

PUCY MP_02

Kacper Szkudlarek, Maciej Stefańczyk

25 maja 2010

Spis treści

1	Realizacja ogólna	2
1.1	Treść zadania	2
1.2	Założenia projektowe	2
1.3	Schemat blokowy układu	4
2	Opis komponentów	5
2.1	Moduł CPU	5
2.1.1	Wczytywanie instrukcji, operandów i rejestrów	5
2.1.2	Opis stanów procesora	6
2.2	Pamięci wewnętrzne	6
2.2.1	Pamięć RAM	6
2.2.2	Pamięć ROM	7
2.3	Moduł operacji złożonej – mnożenie Booth’a	8
2.4	Moduł wejścia – klawiatura PS/2	9
2.5	Moduł wyjścia – drukarka LPT	9
2.5.1	Moduł zamiany liczb U2 na BCD	10
2.6	Moduł testowy – wyprowadzanie danych w postaci binarnej	11
3	Assembler	12
3.1	Opis	12
3.2	Instrukcja użycia	12
3.2.1	Definicja języka	12
3.2.2	Pliki źródłowe	13
3.2.3	Kompilacja	14
3.3	Przykłady	14
4	Kody źródłowe	17
4.1	Mikrokontroler	17
4.2	Assembler	38

Rozdział 1

Realizacja ogólna

1.1 Treść zadania

Zadanie polegało na zaprojektowaniu 8-bitowego mikrokontrolera ogólnego przeznaczenia, wyposażonego w kilka wbudowanych podzespołów (pamięć ROM, pamięć RAM, układ wejściowy wczytujący dane z klawiatury, układ wyjściowy drukujący dane na drukarce) oraz umożliwiającego podłączanie kolejnych podzespołów poprzez wyprowadzoną na zewnątrz szynę systemową. Bloki funkcjonalne mikrokontrolera powinny być zaprojektowane jako niezależne elementy współpracujące ze sobą za pośrednictwem wewnętrznej, trójstanowej szyny systemowej.

1.2 Założenia projektowe

Dane wejściowe i wyjściowe wprowadzane i wypisywane są w postaci 4 znaków ASCII w postaci ZDDD, wyznaczających wartość dziesiętną ze znakiem z przedziału $[-128, +127]$. Dane na wejściu podawane są bez błędów, jednak mimo tego zabezpieczyliśmy się przed podaniem liczb spoza zakresu.

Zestaw instrukcji

Procesor powinien obsługiwać następujący zestaw poleceń: (patrz też: 3.3)

- Instrukcje sterujące

STOP

Zatrzymanie wykonywania programu, wyjście z tego stanu możliwe jedynie przez reset układu.

RESET

Powrót programu do pierwszej instrukcji. $PC \leq 0$

JMP A

Skok bezwarunkowy pod adres A (8 bitów). $PC \leq A$

JZ Rd A

Skok warunkowy pod adres A. $Rd == 0 \Rightarrow PC \leq A$

- Operacje arytmetyczne i logiczne

ADD Rd, Ra, Rb

Dodawanie. $Rd \leftarrow Ra + Rb$

SUB Rd, Ra, Rb

Odejmowanie. $Rd \leftarrow Ra - Rb$

MUL Rd, Ra, Rb

Mnożenie (wykonywane metodą Bootha). $Rd \leftarrow Ra * Rb$

AND Rd, Ra, Rb

Iloczyn bitowy. $Rd \leftarrow Ra \text{ and } Rb$

OR Rd, Ra, Rb

Suma bitowa. $Rd \leftarrow Ra \text{ or } Rb$

- Operacje na pamięci oraz wejścia/wyjścia

LDI Rd, D

Łaadowanie wartości do rejestru Rd. $Rd \leftarrow D$

LD Rd, Ra, Rb

Wczytanie danych z pamięci spod adresu wskazywanego przez rejestry Ra i Rb. $Rd \leftarrow (Ra, Rb)$

STORE Rd, A

Zapisanie zawartości rejestru Rd do pamięci pod adres A (16-bitowy). $Rd \rightarrow (A)$

IN Rd, A

Wczytanie do rejestru Rd danych z urządzenia wejściowego o adresie A (8-bitowy). $Rd \leftarrow IN(A)$

OUT Rd, A

Wyprowadzenie zawartości rejestru Rd do urządzenia wyjściowego o adresie A (8-bitowy). $Rd \rightarrow OUT(A)$

Wszystkie operacje wykonywane są na liczbach 8-bitowych zapisanych w kodzie U2. Mikrokontroler posiada 8 rejestrów ogólnego przeznaczenia (R0..R7).

Szyna systemowa

Szyna systemowa składa się z 3 grup sygnałów: linii adresowych (16 bitów), linii danych (8 bitów) oraz sygnałów sterujących, które dokładniej opisane są niżej.

/RD Sygnał żądania odczytu danych

/WR Sygnał żądania zapisu danych

/MREQ Sygnał żądania dostępu do przestrzeni adresowej pamięci (16 bitów)

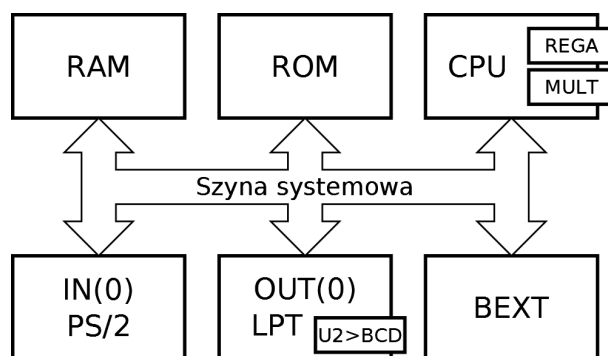
/IOREQ Sygnał żądania dostępu do przestrzeni adresowej wejścia/wyjścia (8 bitów)

WT Sygnał oczekiwania sygnalizujący trwającą operację, procesor powinien zaczekać

Dodatkowo do każdego komponentu doprowadzone są sygnały GEN (zegar systemowy 40MHz) oraz CR (asynchroniczne zerowanie układu).

1.3 Schemat blokowy układu

Na rysunku 1.1 przedstawiony został schemat blokowy zaprojektowanego układu. Każdy bloczek oznacza w ogólności pojedynczy moduł systemu (zaprojektowany osobno i niezależnie od pozostałych). Bloczki zawarte wewnątrz innych są ich częścią składową i poza nimi nie ma do nich dostępu (np. blok mnożenia i rejestry CPU, zamiana postaci U2 na format BCD w układzie wyjściowym).



Rysunek 1.1: Ogólny schemat blokowy projektowanego układu.

Rozdział 2

Opis komponentów

2.1 Moduł CPU

2.1.1 Wczytywanie instrukcji, operandów i rejestrów

Główny moduł mikrokontrolera został zaimplementowany w postaci jednej maszyny stanów, której poszczególne stany są dokładniej omówione w dalszej części opisu. Przy projektowaniu została podjęta decyzja, że zamiast warunkowego pobierania drugiego i ewentualnie trzeciego bajtu z pamięci w zależności od instrukcji, a także od warunkowego pobierania wartości z rejestrów, zawsze będą pobierane 3 słowa z pamięci i odczytywane trzy rejestry. Działa to w sposób następujący:

- Z pamięci odczytywane jest pierwsze słowo (zawierające kod rozkazu i ewentualnie numer rejestru)
- Z pamięci odczytywane jest drugie słowo, a jednocześnie do zmiennej pomocniczej wczytywana jest zawartość rejestru o numerze zakodowanym w pierwszym słowie
- Z pamięci wczytywane jest trzecie słowo, a z rejestrów wczytywane są te o numerach zakodowanych w drugim słowie

W ten sposób zawsze do trzech zmiennych IC1..IC3 wczytywane są trzy słowa z pamięci programu, a do zmiennych TMP0..TMP2 wczytywana jest zawartość trzech rejestrów. Dzięki temu w późniejszym etapie przy wykonywaniu instrukcji wszystkie potrzebne dane są już przygotowane i wykonanie operacji najczęściej sprowadza się do zapisania wyniku do pamięci i odpowiedniego zwiększenia licznika rozkazów PC. Jeśli instrukcja, którą obsługujemy jest krótsza niż 3 słowa (a większość jest), to po zwiększeniu PC "niewykorzystane" słowa zostaną ponownie wczytane i zinterpretowane przez następną komendę.

Niewątpliwą zaletą takiego rozwiązania jest duże uproszczenie funkcji każdej instrukcji, przy poświęceniu niewiele dłuższego czasu na obsługę całego cyklu (różnica jest praktycznie niezauważalna, a w niektórych przypadkach rozwiązanie zastosowane przez nas jest szybsze od pobierania kolejnych słów instrukcji i wczytywania zawartości rejestrów już w funkcji obsługi danej instrukcji).

2.1.2 Opis stanów procesora

Pojedynczy cykl pracy procesora, a więc przejście od wczytania instrukcji i operandów aż do jej wykonania i zapisania wyników, może wymagać przejścia przez następujące stany:

ST0

Sprawdzenie, czy nie został ustawiony sygnał STOP (poprzez wykonanie instrukcji STOP bądź niedozwoloną sekwencję sterującą), oraz pobranie z pamięci słowa spod adresu wskazywanego przez PC.

ST1

Odczekanie ustalonej liczby taktów na ustalenie się danych odczytanych z pamięci¹

ST2

Zapisanie danych do odpowiednich zmiennych pomocniczych (dane wczytane z pamięci do zmiennych IC1..IC3, wczytanie zawartości odpowiednich rejestrów do zmiennych TMP0..TMP2)

ST3

Interpretacja oraz wykonanie instrukcji wczytanej z pamięci. Wszystkie operandy są już wczytane, a więc w większości przypadków wymagane jest jedynie zapisanie wyniku odpowiedniego działania do pamięci bądź do rejestru. W przypadku operacji wymagających odczekania na sygnał WAIT jest to realizowane już wewnątrz danego bloku instrukcji.

ST_WAIT

Stan oczekiwania przed rozpoczęciem nowego cyklu pracy, wykorzystywany podczas zapisu wyników wykonania poprzedniej instrukcji do pamięci.

