Akademia Górniczo-Hutnicza Im. Stanisława Staszica w Krakowie
Wydział Informatyki Elektroniki i Telekomunikacji

Praca dyplomowa

Realizacja pamięci asocjacyjnej w technologii rekonfigurowalnych układów SoC

Implementation of the Associative Memory in Reprogrammable SoC Technology

Autor: Inż. Zachariasz Mońka
Kierunek: Elektronika i Telekomunikacja
Opiekun pracy: Dr hab. inż. Paweł Russek, Profesor AGH

Kraków 2024

Spis treści

Podziękowania

Chciałem zadedykować moją pracę wszystkim tym wokół mnie, którzy nie przestają zadawać pytań, a gdy już usłyszą odpowiedź, zawsze podchodzą do niej z dystansem i nieufnością.

„Dlaczego łyżwy ślizgają się po lodzie?”

„Po jakim czasie od przełączenia przełącznika zapali się światło?”

„Jak kalkulator oblicza pierwiastki?”

„Jak działa elektronika w kosmosie?”

„Skąd się wziął taki, a nie inny wzór na deltę?”

„Dlaczego niebo w nocy jest ciemne?”

„Dlaczego liczb wymiernych jest mniej niż niewymiernych?”

„Skąd się bierze kryptowaluta, którą można wykopać na swoim komputerze?”

„Dlaczego szerokość opon wpływa na drogę hamowania?”

„Jak działa wyłącznik różnicowo-prądowy?”

„Jak to jest z tym wszechświatem, że jest nieskończenie wielki?”

„Jak to jest w tym kosmosie, że wszystko tam jest i nie spada?”

„Czy umiesz mi wyjaśnić, skąd się bierze prąd?”

„Skąd się bierze kurz w pustych pomieszczeniach?”

„Dlaczego do mikrofalówki można wkładać niektóre metalowe przedmioty?”

Na te pytania nie udzielę odpowiedzi w mojej pracy. Niemniej już od samego początku jej pisania starałem się, aby wyjaśniała wszystkie zjawiska w sposób zrozumiały również dla Was.

Rozdział 1

Wstęp

1.1 Czym właściwie jest pamięć

Pytanie o to, czym jest pamięć może wydawać się trywialne. Uważam jednak, że warto się tutaj zatrzymać na chwilę i uporządkować terminologię w celu lepszego zrozumienia dalszych rozdziałów.

W najogólniejszym ujęciu pamięć to zdolność przechowywania i przywoływania informacji. Zgodnie z tą definicją pamięcią można nazwać, zarówno fragment żywego organizmy, książkę a nawet różne materiały których to mechaniczne odkształcenie może przechowywać informacje. W mojej pracy będę odnosił się jedynie do pamięci elektronicznej, czyli takiej, gdzie informacja jest zapisywana i odczytywana za pomocą sygnałów elektrycznych.

Mimo bardzo wielu rodzajów pamięć (elektronicznych) [1], ich działanie najczęściej jest bardzo podobne. Pamięć [2] nie przekazuje stale informacji, która jest w niej zapisana. Informuje o niej dopiero gdy zostanie niejako "zapytania". Proces "pytania" o jej poszczególne komórki nazwany jest adresacją. W większości rodzajów pamięci adresowanie wygląda tak samo: podajemy adres będący numerem komórki i otrzymujemy w rezultacie informacje (zwaną często danymi) będącą uprzednio zapisaną pod tym konkretnym adresem. W przypadku pamięci z których to można nie tylko doczytywać ale również zapisywać do nich dane, adresowanie najczęściej odbywa się za pomocą tych samych adresów, które to są wykorzystywane do odczytywania.

Załóżmy, że blok naszej pamięci składa się z szesnastu komórek ośmiobitowych [3]. Jeśli chcemy zapisać do takiej pamięci pewne słowo, przykładowo "TEST" możemy to zrobić używając kodowanie ASCII [4], które przyporządkowuje każdej literze konkretną wartość liczbową. Następnie zapisać wszystkie znaki w kolejnych komórkach pamięci, co

zostało przedstawione w tabeli 1.1. Widać w niej że każda z komórek pamięci ma swój adres, począwszy od 0x00, czyli liczby 0 zapisanej w systemie szesnastkowym (zwanym też heksadecymalnym) [5]. Komórki, w których nie zostały zapisane żadne dane, zawierają wartość 0xFF, a ostatnia komórka napisu zawiera 0x00, czyli znak "NULL" [6], oznaczający koniec napisu. Jest to bardzo popularna konwencja zapisywania danych. Oczywiście nie jest to jedyny sposób zapisu dany, lecz do celów przykładowych to właśnie ten sposób, przez swoją prostotę posłuży w dalszym omawianiu pamięci.

Adres [hex]	Dane [bin]	Dane [hex]	Dane [ASCI]
0x00	01010100	0x54	'T'
0x01	01000101	0x45	'E'
0x02	01010011	0x53	'S'
0x03	01010100	0x54	'T'
0x04	00000000	0x00	NULL
0x05	11111111	0xFF	
0x06	11111111	0xFF	
0x07	11111111	0xFF	

Tabela 1.1: Przykładowa zawartość pamięci

Warto jeszcze dodać, że opisany powyżej rodzaj pamięci zwykle nazywany jest po prostu jak pamięć RAM [7] (ang. random-access memory). Jak sama nazwa wskazuje, jest to pamięć, w której można odczytywać dane w dowolnej kolejności, w odróżnieniu od pamięci o dostępie sekwencyjnym, na przykład rejestrów przesuwnych.

1.2 Czym jest pamięć asocjacyjna

Pamięci można podzielić [1] na wiele rodzajów w zależności od tego jak są przechowywane informacje, metody jej odczytywania oraz zastosowań. Pomimo tych różnić wszystkie one działają na podobnej zasadzie - czego nie można z pewnością powiedzieć o pamięci asocjacyjnej. Pamięć asocjacyjna [8], zwana również [9] [10] pamięcią skojarzeniową, pamięcią adresowaną zawartością oraz CAM (od ang. content-addressable memory) to specyficzny rodzaj pamięci w której to występuje, odwrócona kolej rzeczy. W przypadku tej pamięci nie odwołujemy się do jej komórek za pomocą ich adresów, w rezultacie otrzymując dane, lecz za pomocą ich zawartości (danych) a w rezultacie otrzymując adresy.

Przykładowo, jeśli zapiszemy w pamięci CAM napis "TEST", możemy szybko dowiedzieć się, na której pozycji znajduje się liter "E". Wystarczy podać tą literę na wejście adresowe, a pamięć zwróci jej adres. Czyli w naszym przypadku komórkę 0x01[1].

Na tym etapie można by stwierdzić że każdą pamięć można nazwać pamięcią adresową zawartością, jeśli tylko aplikacja z niej korzystająca przeszukuje jej zawartość. W dalszej części mojej pracy będę się jedynie odnosił do pamięci, które mają "sprzętowo" zaimplementowany proces wyszukiwania. Proces "szukania" w takiej pamięci zajmuje tyla samo czasu lub czas porównywalny z czasem zapisu. Większość źródeł [11] podaje że pamięć asocjacyjna zwraca adres po jednym takcie zegara. W mojej pracy zdecydowałem się jednak na rozsądne wydłużanie tego czasu, aby porównać klasyczne implementacje z nieco wolniejszymi i aby przedstawić zalety również tych wolniejszych rozwiązań.

1.3 Czym są re-konfigurowalne układy SoC

Jeśli już wiemy, czy jest pamięć asocjacyjna, to warto również powiedzieć czym są konfigurowalne układy SoC. I dlaczego zdecydowałem się na realizowanie pamięci właśnie przy ich użyciu.

Terminem System on Chip (SoC) określa się kompletny system elektroniczny, łączący w jednym układzie scalonym wszystkie lub większość potrzebnych do działania komponentów. Systemy takie zawierają podzespoły takie jak mikroprocesory, bloki pamięci, układy cyfrowo-analogowe, układy analogowe (w tym radiowe) oraz programowalne układy bramek logicznych. To właśnie te ostatnio układy będą w mojej pracy szczególnie interesujące. Pozwalają one na tworzenie dowolnych układów logicznych opartych o bramki logiczne[12][13]. Układy takie zachowują się tak jak by mieściły w sobie wiele bramek których połączenia są definiowane przez nas w takcie programowania układu. W rzeczywistości rozwiązania takie jak FPGA[14] (ang. field-programmable gate array) zawierają wiele połączonych ze sobą pamięci LUT[15] które to przechowują tablice prawdy definiujące funkcje logiczne. Rozwiązanie takie jest bardziej wydajne niż stosowanie konfigurowalnych połączeń pomięć bramkami.

Układy SoC, pomimo tego, że za pomocą bramek logicznych można stworzyć dowolną logikę cyfrową, posiadają również klasyczne bloki cyfrowe, takie jak procesory. Wynika to z faktu, iż koszt tworzenia dużych struktur logicznych za pomocą układów FPGA wymaga więcej zasobów krzemowych niż stworzenie dedykowanego układu (zwanego ASIC), który spełnia konkretną funkcję i nie musi być konfigurowany. Fakt ten

[1]Warto zauważyć że pamięć zwróci 0x01, a nie 0x02 ponieważ zaczynamy numeracje od 0x00

sprawia, że tak popularne stało się łączenie ze sobą procesorów (wykonujących standardowe zadania) z układami programowalnymi, które mogą przyjmować formy bardziej specjalistycznych układów, takich jak pamięć CAM.

1.4 Zawartość pracy

Wszystkie przedstawione w tej pracy kody źródłowe można znaleźć w repozytorium GitHub pod adresem:

github.com/ZachariaszMonka/Master-thesis

Zawartość pracy składa się z plików VHDL przedstawiających konkretne metody implementacji oraz testów jednostkowych do tych plików. W tabeli 1.2 przedstawiłem spis plików z podziałem na implementacje w nich zastosowane.

Nazwa pliku	Rozdział	Rodzaj implementacji	Rodzaj pliku
basic_ram	3.1.1	Pamięć RAM	Plik źródłowy
basic_ram_tb	3.1.3	Pamięć RAM	Test manualny
basic_cam	3.1.2	Najprostsza implementacja	Plik źródłowy
basic_cam_tb	3.1.3	Najprostsza implementacja	Test manualny
basic_tb	3.1.4	Najprostsza implementacja	Test automatyczny
basic_ram_without_generic	3.1.4	Najprostsza implementacja	Alternatywna wersja
basic_bug1_tb	4.2	Najprostsza implementacja	Plik z błędem
inv_cam	3.2.2	Odwrotna pamięć	Plik źródłowy
inv_cam_tb	3.2.3	Odwrotna pamięć	Test manualny
inv_tb	3.2.3	Odwrotna pamięć	Test automatyczny
dual_cam	3.3	Podwójna pamięć	Plik źródłowy
dual_cam_tb	3.3.2	Podwójna pamięć	Test manualny
dual_tb	3.3.2	Podwójna pamięć	Test automatyczny
dual_bug2_tb	4.3	Podwójna pamięć	Plik z błedem
crc	3.4	Wyznaczanie CRC	Moduł
crc_cam	3.4	Skrót CRC w CAM	Plik źródłowy
crc_cam_tb	3.4.5	Skrót CRC w CAM	Test manualny
crc_m_cam	3.4	Złożona wersja z CRC	Plik źródłowy
crc_m_cam_tb	3.4.5	Złożona wersja z CRC	Test manualny
cuckoo_cam	3.5	Algorytm cuckoo	Plik źródłowy
cuckoo_cam_tb	3.5.3	Algorytm cuckoo	Test manualny

Tabela 1.2: Zawartość pracy

Rozdział 2

Założenia

2.1 Jaki jest cel stawiania założeń?

Korzystając z mojego kilkuletniego doświadczenia zawodowego jako programisty układów FPGA pracujących w przestrzeni kosmicznej, mogę stwierdzić, że rozpoczęcie jakiejkolwiek implementacji dobrze jest zacząć od postawienia założeń. Pozwala to znacząco uporządkować pracę i nadać jej kierunek. Zauważyłem, że bardzo często można zaobserwować sytuacje, gdy ktoś próbuje rozwiązać jakiś problem aż do momentu, gdy ktoś inny zapyta "co tak właściwie chcesz osiągnąć?". Okazuje się wtedy, że dobre zdefiniowanie problemu stanowi już niespodziewanie dużą cześć jego rozwiązanie.

Dlatego też projekt ten chciałbym rozpocząć od zdefiniowania założeń, których będę się starał trzymać w granicach rozsądku do samego końca implementacji, aby na koniec móc dumnie do nich wrócić i ocenić ile udało się osiągnąć. Oczywiście zdaje sobie sprawę, że bardzo często założenia postawione na początku projektu okazują się być nierealnymi do wykonania przy obecnych zasobach (zarówno czasowych, sprzętowych, finansowych jak i intelektualnych). Problem ten jest tym bardziej widoczny im bardziej rewolucyjny ma być nasz projekt i stanowi wręcz jeden z klasycznych scenariuszy w zespołach R&D[1]. Mimo tego nie chcę modyfikować tego rozdziału w zależności od wyników mojej pracy. Chciałbym na koniec sprawiedliwie podsumować, co udało się osiągnąć, a czego nie, a także zastanowić się, dlaczego.

[1]R&D to skrót od ang. "Research and Development" i oznacza działania polegające głównie na wdrażaniu nowych technologi. Zwykle są to technologie nie wykorzystywanych jeszcze nigdy w przemyśle lub nie w tej konkretnej formie.

2.2 Język VHDL

Czym są układy SoC oraz układy FPGA już wiemy z rozdziału 1.3. Teraz warto się zastanowić w jakim języku najlepiej programować takie układy. Dwoma głównymi (a można by nawet powiedzieć, że jedynymi) językami służącymi do opisu sprzętu są VHDL [16] oraz Verilog[17]. Warto dodać, że drugi z nich występuje również w rozbudowanej wersji[18] System Verilog [17].

Różnice [19] miedzy tymi językami z pewnością nie są pomijalne. Podobnie jak przy porównywaniu innych języków, cecha, która w jednej aplikacji może być wadą, w innej może okazać się zaletą. Na przykład brak określonego mechanizmu może spowalniać tworzenie aplikacji, ale często mniejsza ilość mechanizmów ułatwia analizę i kontrolę nad aplikacją, co sprawia, że trudno jednoznacznie wskazać, który język jest lepszy.

Patrząc na problem wyboru języka, można przyjąć podejście, w który zostaną wykorzystane oba języki oraz porównane ich wyniki. Jednak nie spodziewam się znaczących różnić pomiędzy wynikami, a na pewno będą one pomijalne w porównaniu do różnic wynikających z rożnych metod implementacji.

Innym podejście byłoby stosowanie różnych języków w zależności od tego, który się bardziej nadaje do danej aplikacji. Uważam jednak, że takie rozwiązanie jest bardzo mało czytelne, nieprofesjonalne i co najważniejsze, utrudnia łączenie fragmentów kodu.

Reasumując, uważam, że wybór języka w mojej pracy nie wpłynie znacząco na konkretne wyniki; ważna jest jedynie konsekwencja stosowania tego samego języka przez całą pracę. Kierując się osobistą sympatią do języka VHDL zdecydowałem się przedstawiać wszystkie fragmenty kodu właśnie w nim.

2.3 Parametryzacja wielkość pamięci

W trakcie tworzenie oprogramowania bardzo ważną cechą jest utrzymywanie kodu w czytelnej i łatwo skalowalnej formie. Nie zamierzam tutaj szczegółowo omawiać kwestii czytelności, ponieważ uważam, że jest to oczywiste dla każdego projektu. Chciałbym jednak poruszyć temat skalowalności. Wyobraźmy sobie sytuacje, w której stworzyliśmy już blok pamięci, ale okazuje się, że potrzebujemy go w nieco większym rozmiarze. Wówczas możemy skopiować nasz projekt i na podstawie mniejszego bloku stworzyć większy. Takie rozwiązanie ma jednak niestety wiele wad. Nie tylko musi szukać za każdym razem w kodzie miejsc, w których należy zmieniać jego rozmiar, ale także muśmy musimy tworzyć oddzielne pliki dla każdej wielkości pamięci zamiast mieć jeden plik, który mógłby być uruchamiany z różnymi parametrami. Ostatnim, lecz paradoksalnie

największym problemem jest to, że gdy zmieniamy rozmiar pewnej stałej w kodzie, często nie mamy pewności, czy autor kodu przewidział takie modyfikacje. W ten sposób łatwo natrafić na przypadek, który nie został prawidłowo przetestowany. Przykładowo, możemy zaimplementować tak dużą pamięć, że ustalona na sztywno przestrzeń adresowa nie będzie jej obsługiwać.

Rozwiązaniem tych problemów jest mechanizm języka VHDL o nazwie "Generic" [20][21]. Umożliwia on na zdefiniowanie szerokości wejść oraz wyjść za pomocą parametrów podawanych podczas deklaracji nowego obiektu. W celu porównanie zostały przedstawione dwa fragmentu kodu, przedstawiające deklaracje portów; jeden z użyciem mechanizmu "generic", a drugi z bezpośrednim wpisaniem wartości w każdym miejscu kodu oddzielnie.

Listing 2.1: `basic_ram_without_generic.vhd` przykład portów

```
19      port (
20          clk      : in  std_logic;
21          w_en     : in  std_logic;
22          w_addr   : in  std_logic_vector(1 downto 0);
23          w_data   : in  std_logic_vector(7 downto 0);
24          r_addr   : in  std_logic_vector(1 downto 0);
25          r_data   : out std_logic_vector(7 downto 0)
26      );
```

Listing 2.2: `basic_ram.vhd` przykład zastosowania generic

```
19      generic (
20          DATA_WIDTH : integer := 8;
21          ADDR_WIDTH : integer := 2
22      );
23      port (
24          clk      : in  std_logic;
25          w_en     : in  std_logic;
26          w_addr   : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
27          w_data   : in  std_logic_vector(DATA_WIDTH-1 downto 0);
28          r_addr   : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
29          r_data   : out std_logic_vector(DATA_WIDTH-1 downto 0)
30      );
```

2.4 Różne metody implementacji

Zdecydowanie najważniejszym założeniem mojej pracy jest przedstawienie w niej nie jednej, lecz wielu różnych implementacji pamięci asocjacyjnej. Mam nadzieję, że w trakcie pracy uda się wyłonić najbardziej efektywne podejście lub przynajmniej określić, które z nich lepiej sprawdzają się w konkretnych zastosowaniach.

