Programowo-sprzętowa realizacja algorytmów

materiały do ćwiczeń laboratoryjnych

Wesja 2018.3

Tomasz Kryjak

PUBLISHED BY AGH

First printing, March 2016

Contents

1 — Obsługa PS na płytce Zybo

1.1 Opis

W ramach ćwiczenia zrealizowana zostanie prosta aplikacja typu “Hello World” dla systemu procesorowego (ang. *processing system* – PS) układu Zynq na karcie Zybo. Cele są następujące:

- wstępne zapoznanie ze środowiskiem Vivado i SDK,
- demonstracja konfiguracji procesora oraz pisania (czy raczej uruchamiania prostego programu),
- dalsze zapoznanie z platformą Zybo.

1.2 Przebieg ćwiczenia

1. Uruchom program Vivado HLx Editions (wersję dostępną w laboratorium).
2. Wybierz *Create Project*. Otworzy się kreator.
3. Nazwa (np. ald – *abandoned luggage detection*) i ścieżka “swojego folderu”. Typ projektu RTL. Źródeł i ograniczeń nie dodajemy. Wybieramy płytkę Zybo (Vendor digilent.com). Uwaga. Wersja ”stara” tj. Zybo (nie Zybo-Z10, czy Z-20). Tworzymy projekt.
4. W zakładce *Flow Navigator*, w polu *IP INTEGRATOR* wybieramy *Create Block Design*. Możemy go nazwać *ald*.
5. Dodajemy nowy moduł (*Add IP*). Można skorzystać z znaku plus (u góry na pasku lub na środku pustego schematu). Wpisujemy w pole wyszukiwania *Zynq* i wybieramy moduł *ZYNQ7 Processing System*. Moduł pojawi się na ekranie.
6. Klikamy dwukrotnie na stworzony moduł i przechodzimy do konfiguracji. Otworzy się schemat procesora. Należy zauważyć, że domyślnie nie są aktywne żadne peryferia. Ponadto w rozważanej wersji aplikacji Vivado nie ma bezpośredniego wsparcia dla karty Zybo, co jest przyczyną pewnych trudności.
7. Ładujemy konfigurację domyślną (*Presets->Default*). Naciskamy OK i zamykamy schemat. W górnej części okna, na zielonym tle, pojawi się opcja *Run Block Automation*. Należy ją uruchomić. Moduł ”urośnie”. Warto zwrócić uwagę, że podłączona zostanie pamięć DDR oraz stałe wejścia i wyjście (*FIXED_IO*).
8. Ponownie otwieramy konfigurację procesora. Tym razem mamy znaczny nadmiar peryferiów. Przechodzimy do zakładki *Peripheral I/O Pins* i odznaczamy wszystko oprócz UART 1. Zatwierdzamy OK.

9. Wybieramy opcję *Validate design* – w *Tools* lub przycisk na pasku lub F6. Pojawi się błąd o braku podłączenia zegara. Naprawiamy go łącząc wyjście *FCLK_CLK0* z textitM_AXI_GP0_ACLK. Ponownie uruchamiamy walidację – powinien pojawić się komunikat o braku błędów.
10. W zakładce *Sources*, w folderze *Design sources* naciskamy prawym przyciskiem myszy na nazwę naszego schematu *ald* i wybieramy *Create HLD Wrapper*. Na oknie, które się pojawi, wybieramy OK.
11. Ponownie otwieramy to samo menu i wybieramy opcję *Generate Output Products*. Operacja chwilę trwa...
12. Kolejny krok to synteza, implementacja i generacja pliku konfiguracyjnego (*bitstream*). Z *Flow Navigator* należy wybrać *Generate Bitstream* i chwilę poczekać.
13. Po tym kroku część "sprzętową" uznajemy za zamkniętą. Przechodzimy zatem do tworzenia aplikacji na procesor.
14. Wybieramy *File->Export Hardware* i zaznaczamy *Include bitstream*. Następnie wybieramy *File->Lunch SDK*. Otworzy się aplikacja SDK (bazująca na Eclipse), w której tworzy się kod na procesor.
15. Proszę upewnić się, że karta Zybo jest podpięta do komputera kablem USB i włączona.
16. Wybieramy *File->New->Application Project*. W kreatorze ustalamy nazwę np. *hello_world*, pozostałe opcje na razie bez zmian. Wybieramy *Next*. Domyślnie zaznaczona jest opcja *Hello World* i taką zostawiamy. Warto zobaczyć jakie inne aplikacje są dostępne. Zamykamy kreator.
17. Stworzony projekt budujemy – prawy klawisz myszy i *Build Project*. Następnie programujemy FPGA – *Xilinx->Program FPGA* lub ikonka ze strzałką i trzema zielonymi kwadratami.
18. Otwieramy program – plik *helloworld.c*. Kod jest bardzo prosty. Warto dodać pętlę `while`, tak aby napis wyświetlał się ciągle.
19. Do połączenia z kartą użyjemy programu CuteCom. Po pierwsze należy sprawdzić jak "nazywa się" nasza karta. Uruchamiamy konsolę i przechodzimy do katalogu `dev` (`cd /dev`). Wyświetlamy jego zawartość (`ls`). Sprawdzamy `ttyUSBX` (X – liczba). Włączając/wyłączając kartę Zybo można stwierdzić, który numer odpowiada szukanemu urządzeniu. Ja to już wiemy, to uruchamiamy program `cutecom` (z konsoli). Jeśli nie mamy dostępu do portu, to należy uruchomić program z opcją `sudo` (jako administrator).
20. Aby uruchomić aplikację należy nacisnąć prawy klawisz myszy na projekcie i wybrać *Run As* i opcję 1. Jeśli wszystko zostało wykonane poprawnie to na konsoli powinien pojawić się napis *Hello World*.

1.2.1 A jakbyśmy jednak chcieli C++ ?

W przypadku zaawansowanych aplikacji lepiej jest wykorzystać możliwości jakie daje C++. Poniżej pokazano jak uruchomić tą samą aplikację w C++ – niestety firma Xilinx nie dostarcza (od lat) stosownego przykładu.

Tworzymy nowy projekt.