Kod modułu został umieszczony na listingu 4.1.

2.2 Pamięci wewnętrzne

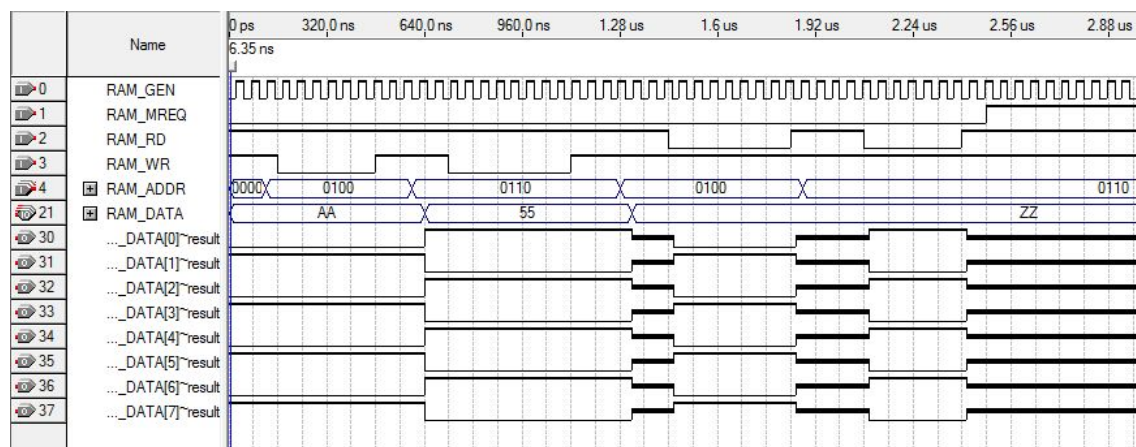
Do realizacji pamięci wewnętrznych zostały wykorzystane moduły zawarte w bibliotece LPM firmy Altera. Biblioteka zawiera konfigurowalne definicje podsawowych, najczęściej używanych bloków funkcjonalnych, które można zrealizować na układach FPGA.

2.2.1 Pamięć RAM

Do realizacji pamięci RAM został użyty moduł `lpm_ram_dq`. Jest to pamięć RAM z rozdzielonym wejściem i wyjściem. Może być używana jako pamięć synchroniczna lub asynchroniczna. W projekcie pamięć została zdefiniowana jako synchroniczna. Synchronizowany jest dostęp do danych, jak i szyny adresowej. Sygnałem synchronizującym operacje zapisu i odczytu danych jest zegar systemowy. Ponieważ szyna danych, do której podłączony jest moduł, jest trójstanowa, niezbędne było także takie skonfigurowanie elementu, by

¹Podczas testów okazało się, że pamięć nie nadąża z ustawieniem danych na liniach, przez co procesor dostawał błędne odczyty

w momencie, gdy nie jest odpowiednio wystawiony do pracy, pozostawiał szynę danych wolną dla innych modułów.



Rysunek 2.1: Przebiegi czasowe na poszczególnych liniach modułu pamięci RAM.

Kod modułu został umieszczony na listingu 4.2.

2.2.2 Pamięć ROM

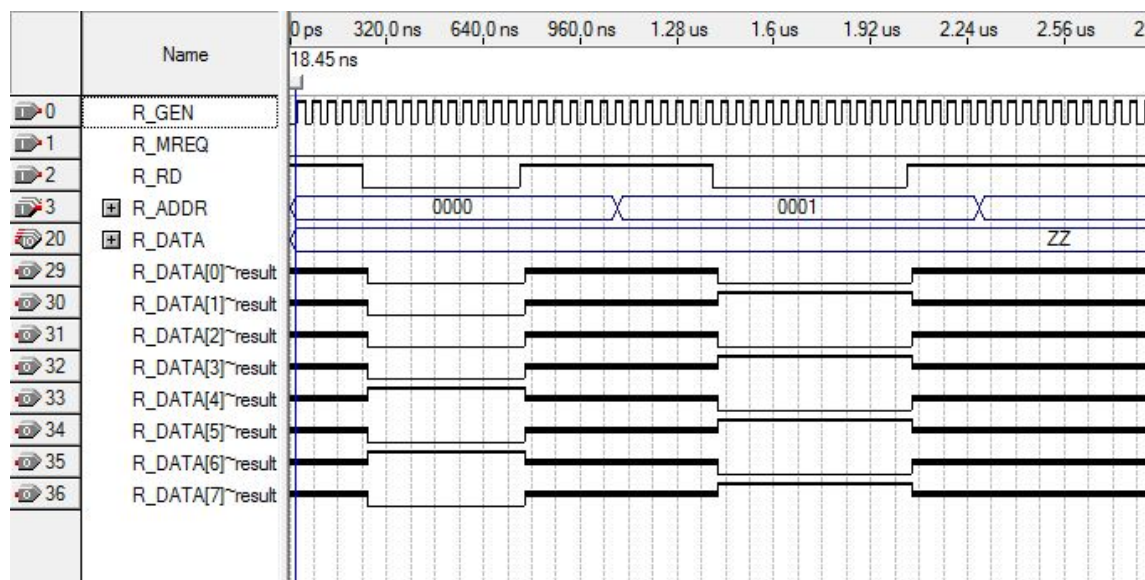
Pamięć ROM również w oparciu o moduł z biblioteki LPM - lpm_rom. Moduł może być używany zarówno jako pamięć synchroniczna i asynchroniczna. W projekcie pamięć została użyta w postaci asynchronicznej. Sterowany jest jedynie dostęp modułu do trójstanowej szyny danych. W momencie, gdy nie następuje odczyt danych, moduł przechodzi w stan wysokiej impedancji, pozostawiając wolny dostęp do szyny danych. W symulacji przedstawionej poniżej 2.2 użyty został następujący plik konfiguracyjny zawartości pamięci:

Listing 2.1: Plik źródłowy dla symulacji 2.2

```

1
2 WIDTH = 8;
3 DEPTH = 32;
4
5 ADDRESS_RADIX = HEX;
6 DATA_RADIX = BIN;
7
8 CONTENT BEGIN
9     0 : 01010000;
10    1 : 10101010; -- R0 <= 0xAA
11    2 : 01010001;
12    3 : 00000001; -- R1 <= 0x01
13    4 : 00100010;
14    5 : 00000001; -- R2 <= R0 + R1 = 0xAB
15    6 : 00100000;
16    7 : 00100001; -- R0 <= R2 + R1 = 0xAC
17    8 : 00000000; -- STOP
18 END;
```


19
20
21 **END ;**



Rysunek 2.2: Przebiegi czasowe na poszczególnych liniach modułu pamięci ROM.

Kod modułu został umieszczony na listingu 4.3.

2.3 Moduł operacji złożonej – mnożenie Bootha

Algorytm Bootha jest wykorzystywany do mnożenia dwóch liczb binarnych zapisanych w postaci U2.

Algorytm polega na sukcesywnym dodawaniu do sumy częściowej P jednej z dwóch predefiniowanych wartości – A bądź S. Oznaczmy przez m i r mnożną i mnożnik naszego działania, a przez n ich ilość bitów.

1. Ustalenie początkowych wartości A, S i P (każde z nich powinno być długości $2n+1$ bitów)
 - (a) Wypełnienie najstarszych n bitów liczby A wartością m, pozostałe bity zerujemy ($A = \{m, 0..0\}$)
 - (b) Wypełnienie najstarszych n bitów liczby S wartością -m (w dopełnieniu do 2), pozostałe bity zerujemy ($S = \{-m, 0..0\}$)
 - (c) Wypełniamy najstarsze bity P zerami, następnie wstawiamy liczbę r, a najmłodszy bit ustawiamy na 0 ($P = \{0..0, r, 0\}$)
2. Jeśli dwa najmłodsze bity sumy częściowej P wynoszą:
 - (a) 01: $P = P + A$

(b) 10: $P = P + S$

(c) 00 lub 11: P pozostaje bez zmian

3. Przesunięcie arytmetyczne P o jeden bit w prawo

4. Powtórzenie kroków od 2 i 3 w sumie n razy

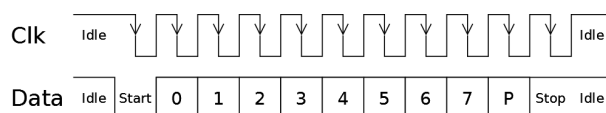
5. Wynikiem mnożenia jest P po usunięciu najmłodszego bitu (przesunięciu bitowym w prawo)

W projekcie mnożenie zostało zrealizowane jako osobny blok funkcjonalny, do którego podawane są mnożna i mnożnik. Blok ten posiada własny sygnał WAIT oznaczający trwające obliczenia, a po ich zakończeniu wynik jest udostępniany przy pomocy odpowiednich sygnałów. Obie liczby na wejściu muszą być tej samej długości (w projekcie przyjęto $n=8$, ale łatwo jest to zmienić), wynik natomiast jest dwukrotnie dłuższy (ze względu na architekturę projektowanego mikrokontrolera odczytywane jest tylko 8 młodszych bitów).

Kod modułu został umieszczony na listingu 4.4.

2.4 Moduł wejścia – klawiatura PS/2

Moduł odpowiada za komunikację układu z urządzeniami zewnętrznymi za pomocą portu PS2. Transmisja danych przychodzących synchronizowana jest poprzez zewnętrzny zegar (ok. 15 kHz) dostarczany przez urządzenie komunikujące się z układem. Wewnętrzna struktura modułu oparta jest o maszynę stanów taktowana wewnętrznym sygnałem (ok. 20 MHz). Maszyna stanów bada kolejność wprowadzanych znaków, jeżeli jest poprawna, wprowadzana liczba konwertowana jest na U2. Konwersja odbywa się poprzez sumowanie odpowiednio przemnożonych kolejnych cyfr dziesiętnych wprowadzanej liczby i odpowiednie ustawienie bitu znaku, jeżeli liczba należy do przedziału $[-128; 127]$

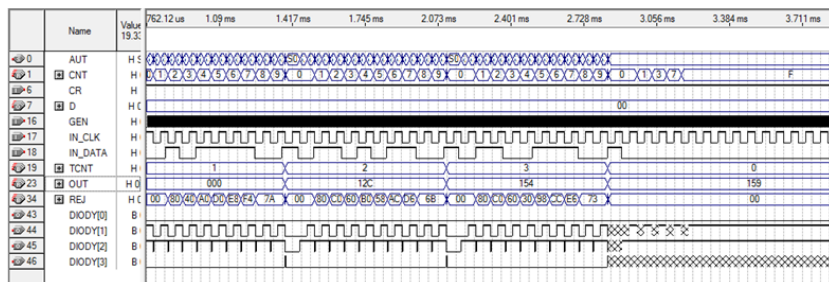


Rysunek 2.3: Przebiegi czasowe na liniach portu PS/2 podczas transmisji pojedynczego znaku z klawiatury.

