Multi-objective Optimisation of RISC-V CV32A6 for ML application

Bastien HUBERT, Omar HAMMAMI
Computer Science and Systems Engineering Laboratory (U2IS)
ENSTA Paris
Palaiseau, France
{bastien.hubert, omar.hammami}@ensta-paris.fr

***Abstract*—RISC-V architectures are rapidly gaining in popularity in embedded systems, in which each mW counts. Since AI-related applications such as image recognition or neural networks tend to be highly energy consuming, low-power techniques are required to optimise the autonomy of systems using SoCs to run such applications.**

However, the trade-off between energy consumption, application performance and resource use requires a multi-objective optimisation with a potentially very important number of optimisation parameters to be performed on the SoC. As they rely on heuristics that are likely to only return locally optimal solutions, empirical methods must be excluded.

To address this issue, a technology-agnostic mathematical model is introduced to represent how optimisations are applied to a SoC, and a workflow designed to perform an intelligent exhaustive exploration of the optimisation space has been developed to highlight a subset of optimal processor configurations.

Optimisation of the ARIANE/CV32A6 RISC-V processor, running a CNN propagation on a Xilinx Zynq 7020 FPGA, has shown very encouraging results using low degree configurations, and is likely to perform even better with higher degree configurations.

***Index Terms*—RISC-V, low-power design, embedded systems, multi-objective optimisation, mathematical model, floorplanning, FPGA, simulation**

CONTENTS

I. INTRODUCTION

The RISC-V architecture is becoming increasingly popular and exploited in many application areas. Predictions about the use of RISC-V architectures in AI indicate that they will be intensively exploited in the near future. The success of the recent RISC-V 2022 Spring Week [1] also reasserted the commitment of the scientific community in its modularity, making it a highly popular choice in embedded system applications.

The goal of this paper is to optimise the energy consumption, application performance and resource use of the ARIANE/CV32A6 RISC-V processor, written in SystemVerilog, on FPGA.

To achieve this, we will propose a mathematical model describing a processor configuration based on the ordered list of optimisation parameters (e.g. whether or not to activate a unit, techniques for reducing power dissipation or tool parameters) used. Combinatorics, set theory and topological considerations will be taken into account to address the high-dimensional problems raised by our model.

State of the Art low power techniques will then be presented to be used as optimisation parameters. A technology independent workflow, able to perform an intelligent exploration of the configuration space from the multi-objective energy/performance/resource trade-off point of view, will be presented and run for a CNN application on CVA6, using Xilinx Vivado as main synthesis tool. Energy simulations will be realised using the Questa Sim simulator, and testbench executions will be run on the Zybo Z7 development board, hosting the ZYNQ 7020 FPGA.

A Pareto frontier extraction will eventually be performed to highlight optimal solutions, and statistics regarding the

configuration energy consumption, performance and resource use distribution of the configurations will be briefly discussed.

II. Mathematical and Physical Foundations

A. Physical Foundations

The electric circuit depicted in Fig. 1 is the simplest representation of a real electronic component, its pMOS and nMOS networks both reduced to a single component. It is controlled by a clock of frequency f_{CK} and uses a capacitor to prevent transistor-induced voltage spikes that happen during switching. The blocking diode is used to represent leakage current.

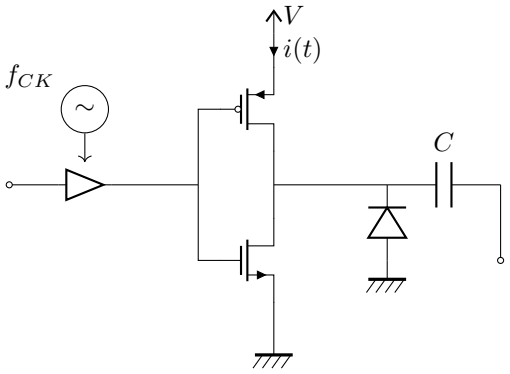

Fig. 1. A simple synchronous electric circuit

In such a circuit, [2] shows that power consumption can be divided into three different categories:

- **Leakage power** P_{leak}, which is static power, corresponds to the power induced by the current flowing through the transistors while there is no activity.
- **Internal power** or short-circuit power P_{SC}, which is a portion of dynamic power, is due to the momentary short circuit that may happen during transistors switching, if both nMOS and pMOS networks are momentarily passing at the same time.
- **Switching power** P_{sw}, which is also accounts for dynamic power, is the fraction of the total power that comes from charging and discharging the capacitors after a change of state of the transistors in the circuit.

The leakage power equation is given by the following formula:

$$P_{leak} = V \times I_{leak} \tag{1}$$

Where the formula for the leakage current I_{leak} can be given by the equations governing CMOS transistors. However, [3] shows that leakage power is usually negligible compared with dynamic power unless considering a high density device, so it will not be taken into account when trying to optimise energy consumption.

Similarly, the internal power equation is given by the following formula:

$$P_{SC} = V \times I_{SC} \tag{2}$$

While a formula for I_{SC} can also be given [4], its most important characteristic is the very short amount of time τ_{SC} during which it is generated. Because this current can only be generated by a switching activity, internal power be can approximated as existing at all time, but reduced by a coefficient of $\tau_{SC} \times f_{CK}$. In particular, $\tau_{SC} << 1/f_{CK}$, and [5] has found that the internal power only accounts to approximately 7% of total the dynamic power. This is why it also won't be considered it in our quest to reduce energy consumption.

The total power in the electric circuit thus has the following distribution:

$$P_{tot} = \underbrace{V \times I_{leak}}_{P_{static}} + \underbrace{V \times I_{SC} \times \tau_{SC} \times f_{CK} + P_{sw}}_{P_{dynamic}} \tag{3}$$

Given that the switching power accounts for most of the total power used, we need to dive into the switching power equation in order to determine which parameters can be modified to improve the energy consumption of the circuit [6].

On average, the electric circuit described in Fig. 1 changes state α times between two rising edges of the clock. α is called **activity factor**, and allows us to define a frequency f, called **effective frequency**, such that the circuit changes state on average as many times on the same period of time as if it were simply clocked by a signal of frequency f. Therefore, $f = \alpha \times f_{CK}$.

The instantaneous switching power at any time t of such a circuit is then given by the formula:

$$P_{sw,instant}(t) := i_{sw}(t) \times V \tag{4}$$

The (average) **switching power** is defined as the average of the instantaneous switching power over the time interval $[t_1,\ t_2]$:

$$P_{sw}^{t_1,\,t_2} := \frac{1}{t_2 - t_1} \int_{t_1}^{t_2} P_{sw,instant}(t)\,\mathrm{d}t \tag{5}$$

Assuming that $t_2 - t_1 >> 1/f$, side effects can be neglected in order approximate that the circuit is only driven by a clock of frequency f. At each rising edge of the clock, a charge $Q = C \times V$ is needed to charge a capacitor of capacity C, which is then discharged at each falling edge. The integral of the current flowing in the circuit between two moments t_1 and t_2 is thus given by the following formula:

$$\int_{t_1}^{t_2} i_{sw}(t)\,\mathrm{d}t = \int_{t_1}^{t_2} \frac{\mathrm{d}Q}{\mathrm{d}t}\,\mathrm{d}t = C \times V \times (t_2 - t_1) \times f \tag{6}$$

The switching power equation can now be deduced from these results:

$$\begin{aligned} P_{sw}^{t_1,\,t_2} &= \frac{1}{t_2-t_1}\int_{t_1}^{t_2} P_{sw,instant}(t)\,\mathrm{d}t \\ &= \frac{1}{t_2-t_1}\int_{t_1}^{t_2} i_{sw}(t)\times V\,\mathrm{d}t \\ &= \frac{V}{t_2-t_1}\int_{t_1}^{t_2} i_{sw}(t)\,\mathrm{d}t \\ &= \frac{V}{t_2-t_1}\times C\times V\times(t_2-t_1)\times f \\ &= C\times V^2\times f \\ P_{sw}^{t_1,\,t_2} &= C\times V^2\times\alpha\times f_{CK} = P_{sw} \end{aligned} \tag{7}$$

In particular, P_{sw} doesn't depend of t_1 and t_2, which means they can be arbitrarily chosen provided that $t_2 - t_1 >> 1/f$. Eq. 7 is a classic result that has many direct applications to reduce the energy consumption of an electric design.

Let's consider a specific couple of CPU configuration and application to be evaluated: this application is executed in n_{cycles} cycles, which translates to $T = n_{cycles}/f_{CK}$ seconds on the configuration.

Assuming that $P_{tot} \propto P_{sw}$, and considering C and V determined by the hardware technology of the circuit, while increasing (resp. decreasing) f_{CK} directly increases (resp. decreases) the switching power, the total energy $E = P_{tot}\times T$ required to run the application doesn't depend of f_{CK}.

This means that clock frequency has a linear impact on the processor performance while not impacting the resource use nor the energy consumption, making it a very easily predictable parameter. For this reason, f_{CK} will not be considered as an optimisation parameter, but rather as a technological, hardware characteristic of the system to optimise. By choosing a different technology, C, V and f_{CK} can be changed and greatly reduce switching power. However, from a technology-independent point of view, the energy consumption can only be impacted by a modification of the activity factor.

B. Mathematical Foundations

Let $N \in \mathbb{N}$ and $c \in \mathbb{N}^N$. c is called a **configuration** if:

$$\exists\, k \in \mathbb{N} \,|\, \begin{cases} \forall i \in [\![1,\,k]\!], & \exists!\, j \in [\![1,\,N]\!] : \quad c_j = i \\ \forall i > k, & \nexists\, j \in [\![1,\,N]\!] : \quad c_j = i \end{cases} \tag{8}$$

If k exists, it is unique and is called the **degree** of the configuration: $deg(c) := k$.

Any given configuration of degree k is composed of:

- exactly one of each integer between 1 and k. The indices of those integers match with the optimisation parameters used in the configuration, while the values correspond to the order in which those parameters are implemented.
- $N - k$ zeros (as c cannot have an integer greater than $N - k$). The indices of those zeros correspond to the optimisation parameters that *aren't* used in the configuration.

For example, if $N = 3$, valid configurations include: $(0,\,0,\,0)$, $(0,\,1,\,0)$, $(2,\,0,\,1)$ and $(2,\,3,\,1)$. On the other hand, $(1,\,0,\,3)$, $(2,\,2,\,1)$, and $(0,\,0,\,-1)$ are not configurations. In particular, $0_{\mathcal{C}} := (0,\,0,\,...\,0) = 0_{\mathbb{N}^N}$ is a configuration, and is called **standard configuration**.

We can thus define $\mathcal{C}$ the set of all configurations, and a family of subsets corresponding to the configurations of a certain degree:

$$\forall k \in \mathbb{N},\, \mathcal{C}_k := \{c \in \mathcal{C} \mid deg(c) = k\} \tag{9}$$

The elements of $\mathcal{C}_k$ are composed of one of each integer between 1 and k, in any order, and $N - k$ zeros. Thus:

$$\#\mathcal{C}_k = k! \times \binom{N}{N-k} = \frac{N!}{(N-k)!} \tag{10}$$

Furthermore, the Taylor-Lagrange formula applied to $f : x \in \mathbb{R} \mapsto e^x \in \mathbb{R}$ and evaluated in $x := 1$ and $a := 0$ states that:

$$\begin{aligned} & f(x) = \sum_{i=0}^{N} \frac{f^{(i)}(a)}{i!}(x-a)^i + \int_a^x \frac{f^{(N+1)}(t)}{N!}(x-t)^N\,\mathrm{d}t \\ \Rightarrow \quad & e^x = \sum_{i=0}^{N} \frac{e^a}{i!}(x-a)^i + \int_a^x \frac{e^t}{N!}(x-t)^N\,\mathrm{d}t \\ \Rightarrow \quad & e^1 = \sum_{i=0}^{N} \frac{e^0}{i!}1^i + \int_0^1 \frac{e^t}{N!}(1-t)^N\,\mathrm{d}t \\ \Rightarrow \quad & e = \sum_{i=0}^{N} \frac{1}{i!} + \int_0^1 \frac{(1-t)^N}{N!}e^t\,\mathrm{d}t \end{aligned} \tag{11}$$

Given that $\mathcal{C} = \bigsqcup_{k=0}^{N} \mathcal{C}_k$, the formula can be used with $i = N - k$ to give a simple equivalent of $\#\mathcal{C}$:

$$\begin{aligned} \#\mathcal{C} &= \sum_{k=0}^{N} \#\mathcal{C}_k \\ &= \sum_{k=0}^{N} \frac{N!}{(N-k)!} \\ &= \sum_{i=0}^{N} \frac{N!}{i!} \\ &= N! \times \left(e - \int_0^1 \frac{(1-t)^N}{N!}e^t\,\mathrm{d}t\right) \end{aligned}$$

$$= N!\, e - \underbrace{\int_0^1 \overbrace{(1-t)^N e^t}^{\leq 1,\ \forall t \in [0,\,1]}\, \mathrm{d}t}_{\leq 1}$$

$$\#\mathcal{C} \underset{N \to \infty}{\sim} N!\, e \tag{12}$$

The fact that the exploration space grows as a factorial (the approximation in Eq. 12 is precise to 10^{-8} as soon as $N \geq 10$) leads to a **combinatorial explosion** even for relatively small values of N. For instance, $\#\mathcal{C} \approx 10^7$ for $N = 10$.

