Implementacja systemu mikroprocesorowego z procesorem zgodnym z 6502 na platformie FPGA

Złożone systemy cyfrowe 2023

Norbert Morawski

12 czerwca 2023

Spis treści

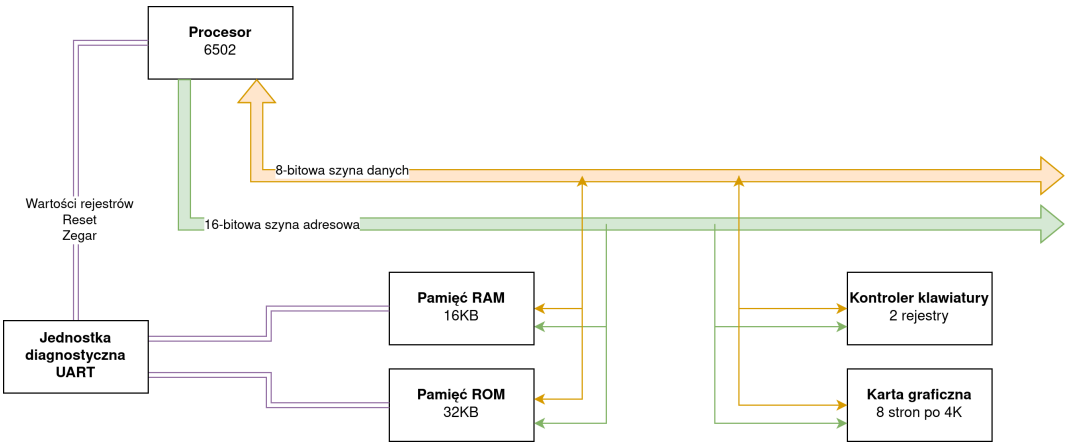
Rozdział 1

Wstęp

Celem projektu jest wykonanie, w miarę możliwości, jak najdokładniejszej kopii procesora 6502 firmy MOS Technology w układzie FPGA. Dodatkowo, aby zaprezentować jego działanie, do zestawu uruchomieniowego zostanie podłączona klawiatura oraz monitor.

Rozdział 2

Założenia projektowe



Rysunek 2.1: Schemat blokowy komputera

System mikroprocesorowy będzie się składał z niżej wymienionych części.

2.1 Procesor

Procesor zostanie zaimplementowany w środowisku Quartus Prime Lite w języku SystemVerilog i uruchomiony na platformie DE10 Lite z układem MAX 10. Parametry procesora:

- Pełna kompatybilność z zestawem instrukcji MOS Technology 6502,
- Implementacja wybranych rozszerzeń z układu 65c02 firmy Western Design Center,
- W miarę możliwości zgodność cykli maszynowych,
- Dekodowanie instrukcji z użyciem tzw. *random logic*, bez wykorzystania mikrokodu.

2.2 Pamięć

Pamięć RAM i ROM będą stanowiły odpowiednie moduły w środowisku Quartus oparte o pamięć M9K znajdującą się bezpośrednio w układzie MAX 10.

Pamięć ROM Zostanie tam umieszczony program do wykonania przez procesor. Pomimo, że procesor będzie mógł tylko odczytywać pamięć ROM, układ diagnostyczny powinien mieć możliwość zapisu do pamięci stałej.

Pamięć RAM W pamięci ulotnej będzie znajdować się strona zerowa, stos i inne zmienne dane potrzebne do działania komputera.

Zmiana rozmiarów pamięci Być może okaże się konieczne poszerzenie pamięci RAM kosztem pamięci ROM.

2.3 Jednostka diagnostyczna

Układ *debug* powinien mieć możliwość:

- odczyt z pamięci RAM i ROM,
- zapis do pamięci RAM i ROM,
- odczyt rejestrów procesora
- zarządzanie wykonaniem programu (sygnałem zegarowym)

Komunikacja z użytkownikiem będzie przebiegać poprzez interfejs UART. Pozwoli to na szybkie sprawdzanie różnych funkcji procesora bez konieczności kompilacji programu FPGA.

2.4 Karta graficzna

Inspirowana paletą kolorów kart CGA, karta graficzna będzie obsługiwać, w trybie tekstowym:

- 16 kolorów tła (3 bity RGB + 1 bit jasności),
- 8 kolorów znaków + bit migania.

W trybie graficznym:

- taka sama paleta kolorów jak w trybie tekstowym, z podziałem na siatkę znaków,
- możliwość adresowania linii 8 pikseli i ustawienia pojedynczych bitów.

Rozdzielczość karty została przewidziana tak, aby tryb tekstowy zajmował 4KB, a tryb graficzny 32KB pamięci (bez uwzględnienia kolorów i atrybutów). Wynosi ona 400x300 pikseli (1/4 standardowej rozdzielczości 800x600). Daje to 50x37 znaków.

Atrybuty będą przechowywane w osobnej pamięci o rozmiarze 4KB (siatka znaków).

Generator znaków będzie oparty o pamięć 2KB.

2.5 Kontroler klawiatury PS/2

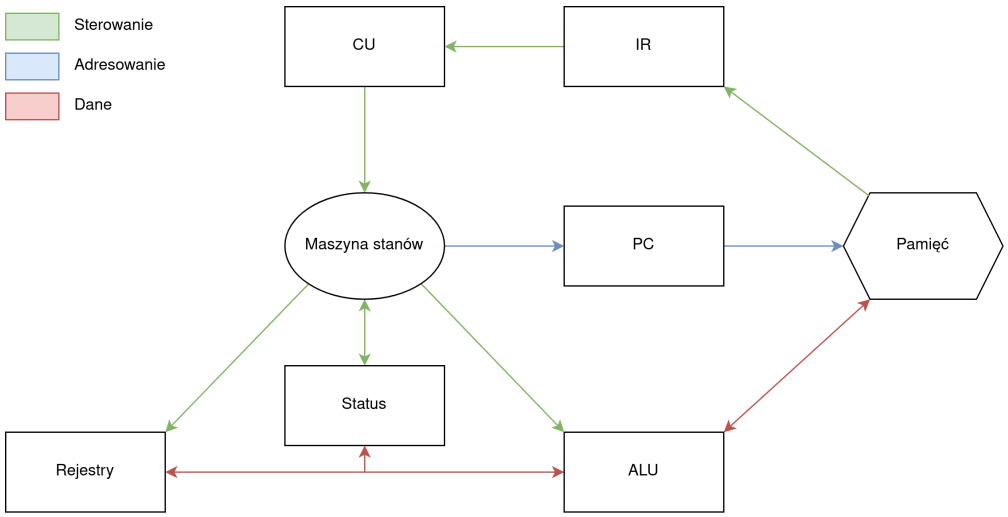
Będzie on umożliwiał odczyt znaków klawiatury wraz ze sprzętowym dekodowaniem specjalnych bajtów danych takich jak $E0_{(16)}$ czy $F0_{(16)}$ oraz będzie obsługiwał wywoływanie przerwań mikroprocesora.

Rozdział 3

Implementacja

3.1 Procesor 6502

Procesor przechodził wiele zmian. Od prostej maszyny stanów, poprzez wiele zmiennych sterujących (maszyny stanów prawie nie było), aż spowrotem do maszyny stanów, gdzie w końcu udało się obsłużyć wszystkie tryby adresowe.



Rysunek 3.1: Uproszczony schemat blokowy procesora

Nazwa	Opis
`IR`	rejestr instrukcji
`CU`	jednostka kontrolna (dekoduje tryby adresowe/akcje procesora)
`PC`	licznik programu
`ALU`	jednostka arytmetyczno-logiczna

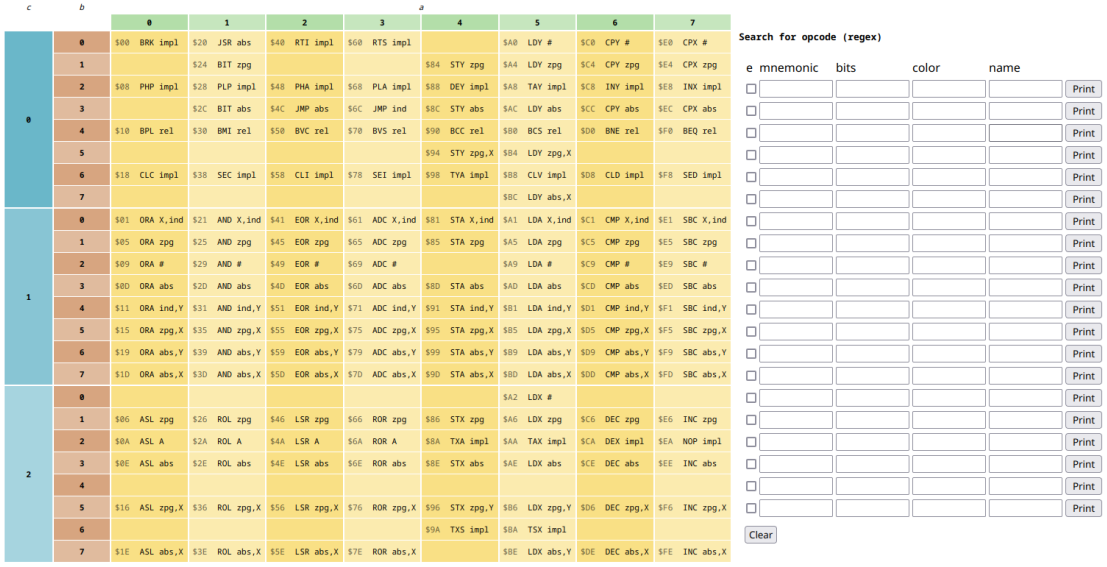
3.1.1 Dekodowanie instrukcji

Dekodowanie trybu adresowego i rodzaju operacji wykonywanej przez procesor było jednym z bardziej wymagających zadań wykonanych w projekcie. Starano się zdekodować je

jak najprościej ale też tak, aby użycie zdekodowanej postaci nie sprawiało problemów.

