Advanced Programmable System Interface (APSI)

Opis komend programu apsi.exe
oraz płyty XSV pod kątem współpracy z programem apsi.exe
Autor: Ernest Jamro, jamro@agh.edu.pl

Spis treści

1 Wstęp

Narzędzie to umożliwia programowanie układu FPGA oraz umożliwia transfer danych pomiędzy komputerem a pamięcią zewnętrzną SRAM lub wewnętrzną układu Virtex (BlockRAM lub Distributed RAM). Ponadto narzędzie APSI posiada interpreter specjalnego języka APSI, który umożliwia łatwą komunikacje z płytami programowalnymi i wykonywanie zaawansowanych funkcji.

2 Interpreter skryptu – program apsi.exe

Program apsi.exe interpreterem skryptu, specjalnego języka opracowanego do celów komunikacji z systemami programowalnymi. Program ten umożliwa programowanie układu FPGA oraz transfer danych przez port równoległy EPP. W przyszłości planowana jest również komunikacja przez inne porty. Domyślnym plikiem skryptem, w którym zawarte są komendy do wykonania jest plik apsi.txt. Plik ten można jednak zmienić poprzez uruchomienie programu apsi.exe z parametrem: *apsi.exe nazwa_skryptu*. Skrypt może posiadać opisane poniżej komendy, które należy wpisywać od nowej linii (znakiem poprzedzającym może być jedynie spacja lub tabulacja):

// - komentarz – może być umieszczony w dowolnym miejscu i powoduje ignorowanie wszystkich znaków aż do końca linii.
/* */ - Powoduje rozpoczęcie ‘/*’ lub zakończenie ‘*/’ (podobnie jak w języku C) komentarza zawierającego wiele linii. Uwaga – w jednej linii może wystąpić tylko jeden komentarz: ‘//’ ‘/*’ ‘*/’ – pozostałe są ignorowane.

_ - można umieścić w dowolnym miejscu i jest ignorowany (tak jakby go nie było) przykład write_byte nazwa_pliku.hex 20_0000 20_1000 jest traktowany tak jak: writebyte nazwapliku.hex 200000 201000

Aby komunikacja z płytą przebiegała poprawnie przełączniki na płycie XSV powinny być w trybie OFF

2.1 Komendy programu apsi.exe

2.1.1 Komendy podstawowe

config nazwa_pliku (rozszerzenie bez znaczenia) – powoduje zaprogramowanie układu FPGA podanym plikiem konfiguracyjnym *.bit lub *.cfg. Plik *.bit jest generowany przez Xilinx ISE (należy ustawić opcje: generate bit file) podczas implementacji projektu i znajduje się zwykle w katalogu implement/ver1/rev1. Program apsi.exe korzysta jednak tylko z pliku *cfg, który jest automatycznie generowany na podstawie pliku *bit. Plik *.bit zaraz po wykorzystaniu jest mazany dlatego następnym razem wykorzystywany będzie tylko plik *.cfg.
Procedura jest następująca: jeżeli program apsi.exe nie potrafi znaleźć pliku *nazwa_pliku.bit* to zakłada że plik *nazwa_pliku.cfg* już istnieje i na podstawie tego pliku programuje układ FPGA. Jeżeli jednak program apsi.exe znajdzie *nazwa_pliku.bit* to zakłada, że plik *nazwa_pliku.cfg* nie istnieje lub zawiera starszą wersję. Dalej na podstawie *nazwa_pliku.bit* generowane są pliki tymczasowe *nazwa_pliku.hex* oraz *nazwa_pliku.prm* a dalej na podstawie tych plików tymczasowych generuje plik ostateczny *nazwa_pliku.cfg*. Następnie program apsi.exe maże pliki: *nazwa_pliku.bit*, *nazwa_pliku.hex* i *nazwa_pliku.prm*. Podsumowując przy następnym uruchomieniu procedury *config nazwa_pliku* plik *nazwa_pliku.bit* nie istnieje (został uprzednio zmazany) i program automatycznie ładuje nazwa_pliku.cfg. Jeżeli zmienia się konfiguracja układu FPGA to należy tylko przekopiować plik *.bit, na podstawie którego zostanie utworzony plik *.cfg. Procedura ta została zastosowana ponieważ generacja pliku *.cfg na podstawie pliku *.bit zabiera około 5s. Alternatywnym rozwiązaniem jest ręczna generacja pliku *.cfg, program promgen.exe komendą: promgen -u 0 nazwa_pliku.bit -p hex -s 2048. Następnie można wygenerować plik *.cfg przy pomocy komendy programu apsi.exe: fileconv bazwa_pliku.hex nazwa_pliku.cfg.
config nazwa_pliku liczba – podobnie jak poprzednio, ale liczba określa co ile transferów ma być sprawdzany stan końcówek w celu wykrycia ewentualnych błędów. Domyślna wartość liczba=256, w przypadku błędów polecana jest wartość 1, aby sprawdzenie odbywało się po każdym transferze. Zob. również instrukcje ver i eppmode w celu poprawienia błędu w czasie transferu.

writeblock [nazwa_pliku.bin(.hex)] [adr_start (hex)] [adr_stop (hex)]– powoduje zapisanie zawartości pliku nazwa_pliku od adresu adr_start (w formacie hex) do pamięci, wielkość transferu jest określona rozmiarem pliku lub adresem adr_stop, wybierana jest wartość mniejsza. Adres może być modyfikowany instrukcją baseadr, która określa adres bazowy danego urządzenia. Istnieją dwa możliwe pliki wejściowe: plik *.hex (niekompatybilny z formatem Intela). W pliku *.hex występują tylko dane (nie ma komend oraz adresów) zapisane po kolei od najniższego do najwyższego adresu, analizowane dane powinny zawierać symbole: 0-9 lub a-f (A-F), pozostałe symbole są ignorowane. Alternatywnym formatem jest *.bin – plik binarny, uwaga: każdy plik z rozszerzeniem rożnym od *.hex będzie traktowany jako plik binarny (np. plik danych obrazu *.raw będzie również traktowany jako plik binarny), nazwa pliku musi zawierać rozszerzenie (a dokładnie kropkę, która zaczyna rozszerzenie). Jeżeli została użyta komenda fielappend 1 to zapis powoduje

dołączenie odczytanych danych do już istniejącego pliku. Domyślnie wszystkie pliki są odczytywane i zapisywane w formacie Big Endian – czyli najstarszy bajt ma adres 0.
writeblock adr_start adr_stop - W przypadku pominięcia nazwy pliku, pamięć jest wypełniana wartością rejestru statusowego (4-bajtowo) (zob. Instrukcje działające na rejestrze statusowym programu).
writeblock nazwa_pliku.bin (.hex) – W przypadku podania tylko nazwy pliku adr_start jest określony przez zawartość pamięci statusowej 0 a adr_stop przez zawartość pamięci statusowej 1.
writeblock – W przypadku wywołania instrukcji writeblock bez parametrów następuje zapis do pamięci wartości rejestru statusowego. Adres początkowy jest określony przez pamięć statusową 0, a adres końcowy przez pamięć statusową 1.

