALU operations, writes data to memory and sets up address lines for load instructions.
- the *write back* stage reads the result from memory (if any) and writes the register file.

```
typedef enum {
Fetch, Decode, Execute, WriteBack
} State;

Reg#(State) state <- mkRegU;

rule doFetch(state == Fetch);
  mem.req(pc);      //requests the instruction
  state <= Decode;  //pass to the decode state
endrule

rule doDecode(state == Decode);
  DecodedInst dInst = decode(mem.resp());
  //... reads from registers
  state <= Execute;
endrule

rule doExecute(state == Execute);
  //performs ALU operation and request data if load instruction
  state <= WriteBack;
endrule

rule doWriteBack(state == WriteBack);
  //reads from memory if load and writes back to registers
  state <= Fetch;
endrule
```

Code snippet 7 Four cycle processor simplified code

4.4 Two-stage pipelined

In the above-described designs, when the processor was in a particular state, the rest of the hardware remained unused. A **pipeline** approach is aimed at improving performance by parallelizing instructions execution. This, however, brings up **data** and **control hazards** that we ignored up to now. In order

to make a step at the time, I started with a **two-stage pipelined** processor (bringing back the good old unrealistic combinational reads) that rules out data hazards (read registers, execution and write-back done in the same clock cycle).

While the stage structure is similar to the two clock design (fetch/decode in one stage, execute/memory/writeback in the second), we need some form of **next instruction prediction** (the fetch stage has to fetch the instruction before the previous one has terminated) and a way to kill wrong path instructions in case of a **misprediction**.

This can be achieved by using a `FIFO` queue as an intermediate register between the stages and, in case of a misprediction, clear said `FIFO` (as it contains a wrong-path instruction) and redirect the `pc` with the correct address. This pc-overwriting action is possible due to `EHR` (*Ephemeral History Register*), an hardware structure that allows to have multiple write with different 'priorities'.

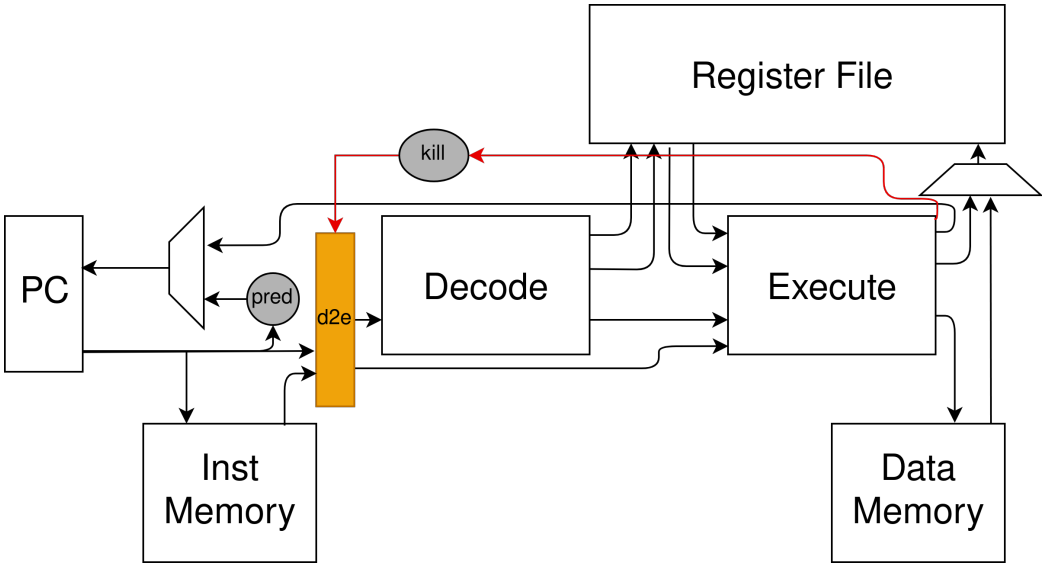

Figure 7: Two stage pipelined processor architecture

4.4.1 Next address prediction

The next instruction prediction can be carried out in multiple ways. At first I used a simple `pc+4` as a predictor for this design and then replaced it in a second moment with a `BTB` (*Branch Target Buffer*), a cache storing the target address for the next instruction after a branch. In the fetch stage the `BTB` is accessed using the n lowest bit of the current pc as an index to the cache. If an entry for the `pc` is present it is selected as the next `pc`, otherwise `pc+4` is chosen.