Rozdział 3

Przegląd rozwiązań

3.1 Wykorzystanie komparatorów

3.1.1 Implementacja RAM

Analogicznie jak we wstępie do rozdziale 1.1 najpierw została przedstawiona pamięć RAM, aby na jej podstawie omówić pamięć CAM. Podobnie tutaj, najpierw zaprezentuje implementacje pamięci RAM. Takie rozwiązanie będzie zarówno bardziej czytelne, jak i praktyczne, bo implementacja pamięci RAM okaże się później przydatna do testowania.

Implementacja pamięci RAM została wykonana w możliwe najprostszy sposób, przy jednoczesnym zachowaniu założeń przedstawionych w rozdziale 2.3. Oznacza to, że wielkość bloku pamięci nie jest stała, lecz może być modyfikowana za pomocą mechanizmu "generic"[20] [21].

Zdecydowałem się na zastosowanie dwóch osobnych portów (oznaczanych `w_*` oraz `r_*`) dla zapisywania i odczytywania danych. Uważam, że takie rozwiązanie jest bardziej czytelne i ułatwia implementację. Jeśli jednak aplikacja wymagałaby, aby pamięć posiadała jeden port z możliwością przełączania między zapisem a odczytem, można łatwo stworzyć odpowiedni adapter.

Listing 3.1: `basic_ram.vhd` opis wejść i wyjść

```
19      generic (
20          DATA_WIDTH : integer := 8;
21          ADDR_WIDTH : integer := 2
22      );
23      port (
24          clk      : in  std_logic;
25          w_en     : in  std_logic;
26          w_addr   : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
27          w_data   : in  std_logic_vector(DATA_WIDTH-1 downto 0);
28          r_addr   : in  std_logic_vector(ADDR_WIDTH-1 downto 0);
29          r_data   : out std_logic_vector(DATA_WIDTH-1 downto 0)
30      );
```

Listing 3.2: `basic_ram.vhd` Behavioral

```
33  architecture Behavioral of basic_ram is
34      type memory_array is array (0 to 2**ADDR_WIDTH-1) of std_logic_vector(DATA_WIDTH-1 downto 0);
35      signal mem : memory_array := (others => (others => '0'));
36  begin
37
38      write_process : process(clk)
39      begin
40          if rising_edge(clk) then
41              if w_en = '1' then
42                  mem(to_integer(unsigned(w_addr))) <= w_data;
43              end if;
44          end if;
45      end process;
46
47      read_process : process(clk)
48      begin
49          if rising_edge(clk) then
50              r_data <= mem(to_integer(unsigned(r_addr)));
51          end if;
52      end process;
53  end Behavioral;
```

3.1.2 Implementacja CAM

Zgodnie z zasadą, że pierwsza implementacja powinna być możliwie najprostsza, aby stanowiła bazę do dalszego rozwoju, pamięć CAM w swojej pierwszej iteracji nie zawiera praktycznie żadnych dodatkowych funkcji.

Porty pamięci CAM właściwe nie różni się za wiele od pamięci RAM przedstawionej w rozdziale 3.1.1. Jak pokazano w listingu 3.3 jedyną różnicą jest kierunek portów `r_data` oraz `r_addr`. Wynika to z faktu, że w przypadku pamięci CAM nie podajemy adresu, lecz zawartość, aby uzyskać odpowiedni adres.

Implementacja w stylu Behavioral jest bardzo zbliżona do implementacji pamięci RAM z lisningu 3.2. Procedura zapisu (przedstawiona na listingu 3.4) nie różni się od tej wykorzystanej w pamięci RAM, ponieważ zapis odbywa się w ten samo sposób. Różnica natomiast jest zauważalna w części odpowiedzialnej za odczyt z pamięci (listing 3.5).

Listing 3.3: `basic_cam.vhd` opis wejść i wyjść

```
19      generic (
20          DATA_WIDTH : integer := 8;
21          ADDR_WIDTH : integer := 4
22      );
23      port (
24              clk             : in  std_logic; -- comon clock for in and out
25              w_en            : in  std_logic; -- '1' write is active
26              w_addr          : in  std_logic_vector(ADDR_WIDTH - 1 downto 0);
27              w_data          : in  std_logic_vector(DATA_WIDTH - 1 downto 0);
28              r_addr          : out std_logic_vector(ADDR_WIDTH - 1 downto 0);
29              r_data          : in  std_logic_vector(DATA_WIDTH - 1 downto 0)
30      );
```

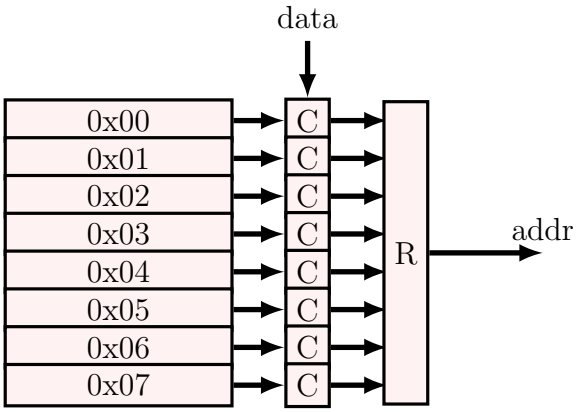
Listing 3.4: `basic_cam.vhd` opis procedury zapisu

```
33  architecture Behavioral of basic_cam is
34      type memory_array is array (0 to 2**ADDR_WIDTH-1) of std_logic_vector(DATA_WIDTH-1 downto 0);
35      signal      mem         : memory_array := (others => (others => '0'));
36  begin
37
38      write_process : process(clk)
39      begin
40          if rising_edge(clk) then
41              if w_en = '1' then
42                  mem(to_integer(unsigned(w_addr))) <= w_data;
43              end if;
44          end if;
45      end process;
```

Listing 3.5: `basic_cam.vhd` opis procedury odczytu

```
47      read_process : process(clk)
48          variable   detected_data : std_logic_vector(2 ** ADDR_WIDTH downto 0) := (others => '0');
49      begin
50          if rising_edge(clk) then
51              for addr in 0 to 2**ADDR_WIDTH - 1 loop
52                  if r_data = mem(addr) then
53                      detected_data(addr) := '1';
54                  else
55                      detected_data(addr) := '0';
56                  end if;
57              end loop;
58             r_addr <= (others => '0');
59             for addr in 0 to 2**ADDR_WIDTH - 1 loop
60                if detected_data(addr) = '1' then
61                    r_addr <= std_logic_vector(to_unsigned(addr, r_addr'length));
62                end if;
63             end loop;
64          end if;
65      end process;
66  end Behavioral;
```

Odczyt z pamięci asocjacyjnej odbywa się w dwóch etapach (przedstawionych na diagramie 3.1). Pierwszym etapem jest porównywanie każdej komórki pamięci z wprowadzonymi danymi. Odpowiada za to blok "C" z diagramu 3.1 oraz linie kody 52-58 z listingu 3.5. W rezultacie każda komórka pamięci, w której znajduje się poszukiwana zawartość, ustawia flagę (wektor `detected_data`). Następnie wszystkie ustawione flagi są porównywane przez blok przedstawiony na diagramie 3.1 jako "R" a w listingu 3.5 jako linie 59-64.



Rysunek 3.1: Schemat prostej pamięci CAM

Warto zastanowić się tutaj nad dwoma ciekawymi przypadkami. Pierwszym z nich jest sytuacja, w której żadna komórka pamięci nie zawiera "poszukiwanych" danych. W takim przypadku pamięć zwróci adres 0x00, za co jest odpowiedzialna linia kodu 59.

W bardziej zaawansowanych rozwiązaniach, można informować użytkownika, czy adres 0x00 oznacza, że poszukiwany zestaw danych znajduje się właśnie w nim, czy też że dane nie zostały znalezione. Jednak w najprostszej implementacji zdecydowałem się nie komplikować kodu tym mechanizmem.

Drugim ciekawym scenariuszem jest sytuacja, gdy więcej niże jedna komórka pamięci zawiera takie same dane. W takim przypadku zostanie zwrócony najmniejszy z znalezionych adresów. Podobnie jak w poprzednim przypadku, również tutaj istnieje spore pole do ulepszenia interfejsu użytkownika oraz informowania go o ilości znalezionych adresów.

3.1.3 Testy manualne

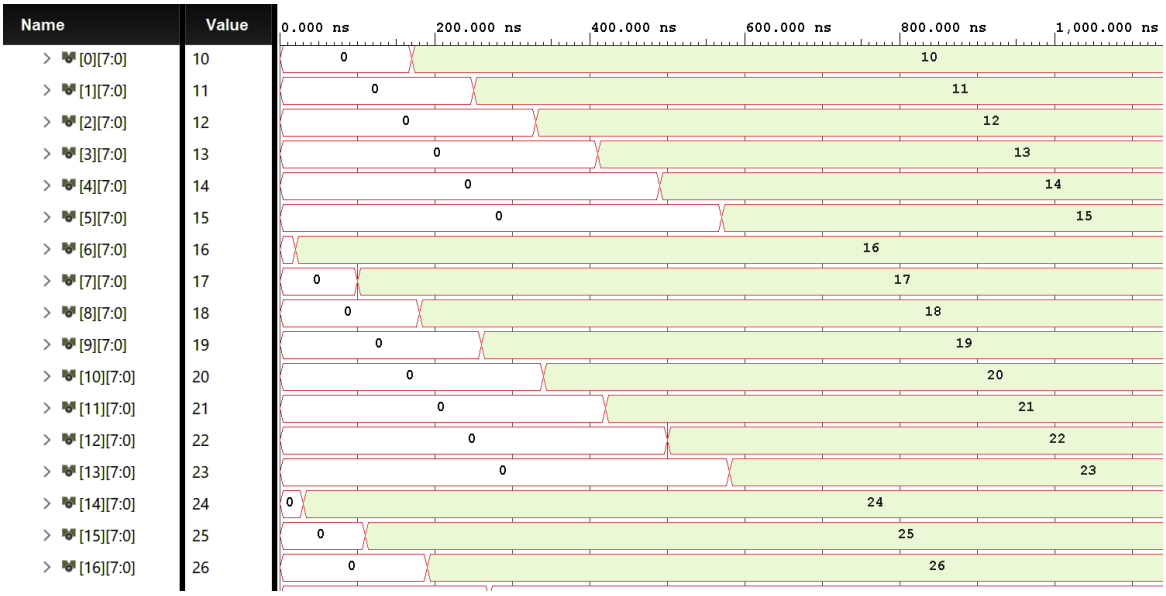
Pomimo istnienia bardzo wielu narzędzi do testowania kodu w VHDL, uważam, że warto aby każdy moduł vhdl posiadał swój własny plik testowy, często zwany "test bench". Zazwyczaj nosi on nazwę taką samą jak plik, który testuje z dopiskiem "_tb". Testy dotyczące poszczególnych plików nie mogę zwykle zastąpić złożonych testów integracyjnych, jednak są znacznie wygodniejsze przy testowaniu konkretnych scenariuszy. Rozbudowane zestawy testów automatycznych[1] zostaną jeszcze tutaj omówione, ale bardzo ważne jest, aby w razie potrzeby móc przedstawić działanie konkretnych modułów, nawet w sposób manualny, gdy pojawią się wątpliwości.

Listing 3.6: "losowanie" adresów w basic_cam_tb.vhd

```
79          --save all cell in memory by (number of cell + 10)
80          for i in 0 to 2**ADDR_WIDTH - 1 loop
81              if  i = 2**ADDR_WIDTH - 1 then
82                  write_addr := 2**ADDR_WIDTH - 1;
83              else
84                  write_addr := (write_addr + 2 ** ADDR_WIDTH / 8 )mod(2 ** ADDR_WIDTH - 1);
85              end if;
86              write_data := (write_addr + 10)mod(2 ** DATA_WIDTH);
87              w_en    <= '1';
88              w_addr  <= std_logic_vector(to_unsigned(write_addr, w_addr'length));
89              w_data  <= std_logic_vector(to_unsigned(write_data, w_data'length));
90              mem_tb(write_addr) <= std_logic_vector(to_unsigned(write_data, w_data'length));
91              wait for clk_period;
```

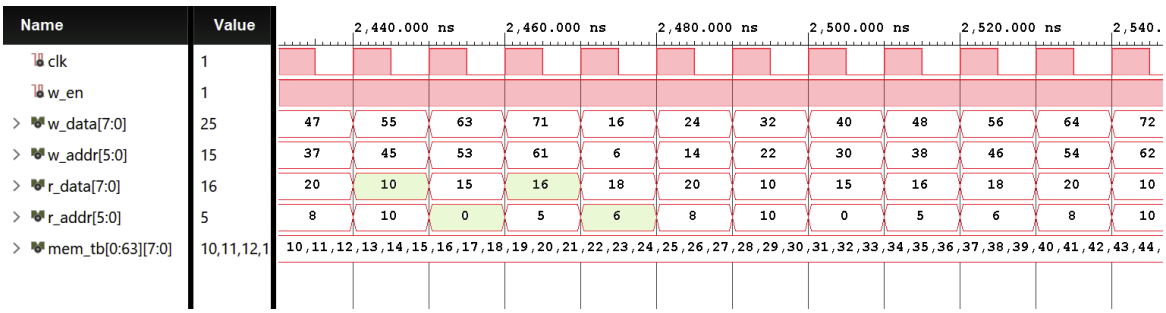
[1]Testy automatyczne (w odróżnieniu od manualnych) to takie które nie wymagają ręcznego sprawdzania wszystkich parametrów lecz pokazują na koniec w konsoli czy test przeszedł pozytywnie

Na listingu 3.6 przedstawiono oryginalną metodę zapisywanie kolejnych komórek pamięci. Polega ona na zapisywaniu co ósmej komórki, wykorzystując operację modulo. Takie podejście pozwala na wprowadzenie pozornej losowości zapisywania danych, co umożliwia przetestowanie czy zapisywanie losowych komórek pamięci działa prawidłowo. Kolejnym zastosowanym ułatwieniem jest zapisywanie do komórek pamięci wartości innej od adresu komórki (aby poprawnie przetestować pamięć), ale mającej prostą zależność liczbową z adresem. Przykładowo jest to wartość adresu powiększoną o dziesięć. Metoda ta pozwala na bardzo wygodne testowanie - programista może łatwo dodać dziesięć do adresu, by sprawdzić czy zawartość jest prawidłowa. Jak to również przydatne dla algorytmu, który nie musi pamiętać jaka wartość została zapisana do danej komórki (pomimo pseudolosowej kolejności zapisu), ponieważ może ją zawsze wyliczyć na podstawie adresu.



Rysunek 3.2: Symulacja procesy zapisywania do pamięci CAM

Rezultat zapisu został przedstawiony na rysunku 3.2. Dla zwiększenia czytelności przebiegów dane zapisane zostały pokolorowane, natomiast dane jeszcze nie zapisane pozostawiono w kolorze białym.



Rysunek 3.3: Rezultat symulacji basic_cam

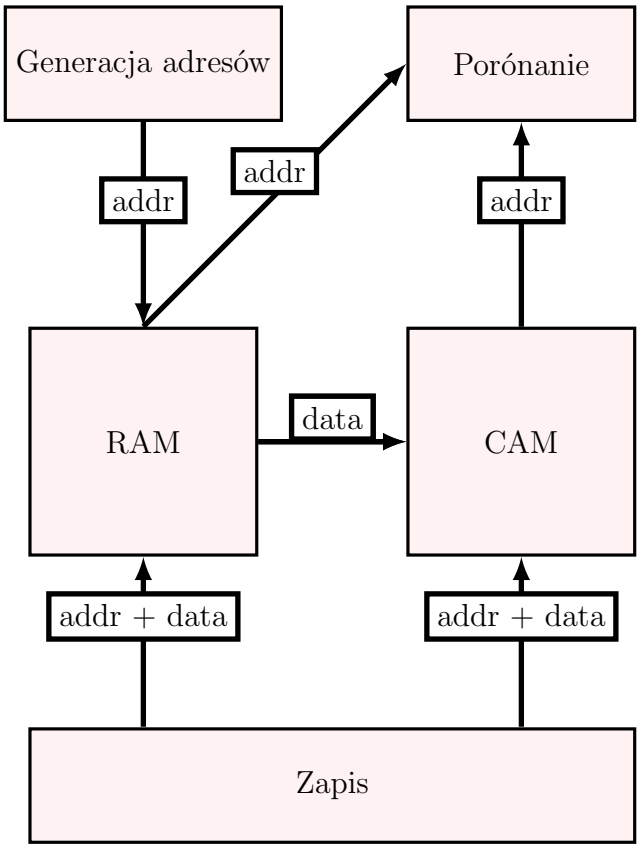
W przypadku testów manualnych rezultaty zapisywania i odczytywania pamięci można sprawdzić jedynie za pomocą ręcznego analizowania przebiegów. Rysunek 3.3 przedstawia fragment symulacji. Widać na nim, że po podaniu na wejście danych wartości `16` pamięć CAM zwraca `6` czyli adres zawierający te dane. Podobnie sytuacja wygląda dla zestawu danych-adresów `10` i `0`.

3.1.4 Testy automatyczne

Pomimo dużej wygody w tworzeniu testów manualnych, ich uruchomienie jest mało wygodne, ponieważ wymagają ciągłego monitorowania wszystkich sygnałów przez programistę. Rozwiązaniem tego problemu są testy automatyczne, które to po uruchomieniu prezentują wynik testu. Podejście takie pozwala na przeprowadzanie testów po każdej większej zmianie i sprawdzaniu, czy program nadal działa poprawnie mimo wprowadzonych modyfikacji.

W celu wykonania testów automatyczny zdecydowałem się na jednoczesne testowanie pamięci RAM oraz CAM. Alternatywnie, zakładając, że pamięć RAM jest dobrze przetestowana, można testować pamięci CAM wykorzystując pamięci RAM jako odniesienie. Mechanizm działania został przedstawiony na rysunku 3.4. Widać na nim że do obu pamięci zapisywane są te same dane. Gdy cała pamięć zostanie zapisana testowymi danymi, rozpoczyna się proces testowania, który polega na podawaniu losowych adresów do pamięci RAM. Pamięć ta zwraca dane zawarte pod danym adresem, które następnie trafiają na wejście pamięci CAM. Prawidłowo działająca pamięć CAM powinna zwrócić adres, pod którym znajdują się dane, będący tym samym adresem, który został podany do pamięci RAM. Taki proces umożliwia łatwe testowanie poprzez sprawdzanie czy adres podany na wejście pamięci RAM jest taki sam jak ten otrzymany na wyjściu pamięci CAM. Warto jedna zauważyć, że podczas porównywania należy

zestawić bieżące wyjście pamięci CAM z wejście pamięci RAM opóźnionym o dwa takty zegara. Opóźnienie to wynika z czasu operacji obu pamięci.

