Wydział Elektroniki i Technik Informacyjnych

Systemy komputerowe (SYKOM)

Podstawy tworzenia i uruchamiania oprogramowania dla procesora Cortex-A9 pracującego bez systemu operacyjnego za pomocą narzędzi Vitis i Vivado.

Kacper Capiga 331463
Karol Godlewski 331474

6 kwietnia 2025

Spis treści

1 Cel laboratorium

Celem laboratorium jest zapoznanie się z obsługą 32 bitowego procesora Cortex-A9 w architekturze ARM v7 oraz ze strukturą drzewa urządzeń (Device Tree Source) za pomocą oprogramowania Vivado i Vitis w wersji 2024.1 firmy Xilinx. Do realizacji laboratorium wykorzystano płytkę rozwojową PYNQ-Z2 z układem (System on Chip) Zynq-7020 z rodziny Zynq-7000 w skład którego wchodzi wspomniany procesor Cortex-A9.

2 Opis oprogramowania i sprzętu

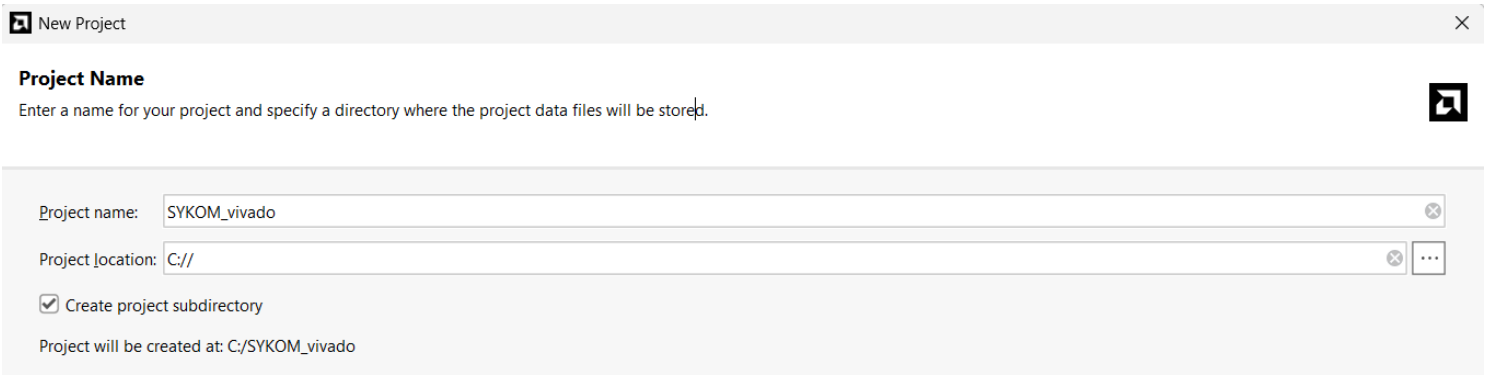
Vivado to zintegrowane środowisko projektowe firmy Xilinx do projektowania układów FPGA. Umożliwia tworzenie, symulację, syntezę oraz implementację układów cyfrowych zdefiniowanych za pomocą języków opisu sprzętu. Umożliwia generowanie plików XSA (Xilinx Shell Archive) opisujących konfigurację sprzętową, która będzie wykorzystywana w trakcie laboratorium.

Vitis to środowisko programistyczne firmy Xilinx do tworzenia oprogramowania dla systemów z procesorami wbudowanymi, takich jak Cortex-A9 w układach Zynq-7000. Umożliwia on kompilację na platformę docelową kodu w języku C, dekompilację drzewa urządzeń (zawartego w XSA) oraz przygotowanie obrazu uruchomieniowego platformy z utworzonym wcześniej oprogramowaniem.

PYNQ-Z2 to płytka rozwojowa oparta na układzie Xilinx Zynq-7020 SoC, łączącym dwurdzeniowy procesor Cortex-A9 o częstotliwości 650 MHz, kontroler pamięci DDR3, kontrolery peryferyjne (USB 2.0, SPI, UART, I2C) oraz logikę programowalną (FPGA) Artix-7. Na płytce znajduje się także 512 MB pamięci DDR3 z 16 bitową szyną, 16 MB pamięci flash Quad-SPI oraz gniazdo na kartę SD.