Kod modułu został umieszczony na listingu 4.5.

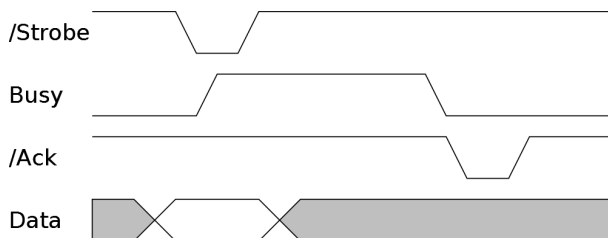
2.5 Moduł wyjścia – drukarka LPT

Moduł odpowiada za komunikację z urządzeniami zewnętrznymi przy użyciu portu LPT. Moduł oparty jest o maszynę stanów taktowaną wewnętrznym zegarem (ok. 20 MHz). Poszczególne stany automatu odpowiadają etapom transmisji danych poprzez łącze równoległe. Moduł korzysta z opisanego poniżej modułu służącego do konwersji liczb z postaci U2 do BCD. Transmisja danych opiera się o wystawianie i utrzymywanie sygnałów sterujących zgodnie wymogami czasowymi transmisji (rys. 2.5), a także badanie stanu linii



Rysunek 2.4: Przebiegi czasowe na liniach modułu PS2.

sterujących wejściowych. Moduł dokonuje także zamiany poszczególnych znaków liczby w postaci BCD na ich odpowiedniki w kodzie ASCII, które są następnie wysyłane do urządzenia zewnętrznego (drukarki zgodnej ze standardem Centronix).



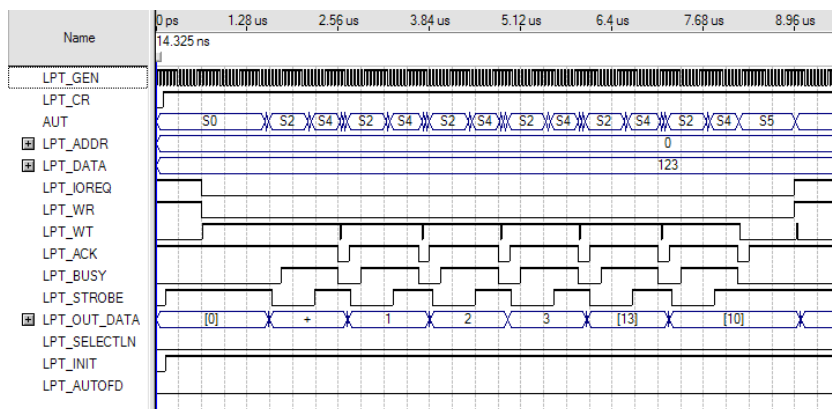
Rysunek 2.5: Przebiegi czasowe na najważniejszych liniach portu LPT podczas wysyłania danych.

Kod modułu został umieszczony na listingu 4.6.

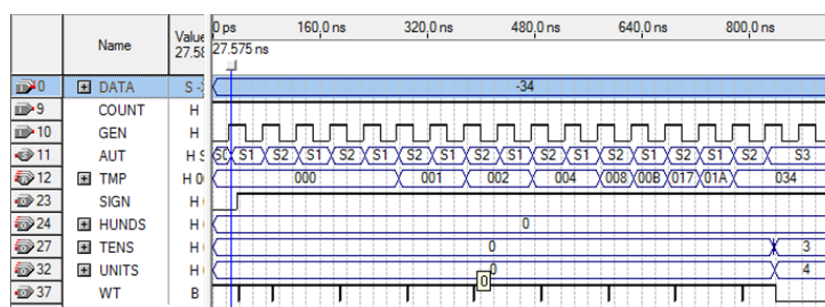
2.5.1 Moduł zamiany liczb U2 na BCD

Pierwszym krokiem konwersji liczb jest sprawdzenie i zapisanie znaku liczby w kodzie U2 i zamiana jej wartości bezwzględnej na liczbę w postaci NKB. Liczba z postaci NKB do postaci BCD konwertowana jest za pomocą algorytmu Shift and Add 3. Ogólna zasada działania algorytmu opiera się na stworzeniu bufora odpowiadającego sumarycznej wielkości znaków wyjściowych (u nas 10 bit, po 4 bit na dziesiątki i jednostki, 2 bity na cyfrę setek) i odpowiednim wsuwaniu z prawej strony liczby binarnej i testowaniu jej wartości. Jeżeli wartość kolumnie odpowiadającej którejś z cyfr liczby dziesiętnej jest większa bądź równa 5 to do tej kolumny dodajemy 3. Po wsunięciu wszystkich bitów konwertowanej liczby algorytm się kończy. W każdej z kolumn otrzymujemy wartość binarnie zapisaną wartość dziesiętną danej cyfry.

Kod modułu został umieszczony na listingu 4.7.



Rysunek 2.6: Wynik symulacji modułu LPT. Na wejście (LPT_DATA) podana została liczba 123. Widać 6 cykli zapisu do drukarki (inicjowane zmianą sygnału LPT_STROBE) oraz oczekiwanie na zakończenie cyklu pracy aż do zdjęcia sygnałów LPT_WR oraz LPT_IOREQ.



Rysunek 2.7: Symulacja działania modułu zamiany liczb na postać BCD.

2.6 Moduł testowy – wyprowadzanie danych w postaci binarnej

W czasie testowania stworzonego mikrokontrolera powstał jeszcze jeden dodatkowy moduł testowy. Pozwala on na wyprowadzanie 8-bitowej danej w postaci binarnej na zewnątrz. Do modułu tego może zostać podłączony dowolny moduł (systemu SML3) potrafiący wyświetlić takie dane, np. wyświetlacz 7-segmentowy (dane zostaną pokazane w formie dwóch cyfr szesnastkowych) bądź zestaw diod LED (wtedy każda dioda będzie odpowiadała stanowi jednego z bitów).

Moduł okazał się potrzebny w sytuacji, kiedy niezbędne było podglądanie i ”zamrażanie” stanu jednej ze zmiennych w układzie, która jedynie przez krótki czas przechowywała wynik obliczeń, a po za nim zawsze się zerowała (a więc proste jej wyprowadzenie na zewnątrz nie dawało zadowalających rezultatów).

Kod modułu został umieszczony na listingu 4.8.

Rozdział 3

Assembler

3.1 Opis

W celu ułatwienia testowania układu stworzony został prosty program mający na celu tłumaczenie kodu programu z postaci źródeł assemblera na pliki .mif wczytywane przez program Quartus. Program, pomimo swojej prostoty, w znaczący sposób skrócił czas tworzenia programów testowych, a także wyeliminował potencjalne błędy mogące powstać podczas ręcznego uzupełniania zawartości ROMu.

Prostota programu nie oznacza jednak, że jest on bardzo ograniczony. Wręcz przeciwnie – został napisany w sposób umożliwiający bardzo łatwe zaadaptowanie go do różnych architektur i zestawów instrukcji. Poprzez modyfikację pliku z definicją języka można zmienić długość kodów operacji, dodać nowe instrukcje, modyfikować ilość rejestrów wewnętrznych procesora, tworzyć nowe typy instrukcji (różne ilości i typy operandów).

3.2 Instrukcja użycia

3.2.1 Definicja języka

Plik z definicją języka podzielony jest na 3 sekcje:

- definicja instrukcji
- definicja rejestrów
- definicja typów instrukcji

Każdy z sektorów musi zaczynać się liczbą oznaczającą ilość jego wpisów, a cały plik nie powinien zawierać pustych linii. Każdy wpis powinien być umieszczony dokładnie w jednej linii.

Definiowanie instrukcji

Ogólna postać definicji pojedynczej instrukcji składa się z 3 części:

- Kod operacji w postaci binarnej – wstawiany jest bezpośrednio w pliku wynikowym. Jego długość nie jest określona, ale dla danego typu instrukcji powinna być stała (każda instrukcja powinna w sumie składać się z pełnych bajtów).
- Mnemonik instrukcji – tekstowy odpowiednik instrukcji używany w kodach źródłowych, podczas parsowania zastępowany odpowiednim kodem.
- Typ instrukcji – jeden z typów określonych w 3 sekcji pliku z definicją języka. Określa ilość i typy parametrów dla danej instrukcji.

Rejestry procesora

W drugiej sekcji zawarta jest prosta definicja rejestrów dostępnych dla danej architektury. Każda linia składa się z nazwy symbolicznej rejestru i następującego po niej binarnego kodu danego rejestru (który wykorzystywany jest podczas tworzenia pliku wynikowego jako część instrukcji).

Definiowanie typów instrukcji

Trzecia sekcja pliku z definicją języka zawiera zdefiniowane typy instrukcji (do nich właśnie odnoszą się typy z sekcji pierwszej). Każdy typ składa się z 2 głównych części: liczbiwego kodu typu oraz definicji składni instrukcji danego typu.

