EN202 Projet VHDL

Etienne MEIDINGER
Alexis TREILLES

Décembre 2023

Encadrant :
RENAUD Sylvie

Table des matières

1 Introduction **3**

2 Fonctionnement du séquenceur **3**
2.1 Description du projet 3
2.2 Architecture du séquenceur 5
2.3 Description detaillée du module lecture2 6
2.4 Description detaillée du module read 7
2.5 Simulation du bloc gene_pis 8

3 Analyse des performances **10**
3.1 Timing report 10
3.2 Utilization report 11
3.3 Power report 11

1 Introduction

Notre projet de séquenceur/boîte à rythme, implémenté en VHDL sur la carte Nexys A7 de Digilent, offre une approche innovante pour la création de séquences rythmiques personnalisées. Dans ce document, nous détaillerons en profondeur le fonctionnement de chaque composant, offrant ainsi une compréhension complète de l'architecture et des performances du séquenceur/boîte à rythme.

2 Fonctionnement du séquenceur

2.1 Description du projet

Ce projet de séquenceur/boîte à rythme vise à offrir une plateforme interactive pour la création de séquences rythmiques personnalisées. En utilisant une combinaison de mémoire ROM, boutons de contrôle et indicateurs visuels, il permet une composition musicale dynamique et intuitive.

Architecture Globale

Le système se compose de trois principales parties : la mémoire ROM contenant les sons de batterie, la logique de contrôle gérant le partitionnement temporel des sons, et le bloc de lecture qui joue les sons en fonction de la partition générée. De plus, les 16 LEDs représentent visuellement le curseur temporel, et l'affichage à 7 segments est utilisé pour afficher la valeur du tempo.

Mémoire ROM

Une mémoire ROM est utilisée pour stocker les trois sons de batterie - kick, snare, et hi-hat. Chaque son est identifié par un index, et ces indices sont utilisés pour sélectionner le son à partitionner.

Contrôle de Partitionnement

1. **Sélection de Son:**
   - Trois boutons, B_LEFT, B_CENTER, et B_RIGHT, permettent de choisir le type de son à partitionner (kick, snare, ou hi-hat).
   - Un premier appui sur l'un de ces boutons active le mode de sélection pour le son correspondant.
   - Un second appui désactive le mode de sélection et revient à l'état initial.
2. **Partitionnement Temporel:**
   - 16 commutateurs sont utilisés pour représenter la partition temporelle, où chaque commutateur représente une unité de temps (jouée ou non jouée).
   - Un curseur se déplace sur les commutateurs à une fréquence prédéfinie par l'utilisateur (4 fois le tempo).
   - Les positions du curseur actif déterminent la partition temporelle du son sélectionné.
3. **Réglage du Tempo:**
   - Deux boutons, B_UP et B_DOWN, permettent d'ajuster le tempo.
   - Appuyer sur B_UP augmente le tempo, tandis que B_DOWN le diminue.
   - La valeur du tempo est affichée sur l'affichage à 7 segments, avec une valeur par défaut de 120 bpm.
4. **Affichage du Curseur Temporel:**
   - Les 16 LEDs disposées au-dessus des commutateurs représentent le curseur temporel qui défile dans le temps.
   - L'allumage des LEDs indique la position actuelle du curseur dans la partition temporelle.

Bloc de Lecture

1. **Instructions de Lecture:**
   - Les positions du curseur actif sont utilisées pour générer des instructions de lecture.
   - Ces instructions indiquent au bloc de lecture quand jouer le son en mémoire.
2. **Bloc de Lecture:**
   - Le bloc de lecture récupère les instructions générées et joue les sons correspondants à partir de la mémoire ROM.

Fonctionnement Général

1. **Sélection du Son:**
   - L'utilisateur choisit le son à l'aide des boutons B_LEFT, B_CENTER, et B_RIGHT.
2. **Partitionnement Temporel:**
   - En mode de sélection, l'utilisateur utilise les 16 commutateurs pour définir la partition temporelle du son sélectionné.
3. **Réglage du Tempo:**
   - L'utilisateur peut ajuster le tempo à l'aide des boutons B_UP et B_DOWN.
   - La valeur du tempo est affichée sur l'affichage à 7 segments.
4. **Curseur et Génération d'Instructions:**
   - Un curseur se déplace à la fréquence définie par l'utilisateur sur les commutateurs, générant des instructions de lecture.
   - Les LEDs indiquent visuellement la position actuelle du curseur dans la partition temporelle.
5. **Lecture des Sons:**
   - Les instructions sont transmises au bloc de lecture qui joue les sons correspondants à partir de la mémoire ROM.

