RISC-V – budujemy własny mikrokontroler (3)

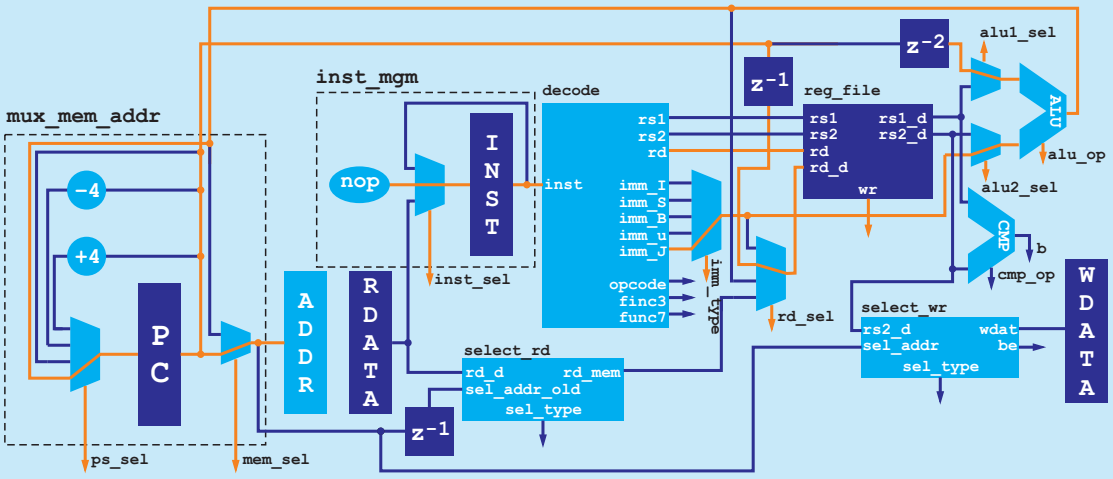
Otwarte oprogramowanie jest spotykane w wielu dziedzinach od kilkudziesięciu lat. W wielu przypadkach wypierało swoich płatnych konkurentów. Zadaniem, jakie przed sobą stawiają członkowie projektu RISC-V, jest przeprowadzenie podobnej rewolucji w dziedzinie procesorów.

FTP WSZYSTKIE POPRZEDNIE ODCINKI KURSU SĄ DOSTĘPNE NA STRONIE WWW.MEDIA.AVT.PL

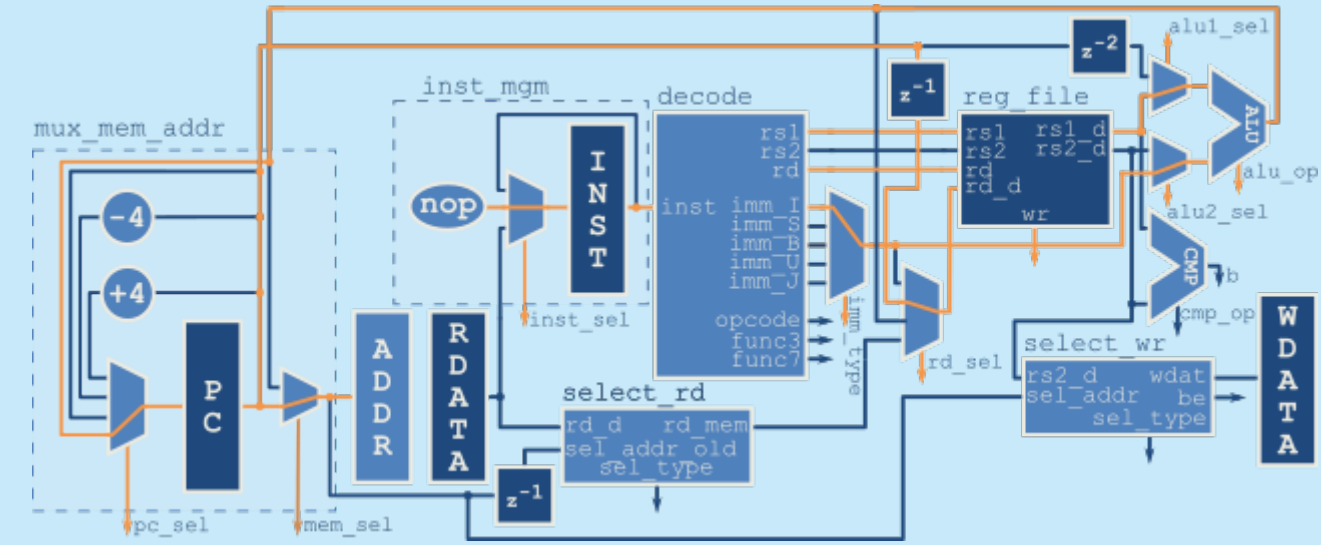
Skoki bezwarunkowe

Przyjrzyjmy się teraz trochę bardziej skomplikowanym poleceniom: skokom bezwarunkowym. Pierwszy z nich ***jal*** jest pokazany na **rysunku 29**. Jak już wiemy pozwala on na dodanie do rejestru PC stałej przy równoczesnym zapisaniu adresu następnej instrukcji do rejestru. Multiplekser *imm_type* wybiera stałą typu J, a *alu2_sel* wpuszcza ją do ALU. Pierwszym operandem wybranym przez *alu1_sel* jest „płynąca wraz z rozkazem" wartość rejestru PC. Ścieżka *alu_op* wybiera dodawanie. *pc_sel* podaje wynik do rejestru PC. Ponieważ rozkaz z pod nowego adresu wejdzie do fazy wykonania dopiero za dwa cykle część kontrolna będzie musiała zastąpić dwa kolejne rozkazy instrukcjami *nop*, co obrazuje konfiguracja rejestru *inst_sel*. Natomiast do rejestru *rd* zostanie wpisany adres następnego rozkazu, który byłby wykonany gdyby skok nie nastąpił. Jest to wartość PC opóźniona o 1 takt zegara.

Drugim rodzajem skoku bezwarunkowego jest instrukcja ***jalr***. Jak widzimy na **rysunku 30** jest ona bardzo podobna. Jednak tym razem kolejny rozkaz jest pobrany z adresu będącego sumą wartości



Rysunek 29. Przepływ danych dla instrukcji *jal*



Rysunek 30. Przepływ danych dla instrukcji *jalr*

z rejestru i (krótszej) stałej. Ponieważ najważniejsze elementy obsługi obu rozkazów są analogiczne przyjrzymy się dokładnie tylko pierwszej z nich: *jal*. Jako przykład posłuży nam króciutki kod z **listingu 2**. Dla przypomnienia został on przedstawiony w **tabeli 5**, ale tym razem, także w postaci binarnej i z podaniem adresów w pamięci. Pierwsza instrukcja zeruje rejestr x5, a następnie w nieskończonej pętli następuje jego inkrementacja.