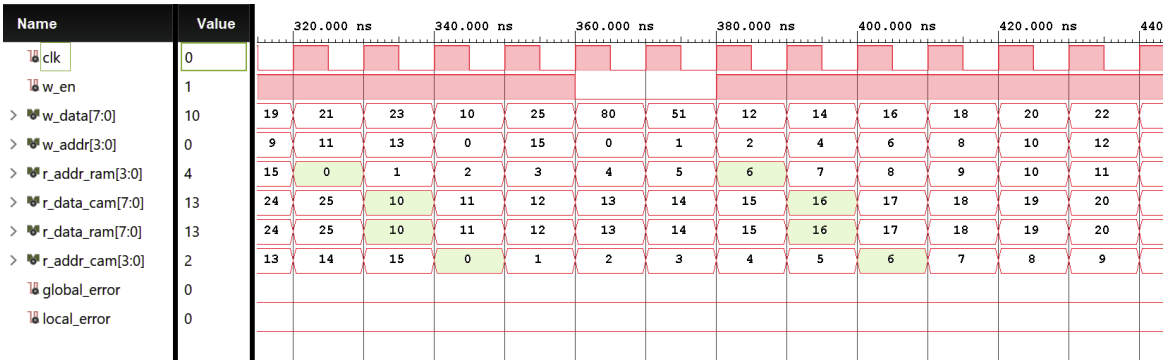


Rysunek 3.4: Schemat testów automatycznych

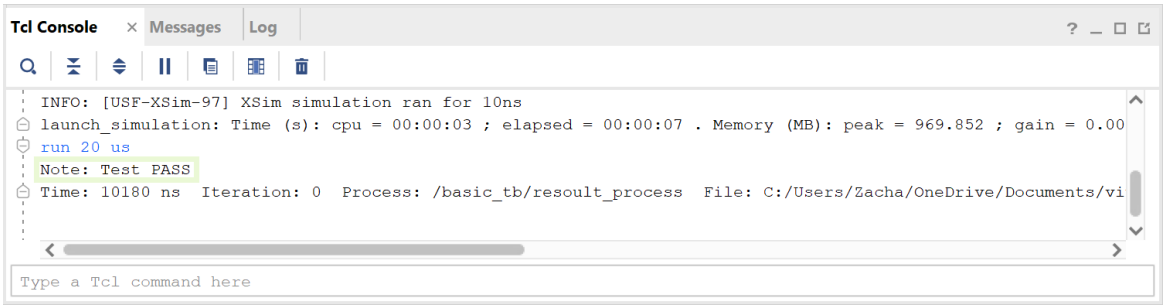
Rezultat testów automatycznych został przedstawiony na rysunku 3.5. Widać na nim prawidłowe działanie pamięć RAM oraz CAM. W odróżnieniu od testów manualnych, w tym przypadku programista nie musi ręcznie analizować przebiegów. Sygnał `local_error` przyjmuje stan wysoki za każdym razem, gdy pamięć zadziała w sposób nieprawidłowy. Aby dodatkowo ułatwić obsługę symulacji (oraz zapobiec przeoczeniu błędu), sygnał `global_error` zatrzaskuje[2] wykryty błąd do końca symulacji.

W celu jeszcze łatwiejszego analizowanie rezultatów zdecydowałem się na informowaniu o rezultacie testów w konsoli programu vivado, co przedstawiono na rysunku3.6.

[2]zatrzaskiwanie to proces w którym raz ustawiony sygnał nie jest modyfikowany aż do resetu



Rysunek 3.5: Rezultat symulacji basic_tb



Rysunek 3.6: Rezultat symulacji automatycznej basic_tb

Pomimo tak licznych zalet rozwiązania jakim są automatyczne testy, rozwiązanie to nie może zostać zastosowane w każdym przypadku. Lecz ten problem zostanie serzej przedstawiony dopiero w rozdziale 4.3.

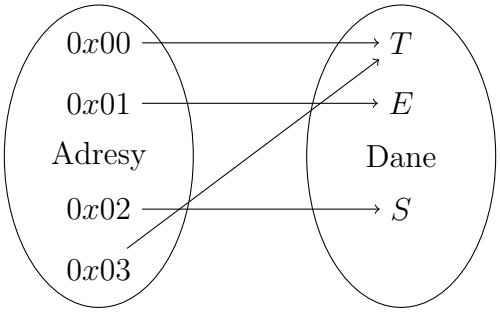
3.2 Metoda zamiany portów

3.2.1 Teoria - Iniekcyjna

Jeszcze przed implementacją kolejnych rozwiązań chciałbym na chwilę zatrzymać się i poruszyć problem, który pojawia się, gdy dwa adresy zawierają te same dane. Problem taki nie jest aż tak abstrakcyjny, jak może się wydawać i występuje nawet w przykładzie z rozdziału 1.1, gdzie to w tabeli 1.1 można zauważyć, że znak 'T' znajduje się w dwóch różnych komórkach.

Rozwiązaniem takiego problemu może być zwracanie najniższego adresu zawierającego "poszukiwane" dane, tak jak to ma miejsce w przypadku pamięci `basic_cam.hdl` przedstawionej w rozdziale 3.1.2. Takie podejście uniemożliwia poznanie adresów kolejnych wystąpień tych danych. Jest to forma zabezpieczenia przed nieprzewidzianą sytuacją w systemie, w którym nie spodziewamy się zapisywać tych samych danych do rożnych komórek pamięci.

Podchodząc do tematu bardzo teoretycznie, można by potraktować pamięć RAM jako funkcje matematyczną[22], czyli przyporządkowanie każdemu adresowi (z przestrzeni adresowej, czyli dziedziny naszej funkcji) pewnej wartości liczbowej będącą daną (z przeciwdziedziny). Przykład takiej funkcji matematycznej został przedstawiony na diagramie z rysunku 3.7, gdzie każdy adres ma przyporządkowaną dokładnie jedną daną (nie oznacza to jednak, że każda dana znajduje się tylko pod jednym adresem).



Rysunek 3.7: Przykład funkcji matematycznej

Pozostając przy analogi pamięci RAM do funkcji matematycznych, można zdefiniować funkcję odwrotną będącą również funkcją (czyli w naszym przypadku pamięcią). Funkcja taka jest pamięcią CAM, ponieważ podobnie jak w przypadku funkcji, tak i tutaj następuje zamiana argumentów z wartościami.

Analiza matematyczna jednoznacznie definiuje, które funkcje posiadają swoje funkcje odwrotne, co można odnieść do pamięci i określić jakie pamięci można przekształcić w pamięci asocjacyjne. Aby daną funkcje można było odwrócić powinna one być iniekcją oraz[3] suriekcją.

Iniekcją funkcji nazywamy sytuacje gdzie funkcja $f : X \to Y$ dla każdego $a, b \in X$ spełnia warunek $f(a) \neq f(b)$. Funkcja taka jest nazywana często funkcją różnowartościową, ponieważ w jej przeciw dziedzinie nie występują dwa takie same elementy. W przypadku funkcji przedstawionej na rysunku 3.7 nie można powiedzieć, że jest różnowartościowa, ponieważ w przeciwdziedzinie występuje dwa razy ta sama dana jaką jest litera "T". Natomiast w naszej analogi do pamięci, iniekcją jest sytuacja, gdzie konkretna pamięć nie zawiera dwóch takich samych danych. Tutaj od razu pojawia się problem, ponieważ każda pamięć, gdy nie jest jeszcze zapisana posiada we wszystkich komórkach tą samą wartość. Dla uproszczenia uznajmy jednak, że komórki pamięć (pary adres-dana) przed swoim zapisaniem nie są częścią naszej pamięci. Uproszczenie takie nie zmienia znacząco sytuacji, ponieważ w przypadku pamięci CAM nie będziemy poszukiwać braku danych lecz konkretne dane.

Drugim warunkiem, aby funkcja posiadała funkcje odwrotną, jest suriekcja (zwana często "funkcją na"). Oznacza to, że funkcja przyjmuje wszystkie wartości z przeciw dziedziny. Co w przypadku pamięci oznacza, że każda dana jest przyporządkowana do jakiegoś adresu.

Reasumując powyższe rozważania, aby pamięć RAM można było bez żadnych komplikacji odwrócić do pamięci CAM, muszą byś spełnione następujące warunki: dwie komórki pamięci nie mogą zawierać tych samych danych, oraz nie mogą występować dane nie powiązane z żadnym adresem. Oczywiście, w przypadku nie spełnienia tych warunków nadal można stworzyć taką pamięć w postaci elektronicznej, ponieważ jej działanie nie musi być opisane funkcją matematyczną. Możemy na przykład zwracać kilka adresów, pod którymi znajdują się konkretne dane. Co w przypadku funkcji matematycznej jest niemożliwe, ponieważ zaprzecza definicji funkcji, według której każdy argument posiada dokładnie jedną wartość.

Warto tutaj zauważyć, że iniekcja oraz suriekcja oznaczają, że ilość elementów dziedziny jest równa ilości elementów przeciwdziedziny. Co za tym idzie, wielkość przestrzeni

[3]Funkcje będące jednocześnie iniekcją oraz suriekcją nazywamy zwykle bijekcją

adresowej naszej pamięci musi być równa wielkości adresów. W mojej pracy jednak nie wszystkie pamięci są pamięciami, które można odwrócić bez dodatkowych trudności, dlatego też podczas deklaracji pamięci w listingu 3.3 zostały osobno zdefiniowane parametry `DATA_WIDTH` oraz `ADDR_WIDTH`.

3.2.2 Implementacja

W tym rozdziale chciałbym przedstawić zupełnie inne podejście do implementacji, które to uważam osobiście za bardzo eleganckie w swojej prostocie. Polega ono na zapisywaniu adresów w komórkach pamięci odpowiadających danym powiązanym z tymi adresami. Rozwiązanie to posiada niestety duże ograniczenie (wynikające z konieczności wystąpienia iniekcji) jakim jest brak możliwości zapisywania tych samych danych do rożnych komórek pamięci. Z tego też powodu posłużenie się przykładem z tabeli 1.1 przedstawiającej zapisany tekst "TEST", okazało się niemożliwe. Tekstem który posłuży nam jako przykład będzie napis "AGH" przedstawiony w tabeli 3.1.

Adres [hex]	Dane [ASCII]	Dane [hex]
0x00	'A'	0x41
0x01	'G'	0x47
0x02	'H'	0x48
0x03		0xFF
0x04		0xFF
0x05		0xFF
0x06		0xFF
0x07		0xFF
0x08		0xFF

Tabela 3.1: Przykladowa zawartosc pamieci 2

Przykład zapisu adresów[4] w komórkach pamięci został przedstawiony w tabeli 3.2. Taki sposób zapisu sprawia, że dostęp do adresów konkretnych danych jest bardzo prosty, jednak podejście to uniemożliwia odczytywanie danych w tradycyjny sposób, co za tym idzie przykładowe odczytanie "po kolei" całego tekstu jest niemożliwe. Warto tutaj

[4]W przypadku zapisywania adresów w pamięci oraz traktowania adresów jako dane występuje spory konflikt nazewnictwa. W tym rozdziale adresy oraz dane będę nazywał z punktu widzenia pamięci CAM, nie wewnętrznej pamięci w której to ta para została odwrócona.

zauważyć, że poza brakiem możliwości zapisywanie wyrazów posiadających powtarzające się znaki, nie możliwe jest również zapisanie znaków końca wyrazu. Ograniczenie to wynika z faktu, że nie można zapisać do komórki pamięci (odpowiadającej konkretnemu znakowi) więcej niż jednego adresu jej wystąpienia.

Adres [hex]	Adres [ASCII]	Dane [hex]
0x41	'A'	0x00
0x42	'B'	0xFF
0x43	'C'	0xFF
0x44	'D'	0xFF
0x45	'E'	0xFF
0x46	'F'	0xFF
0x47	'G'	0x01
0x48	'H'	0x02
0x49	'I'	0xFF

Tabela 3.2: Przykład odwróconej adresacji

Przykład zapisu przedstawiony w tabeli 3.2 pokazuje znaczną różnice w wielkości takiej pamięci względem pamięci przedstawionej w rozdziale 3.1.2. W przypadku zapisu klasyczną metodą, aby zapisać cztery (załóżmy, że niepowtarzające się) litery potrzebujemy czterech komórek pamięci. W przypadku odwróconej adresacji do zapisu tej samej ilości liter, potrzebujemy tyle komórek pamięci ile jest liter w alfabecie. Natomiast, gdybyśmy chcieli móc zapisać też inne znaki, to liczba potrzebnych komórek wzrosłaby o tyle ile jest znaków przykładowo w tabeli ASCII lub UTF-8. Mimo, iż ta cecha konkretnej implementacji wydaje się jej znaczną wadą, w praktyce wiele pamięci nie wymaga operowania na długich danych, które to wymagały by stworzenia bardzo wielu komórek pamięci. Podobnie sytuacja wygląda jeśli chodzi o ograniczenie jakim jest brak możliwości zapisywania wielokrotnie tych samych danych pod różnymi adresami. O ile w przypadku zapisywania tekstu jest to bardzo duże ograniczenia, to w innych zastosowaniach takich jak powiązanie adresów MAC z adresami IP nie stanowi to problemu, ponieważ w konkretnej sieci ani jeden, ani drugi adres, nie powinien się postarzać.

Warto tutaj jeszcze zauważyć, że formalnie, aby można było mówić o iniekcji, pamięć nie może mieć niezapisanych komórek. Każda niezapisana komórka wskazuje na adres 0xff (będący najwyższym adresem w 8 bitowej przestrzeni adresowej) co oznaczałoby, że pod tym adresem znajdują się wszystkie niewystępujące dane w pamięci.

Dla uproszczenie przyjmę założenie, że zwrócenie maksymalnego adresu z przestrzeni oznacza brak zapisanych danych. Takie rozwiązanie zmniejszy ilość możliwych do przechowania adresów o jeden oraz pozwoli uniknąć dodawanie flagi do każdej komórki.

Adres [hex]	Adres [ASCII]	Dane [hex]
0x41	'A'	0x00
0x42	'B'	0xFF
0x43	'C'	0xFF
0x44	'D'	0x02
0x45	'E'	0xFF
0x46	'F'	0xFF
0x47	'G'	0x01
0x48	'H'	0x02
0x49	'I'	0xFF

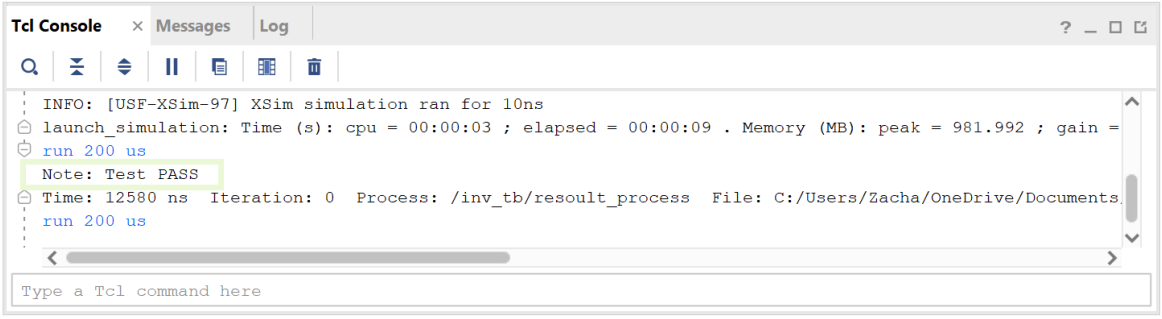
Tabela 3.3: Nieprawidłowy przykład odwróconej adresacji

Przed przedstawieniem implementacji chciałbym zwrócić uwagę na jeden bardzo ważny punkt procedury zapisywania. Składa się ona w rzeczywistości z dwóch czynności: zapisania adresu nowych danych w konkretnej komórce oraz usunięcia adresu starych danych znajdujących się w komórce pamięci odpowiadającej tym danym. Przykładem sytuacji, w której pominięcie drugiego kroku może doprowadzać do niepożądanej sytuacji zapisu może być, zamiana napisu "AGH" na "AGD". Sam zapis sprawia, że poszukując litery "D" prawidłowo zostanie nam wskazana trzecia litera wyrazu. Niestety, gdy "zapytamy" pamięć o literę "H" nadal pozostanie nam wskazana trzecia litera. Problem ten rozwiąże się wówczas, gdy przypiszemy literę "H" do innego adresu, albo gdy będziemy przy każdym zapisie usuwać adres z komórki odpowiadającej danym znajdującym się wcześniej pod nim. Niestety, taka operacja jest trudna, ponieważ wymaga wiedzy pod jakimi danymi został zapisany adres, którego zawartość została zmieniona.

Taka operacja wymaga pamięci CAM, co niestety sprowadza nas do punktu wyjście. W kolejnej sekcji postaram się rozwiązać ten problem przedstawiając zmodyfikowaną wersje pamięci. Natomiast, tutaj chciałbym skupić się na najprostszym rozwiązaniu, które zakłada ograniczenie jakim jest nie zmienianie już raz zapisanych danych w inny sposób niż reset całej pamięci.

