Układy mnożące przez stały współczynnik implementowane w układach programowalnych FPGA

Kazimierz Wiatr, Ernest Jamro

Katedra Elektroniki, Akademia Górniczo-Hutnicza w Krakowie

Otrzymano 2000.11.21

Autoryzowano 2001.01.24

Poniższy artykuł przedstawia różne architektury równoległe układów mnożących o stałym współczynniku mnożenia, implementowanych w układach programowalnych FPGA. W pierwszej części artykułu zostały opisane układy mnożące bezmnożne MM (ang. *Multiplierless Multiplication*). Układy MM wykorzystują reprezentację kanoniczną cyfry ze znakiem CSD (ang. *Canonic Sign Digit*) lub/i dzielnie wspólnej podstruktury SS (ang. *Sub-structure Sharing*). Opisany został również nowy, zoptymalizowany pod kątem generowanego układu MM algorytm konwersji z kodu uzupełnień do dwóch do reprezentacji CSD. Druga część artykułu została poświęcona układom mnożącym wykorzystującym pamięć typu LUT (ang. *Look-Up Table*) i nazywanym w skrócie LM (ang. *LUT based Multiplication*). W konsekwencji opisano wykorzystywanie różnych modułów pamięci oraz znajdowanie optymalnej kombinacji pamięć – układ dodający. Dla układów mnożących LM rozważona została również redukcja szerokości magistrali adresowej dla każdej komórki pamięci jak również możliwość dzielenia wspólnej pamięci dla komórek pamięci o tej samej zawartości. W ostatniej części artykułu podano wyniki implementacji dla układów firmy Xilinx serii XC4000 oraz Virtex.

Słowa kluczowe: układy dedykowane, procesory specjalizowane, układy programowalne, arytmetyka rozproszona

1. Wprowadzenie

Mnożenie jest jedną z podstawowych operacji w cyfrowym przetwarzaniu danych. Niestety dla niektórych aplikacji, układ procesora ogólnego przeznaczenia jak również układ procesora DSP nie jest wstanie sprostać postawionym wymaganiom czasowym, dlatego w tej sytuacji należy zastosować układy dedykowane ASIC (ang. *Application Specific Integrated Circuit*) lub układy programowalne FPGA (ang. *Field Programmable Gate Array*). Architektura układu

mnożącego dla układów ASIC i FPGA wydaje się być taka sama - implementacja układu mnożącego przebiega w podobny sposób i wykorzystywany jest podobny algorytm, jak np. układ mnożarki równoległej typu macierzowego (ang. *parrallel-array*) [1] czy też z wykorzystaniem drzewka Wallace [2]. Jednakże najbardziej znaczącą różnicą pomiędzy układami FPGA a układami ASIC jest to że układy FPGA mogą być łatwo i szybko przeprogramowane. To z kolei umożliwia zmianę współczynnika mnożenia dla układów FPGA albo przez zmianę wartości jednego z argumentu mnożenia dla układu mnożącego o zmiennym współczynniku mnożenia VCM (ang. *Variable Coefficient Multiplier*) lub też zamianę architektury układu mnożącego o stałym współczynniku mnożenia KCM (ang. *Constant Coeficient Multiplier*), dla którego współczynnik mnożenia jest określony przez strukturę układu. Układ mnożący KCM w porównaniu z układem VCM zajmuje dużo mniej powierzchni (26-33% układu VCM dla FPGA [3, 4]) i dlatego jest układem zalecanym pod warunkiem, że współczynnik mnożenia jest relatywnie stały [5] podczas procesu obliczeniowego.

W układach FPGA logika jest implementowana poprzez wykorzystanie pomięci typu LUT (ang. *Look-Up Table*), dlatego naturalną architekturą implementacji mnożenia KCM wydaje się być mnożenie z wykorzystaniem pamięci LUT w układach LM (ang. *LUT based Multiplication*). W celu zmieszczenia rozmiaru pamięci LUT, duża pamięć LUT jest dzielona na mniejsze pamięci w połączeniu z odpowiednimi układami dodającymi [3, 4, 6]. Z drugiej strony w układach ASIC, KCM jest implementowany z wykorzystaniem układów mnożących bezmnożnych MM (ang. *Multiplierless Multiplication*) wykorzystując operacje dodawania i odejmowania [7]. W konsekwencji architektury KCM implementowane w układach ASIC i FPGA są inne. Układy FPGA nowszej generacji posiadają wbudowany układ propagacji przeniesienia dla układu dodającego z przeniesieniem skrośnym (ang. *ripple-carry adder*) dzięki czemu układ dodający zajmuje dwa razy mniejszą powierzchnię układu scalonego i jego szybkość uległa znaczącej poprawie [8]. Jednakże ulepszenie układu dodającego nie zostało uwzględnione w efektywności architektur MM i LM dla układów FPGA i w konsekwencji architektura MM nie jest zalecana dla układów FPGA. Dlatego celem tej publikacji jest porównanie wyników implementacji LM i MM w układach FPGA z uwzględnieniem ich wzajemnej relacji czasowo-powierzchniowej.

W pierwszej części artykułu opisane zostały układy MM wykorzystujące reprezentację kanoniczną cyfry ze znakiem CSD (ang. *Canonic Sign Digit*) i dzielenie wspólnej podstruktury

SS (ang. *Sub-structure Sharing*). Zaprezentowany został również nowy, zmodyfikowany algorytm konwersji z kodu uzupełnień do dwóch do kodu CSD, który uwzględnia budowę układu dodającego w układach FPGA. W dalszej części artykułu szczególnie rozpatrywana jest architektura LM. Układy FPGA posiadają różne moduły pamięci, np. układy Virtex firmy Xilinx posiadają pamięci 16×1, 32×1, 256×1, itd.), dlatego znalezienie optymalnej kombinacji tych pamięci i układów dodających jest skomplikowanym zadaniem, któremu poświęcono dużo uwagi. Ponadto, niektóre komórki pamięci mogą posiadać skróconą szerokość magistrali adresowej lub też dwie (a nawet więcej) komórki pamięci posiadające tę samą zawartość, przez co mogą być zastąpione tylko przez jedną, wspólną komórkę pamięci. W tym miejscu należy zwrócić uwagę, że każdy etap referowanej pracy jest potwierdzony rzeczywistymi wynikami badań, weryfikując w ten sposób prowadzone rozważania i stawiane tezy dotyczące różnych architektur.

2. Układy mnożące bezmnożne (MM)

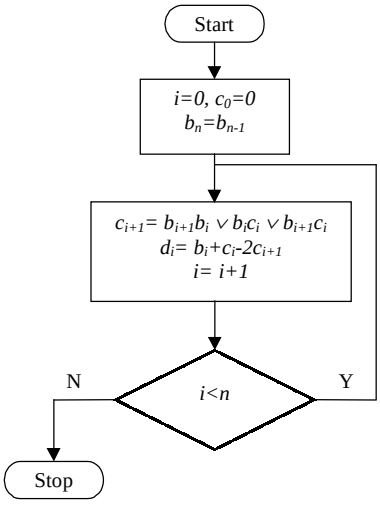
Układ mnożący o stałym współczynniku mnożenia KCM jest najczęściej implementowany w postaci bezmnożnej MM (ang. *Multiplierless Multiplication*) czyli przy wykorzystaniu tylko układów dodających i odejmujących oraz przesunięcia bitowego, których struktura jest określona przez wartość współczynnika mnożenia, a konkretnie jego binarnej reprezentacji BR. Dla przykładu: $A \cdot B$, gdzie $B = 14 = 1110_2$ może być zaimplementowane jako $(A<<1) + (A<<2) + (A<<3)$, gdzie „$A<<n$" oznacza przesunięcie bitowe liczby A o n bitów w lewo. Należy podkreślić, że koszt (zajmowany obszar układu scalonego) układu mnożącego zależy bezpośrednio od wartości współczynnika mnożącego - liczba jedynek w jego binarnej reprezentacji powinna być jak najmniejsza. W związku z tym wiele algorytmów zostało tak zorganizowanych, aby sprostać temu wymogowi w celu zmniejszenia kosztów implementacji układu mnożącego, np. dla filtrów FIR [9]. Jednakże w tym artykule wartość współczynnika mnożenia jest narzuconym parametrem wejściowym, którego wartość nie może być zmieniona.