Potok instrukcji dla początku i dwóch pierwszych obiegów pętli pokazuje **rysunek 31**. Jak już wiemy zaraz po resecie przez dwa takty zegara część sterująca wymusza dwie instrukcje *nop*. W trzecim cyklu instrukcja spod adresu 0x00 jest wykonywana, czyli następuje wyzerowanie rejestru x5. W kolejnym takcie wykonana jest pierwsza instrukcja z ciała pętli. Widzimy jednak, że w tym kroku ustawiony już jest adres 0x0c. Kolor czerwony symbolizuje, że ten fragment pamięci nie zawiera żadnych sensownych danych. Na tym etapie mikrokontroler jeszcze nie wie, że za chwile nastąpi skok, a ładowany adres jest błędny. Dopiero w następnym cyklu jest wykonana instrukcja skoku. W tym momencie w potoku znajdują się już dwie nieprawidłowe instrukcje. Tu wchodzi część kontrolna, która dokładnie tak samo jak w przypadku resetu wymusza zastąpienie dwóch kolejnych rozkazów instrukcją *nop*. Widzimy, że przez ten czas rdzeń wyczyści potok z nieprawidłowych rozkazów, a docelowa instrukcja spod adresu 0x04 dojdzie do fazy wykonaj. Widzimy więc, że w przeciwieństwie do operacji na stałych i rejestrach skok zajmuje aż trzy takty zegara. Jest to ceną jaką płacimy za potokowe przetwarzanie instrukcji. Jak łatwo możemy policzyć jeden obieg pętli zajmuje cztery cykle zegara.

Zachęcam do samodzielnego uruchomienia symulacji rdzenia z przygotowanym programem. Tak jak wcześniej uruchamiamy ModelSima, przechodzimy do głównego folderu repozytorium i uruchamiamy:

```
do scripts/simulate_jal.do
```

Powinniśmy zobaczyć przebiegi podobne do tych z **rysunku 32**. Jeżeli interesują nas przebiegi innych sygnałów możemy je przeciągnąć z panelu *Objects* (ang. obiekty) i umieścić w panelu *Wave* (ang. fale). Następnie musimy wrócić na początek symulacji wpisując polecenie:

```
restart -f
```

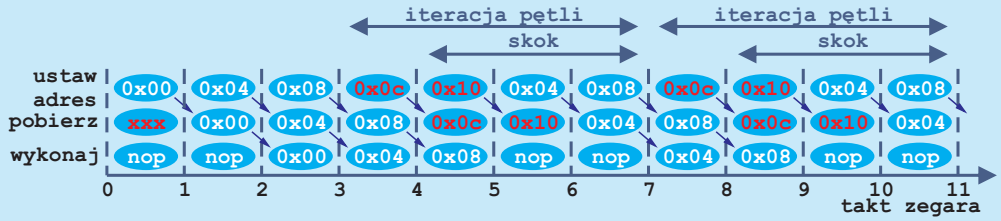
A następnie uruchomić symulację (w tym przypadku na 13 µs):

```
run 13us
```

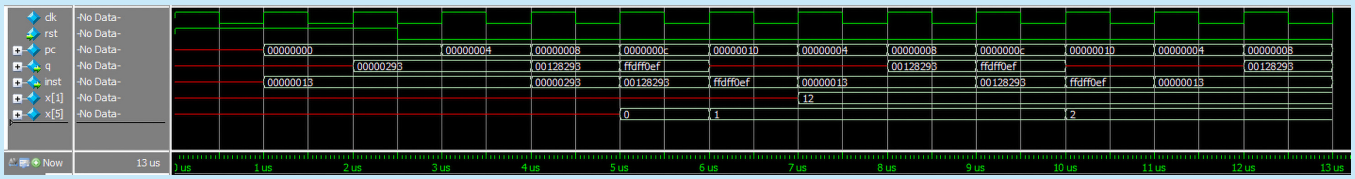
Podobnie jak poprzednio w pierwszych dwóch wierszach mamy sygnał zegarowy i reset. Gdy reset opadnie rozpoczyna się praca rdzenia. Aż do momentu, gdy zostanie wykonana pierwsza instrukcja wartość rejestru x5 (ostatni wiersz) jest nieustalona. Symbolizuje to czerwona linia. Dopiero pierwszy rozkaz powoduje jego wyzerowanie. W kolejnym takcie następuje pierwsza inkrementacja x5. W kolejnym cyklu rozpoczyna się wykonanie skoku. Widzimy, że do rejestru x1 wpisany został adres powrotu, czyli 12 (szesnastkowo 0x0c). Warto też zwrócić uwagę, że odczyt niezainicjalizowanej pamięci powoduje pojawienie się w symulacji czerwonej kreski,

Tabela 5. Program demonstrujący skok bezwarunkowy (code/jal.S)

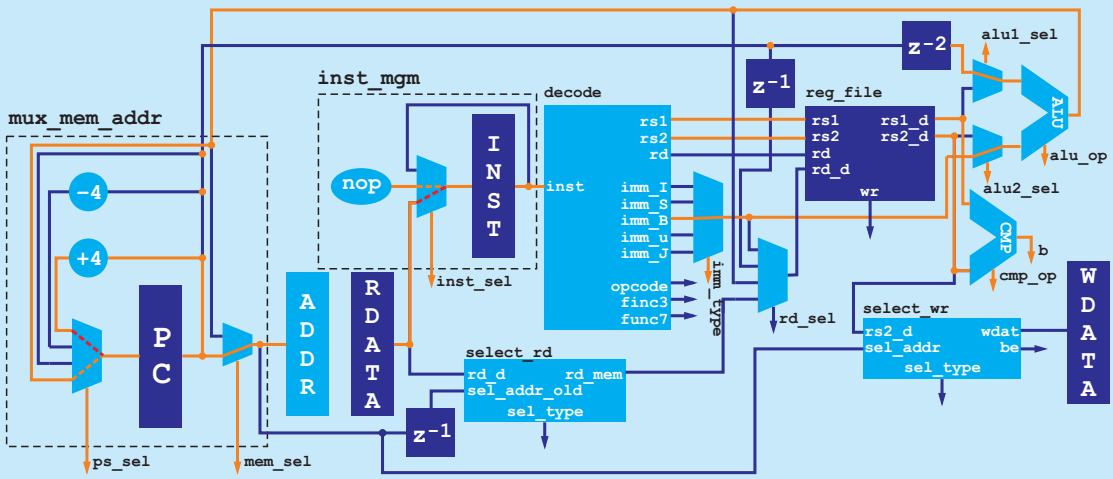
Adres	Binarny	Po assemblacji	Kod
0x00	0x00000293	addi x5, x0, 0	addi x5, x0, 0
			loop:
0x04	0x00128293	addi x5, x5, 1	addi x5, x5, 1
0x08	0xffdff0ef	jal x1, -4	jal x1, loop