writebyte address data (hex) – powoduje zapisanie danej (maksymalnie 8-bajtowej) zapisanej w hex pod podany adres (hex). Rozmiar zapisu jest określony przez liczbę cyfr danej. Rozmiar zapisu nie musi być potęgą liczby 2. Podczas wykonywania komendy wyświetlana jest zapisywana liczba zarówno w kodzie szesnastkowym jak i dziesiętnym. W przypadku kodu dziesiętnego brane są pod uwagę tylko pierwsze 4 bajty zapisywanej danej.

readblock nazwa_pliku.bin(.hex) adr_start adr_stop – powoduje odczytanie zawartości pamięci od adresu adr_start (hex) do adresu adr_stop (hex) do pliku nazwa_pliku.bin (hex) (zob. writeblock). Instrukcja readblock wraca wartość ostatnich 4 przeczytanych bajtów (zob. instrukcje na rejestrze statusowym)

readbyte adr_start [adr_stop] – powoduje odczytanie niewielkiej ilości danych (maksymalnie 1024) spod wskazanej lokacji adresowej (zapisanej w formacie hex). Wartość ta jest następnie wyświetlona na ekranie. W przypadku nie podania adresu końcowego odczytywany jest tylko jeden bajt danej.

baseadr adr – ustawienie adresu bazowego, instrukcja ta powoduje, że instrukcje readblock, readbyte, writeblock, writebyte będą odwoływać się do adresu *adr+adr_start* oraz *adr+adr_stop*. Przykład: baseadr 200000, writebyte 5 FF powoduje zapisanie pod adres 200005 danej FF, aby unieważnić instrukcje baseadr należy użyć instrukcji baseadr 0

ver – zwraca wersję programującą układ CPLD oraz rodzaj układu Virtex oraz stan nóżek programujących. Instrukcja ta jest zalecana w momencie wystąpienia problemów z komunikacją z płytą XSV lub XSB ponieważ pozwala ona na przetestowanie tej komunikacji i ewentualne podanie błędów. Instrukcja ver automatycznie ustawia optymalną wartość eppmode, ale w razie problemów należy ręcznie ustawić optymalny tryb pracy instrukcją eppmode.

eppmode [liczba] (hex) – ustawia tryb pracy portu EPP. Kiedy liczba>= 0x10 to program komunikuje się z płytą XSB w przeciwnym przypadku z płytą XSV. Dla płyty XSV znaczenie liczby jest następujące: *liczba*= b_3, b_2, b_1, b_0 (wartość początkowa: *liczba*= 3)
b_0 – tryb zapisu, b_0= 0 – zapis w trybie EPP, b_0= 1 – software'owa emulacja zapisu
b_1 – tryb odczytu, b_1= 0 – odczyt w trybie EPP, b_1= 1 – software'owa emulacja odczytu
b_2 – tryb czterobajtowy magistrali EPP, b_2= 0 – odczyt / zapis EPP w trybie jednobajtowym, b_2= 1 – odczyt / zapis w trybie czterobajtowym (komputer wykonuje tylko jedną instrukcję aby wysłać cztery bajty poprzez magistralę EPP (kontroler portu równoległego samodzielnie wykonuje 4 niezależne 1-bajtowe transfery na porcie EPP)
b_3 – wyłączenie driveru, b_3= 0 – komunikacja przy pomocy driver'u obsługującego port równoległy (opcja wymagana w Windowsach NT/2000 ze względy na tryb ochrony dostępu do portów we/wy) b_3= 1 – dostęp do portu równoległego poprzez instrukcje asemblera in, out (opcja możliwa w Windowsach 98, nie wymaga kopiowania dodatkowych plików oraz działająca szybciej)
Uwaga 1: Emulacja softwar'owa polega na programowym wymuszeniu każdej zmiany sygnałów na porcie równoległym w sposób podobny jak to robi kontroler EPP, wymaga więc ona szeregu zapisów do portu równoległego wykonywanych przez software, więc opcja ta jest

dużo wolniejsza od normalnej pracy. W przypadku błędów podczas transmisji zaleca się jednak użycia trybu emulacji. W przypadku błędów transmisji występujących nawet podczas trybu emulacji zaleca się przełączenie w BIOSie trybu portu równoległego na tryb SPP. Instrukcja: ver automatycznie ustawia optymalny tryb eppmode w programie apsi, jednak bardzo często ustawienie automatycznie nie wykrywa wszystkich błędów transmisji dlatego w przypadku błędów należy użyć instrukcji eppmode i ręcznie zadeklarować wartość: *liczba*.
Uwaga 2: Bit b_2 przestaje mieć znaczenie podczas pracy w trybie emulacji.
Uwaga 3: Aby zobaczyć aktualnie ustawioną wartość eppmode należy wywołać funkcje bez parametru lub z parametrem ujemnym. Zwrócona wartość może być dalej wykorzystywana do instrukcji warunkowych, np. gobitset.
Uwaga 4: W przypadku wykonywania instrukcji: *run apsi.exe nazwa_skryptu* tryb epp jest ustawiany w nowo wykonywanym skrypcie w stan początkowy = 3 (parametr eppmode nie jest przekazywany do nowego skryptu)
Przykład 1: eppmode //wyświetla aktualny tryb pracy
Przykład 2: eppmode C // tryb pracy bez driver'a w trybie EPP 4-ro bajtowym
Przykład 3: eppmode 3 // (domyślny) praca z driver'em z softwar'ową emulacją zapisu i odczytu.