Definicja składni może zawierać jeden bądź wiele elementów składowych, z których każdy oznacza, jak wynikowo powinna zostać złożona instrukcja, a także jakiego typu parametry są dozwolone. Możliwe wartości do wstawienia w tym miejscu to:

OPCODE – kod instrukcji (mnemonik w kodzie źródłowym, binarny kod operacji w pliku wynikowym)

Rd – symboliczna nazwa jednego z rejestrów, zamieniana na jego kod w pliku wynikowym

IM8 – liczba całkowita (w postaci dziesiętnej bądź szesnastkowej poprzedzonej '0x'), w pliku wynikowym zapisywana w postaci 8-bitowej

IM16 – jak IM8, tyle że w pliku wynikowym zamieniana na liczbę 16-bitową

0 – w tym miejscu w pliku wynikowym zostanie wstawione jedno 0, nie wpływa na postać instrukcji w kodzie źródłowym. Używane do wyrównywania instrukcji do wielokrotności 8 bitów.

3.2.2 Pliki źródłowe

Format plików źródłowych jest dość prosty, podobny do standardowych plików assemblera. Instrukcje zapisywane są przy pomocy kodów określonych w definicji języka, argumenty oddzielone mogą być przecinkiem, spacją, tabulatorem bądź dowolną ich kombinacją.

Komentarze dostępne są jedynie w wersji całej linii (linia komentarza powinna zaczynać się od średnika). Wszystkie komentarze umieszczone na początku pliku przed pierwszą

linią z kodem bądź przed pierwszą pustą linią są umieszczane na początku pliku wynikowego, pozostałe komentarze umieszczane są w pliku wynikowym bezpośrednio przed kodem wygenerowanym z następnej instrukcji.

3.2.3 Kompilacja

3.3 Przykłady

Assembler dla procesora z projektu

Listing 3.1: Definicja języka

```
1
2 14
3 00000 STOP 0
4 00001 RESET 0
5 00010 JMP 1
6 00011 JZ 2
7 00100 ADD 3
8 00101 SUB 3
9 00110 LD 3
10 01000 OR 3
11 01001 AND 3
12 01010 LDI 2
13 01011 IN 2
14 01100 OUT 2
15 01110 STORE 4
16 10100 MUL 3
17 8
18 R0 000
19 R1 001
20 R2 010
21 R3 011
22 R4 100
23 R5 101
24 R6 110
25 R7 111
26 5
27 0 OPCODE 0 0 0
28 1 OPCODE 0 0 0 IM8
29 2 OPCODE Rd IM8
30 3 OPCODE Rd 0 Rd 0 Rd
31 4 OPCODE Rd IM16
```

Dostępnych jest 14 instrukcji 5 typów, każda z nich ma 5-bitowy kod. Procesor wyposażony jest w 8 rejestrów, każdy z nich o 3-bitowym kodzie. Typy instrukcji można przetłumaczyć następująco:

0. instrukcja składa się jedynie z samego kodu operacji, pozostałe zera służą do dopełnienia długości kodu wynikowego do wielokrotności 8 bitów. Instrukcja bezargumentowa.

Przykład: STOP – zatrzymanie wykonywania programu

1. instrukcja posiadająca jeden argument typu całkowitoliczbowego, 8 bitowego.

Przykład: JMP 0x20 – skok bezwarunkowy pod podany adres

2. instrukcja posiadająca dwa argumenty – w kolejności symbol rejestru oraz stałą liczbową 8-bitową.

Przykład: LDI R3, -123 – załadowanie do rejestru R3 wartości -123

3. instrukcja posiadająca 3 argumenty, wszystkie będące symbolami rejestrów.

Przykład: ADD R2, R3, R4 – dodanie rejestrów R3 i R4, umieszczenie wyniku w rejestrze R2

4. instrukcja posiadająca dwa argumenty - symbol rejestru oraz stałą liczbową 16-bitową.

Przykład: STORE R5, 0x1234 – zapisanie wartości rejestru R5 do pamięci RAM pod adres 0x1234.

Listing 3.2: Przykładowy kod programu

```
1 ; Simple test program for VHDL general-purpose CPU
2 ; Authors:
3 ;     Maciej Stefanczyk
4 ;     Kacper Szkudlarek
5
6     IN      R0 , -1
7     IN      R1 , -1
8     ADD     R2 , R1 , R0
9     OUT     R2 , 0
10    IN      R1 , -1
11    MUL     R3 , R1 , R2
12    OUT     R3 , 0
13
14 ; stop operation
15     STOP
```

Listing 3.3: Plik wynikowy dla źródła z listingu 3.2

```
1 -- Simple test program for VHDL general-purpose CPU
2 -- Authors:
3 --     Maciej Stefanczyk
4 --     Kacper Szkudlarek
5 WIDTH = 8;
6 DEPTH = 32;
7 ADDRESS_RADIX = DEC;
8 DATA_RADIX = BIN;
9
10 CONTENT BEGIN
11 -- 6: IN      R0 , -1
12     0 :      01011000;
13     1 :      11111111;
14 -- 7: IN      R1 , -1
```



```

15      2 :    01011001;
16      3 :    11111111;
17 -- 8: ADD R2, R1, R0
18      4 :    00100010;
19      5 :    00010000;
20 -- 9: OUT  R2, 0
21      6 :    01100010;
22      7 :    00000000;
23 -- 10: IN   R1, -1
24      8 :    01011001;
25      9 :    11111111;
26 -- 11: MUL  R3, R1, R2
27      10 :   10100011;
28      11 :   00010010;
29 -- 12: OUT  R3, 0
30      12 :   01100011;
31      13 :   00000000;
32 -- stop operation
33 -- 15: STOP
34      14 :   00000000;
35 END;

```

Rozdział 4

Kody źródłowe

4.1 Mikrokontroler

Listing 4.1: cpu.vhd

```
1  -----
2  -- CPU module
3  -----
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  use ieee.std_logic_arith.all;
8
9  library lpm;
10 use lpm.lpm_components.lpm_ram_dp;
11
12 entity cpu is
13 port ( CPU_CR : in std_logic;
14        CPU_GEN : in std_logic;
15        CPU_DATA : inout std_logic_vector (7 downto 0);
16        CPU_ADDR : buffer std_logic_vector (15 downto 0);
17        CPU_MREQ : buffer std_logic;
18        CPU_IOREQ : buffer std_logic;
19        CPU_RD : buffer std_logic;
20        CPU_WR : buffer std_logic;
21        CPU_WT : in std_logic);
22 end entity cpu;
23
24 architecture cpu of cpu is
25 COMPONENT booth_multiply is
26     GENERIC ( n : INTEGER := 8 );
27     PORT
28     (
29         M      : IN STD_LOGIC_VECTOR(n-1 DOWNT0 0);
30         R      : IN STD_LOGIC_VECTOR(n-1 DOWNT0 0);
31         WYN     : OUT STD_LOGIC_VECTOR(2*n-1 DOWNT0 0);
32         MUL_WT  : OUT STD_ULOGIC;
33         MUL_GEN : IN STD_ULOGIC;
34         MUL_CR  : IN STD_ULOGIC;
```

```

35         MUL      :      IN STD_ULOGIC
36     );
37 END COMPONENT booth_multiply;
38
39 type stany is (ST0, ST1, ST2, ST3, ST4, ST5, ST6, ST7, ST_WAIT);
40 shared variable STAN : stany;
41
42 -- sterowanie pamiecia rejestrow
43 shared variable REG_A : std_logic_vector (2 downto 0);
44 shared variable REG_D : std_logic_vector (2 downto 0);
45 shared variable D_A : std_logic_vector (7 downto 0);
46 shared variable R_D : std_logic_vector (7 downto 0);
47 shared variable WR_ENA : std_logic;
48
49 -- instruction cache
50 shared variable IC1 : std_logic_vector (7 downto 0);
51 shared variable IC2 : std_logic_vector (7 downto 0);
52 shared variable IC3 : std_logic_vector (7 downto 0);
53
54 -- register cache
55 shared variable TMP0 : std_logic_vector (7 downto 0);
56 shared variable TMP1 : std_logic_vector (7 downto 0);
57 shared variable TMP2 : std_logic_vector (7 downto 0);
58
59 shared variable TMP : std_logic;
60
61 -- OUT_LPT
62 shared variable OUT_WT: std_logic;
63
64 -- IN_PS2
65 shared variable IN_WT: std_logic;
66
67 -- mnozarka
68 shared variable B_M : std_logic;
69 shared variable M_WT : std_logic;
70 shared variable M_WY : std_logic_vector (7 downto 0);
71
72 -- flaga ustawiana po instrukcji STOP, zawiesza dzialanie procesora
73 shared variable STOP : std_logic;
74
75 -- program counter
76 shared variable PC : integer range 0 to 63;
77
78 -- pomocnicze liczniki
79 shared variable CNT : std_logic_vector(1 downto 0);
80 --shared variable DELAY : integer := 0;
81 shared variable DELAY : std_logic_vector(2 downto 0) := "000";
82
83 -----
84 -- funkcje pomocnicze
85 -----
86 function READ_REG(ADR : in std_logic_vector) return std_logic_vector is
87 begin
88     REG_A := ADR;

```

```

89     return (D_A);
90 end function READ_REG;
91
92 function WRITE_REG(ADR : in std_logic_vector; D : in std_logic_vector)
    return std_logic is
93 begin
94     R_D := D;
95     REG_D := ADR;
96     WR_ENA := '1';
97     return('1');
98 end function WRITE_REG;
99
100 function READ_RAM(ADR : in std_logic_vector) return std_logic is
101 begin
102     CPU_ADDR <= ADR;
103     CPU_RD <= '0';
104     CPU_WR <= '1';
105     CPU_MREQ <= '0';
106     CPU_IOREQ <= '1';
107     return('1');
108 end function READ_RAM;
109
110 function WRITE_RAM(ADR : in std_logic_vector) return std_logic is
111 begin
112     CPU_ADDR <= ADR;
113     CPU_RD <= '1';
114     CPU_WR <= '0';
115     CPU_MREQ <= '0';
116     CPU_IOREQ <= '1';
117     return('1');
118 end function WRITE_RAM;
119
120 function WRITE_OUT(ADR : in std_logic_vector) return std_logic is
121 begin
122     CPU_ADDR <= (x"00" & ADR);
123     CPU_RD <= '1';
124     CPU_WR <= '0';
125     CPU_MREQ <= '1';
126     CPU_IOREQ <= '0';
127     return('1');
128 end function WRITE_OUT;
129
130 function READ_IN(ADR : in std_logic_vector) return std_logic is
131 begin
132     CPU_ADDR <= (x"00" & ADR);
133     CPU_RD <= '0';
134     CPU_WR <= '1';
135     CPU_MREQ <= '1';
136     CPU_IOREQ <= '0';
137     return('1');
138 end function READ_IN;
139
140 begin
141 mul0: booth_multiply port map ( M => TMP1, R => TMP2, WYN(7 downto 0)

