Ernest JAMRO [1,2], Kazimierz WIATR [1,2]
[1] AGH - AKADEMIA GÓRNICZO-HUTNICZA, KATEDRA ELEKTRONIKI, Al. Mickiewicza 30, 30-059 Kraków
[2] ACK CYFRONET AGH , ul. Nawojki 11, 30-950 Kraków

Implementacja w układach FPGA dekompresji danych zgodnie ze standardem Deflate

Dr inż. Ernest JAMRO

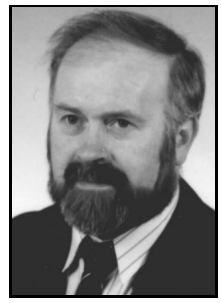
Ukończył studia na AGH na kierunku Elektronika oraz na University of Huddersfield (UK) na kierunku Elektronika i Telekomunikacja. Obronił pracę doktorską w 2001 roku na AGH na wydziale Elektrotechniki, Automatyki, Informatyki i Elektroniki. Aktualnie jest adiunktem w Katedrze Elektroniki na AGH. Jego zainteresowania naukowe to sprzętowa akceleracja obliczeń, niskopoziomowe przetwarzanie obrazów, sieci neuronowe.

e-mail: jamro@agh.edu.pl

Prof. dr hab. inż. Kazimierz WIATR

Studia AGH Kraków (1980), dr nauk technicznych (1987), dr habilitowany (1999) i profesor (2002). Profesor zwyczajny na Akademii Górniczo-Hutniczej oraz Dyrektor Akademickiego Centrum Komputerowego Cyfronet AGH. Prowadzone prace badawcze dotyczą komputerowego sterowania procesami, systemów wizyjnych, systemów wieloprocesorowych, układów programowalnych, rekonfigurowalnych systemów obliczeniowych i sprzętowych metod akceleracji obliczeń.

e-mail: wiatr@agh.edu.pl



Streszczenie

Otwarty standard kompresji danych, Deflate, jest szeroko stosowanym standardem w plikach .gz / .zip i stanowi kombinację kompresji metodą LZ77 / LZSS oraz kodowania Huffmana. Niniejszy artykuł opisuje implementację w układach FPGA dekompresji danych według tego standardu. Niniejszy moduł jest w stanie dokonać dekompresji co najmniej 1B na takt zegara, co przy zegarze 100MHz daje 100MB/s. Aby zwiększyć szybkość, możliwa jest praca wielu równoległych modułów dla różnych strumieni danych wejściowych.

Słowa kluczowe: kompresja danych, FPGA, kodowanie Huffmana.

FPGA Implementation of Deflate standard data decompression

Abstract

This paper describes FPGA implementation of the Deflate standard decoder. Deflate [1] is a commonly used compression standard employed e.g. in zip and gz files. It is based on dictionary compression (LZ77 / LZSS) [4] and Huffman coding [5]. The proposed Huffman decoded is similar to [9], nevertheless several improvements are proposed. Instead of employing barrel shifter a different translation function is proposed (see Tab. 1). This is a very important modification as the barrel shifter is a part of the time-critical feedback loop (see Fig. 1). Besides, the Deflate standard specifies extra bits, which causes that a single input word might be up to 15+13=28 bits wide, but this width is very rare. Consequently, as the input buffer might not feed the decoder width such wide input date, a conditional decoding is proposed, for which the validity of the input data is checked after decoding the input symbol, thus when the actual input symbol bit widths is known. The implementation results (Tab. 2) show that the occupied hardware resources are mostly defined by the number of BRAM modules, which are mostly required by the 32kB dictionary memory. For example, comparable logic (LUT / FF) resources to the Deflate standard decoder are required by the AXI DMA module which transfers data to / from the decoder.

Keywords: data compression, Deflate, Huffman, FPGA.

