Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Kierunek: **Informatyka techniczna (ITE)**
Przedmiot: **Niezawodność i Diagnostyka układów cyfrowych**

SPRAWOZDANIE Z PROJEKTU

Jednostka arytmetyczna z detekcją błędów

Krzysztof Czerniachowicz

Prowadzący
Dr hab inż. Stanisław Piestrak prof. uczelni

WROCŁAW, 2025

Spis treści

Spis rysunków

Rozdział 1

Wstęp

1.1. Cel i zakres pracy

Celem niniejszej pracy jest analiza niezawodności układów arytmetycznych narażonych na występowanie błędów przemijających, na przykładzie czterobitowego sumatora wyposażonego w układ samosprawdzający oparty na arytmetyce resztowej modulo 3. W szczególności badaniu poddano skuteczność detekcji błędów poprzez symulację pojedynczych uszkodzeń logicznych typu *stuck-at-0* oraz *stuck-at-1*.

Praca obejmuje zarówno część teoretyczną - przedstawiającą podstawy działania kodów resztowych i ich właściwości w kontekście niezawodności układów cyfrowych - jak i część eksperymentalną, obejmującą symulację działania układu oraz analizę uzyskanych wyników.

1.2. Tło i znaczenie problemu

Wraz z miniaturyzacją technologii półprzewodnikowych rośnie podatność układów cyfrowych na zakłócenia promieniowania kosmicznego i jonizującego. Cząstki wysokoenergetyczne mogą powodować chwilowe zmiany stanów logicznych w tranzystorach, prowadząc do wystąpienia błędów przemijających (*soft errors*). Tego typu zjawiska nie powodują trwałego uszkodzenia sprzętu, lecz mogą skutkować błędnymi wynikami obliczeń.

W celu ograniczenia wpływu takich błędów opracowano szereg metod detekcji i korekcji, w tym techniki oparte na kodach resztowych, które pozwalają na bieżące sprawdzanie poprawności obliczeń bez znaczącego wzrostu złożoności układu. Kody resztowe, a w szczególności kod modulo 3, umożliwiają realizację tzw. układów samosprawdzających, w których każda operacja arytmetyczna ma równoległy tor kontrolny pracujący w arytmetyce resztowej.

1.3. Założenia eksperymentu

W ramach eksperymentu zbudowano model kombinacyjnego układu arytmetycznego składającego się z:

- czterobitowego sumatora głównego (RCA i CLA),
- generatora reszty modulo 3 z pięciobitowego wyniku sumowania,
- sumatora kontrolnego realizującego operacje w arytmetyce modulo 3,
- komparatora porównującego reszty toru głównego i kontrolnego.

Do modelu wprowadzono mechanizm wstrzykiwania pojedynczych uszkodzeń logicznych typu *stuck-at*, umożliwiający wymuszenie stałej wartości logicznej na wyjściu wybranego elementu układu. Każde uszkodzenie aktywowane jest indywidualnie i analizowane niezależnie.

Dla każdej kombinacji wejść oraz aktywnego uszkodzenia porównywany jest wynik toru głównego z wartością referencyjną, a sygnał błędu generowany przez tor kontrolny służy do klasyfikacji zachowania układu.

Wyniki symulacji klasyfikowane są jako:

- **błąd wykryty (true positive)** – wynik arytmetyczny jest niepoprawny, a układ zgłasza błąd,
- **błąd niewykryty (false negative)** – wynik arytmetyczny jest niepoprawny, lecz błąd nie został zgłoszony,
- **fałszywy alarm (false positive)** – wynik arytmetyczny jest poprawny, lecz układ błędnie zgłasza błąd,
- **brak błędu (true negative)** – wynik poprawny i brak sygnalizacji błędu.

Takie podejście pozwala na ilościową ocenę skuteczności detekcji błędów przez układ samosprawdzający oraz identyfikację elementów najbardziej wrażliwych na uszkodzenia logiczne.

Rozdział 2

Podstawy teoretyczne

2.1. Model błędów i uszkodzeń

W układach cyfrowych występują trzy główne typy nieprawidłowości: **uszkodzenia** (*faults*), **błędy** (*errors*) i **awarie** (*failure*).[1] Uszkodzenie jest zjawiskiem fizycznym lub logicznym powodującym trwałe lub chwilowe odstępstwo działania układu od jego specyfikacji, natomiast błąd jest skutkiem tego uszkodzenia widocznym w danych lub wynikach obliczeń. Jeśli błąd nie zostanie wykryty, może prowadzić do awarii systemu, objawiającej się nieprawidłowym działaniem całego układu.

Szczególne znaczenie w kontekście współczesnych technologii mają **uszkodzenia (jakie?)** (*transient faults*). Powstają one w wyniku zjawisk losowych, takich jak oddziaływanie cząstek promieniowania kosmicznego lub promieniowania jonizującego, które chwilowo zaburzają stan logiczny tranzystorów lub węzłów sygnałowych. Takie zjawiska nie powodują trwałego uszkodzenia struktury układu, ale mogą prowadzić do błędnych wyników obliczeń — szczególnie w urządzeniach reprogramowalnych, takich jak FPGA. Model uszkodzenia (jakiego) w układach kombinacyjnych często opisuje się jako **uszkodzenie typu stuck-at**, polegające na trwałym wymuszeniu wartości logicznej „0” lub „1” na danym sygnale, niezależnie od rzeczywistej funkcji logicznej.

2.2. Metody wykrywania błędów w układach arytmetycznych

Wykrywanie błędów w czasie rzeczywistym (*on-line error detection*) jest kluczowe dla systemów o podwyższonej niezawodności. W praktyce stosuje się kilka podejść, różniących się skutecznością i złożonością sprzętową:

2.2.1. Kod parzystości

Najprostszym mechanizmem wykrywania błędów jest **kod parzystości**, w którym do każdego słowa binarnego dodaje się dodatkowy bit informujący o parzystości liczby jedynek. Kod ten umożliwia wykrycie wszystkich pojedynczych błędów bitowych oraz błędów o nieparzystej krotności. Jego główną wadą jest jednak niska skuteczność w przypadku układów arytmetycznych, gdzie występuje propagacja przeniesień — nawet pojedyncze uszkodzenie może spowodować zmianę kilku bitów, a wynikowy błąd może mieć parzystą krotność i pozostać niewykryty. Ponadto, aby układ był samosprawdzający, cała struktura arytmetyczna musiałaby być przeprojektowana w celu przewidywania i propagacji bitów parzystości.

2.2.2. Metoda duplikacji z porównywaniem

Drugim klasycznym podejściem jest **duplikacja z porównywaniem** (*duplication with comparison, DWC*), w której funkcjonalny blok obliczeniowy jest zdublowany, a wyniki obu torów są porównywane przez komparator. Rozwiązanie to pozwala na wykrycie dowolnego pojedynczego błędu w jednym z torów, jednak kosztem dwukrotnego zwiększenia liczby zasobów sprzętowych i poboru mocy. Z tego względu DWC stosuje się głównie w systemach krytycznych, gdzie koszt jest drugorzędny wobec niezawodności.

2.2.3. Kody resztowe (Residue Codes)

Kody resztowe (*residue codes*) stanowią efektywną metodę wykrywania błędów w układach arytmetycznych. Wykorzystują one własności arytmetyki modularnej, przypisując każdej liczbie dodatkową część kontrolną — jej resztę z dzielenia przez ustaloną podstawę A:

$$|X|_A = X \bmod A.$$

W układzie chronionym kodem resztowym obliczenie zasadnicze i jego odpowiednik modularny są wykonywane równolegle, a następnie porównywane. Jeśli wyniki nie są przystające modulo A, oznacza to wystąpienie błędu.

Kody resztowe są kodami **separowalnymi**, co pozwala dodać tor kontrolny bez modyfikacji głównego układu arytmetycznego. Wyróżniają się niskim kosztem implementacji i wysoką skutecznością detekcji — wykrywają wszystkie błędy pojedyncze oraz znaczną część błędów wielokrotnych.

Szczególnie korzystny w praktyce jest kod **modulo 3**, który wymaga jedynie dwóch bitów kontrolnych i prostych bloków logicznych, dzięki czemu znajduje zastosowanie w samosprawdzających sumatorach i jednostkach arytmetycznych.

2.3. Arytmetyka resztowa

Arytmetyka resztowa (zwana także modularną) jest działem matematyki zajmującym się operacjami na liczbach całkowitych z uwzględnieniem ich reszty z dzielenia przez pewną ustaloną liczbę całkowitą m, nazywaną **modułem**. W arytmetyce tej interesują nas nie same wartości liczb, lecz ich reszty z dzielenia przez moduł, co pozwala znacząco uprościć obliczenia i ograniczyć zakres wartości liczbowych.[2]

Działania w arytmetyce modularnej znajdują szerokie zastosowanie w informatyce i elektronice, m.in. w algorytmach kryptograficznych, kodach korekcyjnych oraz układach samosprawdzających, takich jak stosowane w niniejszym projekcie.