Ce système offre une flexibilité pour créer des partitions rythmiques en temps réel, permettant à l'utilisateur de personnaliser la séquence de batterie selon ses préférences.

2.2 Architecture du séquenceur

L'architecture VHDL de notre système est divisée en cinq blocs interconnectés, chacun jouant un rôle spécifique dans le fonctionnement global de la séquenceur/boîte à rythme sur la carte Nexys A7 de Digilent.

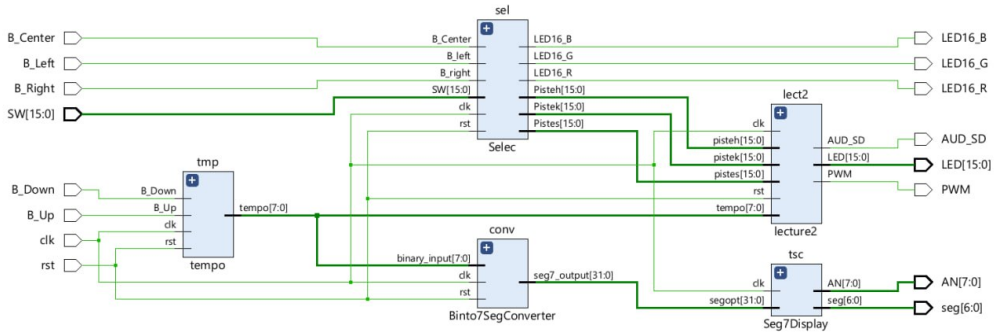

Figure 1: Description de l'architecture du système

Tempo :

- **Entrées :** clk, rst, B_UP, B_DOWN.
- **Sortie :** tempo (std_logic_vector[7 downto 0]).
- **Fonction :** Génère et maintient la valeur du tempo en fonction des boutons B_UP et B_DOWN.

Binto7SegConverter :

- **Entrées :** clk, rst, tempo.
- **Sortie :** seg7_output (std_logic_vector[31 downto 0]).
- **Fonction :** Convertit la valeur du tempo en un signal approprié pour l'affichage sur les 7 segments.

Seg7Display :

- **Entrées :** clk, seg7_output.
- **Fonction :** Utilise le signal converti pour afficher le tempo actuel sur l'affichage 7 segments.

Selec :

- **Entrées :** clk, rst, SW[15 downto 0], B_LEFT, B_CENTER, B_RIGHT.
- **Sorties :** Pistek[15 downto 0], Pistes[15 downto 0], Pisteh[15 downto 0], LED16_B, LED16_R, LED16_G.
- **Fonction :** Machine d'état permettant d'assigner les valeurs des switches aux pistes appropriées selon le mode de sélection. La couleur des LEDs 16 indique le mode sélectionné.

lecture2 :

- **Entrées :** clk, rst, tempo, Pistek, Pistes, Pisteh.
- **Sorties :** AUD-SD, LED[15 downto 0], PWM.
- **Fonction :** Bloc complexe comprenant le bloc read avec 8 sous-blocs. Son rôle est de jouer les sons en fonction des pistes sélectionnées. Les LEDs 0 à 15 défilent avec le curseur.

Maintenant que nous avons décrit l'architecture générale du système, concentrons-nous sur une analyse détaillée du bloc lecture2. Ce bloc central est essentiel pour comprendre comment les sons sont lus et joués en fonction des pistes sélectionnées et du tempo défini.

2.3 Description detaillée du module lecture2

Le bloc `lecture2` est le cœur du séquenceur/boîte à rythme, composé de quatre blocs périphériques, chacun ayant un rôle spécifique dans la génération et la lecture des séquences audio.

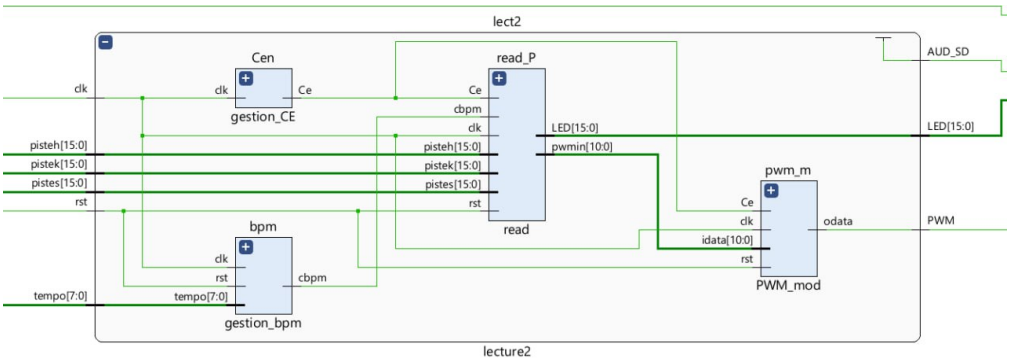

Figure 2: Description de l'architecture du bloc lecture2

gestion_CE :