2. 1. Postać kanoniczna cyfry ze znakiem (CSD)

W reprezentacji CSD redukcje obszaru zajmowanego przez układ mnożący uzyskuje się poprzez redukcję jedynek w reprezentacji liczby [10]. W reprezentacji CSD w porównaniu z BR (reprezentacja binarna) wprowadzona została dodatkowa cyfra $\bar{1}$ mająca wartość –1 i symbolizująca operację odejmowania. Użycie reprezentacji CSD powoduje, że mnożenie może być przeprowadzone z użyciem mniejszej liczby operacji dodawania (odejmowania) w porównaniu z reprezentacją binarną. W poprzednim przykładzie dla BR: mnożenie $A \cdot B$, gdzie $B= 14= 1110_2= 100\bar{1}0_{CSD}$ wymagało 2 operacji dodawania, natomiast dla CSD wymagana jest tylko jedna operacja odejmowania: *(A<<4) – (A<<1)*. Przeciętnie reprezentacja CSD zawiera o 33% mniej jedynek niż BR [10], co pociąga oszczędności rzędu 33%.

Dodatkowa cyfra $\bar{1}$ umożliwia zakodowanie cyfry binarnej na wiele różnych sposobów. Przykładowo liczba $11= 1011_2$ może być zakodowana w następujący sposób: 1011_{CSD}= 8+2+1, $110\bar{1}_{CSD}$= 8+4-1, $10\bar{1}0\bar{1}_{CSD}$= 16-4-1. Sposób kodowania liczby w kodzie CSD decyduje o koszcie układu mnożącego, dlatego konwersja do kodu CSD zostanie tutaj przybliżona.

Standardowa konwersja z reprezentacji binarnej (lub uzupełnień do dwóch - TC) liczby $B=b_{n-1}, b_{n-2}, ..., b_0$ do reprezentacji CSD $D= d_{n-1}, d_{n-2}, ..., d_0$ przebiega według algorytmu przedstawionego na rysunku 1[7].



Rys. 1. Algorytm konwersji do reprezentacji CSD

Podstawą działania powyższego algorytmu jest założenie, że w kodzie CSD nie mogą wystąpić obok siebie dwie (lub więcej) jedynki. W przeciwnym przypadku na pozycję najmniej

znaczącej jedynki wpisywany jest symbol $\bar{1}$. Dzieje się to według zależności 011..11 = 100..0$\bar{1}$. Jak widać w rezultacie działania tego algorytmu liczba operacji odejmowania lub dodawania ulega zmniejszeniu lub pozostaje niezmieniona. Warto w tym miejscu podkreślić, że powyższy algorytm konwersji do CSD traktuje operację dodawania i odejmowania jako operacje równoważne, tzn. zajmujące taką samą powierzchnię układu scalonego. Dla przykładu, liczba 3= 11_2 podlega konwersji na $10\bar{1}$ co nie zmniejsza liczbę operacji a wprowadza odejmowanie zamiast dodawanie.

W przypadku układów FPGA operacje dodawania i odejmowania nie są sobie równoważne. Aby to wykazać przybliżona zostanie struktura układu dodającego i odejmującego implementowanego w układach FPGA. Operacja dodawania $S= A+B$ wykorzystuje następujące operacje:

$$s_i = a_i \; xor \; b_i \; xor \; c_i \tag{1}$$

$$if \; a_i = b_i \; then \; c_i = a_i \; else \; c_i = c_{i-1}. \tag{2}$$

Podobnie operacja odejmowania $S= A – B$ może być wyrażona jako:

$$s_i = a_i \; xor \; not \; b_i \; xor \; c_i \tag{3}$$

$$if \; a_i = not \; b_i \; then \; c_i = a_i \; else \; c_i = c_{i-1}. \tag{4}$$

Dla układów FPGA (np. uklady serii XC4000 firmy Xilinx) operacje (1) i (3) są implementowane w czterowejściowych pamięciach LUT, które mogą zaimplementować dowolną funkcję logiczną czterech argumentów. Operacje (2) i (4) są implementowane w dedykowanym układzie logiki przeniesienia [8], dzięki czemu sygnał przeniesienia propagowany jest 5-50 razy szybciej niż w przypadku standardowych operacji oraz nie zajmuje dodatkowej powierzchni układu. Porównując operacje (1-4) można zobaczyć, że operacja odejmowania wymaga tylko dodatkowej operacji negacji bitów b_i, dlatego założenie równoważności (tego samego kosztu) operacji odejmowania i dodawania wydaje się być spełnione, co zresztą jest potwierdzone w ogólnym przypadku dla układów FPGA. Jednakże dla dodawania, dla którego jeden z argumentów jest przesunięty bitowo w lewo, bity mniej znaczące (LSB) argumentu drugiego są

bezpośrednio kopiowane na wyjście, dzięki czemu zmniejsza się koszt układu dodającego. Niestety w przypadku operacji odejmowania bity LSB odjemnika nie mogą być kopiowane na wyjście ponieważ bity te muszą być zanegowane i musi zostać dodana jedynka do najmniej znaczącego bitu (c_0= *1* w przypadku odejmowania, patrz operacje (3, 4)). W konsekwencji operacje dodawania i odejmowania nie są równoważne. Na przykład dla operacji mnożenia przez $3= 11_2= 10\bar{1}_{CSD}$ (dla BR) LSB może być skopiowany bezpośrednio na wyjście, natomiast dla CSD nie jest to możliwe. Dla układów ASIC, dodatkowe operacje negacji i dodanie jedynki do LSB mogą być zaimplementowane na półsumatorach, których koszt jest znacznie niższy. Niestety w przypadku układów FPGA koszt pełnego sumatora i półsumatora jest w przybliżeniu taki sam.

2. 2. Zmodyfikowany algorytm konwersji do CSD

Standardowy algorytm konwersji reprezentacji BR do kodu CSD zakłada, że operacje dodawania i odejmowania są sobie równoważne, czyli nie uwzględnia wcześniejszych rozważań. W konsekwencji algorytm konwersji został zmodyfikowany tak, aby konwersja do symbolu $\bar{1}$ (symbolizującego operacje odejmowania) została przeprowadzona tylko wtedy, gdy liczba operacji (liczba niezerowych symboli w reprezentacji CSD) ulega zmniejszeniu.

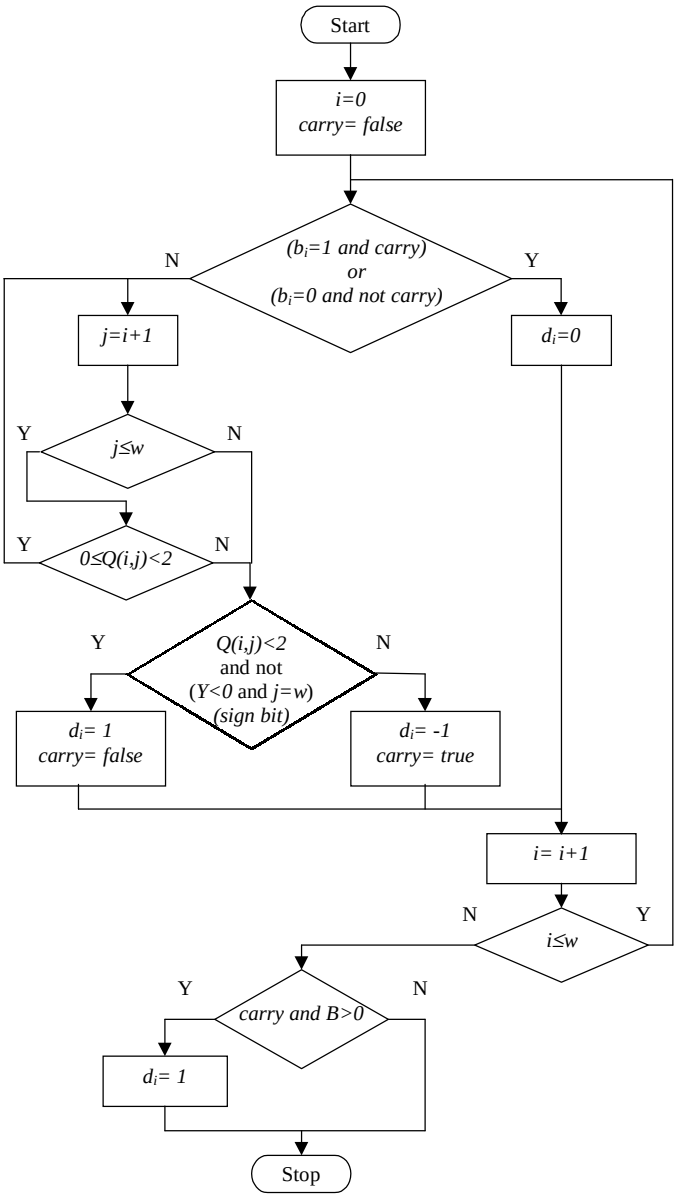
W celu opisania zmodyfikowanego algorytmu konwersji do CSD (nazywanego dalej MCSD) zostanie wprowadzona nowa funkcja *Q(i,j),* gdzie j≥i. Niech b_j reprezentuje *j*-ty bit reprezentacji binarnej (lub uzupełnień do dwóch) współczynnika mnożenia *B (M= A·B)*. Funkcja *Q(i, j)* niech będzie zdefiniowana iteracyjnie:

$$Q(i,i) = 0$$
$$Q(i, j+1) = \begin{cases} Q(i, j) + 1 & gdy \quad b_{j+1} = 1 \\ Q(i, j) - 1 & gdy \quad b_{j+1} = 0 \end{cases} \tag{5}$$

Funkcja *Q(i,j)* jest inkrementowana jeżeli binarny symbol b_{j+1} jest jedynką i dekrementowana jeżeli jest zerem.