Pomocne narzędzie

Na podstawie artykułu[3] o zestawie instrukcji 6502 stworzona została przeze mnie prosta strona internetowa.



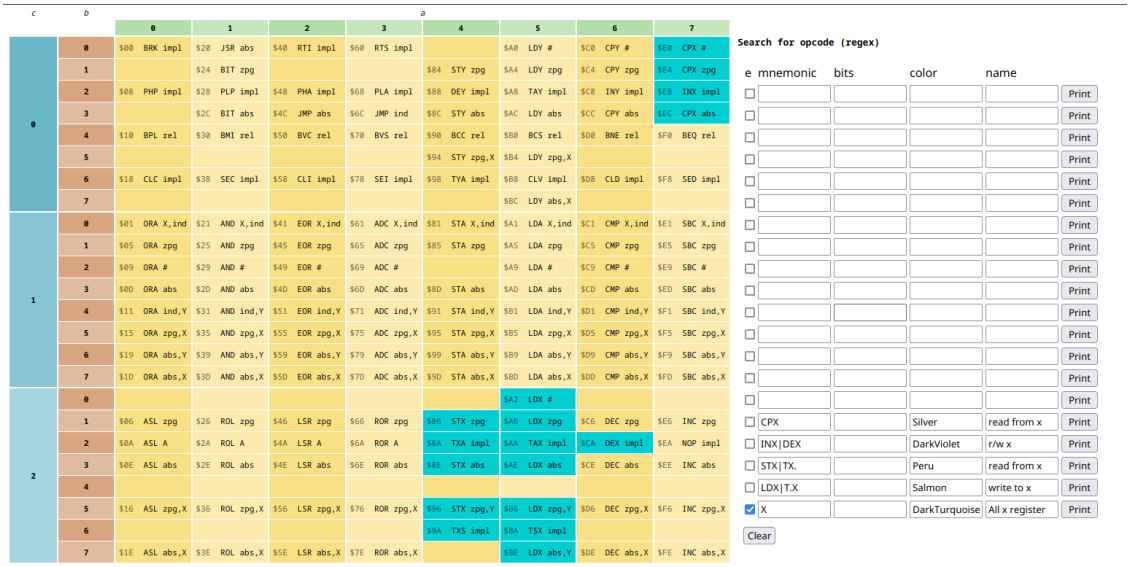
Rysunek 3.2: Instrukcje procesora z wyszukiwarką

Pola wyszukiwarki:

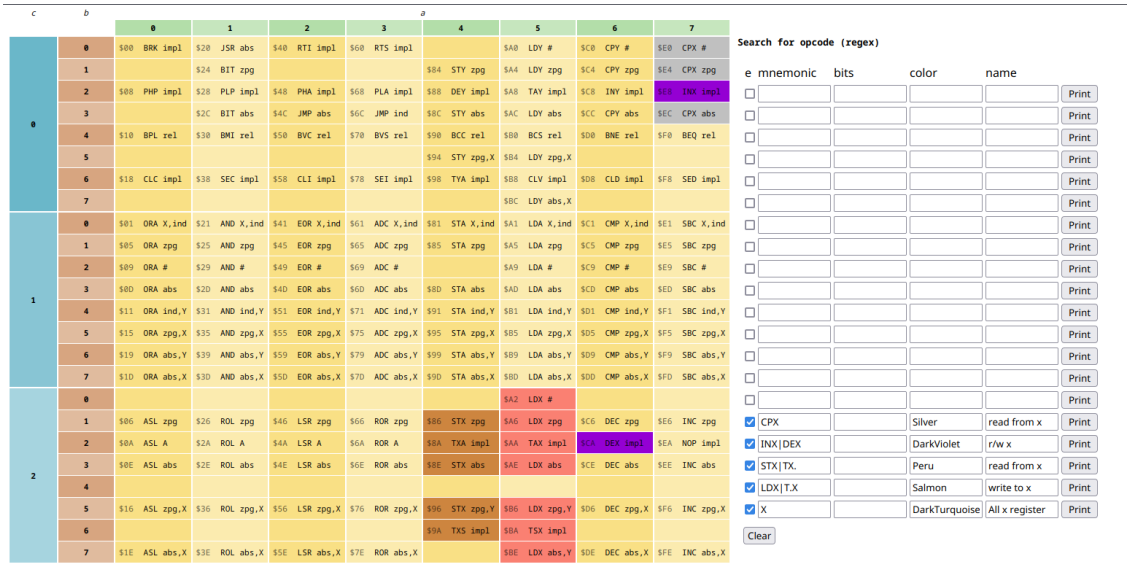
- *e (enable)* – włącza dany filtr,
- *mnemonic* – wyszukuje nazwę instrukcji wyrażeniem regularnym (pominięte gdy puste),
- *bits* – wyszukuje binarną reprezentację kodu instrukcji, format kompatybilny w Verilogiem. Priorytet niższy niż *mnemonic*,
- *color* – jaki kolor ma mieć dopasowana komórka (podwójne kliknięcie generuje losowy kolor),
- *name* – nazwa opisowa.

Strona zapisuje bieżące wyszukiwania w cookies oraz obsługuje funkcje *save()* oraz *load(json)* w konsoli deweloperskiej. Funkcja *save()* zwraca zapisane filtry w postaci JSON, a funkcja *load(json)* ładuje przekazany jako parametr JSON z funkcji *save()*. Pola niżej mają niższy priorytet.

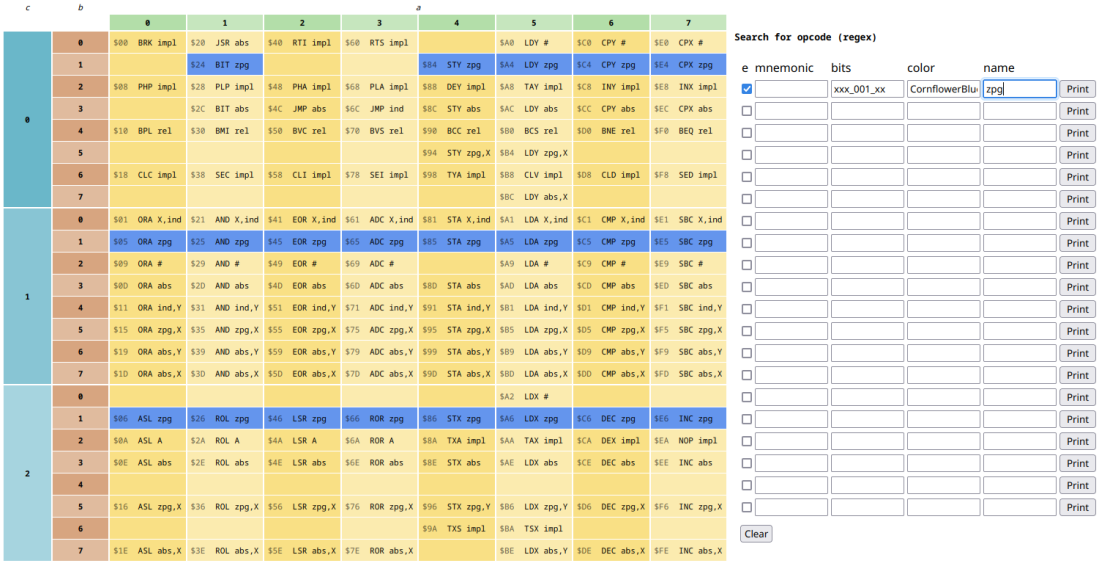
Przykłady wykorzystania narzędzia Strona okazało się pomocna zarówno w dekodowaniu trybów adresowych jak i rodzajów instrukcji.