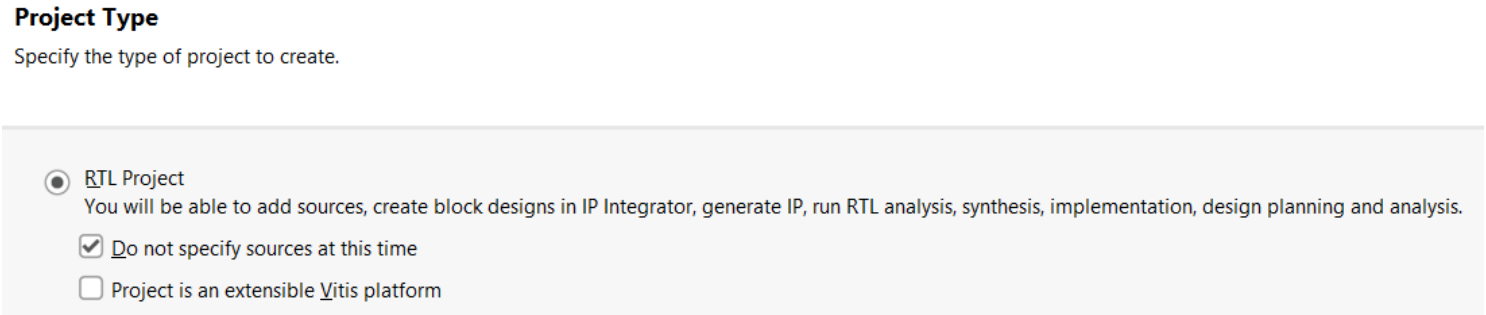
3 Przygotowanie platformy

Do przygotowania pliku z opisem wykorzystanego sprzętu użyto narzędzia Vivado. Za jego pomocą wygenerowano bitstream prostego projektu zawierającego jedynie PS (ang. *Processing System*). Po uruchomieniu narzędzia należy stworzyć nowy projekt *Create New Project* a następnie wskazać miejsce w którym będzie on przechowywany (rysunek 1).



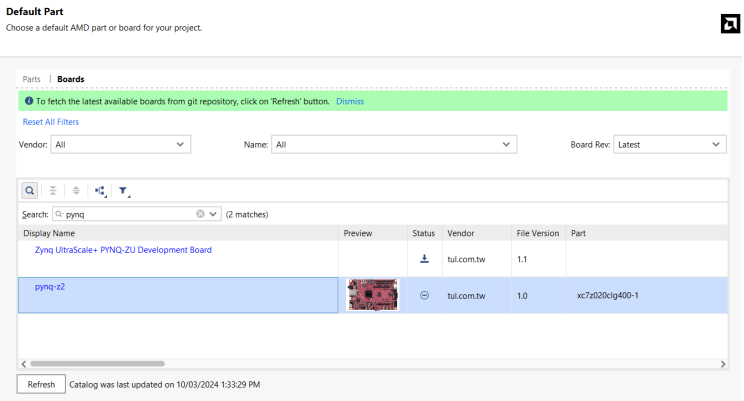
Rysunek 1: Widok kreatora nowego projektu w programie Vivado.

Przy wyborze typu projektu zaznaczono RTL project i wybrano opcję *Do not specify sources at this time* (rysunek 2), żeby nie załączać zewnętrznych źródeł do projektu.



Rysunek 2: Wybór typu projektu w oknie kreatora.

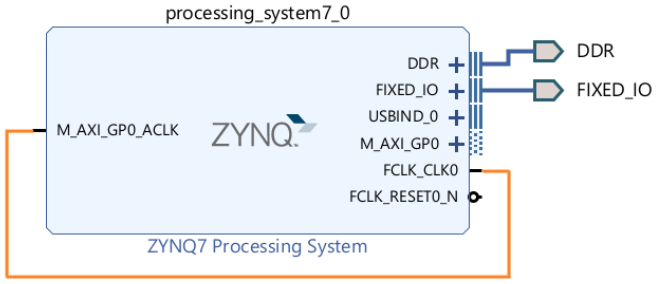
Przy wyborze platformy deweloperskiej wybrano zakładkę "Boards" i wyszukano docelową platformę *pynq-z2* (rysunek 3).