2.3.1. Przystawanie liczb

Podstawowym pojęciem w arytmetyce modularnej jest **przystawanie liczb**. Dwie liczby całkowite a oraz b nazywamy **przystającymi modulo** m, jeśli ich różnica jest całkowitą wielokrotnością liczby m. Formalnie zapisuje się to jako:

$$a \equiv b \pmod{m} \quad \Leftrightarrow \quad m \mid (a - b)$$

co oznacza, że liczby a i b dają tę samą resztę z dzielenia przez m. Na przykład:

$$14 \equiv 2 \pmod{12}$$

ponieważ $14 - 2 = 12$, a więc różnica jest podzielna przez 12.

Przystawanie jest relacją równoważności i zachowuje zgodność wobec działań arytmetycznych. Jeśli zachodzą przystawania:

$$a \equiv b \pmod{m} \quad i \quad c \equiv d \pmod{m},$$

to również:

$$a + c \equiv b + d \pmod{m}$$

oraz

$$a \cdot c \equiv b \cdot d \pmod{m}.$$

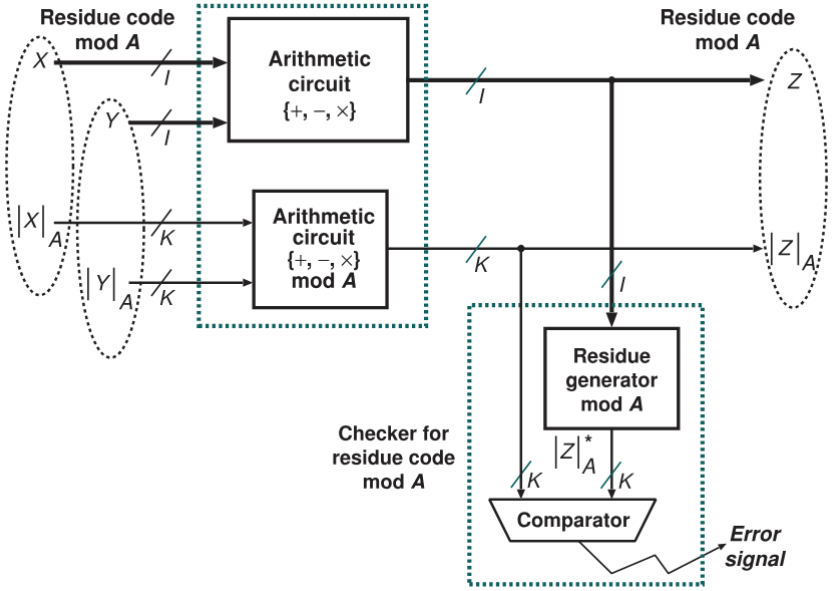
Oznacza to, że dodawanie i mnożenie liczb przystających zachowuje przystawanie – co stanowi podstawę działania układów arytmetycznych w kodach resztowych.

Rozdział 3

Plan eksperymentu

3.1. Struktura układu

Badany układ stanowi przykład arytmetycznego układu samosprawdzającego opartego na kodzie resztowym modulo 3. Jego struktura została zaprojektowana w postaci dwóch równoległych torów obliczeniowych: toru głównego realizującego operację sumowania w arytmetyce binarnej oraz toru kontrolnego pracującego w arytmetyce modulo 3. Ogólną strukturę układu przedstawiono na rys. 3.1.



Rysunek 3.1: Schematyczny model arytmetyczego układu samosprawdzającego modulo 3. Źródło: Design of the coarse-grained reconfigurable architecture DART with on-line error detection.[1]

W badanym rozwiązaniu obliczenia wykonywane są równolegle w dwóch torach:

- torze głównym, w którym realizowane jest klasyczne sumowanie binarne,
- torze kontrolnym, w którym operacje wykonywane są w arytmetyce modulo 3.

Porównanie wyników obu torów umożliwia wykrycie rozbieżności będących skutkiem uszkodzeń logicznych w strukturze układu.

3.1.1. Sumator i generator reszt modulo 3

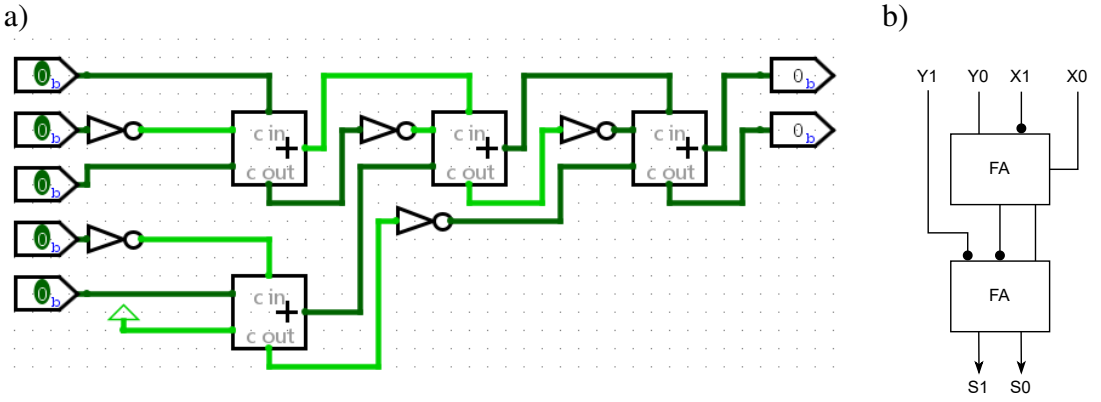
Tor kontrolny układu realizuje operacje w arytmetyce resztowej modulo 3, przy czym reszty operandów wejściowych $A_{\text{mod }3}$ oraz $B_{\text{mod }3}$ są w badanym modelu dostarczane bezpośrednio na wejście toru kontrolnego i nie są wyznaczane wewnątrz układu.

Generator reszty modulo 3 dla wyniku sumowania binarnego oblicza wartość

$$S_{\text{mod }3} = (A + B) \bmod 3$$

na podstawie pięciobitowego wyjścia głównego toru arytmetycznego. Struktura generatora oparta jest na kaskadzie sumatorów pełnych, zgodnie z rozwiązaniami opisanymi w artykule [1]. Schemat generatora reszty modulo 3 zastosowany w eksperymencie przedstawiono na rys. 3.2a).

Zastosowana architektura wykorzystuje własność sumatora pełnego polegającą na redukcji trzech sygnałów binarnych do dwóch. Kolejne stopnie generatora stopniowo zmniejszają liczbę sygnałów reprezentujących wartość wejściową, aż do uzyskania dwubitowej reprezentacji reszty modulo 3. W generatorze reszty dla pięciobitowej wartości zastosowano cztery bloki FA. Struktura ta jest równoważna generatorowi sześciobitowemu, przy czym najwyższy bit wejściowy jest na stałe ustawiony na wartość logiczną 1. Odzwierciedlenie w kodzie widoczne jest na listingu A.8.



Rysunek 3.2: a) Schemat 5 bitowego generatora reszt mod3, b) Sumator mod3

Sumator modulo 3 realizuje operację

$$(A_{\text{mod }3} + B_{\text{mod }3}) \bmod 3$$

i stanowi element toru kontrolnego odpowiedzialny za wyznaczenie reszty kontrolnej na podstawie operandów wejściowych. Jego struktura wewnętrzna jest analogiczna do generatora reszty modulo 3 dla czterech sygnałów wejściowych i również opiera się na kaskadzie sumatorów pełnych. W implementacji A.7 wykorzystano dwa bloki FA. Schemat sumatora modulo 3 przedstawiono na rys. 3.2b).

3.1.2. Komparator reszt modulo 3

Komparator reszt modulo 3 odpowiada za porównanie wartości reszt uzyskanych w torze głównym oraz torze kontrolnym. W odróżnieniu od klasycznego porównania binarnego, układ ten musi uwzględniać **podwójną reprezentację zera**, charakterystyczną dla kodu resztowego modulo 3, w której kombinacje bitowe `00` oraz `11` reprezentują tę samą wartość reszty równą 0.

Wyjście komparatora c przyjmuje wartość:

$$c = 0 \quad \text{(reszty równe)}, \qquad c = 1 \quad \text{(reszty różne)}.$$

Zgodnie z pracą [1], funkcja logiczna realizowana przez komparator reszt modulo 3 dana jest zależnością:

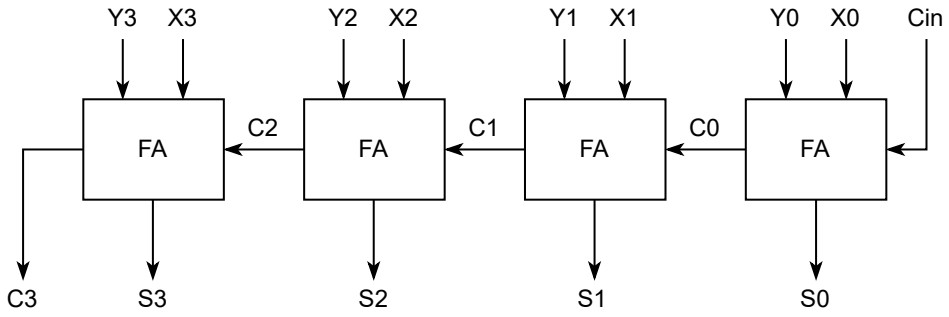
$$c = (\bar{y}_1 y_0) \oplus (\bar{x}_1 x_0) + (y_1 \bar{y}_0) \oplus (x_1 \bar{x}_0). \tag{3.1}$$

Zastosowana funkcja w sposób jawny eliminuje wpływ podwójnej reprezentacji zera, ponieważ kombinacje odpowiadające wartościom `00` oraz `11` nie aktywują żadnego z iloczynów logicznych. Dzięki temu sygnał wyjściowy komparatora wskazuje wyłącznie na rzeczywistą niezgodność wartości reszt modulo 3, niezależnie od ich binarnej reprezentacji. Bezpośrednia implementacja pokazana jest na listingu A.9.