1. Wstęp

Otwarty standard kompresji danych Deflate [1] jest szeroko stosowanym standardem kompresji danych np. w plikach .gz oraz .zip. Niniejszy standard powstał w roku 1996, w dniu dzisiejszym istnieją lepsze standardy kompresji np. bzip2, niemniej niniejszy standard wydaje się być optymalny jeśli chodzi o stosunek współczynnika kompresji do szybkości kompresji / dekompresji oraz wymaganych zasobów sprzętowych. Algorytm Deflate stanowi kombinację dwóch metod kompresji: kodowania słownikowego LZ77 / LZSS [2, 3, 4] oraz kodowania Huffmana [4, 5]. Kodowanie słownikowe polega na wyszukiwaniu w poprzednio zakodowanych danych (czyli słowniku) takiego samego ciągu danych o długości co najmniej 3 znaki (bajty). Na przykład, ciąg wejściowy: *ABCDABCF*, można zakodować jako ciąg wyjściowy: *ABCD(l=3, d=4)F*, gdzie *l* oznacza długość znalezionego ciągu a *d* oznacza dystans (odległość) znalezionego ciągu w słowniku. Warto podkreślić, że według standardu Deflate, kodowanie słownikowe umożliwia również kodowanie długości serii - Run Length Encoding (RLE). Na przykład ciąg wejściowy *ABCABCABCA*, może być zakodowany jako: *ABC(l=7, d=3)*. Kodowanie RLE jest użyte dla *l>d*.

Kodowanie słownikowe zastosowane w standardzie Deflate jest bliższe kodowaniu LZSS niż kodowaniu LZ77 i jest skojarzone z kodowaniem Huffmana. W porównaniu z kodowaniem LZSS, w algorytmie Deflate nie jest tworzony dodatkowy bit świadczący o tym, czy kodowany jest standardowy znak (literał) czy też kodowana jest para (*l, d*). Zamiast tego alfabet kodowanych literałów (zwykle znaki od 0 do 255) jest zwiększany o dodatkowe znaki takie jak: 256- End of Block czyli znak końca bloku; 257- długość znalezionego ciągu $l = 3$; 258 – $l = 4$; itd. Aby ograniczyć liczbę dodatkowych symboli, dla długości znalezionego ciągu większej niż 10, stosuje się dodatkowe bity *extra,* które nie podlegają kodowaniu Huffmana. Na przykład, *l=12* jest kodowane jako symbol *266* i jeden bit *extra*=1. Liczba bitów *extra* jest zwiększana dla coraz to większej długości, i tak dla długości *l=256* wynosi 5. Maksymalna liczba symboli wynosi 285 i odpowiada maksymalnej długości ciągu *l*=258. Po każdym symbolu o wartości większej niż 256 kodowany jest dystans *d.* Podobnie jak dla długości kopiowanego ciągu, jest on kodowany w postaci odpowiedniego kodu (od 0 do 29) oraz dodatkowych bitów *extra.*

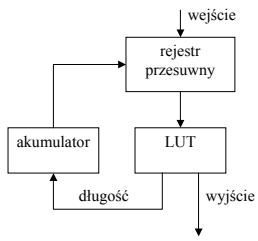
Po kompresji słownikowej użyta jest kompresja danych metodą Huffmana, która polega na przyporządkowaniu każdemu symbolowi (literał/długość lub niezależna książka kodowa dla dystansu) kodu o różnej długości bitowej, w zależności od jego prawdopodobieństwa wystąpienia. Symbole występujące często mają krótsze słowa kodowe, a występujące rzadko mają dłuższe słowa kodowe. Dzięki temu sumaryczna liczba bitów konieczna do zapisania ciągu znaku ulega zmniejszeniu. W dekoderze (nazywanym również jako inflate – w języku angielskim przeciwieństwo słowa deflate) kolejność dekodowania jest odwrotna, najpierw następuje dekodowanie Huffmana a następnie dekodowanie słownikowe.

Algorytm Deflate jest niezmiernie popularny, już twórca tego standardu udostępnił odpowiedni program napisany w języku C, umożliwiający kompresję i dekompresję danych. Istnieje wiele implementacji sprzętowych niniejszego standardu, np. AHA gzip compressor / decompressor [6], niemniej niniejszy moduł może obsługiwać tylko statyczne kodowanie Huffmana. Innym rozwiązaniem jest produkt firmy Indra [7], która z użyciem układu FPGA jest w stanie dokonać kompresji i dekompresji na poziomie 110MB/s. Pomimo tego, że na rynku istnieją podobne produkty, podjęto decyzję o projektowaniu własnego ze względu na jego znaczenie oraz możliwość uniezależnienia się od innych firm. W momencie, kiedy o szybkości przetwarzania danych w układach FPGA często decyduje nie sama szybkość przetwarzania tych danych, ale szybkość dostarczania danych do układu FPGA, szybka dekompresja danych jest kluczowa. Przykładem może być przeszukiwanie tekstu [8], którego szybkość jest ograniczona do szybkości