Rysunek 3: Wybór platformy docelowej

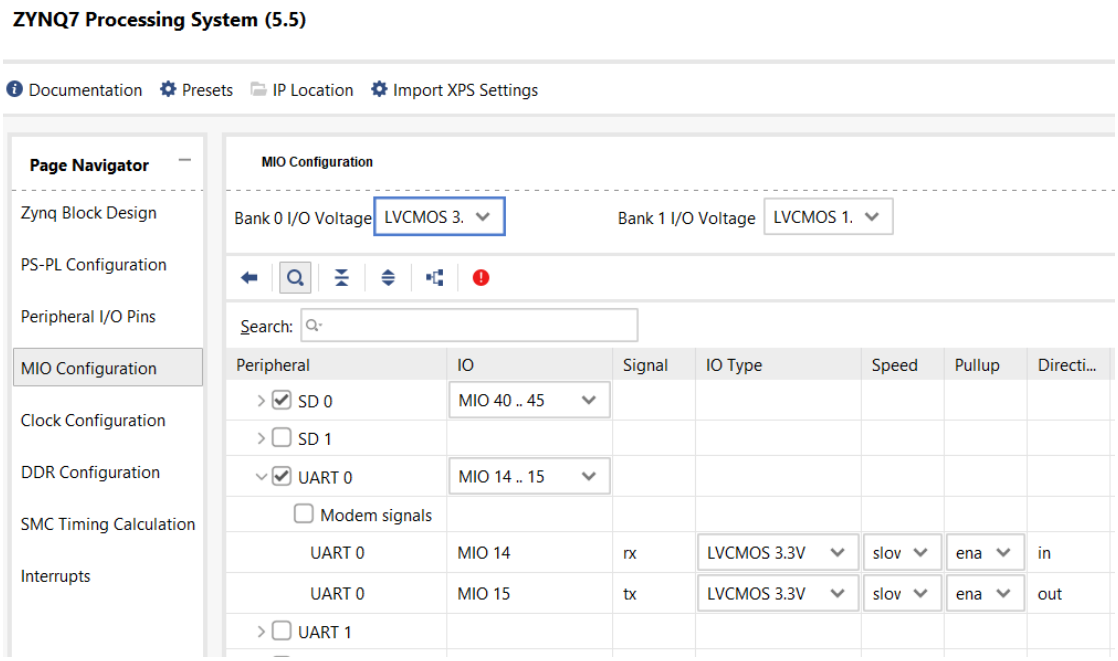
Po otworzeniu projektu w oknie *Flow Navigator* utworzono nowy schemat blokowy za pomocą przycisku *Create Block Design*. W przestrzeni roboczej pojawiło się okno *Diagram* w którym dodano nowe IP (ang. *Intellectual Property*) za pomocą skrótu klawiszowego *Ctrl+I* i wybrano *ZYNQ7 Processing System*.

Po dodaniu IP wybrano opcję *Run Block Automation*, która na podstawie wybranej platformy dostosuje projekt do jej specyfikacji. Następnie połączono główny zegar z portem *M_AXI_GP0_ACLK*. Zweryfikowano poprawność schematu blokowego za pomocą skrótu klawiszowego *F6*. Schemat blokowy powinien wyglądać jak na rysunku 4.



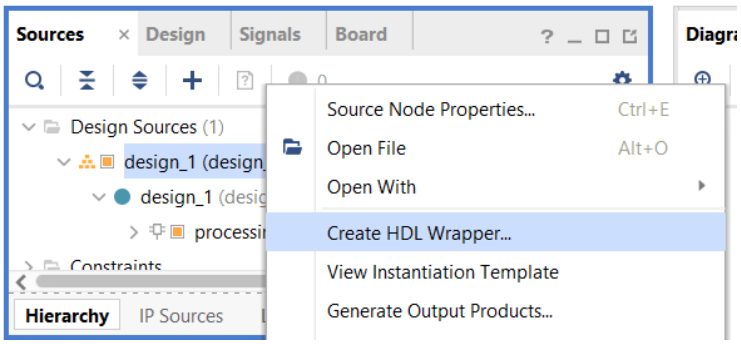
Rysunek 4: Widok schematu blokowego.

Przed wygenerowaniem bitstreamu upewniono się, że automatyczna konfiguracja włączyła UART0 i przypisała mu odpowiednie piny *Multiplexed Input Output* (MIO), co zostało przedstawione na rysunku 5. Zwrócono uwagę, że niektóre wersje programu Vivado na tym etapie nie przypisują ich automatycznie i należy zrobić to ręcznie. Numery pinów uzyskano ze strony 17 user manuala dostępnego pod adresem: https://dpoauwgwqsy2x.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf.



Rysunek 5: Przypisanie pinów MIO do UART0.

Przed generacją bitstreamu należy stworzyć jeszcze *top-level-module*. W tym celu w oknie *Sources* naciśnięto prawym przyciskiem myszy na schemat blokowy i wybrano opcję *Create HDL Wrapper...* (rysunek 6).



Rysunek 6: Tworzenie *HDL Wrapper* schematu blokowego.

Po stworzeniu wrappera w oknie *Flow Navigator* wybrano opcję *Generate Bitstream* na dole strony.