2.1.2 Komendy służące do sterowania programem

sleep czas_ms – podaje czas uśpienia wykonywania programu w ms.

run komenda – powoduje uruchomienie komendy w linii poleceń systemu operacyjnego, w szczególności możliwe jest uruchomienie innego skryptu poprzez komendę: run apsi.exe nazwa_skryptu

exit – powoduje bezwarunkowe zakończenie wykonywania skryptu.

waitforkey – wstrzymaj wykonywanie skryptu do czasu naciśnięcia jakiegoś klawisz

print tekst – powoduje wyświetlenie na ekranie napisu: *tekst.*

comment liczba– powoduje, że jest (liczba!=0) lub nie jest (liczba=0) wyświetlana aktualnie wykonywana komenda.

goto etykieta **–** instrukcja skoku do etykiety

:etykieta – każda etykieta jest poprzedzona znakiem dwukropka. Dwukropek musi być pierwszym znakiem w linii (nie może być poprzedzony spacją lub tabulacją).

loop liczba_wykonanian etykieta – powoduje przejście do *etykieta* przez (*liczba_wykonan-1)* [hex] razy (instrukcje są wykonywane *liczba_wykonan razy)*. Uwaga: Pętle nie mogą być zagnieżdżone – w jednym czasie może być wykonywana tylko jedna pętla, w przeciwnym wypadku instrukcja pętli nie działa poprawnie. W przypadku bardziej skomplikowanych pętli należy więc używać operacji na rejestrze statusowym. Instrukcja pętli ustawia rejestr statusowy na aktualną wartość licznika pętli (ilość skoków jeszcze do wykonania).

Przykład:

```
:begin
readbyte 0
readbyte 1
loop 100 begin // powoduje 100 (hex) krotne wykonanie operacji readbyte 0 i readbyte 1
:read2
readbyte 2
loop 10 read2 // powoduje wykonywanie 10 (hex) razy readbyte2
```

goto begin //powoduje rozpoczęcie całej instrukcji od początku (nie można tutaj zamiast instrukcji goto użyć instrukcji loop ponieważ powodowałoby to zagnieżdżenie pętli

2.1.3 Instrukcje warunkowe

Instrukcje warunkowe są wykonywane na podstawie rejestru statusowego, którego wartości jest określona przez poprzedzającą instrukcje zmieniającą status. Do instrukcji zmieniających status należą instrukcje: readbyte, readblock, filecomp, eppmode oraz stat. Instrukcje te nie muszą bezpośrednio poprzedzać instrukcje warunkową a jedynie liczy się ostatnio wykonana instrukcja zwracająca wartość. Wartość początkowa rejestru statusowego wynosi 0.

waitbit0 maska_bitowa (hex) adres (hex) [timeout] – powoduje czekanie do momentu aż dana odczytana spod adresu *adres* oraz po dokonaniu operacji bitowej AND z maską *maska_bitowa* będzie równa zeru. Podczas wykonywania tej instrukcji gdy nie jest spełniony warunek jest podawana właśnie przeczytana wartość. Opcjonalnie można podać liczbę odczytów (jeden odczyt to około! 10μs) po którym program przejdzie do następnej instrukcji.
Przykład:
waitbit0 01 20_C010 – powoduje czekanie (zatrzymanie programu) aż do momentu kiedy dana odczytana spod adresu 20_C010 będzie maiła wyzerowany bit zerowy.
waitbit0 03 20_0000 – powoduje czekanie aż do momentu aż bity zerowy oraz pierwszy będą wyzerowane pod adresem 20_0000.

waitbit1 maska_bitowa (hex) adres (hex) – powoduje czekanie do momentu aż dana odczytana spod adresu *adres* oraz po dokonaniu operacji bitowej AND z maską *maska_bitowa* będzie różna od zera.

gobit1 maska_bitowa (hex) etykieta –skok do etykiety jeżeli wartość rejestru statusowego programu apsi po dokonaniu operacji bitowej AND jest różna od zera.
Przykład: gobit1 0F label // skok dla rejestru statusowego równego np. 1, 2, 3, 0xF, 0x23. Brak skoku dla rejestru równego np. 0, 0x10, 0xF0.

gobit0 maska_bitowa (hex) etykieta– instrukcja skoku do etykiety jeżeli wartość rejestru statusowego programu apsi po dokonaniu operacji bitowej AND jest równa zero (zaprzeczenie instrukcji gobit1).

go> wartosc (hex) etykieta– instrukcja skoku do etykiety jeżeli wartość rejestru statusowego programu apsi jest większa od *wartosc*. Uwaga nie wolno wstawiać spacji pomiędzy słowem go a znakiem '<'.

go< wartosc (hex) etykieta – instrukcja skoku do etykiety jeżeli wartość rejestru statusowego programu apsi jest mniejsza od *wartosc*.

go= wartosc (hex) etykieta – instrukcja skoku do etykiety jeżeli wartość rejestru statusowego programu apsi jest równa *wartosc*.

2.1.4 Operacje na rejestrze statusowym

Rejestr statusowy programu apsi służy do lepszej możliwości kontroli nad biegiem programu. Jest to w ogólnie rzecz biorąc rejestr 32-bitowy bez znaku, lecz w niektórych instrukcjach np. writebyte jest używana tylko jego najmłodszy bajt. Uwaga: w instrukcjach tych nie wolno wstawiać spacji pomiędzy instrukcją stat a odpowiednimi znakami np. '?'. Oprócz rejestru statusowego program posiada 256 pamięć rejestru statusowego, co pozwala na zapis i odczyt tych rejestrów spod indeksów 0÷FF.

stat? – powoduje wyświetlenie aktualnej wartości rejestru statusowego
stat= wartosc (hex) – powoduje ustawienie rejestru statusowego zgodnie ze zmienną *wartosc*.