Zmodyfikowany algorytm konwersji (MCSD) jest pokazany na rysunku 2. Konwersja do symbolu $\bar{1}$ ma miejsce tylko wtedy gdy ogólna liczba operacji ulegnie zmniejszeniu.

W konsekwencji liczba jedynek z rzędu w BR musi być równa co najmniej 3. Jeżeli natomiast zero przerywa sąsiadujące jedynki wtedy liczba jedynek z rzędu (po odrzuceniu zera) powinna być zwiększona o 2, lub też równoważnie, licznik jedynek powinien być zmniejszony o 1, jak to ma miejsce w przypadku funkcji *Q(i,j)*. Proces liczenia jedynek (funkcja *Q(i,j)*) powinien być zakończony w momencie kiedy funkcja *Q(i,j)* jest mniejsza od zera – wstawiony ma być symbol 0 lub 1, lub też jeżeli jest równa 2 – wstawiony symbol $\bar{1}$.



Rys. 2. Zmodyfikowany algorytm konwersji do CSD, b_i –i-ty bit BR, Q(i,j) - funkcja zdefiniowana w równaniu (5), w- szerokość współczynnika mnożenia-1 (liczbę bitów, na których należy zapisać współczynnik mnożenia-1)

Przykładowe wyniki dla standardowego i zmodyfikowanego algorytmu konwersji podane są w Tablicy 1. Dla liczby *3* i *11* standartowy algorytm wprowadził symbol $\bar{1}$, mimo tego, że liczba operacji nie uległa zmianie a jedynie operacja dodawania jest zastąpiona operacją odejmowania. Zmodyfikowany algorytm w tym przypadku nie wprowadza żadnych zmian co jest w zgodzie z przyjętym założeniem wprowadzania zmian w kodzie tylko w wypadku redukcji liczby operacji. W przypadku liczby 7 zastosowanie znaku $\bar{1}$ zmniejsza liczbę operacji dlatego oba algorytmy konwersji wprowadzają ten symbol. W przypadku liczby 23 zmodyfikowany algorytm wprowadza tylko jeden symbol $\bar{1}$ natomiast algorytm standardowy aż dwa symbole $\bar{1}$.

Tablica 1

Przykład wyników dla standardowego i zmodyfikowanego algorytmu konwersji do CSD

Współczynnik	BR	CSD	MCSD
3	11	$10\bar{1}$	11
7	111	$100\bar{1}$	$100\bar{1}$
11	1011	$10\bar{1}0\bar{1}$	1011
23	10111	$10\bar{1}00\bar{1}$	$1100\bar{1}$

2. 3. Dzielenie wspólnej podstruktury (SS)

Inną metodą redukcji obszaru zajmowanego przez układ mnożący jest dzielnie wspólnej podstruktury SS (ang. *Sub-structure Sharing*) [11]. Na przykład, mnożenie przez liczbę $27 = 11011_2$ może być zrealizowane z użyciem dodatkowej zmiennej pomocniczej *tmp,* tak jak to pokazano na poniższym równaniu, dla którego liczba operacji dodawania została zmniejszona z 3 do 2.

$$tmp = a + (a<<1) \qquad (6)$$
$$27 \cdot a = tmp + (tmp<<3)$$

Operacja SS może być również przeprowadzona na reprezentacji CSD, dlatego kombinacja SS i CSD została również uwzględniona podczas implementacji algorytmu optymalizacji generowanego układu. Z drugiej jednak strony, CSD może zakłócać optymalizację SS, dlatego algorytm SS został zaimplementowany zarówno na BR jak i CSD, a wybrany został lepszy rezultat.

2. 4. Wyniki doświadczalne

Porównanie opisanych w tym rozdziale technik jest trudne ze względu na to, że wynik i wybór optymalnego algorytmu w dużym stopniu zależy od wartości współczynnika mnożenia. Jednakże można wyciągnąć ogólne wnioski dotyczące średniego i maksymalnego kosztu układu oraz liczby wystąpień najlepszego algorytmu.

Wyniki doświadczalne zostaną przedstawione dla 8-bitowego wejścia bez-znaku (najbardziej popularny format danych dla przetwarzania obrazu) oraz współczynników mnożenia o szerokości *K= 3-12* bitów. Wyniki podane są w Tablicy 2.

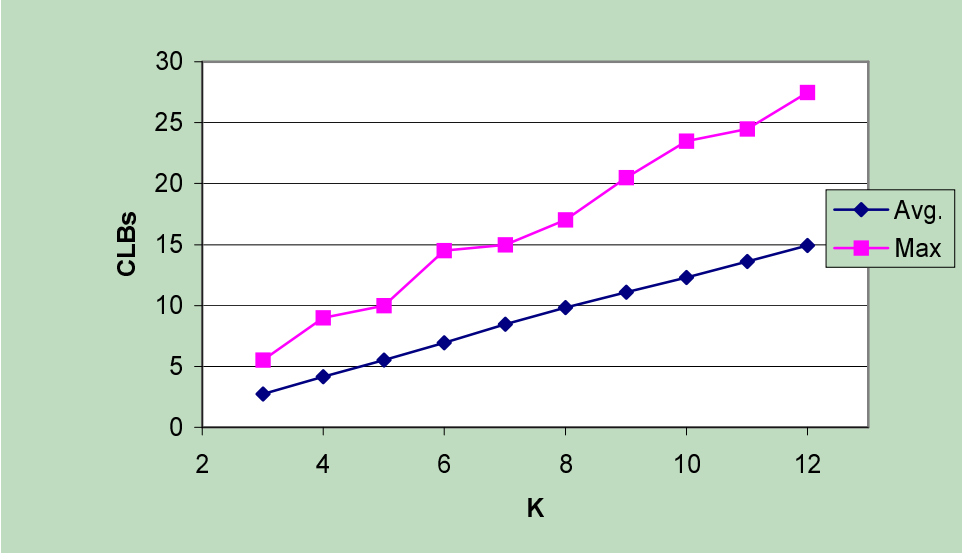
Tablica 2

Średnia i maksymalna liczba CLB dla XC4000 oraz odpowiadająca liczba operacji dodawania (odejmowania) L dla najlepszego algorytmu (wybieranego osobno dla każdego współczynnika mnożenia). K- maksymalna szerokość współczynnika mnożenia (współczynnik mnożenia w zakresie [1, 2^K-1]). Najlepszy algorytm - liczba współczynników mnożenia dla których podany algorytm daje najlepszy rezultat, jeżeli rezultat (koszt w CLB a nie liczba operacji) dwóch algorytmów jest taki sam to wybierany jest algorytm wcześniejszy (w tabeli bardziej na lewo)

K	Średnio		Max.			Najlepszy algorytm			
	CLB	L	CLB	L	Współ.	BR	CSD	SS	CSD-SS
3	2,71	0,57	5,5	1	7	6	1	0	0
4	4,13	0,87	9	2	11	12	3	0	0
5	5,52	1,16	10	2	23	22	9	0	0
6	6,92	1,44	14,5	3	43	39	17	4	0
7	8,44	1,75	15	3	75	68	43	16	0
8	9,8	2,03	17	3	183	114	94	44	3
9	11,1	2,31	20,5	4	309	188	193	107	15
10	12,3	2,57	23,5	4	747	300	407	254	62
11	13,6	2,83	24,5	4	1463	478	797	579	193
12	14,9	3,08	27,5	4	3381	746	1510	1285	554