- **Entrée :** clk
- **Sortie :** Ce (horloge à 44,1 kHz)
- **Fonction :** Génère une horloge à une fréquence de 44,1 kHz, correspondant à la fréquence d'échantillonnage des sons en mémoire ROM.

gestion_bpm :

- **Entrées :** clk, rst, tempo[7 downto 0]
- **Sortie :** cbpm (horloge à la fréquence sélectionnée par l'utilisateur)
- **Fonction :** Génère un signal d'impulsion à une fréquence équivalente à celle définie par le tempo sélectionné.

read :

- **Entrées :** Ce, cbpm, clk, rst, Pistek, Pistes, Pisteh
- **Sorties :** pwmin[10 downto 0], LED[15 downto 0]
- **Fonction :** Composé de 8 sous-blocs, lit en boucle les sons en mémoire ROM selon les partitions données en entrée et à la fréquence du tempo. Les sorties comprennent les données pour le bloc `PWM_mod` et les LEDs pour représenter le curseur.

PWM_mod :

- **Entrées :** Ce, clk, rst, pwmin
- **Sortie :** PWM
- **Fonction :** Gère les séquences audio échantillonnées à 44,1 kHz, calculant la résolution du module PWM en fonction de la fréquence d'horloge du FPGA (100 MHz).

AUD_SD est fixé à 1. Il indique que la sortie audio est activée.

En approfondissant l'analyse, nous allons maintenant explorer plus en détail le fonctionnement du bloc `read` et de ses 8 sous-blocs. Cette section permettra de comprendre comment les partitions temporelles influencent la lecture des sons stockés en mémoire ROM.

2.4 Description detaillée du module read

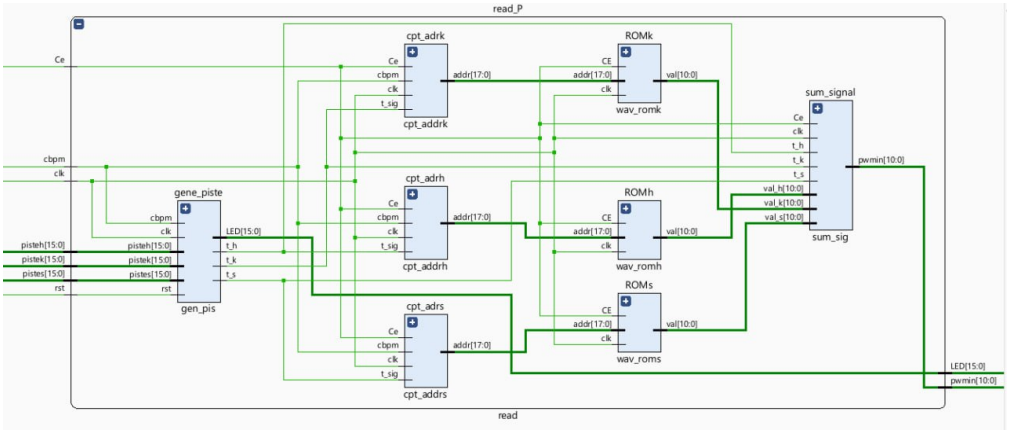

Figure 3: Description de l'architecture du bloc read

Le bloc `read` est composé de 8 sous-blocs, dont le premier est `gen_piste`. Il génère le curseur temporel qui défile sur les vecteurs piste, permettant ainsi la création du curseur LED et les trois signaux temporels `t_k`, `t_s`, `t_h`. Le code VHDL du bloc `gen_piste` est le suivant :