- *File->New->Application Project*,
- wybieramy nazwę oraz język C++,
- niestety, zbiór szablonów aplikacji dostępny jest tylko dla C. Zatem czeka nas trochę „kombinowania".
- otwieramy plik main.cc. Dodajemy następujący kod (można skopiować z projektu w C):

```
#include "xparameters.h"
#include "platform.h"
#include "xil_printf.h"
```

```
#include "xil_io.h"
```

- okaże się, że brakuje pliku *platform.h*. Jest on dostępny w projekcie z C stworzonym wcześniej. Trzy pliki: platform.h, platform.c oraz platform_config.h trzeba przekopiować do folderu src w katalogu .sdk/nazwa_aplikacji. Można to zrobić „po prostu" w SDK. Ew. po tej operacji odświeżamy Project Explorer (PPM->Refresh).
- do main.cc kopiujemy kod z helloworld.c.
- zmieniamy funkcję `print` na `xil_printf` (taki Xilinx'owy printf).
- ponownie uruchamiamy projekt (ew. może być potrzeby reset karty – ogólnie jest to jedna z pierwszych czynności, które warto podjąć, jak coś nie działa jak powinno).

Stara góralska zasadna pracy z Vivado mówi – zachowywać kopie zapasowe. Robienie tego w git jest utrudnione, zatem zip i do jakiegoś backup. A dopiero później dodawać kolejne komponenty.

2 — Tor wizyjny w formacie AXI Stream

2.1 Wstęp

W ramach ćwiczenia do projektu zostanie dodana konwersja standardowego strumienia wizyjnego do AXI Stream (oraz z AXI Stream do standardowego). Warto zaznaczyć, że format AXI Stream jest stosowany (oraz mocno promowany) przez firmę Xilinx. Nam konwersja będzie przydatna do realizacji segmentacji obiektów pierwszoplanowych.

2.2 Vivado

W pierwszym kroku, do schematu stworzonego w rozdziale 1, należy dodać moduły: dekodujące sygnał HDMI oraz obsługujący port VGA. Jest to procedura analogiczna do tej realizowanej w ramach kursu Systemy Rekonfigurowalne. W celu ułatwienia pracy na stronie kursu przygotowano archiwum z wstępnie przygotowanym projektem (tor wizyjny + procesor).

1. Wypakuj projekt z torem wizyjnym i procesorem. Ponadto do tego samego folderu wypakuj zawartość archiwum `ip_repo`.
2. Przeprowadź implementację projektu oraz uruchom go na karcie Zybo. Sprawdź działanie toru wizyjnego (dla przypomnienia przydatne może być polecenie Linux `xrand`). Sprawdź też działanie systemu procesorowego – aplikacja (*hello world*). Jeśli wszystko działa, to tworzymy kopię zapasową i rozpoczynamy realizację ćwiczenia.
3. Otwieramy schemat blokowy (*Open Block Design*). Dodajemy trzy moduły (*Add IP* np. pod prawym klawiszem myszy lub znak „plus” na pasku):
   - Video In to AXI4-Stream,
   - AXI4-Stream to Video Out,
   - Video Timing Controler (VTC).

   Rola pierwszych dwóch jest oczywista, a VTC odpowiada za potrzebne generowanie sygnałów synchronizacji (w podejściu Xilinx'a nie są obe przesyłane). Uwaga. Należy zaznaczyć, że także w strumieniu AXI Stream nie występują pola wygaszania poziomego i pionowego – przesyłane są tylko „poprawne” piksele”. Przystępujemy do konfiguracji modułów.
4. *Video In to AXI4-Stream*. Parametry modułu (bez zmian – pozostawiamy domyślne). Wejścia:
   - `vid_io_in` – podłączamy do RGB z *dvi2rgb_0*

- `vid_io_in_ce` – podłączamy do VCC (bloczek Constant z ustaloną na stałe wartością 1),
- `aclk` – podłączamy do `PixelClk`,
- `aclken` – podłączamy do VCC,
- `aresentn` – podłączamy do VCC (port aktywowany stanem niskim),
- `axis_enable` – podłączamy do VCC.

5. *Video In to AXI4-Stream*. Parametry modułu (bez zmian – pozostawiamy domyślne). Wejścia:
   - `video_in` – do `video_out` z *Video In to AXI4-Stream*,
   - `vtiming_in` – do `vtiming_out` z *Video Timing Controler,*
   - `aclk` – do `PixelClk`,
   - `aclken` – do VCC,
   - `aresetn` – do VCC,
   - `vid_io_out_ce` – do VCC,
   - `fid` – pozostaje niepodłączone.

   Wyjścia:
   - `vid_io_out` do `vid_in` z *RGB to VGA output*.
6. *VTC*. Parametry modułu:
   - *Max Clocks Per Line* – 2048,
   - *Max Lines Per Frame* – 2048,
   - zaznaczamy *Auto Generation Mode*,
   - reszta bez zmian.

   Wejścia:
   - `vtiming_in` do `vtiming_out` z *Video In to AXI-4 Stream*,
   - `clk` do `PixelClk`,
   - `clken` – do VCC,
   - `s_axi_aclken` – do VCC,
   - `det_clken` – do VCC,
   - `resetn` – do VCC,
   - `gen_clken` -> `vtg_ce` z *AXI4Stream to Video Out*.
7. Kolejny krok do połączenie modułu *VTC* do systemu procesorowego z wykorzystaniem magistrali AXI4-Lite. Zadanie jest uproszczone – należy wybrać dostępną w górnej części schematu opcję *Run Connection Automation*. Kreator samodzielnie stworzy niezbędne połączenia. Warto zauważyć, że pojawia się także dodatkowy moduł AXI Interconnect.
8. Do „posprzątania" schematu służy opcja *Regenerate Layout*.
9. Dla potrzeb eliminowania ew. błędów dobrze jest dodać wyświetlanie flag statusu *overflow* i *underflow* z obu modułów. Można to zrobić poprzez „wyświetlnie" ich wartości na diodach LED. W tym celu:
   - należy otworzyć plik xdc (plik ograniczeń użytkownika). W hierarchii *Constraints->constrs_1>Zybo_HDMI.xdc*.
   - odszukać i od komentować 4 linijki, w których opisane są diody LED,
   - na schemacie dodać port wyjściowy (PPM->Create Port). Nazwać go `led` (uwaga nazwa musi być taka sama jak w xdc), *Direction->Output*, *Vector [3:0]*.
   - do schematu dodajemy bloczek *Concat* (*Add IP*), ustawiamy liczę portów na 4
   - wszystko łączymy.
10. Generujemy plik konfiguracyjny. Wykonujemy eksport konfiguracji (wraz z plikiem bit) i przechodzimy do SDK. Uwaga. Jeśli SDK nie uruchamia się – pojawia się ekran startowy, który po chwili znika – pomaga wykasowanie projektu (w folderze rozszerzenie *.sdk*). Oczywiście wcześniej warto zrobić kopię zapasową plików źródłowych.