stat+= wartosc (hex) – powoduje dodanie do rejestru statusowego zmienną *wartosc*.
stat-= wartosc (hex) – powoduje odjęcie od rejestru statusowego zmienną *wartosc*.
stat*= wartosc (hex) – powoduje pomnożenie rejestru statusowego przez zmienną *wartosc*.
stat/= wartosc (hex) – powoduje podzielenie rejestru statusowego przez zmienną *wartosc*.
stat&= wartosc (hex) – powoduje wykonanie bitowej operacji AND rejestru statusowego oraz zmienej *wartosc*.
stat|= wartosc (hex) – powoduje wykonanie bitowej operacji OR rejestru statusowego oraz zmienej *wartosc*.
stat^= wartosc (hex) – powoduje wykonanie bitowej operacji XOR rejestru statusowego oraz zmienej *wartosc*.

stat=>m index (hex)– powoduje zapisanie aktualnej wartości rejestru statusowego w pamięci wewnętrznej programu pod adresem index. Uwaga na spacje pomiędzy m a indeksem.
stat<=m index (hex) – powoduje odtworzenie uprzednio zapisanej wartości rejestru statusowego z pod adresu index.
stat<=>m index (hex) – powoduje zamianę wartości rejestru statusowego i pamięci statusowej.
stat+=m index (hex) – powoduje dodanie do rejestru statusowego wartości pamięci o adresie *indeks*.
stat-=m index (hex) – powoduje odjęcie od rejestru statusowego wartości pamięci o adresie *indeks*.

Przykład: Powoduje zapisanie pamięci 20 0000 ÷ 20 00FF kolejnymi danymi 0, 1, ... FF I zapisanie jej do pliku tmp.hex

```
config epp2sram300 // konfiguracja układu FPGA
status= FF // zaczynany od największego adresu
:beg
status=>m 1 // wartość danej do zapisu
status+= 200000 // dodaj offset pamięci wewnętrznej
status=>m 0 // zapis pod domyślnym rejestrem adresowym
status<=m 1 // odtwórz wartość danej do zapisu
writebyte // zapis pod adres zapisany instrukcją savestatus 0 i daną w rejestrze statusowym
loop beg FF
writeblock tmp.hex 200000 2000FF
```

2.1.5 Operacje na plikach

Wszystkie operacje na plikach są wykonywane przy założeniu, że format pliku jest Big Endian, czyli najstarszy bajt jest pod adresem 0. Nie ma to żadnych konsekwencji na pracę programu apsi pod warunkiem, że plik binarny wynikowy nie jest wykorzystywany dalej przez inny program na komputerze PC. W przeciwnym wypadku należy uruchomić instrukcję fileconv zamieniającą format pliku binarnego lub też skorzystać z instrukcji file2txt konwertującej na postać tekstową.

fileconv nazwa_źródła nazwa_przeznaczenia [nr_bit] – powoduje utworzenie nowego pliku w formacie określonym na podstawie rozszerzenia: (.hex – plik w formacie hex, inne rozszerzenie plik w formacie binarnym). Komenda ta umożliwia np. edytowanie plików w postaci hex a następnie konwersje do plików *.bin. Jeżeli fileappend=1 to umożliwia również dołączenie jednego pliku do drugiego. Parametr *nr_bit* służy do konwersji formatu Big Endian na Little Endian i na odwrót. Parametr ten określa wielkość pojedynczego słowa w ramach którego dokonywana jest konwersja. Dla nr_bit<=8 konwersja nie jest dokonywana. Wartość domyślna nr_bit= 0.

filecomp plik1 plik2 – porównuje dwa pliki (mogą być innego formatu i długości) i zwraca numer bajtu na którym znaleziono pierwszą różnicę (liczenie rozpoczyna się od jeden) (zob. np. gobitset), jeżeli pliki są takie same to zwraca 0.

fileappend value– powoduje zmianę metody zapisu do plików. Od tej pory wszystkie zapisy do plików będą wykonywane w trybie stwarzania nowego pliku *value= 0* (domyślnie) lub też w trybie dołączania do pliku *value=1*.

Zaawansowane operacje na plikach

fileselect file_src file_dst start stop [sample_size record_size] – powoduje wybranie tylko części pliku źródłowego *file_src* i zapisanie go do pliku docelowego *file_dst.* W konsekwencji z pliku *file_src* do pliku *file_dst* zapisywane są dane od bajtu *start* [hex] do bajtu *stop* [hex]. Liczenie zaczyna się od wartości 0. Przykład: aby zapisać bajty 17 do 32 z pliku *a.bin* do pliku *b.hex* (w formacie hex, zob. fileconv) należy uruchomić: fileselect a.bin b.hex 10 1F. Dodatkowym parametrem jest sample_size [bit] i record_size [bit]. Pozwalają one wybrać tylko niektóre próbki, czyli z record_size (w bitach) danych wybierane jest tylko sample_size danych (czyli wybierany jest jeden kanał danych spośród wielu innych kanałów).
Przykład: dla pliku wejściowego [hex] 0011223344556677 oraz sample_size= 8 oraz record_size= 16 otrzymamy plik [hex]: 00224466. Możliwe jest podani wartości sample_size > record_size, spowoduje to dodanie dodatkowych bajtów zerowych na najmłodszych bitach tak aby pojedyncza próbka miała rozmiar sample_size.
Przykład: fileselect file1 file2 0 5 32 24 oraz pliku wejściowego 112233445566 otrzymamy plik: 1122330044556600.
Aby dopisać zera na najstarszej pozycji należy najpierw utworzyć plik z (sample_size – record_size)-bitowym zerem a następnie wykonać instrukcję *fileappend 1* oraz instrukcje z poprzedniego przykładu. Lub też wyedytować plik i dopisać zero na najstarszej pozycji.
Podanie wartości *stop* większej niż wielkość pliku powoduje automatyczną aktualizację tej wartości do wartości wielkość pliku-1.

file2txt file_src file_dst nr_bit nr_column – powoduje konwersję pliku binarnego (lub hex) *file_src* do pliku tekstowego *file_dst*. Plik tekstowy może być dalej wykorzystywany np. do przeglądania (liczby zapisane w kodzie dziesiętnym) lub wykreślania np. Excelu. Zmienna *nr_bit* [dec] określa na ilu bitach zapisana jest pojedyncza dana binarna. Zmienna *nr_column* [dec] określa w ilu kolumnach maja być zapisywane dane zanim nastąpi przejście do następnej linii, zmienna ta może być użyteczna w przypadku zapisywania wielu kanałów. Przyjmuje się że liczy są zapisane w formacie BigEndian (czyli najstarszy bit jest zapisywany jako pierwszy).