dostarczanych danych. Dlatego poprzez zastosowanie dekompresji danych w jednym układzie FPGA z przeszukiwaniem tekstu istnieje możliwość znaczącego przyśpieszenia samego przetwarzania, niemniej wymaga to dalszych prac badawczych. Warto podkreślić, że oprócz zwiększenia szybkości przetwarzania istnieje możliwość zmniejszenia powierzchni zajmowanej przez dane np. na zasobach dyskowych. Ma to duże znaczenie szczególnie dla szybkiego przetwarzania dużej liczy danych (a z taką liczbą mamy do czynienia w przypadku projektu SYNAT) zgromadzonych na bardzo szybkich dyskach SSD, które są bardzo szybkie niemniej mają stosunkowo wysoki stosunek ceny do pojemności.

2. Dekoder kodu Huffmana

Największym wyzwaniem niniejszej pracy był projekt dekodera dynamicznego kodu Huffmana. Dekodowaniem nagłówka Deflate zajmuje się procesor, np. MicroBalaze, x86. W standardzie Deflate, książka kodowa jest najpierw kodowana kodem RLE (Run Length Encoding) a następnie kodem Huffmana. Niemniej rozmiar książki kodowej jest na tyle mały, a algorytm na tyle rozbudowany, że w realizacji tej funkcji dobrze sprawdza się procesor. Moduł sprzętowy natomiast dostaje już gotową książkę kodową wraz z długością kodu. Architektura sprzętowa dekodera została opisana w [9]. Zmiana książki kodowej dekodera dokonuje się poprzez odpowiednie zaprogramowanie pamięci LUT, która posiada dwa wyjścia danych: wyjście zdekodowanej danej oraz długość (w bitach) wejściowego kodu. Głównym problemem dekodera jest to, że szerokość bitowa wejściowych słów kodowych jest zmienna, co powoduje, że na wejściu LUT konieczne jest stosowanie skomplikowanego rejestru przesuwnego. Warto w tym miejscu podkreślić, że dekodowanie jest procesem sekwencyjnym, to znaczy aby zdekodować następny symbol wejściowy konieczna jest znajomość, gdzie on się zaczyna, a miejsce to z kolei jest znane dopiero po zdekodowaniu aktualnego symbolu.



Rys. 1. Uproszczona architektura dekodera Huffmana
Fig. 1. Simplified architecture of the Huffman decoder

Następnym problemem podczas implementacji dekodera jest wielkość pamięci LUT, a dokładnie szerokość magistrali adresowej. W przypadku kodera [10] wynosi ona tyle bitów na ilu zapisywany jest pojedynczy symbol, czyli 8 ewentualnie 9 bitów w przypadku standardu Deflate. W przypadku dekodera, szerokość magistrali adresowej powinna wynosić tyle ile wynosi maksymalna szerokość zakodowanego symbolu, czyli w przypadku standardu Deflate – 15 bitów. W konsekwencji wymagana jest wielkość tej pamięci na poziomie 64kB (2^{15}×16-bitów). Ponadto podczas programowania pamięci LUT, wymaganych jest 2^{15} wpisów (taktów zegara), co w przypadku dekompresji małych plików zwielokrotniałoby czas dekompresji.

Rozwiązanie tego problemu, wykorzystuje fakt, że dla kodowania kanonicznego Huffmana [4], kody krótsze otrzymują przedrostek w postaci bitu 0, a kody dłuższe – przedrostek 1. Dlatego kody rozpoczynające się od bitu '0' mogą mieć długość maksymalną 8-bitów dla liczby symboli nie przekraczającej 256+8=264. Liczba ta jest otrzymana dla najgorszego przypadku, dla którego symbole rozpoczynające się od bitu '0' generują najwyższe drzewo (np. dla ciągu Fibonacciego – drzewo to zawiera tylko 8 symboli dla wysokości 8, zob. [4]). Z kolei symbole rozpoczynające się na '1' generują najniższe drzewo – wszystkie symbole mają 9-bitów długości, czyli jest ich 256. W standardzie Deflate maksymalna liczba symboli wynosi 285. Dlatego może się zdarzyć, że układ ten nie będzie działał poprawnie, niemniej najgorszy przypadek jest bardzo mało prawdopodobny i właściwie nie występuje w praktyce (nie zaobserwowano go do tej pory).