Kroki do tego momentu mogą zostać wykonane bezpośrednio przez skrypt w formacie Tool Command Language (TCL). Można je również obserwować podczas wykonywania poszczególnych kroków w oknie *Tcl Console*. Tym samym przeklikiwanie okien można zastąpić uruchomieniem skryptu Skrypt 1. Służy do tego funkcja *Tools* > *Run Tcl Script...* po której należy wybrać plik z rozszerzeniem .tcl z poniższą zawartością.

```
start_gui
create_project SYKOM_vivado C:/SYKOM_vivado -part xc7z020clg400-1
set_property board_part tul.com.tw:pynq-z2:part0:1.0 [current_project]
create_bd_design "design_1"
update_compile_order -fileset sources_1
startgroup
create_bd_cell -type ip -vlnv xilinx.com:ip:processing_system7:5.5 processing_system7_0
endgroup
apply_bd_automation -rule xilinx.com:bd_rule:processing_system7 -config {make_external
    "FIXED_IO, DDR" apply_board_preset "1" Master "Disable" Slave "Disable" } [get_bd_cells
    processing_system7_0]
connect_bd_net [get_bd_pins processing_system7_0/FCLK_CLK0] [get_bd_pins
    processing_system7_0/M_AXI_GP0_ACLK]
save_bd_design
make_wrapper -files [get_files
    C:/SYKOM_vivado/SYKOM_vivado.srcs/sources_1/bd/design_1/design_1.bd] -top
add_files -norecurse
    c:/SYKOM_vivado/SYKOM_vivado.gen/sources_1/bd/design_1/hdl/design_1_wrapper.v
launch_runs impl_1 -to_step write_bitstream -jobs 4
```

Skrypt 1: Skrypt tcl realizujący kroki wykonane do generacji bitsream'u

Po zakończeniu generacji bitstream'u wyeksportowano plik zawierający opis sprzętu w formacie .xsa za pomocą *File* > *Export* > *Export Hardware...* Można to również zrobić za pomocą skryptu Skrypt 2. Należy zapamiętać miejsce w którym zapisano plik .xsa.

```
write_hw_platform -fixed -include_bit -force -file C:/SYKOM_vivado/design_1_wrapper.xsa
```

Skrypt 2: Skrypt tcl eksportujący opis sprzętu z załączonym bitstream'em do pliku .xsa

Po eksporcie należy otwarto narzędzie Vitis, w którym utworzono aplikację, która została uruchomiona na zdefiniowanym w pliku .xsa systemie, w tym celu wybrano *Tools* > *Launch Vitis IDE* co otworzyło narzędzie w nowym oknie.

4 Uzupełnienie funkcji w kodzie

Aplikacja wykonująca zadane polecenie została podzielona na funkcje umieszczone w oddzielnych plikach. Główny plik zawierający wywołania pozostałych funkcji to *main.c*, w którym znajduje się funkcja *main*. Korzysta on z pliku nagłówkowego *functions.h*, który deklaruje wszystkie używane funkcje i umożliwia ich integrację.
Funkcja *main* w pierwszej kolejności wywołuje funkcję *send_UART_chars*, aby wyświetlić aktualny dzień, godzinę oraz ścieżkę, w której znajdował się plik *main.c* w momencie kompilacji. Następnie funkcja wypisuje na konsoli identyfikator urządzenia w formacie heksadecymalnym i kończy działanie symulacji, wywołując funkcję *exit_simulation*.

```c
#include "functions.h"

int main() {
    send_UART_chars("\nARMv7-APP: SYKOM lab.1 ("__FILE__", "__DATE__", "__TIME__")\n");
    send_UART_chars("0x");
    print_ulong(get_id());
    send_UART_chars("\n");
    exit_simulation();
    return 0;
}
```

Kod 1: main.c

Makro *RAW_SPACE(addr)* umożliwia bezpośredni dostęp do pamięci pod wskazanym adresem, traktując go jako wskaźnik do zmiennej typu *unsigned long* oraz uniemożliwiając kompilatorowi optymalizację operacji odczytu i zapisu pod podany adres dzięki słowu kluczowemu *volatile*. Pozostałe funkcje odpowiadają za komunikację przez UART, wypisywanie wartości liczbowych, pobranie identyfikatora oraz zakończenie symulacji.

```c
#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#define RAW_SPACE(addr) (*(volatile unsigned long *)(addr))

void send_UART_char(char s);
int send_UART_chars(const char *s);
void exit_simulation();
void print_ulong(unsigned long v);
unsigned long get_id();

#endif
```

Kod 2: functions.h

Zadaniem funkcji *send_UART_char* oraz *send_UART_chars* jest odpowiednie wypisanie przez UART jednego znaku lub ciągu znaków, korzystając z odpowiednich rejestrów XUART. Funkcja *send_UART_char* czeka na dostępność miejsca w FIFO, monitorując stan rejestru statusu *XUARTPS_SR*. Gdy miejsce w FIFO jest dostępne, znak jest zapisywany do rejestru *XUARTPS_FIFO*. Funkcja *send_UART_chars* wykorzystuje *send_UART_char* do wysyłania każdego znaku z łańcucha, automatycznie dodając znak powrotu karetki $\backslash r$ po wykryciu znaku nowej linii $\backslash n$, co jest wymagane przez niektóre urządzenia komunikacyjne.

```c
#include "include/functions.h"

#define XUART0_PS 0xe0000000
#define XUARTPS_SR_OFFSET (XUART0_PS + 0x0000002C)
#define XUARTPS_SR_TXFULL 1<<4
#define XUARTPS_FIFO_OFFSET (XUART0_PS + 0x00000030)

void send_UART_char(char s) {
  while ((RAW_SPACE(XUARTPS_SR_OFFSET) & XUARTPS_SR_TXFULL) > 0);
  RAW_SPACE(XUARTPS_FIFO_OFFSET) = (unsigned int) s;
}

int send_UART_chars(const char *s) {
    while(*s != '\0') {
            if(*s == '\n') send_UART_char('\r');
            send_UART_char(*s);
            s++;
    }
    return 0;
}
```