Rysunek 3.3: Dopasowanie bazowe wszystkich mnemoników z X



Rysunek 3.4: Budowanie grup dopasowań rodzajów instrukcji



Rysunek 3.5: Dopasowanie instrukcji z adresowaniem zpg

Wyrażenie powyżej można bezpośrednio skopiować do *casex* w Verilogu.

Generacja dopasowań

Strona posiada także funkcję generowania tego co dopasowane przyciskiem *Print*.

List of matched opcodes

```
8'hA8,    // TAY impl
8'hAA,    // TAX impl
```

```
// 101 010 00 $A8 TAY impl
// 101 010 10 $AA TAX impl
```

```
/* 101 010 00 $A8 TAY impl  */  { .opcode = 0xA8, .adr_mode = IMPL },
/* 101 010 10 $AA TAX impl  */  { .opcode = 0xAA, .adr_mode = IMPL },
```

Rysunek 3.6: Wygenerowane opisy dla kodów TA. (regex)

Generowanie to służy różnym celom i jest zmieniane w razie potrzeby. Powyższe posłużyło kolejno do:

- bezpośrednie dopasowania w Verilogu,
- komentarze do częściowych dopasowań,

- inicjalizatory struktur do automatycznego testowania.

3.1.2 Dekodowanie trybu adresowania

Poniżej *IR* oznacza *Instruction Register*, czyli rejestr zawierający kod aktualnie załadowanej instrukcji.

Tryby adresowania zostały określone wg dokumentacji procesora 65c02 [2].

Table 6-5 Instruction Timing Chart

Address Mode	Note	Cycle	VPB	MLB	SYNC	Address Bus	Data Bus	RWB
1a. Absolute a ADC, AND, BIT, CMP, CPX, CPY, EOR, LDA, LDX, LDY, ORA, SBC, STA, STX, STY, STZ 16 OpCodes, 3 bytes, 4&5 cycles	(6)	1	1	1	1	PC	OpCode	1
		2	1	1	0	PC+1	AAL	1
		3	1	1	0	PC+2	AAH	1
		4	1	1	0	AA	Data	1/0
1b. Absolute (R-M-W) a ASL, DEC, INC, LSR, ROL, ROR, TRB, TSB 8 OpCodes, 3 bytes, 6 cycles		1	1	1	1	PC	OpCode	1
		2	1	1	0	PC+1	AAL	1
		3	1	1	0	PC+2	AAH	1
		4	1	0	0	AA	Data	1
		5	1	0	0	AA	IO	1
		6	1	0	0	AA	Data	0
1c. Absolute (JUMP) a JMP (4C) 1 OpCode, 3 bytes, 3 cycles		1	1	1	1	PC	OpCode	1
		2	1	1	0	PC+1	New PCL	1
		3	1	1	0	PC+2	New PCH	1
		1	1	1	1	New PC	New OpCode	1
1d. Absolute (JUMP to subroutine) a JSR (20) 1 OpCode, 3 bytes, 3 cycles (different order from N6502)		1	1	1	1	PC	OpCode	1
		2	1	1	0	PC+1	New PCL	1
		3	1	1	0	S	IO	1
		4	1	1	0	S	PCH	0
		5	1	1	0	S+1	PCL	0
		6	1	1	0	PC+2	New PCH	1
		1	1	1	1	New PC	New OpCode	1
2. Absolute Indexed Indirect (a, x) JMP (7C) 1 OpCode, 3 bytes, 6 cycles	(1)	1	1	1	1	PC	OpCode	1
		2	1	1	0	PC+1	AAL	1
		3	1	1	0	PC+2	AAH	1
		4	1	1	0	PC+2	IO	1
		5	1	1	0	AA+X	New PCL	1
		6	1	1	0	AA+X+1	New PCH	1
		1	1	1	1	New PC	OpCode	1
3a. Absolute , X a, x ADC, AND, BIT, CMP, EOR, LDA, LDY, ORA, SBC, STA, STZ 11 OpCodes, 3 bytes, 4,5 and 6 cycles	(1)	1	1	1	1	PC	OpCode	1
		2	1	1	0	PC+1	AAL	1
		3	1	1	0	PC+2	AAH	1
	(6)	4	1	1	0	AA+X	Data	1/0
3b. Absolute, X(R-M-W) a, x ASL, DEC, INC, LSR, ROL, ROR 6 OpCodes, 3 bytes, 7 cycles	(1)	1	1	1	1	PC	OpCode	1
		2	1	1	0	PC+1	AAL	1
		3	1	1	0	PC+2	AAH	1
		4	1	1	0	AAH,AAL+X	IO	1
		5	1	0	0	AA+X	Data	1
		6	1	0	0	AA+X+1	IO	1
		7	1	0	0	AA+X	Data	0

Rysunek 3.7: Tabela w dokumentacji 65c02

```
`define ADR_INVAL       5'd0      // Invalid address mode
`define ADR_DONT_CARE   5'bXXXXX  // don't cares (correct width)

`define ADR_ABS         5'd1    // 1a absolute
`define ADR_ABS_RMW     5'd2    // 1b absolute (RMW)
`define ADR_ABS_JMP     5'd3    // 1c absolute JUMP
`define ADR_ABS_JSR     5'd4    // 1d absolute JUMP to subroutine

`define ADR_ABS_X_IND   5'd5    // 2  absolute indexed X indirect M(op+X

`define ADR_ABS_X_Y     5'd6    // 3a absolute indexed X op+X / 4 absolu
`define ADR_ABS_X_RMW   5'd7    // 3b absolute indexed X op+X (RMW)
```

Rysunek 3.8: Określone kody dla poszczególnych trybów

Przy pomocy powyższego narzędzia dopasowane zostały poszczególne grupy instrukcji do kodów trybów adresowania.

```
if (IR == 8'h20)
    adr_mode = `ADR_ABS_JSR;
else if (IR == 8'h6C)
    adr_mode = `ADR_ABS_IND;
else if (IR == 8'h4C)
    adr_mode = `ADR_ABS_JMP;
else casex (IR)
// IR = aaa bbb cc
// b odd -- well defined
    8'bxxx_001_0x,
    8'b10x_001_1x: adr_mode = `ADR_ZPG;
    8'b0xx_001_1x,
    8'b11x_001_1x: adr_mode = `ADR_ZPG_RMW;

    8'bxxx_011_0x, // 4C, 6C -> see above if
    8'b10x_011_1x: adr_mode = `ADR_ABS;
    8'b0xx_011_1x,
    8'b11x_011_1x: adr_mode = `ADR_ABS_RMW;

    8'bxxx_101_0x,
    8'b10x_101_1x: adr_mode = `ADR_ZPG_X_Y;
    8'b0xx_101_1x,
    8'b11x_101_1x: adr_mode = `ADR_ZPG_X_RMW;

    8'bxxx_111_0x,
    8'b10x_111_1x: adr_mode = `ADR_ABS_X_Y;
    8'b0xx_111_1x,
    8'b11x_111_1x: adr_mode = `ADR_ABS_X_RMW;
```

Rysunek 3.9: Określone tryby adresowania dla grup instrukcji

3.1.3 Dekodowanie rodzaju instrukcji

W celu uproszczenia logiki, zdecydowano się najpierw na dekodowanie mnemoników, a dopiero późniejsze tworzenie z nich zmiennych sterujących.

```
casex (IR)
    8'b011_xxx_01:
        ADC = 1;

    default: ADC = 0;
endcase

casex (IR)
    8'b100_xxx_01:
        STA = 1;

    default: STA = 0;
endcase

casex (IR)
    8'b101_xxx_01:
        LDA = 1;

    default: LDA = 0;
endcase
```

Rysunek 3.10: Dekodowanie mnemoników

```
// output logic
// register read/write
assign from_A = STA | CMP            | TAY | TAX | ORA | AND | EOR | ADC | SBC | ASL_A | ROL_A | LSR_A | ROR_A | PHA;
assign to_A   = LDA                  | TYA | TXA | ORA | AND | EOR | ADC | SBC | ASL_A | ROL_A | LSR_A | ROR_A | PLA;
assign from_X = STX | CPX | INX | DEX | TXA | TXS;
assign to_X   = LDX       | INX | DEX | TAX | TSX;
```