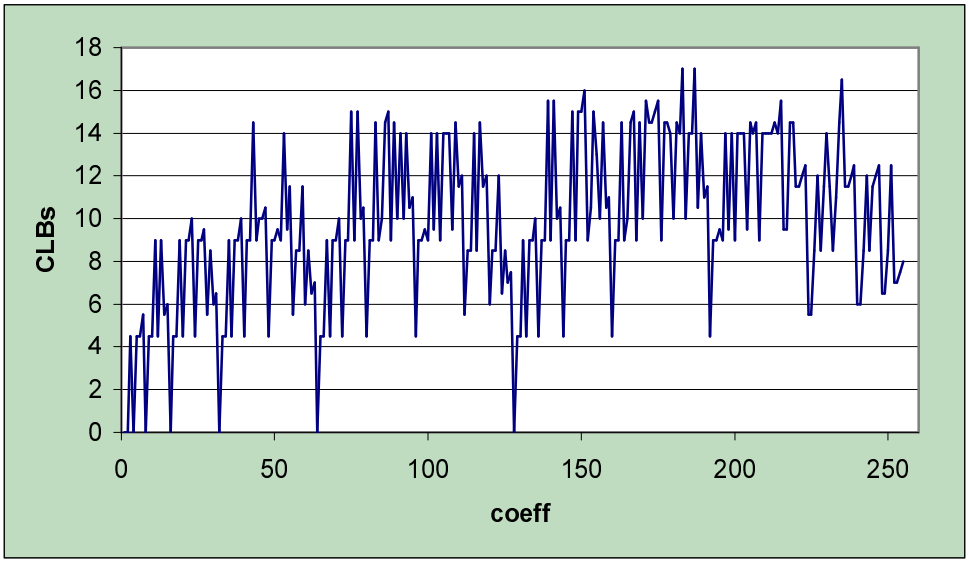
Analizując wyniki w Tablicy 2 można zauważyć, że algorytmy optymalizacji (CSD, SS, CSD-SS) są bardziej skuteczne dla szerszych współczynników mnożenia. Średnia liczba operacji dla BR jest w przybliżeniu równa *K/2-1*, co dla *K= 12* daje *5*, w porównaniu z *3,08* z zastosowaniem technik optymalizacji. Średni koszt układu mnożącego wzrasta w przybliżeniu liniowo ze wzrostem szerokości współczynnika mnożenia *K*, jak to jest pokazane na wykresie (rys. 3 – linia Avg). Podobny jest przebieg maksymalnego kosztu (na wykresie linia Max),

chociaż dla tego przebiegu odchylenia są znacznie większe, szczególne dla *K*, dla którego następuje wzrost liczby maksymalnej liczby operacji *L* (dla *K= 4, 6, 9*). Warto zwrócić uwagę, że występuje wzrost kosztu maksymalnego pomimo, że liczba operacji pozostaje stała. W konsekwencji, wybór najlepszego algorytmu nie zależy tylko od liczby operacji dodawania (odejmowania), ale musi zostać uwzględniony całkowity koszt układu mnożącego.



Rys. 3. Średnia i maksymalna powierzchnia zajmowany przez 8-bitowe układ mnożący dla różnej szerokości współczynnika mnożącego K

Koszt układu mnożącego silnie zależy od wartości współczynnika mnożącego co jest pokazane na wykresie (rys. 4). Można zauważyć, że najbardziej kosztowny układ mnożący występuje dla współczynnika o wartości 60-75% pełnego zakresu binarnego.

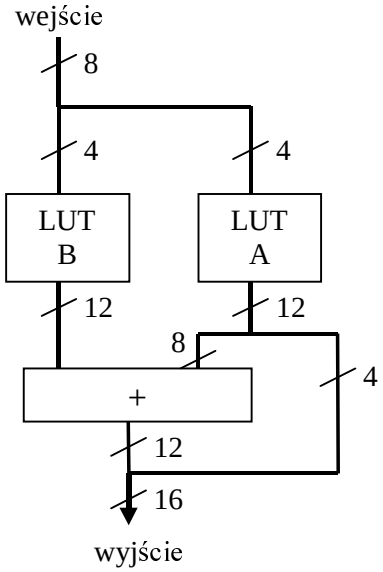


Rys. 4. Koszt układu mnożącego o wejściu 8-bitowym dla różnych współczynników mnożenia. Układ zaimplementowany w XC4000

3. Układy mnożące z wykorzystaniem pamięci LUT (LM)

3. 1. Zasada działania

Obliczenie dowolnej ograniczonej wartościowo funkcji możliwe jest z wykorzystaniem pamięci typu LUT, na której wejście adresowe podane są wartości argumentów, a na wyjściu otrzymuje się wynik funkcji. Teoretycznie zastosowanie pamięci LUT daje najszybszą implementację funkcji ponieważ nie jest wymagane zastosowanie żadnych układów arytmetycznych. Niestety, zastosowanie w praktyce pojedynczej pamięci LUT dla mnożenia jest możliwe tylko dla małych wartości argumentu, ponieważ wielkość pamięci wzrasta wykładniczo wraz z szerokością argumentu. Na przykład, dla *L*-bitowego argumentu i *K*- bitowego współczynnika mnożenia, wymagana jest pamięć o rozmiarze $(L+K)\cdot 2^L$. Dlatego praktycznym rozwiązaniem jest zastosowanie kombinacji małych pamięci LUT i układów dodających, poprzez rozbicie argumentu, użycie małych pamięci LUT i następnie odpowiednie dodanie wyjść pamięci LUT [3,6,12]. Przykład takiego układu mnożącego jest pokazany na rysunku 5, gdzie operacja mnożenia $Y= A\cdot B$ jest przeprowadzona według zależności $Y= 2^4\cdot B\cdot A_{bity\ 7\text{-}4} + B\cdot A_{bity\ 3\text{-}0}$.



Rys. 5. Układ mnożący LM dla szerokości argumentu L=8 i szerokości współczynnika mnożącego K=8

Zawartość pamięci LUT dla mnożenia $Y = A \cdot B$ może być obliczona bezpośrednio z mnożenia tak jak to jest pokazane na przykładzie w Tablicy 3.

Tablica 3

Zawartość komórek pamięci LUT ($y_5 - y_0$) dla różnych wartości argumentu podanego na magistralę adresową; współczynnik mnożenia wynosi 19; szerokość adresu – szerokość magistrali adresowej dla poszczególnych komórek pamięci

Adres	wartość	y_5	y_4	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0
1	19	0	1	0	0	1	1
2	38	1	0	0	1	1	0
3	57	1	1	1	0	0	1
Szerokość adresu		1	1	2	2	2	1

Warto zwrócić uwagę, że zawartość komórek pamięci LUT zależy tylko od linii adresowych o wadze mniejszej lub równej wadze komórki wyjściowej. Dlatego w przedstawionym przykładzie w Tablicy 3, komórka pamięci y_0 zależy tylko od linii adresowej a_0; y_1 zależy od a_0 i a_1; itd. W rezultacie szerokość linie adresowej wzrasta z wagą bitu wyjściowego pamięci LUT. Maksymalna szerokość magistrali adresowej jest jednak ograniczona od góry przez szerokość *n* magistrali adresowej pamięci LUT. W rezultacie szerokość magistrali adresowej komórki y_i w ogólnym przypadku wynosi *MAX(i+1, n)*. W konsekwencji *n-1* najmniej znaczących bitów pamięci wymaga mniejszej szerokości magistrali adresowej, a przez to i mniejszych rozmiarów pamięci, co powoduje znaczącą redukcję kosztów pamięci i będzie oznaczane dalej jako LAWR (ang. *LSBs Address Width Reduction*).