2.3 SDK

1. W pierwszym kroku testujemy, czy (dalej) działa *Hello world*. Jeśli nie (a powinien), to można spróbować stworzyć nowy projekt z przykładową aplikacją, uruchomić (powinna działać) i wrócić do „naszej”. Inna opcja to wybrać – *Lunch on Hardware (GDB)*.
2. Kolejny krok to „obsługa” *VTC* z wykorzystaniem *AXI4-Stream Lite*. Uwaga. Jest to dość wygodne, gdyż konfigurowanie modułu z poziomu aplikacji jest dużo szybsze, niż testowanie kolejnych parametrów w sprzęcie (i czekanie na skończenie kolejnych implementacji).
3. Otwieramy plik *xparameters.h*. Znajduje się on w *nazwa_aplikacj_bsp/ps7_cortex9_0/include*. W pliku wyszukujemy „VTC” (najlepiej Ctrl+F) Powinna znajdować się tam cała sekcja dotycząca tego urządzenia. Nas w tym momencie interesuje `XPAR_VTC_0_BASEADDR` (adres urządzenia w systemie). Warto w tym miejscu zaznaczyć, że ten sam adres można zobaczyć/ustawić (domyślnie przydzielane są one automatycznie) w *Address Editor*, w *Vivado*. Co więcej, niekiedy dzieje się tak, że adresy z Vivado do SDK przekazywane są niepoprawnie – zatem należy pamiętać, że można to tutaj sprawdzić.
4. Dodajemy inicjalizację *VTC* (przed pętlą `while`).

```
// Inicjalizacja VTC
Xil_Out32(XPAR_V_TC_0_BASEADDR,0xC);
```

5. Następnie w pętli odczyt rozdzielczości strumienia wizyjnego:

```
// Pobranie rozdzielczosci
int hsize_input = Xil_In32(XPAR_V_TC_0_BASEADDR+0x0030);
int vsize_input = Xil_In32(XPAR_V_TC_0_BASEADDR+0x0034);

// Pobranie rozdzielczosci aktywnej
int active_input = Xil_In32(XPAR_V_TC_0_BASEADDR+0x0020);
int vsize_active_input = (active_input & 0xFFFF0000) >> 16;
int hsize_active_input = active_input & 0x0000FFFF;

// Wyswietlenie komunikatu
xil_printf("Input␣video␣stream:␣␣␣%d␣x␣%d␣\n\r",hsize_input,vsize_input
    );
xil_printf("Input␣active␣video␣stream:␣␣%d␣x␣%d␣\n\r",
    hsize_active_input,vsize_active_input);
sleep(1);
```

6. Trzeba dołączyć:

```
#include "sleep.h"
```

7. Po wgraniu nowej aplikacji, w terminalu *CuteCom* powinniśmy zobaczyć informację o rozdzielczości strumienia wizyjnego. Oczywiście tor wizyjny również powinien działać.

3 — PS – PL z AXI4-Lite i bankiem rejestrów

3.1 Wstęp

]

Najprostszą, obok pinów EMIO (ang. *Extended Multiplexted Input/Output*, formą wymiany informacji pomiędzy systemem procesorowym, a częścią rekonfigurowalną jest magistrala AXI4 w wersji *Lite* (tj. bez wsparcia trybu burst). Vivado dostarcza szablon modułu, który stanowi bank rejestrów i może zostać użyty np. do wymiany parametrów pomiędzy aplikacją uruchomioną na PS, a modułami w części FPGA. Reprezentatywnym przykładem jest wyświetlanie prostokąta otaczającego na strumieniu wideo, którego współrzędne wyznaczane są w procesorze (jest to potrzebne w naszej aplikacji docelowej). Dodatkowo dodamy odczytywanie stanu przycisków jako przykład komunikacji PL-PS.

3.2 Realizacja

1. Zaczynamy pracę w poprzednim projekcie tj. aplikacji „hello-world” w wersji dla C++. Proszę nie zapomnieć o robieniu kopii zapasowych działających wersji.
2. Otwieramy program Vivado i tworzymy nowy moduł IP do komunikacji po AXI.
   - Wybieramy *Tools->Create and Package New IP, Next*,
   - Wybieramy*Create a new AXI4 Peripherial*, nazwijmy go `parameter_register`. Proszę zwrócić uwagę, gdzie zostanie on utworzony.
   - Na następnym ekranie wybieramy:
     - Name -> SOO_AXI
     - Interface Type -> Lite (chcemy moduł AXI4-Lite),
     - Interface Mode -> Slave (*masterem* będzie PS),
     - Data Width (Bits) -> 32,
     - Memory Size (Bytes) -> 64
     - Number of Registers -> 10 (na przyszłość).
   - Uruchamiamy generację modułu (*Next, Finish*).
3. Utworzony moduł dodajemy do schematu (tj. *Add IP* i wyszukujemy po nazwie).
4. Następnie uruchamiamy *Run Connection Automation*. Otworzy się okno dialogowe. Ustawienia pozostawiamy domyślne, ale proszę zwrócić uwagę do czego zostanie moduł podpięty (*Master*).