Rysunek 3.11: Tworzenie zmiennych sterujących

Testowanie dekodera Napisano program w C++, który ma zawarte testy co powinien zwracać dekoder. Testy te zostały wygenerowane automatycznie przez w.w. stronę internetową, aby zminimalizować ryzyko pomyłki. Przy użyciu narzędzia Verilator[9] przekształcono kod języka Verilog do klasy w C++. Dzięki temu stało się możliwe porównywanie wyników testów i modułu dekodera. Fragmenty kodu testera:

```
{ .opcode = 0xD0, .adr_mode = REL },
{ .opcode = 0xF0, .adr_mode = REL },
{ .opcode = 0x94, .adr_mode = ZPG_I, .index = INDEX_X },
{ .opcode = 0xB4, .adr_mode = ZPG_I, .index = INDEX_X },
{ .opcode = 0x18, .adr_mode = IMPL },
{ .opcode = 0x38, .adr_mode = IMPL },
```

```
// STA
for (int opcode : { 0x81, 0x85, 0x8D, 0x91, 0x95, 0x99, 0x9D }) {
        tc = test_case_by_opcode(opcode);
        tc->from_A = 1;
        tc->to_mem = 1;
}
```

3.1.4 Maszyna stanów

Maszyna stanów do swojego działania wykorzystuje głównie informacje o aktualnym trybie adresowania oraz stanie w którym się znajduje. Stan 0 jest unikalny – wczytywany jest wtedy IR i testowane jest wejście przerwania. Działanie kolejnych stanów jest uzależnione od trybu adresowania.

```
/* --------------------- Absolute --------
{`ADR_ABS, 3'd1}: addr_abs_step_1();
{`ADR_ABS, 3'd2}: addr_abs_step_2();
{`ADR_ABS, 3'd3}: exec_step_3();

{`ADR_ABS_RMW, 3'd1}: addr_abs_step_1();
{`ADR_ABS_RMW, 3'd2}: addr_abs_step_2();
{`ADR_ABS_RMW, 3'd3}: exec_rmw_step_1();
{`ADR_ABS_RMW, 3'd4}: exec_rmw_step_2();
{`ADR_ABS_RMW, 3'd5}: exec_rmw_step_3();
```

Rysunek 3.12: Adresowanie bezwzględne

```
/* --------------------- Stack Interrupt
// save PCH
{`ADR_STACK_INT, 3'd1}:
begin
    adr_bus <= { 8'h01, S };
    data_bus_out_buf <= PC[15:8];

    if (!reset_routine)
        RW = `RW_WRITE;

    alu_B <= S;
    alu_set_dec();

    next_state_only();
end

// save PCL
{`ADR_STACK_INT, 3'd2}:
begin
    adr_bus <= { 8'h01, alu_out };
    data_bus_out_buf <= PC[7:0];

    alu_B <= alu_out;
    next_state_only();
end
```

Rysunek 3.13: Adresowanie przy przerwaniu

Działanie maszyny stanów Maszyna stanów wykonuje krok przy zboczu opadającym. Reszta systemu reaguje na sygnały na magistralach przy zboczu rosnącym.

3.1.5 Rejestry użytkowe

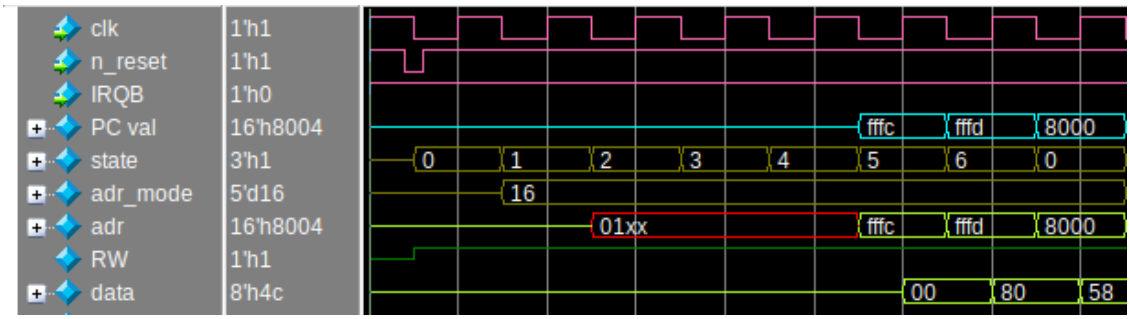
Nazwa	Opis
`A`	akumulator
`X`	rejestr indeksowy
`Y`	rejestr indeksowy
`S`	wskaźnik stosu
`P`	rejestr stanu

3.1.6 Reset procesora

Przebieg restartu jest zgodny z pierwowzorem. Trwa 7 cykli, podczas których ładowany jest licznik programu zawartością pamięci z komórek `$FFFC` i `$FFFD`. Wewnętrznie zrealizowany jest (wg dokumentacji) jako przerwanie, ale zamiast zapisu adresu powrotu i stanu, są odczyty (wartości są ignorowane).

10a. Stack s ABORTB, IRQB, NMIB, RESB 4 hardware interrupts, 0 bytes, 7 cycles	(5)	1	1	1	1	PC	not used	1
		2	1	1	0	PC	not used	1
		3	1	1	0	01,S	Return PCH	0
		4	1	1	0	01,S-1	Return PCL	0
		5	1	1	0	01,S-2	Return P	0
		6	0	1	0	VA	New PCL	1
		7	0	1	0	VA+1	New PCH	1
		1	1	1	1	New PC	New OpCode	1

Rysunek 3.14: Restart procesora jako przerwanie (dokumentacja)



Rysunek 3.15: Restart procesora

Ustawienie stanu na 0 jest częścią "poprzedniej instrukcji"i nie jest liczone do liczby cykli.

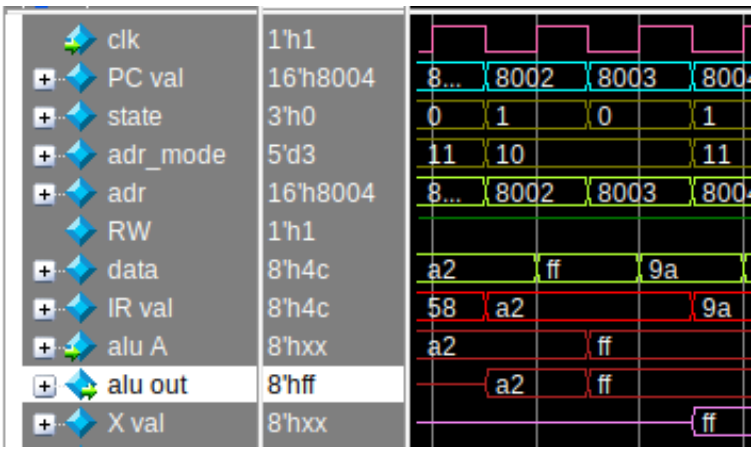
Cykl	Zbocze	Wykonane czynności
1	↓	załadowanie trybu adresowego *stack*
2	↓	przygotowanie adresu
3	↓	przygotowanie adresu, ignorowany odczyt
4	↓	przygotowanie adresu, ignorowany odczyt
5	↓	przygotowanie adresu, ignorowany odczyt
6	↓	przygotowanie adresu, odczyt młodszego bajtu licznika programu
7	↓	odczyt starszego bajtu licznika programu, wyzerowanie stanu

3.1.7 Realizacja dostępu z pamięci

Tutaj zazwyczaj zaczynały się pierwsze problemy, jeżeli architektura maszyny stanów była nieodpowiednia.

[Odczyt] wartość natychmiastowa

Odczyt wartości zawsze wykorzystuje ALU. Jest to o tyle wygodne, że przy operacjach arytmetyczno-logicznych wystarczy tylko je aktywować (domyślnie ALU przekazuje jedno wejście na wyjście). Niesie to ze sobą konieczność zapisu z rejestru wyjściowego ALU do faktycznego rejestru użytkowego, do czego wykorzystywany jest stan 0. Łącznie odczyt trwa 2 cykle + 1 cykl na *writeback* (zapis do rejestru), ale dodatkowy cykl już ładuje następną instrukcje. Nie ma żadnej straty czasowej.

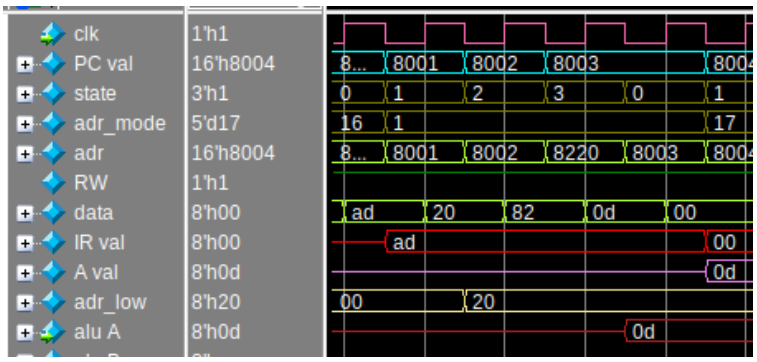


Rysunek 3.16: Odczyt wartości natychmiastowej (2+1 cykli)

Cykl	Zbocze	Wykonane czynności
1	⌝⌞	załadowanie `IR`, zdekodowanie trybu adresowania
2	⌝⌞	zatrzaśnięcie do ALU
+1	⌝⌞	*writeback* do rejestru `X`, załadowanie kolejnej instrukcji

[Odczyt] adres bezwzględny

Procedura jest taka sama jak wcześniej (nadal używamy ALU, jeden dodatkowy cykl na zapis do rejestru pokrywający się z następną instrukcją).