Given that $\mathcal{C} \subset \mathbb{N}^N \subset \mathbb{R}^N$, can define the **canonical distance**, or genotypical distance, making $(\mathcal{C},\, d)$ a metric space:

$$d : \begin{cases} \mathcal{C} \times \mathcal{C} & \to & \mathbb{R}_+ \\ (c,\, c') & \mapsto & \sqrt{\sum_{k=1}^{N} (c_k - c'_k)^2} \end{cases} \tag{13}$$

However, as N increases, d loses its "discrimination ability", meaning that it is increasingly difficult to distinguish pairs of elements using only d. This means that navigating through $\mathcal{C}$, let alone extracting a Pareto surface to isolate Pareto-equivalent configurations in the E/P/R trade-off, is mathematically difficult.

This phenomenon is referred as the **curse of dimensionality**, and is a common issue when dealing with high-dimensional metric spaces. One way to illustrate this curse of dimensionality is to consider M outcomes $(v_1,\, v_2,\, ...v_M)$ of a continuous uniform distribution on $[0,\, 1]^N$, and plot the discrimination ratio R as a function of N, as shown by [7] in Fig. 2.

$$R := \frac{max_{(i,\,j) \in [\![1,\,M]\!]^2,\, i \neq j} \{d(v_i,\, v_j)\}}{min_{(i,\,j) \in [\![1,\,M]\!]^2,\, i \neq j} \{d(v_i,\, v_j)\}} \xrightarrow[N \to \infty]{} 1 \tag{14}$$

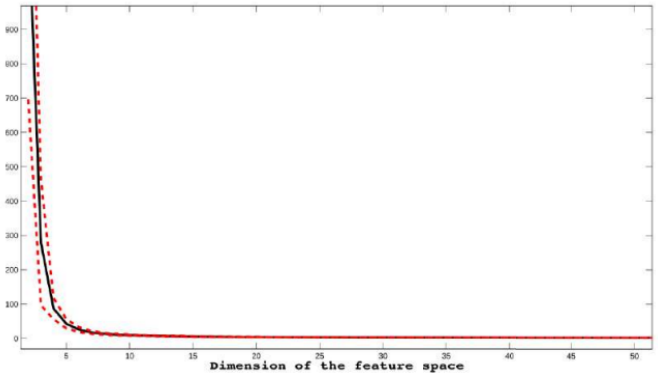

Fig. 2. R as a function of N (M = 500)

To solve this issue, we need to work in a smaller topological space, called **effective space**. This space, noted $\mathcal{EPR} := {\mathbb{R}_+}^3$, corresponds to the 3D plot of the standardised energy consumption, performance and resource use of each configuration. In order to represent a configuration in this space, we need to define a **fitness function** f:

$$f : \begin{cases} \mathcal{C} & \to & \mathcal{EPR} \\ c & \mapsto & (\, E^0(c),\, P^0(c),\, R^0(c)\,) \end{cases} \tag{15}$$

Where E^0, P^0 and R^0 are the **standardised energy, performance and resource functions** respectively. By noting E, P and R the functions used to evaluate the corresponding metrics for a configuration, the standardised functions can be defined as such:

$$E^0,\, P^0,\, R^0 : \begin{cases} \mathcal{C} & \to & \mathbb{R}_+ \\ c & \mapsto & \frac{E(0_\mathcal{C})}{E(c)},\, \frac{P(c)}{P(0_\mathcal{C})},\, \frac{R(0_\mathcal{C})}{R(c)} \end{cases} \tag{16}$$

Since we want to maximise performance while minimising energy consumption and resource use, the standardised functions are designed so as to be maximising ratios. This is meant to make plots in the $\mathcal{EPR}$ space and the Pareto surface easier to read while having no impact on the mathematical meaning.

Using f, new function called **effective distance** or phenotypical pseudometric can be defined as follows:

$$d_{eff} : \begin{cases} \mathcal{EPR} \times \mathcal{EPR} & \to & \mathbb{R}_+ \\ (x,\, y) & \mapsto & \sqrt{\sum_{k=1}^{3} (x_k - y_k)^2} \end{cases} \tag{17}$$

Given that f is not injective, these functions only grant pseudometric proprieties to $(\mathcal{C},\, (f,\, f) \circ d_{eff})$, as two different configurations may have the same standardised energy consumption, performance and resource use. Such points are indistinguishable in the effective space. Because f is used before applying d_{eff}, only three coordinates are used when calculating the distance, effectively eliminating the curse of dimensionality.

However, it is not the only problem raised by the combinatorial explosion of Eq. 12. Since N can be virtually as big as we want, it will be computationally extremely long to explore the entire space. To give an order of magnitude, it takes 3h to evaluate the fitness of a single configuration using the experimental setup of section V-E, so a complete evaluation of $\mathcal{C}$ roughly takes to 3500 years of computation time for just 10 optimisation parameters. To avoid exhaustive exploration, techniques to reduce the exploration space have been used.

The simplest way to reduce the cardinal of a set without it losing its topological properties is to define a **binary equivalence relation** between the elements of the set, and to consider the space quotiented by the equivalence relation, i.e.

only take into account one element of every equivalence class.

In the case of $\mathcal{C}$, a family of binary equivalence relations indexed by $\mathcal{P}(\llbracket 1,\ N \rrbracket)$ can be defined as follows:

$$\begin{gathered}\forall L \in \mathcal{P}(\llbracket 1,\ N \rrbracket),\ \forall\ (c,\ c') \in \mathcal{C}^2, \\ c\ \mathcal{R}_L\ c' \overset{def}{\iff} \forall\ i \in \llbracket 1,\ N \rrbracket \backslash L,\ c_i = c'_i\end{gathered} \tag{18}$$

Let $L \in \mathcal{P}(\mathcal{C})$, the relation defined in Eq. 18 is called ***L*-distinction**, and defines a level of difference between two configurations. Given a configuration c:

- any parameter whose index is *not* in L used in c will also be used by every L-distinct configuration
- any unused parameter whose index is *not* in L will won't be used by any L-distinct configuration

The equivalence class of c is noted $[c]_L$, and the set of all equivalence classes is noted $\mathcal{C}/\mathcal{R}_L := \{[c]_L,\ c \in \mathcal{C}\}$. In particular, $[c]_\emptyset = \{c\}$ and $[c]_{\llbracket 1,\ N \rrbracket} = \mathcal{C}$.

Unfortunately, the elements of a L-distinct class do not share enough proprieties to consider only evaluating a representative of each equivalence class instead of the entire configuration space. For instance, L-distinct configurations can have a different degree: $(1,\ 0,\ 0)\ \mathcal{R}_{\{2\}} (1,\ 2,\ 0)$, but $deg((1,\ 0,\ 0)) = 1$ and $deg((1,\ 2,\ 0)) = 2$. A new, degree-invariant subset of $[c]_L$ can be defined as such:

$$[c]^0_L\ := \{c' \in [c]_L \mid \forall\ i \in L,\ c'_i \neq 0\} \tag{19}$$

An element of $[c]^0_L$ is called a ***L*-variation** of c. It should be noted that c is not always a L-variation of itself, as it is the case for $(1,\ 0,\ 0)$ and $L = \{2\}$. However, $[c]^0_L$ cannot be an empty set because it is always possible to replace an arbitrary number of zeros in c by $deg(c) + 1$, $deg(c) + 2$ and so on.

Let's consider $c' \in [c]^0_L$. Because the values corresponding to the indices of L cannot be zeros, potential zeros of c' have to be indexed by elements outside of L. Since $c\ \mathcal{R}_L\ c'$, c has exactly the same values for those indices, which means they have the same number of zeros, and thus the same degree.

This also means that, given that the zeros of every L-variation of c are at the same positions, a permutation of $\llbracket 1,\ N \rrbracket$ can be applied to the indices of every element of $[c]^0_L$ to move the zeros to the last $N - deg(c)$ indices. The tuples extracted by only considering the first $deg(c)$ indices of each L-variations are thus permutations of $\llbracket 1,\ deg(c) \rrbracket$. Since these configurations are L-distinct of c, another permutation can be applied to the tuples in order to move the values corresponding to the indices of L to the last indices.

Hence, L-variations can only differ from each other by a permutation of the values indexed by L. In particular, L-variations of $0_\mathcal{C}$ correspond to every configurations using exactly the parameters whose indexes are listed in L, and their degree is $\#L$. The two following relations can be deduced from these considerations:

$$\forall k \in \llbracket 1,\ N \rrbracket,\ \mathcal{C}_k = \bigsqcup_{L \in \{\mathcal{P}(\llbracket 1,\ N \rrbracket)\ \mid\ \#L = k\}} [0_\mathcal{C}]^0_L \tag{20}$$

$$\mathcal{C} = \bigsqcup_{L \in \mathcal{P}(\llbracket 1,\ N \rrbracket)} [0_\mathcal{C}]^0_L \tag{21}$$

It is worth noting that using a representative of $[0_\mathcal{C}]^0_L$ for each $L \in \mathcal{P}(\llbracket 1,\ N \rrbracket)$, as suggested by Eq. 21, is equivalent to considering that parameter order has no impact on the fitness of a configuration, effectively reducing the number of configurations to evaluate from $N!\ e$ to 2^N.

Even in the general case where the order in which parameters are implemented has an impact on the fitness of a configuration, Eq. 20 can be used to create **meta parameters**. A meta parameter correspond to the ordered tuple of used parameters (i.e. indices of strictly positive values) in a representative of $[c]^0_L$ for a given L. Two simple methods can be used to determine which meta parameters must be taken into account when reducing the cardinal of the exploration space: the ***ε*-clustering** and the **n-clustering**.

These two techniques consist in evaluating, for a given value of $k \in \llbracket 1,\ N \rrbracket$, the maximum effective distance between two elements of $[0_\mathcal{C}]^0_L$, called **configuration diameter**, for every $L \in \{l \in \mathcal{P}(\llbracket 1,\ N \rrbracket) \mid \#l = k\}$:

$$D(L) := \max_{(c,\ c') \in \left([0_\mathcal{C}]^0_L\right)^2} \{d_{eff}(f(c), f(c'))\} \tag{22}$$

In the case of ε-clustering, if there is L_0 that verifies $D(L_0) < \varepsilon$, the parameters corresponding to L_0 will be removed from the list of all parameters, and replaced by the corresponding meta parameter. For example, if the studied parameters were $(a,\ b,\ c)$ and $L_0 = \{1,\ 3\}$, a meta parameter can be $(c,\ a)$, and the new list of parameters will be $(b,\ (c,\ a))$. This leads to a diminution of N and the total number of configurations: a new configuration space $\widetilde{\mathcal{C}}$ is created, and the ε-clustering can be performed again while it is possible to find a L_0.

The n-clustering iterates on the exploration space exactly as the ε-clustering, but instead of stopping when every cluster has a large diameter, it stops after n iterations. For the sake of clarity, $\widetilde{\mathcal{C}}$ will denote the exploration space at the end of the final clustering iteration.

Now that the number of parameters can be reduced at will, exhaustive exploration of $\widetilde{\mathcal{C}}$ can be realised, and evaluated configurations can be represented in the $\mathcal{EPR}$ space. But the question of selecting the best configuration for a given use

case remains.

Given that there are no obvious linear relations between energy consumption, performance and resource use, we cannot try to optimise a single function of the form:

$$J : \left\{ \begin{array}{ccc} \mathcal{C} & \rightarrow & \mathbb{R}_+ \\ c & \mapsto & \alpha E^0(c) + \beta P^0(c) + \gamma R^0(c) \end{array} \right. \quad (23)$$

This means that mutlicriteria optimisation tools must be used in order to extract the best configuration(s) from the $\mathcal{EPR}$ space. In this paper, we will use the Pareto frontier extraction method to highlight configurations c such that there is no other configurations that has a better value for *every* metric. Mathematically, a partial order relation, called **domination**, can be defined between a 3D point x and another y as follows:

$$x \succeq y \overset{def}{\iff} \forall k \in [\![1,\, 3]\!],\, x_k \geq y_k \quad (24)$$

With this definition, a **Pareto-efficient** configuration is a configuration whose image by f is not dominated by that of any other configuration:

$$\nexists\, c' \in \mathcal{C} \backslash \{c\} \mid f(c') \succeq f(c) \quad (25)$$

The set of all Pareto-efficient configurations is called the **Pareto frontier** of the space, and is noted $\mathbf{P}(\mathcal{C}) \subset \mathcal{C}$. Fig. 3 shows a 2D example of a Pareto frontier, using Python 3.10 [8].

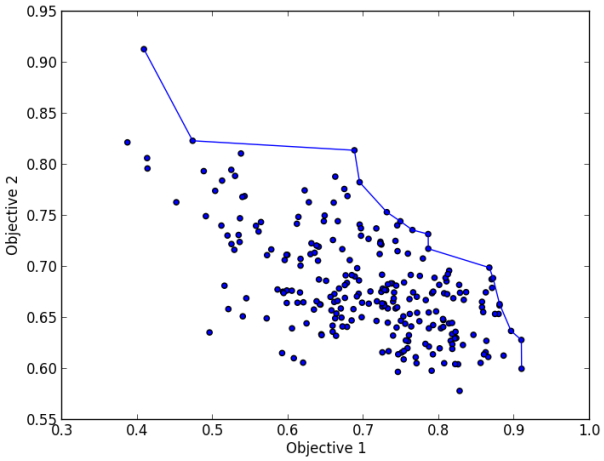

Fig. 3. A 2D Pareto frontier and the space it has been extracted from

III. State of the Art

As detailed in section II-A, the most important factor that must be taken into account when trying to reduce energy consumption is the activity factor α. In particular, Eq. 7 proves that α has a linear impact on switching power. Since P_{leak} and P_{SC} have both been neglected, we can approximate that there is a linear relation between the total energy used by the circuit and its activity factor.