3.1.3. Badane układy arytmetyczne

W ramach eksperymentu analizie poddano dwa czterobitowe układy sumujące: sumator kaskadowy typu Ripple Carry Adder (RCA) oraz sumator z przyspieszonym przeniesieniem typu Carry Lookahead Adder (CLA).

Sumator RCA zbudowany jest z czterech sumatorów pełnych połączonych kaskadowo. Każdy stopień realizuje operację sumowania jednego bitu operandów wraz z przeniesieniem z poprzedniego stopnia. Sygnał przeniesienia propagowany jest sekwencyjnie przez wszystkie sumatory pełne, co powoduje, że czas ustalania wyniku zależy liniowo od liczby bitów. Schemat badanego sumatora RCA przedstawiono na rys. 3.3, a implementacje na listingu A.3.



Rysunek 3.3: Schemat układu sumatora 4-bitowego RCA

Sumator CLA posiada odmienną strukturę wewnętrzną. Składa się on z zestawu sumatorów częściowych 3.4 A.4, które dla każdego bitu operandów generują sygnały sumy S, propagacji P oraz generacji G. Na podstawie sygnałów P i G wyznaczane są przeniesienia w dedykowanym module obliczania przeniesień. W pracy przedstawiono również schemat sumatora częściowego zrealizowanego na poziomie bramek logicznych. Schemat całego układu CLA pokazano na rys. 3.5, implementacja znajduje się na listingu A.11.

Moduł obliczania przeniesień w sumatorze CLA realizuje zależności:

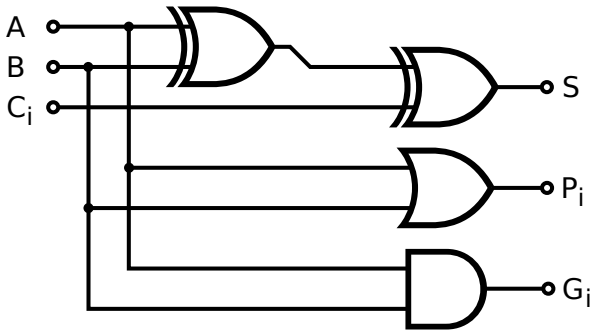
$$C_1 = G_0 + P_0 \cdot C_0$$

$$C_2 = G_1 + P_1 \cdot C_1$$

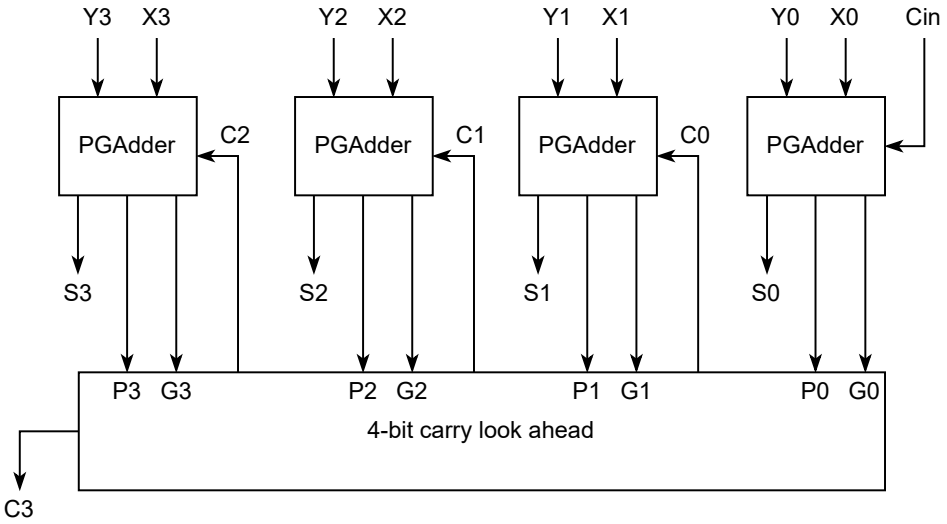
$$C_3 = G_2 + P_2 \cdot C_2$$

$$C_4 = G_3 + P_3 \cdot C_3$$

gdzie C_0 oznacza wejściowe przeniesienie do sumatora.



Rysunek 3.4: Schemat układu sumatora częściowego z wyjściami generacji i propagacji



Rysunek 3.5: Schemat układu sumatora 4-bitowego CLA

3.2. Implementacja modelu symulacyjnego

Model badanego układu został zaimplementowany w języku opisu sprzętu *Verilog* i symulowany w środowisku *ModelSim*. Implementacja obejmuje zarówno tor główny realizujący sumowanie w arytmetyce binarnej, jak i tor kontrolny oparty na arytmetyce resztowej modulo 3. Każdy z bloków przedstawionych wcześniej w pracy posiada odpowiadający mu model strukturalny wykorzystywany w symulacji.

Projekt przyjęto w postaci hierarchicznej. Elementarne komponenty, takie jak sumatory pełne czy sumatory częściowe, zostały opisane jako niezależne moduły i następnie złożone w bardziej złożone struktury, w tym sumatory RCA i CLA. Takie podejście umożliwia jednoznaczne wskazanie punktów potencjalnych uszkodzeń oraz ich systematyczne wstrzykiwanie.

Modele układów mają charakter funkcjonalny i są wykorzystywane wyłącznie do analizy poprawności logicznej oraz skuteczności detekcji błędów.

zaprojektowana w celu analizy odporności układu na pojedyncze uszkodzenia przemijające (ang. *transient faults*), których fizycznym źródłem mogą być m.in. cząstki promieniowania kosmicznego powodujące chwilową zmianę stanu logicznego. Na potrzeby symulacji każde uszkodzenie modelowane jest w postaci sklejenia stanu logicznego typu *stuck-at*, polegającego na wymuszeniu stałej wartości logicznej 0 lub 1 na wyjściu wybranego komponentu.

Przyjęto założenie, że uszkodzeniu może ulec dowolny sygnał wyjściowy elementarnego komponentu budującego badany układ. Za komponenty elementarne uznano:

- sumatory pełne (FA) A.2 w strukturze sumatora RCA, generatora reszt modulo 3 oraz sumatora modulo 3,
- sumatory częściowe A.4 w strukturze sumatora CLA,
- moduł generowania przeniesień A.5 w sumatorze CLA,
- moduł komparatora modulo 3 A.9.

Każdy z wymienionych komponentów został w modelu symulacyjnym wyposażony w dodatkowy moduł `fault_mux` A.1, umożliwiający wymuszenie uszkodzenia na jego wyjściu. Sterowanie iniekcją błędu realizowane jest przy pomocy wspólnej magistrali `fault_en_bus`, co pozwala na jednoznaczne i systematyczne testowanie wszystkich pojedynczych punktów potencjalnej awarii.

3.2.1. Zakres testów

Testy wykonywane są dla wszystkich 4-bitowych kombinacji operandów A i B. Dla każdej konfiguracji wejść wyznaczana jest referencyjna suma binarna S_{ref} oraz generowany jest wynik badanego układu wraz z sygnałem błędu `err` pochodzącym z toru kontrolnego.

Analiza prowadzona jest dla następujących wariantów pracy układu:

- brak aktywnego uszkodzenia,
- pojedyncze uszkodzenie typu *stuck-at-0*,
- pojedyncze uszkodzenie typu *stuck-at-1*.

W przypadku testów z iniekcją błędu aktywowany jest dokładnie jeden sygnał odpowiadający wyjściu wybranego komponentu elementarnego. Każde uszkodzenie analizowane jest niezależnie dla pełnego zbioru kombinacji wejściowych.

3.2.2. Kryterium oceny

Dla każdej kombinacji wejść wynik badanego układu porównywany jest z wartością referencyjną S_{ref}, a następnie analizowany jest stan sygnału `err`. Na tej podstawie przypadki klasyfikowane są do jednej z czterech kategorii:

- **Błąd wykryty (TP, *True Positive*)** — wynik niezgodny z referencyjnym oraz `err = 1`,
- **Brak błędu (TN, *True Negative*)** — wynik zgodny z referencyjnym oraz `err = 0`,
- **Fałszywy alarm (FP, *False Positive*)** — wynik zgodny z referencyjnym oraz `err = 1`,
- **Błąd niewykryty (FN, *False Negative*)** — wynik niezgodny z referencyjnym oraz `err = 0`.