5. Przystępujemy do edycji modułu `parameter_register`. Klikamy na niego PPM i wybieramy opcję *Edit in IP Packager*. Otworzy się nowe Vivado z tymczasowym projektem (dla przypomnienia tak w Vivado obsługuje się „spakowane" moduły). Sam moduł składa się z dwóch plików v. Do obu musimy dodać odpowiedniej wyjścia:
   (a) W pliku nadrzędnym (tu `parameter_register_v1_0.v`), w sekcji `// Users to add ports here` dodajemy dwa porty: wyjściowy 64-bitowy port `bbox0` oraz wejściowy 4-bitowy port `sw`. Dla przypomnienia: `output wire [63:0] bbox0` oraz `input wire [3:0] sw`.
   (b) Następnie do instancji modułu `parameter_register_v1_0_S00_AXI` (dolna część pliku) dodajemy podłączenia do tych portów (`.sw(sw), .bbox0(bbox0)`). Uwaga. Proszę pamiętać, że po ostatnim przypisaniu ma nie być przecinka.
   (c) W pliku `parameter_register_v1_0_S00_AXI` dodajemy takie same porty.
   (d) W wolnej chwili (albo tzw. międzyczasie, jak projekt się będzie budował) zachęcamy do analizy tej w sumie dość prostej maszyny stanu.
   (e) Komunikacja PS-PL. W sekcji `// Add user logic here` tj. na samym dole dodajemy przypisanie dwóch wybranych przez nas rejestrów do sygnału `bbox0`. Dla przypomnienia: `assign bbox0[63:32] = slv_reg0;`.
   (f) Komunikacja PL-PS. Ten przypadek jest nieco bardziej złożony. Wybieramy rejestr np. `slv_reg3`. W maszynie stanu, zakomentowywujemy linijkę kodu, w której następuje przypisanie. W ten sposób rejestr staje się „tylko do odczytu". Odszukujemy fragment, gdzie dane z rejestrów `slv_` są odczytywane i wpisywane do rejestru `reg_data_out`. Modyfikujemy fragment kodu, tak aby wpisać tam stan przełączników (`sw`). Pozostałe bity uzupełniamy zerami – `28'b0,sw`.
   (g) Po skończonej edycji przechodzimy do zakładki *Package IP*. Jeśli w kolumnie *Packaging Steps* nie ma zielonego symbolu w poszczególnych pozycjach, to trzeba je otworzyć i zwykle kliknąć w górną cześć okna (coś ustawić lub połączyć). Ostatecznie przechodzimy do *Review and Package* i naciskamy *Re-Package IP* oraz akceptujemy zamknięcie projektu.
6. W „podstawowym" Vivado pojawi się na żółtym pasku informacja, że moduł powinien zostać uaktualniony. Proszę wybrać *Show IP Status* (ta przydatna opcja jest też dostępna w *Tools->Report->Report IP Status*). Otworzy się okno w dolnej części Vivado. Należy wybrać, `Rerun`, a później `Upgrade Selected`. Następnie należy też zaakceptować *Generate Output Products*. Ew. ostrzeżenie *Critical warning* proszę zignorować – można przeczytać informację w podanym pliku, ale dotyczy ona faktu pojawienie się nowych portów. Pojawi się też drugie ostrzeżenie o niepodpiętych portach.
7. Dla `sw` należy utworzyć port. Dla przypomnienia – najechać na port modułu, PPM i wybrać `Create Port`. Należy też dokonać odpowiedniej modyfikacji pliku *xdc*
8. Następnie należy dodać moduł wyświetlający prostokąty otaczające.
   - pobieramy archiwum *draw_bbox.zip*,
   - kopiujemy folder *draw_bbox* do katalogu *ip_repo*,
   - odśwież IP Catalog. Powinien się pojawić nowy moduł (User Repository->User IP),
   - należy go podłączyć pomiędzy *AXI4-Stream to Video Out*, a *RGB to VGA*. Uwagi:
     – interfejsy `vid_io_out` i `vid_in` rozszerzyć. Sygnał `vid_active_video` to to samo co `de`.
     – `clk` – `PixelClk`,
     – `bbox0` – z naszym modułem *parameter_register*.
9. Uruchamiamy implementację oraz generację pliku bit. Uwaga. Istnieje opcja **przyspieszenia** tego procesu. Klikając PPM na *Implementation* i wybierając *Implementation settings* możemy ustawić strategię implementacji na **Flow_Quick**. Proces będzie istotnie szybszy,

kosztem spełnienia ograniczeń czasowych (tj. raczej nie będą one spełnione). Tym niemniej doświadczenie pokazuje, że system zwykle działa. Należy jednak pamiętać, że jeśli działać nie będzie, to powyższe postępowanie może być przyczyną.

10. Eksportujemy projekt do SDK.
11. W pliku *xparameters.h* powinien pojawić się nasz moduł z właściwym adresem. Uwaga 1, jeśli plik był otwarty, to trzeba go odświeżyć – F5. Uwaga 2, jeśli pojawią się „dziwne" błędy w SDK, to najlepiej jest je zamknąć i uruchomić raz jeszcze. Czasami trzeba też zresetować kartę.
12. Możemy teraz przetestować stworzoną logikę. Zapis do rejestru to funkcja: `Xil_Out32(adres,wartosc)`, a odczyt to `x = Xil_In32(adres)`. Jako adres najwygodniej podać odpowiednią nazwę modułu z pliku `xparameters.h`. W przypadku rejestru do odczytu, do wartości adresu należy dodać $\times 4$, gdzie $\times$ to numer rejestru (bo 1 rejestr to 4 bajty danych). Proponuje się w pętli odczytać wartość z przełączników i wysłać na konsolę.
13. Przesyłanie danych PS-PL i jednocześnie wyświetlanie prostokąta otaczającego testujemy następującym kodem:

```
// Wyswietlenie bbox'a
int x2 = 200;
int x1 = 100;
int y2 = 500;
int y1  = 50;
Xil_Out32(XPAR_PARAMETER_REGISTER_0_S00_AXI_BASEADDR + 0, (x2 << 16) +
    x1);
Xil_Out32(XPAR_PARAMETER_REGISTER_0_S00_AXI_BASEADDR + 4, (y2 << 16) +
    y1);
```

    Jeśli wszystko zostało dobrze zrobione, to na ekranie powinien pojawić się zielony prostokąt.
14. Uwaga. Jeśli nasz system działa, to robimy kopię zapasową projektu.

4 — PS – PL z BRAM AXI

Opisana w rozdziale 3 metoda jest prosta, ale ma ograniczenia. Rozważamy aplikację wieloskalowego wykrywania obiektów. Dla ustalenia uwagi niech system zawiera 4 detektory w logice reprogramowalnej. Każdy z nich generuję listę kandydatów w postaci parametrów prostokąta otaczającego. Wyniki te trzeba poddać filtracji (np. eliminacja wielokrotnych wykryć tego samego obiektu) i integracji (jeden wynik na obrazie wyjściowym). Opisane operacje dużo łatwiej zrealizować w systemie procesorowym (analiza prowadzona jest dla kilku, kilkunastu prostokątów, ale może być złożona – odpowiadająca maszyna stanu dość skomplikowana). Powstaje pytanie jak przesłać dane z PS do PL. Poznanego wcześniej mechanizmu raczej nie zastosujemy, bo liczba rejestrów jest stała i ograniczona, a liczba danych zmienna.

Innym rozwiązaniem jest zapisanie tych danych do modułu pamięci BRAM i odczytanie ich po stronie procesora – zostanie to zademonstrowane w ramach niniejszego ćwiczenia. W przykładzie, zaimplementujemy przesyłanie niewielkiego fragmentu obrazu z PL do PS poprzez BRAM.

