FPGA a open source

Prosty mikrokontroler DO6502

Materiały dodatkowe
https://bit.ly/2SBK3VW

W cyklu artykułów z EP 9/20 i EP 2/21 poznaliśmy otwarte narzędzia przeznaczone do syntezy FPGA oraz płytkę ewaluacyjną z układem ICE40 zrealizowaną jako projekt open hardware. Dysponując kompletem potrzebnych narzędzi pokażemy, w jaki sposób zbudować syntezowany mikrokontroler w oparciu na rdzeniu 65C02, znanym z pierwszych komputerów domowych lat 80. Prezentowany projekt ma charakter przede wszystkim edukacyjny. W praktycznych zastosowaniach lepszym rozwiązaniem będzie użycie rdzenia RISC-V. Jednak istotnym aspektem wyboru rdzenia jest jego prostota oraz sentyment do starych komputerów z lat 80. Projekt możemy uruchomić na płytce ewaluacyjnej „IceCore ICE40 HX" opisanej w poprzednim artykule.

Mikrokontroler typu DO6502 charakteryzuje się następującymi parametrami:

- rdzeń 65C02 taktowany zegarem 6,25 MHz,
- pamięć RAM 4 kB,
- pamięć ROM 8 kB,
- interfejs USART z 16-bajtowym buforem FIFO o prędkości transmisji 57600 bps,
- wyjściowy port GPIO sterujący 4 diodami LED.

Zestaw peryferiów prezentowanego mikrokontrolera jest skromny i ma na celu sprawdzenie, czy otwarte narzędzia do syntezy FPGA mogą być stosowane do projektowania rozbudowanych projektów, takich jak implementacja własnego mikrokontrolera. W miarę potrzeb projekt możemy rozbudować o dodatkowe układy peryferyjne, ponieważ obecnie zajmuje on jedynie 15% bloków układu ICE40, a zatem mamy wystarczająco dużo zasobów, aby znacząco rozbudować mikrokontroler.

Budowa mikrokontrolera

Schemat blokowy mikrokontrolera DO6502 został pokazany na **rysunku 1**. Sercem układu jest miękki rdzeń 65C02 pierwotnie opracowany przez Arleta Ottensa, a następnie zmodyfikowany przez Davida Banksa, i dostępny jest na platformie GitHub. Rdzeń jest w pełni zgodny z listą instrukcji mikroprocesora 65C02 i bez problemu przechodzi test Dorman 65C02 potwierdzający zgodność programową z listą instrukcji oryginalnego 65C02.

Interfejs zewnętrzny został dostosowany do specyfiki implementacji FPGA i ma dwie oddzielne magistrale: danych wejściowych (DI) oraz wyjściowych (DO). Mikroprocesor może współpracować jedynie z pamięciami synchronicznymi, gdzie dane na magistrali DI są oczekiwane w kolejnym cyklu po wystawieniu sygnału na magistrali adresowej ADDR. Pozostałe sygnały takie jak RDY, NMI, IRQ są zgodne z oryginalnym mikroprocesorem 65C02. Przykładowe sygnały na magistrali podczas wykonania programu z pamięci synchronicznej zostały pokazane na **rysunku 2**.

Tabela 1. Mapa pamięci układu

Adres	Rozmiar	Linia	Opis
0x0000	2048	RAMCS	Pamięć RAM
0x0400	256	PERIPHCS	Przestrzeń układów peryferyjnych
0xE000	4096	ROMCS	Wybór pamięci ROM

Jeżeli mikroprocesor chce odczytać dane z pamięci, linia WE ustawiana jest w stan 0, a następnie w kolejnym cyklu mikroprocesor odczytuje dane z magistrali DI. W przypadku gdy procesor chce zapisać dane, wystawia sygnał WE, jednocześnie wystawiając dane do zapisu na magistrali WE.

16-bitowa przestrzeń adresowa podzielona jest na 3 obszary za pomocą głównego dekodera adresowego oznaczonego na schemacie jako ADDR dekoder. Dekoder na wyjściu ma sygnały wyboru pamięci RAMCS, sygnały wyboru pamięci ROMCS oraz sygnału wyboru przestrzeni układów peryferyjnych PERIPHCS. Mapa pamięci znajduje się w **tabeli 1**.