Kod 3: send_UART.c

Funkcja *print_ulong* korzysta z wcześniej opisanej funkcji *send_UART_char*, do której przekazuje kolejne znaki ASCII tworzące cyfry przekazanej liczby w postaci heksadecymalnej. Funkcja ta przekształca liczbę na ciąg znaków, reprezentujących 4-bitowe fragmenty liczby, począwszy od MSN (Most Significant Nibble). Funkcja *val2hex* odpowiada za konwersję pojedynczych 4-bitowych fragmentów liczby na odpowiadające im znaki heksadecymalne ('0'-'9' oraz 'A'-'F').

```c
#include "include/functions.h"

char val2hex(unsigned int i) {
    if (i < 10) return '0' + i;
    else return 'A' + (i - 10);
}

void print_ulong(unsigned long v) {
    int i;
    for(i = 0 ; i < 8; i++) {
        char x = (v >> (28 - 4*i)) & 0xF;
        send_UART_char(val2hex(x));
    }
}
```

Kod 4: print_ulong.c

Funkcja *get_id* zwraca liczbę typu *unsigned long* zdefiniowaną wcześniej jako *ID_ADDR* - w tym przypadku adresowi bezwzględnemu rejestru *APER_CLK_CTRL* kontrolującego zegary. [3]

```
#include "include/functions.h"

#define ID_ADDR (0xF800012C)

unsigned long get_id() {
       return RAW_SPACE(ID_ADDR);
}
```

Kod 5: get_id.c

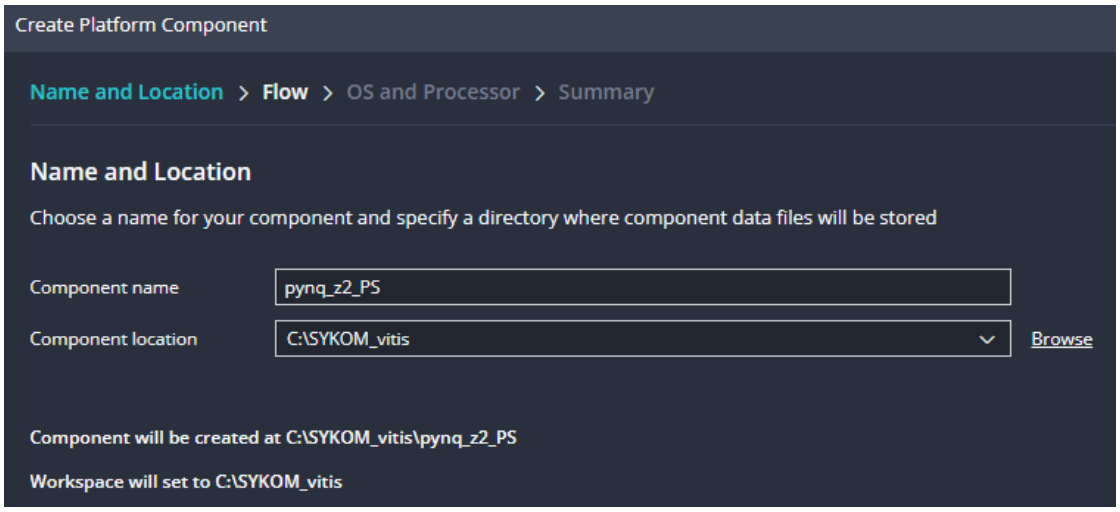
Funkcja *exit_simulation* inicjuje reset systemu, zapisując wartość *PSS_RST_VAL* do rejestru sterowania resetem *PSS_RST_CTRL*.

```
#include "include/functions.h"

#define PSS_RST_CTRL (*(volatile unsigned long *)((0xF80000000 + 0x200)))
#define PSS_RST_VAL (0x00000000)

void exit_simulation() {
    PSS_RST_CTRL = PSS_RST_VAL;
}
```