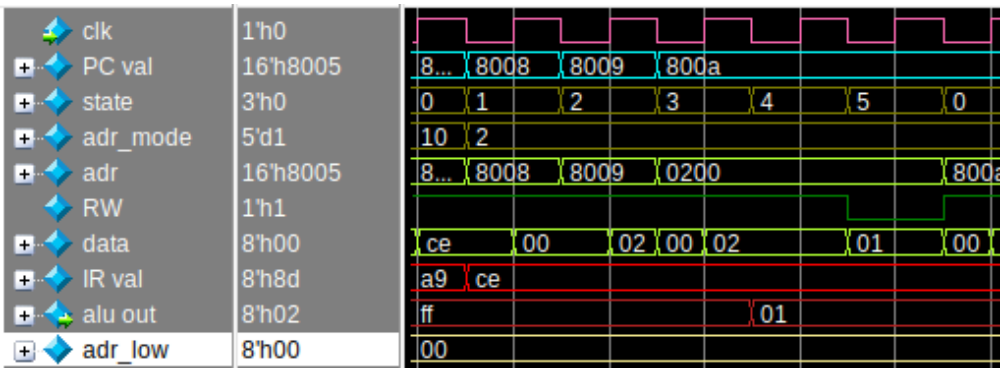


Rysunek 3.17: Odczyt wartości spod adresu bezwzględnego (4+1 cykli)

Cykl	Zbocze	Wykonane czynności
1	↘	załadowanie `IR`, zdekodowanie trybu adresowania
2	↘	odczyt adresu, zatrzaśnięcie do `adr_low` (bufor młodszego bajtu)
3	↘	odczyt adresu, wystawienie adresu na magistralę adresową
4	↘	zatrzaśnięcie do ALU
+1	↘	*writeback* do rejestru `A`, załadowanie kolejnej instrukcji

[Odczyt-Modyfikacja-Zapis] Adres bezwzględny

Procedura jest taka sama jak wcześniej (nadal używamy ALU, jeden dodatkowy cykl na zapis do rejestru pokrywający się z następną instrukcją).

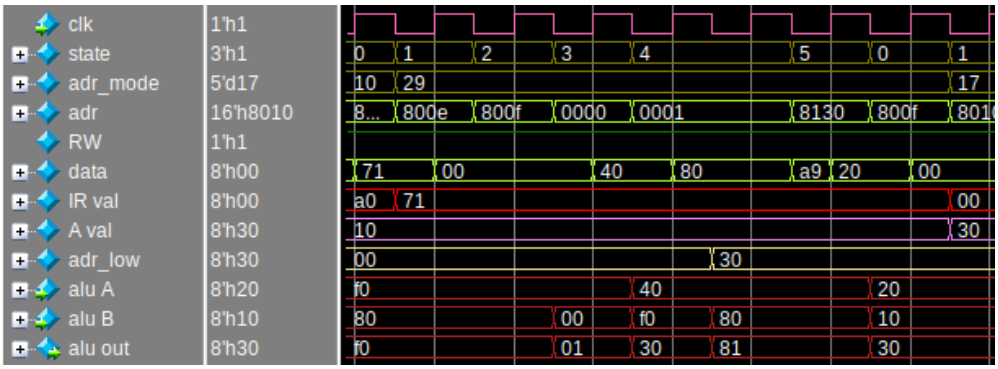


Rysunek 3.18: Odczyt wartości spod adresu bezwzględnego (4+1 cykli)

Cykl	Zbocze	Wykonane czynności
1	↘	załadowanie `IR`, zdekodowanie trybu adresowania
2	↘	odczyt adresu, zatrzaśnięcie do `adr_low` (bufor młodszego bajtu)
3	↘	odczyt adresu, wystawienie adresu na magistralę adresową
4	↘	zatrzaśnięcie do ALU
5	↘	wystawienie wyniku na magistralę danych, zapis (`RW=0`)
6	↘	przygotowanie do odczytania następnej instrukcji

Pośrednie adresowanie stroną zerową z indeksacją

Jednym z bardziej przydatnych trybów adresowania jest tryb *Zero Page Indirect Indexed with y* (symbolicznie `ind, y`). Ładuje on adres ze zmiennej w stronie zerowej, a następnie dodaje do niego wartość rejestru `Y` i tworzy faktyczny adres. Jest to jedyny tryb gdzie programowo możemy dostać się do dowolnego adresu z indeksacją. Przydatny w operowaniu z dużymi obszarami pamięci (większymi niż 256 bajtów, które pozwoliłby zaadresować zwykły tryb indeksowy). Przy dodawaniu Y do adresu może nastąpić przejście granicy stron. Tak też się dzieje w tym przypadku. Pod adresem `$0` mieści się wartość `$8040`, a w rejestrze `Y` wartość `$F0`. Po dodaniu nastąpi przeniesienie. Finalny adres powinien wynieść `$8130`.



Rysunek 3.19: Instrukcja dodawania `adc ind, y` (7+1 cykli)

Cykl	Zbocze	Wykonane czynności
1	↘	załadowanie `IR`, zdekodowanie trybu adresowania
2	↘	odczyt adresu strony zerowej, zatrzaśnięcie do `adr_low` (bufor)
3	↘	zatrzaśnięcie do ALU, inkrementacja adresu
4	↘	odczyt młodszego bajtu adresu, zatrzaśnięcie do ALU, dodanie `Y`
5	↘	odczyt starszego bajtu adresu, wykrycie przejścia granicy stron (inkrementacja)
6	↘	wystawienie adresu na magistralę
7	↘	odczyt operandu
+1	↘	*writeback* sumy do rejestru `A`, załadowanie kolejnej instrukcji

3.1.8 Skok bezwarunkowy

W poprzednich wersjach procesora największym problemem był dostęp do pamięci. Teraz, gdy ten etap został zrealizowany, nadszedł czas na kolejny, bardziej złożony, etap. Przy odpowiednim zdekodowaniu trybu adresowego, dodanie skoku okazało się bardzo proste.

```
/* ---------------------- Absolute JMP ----------------------
{`ADR_ABS_JMP, 3'd1}: addr_abs_step_1();
{`ADR_ABS_JMP, 3'd2}:
begin
    set_PC_adr_bus({data_bus_in, adr_low});
    state_reset();
end
```

Rysunek 3.20: Obsługa instrukcji skoku bezwarunkowego

Wykorzystanie części kodu z trybu adresowania bezwzględnego pozwoliło na dalsze uproszczenia.

3.1.9 Skoki warunkowe

Przed implementacją skoków warunkowych należało dodać rejestr statusowy. Instrukcje aktualizują go zgodnie z dokumentacją procesora [2]. Jednostka kontrolna generuje odpowiednie sygnały:

Nazwa	Opis
`branch_value`	wykonaj skok gdy wartość flagi równa tej zmiennej
`branch_neg`	sprawdź wartość flagi `negative`
`branch_ov`	sprawdź wartość flagi `overflow`
`branch_carry`	sprawdź wartość flagi `carry`
`branch_zero`	sprawdź wartość flagi `zero`

Sygnały te są wykorzystywane w logice skoków.

```
if ((cu_branch_neg   && cu_branch_value == flag_neg  ) ||
    (cu_branch_ov    && cu_branch_value == flag_ov   ) ||
    (cu_branch_carry && cu_branch_value == flag_carry) ||
    (cu_branch_zero  && cu_branch_value == flag_zero ))

    // take branch
    next_consume_put();
else

    // don't take branch
    next_rst_consume();
```

Rysunek 3.21: Logika skoków warunkowych

Jeżeli skok ma być wykonany, maszyna przechodzi do następnego stanu gdzie dokonuje obliczeń przesunięcia. Natomiast gdy skok ma być pominięty, resetuje ona stan do 0 i wykonuje następną instrukcje ignorując bajt przesunięcia.

3.1.10 Podprogramy

Zaprogramowanie obsługi podprogramów, także nie sprawiło większych problemów przy dobrej implementacji maszyny stanów i właściwemu dekodowaniu trybów adresowych (skok i powrót z podprogramu to osobne tryby adresowe obejmujące po jednej instrukcji). Pomocna przy programowaniu była dokumentacja w której opisano co i kiedy jest zapisywane/odczytywane z pamięci.