W konsekwencji poprzez dekodowanie wiodących jedynek możliwe jest ograniczenie wielkości pamięci do 2^7 wpisów dla określonej liczby wiodących jedynek. Pokazuje to Tab. 1 oraz kolumna *adres 1*, dla której bity wejściowe: *ab..o* są odpowiednio przesuwane w rejestrze przesuwnym zgodnie z liczbą wiodących jedynek [9]. Na podstawie tab. 1, można zauważyć, że liczba potrzebnych pamięci o 128 wpisach wynosi 9, w konsekwencji wymagana wielkość całkowitej pamięci wynosi 9×128= 1152 lokacje adresowe, które można zakodować na 11-bitach adresowych.

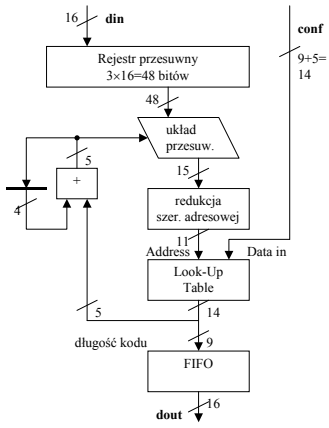
Tab. 1. Mapowanie bitów podczas skracania szerokości adresowej pamięci LUT
Tab. 1. Bit mapping for shortening the address width of the Look-Up Table

Dł	wejście	adres 1	adres 2
≤8	0bcd efgh	0000 bcdefgh	0000 bcdefgh
≤9	10cd efgh i	0001 cdefghi	0001 icdefgh
≤10	110d efgh ij	0010 defghij	0011 ijdefgh
≤11	1110 efgh ijk	0011 efghijk	0010 ijkefgh
≤12	1111 0fgh ijkl	0100 fghijkl	0110 ijklfgh
≤13	1111 10gh ijkl m	0101 ghijklm	0111 ijklmgh
≤14	1111 110h ijkl mn	0110 hijklmn	0101 ijklmnh
≤15	1111 1110 ijkl mno	0111 ijklmno	0100 ijklmno
≤15	1111 1111 ijkl mno	1000 ijklmno	1100 ijklmno

W tab. 1 w kolumnie *adres 2* zaproponowano autorską metodę optymalizacji algorytmu skracania szerokości bitowej magistrali adresowej pamięci LUT. Otóż zamiast użycia rejestru przesuwnego [9], zaproponowano zastosowanie dowolnego innego jednoznacznego kodowania, takiego aby liczba bitów wejściowych doprowadzonych do określonego wejścia adresowego była jak najmniejsza. Na przykład, w oryginalnym algorytmie z użyciem rejestru przesuwnego, bit najmłodszy adresu jest wybierany z wejść: *hijklmno*, co wymaga użycia multipleksera 8:1. W zaproponowanym algorytmie najmłodszy bit adresu jest wybierany tylko spośród dwóch bitów: *h* lub *o*, co wymaga zdecydowanie mniejszych zasobów logicznych. Podobnie kodowane są 4 najstarsze bity adresowe, które określają, która pamięć o 2^7 lokacjach jest aktualnie użyta. Bity te są uzależnione od liczby wiodących jedynek, i są kodowane w kodzie Greya zamiast w kodzie binarnym. Kod Greya zapewnia minimalną możliwą liczbę zmian, przez co logika jest uproszczona. Warto podkreślić, że logika redukcji szerokości adresowej znajduje się w krytycznej czasowo ścieżce sprzężenia zwrotnego, w ramach której nie jest możliwe zastosowanie architektury potokowej. W konsekwencji uproszczenie tej logiki nie tylko powoduje zmniejszenie zajmowanych zasobów sprzętowych, ale przede wszystkim umożliwia zwiększenie maksymalnej częstotliwości działania dekodera.