3.2.3 Testy

Zgodnie z podejściem przedstawionym w rozdziale 3.1.4, do sprawdzenia poprawności działania implementacji został zastosowany test automatyczny. Podejście taki pozwoliło zrezygnować z ręcznego analizowania przebiegów. Warto tutaj jednak zauważyć że test ten zapisuje jedynie jednokrotnie dane do pamięci, przez co nie wykrywa problemu opisanego na końcu rozdziału 3.2.2. Do tematu tej luki w testach jeszcze powrócę w rozdziale 4.3



Rysunek 3.8: Rezultat symulacji automatycznej inv_tb

3.3 Zastosowanie podwójnej pamięci

3.3.1 Implementacja

W ślad za rozdziałem 3.2.2 postaram się tutaj udoskonalić rozwiązanie polegające na zapisywaniu do pamięci adresów w miejscu "poszukiwanych" danych. Największym problemem poprzednio poprzedniego rozwiązania, był problem z brakiem możliwość odczytywania pamięci w klasyczny sposób. Ograniczenie to ujawniało się nie tylko wtedy, gdy chcielibyśmy skorzystać z pamięci jak z pamięci RAM, ale nawet gdy chcieliśmy korzystać z pamięci jedynie jako pamięci CAM i aktualizować jej zawartość. Aby uniknąć tego problemu, najbardziej naturalnym rozwiązaniem będzie zastosowanie drugiej, bliźniaczej pamięci,w której dane będą zapisywane klasycznie pod ich adresami. Posługując się tutaj ponownie przykładem tabeli zawierającej zawartość pamięci: pierwsza pamięć działająca jako pamięć CAM, jest reprezentowana przez tabelę 3.2, natomiast druga pamięć, służąca jako klasyczna pamięć RAM, jest reprezentowana jako tabela 3.1.

Listing 3.7: `Procedura zapisu dual_cam.vhd`

```
33 architecture Behavioral of dual_cam is
34     type memory_cam is array (0 to 2**DATA_WIDTH-1) of std_logic_vector(ADDR_WIDTH-1 downto 0);
35     type memory_ram is array (0 to 2**ADDR_WIDTH-1) of std_logic_vector(DATA_WIDTH-1 downto 0);
36     signal      mem_cam        : memory_cam := (others => (others => '0'));
37     signal      mem_ram        : memory_ram := (others => (others => '0'));
38 
39 begin
40 
41     write_process : process(clk)
42         variable    old_cam_data  : std_logic_vector(DATA_WIDTH - 1 downto 0) := (others => '0');
43         variable    old_ram_addr  : std_logic_vector(ADDR_WIDTH - 1 downto 0) := (others => '0');
44     begin
45         if rising_edge(clk) then
46             if w_en = '1' then
47                 --write new data
48                 mem_cam(to_integer(unsigned(w_data))) <= w_addr;
49                 mem_ram(to_integer(unsigned(w_addr))) <= w_data;
50                 --remove old data
51                 old_cam_data := mem_ram(to_integer(unsigned(w_addr)));
52                 mem_cam(to_integer(unsigned(old_cam_data))) <= (others => '1');
53                 old_ram_addr := mem_cam(to_integer(unsigned(w_data)));
54                 mem_ram(to_integer(unsigned(old_ram_addr))) <= (others => '1');
55             end if;
56         end if;
57     end process;
```

Kluczową różnicę pomiędzy tą a poprzednią implementacją widać na listingu 3.7. Widać na nim że procedura zapisu składa się z dwóch etapów. Jednoczesnego zapisania danych i adresów do pamięci CAM i RAM (48 i 49 lina kodu), oraz wyczyszczenia pamięci która to zaburzała ny iniekcje funkcji (51 - 54 linia kodu). Uśnięcie odbywa się w dwóch krokach. W przypadku pamięci RAM pierwszym jest pozyskanie adresu pod którym znajduje się identyczna dana jak tą którą chcemy zasiać (do pozyskania tego wykorzystana jest pamięć CAM). Drugim krokiem jest wyczyszczenie tek krokiem tej oto komórki klasycznie adresując ją za pomocą właśnie pozyskanego adresu. W przypadku pamięci CAM procedura przebiega analogicznie. Pozyskujemy dane pod jakimi znajdował się adres poprzedniego wystąpienia adresu który będzie teraz zapisywanie oraz w kolejnym kroku ustawiamy tam wartość wskazującą na brak pary adres-data.

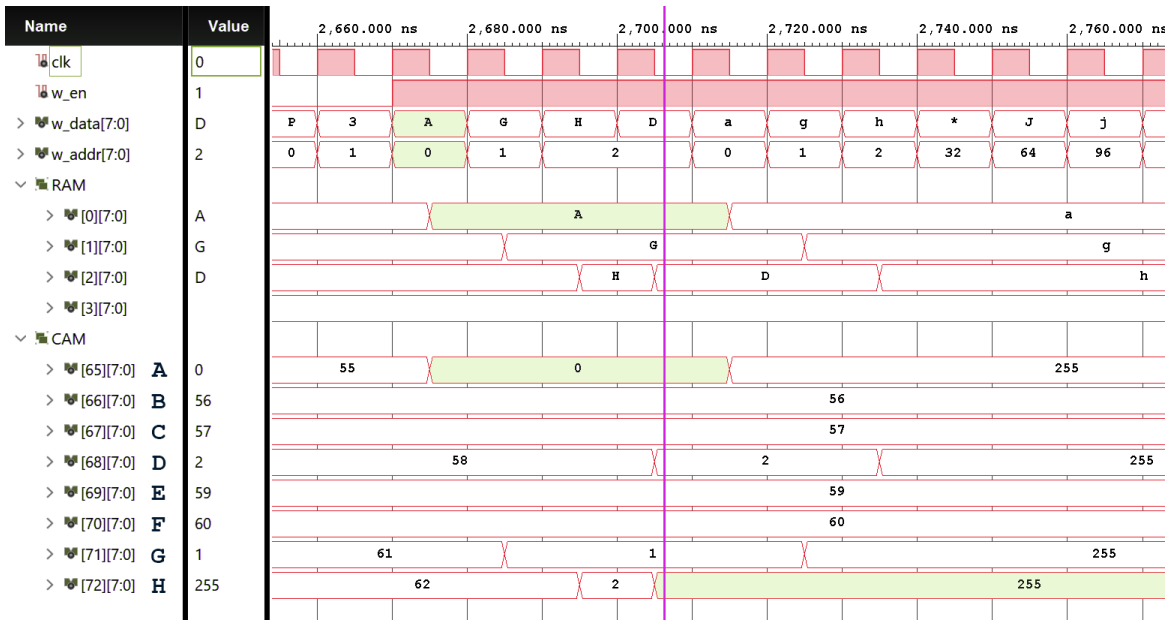
3.3.2 Testy

Działanie pamięci podwójnej najlepiej przedstawia symulacja przedstawiona na rysunku 3.9. Został na mim podkreślony proces zapisywanie litery "A" pod zerowy adres pamięci, oraz sytuacje opisaną w rozdziale 3.2.2 gdzie to litera "H" została zmieniona na litera "D". Działanie takie spowodowało (poza zapisem litery "D" oraz jej adresu) usunięcie innych wystąpień litery "D" z pamięci RAM, oraz usunięcie adresu litery "H" z pamięci CAM. Proces usuwania adresu z pamięci CAM odbywa się przez zapisanie do pamięci samych jedynek. Został on również podkreślony na rysunku 3.9.

3.4 Zastosowanie funkcji skrótu

3.4.1 Funkcje skrótu

Rozpoczynając rozdział poświęcony zastosowaniu funkcji skrótu (haszowaniu), warto rozpocząć od wyjaśnienia, czym one właściwie są. Terminem funkcji haszującej [23][24], zwanej też funkcją skrótu, nazywamy funkcje która przyporządkowuje danemu ciągowi znaków (zwanemu argumentem funkcji) ciąg krótszy (zwany skrótem lub haszem). Skrót funkcji cechuje się stałą długością, niezależnie od długości argumentu. Proces skracania długości funkcji wiąże się z nieodwracalną utratą informacji, co uniemożliwia jednoznacznie odwrócenie operacji. Ponadto, w przypadku funkcji skrótu stosowanych w informatyce, cecha ta jest kluczowa - poszukujemy takich funkcji, których to nie tylko nie da się odwrócić, ale nie da się nawet (w prosty sposób) wyznaczyć żadnego z rodziny argumentów generujących dany skrót. Na rysunku 3.10 została



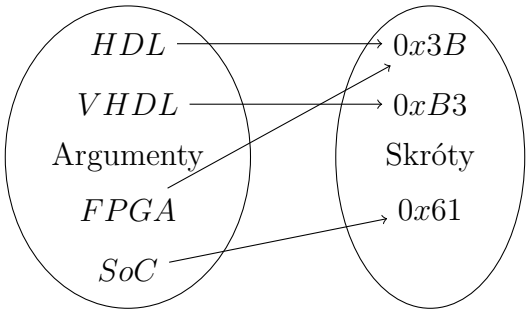
Rysunek 3.9: Rezultat symulacji dual_cam_tb

przedstawiona funkcja skrótu[5] przedstawiająca przekształcenia dłuższego ciągu (jakim są napisu) w krótszy (jakim są hasze). Na rysunku również można zauważyć, że dwa różne argumenty dają w rezultacie ten sam skrót, dzięki czemu przeciwdziedzina funkcji jest mniejsza od dziedziny.

Sytuacja, w której to dwa lub więcej adresów dają ten nam skrót jest nazywana kolizją (ang. hash collisions). W większości zastosowań jest to zjawisko niepożądane. Niestety, nie można go całkowicie wyeliminować, jeśli chcemy, aby skrót był rzeczywiście krótszy niż jego argument. Ograniczenie to wynika z faktu, że skrócenie ciągu wiąże się z utratą informacji. Możliwe jest zdefiniowanie tak zwanej doskonałej funkcji haszującej, w której to nie występowały by kolizje. Niestety, w rzeczywistości tak funkcja nie "skracałaby" ciągu wejściowego, albo mogła by działać tylko na specyficznym zbiorze danych wejściowych, który nie przyjmowałby dowolnych wartości.

[5]Przedstawioną funkcją skrótu jest algorytm CRC-8(SMBUS) wykonany na ciągu znaków zapisanych w formacie UTF-8

Z racji tego, że w mojej pamięci dane mogą zawierać dowolne wartości, zastosowanie takiej funkcji nie miało by żadnego sensu.



Rysunek 3.10: Przykład funkcji skrótu

3.4.2 Zastosowanie funkcji skrótu

Wracając do implementacji, chciałbym rozwiązać problem powstały przy implementacji przedstawionej w rozdziale 3.2.2. Już w przykładzie pokazanym w tabeli 3.2 widać, że do zapisania trzech dowolnych znaków potrzebujemy pamięci zdolnej pomieścić wszystkie możliwe znaki.

Podejście, które chciałbym zaproponować opiera się na zapisywaniu danych do pamięci pod adresami, będącymi w pewien sposób powiązanymi z danymi, które to pod nimi się znajdują. Powiązaniem danych z adresem będzie właśnie funkcja skrótu, co sprawia, że gdy będziemy poszukiwać konkretnych danych, będziemy mogli wyznaczyć adres pod którym będziemy się jej spodziewać. Fakt, iż funkcja skrótu "skraca", sprawia, że pamięć nie będzie musiała mieć aż tylu adresów, aby zapewnić pokrycie wszystkim kombinacjom danych.

W celu lepszego zrozumienia posłużę się przykładem z wykorzystaniem funkcji[6]

$$f(x) = (x - 65) mod 8$$

gdzie x jest znakiem zapisanym w formacie ASCII, a $f(x)$ jest skrótem odpowiadającym w naszym przepadku adresowi pod którym chcemy zapisać dane. Funkcja taka posiada w swojej przeciw dziedzinie tylko osiem elementów, którymi są liczby od 0 do 7. Zostało to przedstawione w tabeli 3.5.

[6]mod wstępujące w równaniu oznacza operacje modulo. Jest to operacja reszty z dzielenia zdefiniowana jako: $reszta = (dzielna) mod (dzielnik)$ przykładowo $1 = 16 mod 5$

Adres [dec]	Skrót(dane) [dec]	Dane [bin]	Dane [ASCI]
0	-	-	-
1	-	-	-
2	2	83	'S'
3	2	67	'C'
4	-	-	-
5	-	-	-
6	6	111	'o'
7	-	-	-

Tabela 3.4: Przykładowa zawartość pamięci - skrót

Chcąc zapisać do pamięci ciąg danych jakim jest napis "SoC" napotykamy szybko na problem w postaci kolizji. Zarówno litera "S" jak i litera "C" posiadają ten sam skrót, którym jest wartość 2. Bez rozwiązania tego problemu trudno mówić o zaoszczędzeniu pamięci, gdyż mimo posiadania ośmiu komórek pamięci nie możemy zapisać nawet trzech z nich. Rozwiązaniem tego problemu (tab 3.4) może okazać się zapisywanie danych do następnej komórki w sytuacji, gdy ta wskazana przez skrót jest już zajęta. Taka metoda wymaga sprawdzania w trakcie odczytu dwóch komórek, a gdybyśmy chcieli mieć pewność, że nie wystąpi kolizja, to kilku kolejnych komórek. Mimo że takie rozwiązanie wymaga porównywania wielu rożnych komórek z poszukiwanymi danymi, jest to spore zaoszczędzenie zasobów w porównaniu z metodą przedstawioną w rozdziale 3.1.

x [ASCII]	x [dec]	f(x) [dec]
'A'	65	0
'B'	66	1
'C'	67	2
'D'	68	3
'E'	69	4
...	...	...
'J'	74	1
'K'	75	2
'L'	76	3
'M'	77	4
'N'	78	5
'O'	79	6
'P'	80	7
'Q'	81	0
'R'	82	1
'S'	83	2
...	...	...
'a'	97	0

Tabela 3.5: funkcja skrótu modulo

3.4.3 Funkcje CRC

Opisana poprzednio funkcja skrótu opierająca się o operacje modulo niestety nie jest najlepszym wyborem funkcji skrótu. Oczywiście, dobór innej funkcji skrótu o tej samej długości, nie zmniejszy prawdopodobieństwa kolizji, ponieważ nie zmieni to ilości adresów która przypada[7] na jeden skrót. Zakładając że rozkład funkcji skrótu jest zawsze jednolity[8].

Różnice w funkcjonalności wynikające z wyboru rożnych funkcji podyktowane są tym, jak konkretny rozkład takiej funkcji wpasowuje się charakter danych, które będą przechowywane w pamięci. Trudno zatem jednoznacznie określić, który skrót jest lep-

[7]W przypadku gdy funkcja skrótu ma cztero bitową dziedzinę oraz trzy bitową przeciw dziedzinę (przyporządkowuje ciągowi 1001 ciąg 001) na każdą wartość skrótu (przykładowo 001) przypadają dwie wartości argumentów(1001 oraz 0001).

[8]Jednolity rozkład funkcji skrótu oznacza że na każdy skrót przypada tyle samo argumentów. Innymi słowy każdy argument z dziedziny ma tyle samo warowności z przeciw dziedziny.

szy, a który gorszy. Niemniej wybór skrótu jakim jest operacja modułu posiada pewnie niedogodności, widoczne już na pierwszy rzut oka. Z uwagi na to, że operacja ta jest na podstawie reszty z dzielenia przez kolejne potęgi dwójki, udział w procesie wyznaczania wartości modulo udział biorą tylko najstarsze (LSB) wartości danych. W przypadku zapisywania liter za pomocą tablicy ASCII nie stanowi to problem, ponieważ to właśnie najstarsze bity są najbardziej zróżnicowane. Jednak gdyby sytuacja wyglądała odwrotnie (na przykład gdybyśmy zapisywali pomiary o mniejszej rozdzielczość, wyrównane do MSB) to wszystkie wartości dawałyby w rezultacie ten sam skrót. Z uwagi na to, że tworząc pamięć często nie wiemy, do jakich zastosowań będzie ona zastosowana, uważam, że zagrożenie płynące za funkcją skrótu, która daje taki sam skrót wszystkich interesującym nas danym, nie jest adekwatne do szans na stworzenie w ten sposób funkcji skrótu idealnie dopasowanej, jak w pierwszym przypadku. W związku z tym warto skorzystać z takiej funkcji skrótu, która byłaby wrażliwa na zmianę każdego z bitów.

Podobnie jak i wiele innych problemów w technice tak i ten doczekał się specjalnej metryki[9] służącej do ilościowego[10] opisu. Metryką tą jest odległość Hamminga, a definiuje się ją pomiędzy dwoma wektorami (w naszym przypadku wartościami skrótu) jako ilość różniących się w nich wartości.
Innymi słowy jest to minimalna ilość zmian jakie było by trzeba wprowadzić w dowolnym wektorze aby wektory te były identyczne. Posługując się przykładem dla ciągów

$$10100000$$

$$10100010$$

odległość Hamminga wynosi 1 ponieważ tylko jeden bit rożni się pomierzy ciągami.

Wracając do opisu funkcji skrótu: minimalną odległością Hamminga dla operacji modulo pomiędzy dwoma ciągami, które różnią się na wejściu jednym bitem, jest zero. Ponieważ, może okazać się, że żaden bit się nie zmieni, pomimo zmiany wartości wejściowej. Funkcja, którą chciałem w tym rozdziale zaproponować jako funkcje skrótu, posiada tak specyficznie ułożony rozkład odwzorowań w funkcji skrótu, że dla zmiany jednego bitu zawsze gwarantuje inny skrót. Co oznacza, że minimalna odległość Hamminga wynosi dla niej jeden. Oczywiście fakt, że zmiana jednego bitu skutkuje wygenerowaniem innej funkcji skrótu, nie oznacz, że zmiana większej ilości bitów również gwarantuje minimalną odległość Hamminga większą lub równą jedynce. Jest to wręcz

[9]Metryka to funkcja, która definiuje odległość, nie tylko w znaczeniu geometrii kartezjańskiej. Nawet w konkretnej przestrzeni (np. Kartezjańskiej dwu wymiarowej) może istnieć wiele definicji metryki, czyli potocznie mówiąc odległości zdefiniowanych w różny sposób.

[10]Opis ilościowy przedstawia wyniki w formie liczbowej. Pozwala na porównanie zjawiska.

niemożliwe, ponieważ ilość kolizji nie zmniejsza się w zależności od wyboru rozkładu, a jedynie jest przeniesiona w inne obszary.

Cykliczny kod nadmiarowy lub cykliczna kontrola nadmiarowa (ang. Cyclic Redundancy Code[25], Cyclic Redundancy Check) (często nazywany CRC[26]) to funkcja skrótu, którą przedstawię w tym podrozdziale. Jest ona głównie stosowana w technice jako redundancja, czyli informacja nadmiarowa służąca do weryfikacji poprawności przesyłania danych. Skrót CRC oblicza się na podstawie danych i przesyła razem z nimi, aby odbiorca mógł sprawdzić, czy wyliczony przez niego skrót jest zgodny z przesłanym przez nadawcę. Zgodność skrótu z dużym prawdopodobieństwem wskazuje na brak błędów w transmisji.