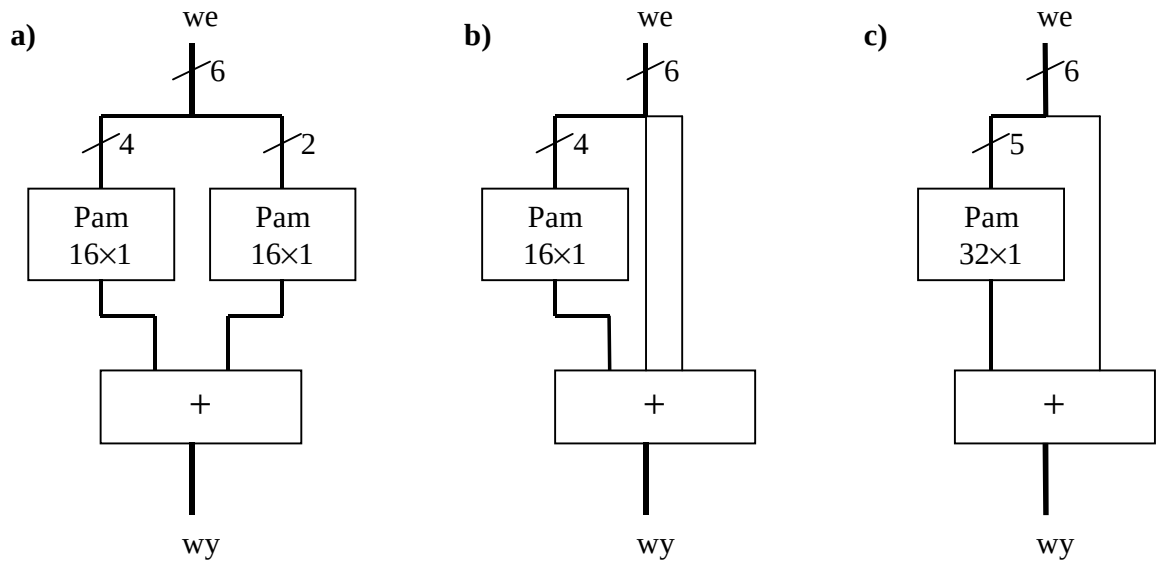
Dodatkową redukcję szerokość magistrali adresowej można zaobserwować dla komórek pamięci, dla których zawartość pamięci nie zależy od pewnej linii adresowej. Osiągnięte dzięki tej metodzie oszczędności będą dalej w skrócie nazywane jako DAWR (ang. *Don't care Address Width Reduction*). DAWR jest z reguły obserwowany dla najbardziej znaczących bitów pamięci LUT, co potwierdza przykład w Tabeli 3, dla którego DAWR występuje dla komórek pamięci y_5 i y_4. Dodatkowe oszczędności można osiągnąć dzięki dzieleniu wspólnych komórek pamięci MS (ang. *Memory Sharing*), jeżeli te komórki mają taką samą zawartość. Na przykładzie Tabeli 3 można zauważyć, że komórki pamięci y_0 i y_4 są takie same, dlatego tylko jedna z nich może być zastosowana. Warto w tym miejscu podkreślić, że redukcję szerokości magistrali adresowej DAWR oraz dzielenie wspólnej pamięci MS nie można uogólnić i zależą one silnie od wartości współczynnika mnożenia i szerokości magistrali adresowej. Ponadto, wymagany jest zaawansowany algorytm przeszukujący i implementujący wspomniane techniki. W przedstawionych rozważaniach zastosowano algorytm porównujący wszystkie możliwe kombinacje komórek i linii adresowych.

3. 2. Implementacja układu LM w układach FPGA

W układzie mnożącym rozbicie dużej pamięci LUT na mniejsze powinno być przeprowadzone z uwzględnieniem relacji koszt-rozmiar pamięci oraz koszt-rozmiar układów dodających. Układy serii XC4000 firmy Xilinx mają wbudowane pamięci 16×1 oraz 32×1, jednakże koszt pamięci 32×1 jest równy 1 CLB i jest dwa razy większy niż dla pamięci 16×1 (½

CLB). Koszt układu dodającego jest równy ½ CLB/bit. Ponadto, istnieje dodatkowa wirtualna pamięć 2×1, która może być zaimplementowana jako bezpośrednie połączenie (z ominięciem pamięci – zerowy koszt) do układu dodającego.

W konsekwencji, znalezienie optymalnej kombinacji różnych modułów pamięci oraz układów dodających jest skomplikowanym zadaniem, które zależy od rozmiaru danej wejściowej i narzuconej wartości współczynnika mnożenia. Na podstawie przeprowadzonych badań doświadczalnych, dla danej wejściowej dużo większej niż 4 preferowanym rozmiarem pamięci jest pamięć 16×1. Jeżeli jednak szerokość argumentu nie dzieli się przez 4, mogą być użyte różne bloki pamięci.

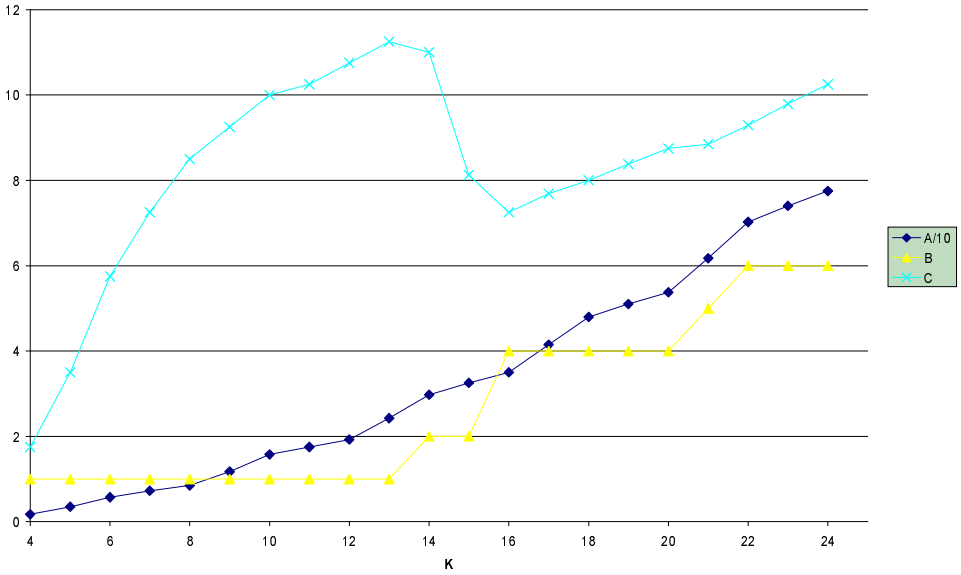


Rys. 6. Różne korzystne metody implementacji układu mnożącego LM dla szerokości danej wejściowej równej 6

Rysunek 6 przedstawia kilka dobrych rozwiązań układu mnożącego; koszt układów mnożących implementowanych w układach XC4000 wyniesie dla poszczególnych rozwiązań odpowiednio: a) 11½ CLB, b) 12 CLB, c) 12 CLB. Jak widać różnica kosztów dla poszczególnych rozwiązań jest niewielka. Uogólniając, jednym z najlepszych rozwiązań jest układ, który używa tylko pamięci 16×1. Jeżeli pozostaje tylko jedna linia adresowa, która jest niepodzielna przez 4, należy użyć bezpośredniego połączenia do układu dodającego.

Układy serii Virtex firmy Xilinx oferują jeszcze bogatszy wybór pamięci. Oprócz pamięci 16×1 i 32×1 wprowadzono w układach Virtex dodatkowe duże bloki pamięci BSR (ang. *Block*

Select RAM) o rozmiarze 4kb i różnej szerokości magistrali danych: 4k×1, 2k×2, 1k×4, 512×8, 256×16 [13]. Powierzchnia krzemu zajmowana przez pojedynczy BSR jest porównywalna do powierzchni zajmowanej przez 16 Virtex CLB (64 RAM 16×1). Jednakże rzeczywisty koszt tych pamięci może zależeć od wolnych zasobów układu FPGA, np. układ ma już zajęte wszystkie CLB, ale ma jeszcze wolne zasoby BSR, dlatego rzeczywistą zależność kosztów tych pamięci należy dostosować do aktualnego projektu. Wykres na rysunku 7 przedstawia ekwiwalentny koszt pamięci BSR (dodatkowy koszt układu po zastąpieniu pamięci BSR pamięciami 16×1 i 32×1 i podzieleniu przez liczbę użytych BSR) dla różnych szerokości wejścia i współczynnika mnożenia. Na podstawie tego wykresu można odczytać, że pojedyncza pamięć BSR może być zastąpiona przez średnio 8-11 VCLB (około 32-44 RAM 16×1).



Rys. 7. Powierzchnia układu LM dla różnej szerokości magistrali wejściowej i współczynnika mnożenia K. A) powierzchnia (podawana dla Virtex CLB (1 VCLB= 2 XC4000 CLB) w skali 1:10 dla LM z użyciem tylko pamięci 16×1 i 32×1; B) liczba użytych bloków BSR 256×16; C) ekwiwalentny koszt (podawany dla VCLB) jednego bloku BSR w porównaniu z rozwiązaniem przy użyciu tylko pamięci 16×1 i 32×1