```

```

=> M_WY, MUL_WT => M_WT, MUL_GEN => CPU_GEN, MUL_CR => CPU_CR, MUL
=> B_M);
142
143 regA: lpm_ram_dp
144     generic map (LPM_WIDTH => 8, LPM_WIDTHAD => 3, LPM_NUMWORDS =>
145         8,
146         LPM_INDATA => "REGISTERED", LPM_OUTDATA =>
147             "UNREGISTERED",
148             LPM_RDADDRESS_CONTROL => "UNREGISTERED",
149             LPM_WRADDRESS_CONTROL => "REGISTERED")
150     port map ( rdaddress => REG_A, q => D_A,
151         wraddress => REG_D, data => R_D, wren => WR_ENA,
152         wrclken => '1', wrclock => CPU_GEN);
153
154 p0: process (CPU_GEN, CPU_CR) is
155     begin
156         if(CPU_CR = '0') then
157             PC := 0; -- poczatek adresow pamieci ROM
158             CNT := "00";
159             CPU_DATA <= (others => 'Z');
160             CPU_RD <= '1';
161             CPU_WR <= '1';
162             CPU_MREQ <= '1';
163             CPU_IOREQ <= '1';
164             STAN := ST0;
165             STOP := '0';
166         elsif (rising_edge(CPU_GEN)) then
167
168             case STAN is
169
170                 when ST0 => --jesli PC < od 32 (max pamieci), odczyt
171                     instrukcji z ROM
172                     IF (STOP = '1') THEN
173                         STAN := ST0;
174                     ELSE
175                         STAN := ST1;
176                         CPU_ADDR <= conv_std_logic_vector(PC, 16);
177                         CPU_RD <= '0';
178                         CPU_WR <= '1';
179                         CPU_MREQ <= '0';
180                         CPU_IOREQ <= '1';
181                         CPU_DATA <= (others => 'Z');
182                         WR_ENA := '0';
183                         B_M := '0';
184                     END IF;
185
186                 when ST1 => --oczekiwanie na odczyt z pamieci
187                     if (UNSIGNED(DELAY) < 3) THEN
188                         DELAY := UNSIGNED(DELAY) + 1;
189                         STAN := ST1;
190                     ELSE
191                         STAN := ST2;
192                         DELAY := "000";

```

```

190         END IF;
191
192     when ST2 => -- wstepne pobranie instrukcji i operandow
193         case CNT is
194             when "00" =>
195                 IC1 := CPU_DATA;
196                 STAN := ST0;
197                 CNT := "01";
198                 PC := PC + 1;
199             when "01" =>
200                 IC2 := CPU_DATA;
201                 STAN := ST0;
202                 CNT := "10";
203                 PC := PC + 1;
204                 TMP0 := READ_REG(IC1(2 downto 0));
205             when "10" =>
206                 TMP0 := D_A;
207                 IC3 := CPU_DATA;
208                 STAN := ST1;
209                 CNT := "11";
210                 PC := PC - 2;
211                 TMP1 := READ_REG(IC2(6 downto 4));
212             when "11" =>
213                 TMP1 := D_A;
214                 TMP2 := READ_REG(IC2(2 downto 0));
215                 STAN := ST3;
216                 CNT := "00";
217         end case;
218         CPU_RD <= '1';
219         CPU_WR <= '1';
220         CPU_MREQ <= '1';
221         CPU_IOREQ <= '1';
222
223     when ST_WAIT =>
224         if (UNSIGNED(DELAY) < 1) THEN
225             DELAY := UNSIGNED(DELAY) + 1;
226             STAN := ST_WAIT;
227         ELSE
228             STAN := ST0;
229             DELAY := "000";
230         END IF;
231
232     when ST3 =>
233         TMP2 := D_A;
234
235         case IC1(7 downto 3) is
236             when "00000" => -- OK STOP
237                 STOP := '1';
238                 STAN := ST0;
239
240             when "00001" => -- JMP 0
241                 PC := 0;
242                 STAN := ST0;
243

```

```

244      when "00010" => --      JMP A
245          PC := CONV_INTEGER(UNSIGNED(IC2));
246          STAN := ST0;
247
248      when "00011" => --      JZ Rd, A
249          IF (TMP0 = x"00") THEN
250              PC := CONV_INTEGER(UNSIGNED(IC2));
251          END IF;
252          STAN := ST0;
253
254      when "00100" => -- OK Rd <= Ra + Rb
255          TMP := WRITE_REG(IC1(2 downto 0),
256              UNSIGNED(TMP1) + UNSIGNED(TMP2));
257          PC := PC + 2;
258          STAN := ST0;
259
260      when "00101" => -- OK Rd <= Ra - Rb
261          TMP := WRITE_REG(IC1(2 downto 0),
262              UNSIGNED(TMP1) - UNSIGNED(TMP2));
263          PC := PC + 2;
264          STAN := ST0;
265
266      when "00110" => -- OK Rd <= RAM(RaRb)
267          if (DELAY = "000") then
268              TMP := READ_RAM( (TMP1 & TMP2) );
269              DELAY := UNSIGNED(DELAY) + 1;
270          elsif(UNSIGNED(DELAY) < 3) then
271              DELAY := UNSIGNED(DELAY) + 1;
272          else
273              CPU_RD <= '1';
274              TMP := WRITE_REG(IC1(2 downto 0),
275                  CPU_DATA);
276              DELAY := "000";
277              PC := PC + 2;
278              STAN := ST0;
279          end if;
280
281      when "01000" => -- OK Rd <= Ra # Rb
282          TMP := WRITE_REG(IC1(2 downto 0), TMP1 or
283              TMP2);
284          PC := PC + 2;
285          STAN := ST0;
286
287      when "01001" => -- OK Rd <= Ra & Rb
288          TMP := WRITE_REG(IC1(2 downto 0), TMP1 and
289              TMP2);
290          PC := PC + 2;
291          STAN := ST0;
292
293      when "01010" => -- OK Rd <= DIN
294          TMP := WRITE_REG(IC1(2 downto 0), IC2);
295          PC := PC + 2;
296          STAN := ST0;

```

```

293
294     when "01110" => -- OK Rd => RAM(A)
295         CPU_DATA <= TMP0;
296         TMP := WRITE_RAM((IC2 & IC3));
297
298         PC := PC + 3;
299         STAN := ST_WAIT;
300
301     when "10100" => --      Rd <= Ra * Rb
302         TMP := M_WT;
303         if(UNSIGNED(DELAY) < 3) then
304             B_M := '1';
305             DELAY := UNSIGNED(DELAY) + 1;
306         else
307
308             if (M_WT = '1') then
309                 STAN := ST3;
310                 TMP0 := M_WY;
311             else
312                 B_M := '0';
313                 TMP := WRITE_REG(IC1(2 downto 0),
314                                     M_WY);
315
316                 PC := PC + 2;
317                 STAN := ST0;
318                 DELAY := "000";
319             end if;
320         end if;
321
322     when "01100" => -- Rd => OUT(A)
323         if(UNSIGNED(DELAY) < 3) then
324             CPU_DATA <= TMP0;
325             TMP := WRITE_OUT(IC2);
326             DELAY := UNSIGNED(DELAY) + 1;
327         else
328             if(CPU_WT = '1') then
329                 STAN := ST3;
330             else
331                 PC := PC + 2;
332                 STAN := ST0;
333                 DELAY := "000";
334             end if;
335         end if;
336
337     when "01011" =>
338         if(UNSIGNED(DELAY) < 3) then
339             TMP := READ_IN(IC2);
340             DELAY := UNSIGNED(DELAY) + 1;
341         else
342             if(CPU_WT = '1') then
343                 STAN := ST3;
344
345             else
346                 TMP := WRITE_REG(IC1(2 downto 0),

```



```

346         CPU_DATA);
347         PC := PC + 2;
348         STAN := ST0;
349         DELAY := "000";
350     end if;
351
352     when others =>
353         STOP := '1';
354         STAN := ST0;
355     end case; -- IC1
356
357     when others =>
358         STOP := '1';
359         STAN := ST0;
360
361     end case;
362
363     end if;
364
365     end process p0;
366
367 end architecture cpu;

```

Listing 4.2: ram.vhd

```

1  -----
2  -- RAM
3  -- pamiec dostepna pod adresami 0x100 - 0x1FF
4  -----
5
6  library ieee;
7  use ieee.std_logic_1164.all;
8  library lpm;
9  use lpm.lpm_components.lpm_ram_dp;
10 use lpm.lpm_components.lpm_ram_dq;
11
12 entity ram is
13 port (   RAM_GEN : in std_logic;
14         RAM_DATA : inout std_logic_vector (7 downto 0);
15         RAM_ADDR : in std_logic_vector (15 downto 0);
16         RAM_MREQ : in std_logic;
17         RAM_WR   : in std_logic;
18         RAM_RD   : in std_logic);
19 end entity ram;
20
21 architecture pamiec_RAM of ram is
22 shared variable DOUT : std_logic_vector (7 downto 0);
23 signal ENAB : std_logic;
24
25 begin
26 f0: lpm_ram_dq
27     generic map (LPM_WIDTH => 8, LPM_WIDTHAD => 8, LPM_NUMWORDS =>
28                 256,