1d. Absolute (JUMP to subroutine) a		1	1	1	1	PC	OpCode	1
JSR (20)		2	1	1	0	PC+1	New PCL	1
1 OpCode, 3 bytes, 3 cycles		3	1	1	0	S	IO	1
(different order from N6502)		4	1	1	0	S	PCH	0
		5	1	1	0	S+1	PCL	0
		6	1	1	0	PC+2	New PCH	1
		1	1	1	1	New PC	New OpCode	1

Rysunek 3.22: Skok do podprogramu (dokumentacja)

10d. Stack (Return from subroutine) s		1	1	1	1	PC	OpCode	1
RTS		2	1	1	0	PC+1	not used	1
1 OpCode, 1 byte, 6 cycles		3	1	1	0	PC+1	not used	1
		4	1	1	0	S+1	Return PCL	1
		5	1	1	0	S+2	Return PCH	1
		6	1	1	0	PC+1	IO	1
		1	1	1	1	Return PC	New OpCode	1

Rysunek 3.23: Powrót z podprogramu (dokumentacja)

3.1.11 Przerwania

Zostało zaimplementowane jedno przerwanie (z dwóch): aktywowane poziomem niskim, maskowalne. `NMI` (*Non-Maskable Interrupt*) nie zostało zaimplementowane z uwagi na podobny mechanizm działania. Maskowalne przerwanie używane jest w finalnym systemie do odczytywania znaków z kontrolera klawiatury PS/2.

3.1.12 Rozszerzenia 65c02

Nowe instrukcje w 65c02 względem 6502:

Mnemonik	Opis
`BRA`	skok warunkowy zawsze (jeden bajt mniej kodu niż `JMP`)
`TSB` / `TRB`	sprawdź i ustaw/zresetuj pin
`BBR` / `BBS`	skok warunkowy w zależności od wartości bitu
`RMB` / `SMB`	ustawienie lub reset bitu
`STZ`	zapisz zero
`INC A` / `DEC A`	inkrementacja/dekrementacja akumulatora
`PHY` / `PLY`	operacje z rejestrem `Y` na stosie
`PHX` / `PLX`	operacje z rejestrem `X` na stosie
`WAI`	czekaj na przerwanie
`STP`	stop

Nowe tryby adresowe:

- `ZPG_IND` – Pośrednie adresowanie stroną zerową
- `ABS_X_IND` – Pośrednie adresowanie adresem bezwzględnym z indeksacją (wykorzystywane tylko przy instrukcji `JMP (abs, x)`)
- `ADR_IMPL_STP` – Instrukcja `STP`
- `ADR_IMPL_WAI` – Instrukcja `WAI`
- `ADR_REL_BIT` – Adresowanie względne, instrukcje `BBR` / `BBS`
- `ADR_ZPG_BIT` – Adresowanie strony zerowej, instrukcje `RMB` / `SMB`

Nie zaimplementowano żadnych rozszerzeń z nowszej wersji 6502 od producenta *Western Design Center*.

3.1.13 Kompletność implementacji

Nie zaimplementowano następujących trybów adresowania:

- `ADR_STACK_BRK` – Używany przez instrukcje `BRK` (przerwanie programowe)

Nie zaimplementowano trybu dziesiętnego.
Nie zaimplementowano następujących instrukcji:

- `BRK` – Przerwanie software

Nie zaimplementowano przerwania NMI.
Procesor nie posiada pinów statusowych.

3.2 Jednostka diagnostyczna

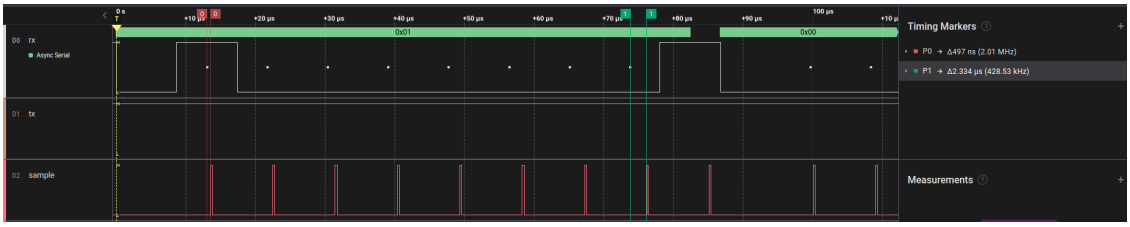
Układ diagnostyczny stał się konieczny, gdy procesor zaczął być testowany na płytce FPGA (wcześniej testowany był w symulatorze *Questa*). Relatywnie długi czas kompilacji całego projektu utrudniałby aktualizację pamięci ROM o nowe instrukcje do przetestowania.

3.2.1 UART

Warstwa fizyczna komunikacji opiera się o protokół UART. Ma to tę zaletę, że dostępne są na rynku odpowiednie przystawki UART do komputerów PC, co pozwala na bezpośrednią komunikację z płytką.

Próbkowanie

Wybrana została popularna częstotliwość sygnałowa 115200 znaków na sekundę (brak parzystości, jeden bit stopu). Jako bazę do pobierania próbek stanowi zegar jak najbliższy 3.6864MHz (realnie 3.684211MHz, 0.06% różnicy). Taki zegar dzieli się przez wiele popularnych częstotliwości (dodatkowo produkowane są kwarce z dokładnie takim zegarem). Dla 115200b/s należy próbkować sygnał wejściowy co 32 cykle takiego zegara. Na poniższym rysunku widać, że próbkowanie działa. Próbka mieści się w danym bicie.



Rysunek 3.24: Próbkowanie sygnału UART

Obsługa

```
module UART (
    input wire clk,
    input wire n_reset,

    input wire rx,
    output reg rx_ready, // module sets it to 1 when transfer finishes
                         // for one clock period
    output reg [7:0] rx_data,

    output reg tx,
    input wire tx_write,    // user sets to 1 to start a transfer
                            // one pulse is sufficient
    output reg tx_finished, // module sets to 1 when transfer finished
                            // for one clock pulse
    input wire [7:0] tx_data,
```

Rysunek 3.25: Definicja modułu UART

Sterowanie jednostką UART realizowane jest przez 2 sygnały:

Nazwa	Opis
`rx_ready`	odbiór bajtu zakończony, można odczytać `rx_data`
`tx_write`	rozpocznij nadawanie bajtu znajdującego się na `tx_data`

Jednostka zgłasza zakończenie nadawania sygnałem `tx_finished`. Sygnały `rx` i `tx` to sygnały danych UART. `clk` i `n_reset` to sygnały systemowe (zegar i reset).

Podłączenie

Przystawka UART do PC działa z napięciem 3.3V, więc obyło się bez dodatkowych elementów. Pin `Rx` konwertera podpięty został do `GPIO0`, a pin `Tx` do `GPIO1`.

3.2.2 *Debug unit*

Sama jednostka diagnostyczna (ang. *debug unit*, skrót. `dbgu`). Jest zbudowana na podstawie modułu UART. Obsługuje odbiór/nadawanie dwóch bajtów danych. Poniżej zostały opisane jej funkcje.

Dostęp do pamięci

Podstawową i najważniejszą funkcją modułu jest dostęp do pamięci ROM i RAM. Pozwala to na modyfikowanie programu jak i sprawdzanie wyników obliczeń w pamięci RAM. Wewnątrz modułu zaimplementowano rejestr wskaźnikowy, który po każdej instrukcji dostępu do pamięci jest inkrementowany. Dzięki temu przyspieszony został dostęp sekwencyjny. Zdefiniowano 5 instrukcji dostępu do pamięci:

	Request		Response		
	16-bit		16-bit		Always 16-bit of data is send and received
	Byte 0	Byte 1	Byte 0	Byte 1	
	Instruction	Data	Data		
Set address pointer low	0x01	Addr low	echo request		sets address pointer, it's incremented
Set address pointer high	0x02	Addr high	echo request		After every access to memory
Get address pointer	0x03		Addr ptr low	Addr ptr high	gets address pointer
Write to memory 1-byte	0x04	Value (ptr)	echo request		uses address pointer, increments by 1
Read memory 2-byte	0x05		Value (ptr)	Value (ptr+1)	uses address pointer, increments by 2

Rysunek 3.26: Instrukcje dostępu do pamięci modułu `dbgu`

Dostęp do rejestrów

Drugą ważną funkcją jest dostęp do aktualnych wartości rejestrów procesora. 8-bitowe rejestry zostały zgrupowane w pary, aby zaoszczędzić cykle odczytu.

	Request		Response		
	16-bit		16-bit		Always 16-bit of data is send and received
	Byte 0	Byte 1	Byte 0	Byte 1	
	Instruction	Data	Data		
Get A + S	0x10		A	S	
Get X + Y	0x11		X	Y	
Get IR + P	0x12		IR	P	
Get PC	0x13		PC low	PC high	

Rysunek 3.27: Instrukcje dostępu do rejestów modułu `dbgu`

Zarządzanie wykonaniem

Trzecią grupę instrukcji stanowią instrukcje zarządzające pracą samego procesora. Odpowiadają one za reset procesora, uruchomienie na określoną i nieokreśloną liczbę cykli.