Kod 6: exit_simulation.c

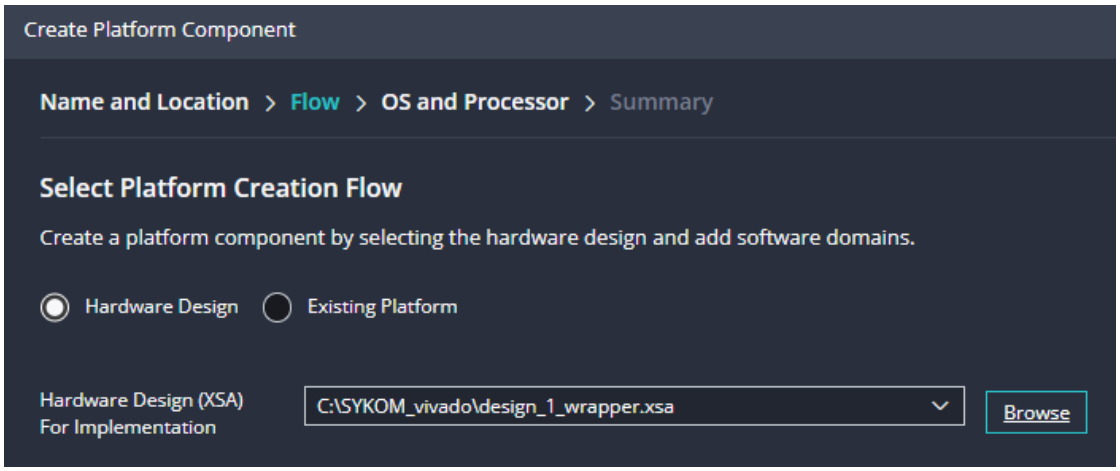
5 Kompilacja na platformę docelową z użyciem Vitis IDE

Pracę w narzędziu Vitis zaczęto od dodania platformy, na której uruchomiona będzie aplikacja. W tym celu wybierano opcję *File* > *New Component* > *Platform*. Następnie wybrano lokalizację przestrzeni roboczej oraz nazwę komponentu, co przedstawia rysunek 7.



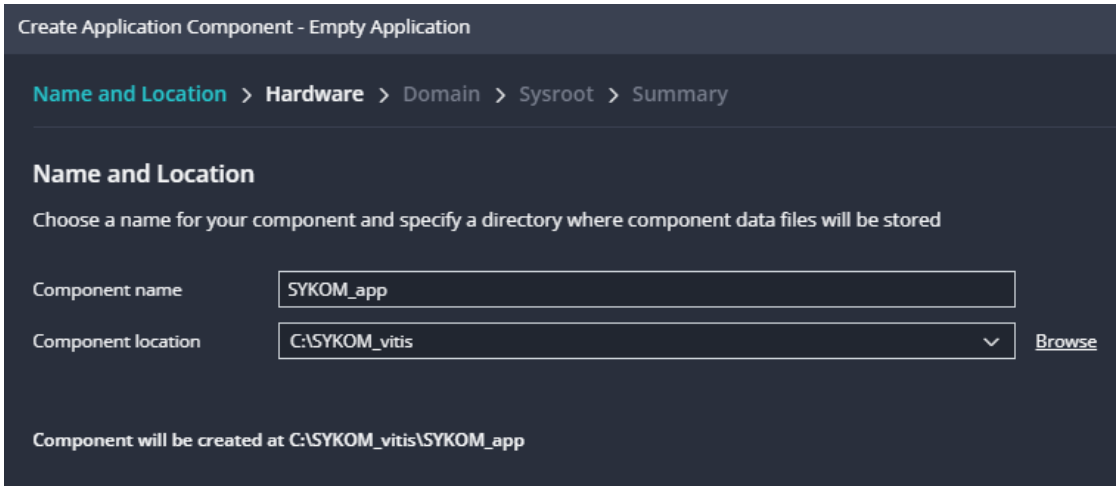
Rysunek 7: Tworzenie nowego projektu platformy.

W kolejnym kroku wybrano opis sprzętu w formacie .xsa (rysunek 8) wygenerowany z Vivado w sekcji 3. Na podstawie tego opisu wygenerowane zostało *device tree* oraz *linker file*.



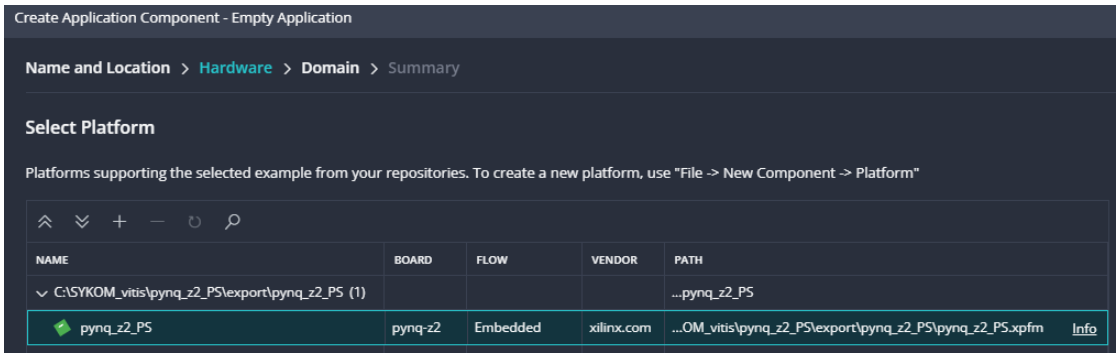
Rysunek 8: Dodanie opisu sprzętu w formacie .xsa.