```

```

28         LPM_INDATA => "REGISTERED", LPM_OUTDATA => "REGISTERED",
29         LPM_ADDRESS_CONTROL => "REGISTERED")
30     port map ( address => RAM_ADDR(7 downto 0), q => DOUT,
31               data => RAM_DATA, we => (not RAM_WR) and (not
32                                     RAM_MREQ) and ENAB , inclock => RAM_GEN,
33               outclock => RAM_GEN);
34
35     --Ustawianie szyny danych w stan wysokiej impedancji, gdy nie jest
36     --używana
37     RAM_DATA <= DOUT when ((not RAM_RD) and (not RAM_MREQ) and ENAB) =
38         '1'
39         else (others => 'Z');
40
41     ENAB <= '1' when (RAM_ADDR(15 downto 8) = x"01")
42         else '0';
43
44 end architecture pamiec_RAM;

```

Listing 4.3: rom.vhd

```

1  -----
2  -- ROM
3  -- pamiec dostepna pod adresami 0x000 - 0xFF
4  -----
5
6  library ieee;
7  use ieee.std_logic_1164.all;
8
9  library lpm;
10 use lpm.lpm_components.lpm_rom;
11
12 entity rom is
13 port ( R_GEN : in std_logic;
14        R_DATA : inout std_logic_vector (7 downto 0);
15        R_ADDR : in std_logic_vector (15 downto 0);
16        R_MREQ : in std_logic;
17        R_RD : in std_logic);
18 end entity rom;
19
20 architecture pamiec_ROM of rom is
21 signal ENAB : std_logic;
22
23 begin
24
25 e0: lpm_rom
26     generic map(LPM_WIDTH => 8, LPM_WIDTHAD => 8, LPM_NUMWORDS => 256,
27                LPM_FILE=>"none.mif",
28                LPM_OUTDATA => "UNREGISTERED",
29                LPM_ADDRESS_CONTROL => "UNREGISTERED",
30                LPM_HINT=>"UNUSED")
31     port map ( address => R_ADDR(7 downto 0), q => R_DATA, memenab=>
32               (not R_RD) and ENAB and (not R_MREQ));
33
34     ENAB <= '1' when (R_ADDR(15 downto 8) = x"00")

```

```

33         else '0';
34
35 end architecture pamiec_ROM;

```

Listing 4.4: booth_multiply.vhd

```

1  -----
2  -- MNOZARKA
3  -----
4  -- zrodlo:
5  -- http://en.wikipedia.org/wiki/Booth's\_multiplication\_algorithm
6  -----
7
8  library ieee;
9
10 use ieee.std_logic_1164.all;
11 use ieee.std_logic_arith.all;
12
13 entity booth_multiply is
14 generic (n : positive := 8);
15 port ( M, R: in std_logic_vector (n - 1 downto 0); --liczby do
        pomnozenia
16         WYN: out std_logic_vector (2*n - 1 downto 0); --wynik mnozenia
17         MUL_WT : out std_ulogic; -- sygnal trwania przetwarzania
18         MUL_GEN: in std_ulogic; -- sygnal zegarowy
19         MUL_CR : in std_ulogic; -- sygnal reset
20         MUL : in std_ulogic);
21 end entity booth_multiply;
22
23 architecture booth_multiply_arch of booth_multiply is
24 shared variable A, S, P : std_logic_vector(2*n downto 0); -- dlugosc
        2n+1
25 shared variable Z1 : std_logic_vector(n downto 0);
26 shared variable Z2 : std_logic_vector(n-1 downto 0);
27 shared variable MM : std_logic_vector(n-1 downto 0);
28 shared variable CNT : integer range 0 to 31;
29 type stany is (ST0, ST1, ST2, ST3, ST4);
30
31 shared variable STAN : stany;
32
33 begin
34     p0 : process (MUL_GEN, MUL_CR, M, R, MUL) is
35     begin
36
37         if(MUL_CR = '0') then
38             STAN := ST0;
39             MUL_WT <= '0';
40             WYN <= (others => '0');
41             Z1 := (others => '0');
42             Z2 := (others => '0');
43         else
44             if rising_edge(MUL_GEN) then
45                 case (STAN) is
46                     when ST0 =>

```

```

47         WYN <= (others => '0');
48         if (MUL = '1') then
49             MUL_WT <= '1';
50             STAN := ST1;
51             MM := not M;
52             MM := UNSIGNED(MM) + 1;
53             A := (M & Z1);
54             S := (MM & Z1);
55             P := (Z2 & R & '0');
56             CNT := 0;
57         else
58             MUL_WT <= '0';
59             STAN := ST0;
60         end if;
61     when ST1 =>
62         MUL_WT <= '1';
63         if (P(1 downto 0) = "01") then
64             P := UNSIGNED(P) + UNSIGNED(A);
65         elsif (P(1 downto 0) = "10") then
66             P := UNSIGNED(P) + UNSIGNED(S);
67         end if;
68         STAN := ST2;
69     when ST2 =>
70         MUL_WT <= '1';
71         if (CNT < n) then
72             STAN := ST1;
73             P := (P(2*n) & P(2*n downto 1));
74             CNT := CNT + 1;
75         else
76             STAN := ST3;
77             CNT := 0;
78         end if;
79     when ST3 =>
80         MUL_WT <= '0';
81         if (MUL = '1') then
82             WYN <= P(2*n downto 1);
83             STAN := ST3;
84         else
85             WYN <= (others => '0');
86             STAN := ST0;
87         end if;
88     when others =>
89         MUL_WT <= '0';
90     end case;
91 end if;
92 end if;
93
94 end process p0;
95
96 end architecture booth_multiply_arch;

```

Listing 4.5: input_ps2.tdf

1 TITLE "Uklad wejsciowy mikroprocesora - PS2";

```

2
3 constant addr = B"00000000";
4
5
6 subdesign input_ps2(
7     PS2_DATA[7..0] : bidir; %Szyna danych %
8     PS2_GEN : input;      %Zegar ok. 20 MHz%
9     PS2_CR : input;
10
11     --PS2_OUT[7..0] : output;
12
13
14     PS2_ADDR[7..0] : input;
15     PS2_IOREQ : input;
16     PS2_RD : input;
17     PS2_WR : input;
18     PS2_WT : output;
19
20
21     PS2_IN_DATA : input;      %Dane przychodzace z lacza PS2 %
22     PS2_IN_CLK : input;      %Sygnal synchornizujacy przychodzace dane %
23
24 )
25
26 variable
27     AUT : machine of bits (Q[3..0]) %Automat obslugujacy komunikacje%
28         with states (START = 0, S0 = 1, S1 = 2, S2 = 3, S3 = 4, S4
29             = 5, S5 = 6, S6 = 7, S7 = 8);
30     --DATA[7..0] : DFF; %Dane calkowicie odczytane z portu%
31     REJ[7..0] : DFF;      %Dane odczytywane z portu%
32     PAR : DFF;            %Badanie parzystosci%
33
34     --PS2_DATA[7..0] : DFF;
35     TMP_DATA[7..0] : DFF;
36
37     PS2_CLK : DFF;
38     CNT[3..0] : DFF;      %Licznik ilosci odczytanych bitow%
39     OK : DFF;            %Czy dane sa poprawne%
40     SH : DFF; %czy wyswietlono%
41
42     NUM[1..0] : DFF;
43     READ : DFF;
44     OUT[9..0] : DFF;
45     TCNT[2..0] : DFF;
46     SIGN : DFF;
47 begin
48
49     %Podpiecie zegara do poszczegolnych elementow%
50     REJ[].clk = PS2_GEN;
51     PS2_CLK.clk = PS2_GEN;
52     CNT[].clk = PS2_GEN;
53     AUT.clk = PS2_GEN;
54     PAR.clk = PS2_GEN;

```

```

55     OK.clk = PS2_GEN;
56     SH.clk = OK;
57     READ.clk = PS2_GEN;
58     OUT[].clk = PS2_GEN;
59     TCNT[].clk = PS2_GEN;
60     SIGN.clk = PS2_GEN;
61     TMP_DATA[].clk = PS2_GEN;
62
63
64     %Podpięcie sygnału resetu%
65     REJ[].clrn = PS2_CR;
66     PS2_CLK.clrn = PS2_CR;
67     CNT[].clrn = PS2_CR;
68     PAR.clrn = PS2_CR;
69     AUT.reset = !PS2_CR;
70     OK.clrn = PS2_CR;
71     SH.clrn = PS2_CR;
72     READ.clrn = PS2_CR;
73     OUT[].clrn = PS2_CR;
74     TCNT[].clrn = PS2_CR;
75     SIGN.clrn = PS2_CR;
76     TMP_DATA[].clrn = PS2_CR;
77
78     SH = !SH;
79
80     NUM[].clk = PS2_GEN;
81     NUM[].clrn = PS2_CR;
82
83
84     PS2_CLK = PS2_IN_CLK;
85
86     --Ustawianie szyny w stan wysokiej impedancji
87     PS2_DATA[0] = TRI(TMP_DATA[0], !PS2_IOREQ & !PS2_RD & (PS2_ADDR[]
      == addr));
88     PS2_DATA[1] = TRI(TMP_DATA[1], !PS2_IOREQ & !PS2_RD & (PS2_ADDR[]
      == addr));
89     PS2_DATA[2] = TRI(TMP_DATA[2], !PS2_IOREQ & !PS2_RD & (PS2_ADDR[]
      == addr));
90     PS2_DATA[3] = TRI(TMP_DATA[3], !PS2_IOREQ & !PS2_RD & (PS2_ADDR[]
      == addr));
91     PS2_DATA[4] = TRI(TMP_DATA[4], !PS2_IOREQ & !PS2_RD & (PS2_ADDR[]
      == addr));
92     PS2_DATA[5] = TRI(TMP_DATA[5], !PS2_IOREQ & !PS2_RD & (PS2_ADDR[]
      == addr));
93     PS2_DATA[6] = TRI(TMP_DATA[6], !PS2_IOREQ & !PS2_RD & (PS2_ADDR[]
      == addr));
94     PS2_DATA[7] = TRI(TMP_DATA[7], !PS2_IOREQ & !PS2_RD & (PS2_ADDR[]
      == addr));
95
96     case AUT is
97         when START => TMP_DATA[] = TMP_DATA[];
98             if(!PS2_IOREQ & !PS2_RD & (PS2_ADDR[] == addr)) then
99                 PS2_WT = VCC;
100                 AUT = S0;