	Request		Response		
	16-bit		16-bit		Always 16-bit of data is send and received
	Byte 0	Byte 1	Byte 0	Byte 1	
	Instruction	Data	Data		
Run cycles	0x20	Cycles	echo request		run processor for provided number of cycles
Perform cpu reset on next cycle	0x21		echo request		
Set CPU on free run	0x22	Enabled	echo request		

Rysunek 3.28: Instrukcje zarządzania wykonaniem modułu `dbgu`

3.2.3 Automatyczne testy

Implementacja komunikacji z procesorem umożliwiła na napisanie automatycznych testów różnych funkcji. Skrypt w języku Python bazując na obsłudze wyżej opisanego protokołu mógł dowolnie manipulować pamięcią, odczytywać wartości rejestrów i uruchamiać procesor na określoną liczbę cykli zegara. Został wykorzystany assembler `vasm` [10], dzięki któremu można pisać kod w assemblerze 6502 zamiast w kodzie maszynowym.

Konstruktor klasy `Test` przyjmuje argumenty:

- nazwa testu
- program w assemblerze (wielolinijkowy string)
- liczba cykli
- wiele argumentów oznaczające testowane wartości

Poniżej przykładowe testy:

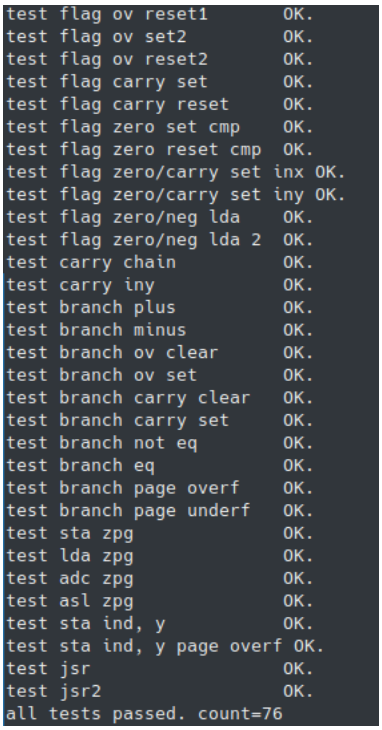
```
tests.append(Test("load imm A/X/Y",
    """
      .org $8000
      lda #$20
      ldx #$21
      ldy #$22
    """, 1+2*3,
    "A=20", "X=21", "Y=22"))
```

Rysunek 3.29: Test ładowania wartości

```
tests.append(Test("asl zpg",
    """
      .org $8000
      lda #$28
      sta $00
      asl $00
    """, 1+2+3+5,
    "M(00)=50"))
```

Rysunek 3.30: Test przesunięcia w lewo wartości w pamięci

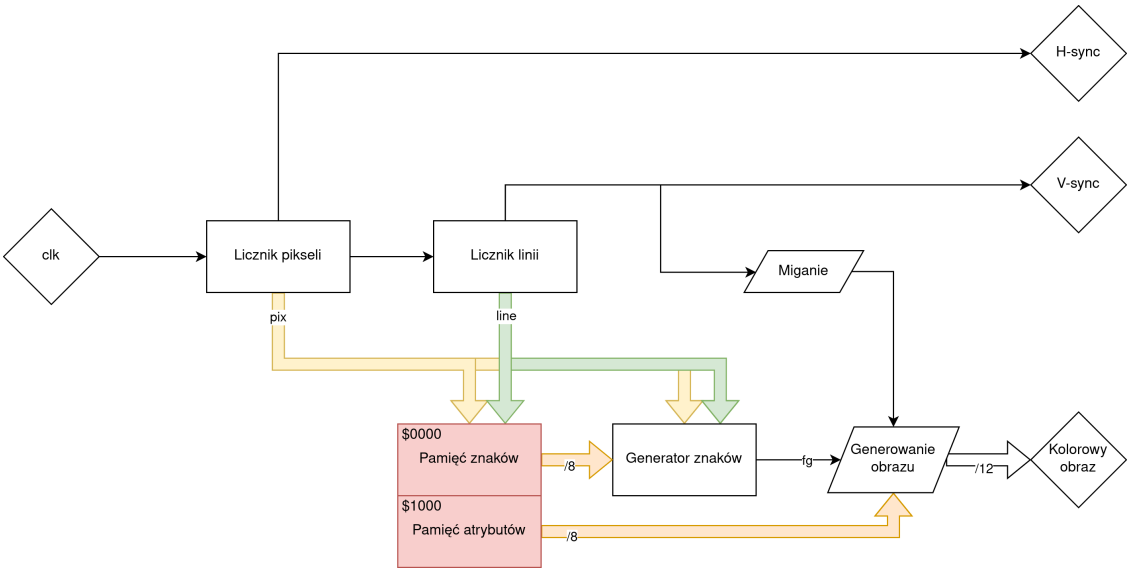
Program wykonujący testowanie:



Rysunek 3.31: Automatyczne wykonanie testów

3.3 Karta graficzna

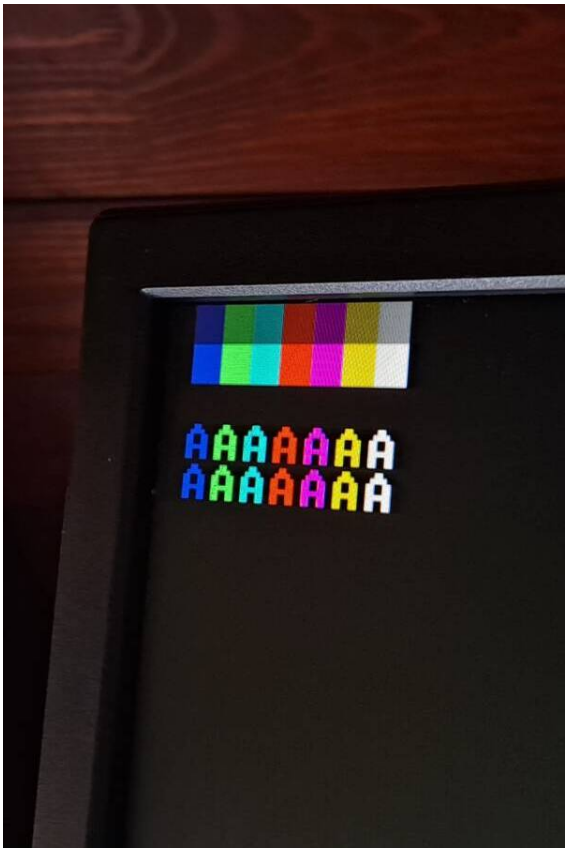
Karta grafiki została zaimplementowana jako prosty moduł oparty o 2 liczniki. Jeden z nich liczy piksele w linii i generuje sygnał synchronizacji poziomej. Drugi liczy linie na ekranie i generuje sygnał synchronizacji pionowej. Indeksy piksela i linii są przekazywane do pamięci znaków i atrybutów a także do generatora znaków, które generują obraz.



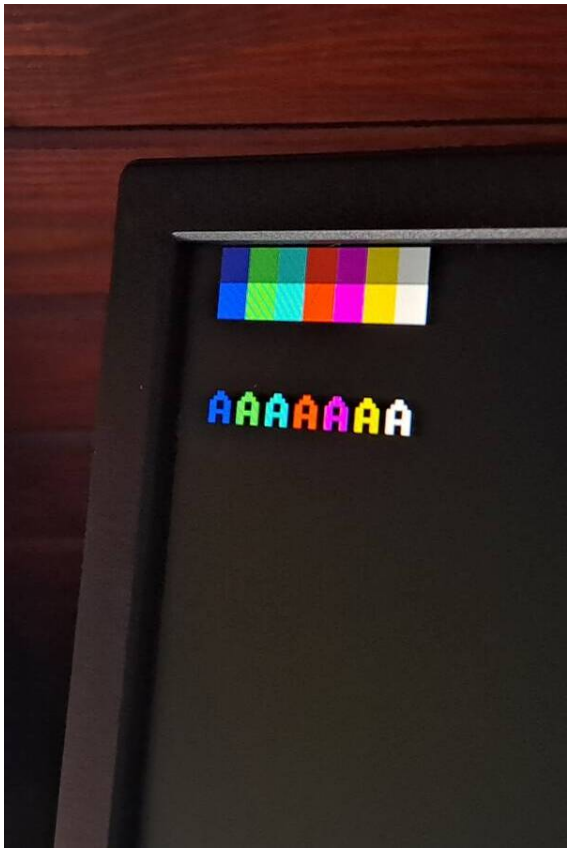
Rysunek 3.32: Schemat blokowy karty graficznej

Pamięć atrybutów jest przesunięta względem pamięci znaków o `$1000` (4096 bajtów). Są one zrealizowane jako ta sama pamięć. Najpierw odczytywany jest kod znaku. Kod jest przekazywany do generatora znaków (pamięć stała 2KBit), a z pamięci znaków/atrybutów odczytywany jest atrybut. Generator znaków generuje informacje o tym, czy dany piksel w danej linii w znaku jest tłem czy kolorem pierwszoplanowym. Atrybuty generują kolor i dodatkowe cechy. Postać wartości atrybutów (R/G/B – składowa czerwona/zielona/niebieska koloru):

Bit	**Nazwa**	**Opis**
7	miganie znaku	Gdy 0 – znak miga
6	R pierwszego planu	
5	G pierwszego planu	
4	B pierwszego planu	
3	tło jaśniejsze	Gdy 1 – tło jest jaśniejsze
2	R tła	
1	G tła	
0	B tła	



Rysunek 3.33: Test możliwości



Rysunek 3.34: Test możliwości – miganie

3.3.1 Kompletność implementacji

Karta graficzna wbrew początkowym założeniom posiada tylko tryb tekstowy.