Kolejne opcje pozostawiono bez zmian. Po otrzymaniu od środowiska powiadomieniu o ukończeniu tworzenia platformy, stworzono nową aplikację poprzez *File* > *New Component* > *Application*. Pierwszy krok wygląda tak samo jak w przypadku tworzenia platfomy (rysunek 9).



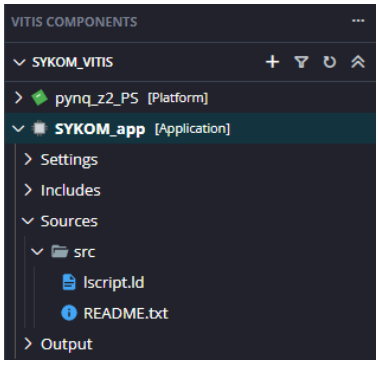
Rysunek 9: Tworzenie nowej aplikacji.

Przy wyborze platformy docelowej wybrano właśnie utworzoną na podstawie pliku .xsa platformę (rysunek 10), a następnie zostawiono domyślne opcje.



Rysunek 10: Wybór utworzonej wcześniej platformy.

Po stworzeniu projektu platformy i aplikacji okno komponentów wygląda jak na rysunku 11. Dwie najważniejsze ścieżki na które zwrócono uwagę to *Application/Sources/src*, skąd Vitis pobiera pliki do skompilowania na docelową platformę, oraz *Platform/Sources/hw/include*, gdzie można obejrzeć device tree systemu (plik *system-top.dts*).



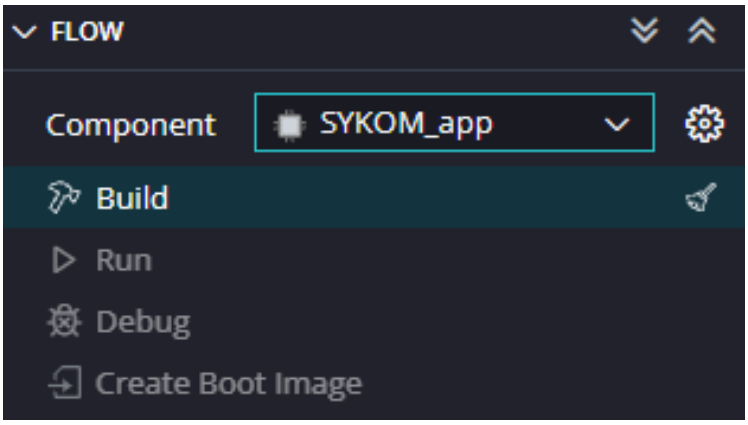
Rysunek 11: Drzewo komponentów projektu.

W celu skompilowania programu opisanego w sekcji 4 przekopiowano pliki do wspomnianego folderu src aplikacji, przy czym plik *functions.h* umieszczono w dodatkowym folderze *include*, który następnie wskazano w pliku *Application/Settings/UserConfig.cmake* w linii 29 poprzez wpis Kod 7.

```
1 set(USER_INCLUDE_DIRECTORIES
2     ${CMAKE_SOURCE_DIR}/include
3 )
```

Kod 7: UserConfig.cmake

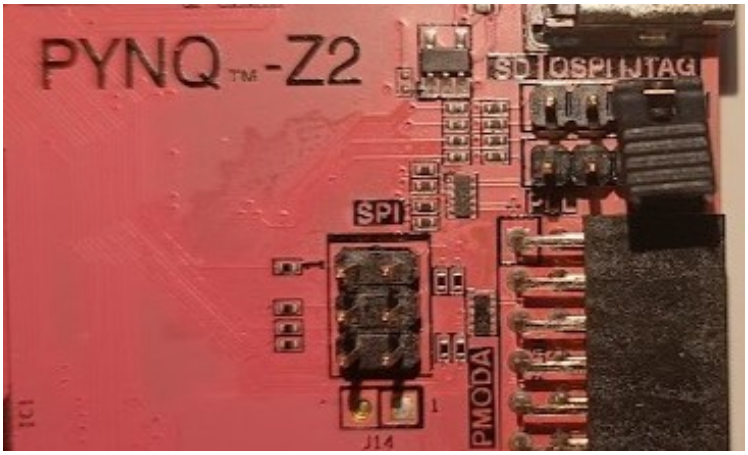
Po wykonaniu wymienionych czynności zbudowano projekt, za pomocą przycisku *Build* w oknie *Flow* które przedstawiono na rysunku 12. W wyniku tej operacji została wykonana kompilacja kodu źródłowego za pomocą odpowiedniego *toolchain'a*, a następnie linkowanie do pliku wykonywalnego .elf.



