Termin zajęć: **CZ/P/8.15** Wrocław, 13.06.2007

Grzegorz Pietrzak, 133329

Jacek Symonowicz, 133375

Ocena:

Oddano:

ALU zabezpieczona kodem korekcyjnym

projekt z przedmiotu „Systemy Tolerujące Uszkodzenia (FTC)”

Rok akad. 2006/2007, kierunek: INF

PROWADZĄCY:
dr inż. Piotr Patronik

Spis treści:

1. Wstęp

1.1. Rodzaje nadmiarowości

Projektowanie niezawodnych układów cyfrowych pociąga za sobą konieczność wykorzystania pewnej nadmiarowości, która objawiać się może w różnych aspektach, zależnie od zapotrzebowania na określone parametry niezawodnościowe układu. Wyróżnić można nadmiarowość sprzętową, czasową oraz informacyjną.

Jako prosty sposób na zwiększenie niezawodności można podać wprowadzenie nadmiaru sprzętowego, który obejmuje takie metody, jak dublowanie, dublowanie z inwersją oraz powielanie z głosowaniem. Polegają one na zwielokrotnieniu układu początkowego w tej samej formie lub odpowiednio zmodyfikowanej. Do wykrycia i ewentualnego poprawienia znalezionych błędów nieodzowny jest tu dodatkowy układ kontrolny.

Inną metodą jest rozbudowa sprzętu o dodatkowe moduły kontrolne z jak najmniejszym nakładem sprzętowym. Wykorzystywane jest to na przykład przy projektowaniu układów arytmetycznych odpornych na uszkodzenia. Wypracowano dla nich uniwersalne metody zabezpieczania, takie jak kody resztowe, bity parzystości oraz kody AN.

Do zmniejszenia ryzyka błędnego przesłania danych stosuje się oddzielną grupę zabezpieczeń. Należą do niej przede wszystkim kody korekcyjne (cykliczne kody CRC, kody Reeda-Salomona i inne), ale także detekcyjne (bity parzystości, sumy kontrolne itp.) [1].

1.2. Kody detekcyjne i korekcyjne

Do grupy kodów detekcyjnych zaliczamy te wszystkie kody, w których po wykryciu błędu nie jest jednoznacznie określone, na której pozycji (lub pozycjach) nastąpiło przekłamanie. Do tej klasy można zaliczyć zwykłe kody z bitem parzystości i inne bazujące na prostych działaniach arytmetycznych.

Kody korekcyjne, oprócz wykrywania pewnej liczby błędów, potrafią także pewną ich liczbę poprawić. Należą do nich na przykład kody Hamminga, kody BCH, splotowe, czy kod Turbo. O każdym z nich mówimy, że ma określone właściwości korekcyjne i detekcyjne, co wiąże się z ilością błędów przez niego poprawianych i wykrywanych.

Z niektórych kodów, które nie posiadają właściwości korekcyjnych, można tworzyć układy posiadające takie własności. Na przykład zwykły kod Hamminga tworzy się obliczając bity parzystości odpowiednich pozycji zakodowanej liczby. Podobnie można postąpić z kodowaniem resztowym. Dodając do zwykłego (niezabezpieczonego) układu arytmetycznego moduł generujący w odpowiedni sposób reszty z dzielenia argumentów przez odpowiednią liczbę i operujący na tych resztach w analogiczny sposób, jak na danych, otrzymuje się układ o zdolnościach detekcyjnych. Dodatkowo dołączenie drugiego po-

dobnego układu (lub więcej), dzielącego modulo przez inną liczbę skutkuje pojawieniem się właściwości korekcyjnych – układ może już korygować pojedynczy błąd. Tak powstałe kody noszą nazwę dwuresztowych (biresidue) [2].

1.3. Kody dwuresztowe

Załóżmy, że liczbę całkowitą N kodujemy jako wektor $(N, |N|_A, |N|_B)$, gdzie A i B są dwiema liczbami całkowitymi względnie pierwszymi. Liczby $|N|_A$, $|N|_B$ nazwiemy resztami z dzielenia N przez A i B. Definiujemy dodawanie dwóch słów kodowych $(N_1, |N_1|_A, |N_1|_B)$, $(N_2, |N_2|_A, |N_2|_B)$ jako skutkujące powstaniem słowa $(N_1+N_2, |N_1|_A+ +|N_2|_A, |N_1|_B+|N_2|_B)$, w którym sumowanie reszt odbywa się z zachowaniem działań względem odpowiedniego modułu. Składniki trójki kodowej nazwiemy odpowiednio: część akumulatorowa, korektor A i korektor B.

Wektor (X, Y, Z) jest kodem dwuresztowym względem modułów A i B wtedy i tylko wtedy, gdy $Y = |X|_A$ oraz $Z = |X|_B$.

Syndromem dla wektora (X, Y, Z) względem modułów A i B, zapisywanym jako $S(X, Y, Z)$ jest para (s_a, s_b), gdzie $s_a = |X - Y|_A$ oraz $s_b = |X - Z|_B$.

Z powyższych wynika, że wektor (X, Y, Z) jest słowem kodowym kodu dwuresztowego względem modułów A i B wtedy i tylko wtedy, gdy jego syndrom $S(X, Y, Z)$ względem tych samych modułów A i B jest równy $(0, 0)$;

Załóżmy teraz, że (X, Y, Z) jest słowem kodowym kodu dwuresztowego względem modułów A i B i że nastąpił błąd e w którejkolwiek z trzech części. Mamy teraz trzy możliwości:

- Przypadek I: błąd w części akumulatorowej: wektor z błędem będzie teraz zapisywany jako (X', Y, Z), gdzie $X' = X + e$. Syndrom $S(X', Y, Z) = = (|X' - Y|_A, |X' - Z|_B) = (|e|_A, |e|_B)$. Błąd przechodzi niezauważenie tylko w przypadku, kiedy $|e|_A = |e|_A = 0$.
- Przypadek II: błąd w części korektora A: wektor z błędem zapisywany jako (X, Y', Z), gdzie $Y' = Y + e$. Syndrom $S(X, Y', Z) = (|X - Y'|_A, |X - Z|_B) = (|-e|_A, 0)$.
- Przypadek III: błąd w części korektora B: wektor z błędem zapisywany jako (X, Y, Z'), gdzie $Z' = Z + e$. Syndrom $S(X, Y, Z') = (|X - Y|_A, |X - Z'|_B) = (0, |-e|_A)$.

Jeśli weźmiemy teraz pod uwagę taką klasę błędów e, że $|e|_A \neq 0$, błąd w dowolnej części będzie wykrywany i lokalizowany na podstawie powstałego syndromu. Jeśli błąd powstanie w jednym z korektorów, ten wynik będzie mógł być wygenerowany ponownie z części akumulatorowej. Z drugiej strony, jeśli błąd powstanie w części akumulatorowej, pozycja i poprawna wartość może być jednoznacznie wyznaczona z wartości syndromu (s_a, s_b). Tutaj należałoby założyć, że długość liczby wejściowej jest nie większa niż określona przez moduły A i B. *[13]*

W naszym przypadku długość liczby wejściowej będzie ograniczona przez dwa moduły: 7 (czyli 2^3-1) oraz 15 (czyli 2^4-1), będzie zatem ograniczona do $3 \cdot 4 = 12$ pozycji (bitów). Dla

każdej pozycji można obliczyć syndromy. Jest to bardzo łatwe, bo dla i-tej należy tylko obliczać resztę z dzielenia i-tej potęgi dwójki przez oba moduły. Drugi syndrom (ujemny) powstaje przez odjęcie pierwszego od wektora (7, 15). Wszystkie przedstawia poniższa tabela:

Tabela 1: Syndromy

i	**syndrom $+2^i$**	**syndrom -2^i**
0	(1, 1)	(6, 14)
1	(2, 2)	(5, 13)
2	(4, 4)	(3, 11)
3	(1, 8)	(6, 7)
4	(2, 1)	(5, 14)
5	(4, 2)	(3, 13)
6	(1, 4)	(6, 11)
7	(2, 8)	(5, 7)
8	(4, 1)	(3, 14)
9	(1, 2)	(6, 13)
10	(2, 4)	(5, 11)
11	(4, 8)	(3, 7)

Należy zauważyć, że w powyższej tabeli żadne z 24 syndromów nie powtarzają się. Można przypisać je bezpośrednio do wszystkich 24 możliwych do otrzymania pojedynczych błędów. Na przykład po otrzymaniu syndromu (2, 8) można z łatwością ustalić, że na pozycji 7 otrzymano '1', a powinno być '0'. Aby skorygować wynik, należy teraz odjąć od niego wartość $2^7 = 128$.