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity gen_pis is
    Port ( clk : in STD_LOGIC;
           rst : in STD_LOGIC;
           cbpm : in STD_LOGIC;
           pistek : in STD_LOGIC_VECTOR (15 downto 0);
           pistes : in STD_LOGIC_VECTOR (15 downto 0);
           pisteh : in STD_LOGIC_VECTOR (15 downto 0);
           LED : out std_logic_vector ( 15 downto 0);
           t_s : out STD_LOGIC:='0';
           t_k : out STD_LOGIC:='0';
           t_h : out STD_LOGIC:='0');
end gen_pis;

architecture Behavioral of gen_pis is
    signal internal_counter : INTEGER range 0 to 15 := 15;
    signal last_cbpm : STD_LOGIC := '0';
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if rst='0' then
                internal_counter<=15;
            else
            -- Detecter le front montant de cbpm
            if last_cbpm = '0' and cbpm = '1' then
                if internal_counter>0 then
                    t_s <= pistes(internal_counter);
                    t_k <= pistek(internal_counter);
                    t_h <= pisteh(internal_counter);
                    internal_counter <= internal_counter - 1;
                    LED <= (others => '0');
                    LED(internal_counter) <= '1';
```

```
38
39                  else
40                      t_s <= pistes(internal_counter);
41                      t_k <= pistek(internal_counter);
42                      t_h <= pisteh(internal_counter);
43                      LED <= (others => '0');
44                      LED(internal_counter) <= '1';
45                      internal_counter<=15;
46                  end if;
47
48              end if;
49              end if;
50              last_cbpm <= cbpm;
51          end if;
52      end process;
53  end Behavioral;
```

Ce bloc utilise un compteur interne `internal_counter` pour générer le curseur temporel et ajuster les signaux `t_k`, `t_s`, et `t_h`. La valeur du curseur est utilisée pour sélectionner les éléments correspondants des vecteurs `pistek`, `pistes`, et `pisteh`. Le curseur `LED` est également généré en fonction de la position du curseur temporel.

2.5 Simulation du bloc gene_pis

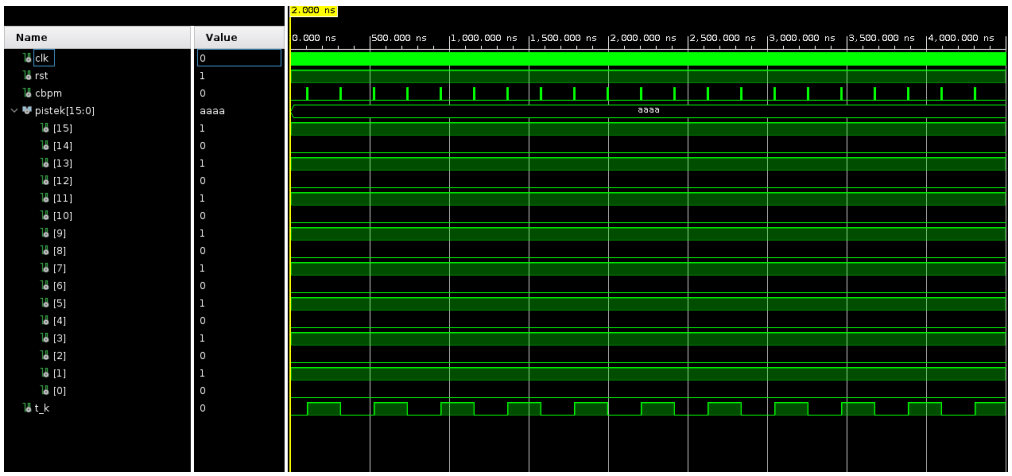

Figure 4: Simulation du bloc gene_pis

Ensuite, trois compteurs d'adresse (`cpt_addrk`, `cpt_addrs`, `cpt_addrh`) sont remis à zéro à chaque front montant de cbpm et s'incrémente à la fréquence d'échantillonnage de 44,1 kHz si le signal temporel correspondant est activé (`t_k`, `t_s`, `t_h`). Ces compteurs d'adresse génèrent le vecteur d'adresse `addr[17 downto 0]` qui est utilisé pour accéder aux blocs ROM (`wav_romk`, `wav_roms`, `wav_romh`) qui contiennent les sons kick, snare et hi-hat.

Le bloc `sum_sig` prend en entrée les signaux `val_k`, `val_s`, `val_h` qui représentent les valeurs échantillonnées des sons kick, snare, et hi-hat respectivement. Il utilise également les signaux temporels `t_k`, `t_s`, `t_h` pour déterminer quels sons sont actifs à un moment donné. Le bloc effectue une somme pondérée des signaux audio, ajustant le volume en fonction du nombre de sons actifs.

Le code VHDL du bloc `sum_sig` est le suivant :