3.2.3. Cel badania

Celem przeprowadzonych testów jest:

- potwierdzenie, że żadne pojedyncze uszkodzenie nie prowadzi do niewykrytego błędu arytmetycznego,
- ilościowa ocena liczby fałszywych alarmów generowanych przez układ kontrolny.

Rozdział 4

Analiza wyników

W niniejszym rozdziale przedstawiono analizę wyników symulacji odporności na pojedyncze uszkodzenia typu *stuck-at* dla dwóch wariantów sumatora binarnego: RCA oraz CLA. W obu przypadkach analizie poddano zarówno tor arytmetyczny, jak i tor kontrolny oparty na kodzie resztowym modulo 3.

4.1. Wyniki symulacji dla układu RCA

Listing 4.1: Wyniki symulacji dla układu RCA A.10

```
# === START SYMULACJI ===
# === RCA ref ===
# TP=0
# TN=256
# FP=0
# FN=0
# === RCA ===
# TP=2048
# TN=2048
# FP=0
# FN=0
# === Mod3 Adder ===
# TP=0
# TN=1024
# FP=1024
# FN=0
# === Komparator ===
# TP=0
# TN=768
# FP=256
# FN=0
# === Generator reszt mod3 ===
# TP=0
# TN=2048
# FP=2048
# FN=0
# === KONIEC SYMULACJI ===
```

Najważniejszą obserwacją dla układu RCA jest całkowity brak błędów niewykrytych (**FN = 0**) we wszystkich badanych przypadkach. Oznacza to, że każde uszkodzenie prowadzące do błędu arytmetycznego zostało poprawnie zasygnalizowane przez tor kontrolny.

W przypadku samego sumatora RCA uszkodzenia wstrzykiwane do toru arytmetycznego powodowały błąd wyniku w dokładnie 50% przypadków. Pozostałe 50% uszkodzeń nie miało wpływu na wynik obliczeń. Charakterystyczne jest przy tym równomierne rozłożenie przypadków *TP* i *TN*.

Uszkodzenia wprowadzane do elementów toru kontrolnego nie powodowały błędów wyniku, lecz skutkowały pojawieniem się fałszywych alarmów (*FP*). Dla generatora reszt modulo 3 oraz sumatora modulo 3 liczba fałszywych alarmów była równa liczbie przypadków gdzie uszkodzenie nie powodowało błędu ($FP = TN$), natomiast w przypadku komparatora fałszywy alarm występował w 25% badanych przypadków.

4.2. Wyniki symulacji dla układu CLA

Listing 4.2: Wyniki symulacji dla układu CLA A.11

```
# === START SYMULACJI ===
# === CLA ref ===
# TP=0
# TN=256
# FP=0
# FN=0
# === CLA ===
# TP=3140
# TN=5052
# FP=0
# FN=0
# === Mod3 Adder ===
# TP=0
# TN=1024
# FP=1024
# FN=0
# === Komparator ===
# TP=0
# TN=768
# FP=256
# FN=0
# === Generator reszt mod3 ===
# TP=0
# TN=2048
# FP=2048
# FN=0
# === KONIEC SYMULACJI ===
```

Podobnie jak w przypadku układu RCA, również dla sumatora CLA nie odnotowano żadnego błędu niewykrytego (**FN = 0**). Oznacza to, że zastosowany mechanizm samosprawdzania zachowuje pełną skuteczność detekcji pojedynczych uszkodzeń także dla bardziej złożonej struktury arytmetycznej.

W przeciwieństwie do RCA, w układzie CLA odsetek uszkodzeń prowadzących do błędu wyniku jest wyraźnie mniejszy i wynosi około 38%. Większość wstrzykiwanych uszkodzeń zo-

staje zamaskowana przez strukturę sumatora, co można wiązać z równoległym wyznaczaniem przeniesień oraz większą liczbą ścieżek logicznych.

Zachowanie toru kontrolnego dla układu CLA jest identyczne jak w przypadku RCA. Uszkodzenia wprowadzane do generatora reszt modulo 3, sumatora modulo 3 oraz komparatora prowadzą wyłącznie do fałszywych alarmów i nie wpływają na poprawność wyniku arytmetycznego. Jest to bezpośrednią konsekwencją separacji toru obliczeniowego i kontrolnego oraz stanowi oczekiwany efekt przyjętej architektury.

Rozdział 5

Podsumowanie i wnioski końcowe

Celem niniejszej pracy była analiza skuteczności detekcji pojedynczych uszkodzeń przemijających w arytmetycznych układach samosprawdzających opartych na kodzie resztowym modulo 3. Badaniom poddano dwie struktury sumatorów binarnych: sumator z propagacją przeniesienia (RCA) oraz sumator z wyprzedzającym generowaniem przeniesień (CLA), współpracujące z torem kontrolnym realizującym obliczenia w arytmetyce modulo 3.

Modele badanych układów zostały zaimplementowane w języku Verilog i poddane symulacji w środowisku ModelSim. Przyjęty model uszkodzeń zakładał możliwość wystąpienia pojedynczego uszkodzenia typu *stuck-at-0* lub *stuck-at-1* na dowolnym sygnale wyjściowym elementarnych komponentów, takich jak sumatory, generatory przeniesień, moduły resztowe oraz komparator. Takie podejście umożliwiło systematyczne i jednoznaczne przetestowanie wszystkich potencjalnych punktów awarii w strukturze układu.

Przeprowadzone eksperymenty wykazały, że zarówno w przypadku struktury RCA, jak i CLA, nie wystąpił ani jeden przypadek niewykrytego błędu arytmetycznego (*False Negative*). Oznacza to, że zastosowany mechanizm kontroli modulo 3 zapewnia pełne pokrycie pojedynczych uszkodzeń prowadzących do błędnego wyniku obliczeń. Z punktu widzenia niezawodności obliczeń jest to wynik jednoznacznie pozytywny i potwierdzający własności samosprawdzające badanych układów.

Jednocześnie zaobserwowano, że znaczna część wstrzykiwanych uszkodzeń w torze arytmetycznym nie wpływa na końcowy wynik sumowania. W przypadku sumatora RCA dokładnie połowa testowanych uszkodzeń nie powodowała zmiany wyniku, co wynika z naturalnej redundancji logicznej oraz maskowania błędów w strukturze propagacji przeniesień. W strukturze CLA odsetek uszkodzeń prowadzących do błędnego wyniku był niższy i wynosił około 38%, co można wiązać z bardziej złożoną strukturą i większą ilością sygnałów wewnętrznych.

Uszkodzenia wstrzykiwane w elementy toru kontrolnego nie prowadziły do błędów arytmetycznych, lecz skutkowały występowaniem fałszywych alarmów (*False Positive*). Zjawisko to było szczególnie widoczne w przypadku generatora reszt modulo 3 oraz sumatora modulo 3, gdzie liczba fałszywych alarmów była porównywalna z liczbą przypadków poprawnej pracy. Dla komparatora modulo 3 fałszywy alarm występował w około 25% przypadków. Taki charakter błędów jest zgodny z oczekiwaniami i stanowi kompromis w układach samosprawdzających pomiędzy pełną detekcją błędów a nadmiarowością sygnałów alarmowych.

Podsumowując, wyniki pracy potwierdzają, że arytmetyczne układy samosprawdzające oparte na kodzie resztowym modulo 3 skutecznie wykrywają wszystkie pojedyncze uszkodzenia prowadzące do błędnych wyników obliczeń, zarówno w strukturach RCA, jak i CLA. Różnice pomiędzy badanymi sumatorami ujawniają się głównie w podatności toru arytmetycznego na uszkodzenia, natomiast zachowanie toru kontrolnego pozostaje niezależne od zastosowanej architektury sumatora.

Bibliografia

[1] S. Jafri, S. Piestrak, O. Sentieys, S. Pillement. Design of the coarse-grained reconfigurable architecture dart with on-line error detection. *Microprocessors and Microsystems*, 38(2):124–136, 2014.

[2] Wikipedia contributors. Arytmetyka modularna. `https://pl.wikipedia.org/wiki/Arytmetyka_modularna`, dostęp: 2025-11-04.