```

```

101         else
102             PS2_WT = GND;
103             AUT = START;
104         end if;
105     when S0 => TMP_DATA[] = TMP_DATA[]; PAR = GND; CNT[] = 0;
106             REJ[] = 0; OK = OK; READ = READ; OUT[] = OUT[];
107             PS2_WT = VCC;
108             TCNT[] = TCNT[]; SIGN = SIGN;
109             if (!PS2_CLK & !PS2_IN_DATA) then
110                 OK = GND;
111                 AUT = S2;
112             else AUT = S0;
113             end if;
114     when S1 => TMP_DATA[] = TMP_DATA[]; OK = OK; READ = READ;
115             OUT[] = OUT[];
116             TCNT[] = TCNT[]; SIGN = SIGN; PS2_WT = VCC;
117             if (CNT[] < 8) then
118                 REJ[] = (PS2_IN_DATA, REJ[7..1]);
119                 PAR = PAR xor PS2_IN_DATA;
120                 AUT = S2;
121             elsif (CNT[] == 8) then
122                 REJ[] = REJ[];
123                 PAR = PAR xor PS2_IN_DATA;
124                 AUT = S2;
125             elsif (CNT[] == 9) then
126                 REJ[] = REJ[];
127                 OK = PAR and PS2_IN_DATA;
128                 AUT = S4;
129             end if;
130             CNT[] = CNT[] + 1;
131     when S2 => TMP_DATA[] = TMP_DATA[]; PAR = PAR; CNT[] = CNT[];
132             REJ[] = REJ[]; OK = OK; READ = READ; OUT[] = OUT[];
133             TCNT[] = TCNT[]; SIGN = SIGN; PS2_WT = VCC;
134             if (PS2_CLK == VCC) then AUT = S3;
135             else AUT = S2; end if;
136     when S3 => TMP_DATA[] = TMP_DATA[]; PAR = PAR; CNT[] = CNT[];
137             REJ[] = REJ[]; OK = OK; READ = READ; OUT[] = OUT[];
138             TCNT[] = TCNT[]; SIGN = SIGN; PS2_WT = VCC;
139             if (PS2_CLK == GND) then AUT = S1;
140             else AUT = S3; end if;
141     when S4 => TMP_DATA[] = TMP_DATA[]; OK = OK; OUT[] = OUT[];
142             PS2_WT = VCC;
143             if (OK) then
144                 if (READ) then
145                     case REJ[] is
146                         WHEN X"70" => REJ[] = 0; TCNT[] =
147                             TCNT[]; SIGN = SIGN;
148                         WHEN X"69" => REJ[] = 1; TCNT[] =
149                             TCNT[]; SIGN = SIGN;
150                         WHEN X"72" => REJ[] = 2; TCNT[] =
151                             TCNT[]; SIGN = SIGN;
152                         WHEN X"7A" => REJ[] = 3; TCNT[] =
153                             TCNT[]; SIGN = SIGN;
154                         WHEN X"6B" => REJ[] = 4; TCNT[] =

```

```

146         TCNT[]; SIGN = SIGN;
        WHEN X"73" => REJ[] = 5; TCNT[] =
147         TCNT[]; SIGN = SIGN;
        WHEN X"74" => REJ[] = 6; TCNT[] =
148         TCNT[]; SIGN = SIGN;
        WHEN X"6C" => REJ[] = 7; TCNT[] =
149         TCNT[]; SIGN = SIGN;
        WHEN X"75" => REJ[] = 8; TCNT[] =
150         TCNT[]; SIGN = SIGN;
        WHEN X"7D" => REJ[] = 9; TCNT[] =
151         TCNT[]; SIGN = SIGN;
        WHEN X"79" => REJ[] = 10; SIGN = GND;
152         TCNT[] = 0;
        WHEN X"7B" => REJ[] = 10; SIGN = VCC;
        TCNT[] = 0;
153         WHEN OTHERS => REJ[] = 14; TCNT[] = 0;
        SIGN = SIGN;
154     end case;
155     AUT = S5;
156     READ = GND;
157 else
158     SIGN = SIGN;
159     TCNT[] = TCNT[];
160     AUT = S0;
161     IF (REJ[] == X"F0") THEN
162         READ = VCC;
163     ELSE
164         READ = GND;
165     END IF;
166 end if;
167 SIGN = SIGN;
168 else
169     TCNT[] = TCNT[];
170     SIGN = SIGN;
171 end if;
172 when S5 => TMP_DATA[] = TMP_DATA[]; REJ[] = REJ[]; SIGN =
    SIGN; PS2_WT = VCC;
173     if (REJ[] == 14) then
174         OUT[] = 0;
175         TCNT[] = 0;
176     elsif ( (TCNT[] == 0) and (REJ[] != 10) ) THEN
177         OUT[] = 1;
178         TCNT[] = 0;
179     else
180         CASE TCNT[] IS
181             WHEN 0 => OUT[] = 0; TCNT[] = 1;
182             WHEN 1 => OUT[] = (REJ[3..0], 0, 0, 0, 0,
                0, 0) + (REJ[4..0], 0, 0, 0, 0, 0) +
                (REJ[7..0], 0, 0); TCNT[] = 2;
183             WHEN 2 => OUT[] = OUT[] + (REJ[6..0], 0, 0,
                0) + (0, REJ[7..0], 0); TCNT[] = 3;
184             WHEN 3 => OUT[] = OUT[] + (0, 0, REJ[]);
185         END CASE;
186     end if;

```



```

187
188         if(TCNT[] == 3) then
189             AUT = S6;
190         else
191             AUT = S0;
192         end if;
193     when S6 => SIGN = SIGN; PS2_WT = VCC;
194         if(SIGN == VCC) then
195             if(OUT[] > 128) then
196                 TCNT[] = 0;
197                 OUT[] = 0;
198                 TMP_DATA[] = 0;
199                 AUT = S0;
200             else
201                 TMP_DATA[] = -OUT[7..0];
202                 OUT[] = OUT[];
203                 AUT = S7;
204             end if;
205         else
206             if(OUT[] > 127) then
207                 TCNT[] = 0;
208                 OUT[] = 0;
209                 TMP_DATA[] = 0;
210                 AUT = S0;
211             else
212                 TMP_DATA[] = OUT[7..0];
213                 OUT[] = OUT[];
214                 AUT = S7;
215             end if;
216         end if;
217     when S7 => OUT[] = OUT[]; TMP_DATA[] = TMP_DATA[]; PS2_WT =
                GND;
218         if(!PS2_IOREQ & !PS2_RD) then
219             AUT = S7;
220         else
221             AUT = START;
222         end if;
223
224     end case;
225
226 end;

```

Listing 4.6: output_lpt.tdf

```

1 TITLE "Uklad wyjsciowy portu LPT";
2
3 include "u2tobcd.inc";
4
5 constant addr = B"00000000";
6
7 subdesign output_lpt(
8     LPT_GEN : input;      %Zegar ok. 20 MHz%
9     LPT_CR  : input;
10

```

```

11     LPT_DATA[7..0] : input;
12     LPT_ADDR[7..0] : input;
13     LPT_IOREQ : input;
14     LPT_RD : input;
15     LPT_WR : input;
16     LPT_WT : output;
17
18     LPT_OUT_DATA[7..0] : output;
19     LPT_BUSY : input;
20     LPT_ACK : input;
21     LPT_STROBE : output;
22     LPT_SELECTLN : output;
23     LPT_SEL : input;
24     LPT_INIT : output;
25     LPT_AUTOFD : output;
26 )
27 variable
28     LPT_OUT_DATA[7..0] : DFF;
29     LPT_STROBE : DFF;
30     LPT_SELECTLN : DFF;
31     LPT_INIT : DFF;
32     LPT_AUTOFD : DFF;
33
34     CNT[3..0] : DFF;
35     NUM[2..0] : DFF;
36
37     AUT : machine of bits (Q[2..0])
38         with states (S0=0, S1=1, S2=2, S3=3, S4=4, S5=5);
39
40     BCD_MOD : u2tobcd;
41     SIGN : DFF;
42     HUNDS[1..0] : DFF;
43     TENS[3..0] : DFF;
44     UNITS[3..0] : DFF;
45
46
47 begin
48     LPT_OUT_DATA[].clk = LPT_GEN;
49     LPT_STROBE.clk = LPT_GEN;
50     LPT_SELECTLN.clk = LPT_GEN;
51     LPT_INIT.clk = LPT_GEN;
52     LPT_AUTOFD.clk = LPT_GEN;
53     CNT[].clk = LPT_GEN;
54
55     LPT_OUT_DATA[].clrn = LPT_CR;
56     LPT_STROBE.clrn = LPT_CR;
57     LPT_SELECTLN.clrn = LPT_CR;
58     LPT_INIT.clrn = LPT_CR;
59     LPT_AUTOFD.clrn = LPT_CR;
60     CNT[].clrn = LPT_CR;
61
62     NUM[].clk = LPT_GEN;
63     NUM[].clrn = LPT_CR;
64

```

```

65     SIGN.clk = LPT_GEN;
66     HUNDS[].clk = LPT_GEN;
67     TENS[].clk = LPT_GEN;
68     UNITS[].clk = LPT_GEN;
69
70     SIGN.clrn = LPT_CR;
71     HUNDS[].clrn = LPT_CR;
72     TENS[].clrn = LPT_CR;
73     UNITS[].clrn = LPT_CR;
74
75     AUT.clk = LPT_GEN;
76     AUT.reset = !LPT_CR;
77
78
79     LPT_SELECTLN = VCC;
80     LPT_AUTOFD   = GND;
81     LPT_INIT     = GND;
82
83
84     BCD_MOD.GEN = LPT_GEN;
85
86     case (AUT) is
87         %Czekamy na wystawienie adresu i danych na szynę%
88         when S0 => CNT[] = 0; LPT_OUT_DATA[] = 0; NUM[] = 0;
89                     LPT_STROBE = VCC;
90
91                     if(!LPT_IOREQ & !LPT_WR & (LPT_ADDR[] == addr)) then
92                         LPT_WT = VCC;
93                         BCD_MOD.COUNT = VCC;
94                         BCD_MOD.DATA[] = LPT_DATA[];
95                         if(BCD_MOD.WT) then
96                             AUT = S0;
97                         else
98                             SIGN = BCD_MOD.SIGN;
99                             HUNDS[] = BCD_MOD.HUNDS[];
100                             TENS[] = BCD_MOD.TENS[];
101                             UNITS[] = BCD_MOD.UNITS[];
102                             AUT = S1;
103                         end if;
104                     else
105                         LPT_WT = GND;
106                         AUT = S0;
107                     end if;
108
109         when s1 => LPT_STROBE = VCC; LPT_WT = VCC;
110                     SIGN = SIGN; HUNDS[] = HUNDS[]; TENS[] = TENS[];
111                     UNITS[] = UNITS[];
112                     case (NUM[]) is
113                         when 0 => if(SIGN) then %drukuj znak liczby
114                                     +/-%
115                                     LPT_OUT_DATA[] = 45;
116                                     else
117                                     LPT_OUT_DATA[] = 43;