1. W Vivado rozwijamy dalej nasz projekt. Przypominamy – jeśli poprzednia wersja działa to robimy kopię zapasową.
2. W pierwszym kroku dodajemy moduł *AXI BRAM Controller* – kontroler pamięci blokowej RAM w standardzie AXI oraz sam moduł RAM (*Block Memory Generator*).
3. Zaczynamy od konfiguracji BRAM. Wybieramy: Mode->Stand Alone. Memory Type -> True Dual Port RAM. Port A Options -> Write/Read Depth 4096, rozmiar 32. Port B -> ustawi się automatycznie. Kończymy konfigurację.
4. Przechodzimy do konfiguracji modułów *AXI BRAM Controler*. W obu ustalamy liczbę interfejsów na 1 (*Number of BRAM interfaces*).
5. Uruchamiamy konfigurację procesora. Przechodzimy do zakładki *PS-PL Configuration*. Uruchamiamy oba interfejsy *master* tj. *AXI Non Secure Enablement -> M AXI GP0 i M AXI GP1*. No procesorze pojawi się port *M_AXI_GP1*.
6. Uruchamiamy *Run Connection Automation*.
   (a) axi_bram_ctrl_0:
      - BRAM_PORT_A – podłączamy do pamięci BRAM (domyślnie),
      - S_AXI — podłączamy do portu M_AXI_GP1 – master AXI po stronie procesora.
7. Kolejny etap to dodanie zapisu fragmentu obrazu do pamięci RAM. Otwieramy nowe

Vivado i tworzymy projekt np. *write_video_bram*. Tworzymy plik o takiej samej nazwie. Ponadto otwieramy do edycji moduł *draw_bbox* – będziemy się na nim wzorować.

8. Po pierwsze kopiujemy interfejs wejściowy (bez *bbox* oraz parametry). Potrzebne sygnały wyjściowe należy „odczytać" z portu BRAM (adres, dane do zapisu, reset, *eneble* oraz *write eneble (we)*). Po drugie kopiujemy sekcję odpowiedzialną za generację liczników (maszyna stanów i deklaracje). Oczywiście, teoretycznie można by to uwspólnić pomiędzy oba moduły, ale byłaby to duża komplikacja.
9. Potrzebujemy teraz logikę, która nam zapisze fragment obrazu. Załóżmy, że o rozmiarze 64x64. Potrzebujemy rejestru na adres, dane oraz *write enable*. Wewnątrz instrukcji `always` sprawdzamy dwa warunki. Jeśli sygnał `vsync_in` ma wartość 1 (nowa ramka), to resetujemy adres oraz ustawiamy na zero sygnał `we`. Jeśli liczniki spełniają nasze założenia tj. są oba mniejsze od 64 (jak ktoś chce pobrać inny fragment to musi odpowiednio zmodyfikować ten warunek) to przypisujemy do rejestrów dane (najstarszy bajt ustalamy na 0), inkrementujemy adres oraz ustawiamy `we` na 1. W przeciwnym przypadku (tj. jesteśmy poza zakresem) ustawiamy `we` na 0. Na końcu rejestry oraz pozostałe sygnały przypisujemy do wyjść (reset na zero, `en` na 1).
10. Pakujemy stworzony moduł (*Create and Package New IP*). Pozostałe opcje domyślne.
11. Wracamy do „bazowego" Vivado. W *IP Catalog* dodajemy ścieżkę do stworzonego modułu (*Add Repository*), a następnie moduł do projektu. Wejścia podłączamy analogicznie do *draw_bbox*, a wyjście do BRAM. Zegar portu BRAM to zegar piksela.
12. Wykorzystamy narzędzie ILA (Integrated Logic Analyzer) do sprawdzenia, czy nasza maszyna działa poprawnie. W tym celu dla połączeń `din`, `addr` oraz `we` zaznaczamy opcję *Debug* (dostępna pod PPM). Pojawi się opcja *Run Connection Automation*, uruchamiamy ją, zaznaczamy wszystkie sygnały. Na schemacie pojawi się moduł *System ILA*.
13. Przechodzimy do generacji pliku bit, eksportujemy konfigurację i na końcu uruchamiamy SDK (doświadczenie uczy, że lepiej jest je zamknąć i uruchomić, niż uaktualniać tak na bieżąco – nie zawsze wszystko się poprawnie odświeży). Uwaga. Może zdarzyć się też tak, że SDK z Vivado nie będzie chciało się uruchomić (np. przy przenoszeniu z komputer na komputer). Wtedy jednym z wyjść jest skasowanie folderu „.sdk" z projektu i wygenerowanie go na nowo. Oczywiście wcześniej trzeba skopiować pliki źródłowe (z cyklu uroki Vivado/SDK).
14. W pliku `xparameters.h` pojawią się parametry stworzonego kontrolera. Nas interesuje jego adres. Wykonujemy następujący test. W pętli odczytujemy nasz obrazek. Stosujemy znane już `Xil_In32`. Proszę zwrócić uwagę na odpowiednie adresowanie (inkrementacja co 4). Wyniki należy zapisać na konsolę i niewielkim nakładem pracy wczytać do np. Matlaba i stworzyć z nich obraz.
15. Aby wykorzystać możliwości jakie daje ILA należy postąpić nieco inaczej. W Vivado otworzyć *Hardware Manager* i tam zaprogramować układ, a następnie dać *Refresh Device*. Powinien pojawić się interfejs ILA. Jego obsługa jest raczej intuicyjna. Do okna z przebiegami można dodawać kolejne sygnały. Osobno używa się *triggerów* – dodaje się sygnał i ustala warunek przy jakim ma on zadziałać. Proszę „podglądnąć" adresy oraz dane, a za *trigger* ustawić sygnał *we*. Uwaga 1. Układ Zynq programować PL w Vivado, a PS w SDK. Uwaga 2. ILA jest bardzo przydatne w przypadku, kiedy logika zweryfikowana symulacyjne nie działa w sprzęcie (niestety czasem tak bywa). Należy tylko wybrać odpowiednie „punkty pomiarowe", które pozwolą szybko zweryfikować, który fragment i w jakim przypadku nie działa poprawnie.

5 — Przerwania

Niekiedy chcielibyśmy uruchomić przetwarzanie pewnych danych na procesorze w określonym momencie – np. po zakończeniu ramki obrazu. W tym celu najlepiej wykorzystać mechanizm przerwań z części PL, który dostępny jest w rozważanym systemie. W ramach tego ćwiczenia poznamy jak uruchomić takie przerwanie.

1. Dalej rozwijamy nasz projekt, zatem warto zrobić **kopię zapasową**.
2. Otwieramy konfigurację procesora: *Interrupts->Fabric ..->PL-PS...*. Zaznaczamy IRQ. Po zatwierdzeniu dodatkowy port pojawi się na procesorze (*IRQ*).
3. Podpinamy do niego sygnał `vsync`. Wybieramy *Generate Bitstream*.
4. Przechodzimy do SDK. Sprawdzamy, czy w `xparameters.h` pojawiły się nowe linie (najprościej CTRL+F i poszukać „_INTR”).
5. Kod inicjalizujący obsługę przerwań jest dość rozbudowany. Dla porządku zamieszono go poniżej, ale lepiej skorzystać z pliku dostępnego na stronie kursu. Na jego podstawie proszę przygotować aplikację, w która na konsolę wyśle informację o pojawieniu się nowej ramki. Docelowo taką funkcjonalność wykorzystamy do przesyłania parametrów obiektów.