Dodatek A

Listingi źródłowe

Listing A.1: Moduł multipleksera wstrzykującego uszkodzenie

```
module fault_mux #(
    parameter NG = 128,
    parameter GID = 0
)(
    input  wire in,
    input  wire [NG-1:0] fault_en_bus,
    input  wire fault_val,
    output wire out
);
    assign out = (fault_en_bus[GID]) ? fault_val : in;
endmodule
```

Listing A.2: Moduł sumatora pełnego

```
module FA #(
        parameter NG = 128,
        parameter GID_SUM = 0,
        parameter GID_COUT = 1
)(
        input wire a, b, cin,
        input wire [NG-1:0] fault_en_bus,
        input wire fault_val,
        output wire sum,
        output wire cout
);
        wire sum_int = a ^ b ^ cin;
        wire cout_int = (a & b) | (a & cin) | (b & cin);

        //fault injection
        fault_mux #(.GID(GID_SUM), .NG(NG)) fm_sum (sum_int,
           ↪ fault_en_bus, fault_val, sum);
        fault_mux #(.GID(GID_COUT), .NG(NG)) fm_cout (cout_int,
           ↪ fault_en_bus, fault_val, cout);
endmodule
```

Listing A.3: Moduł sumatora 4 bitowego RCA

```
module RCA #(
```

```
        parameter NG = 128,
        parameter GID_BASE = 0
)(
        input wire [3:0] X,
        input wire [3:0] Y,
        input wire Cin,
        input wire [NG-1:0] fault_en_bus,
        input wire fault_val,
        output wire [3:0] S,
        output wire Cout
);
        wire s0, c0;
        FA #(
                .NG(NG),
                .GID_SUM(GID_BASE + 0),
                .GID_COUT(GID_BASE + 1)
        ) fa0 (
                .a(X[0]), .b(Y[0]), .cin(Cin),
                .fault_en_bus(fault_en_bus),
                .fault_val(fault_val),
                .sum(s0),
                .cout(c0)
        );

        wire s1, c1;
        FA #(
                .NG(NG),
                .GID_SUM(GID_BASE + 2),
                .GID_COUT(GID_BASE + 3)
        ) fa1 (
                .a(X[1]), .b(Y[1]), .cin(c0),
                .fault_en_bus(fault_en_bus),
                .fault_val(fault_val),
                .sum(s1),
                .cout(c1)
        );

        wire s2, c2;
        FA #(
                .NG(NG),
                .GID_SUM(GID_BASE + 4),
                .GID_COUT(GID_BASE + 5)
        ) fa2 (
                .a(X[2]), .b(Y[2]), .cin(c1),
                .fault_en_bus(fault_en_bus),
                .fault_val(fault_val),
                .sum(s2),
                .cout(c2)
        );

        wire s3, c3;
        FA #(
                .NG(NG),
```

```
                .GID_SUM(GID_BASE + 6),
                .GID_COUT(GID_BASE + 7)
        ) fa3 (
                .a(X[3]), .b(Y[3]), .cin(c2),
                .fault_en_bus(fault_en_bus),
                .fault_val(fault_val),
                .sum(s3),
                .cout(c3)
        );

        assign S = {s3, s2, s1, s0};
        assign Cout = c3;
endmodule
```

Listing A.4: Moduł sumatora częściowego z sygnałami generacji i przeniesienia

```
module PGA #(
    parameter NG = 128,
    parameter GID_S = 0,
    parameter GID_P = 1,
    parameter GID_G = 2
)(
    input  wire A,
    input  wire B,
    input  wire Cin,
    input  wire [NG-1:0] fault_en_bus,
    input  wire fault_val,
    output wire S,
    output wire P,
    output wire G
);

    wire p_int = A ^ B;        // propagacja
    wire g_int = A & B;        // generacja
    wire s_int = p_int ^ Cin; // suma

    // fault injection na kazdym sygnale
    fault_mux #(.GID(GID_P), .NG(NG)) fm_p (p_int, fault_en_bus,
       ↪ fault_val, P);
    fault_mux #(.GID(GID_G), .NG(NG)) fm_g (g_int, fault_en_bus,
       ↪ fault_val, G);
    fault_mux #(.GID(GID_S), .NG(NG)) fm_s (s_int, fault_en_bus,
       ↪ fault_val, S);
endmodule
```

Listing A.5: Moduł generatora przeniesień w CLA

```
module CLA_CarryGen #(
    parameter NG = 128,
    parameter GID_C1 = 0,
    parameter GID_C2 = 1,
    parameter GID_C3 = 2,
    parameter GID_C4 = 3
```

```
)(
    input wire C0,
    input wire [3:0] P,
    input wire [3:0] G,
    input wire [NG-1:0] fault_en_bus,
    input wire fault_val,
    output wire C1,
    output wire C2,
    output wire C3,
    output wire C4
);

    // surowe przeniesienia
    wire C1_int = G[0] | (P[0] & C0);
    wire C2_int = G[1] | (P[1] & C1_int);
    wire C3_int = G[2] | (P[2] & C2_int);
    wire C4_int = G[3] | (P[3] & C3_int);

    // wstrzykniecia uszkodzen
    fault_mux #(.GID(GID_C1), .NG(NG)) fm_c1 (C1_int, fault_en_bus,
       ↪ fault_val, C1);
    fault_mux #(.GID(GID_C2), .NG(NG)) fm_c2 (C2_int, fault_en_bus,
       ↪ fault_val, C2);
    fault_mux #(.GID(GID_C3), .NG(NG)) fm_c3 (C3_int, fault_en_bus,
       ↪ fault_val, C3);
    fault_mux #(.GID(GID_C4), .NG(NG)) fm_c4 (C4_int, fault_en_bus,
       ↪ fault_val, C4);

endmodule
```

Listing A.6: Moduł sumatora 4 bitowego CLA

```
module CLA #(
    parameter NG = 128,
    parameter GID_BASE = 0
)(
    input wire [3:0] X,
    input wire [3:0] Y,
    input wire Cin,
    input wire [NG-1:0] fault_en_bus,
    input wire fault_val,
    output wire [3:0] S,
    output wire Cout
);

    wire [3:0] s_int, P, G;
    wire C1, C2, C3, C4;

// inicjalizacja 4 sumatorow
    PGA #(
        .NG(NG),
        .GID_S(GID_BASE + 0),
        .GID_P(GID_BASE + 1),
```

```
        .GID_G(GID_BASE + 2)
    ) pga0 (
        .A(X[0]),
        .B(Y[0]),
        .Cin(Cin),
        .fault_en_bus(fault_en_bus),
        .fault_val(fault_val),
        .S(s_int[0]),
        .P(P[0]),
        .G(G[0])
    );

    PGA #(
        .NG(NG),
        .GID_S(GID_BASE + 3),
        .GID_P(GID_BASE + 4),
        .GID_G(GID_BASE + 5)
    ) pga1 (
        .A(X[1]),
        .B(Y[1]),
        .Cin(C1),
        .fault_en_bus(fault_en_bus),
        .fault_val(fault_val),
        .S(s_int[1]),
        .P(P[1]),
        .G(G[1])
    );

    PGA #(
        .NG(NG),
        .GID_S(GID_BASE + 6),
        .GID_P(GID_BASE + 7),
        .GID_G(GID_BASE + 8)
    ) pga2 (
        .A(X[2]),
        .B(Y[2]),
        .Cin(C2),
        .fault_en_bus(fault_en_bus),
        .fault_val(fault_val),
        .S(s_int[2]),
        .P(P[2]),
        .G(G[2])
    );

    PGA #(
        .NG(NG),
        .GID_S(GID_BASE + 9),
        .GID_P(GID_BASE + 10),
        .GID_G(GID_BASE + 11)
    ) pga3 (
        .A(X[3]),
        .B(Y[3]),
        .Cin(C3),
```

```
        .fault_en_bus(fault_en_bus),
        .fault_val(fault_val),
        .S(s_int[3]),
        .P(P[3]),
        .G(G[3])
    );


// inicjalizacja modulu generujacego przeniesienia
    CLA_CarryGen #(
        .NG(NG),
        .GID_C1(GID_BASE + 12),
        .GID_C2(GID_BASE + 13),
        .GID_C3(GID_BASE + 14),
        .GID_C4(GID_BASE + 15)
    ) cla_gen (
        .C0(Cin),
        .P(P),
        .G(G),
        .fault_en_bus(fault_en_bus),
        .fault_val(fault_val),
        .C1(C1),
        .C2(C2),
        .C3(C3),
        .C4(C4)
    );

// przypisanie sygnalow wyjsciowych
    assign S[0] = s_int[0];
    assign S[1] = s_int[1];
    assign S[2] = s_int[2];
    assign S[3] = s_int[3];

    assign Cout = C4;

endmodule
```

Listing A.7: Moduł sumatora modulo 3

```
module adder_mod3 #(
        parameter NG = 128,
        parameter GID_BASE = 0
)(
        input wire [1:0] X,
        input wire [1:0] Y,
        input wire [NG-1:0] fault_en_bus,
        input wire fault_val,
        output wire [1:0] R
);
        wire x0 = X[0];
        wire x1 = ~X[1];
        wire y0 = Y[0];
        wire y1 = ~Y[1];
```

```
        wire s0, c0;
        FA #(
                .NG(NG),
                .GID_SUM(GID_BASE + 0),
                .GID_COUT(GID_BASE + 1)
        ) fa0 (
                .a(x1), .b(y0), .cin(x0),
                .fault_en_bus(fault_en_bus),
                .fault_val(fault_val),
                .sum(s0),
                .cout(c0)
        );

        wire s1, c1;
        FA #(
                .NG(NG),
                .GID_SUM(GID_BASE + 2),
                .GID_COUT(GID_BASE + 3)
        ) fa1 (
                .a(y1), .b(~c0), .cin(s0),
                .fault_en_bus(fault_en_bus),
                .fault_val(fault_val),
                .sum(s1),
                .cout(c1)
        );

        // Wynik modulo 3 - juz nie poddawany wstrzykiwaniu uszkodzen
        assign R = {c1, s1};
endmodule
```