The literature about low power design is very extensive, but some techniques are particularly common when optimising the power consumption of SoCs. We have tried to isolate the ones we thought were the most significant while staying as technology independent as possible.

The most commonly used low-power technique to reduce α is called **clock gating**, and consists in adding a logical AND gate before the clock input of sub-circuit. Fig. 4 shows an example of a clock gated data flip-flop (DFF).

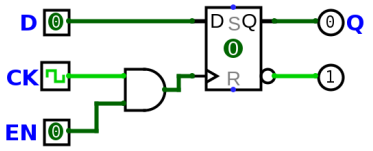

Fig. 4. Diagram of a clock gated DFF

While a signal EN, called an **enable**, is low, the clock signal CK will not be propagated through the clock gated sub-circuit, preventing transistors to switch, and thus only dissipating static power. This effectively contributes to reducing the activity factor. Considering that the circuit initially had an activity factor of 1, as is the case of a non clock gated DFF, the power gain G of using clock gating is:

$$G := 1 - \left(\alpha \times \frac{P_{sw}}{P_{tot}} + \frac{P_{tot} - P_{sw}}{P_{tot}} \right) = (1 - \alpha) \times \frac{P_{sw}}{P_{tot}} \quad (26)$$

According to [9], switching power accounts for 70% to 90% of the total dissipated power, so the energy saved by clock gating represents at least $0.7 \times (1 - \alpha)$ of the total energy initial dissipated.

However, clock gated circuits are still connected to a power source, meaning that they are still a source of energy dissipation through transistor leakage current. Though we stated that leakage power is negligible compared with dynamic power in most cases, we want to be as large in our description of the state of the art and include a low power technique, called **power gating** [10] [11], and that can be applied to high-density electric circuits to reduce their leakage power.

Similarly to clock gating, the idea to power gating is to disconnect the power supply to any sub-circuit that isn't active at a given time. Fig. 5 pictures an example of a power gated sub-circuit. Because the isolated sub-circuit will only be powered when necessary, additional combinatorial logic must be inserted before each signal leaves the sub-circuit.

However, the power gating technique comes with a drawback: delays are created between powered (active) and

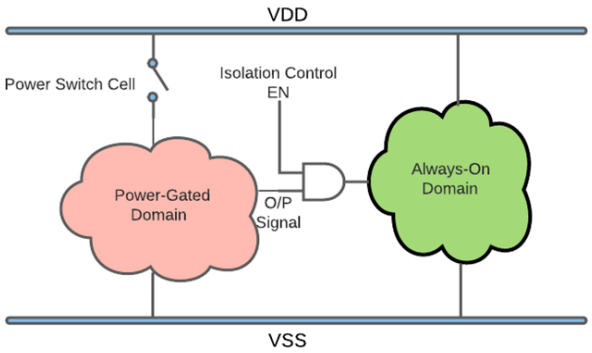

Fig. 5. A power gated sub-circuit and a non power gated sub-circuit coexisting

non powered (sleep) states due to charge and discharge of capacitors. These delays are called falling-asleep and **waking-up** delays. While falling-asleep delay doesn't impact the logic of the sub-circuit, waking-up delay means that the signal has to "wait" for the sub-circuit to reach its active voltage before being propagated.

Moreover, the state of the sub-circuit itself is lost during sleep mode. To prevent this, **retention power gating**, consisting in saving the value of certain sub-circuit signals in a powered area, can be implemented. The necessary additional routing is a trade-off to reducing waking-up delays, and is to be used only in critical-timing scenarios.

A less manichean version of clock and power gating also exist under the term of **frequency and voltage islands**. Certain high performance sub-circuits with long propagation times may require to operate at a higher frequency to reduce the global critical path. Because there is a relation between the operating voltage and the maximum frequency at which a circuit can be clocked, such units need to be "overvolted". On the other hand, sub-circuits that have short propagation times may be idling while waiting for longer sub-circuits and can be clocked at a lower frequency, allowing to save energy by "undervolting" them.

It must be noted that creating frequency and voltage islands in an electronic circuit do not contradict what was said about the *global* clock frequency and operating voltage, as it merely creates new clocks and power sources at a determined fraction of the their parents: changing the main clock frequency will also result in the same frequency change for its children, and should be interpreted as a modification of the activity instead. Only by decreasing (resp. increasing) the frequency and voltage of a specific portion of the circuit can the fraction of time during which it (resp. the rest of the circuit) idles be minimised.

Eventually, the dynamic counterpart of frequency and voltage islands is called **dynamic voltage and frequency scaling** (DVFS) [12]. Instead of always operating at specific fractions of f_{CK} and V, a **dynamic power management** (DPM) monitors the logic propagation throughout the entire circuit, and changes the values of region-specific clock frequency and operating voltage accordingly. Fig. 6 [2] gives an example of dynamic frequency and voltage scaling: to prevent a sub-circuit from idling until a delay D is over, the DPM changes the distribution of the **workload** $W := V \times T$ by reducing the frequency so that $T = D$. This way, V can be reduced by a factor of $\frac{W/V}{D}$.

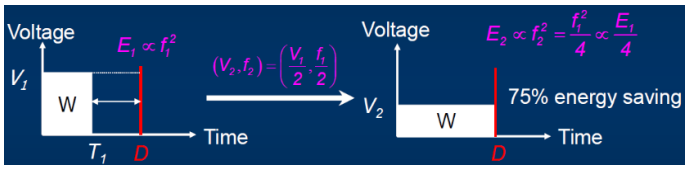

Fig. 6. Energy can be saved by dynamically reducing sub-clock frequency

IV. Methodology

The complete workflow used to in this paper to evaluate Pareto-efficient configurations is described in Fig. 7, and Fig. 8 gives an overview of the software for the different stages. It consists of three main stages: generating the exploration space $\mathcal{C}$, evaluating the values of $E(c)$, $P(c)$ and $R(c)$ for each configuration during the **fitness evaluation** process, and extracting the Pareto frontier from $\mathcal{EPR}$.

Before trying to extract the Pareto frontier to determine which configurations are best suited for different scenarios, we have to define the exploration space $\mathcal{C}$ by choosing which optimisation parameters will taken into account. This can be achieved by three different ways:

- An application profiling of our testbench can be performed using the standard configuration to see which functions or methods were the most time or energy consuming, and determine the impact of different processor components on application performance
- Low power techniques, elaborated based on our mathematical and physical understanding of the couple processor/application, can then be applied to all or some components of the processor
- We can finally choose to include some of the built-in options provided by the tools used to generate a processor configuration

Having defined the list of optimisation parameters, and thus $\mathcal{C}$, we may reduce its size using the ε-clustering or n-clustering methods. We can then begin to run fitness evaluation iterations for every configurations $c \in \widetilde{\mathcal{C}}$.

The first step to obtain those values is to alter the code of the standard configuration, which is composed of a main structure and an instruction set architecture, to introduce

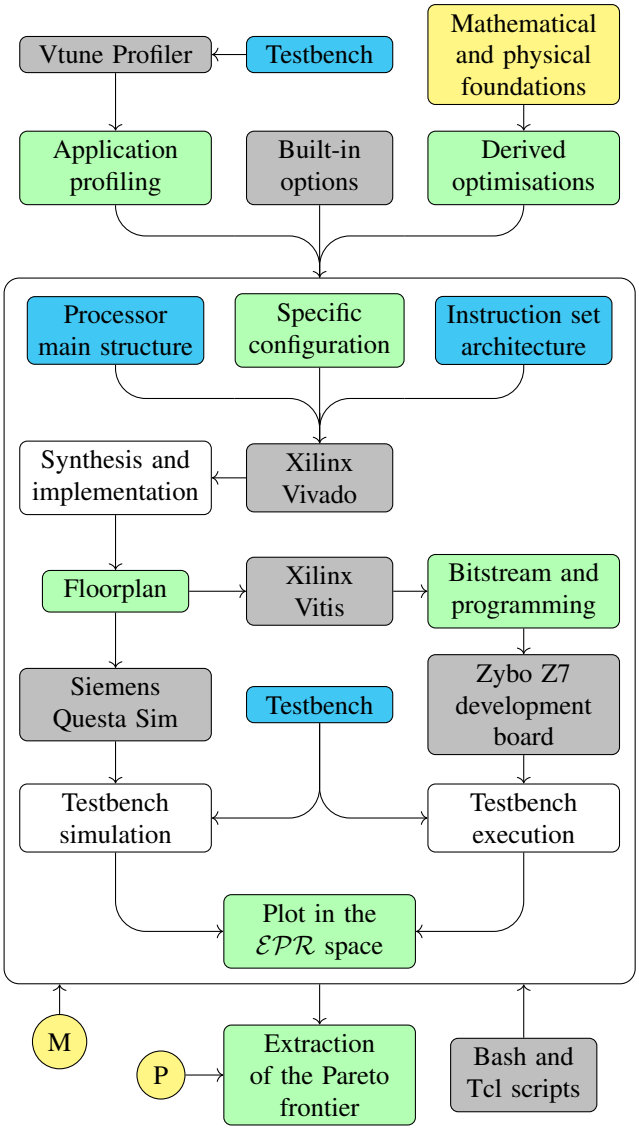

Fig. 7. The complete workflow with its fitness evaluation component ($\mathrm{M} = \#\mathcal{C} \approx N!\, e$, $\mathrm{P} = \#\mathbf{P}(\mathcal{C})$)

RTL analysis	Xilinx Vivado
Behavioural simulation	Xilinx Vivado
Synthesis	Xilinx Vivado
Implementation	Xilinx Vivado
Bitsream generation	Xilinx Vivado
Testbench execution	Xilinx Vitis
Testbench simulation	Siemens Questa Sim

Fig. 8. The software used to perform each stage of the workflow

the variations corresponding to the parameters used in this specific configuration in the correct order.

This gives a set of files written using a hardware description language (HDL) that can be interpreted by a RTL analysis tool as a **register-transfer level (RTL) design**, which is a sequential and combinatorial logic model of a HDL code. It converts code into abstract logic gates (for the combinatorial part) and registers (for the sequential part) that can be displayed as shown in Fig. 9.

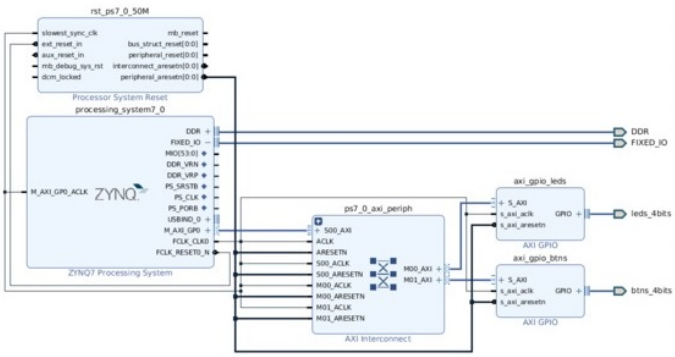

Fig. 9. A RTL design displayed using Xilinx Vivado 2020.1

After having generated the RTL design of the configuration, a **behavioural simulation** may be performed to ensure the design behaves the way it was intended to. In practice, as this adds computation time to an already long iteration, behavioural simulation was only performed on the standard configuration. This step is purely for verification purposes as the testbench application will be executed at the end of the iteration: it only saves computation time in case the application didn't run correctly on the specific configuration.

The following step is **synthesis**, which is the act of transforming the RTL design into a **netlist**. A netlist is a description of the electronic connections in a circuit, represented as an oriented graph where the nodes are the electronic components of the circuit. At this point, the components are still abstract, meaning that there can be as many as wanted and that there is no physical, geometrical constraint between them.

In order to be one step closer to the physical disposition of the components, the netlist must go through **placement and routing**, or implementation. During this step, the place and route (P&R) tool will try to match the abstract gates and registers with actual circuit components. The placement and routing process can be greatly customised to perform multiple optimisations in the geometrical distribution of the components. Increasing component density is the best way to reduce the value of $R(c)$, so configurations with a high amount of P&R-related parameters will usually have much better resource use than others. Moreover, these parameters

are amongst the most sensitive to the order in which they are chosen, so configurations using the same placement and routing optimisations in different order may result in vastly different FPGA circuits. The P&R workflow, shown in Fig. 10 [13], is quite complex so we won't dive into how it works, but rather use it as a highly customisable tool.