```
1  library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity sum_sig is
    Port ( clk : in STD_LOGIC;
           Ce : in STD_LOGIC;
           t_s : in std_logic;
           t_k : in STD_LOGIC;
           t_h : in std_logic;
           val_k : in STD_LOGIC_VECTOR (10 downto 0);
           val_s : in STD_LOGIC_VECTOR (10 downto 0);
           val_h : in STD_LOGIC_VECTOR (10 downto 0);
           pwmin : out STD_LOGIC_VECTOR (10 downto 0));
end sum_sig;

architecture Behavioral of sum_sig is
    signal sum ,kf,sf,hf,add : INTEGER;
    signal vect11 : std_logic_vector(11 downto 0);
    signal vect12 : std_logic_vector(10 downto 0);
    signal vect13 : std_logic_vector(12 downto 0);

begin
    process(clk, Ce)
    begin
        if rising_edge(clk) then
            if Ce = '1' then
               if t_k ='1' then
                    kf <= to_integer(unsigned(val_k));
                else
                    kf <= 0;
                end if;
                if t_s ='1' then
                    sf <= to_integer(unsigned(val_s));
                else
                    sf <= 0;
                end if;
                if t_h ='1' then
                    hf <= to_integer(unsigned(val_h));
                else
                    hf <= 0;
                end if;
                if t_k='1' and t_s='1' and t_h='0' then
                    add <= sf + kf;
                    vect11 <= std_logic_vector(to_unsigned(add,12));
                    vect12 <= vect11(11 downto 1);
                    sum <= to_integer(UNSIGNED(vect12));
                elsif t_k='1' and t_s='0' and t_h='1' then
                    add <= hf + kf;
                    vect11 <= std_logic_vector(to_unsigned(add,12));
                    vect12 <= vect11(11 downto 1);
                    sum <= to_integer(UNSIGNED(vect12));
                elsif t_k='0' and t_s='1' and t_h='1' then
                    add <= sf + hf;
                    vect11 <= std_logic_vector(to_unsigned(add,12));
                    vect12 <= vect11(11 downto 1);
                    sum <= to_integer(UNSIGNED(vect12));
                elsif t_k='1' and t_s='0' and t_h='0' then
                    sum <= kf;
                elsif t_k='0' and t_s='1' and t_h='0' then
                    sum <= sf;
                elsif t_k='0' and t_s='0' and t_h='1' then
                    sum <= hf;
```