file2raw file_src file_dst x_size y_size nr_bit [nr_channel [view_option]]. Instrukcja powoduje wygenerowanie pliku graficznego w formacie *.raw pokazującego przebieg zarejestrowanych próbek. Parametr *file_src* określa plik binarny, z którego będą czytane zarejestrowane próbki. Parametr *file_dst* określa nazwę generowanego pliku graficznego (zaleca się aby nazwa miała rozszerzenie *.raw). Parametry *x_size* oraz *y_size* określają wielkość generowanego obrazka. Parametr *nr_bit* oreśla liczbę bitów na których jest prezentowana pojedyncza próbka. Parametr *nr_channel* określa liczbę niezależnych kanałów, które mają być wyświetlane (domyślnie 1). Parametr view_option określa opcje wyświetlania poszczególnych kanałów: 0- (domyślna wartość) wszystkie kanały są rysowane w jednym miejscu, czyli kanały mogą się pokrywać i zamazywać; 1- kanały są rysowane jeden pod drugim na niezależnym miejscu, czyli nie mogą się nakładać, niestety składkowa y przypadająca na każdy wykres ulega zmniejszeniu; >1 wykresy są rysowane jeden pod drugim ale są przesunięte w osi y o view_option pikseli. Dzięki temu wykresy mogą się pokrywać ale prawdopodobieństwo pokrywania maleje. Format *raw jest czytany przez większość programów graficznych, np. darmowy InfranView. Podczas wczytywania tego pliku w programie graficznym należy wybrać opcje: bez nagłówka, obraz monochromatyczny 8 bitów/piksel.

Operacje matematyczne na plikach

filestatistic file_src nr_bit [nr_channel]. Powoduje wyświetlenie podstawowych danych statystycznych: wartości minimalnej (min), maksymalnej (max), różnicy pomiędzy wartością maksymalną i minimalną, oraz podanie wartości średniej zapisanej w pliku. Parametr *file_src* określa nazwę pliku wejściowego, *nr_bit* [dec]– liczbę bitów na której jest zapisana dana (dozwolona wartość to 16-bitów bez znaku). Parametr nr_channel [dec] określa liczbę kanałów na których zapisane są dane (domyślnie 1 kanał).

filehistogram file_src file_dst [nr_bit]. Powoduje obliczenie histogramy danego pliku wejściowego i zapisanie wyniku do pliku *file_dst*. Parametr *nr_bit* określa liczbę bitów na których jest zapisana pojedyncza próbka wejściowa, domyślna wartość to 8-bitów. W przypadku kiedy nr_bit<8 pojedyncza próbka musi zajmować cały bajt (0 na najstarszych bitach). Podobnie jest w przypadku nr_bit= 9..15 kiedy to pojedyncza próbka wejściowa musi zajmować 2 bajty. Próbka wynikowa jest zapisywana w formacie 32-bitowym (big endian).
Przykład: filehistogram lena.raw lena.his // obliczenie histogramu
file2raw lena.his histogram.raw 256 256 32 // narysowanie obliczonego histogramu do pliku graficznego

2.1.6 Testbench dla VHDL

vhdlsim – przejście w tryb symulacji. Generacja pliku apsi.tst służącego do symulacji oraz odczytanie i przetworzenie pliku apsi.out, który zawiera wyniki symulacji vhdl odczytane przez port równoległy.

W celu dokonania symulacji VHDL całego systemu współpracującego z modułem APSI, dokonano specjalnej modyfikacji modułu APSI, który po dodaniu komendy *vhdl_sim* przestaje komunikować się płytą z układem FPGA i przechodzi w tryb symulacji. Dokładniejszy opis całego systemu znajduje się w osobnych tutorialach.

Niestety bardzo trudna jest symulacja na poziomie języka VHDL wszystkich możliwości programu *apsi.exe* komunikującego się przez port równoległy z płytą XSV/VSB. Dotyczy to szczególnie komend skoku warunkowego, dla którego skok jest wykonywany w zależności od wartości odczytanej poprzez port równoległy. Warto podkreślić, że instrukcja *waitbit* symuluje się poprawnie.

Komendy stosowane w pliku apsi.tst

1, a – write address a (byte) – wystawienie na porcie EPP adresu do zapisu, 4 transfery są wymagane aby przesłać jeden adres
2, d – write data d (byte) – wystawienie danej do zapisu na porcie EPP.
3 – read – powoduje odczyt danej z portu EPP
10, t – sleep time – czas wstrzymania aktywności w ms (t<256)
11, b, a_3, a_2, a_1, a_0, m, - waitbit0, waitbit1 – zob. instrukcje skryptu. b=0 waitbit0, b=1 waitbit1, a_i –adres (4 transfery, m- maska)
255 – End Of File – zakończenie, koniec pliku.

2.1.7 Komendy obsługujące poszczególne urządzenia

2.1.7.1 Reset układu

reset –moduł opb_epp posiada wewnętrzny układ umożliwiający ustawienie sygnału reset w stan wysoki na krótki okres czasowy (około 10us). Aby tego dokonać można użyć komendy reset lub też komendy writebyte FFFF_FFFF 01 (ustaw reset w stan wysoki) i następnie writebyte FFFF_FFFF 00 (ustaw reset w stan niski).

2.1.7.2 Analizator Stanów logicznych

LA – Internal Logic State Analyser – moduły opb_epp dla c_la_mwidth≥4 lub opb_la.
Dodatkowy opis tych instrukcji można znaleźć w opisie modułu opb_la.