Figure 8 shows the different performance of the design when using a `BTB` with respect to a plain pc + 4 predictor.

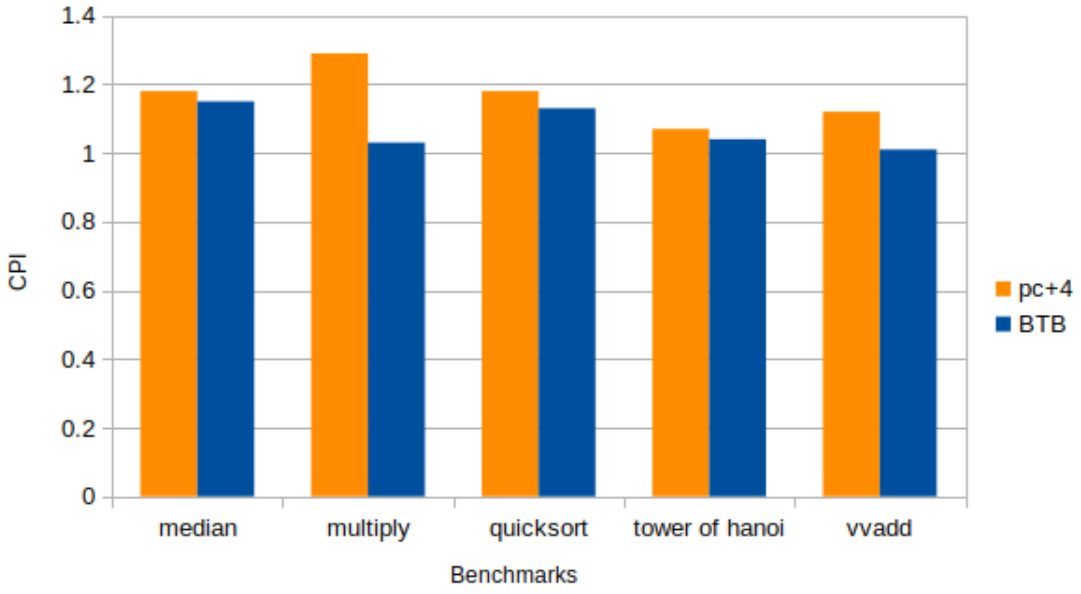

Figure 8: CPI comparison with different benchmarks

4.5 Six stage pipelined

For the last design I have implemented a six stage pipelined processor, getting rid of combinational reads of the two stage one.

This, however, creates the possibility of having data hazards. Since the execution will be in-order, the only hazard we should take care of are **RAW** (Read After Write); an instruction might read a register that would have been written by a previous instruction that has not committed yet.

In this design I solved this type of conflict using **Scoreboard**. Scoreboard is a data structure for *centralized hazard management* that keeps track of the registers currently involved in the pipelined instructions.

Every instruction goes through the `scoreboard`: if no conflict are found the instruction proceeds in the pipeline and the used registers are saved in the `scoreboard`; if a conflict is found, the instruction stalls.

```
1  Scoreboard#(10)   sb <- mkCFScoreboard;
2
3  rule doRegisterFetch();
4    if (!sb.search1(inst.src1) && !sb.search2(inst.src2))
5    //search if the operand register are present in the
       scoreboard
6      begin sb.insert(inst.dst);
7      //...
8      end
9    else
10     // STALL
11 endrule
```

```
12
13 rule doWriteBack();
14   sb.remove   //removes the instruction from scoreboard
15 endrule
```

Code snippet 8 Scoreboard integration in the pipeline

The six stage pipeline is divided in the following stages:

- **Instruction fetch**: requests instruction from memory and updates PC using the BTB.
- **Decode**: receives instruction from memory and decodes it.
- **Register fetch**: reads from register file.
- **Execute**: executes the instruction and corrects mispredictions.
- **Memory**: sends memory requests for load/store.
- **Write back**: receive responses and writes to register file.

The processor needs an intermediate `FIFO` queue between each stage carrying the necessary information forward.