Listing 1. Dekoder adresowy zrealizowany jako standardowy układ kombinacyjny

```
module main_address_decoder(
    input [15:0] addr_bus,
    output rom_cs,
    output ram_cs,
    output periph_cs
);

    function inbetween(input [15:0] low, input [15:0] value, input [15:0] high);
        inbetween = (value >= low && value <= high);
    endfunction

    function inany(input [15:0] value);
        inany = (^value===1'bX);
    endfunction

    reg [2:0] decoder_al;
    always @(addr_bus) begin
        (* full_case,parallel_case *)
        case(1'b1)
            inany(addr_bus): decoder_al <= 3'b100;
            inbetween(16'h0000,addr_bus,16'h0FFF): decoder_al <= 3'b001;        //RAM
            inbetween(16'h0400,addr_bus,16'h04FF): decoder_al <= 3'b010;        //Periph
            inbetween(16'hE000,addr_bus,16'hFFFF): decoder_al <= 3'b100;        //BootROM
        endcase
    end
    assign ram_cs =    decoder_al[0];
    assign periph_cs = decoder_al[1];
    assign rom_cs =    decoder_al[2];
endmodule
```

Dekoder adresowy

Dekoder został zrealizowany jako standardowy układ kombinacyjny, którego wejście stanowią sygnały adresowe, natomiast wyjściem są linie sygnałowe wyboru pamięci oraz układów peryferyjnych (**listing 1**). Wejście dekodera stanowi 16-bitowa magistrala input, która jest porównywana z zadeklarowanymi przedziałami adresowymi z wykorzystaniem funkcji `in_between()`. W przypadku spełnienia przedziału dla danego zakresu na odpowiadających liniach wyboru peryferiów wystawiany jest stan wysoki. W aktualnej implementacji mikroprocesora w następnym cyklu zegarowym po zdjęciu sygnału RESET młodszy bajt magistrali adresowej znajduje się w stanie nieustalonym, zatem konieczne stało się dodanie do dekodera adresowego rozpoznawania tego stanu. W przypadku wykrycia stanu nieustalonego na dowolnej linii adresowej wybierana jest pamięć ROM, co umożliwia odczyt pierwszego rozkazu z pamięci oraz prawidłowy start mikroprocesora.