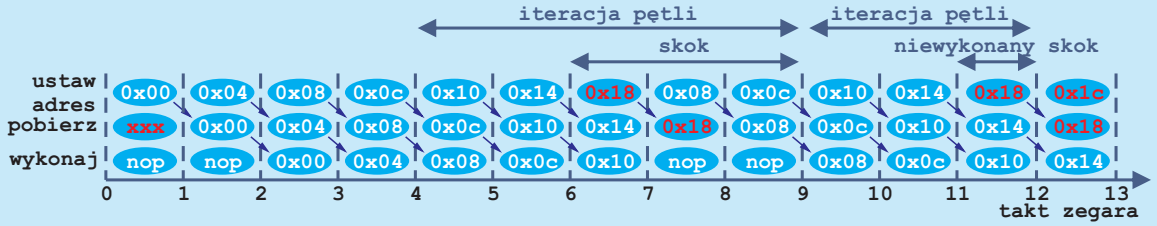
Rysunek 31. Potok instrukcji przy wykonywaniu programu z tabeli 5. Numery określają spod jakiego adresu pochodzi rozkaz



Rysunek 32. Symulacja wykonania kodu z tabeli 5



Rysunek 33. Przepływ danych dla instrukcji warunkowych. Ścieżki zależne od warunku są zaznaczone liniami przerywanymi: pomarańczowe dla wykonanego, a czerwone dla niewykonanego skoku.



Rysunek 34. Potok instrukcji przy wykonywaniu programu z tabeli 6. Numery określają spod jakiego adresu pochodzi rozkaz

czyli nieznanego stanu. Nie zostanie on jednak wykonany, ponieważ przez kolejne dwa cykle będzie ona nadpisany instrukcją o kodzie 0x00000013, czyli *nop*. Licząc liczbę cykli przypadających na kolejne wartości w rejestrze x5 widzimy, że zgodnie z oczekiwaniem obrót pętli trwa cztery cykle zegara.

Skoki warunkowe

Skoki bezwarunkowe pozwalają na wywoływanie i powroty z podprogramów. Jeżeli jednak chcemy, aby nasz program podejmował decyzję potrzebujemy skoku warunkowego, czyli takiego, które zachodzi jedynie, gdy spełniony jest określony warunek. Przepływ danych przedstawia **rysunek 33**. Z rozkazu odczytywane są numery rejestrów, których wartości są przekazywane do CMP. Za pomocą ścieżki *cmp_op* wybierany jest rodzaj warunku do sprawdzenia. Jeżeli został on spełniony ścieżka b przybiera stan wysoki. Równocześnie ALU sumuje wartość rejestru PC (opóźnioną o dwa cykle) ze stałą bezpośrednią. Na podstawie stanu ścieżki b część kontrolna wybiera, czy kolejna wartość PC zostanie załadowana z ALU (skok), czy przyjmie wartość o 4 większą od poprzedniej (brak skoku). Jeżeli skok nastąpi, to analogicznie jak dla instrukcji *jal* koniecznie będzie wyczyszczenie potoku poprzez wymuszenie instrukcji *nop* przez dwa cykle zegara. W **tabeli 6** znajduje się znany nam już kod z **listingu 4**, ale tym razem z podaną lokalizacją poszczególnych rozkazów w pamięci. Na **rysunku 34** pokazany jest potok instrukcji wykonywanych podczas obu iteracji wewnętrznej pętli.

Tak jak we wcześniejszych przykładach efekt pracy pierwszego rozkazu zostanie zapisany na 3 zboczu zegara. Pierwszy skok warunkowy ma miejsce podczas taktów od siódmego do dziewiątego. Ponieważ został on wykonany, konieczne jest wyczyszczenie potoku. Następnie w takcie 12 następuje drugie wywołanie instrukcji. Tym razem warunek nie jest spełniony. Ponieważ zwykły potok nie został zaburzony, wykonanie rozkazu zajęło tylko jeden takt zegara. W następnym kroku następuje przejście do rozkazu z adresu 0x14, czyli znanego już nam skoku bezwarunkowego do początku programu.

Tabela 6. Program demonstrujący skok bezwarunkowy (code/branch.S)

Adres	Binarny	Po assemblacji	Kod
			start:
0x00	0x00200093	addi x1, x0, 2	addi x1, x0, 2
0x04	0x00000113	addi x2, x0, 0	addi x2, x0, 0
			loop:
0x08	0x00110133	add x2, x2, x1	add x2, x2, x1
0x0c	0xfff08093	addi x1, x1, -1	addi x1, x1, -1
0x10	0xfe101ce3	bne x0, x1, -8	bne x0, x1, loop
0x14	0xfedff06f	jal x0, -20	j start

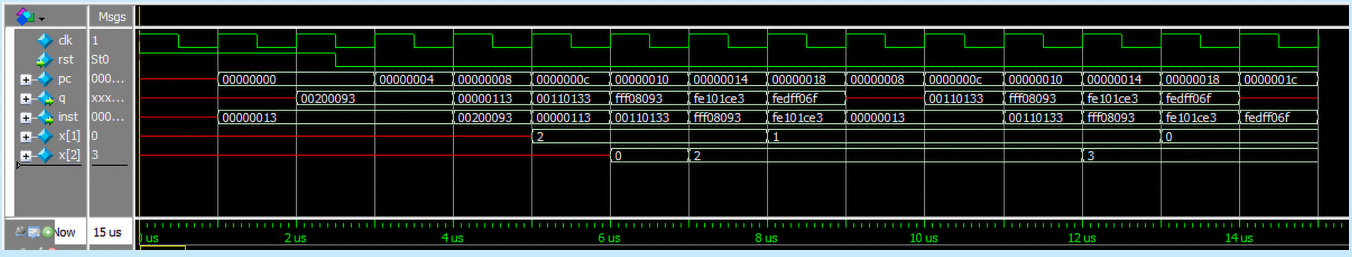
Jego działanie poznaliśmy już w poprzednim kroku. Warto zwrócić uwagę, że ostatni obieg pętli, w którym skok nie zostaje wykonany trwa o dwa takty zegara krócej niż wcześniejsze. Jeżeli chcielibyśmy użyć pętli do wygenerowania opóźnienia powinniśmy uwzględnić ten fakt. Tak jak poprzednio zachęcam do zasymulowania działania tego programu w ModelSimie:

```
do scripts/simulate_b.do
```

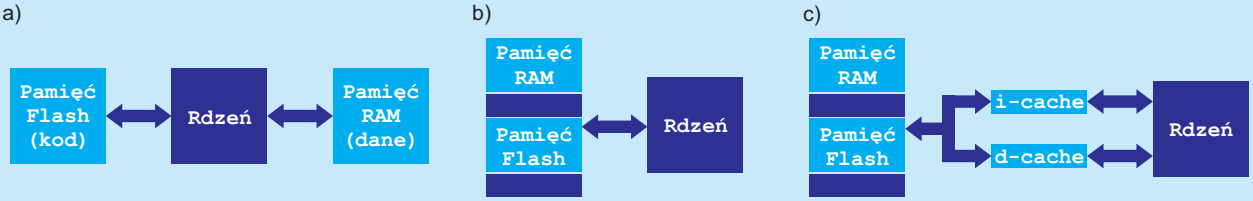
Fragment uzyskanych przebiegów pokazuje **rysunek 35**. Warto go porównać z rysunkiem 34. Ponieważ rdzeń rozpoczyna pracę po wyzerowaniu sygnału reset zerowy cykl zegara ma miejsce w 2 µs symulacji.