Ze względu na swoje specyficzne zastosowanie, polegające na wykrywaniu różnic pomiędzy ciągami o małej odległości Hamminga, algorytm CRC może działać mniej efektywnie w przypadku danych o dużej różnorodności, co może zwiększać ryzyko wystąpienia kolizji. Temat ten omówię bardziej szczegółowo w kolejnym rozdziale.

Generacja kodów CRC na podstawie ciągu danych odbywa się za pomocą operacji mnożenia wielomianów modulo dwa (ang. division of polynomials Mod-2). Operacja ta jest analogiczna do mnożenia wielomianów, lecz wykonywana w dziedzinie modulo. Oznacza to, że w trakcie obliczeń pojawiają się wyłącznie wartości 0 oraz 1.
Dzięki temu podstawowe operacje matematyczne w tej dziedzinie mają następujące wyniki.

$$\begin{array}{llll} 0+0=0 & 0-0=0 & 0\cdot 0=0 & 0/1=0 \\ 0+1=1 & 0-1=1 & 0\cdot 1=0 & 1/1=1 \\ 1+0=1 & 1-0=1 & 1\cdot 0=0 & \\ 1+1=0 & 1-1=0 & 1\cdot 1=1 & \end{array}$$

Warto tutaj wyjaśnić, że odmienność wyników działań modulo wynika z faktu, iż od każdej wartości większej od wartości modulo odejmowana jest wartość modulo, aż do uzyskania wyniku mieszczącego się w zakresie modulo. Przykładowo:

$$(1+1)mod2 = (2)mod2 = 0$$

Natomiast dla wartości mniejszych niż minimalna wartość modulo, do wyniku dodawana jest wartość modulo, aż do osiągnięcia wyniku mieszczącego się w tym zakresie.

$$(0-1)mod2 = (-1)mod2 = (2-1)mod2 = (1)mod2 = 1$$

Po omówieniu podstaw matematycznych modulo, przejdźmy teraz do zagadnienia obliczania CRC. Proces ten polega na obliczaniu reszty z dzielenia modulo wielomianu odpowiadającego wektorowi danych przesuniętych w lewo o długość crc przez wielomian

generacyjny (konkretny ciąg specyficzny do konkretnej odmiany CRC). Wektor bitowy (jakimi jest ciąg zer i jedynek które chcemy poddać operacji) zamieniamy na wielomian poprzez wpisanie do jednomianów o kolejnych stopniach, współczynników z odpowiadających stopniom indeksom bitów. Przykładowo ciąg "110100" można zapisać:

$$\begin{array}{cccccc} 1 & 1 & 0 & 1 & 0 & 0 \\ 1 \cdot x^5 & +1 \cdot x^4 & +0 \cdot x^3 & +1 \cdot x^2 & +0 \cdot x^1 & +0 \cdot x^0 \end{array}$$

Natomiast operacje dzielenia (w formie pisemnej) wygląda tak jak przedstawiono w tabeli 3.6):

$$\begin{array}{rl} x^6+x^5+x^4 \quad +x^2 \quad +x^0 & \\ \hline\hline x^8+x^7 \quad +x^5 & :(x^2+x^0) \\ \underline{-(x^8 \quad +x^6)} & \\ x^7+x^6+x^5 & \\ \underline{-\ (x^7 \quad +x^5)} & \\ x^6 & \\ \underline{-\ (x^6 \quad +x^4)} & \\ +x^4 & \\ \underline{-\ (x^4+x^2)} & \\ +x^2 & \\ \underline{-\ (x^2 \quad +x^0)} & \\ x^0 & \end{array}$$

Tabela 3.6: Dzielenie wielomianów modulo 2 pisemne

Warto tutaj zauważyć, że interesuje nas tak naprawdę jedynie reszta, a nie wynik dzielenia. Mając na uwadze tę własność oraz przedstawione powyżej właściwości odejmowania modulo, które poosiada takie właściwości jak dodawanie modulo dwa (zwane również operacją XOR), można poczynić spore uproszenia przy wyznaczaniu CRC przenosząc wszystkie operacje do postaci binarnej. Algorytm takich obliczeń wygląda następująco:

1. Dopisanie na miejscu LSB do wektora danych zer odpowiadających długością długości CRC.

2. Wypisanie wielomianu dzielącego tak aby jego początek odpowiadał początkowi wielomianu dzielonego.

3. Wykonanie operacji XOR, jeśli bit nad początkiem wielomiany jest jedynką.

4. Wypisanie wielomiany na pozycji przesuniętej o jeden w prawo względem poprzedniej.

5. Powtórzenie operacji warunkowego XOR oraz przesunięcia (punkty 3,4) do momentu, aż nie zostanie wykonana operacja na ostatnim bicie.

6. Wartością obliczonego CRC jest wynik ostatniej operacji XOR.

Algorytm ten przedstawiony na tym samym przykładzie co poprzedni przedstawiłem w tabeli 3.7. Jak widać, wynikami operacji CRC na ciągu "01101000" (literze "h" w kodzie ASCII) jest ciąg "01". Warto tutaj jeszcze zauważyć, że wynik jest zawsze o jeden bit krótszy, niż długość wielomianu generacyjnego.

0	1	1	0	1	0	0	0	0	0
⊕	1	0	1						
0	0	1	1	1	0	0	0	0	0
	⊕	1	0	1					
0	0	0	1	0	0	0	0	0	0
		⊕	1	0	1				
0	0	0	0	0	1	0	0	0	0
				⊕	1	0	1		
0	0	0	0	0	0	0	1	0	0
						⊕	1	0	1
0	0	0	0	0	0	0	0	0	1

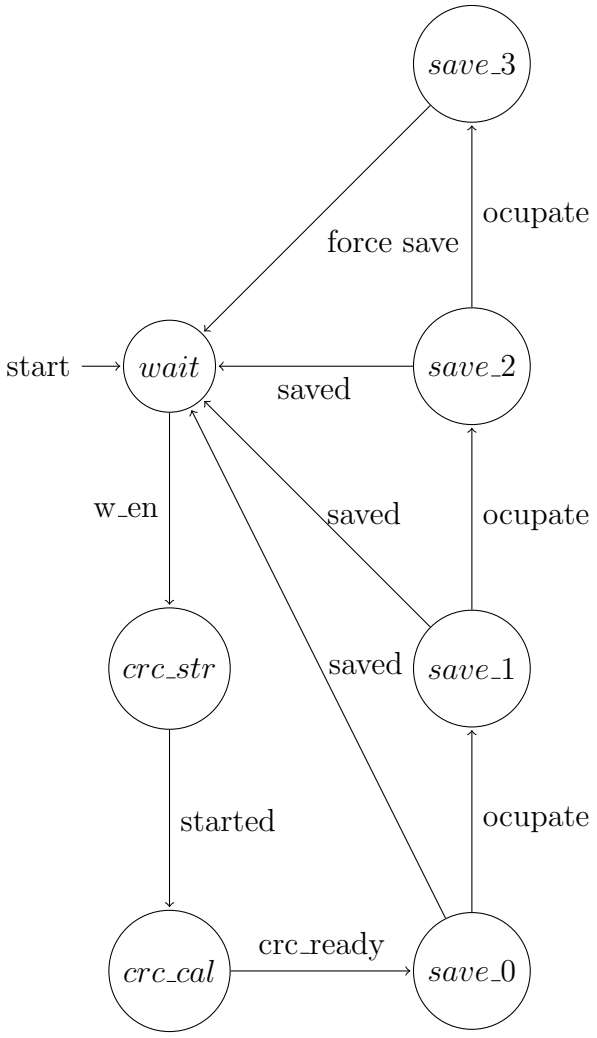
Tabela 3.7: Operacja CRC

3.4.4 Implementacja

Rozpoczynając implementacji pamięci asocjacyjnej, zapisującej danej pod adresami funkcji skrótu wyznaczonego przez CRC, pierwszą napotkaną przeze mnie trudnością był fakt, że pamięć taka może mieć różną wielkość. Co za tym idzie również funkcja skrótu powinna przyjmować różne wielkości przeciwdziedziny. Wiąże się to z nie tylko z koniecznością dostosowania algorytmu do różnej długości danych, ale również do parametrycznej (zależnej od generic) ilości cykli potrzebnych na wyznaczenie adresu, oraz co najważniejsze stosowania różnych wielomianów generacyjnych dla różnych wartości. Tą ostatnią trudność udało mi się rozwiązać stosując tablice zawierającą wielomiany generacyjne dla wszystkich wielkości pamięci w zakresie 1 do 16 bitów, co zostało przedstawione na listingu 3.8.

Listing 3.8: `crc.vhd` tablica wielomianów generacyjnych

```
16 package crc_polynomials_pkg is
17     constant MAX_CRC_WIDTH : integer := 16;
18     type polynomial_array_t is array (1 to MAX_CRC_WIDTH) of STD_LOGIC_VECTOR(MAX_CRC_WIDTH downto
           0);
19     constant CRC_POLYNOMIALS : polynomial_array_t := (
20         1  => "00000000000000011" , --x + 1
21         2  => "00000000000000111" , --x^2 + x + 1
22         3  => "00000000000001011" , --x^3 + x + 1
23         4  => "00000000000010011" , --x^4 + x + 1
24         5  => "00000000000101001" , --x^5 + x^3 + 1
25         6  => "00000000001000011" , --x^6 + x + 1
26         7  => "00000000010001001" , --x^7 + x^3 + 1
27         8  => "00000000100000111" , --x^8 + x^2 + x + 1
28         9  => "00000001000100001" , --x^9 + x^5 + 1
29         10 => "00000011000110011" , --x^10 + x^9 + x^5 + x^4 + x + 1
30         11 => "00000101000000001" , --x^11 + x^9 + 1
31         12 => "00001100000001111" , --x^12 + x^11 + x^3 + x^2 + x + 1
32         13 => "00011000000000111" , --x^13 + x^12 + x^2 + x + 1
33         14 => "00110000000100011" , --x^14 + x^13 + x^5 + x + 1
34         15 => "01000000000000001" , --x^14 + 1
35         16 => "10001000000100001"   --x^16 + x^12 + x^5 + 1
36     );
37 end package crc_polynomials_pkg;
```



Rysunek 3.11: Automat stanów pamięć CAM wykorzystującej skrót

Przechodząc do implementacji pamięci asocjacyjnej, zapisującej danej pod adresami funkcji skrótu wyznaczonego przez CRC, pierwszą napotkaną przeze mnie trudnością był fakt, że pamięć taka może mieć różną wielkość. Co za tym idzie również funkcja skrótu powinna przyjmować różne wielkości przeciwdziedziny. Wiąże się to z nie tylko dostosowaniem algorytmu do różnej długości danych, ale również do parametrycznej (zależnej od generic) ilości cykli potrzebnych na wyznaczenie adresu, a także do stosowania różnych wielomianów generacyjnych dla różnych wartości. Tą ostatnią trudność udało mi się rozwiązać stosując tablice zawierającą wielomiany generacyjne dla wszyst-

kich wielkości pamięci w zakresie 1 do 16 bitów, co zostało przedstawione na listingu

W odróżnieniu do programu przedstawionego w rozdziale 3.3, który to wykorzystywał specyficzne dane jakimi były same jedynki do informowana o tym, że pamięć jest pusta. Pamięć przedstawiona w tym pod rozdziale zawiera specjalny rejestr przechowujący informacje o zajętości pamięci. Trudno jednoznacznie stwierdzić, które rozwiązanie jest lepsze. Przechowanie tej informacji w postaci konkretnych zarezerwowanych danych może wydawać się marnowaniem miejsca, lecz moim zdaniem sytuacja wygląda dokładnie odwrotne. W ten sposób tracimy tylko możliwość zapisu jednego elementu (jakim jest 0b111...1). Natomiast dodatkowy rejestr gdyby został potraktowany jako cześć pamięci (a uważam, że powinien być tak potraktowany, ponieważ wykorzystuje tyle zasobów jak by właśnie nim był) sprawiłby, że tracilibyśmy możliwość zapisu połowy danych. Niezależnie od wartości danych przechowywanych w bloku pamięci, nie mają one znaczenia gdy flaga tej komórki jest ustawiona na wartość "empty".

Pozostając przy temacie optymalnego wykorzystania pamięci, chciałem w tym miejscu zwrócić uwagę na jeszcze jeden aspekt do możliwej optymalizacji. W sytuacji, gdy dane są zapisywane do komórek wyznaczalnych przez funkcje skrótu, sam adres zawiera już pewne informacje o swojej zawartości. Oczywiście, informacje te nie są wystarczające do odtworzenia danych, lecz sprawia to, że informacja zawarta w komórce danych jest nadmiarowa. Fakt ten jest szczególnie widoczny w przypadku funkcji skrótu zrealizowanej za pomocą operacji modulo. Jeśli skrót zawiera ostanie kilka bitów danych (w formie jawnej), można skorzystać z tego faktu, zapisując w komórce pamięci jedynie brakujące dane. Teoretycznie rozwiązanie takie może być wprowadzone również dla funkcji skrótów bardziej złożonych niż operacja modulo. Mimo iż funkcje skrótu takie jak CRC czy SHA nie zapisują w formie jawnej przechowywanej przez nie części informacji, to informacja ta nadal tam jest. Wystarczyłoby przechowywać w komórkach pamięci jedynie informacje potrzebne do rozwiązanie kolizji. Niestety, jednak takie rozwiązanie jest bardzo trudne w realizacji dla większości funkcji skrótu. Wynika to z trudności odwracania większości operacji skrótu celem uzyskania rodziny wektorów, sposób których znajdują się pierwotne dane. Nawet gdyby operacja taka była możliwa do sprawnego wykonania na układzie FPGA, to wiązała by się z rezygnacją z funkcjonalności jaką jest możliwość zapisywania rożnych danych o tej samej funkcji skrótu. W przypadku zapisania danych pod innym adresem niż ich funkcja skrótu cześć informacji zostałaby utracona. Reasumując, uważam, że temat dalszej eksploracji w tym kierunku nie jest wystarczająco opłacalny, w aspekcie zaoszczędzenia zasobów.

Pomimo że informacji zawartej w adresie nie można prosto wykorzystać do zredukowania wielkości komórek pamięci, nadmiarowość tą można wykorzystać w celu weryfikacji poprawności danych zapisanych w pamięci. W tym przypadku stosowanie kodów

kontroli błędów takich jak CRC jako funkcji skrótu pozwala idealnie wykorzystać jego właściwość w zakresie dla którego zostały zaprojektowane. Za każdym razem gdy dane są odczytywane z pamięci, można ponownie wyznaczyć CRC oraz sprawdzić czy zgadza się z adresem z którego pochodzą te dane (z dokładnością do kilku adresów, ponieważ gdy komórka była zajęta zapis mógł się odbyć w kolejnej).

Czy na pewno wprowadzenie takiej funkcjonalność ma sens? Odpowiedź na to pytanie brzmi: nie. Taka funkcjonalność niejako przypadkiem, już jest zaimplementowana. W momencie, gdy chcemy odczytać adres dopowiadający "poszukiwanym" danym. Wyliczamy wartość CRC a następnie sprawdzamy czy w komórce odpowiadającej tym danym znajduje się poszukiwana wartość. Gdyby wartość ta uległa zmianie (przykładowo w wyniki promieniowania kosmicznego[27]) to w procesie porównywania błąd ten zostałby wykryty i objawiłby się w postaci informacji o poszukiwanych danych.

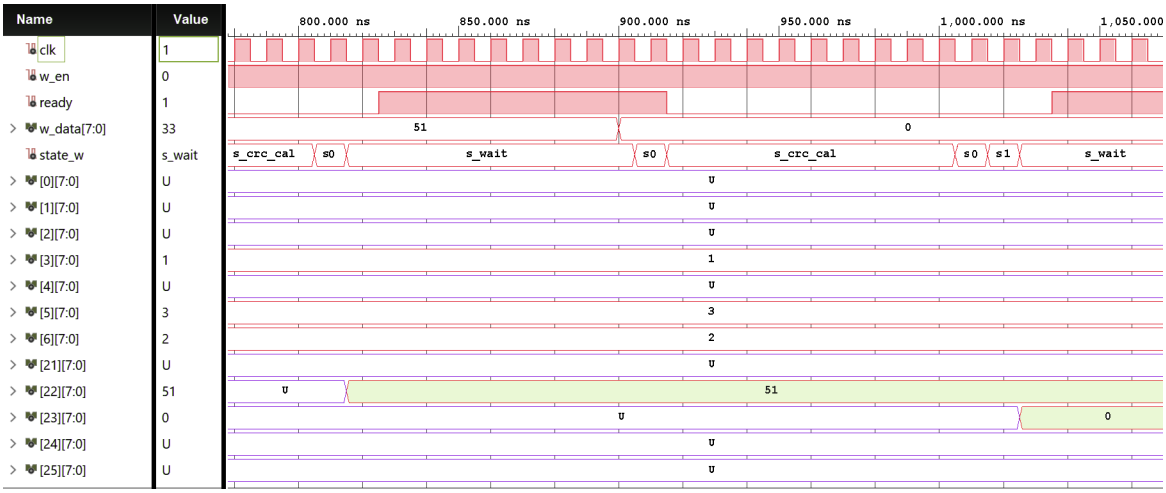
Na koniec tego pod rozdziału chciałbym jeszcze zaznaczyć, że: przedstawiona powyżej pamięć może wykorzystywać wiele różnych funkcji skrótu - nie musi to być koniecznie CRC. Fragment logiki wyznaczający funkcje skrótu został wyodrębniony do osobnego pliku `crc.vhd` dzięki czemu może zostać łatwo podmieniony na inną funkcją taką jak przykładowo: SHA-256, MD5, RIPEMD, BLAKE-256 lub XXH3.

3.4.5 Rezultat

Rezultat działania przedstawionej wyżej idei najlepiej pokazuje zestawienie działania modułu `crc_cam.vhd` z `crc_m_cam.vhd`. Pierwszy z wymienionych jest wczesną wersją, bez zaimplementowanego wpisywania danych pod kolejne adresy w przypadku braku miejsca. Rysunek 3.12 przedstawia proces zapisu wartości 51, która to następnie jest nadpisana poprzez wartość 0. Jest to idealny przykład obrazujący problem, jakim jest fakt, że gdy występuje kolizja może się okazać, że zapisanie dwóch komórek jest niemożliwe, mimo całej wolnej pamięci.