Pamięć RAM

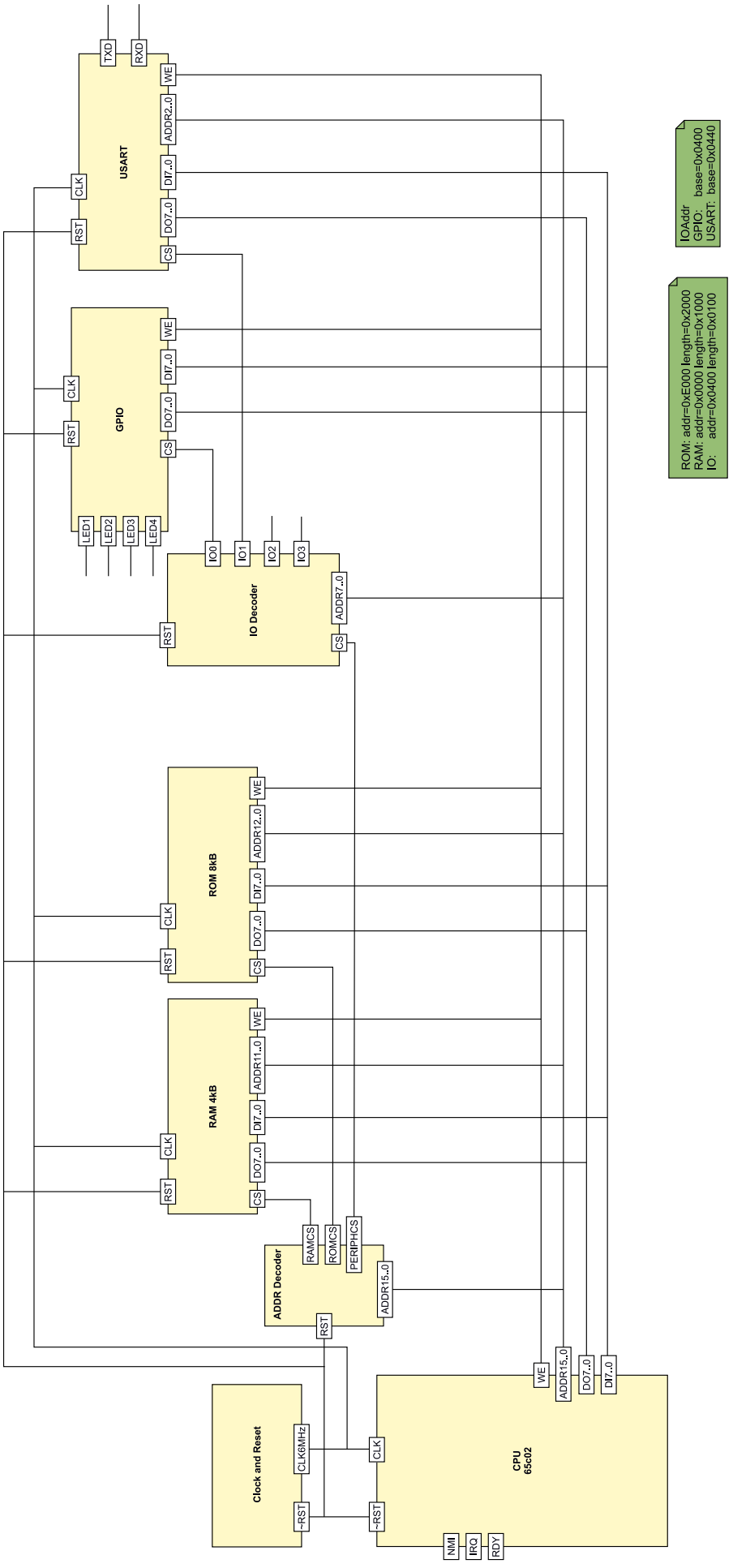
Blok pamięci RAM został zaimplementowany bezpośrednio w VERILOG-u bez użycia modułów specyficznych dla ICE40 celem sprawdzenia, jak otwarte narzędzie nextpnr poradzi sobie z rozpoznaniem bloków, które mogą być zaimplementowanie w pamięci BlockRAM (**listing 2**).

Blok synchronicznej pamięci RAM (*internal_ram_notristate*) zawiera parametr konfiguracyjny BIT, który pozwala określić rozmiar pamięci poprzez liczbę bitów magistrali adresowej. Sygnał CLK jest sygnałem zegara pamięci synchronicznej, sygnał WE określa, czy dane będą odczytane (stan 0), czy zapisywane (stan 1). Linie danych pamięci RAM zostały podzielone na magistralę wejściową (DI) oraz magistralę wyjściową (DO), która dodatkowo wyposażona jest w bufor pozwalający na opóźnienie danych o jeden cykl zegarowy, zgodnie z wymaganiami stawianymi przez rdzeń. W bloku warunkowym *__ICARUS* znajduje się wstępne zerowanie pamięci RAM wymagane przez symulator ICARUS VERILOG i jest ono syntezowane tylko w przypadku symulacji.

Pamięć ROM

W analogiczny sposób zrealizowano blok pamięci ROM (**listing 3**), jedyną różnicą w stosunku do pamięci RAM jest brak linii WE. Zawartość pamięci ROM jest odczytywana na etapie syntezy za pomocą polecenia `readmemh` z pliku *capp.hex*. Plik ten generowany jest za pomocą narzędzia hexdump, które zamienia plik binarny wygenerowany przez kompilator na plik zawierający heksadecymalne wartości w postaci tekstu.

Moduły pamięci *internal_rom_no_tristate* *internal_ram_no_tristate* nie mogą być bezpośrednio dołączone do procesora, ponieważ



Rysunek 1. Schemat blokowy mikrokontrolera DO6502

Rysunek 2. Przykładowe sygnały na magistrali podczas wykonania programu z pamięci synchronicznej