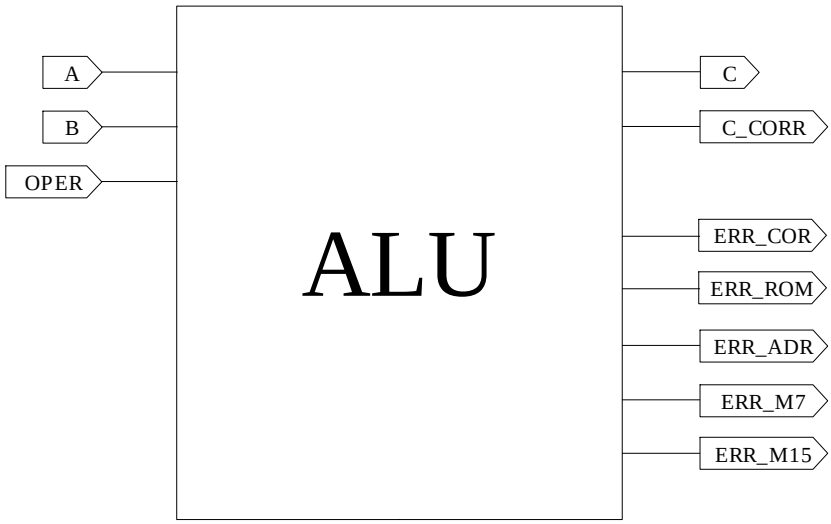
2. Układ arytmometru korygujący za pomocą kodu dwuresztowego

2.1 Wybór kodowania

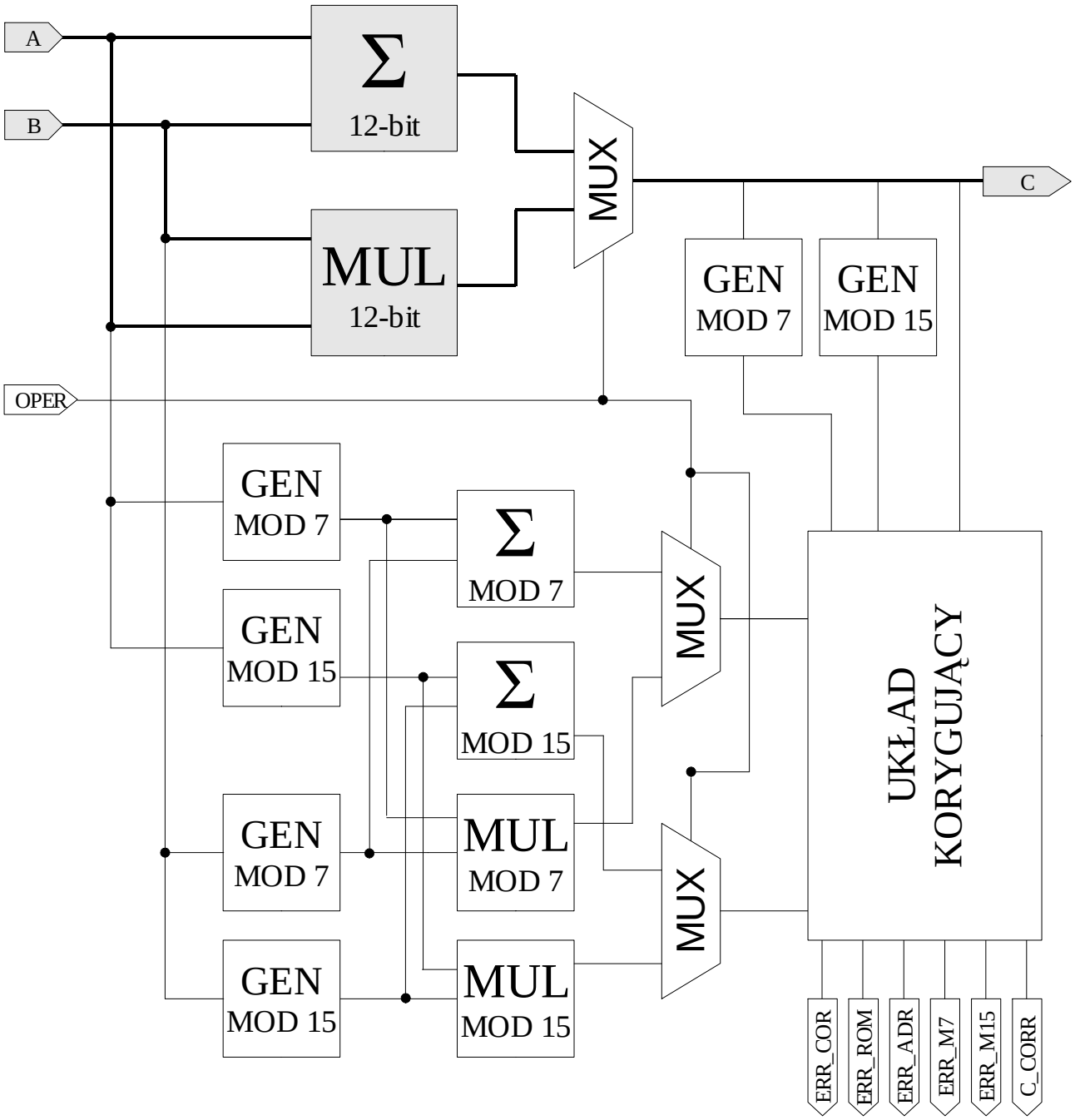
Przy wyborze kodowania dla ALU mającej samoczynnie korygować pojedyncze błędy wybrano początkowo kod Hamminga. Jest to kod cykliczny mogący korygować pojedynczy błąd. Skonstruowany został sumator dodający najpierw modulo 2 dwie liczby (przy pomocy bramek XOR), a następnie wykorzystujący właściwości liniowości kodu Hamminga do korygowania kolejnych pozycji w przypadku wystąpienia przeniesienia. W ten sposób wszystkie błędy nie powodujące przeniesienia powinny być korygowane, a pozostałe wy-

krywane (każdy pojedynczy błąd w układzie skutkowałby w wyjściu nie będącym słowem kodowym). Niestety przy rzeczywistej realizacji tych założeń wystąpił szereg problemów, głównie z generacją i kontrolowaniem przeniesień – w niektórych przypadkach uszkodzenie mogło przejść przez układ niezauważone.

Ostatecznie użyto podwójnych kodów resztowych (mod 7/mod 15), na poniższych schematach znajduje się układ arytmetyczny wykonujący operacje mnożenia, dodawania i odejmowania w kodzie uzupełnieniowym do 2.