Zapis do pamięci

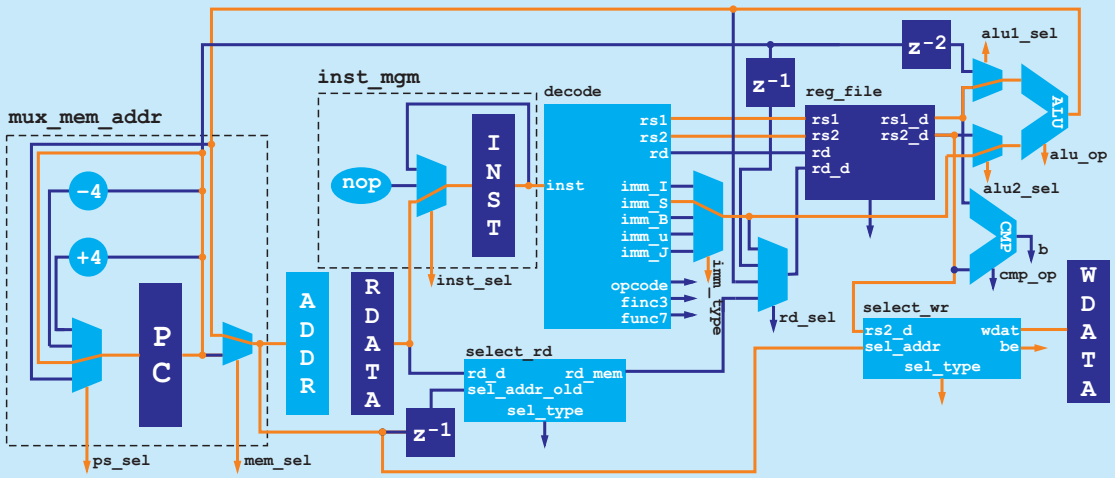
Żeby zaprojektować obsługę zapisu i odczytu z pamięci musimy wrócić do architektury naszego rdzenia. Dla przypomnienia różne jej rodzaje są pokazane na **rysunku 36**. Część a przedstawia architekturę Harwardzką występującą na przykład w popularnych mikrokontrolerach AVR. Pamięci danych jest w niej niezależna od pamięci programu. Dzięki temu odczyt i zapis do RAM nie wpływają na wczytywanie kolejnych instrukcji. Ma ona też swoje wady. Odczytywanie



Rysunek 35. Symulacja wykonania kodu z tabeli 6



Rysunek 36. a) Architektura Harwardzka z rozdzieloną przestrzenią adresową programu i danych (AVR). b) Architektura von Neumana (ARM Cortex-M0). c) Architektura Harwardzka z cachem (ARM Cortex-M7)



Rysunek 37. Przepływ danych podczas zapisu do pamięci

stałych z pamięci Flash jest dużo trudniejsze, a uruchomienie programu z pamięci RAM nie jest możliwe.

Na rysunku 36b pokazana jest architektura von Neumana. Tutaj zarówno program jak i dane znajdują się we wspólnej przestrzeni adresowej, pomimo że mogą być one wykonane w różnej technologii. Dostęp do nich odbywa się przez tą samą szynę. Przykładem mikrokontrolera z taką organizacją jest ARM Cortex-M0.

Dla szybszych rdzeni stosuje się jeszcze inne podejście. Program i dane nadal znajdują się we wspólnej przestrzeni adresowej, ale dostęp do nich odbywa się poprzez *cache* (pamięć notatnikowa). Tę sytuację pokazuje rysunek 36c. Jest to stosunkowa mała, ale szybka pamięć, która pośredniczy w dostępie do przestrzeni adresowej. Rozwiązuje konflikt pomiędzy wykorzystaniem magistrali dla dostępu do danych i do pobierania programu. Przykładem mikrokontrolera, w którym zastosowano taką architekturę jest ARM Cortex-M7. Występuje także w większości procesorów (Intel, ARM Cortex-A). Cache same w sobie są ciekawym (i dość złożonym) zagadnieniem. Zainteresowanym czytelnikom polecam wykłady [5], [14] oraz książkę [12].

W budowanym przez nas rdzeniu zastosowana jest architektura von Neumana (rysunek 36b).

Rysunek 37 pokazuje przepływ danych przy zapisie do pamięci. Adres pamięci jest ustawiony na wartość z wyjścia ALU, czyli sumę stałej bezpośredniej oraz wartości z rejestru. Wartość rejestru PC nie ulega zmianie, ponieważ na następnym zboczu zegara nie nastąpi odczyt kolejnej instrukcji.

Do pamięci zostanie zapisana wartość z drugiego odczytanego rejestru. Przed zapisem musi jednak zostać odpowiednio przygotowana, za co odpowiada blok *select_wr*. Ścieżka *sel_type* konfiguruje, czy zapis ma dotyczyć pojedynczego bajtu, połowy, czy całego słowa. Tu pojawia się problem. Pamięć obsługuje jedynie dostęp do pełnych słów, spod adresów wyrównanych do czterech bajtów. Na szczęście udostępnia jednak dodatkowy czterobitowy sygnał *be* (*byte enable* – bajt włączony). Zapisane zostaną tylko te bajty słowa, dla których odpowiedni bit jest ustawiony.