Listing 2. Kod opisujący blok pamięci RAM

```
/* RAM memory */
module internal_ram_no_tristate
#(parameter BITS=10)
(
    input clk,                  // Clock input
    input we,                   // WR/RD signal
    input [BITS-1:0] addr,      // Address input
    input [7:0] din,            // Data input
    output reg [7:0] dout       //Data output
);
    reg [7:0] ram_memory[0:2**BITS-1];
`ifdef __ICARUS__
    integer i;
    initial begin
        for(i=0;i<(2**BITS-1);++i)
            ram_memory[i] = 0;
    end
`endif
    always @(posedge clk) begin
        if(we) ram_memory[addr] <= din;
        else dout <= ram_memory[addr];
    end
endmodule

module internal_ram
#(parameter BITS=10)
(
    input clk,                  // Clock input
    input rst,                  // Reset signal
    input we,                   // WR/RD signal
    input oe,                   // Chip select signal
    input [BITS-1:0] addr,      // Address input
    input [7:0] din,            // Data input
    output [7:0] dout           //Data output
);
    wire [7:0] idata;
    internal_ram_no_tristate #(.BITS(BITS)) ram(
        .clk(clk),
        .we(we&oe),
        .addr(addr),
        .din(din),
        .dout(idata)
    );

    reg oe_del;
    always @(posedge clk or negedge rst)
        if(~rst) oe_del <= 1'b0;
        else oe_del <= oe;

    tristate_buf tbuf(
        .oe(oe_del),
        .in(idata),
        .out(dout)
    );
endmodule
```

Listing 3. Kod realizujący blok pamięci ROM

```
module internal_rom_no_tristate
#(parameter BITS=9)
(
    input clk,
    input [BITS-1:0] addr,
    output reg [7:0] data
);

    localparam MEM_INIT_FILE="capp.hex";
    reg [7:0] rom_memory[0:(2**BITS-1)];
    initial begin
        if(MEM_INIT_FILE!="")
            $readmemh(MEM_INIT_FILE, rom_memory);
    end
    always @(posedge clk)
        data <= rom_memory[addr];
endmodule

/* ROM memory */
module internal_rom
#(parameter BITS=10)
(
    input clk,                  // Clock input signal
    input rst,
    input oe,                   // Enable signal
    input [BITS-1:0] addr,      //Adress input
    output [7:0] data           // Data output
);
    wire [7:0] ntsdata;
    internal_rom_no_tristate #(.BITS(BITS)) rom(
        .clk(clk),
        .addr(addr),
        .data(ntsdata)
    );
    reg oe_del;
    always @(posedge clk or negedge rst)
        if(~rst) oe_del <= 1'b0;
        else oe_del <= oe;


    tristate_buf tbuf(
        .oe(oe_del),
        .in(ntsdata),
        .out(data)
    );
endmodule
```

Listing 4. Dodatkowy dekoder adresowy IO decoder

```
module device_address_decoder(
    input [7:0] addr_in,
    input periph_cs,
    output periph1,
    output periph2,
    output periph3,
    output periph4
);
    reg [3:0] decoder;
    wire [2:0] addr = { periph_cs, addr_in[7:6] };
    always @(addr) begin
        (* full_case,parallel_case *)
        casex(addr)
            3'b100: decoder = 4'b0001;
            3'b101: decoder = 4'b0010;
            3'b110: decoder = 4'b0100;
            3'b111: decoder = 4'b1000;
            default: decoder = 4'b0000;
        endcase
    end
    assign periph1 = decoder[0];
    assign periph2 = decoder[1];
    assign periph3 = decoder[2];
    assign periph4 = decoder[3];
endmodule
```

nie mają wyjścia trójstanowego aktywowanego sygnałem CS. Moduły *internal_ram* oraz *internal_rom* korzystają ze wspomnianych wcześniej modułów, wyposażając je w dodatkowy bufor trójstanowy aktywowany linią wyboru CS. Doświadczenia pokazały, że nextpnr bez większych problemów rozpoznaje i dokonuje syntezy pamięci ROM i RAM, umieszczając ją w BlockRAM bez potrzeby bezpośredniego odnoszenia się do komponentów specyficznych dla układu ICE40.

Dodatkowy dekoder

Główny dekoder adresowy ma tylko jedną linię CS, dla układów peryferyjnych koniecznie jest użycie dodatkowego dekodera oznaczonego na schemacie jako IO decoder (**listing 4**) Jego zadaniem jest podział 256-bajtowej przestrzeni IO na 4 obszary po 64 bajty każdy. Blok zawiera 8-bitowy sygnał wejściowy, linię wejściową CS (pochodzącą z głównego dekodera) oraz 4 linie wyjściowe IO0...IO3 wyznaczające wybór układu w obszarze wejścia-wyjścia. Dekoder został zrealizowany w analogiczny sposób do głównego dekodera, z tym że nie ma tutaj konieczności rozpoznawania stanu nieustalonego.