Przykład obsługi przerwań: Interrupt handling example:

```
#include "xparameters.h"
#include "platform.h"
#include "xil\_printf.h"
#include "xil\_io.h"
#include "xscugic.h"

XScuGic InterruptController;      /* Instance of the Interrupt Controller */
static XScuGic_Config *GicConfig; /* The configuration parameters of the
    controller */

extern char inbyte(void);

void Btn_InterruptHandler(void *data) {
        xil_printf("␣Interrupt␣acknowledged␣\n␣\r␣");
        xil_printf("\r\n");
        xil_printf("\r\n");
}

int ScuGicInterrupt_Init() {
```

```
        int Status;
        /*
        * Initialize the interrupt controller driver so that it is ready to
        * use.
        * */
        Xil_ExceptionInit();

        GicConfig = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID);
        if (NULL == GicConfig) {
                return XST_FAILURE;
        }

        Status = XScuGic_CfgInitialize(&InterruptController, GicConfig,
                GicConfig->CpuBaseAddress);

        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        /*
        * Setup the Interrupt System
        * */

        /*
        * Connect the interrupt controller interrupt handler to the hardware
        * interrupt handling logic in the ARM processor.
        */
        Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_IRQ_INT,
                (Xil_ExceptionHandler) XScuGic_InterruptHandler,
                (void *) &InterruptController);

        /*
        * Connect a device driver handler that will be called when an
        * interrupt for the device occurs, the device driver handler
    performs
        * the specific interrupt processing for the device
        * since the xparameters.h file doesnt detect the external interrupts
    , we have to manually
        * use the IRQ_F2P port numbers ; 91, 90, ect..
        */
        Status = XScuGic_Connect(&InterruptController, XPAR_FABRIC_BTN_INTR,
     (Xil_ExceptionHandler)Btn_InterruptHandler, (void *)&InterruptController
    );
        XScuGic_Enable(&InterruptController, XPAR_FABRIC_BTN_INTR);

        /*
        * Enable interrupts in the ARM
        */
        Xil_ExceptionEnable();

        XScuGic_SetPriorityTriggerType(&InterruptController,
    XPAR_FABRIC_BTN_INTR, 0xa0, 3);

        if (Status != XST_SUCCESS) {
                return XST_FAILURE;
        }

        return XST_SUCCESS;
}
```

```
int main()
{
        int xstatus;
        init_platform();

        xstatus = ScuGicInterrupt_Init();
        if (xstatus != XST_SUCCESS) {
                return XST_FAILURE;
        }
        xil_printf("GIC Init Success\n\r");


        while(1) {
        }

        cleanup_platform();
        return 0;
}
```

6 — Detekcja porzuconych bagażów – koncepcja

System detekcji porzuconych obiektów składa się z następujących modułów:

- segmentacja obiektów pierwszoplanowych,
- segmentacja obiektów ruchomych,
- filtracja maski obiektów (opcjonalna),
- indeksacja,
- analiza wykrytych obiektów.

W rozdziale zostaną one krótko omówione – będzie to podstawa do stworzenia modelu programowego.

6.1 Segmentacja obiektów pierwszoplanowych

Na potrzeby niniejszego systemy wykorzystana zostanie prosta metoda znana jako *sigma-delta* lub aproksymacja mediany. Jej działanie opisuje wzór:

$$BG_{i+1} = BG_i + sign(I_i - BG_i) \tag{6.1}$$

gdzie: BG to model tła, I – bieżąca ramka. Możemy założyć, że obraz i model są kolorowe (format RGB) oraz, że składowe są niezależne (co nie jest prawdziwe).

Segmentacja sprowadza się do prostego odejmowania obrazu od modelu tła. Do obliczenia odległości można wykorzystać normę L1 (łatwiejsza implementacja sprzętowa). Należy też dobrać stały próg binaryzacji.

Uwaga. Proszę rozważyć dwa podejścia: liberalne (aktualizujemy wszystko) i konserwatywne (aktualizujemy tylko elementy uznane za tło). Przy czym w obu przypadkach najpierw wykonujemy segmentację, a dopiero później aktualizację.

Efektem tego etapu powinna być maska obiektów pierwszoplanowych.

6.2 Segmentacja obiektów ruchomych

Na potrzeby niniejszego systemy wykorzystana zostanie najprostsze odejmowanie dwóch kolejnych ramek w RGB (norma L1). Informacja o ruchu jest nam potrzebna na etapie analizy – obiekt porzucony to **nieruchomy obiekt pierwszoplanowy**.

6.3 Filtracja

Obie maski – obiektów pierwszoplanowych oraz ruchomych można poddać jednej ze znanych metod filtracji obrazów binarnych np. otwarcie/zamknięcie morfologiczne lub mediana.

6.4 Indeksacja

W rozważanej aplikacji potrzebne nam jest rozwiązanie *hybrydowe*. Typową indeksację wykonujemy na masce obiektów pierwszoplanowych – wyliczamy parametry: pole, środek ciężkości i prostokąt otaczający. Następnie dla wykrytych obiektów określamy liczbę „ruchomych" pikseli (tj. należących do maski obiektów ruchomych). W ten sposób wiemy, czy obiekt się rusza.

6.5 Analiza wykrytych obiektów

1. Po pierwsze odrzucane są obiekty, które są mniejsze niż zadany próg. Wynika to z (nieuniknionej) obecności szumu na masce obiektów pierwszoplanowych. Nawet po zastosowaniu wspomnianej filtracji medianowej czy morfologicznej.
2. Po drugie obiekty, które uznane zostaną za statyczne (odpowiednio mało pikseli się porusza) poddawane są dalszej analizie.
3. Dopasowywanie obiektów. Koncepcja: mamy listę obiektów statycznych bieżących i z poprzedniej ramki. Chcemy obiekty bieżące dopasować do tych poprzednich. Dopasowanie opieramy na analizie rozmiaru części wspólnej prostokątów otaczających. Uwaga! Możliwy jest również wariant, w którym analizujemy wprost maski obiektów, ale jest on bardziej skomplikowany (szczególnie w kontekście implementacji sprzętowej). Poza tym, bagaż jest zwykle zbliżony do prostokąta.
   Opis obiektu stycznego:
   - środek ciężkości,
   - pole,
   - prostokąt otaczający,
   - ..widoczny" – flaga,
   - „jak długo statyczny" – licznik (w ramkach).