Rysunek 3.13 przedstawia natomiast proces rozwiązania konfliktu przez udoskonaloną wersje pamięci. Widać na nim, że wartość 0 zostaje wpisana do następnej komórki, gdy pierwotna jej destynacja jest zajęta przez 51.

Rysunek 3.12: Rezultat symulacji crc_cam_tb



Rysunek 3.13: Rezultat symulacji crc_m_cam_tb

3.5 Cuckoo hashing

3.5.1 Opis algorytmu

Rozwiązanie przedstawione w rozdziale 3.4, pomimo że częściowo rozwiązuje problem kolizji, nie jest w tej kwestii idealne. Wpisywanie danych do kolejnych adresów, szybko zaczyna znacznie je oddalać do adresu wskazanego przez skrót. Powoduje to zwiększenie czasu dostępu do pamięci, gdy sprawdzane są one w sekwencji, a w przypadku równoległych układów szukających wymaga to dodatkowych układów logicznych.

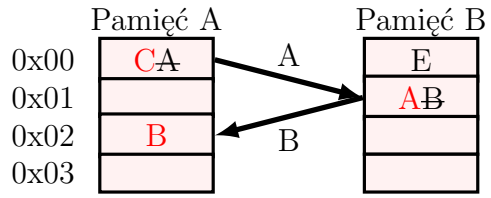
Algorytmem, który pozwoli na znaczne zredukowanie tego problemu, jest Cuckoo hashing. Wykorzystuje on dwie bliźniacze pamięci oraz dwie różne funkcje skrótu. Każda z pamięci jest adresowana poprzez swoją funkcje skrótu. Algorytm zapisu polega na zapisywaniu danych pod jeden z dwóch adresów. Natomiast, gdy obydwa adresy (w obu pamięciach) będą zajęte, algorytm przenosi znajdujące się tam dane pod ich drugą lokalizacje (w drugiej pamięci, w której to obowiązuje inna funkcja skrótu, więc nadaje jej tam inny adres). Tym samym zwalnia miejsce dla danych, które chcemy zapisać. Jeśli przeniesione dane napotkają na konflikt, procedura powtarza się. Przerzucając dane pod ich bliźniacze lokacje, robiąc tym miejsce dla nowszych danych. Algorytm wykonuje tą operacje do określonego momentu - w zależności od implementacji może wykonać taką operacje nie więcej niż kilka razy, lub do wykrycie zapętlenia.

Adres w pamieci [hex]	Możliwa pamieci A [ASCII]	Możliwa pamieci B [ASCII]
0x00	'A' 'C'	'C' 'E'
0x01	'E' 'F'	'A' 'B'
0x02	'B' 'D'	'D' 'F'

Tabela 3.8: Przykładowe funkcje skrótu dla pamięci cuckoo

Warto zauważyć, że funkcje skrótu muszą być różne. Gdyby były takie same, dana którą chcemy przenieś, posiadałaby w drugiej pamięci również ten sam adres co nowa dana. Sprawiłoby to, że w przypadku zajętości tej komórki wrócilibyśmy do punktu wyjścia.

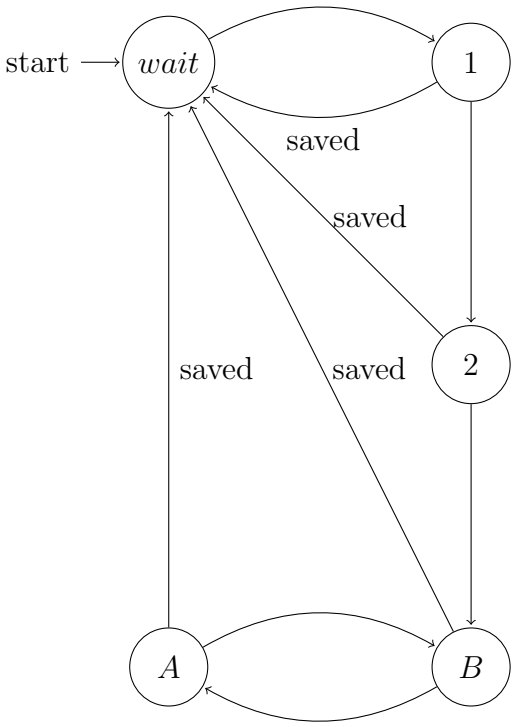
W celu lepszego przedstawienia działania algorytmu posłużę się przykładem przedstawionym na rysunku 3.14. Przedstawia on proces zapisu litery "C" do pamięci. Funkcje skrótu dla oby pamięci zostały przedstawione w tabeli 3.8. Widać na niej, że litera "C" koliduje z literą "A". W związku z tym litera "A" zostaje przeniesiona pod swoją drugą pozycje, w której też następuje kolacji. Litera "A" zostaje zapisana pod swoją drugą pozycją, a znajdująca się tam litera "B" przeniesiona do swojej drugiej komórki. W tym momencie nie występuje już żadna kolizja i pamięć jest gotowa do przyjęcia kolejnych danych.



Rysunek 3.14: Schemat działanie pamięci cuckoo

3.5.2 Implementacja

Kluczowym elementem implementacji jest tutaj automat stanów przedstawiony w uproszczonej formie na rysunku 3.15. Przedstawia on pięć stanów, w których może znajdować się pamięć. Pierwszym z nich jest stan oczekiwania na dane do zapisania, automat wychodzi z tego stanu, gdy na wejście układu zostaną podane nowe dane, a flaga `w_en` będzie wskazywała na aktywny zapis. Drugim stanem jest pierwsza próba zapisu, polegająca na próbie zapisu do pamięci A. Trzecim stanem jest druga próba zapisania danych do pamięci B. Jeśli żadna z nich się nie powiedzie, automat przechodzi do kolejnego stanu, jakim jest stan "A". Podobnie jak poprzednie stany, sprawdza, czy komórka pamięci jest wolna, lecz w odróżnieniu od poprzednich, nawet gdy jest zajęta, zapisuje pod nią dane. Gdy sprawdzenie wykaże, że komórka była przed zapisem wolna, automat powraca do stanu oczekiwania na kolejne dane. Jeśli jednak komórka była zajęta, to jej wcześniejsza wartość zostaje przekazana do kolejnego stanu. Stan "B" jest analogiczny do stanu "A", a jedyna różnica pomiędzy nimi polega na zapisywania do innych banków pamięci.



Rysunek 3.15: Automat działanie pamięci cuckoo

Przedstawiony algorytm jest jedynie uproszczoną forma algorytmu zaimplementowanego w układzie FPGA. W rzeczywistości każdy z stanów (za wyjątkiem "wait") składa się z dwóch stanów: uruchomienia procedury wyznaczanie funkcji skrótu oraz oczekiwania na wykonanie obliczeń. Warto również zauważyć, że w celu uniknięcia nieskończonej pętli próbującej zapisać dane, został wprowadzony rejestr `first_addr`. Zawiera on adres pod jakim stan "1" zapisał dane. Przed każdym zapisem danych przez stan "A" porównywana jest wartość tego rejestru z obecnym adresem zapisu w celu wykrycia powtórzeń.

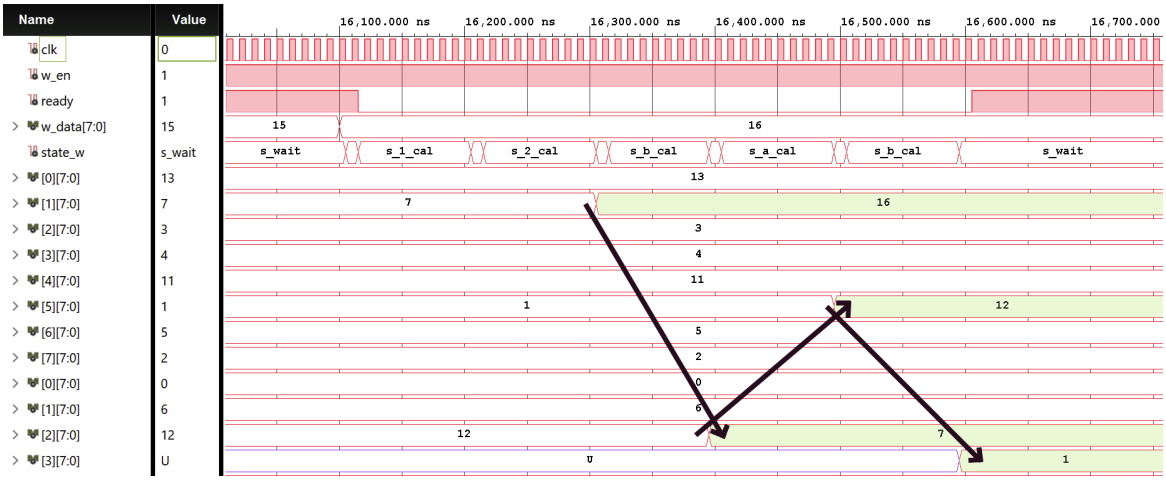
Dużą zaletą pamięci cuckoo jest bardzo prostu proces odczytu. W odróżnieniu od złożonego zapisu, składa się one jedynie z obliczenia dwóch skrótów dla poszukiwanej wartości, a następnie sprawdzenia pod którym z adresów (jeśli w ogóle) znajduje się poszukiwana dana. Warto tutaj jeszcze zauważyć, że wskazanie adresu nie determinuje miejsca w którym można spodziewać się danych, ponieważ mogą być zlokalizowane w dwóch rożnych bankach pamięci. W tym celu (jak widać na listingu 3.9) została wprowadzony dodatkowy sygnał (`destination`) zawierający informacje o tym w której pamięci znajdują się dane.

Listing 3.9: `Inicjalizacja portów cuckoo_cam_tb.vhd`

```
17 entity cuckoo_cam is
18     generic (
19         DATA_WIDTH : integer := 8;
20         ADDR_WIDTH : integer := 4
21     );
22     port (
23         clk         : in  std_logic;         -- work on falling edge
24         w_en        : in  std_logic;         -- 1-> write to memory enable
25         ready       : out std_logic := '0'; -- 1-> memory is ready to write next data
26         valid       : out std_logic := '0'; -- 1-> r_addr is valid
27         not_found   : out std_logic := '0'; -- 1-> not found data in memory
28         destination : out std_logic := '0'; -- 1-> found data in memory B, 0-> ... memory A
29         w_data      : in  std_logic_vector(DATA_WIDTH-1 downto 0);
30         r_addr      : out std_logic_vector(ADDR_WIDTH-1 downto 0);
31         r_data      : in  std_logic_vector(DATA_WIDTH-1 downto 0)
32     );
33 end cuckoo_cam;
```

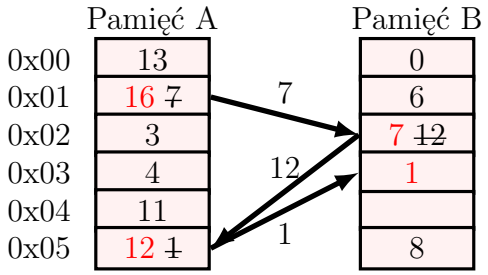
3.5.3 Testy

W celu przetestowania działania pamięci, zdecydowałem się na zastosowanie testów manualnych, aby móc skupić się na najciekawszych przypadkach, w których to można było spodziewać się potencjalnych błędów. Ze względu na trudną odwracalność funkcji skrótu zdecydowałem się przeprowadzić testy na pamięci trój bitowej. Zastosowanie małej przestrzenie adresowej sprawiło, że kolizje występowały częściej a co za tym idzie algorytm cuckoo mógł zostać zaprezentowany na najbardziej złożonych scenariuszach.



Rysunek 3.16: Rezultat symulacji cuckoo_cam_tb

Na rysunku 3.16 został przedstawiony przebieg symulacji pamięci cuckoo. Widać na nim próbę zapisu wartości 16 do pamięci. Ponieważ oba jej adresy są zajęte (odpowiednio wartościami 7 oraz 1). Algorytm przechodzi do stanu "B" w którym to zapisuje danej w pierwszej destynacji, a poprzednio znajdujące się tam dane przenosi do kolejnej lokacji. Proces ten powtarza się aż do przeniesienia wartości 1, do komórki która była uprzednio pusta. Dla lepszego zobrazowania działań symulacji proces zapisu do pamięci został przedstawiony na rysunku 3.17.



Rysunek 3.17: Schemat działanie pamięci cuckoo na przykładzie z symulacji

Rozdział 4

Napotkane problemy

4.1 Idea opisywania problemów

Mimo że stara maksyma mówi, iż uczymy się tylko na swoich błędach, wierze jednak, że możemy również wiele nauczyć się na cudzych. Dlatego powstał ten rozdział, aby przedstawić kilka ciekawych problemów jakie napotkałem w swojej pracy. Mam nadzieję, że będą one dobrą lekcją nie tylko dla mnie.

4.2 Problem z symulacją

4.2.1 Opis zjawiska

Pierwszym problemem, na który natknąłem się w swojej pracy, pojawił się już w rozdziale 3.1.1. Po zaimplementowaniu pamięci CAM w swojej najprostszej formie okazało się, że pamięć co prawda działa (zwraca prawidłowe adresy), jednak wyniki pojawiają się na wyjściu wcześniej, niż się spodziewałem. Jak widać na rysunku 4.1 zapytanie o daną `16` skutkuje natychmiastowym zwróceniem adresu `6`. Mimo że takie działanie pamięci jest prawidłowe i powinno nas cieszyć (ponieważ chcemy jak najszybciej otrzymać wynik), nie jest tak naprawdę niczym dobrym.

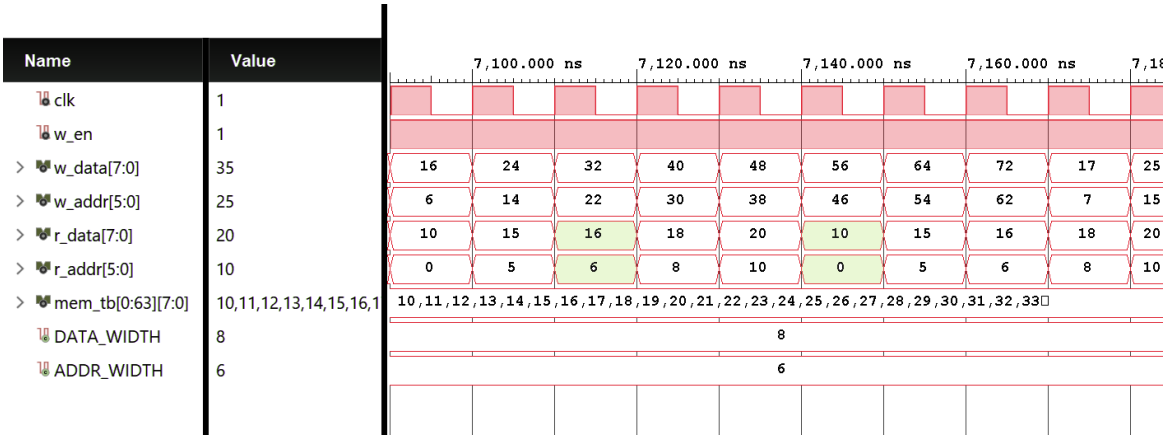
Rezultaty pojawiające się na wyjściu wcześniej niż tego oczekiwałem były spowodowane jednoczesną zmianą sygnału zegarowego oraz sygnałów danych. Pierwszym moim podejrzeniem było zjawisko hazardu [28] [29]. Jest to zjawisko którego wynika z nie zerowego czas propagacji sygnałów. W sytuacji, gdy do pewnej logiki docierają pewne sygnały z rożnymi opóźnieniami, może wystąpić sytuacja, gdzie stan na wyjściu układu

będzie zależał od kolejności pojawiania się danych na wejściu. Co gorsza, bardzo często rezultat może okazać się trudny do przewidzenie, ponieważ czasy propagacji zmieniają się na tyle, że kolejność dotarcia sygnałów też może być różna.

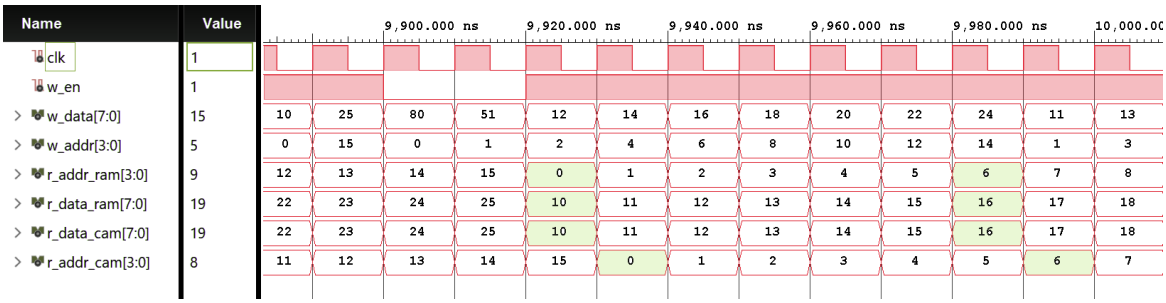
Jednak w przypadku układów synchronicznych jakimi są układy FPGA zjawisko hazardu nie powinno występować. Pomimo iż powszechnie unika się zmieniania danych w rytm sygnału zegarowego w protokołach komunikacyjny (takich jak UART, CAN a nawet AXI) jest to jednak powszechna praktyka w układach synchronicznych (przykładowo w licznikach synchronicznych). W takim przypadku dane powinny zostać pobrane z chwili tuż przed wystąpieniem zobaczą, a na wyjściu układ powinny się pokazać tuż po zboczu. W rzeczywistych układach jest to spowodowane zwłoką w zmianie stanów logicznych wynikających chociażby z pojemność elektrycznej.

Kolejnym moim podejrzeniem było że problem może wynikać z faktu iż dane są pobieranie po wystąpieniu zobaczą na nie przed nim. Według mnie było to spowodowane nieprawidłową symulacją. Symulacja każdej zmiany stanów w programie Vivado odbywa się w wielu krokach oznaczonych kolejno $\delta 0$, $\delta 1$, ... W pierwszym z tych kroku wyznaczana jest wartość wyjść wszystkich układów która których wejścia są wejściami symulowanego układu, w następnym kroku wyznaczane są stany wyjściowe tych układów które to mają te poprzednio wyznaczone na wejściach. Proces taki powtarza się tak długo aż będzie możliwe wyznaczenie wartości wyjść symulowanego układu. Warto tutaj jednak zauważyć że nawet w sytuacji gdy dane do konkretnego układu docierają w symulacji rożnymi drogami, nie może wystąpić zjawisko bliźniacze do hazardu ponieważ stan takiego układa zostanie policzony dopiero w kroku w którym dotrą do niego wszystkie sygnały wejściowe.

Powagę problemu pokazuje rysunek 4.2, gdzie widać fragment testu automatycznego w którym zgodnie w rysunkiem 3.4 moduł `basic_cam.vhd` jest jedną ze składowych całego systemu. Przykład ten pokazuje, że nawet niewielka zmiana może spowodować inną kolejność symulowania układu, co w rezultacie może prowadzić do zupełnie innych wyników na wyjściu. Fakt ten skłonił mnie jednak do podejrzeń że symulacja vivado 2019.2 nie działa prawidłowo.



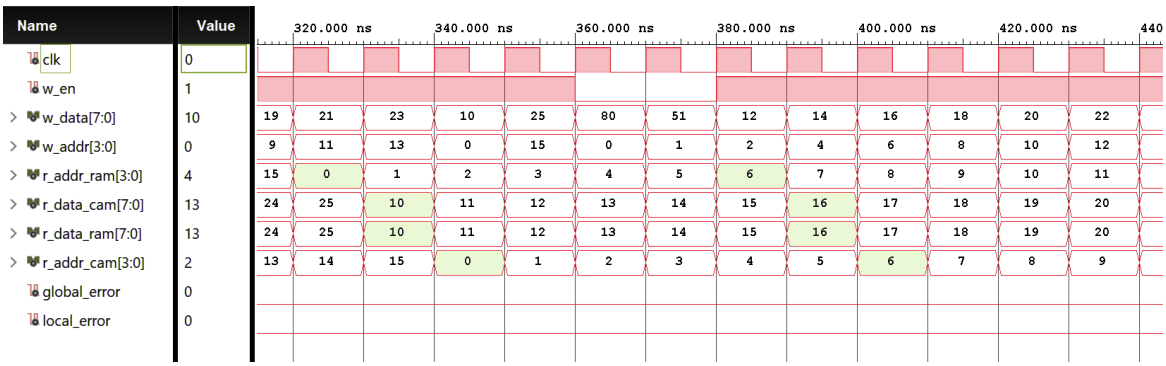
Rysunek 4.1: Rezultat symulacji basic_cam_bug1_tb



Rysunek 4.2: Rezultat symulacji basic_bug1_tb

4.2.2 Rozwiązanie problemu

Rozwiązaniem problemu może być pobieranie danych na innym zboczu (zegara) niż następuje ich zmiana. Warto tutaj jednak zauważyć, że gdy układ jest taktowany zboczem opadającym, dane na jego wyjściu również pojawiają się na zboczy opadającym. Takie rozwiązanie może utrudniać potencjalne łączenie dwóch takich układów. Uważam, że takie podejście jest bardzo mało czytelne, ponieważ trudno mówić o jakiejkolwiek kompatybilności układu z innymi modułami, jeśli nie jest on kompatybilny sam ze sobą. Dlatego zdecydowałem się przyspieszyć farze sygnału zegarowego w symulacji o 1/20 okresu. Zabieg takie zagwarantował pobieranie danych po odpowiedniej stronie zbocza zegarowego, co zapewniło prawidłowe działanie pamięci, przedstawione na rysunku 4.3.



Rysunek 4.3: Rezultat symulacji poprawionego modułu basic_cam

Warto tutaj jednak zauważyć że problem tutaj przedstawiony był jedynie wynikiem nieprawidłowego testowanie. Układ zaimplementowany na układzie FPGA oraz przetestowany za pomocą wewnętrznego analizatora stanów logicznych ILA działał prawidłowo.

4.3 Niedoskonałe testy

4.3.1 Przedstawienie problemu

Kolejny problem z jakim przyszło mi się zmierzyć podczas pisania mojej pracy, nie dotyczył bezpośrednio implementacji, lecz problemem w prawidłowym testowaniu. Okazało się, że testy automatyczne, które to tak dobrze sprawdzały się w przypadku danych różnowartościowych, niestety nie działają prawidłowo gdy dane te powtarzają się. Problem ten w rzeczywistości okazał się dwoma bliźniaczymi problemami.

Pierwszym problemem jest brak przystosowania struktury testu przedstawionej na rysunku 3.4 do powtarzających się danych. W rezultacie, gdy za każdym razem pod dwoma adresami znajduje się ta sama dana, pamięć CAM zwraca adres pierwszego wystąpienia. Działania to jest prawidłową cechą pamięci CAM i nie może być nazwane błędem. Jednak z punktu widzenie symulacji, jest to spory problem, ponieważ gdy dochodzi do porównania adresów może okazać się, że adres podany na pamięć RAM nie zgadza się z tym otrzymanym na wyjściu pamięci CAM.

Metod na rozwiązanie pierwszego problemu jest co najmniej kilka, lecz zanim przedstawię którąś z nich warto przedstawić drugi problem, ponieważ cała trudność polega na znalezieniu metody rozwiązującej oba problemy.

Drugim problemem, związanym z metodą testów automatycznych, jest sytuacja w której to pamięć musi dokonać pewnych (charakterystycznych dla konkretnej implementacji) modyfikacji w parach adres-data. Przykładowo, jeśli przedstawiona w rozdziale 3.3 usuwa poprzednie wystąpienia danej, to test automatyczny musi robić identyczną operacje w pamięci RAM. W przeciwnym wypadku różnica w zawartości pomiędzy pamięcią RAM a CAM zostanie odebrana jako błąd w działaniu pamięci CAM.

4.3.2 Rozwiązania problemu

Przedstawione powyżej dwa problemy, będąc związane razem, skutecznie uniemożliwiają stworzenie jednego uniwersalnego testu, który to w zależności od implementacji musiałby przyjmować inną formę testowania. Nasuwa to automatycznie rozwiązanie jakim byłoby stworzenie automatycznych testów do każdej implementacji osobno. Podejście takie niestety, również nie jest pozbawione wad. Tworząc do każdej implementacji oddzielny test oraz zawierając w nim bardzo specyficzne metody zapisu, a nie wzorując się na podstawowych zasadach odwrotności działań jak to miało miejsce w rozdziale 3.1.4, bardzo łatwo jest popełnić ten sam błąd w programie testującym oraz testowanym. Ponadto głównym powodem zastosowania testów automatycznych miał być fakt przyspieszenia prac oraz testowania tą samą metodą wielu implementacji. Rozwiązanie takie miało pozwolić na sprawniejsze wykrywanie różnic pomiędzy implementacjami, które to mogą świadczyć o potencjalnych błędach.

W rezultacie zdecydowałem się nie tworzyć całej rodziny testów automatycznych i pozostać przy dotychczasowym teście automatycznym sprawdzającym jedynie przypadek, gdy dane nie powtarzają się. Test taki jest pierwszym wskaźnikiem, który może zwrócić uwagę na błędne działanie pamięci, lecz nie jest wystarczający do stwierdzenie poprawności mechanizmów. Co za tym idzie nie wyklucza stosowanie testów manualnych z procesu testowania.

Rozdział 5

Porównanie rozwiązań

5.1 Funkcjonalności

W niniejszej pracy przedstawiłem wiele rożnych metod implementacji pamięci CAM. Warto w tym miejscu zastanowić się nad tym, jakie są ich zalety a jakie wady. Pomimo iż w trakcie implementacji starałem się aby wszystkie pamięci były możliwie jak najbardziej podobne do siebie pod kątem funkcjonalności (co miało na celu ułatwić ich ilościowe porównanie), różnią się one od siebie czasem w znacznym stopniu. Wynika to z specyficznych właściwości pewnych rozwiązań.

5.1.1 Kontrola nad adresami

Pierwszym z kryteriów jest możliwość zapisywania danych pod dowolnymi adresami. Funkcjonalność ta, mimo iż może wydawać się oczywista, w niektórych implementacjach jest niedostępna. Rozwiązania wykorzystujące funkcje skrótu, niejako same decydują o destynacji danych. Cecha ta w większości zastosowań nie dyskwalifikuje takiej pamięci. Może to jednak skutkować trudnościami w kompatybilności i swobodnemu podmienianiu rozwiązań.

5.1.2 Iniekcja

Drugim kryterium jest możliwość zapisywania tych samych danych wielokrotnie do pamięci, ta cecha niestety występuje jedynie w najprostszej formie pamięci (`basic_cam`). Jest to wbrew pozorom bardzo problematyczna funkcjonalność. Nawet w przypadku pamięci `basic_cam` należy zachować ostrożność, ponieważ cecha jaką jest zwracanie adresu pierwszego wystąpienia czasem może być myląca.

5.1.3 Kolizje

Trzecim jakościowym kryterium dzielącym pamięci jest fakt występowanie kolizji. Pamięci takie jak `basic_cam`, `inv_cam`, `dual_inv_cam` są całkowicie pozbawiane tego problemu. Pozwala to na zapisanie w pamięci wszystkich jej komórek. Zapis taki jest niestety niemożliwy w przypadku pamięć opartych na funkcji skrótu. W skrajnym przypadku pamięci `crc_cam` opartej na funkcji skrótu bez możliwości zapisaniu w przypadku kolizji pod kolejnym adresem, może się okazać, że pamięć nie będzie mogła przyjąć nawet dwóch danych niezalezienie od swojej wielkości.

5.1.4 Klasyczny dostęp

Kolejnym kryterium jest możliwość klasycznego dostępu do pamięć. Pomimo iż żadna z pamięci przedstawiona w mojej pracy nie posiadała wyprowadzonych portów w celu klasycznej dostępu do pamięci (adresowanej adresami), w większości z nich funkcjonalność ta jest bardzo prosta do uzyskana. Warto tutaj jednak zauważyć, że pamięć `inv_cam` nie przechowuje danych w klasycznej formia. Przez co odwołanie się do konkretnych danych poprzez jej adres jest nie możliwe.

5.1.5 Zewnętrzna pamięć

Ostatnim kryterium podziału pamięci jest zastosowanie zewnętrznej pamięci. Pomimo że w żadnym z przedstawionych tutaj rozwiązań nie korzystałem z zewnętrznej pamięci, wszystkie rozwiązania za wyjątkiem `basic_cam`, można w łatwy sposób przekształcić do korzystania z wewnętrznej pamięci. W takiej implementacji układy logiczne wymagane są jedynie do sterowania algorytmem "szukania"/"zapisuwania" danych. Podejście to jest niestety nie możliwe w przypadku pamięci `basic_cam` ponieważ potrzebuje ona równoległego dostępu do wszystkich komórek pamięci jednocześnie w celu porównywania ich z wartością wejściową.

5.1.6 Podsumowanie

Wszystkie przedstawione powyżej kryteria ilościowe zostały zestawione ze sobą w tabeli

	Kontrola adresami	Iniekcja	Kolizje	Klasyczny dostęp	Zewnętrzna pamięć
basic	T	N	N	T	N
inv	T	T	N	N	T
dual_inv	T	T	N	T	T
crc	N	T	T	T	T
crc_m	N	T	T	T	T
cuckoo	N	T	T	T	T

Tabela 5.1: Rożnicę jakościowe pomiędzy rozwiązaniami

5.2 Wymagane zasoby

5.2.1 Metodologia pomiarów

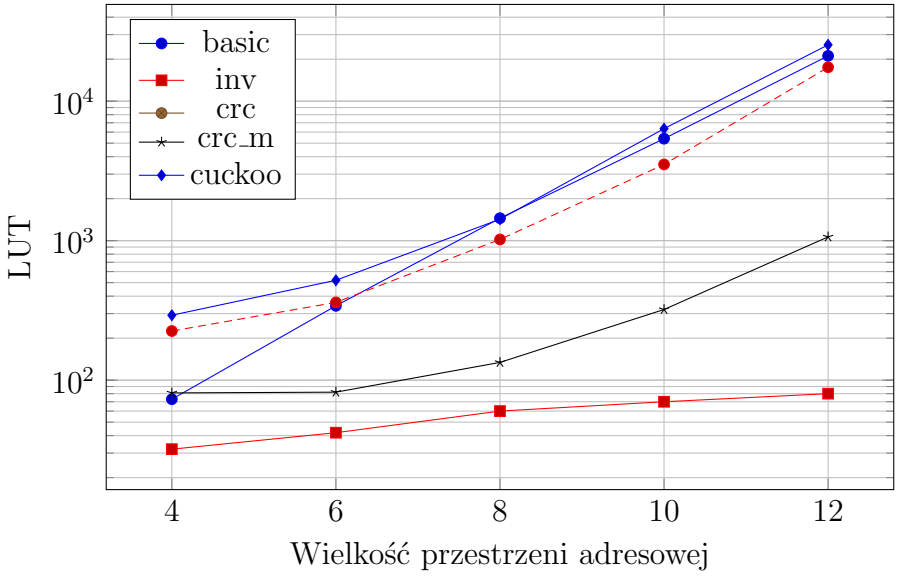
W celu ilościowego porównania przedstawionych rozwiązań zdecydowałem się na zestawienie wykorzystania zasobów sprzętowych poprzez różne rozwiązania. W tym calu porównywałem parametry jakimi była ilość wykorzystanych tablic logicznych LUT, LUTRAM oraz przerzutników FF. Warto tutaj zwrócić uwagę, że dla łatwiejszego porównywania wyników zdecydowałem się na niewykorzystywanie pamięci BRAM. Co za tym idzie, wszystkie komórki pamięci są realizowane za pomocą LUTRAM. W celu wymuszenie w procesie syntezy niewykorzystywanie pamięci BRAM przed każda inicjalizacją pamięci została dodana dyrektywa przedstawiona na listingu 5.1 w liniach 36 oraz 37.

Listing 5.1: `Blokada pamięci BRAM inv_cam.vhd`

```
33 architecture Behavioral of inv_cam is
34     type memory_array is array (0 to 2**DATA_WIDTH-1) of std_logic_vector(ADDR_WIDTH-1 downto 0);
35     signal      mem             : memory_array := (others => (others => '0'));
36     attribute   ram_style       : string;
37     attribute   ram_style of mem : signal is "distributed";
```

5.2.2 Rezultaty

Przedstawiony na rysunku 5.1 wykres obrazuje wykorzystanie bloków logicznych LUT w zależności od długości adresów pamięci. Najmniejszym wykorzystaniem zasobów cechuje się pamięć "inv". Wynika to z braku logiki sterującej tą pamięcią oraz realizacji jej głównie za pomocą bloków pamięci (LUTRAM).

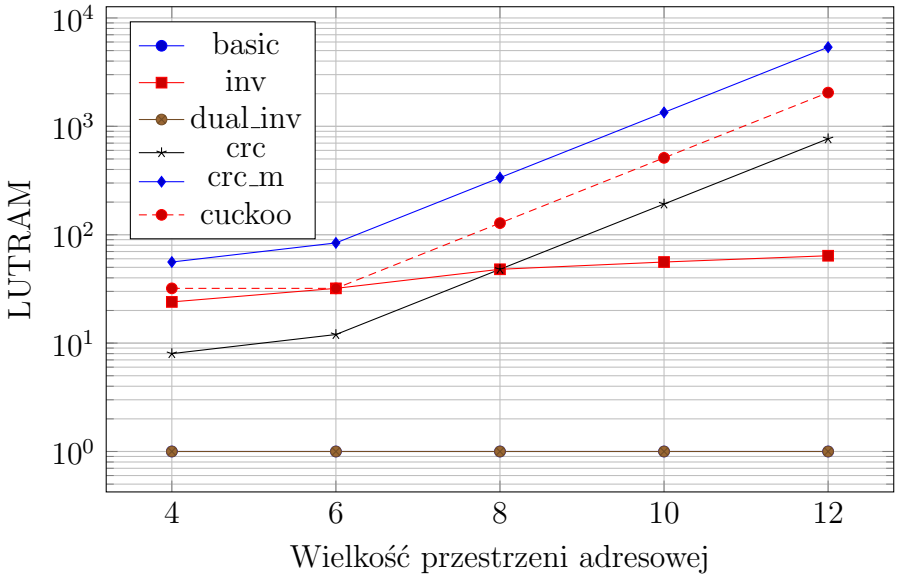


Rysunek 5.1: Wykorzystanie LUT dla pamięci 8bitiwej

Po analizie wykresu można dojść do błędnego wniosku, że pamięci "basic" oraz "cuckoo" wykorzystują taką samą ilość zasobów, a implementacja "cuckoo" jest nieopłacalna ze względu na swoją złożoność. Relacja ta wynika z faktu, że rozbudowany układ sterujący oraz wyznaczanie wielu funkcji skrótu wymagają podobnej ilości zasobów co zastosowanie licznych komparatorów. Różnica jest jednak widoczna w przypadku pamięci 16-bitowej, co przedstawiono na wykresie 5.4.

W przypadku pamięci 16-bitowej wpływ układu sterującego staje się coraz bardziej pomijalny. W pamięci "basic" każdy kolejny bit (poza zwiększeniem wielkości pamięci) proporcjonalnie zwiększa liczbę wymaganych komparatorów (na przykład dla 12-bitowej przestrzeni adresowej potrzebne są kolejne 2^{12} komparatory). Natomiast w przypadku pamięci "cuckoo" zwiększenie wielkości danych o jeden bit (poza zwiększeniem wielkości pamięci) wymaga jedynie dodania jednego bitu w kilku rejestrach algorytmu obliczania funkcji skrótu.