Możliwe jest równoczesne obsługiwanie więcej niż jednego modułu analizatora stanów logicznych. Przy każdym odwołaniu do innego modułu należy jednak odpowiednio ustawić parametry labaseadr (oraz laawidth tylko w przypadku symulacji VHDL).

labaseadr adr – powoduje ustawienie adresu *adr* bazowego analizatora stanów logicznych – instrukcja podobna do baseadr ale dotyczy tylko analizatora stanów logicznych. Dla projektów opb_epp wartość ta powinna być ustawiona na FFFF_0000. Dla modułu opb_la powinna ona mieć wartość c_baseaddr. Podczas wykonywania tej instrukcji program *apsi.exe* sprawdza czy pod podaną lokacją adresową znajduje się analizator stanów logicznych i odczytuje jego parametry. Dlatego nie jest konieczne ustawianie jego parametrów za pomocą komend *laawidth* lub *lahighadr*. Jest to konieczne tylko podczas symulacji VHDL.

laawidth liczba (dec) – Komenda używana tylko podczas wstępnej symulacji VHDL gdy analizator nie został jeszcze poprawnie wykryty; normalnie komanda ta jest ignorowana – laawidth jest określone poprzez komunikacje z analizatorem stanów logicznych. Komenda ta określa szerokości magistrali adresowej analizatora – powinien to być to ten sam parametr co dla entity log_anal generic adr_width. Dla modułu opb_epp należy ustawić laawidth= c_la_mwidth + log2(c_la_dwidth/8) + 1. Dla modułu opb_la należy ustawić log2(c_highaddr+1-c_baseaddr) lub użyć instrukcji *lahighadr*. Instrukcja powinna być użyta przed instrukcją labaseadr.

lahighadr adr – Podobnie jak instrukcja *laawidth*, instrukcja ta jest używana tylko podczas wstępnej symulacji VHDL, kiedy nie ma poprawnej komunikacji z modułem analizatora stanów logicznych. Instrukcja ta powoduje ustawienie adresu końcowego analizatora stanów logicznych. Instrukcja ta powinna być użyta zamiennie z instrukcją *laawidth*. Parametr *adr* powinien być taki sam jak parametr *c_highaddr* użyty w parametrach modułu *opb_la*. Instrukcja powinna być użyta przed instrukcji labaseadr.

latvalue liczba (hex) – określa wartość triggera, czyli takiej sekwencji danych, która powoduje wyzwolenie analizatora. Miejsce wyzwolenia (w oglądanym przebiegu) określa instrukcja lastart. Zob. lacare.

Warto podkreślić, że sygnał wyzwalający jest ignorowany do czasu kiedy nastąpi wypełnienie bufora pamięci w części określonej przez miejsce sygnału wyzwalającego (np. jeżeli trigger jest na końcu rejestrowanych próbek to najpierw musi być wypełniony cały bufor pamięci a następnie dopiero stan sygnału wyzwalającego jest brany pod uwagę). Nawet w momencie kiedy trigger jest na samym początku musi być zarejestrowana co najmniej jedna dana. Jest to szczególnie ważny warunek w momencie kiedy używa się logiki zezwolenia zegara dla rejestrowanych próbek.

latcare liczba (hex) – określa czy dany bit triggera jest brany pod uwagę (1) czy też nie (0). Dla przykładu dla: *latvalue 50* oraz *latcare F0*, dane zostaną zarejestrowane jeżeli na wejściu triggera wystąpi sekwencja 5X (gdzie x oznacza poziom dowolny).

lacevalue liczba (hex) – określa wartość sygnału wejściowego la_ce_Dbus (moduł opb_la lub opb_epp), przy którym nastąpi uaktywnienie sygnału zezwolenia zegara (CE) dla rejestrowania próbek lub też uaktywnienie dodatkowego triggera. Instrukcja jest ważna tylko wtedy kiedy został wybrany parametr c_la_ce_dtype>0. Instrukcja działa podobnie jak instrukcja latvalue lecz dotyczy sygnału CE lub dodatkowego triggera. Zob. latcare oraz lactr.

lacecare liczba (hex) – instrukcja podobna jak instrukcja lacevalue ale decyduje czy odpowiedni bit wartości zadeklarowana instrukcją lacevalue jest brany pod uwagę (1) lub nie jest brany pod uwagę (0).

lacectr liczba (hex) – powoduje zapis do rejestru kontrolnego sterującego funkcją CED (zezwolenia ze do rejestrowanych danych lub rejestru kontrolnego). Zapis jest ważny tylko wtedy kiedy parametr c_la_ce_dtype>0. *liczba* jest określana bitowo na podstawie poniższych bitów. Domyślny wpis do wszystkich bitów to 0.

Bit #	**Opis**
0 ced_acti ve_low	Określa poziom wyjścia logiki CED, który aktywuje logikę zezwolenia zegara lub dodatkowego triggera. 0 – Powoduje, że spełnienie warunku lacevalue i lacecare aktywuje logikę. 1 – Powoduje, że nie spełnienie tego warunku aktywuje logikę.
1 ced_reg	0 – wyjście logiki CED jest bezpośrednie (nie opóźnione) 1 – dodatkowy przerzutnik opóźniający o jeden takt zegara. Opóźnienie to powoduje, że możliwe staje się wykrywanie zmian sygnału – jeżeli użyto logiki dodatkowego triggera.
2 ce_dat	0 – Sygnał jest rejestrowany niezależnie od logiki CED. 1 – Wyjście logiki CED pełni funkcje zezwolenia zegara dla rejestrowanych próbek – próbki są rejestrowane tylko wtedy gdy spełniony jest bądź nie (zob. bit 0) warunek value i care.
3 and_trig	0 – Bramka AND dla dodatkowego triggera jest nieaktywna. Trigger główny (lub bit 4) decyduje o momencie wyzwolenia. 1 – Analizator zostanie wyzwolony w momencie, kiedy równocześnie trigger główny oraz logika CED są aktywne (zob. Bit 0, 1)
4 or_trig	0 – Bramka OR jest nieaktywna. Wyzwolenie następuje tylko dla triggera głównego (zob. bit 3). 1- Analizator zostanie wyzwolony kiedy spełniony jest warunek triggera głównego **lub** logiki CED
5 RLC (not)	0 – używaj kodowania RLC (oczywiście jeżeli został ustawiony odpowiedni parametr – use_run_length_coding 1 – nie używaj kodowania RLC. Bit ten należy ustawić w momencie kiedy analizator nie potrafi zakończyć rejestracji pamięci lub nie ulega długo wyzwoleniu.