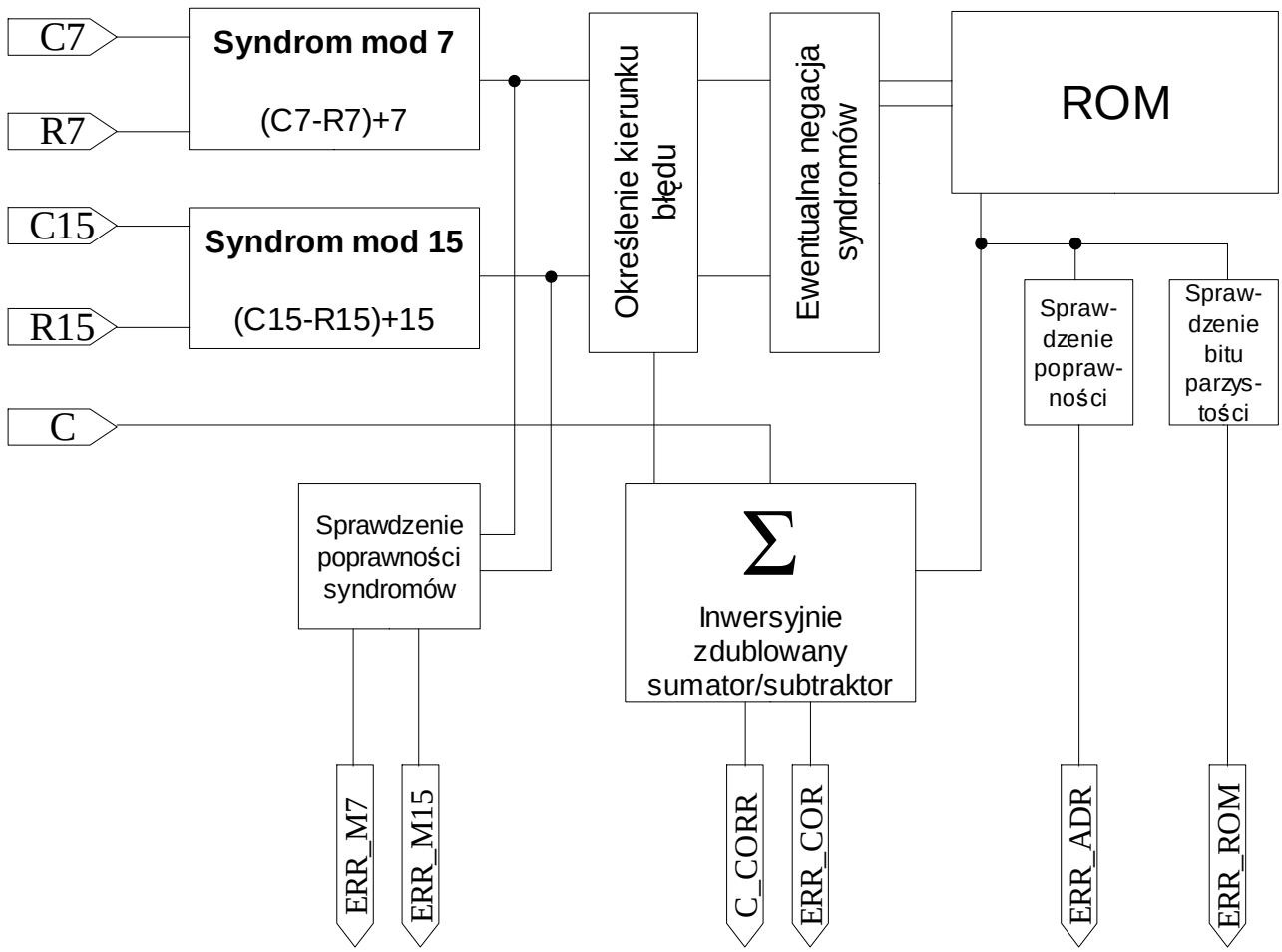


Rysunek 1: Schemat nadrzędny układu



Rysunek 2: Użyte moduły

Operacje wykonywane są na 12-bitowych argumentach *A* i *B*, bezpośrednim, nieskorygowanym wynikiem jest tutaj *C* (część schematu zaznaczona na szaro z pogrubionymi liniami oznacza tu praktycznie niezależną od układu korygującego jednostkę arytmetyczną). Dla każdego z argumentów generowane są reszty modulo 7 oraz 15 (*A7*, *B7*, *A15* i *B15*), na których przeprowadza się, równolegle do obliczeń na *A* i *B*, odpowiednią operację arytmetyczną modulo (w zależności od wejścia oper będzie to $R7 = A7 +_{\text{mod } 7} B7$ lub $R7 = A7 *_{\text{mod } 7} B7$ oraz $R15 = A15 +_{\text{mod } 15} B15$ lub $R15 = A15 *_{\text{mod } 15} B15$). Reszty generowane są następnie dla wyniku *C* ($C \bmod 7 = C7$ oraz $c \bmod 15 = C15$), w celu porównania z *R7* i *R15*. Na podstawie otrzymanych danych możliwa jest już korekcja, poniżej znajduje się schemat układu korekcyjnego.



Rysunek 3: Schemat układu korekcyjnego

Aby skorygować błąd trzeba najpierw wyliczyć syndromy *S7* = (*C7-R7*)+7 oraz *S15*= (*C15-R15*)+15. Można wyróżnić następujące sytuacje:

1. jeżeli oba syndromy są zerami operację arytmetyczną wykonano bezbłędnie,
2. jeżeli oba są niezerowe – nastąpił błąd w układzie arytmetycznym,
3. jeżeli jeden z syndromów jest zerem, a drugi jest niezerowy – wystąpił błąd w jednym z układów sprawdzających.

W pierwszym z przypadków dalsze operacje nie powinny doprowadzić do korekcji wyniku. W drugim przypadku należy ustalić którą pozycje wyniku wymagają korekcji. W zależności od wartości syndromów należy albo dodać, albo odjąć jedynkę na danej pozycji błędnej liczby.

Z właściwości kodów dwuresztowych wynika, że w przypadku korygowania poprzez odejmowanie każdy z syndromów składa się z jednej jedynki i reszty zer (np. *100, 010, 001* w przypadku *S7*). Przy dodawaniu syndromy składają się z jednego zera i reszty jedynek (odpowiednio *011, 101, 110*), przy czym zestawy syndromów zwykłych i zanegowanych dotyczą korekcji tych samych pozycji (np. $S7 = 1_{10} = 001_2$, $S15 = 4_{10} = 0100_2$ - błąd polegający

na dodaniu jedynki na pozycji 6, po zanegowaniu syndromów $S7 = 110_2$, $S15 = 1011_2$ oznacza błąd polegający na odjęciu jedynki na pozycji 6). Układ określający kierunek błędu zwraca operację, która do błędu doprowadziła (1 – odjęcie jedynki, 0 – dodanie jedynki), a następnie przekazuje wynik do negatora, gdzie opcjonalnie następuje inwersja bitów w obu resztach.

Na wyjściu negatora powinna się pojawić liczba złożona z 7 bitów z dwoma jedynkami i resztą zer. Stanowi ona adres dla pamięci ROM, w której przechowywane są ciągi korygujące (np. dla wspomnianych już syndromów *S7=1* i *S15=4* ciąg będzie równy *K=000001000000*) zabezpieczone bitem parzystości. Ciąg (zdublowany inwersyjnie) doprowadzany jest do zdublowanego inwersyjnie sumatora/subtraktora, gdzie następuje korekcja i zwrócenie poprawnego wyniku *C_CORR*. Jeżeli wszystkie sygnały błędów w układzie są równe zeru możemy bezpiecznie założyć, że wartość *C_CORR* jest poprawna. Jeżeli nastąpił błąd musimy zdać się na wyjście *C* omijające układ korekcyjny.

Reasumując, zdolności korekcyjne i detekcyjne są następujące:

- możliwość korekcji każdego pojedynczego błędu, który wystąpi w części arytmetycznej układu
- możliwość detekcji każdego pojedynczego błędu, który wystąpi w części korekcyjnej układu

2.2 Implementacja

Układ został zaimplementowany w języku Verilog, w środowisku Cver, przy użyciu niskopoziomowych instrukcji operujących na bramkach i wyrażeniach logicznych. Poniżej znajduje się opis poszczególnych modułów używanych w projekcie

1. Moduł *mux2x1*

- Nagłówek: **mux2x1(a1, a2, adr, wy)**,
- wejścia: *a1[WID-1:0]* – pierwsza linia wejściowa, *a2[WID-1:0]* – druga linia wejściowa, *adr* – adres do wyboru,
- wyjścia: *wy[WID-1:0]* – linia wyjściowa,
- parametry: *WID* – szerokość słowa.

Multiplekser ten jest wykorzystywany przy określaniu rodzaju operacji do wykonania. Mnożenie i dodawanie wykonywane są równolegle, wyniki doprowadza się na wejścia *a1* i *a2* multipleksera, gdzie następuje decyzja o wyborze jednego z nich na podstawie adresu *adr* (dodawanie/odejmowanie dla *adr*=0 lub mnożenie dla *adr*=1).