Pierwszym zadaniem bloku *select_wr* jest odpowiednie przesunięcie danych do zapisania, aby odpowiadały docelowej pozycji w pamięci. Odpowiedzialny za to fragment kodu pokazuje **listing 12**. Przy zapisie pojedynczego bajtu (SB) jak łatwo się domyślić dostępne są cztery możliwości. Przy zapisie połowy słowa (SH) sprawa jest trudniejsza. Teoretycznie także mamy cztery możliwości, ale tylko dwie z nich w całości mieszczą się w pojedynczym słowie w pamięci. Dzieje się tak gdy adres jest podzielny przez liczbę bajtów. O takim adresie mówimy, że jest wyrównany (*aligned*). Zapis (a także odczyt) pod niewyrównany adres wymaga rozbicia go na dwie operacje w pamięci. RISC-V ISA zostawia twórcą poszczególnych implementacji

Listing 12. Fragment modułu *select_wr* (*core/select_wr.sv*) przesuwający dane, aby pasowały do odpowiedniego słowa pamięci

```
20 always_comb
21  case (sel_type)
22      selectPkg::SB:
23          case (sel_addr)
24          2'b00: wdata = {24'b0, rs2_d[7:0]};
25          2'b01: wdata = {16'b0, rs2_d[7:0], 8'b0};
26          2'b10: wdata = {8'b0, rs2_d[7:0], 16'b0};
27          2'b11: wdata = {rs2_d[7:0], 24'b0};
28          default: wdata = {24'b0, rs2_d[7:0]};
29          endcase
30      selectPkg::SH:
31          case (sel_addr)
32          2'b00: wdata = {16'b0, rs2_d[15:0]};
33          2'b10: wdata = {rs2_d[15:0], 16'b0};
34          default: wdata = {16'b0, rs2_d[15:0]};
35          endcase
36      selectPkg::SW: wdata = rs2_d;
37          default: wdata = rs2_d;
38  endcase
```

Listing 13. Fragment modułu *select_wr* (*core/select_wr.sv*) przygotowujący sygnał *be*

```
40 always_comb
41  case(sel_type)
42      selectPkg::SB: be = 4'b0001 << sel_addr;
43      selectPkg::SH: be = 4'b0011 << sel_addr;
44      selectPkg::SW: be = 4'b1111;
45      default: be = 4'b1111;
46  endcase
47 endmodule
```

wybór, czy taki dostęp do pamięci będzie obsługiwany sprzętowo, programowo, bądź spowoduje zgłoszenie wyjątku. Ponieważ w naszej minimalistycznej implementacji nie są one wywoływane przyjmiemy, że wyniki niewyrównanych operacji na pamięci jest niezdefiniowany. Ostatni przypadek to zapis całego słowa (SW). W tym przypadku wyrównany adresu musi być podzielny przez cztery, więc mamy tylko jedną możliwość: po prostu przesyłamy niezmienioną wartość z rejestru do pamięci.

Drugim zadaniem *select_wr* jest przygotowanie sygnału *be*. Odpowiadającą za to logikę znajdziemy na **listingu 13**. Dla pojedynczego bajtu ustawiona jest jedna 1 na polu odpowiadającemu dwóm najmłodszym bitom adresu. Dla połowy słowa ustawione są dwa młodsze labo dwa starsze bity. Przy zapisie całego słowa ustawione są wszystkie cztery bity sygnału ***be***.

Tabela 7. Program demonstrujący zapis danych do pamięci (code/store_sim.S)

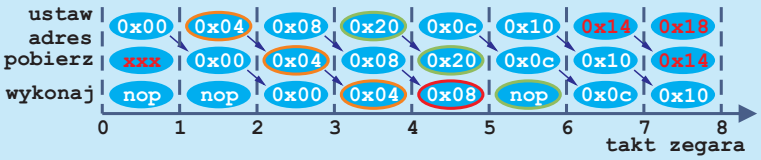
Adres	Binarny	Po assemblacji	Kod
0x00	0x00a00093	addi x1 x0 10	li x1, 10
0x04	0x02102023	sw x1 32(x0)	sw x1, 0x20, x0
0x08	0x00100093	addi x1 x0 1	li x1, 1
0x0c	0x00200093	addi x1 x0 2	li x1, 2
0x10	0x00300093	addi x1 x0 3	li x1, 3

Wiemy już w jaki sposób przygotowane są dane do zapisu. Musimy obsłużyć jeszcze jeden problem. Ponieważ pamięć danych i programu mają wspólną magistrale, wartość spod zapisywanego adresu zostanie wciągnięta do potoku instrukcji. Musimy więc wykryć w którym miejscu się pojawi i zastąpić ją bezpiecznym rozkazem *nop*. Aby poczuć problem przeanalizujemy wykonanie kodu z **tabeli 7**. Pierwsza instrukcja ładuje do rejestru liczbę dziesięć, która zostanie załadowana pod adres 0x20. Pozostałe trzy wpisują do rejestru x1 kolejno liczby 1, 2 i 3. Są one dodane po to, aby łatwiej było zaobserwować, w który momencie w potoku znajduje się „nieproszona" wartość. Na **rysunku 38** pokazano przepływ instrukcji. Kolorem pomarańczowym zaznaczona jest instrukcja *sw*. Gdy wchodzi do fazy wykonania adres jest ustawiony na 0x20. Przepływ tej instrukcji przez potok zaznaczono kolorem zielonym. Warto zwrócić uwagę na instrukcję 0x08. W momencie gdy dokonywany jest zapis znajduje się już ona w fazie pobierz. Dzięki temu może być ona wykonana od razu po zapisie, co zaznaczono kolorem czerwonym. Jest tu jednak mała pułapka. Gdyby zapis odbywał się pod adres 0x08 wykonana byłaby instrukcja znajdująca się pod tym adresem przed dokonaniem zapisu, mimo że w kodzie znajduje się ona za nim! W ten sposób zyskujemy skrócenie trwania zapisu z 3 do 2 cykli zegara kosztem dodania bardzo nieoczywistego zachowania w pewnym szczególnym przypadku. Ponieważ podobne sztuczki są także implementowane w większości procesorów dodano specjalną rodzinę instrukcji zwanych barierami, które gwarantują kolejności zapisu i odczytu danych. Dla naszego prostego rdzenia taką barierą będzie po prostu dodanie instrukcji *nop* po zapisie, który mógłby sprawić problem.