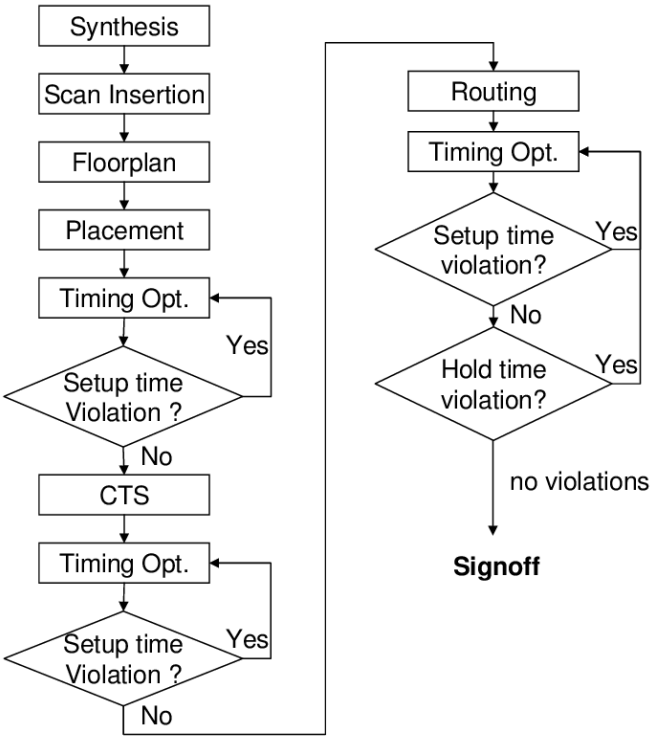

Fig. 10. The Placement and Routing workflow

After having synthesised and implemented the configuration, a post placement and routing **floorplan** has been generated. It is the physical mapping of the electronic components that make up the configuration's circuit. To draw a parallel with genetic algorithms, a floorplan is the phenotype expressed by a genotype: the configuration. In this case, the fitness function has been defined in Eq. 15, while the selection function corresponds to the extraction of a Pareto frontier. An example of floorplan using Vivado is shown in Fig. 11.

Now that we have obtained a floorplan from a configuration, we need to calculate $f(c)$. While computing the resource use of the configuration can be achieved by counting the number of logic cells used on the floorplan, the same cannot be said for the performance and the energy consumption: we need run the testbench on the processor to determine their values. From here, there are two options, each with its own advantages and disadvantages: **execution** and **simulation**.

On the one hand, executing the testbench on a FPGA programmed with the studied configuration gives the execution

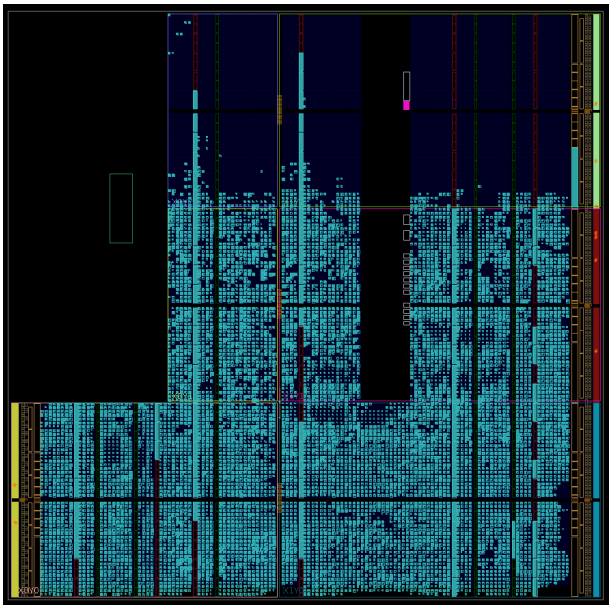

Fig. 11. A post-P&R floorplan displayed using Vivado 2020.1

time with high accuracy, because timestamps can easily be added to the source code of the testbench. The computation time is given by subtracting the timestamp of the end with that of the beginning of the execution. On the other hand, the physical nature of a FPGA-based processor makes it difficult to accurately measure its energy consumption. According to microelectronics research engineer F. Dulucq [14], the lack of precision in measurements is usually so important that the variations in energy consumption obtained are most often within the uncertainty range. Moreover, while the amount of time required to program a FPGA and execute the testbench is relatively small compared with synthesis and implementation, communication with the electronic board makes it difficult to automate and parallelise.

Regarding simulation, the major issue is computation time: simulating only 60ms of processor activity takes approximately 40 to 60 minutes using the setup of described in section V-E, and can take much longer if more accuracy is needed. This downside is compensated by the high precision of the energy consumption given by the simulation tool, and the fact that it can easily be automated using Tcl scripts. Because Tcl scripts can also be used to automate synthesis and implementation, little work is required to include simulation in the workflow without causing any software conflict. However, while $E(c)$ can be measured with high accuracy, the level of precision necessary for a precise estimation of $P(c)$ would drastically increase the duration of the simulation beyond reason.

The advantages and disadvantages of these two methods are summarised in Fig. 12.

	Execution	Simulation
Duration	low	high
Precision of $P(c)$	high	none
Precision of $E(c)$	low	high
Automation	difficult	easy

Fig. 12. Comparison between execution and simulation

Because the qualities of one compensate for the defects of the other, both execution and simulation are performed on the implemented design in order to compute the most accurate metrics without increasing the total computation time per iteration too much.

Before being capable of executing the testbench application, the design need to be run through a bitstream generation software to obtain the **bitstream** necessary to program the FPGA with the configuration of the processor. A bitstream is the sequence of data describing the physical setup of an electronic circuit required to configure the programmable logic (PL) and its interactions with the processing system (PS) of the FPGA. An integrated development environment must then compile the testbench using a compiler to the instruction set architecture used by the process and load the bitstream and the executable file into the FPGA. Fig. 13 shows some tabs of such an IDE.

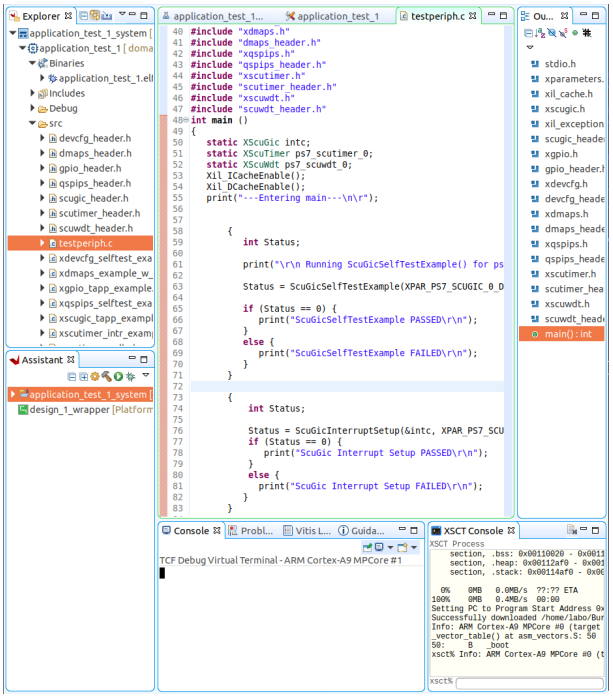

Fig. 13. A testbench application in the 2020.1 Vitis IDE

Eventually, the IDE needs to maintain communication with the FPGA in order to send the signal to run the testbench and display the received output on a console terminal. The console output corresponding to the application shown in Fig. 13 is presented in Fig. 14.

```
---Entering main---

 Running ScuGicSelfTestExample() for ps7_scugic_0...
ScuGicSelfTestExample PASSED
ScuGic Interrupt Setup PASSED

Running GpioInputExample() for axi_gpio_btns...
GpioInputExample PASSED. Read data:0x0

Running GpioOutputExample() for axi_gpio_leds...
GpioOutputExample PASSED.

 Running DcfgSelfTestExample() for ps7_dev_cfg_0...
DcfgSelfTestExample PASSED

 Running XDmaPs_Example_W_Intr() for ps7_dma_s...
Test round 0
XDmaPs_Example_W_Intr PASSED

 Running QspiSelfTestExample() for ps7_qspi_0...
QspiPsSelfTestExample PASSED

 Running ScuTimerPolledExample() for ps7_scutimer_0...
ScuTimerPolledExample PASSED

 Running Interrupt Test  for ps7_scutimer_0...
ScuTimerIntrExample PASSED

 Running Interrupt Test  for ps7_scuwdt_0...
ScuWdtIntrExample PASSED
---Exiting main---
```

Fig. 14. The corresponding output in a terminal

In parallel with the execution branch calculating $P(c)$, the simulation branch computes the value of $E(c)$ using a post placement and routing simulation software. Such a software is able simulate a processor running an application at logic gates level, and can display binary signals as a function of time in a diagram, as shown in Fig. 15. Monitoring specific signals, such as pipelined contol signals or flag registers, can be useful to ensure that the simulation runs smoothly, or to facilitate debugging if it does not.

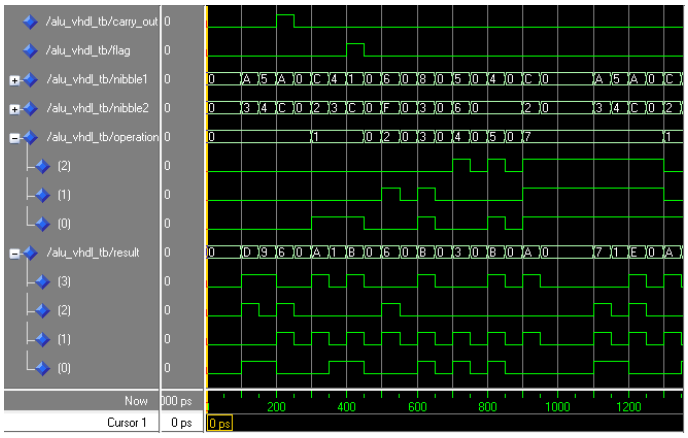

Fig. 15. A waveform generated by Questa Sim 10.7

We have obtained the values of $E(c)$, $P(c)$ and $R(c)$ during the simulation, execution and implementation steps of this iteration respectively, $f(c)$ can now be plotted in the $\mathcal{EPR}$ space to conclude the fitness evaluation of c.

After having evaluated every configuration in $\widetilde{\mathcal{C}}$, the Pareto frontier can finally be extracted from the $\mathcal{EPR}$ space. Based on the physical constraints of the studied embedded system and taking into account the desired ratio in the E/P/R trade-off, we can eventually choose the Pareto-efficient configuration(s) that best suits the use case scenario.

V. Experimental Setup

A. The RISC-V Architecture

The RISC-V architecture [15] [16] [17] is an open source **reduced instruction set computer** (RISC) architecture (ISA) originating from the Berkeley University of California. It was first developed in 2010 by Prof. Krste Asanović and graduate students Yunsup Lee and Andrew Waterman as part of a five year project in computing science, and has since grown into a thriving community: in 2015, the RISC-V Foundation was created with over 100 members and a board consisting of companies like Google, Nvidia, Western Digital, NXP Semiconductors, Microsemi, and Bluespec, as well as representatives from UC Berkeley. Since then, companies such as AMD, Qualcomm, and IBM have joined. The Foundation is a non-profit association which organises the development and adoption of RISC-V by freely publishing its intellectual property. Relocated to Switzerland in 2019, the Foundation has become RISC-V International [18] in 2020 with more than 2000 members across the world, and permits unrestricted use of the ISA for design of both software and hardware since then.

One of the strengths of the RISC-V instruction set is its modularity, offering four base ISA, shown in Fig. 16, and multiple independent extensions, a non-exhaustive list of which can be found in Fig. 17.

RV32I	Base Integer Instruction Set, 32-bit
RV32E	Base Integer Instruction Set, (embedded), 32-bit, 16 registers
RV64I	Base Integer Instruction Set, 64-bit
RV128I	Base Integer Instruction Set, 128-bit

Fig. 16. Base RISC-V instruction set architectures

The instructions defined by the RISC-V ISA are **little-endian**, meaning that the least significant byte is stored at that smallest address. There are four base instruction format: register (R), immediate (I), store (S) and upper immediate (U), but two additional formats, branch (B) and jump (J), have been added as variations of S and U formats respectively to handle immediates. Fig. 18 describes the field distribution for each instruction format.

CV32A6, described in section V-B and which is the processor studied in this paper, is a 32-bit RISC-V processor, so we will only consider the RV32I base instruction set

M	Integer Multiplication and Division
A	Atomic Instructions
F	Single-Precision Floating-Point (32 bits)
D	Double-Precision Floating-Point (64 bits)
Zicsr	Control and Status Register
Zifencei	Instruction-Fetch Fence
Q	Quad-Precision Floating-Point (128 bits)
L	Decimal Floating-Point
C	Compressed Instructions
B	Bit Manipulation
V	Vector Operations
M	Integer Multiplication and Division

Fig. 17. RISC-V extensions

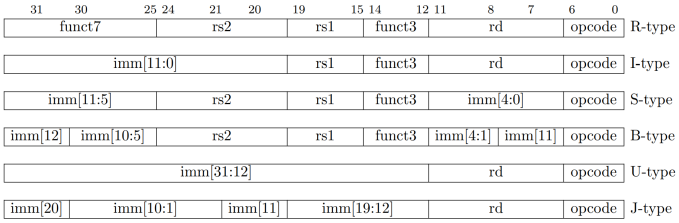

Fig. 18. The six RISC-V instruction formats

architecture. For the same reason, since CV32A6 currently only supports M, A and C extensions, only IMAC instructions will be studied in this section, summarised in Fig. 19.