2. Moduł *gen_u2_6*

- Nagłówek: **gen_u2_6(a, sum)**,
- wejścia: *a[5:0]* – 6-bitowa liczba,
- wyjścia: *sum[5:0]* – wynik działania.

Moduł ten neguje liczbę 6-bitową na wejściu *a*, a następnie dodaje na najmłodszej pozycji jedynkę. Jest to potrzebne przy konwersji do systemu uzupełnieniowego do dwóch (U2) ostatniego cząstkowego iloczynu w multiplikatorze *mul12*.

3. Moduł *mul12*

- Nagłówek: **mul12(a, b, w)**,
- wejścia: *a[5:0]* – 6-bitowa mnożna, *b[5:0]* – 6-bitowy mnożnik,
- wyjścia: *w[11:0]* – 12-bitowy wynik działania.

Moduł *mul12* jest standardowym multiplikatorem mnożącym dwie 6-bitowe liczby *a* i *b* zapisane w systemie U2, zwracającym 12-bitowy wynik *w*. Poszczególne iloczyny cząstkowe są dopełniane zgodnie ze znakiem mnożnej, przy ostatnim iloczynie cząstkowym należy zanegować mnożną i dodać na jej najmłodszym bicie jedynkę (moduł *gen_u2_6*).

4. Moduły *mul6, mul8*

- Nagłówek: **mul6(a, b, w), mul8(a, b, w)**,
- wejścia: *a[2/3:0]* – 3 lub 4-bitowa mnożna, *b[2/3:0]* – 3 lub 4-bitowy mnożnik,
- wyjścia: *sum[5/7:0]* – 6 lub 8-bitowy wynik działania.

Moduły te mają za zadanie obliczenie iloczynu z dwóch liczb 3 lub 4-bitowych zapisanych w NKB. Brakuje tu części odpowiedzialnej za liczenie w systemie U2. *Mul6* i *mul8* używane są do operacji na resztach, które są zawsze nieujemne.

5. Moduł *adder12*

- Nagłówek: **adder(a, b, sum)**,
- wejścia: *a[11:0]* – 12-bitowy składnik, *b[11:0]* – 12-bitowy składnik,
- wyjścia: *w[11:0]* – 12-bitowa suma.

12-bitowy sumator nie uwzględniający przeniesienia wejściowego i wyjściowego. Z racji braku obsługi przepełnień należy tu uważać na zbyt duże argumenty *a* i *b*, jeżeli w wyniku operacji arytmetycznej wystąpi przepełnienie wynik *w* będzie niepoprawny, ale wynik działania na resztach pozostanie prawidłowy.

6. Moduły *genmod7*, *genmod15*

- Nagłówek: **genmod7(we, wy), genmod15(we, wy)**,
- wejścia: *we[11:0]* – 12-bitowy argument do skonwertowania,
- wyjścia: *wy[2/3:0]* – reszta z argumentu modulo 7/15.

Generatory modulo wykorzystują sumatory z przechowywaniem przeniesień (CSA - carry-save adder). Końcowy moduł generatora realizuje się zazwyczaj jako sumator ze sprzężeniem zwrotnym (EAC – end-around carry), niestety ograniczenia nałożone przez Cver wymusiły zastosowanie zastępczego rozwiązania, czyli dwóch poziomów sumatorów RCA (moduł *summod7* / *summod15*). Reszta z dzielenia jest tu zawsze dodatnia, co ułatwia dalsze operacje wykonywane w ALU.

7. Moduły *summod7*, *summod15*

- Nagłówek: **summod7(a,b,s), summod15(a,b,s)**,
- wejścia: *a[2/3:0]* – 2- lub 3-bitowy składnik, *b[2/3:0]* – 2- lub 3-bitowy składnik,
- wyjścia: *wy[2/3:0]* – suma modulo 7/15.

Moduł ten realizuje sumę modulo 7 lub 15. Jak już zostało wspomniane w opisie modułów *genmod* powinien zostać tu użyty sumator ze sprzężeniem zwrotnym, czego nie udało się zrealizować ze względu na ograniczenia symulatora. Wersja EAC po otrzymaniu składników sumy stabilizuje się dość długo, ale pozwala na znacznie mniejszy nadmiar sprzętu, niż w przypadku dwupoziomowego RCA.

8. Moduły *kon_u2_uj7*, *kon_u2_uj15*

- Nagłówek: **kon_u2_uj7(we, wy), kon_u2_uj15(we, wy)**,
- wejścia: *we[2/3:0]* – reszta do konwersji,
- wyjścia: *wy[3/4:0]* – wynik działania.

Są to moduły konwertujące 2- lub 3-bitową resztę do postaci ujemnej poprzez negację wszystkich bitów i dodanie jedynki na najmłodszej pozycji. Wynik jest o jeden bit szerszy niż wejście wskutek doklejenia na początku bitu znaku równego 1.

9. Moduły *sum[4/5/6/8]bit*

- Nagłówek: **sum[4/5/6/8]bit(a, b, ci, sum)**,
- wejścia: *a[x:0]* – pierwszy składnik (x = 4/5/6/8), *b[x:0]* – drugi składnik (x = 4/5/6/8), *ci* – przeniesienie na wejściu,
- wyjścia: *sum[y:0]* – suma (y = 4/5/6/8).

Zestaw zwykłych sumatorów używanych przy wyliczaniu syndromów oraz dodawaniu iloczynów cząstkowych w multiplikatorach. Sumatory te nie uwzględniają na wyjściu przeniesienia *cout*.

10. Moduł *rom*

- Nagłówek: **rom(adr, out, nout)**,
- wejścia: *adr[6:0]* – adres,
- wyjścia: *out[12:0]* – dana spod adresu *adr* (12 bit + parity), *nout[12:0]* – dana komplementarna do *out*, ale z wyliczonym od nowa bitem parzystości.

Moduł ten jest pamięcią o rozmiarze 128x13 (128 13-bitowych słów), zawierającą ciągi korekcyjne. Każde słowo w pamięci jest zabezpieczone bitem parzystości, wyjścia układu są komplementarne (nie powoduje to dużego nadmiaru, gdyż i tak potrzebujemy negacji *out* w zdublowanym inwersyjnie sumatorze). Jeżeli adres jest z nieobsługiwanego zakresu, to zostaje zwrócony błąd w postaci słowa 1111111111100 (wartość bezpieczna – dozwolone są słowa, w których znajduje się oprócz bitu parzystości tylko jedna jedynka). Ze względów praktycznych (1664 przechowywane bity) moduł jako jedyny w całym układzie został zrealizowany poprzez instrukcję warunkową *case*, co pozwoliło na znaczną redukcję długości kodu.

11. Moduł *parity*

- Nagłówek: **parity(we, p)**,
- wejścia: *we[11:0]* – argument wejściowy,
- wyjścia: *p* – bit parzystości.

Moduł *parity* oblicza bit dopełnienia do parzystości dla 11-bitowej liczby. Używane są tu kaskadowo ułożone bramki XOR.

12. Moduł direction

- Nagłówek: **direction(s7, s15, odejm, oper, adr)**,
- wejścia: *s7[2:0]* – syndrom mod 7, *s15[3:0]* – syndrom mod 15,
- wyjścia: *odejm* – czy kierunkiem jest odejmowanie, *oper* – negacja odejm, *adr[6:0]* – adres do pamięci ROM.

Odbywa się tutaj sprawdzenie rodzaju błędu, jakim obciążony jest wynik działania arytmetycznego. Jeżeli w jednym z syndromów znajdują się więcej niż 2 jedynki, wówczas musimy oba syndromy zanegować przed podaniem ich jako adres do pamięci ROM.

13. Moduł fulladder_ftc

- Nagłówek: **fulladder_ftc(a, b, ci, na, nb, nci, co, nco, s, ns),**
- wejścia: *a* – pierwszy składnik sumy, *na* – negacja *a*, *b* – drugi składnik sumy, *nb* – negacja *b*, *c* – przeniesienie na wejściu, *nc* – negacja *c*,
- wyjścia: *co* – przeniesienie na wyjściu, *nco* – negacja *co*, *s* – suma, *ns* – negacja *s*.