Listing A.8: Moduł 5 bitowego generatora reszt modulo 3

```
module residue_mod3 #(
        parameter NG = 128,
        parameter GID_BASE = 0
)(
        input wire [4:0] A,
        input wire [NG-1:0] fault_en_bus,
        input wire fault_val,
        output wire [1:0] R
);
        wire x0 = A[0];
        wire x1 = ~A[1];
        wire x2 = A[2];
        wire x3 = ~A[3];
        wire x4 = A[4];
        wire x5 = 1;

        wire s0, s1, s2, s3;
        wire c0, c1, c2, c3;

        FA #(
                .NG(NG),
                .GID_SUM(GID_BASE + 0),
```

```
                .GID_COUT(GID_BASE + 1)
        ) fa0 (
                .a(x1), .b(x2), .cin(x0),
                .fault_en_bus(fault_en_bus),
                .fault_val(fault_val),
                .sum(s0),
                .cout(c0)
        );

        FA #(
                .NG(NG),
                .GID_SUM(GID_BASE + 2),
                .GID_COUT(GID_BASE + 3)
        ) fa1 (
                .a(x4), .b(x5), .cin(x3),
                .fault_en_bus(fault_en_bus),
                .fault_val(fault_val),
                .sum(s1),
                .cout(c1)
        );

        FA #(
                .NG(NG),
                .GID_SUM(GID_BASE + 4),
                .GID_COUT(GID_BASE + 5)
        ) fa2 (
                .a(s1), .b(~c0), .cin(s0),
                .fault_en_bus(fault_en_bus),
                .fault_val(fault_val),
                .sum(s2),
                .cout(c2)
        );

        FA #(
                .NG(NG),
                .GID_SUM(GID_BASE + 6),
                .GID_COUT(GID_BASE + 7)
        ) fa3 (
                .a(~c1), .b(~c2), .cin(s2),
                .fault_en_bus(fault_en_bus),
                .fault_val(fault_val),
                .sum(s3),
                .cout(c3)
        );

        // Wynik modulo 3 - juz nie poddawany wstrzykiwaniu uszkodzen
        assign R = {c3, s3};
endmodule
```

Listing A.9: Moduł komparatora modulo 3

```
module mod3_comparator#(
    parameter NG = 128,
    parameter GID_OUT = 0
```

```
)(
    input  wire [1:0] X,   // reszta mod3 pierwszego operand
    input  wire [1:0] Y,   // reszta mod3 drugiego operand
    input  wire [NG-1:0] fault_en_bus,
    input  wire fault_val,
    output wire eq_err     // 1 = reszty rozne, 0 = takie same
);
    wire x1 = X[1];
    wire x0 = X[0];
    wire y1 = Y[1];
    wire y0 = Y[0];

    wire f_raw =
        (~y1 & y0) ^ (~x1 & x0) |
        (y1 & ~y0) ^ (x1 & ~x0);


    // fault injection na wyjsciu komparatora
    fault_mux #(.GID(GID_OUT), .NG(NG)) fm_cmp (
        .in(f_raw),
        .fault_en_bus(fault_en_bus),
        .fault_val(fault_val),
        .out(eq_err)
    );
endmodule
```