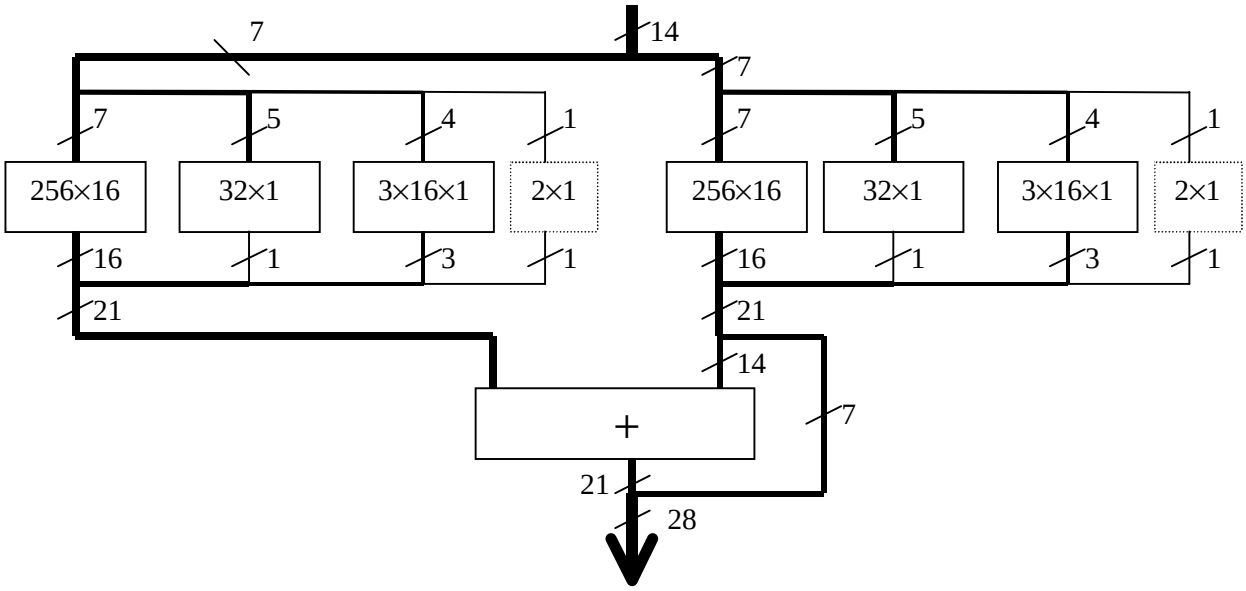
Bloki pamięci BSR są relatywnie duże i dlatego trudno jest znaleźć architekturę układu mnożącego, dla której są one całkowicie wykorzystane, a stopień wykorzystania bloków pamięci bardzo wpływa na ich ekwiwalentny koszt. Konsekwentnie, dla małej szerokości magistrali

wejściowej i współczynnika mnożenia K ($K<8$) ekwiwalentny koszt BSR jest raczej mały (rys. 7), co nie zachęca do użycia tych pamięci. Dla $K\leq13$ ekwiwalentny koszt BSR wzrasta. Natomiast dla $K>13$, dwa lub więcej bloki BSR są użyte co wymaga użycia dodatkowych układów arytmetycznych oraz ponownie pojawia się problem pełnego wykorzystania pamięci BSR (występuje efekt ziarnistości dużych bloków BSR), przez co zmniejsza się ekwiwalentny koszt pamięci BSR. Dla $K=16$, liczba użytych bloków BSR wzrasta gwałtownie, jako że bloki BSR są grupowane w pary stanowiące w sumie dużą pamięć 256×32, co również zmniejsza ekwiwalentny koszt BSR. Dalsze zwiększanie K powoduje wzrost ekwiwalentnego kosztu, wydaje się jednak, że wartość maksymalna 11 nie zostanie przekroczona.

Wykres A na rysunku 7 pokazuje koszt LM przy użyciu tylko pamięci 16×1 i 32×1. Można zauważyć gwałtowny wzrost kosztu układu mnożącego dla $K= 9, 13, 17, 23$, kiedy to K przekracza liczbę podzielną przez 4 – liczbę wejść do pamięci 16×1.

W przedstawionych rozważaniach w celu znalezienia optymalnej architektury układu mnożącego wykorzystano technikę przeszukiwania wszystkich możliwych kombinacji (ang. *exhausted search*). Wszystkie możliwe i korzystne kombinacje pamięci BSR (32×1 i 16×1) oraz układów dodających zostały rozpatrzone i został wybrany najlepszy układ. W celu wizualizacji rozważanych architektur na rysunku 8 przedstawiono przykład układu mnożącego, w którym użyto kombinacje różnych pamięci RAM i układu dodającego. W przykładzie tym (jak i we wszystkich rozważaniach w tym podrozdziale) nie uwzględniono możliwej optymalizacji MS i DAWR, ze względu na ich ścisłe powiązanie z wartością współczynnika mnożenia. Dlatego po uwzględnieniu tych optymalizacji schemat na rysunku 8 może ulec dalszej komplikacji.

Warto podkreślić, że opisane tutaj wyniki zostały uzyskane dzięki specjalne zaprojektowanemu w tym celu pakietowi AuToCon napisanemu w języku C++ i VHDL, który automatycznie generuje optymalną architekturę układu mnożącego. Dokładne opisanie działania tego narzędzia przekracza ramy tego artykułu. Program AuToCon nie zakłada jakichkolwiek związków pomiędzy kosztami modułów pamięci i układów dodających, dlatego moduły pamięci mogą być dowolnie zdefiniowane i program może być użyty dla różnych rodzin układów FPGA, a także dla projektów ASIC. Parametrami wejściowymi jest koszt układów dodających oraz rozmiar i koszt modułów pamięci. Wyjściem programu jest plik tekstowy VHDL, który po syntezie może być zaimplementowany do dowolnego układu FPGA.



Rys. 8. Układ mnożący LM dla K= 14

W przedstawionych rozważaniach uwzględniono jedynie układy serii XC4000 i Virtex firmy Xilinx, ale podobne właściwości mają również inne układy FPGA. Na przykład, układy Apex 20k firmy Altera [14] mają podobne relacje kosztów jak w przypadku układów Virtex. Apex posiada czterowejściową pamięci LUT 16×1 i dedykowany układ generacji przeniesienia w każdym elemencie logicznym LE (ang. *Logic Element*) oraz posiada duże bloki pamięci (128×16, 256×8, 512×4, 1024×2 i 2048×1) w ESB (ang. *Embedded System Block*). Rozmiar pamięci dla układów Apex jest o połowę mniejszy niż w przypadku BSR układów Virtex, jakkolwiek w przeważającej większości 4kb BSR nie jest w pełni wykorzystany i dlatego równie znaczącą różnicą jest brak pamięci 32×1 dla układów Apex.

Należy zwrócić uwagę, że liczba użytych bloków BSR zależy od relacji kosztów. Na przykład, dla 16-bitowego układu mnożącego (*K=16*) liczba BSR wzrasta z malejącym kosztem pamięci BSR, jak to jest pokazane w Tablicy 4.

Tablica 4

Wpływ kosztu bloku pamięci BSR na architekturę LM dla K=16; dla kosztów: 16×1= 0.25 VCLB oraz sumator = 0,25 VCLB/bit

Koszt BSR [VCLB]	liczba BSRs	Koszt LUT RAM [VCLB]	Koszt Sumatora [VCLB]	Ekw. koszt BSR [VCLB]
≥7,75	0	19	16	-
≥6,5	2	9,5	10	7,75
≤6,25	4	0	6	7,25

Bloki pamięci BSR podczas implementacji bardzo często nie są w pełni wykorzystywane (patrz przykładowy schemat z rysunku 8, w którym jedna linia adresowa pamięci BSR nie została użyta), co może sugerować o ich nieoptymalnych rozmiarach. Pamięci BSR: 4k×1, 2k×2, 1k×4 oraz 512×8 nigdy nie zostały użyte, a wykorzystywane były tylko pamięci 256×16. W konsekwencji korzystne wydaje się wprowadzenie pamięci o szerszej magistrali danych, np. 128×32. Uogólniając, można stwierdzić, że optymalna wielkość pamięci, aby została w pełni wykorzystana powinna być następująca:

$$W_D = W_C + W_A - W_L \qquad (7)$$

gdzie: W_D*- szerokość magistrali danych;* W_A*- szerokość magistrali adresowej;* W_L*- szerokość magistrali adresowej następnej mniej kosztownej pamięci, dla układu Virtex* $W_L = 5$.

Podstawiając do równania (7) $W_C = 13$, $W_A = 8$, $W_L = 5$ otrzymujemy $W_D = 16$, co odpowiada pamięci 256×16, i dlatego maksymalny ekwiwalentny koszt pamięci BSR jest obserwowany na rysunku 7 dla szerokości współczynnika mnożenia $K = 13$.

Dodatkowe oszczędności można uzyskać w przypadku, gdy binarny zakres wejścia nie jest w pełni wykorzystywany. Na przykład, dla zakresu danej wejściowej 0-127 (zakres binarny) oraz

dla zakresu 0-99 (zakres dziesiętny) i współczynnika mnożenia równego 81, wynik implementacji układu mnożącego wynosi odpowiednio: 14,5 CLB oraz 13,5 CLB (dla XC4000).

Ponadto zoptymalizować można arytmetykę liczb ujemnych. Generalnie obszar optymalizacji może być podzielony na cztery zakresy:

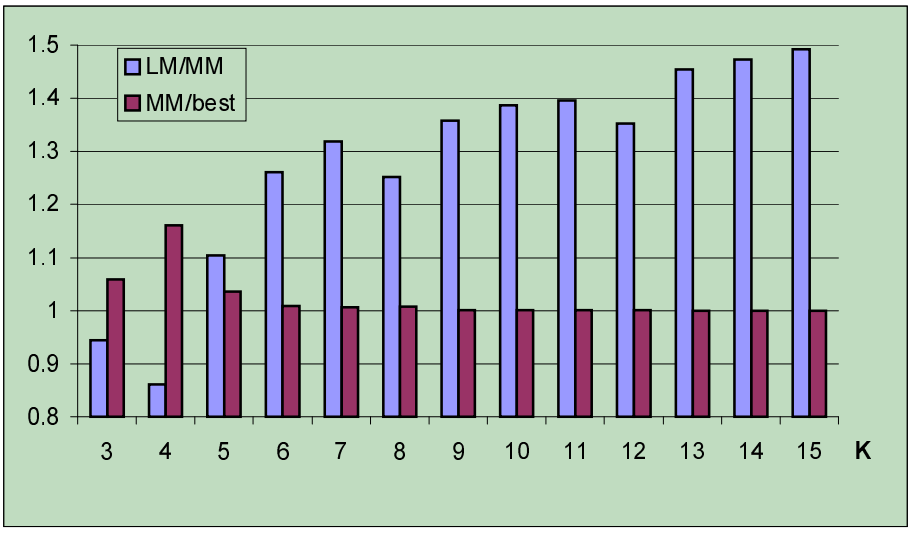
- Współczynnik mnożenia i dana wejściowa jest dodatnia – nie uzyskuje się żadnej optymalizacji.
- Współczynnik jest dodatni, a dana wejściowa jest zapisana w kodzie uzupełnień do dwóch. W tym przypadku tylko najbardziej znaczący blok pamięci operuje na liczbach ujemnych i dodatnich, pozostałe bloki tylko na liczbach dodatnich. W tym przypadku również nie można uzyskać żadnej optymalizacji.
- Ujemny współczynnik mnożenia i dodatnia dana wejściowa. Wszystkie pamięci LUT operują na danych ujemnych (w kodzie uzupełnień do dwóch - TC), dlatego dodawanie może być zastąpione odejmowaniem i dzięki temu wszystkie pamięci będą pracowały tylko na dodatnich liczbach (współczynnik mnożenia jest zanegowany, a przez to dodatni). W konsekwencji bit znaku nie będzie musiał być implementowany i wszystkie pamięci LUT będą o jeden bit węższe. Niestety podwójne odejmowanie ($s= -a - b$) nie może być zaimplementowane w układach FPGA i musi być odłożone do następnego poziomu dodającego ($s= -(a+b)$), co powoduje, że wynik dodawania jest zanegowany ($s= -a - b - c - ... = -(a + b + c + ...)$), a także na koniec konieczne jest dokonanie dodatkowej operacji negacji co jest raczej kosztowne. Dlatego najlepszym rozwiązaniem jest użycie odejmowania dla wszystkich pamięci LUT z wyjątkiem ostatniej najmniej znaczącej. Dzięki temu przełamany jest łańcuch podwójnego odejmowania. Wybór padł na LSB LUT ponieważ w tym wypadku możliwe jest kopiowanie bitów LSB przy przesunięciu bitowym dwóch argumentów jak to było opisane w podrozdziale 2.1.
- Ujemny współczynnik mnożenia i dana wejściowa w formacie uzupełnień do dwóch. W tym przypadku wszystkie pamięci oprócz MSB LUT operują na danych ujemnych i dlatego powinny być zaimplementowane tak jak w poprzednim zakresie. MSB LUT operuje jednak na danych w kodzie uzupełnień do dwóch dlatego jego wyjście może być dodawane do wyniku. Jednakże ta operacja dodawania może zakłócić etap bezpośredniego kopiowania LSB dla pozostałych operacji odejmowania. Dlatego korzystne wydaje się również dla tej

pamięci zastosowanie operacji odejmowania, czyli układ działa podobnie jak dla poprzedniego zakresu.

4. Porównanie układów mnożących

4. 1. Powierzchnia układu

W przedstawionych rozważaniach omówiono dwie różne techniki implementacji KCM: LM oraz MM. Dlatego powstaje pytanie, która z nich jest bardziej efektywna. Wykres na rysunku 9 przedstawia ich wzajemne zależności uwzględniając wartości średnie zajmowanej powierzchni. Analizując wykres na rysunku 9 można zauważyć, że dla XC4000 układ LM jest preferowany dla K≤4, a dla K≥5 najlepszy rezultat jest z reguły osiągnięty dla układu MM. Warto jednak zwrócić uwagę, że wybór pomiędzy układami LM i MM zależy od wybranego współczynnika mnożenia i wykres z rysunku 9 pokazuje tylko statystyczną relację pomiędzy tymi układami. Dlatego obie architektury powinny być przeanalizowane i lepsza z nich wybrana dla konkretnego przypadku. Jednakże, analizując stosunek MM/best można dojść do wniosku, że dla $K>5$ zysk z użycia układu LM jest minimalny i jego znaczenie spada wraz ze wzrostem K.



Rys. 9. Relacja pomiędzy stosunkiem średnich powierzchni układu XC4000 zajętych przez LM / MM oraz MM /best. Wynik dla różnej szerokości K danej wejściowej (zakres danej wejściowej $0 \div 2^K-1$) i współczynnika mnożenia (zakres: $1 \div 2^K-1$)

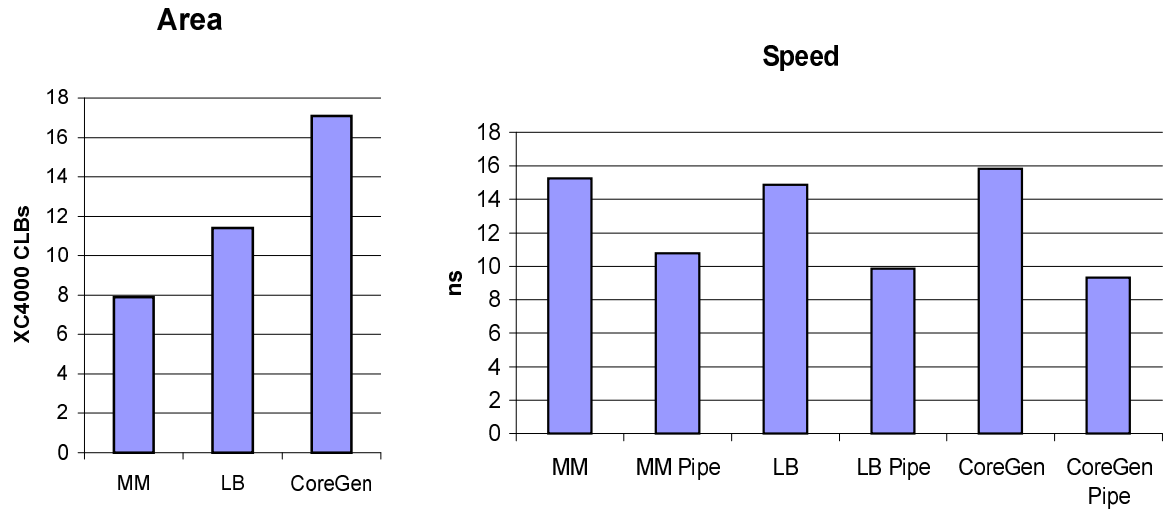
Z wykresu na rysunku 9 i Tabeli 2 można wysnuć bardziej ogólny wniosek: dla układu MM optymalizacja typu CSD i SS jest coraz bardziej skuteczna wraz ze wzrostem *K*.

Kolejnym aspektem wzajemnych porównań jest badanie relacji ilościowych jak dużą redukcję powierzchni można osiągnąć dzięki zastosowaniu technik optymalizacji MS i DAWR dla LM. Według wyników doświadczalnych uzyskuje się średnio zmniejszenie powierzchni układu o 5÷20% w zależności od szerokości wejścia *K*, przy czym optymalizacja ta jest bardziej skuteczna dla małych pamięci LUT.