   Realizacja:
   - w pierwszym kroku zakładamy, że wszystkie obiekty z poprzedniej ramki mają status „niewidoczne". Jest to potrzebne do usuwania ze listy takich, które zniknęły (wbrew pozorom „chwilowych" obiektów statycznych jest całkiem sporo – np. „oderwany" fragment cienia).
   - następnie w ramach podwójnej pętli, każdy bieżący obiekt próbujemy dopasować do każdego z poprzednich. Nie jest to może najbardziej eleganckie rozwiązanie, ale w praktyce takich obiektów jest bardzo mało (do kilkunastu) zatem nie wystąpią problemy z wydajnością. Obliczamy:
     – pole części wspólnej dwóch prostokątów,
     – stosunek części wspólnej do pola obiektu historycznego,
     – stosunek pól obu obiektów.

     Sprawdzamy warunki na dopasowanie:
     – po pierwsze interesuje nas „najbardziej dopasowany" obiekt (największe pole części wspólnej),
     – jednakże ma też być podobnego rozmiaru. W przeciwnym przypadku przesłaniający większy obiekt (np. człowiek) „dziedziczyłby" historię po mniejszym (plecaku).

     Jeśli warunki są spełnione to zapisujemy numer obiektu.

- W kolejnym kroku mamy dwie możliwości:
  - nie udało nam się dopasować obiektu – oznacza to, że obiekt jest nowy i taki dodajemy go do listy,
  - udało się dopasować – dokonujemy aktualizacji parametrów istniejącego (pole, prostokąt otaczający oraz licznik jak długo pozostawał statyczny). Zaznaczamy ponadto, że jest on widoczny.
- Usuwamy ze struktury obiekty niewidoczne.
- Detekcja porzuconego bagażu sprowadza się do analizy licznika (przykładowo 200 kolejnych ramek).

6.6 Dyskusja możliwych usprawnień systemu

Zaprezentowana, podstawowa wersja systemu, działa poprawnie tylko w bardzo prostych przypadkach. W szczególności dość „problematyczne" są dwie sytuacje:

1. Przesłanianie obiektów. Jeśli analizowany obiekt, widoczny powiedzmy od 100 klatek, zostanie chwilowo przesłonięty, to licznik zostanie wyzerowany. W pesymistycznym wariancie taki obiekt nigdy nie zostanie uznany za potencjalnie niebezpieczny. Z taką sytuacją mamy przykładowo do czynienia jak bagaż jest na drugim planie, a na pierwszych przemieszają się osoby.
   Chcielibyśmy zatem, aby chwilowe przesłonięcie nie „kasowało" licznika. Uzyskanie takiej funkcjonalności jest możliwe np. poprzez wprowadzenie dodatkowego statusu (flagi) – *przesłonięty* . Jeśli obiekt nie jest widoczny w danej ramce, to zamiast go kasować (jak w podstawowej wersji) uznajemy, że jest przesłonięty (przy czym warto tak postępować dla obiektów obserwowanych przez kilkanaście/kilkadziesiąt kolejnych ramek).
   Dla obiektów przesłoniętych zmniejszamy pewien licznik (może to być ten, który określa czas widoczności lub też inny tzw. *time to live* – TTL). Jeśli osiągnie on małą wartość, to obiekt usuwamy.
   Takie postępowanie, w pewnym stopniu, zabezpiecza nas przed opisaną sytuacją przesłonięcia.
2. Ghosty.
   Proponowany algorytm z tzw. konserwatywnym podejściem do aktualizacji tła zakłada (niejawnie), że pierwsza ramka obrazu jest „pusta". Oczywiście nie musi to być prawdą. Poza tym na scenie coś zawsze może się zmienić (np. ktoś przesunie kosz na śmieci). Dlatego bardzo pożądany jest mechanizm eliminacji fałszywych detekcji.
   Jednym z możliwych podejść jest wykorzystanie układu krawędzi maski, sceny oraz modelu tła. Dla tych trzech obrazów (de-facto ich fragmentów) liczymy krawędzie (Sobel/Canny). Następnie „patrzymy", czy układ krawędzi obiektu (tj. maski binarnej) jest bardziej zbliżony do układu krawędzi na bieżącej scenie, czy do modelu tła. Pierwszy przypadek oznacza, że obiekt istnieje na scenie, drugi, że mamy do czynienia z obiektem fikcyjnym (ghostem) – takie oczywiście usuwamy poprzez aktualizację.
   Uwaga. W teorii metoda jest zgrabna, elegancka i co możliwa do realizacji w FPGA. Niestety, tego typu heurystyki są zwykle „mieczem obosiecznym". Doświadczenie uczy, że pozwalają wyeliminować sporo błędów, ale też jakąś część prawidłowych obiektów. Ponadto źle działają, jeśli tło sceny jest bogate w teksturę.
3. Ludzie. Jeśli chcielibyśmy rozróżniać ludzi i inne obiekty to w OpenCV jest stosowana funkcjonalność. Warto spróbować (HOG+SVM).
4. Podsumowanie.
   - proszę traktować powyższe propozycje jako przykładowe możliwości – nie muszą to być najlepsze rozwiązania,

- gorąco zachęcam do poszukiwania własnych rozwiązań. Stosujemy podejście *problem based* tj. uruchamiamy system na sekwencji i patrzymy jak działa. Jeśli występują błędy, to po pierwsze myślimy z czego dokładnie wynikają, a po drugie jak je usunąć. Ale uwaga. Nasze poprawki trzeba sprawdzić na całym zbiorze danych (bywa żmudne). Doświadczenie uczy, że metoda poprawiająca sytuację „A", spowoduje powstanie błędów w sytuacji „B".

7 — Model programowy – C++/OpenCV

W rozdziale przedstawione zostaną kolejne kroki, które pozwolą zaimplementować opisany w rozdziale 6 system detekcji porzuconych obiektów. Uwaga zaprezentowane rozwiązanie należy traktować jako **przykładowe**. W szczególności zastosowane konstrukcje języka C++ mogą być ulepszone, a sama aplikacja zaprojektowana jako obiektowa. Jednak tutaj skupiamy się przede wszystkim na funkcjonalności.