Abu uruchomić zaprezentowany program w symulatorze ModelSim wywołujemy polecenie:

```
do scripts/simulate_store.do
```

Po jego wykonaniu zobaczymy efekt podobny do tego z **rysunku 39**. W ostatnim wierszu wyświetlona jest zawartość ósmego słowa



Rysunek 38. Przepływ instrukcji podczas wykonywania programu z tabeli 7

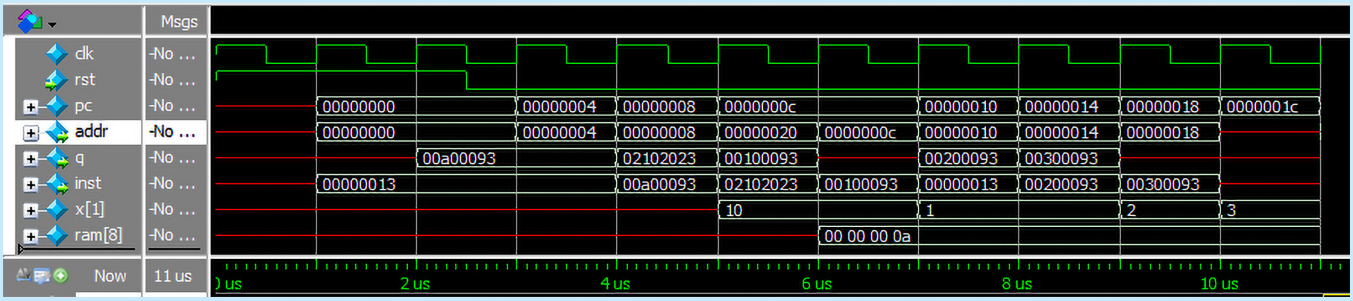
Tabela 8. Program demonstrujący odczyt danych z pamięci (code/load_sim.S)

Adres	Binarny	Po assemblacji	Kod
0x00	0x00a00093	addi x1 x0 10	li x1, 10
0x04	0x02102023	lw x1 20(x0)	lw x1, 0x14, x0
0x08	0x00100093	addi x1 x0 1	li x1, 1
0x0c	0x00200093	addi x1 x0 2	li x1, 2
0x10	0x00300093	addi x1 x0 3	li x1, 3
0x14	0x000000ff		

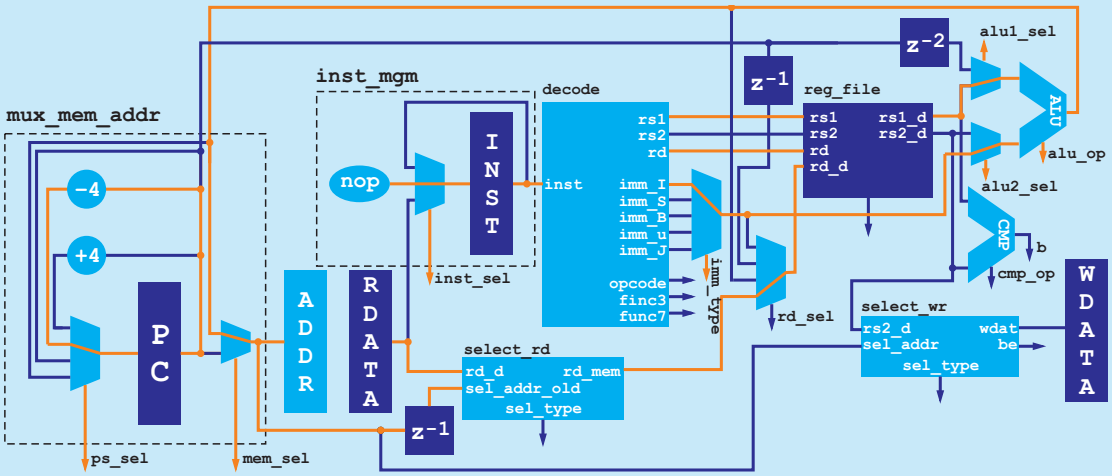
pamięci RAM, które zawiera bajty od 32 do 36. Widzimy że w 6 µs następuje zapis do niego zawartości rejestru x1. Następnie w kolejnym takcie do x1 wpisywana jest jedynka. Na zboczu zegara w 8 µs wykonana jest wstrzyknięta instrukcja *nop*, która zastępuje dane wczytane w czasie zapisu. Ponieważ wcześniej to słowo pamięci nie było zapisane oznaczone jest czerwoną kreską, czyli jako niezdefiniowane. Wpisanie do x1 liczby 2 ma miejsce dopiero podczas kolejnego taktu (9 µs).

Odczyt z pamięci

Odczyt danych z pamięci jest operacją tylko trochę bardziej skomplikowaną niż zapis. Jak widzimy na **rysunku 40** wybór adresu przebiega analogicznie. Tym razem jednak z samym zapisem do rejestru musimy zaczekać, aż dane pojawią się w rejestrze RDATA. Ma to miejsce dopiero na kolejnym zboczy zegara po tym jak instrukcja



Rysunek 39. Symulacja wykonania kodu z tabeli 7



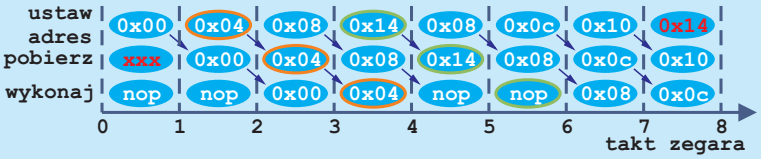
Rysunek 40. Przepływ danych podczas odczytu z pamięci

znajdzie się w fazie wykonaj. Wtedy dopiero zostanie ustawiona ścieżka *wr*, a sam zapis do rejestru nastąpi dopiero na trzecim takcie zegara. Blok *select_rd* pełni odwrotną funkcję do *select_wr*. Na podstawie dwóch najmłodszych bitów adresu oraz rozmiaru wczytywanych danych wycina z odczytanego słowa pamięci potrzebne dane. Następnie przesuwane są one na początek rejestru. Aby adres „płynął" razem z danymi przed podaniem na blok *select_rd* jest on opóźniony o jeden takt zegara.

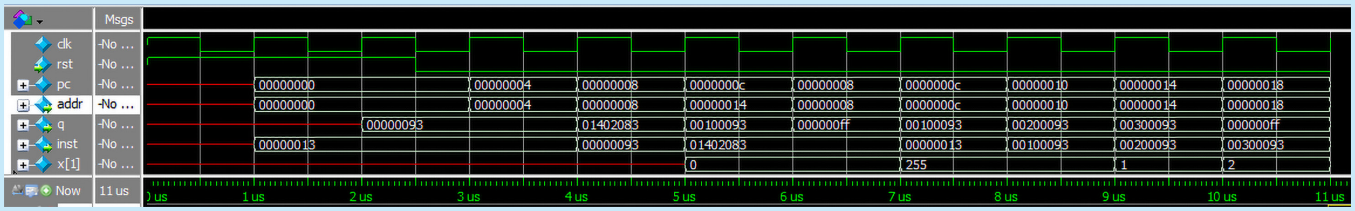
W przypadku odczytu całego słowa jest ono po prostu wpisywane do rejestru. Natomiast dla połowy słowa oraz bajtu dochodzi dodatkowa możliwość. Rozkazy *sh* i *sb* traktują pobieraną wartość jako liczbę ze znakiem, więc na niewykorzystane bity rejestru dopełniają najstarszym bitem zmiennej. Jednak jeżeli chcemy reprezentować tylko liczby dodatnie możemy skorzystać z instrukcji *shu* i *sbu*, które zawsze dopełnią je zerami. Różnicę

Listing 14. Różnica pomiędzy ładowaniem liczb z i bez znaku (*core/select_rd.sv*).

```
29 selectPkg::SH:
30  case (sel_addr_old)
31      2'b00: rd_mem = {rdata[15] ? 16'hffff : 16'h0, rdata[15:0]};
32      2'b10: rd_mem = {rdata[31] ? 16'hffff : 16'h0, rdata[31:16]};
33      default: rd_mem = 'b0;
34  endcase
44 selectPkg::SHU:
45  case (sel_addr_old)
46      2'b00: rd_mem = {16'h0, rdata[15:0]};
47      2'b10: rd_mem = {16'h0, rdata[31:16]};
48      default: rd_mem = 'b0;
49  endcase
```



Rysunek 41. Przepływ instrukcji podczas wykonywania programu z tabeli 8



Rysunek 42. Symulacja wykonania kodu z tabeli 8

pomiędzy implementacją obu tych przypadków obrazuje fragment kodu z **listingu 14**.