Przykład:
lacectr 0 // logika CED jest nieaktywna czyli tak jakby jej w ogóle nie było (wartość początkowa)

lacectr 4 // powoduje rejestrowanie próbek tylko wtedy kiedy jest spełniony warunek logiki CED (czyli wejście logiki CE spełnia warunek lacevalue oraz lacecare

lacectr 11 // Wyzwolenie analizatora następuje tylko wtedy kiedy jest spełniony warunek latvalue latcare oraz nie jest spełniony warunek lacevalue lacecare.

latvalue 05
latcare 0F
lacevalue 0A
lacecare 0F
lacectr A // powoduje wyzwolenie analizatora w momencie kiedy sygnał na danych D0-D3 zmieni się z A na 5 w jednym takcie zegara (pod warunkiem że c_la_ce_type= 3)

lastart liczba (hex) – określa miejsce położenia triggera względem obserwowanych próbek. *Liczba*= 00 – trigger na poczatku, liczba= 20- trigger w środku, liczba= 3F – trigger na końcu, 10xx_xxxx (bin) – wyzwól LA w chwili zapisu do rejestru – miejsce triggera jest określone przez 6 najmłodszych bitów. Instrukcja lastart jest równoważna zapisowi pod rejestr statusowy układu LA (zob. log_anal.doc(pdf))

laread [nazwa_pliku, [timeout]]– powoduje oczekiwanie aż zostanie zakończony proces akwizycji danych oraz powoduje zapis zarejestrowanych danych do pliku la_data.bin, który jest następnie czytany prze entity la_view.vhd. Plik la_view automatycznie wyświetla

zapisane dane w pliku la_data.bin. Możliwa jest zmiana domyślnej nazwy pliku la_data.bin, do którego zapisywane są pobrane dane na plik *nazwa_pliku*. Po liczbie odczytów *timeout* (jeden odczyt to około! 10μs) po której analizator nie zakończy akwizycji danych następuje zapis pod rejestr kontrolny ced o wartości 0x20 – co powoduje wyłączenie logiki zezwolenia zegara, dodatkowego triggera oraz wyłączenie kompresji danych RLC. Następnie procedura zostanie powtórzona. Jeżeli powtórnie po wykonaniu *timeout* odczytów nie zostanie zapełniony bufor pamięci analizatora, to program dokona odczytu pomimo tego, że część bufora (ta na początku rejestracji) nie zawiera poprawnych danych.
Podczas akwizycji po instrukcji la_read jest wyświetlana dana, rejestr stanu analizatora stanów logicznych zob, log_anal.doc. Jeśli bit 0x80 jest równy zero to analizator nie został wyzwolony i dlatego proces akwizycji nie mógł przebiec poprawnie. W tym wypadku należy np. zmienić wartość triggera. Jeżeli bit 0x80 jest ustawiony to następuje proces akwizycji danych, zakończy się on kiedy zostanie wypełniona pamięć próbek czyli odczytana dana (rejestr statusowy) osiągnie wartość 0xC0. Jeśli jednak inkrementacja tego rejestru do wartość 0xC0 przebiega zbyt wolno należy albo zmienić wartość logiki zezwolenia zegara (CE) instrukcją la_cecare / la_cevalue lub też zrezygnować z kompresji danych RLC (zob. lacectr).
Przykład dla parametrów domyślnych opb_epp.

```
la_base_adr FF_0000
la_tvalue 1
la_tcare 1 // przy wyzwalaniu liczą się tylko bity 1,3-7
la_start 01 // zarejestruj moment wyzwalania trigger na początku
// tu umiesc cos co chcesz oglądnąć
la_read // powoduje odczyt zarejestrowanych danych
```

Aby ostatecznie móc oglądnąć zarejestrowane dane należy w symulatorze VHDL wybrać jako plik nadrzędny moduł la_view.vhd wraz z zapisanym uprzednio plikiem zarejestrowanych danych (domyślnie la_data.bin). Zobacz dokładniejszą instrukcje log_anal.doc.

2.1.7.3 I^2C oraz Video ADC (urządzenie z OpenCores)

i2cwrite sub_adr dana – powoduje zapisanie danej *dana* pod określony pod-adres urządzenia ADC Video. Uwaga: Aby dana instrukcja zadziałała poprawnie należy określić adres bazowy urządzenia I^2C master instrukcją base_adr oraz określić częstotliwość zegara i uaktywnić urządzenie I^2C master instrukcjami writebyte pod adres 0-2.
Przykład:

```
baseadr 20_C000 // określenie adresu bazowego układu I2C master (zależy od projektu w VHDLu)
writebyte 0 63 //= 99 (dec) daje czestotliwosc i2c 100kHz dla zegara 50MHz
writebyte 1 0 // // czestotliwosc zegara (starszy bajt)
writebyte 2 80 // Control register (aktywuj i2c oraz wylacz przerwanie)
i2cwrite 40 12 //zapis pod sub-adres 40 liczby 12
i2cread 40 // odczyt poprzednio zapisanej danej
```

i2cread sub_adr – powoduje odczytanie danej spod sub-adresu *sub_adr*. Odczytana dana jest zapisana w rejestrze statutowym. Patrz również przykład i2cwrite.

2.2 Układ CPLD (XC95108)

Układu CPLD jest niedostępny od strony użytkownika. Warto jednak podkreślić, że aby cały układ działał poprawnie musi być właściwie zaprogramowany układu CPLD. Ponadto nie jest możliwe równoczesne programowanie układu CPLD i korzystanie z portu równoległego EPP ponieważ pin 11 portu równoległego jest połączony z dwoma końcówkami układu CPLD (końcówka 63 oraz 83 – przez negator i rezystor 300Ω). Dlatego aby zaprogramować układ

CPLD należy ustawić DIPSW8 w pozycji ON. Układ CPLD można przeprogramowywać tylko na wyraźne polecenie prowadzącego.
Uwaga: Aby była poprawna komunikacja przez złącze EPP, DIP SWITCH 8 musi być w pozycji OFF).