Sumator ten pozwala na wykrycie pojedynczego błędu w swojej strukturze, jego własności detekcyjne przydadzą się w ustaleniu, czy układ korekcyjny działa poprawnie.

14. Moduł bugfind

- Nagłówek: **bugfind(m7, m15, err7, err15)**,
- wejścia: *m7* – syndrom mod 7, *m15* – syndrom mod 15,
- wyjścia: *err7* – błąd w układzie sprawdzającym mod 7, *err15* - błąd w układzie sprawdzającym mod 15.

Moduł bugfind służy do wykrycia sytuacji, kiedy to $S7 = 0$ i $S15 \neq 0$ lub $S7 \neq 0$ i $S15 = 0$. Oznacza to sytuację, kiedy jeden z układów operujących na resztach jest uszkodzony.

15. Moduł corrector

- Nagłówek: **corrector(wynik, kor, nkor, oper, noper, err, out)**,
- wejścia: *wynik[11:0]* – wynik operacji arytmetycznej do korekcji, *kor[11:0]* – ciąg korekcyjny, *nkor[11:0]* – komplementarny ciąg korekcyjny, *oper* – rodzaj operacji do wykonania (1 – odejmowanie), *noper* – negacja *oper*,
- wyjścia: *err* – wystąpił błąd w układzie korekcyjnym, *out* – skorygowany wynik.

Moduł *corrector* koryguje wynik operacji arytmetycznej na podstawie doprowadzonego ciągu korekcyjnego, sumowanie odbywa się w 12-bitowym inwersyjnie zdublowanym sumatorze. Jeżeli w sumatorze zostanie wykryty błąd, wyjście *err* przyjmie wartość 1.

16. Moduł alu

- Nagłówek: **alu(a, b, oper, c, syn7, syn15, corr, ERR_M7, ERR_M15, ERR_ROM, ERR_COR, ERR_ADR)**,
- wejścia: *a[11:0]* – argument 1, *b[11:0]* – argument 2, *oper* – rodzaj operacji (0 – dodawanie/odejmowanie, 1 - mnożenie),
- wyjścia: *c[11:0]* – wynik bez korekcji, *syn7[2:0]* – syndrom mod 7, *syn15[3:0]* – syndrom mod 15, *corr[11:0]* – wynik po korekcji, *ERR_M7* – błąd przy operacjach mod 7, *ERR_M15* – błąd przy operacjach mod 15, *ERR_ROM* – błąd pamięci ROM, *ERR_COR* – błąd w korektorze, *ERR_ADR* – błąd w adresie pamięci ROM.

ALU jest modułem nadrzędnym projektu, integruje pozostałe moduły według przedstawionych wcześniej schematów. Argumenty wejściowe w przypadku dodawania nie mogą spowodować nadmiaru, w przypadku mnożenia – nie mogą przekraczać 6 bitów.

Jeżeli na wyjściu ALU wszystkie błędy *ERR_* są równe zeru, wówczas mamy pewność, że część korekcyjna działa poprawnie. Jeżeli którekolwiek z wyjść błędów uaktywnia się, nie można ufać otrzymanemu wynikowi *corr*, zdając się tym samym na nieskorygowany wynik *c*. Należy zaznaczyć, że jedne błędy mogą powodować drugie, stąd np. błąd w korektorze może mieć za pośrednią przyczynę uszkodzoną pamięć ROM.

3. Program testujący

Jak opisano wcześniej, układ powinien być odporny na wszystkie błędy pojedyncze, przy czym błędy w części arytmetycznej są korygowane, a w części korekcyjnej tylko wykrywane. Należy przyjąć, że każda bramka logiczna wchodząca w skład układu może ulec uszkodzeniu. Ponadto, błędy mogą być różnego typu. Najpopularniejsze są sklejenia z logiczną jedynką oraz sklejenia z logicznym zerem. Są to zresztą jedyne zdatne do predykcji uszkodzenia bez znajomości topografii układu. Rozpatrując zagadnienie w kategoriach bramek logicznych należy zaznaczyć, że należy osobno traktować uszkodzenia wejść bramki oraz jej wyjścia. Z tego powodu liczba uszkodzeń układu ALU, który jest rozpatrywany, jest na tyle duża, iż niemożliwe jest przetestowanie układu dla wszystkich kombinacji błędów i stanów wejść. To stwierdzenie odnosi się również do błędów pojedynczych.

Dlatego też zdecydowano o przeprowadzeniu systematycznych testów mających sprawdzić odporność układu na wszystkie błędy pojedyncze występujące w części arytmetycznej oraz na losowo rozmieszczone błędy w części korekcyjnej. Do stworzenia systemu symulacji uszkodzeń i automatycznych testów został napisany program w języku C++ w środowisku Linux, pod kompilator gcc.

Głównym założeniem programu jest uruchamianie jako procesu potomnego symulatora Cver oraz przetwarzanie zwróconych przez niego wyników symulacji. W ten sposób, odczytywane są informacje niezbędne do stwierdzenia, jakie były dane wejściowe i wyjściowe w symulowanym układzie, a w konsekwencji przeprowadzana jest kontrola poprawności wykonywanych działań przy wprowadzonych uszkodzeniach. Jako plik wejściowy Cver otrzymuje przy każdym uruchomieniu inną wersję układu, różniącą się od poprzedniej jedynie miejscem wprowadzenia uszkodzenia. Miejsce uszkodzeń zostały zdefiniowane przed uruchomieniem symulacji. Podczas działania aplikacji testującej zbierane są statystyki - zarówno dla pojedynczego przebiegu testu, jak i dla całego zdefiniowanego zestawu testów.

3.1. Implementacja programu testującego

Działanie programu najłatwiej opisać śledząc kolejne kroki przeprowadzania testu. Każdy test opisany jest w funkcji *main()* za pomocą struktury *testbench*.

```
struct testbench
{
        char nazwa[100];
        char plik_out[50];
        char plik_pocz[50];
        int dol, gora, typ;
};
```

Listing 1: Struktura definiująca test

Przykładowe użycie demonstruje poniższy fragment kodu wyjęty z funkcji *main()*:

```
struct testbench testy[] = {
        {"Test s-z-0 - dodawanie", "./test0a.v", "./testowy_zero_dodaj.v", -10, 10, 0},
        {"Test s-z-0 - mnożenie", "./test0b.v", "./testowy_zero_mnoz.v", -10, 10, 1},
        {"Test s-z-1 - dodawanie", "./test1a.v", "./testowy_one_dodaj.v", -10, 10, 0},
        {"Test s-z-1 - mnożenie", "./test1b.v", "./testowy_one_mnoz.v", -10, 10, 1}
};
```

Listing 2: Definiowanie testu

W powyższym przykładzie zdefiniowano cztery testy. Zawartość kodu veriloga pobierana jest z plików *testowy_***_****.v*, zaś po wykonaniu zmian w układzie zapisywana jest w plikach *test**.v*. Każde przejście testu będzie dokonywane przy wszystkich kombinacjach wprowadzanych na wejście układu liczb z przedziału <-10, 10>. Jeśli ostatnim parametrem jest 0, wykonywane będzie dodawanie, jeśli zaś 1 - mnożenie.