Listing A.10: Testbench układu samosprawdzającego z sumatorem RCA

```
`timescale 1ns/1ps

module tb_RCA;

    //----------------------
    localparam NG = 128;     // liczba mozliwych GID
    // 8 mozliwych uszkodzen w RCA
    localparam GID_RCA_BASE      = 0;
    localparam GID_RCA_END       = 7;
    // 4 mozliwe uszkodzenia w adder mod3
    localparam GID_MOD3_BASE     = 20;
    localparam GID_MOD3_END      = 23;
    // 2 mozliwe uszkodzenia w komparatorze
    localparam GID_CMP_BASE      = 30;
    localparam GID_CMP_END       = 31;
    // 8 mozliwych uszkodzen w generatorze reszty mod 3
    localparam GID_RESIDUE_BASE  = 40;
    localparam GID_RESIDUE_END   = 47;
    //----------------------
    // Sygnaly testbench
    //----------------------
    reg  [3:0] A, B;
    reg  [1:0] A_mod3, B_mod3;
    reg        Cin;
```

```
reg  [NG-1:0] fault_en_bus;
reg           fault_val;

reg [4:0] ref_sum;

wire [4:0] Sout;
wire       err;

//----------------------
// DUT
//----------------------

// 1) RCA
wire [3:0] rca_s;
wire       rca_cout;

RCA #(
    .NG(NG),
    .GID_BASE(GID_RCA_BASE)
) dut_rca (
    .X(A),
    .Y(B),
    .Cin(Cin),
    .fault_en_bus(fault_en_bus),
    .fault_val(fault_val),
    .S(rca_s),
    .Cout(rca_cout)
);

// po??czone wyj?cie RCA jako 5-bit
assign Sout = {rca_cout, rca_s};


// 2) RESZTA MOD3 z wyniku RCA
wire [1:0] Rsum_mod3;

residue_mod3 #(
    .NG(NG),
    .GID_BASE(GID_RESIDUE_BASE)
) dut_mod3_sum (
    .A(Sout),
    .R(Rsum_mod3),
    .fault_en_bus(fault_en_bus),
    .fault_val(fault_val)
);

// 3) SUMA RESZT (Amod3 + Bmod3)
wire [1:0] R_ab_mod3;

adder_mod3 #(
    .NG(NG),
    .GID_BASE(GID_MOD3_BASE)
) dut_mod3_add (
```

```
        .X(A_mod3),
        .Y(B_mod3),
        .fault_en_bus(fault_en_bus),
        .fault_val(fault_val),
        .R(R_ab_mod3)
    );

    // 4) POROWNANIE RESZT
    mod3_comparator #(
        .NG(NG),
        .GID_OUT(GID_CMP_BASE)
    ) dut_cmp (
        .X(Rsum_mod3),
        .Y(R_ab_mod3),
        .fault_en_bus(fault_en_bus),
        .fault_val(fault_val),
        .eq_err(err)
    );

    //----------------------
    // test wszystkich uszkodzen
    //----------------------
    integer i;
    integer sa;   // stuck-at
    integer a, b, ra, rb;

    integer tp, fp, fn, tn;

    initial begin
        $display("===␣START␣SYMULACJI␣===");

        Cin = 0;
        fault_en_bus = 0;
        fault_val    = 0;
        tp = 0; tn = 0; fp = 0; fn = 0;

         // przej?cie po wszystkich wej?ciach A,B
         for (a = 0; a < 16; a = a + 1) begin
             for (b = 0; b < 16; b = b + 1) begin

                 A = a;
                 B = b;
                 ref_sum = A + B + Cin;

                 A_mod3 = A % 3;
                 B_mod3 = B % 3;

                 #10; // ustalenie sygnalow

                 if ((ref_sum != Sout) && (err == 0)) begin
                         fn = fn + 1;
                         $display(
```

```
                    "NIEWYKRYTY␣BLAD␣ARYTMETYCZNY:␣GID=%0d␣SA=%0d
                       ↪ ␣A=%0d␣B=%0d␣Cin=%0d␣|␣ref=%0d␣Sout=%0d
                       ↪ ",
                    i, sa, A, B, Cin, ref_sum, Sout
                    );
            end
            else if ((ref_sum != Sout) && (err == 1)) tp = tp +
               ↪ 1;
            else if ((ref_sum == Sout) && (err == 1)) fp = fp +
               ↪ 1;
            else if ((ref_sum == Sout) && (err == 0)) tn = tn +
               ↪ 1;
        end
    end

    $display("===␣RCA␣ref␣===");
    $display("TP=%0d", tp);
    $display("TN=%0d", tn);
    $display("FP=%0d", fp);
    $display("FN=%0d", fn);

    tp = 0; tn = 0; fp = 0; fn = 0;

    // Test: wszystkie uszkodzenia ukladu arytmetycznego
    for (i = GID_RCA_BASE; i <= GID_RCA_END; i = i + 1) begin

        // test stuck-at-0 i stuck-at-1
        for (sa = 0; sa <= 1; sa = sa + 1) begin

            fault_val = sa;

            // aktywuj jedno uszkodzenie
            fault_en_bus = 0;
            fault_en_bus[i] = 1;


            // przej?cie po wszystkich wej?ciach A,B
            for (a = 0; a < 16; a = a + 1) begin
                for (b = 0; b < 16; b = b + 1) begin

                    A = a;
                    B = b;
                    ref_sum = A + B + Cin;

                    A_mod3 = A % 3;
                    B_mod3 = B % 3;

                    #10; // ustalenie sygnalow

                    if ((ref_sum != Sout) && (err == 0)) begin
                        fn = fn + 1;
                        $display(
```

```
                        "NIEWYKRYTY␣BLAD␣ARYTMETYCZNY:␣GID=%0
                            ↪ d␣SA=%0d␣A=%0d␣B=%0d␣Cin=%0d␣|␣
                            ↪ ref=%0d␣Sout=%0d",
                        i, sa, A, B, Cin, ref_sum, Sout
                    );
                end
                else if ((ref_sum != Sout) && (err == 1)) tp
                    ↪ = tp + 1;
                else if ((ref_sum == Sout) && (err == 1)) fp
                    ↪ = fp + 1;
                else if ((ref_sum == Sout) && (err == 0)) tn
                    ↪ = tn + 1;
            end
        end
    end
end

$display("===␣RCA␣===");
$display("TP=%0d", tp);
$display("TN=%0d", tn);
$display("FP=%0d", fp);
$display("FN=%0d", fn);

tp = 0; tn = 0; fp = 0; fn = 0;
// Test: pozosta?e uszkodzenia w uk?adzie
for (i = GID_MOD3_BASE; i <= GID_MOD3_END; i = i + 1) begin

    // test stuck-at-0 i stuck-at-1
    for (sa = 0; sa <= 1; sa = sa + 1) begin

        fault_val = sa;

        // aktywuj jedno uszkodzenie
        fault_en_bus = 0;
        fault_en_bus[i] = 1;


        // przej?cie po wszystkich wej?ciach A,B
        for (a = 0; a < 16; a = a + 1) begin
            for (b = 0; b < 16; b = b + 1) begin

                A = a;
                B = b;
                ref_sum = A + B + Cin;

                A_mod3 = A % 3;
                B_mod3 = B % 3;

                #10; // ustalenie sygnalow

                if ((ref_sum != Sout) && (err == 0)) begin
                    fn = fn + 1;
                    $display(
```

```
                        "NIEWYKRYTY␣BLAD␣ARYTMETYCZNY:␣GID=%0
                            ↪ d␣SA=%0d␣A=%0d␣B=%0d␣Cin=%0d␣|␣
                            ↪ ref=%0d␣Sout=%0d",
                        i, sa, A, B, Cin, ref_sum, Sout
                    );
                end
                else if ((ref_sum != Sout) && (err == 1)) tp
                    ↪ = tp + 1;
                else if ((ref_sum == Sout) && (err == 1)) fp
                    ↪ = fp + 1;
                else if ((ref_sum == Sout) && (err == 0)) tn
                    ↪ = tn + 1;
            end
        end
    end
end

$display("===␣Mod3␣Adder␣===");
$display("TP=%0d", tp);
$display("TN=%0d", tn);
$display("FP=%0d", fp);
$display("FN=%0d", fn);

tp = 0; tn = 0; fp = 0; fn = 0;
for (i = GID_CMP_BASE; i <= GID_CMP_END; i = i + 1) begin

    // test stuck-at-0 i stuck-at-1
    for (sa = 0; sa <= 1; sa = sa + 1) begin

        fault_val = sa;

        // aktywuj jedno uszkodzenie
        fault_en_bus = 0;
        fault_en_bus[i] = 1;


        // przej?cie po wszystkich wej?ciach A,B
        for (a = 0; a < 16; a = a + 1) begin
            for (b = 0; b < 16; b = b + 1) begin

                A = a;
                B = b;
                ref_sum = A + B + Cin;

                A_mod3 = A % 3;
                B_mod3 = B % 3;

                #10; // ustalenie sygnalow

                if ((ref_sum != Sout) && (err == 0)) begin
                    fn = fn + 1;
                    $display(
```

```
                        "NIEWYKRYTY␣BLAD␣ARYTMETYCZNY:␣GID=%0
                           ↪ d␣SA=%0d␣A=%0d␣B=%0d␣Cin=%0d␣|␣
                           ↪ ref=%0d␣Sout=%0d",
                        i, sa, A, B, Cin, ref_sum, Sout
                    );
                end
                else if ((ref_sum != Sout) && (err == 1)) tp
                   ↪ = tp + 1;
                else if ((ref_sum == Sout) && (err == 1)) fp
                   ↪ = fp + 1;
                else if ((ref_sum == Sout) && (err == 0)) tn
                   ↪ = tn + 1;
            end
        end
    end
end

$display("===␣Komparator␣===");
$display("TP=%0d", tp);
$display("TN=%0d", tn);
$display("FP=%0d", fp);
$display("FN=%0d", fn);

tp = 0; tn = 0; fp = 0; fn = 0;
for (i = GID_RESIDUE_BASE; i <= GID_RESIDUE_END; i = i + 1)
   ↪ begin

    // test stuck-at-0 i stuck-at-1
    for (sa = 0; sa <= 1; sa = sa + 1) begin

        fault_val = sa;

        // aktywuj jedno uszkodzenie
        fault_en_bus = 0;
        fault_en_bus[i] = 1;


        // przej?cie po wszystkich wej?ciach A,B
        for (a = 0; a < 16; a = a + 1) begin
            for (b = 0; b < 16; b = b + 1) begin

                A = a;
                B = b;
                ref_sum = A + B + Cin;

                A_mod3 = A % 3;
                B_mod3 = B % 3;

                #10; // ustalenie sygnalow

                if ((ref_sum != Sout) && (err == 0)) begin
                    fn = fn + 1;
                    $display(
```

```
                                    "NIEWYKRYTY␣BLAD␣ARYTMETYCZNY:␣GID=%0
                                       ↪ d␣SA=%0d␣A=%0d␣B=%0d␣Cin=%0d␣|␣
                                       ↪ ref=%0d␣Sout=%0d",
                                    i, sa, A, B, Cin, ref_sum, Sout
                                );
                            end
                            else if ((ref_sum != Sout) && (err == 1)) tp
                               ↪ = tp + 1;
                            else if ((ref_sum == Sout) && (err == 1)) fp
                               ↪ = fp + 1;
                            else if ((ref_sum == Sout) && (err == 0)) tn
                               ↪ = tn + 1;
                        end
                    end
                end
            end

            $display("===␣Generator␣reszt␣mod3␣===");
            $display("TP=%0d", tp);
            $display("TN=%0d", tn);
            $display("FP=%0d", fp);
            $display("FN=%0d", fn);

            $display("===␣KONIEC␣SYMULACJI␣===");
            $finish;
        end

endmodule
```