Program demonstracyjny z **tabeli 8** jest bardzo podobny do poprzedniego. Różni się zamianą instrukcji *sw* na *lw* oraz wpisaniem liczby 255 do słowa pamięci pod adresem 0x14. **Rysunek 41** pokazuje potok instrukcji podczas wykonywania przygotowanego programu. Na pomarańczowo oznaczono instrukcję *lw*. W momencie jej wykonania ustawiany jest adres danych do wczytania, co zostało oznaczone kolorem zielonym. Aby kolejne instrukcje były wykonywane dopiero po wczytaniu danych do rejestru wartość PC jest przywracana do poprzedniej pozycji poprzez odjecie 4. Taka realizacja odczytu z pamięci jest prostsza ale zajmuje 3 cykle. Alternatywnym rozwiązaniem mógłby być zapis instrukcji spod adresu 0x08 w dodatkowym rejestrze. Dzięki temu mogłaby być ona załadowana w miejsce drugiej instrukcji *nop* (oznaczonej kolorem zielonym). Spowodowało by to skrócenie czasu wykonania o jeden takt zegara. Zachęcam do zabawy z własnymi modyfikacjami. Na **rysunku 42** pokazano wynik symulacji, którą można uruchomić poleceniem:

```
do scripts/simulate_load.do
```

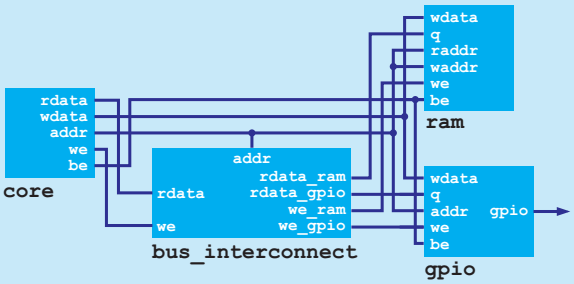
Za wygenerowanie odpowiednich sygnałów sterujących odpowiada moduł *ctrl* zaimplementowany w pliku *core/ctrl.sv*.

Peryferia

Mamy już gotowy rdzeń. Żeby mógł pracować potrzebujemy jeszcze peryferiów. Najprostszym sposobem na ich dodanie są tak zwane MMIO (*Memory-Mapped Input/Output* – wejścia/wyjścia zmapowane do pamięci). Technika ta polega na przypisaniu rejestrom poszczególnych podzespołów adresów. Dzięki czemu dostęp do nich nie wymaga żadnych dodatkowych instrukcji. Po prostu korzystamy z rozkazów z rodzin *store* i *load*.

Zajmiemy się tylko najprostszym możliwym peryferium, czyli wyjściem cyfrowym. Docelowo posłuży ono do sterowania diodami świecącymi.

Rysunek 43 pokazuje schemat naszego miniaturowego układu mikroprocesorowego. Implementacja jego poszczególnych elementów znajduje się w folderze *peripheral*. Kod bloku pamięci RAM

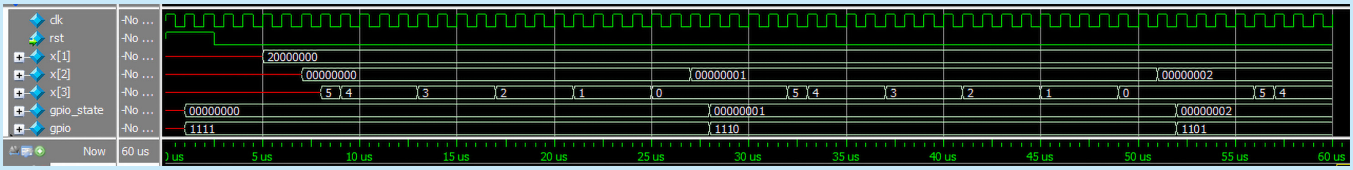


Rysunek 43. Schemat układu mikroprocesorowego

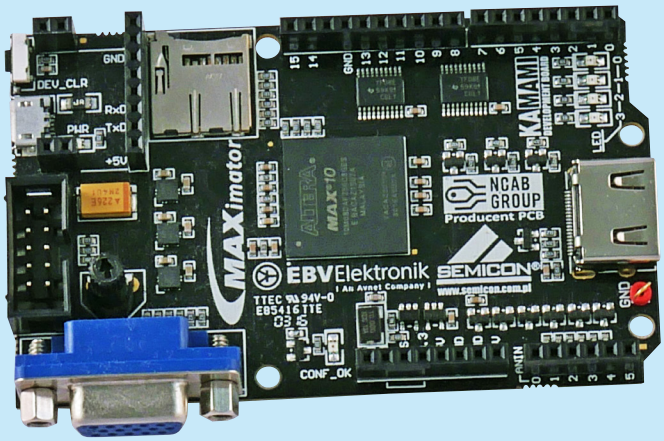
pochodzi z szablonu dostarczonego przez firmę Intel, który gwarantuje że w czasie syntezy nie zostanie on złożony z podstawowych elementów logicznych. Zamiast tego wyinferowana zostanie blokowa pamięć RAM. W projekcie przyjęto, że ma ona rozmiar 1 kB. W przestrzeni adresowej zajmuje pozycję od 0 do 0x400. Moduł wyjść cyfrowych składa się z jednego rejestru. Jego 4 najmłodsze bity są wyprowadzona na ścieżkę *gpio*. Zostaną one podłączone do czterech diod świecących znajdujących się na płytce prototypowej. Ponieważ do wyjść układu FPGA podłączone są ich katody, moduł gpio wystawia zanegowane wartości. Dzięki temu logiczne 1 odpowiada zaświeconej, a 0 zgaszonej diodzie. W przestrzeni adresowej odpowiada im adres 0x2000_0000. Moduł *bus_interconnect* pełni rolę multipleksera. Na podstawie najstarszych bitów adresu ustala czy aktywny ma być RAM, czy *gpio*. Odpowiada za to, aby z jednej strony rdzeń otrzymał odpowiednie dane wejściowe, a z drugiej aby sygnał we trafił do odpowiedniego bloku.