Schemat blokowy dekodera Huffmana został przedstawiony na rys. 2. Szczególną trudność przy projektowaniu niniejszego modułu sprawił wejściowy rejestr przesuwny oraz krytyczna czasowo pętla sprzężenia zwrotnego składająca się z pamięci LUT (pamięć BRAM), akumulatora, układu przesuwnego oraz układu redukcji szerokości magistrali adresowej. Magistrala wejściowa danych *din* ma szerokość 16-bitów i w wejściowym rejestrze przesuwnym mogą znajdować się aż 3 dane 16-bitowe, które są przesuwane w prawo z granulacją 16-bitów. W konsekwencji maksymalnie 15-bitów po prawej stronie może być nieaktualnych. Pamięć BRAM jest pamięcią synchroniczną, czyli dane wyjściowe są opóźnione o jeden takt zegara w stosunku do magistrali adresowej, co zdecydowanie komplikuje projekt układu. Na przykład, początek dekodowanego słowa może znajdować się na bicie zerowym rejestru przesuwnego ale również na bicie 15 + 15 + 13 = (maksymalna liczba nieaktualnych bitów rejestru przesuwnego) + (maksymalna długość kodu Huffmana) + (maksymalna liczba bitów ekstra dla dystansu) = 43. Warto podkreślić, że niniejszy przypadek jest mało prawdopodobny – symbol o długości 15 występuje bardzo rzadko skoro jest kodowany na tak dużej liczbie bitów. Dlatego, w takim wypadku zostanie wstawiony dodatkowy takt oczekiwania, tak aby rozpoczęcie dekodowania rozpoczęło się co najwyżej od bitu 31 oraz zakończyło co najwyżej na bicie 47.

W konsekwencji układ przesuwny to barrel shifter umożliwiający przesunięcie od 0 do 31 bitów. Ograniczenie to zmniejsza nie tylko liczbę zajmowanych zasobów ale również zmniejsza opóźnienie w krytycznej czasowo pętli sprzężenia zwrotnego.



Rys. 2. Schemat blokowy dekodera Huffmana
Fig. 2. Block diagram of the Huffman decoder

Dodatkowym problemem jest znalezienie miejsca zakończenia dekodowania danego symbolu. Ponieważ długość danego kodu jest znana dopiero po jego zdekodowaniu, nie jest możliwe określenie czy rejestr przesuwny posiada wszystkie konieczne do zdekodowania dane. Warto podkreślić, że maksymalna długość pojedynczego symbolu wynosi 15 bitów + 13 bitów extra = 28 bitów. Wartość ta występuje jednak niezmiernie rzadko i dlatego układ nie jest optymalizowany na jej wystąpienie. W konsekwencji proces dekodowania jest inicjalizowany nawet wtedy, kiedy nie wszystkie bity rejestru przesuwnego są ważne, lub też kiedy maksymalne wejściowe słowo kodowe wykracza poza 48-bitowy rejestr przesuwny. W takim wypadku dopiero po zdekodowaniu danego słowa sprawdzane jest czy dana wejściowa była ważna. Problem polega na tym, że długość zdekodowanego symbolu może być przekłamana w momencie, kiedy dana wejściowa była niepoprawna. Aby temu zapobiec, nieważna dana wejściowa jest wypełniana jedynkami. Dla kodowania kanonicznego Huffmana, symbole dłuższe przyjmują bity '1' a krótsze '0' [4]. W konsekwencji wypełnienie jedynkami nieważnych bitów wejściowych gwarantuje, że zostanie zdekodowana największa możliwa długość symbolu, czyli gwarantuje poprawność działania układu.

3. Wyniki Implementacji

Wyniki implementacji dla układu FPGA Xilinx Spartan 6: XC6SLX45CSG324-3 zostały przedstawione w tab. 2. Cały moduł dekompresji (moduł *inflate*) składa się z modułu dekodera Huffmana oraz dekompresji słownikowej. W skład modułu Huffmana wchodzą między innymi moduł *length_dec* służący do konwersji danych po dekodowaniu Huffmana na literał / długość / dystans zgodnie ze standardem Deflate. W skład modułu *Huffman* wchodzi również dodatkowy bufor FIFO (zajmujący głównie 25 LUTRAM) znajdujący się pomiędzy dekoderem Huffmana a słownikiem. Dzięki temu buforowi, dekoder Huffmana może dekodować następne symbole w momencie, kiedy moduł słownika jest zajęty kopiowaniem danych ze słownika na wyjście modułu. Ponadto bufor FIFO jest 16-bitowy co umożliwia kopiowanie 2B (literałów) na takt zegara, w konsekwencji teoretyczna maksymalna szybkość pracy modułu *inflate* jest prawie 2B na takt zegara. W skład dekodera Huffmana wchodzi również moduł *config* służący do konfiguracji pamięci LUT. Moduł ten na podstawie wartości słowa kodowego wejściowego, symbolu wyjściowego oraz długości słowa odpowiednio zapisuje pamięć LUT dekodera Huffmana. Moduł dekompresji słownikowej, w pierwszym przybliżeniu wystawia na wyjście albo dane wejściowe po dekodowaniu Huffmana albo też kopiuje dane znajdujące się w słowniku, czyli w pamięci BRAM. Niestety głównymi zasobami sprzętowymi zajmowanymi przez moduł dekompresji słownikowej jak i całego modułu *inflate* są pamięci BRAM. Pamięć słownika zgodnie ze standardem Deflate musi wynosić 32kB. Oczywiście możliwe jest zmniejszenie tej pamięci, pod warunkiem, że podczas kodowania również użyto mniejszej pamięci słownika, ale w tym wypadku nie będzie możliwa dekompresja dowolnego pliku zgodnego ze standardem Deflate.