```
                elsif t_k='1' and t_s='1' and t_h='1' then
                    add <= sf + kf + hf;
                    vect13 <= std_logic_vector(to_unsigned(add,13));
                    vect12 <= vect13(12 downto 2);
                    sum <= to_integer(UNSIGNED(vect12));
                end if;
                -- Verifier si la somme depasse la taille maximale du vecteur de
                    sortie
                if sum > 2047 then
                    pwmin <= "11111111111"; -- Limite maximale pour 11 bits
                else
                    pwmin <= std_logic_vector(TO_UNSIGNED(sum,11));
                end if;
            end if;
        end if;
    end process;
end Behavioral;
```

Le bloc prend en compte les signaux temporels `t_k`, `t_s`, `t_h` pour déterminer quels sons sont actifs à un instant donné. En fonction de cela, il effectue la sommation pondérée des valeurs échantillonnées des trois sons (kick, snare, hi-hat) et ajuste le volume en conséquence. La sortie `pwmin` représente le signal audio final, et son amplitude est ajustée pour éviter tout dépassement de la plage définie par 11 bits. Ainsi, le bloc `sum_sig` contribue à la synthèse du signal audio combiné à partir des sons individuels.

3 Analyse des performances

3.1 Timing report

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	2,798 ns	Worst Hold Slack (WHS):	0,077 ns	Worst Pulse Width Slack (WPWS):	4,500 ns
Total Negative Slack (TNS):	0,000 ns	Total Hold Slack (THS):	0,000 ns	Total Pulse Width Negative Slack (TPWS):	0,000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	1295	Total Number of Endpoints:	1295	Total Number of Endpoints:	399

All user specified timing constraints are met.

Figure 5: Rapport des timings caractéristiques

Le design satisfait toutes les contraintes de timing avec un WNS de 2,798 ns et un WHS de 0,077 ns, garantissant l'absence de violation de timing à la fréquence de fonctionnement cible. Cela assure une marge adéquate pour les variations de processus, de tension et de température.

Pendant le projet, nous avons rencontré plusieurs problèmes de timing, avec un Worst Negative Slack (WNS) initial de -34 ns. Ce problème était dû à une opération mathématique complexe effectuée sur une seule ligne dans le bloc sum_sig présenté ci-dessus . En divisant cette opération en plusieurs parties sur plusieurs lignes, nous avons introduit des variables tampons, créant ainsi un pipeline. Ce processus avait pour but de répartir le calcul sur plusieurs périodes d'horloge, améliorant ainsi le WNS. Nous avons pris soin de réaliser ce décalage temporel sur nos trois signaux sonores afin de conserver une synchronisation. Cependant, en pratique, étant donné que les sons étaient échantillonnés à 44.1 kHz, une période d'horloge à 100 MHz n'aurait pas fait de différence perceptible pour l'utilisateur.

3.2 Utilization report

Site Type	Used	Fixed	Prohibited	Available	Util%
Slice LUTs	243	0	0	63400	0.38
LUT as Logic	243	0	0	63400	0.38
LUT as Memory	0	0	0	19000	0.00
Slice Registers	364	0	0	126800	0.29
Register as Flip Flop	364	0	0	126800	0.29
Register as Latch	0	0	0	126800	0.00
F7 Muxes	2	0	0	31700	<0.01
F8 Muxes	0	0	0	15850	0.00

Figure 6: Répartition des différents éléments logiques utilisés

L'utilisation des LUTs et des registres est modeste, avec seulement 0.38% des LUTs et 0.29% des registres utilisés, ce qui est justifié par l'implémentation de trois ROMs. Cette utilisation laisse une marge significative pour d'éventuelles extensions du design ou pour intégrer d'autres fonctionnalités sans compromettre les ressources disponibles.

3.3 Power report

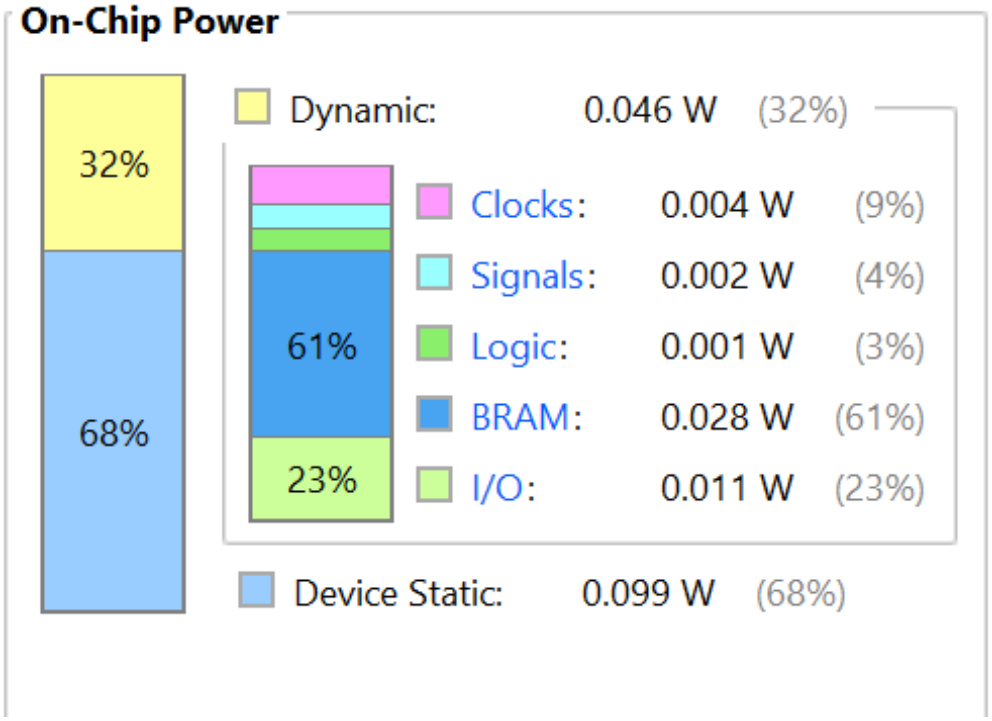

Figure 7: Rapport de la consomation du système

Le total de la puissance consommée sur la puce est de 0.145 W, indiquant une bonne efficacité énergétique pour le design implémenté. La puissance dynamique représente 32% de cette consommation, avec une utilisation prédominante de la mémoire BRAM. La puissance statique, représentant 68%, suggère une opportunité d'optimisation pour réduire encore la consommation énergétique globale.