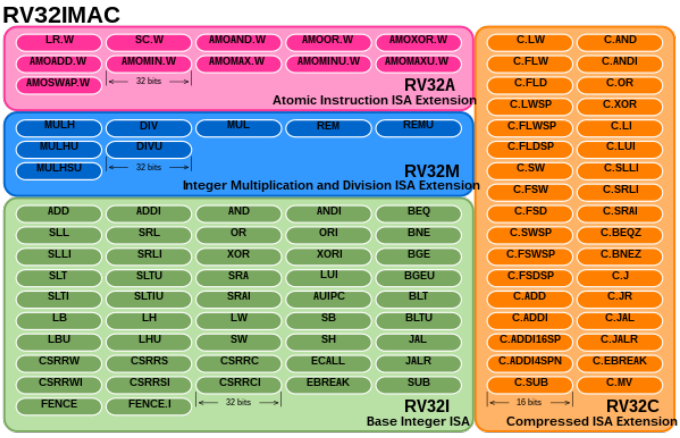

Fig. 19. The 100 IMAC instructions, sorted by extension

These instructions can be grouped as follows:

- integer computational instructions:
  - add, addi, sub: addition and subtraction
  - sll, slli, srl, srli, sra, srai: logical and arithmetic bitwise shifts
  - and, andi, or, ori, xor, xori: logical bitwise operations
  - slt, slti, sltu, sltui: set if less than instructions
  - lui: loads the upper immediate of the destination register with a 20-bit immediate value

  - auipc: adds the immediate value to the 20 high bits of the PC
- flow control instructions:
  - beq, bne, blt, bltu, bge, bgeu: conditional branching
  - jal, jalr: unconditional branching
- memory access instructions:
  - lb, lbu, lh, lhu, lw: loading from memory instructions
  - sb, sh, sw: storage to memory instructions
  - fence: forces memory access scheduling and ensures cache consistency
  - fence.i: synchronises writes to instruction memory and instruction fetches
- system instructions:
  - ecall: system call
  - break: adds a breaking point
  - csr{rw, rwi, rc, rci, rs, rsi}: CSR read and write instructions
- integer multiplication and division instructions:
  - mul, mulh, mulhu, mulhsu, div, divu, rem, remu: multiplication and division
- atomic instructions:
  - lr.w, sc.w: load-reserved and store-conditional
  - amo{swap, add, and, or, xor, max, maxu, min, minu}.w: atomic memory operations
- compressed instructions
  - c.*: 16-bit instructions

Along with its instruction set, the RV32IMAC ISA also specifies 32 registers, listed in Fig. 20. It must be noted that **control and status registers** (CSR) are defined in the RISC-V architecture, but that user-level state instructions can only access timers, counters and floating-point management CSRs, while the other are manipulated by privileged code.

0	Zero	Always zero
1	ra	Return address
2	sp	Stack pointer
3	gp	Global pointer
4	tp	Thread pointer
5	t0	Temporary / alternate return address
6-7	t1-t2	Temporary
8	s0/fp	Saved register / frame pointer
9	s1	Saved register
10-11	a0-a1	Function argument / return value
12-17	a2-a7	Function argument
18-27	s2-s11	Saved register
28-31	t3-t6	Temporary

Fig. 20. The 32 RV32IMAC registers

B. The CVA6 Processor

CVA6 [19] [20] is an open source processor implementing the RISC-V instruction set architecture and written in SystemVerilog HDL. It is the industrial evolution of ARIANE [21], a CPU developed by ETH Zürich and the University of Bologna, and is maintained by the OpenHW Group and Thales Group. Two versions of the processor are currently available: CVA6, implementing a 64-bit RISC-V core, and its CV32A6, a variant implementing a 32-bit RISC-V core. Both versions are fully compatible with I, M, A and C extensions.

CVA6 is a 6-stage, single order CPU, meaning that each of the 6 stages of the **pipeline** only processes one instruction at any time. Fig. 21 gives an overview of the CVA6 pipeline, which is differs slightly from the usual 5-stage pipeline, described in [22], by the presence of an additional frontend stage before the instruction fetch.

The first stage of the frontend is the **PC generation**, which is responsible for generating the next program counter (PC) thanks to its **branch predict unit**. This unit tries to predict the result conditional branching instructions and whether it will lead to a jump in the instruction flow. It is composed of a branch table buffer (BTB), a branch history table (BHT), a return address stack (RAS). Since CVA6 is compatible with RISC-V C extension, the branch unit can also predict branches on a 16-bit instruction.

A BTB is a hash table capable of detecting previously encountered branch instruction before the instruction fetch stage and knowing the branch target address if applicable. If the current PC maps with a table entry, the BTB knows that the next instruction to be fetched is a branch instruction, and the branch unit can then try to predict the result while still before the instruction fetch stage.

A BHT is based on the premise that branches usually behave as they have recently done. It consists of an array of small saturating counters being incremented whenever a specific branch has been taken, and decremented when it hasn't. If the value of a counter is high, meaning that the corresponding branch was taken most of the time in recent history, the BHT predicts that the branch will be taken again. On the other hand, if the counter has a low value, the branch probably won't be taken.

A RAS is a unit that pushes the PC of the next instruction on a specific stack when the PC corresponding to call instruction is read. When the PC next encounters a return instruction, the stored PC is popped and the RAS predicts that the return address is the this PC.

The second stage of the frontend is the **Instruction Fetch** (IF), whose role is to request the instruction corresponding to the PC address transmitted by the previous stage to the **instruction cache** I$ and push it in the **instruction queue**. A **memory management unit** (MMU) can be enabled from the CVA6 source code to ensure memory protection between I$ and the instruction queue. The queue has one write input and

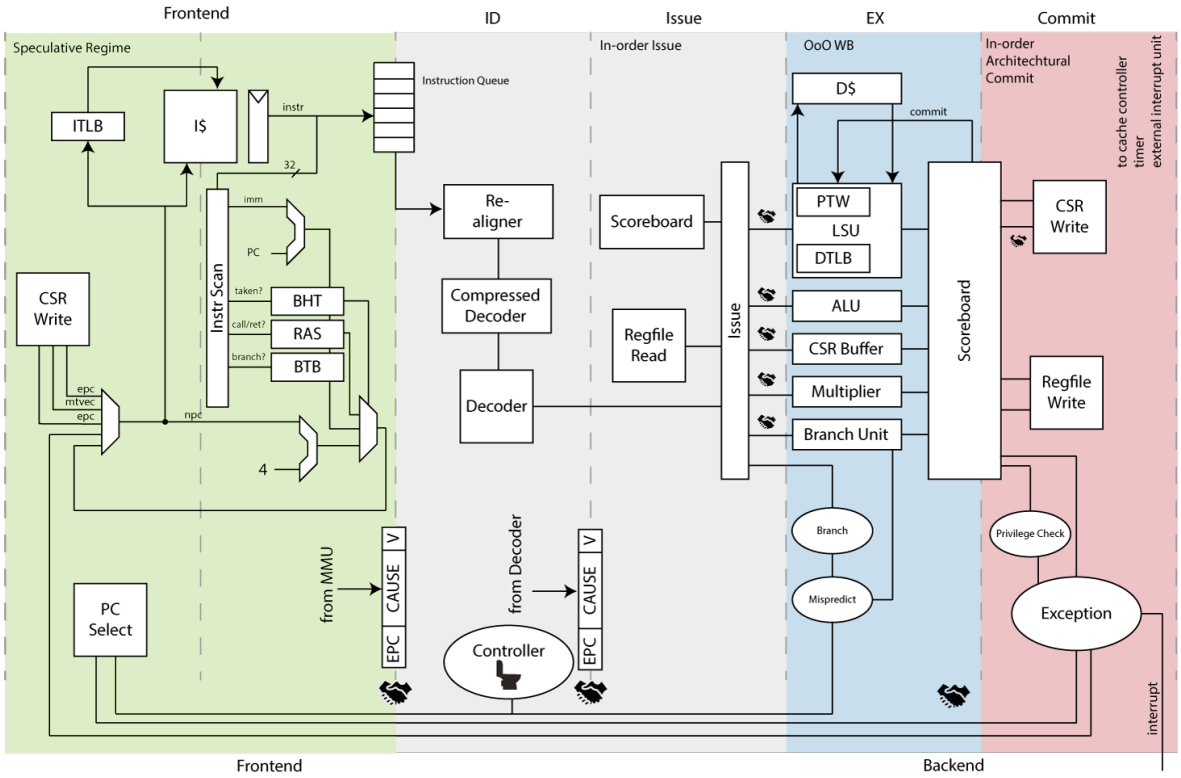

Fig. 21. The functional breakdown of the ARIANE/CVA6 pipeline as described by [21]

two read outputs, potentially allowing a dual-issue pipeline to be implemented, meaning that two instructions could be treated per pipeline stage. It also decouples the frontend from the rest of the pipeline.

The next stage is the **Instruction Decoder** (ID), which performs mainly two tasks. First, it has to search the instruction queue for the next instruction, potentially re-aligning variable length instructions and decompressing compressed instructions. Then, it decodes the properly standardised instruction into the following scoreboard entry:

- PC: PC of instruction
- FU: functional unit to use
- OP: operation to perform in each functional unit
- RS1: register source address 1
- RS2: register source address 2
- RD: register destination address
- Result: result, immediate for unfinished instructions
- Valid: validity of the result
- Use I Immediate: use the immediate as operand b
- Use Z Immediate: use zimm as operand a
- Use PC: use the PC as operand a, PC from exception
- Exception: an exception has occurred
- Branch predict: branch predict scoreboard data structure
- Is compressed: correctly increments the PC in case of a compressed instruction (+2 instead of +4)

After having decoded the instruction, the **Issue** stage has to verify the availability of source operands and whether currently issued instruction will write the same destination register. It then independently communicates with the different functional units, checking their availability, dispatching the issued instructions to them and receiving write-back data from them. It also keeps track of all issued instructions, the status of the functional units and receives the write-back data from the execute stage.

The next stage is the **Execute** (EX) stage, where each functional unit performs the operation they were issued. The CVA6 processor has five functional units, all fully independent from each other: the arithmetic and logic unit (ALU), the multiplier, the CSR buffer, the branch unit, and the load and store unit (LSU). This independence allows the EX stage to execute instructions **out-of-order** while they were issued **in-order**.

The ALU performs every arithmetical and logical operation offered by the RISC-V base ISA in only one clock cycle, whereas the multiplier unit executes integer multiplications in two cycles, and integer divisions and remainder in at most 64 cycles. Because CSR instructions directly impact the instruction flow, they can only be executed if the commit stage, the last stage of the pipeline, allows it: they need to be stored in the CSR buffer while waiting for the commit

signal. The EX branch unit is the backend counterpart of the branch predict unit: it consists of an adder and a comparison unit to perform conditional and unconditional branches in a single cycle, while also being capable of correcting the BHT and setting the PC in case of branch misprediction. Because branches are executed in only one cycle, pipeline stalls are greatly reduced, but it implies that the somewhat complex operations performed by the branch unit are part of the overall critical path. The role of the load and store unit is to communicate with the data cache (D$) to perform load and store operation. Like for the instruction cache, data exchange between the unit and the cache is controlled by a MMU.

The final stage of the pipeline is the **Commit** stage, or write-back (WB). It takes incoming instruction and update the architectural state, which consists in committing (retiring) store instructions and updating the architectural state of the pipeline. This stage also handles exceptions, interrupts and processor stalling, and is responsible of pipeline resets.

The CVA6 processor also integrates an AXI (Advanced eXtensible Interface) bus capable of communicating with a DDR3 (Double Data Rate) SDRAM, a SPI (Serial Peripheral Interface) controller to conncet to an SD card, an Ethernet controller, a JTAG (Joint Test Action Group) port, an UART (Universal Asynchronous Receiver Transmitter) port and a Bootrom. Fig. 22 shows the communication protocols used by CVA6 to communicate with its peripherals.

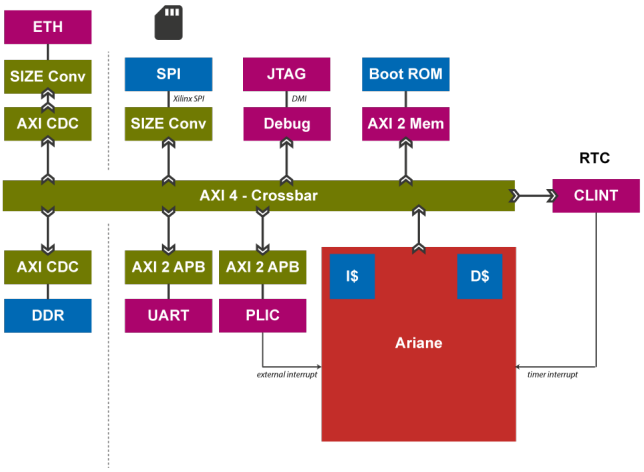

Fig. 22. CVA6 uses AXI protocols to communicate with its peripherals

Even though the OpenHW Group actively maintains the GitHub repository containing the CVA6 source code up to date, the code itself can be very confusing to read, as little information is provided regarding the way the SystemVerilog modules interact with one another. The code is relatively well commented, including a short description of the module at the beginning of each file. However, the seemingly only way to obtain an overview of the code is to open it in a RTL analysis tool and to recursively find our way to the structure of the code, which remains a tedious task to perform.

Furthermore, little to no literature currently exists on CVA6 or how to change its configuration, and only support for the Gensys2 development board [23] is currently provided. In fact, the very recently published [24] and [25] were the only two articles we could find that dived into the CVA6 structure more in depth than presented in the official documentation.