Tab. 2. Wyniki implementacji
Tab. 2. Implementation results

Moduł	# FF	# LUT	#LUTRAM	# BRAM
Inflate	288	784	41	18
• Huffman	167	560	25	2
• length_dec	51	183	0	0
• config	37	56	0	0
• Słownik	118	221	16	16
axis_dmas	711	863	113	0
• axi_master [11]	567	464	5	0
MicroBlaze	990	1350	113	0
System	2661	3940	284	42

Moduł *inflate* jest kompatybilny ze standardem magistrali AXI Stream. Aby działał on poprawnie potrzebny jest dodatkowy moduł DMA dostarczający dane wejściowe (np. z pamięci) i odbierający dane po dekompresji. W niniejszym systemie zastosowano autorski moduł *axis_dmas*, który jest modułem typu master na magistrali AXI. Analizując tab. 2, można zauważyć, że moduł *axis_dmas* wymaga nieznacznie większych zasobów logicznych niż moduł *inflate,* przy czym około połowę zasobów modułu *axis_dmas* zajmuje interfejs typu master magistrali AXI [11]. Dodatkowo podano zasoby zajmowane przez soft-procesor MicroBlaze oraz cały system zbudowany w pakiecie Xilinx EDK.

4. Wnioski

Niniejszy artykuł opisuje implementacje w układach FPGA dekompresji danych według standardu Deflate. Wyniki implementacji pokazują, że niniejszy moduł zajmuje głównie zasoby pamięciowe BRAM, potrzebne do dekompresji słownikowej. Ponadto moduł DMA dostarczający i odbierający dane wejściowe zajmuje nieznacznie większe zasoby logiczne niż opisywany moduł *inflate*. Niniejszy moduł pracował poprawnie z częstotliwością 100MHz, czyli minimalna szybkość dekompresji wynosi 100MB/s. Istnieje możliwość równoległej pracy wielu takich modułów dla niezależnych strumieni.

5. Literatura

[1] Deutsch P.: DEFLATE Compressed Data Format Specification version 1.3: RFC1951, 05/1996.
[2] Ziv, Jacob; Lempel, Abraham (May 1977). "A Universal Algorithm for Sequential Data Compression". IEEE Transactions on Information Theory 23 (3): 337-343.
[3] Lempel–Ziv–Storer–Szymanski (LZSS) http://en.wikipedia.org/
[4] Salomon D.: A Concise Introduction to Data Compression, Springer, 2008.
[5] Huffman D.A.: A method for the construction of minimum-redundancy codes, Proc. Inst. Radio Eng, Vol.40, No.9, pp.1098-1101, Sep. 1952.
[6] AHA, www.aha.com
[7] Indra, http://www.indranetworks.com
[8] Jamro E., Russek P., Dąbrowska-Boruch A., Wielgosz M., Wiatr K.: The implementation of the customized, parallel architecture for a fast word-match program, Computer Systems Science and Engineering, 2011 vol. 26 iss. 4 s. 285–292.
[9] Aspar Z., Yusof Z. M., Suleiman I.: Parallel Huffman Decoder with an Optimize Look Up Table Option on FPGA, TENCON 2000, Proceedings , 1 . pp. 73-76.
[10] Jamro E., Wielgosz M., Wiatr K.: FPGA Implementation of the Dynamic Huffman Encoder, Proc. IFAC Workshop on Programmable Devices and Embedded Systems, Brno, Feb. 14-16, 2006, pp.60-65.
[11] Xilinx Inc. LogiCORE IP AXI Master Burst (axi_master_burst) (v1.00.a), DS844 June 22, 2011.

otrzymano / received: 09.05.2013
przyjęto do druku / accepted: 03.07.2013 *artykuł recenzowany / revised paper*