Port GPIO

Do linii IO0 dekodera adresowego urządzeń dołączony został prosty wyjściowy Port GPIO mający 4 wyjścia (**listing 5**). Port ten został zrealizowany w najprostszy sposób i składa się z 4 przerzutników D z linią aktywacji układu CS. Linie D przerzutników dołączone są do 4 najmłodszych linii magistrali danych DI. Linie wyjściowe portu GPIO zmapowane są na 4 diody LED na płytce ewaluacyjnej ICE-CORE. Zatem zmieniając stan bitów w rejestrze GPIO, możemy sterować diodami LED.

Port szeregowy USART

Drugim urządzeniem naszego eksperymentalnego mikrokontrolera jest dwukierunkowy port szeregowy USART z 16-bajtowym buforem FIFO,

Listing 5. Kod prostego wyjściowego portu GPIO

```
module output_gpio(
    input clk,                  // Clock
    input rst,                  // Reset
    input [7:0] di,             // Data input
    input oe,                   // Output enable
    input we,                   // Write enable
    output reg [3:0] port       //Output port
);

    always @(posedge clk or negedge rst) begin
        if( ~rst ) begin
            port <= 4'd0;
        end
        else begin
            if( oe && we )
                port <= di[3:0];
        end
    end
endmodule
```

dostosowany do magistrali mikroprocesora 65C02 (**listing 6**). Interfejs modułu *tiny_usart()* jest zgodny z magistralą 65C02 oraz ma dodatkowo linię adresową wyboru układu CS, która jest dołączona do przestrzeni adresowej IO2 drugiego dekodera adresowego. Wyjście modułu stanowią dwie linie portu szeregowego TXD oraz RXD. Od strony programowej do dyspozycji mamy 3 rejestry. Rejestr danych DR, do którego należy wpisać daną, jaką chcemy wysłać lub odczytać daną odebraną. Rejestr statusu SR, który zawiera dwa bity: bit 0 (TXE), który mówi nam, że bufor nadawczy nadajnika jest pusty; bit 1 (RXNE) mówiący o tym, że w buforze odbiornika został odebrany znak i możemy go odczytać z rejestru danych. Rejestr kontrolny CR zawiera tylko jeden bit 0 (STX), ustawienie tego bitu powoduje rozpoczęcie transmisji.

Do realizacji zintegrowanego portu szeregowego zastosowano moduł *usart_tx*, który odpowiedzialny jest za realizację nadajnika oraz *usart_rx* zawierający implementację odbiornika. Moduł *usart_tx_fifo* zawiera magistralę danych oznaczoną jako data, oraz sygnał kontrolny valid, który należy ustawić, aby rozpocząć transmisję. W odpowiedzi na rozpoczęcie transmisji sygnał ready jest ustawiany po zakończeniu nadawania.

Analogiczny interfejs ma moduł odbiornika *usart_rx_fifo* oraz dodatkowo sygnał overflow oznaczający przepełnienie wewnętrznej kolejki FIFO. Sercem portu szeregowego jest maszyna stanów odpowiedzialna za generowanie sygnałów dla modułu nadajnika i odbiornika na podstawie wartości wpisywanych do rejestrów przez procesor 65C02.

Prędkość portu szeregowego została ustalona na stałą wartość 57600 bitów na sekundę i może być zmieniana na etapie syntezy poprzez zmianę współczynników dzielnika fraktalnego, które są parametryzowane za pomocą parametrów SPEED_KDIV oraz SPEED_NDIV. Port szeregowy, w razie potrzeby, może być łatwo rozbudowany o linię IRQ służącą do zgłaszania przerwań, jednak z uwagi na eksperymentalny charakter projektu została ona pominięta.

Przykładowy program

Celem demonstracji możliwości mikrokontrolera przygotowano prosty program, którego zadaniem jest sprawdzenie poprawności działania rdzenia oraz jego skromnych peryferiów. Program został napisany w języku C i jest kompilowany za pomocą otwartoźródłowego kompilatora dla cc65 dla rodziny mikroprocesorów 6502.