4.5.1 BHT

In order to further improve branch prediction I have implemented and integrated a **2-bit BHT** (*Branch History Table*). It is a table with 2 bits for each entry that are updated based the previous behavior of the branch. By checking said bits, the `BHT` predicts whether to take or not the branch.

The `BHT` is accessed in the decode stage (we need to know that the instruction is a branch in order to use the table) and redirects the `pc`, killing wrong-path instructions, in case the prediction of the `BTB` was not in line with the decision of the `BHT`.

Figure 9 shows how the number of clock cycles per instruction changes in different benchmarks when using the `BHT` or not.

4.5.2 RAS

As the last improvement to branch predictions I have implemented a **RAS**. The `Return Address Stack` is a predictor aimed at improving *Jump register* instructions (`JALR` and `JAL`). According to the RISC-V specification, this two types of unconditional jumps save the address of the instruction following the jump (`pc+4`) in register x1 (`JAL`) or in register *rd* specified in the instruction (`JALR`). Most `JAL` and `JALR` instructions found in programs are used to call or return from functions.

The `RAS`, implemented along with `BHT` in the decode stage, works is by maintain a stack, pushing return addresses when a jump calling a routine is

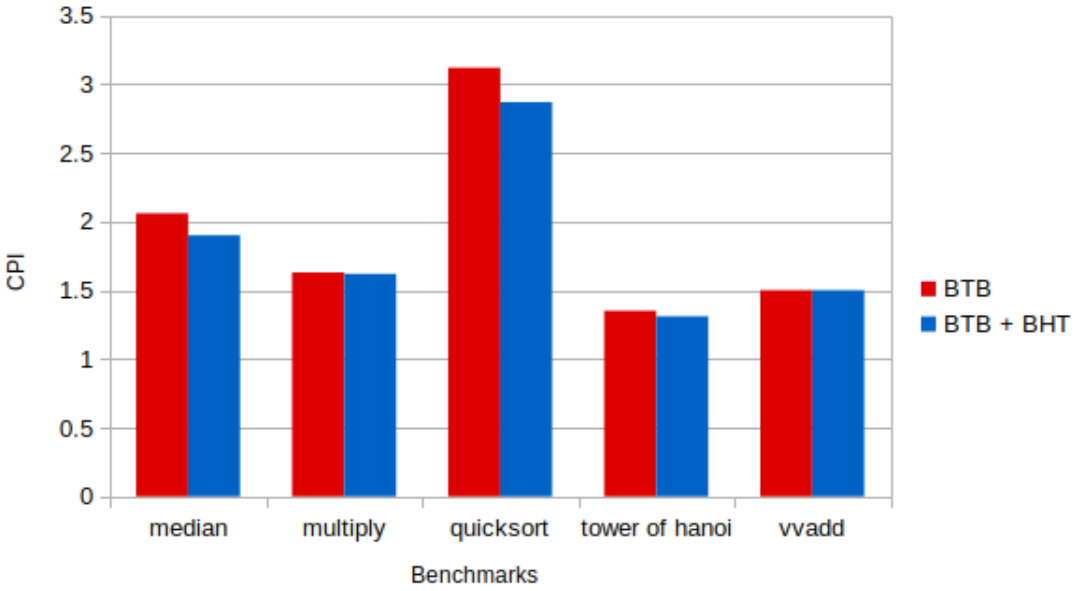

Figure 9: CPI comparison with different benchmarks

found and popping said addresses when another jump instruction returning from a function is detected. Depending on the frequency of jumps in a program, this predictor might slightly improve performances.

References

[1] Bluespec Compiler. URL: https://github.com/B-Lang-org/bsc.

[2] Andrew Waterman et al. "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA". In: May 2011. URL: https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.pdf.

[3] David Patterson and Krste Asanovic. *Instruction Sets Should Be Free: The Case For RISC-V*. Aug. 2014. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.pdf.

[4] Bluespec Inc. *Bluespec(TM) System Verilog Reference Guide*. 2008. URL: http://csg.csail.mit.edu/6.S078/6_S078_2012_www/resources/reference-guide.pdf.

[5] Connectal. URL: https://www.connectal.org/.