3.4 Kontroler klawiatury PS/2

Moduł komunikacji z klawiaturą został zrealizowany w podobny sposób do modułu UART. Dane są pobierane z linii danych gdy zegar systemowy wykryje zbocze opadające na linii zegarowej. Dodatkowo, aby poprawić stabilność pracy, sygnały `ps2_clk` i `ps2_data` przechodzą przez podwójne przerzutniki D. Po odebraniu 11 bitów (bit startu + 8 bitów danych + parzystość + bit stopu) zgłaszane jest przerwanie linią `sys_irq`. Moduł posiada 2 rejestry:

Rejestr statusowy

Indeks 0.

Bit	Nazwa	Opis
7	odbieranie zakończone	
6	błąd odbioru	błąd bitu startu/stopu, błąd parzystości
5	kod `F0`	odbiór kodu klawisza poprzedzony kodem `F0` (*break*)
4	kod `E0`	analogicznie j.w. dla kodu `E0` (*extended*)

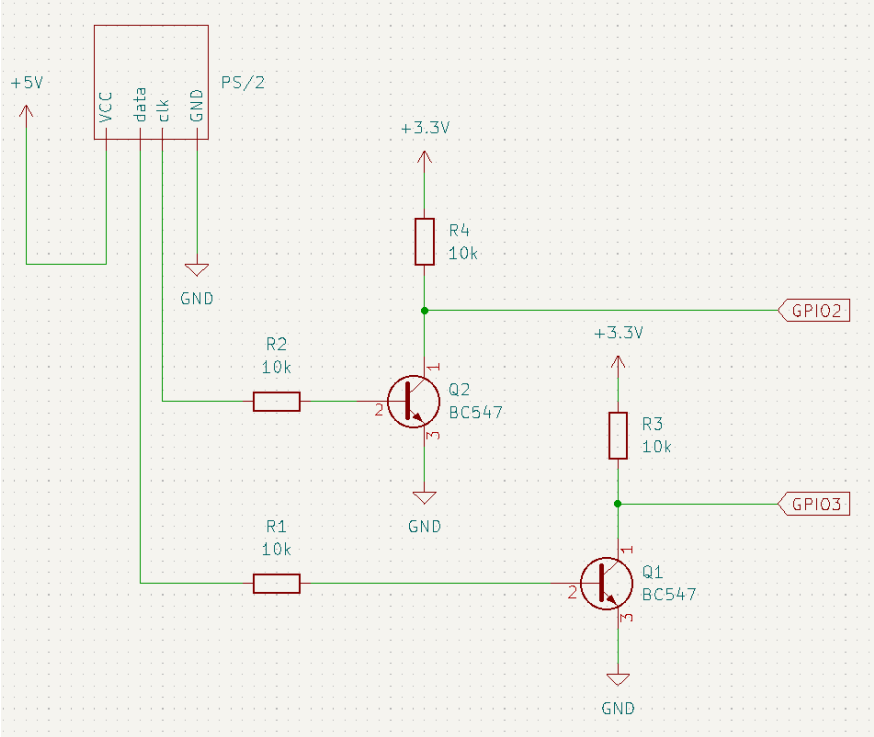
Reszta bitów nie jest używana.

Rejestr danych

Indeks 1.
Zawiera kod naciśniętego klawisza.

Konwersja poziomów

Klawiatura PS/2 działa z napięciem 5V, a płytka FPGA z napięciem 3.3V. Konieczna była konwersja poziomów.



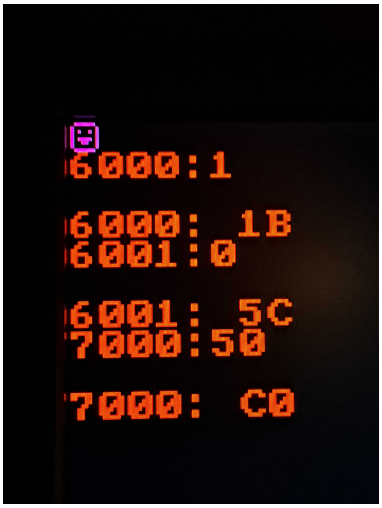
Rysunek 3.35: Podłączenie klawiatury PS/2 do płytki FPGA

Tranzystory odwróciły sygnał. Ponowna negacja zachodzi już w układzie FPGA.

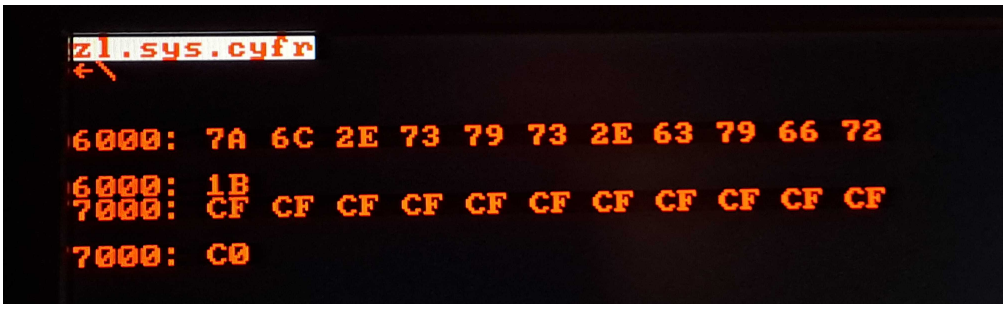
Rozdział 4

Efekt końcowy

Na komputerze udało się uruchomić program **WozMon** [11]. Program ten został napisany przez Steve'a Wozniak'a w 1976 roku na komputer Apple I. Posiadał on procesor 6502, więc lista rozkazów jest zgodna z moim komputerem. Koniecznie było tylko zmodyfikowanie części kodu odpowiedzialnych za wczytywanie, dekodowanie i wyświetlanie znaków.



Rysunek 4.1: Procedura wypisania znaku i zmiany koloru



Rysunek 4.2: Wyświetlenie napisu

Filmy pokazujące działanie WozMon'a zostały dostarczone w folderze `filmy`. Lista filmów:

Film	Opis
`list_chars`	uruchomienie programu testowego Apple I
`mem_rw`	Zapis/odczyt pamięci
`vga_clr_scr`	Wyczyszczenie ekranu poprzez napisanie procedury (instrukcje `JSR` i `JMP`)
`vga_colors`	Zmiana kolorów znaków
`vga_scr_color`	Zmiana wartości zmiennej odpowiedzialnej za kolor ekranu + wyczyszczenie

Bibliografia

[1] *Build a 6502 computer* – Oryginalna inspiracja do próby zrozumienia komputerów poprzedniego stulecia.

[2] *W65C02S 8–bit Microprocessor* – Dokumentacja procesora 65c02 (rok 2004)

[3] *6502 Instruction Set* – Obszerny artykuł o próbie zdekodowania instrukcji 6502.

[4] *6502 Colorized Block Diagram* – kolorowana wersja schematu blokowego 6502. Niewiążąca inspiracja.

[5] *Let's build a video card!* – Oryginalna inspiracja do budowy własnej karty graficznej.

[6] *Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!* – Artykuł o nieblokujących przypisaniach w Verilogu.

[7] *Verilog HDL Coding* – Jak pisać dobry kod w Verilogu.

[8] *Quick Start Guide to Verilog* – Książka z której nauczyłem się Veriloga.

[9] *Verilator* – narzędzie do transpilacji Veriloga do C/C++. Umożliwiło automatyczne testowanie różnych modułów.

[10] *VASM* – assembler obsługujący m.in. 6502

[11] *WozMon* – główny program Apple I