2.3 Układ Virtex.

Wielokrotna rekonfiguracja

Układ Virtex składa się z wielu podbloków, które w dużej mierze zależą od użytkownika. Istnieje jednak już wykonany projekt epp2sram, który służy do przesyłania danych z/do pamięci zewnętrznej SRAM poprzez port EPP. Blok ten można użyć bezpośrednio w postaci konfiguracji układu FPGA, plik epp2sram.bit (uwaga należy zwrócić uwagę jaki układ virtex jest na płycie: dla układu Virtex XCV300 PQ240 długość pliku epp2sram.bit wynosi 219 047 bajtów, dla układu XCV800 PQ240 odpowiednio 589 529 bajtów). Po zaprogramowaniu układu Virtex plikiem epp2sram.bit (komenda config epp2sram) można dalej transferować dane do pamięci zewnętrznej komendami writeblock lub transferować dane z pamięci sram do komputera przy pomocy komend readblock, readbyte (patrz punkt 1.1). Następnie można reprogramować układ Virtex swoim własnym programem przy pomocy komendę config mój_modul.hex. W ramach gotowego projektu można również przesyłać dane do pamięci wewnętrznej oraz oglądać sygnały za pomocą wewnętrznego analizatora stanów logicznych, szczegóły można znaleźć w następnym podpunkcie

Jedna konfiguracja

Alternatywnym i zarazem zalecanym rozwiązaniem jest dołączenie modułów transferu danych do swojego projektu. W przypadku projektu EDK należy dołączyć moduł opb_epp oraz opb_sram. Rozwiązanie to wymaga zapoznania się z tymi modułami, co może być na początku kłopotliwe, jednakże dodatkowy nakład pracy może się bardzo szybko zwrócić, ponieważ projekt w ten sposób zyskuje dodatkowy wymiar. Można przesyłać dane bez dodatkowego przeprogramowania układu FPGA oraz co wydaje się jeszcze ważniejsze, można kontrolować swój program poprzez złącze EPP, można korzystać z wewnętrznego analizatora stanów logicznych, pamięci wewnętrznej. Dlatego rozwiązanie to jest zalecane przez prowadzącego!!!.

2.4 Moduły służące do symulacji i testowania

test.vhd – moduł nadrzędny podczas testowania, w którym oprócz modułu układu Virtex (w przypadku projektu EDK – system.vhd) podłącza się dodatkowe moduły emulujące zewnętrzen urządzenia znajdujące się na płycie XSV.
sram_model.vhd – moduł służący do emulacji pamięci SRAM – moduł ten służy do symulowania działania pamięci SRAM. Pamięć SRAM jest inicjalizowana zawartością pliku binarnego: sramin.bin, a po czasie określonym przez parametr mem_write_time zawartość pamięci jest zapisywana do pliku sramout.bin
la_view.vhd – moduł służący do odczytu (wyświetlania) danych zapisanych przez analizator stanów logicznych log_anal.vhd. Należy przeczytać instrukcje do modułu opb_epp oraz opb_la.
epp_model.vhd – moduł emulujący działanie portu równoległego oraz programu apsi.exe. Pełniejszy opis działania tego modułu znajduje się w opisie komendy programu apsi.exe vhdlsim**.**

2.5 Ustawienie przełączników na płycie XSV

Wszystkie przełączniki DIP SWITCH powinny być w pozycji OFF (wyjątki skonsultować z prowadzącym)
DIPSW8 – (nóżka adresowa A20 pamięci FLASH) powinien być ustawiony w pozycji ON w momencie kiedy chcemy przeprogramować układ CPLD (układ CPLD powinien być przeprogramowywany tylko za wyraźnym pozwoleniem prowadzącego).
J23 – rozwarty - normalna praca; zwarty – programowanie CPLD.
J29 – 1-2
J30 – 1-2
J31 – 2-3

3 Opis projektu dla projektanta

3.1 Odczyt wersji programu zapisanego w CPLD oraz stanu nóżek programujących.

Płyta XSV
Aby sprawdzić stan płyty XSV należy dokonać odczyty adresu zgodnie ze standardem EPP. Stan poszczególnych bitów oraz ich znaczenie jest podany w poniższej tabeli:

1	0	Vv	v	1	0	Done	InitN

gdzie:
v – bit które mówią o wersji programującej CPLD.
vv – wersja układu Virtex 800 lub 300. vv= 1 układ XCV300 PQ240, vv= 0 układ XCV800 HQ240.
Done- stan linii programującej Done (=1 - poprawnie zaprogramowany układ FPGA)
InitN- stan linii programującej InitN. Końcówka świadcząca o inicjalizacji lub błedu podczas programowania)
Odczyt adresu można wykonać za pomocą instrukcji ver, oraz powinien być wykonany w celu sprawdzenie czy układ CPLD został zaprogramowany właściwym programem. Ponadto odczyt ten jest konieczny podczas sprawdzenie poprawności zaprogramowania układu FPGA.

Płyta XSB
Dla płyty XSB poprzez odczyt adresu odczytujemy stan płyty w następujący sposób:

1	0	v1	v0	0	Progn	Done	InitN

Gdzie v1, v0 – wersja programująca CPLD
Progn – stan koncówki ProgramN układu FPGA (Progn='0' – kasowanie konfiguracji układu FPGA)

3.2 Komunikacja pomiędzy układami CPLD i FPGA

Układ CPLD (oprócz cyklu programowania układu FPGA) kopiuje (z pewnym przybliżeniem) sygnały na magistrali EPP komputera do układu FPGA. Dlatego układ FPGA nie widzi układu CPLD i zachowuje się tak jakby bezpośrednio był połączony z portem równoległym komputera.

Dodatkowe sygnały wystawiane przez układ CPLD a nie mające odpowiednika na magistrali EPP:
fcen (Flash Chip Enable Not) sygnał umożliwiający zapis i odczyt pamięci Flash. W przypadku komunikacji pomiędzy układem CPLD a układem FPGA sygnał powinien być zawsze równy: fcen=1; w przepadku zapisu/ odczytu pamięci FPASH fcen= 0.
M2, M1, M0 – sygnały służące do ustawienia trybu programowania układu FPGA. W trybie programowania Serial Slave: M2= 1, M1= 1, M0= 1.
v_initn – pamięć konfiguracji jest czyszczona, również sygnalizacja błędu programowania układu FPGA.
v_done (P10) – sygnalizacja zakończenia procesu programowania
v_cclk (P12) – sygnał zegarowy (dane programujące są zatrzaskiwane narastającym zboczem zegara