Działanie programu polega na użyciu portu szeregowego jako konsoli szeregowej służącej do sterownia czterema diodami LED dołączonymi do portu GPIO układu. Pętla główna programu wygląda jak na **listingu 7**. Na początku program wysyła na konsolę szeregową komunikat zachęcający do wpisania na klawiaturze znaku z przedziału 1...4, a następnie oczekuje na odebranie pojedynczego znaku przez układ UART. Po odebraniu znaku sprawdzany jest dozwolony zakres znaków. Jeśli nie mieści się on w określonym przedziale, wówczas odsyłany jest komunikat błędu. W przypadku gdy znak mieści się w przedziale 1...4, stan odpowiadającej mu diody LED zmieniany jest na przeciwny. Znak 1 będzie odpowiedzialny za zmianę stanu diody dołączonej do linii 49 i dalej analogicznie: linia 52, linia 55 i linia 56.

Funkcja *led_control()* jest odpowiedzialna za włączenie lub wyłączenie poszczególnych diod LED dołączonych do portu GPIO i sprowadza

Listing 6. Kod dwukierunkowego portu szeregowego USART z 16-bajtowym buforem FIFO

```
module tiny_usart(
    input clk,
    input rst,
    output tx_pin,
    input rx_pin,
    input [2:0] addr,
    input cs,
    input we,
    input [7:0] di,
    output [7:0] do
);
    reg utx_valid;
    wire utx_ready;
    reg [7:0] tx_dr;
    reg [7:0] rx_dr_rdy;
    reg [7:0] idata;
    wire [7:0] urx_bus;
    wire urx_valid;
    reg urx_ready;
    wire urx_overflow;

    localparam DR_ADDR = 3'd0;
    localparam SR_ADDR = 3'd1;
    localparam CR_ADDR = 3'd2;

    always @(posedge clk or negedge rst)
        if(~rst) begin
            tx_dr <= 8'd0;
            utx_valid <= 1'b0;
            urx_ready <= 1'b0;
            rx_dr_rdy <= 1'b0;
        end
        else begin
            if(cs && we) begin
                if(addr==DR_ADDR) begin
                    tx_dr <= di;
                end
                else if(addr==CR_ADDR) begin
                    utx_valid <= di[0];
                end
            end
            else if(cs && !we) begin
                if(addr==SR_ADDR) begin
                    idata <= { 5'd0, urx_overflow,
                               rx_dr_rdy, utx_ready };
                end
                else if(addr==CR_ADDR) begin
                    idata <= { 7'd0, utx_valid };
                end
                else if(addr==DR_ADDR) begin
                    idata <= urx_bus;
                    rx_dr_rdy <= 1'b0;
                end
            end
            if(urx_valid) begin
                urx_ready <= 1'b1;
                rx_dr_rdy <= 1'b1;
            end

            if(utx_valid) utx_valid <= 1'b0;
            if(urx_ready) urx_ready <= 1'b0;
        end


    // Transmit component
    usart_tx_fifo #(.SPEED_NDIV(13)) utx(
        .clk(clk), .rst(rst),
        .valid(utx_valid), .ready(utx_ready),
        .data(tx_dr), .u_tx_pin(tx_pin)
    );

    // Receive component
    usart_rx_fifo #(.SPEED_NDIV(13)) urx( .clk(clk),
        .rst(rst),
        .valid(urx_valid), .ready(urx_ready),
        .overflow(urx_overflow), .data(urx_bus),
        .u_rx_pin(rx_pin)
    );

    reg cs_del;
    always @(posedge clk or negedge rst)
        if(~rst) cs_del <= 1'b0;
        else cs_del <= cs;

    tristate_buf tbuf(
        .oe(cs_del),
        .in(idata),
        .out(do)
    );

endmodule
```

się do wpisania stanu bitów do rejestru portu GPIO znajdującego się pod adresem bazowym 0x400 (**listing 8**).

Wysłanie łańcucha tekstowego na konsolę szeregową polega na cyklicznym wywołaniu funkcji *putchar()*, która przesyła kolejne znaki do bufora portu szeregowego (**listing 9**). Wysłanie pojedynczego znaku polega na sprawdzeniu w rejestrze statusu, czy bufor nadajnika jest wolny. W przypadku spełnienia tego warunku do rejestru DR wpisywany jest kod znaku, który chcemy przesłać, a w rejestrze kontrolnym CR ustawiany jest bit STX, co powoduje wpisanie kodu znaku do wewnętrznej kolejki FIFO portu szeregowego.