- W pierwszym kroku uruchomimy aplikację typu „Hello world" z wykorzystaniem OpenCV. Uruchamiamy środowisko CLion. Tworzymy nowy projekt C++ (ustawienia domyślne), wybieramy folder (najlepiej stworzyć np. *ald_sw_cpp*).
- Do pliku *CMakeLists.txt* dodajemy następujący kod (plik dostępny w repozytorium):

```
# == OpenCV (3.4.5.0) / (4.0.1) configuration ==
# Where to find CMake modules and OpenCV
set(OpenCV_DIR "/opt/opencv_4_0_1/build/")   #check path
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/cmake/"
    )

find_package(OpenCV REQUIRED)
include_directories(${OpenCV_INCLUDE_DIRS})

# Add libs you need
set(OpenCV_LIBS opencv_core opencv_imgproc opencv_highgui
    opencv_imgcodecs)

# Linking
target_link_libraries(ald ${OpenCV_LIBS})
```

  Proszę zwrócić uwagę na ścieżkę oraz nazwę projektu (tu *ald*). Ustawienia trzeba uaktualnić – pojawi się komunikat na pasku powyżej kodu.
- Pobieramy ze strony kursu archiwum *pets_2016* z sekwencją testową. Do folderu z projektem rozpakowujemy sekwencję testową. Do pliku *main* dodajemy następujące biblioteki:

```
#include "opencv2/opencv.hpp"
#include "opencv2/core/core.hpp"
#include "opencv2/highgui/highgui.hpp"
#include "opencv2/imgproc/imgproc.hpp"
```

- Implementujemy wczytywanie pojedynczego obrazka. Przy czym od razu „przygotowujemy grunt" pod wczytanie kilku. Wykorzystujemy poniższy kod:

```
char buffer[100];
cv::Mat inputImage;
int iImage = 1;
sprintf(buffer,"in%06d.jpg",iImage);
std::string s_sequencePath = "../pets2006/";
inputImage = cv::imread(s_sequencePath+buffer);
cv::imshow("Input image",inputImage);
cv::waitKey(0);
```

Należy zwrócić uwagę na ścieżkę do folderu oraz instrukcję `waitKey(0)`, która pozwala wyświetlić obraz (bez niej nie zadziała) oraz zatrzymuje wykonanie programu (parametr 0, można dać 1). Pożądany efekt to wyświetlony obraz wejściowy.

- Kolejny krok to wyświetlanie sekwencji wideo. Dodajemy pętlę `for` ze zmienną *iImage*. Warto dodać od razu trzy parametry ramkę początkową, końcową oraz krok (co ile ramek) – jako `define` lub `const`.
- Możemy przystąpić do implementacji segmentacji obiektów pierwszoplanowych i aktualizacji modelu tła. Należy zaznaczyć, że w OpenCV dostępne są wbudowane metody, jednak są one zbyt zaawansowane do implementacji w FPGA w ramach zajęć (bo „ogólnie" to można je wykonać). Tworzymy obiekt *Mat* dla tła – w naszej prostej metodzie jest to po prostu obraz. Dla ustalenia uwagi niech nazywa się on *bgModel*.
  Po pierwsze implementujemy inicjalizację. Dla pierwszej ramki do modelu tła kopiujemy zawartość bieżącej ramki (metoda `clone()` z `Mat`). Dla kolejnych musimy zaimplementować segmentację i aktualizację.
- Najbardziej czytelne (bo pewnie nie najszybsze) jest podejście z dwoma pętlami *for* i wykorzystaniem operatora `at`. Odpowiedni kod pokazano poniżej:

```
typedef cv::Vec<uchar,3> vec_uchar_3;

for (int jj=0; jj < inputImage.rows; jj++) {
 for (int ii=0; ii < inputImage.cols; ii++) {
  vec_uchar_3 in_pixel = inputImage.at<vec_uchar_3>(jj,ii);
  vec_uchar_3 m_pixel = bgModel.at<vec_uchar_3>(jj,ii);
   ...
 }
}
```

Proszę zwrócić uwagę na konieczność zdefiniowania typu – wektora trzech *uchar* (tak jest wygodniej). Ponadto warto popatrzeć na sposób dostępu do piksela – pierwsza współrzędna to wiersz, a druga to kolumna (odwrotnie, niż zwykle opisuje się obraz np. podając rozdzielczość).

- Segmentacja. Implementujemy normę L1 oraz progowanie (wartość progu zdefiniować). Wyniki należy zapisać do nowego obrazka – *fgMask*. Musimy go stworzyć:

```
fgMask = cv::Mat::zeros(inputImage.rows, inputImage.cols, CV_8U);
```

Najlepiej w ramach inicjalizacji. Przykładowe przypisanie:

```
fgMask.at<uchar>(jj,ii) = 255;
```

- Aktualizacja. Implementujemy metodę *sigma-delta* – dla każdej składowej osobno. Na razie w wersji *liberalnej* tj. dla wszystkich pikseli. Uwaga. Należy zastosować konstrukcję z dwoma `if` (jeśli elementy są równe, to nic nie ma się dziać). Nową wartość modelu tła przypisujemy do naszego obrazka.
- Pierwszy test. Uruchamiamy sekwencję od ramki 1, co 1 (później można zmienić na co 5). Wyświetlamy obraz wejściowy, model tła i maskę. Obserwujemy działanie algorytmu. Zwracamy uwagę na:
  - „smużenie",

  - częściowe i całościowe przenikanie obiektów do tła,
  - *ghosty*,
  - i najważniejsze – co się dzieje z naszym bagażem.

  Wydaje się, że aktualizacja wszystkich pikseli w tym przypadku nie jest dobrym pomysłem. Spróbujemy zatem aktualizować tylko te piksele, które w ramach segmentacji zostały uznane za tło. Wymaga to niewielkiej modyfikacji kodu (na tym etapie).
- Drugi test. Zasadniczo powinno być lepiej. Jednak mogą występować pewne problemy z cieniami oraz wtapianiem się sylwetki osoby. Spróbujemy je wyeliminować w dalszych krokach. Dla tej konkretnej sekwencji naszym celem jest uzyskanie poprawnej maski plecaka (i raczej niczego więcej).
- Dodajemy segmentację obiektów ruchomych tj. odejmowanie sąsiednich ramek. Postępujemy podobnie jak wcześniej. Tworzymy obiekt *Mat* na poprzednią ramkę (*prevImage*) oraz wyniki segmentacji (*movMask*). Inicjalizujemy. Obliczamy różnicę w L1 i binaryzujemy. Na końcu pętli kopiujemy obraz bieżący do poprzedniego (metoda *copyTo*). Maskę wyświetlamy.

  Modyfikujemy również procedurę aktualizacji. Dodajemy warunek na brak ruchu – może wymagać to małej reorganizacji kodu.
- Filtracja. Dodajemy filtrację medianową maski obiektów pierwszoplanowych – funkcja `cv::medianBlur`.
- Pożądany efekt na tym etapie. Mamy do dyspozycji dwa parametry – progi stosowane przy segmentacjach. Za ich pomocą powinno udać się osiągnąć efekt w postaci „stabilnej" detekcji plecaka. Dopuszczalne są pewne niewielkie zakłócenia. Uwaga. Potencjalnie pomocna jest analiza co 3 lub co 5 ramki – ogranicza to negatywne efekty związane z przenikaniem obiektów pierwszoplanowych do tła.
- Indeksacja. W OpenCV dostępna jest wygodna funkcja: