AKADEMIA GÓRNICZO-HUTNICZA

Raport z projektu

Sprzętowa implementacja algorytmu Karatsuba

z przedmiotu

Systemy dedykowane w układach programowalnych

Elektronika i Telekomunikacja, Systemy Wbudowane, rok I

Marcin Maj, Jakub Zimnol

semestr letni 22/23, grupa poniedziałek 18:00

prowadzący: Sebastian Koryciak

19.06.2023

1. Podstawy teoretyczne

Algorytm Karatsuby to rekurencyjny algorytm szybkiego mnożenia typu divide-and-conquer. W algorytmie Karatsuby ilość potrzebnych do wykonania operacji mnożenia jest mniejsza niż w klasycznym algorytmie mnożenia. Jego złożoność obliczeniowa wynosi $O(n^{log_2 3}) \approx O(n^{1.58})$, podczas gdy złożoność obliczeniowa zwykłego mnożenia wynosi $O(n^2)$. Z racji rekurencyjności algorytm Karatsuby jest szybszy od klasycznego algorytmu tylko dla odpowiednio dużych liczb.

Oprócz szybkości, w kontekście FPGA zaletą algorytmu jest mniejsza ilość mnożeń, dzięki czemu wykorzystanie algorytmu redukuje zużycie bloków DSP.

Zasada działania algorytmu:

Aby pomnożyć dwie liczby n-cyfrowe (lub n-bitowe) x i y o podstawie B, należy je rozdzielić na dwie części:

$$x = x_1 B^m + x_2$$
$$y = y_1 B^m + y_2$$

gdzie n = 2m, oraz x_2 i y_2 są mniejsze niż B^m. Wynik mnożenia wynosi wtedy:

$$\begin{aligned} xy &= (x_1 B^m + x_2)(y_1 B^m + y_2) \\ &= x_1 y_1 B^{2m} + (x_1 y_2 + x_2 y_1) B^m + x_2 y_2 \\ &= z_2 B^{2m} + z_1 B^m + z_0 \end{aligned}$$

gdzie:

$$z_2 = x_1 y_1$$
$$z_1 = (x_1 y_2 + x_2 y_1)$$
$$z_0 = x_2 y_2$$

Standardową metodą byłoby pomnożenie czterech czynników osobno i dodanie ich po odpowiednim przesunięciu. Daje to algorytm o złożoności $O(n^2)$. Algorytm Karatsuby polega na policzeniu czynnika z_1 w następujący sposób:

$$z_1 = (x_1 y_2 + x_2 y_1) = [kilka\ przekształceń] = (x_1 + x_0)(y_1 + y_0) - z_2 - z_0$$

A więc ostatecznie:

$$\begin{aligned} xy &= x_1 y_1 B^{2m} + (x_1 y_2 + x_2 y_1) B^m + x_2 y_2 \\ &= x_1 y_1 B^{2m} + [(x_1 + x_0)(y_1 + y_0) - z_2 - z_0] B^m + x_2 y_2 \end{aligned}$$

Pozwala to na zredukowanie całkowitej ilości mnożeń z 4 do 3.

Pseudokod algorytmu (źródło: Karatsuba algorithm - Wikipedia):

```
function karatsuba(num1, num2)
    if (num1 < 10 or num2 < 10)
        return num1 × num2 /* fall back to traditional multiplication */

    /* Calculates the size of the numbers. */
    m = max(size_base10(num1), size_base10(num2))
    m2 = floor(m / 2) /* m2 = ceil (m / 2) will also work */

    /* Split the digit sequences in the middle. */
    high1, low1 = split_at(num1, m2)
    high2, low2 = split_at(num2, m2)

    /* 3 recursive calls made to numbers approximately half the size. */
    z0 = karatsuba(low1, low2)
    z1 = karatsuba(low1 + high1, low2 + high2)
    z2 = karatsuba(high1, high2)
    return (z2 × 10 ^ (m2 × 2)) + ((z1 − z2 − z0) × 10 ^ m2) + z0
```

2. Założenia projektu

Założeniem projektu jest stworzenie IP-core implementującego algorytm mnożenia „Karatsuba”, porównanie wyników syntezy dla klasycznego mnożenia i karatsuby oraz zaimplementowanie komponentu jako moduł współpracujący z procesorem ARM za pomocą interfejsu AXI-Lite.

Plan zakłada implementację mnożenia liczb:

- stałoprzecinkowych
- dodatnich
- ujemnych
- 64, 128, 256 bitowych

3. Opis funkcjonalny

Moduł Karatsuba komunikuje się z procesorem ARM za pomocą interfejsu AXI Slave poprzez protokół AXI-Lite. Zaimplementowane zostało mnożenie dodatnich liczb 64 bitowych stałoprzecinkowych.

4. Opis rejestrów

IP karatsuba zawiera 10 rejestrów, każdy rejestr jest 32 bitowy. Opis rejestrów przedstawiono w tabeli 1. Parametr BASEADDR zadeklarowany jest w pliku main.c, również parametry z offsetem dla poszczególnych rejestrów zostały zapisane w pliku main.c

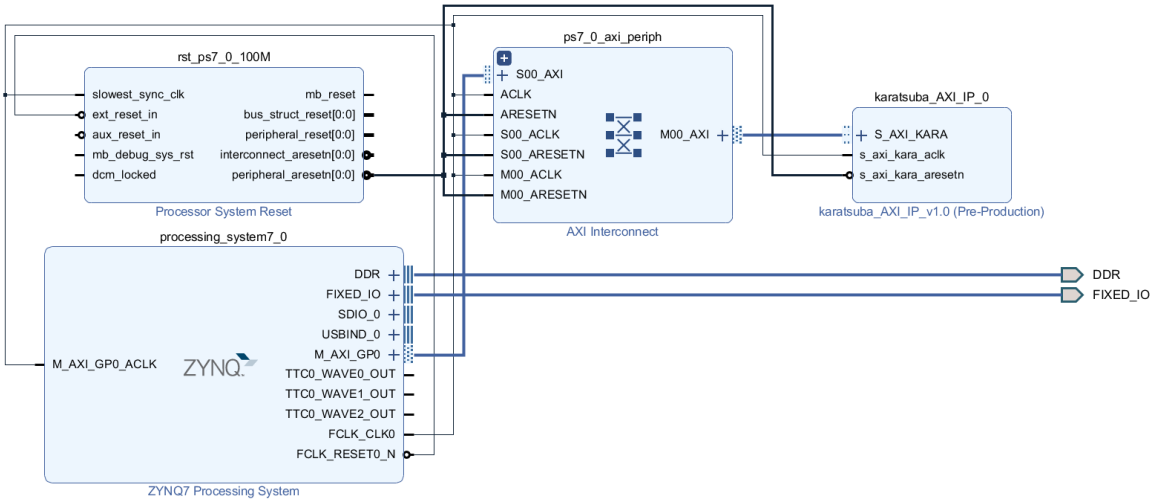
Tabela 1. Opis rejestrów IP karatsuba

Nazwa rejestru	Opis	Adres (adres bazowy + offset)	Nazwa parametru zawierającego wartość offsetu
slv_reg0	Rejestr do zapisu i odczytu, zawiera bity [31:0] pierwszego czynnika mnożenia	BASEADDR + 0x00	A_LOW
slv_reg1	Rejestr do zapisu i odczytu, zawiera bity [63:32] pierwszego czynnika mnożenia	BASEADDR + 0x04	A_HIGH
slv_reg2	Rejestr do zapisu i odczytu, zawiera bity [63:32] drugiego czynnika mnożenia	BASEADDR + 0x08	B_HIGH
slv_reg3	Rejestr do zapisu i odczytu, zawiera bity [31:0] drugiego czynnika mnożenia	BASEADDR + 0x0C	B_LOW
slv_reg4	Rejestr tylko do odczytu zawierający bity [31:0] wyniku	BASEADDR + 0x10	RESULT_0
slv_reg5	Rejestr tylko do odczytu zawierający bity [63:32] wyniku	BASEADDR + 0x14	RESULT_1
slv_reg6	Rejestr tylko do odczytu zawierający bity [95:64] wyniku	BASEADDR + 0x18	RESULT_2
slv_reg7	Rejestr tylko do odczytu zawierający bity [127:96] wyniku	BASEADDR + 0x1C	RESULT_3
slv_reg8	Rejestr tylko do odczytu, jest to wewnętrzny rejestr statusowy. W użyciu są bity [3:0], gdzie każdy z bitów sygnalizuje zapis do odpowiedniego rejestru slv_reg	BASEADDR + 0x20	REG8
slv_reg9	Rejestr tylko do odczytu, najmłodszy bit rejestru sygnalizuje zakończenie mnożenia i gotowość do odczytu	BASEADDR + 0x24	STATUS_REG

5. Schematy blokowe

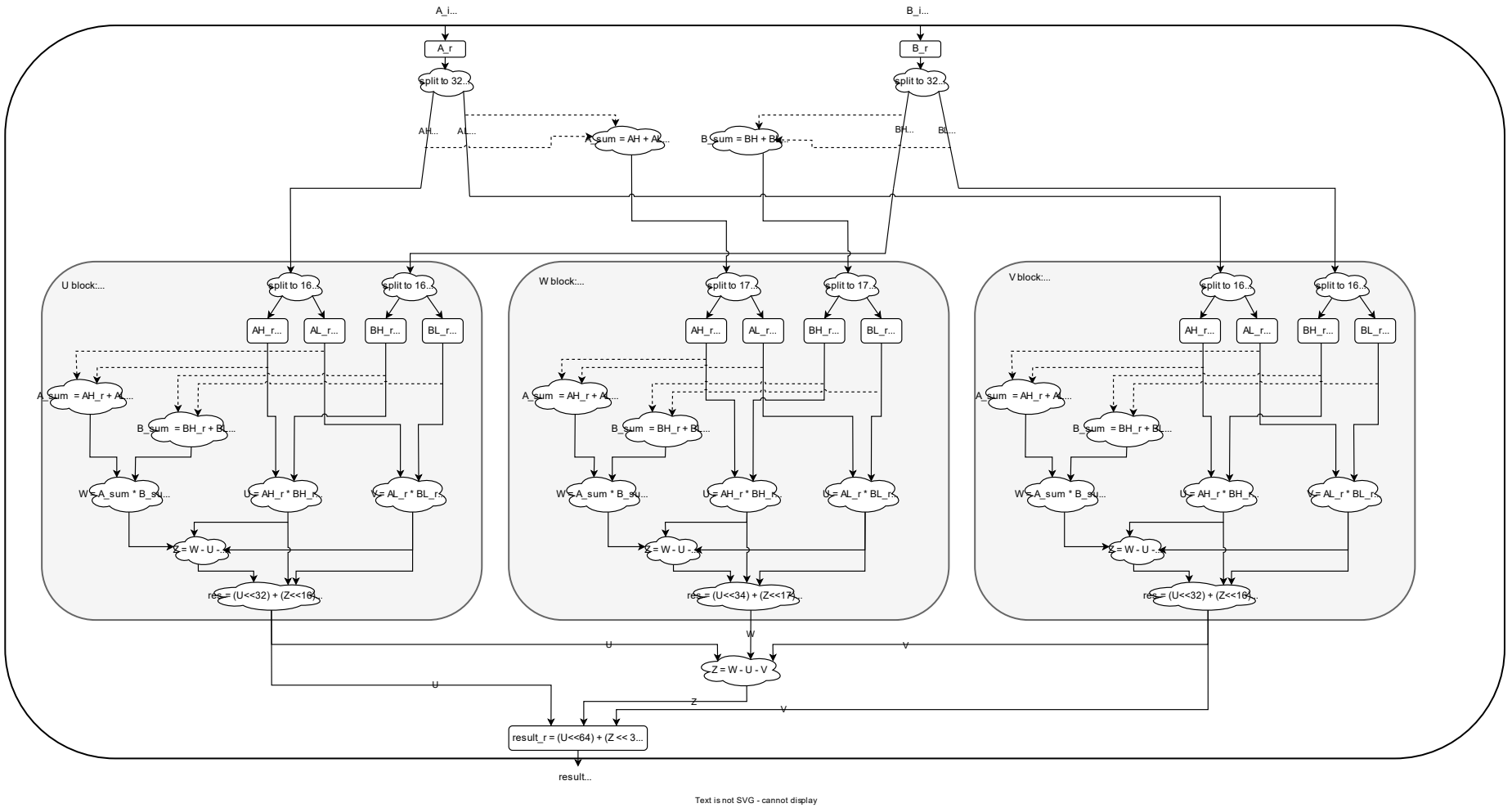
Poniżej znajdują się dwa schematy, pierwszy z nich obrazuje cały system zawierający procesor ARM, moduł Karatsuba oraz potrzebne komponenty. Drugi schemat przedstawia sposób implementacji algorytmu.

a) Schemat top-level



Rys 1. Schemat systemu

b) Schemat blokowy modułu karatsuba, link do obrazka w lepszej jakości: [maj77/karatsuba.svg]



Rys 2. Schemat bloku realizującego algorytm mnożenia „Karatsuba”

6. Weryfikacja

Do zweryfikowania poprawności działania modułu Karatsuba stworzony został testbench oraz skrypt generujący dane wejściowe dla algorytmu. Skrypt generuje dane wejściowe, które zapisuje do pliku, następnie testbench ładuje zawartość pliku do tablicy. Testbench przekazuje liczby z tablicy do modułu Karatsuba oraz oblicza iloczyny, które są porównywane z wynikami z modułu Karatsuba.

Przykładowe logi z symulacji:

```
----------------------------------------------------------------------
time: 5005.00 ns
CORRECT RESULT!
A=   9f27509da72b1362, B=   9479b4d2ea9bf50b       A_tb=9f27509da72b1362, B_tb=9479b4d2ea9bf50b
result : 39a33b1548f155a9fb0a45f195d38270
r_check: 39a33b1548f155a9fb0a45f195d38270

Results summary:
Correct:         249,
Incorrect:           0

----------------------------------------------------------------------
```

Rys 3. Log informujący o prawidłowym wyniku

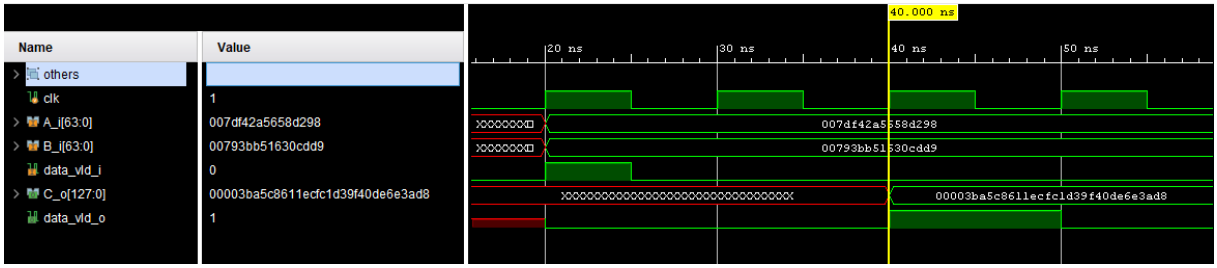
```
----------------------------------------------------------------------
time: 65.00 ns
INCORRECT RESULT!
A=   007df42a5658d298, B=   00793bb51630cdd9       A_tb=007df42a5658d298, B_tb=00793bb51630cdd9
result : 00003ba5c8611ecfc1d39f40de6e3ad8
r_check: 00003ba7c8611ecfc1d39f50de6e3ad8
diff   : 00000002000000000000001000000000

Results summary:
Correct:           0,
Incorrect:           2

----------------------------------------------------------------------
```

Rys 4. Log informujący o nieprawidłowym wyniku

Obliczenie wyniku zajmuje 2 cykle zegara:



Rys 5. Przebiegi otrzymane podczas testowania modułu Karatsuba

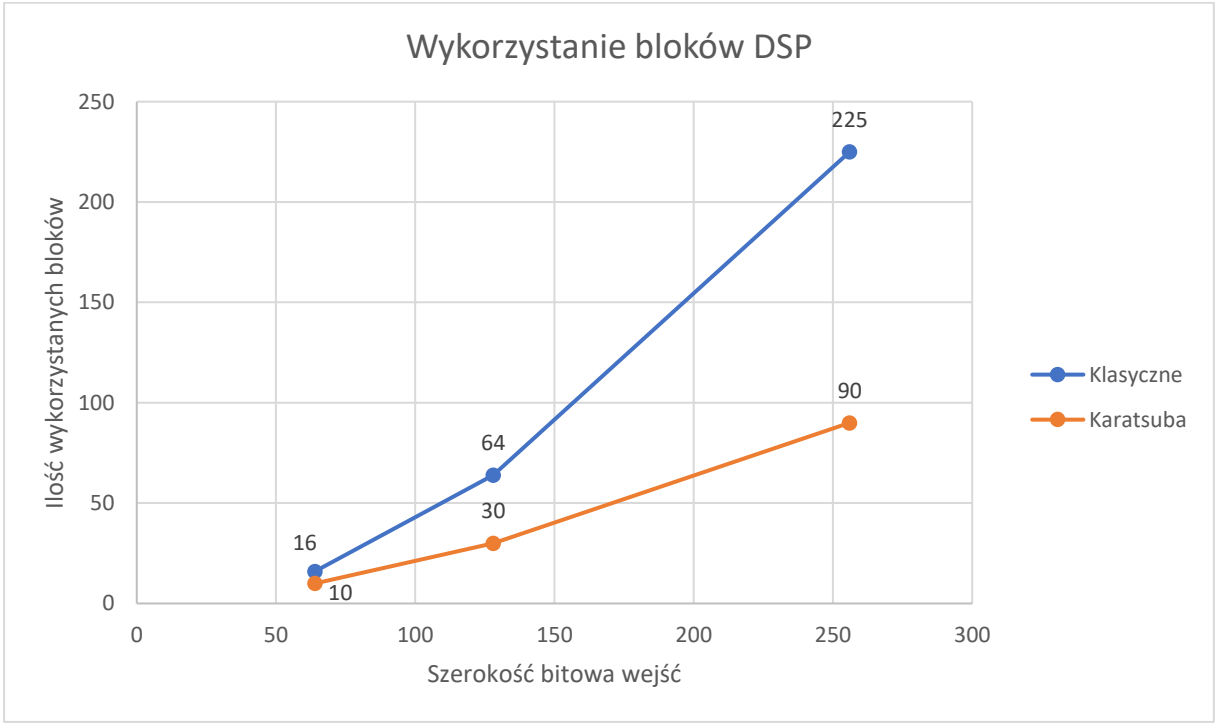
7. Wyniki syntezy

W celu porównania zajętości zasobów przeprowadzono syntezy dla samego modułu Karatsuba oraz dla modułów wykonujących klasyczne mnożenie. Najbardziej interesuje nas ilość wykorzystanych bloków DSP. Pełne raporty z syntezy dla wersji 64 bit (klasyczna i Karatsuba) znajduje się na końcu raportu.

Tabela 2. Zajętość bloków DSP

Typ modułu	Szerokość bitowa wejść	Ilość użytych bloków DSP
Karatsuba	64	10
Karatsuba	128	30[1)]
Karatsuba	256	90[1)]
Klasyczny	64	16
Klasyczny	128	64
Klasyczny	256	225

[1)] jest to przybliżona wartość obliczona na podstawie ilości wykorzystanych bloków DSP dla modułu mnożącego liczby 64 bitowe, w praktyce może różnić się o kilka bloków, prawdopodobnie 5-15 bloków więcej, w zależności od wersji (128/256bit).



Rys 6. Porównanie wykorzystania bloków DSP

8. Podsumowanie

Aktualna wersja projektu posiada zaimplementowany algorytm Karatsuby pozwalający na mnożenie 64 bitowych dodatnich liczb stałoprzecinkowych. Podstawową zaletą algorytmu jest zmniejszone zapotrzebowanie na bloki DSP, w przypadku liczb 64 bitowych różnica wynosi tylko 4. Różnica ta może być zadowalająca w przypadku gdy układ FPGA ma bardzo małą ilość bloków DSP i każdy blok jest „na wagę złota". Algorytm może zastąpić klasyczne mnożenie kiedy wystąpi konieczna potrzeba zredukowania ilości wykorzystanych bloków DSP.

W przypadku mnożenia liczb 128 i 256 bitowych róznica między klasycznym mnożeniem a Karatsubą jest już zadowalająca i wynosi ona odpowiednio ~30 i ~120 bloków DSP.

Podsumowując: W przypadku gdy w projekcie wystąpiłaby potrzeba mnożenia liczb 64 bitowych zdecydowalibyśmy się na zaimplementowanie 64 bitowej wersji algosrytmu Karatsuba tylko w ostateczności ponieważ zysk jest niewielki. W przypadku gdy w projekcie wystąpiłaby potrzeba mnożenia liczb 256 bitowych rozważylibyśmy implementację algorytmu Karatsuby w pierwszej kolejności ponieważ różnica 120 bloków DSP jest znaczna, również w kontekście poboru mocy przez układ.

Możliwości rozwoju projektu:

- Implementacja mnożenia liczb ujemnych
- Implementacja mnożenia liczb 128 i 256 bitowych
- Analiza algorytmu w kontekście czasu wykonywania operacji

Dodatek – Wyniki syntezy algorytmu Karatsuba oraz klasycznego mnożenia

a) Karatsuba

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	538	0	17600	3.06
LUT as Logic	537	0	17600	3.05
LUT as Memory	1	0	6000	0.02
LUT as Distributed RAM	0	0		
LUT as Shift Register	1	0		
Slice Registers	451	0	35200	1.28
Register as Flip Flop	451	0	35200	1.28
Register as Latch	0	0	35200	0.00
F7 Muxes	0	0	8800	0.00
F8 Muxes	0	0	4400	0.00

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	10	0	80	12.50
DSP48E1 only	10			

7. Primitives

Ref Name	Used	Functional Category
FDRE	451	Flop & Latch
LUT2	326	LUT
LUT3	162	LUT
CARRY4	156	CarryLogic
IBUF	130	IO
OBUF	129	IO
LUT4	118	LUT
LUT6	44	LUT
DSP48E1	10	Block Arithmetic
SRL16E	1	Distributed Memory
BUFG	1	Clock

b) Klasyczne mnożenie

1. Slice Logic

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	176	0	17600	1.00
LUT as Logic	176	0	17600	1.00
LUT as Memory	0	0	6000	0.00
Slice Registers	102	0	35200	0.29
Register as Flip Flop	102	0	35200	0.29
Register as Latch	0	0	35200	0.00
F7 Muxes	0	0	8800	0.00
F8 Muxes	0	0	4400	0.00

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	16	0	80	20.00
DSP48E1 only	16			

7. Primitives

Ref Name	Used	Functional Category
IBUF	129	IO
OBUF	128	IO
FDRE	102	Flop & Latch
LUT6	60	LUT
LUT2	47	LUT
LUT4	36	LUT
LUT5	35	LUT
LUT3	31	LUT
CARRY4	24	CarryLogic
DSP48E1	16	Block Arithmetic
BUFG	1	Clock