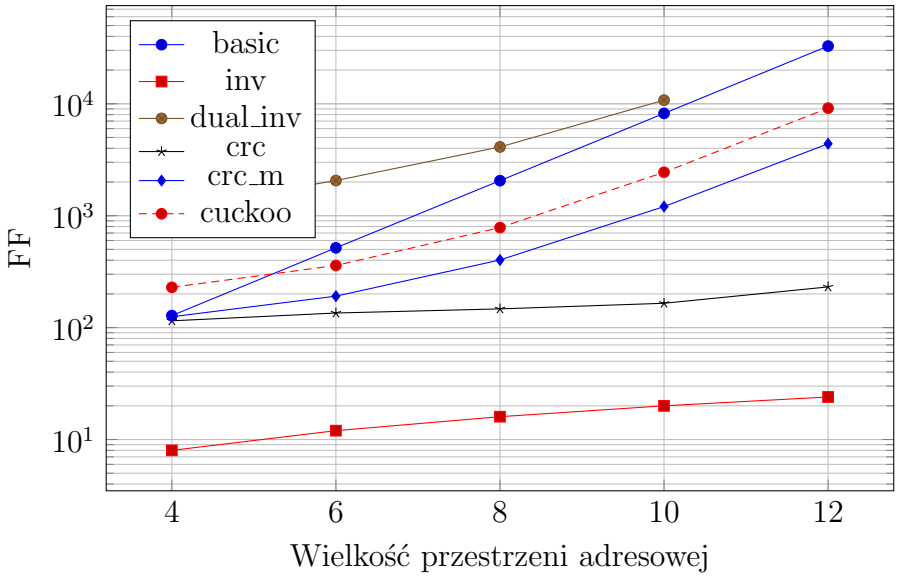
Warto w tym miejscu zwrócić uwagę również na pamięć "inv". W jej przypadku zwiększenie szerokości danych o jeden bit skutkuje zwiększeniem przestrzeni adresowej o kolejny bit, co wiąże się z podwojeniem wielkości układu.



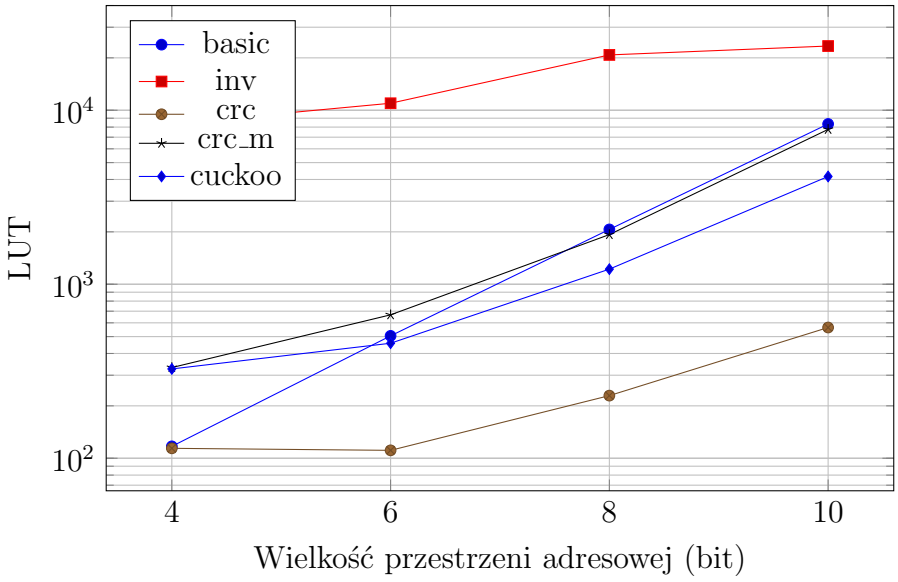
Rysunek 5.2: Wykorzystanie LUTRAM dla pamięci 8bitiwej

Rysunek 5.2 przedstawia wykorzystanie komórek pamięci LUTRAM. Można zauważyć, że nie wszystkie rozwiązania są w stanie wykorzystać ten zasób, podobnie jak nie wszystkie umożliwiają realizację pamięci za pomocą bloków BRAM. Na wykresie widoczne jest również, że wielkość zajmowanej pamięci LUTRAM jest proporcjonalna do wielkości przestrzeni adresowej, co skutkuje asymptotycznym tempem wzrostu na poziomie $O(\text{LUTRAM}(\text{addr'length})) \in 2^{\text{addr'length}}$.

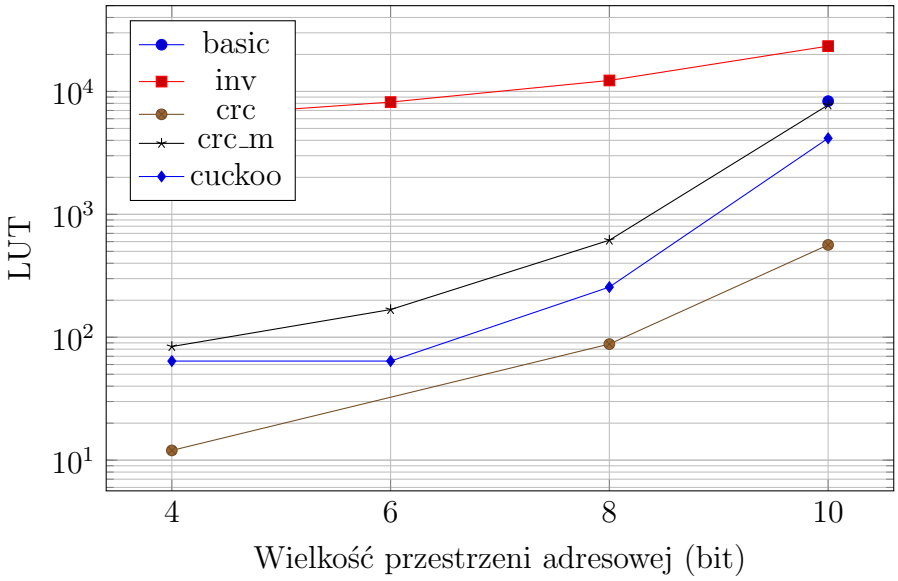
Wyjątek od tej reguły stanowi pamięć wykorzystująca odwrotną adresację. Wielkość tej pamięci zależy bowiem proporcjonalnie od długości adresów, a nie przestrzeni adresowej. Wynika to z faktu, że liczba komórek tej pamięci zależy od szerokości danych, podczas gdy długość adresów przekłada się proporcjonalnie na długość komórek tej pamięci.



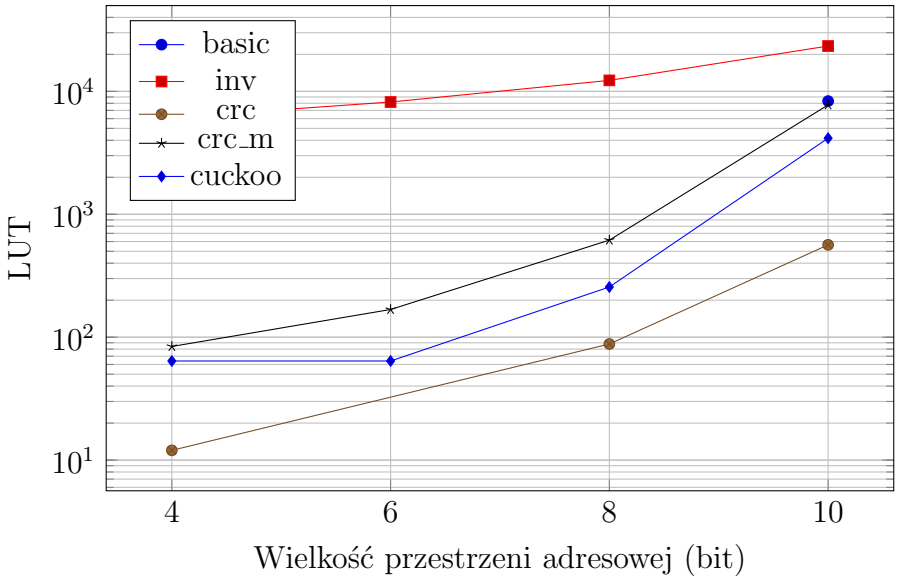
Rysunek 5.3: Wykorzystanie FF dla pamięci 8bitiwej



Rysunek 5.4: Wykorzystanie LUT dla pamięci 16bitiwej



Rysunek 5.5: Wykorzystanie LUTRAM dla pamięci 16bitiwej

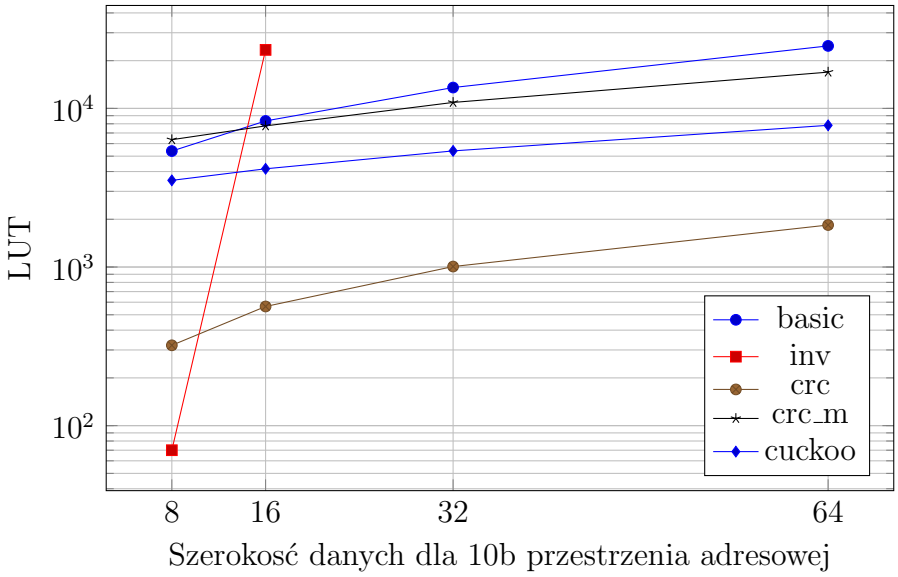


Rysunek 5.6: Wykorzystanie FF dla pamięci 16bitiwej

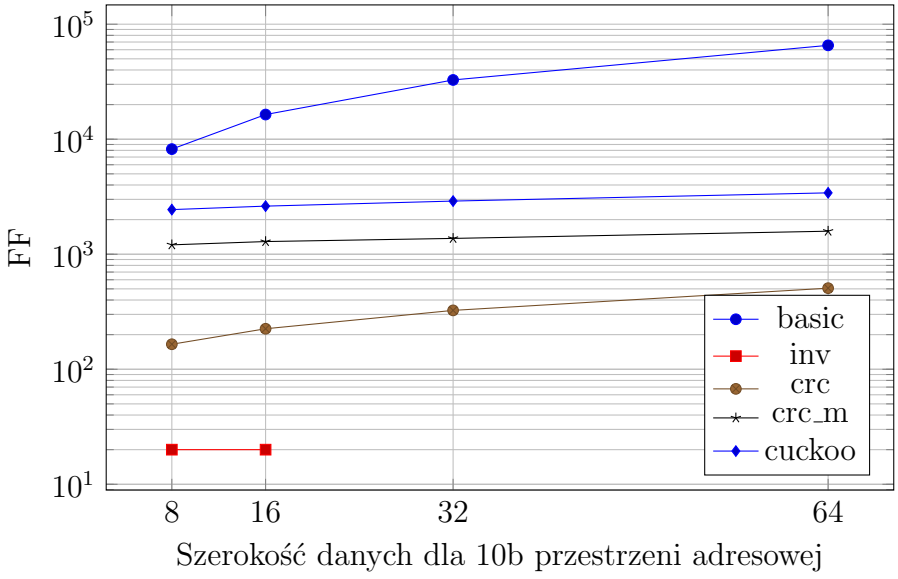
5.2.3 Wpływ szerokość danych

Na sam koniec chciałbym jeszcze przedstawić zależność wykorzystanych zasobów w funkcji szerokości danych. Wykresy 5.7 oraz 5.8 przedstawiają wpływ szerokości danych na wykorzystanie zasobów dla 10-bitowej przestrzeni adresowej. Widać na nich, że implementacja "inv", która cechowała się bardzo niskim wykorzystaniem zasobów dla krótkich komórek pamięci, wraz ze wzrostem szerokości danych bardzo szybko zwiększa wymagane zasoby. Na wykresie zostały przedstawione jej wartości jedynie dla 8- i 16-bitowej szerokości. Nagły wzrost wielkości pamięci wraz ze wzrostem szerokości danych sprawia, że dla 32-bitowej szerokości pamięć taka musiałaby mieć $2^{32} = 4\ 294\ 967\ 296$ komórek. Niestety, pamięci takiej wielkości są trudne do symulowania w programie Vivado.

Na wykresach można również zauważyć, że dla coraz szerszych komórek danych algorytm oparty o CRC lepiej się skaluje niż "siłowa" metoda sprawdzania oparta na komparatorach.



Rysunek 5.7: Wykorzystanie LUT w funkcji szerokości danych



Rysunek 5.8: Wykorzystanie FF w funkcji szerokości danych

Rozdział 6

Podsumowanie

6.1 Wnioski

Przedstawione w rozdziale 5 zestawienie różnych rozwiązań, pokazało że nie istniej idealne rozwiązanie pamięci CAM. Implementacja konkretnej metody powinna zależeć zarówno od parametrów jakościowych (rozdział 5.2) jak i ilościowych (rozdział 5.1).

Chciałbym w tym miejscu zauważyć że przedstawione porównanie jest porównaniem jedynie moich implementacji. Pokazuje one pewne trendy (przykładowo pamięć "inv" gdy nie zapisujemy długich danych, natomiast "cockoo" wyróżnia się nad innymi rozwiązaniami dopiero w przypadku implementacji większych pamięci) lecz warto pamiętać że każda z implementacji może zostać jeszcze dodatkowo zmodyfikowana pod konkretne zastosowania.

6.2 Zakończenie

Pomimo że elektroniką zajmuję się zawodowo od sześciu lat, a hobbystycznie od ponad dwunastu, i moje umiejętności w tej dziedzinie z pewnością nie można uznać za małe, uważam, że przez te wszystkie lata najwięcej nauczyłem się cierpliwości oraz pokory. Praca nad tym projektem była kolejną lekcją, która pokazała mi, że temat ten nadal skrywa niewyobrażalne pokłady fascynujących zagadnień. Było to dla mnie rozwijającym doświadczeniem dostrzec jak, drzewo możliwych rozwiązań rozrastało się niemal w nieskończoność, niezależnie od kierunku, w którym chciałem rozwijać tą pracę.

Spis tabel

Spis rysunków

Listings

Bibliografia

[1] Wojciech Głocki. *Układy Cyfrowe*, chapter 14, pages 262–276. Wydawnictwa Szkolne i Pedagogiczne S.A., 1998.

[2] Paul Horowitz Winfield Hill. *Sztuka Elektroniki*, chapter 14.1-14.2, pages 563–573. Wydawnictwa Komunikacji i Łączności WKŁ, 2018.

[3] Anil K. Maini. *Digital Electronics*, chapter 1.7.1, pages 4–5. john wiley & sons ltd, 2007.

[4] David Money Harris & Sarah L. Harris. *Digital Design and Computer Architecture*, chapter 6.4.5, pages 314–327. Elsevier Ltd. Oxford, 2013.

[5] David A. Patterson John L. Hennessy. *Computer Organization and Design*, chapter 2.4-2.5, pages 73–87. Elsevier Science & Technology, 2020.

[6] Mirosław Kardaś. *Mikrokontrolery AVR Język C*, chapter 3.7.3, pages 151–156. Atnel, 2013.

[7] Paul Horowitz Winfield Hill. *Sztuka Elektroniki*, chapter 14.4.1-14.4.4, pages 597–601. Wydawnictwa Komunikacji i Łączności WKŁ, 2018.

[8] Zahid Ullah, Manish Kumar Jaiswal, Y.C. Chan, and Ray C.C. Cheung. Fpga implementation of sram-based ternary content addressable memory. *IEEE*, 26(978-1-4673-0974-5), 2012.

[9] Wikipedia. Pamięć adresowana zawartością. https://pl.wikipedia.org/wiki/Pamięć_adresowana_zawartością. Dostęp 17.07.2022.

[10] Piotr Ratajczak. *Słownik specjalistyczny elektronika*, page 66. Kanion, 2005.

[11] IEEE Kostas Pagiamtzis and Ali Sheikholeslami. Content-addressable memory (cam) circuits and architectures: A tutorial and survey. *IEEE*, page 1, 2006.

[12] Charles Platt. *Elektronika od praktyki od teori*, chapter 19, pages 181–197. Helion, 2013.

[13] Paweł Borkowski. *Przygoda z elektroniką*, chapter 5, pages 309–354. Helion, 2013.

[14] Anil K. Maini. *Digital Electronics*, chapter 9, pages 299–355. john wiley & sons ltd, 2007.

[15] D.K. KAUSHIK. *Digital Electronics*, chapter 12.4. Dhanpat rai publishing company, 2005.

[16] IEEE. Standard vhdl language reference manual. Standard IEEE 1076, IEEE, January 2000.

[17] IEEE. Standard for systemverilog - unified hardware design, specification, and verification language. Standard IEEE 1800-2012, IEEE, December 2017.

[18] Ankit K Zalawadiya. Review paper on comparison of verilog and systemverilog. *Department of EC, Institute Of Technology, Nirma University*, 2018.

[19] Stephen Bailey. Comparison of vhdl, verilog and systemverilog. Raport, Model Technology, 2023.

[20] IEEE. Standard vhdl language reference manual. Standard IEEE 1076, IEEE, January 2000. Chapter 1.1.1.1 Page 6.

[21] Fabrizio Tappero Bryan Mealy. *Free Range VHDL*, chapter 8.2, pages 127–128. free range factory, 2013.

[22] Zbigniew Skoczylas Marian Gewert. *Analiza Matematyczna 1 Definicje, twierdzenia, wzory*, pages 15–16. GiS, 2001.

[23] Juvet Karnel Sadié René Ndoundam. Collision-resistant hash function based on composition of functions vol 11. *ARIMA*, pages 168 – 169, 2011.

[24] Richa Purohit Abhay Bansal, Upendra Mishra. Design and analysis of a new hash algorithm with key integration. *International Journal of Computer Applications*, page 33, 2013.

[25] Piotr Ratajczak. *Słownik specjalistyczny elektronika*, page 101. Kanion, 2005.

[26] Paweł Kardaś. *Magistrala Can od A do Z*, chapter 3.3.1, page 38. ATNEL, 2020.

[27] Slawosz Uznanski. *Single Event Upsets in Sub-65nm CMOS technologies: Monte-Carlo simulations and contribution to understanding of physical mechanisms*, chapter 1. Lambert, 2011.

[28] Z. Kohavi and N. Jha. *Switching and Finite Automata Theory Third Edition*, chapter 11.2, pages 339–345. cambridge university press, 2010.

[29] David A. Patterson John L. Hennessy. *Computer Organization and Design*, chapter 4.7-4.8, pages 303–324. Elsevier Science & Technology, 2020.