Listing A.11: Testbench układu samosprawdzającego z sumatorem CLA

```
‘timescale 1ns/1ps

module tb_CLA;

    //----------------------
    // Parametry
    //----------------------
    localparam NG = 128;     // liczba mo?liwych GID
    // 8 mozliwych uszkodzen w RCA
    localparam GID_CLA_BASE      = 0;
    localparam GID_CLA_END       = 15;
    // 4 mozliwe uszkodzenia w adder mod3
    localparam GID_MOD3_BASE     = 20;
    localparam GID_MOD3_END      = 23;
    // 2 mozliwe uszkodzenia w komparatorze
    localparam GID_CMP_BASE      = 30;
    localparam GID_CMP_END       = 31;
    // 8 mozliwych uszkodzen w generatorze reszty mod 3
    localparam GID_RESIDUE_BASE  = 40;
    localparam GID_RESIDUE_END   = 47;
    //----------------------
    // Sygnaly testbench
    //----------------------
```

```
reg  [3:0] A, B;
reg  [1:0] A_mod3, B_mod3;
reg        Cin;

reg  [NG-1:0] fault_en_bus;
reg           fault_val;

reg [4:0] ref_sum;

wire [4:0] Sout;
wire       err;

//----------------------
// DUT
//----------------------

// 1) CLA
wire [3:0] cla_s;
wire       cla_cout;

CLA #(
    .NG(NG),
    .GID_BASE(GID_CLA_BASE)
) dut_cla (
    .X(A),
    .Y(B),
    .Cin(Cin),
    .fault_en_bus(fault_en_bus),
    .fault_val(fault_val),
    .S(cla_s),
    .Cout(cla_cout)
);

// po??czone wyj?cie CLA jako 5-bit
assign Sout = {cla_cout, cla_s};


// 2) RESZTA MOD3 z wyniku CLA
wire [1:0] Rsum_mod3;

residue_mod3 #(
    .NG(NG),
    .GID_BASE(GID_RESIDUE_BASE)
) dut_mod3_sum (
    .A(Sout),
    .R(Rsum_mod3),
    .fault_en_bus(fault_en_bus),
    .fault_val(fault_val)
);

// 3) SUMA RESZT (Amod3 + Bmod3)
wire [1:0] R_ab_mod3;
```

```
adder_mod3 #(
    .NG(NG),
    .GID_BASE(GID_MOD3_BASE)
) dut_mod3_add (
    .X(A_mod3),
    .Y(B_mod3),
    .fault_en_bus(fault_en_bus),
    .fault_val(fault_val),
    .R(R_ab_mod3)
);

// 4) POROWNANIE RESZT
mod3_comparator #(
    .NG(NG),
    .GID_OUT(GID_CMP_BASE)
) dut_cmp (
    .X(Rsum_mod3),
    .Y(R_ab_mod3),
    .fault_en_bus(fault_en_bus),
    .fault_val(fault_val),
    .eq_err(err)
);

//----------------------
// test wszystkich uszkodzen
//----------------------

integer i;
integer sa;   // stuck-at
integer a, b, ra, rb;

integer tp, fp, fn, tn;

initial begin
    $display("===␣START␣SYMULACJI␣===");

    Cin = 0;
    fault_en_bus = 0;
    fault_val    = 0;
    tp = 0; tn = 0; fp = 0; fn = 0;

     // przej?cie po wszystkich wej?ciach A,B
     for (a = 0; a < 16; a = a + 1) begin
         for (b = 0; b < 16; b = b + 1) begin

            A = a;
            B = b;
            ref_sum = A + B + Cin;

            A_mod3 = A % 3;
            B_mod3 = B % 3;

            #10; // ustalenie sygnalow
```

```
            if ((ref_sum != Sout) && (err == 0)) begin
                    fn = fn + 1;
                    $display(
                    "NIEWYKRYTY␣BLAD␣ARYTMETYCZNY:␣GID=%0d␣SA=%0d
                       ↪ ␣A=%0d␣B=%0d␣Cin=%0d␣|␣ref=%0d␣Sout=%0d
                       ↪ ",
                    i, sa, A, B, Cin, ref_sum, Sout
                    );
            end
            else if ((ref_sum != Sout) && (err == 1)) tp = tp +
               ↪ 1;
            else if ((ref_sum == Sout) && (err == 1)) fp = fp +
               ↪ 1;
            else if ((ref_sum == Sout) && (err == 0)) tn = tn +
               ↪ 1;
        end
    end

    $display("===␣CLA␣ref␣===");
    $display("TP=%0d", tp);
    $display("TN=%0d", tn);
    $display("FP=%0d", fp);
    $display("FN=%0d", fn);

    tp = 0; tn = 0; fp = 0; fn = 0;

    // Test: wszystkie uszkodzenia ukladu arytmetycznego
    for (i = GID_CLA_BASE; i <= GID_CLA_END; i = i + 1) begin

        // test stuck-at-0 i stuck-at-1
        for (sa = 0; sa <= 1; sa = sa + 1) begin

            fault_val = sa;

            // aktywuj jedno uszkodzenie
            fault_en_bus = 0;
            fault_en_bus[i] = 1;


            // przej?cie po wszystkich wej?ciach A,B
            for (a = 0; a < 16; a = a + 1) begin
                for (b = 0; b < 16; b = b + 1) begin

                    A = a;
                    B = b;
                    ref_sum = A + B + Cin;

                    A_mod3 = A % 3;
                    B_mod3 = B % 3;

                    #10; // ustalenie syganlow
```

```
                    if ((ref_sum != Sout) && (err == 0)) begin
                        fn = fn + 1;
                        $display(
                            "NIEWYKRYTY BLAD ARYTMETYCZNY: GID=%0
                               ↪ d SA=%0d A=%0d B=%0d Cin=%0d | 
                               ↪ ref=%0d Sout=%0d",
                            i, sa, A, B, Cin, ref_sum, Sout
                        );
                    end
                    else if ((ref_sum != Sout) && (err == 1)) tp
                       ↪ = tp + 1;
                    else if ((ref_sum == Sout) && (err == 1)) fp
                       ↪ = fp + 1;
                    else if ((ref_sum == Sout) && (err == 0)) tn
                       ↪ = tn + 1;
                end
            end
        end
    end

    $display("=== CLA ===");
    $display("TP=%0d", tp);
    $display("TN=%0d", tn);
    $display("FP=%0d", fp);
    $display("FN=%0d", fn);

    tp = 0; tn = 0; fp = 0; fn = 0;
    // Test: pozostale uszkodzenia w ukladzie
    for (i = GID_MOD3_BASE; i <= GID_MOD3_END; i = i + 1) begin

        // test stuck-at-0 i stuck-at-1
        for (sa = 0; sa <= 1; sa = sa + 1) begin

            fault_val = sa;

            // aktywuj jedno uszkodzenie
            fault_en_bus = 0;
            fault_en_bus[i] = 1;


            // przej?cie po wszystkich wejsciach A,B
            for (a = 0; a < 16; a = a + 1) begin
                for (b = 0; b < 16; b = b + 1) begin

                    A = a;
                    B = b;
                    ref_sum = A + B + Cin;

                    A_mod3 = A % 3;
                    B_mod3 = B % 3;

                    #10; // ustalenie sygnalow
```

```
                    if ((ref_sum != Sout) && (err == 0)) begin
                        fn = fn + 1;
                        $display(
                            "NIEWYKRYTY␣BLAD␣ARYTMETYCZNY:␣GID=%0
                               ↪ d␣SA=%0d␣A=%0d␣B=%0d␣Cin=%0d␣|␣
                               ↪ ref=%0d␣Sout=%0d",
                            i, sa, A, B, Cin, ref_sum, Sout
                        );
                    end
                    else if ((ref_sum != Sout) && (err == 1)) tp
                       ↪ = tp + 1;
                    else if ((ref_sum == Sout) && (err == 1)) fp
                       ↪ = fp + 1;
                    else if ((ref_sum == Sout) && (err == 0)) tn
                       ↪ = tn + 1;
                end
            end
        end
    end

    $display("===␣Mod3␣Adder␣===");
    $display("TP=%0d", tp);
    $display("TN=%0d", tn);
    $display("FP=%0d", fp);
    $display("FN=%0d", fn);

    tp = 0; tn = 0; fp = 0; fn = 0;
    for (i = GID_CMP_BASE; i <= GID_CMP_END; i = i + 1) begin

        // test stuck-at-0 i stuck-at-1
        for (sa = 0; sa <= 1; sa = sa + 1) begin

            fault_val = sa;

            // aktywuj jedno uszkodzenie
            fault_en_bus = 0;
            fault_en_bus[i] = 1;


            // przejscie po wszystkich wejsciach A,B
            for (a = 0; a < 16; a = a + 1) begin
                for (b = 0; b < 16; b = b + 1) begin

                    A = a;
                    B = b;
                    ref_sum = A + B + Cin;

                    A_mod3 = A % 3;
                    B_mod3 = B % 3;

                    #10; // ustalenie sygnalow

                    if ((ref_sum != Sout) && (err == 0)) begin
```

```
                        fn = fn + 1;
                        $display(
                            "NIEWYKRYTY␣BLAD␣ARYTMETYCZNY:␣GID=%0
                               ↪ d␣SA=%0d␣A=%0d␣B=%0d␣Cin=%0d␣|␣
                               ↪ ref=%0d␣Sout=%0d",
                            i, sa, A, B, Cin, ref_sum, Sout
                        );
                    end
                    else if ((ref_sum != Sout) && (err == 1)) tp
                       ↪ = tp + 1;
                    else if ((ref_sum == Sout) && (err == 1)) fp
                       ↪ = fp + 1;
                    else if ((ref_sum == Sout) && (err == 0)) tn
                       ↪ = tn + 1;
                end
            end
        end
    end

    $display("===␣Komparator␣===");
    $display("TP=%0d", tp);
    $display("TN=%0d", tn);
    $display("FP=%0d", fp);
    $display("FN=%0d", fn);

    tp = 0; tn = 0; fp = 0; fn = 0;
    for (i = GID_RESIDUE_BASE; i <= GID_RESIDUE_END; i = i + 1)
       ↪ begin

        // test stuck-at-0 i stuck-at-1
        for (sa = 0; sa <= 1; sa = sa + 1) begin

            fault_val = sa;

            // aktywuj jedno uszkodzenie
            fault_en_bus = 0;
            fault_en_bus[i] = 1;


            // przejscie po wszystkich wejsciach A,B
            for (a = 0; a < 16; a = a + 1) begin
                for (b = 0; b < 16; b = b + 1) begin

                    A = a;
                    B = b;
                    ref_sum = A + B + Cin;

                    A_mod3 = A % 3;
                    B_mod3 = B % 3;

                    #10; // ustalenie sygnalow

                    if ((ref_sum != Sout) && (err == 0)) begin
```

```
                        fn = fn + 1;
                        $display(
                            "NIEWYKRYTY␣BLAD␣ARYTMETYCZNY:␣GID=%0
                               ↪ d␣SA=%0d␣A=%0d␣B=%0d␣Cin=%0d␣|␣
                               ↪ ref=%0d␣Sout=%0d",
                            i, sa, A, B, Cin, ref_sum, Sout
                        );
                    end
                    else if ((ref_sum != Sout) && (err == 1)) tp
                       ↪ = tp + 1;
                    else if ((ref_sum == Sout) && (err == 1)) fp
                       ↪ = fp + 1;
                    else if ((ref_sum == Sout) && (err == 0)) tn
                       ↪ = tn + 1;
                end
            end
        end
    end

    $display("===␣Generator␣reszt␣mod3␣===");
    $display("TP=%0d", tp);
    $display("TN=%0d", tn);
    $display("FP=%0d", fp);
    $display("FN=%0d", fn);

    $display("===␣KONIEC␣SYMULACJI␣===");
    $finish;
  end

endmodule
```