Za odbiór znaku z portu szeregowego odpowiada funkcja *getchar()* (**listing 10**). W aktywnej pętli sprawdza, czy bit RXNE w rejestrze SR

został ustawiony na 1, co oznacza, że w rejestrze DR znajduje się nowy znak do odczytania.

Aby program działał prawidłowo należy umieścić go w odpowiednim miejscu pamięci, do czego będziemy potrzebować skryptu linkera. Będziemy również potrzebować plik startowy *crt0.S*, który jest odpowiedzialny za ustawienie pamięci zgodnie z wymaganiami języka ANSI C. Mapę pamięci zawartą w pliku linkera pokazuje fragment skryptu:

```
MEMORY {
  ZP:  start = $0, size = $100, type = rw, define =
yes;
  RAM: start = $200, size = $0600, define = yes;
  ROM: start = $E000, size = $2000, file = %O;
}
```

Obszar ZP jest obszarem strony zerowej stanowiącym fragment pamięci RAM i rozpoczyna się od adresu 0. Dalszy obszar pamięci RAM kolejnych stron zawarty jest w sekcji RAM i rozpoczyna się od adresu 0x200. Obszar pamięci ROM rozciąga się od adresu 0xE000 aż do końca obszaru adresowego 0xFFFF, co oznacza, że wektory przerwań znajdować się będą bezpośrednio w pamięci ROM.

Struktura projektu oraz uruchomienie

Jedną z zalet otwartych narzędzi syntezy układów programowalnych jest możliwość wywołania ich z wiersza polecenia w dowolny sposób, bez konieczności używania zintegrowanych środowisk. Do zautomatyzowania procesu syntezy z pomocą przychodzą nam klasyczne narzędzia do automatyzacji budowania oprogramowania. W naszym projekcie użyto narzędzia gnu make, które przetwarza plik reguł Makefile i na tej podstawie stwierdza, które pliki źródłowe wymagają syntezy/kompilacji. Aby zbudować projekt, do dyspozycji mamy kilka głównych reguł zdefiniowanych w pliku Makefile realizujących następujące czynności:

- `make all` – kompletne zbudowanie projektu, które polega na kompilacji programu za pomocą kompilatora CC65 oraz syntezie projektu mikrokontrolera. Do syntezy wykorzystywane jest narzędzie nextpnr-ice40, które na podstawie plików źródłowych Verilog i pliku bin wygenerowanego przez kompilator tworzy plik *retromcu.bin*, który stanowi plik wynikowy przeznaczony do konfiguracji układu ICE40;
- `make clean` – wyczyszczenie wszystkich plików pośrednich oraz wynikowych;
- `make sim` – dokonuje syntezy oraz symulacji projektu z wykorzystaniem pakietu icarus-verilog. W wyniku tego powstają pliki wynikowe LXT, które następnie możemy załadować do narzędzia służącego do wyświetlania wykresów GTK Wave;

Jeśli chodzi o strukturę projektu, to została ona podzielona na katalogi odpowiadające poszczególnym modułom projektu:

- capp – pliki źródłowe aplikacji w języku C i ASM, które następnie mogą być kompilowane za pomocą kompilatora CC65;
- rtl – pliki źródłowe Verilog projektu mikrokontrolera;
- sim – pliki testbench służące do weryfikacji poprawności działania plików źródłowych. Główne jednostki testowe powinny mieć nazwę kończącą się sufiksem *_top.v*. Dla wszystkich plików top generowane są pliki stanowiące wynik symulacji w postaci przebiegów czasowych dla programu GTK Wave.

Aby uruchomić projekt, należy dysponować skonfigurowanym środowiskiem, które zostało opisane w pierwszym odcinku cyklu, a ponadto należy doinstalować kompilator dla mikroprocesora cc65. W przypadku popularnych dystrybucji Linuksa można to zrobić za pomocą polecenia:

```
sudo apt install cc65
```

Jeśli dysponujemy systemem OS-X, możemy skorzystać z HomeBrew:

```
brew install cc65
```

Jeśli mamy gotowy zestaw narzędzi, w kolejnym kroku należy dokonać kompilacji oraz syntezy projektu, co możemy zrealizować za pomocą polecenia: `make`, czego rezultatem będzie powstanie pliku *retromcu.bin*.

Zanim uruchomimy mikrokontroler, do linii P32 stanowiącej wyjście TXD zestawu iCE-Core należy dołączyć wejście RXD przejściówki RS232 ↔ USB, natomiast do linii P33 (RXD) wyjście RXD przejściówki RS232 ↔ USB. Należy również uruchomić program terminalowy na wirtualnym porcie USB z następującymi parametrami transmisji: 57600,N,8,1. Konfiguracji zestawu dokonujemy poprzez przesłanie do wirtualnego portu szeregowego */dev/ttyACM0* pliku konfiguracyjnego za pomocą polecenia:

```
cat retromcu.bin > /dev/ttyACM0
```

Po przesłaniu pliku, w zestawie ICE-CORE powinna zaświecić się dioda CONF. W tym momencie przy wysyłaniu w programie terminalowym kodów klawiszy 1...4 powinny się naprzemiennie włączać diody LD1...LD4 przypisane poszczególnym klawiszom, a na konsolę szeregową będzie wysyłana informacja o włączeniu lub wyłączeniu wybranej diody.

Listing 7. Pętla główna programu testowego

```
int main() {
    unsigned char ch,lds;
    puts("\r\nHello from 65c02 LED test\r\n");
    puts("Press 1-4 for led toggle\r\n");
    led_control(0xff);
    for(lds=0;;) {
        ch = getchar();
        if(ch<'1' || ch > '4') {
            puts("Error: Unknown key only 1-4 is allowed");
        } else {
            ch -= '0';
            lds ^= 1 << (ch-1);
            led_control(~lds);
            puts("Led: "); tohex(ch); puts(" toggled.\r\n");
        }
    }
    return 0;
}
```

Listing 8. Funkcja led_control()

```
#define LED_REG *(volatile unsigned char*)(0x0400)
static void led_control(unsigned char led) {
    LED_REG = led;
}
```

Listing 9. Funkcja putchar()

```
#define USART_DR_REG *(volatile unsigned char*)(0x0440)
#define USART_SR_REG *(volatile unsigned char*)(0x0441)
#define USART_CR_REG *(volatile unsigned char*)(0x0442)
#define USART_SR_TXE 0x01
#define USART_SR_RXNE 0x02
#define USART_CR_STX 0x01

static void putch(char ch) {
    while( !(USART_SR_REG & USART_SR_TXE) ) {}
    USART_DR_REG = ch;
    USART_CR_REG |= USART_CR_STX;
}
```

Listing 10. Funkcja getchar()

```
static unsigned char getchar(void) {
    while( !(USART_SR_REG & USART_SR_RXNE) ) {};
    return USART_DR_REG;
}
```

Zakończenie

Zaprezentowany projekt zawiera jedynie podstawowe układy peryferyjne takie jak: GPIO, UART. W praktycznych zastosowaniach należałoby go wyposażyć w dodatkowe peryferia np. układy czasowo-licznikowe oraz dodatkowy kontroler przerwań. Celem projektu nie było opracowanie praktycznego mikrokontrolera, a sprawdzenie, czy otwarte narzędzia programistyczne pozwalają na syntezę rozbudowanych projektów bazujących na syntezowanych mikroprocesorach. W praktycznych zastosowaniach lepszym rozwiązaniem będzie skorzystanie z bardziej wydajnego 32-bitowego rdzenia RISC-V zamiast z trzydziestoletniego CPU. Praktyczne doświadczenia pokazały, że nextptr-ice40 bez problemu poradził sobie z tego rodzaju zaawansowanym projektem i udało się uzyskać w pełni działający mikroprocesor z częstotliwością pracy 6,25 MHz.

Lucjan Bryndza, EP
lucjan.bryndza@ep.com.pl

Materiały dodatkowe:
- EP 2/21: FPGA a Open Source. Płytka prototypowa IceCore z układem Lattice ICE40, https://bit.ly/3oP5oXM
- EP 9/20: Przegląd otwartych narzędzi dla układów FPGA, https://bit.ly/35XvZJT