Po uruchomieniu programu wykonywane są pętle odpowiedzialne za generowanie kolejnych przebiegów. Zewnętrzna pętla for ma za zadanie przejść przez wszystkie definiowane wcześniej testy. Wewnątrz niej znajduje się pętla while, w której wprowadzane są kolejne uszkodzenia układu. Pomijając wypisywanie komunikatów oraz zerowanie statystyk, zawartość jej przedstawia się następująco:

```
wprowadz_zmiane(testy[i].plik_pocz, "./tmp.v");
unlink(testy[i].plik_pocz);
rename("./tmp.v", testy[i].plik_pocz);
make_test(testy[i].plik_out, testy[i].plik_pocz, testy[i].dol, testy[i].gora, testy[i].typ);
run_test(testy[i].plik_out);
```

Listing 3: Zasadnicza część funkcji *main()*

Wykonywane jest tu kilka kluczowych funkcji. Najpierw wprowadzana jest zmiana do pliku źródłowego. W tym celu korzysta się z tymczasowego pliku *tmp.v*, do którego przepisywany jest kod veriloga z dodanym kolejnym uszkodzeniem i jednocześnie usuniętym poprzednim. Ten plik staje się teraz plikiem źródłowym. Następnie tworzony jest kolejny przebieg, a wynikowy kod zapisywany jest do pliku podanego przy definicji testu. Utworzony kod jest interpretowany przez uruchomienie programu Cver w funkcji *run_test()*.

Działanie funkcji *wprowadz_zmiane()* wymaga wyjaśnień odnośnie struktury pliku źródłowego, do którego wprowadzane są zmiany. Otóż jest to zwyczajny kod veriloga opatrzony w pewnych miejscach specjalnymi komentarzami. Wykorzystywany jest tu fakt, iż Cver uznaje wystąpienie podwójnego znaku '/' za komentarz i nie interpretuje tekstu aż do końca linii, w której się znajduje. W niektórych miejscach wykorzystywany jest także drugi rodzaj komentarza respektowanego przez Cver: "/* */". Każdy taki specjalny komentarz ma określoną strukturę:

```
// $ciąg1$ciąg2$ ^^
```

gdzie *ciąg1* i *ciąg2* są łańcuchami znaków odpowiednio: zastępowanym i zastępującym. Przykładowa linia kodu tak zakomentowana wygląda następująco:

```
assign s0[11] = zn[0];    // $zn[0]$one$ ^^
```

Ważne jest, aby komentarz kończył się dwoma znakami '^'. Po przeczytaniu tej linii przez funkcję *wprowadz_zmiane()* każde wystąpienie przed komentarzem ciągu *"zn[0]"* zostanie zamienione na *"one"*, co w tym wypadku oznacza sklejenie linii *s0* z logiczną jedynką. Sama modyfikacja ma miejsce w funkcji *zamiana()*. Aby zaznaczyć, że linia została zmieniona, ostatni znak '^' zastępuje się znakiem '-'. teraz więc przykładowa linia kodu wygląda tak:

```
assign s0[11] = one;      // $one$zn[0]$ ^-
```

Jak widać, *ciąg1* i *ciąg2* zamienione zostały miejscami. Umożliwi to zmianę powrotną w kolejnym przebiegu. Gdy wtedy napotkana zostanie linia z komentarzem zakończonym przez "^-", będzie zmieniona z powrotem na postać wyjściową, a komentarz zostanie usunięty.

```
assign s0[11] = zn[0];
```

Kiedy funkcja *wprowadz_zmiane()* napotka w pliku komentarz pierwszego typu (zakończony na "^^"), kolejne linie nie są już zmieniane, natomiast komentarz drugiego typu ("^-") nie powoduje zatrzymania wprowadzania zmian. Pozwala to na ciągłe wprowadzanie uszkodzeń pojedynczych w kolejnych miejscach w kodzie.

Funkcja *make_test()* jest odpowiedzialna za złożenie poszczególnych składowych do jednego pliku, który będzie podawany do symulacji programowi Cver. Wykorzystywane wewnątrz niej funkcje *utworz()*, *dopisz_plik()*, *dopisz_tekst()* oraz *zamknij()* operują na globalnym deskryptorze pliku, do którego zapisywane są kolejne części kodu. Najpierw przepisywana jest zawartość pliku określonego w parametrze *zrodlo*, czyli wynik działania funkcji *wprowadz_zmiane()*. Każdy plik źródłowy kończy się następująco:

```
module alu_tb;

reg [11:0] a;
reg [11:0] b;
reg oper;
wire [11:0] sum, corr;
wire [2:0] syn7;
wire [3:0] syn15;
wire ERR_M7, ERR_M15, ERR_ROM, ERR_COR, ERR_ADR;

initial begin

$monitor ("%b %b %b %b %b %d %d %d %d %d %d %d", oper,a,b, sum, corr, syn7, syn15, ERR_M7, ERR_M15, ERR_ROM,
ERR_COR, ERR_ADR);
```

Listing 4: Końcowy fragment pliku źródłowego

Jest to inicjalny fragment modułu *alu_tb*, w którym to module na wejścia układu ALU wprowadzane są kolejne liczby. Poprzez komendę *$monitor* ustalony zostaje konkretny format danych wyjściowych, które przechwytywał będzie program testujący.

Następnie przypisywane są początkowe stany wejść układu: typ działania (dodawanie/mnożenie) i wyzerowane wejścia *a* i *b*. Teraz w podwójnej pętli for zostają zapisane wszystkie kombinacje wejść *a* i *b* z podanego zakresu (*dol*, *gora*). Przykładowy wiersz wygenerowany przez ten fragment kodu to:

```
#100 a = 'b000000010101; b = 'b111111110111;
```

Na koniec dopisywana jest zawartość pliku *koniec.v:*

```
#10 $finish;
end

alu a1(a, b, oper, sum, syn7, syn15, corr, ERR_M7, ERR_M15, ERR_ROM, ERR_COR, ERR_ADR);

endmodule
```

Listing 5: Zawartość pliku koniec.v

i gotowy plik testowy jest zamykany.

Funkcja *run_test()* wykonuje czynności niezbędne do uruchomienia symulacji układu zdefiniowanego w pliku, którego nazwa jest jej parametrem.

Na początku tworzone jest łącze nienazwane (funkcja *pipe()*), po czym poprzez funkcję *fork()* tworzony jest proces potomny, którego standardowe wyjście zostaje przekierowane do tegoż łącza. Teraz funkcja *execlp()* powoduje uruchomienie programu Cver z parametrami "*-q*" oraz nazwą pliku źródłowego. Proces macierzysty zamyka swój koniec łącza służący do zapisu i odczytuje kolejne znaki od procesu potomnego. Po napotkaniu znaku przejścia do nowej linii cały wiersz (jeśli zaczyna się od cyfry 0 lub 1) zostaje zinterpretowany funkcją s*prawdz()*, która odczyta parametry wejściowe i wyjściowe aby stwierdzić, czy wyniki są poprawne i uzupełnić statystyki.

3.2. Kompilacja i uruchomienie programu testującego

W celu skompilowania programu, należy wydać polecenie:

```
gcc -o test test.c
```

Skompilowany program można uruchomić z opcjonalnym parametrem *-v* oraz ewentualnie określić jego wartość. Parametr ten oznacza tryb "verbose" (czyli "gadatliwy"), w którym będzie wypisywane więcej informacji odnośnie przeprowadzanych właśnie testów. Gdy parametr ten zostanie podany, jego wartość będzie ustawiona na 1, chyba że zostanie podana także konkretna wartość:

- 1: wypisywanie zmian w kodzie źródłowym (wprowadzane uszkodzenia), status zwrócony przez program Cver, informacje o każdym przebiegu testu, łącznie ze statystykami,
- 2: wypisywanie wszystkich działań błędnie przeprowadzonych przez część arytmetyczną, nie tylko tych nieskorygowanych lub źle skorygowanych.

Niezależnie od parametrów, przy każdym uruchomieniu program wypisuje na wyjściu statystyki ogólne dla każdego zdefiniowanego zestawu testów oraz wszystkie nieskorygowane lub źle skorygowane błędy w układzie. W celu późniejszej analizy, standardowe wyjście programu może być przekierowane do pliku. Uruchomienie testu może więc wyglądać następująco:

```
./test -v 2 >wyniki.log
```

4. Testowanie

Procedura testowa trwała łącznie cztery doby, objęła ona wszystkie możliwe uszkodzenia części arytmetycznej oraz niektóre uszkodzenia części korekcyjnej układu. Obrane zakresy, ze względu na długi czas przebiegu testów, nie pokrywały wszystkich możliwych kombinacji dwóch liczb, jednakże były dość reprezentatywne aby oszacować odporność na błędy.

Przeprowadzone testy obejmowały cztery rodzaje uszkodzeń: sklejenie z zerem lub jedynką w sumatorze 12-bit, oraz sklejenie z zerem lub jedynką w multiplikatorze mnożącym 6-bitowe argumenty. Pozostałe moduły układu stanowią część kontrolną, w której możliwa jest detekcja błędów bez ich korekcji.

4.1. Implementacja zestawu testów

Ze względu na niewielki rozmiar argumentów (6 bit, czyli przedział od -32 do 31) w przypadku mnożenia poprawność multiplikatora została sprawdzona poprzez przegląd zupeł

ny wszystkich kombinacji danych wejściowych oraz uszkodzeń. W procedurze testowej wykonano niemal 9 mln operacji, wyniki znajdują się w poniższej tabeli.

Tabela 2: Wyniki testów sklejenia z zerem

	sklejenie z zerem	**sklejenie z jedynką**
Wszystkich	4 473 924	4 473 924
Bezbłędnych	3 747 029 **(84 %)**	2 551 368 **(57 %)**
Skorygowanych	726 191 **(16 %)**	1 919 381 **(43 %)**
Niekorygowalnych	704 **(0,01 %)**	3175 **(0,07 %)**
Niezerowych syndromów mod 7	726 671	1 921 628
Niezerowych syndromów mod 15	726 735	1 922 140
Błędy na linii ERR_M7	160	416
Błędy na linii ERR_M15	224	928
Błędy na linii ERR_ROM	704	3175
Błędy na linii ERR_COR	0	0
Błędy na linii ERR_ADR	704	3175

W przypadku obydwu typów sklejeń poprawione zostały niemal wszystkie wyniki mnożenia. Błędy niekorygowalne zostały wykryte na wejściu pamięci ROM, gdzie podany został nieprawidłowy adres. Przykładowo po wprowadzeniu uszkodzenia polegającego na sklejeniu z jedynką wejścia *y* bramki AND sumatora o oznaczeniu *fulladder76* zwrócone zostały 224 niekorygowalne błędy. Po krótkiej analizie jednego z nich łatwo wskazać źródło problemów.

4 * -3 = -1548 --####--> -1546 (-12) x

Mnożenie liczby 4 razy -3 dało tu wynik -1548 zamiast -12. Układ korekcyjny dodał do wyniku 2, co okazało się błędną decyzją.

```
   4      =    0000 0000 0100U2
* -3       =    1111 1111 1101U2
-----------------------------------------
-1548      =    1001 1111 0100U2     ← taki wynik otrzymano
-1546      =    1001 1111 0110U2     ← korekcja +2^i
-12        =    1111 1111 0100U2     ← taki powinien być wynik
```

Rysunek 4: Analiza przykładowego błędnego wyniku mnożenia, występuje tu specyficzny

rodzaj błędu podwójnego, niemożliwy do skorygowania przez kod dwuresztowy
Poprawny wynik różni się od otrzymanego na dwóch bitach, przy czym nie jest możliwa jego korekcja poprzez dodanie lub odjęcie wartości 2^i. Wynika stąd, że błąd nie może zostać skorygowany, możliwe jest tylko jego wykrycie poprzez rozpoznanie niewłaściwego zestawu syndromów (sygnalizowane wyjściem *ERR_ADR*).

Po przeanalizowaniu miejsc, w których wystąpiły niekorygowalne błędy okazało się, że szczególnie wrażliwy na uszkodzenia jest konwerter liczby sześciobitowej do formatu U2, potrzebny przy obliczaniu ostatniego iloczynu cząstkowego w multiplikatorze. Uszkodzenia konwertera powodowały 100% niekorygowalnych błędów w przypadku testów s-z-0 oraz oraz 98,8% w przypadku s-z-1 (pozostałe 1,2% wystąpiło w sumatorach 12-bitowych, w częściach odpowiedzialnych za bit 9 i 10 oraz przy obliczaniu 10 bitu iloczynów cząstkowych).

4.2. Dodawanie

Testowanie sklejeń w przypadku dodawania odbyło się na mocno ograniczonym zestawie działań, głównie ze względu na duży zakres argumentów (są 12-bitowe, aby nie dopuścić do przepełnienia należy dodawać wartości maksymalnie 11-bitowe, co w przeglądzie zupełnym daje 4 miliony kombinacji na jeden przebieg procedury testującej).

Przedziały dla testów sklejenia z 0:

- od -1024 do -768,
- od -768 do -512,
- od -512 do -256,
- od -256 do 0,
- od -128 do 128,
- od 896 do 1023.

Przedziały dla testów sklejenia z 1:

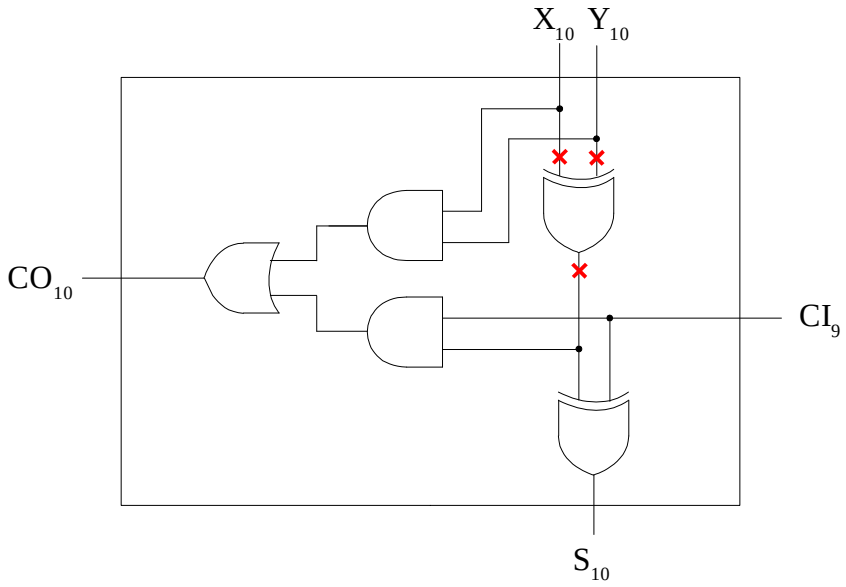
- od -64 do 63,
- od 896 do 1023,
- od -1024 do -897,
- od -600 do -500,
- od 500 do 600,
- od 600 do 700,
- od 200 do 300.

Tabela 3: Wyniki testów sklejenia z jedynką

	sklejenie z zerem	**sklejenie z jedynką**
Wszystkich	46 629 703	15 564 330
Bezbłędnych	30 631 178 **(66 %)**	9 202 743 **(59 %)**
Skorygowanych	15 998 525 **(34 %)**	6 252 129 **(40 %)**
Niekorygowalnych	0	109 458 **(1 %)**
Niezerowych syndromów mod 7	15 998 525	6 361 587
Niezerowych syndromów mod 15	15 998 525	6 361 587
Błędy na linii ERR_M7	0	0
Błędy na linii ERR_M15	0	0
Błędy na linii ERR_ROM	0	109 458
Błędy na linii ERR_COR	0	0
Błędy na linii ERR_ADR	0	109 458

Testy sklejenia z zerem nie wykazały nieprawidłowego działania układu, wszystkie uszkodzenia zostały skorygowane. Bardzo duża liczba poprawnie przeprowadzonych operacji dodawania (szczególnie dla zakresów <-256; 0> oraz <896; 1023>) daje wysokie prawdopodobieństwo, że sumator 12-bitowy jest chroniony przed wszystkimi pojedynczymi sklejeniami z zerem.

Testy sklejenia z jedynką wypadły już słabiej, dodatkowo ze względów czasowych musieliśmy znacznie ograniczyć zakres argumentów. Wystąpiło ponad 100 000 niekorygowalnych błędów, będących skutkiem uszkodzenia bramki XOR w sumatorze operującym na 10. pozycji 12-bitowej liczby.



Rysunek 5: Miejsce w układzie, którego uszkodzenie powoduje niekorygowalne błędy na wyjściu; na czerwono zaznaczone zostały punkty sklejenia z jedynką

Dokonana analiza propagacji błędów wykazała, że nie jest możliwa korekcja wyniku, w którym na skutek uszkodzenia pojawiła się wartość 1 na najstarszej pozycji. Jest to prawdopodobnie związane z dość specyficzną budową modułu obliczającego reszty mod 7 i mod 15, w którym została uwzględniona możliwość operowania na liczbach zapisanych w formacie U2.

512	=	$0010\ 0000\ 0000_{U2}$	
+ 520	=	$0010\ 0000\ 1000_{U2}$	
-2040	=	$1000\ 0000\ 1000_{U2}$	← taki wynik otrzymano
-2038	=	$1000\ 0000\ 10\underline{1}0_{U2}$	← korekcja $+2^i$
1032	=	$\underline{01}00\ 0000\ 1000_{U2}$	← taki powinien być wynik

Rysunek 6: Analiza przykładowego błędnego wyniku dodawania

W przykładzie z rys. 6 teoretycznie jest możliwa korekcja poprzez odjęcie liczby 2^{10}, syndromy jednak nie zostają poprawnie rozpoznane, generując błąd na wejściu adresowym pamięci ROM. Podobne problemy (opisane w poprzednim rozdziale) wystąpiły podczas testowania sklejeń z 1 w multiplikatorze – wszystkie błędy, nie związane z wrażliwym na uszkodzenia konwerterem do formatu U2, dotyczyły 10 bitu wyniku.

4.3. Układ korekcyjny

Układ kontrolny, w przeciwieństwie do testowanego wcześniej sumatora i multiplikatora, nie posiada właściwości korekcji skutków pojedynczych uszkodzeń. Możliwa jest za to detekcja każdego pojedynczego błędu poprzez stosowanie bitu parzystości oraz umieszczenie w układzie modułów zdublowanych inwersyjnie. Procedura testowa części kontrolnej ALU polegała na wprowadzeniu uszkodzeń (sklejeń z zerem i jedynką) na wejściach i wyjściach niektórych bramek logicznych, mających kluczowy wpływ na przebieg korekcji. Prawidłowo zaprojektowany układ korygujący powinien informować poprzez jedno z wyjść ERR o swoim uszkodzeniu, w przeciwnym wypadku nie jest pewne, czy wynik uzyskany na wyjściu C_CORR jest poprawny.

Testy zostały przeprowadzone dla dodawania (przedziały $\langle 896\,;1023\rangle$ oraz $\langle -50\,;50\rangle$) oraz mnożenia (przegląd zupełny). Zbiór uszkodzeń obejmował ok. 200 sklejeń.

Tabela 4: Wyniki testów układu korekcyjnego

Wszystkich	5 584 488
Błędnie skorygowanych	1 436 661
Niezerowych syndromów mod 7	1 053 369
Niezerowych syndromów mod 15	123 082
Błędy na linii ERR_M7	1 084 053
Błędy na linii ERR_M15	123 082
Błędy na linii ERR_ROM	1 299 187
Błędy na linii ERR_COR	0
Błędy na linii ERR_ADR	1 299 187

Niestety nie wszystkie uszkodzenia układu korekcyjnego przełożyły się na sygnał błędu na wyjściu – analiza zwróconych wyników wykazała, że niektóre sklejenia w module inwersyjnie zdublowanego sumatora pozostają niewykryte. Nie udało się znaleźć dokładnej przyczyny błędnego działania sumatora (został on zrealizowany na podstawie materiałów zawartych w [2]), dla pozostałych uszkodzeń funkcjonował zgodnie z założeniami.

5. Wnioski

Głównym założeniem podczas tworzenia ALU była możliwość korekcji wszystkich błędów pojedynczych w części arytmetycznej (tj. w sumatorze 12-bitowym oraz multiplikato-

rze). Przy przeprowadzeniu testów wszystkich możliwych uszkodzeń każdej bramki logicznej 99,6 % błędnych wyników zostało skorygowanych, pozostałe 0,4 % nie podlega korekcji. Warto wspomnieć, że błędy niekorygowalne zostają zawsze wykryte poprzez sygnał ERR_ADR na wyjściu układu, dzięki czemu nie występuje sytuacja, w której nie jesteśmy pewni otrzymanego wyniku.

Sklejenia z zerem okazały się znacznie mniej szkodliwe dla działania ALU, układ sumujący był w stanie skorygować każde przekłamanie. Podczas mnożenia słabym ogniwem okazał się konwerter U2, jednakże liczba nieskorygowanych wyników nie przekroczyła 0,01 % wszystkich otrzymanych. Sklejenia z jedynką są bardziej problematyczne, szczególnie w przypadku dodawania.

Zastosowaną w projekcie ideę budowy ALU można uznać za poprawną, wszelkie napotkane błędy w korekcji wyniku da się wyeliminować poprzez dokładną analizę zbyt wrażliwych na uszkodzenia modułów. Głównym problemem napotkanym przy implementacji była niewielka ilość materiałów źródłowych oraz brak danych na temat realizacji układu korygującego. W rezultacie wszystko poza ogólnie stosowanymi multiplikatorami i sumatorami stanowi pracę autorską.

Kody dwuresztowe, używane do korygowania błędów arytmetycznych, stosowane są niezmiernie rzadko ze względu na dużą redundancję wynikłą z podwojenia generatorów reszt [2]. Należy jednak rozważyć sytuację, w której ponowne przeprowadzenie operacji arytmetycznej jest znacznie bardziej kosztowne niż użycie dodatkowych modułów kontrolnych. Możliwość korekcji błędów pojedynczych (lub wielokrotnych dających się skorygować poprzez dodanie/odjęcie wartości 2^i) stanowi wówczas dość atrakcyjną perspektywę.

6. Bibliografia

[1] K. Berezowski. Materiały do wykładu *Niezawodne systemy cyfrowe.* Wrocław, 2006/2007.

[2] J. Biernat. Materiały do wykładu *FTC – niezawodne systemy cyfrowe*. Wrocław, 2007.

[3] R. E. Blahut. *Algebraic Codes for Data Transmission*. Cambridge University Press, Cambridge, 2007.

[4] P. Kocyan. *Materiały pomocnicze do kursu Kodowanie*. Politechnika Wrocławska, 2004/2005.

[5] I. Koren. *Fault Tolerant Computing, part 5 – Coding II* (prezentacja). University of Massachusetts, 2006.

[6] R. S. Lim. *Concurrent Error-Detecting Codes for Arithmetic Processors.* NASA Technical Paper 1528, sierpień 1979.

[7] W. Mochnacki. *Kody korekcyjne i kryptografia*. Oficyna Wydawnicza Politechniki Wrocławskiej, Wrocław, 2000.

[8] P. Oikonomakos. *High-level Synthesis for On-line Testability.* School of Electronics and Computer Science, University of Southampton, 2004.

[9] B. Parhami. *Fault-Tolerant Computing - Error Correction* (prezentacja). University of California, Santa Barbara, 2006.

[10] S. J. Piestrak. *Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders*. IEEE Transactions on Computers, vol. 423, no. 1, styczeń 1994.

[11] S. J. Piestrak. Materiały do wykładu *systemy tolerujące uszkodzenia*. Wrocław, 2006.

[12] S. J. Piestrak. *Metody tolerowania uszkodzeń w układach i systemach cyfrowych*. Instytut Informatyki, Automatyki i Robotyki Politechniki Wrocławskiej, Wrocław, 2005.

[13] R. N Rao. *Biresidue Error-Correcting Codes for Computer Arithmetic*. IEEE Transactions on Computers, vol. C-19, no. 5, maj 1970.

[14] I. J. Wassell. *3F4 Error Control Coding (prezentacja)*. The University of Cambridge, Cambridge, 2007.

[15] S. Wei, K. Shimizu. *Error Detection of Arithmetic Circuits Using a Residue Checker with Signed-Digit Number System*. Gunma University of Japan.

[16] S. S. Yau, Y. Liu. *Error Correction in Redundand Residue Number System*. IEEE Transactions vol. C-22, no. 1, styczeń 1973.