4. 2. Częstotliwość pracy

W poprzednim rozdziale rozważana była tylko powierzchnia zajmowanego układu. W rzeczywistym układzie należy jednak rozważyć zarówno powierzchnię jak i częstotliwość pracy układu (w szczególności stosunek wartości powierzchni do częstotliwości pracy układu). W konsekwencji, aby zwiększyć przepustowość układu mnożącego należy zastosować architekturę potokową tym bardziej, że po każdym elemencie logicznym układy FPGA zawierają wbudowane rejestry typu D. Dlatego w pierwszym przybliżeniu zastosowanie architektury potokowej nie wymaga dodatkowej powierzchni układu FPGA. Jednakże, niektóre sygnały nie wymagają logiki, dlatego często rejestry potokowe muszą być użyte bez powiązanej logiki, zgodnie z zasadą tnij-wstaw (ang. *cut-set*) [7]. W konsekwencji dla całkowicie potokowej architektury (rejestry występują po każdej logice), powierzchnia zajmowana przez układ jest określona raczej przez liczbę rejestrów niż liczbę użytej logiki (układów dodających i pamięci). Dodatkowa powierzchnia zajmowana przez układ potokowy znajduje się w zakresie 0-50% i redukuje się do zera dla mniejszej liczby etapów potokowości gdy np. rejestry są implementowane co dwa poziomy logiki. Jednakże redukcja liczby etapów potokowych powoduje zmniejszenie częstotliwości taktowania układu. Z drugiej strony, architektura potokowa powoduje znaczne zwiększenie częstotliwości pracy układu i dlatego w większości przypadków można zaakceptować wzrost powierzchni układu potokowego, szczególnie wtedy gdy nie jest on duży. Warto podkreślić, że koszt rejestrów został również brany pod uwagę podczas poszukiwania optymalnej architektury układu mnożącego. Dla przykładu, technika optymalizacji SS ma

tendencję do implementacji dużej liczy rejestrów w przeciwieństwie do techniki CSD i dlatego metoda CSD jest skuteczniejsza dla architektury potokowej.



Rys. 10. Średnia zajmowana powierzchnia układu mnożącego (bez potokowości) i okres zegara z/bez potokowości dla MM, LB i Core Generator [15]. Wynik implementacji dla układu XC4000E-1, 8-bitowej danej bez znaku oraz losowo wybranym współczynnikom mnożenia: 41, 108, 132, 190, 225

Wykresy na rysunku 10 przedstawiają średnią powierzchnię i okres zegara dla układów MM i LM. Widać, że układy MM są bardziej efektywne niż LB i charakteryzują się tylko nieznacznym spadkiem częstotliwości pracy spowodowanej propagacją przeniesienia w dedykowanym układzie dodającym. Spadek szybkości w niektórych przypadkach może powodować, że przy porównywalnych powierzchniach układu bardziej optymalną architekturą jest układ LM. Wykres na rysunku 10 przedstawia również wynik implementacji dla komercyjnego programu Core Generator [15]. Uwidacznia on także, że rozwiązanie układów MM i LM opracowane w ramach referowanych w tym artykule badań przewyższa to rozwiązanie komercyjne.

5. Wnioski

W prezentowanych rozważaniach przedstawiono dwie różne architektury implementacji mnożenia: LM i MM. Wyniki doświadczalne pokazują, że dla małych szerokości wejścia i współczynnika mnożenia układy LM są z reguły lepsze, ale ze wzrostem tych szerokości układy MM stają się coraz bardziej atrakcyjne i przewyższają układy LM. Dzieje się tak dzięki większej efektywności optymalizacji CSD i SS.

Ponadto, podano ulepszony algorytm konwersji z reprezentacji binarnej do kodu CSD. Algorytm ten uwzględnia fakt, że koszt odejmowania jest wyższy od kosztu dodawania ponieważ dla odejmowania nie zachodzi kopiowanie najmniej znaczących bitów przesuniętego w prawo odjemnika. W konsekwencji odejmowanie (symbol $\bar{1}$ w kodzie CSD) jest implementowane tylko wtedy gdy dzięki temu zmniejszy się ogólna liczba operacji. Oprócz tego, dla architektury LM opisane zostało użycie różnych bloków pamięci, co wymaga użycia zaawansowanych technik i wyszukiwania optymalnego rozwiązania. Na zakończenie zostały podane relacje pomiędzy powierzchnią, a szybkością omawianych architektur. Warto w tym miejscu podkreślić, że osiągnięte w ramach prezentowanych badań rozwiązanie przewyższa rozwiązanie komercyjne.

Przedstawiono także różne metody i rezultaty implementacji dla układu mnożącego przez stały współczynnik. Warto podkreślić, że największy zakres pracy, który został wykonany w ramach prowadzonych badań, został włożony w opracowanie zautomatyzowanego oprogramowania narzędziowego AuToCon (ang. *Automated Tool for Convolution processor design*) do syntezy tych układów. Dzięki temu, bez udziału człowieka i bez jego rozległej wiedzy na temat budowy układów mnożących w układach FPGA, generowany jest kod VHDL układu mnożącego, który po syntezie może być implementowany do rzeczywistego układu FPGA. Zbudowane oprogramowanie wspomagające AuToCon nie zakłada jakichkolwiek wstępnych relacji kosztów i dlatego może być użyte dla dowolnego układu FPGA, a nawet układów typu ASIC. Parametrami wejściowymi programu AuToCon są: zakres danej wejściowej, wartość współczynnika mnożenia, koszt układu dodającego i rejestrów oraz rozmiar i koszt pamięci. W konsekwencji dzięki opracowanemu oprogramowaniu AuToCon, nie tylko zredukowany został koszt układu mnożącego przez stały współczynnik, ale również znacząco został skrócony czas realizacji projektu.

Bibliografia

1. S. Waser: *High-speed monolithic multipliers for real-time digital signal processing,* IEEE Computer Magazine, Vol. 11, No. 10, pp.19-29, 1978
2. C. S. Wallace: *A suggestion for a fast multiplier,* IEEE Trans. On Electron. Comput., Vol. EC-13, pp. 14-17, 1964
3. K. Chapman: *Constant Coefficient Multipliers for the XC4000E*. Xilinx Application Note, XAPP 054 December 1996
4. R. Petersen, B. L. Hutchings: *An Assessment of the Suitability of FPGA-Based Systems for Use in Digital Signal Processing*, In 5th International Workshop on Field Programmable Logic and Applications, Oxford England, pp. 293-302, August 1995
5. M. J. Wirthlin, B. L. Hutchings: *Improving Functional Density Through Run-Time Constant Propagation,* ACM/SIGDA International Symposium on Field Programmable Gate Arrays, pp. 86-92 (1997)
6. K. Chapman: *Fast Integer Multiplier fit in FPGA's*, EDN 1993 Design Idea Winner, END May 12th 1994
7. P. Pirsch: *Architectures for Digital Signal Processing*, Wiley 1998
8. Xilinx Co.: *Using the Dedicated Carry Logic in XC4000E,* Xilinx Application Note XAPP 013 July 4, 1996
9. H. Samueli: *An imroved search algorithm for the design of multiplierless FIR filters with power-of-two coefficients,* IEEE Transactions on Circuits and Systems, Vol. 36, pp. 1044-1047, July 1989
10. H. Garner: *Number Systems and Arithmetic,* Advances in Computing, vol. 6, pp. 131-194, 1965
11. R. I. Hartley: *Subexpression Sharing in Filters Using Canonic Signed Digit Multipliers,* IEEE Transactions on Circuits and Systems II – Analog and Digital Signal Processing, vol. 43, no. 10, Oct. 1996
12. A. R. Omondi: *Computer Arithmetic Systems. Algorithms Architecture and Implementations,* Prentice Hall 1994
13. Xilinx Co.: *The Programmable Logic Data Book* 1999

14. Altera Co.: *Apex 20K Programmable Logic Device Family, Data Sheet,* ver. 2.05, Nov. 1999
15. Xilinx Co.: *Core Generator* Foundation 2.1i Software Packet, 1999

K. Wiatr, E. Jamro

Constant Coefficient Multiplication in FPGA Structures

Summary

This paper investigates different architectures implementing bit-parallel constant coefficient multiplication in FPGA structures. At first the multiplierless multiplication (MM) architectures employing Canonic Sign Digit (CSD) and sub-structure sharing methods are addressed, and a novel algorithm for the conversion from two's complement to CSD representation is presented. In the second part of this paper the Look up table based Multiplication (LM) is investigated. Correspondingly, the usage of different memory modules and finding the optimal combination of the memory and adders are considered. The LM architecture considers also reduction of the address width for each memory cell and the possibility of memory sub-structure sharing (the search for the same memory cells is implemented). Finally the implementation results for Xilinx XC4000 and Virtex families are presented. As a result, the MM generally surpasses the LM architecture, however the actual choice between these two architectures is coefficient and input parameters dependent.

Keywords: dedicated circuits, specialised processors, programmable logic devices