Listing 15. Program sterujący diodami LED (*code/blink_slow.S*).

```
01 li       x1, 0x20000000
02 li       x2, 0
03 start:
04 li       x3, 1000000
05 loop:
06 addi x3, x3, -1
07 bnez x3, loop
08 addi x2, x2, 1
09 sw       x2, 0, x1
10 j        start
```



Rysunek 44. Symulacja działania programu migającego diodą



Fotografia 45. Zestaw startowy maXimator

Na **listingu 15** pokazany jest program, który zaświeca diody LED w sekwencji odpowiadającej kolejnym liczbom binarnym. W pierwszej linii do rejestru x1 wpisywany jest wybrany wcześniej adres portów wyjściowych. Rejestr x2 służy do przygotowania kolejnych stanów wyjść. Głowna pętla programu zaczyna się od etykiety *start*. W linach od 4 do 7 znajduje się pętla opóźniająca. Najpierw rejestr x3 jest inicjalizowany, a następnie jest on dekrementowany dopóki nie zostanie wyzerowany. Za pętlą inkrementowany jest rejestr x2. Wpisanie jego zawartości na wyjście cyfrowe następuje w linii 9. Ostatnią instrukcją jest skok do początku pętli głównej. Aby miganie było widoczne dla człowieka wartość wpisywana do x3 musi być stosunkowo duża. Na potrzeby symulacji przygotowano program *code/blink.S*. Jedyną zmianą jest zmniejszenie liczby iteracji do 5. Można ją uruchomić poleceniem:

```
do scripts/simulate_blink.do
```

Uzyskany wynik prezentuje **rysunek 44**. Tym razem nie są na nim pokazane kolejne instrukcje, ponieważ przy tym powiększeniu byłby one nieczytelne. Poza zegarem i resetem pokazany jest stan trzech wykorzystanych rejestrów. Gpio_state obrazuje wartość wpisaną do rejestru wyjść, a gpio przedstawia w systemie binarnym stan podawany na wyjścia układu FPGA.

Podsumowanie

Mam nadzieję, że udało mi się zainteresować czytelników tematem tworzenia własnych mikrokontrolerów. Ostatnim etapem kursu będzie uruchomienie rdzenia „w krzemie”. Naszą docelową platformą będzie płytka MAXimator (**fotografia 45**) wyposażona w układ MAX10 firmy Intel (dawniej Altera) [13].

Rafał Kozik
rafkozik@gmail.com

Bibliografia:

[1] ***http://bit.ly/2MFFapx***, dostęp na dzień 29.04.2019
[2] ***http://bit.ly/2YP0EGY***, dostęp na dzień 29.04.2019
[3] Zeloof S., First IC, ***http://bit.ly/33dwwEx***, dostęp na dzień 29.04.2019
[4] ***http://bit.ly/2MHjQQL***
[5] Computer Structures, MIT 6.004, Spring 2019, ***http://bit.ly/33dx4dL***
[6] MIT 6.004 RISC-V ISA Reference Card, ***http://bit.ly/2TcOS43***
[7] The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version2.2, Editors Andrew Waerman and Krste Asanovic, RISC-V Foundation, May 2017
[8] ***http://bit.ly/31jY2OV***
[9] ***http://bit.ly/2yDgVjP***
[10] Quartus Prime Lite Edition, ***https://intel.ly/2MC6TYp***
[11] Ataki na procesory – PortSmash, TLBleed, Foreshadow, ***http://bit.ly/2M307eL***
[12] Patterson D., Hennessy J., Computer Organization and Design: The Hardware Software Interface (RISC-V Edition), Morgan Kaufmann, 1st ed, 2017
[13] MAXimator Altera MAX10 FPGA Evaluation Board, ***http://bit.ly/2YJneNp***
[14] Great Ideas in Computer Architecture (Machine Structures), CS61C Berkeley with Nicholas Weaver, Spring 2019
[15] Moduł dla Visual Studio Code ***http://bit.ly/2YP0Jug***

Miesięcznik **„Elektronika Praktyczna”**
(12 numerów w roku) jest wydawany
przez AVT-Korporacja Sp. z o.o. we współpracy
z wieloma redakcjami zagranicznymi.

Wydawca:
AVT-Korporacja Sp. z o.o.
03-197 Warszawa, ul. Leszczynowa 11
tel.: 22 257 84 99, faks: 22 257 84 00

Adres redakcji:
03-197 Warszawa, ul. Leszczynowa 11
tel.: 22 257 84 60
faks: 22 257 84 00
e-mail: redakcja@ep.com.pl
www.ep.com.pl

Redaktor Naczelny:
Wiesław Marciniak

Redaktor Programowy,
Przewodniczący Rady Programowej:
Piotr Zbysiński

Zastępca Redaktora Naczelnego,
Redaktor Prowadzący:
Damian Sosnowski

Zastępca Redaktora Naczelnego,
Menedżer Magazynu
Marcin Karbowniczek

Szef Pracowni Konstrukcyjnej:
Grzegorz Becker, tel.: 22 257 84 58

Redaktor strony internetowej *www.ep.com.pl*
Dariusz Welik

Zespół marketingu i reklamy:
Katarzyna Gugała, tel.: 22 257 84 64
Adam Kęska, tel.: 22 257 84 63
Bożena Krzykawska, tel.: 22 257 84 42
Grzegorz Krzykawski, tel.: 22 257 84 60
Maksymilian Hoser, tel.: 22 257 84 65

Sekretarz Redakcji:
Grzegorz Krzykawski, tel.: 22 257 84 60

DTP i okładka:
MAD Sp. z o.o.

Stali Współpracownicy:
Jacek Bogusz, Lucjan Bryndza, Jarosław Doliński, Andrzej Gawryluk, Krzysztof Górski, Tomasz Jabłoński, Michał Kurzela, Szymon Panecki, Sławomir Skrzyński, Ryszard Szymaniak, Adam Tatuś, Robert Wołgajew

Uwaga!
Kontakt z wymienionymi osobami jest możliwy via e-mail, według schematu: imię.nazwisko@ep.com.pl

Prenumerata w Wydawnictwie AVT
www.avt.pl/prenumerata
lub tel.: 22 257 84 22
e-mail: prenumerata@avt.pl
www.sklep.avt.pl, tel.: 22 257 84 66

Prenumerata w RUCH S.A.
www.prenumerata.ruch.com.pl
lub tel.: 801 800 803, 22 717 59 59
e-mail: prenumerata@ruch.com.pl

Wydawnictwo
AVT-Korporacja Sp. z o.o.
należy do **Izby Wydawców Prasy**