Rysunek 12: Budowanie aplikacji SYKOM_app.

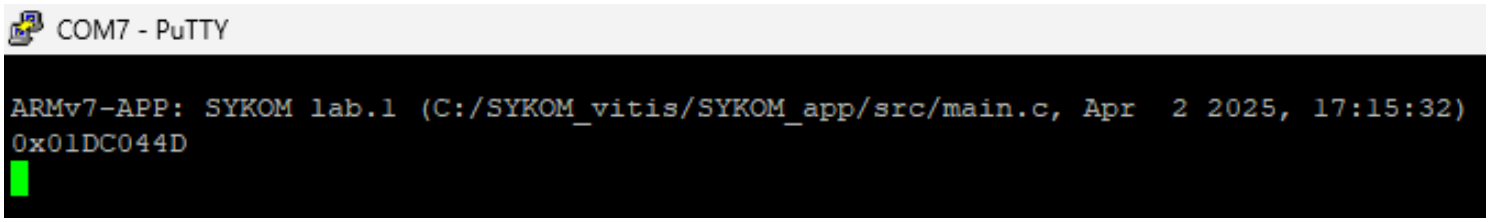
6 Uruchomienie na platformie PYNQ-Z2

W celu uruchomienia skompilowanej aplikacji na docelowej platformie podłączono ją za pomocą kabla *USB-A - micro-USB-B*, ustawiono zworkę boot-select w pozycji *JTAG* oraz ją uruchomiono.



Rysunek 13: Ustawienie zworki w pozycji JTAG.

Następnie zidentyfikowano numer portu szeregowego COM (system Windows) lub plik */dev/ttyUSBx* (gdzie x to numer portu; system Linux) przez który odbywać się będzie komunikacja. W systemie Windows można to zrobić za pomocą *Menadżera Urządzeń (devmgmt.msc)*, a w systemie Linux za pomocą wywołania *dmesg -wH* (opcja *w* powoduje oczekiwanie na nowe komunikaty, a opcja *H* wyświetla komunikaty sformatowane do odczytu przez człowieka).
Do obserwowania standardowego wyjścia urządzenia wykorzystano emulator terminala szeregowego Putty. Można użyć też wbudowanej w Vitis konsoli w zakładce *Vitis > Serial Monitor*. Jeżeli nie ma tej funkcji, to trzeba ją doinstalować poprzez *Vitis > New Feature Preview* i w przedstawionym oknie włączyć funkcję *Serial Monitor*. Gdy wszystko zostało przygotowane kliknięto przycisk *Run*, a środowisko prześle zbudowany już plik .elf (wraz z programem ładującym i opisem sprzętu) na docelową platformę. Gdyby aplikacja wymagała konfiguracji PL, Vitis w tym kroku załadowałby również bitstream. Po wykonaniu poleceń, na konsoli powinna pojawić się wiadomość przedstawiona na rysunku poniżej.



Rysunek 14: Rezultat działania aplikacji w terminalu Putty.

Wartość w postaci heksadecymalnej wyświetlona w drugiej linijce wygląda następująco w postaci binarnej: 0000 0001 1101 1100 0000 0100 0100 1101. Korzystając z dokumentacji technicznej Zynq-7000 SoC [3] możemy zauważyć, że pierwsze 7 bitów rzeczywiście ma wartość zero. Interesujące są również bity 21 i 20 (licząc od prawej), które odpowiadają za UART AMBA Clock control odpowiednio dla UART 1 i UART 0.

7 Wnioski

Zapoznano się z obsługą procesora w architekturze ARM v7 zawartego w układzie SoC Zynq-7020. Przeanalizowano drzewo urządzeń wykorzystywanego układu oraz stworzono podstawowe funkcje do obsługi UART z wykorzystaniem rejestrów z poziomu języka C. Zdobyto istotne umiejętności z dziedziny tworzenia oprogramowania działającego na układach SoC bez systemu operacyjnego oraz analizy dokumentacji układu.

Literatura

[1] AMD. Pynq-z2 reference manual v1.0, 2018. URL: https://dpoauwgwqsy2x.cloudfront.net/Download/pynqz2_user_manual_v1_0.pdf.

[2] rlangoy. Zedboard baremetal examples, 2013. URL: https://github.com/rlangoy/ZedBoard-BareMetal-Examples/tree/master.

[3] Xilinx. Zynq-7000 soc technical reference manual, 2018. URL: https://download.kamami.pl/p574762-ug585-Zynq-7000-TRM.pdf.