A configuration file [26], describing the parameters that can be modified, can however be found amongst other source files. Those parameters are listed below:

- CONFIG_L1I_SIZE
- ICACHE_SET_ASSOC
- ICACHE_INDEX_WIDTH
- ICACHE_TAG_WIDTH
- ICACHE_LINE_WIDTH
- ICACHE_USER_LINE_WIDTH

- CONFIG_L1D_SIZE
- DCACHE_SET_ASSOC
- DCACHE_INDEX_WIDTH
- DCACHE_TAG_WIDTH
- DCACHE_LINE_WIDTH
- DCACHE_USER_LINE_WIDTH
- DCACHE_USER_WIDTH

C. Zybo Z7 and ZYNQ 7020

The FPGA used to program CVA6 is the Digilent Zybo Z7-20 development board [27] and featuring a Xilinx ZYNQ 7020 SoC (System on Chip) [28]. Fig. 23 shows a functional view of ZYNQ 7020.

The SoC consists of two separate parts communicating together using AXI ports: the **Processing System** (PS) and the **Programmable Logic** (PL).

The PS contains a dual-core ARM Cortex A9, an Advanced Microcontroller Bus Architecture (AMBA) interconnect, a DDR3 Memory controller and other peripheral controllers. The Cortex A9 processors act as masters to all the other components of the SoC, including the PL, and can be used to monitor various signals sent by the PL through the AXI ports. Though the PS won't be described any more precisely, Fig. 24 shows an overview of its functional components.

The PL is the configurable part of the FPGA, where CVA6 will be after it has been programmed with the bitstream. It uses a 28nm technology. Fig. 25 gives the number of logic cells and flip-flops on the PL, along with the amount of RAM available.

It can be programmed using a three-stage boot protocol from three different sources: a micro SD slot, a Quad

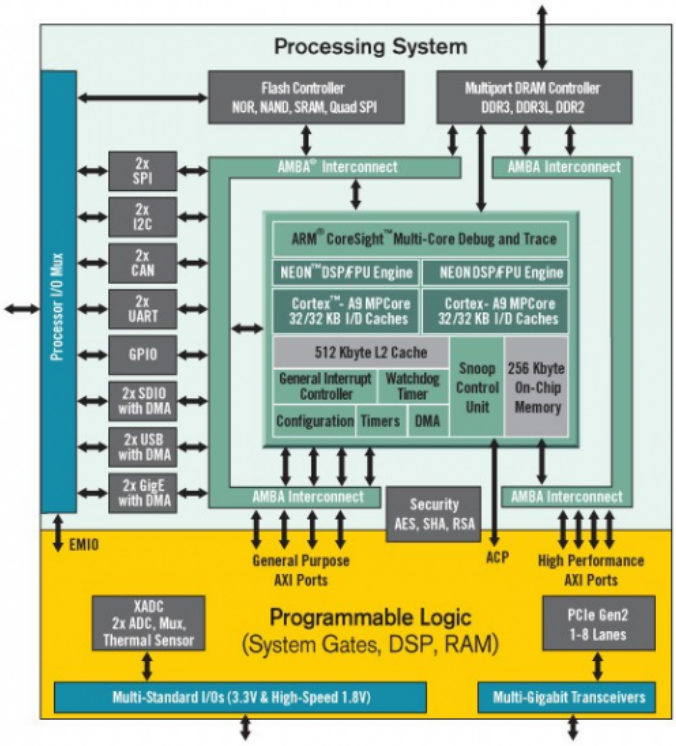

Fig. 23. Functional diagram of the ZYNQ 7020 SoC

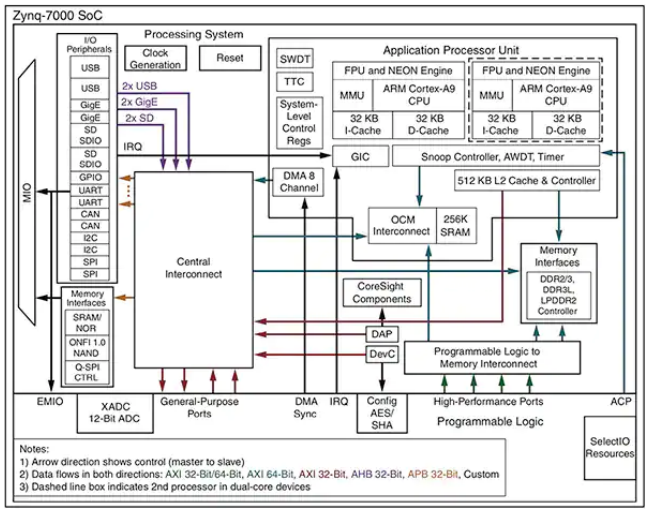

Fig. 24. Functional diagram of the PS

Device Name	Z-7020
Part Number	**XC7Z020**
Xilinx 7 Series Programmable Logic Equivalent	Artix-7 FPGA
Programmable Logic Cells	85K
Look-Up Tables (LUTs)	53,200
Flip-Flops	106,400
Block RAM (# 36 Kb Blocks)	4.9 Mb (140)
DSP Slices (18x25 MACCs)	220
Peak DSP Performance (Symmetric FIR)	276 GMACs
PCI Express (Root Complex or Endpoint)[3]	

Fig. 25. Technical data regarding the PL

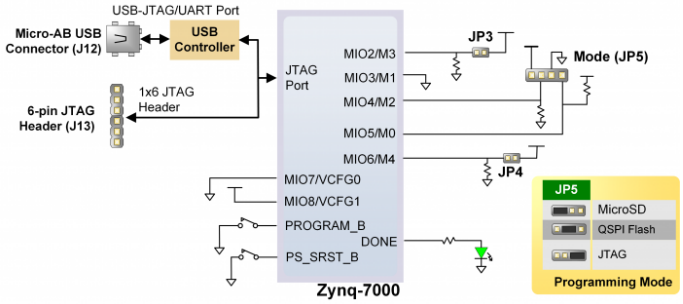

Fig. 26. Diagram of the Zybo Z7 programming source circuit [20]

SPI-flash memory and a JTAG port as described by Fig. 26. Boot stage 0 stage includes the execution of a program called the BootROM, selecting a boot source, and copying the First Step Boot Loader (FSBL). Stage 1 corresponds to executing the FSBL, setting up PS components and configuring the PL by loading and reading the bitstream. Eventually, stage 3 is the execution of any user application.

Aside from the ZYNQ 7020 SoC, the Zybo Z7 features the following components:

- a 1GB DDR3L SDRAM using a 32-bit bus
- a 16MB Quad-SPI flash memory
- a USB port
- a micro-SD socket
- an UART port
- a JTAG port
- a Jack connector
- an Ethernet port
- a Pcam camera connector
- an HDMI input and output
- a 3-port stereo audio codec
- 6 Peripheral module interfaces (Pmod)
- an on/off switch
- 6 push-buttons, including 2 CPU connected
- 4 slide switches
- 5 LEDs, including 1 CPU connected
- 2 RGB LEDs

Fig. 27 shows a picture of the actual development board while Fig. 28 presents the mentioned units and peripherals.

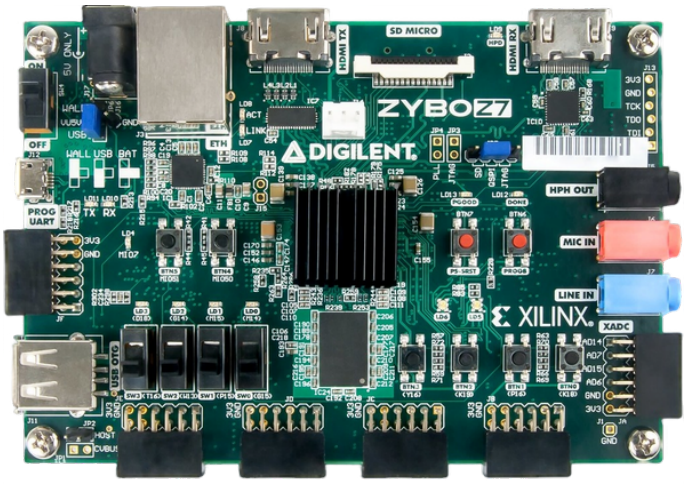

Fig. 27. The Zybo Z7-20 development board

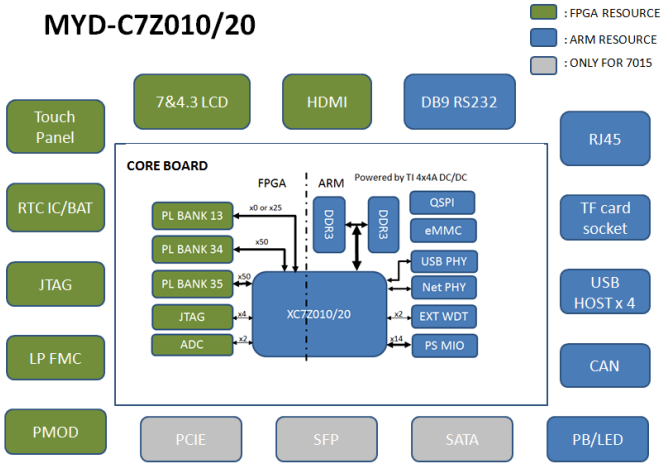

Fig. 28. Functional diagram of the Zybo Z7-20

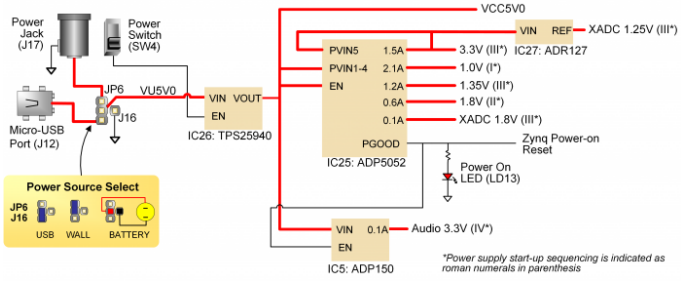

Fig. 29. Diagram of the Zybo Z7 power supply circuit [20]

It provides a 33.333 MHz clock that allows the processor to operate at 663 MHz and the DDR bus at 533 MHz. An external clock of frequency 125 MHz is directly connected to the PL, allowing it to be completely independent from the PS as well a generating two Mixed-Mode Clock Manager (MMCM) and two Phase-Locked Loop (PLL).

The Zybo Z7 board can be connected using three different power sources as described by Fig. 29: using a USB cable, using a Jack connector, or direct plugging a battery into the circuit. It requires a 5V power supply to be operational.

D. Testbench Application

The studied testbench consists in propagating a 24*24 pixels black and white image through a simple **convolutional neural network** (CNN) written in C. It has a total of four layers, two convolution layers and two fully connected layers, which are shown by Fig. 30. Fig. 31 represents the input, output and hidden neuron distribution between the different layers of the CNN.

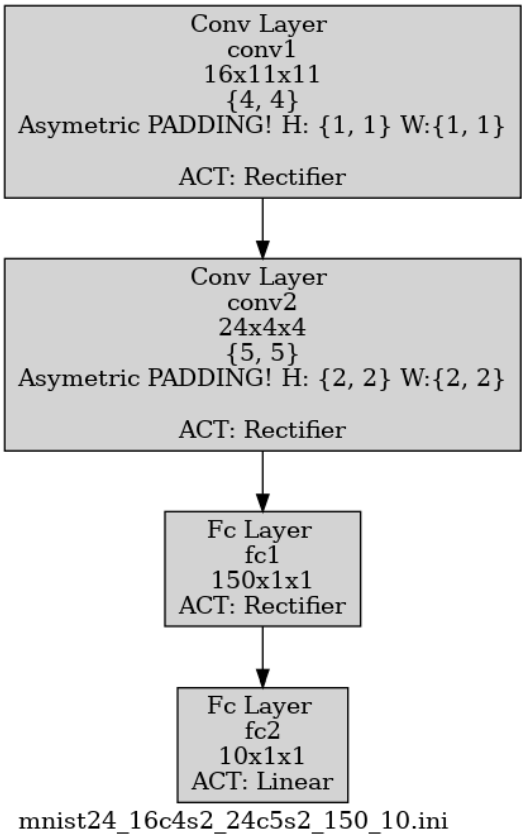

Fig. 30. Topology of the studied CNN

The propagated image corresponds the number 4 in shades of grey, and is represented as a buffer of 8-bit unsigned integers. The CNN is already trained (the weights of each link can be found in a header file), so only the loading and the propagation need to be executed. At the end of the propagation, certain values are displayed, among which:

- the expected value (4)
- the predicted value
- the number of instructions executed

- the number of clock cycles

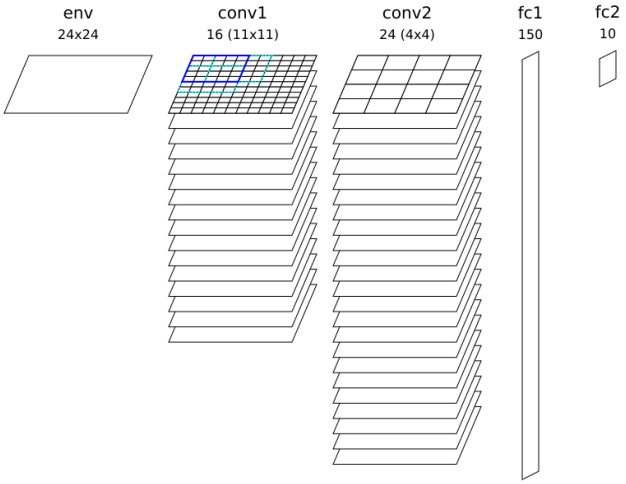

Fig. 31. Neuron distribution in the CNN

It must be compiled using the RISC-V GNU Compiler Toolchain [29], a C/C++ cross-compiler dedicated to produce a RISC-V compatible **executable and linkable format** (ELF) called elf32-littleriscv. The compiler is very similar in its use to the gcc compiler, can be called using the $riscv32\text{-}unknown\text{-}elf\text{-}gcc$ command. We used the following compilation options:

- O3: the highest standard optimisation possible
- static: the program won't require a dependency on dynamic libraries at runtime
- Wall: enables warning messages
- pedantic: adds warnings to encourage strict adherence to ISO C

The final compiled program is 22931 instructions long, and can be directly loaded into the CVA6 instruction cache during the FPGA programming. Similarly, the weight of the CNN and the buffer corresponding to the image are loaded into the data cache during programming.

An application profiling of the CNN was performed using the Intel Vtune Profiler 2022.3. Fig. 32 and Fig. 33 show various metrics corresponding to the execution of 100,000 propagations.

In particular, it should be noted that the testbench is a relatively small application, as displaying the propagation metrics takes more time than actually computing the propagation: 58.8% and 41.2% of CPU time respectively. This is also shown by the intensive port utilisation percentage visible in Fig. 32.

It also shows that the application is well pipelined, as an average of 56% of instructions are retired by the commit stage per cycle. It has an extremely low frontend bound value

Elapsed Time: 21.431s

Clockticks:	53,911,000,000
Instructions Retired:	113,711,000,000
CPI Rate:	0.474
MUX Reliability:	0.995
Retiring:	56.0%
Light Operations:	51.9%
Heavy Operations:	4.1%
Front-End Bound:	7.4%
Front-End Latency:	6.1%
ICache Misses:	2.2%
ITLB Overhead:	5.9%
Branch Resteers:	1.5%
DSB Switches:	0.2%
Length Changing Prefixes:	0.0%
MS Switches:	0.9%
Front-End Bandwidth:	1.3%
Bad Speculation:	0.6%
Branch Mispredict:	0.6%
Machine Clears:	0.0%
Back-End Bound:	36.0%
Memory Bound:	9.2%
L1 Bound:	6.3%
L2 Bound:	2.2%
L3 Bound:	5.3%
DRAM Bound:	0.1%
Store Bound:	0.1%
Core Bound:	26.8%
Divider:	0.0%
Port Utilization:	40.7%

Fig. 32. Runtime metrics obtained by executing the testbench 100000 times

Function	Module	CPU Time	% of CPU Time
printf	msvcrt.dll	6.813s	58.7%
func@0x4017b0	mnist.exe	4.780s	41.2%
[mnist.exe]	mnist.exe	0.004s	0.0%

Fig. 33. CPU time distribution for the testbench

of 7.4%, including only 2.2% of instruction cache misses, which is most likely due to the limited amount of instructions that make up the application. Another remarkably low value is the 0.6% of bad speculation due to cache misses: branches in the CNN are mostly due to loops, which can easily be predicted as not taken most of the time by the BHT unit. While backend bounds represent 36.0% of pipeline slots, the limited amount of data (mostly CNN weights and image buffer) explains why the application is CPU bound for 26.8% of time and only memory bound 9.2% of the time.

These metrics are summarised by a diagram called ***µ*Pipe diagram**, represented in Fig. 34

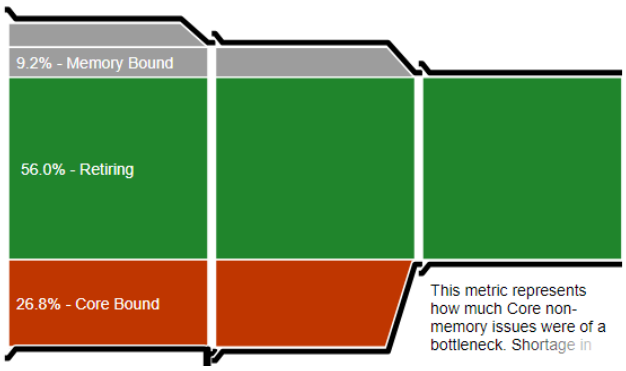

Fig. 34. Pipeline bottlenecks and inefficiencies for the CNN propagation

E. Software Automation

In September 2021, Thales, along with the GDR SoC2 and the CNFM, launched the 2nd National RISC-V Student Contest [30], a competition designed to motivate the interest of french students in electronics in the RISC-V architecture by analysing and modifying an FPGA emulation of a RISC-V processor. Groups of students were given access to a GitHub repository [31] where they could find various files, including:

- The complete source code of the ARIANE/CV(32)A6 processor
- Source codes of multiple testbench applications written in C to be executed on RISC-V architecture processors
- A link to the RISC-V GNU Compiler Toolchain GitHub repository [29] to compile the testbench on the RISC-V ISA
- Elements of Tcl scripts to automate some aspects of the Vivado workflow (synthesis, simulation, implementation)
- Documents describing the required material, software and how to use the scripts

This contest provided us with a solid base to run and automate the workflow described in section IV.

A 32 GB RAM 6 hyper-threaded Intel Core i7 computer using Ubuntu 18.04 was used to compute the entire workflow. To be compliant with the contest, Vivado 2020.1 [32], Vitis 2020.1 [33] and Questa Sim 10.7 [34] were used by the fitness evaluation process. Vtune Profiler 2022.3 [35] was used to perform the application profiling of the testbench, and the $\mathcal{EPR}$ plot and the Pareto frontier extraction were performed under Python 3.10. Communication with the FPGA was established thanks to OpenOCD 0.10 [36] on a minicom terminal. Each fitness evaluation iteration was automated by Tcl scripts while the whole workflow was run by bash scripts.

The clock frequency was set to 45 MHz, as the default value given by the contest organisers.

Due to technical limitations, the execution branch could not return an execution duration, but instead returned the number of instructions and cycles as shown by Fig. 35

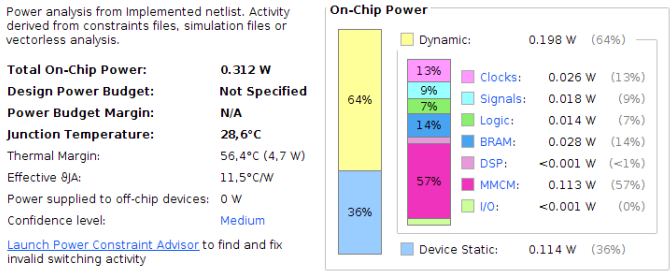

Fig. 35. Power estimation metrics returned by Questa Sim

This is not an issue as execution duration can be obtained by calculating $T = n_{cycles}/f_{CK}$. However, due to branch predict errors, amongst other things, the value of n_{cycles} may be slightly increased because more instructions were executed. To solve this, the value returned by the performance function is the $CPI := n_{cycles}/n_{instr}$ (cycles per instruction) instead of the computation time. We thus have:

$$T = f_{CK} \times n_{instr} \times \underbrace{CPI}_{P(c):=} \tag{27}$$

Since $P^0(c) = P(c)/P(0_{\mathcal{C}})$ and given that clock frequency is the same between every configurations, the only impact of considering CPI instead of T is actually to remove variations in the number of instructions executed, which introduced noise and isn't representative of an optimisation.

Similarly, Questa Sim only gives the total power distribution and not the total energy spent. Because $E = P_{tot} \times T$ and that T is prone to non configuration-related uncertainties, taking power consumption as the value of $E(c)$ also reduces the noise on $E^0(c)$ with changing its meaning.

VI. Results and Analysis

A. Chosen Parameters

As mentioned at the beginning of section IV, implementation using Vivado 2020.1 is highly customisable thanks to numerous design options and directives [37]. Fig. 36 describes the placement and routing flow of Vivado, highlighting three main categories of optimisation parameters: logic, power and physical. These categories roughly correspond to performance, energy and resource optimisations respectively, but the relationships between the components of the trade-off mean that each parameter can impact all three.

Logic optimisations are performed by the Tcl command *opt_design* and can be customised using the following options:

- retarget: retargets one type of block to another
- propconst: simplifies logic by propagating constant values

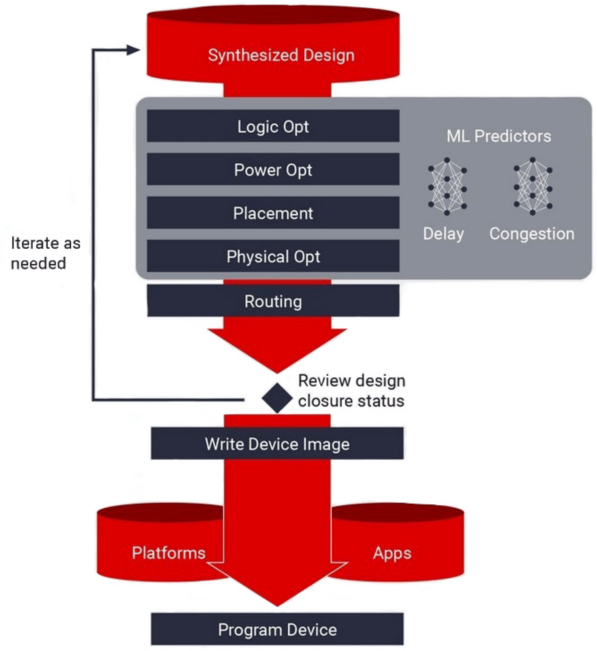

Fig. 36. Vivado Placement and Routing flow [38]

- sweep: removes load-less cells and unconnected nets
- bram_power_opt: enables power optimisation on block RAM cells
- remap: recombines multiple LUTs into a single LUT
- aggressive_remap: more exhaustive version of remap
- resynth_area: re-synthesis to reduce the number of LUTs
- resynth_seq_area: more exhaustive version of resynth _area, including logic optimisations
- resynth_remap: remaps the design to improve critical paths by performing re-synthesis to optimise logic
- muxf_remap: optimises multiplexers into lookup tables (LUT) to improve routability
- bufg_opt: creates global clock buffers on set/reset nets and clock enables
- shift_register_opt: various shift register optimisations
- dsp_register_opt: optimises Digital Signal Processing (DSP) Slices to improve timing. DSP Slices are high performance computation units typically used for matrix multiplication or filter calculus
- carry_remap: optimises Carry chains to LUTs to improve routability
- directive: customisable macro that cannot be used with other options

The use of the directive option forces opt_design to use certain options depending on the specified directive:

- Default: retarget, propconst, sweep, bufg_opt, shift_register_opt, bram_power_opt
- Explore: retarget, propconst, sweep, bufg_opt, shift_register_opt, propconst, sweep, bram_power_opt
- ExploreWithRemap: retarget, propconst, sweep, bufg_opt, shift_register_opt, propconst, sweep, remap, bram_power_opt
- ExploreArea: retarget, propconst, sweep, bufg_opt, shift_register_opt, propconst, sweep, resynth_area, bram_power_opt
- AddRemap: retarget, propconst, sweep, bufg_opt, shift_register_opt, remap, bram_power_opt
- ExploreSequentialArea: retarget, propconst, sweep, bufg_opt, shift_register_opt, propconst, sweep, resynth_area, resynth_seq_area, bram_power_opt
- RuntimeOptimized: retarget, propconst, sweep, bufg_opt, shift_register_opt

Power optimisation is executed by $power\_opt\_design$ and is capable of automatically perform clock gating before Block RAM cells. Because $power\_opt\_design$ is reduced to intelligent clock gating, it can only be customised by running the set_power_opt command, which is used to specify which cells will be considered.

Physical optimisations are performed by $phys\_opt\_design$ and have the same customisation level as opt_design. Optimisation options include:

- rewire: swaps LUTs connections to optimise the logic of critical signals
- fanout_opt: optimises high-fanout nets
- placement_opt: replaces critical path cells to reduce delays
- routing_opt: replaces critical path nets to reduce delays
- bram_register_opt: can move registers out of the Block RAM cell into the logic array and vice versa
- critical_cell_opt: replicates cells in failing paths
- shift_register_opt: can move registers out of the Block RAM cell into the logic array
- uram_register_opt: can move registers out of the Ultra-RAM cell into the logic array and vice versa
- dsp_register_opt: can move registers out of the DSP cell into the logic array and vice versa
- retime: improves the delay on the critical path by moving registers across combinational logic
- clock_opt: creates BUFGs to alter the critical path
- directive: customisable macro that cannot be used with other options. They won't be listed for $phys\_opt\_design$

Unfortunately, the short amount of time available didn't allow us to take every listed option into account, let alone explore the entire configuration space. This is why we chose to only consider opt_design parameters and $power\_opt\_design$ in our definition of $\mathcal{C}$, which already corresponds to 16 optimisation parameters. We evaluated every configuration of degree 1, and also evaluated every configuration corresponding to a $power\_opt\_design$ directive in order to plot some higher degree configurations as well. In particular, the only low power method considered is the clock gating, and no optimisation

parameter deduced from the testbench profiling and the CVA6 structure were taken into account.

B. Experimental Results

Fig. 37 to 40 show the 23 evaluated configurations plotted in the $\mathcal{EPR}$ space.

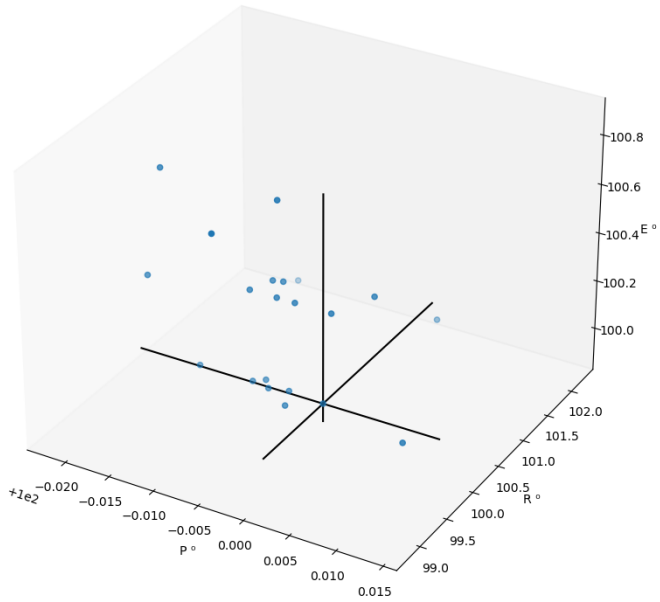

Fig. 37. The $\mathcal{EPR}$ space

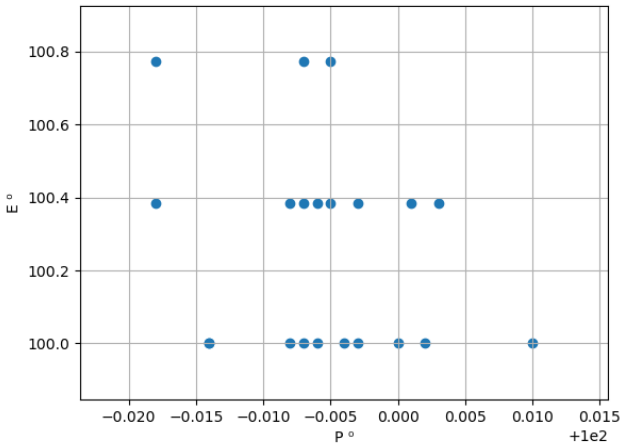

Fig. 38. The $\mathcal{EPR}$ space projected on the $\mathcal{PE}$ plan

In spite of the relatively small number of points, a Pareto Frontier is easily identifiable on the three projections. It must be noted that the power consumption given by Questa were only precise to 1mW, which explains the discretisation that can be observed regarding the distribution of the values of E^0.

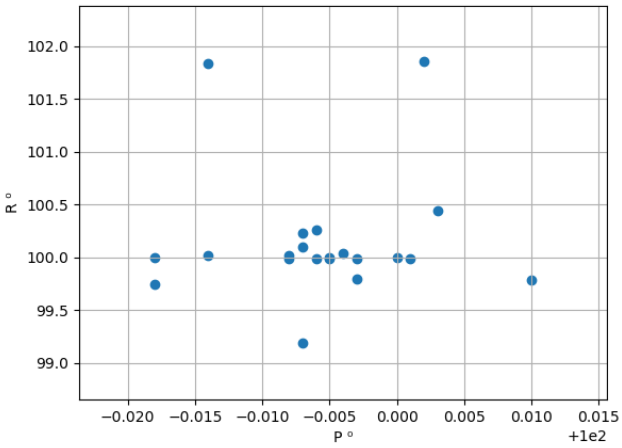

Fig. 39. The $\mathcal{EPR}$ space projected on the $\mathcal{PR}$ plan

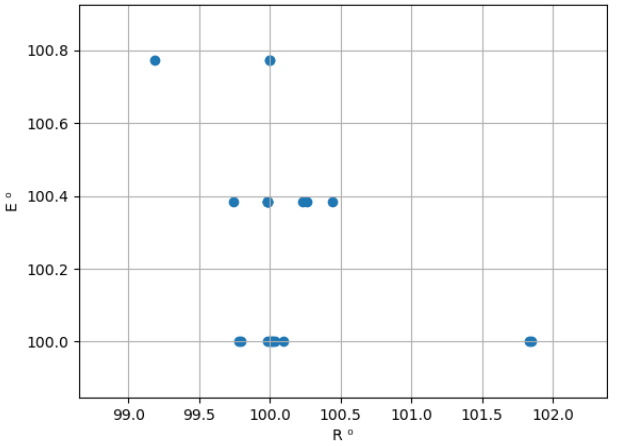

Fig. 40. The $\mathcal{EPR}$ space projected on the $\mathcal{RE}$ plan

The maximum effective distance between two points is 2.77, with an average of $7.14 * 10^{-1}$ and a standard deviation of $6.28 * 10^{-1}$. While those numbers cannot be easily interpreted as such, they show that most evaluated configurations are gathered around $0_{\mathcal{C}}$, with some others being unusually far from them. Analysing the data shows that the first set of points is mostly composed of degree 1 configurations, while outside points usually come from directives. In fact, only considering $\mathcal{C}_1$ drops the maximum distance, average and standard deviation to $7.98 * 10^{-1}$, $3.51 * 10^{-1}$ and $2.06 * 10^{-1}$ respectively. This indicates that the degree of a configuration of a configuration seems to be in direct correlation with the fitness difference from $0_{\mathcal{C}}$: the more optimisations were used in a configuration, the more its energy consumption, performance and resource use differ from the standard configuration.

If we only consider one component of $\mathcal{EPR}$ at a time, more information can be gathered. Fig. 41 summaries the configuration distribution for E^0, P^0 and R^0 gains respectively.

	E^0	P^0	R^0
maximum	7.72 ‰	$1.00 * 10^{-1}$ ‰	18.5 ‰
minimum	0.00 ‰	$-1.80 * 10^{-1}$ ‰	-8.32 ‰
average	2.56 ‰	$-5.57 * 10^{-2}$ ‰	1.52 ‰
std. dev.	2.75 ‰	$6.67 * 10^{-2}$ ‰	5.96 ‰

Fig. 41. Energy, performance and resource gain statistics, in ‰ of $0_{\mathcal{C}}$

As stated before, most degree 1 configurations bring the values of both the average and the standard deviation for each gain closer to 0. This lack of diversity can hopefully can be balanced by evaluated high degree configurations.

The tendency of the $\mathcal{EPR}$ distribution to be slightly shifted towards negative performance gains, along with performance being the least impacted component of the E/P/R trade-off, is most likely due to the fact that no directly performance-related optimisation were considered, dedicating all the optimisation efforts to energy consumption and resource use.

On the other hand, the fact that resource use is by far the most impacted component is explained by the fact that we only considered implementation optimisations (though $power\_opt\_design$ performs a low-power optimisation), meaning that most of the optimisation efforts were used in trying to optimise placement, i.e. resource use.

VII. Future Work

The work presented in this paper is far from being over. In spite of limited time given to run the fitness evaluation process, the obtained results already are very encouraging, and suggests that further evaluations further than degree 1 configurations may lead to more significant benefits. Parallelising configuration evaluations using multiple computers will allow to generate more data.

Furthermore, a lot more work would be required to have a firm grasp on the software, the RISC-V ISA and CVA6. In particular, the absence of extensive documentation about CVA6 considerably slowed the process of choosing useful parameters from both CVA6 itself and Vivado.

Eventually, since ε-clustering and n-clustering techniques induce approximations by altering the configuration space, we would have liked to also try a different approach. A possible alternative we came up with would be to implement genetic algorithms using Pareto-efficient configurations as base for the selection stage.

VIII. Conclusion

In this paper, we have described the physical distribution of power consumption in an electronic circuit and we have proposed a mathematical model to describe how various energy, performance and resource optimisations can be implemented on a processor to create alternative versions, called configurations. We then developed a general, technology-agnostic workflow taking the list of optimisations as an input, and returning a set of Pareto-efficient configurations for the user to choose from. Eventually, we used the RISC-V ISA CV32A6 processor and a CNN propagation testbench as a proof of concept. Due to schedule limitations, we were not able to run the workflow in its entirety, but we obtained promising results that are highly encouraging for the future.

With the explosion of low-power techniques and the ever growing development of computer-assisted tools for embedded systems, there are still too little technology-independent automated conception designs in modern literature, especially for newer processors. Engineers and researchers are forced to use rule of thumbs and highly heuristic methods, and would greatly benefit from having an automated exhaustive workflow from both a theoretical and practical point of view. We hope that our findings will help making non-heuristic time-efficient methods become more widely used in the fields of system engineering and microelectronics.

Acknowledgements

We thank Goran Frehse, director of the U2IS laboratory, for the hardware and software resources he made available for our research.

We thank Hervé Le Provost and Farhat Thabet for their assistance in setting up and using the software.

We are also grateful for Frederic Dulucq, for a shining a different light on our approach.

References

[1] https://open-src-soc.org/2022-05/posters.html
[2] F. Thabet, "Low power techniques overview - for SoC", Master 2 SETI, Paris-Saclay University
[3] M. B. Lin, "Introduction to VLSI Systems: A Logic, Circuit, and System Perspective", 2012, chapter 2, section 2.4.3
[4] V. Natarajan, A. K. Nagarajan, N. Nagarajan Pandian, and V. G. Savithri, "Low Power Design Methodology", in Very-Large-Scale Integration. London, United Kingdom: IntechOpen, 2018 [Online]. Available: https://www.intechopen.com/chapters/59358 doi: 10.5772/intechopen.73729
[5] Y. I. Ismail, E. G. Friedman and J. L. Neves, "Dynamic and short-circuit power of CMOS gates driving lossless transmission lines," in IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications, vol. 46, no. 8, pp. 950-961, Aug. 1999, doi: 10.1109/81.780376.
[6] R. Panwar, "Introduction to CMOS VSLI design - design for low power"
[7] G. Franchi, "Unsupervised Learning", ENSTA Paris
[8] https://sirinnes.wordpress.com/2013/04/25/pareto-frontier-graphic-via-python/

[9] Srinivasan, Nandita & Prakash, Navamitha & Dabhekar, Shalakha & D., Sivaranjani & G., Swetha & B., Bala Tripura Sundari. (2015). Power Reduction by Clock Gating Technique. Procedia Technology. 21. 631-635. 10.1016/j.protcy.2015.10.075.
[10] Hailin Jiang, M. Marek-Sadowska and S. R. Nassif, "Benefits and costs of power-gating technique," 2005 International Conference on Computer Design, 2005, pp. 559-566, doi: 10.1109/ICCD.2005.34.
[11] https://anysilicon.com/power-gating/
[12] M. S. Bhat, Srigowri, V. V. Rao and B. P. V. Pai, "Implementation of dynamic voltage and frequency scaling for system level power reduction," International Conference on Circuits, Communication, Control and Computing, 2014, pp. 425-430, doi: 10.1109/CIMCA.2014.7057837.
[13] Jeong, Kwangok & Kahng, Andrew & Samadi, Kambiz. (2008). Quantified Impacts of Guardband Reduction on Design Process Outcomes. Proceedings of the 9th International Symposium on Quality Electronic Design, ISQED 2008. 790-797. 10.1109/ISQED.2008.155.
[14] https://inspirehep.net/authors/1071553
[15] A. Waterman, K. Asanović, "The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA", CS Division, EECS Department, University of California, Berkeley, December 13, 2019
[16] A. Waterman, K. Asanović, John Hauser, "The RISC-V Instruction Set Manual - Volume I: Unprivileged ISA", CS Division, EECS Department, University of California, Berkeley, December 13, 2019
[17] https://five-embeddev.com/riscv-isa-manual/
[18] https://riscv.org/
[19] https://github.com/openhwgroup/cva6
[20] https://docs.openhwgroup.org/projects/cva6-user-manual/
[21] F. Zaruba and L. Benini, "The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 27, no. 11, pp. 2629-2640, Nov. 2019, doi: 10.1109/TVLSI.2019.2926114.
[22] O. Hammami, "Introduction à l'architecture des microprocesseurs", ENSTA Paris
[23] https://digilent.com/reference/programmable-logic/genesys-2/reference-manual?redirect=1
[24] Martinoli, Valentin et al. "CVA6's Data cache: Structure and Behavior." ArXiv abs/2202.03749 (2022)
[25] Open Virtual Plateform, "OVP guide to using processor podels - model specific information for OpenHWGroup CV32A6", 27 July 2022
[26] https://github.com/openhwgroup/cva6/blob/master/core/include/ariane_pkg.sv
[27] https://digilent.com/reference/_media/reference/programmable-logic/zybo-z7/zybo-z7_rm.pdf
[28] https://docs.xilinx.com/v/u/en-US/ds190-Zynq-7000-Overview
[29] https://github.com/riscv-collab/riscv-gnu-toolchain
[30] https://web-pcm.cnfm.fr/wp-content/uploads/2021/10/Annonce-RISC-V-contest-2021-2022-v1.pdf
[31] https://github.com/ThalesGroup/cva6-softcore-contest
[32] https://www.xilinx.com/products/design-tools/vivado.html
[33] https://www.xilinx.com/products/design-tools/vitis.html
[34] https://eda.sw.siemens.com/en-US/ic/questa/simulation/advanced-simulator/
[35] https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
[36] https://openocd.org/
[37] https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0019-vivado-implementation-hub.html
[38] https://www.xilinx.com/products/design-tools/vivado/implementation.html