```

```

116         end if;
117         NUM[] = 1;
118         AUT = S2;
119
120         when 1 => LPT_OUT_DATA[] = (B"000000",
121             HUNDS[]) + 48;
122             NUM[] = 2;
123         when 2 => LPT_OUT_DATA[] = (B"0000", TENS[])
124             + 48;
125             AUT = S2;
126             NUM[] = 3;
127         when 3 => LPT_OUT_DATA[] = (B"0000", UNITS[])
128             + 48;
129             NUM[] = 4;
130             AUT = S2;
131         when 4 => LPT_OUT_DATA[] = 13;
132             NUM[] = 5;
133             AUT = S2;
134         when 5 => LPT_OUT_DATA[] = 10;
135             AUT = S2;
136             NUM[] = 6;
137     end case;
138     %Opuszczamy STROBE na 500 ns%
139     when S2 => LPT_OUT_DATA[] = LPT_OUT_DATA[]; NUM[] = NUM[];
140         LPT_WT = VCC;
141         SIGN = SIGN; HUNDS[] = HUNDS[]; TENS[] = TENS[];
142         UNITS[] = UNITS[];
143         LPT_STROBE = GND;
144         CNT[] = CNT[] + 1;
145         if ( CNT[] < 10) then
146             AUT = S2;
147         else
148             AUT = S3;
149             CNT[] = 0;
150         end if;
151     when S3 => LPT_OUT_DATA[] = LPT_OUT_DATA[]; NUM[] = NUM[];
152         LPT_WT = VCC;
153         SIGN = SIGN; HUNDS[] = HUNDS[]; TENS[] = TENS[];
154         UNITS[] = UNITS[];
155         LPT_STROBE = GND;
156
157         if (!LPT_BUSY) then
158             AUT = S3;
159         else
160             AUT = S4;
161         end if;
162     %Drukarka zakonczyła przetwarzanie%
163     when S4 => LPT_OUT_DATA[] = LPT_OUT_DATA[]; NUM[] = NUM[];
164         LPT_WT = VCC;
165         SIGN = SIGN; HUNDS[] = HUNDS[]; TENS[] = TENS[];
166         UNITS[] = UNITS[];
167         LPT_STROBE = VCC;
168         CNT[] = 0;
169
170

```

```

161         if (!LPT_BUSY) then
162             AUT = S5;
163         else
164             AUT = S4;
165         end if;
166     when S5 => LPT_OUT_DATA[] = LPT_OUT_DATA[]; NUM[] = NUM[];
167                SIGN = SIGN; HUNDS[] = HUNDS[]; TENS[] = TENS[];
168                UNITS[] = UNITS[];
169                LPT_STROBE = VCC;
170                CNT[] = 0;
171
172                if (LPT_ACK) then
173                    if (NUM[] == 6) then
174                        LPT_WT = GND;
175                        AUT = S0;
176                    else
177                        LPT_WT = VCC;
178                        AUT = S1;
179                    end if;
180                else
181                    LPT_WT = VCC;
182                    AUT = S5;
183                end if;
184    end case;
185
186
187 end;

```

Listing 4.7: u2tobcd.tdf

```

1 TITLE "Konwerter z U2 do BCD";
2
3
4 SUBDESIGN U2TOBCD(
5     DATA[7..0] : input;
6     COUNT : input;
7     GEN : input;
8
9     WT : output;
10
11     SIGN : output;
12     HUNDS[1..0] : output;
13     TENS[3..0] : output;
14     UNITS[3..0] : output;
15 )
16 VARIABLE
17     SIGN : DFF;
18     TMP[9..0] : DFF;
19     CNT[2..0] : DFF;
20     D[7..0] : DFF;
21
22     AUT : machine of bits (Q[1..0])
23         with states (S0 = 0, S1 = 1, S2 = 2, S3 = 3);

```

```

24
25 BEGIN
26     TMP[].clk = GEN;
27     CNT[].clk = GEN;
28     D[].clk = GEN;
29     SIGN.clk = GEN;
30
31     AUT.clk = GEN;
32
33
34     case (AUT) is
35         when S0 => TMP[] = 0; CNT[] = 0;
36                     if (COUNT) then
37                         WT = VCC;
38                         SIGN = DATA7;
39                         if (DATA7) then
40                             D[] = (NOT DATA[]) + 1;
41                         else
42                             D[] = (GND, DATA[6..0]);
43                         end if;
44                         AUT = S1;
45                     else
46                         WT = GND;
47                         AUT = S0;
48                     end if;
49         when S1 => WT = VCC; CNT[] = CNT[];
50                     SIGN = SIGN;
51                     TMP[] = (TMP[8..0], D7);
52                     D[] = (D[6..0], GND);
53                     AUT = S2;
54         when S2 => WT = VCC; SIGN = SIGN;
55                     D[] = D[];
56                     if (CNT[] < 7) then
57                         if (TMP[3..0] >= 5) then TMP[3..0] = TMP[3..0] +
58                             3; else TMP[3..0] = TMP[3..0]; end if;
59                         if (TMP[7..4] >= 5) then TMP[7..4] = TMP[7..4] +
60                             3; else TMP[7..4] = TMP[7..4]; end if;
61                         TMP[9..8] = TMP[9..8];
62                         --TMP[] = (TMP[8..0], D7);
63                         --D[] = (D[6..0], GND);
64                         --AUT = S2;
65                         AUT = S1;
66                     else
67                         TMP[] = TMP[];
68                         AUT = S3;
69                     end if;
70         when S3 => CNT[] = CNT[] + 1;
71                     WT = GND;
72                     TMP[] = TMP[];
73                     SIGN = SIGN;
74                     HUNDS[] = TMP[9..8];
75                     TENS[] = TMP[7..4];
76                     UNITS[] = TMP[3..0];
77                     if (COUNT) then

```

```

76             AUT = S3;
77         else
78             AUT = S0;
79         end if;
80     end case;
81
82 END;

```

Listing 4.8: seg.vhd

```

1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_arith.all;
4
5  entity seg is
6  port (   SEG_CR : in std_logic;
7          SEG_GEN : in std_logic;
8          SEG_DATA : in std_logic_vector (7 downto 0);
9          SEG_ADDR : in std_logic_vector (15 downto 0);
10         SEG_MREQ : in std_logic;
11         SEG_WR : in std_logic;
12         SEG_RD : in std_logic;
13         SEG_OUT : out std_logic_vector(7 downto 0));
14 end entity seg;
15
16 architecture seg of seg is
17 shared variable D_OUT : std_logic_vector(7 downto 0);
18
19 begin
20
21 p0: process (SEG_GEN, SEG_CR, SEG_DATA) is
22     begin
23         if (SEG_CR = '0') then
24             D_OUT := (others => '1');
25         else
26             if rising_edge(SEG_GEN) then
27                 if (SEG_ADDR(7 downto 0) = "00001111") then
28                     D_OUT := SEG_DATA;
29                 end if;
30                 SEG_OUT <= D_OUT;
31             end if;
32         end if;
33
34     end process p0;
35
36 end architecture seg;

```

4.2 Assembler

Listing 4.9: asm.cpp

```

1 #include <iostream>
2 #include <fstream>

```

```

3 #include <map>
4 #include <string>
5 #include <vector>
6 #include <string>
7 #include <algorithm>
8 #include <sstream>
9 #include <cstring>
10
11 using namespace std;
12
13 /*
14  * Command line parsing
15  */
16 typedef std::map<std::string, std::vector<std::string> > CommandLine;
17
18
19 const char SWITCH_CHAR = '-'; // lub '/'
20 const CommandLine ParseCommandLine(int argc, const char* argv[])
21 {
22     CommandLine cl;
23     for (int i = 1; i < argc; )
24         if (*(argv[i]) == SWITCH_CHAR)
25         {
26             std::vector<std::string> p;
27             p.reserve (argc - i);
28
29             int j;
30             for (j = i + 1;
31                  j < argc && (*(argv[j]) != SWITCH_CHAR ||
32                               strstr(argv[j], " "));
33                  ++j)
34                 p.push_back (argv[j]);
35
36             cl.insert (std::make_pair(argv[i] + 1, p));
37             i = j;
38         }
39         else
40             ++i;
41     return cl;
42 }
43
44 /*
45  * Tokenizer
46  */
47 class Tokenizer
48 {
49     public:
50         static const std::string DELIMITERS;
51         Tokenizer(const std::string& str);
52         Tokenizer(const std::string& str, const std::string&
53                   delimiters);
54         bool NextToken();
55         bool NextToken(const std::string& delimiters);

```



```

55         const std::string GetToken() const {
56             return m_token;
57         }
58         void Reset();
59     protected:
60         const std::string m_string;
61         size_t m_offset;
62         std::string m_delimiters;
63         std::string m_token;
64 };
65
66 const string Tokenizer::DELIMITERS(" \t\n\r");
67
68 Tokenizer::Tokenizer(const std::string& s) :
69     m_string(s),
70     m_offset(0),
71     m_delimiters(DELIMITERS) {}
72
73 Tokenizer::Tokenizer(const std::string& s, const std::string&
74     delimiters) :
75     m_string(s),
76     m_offset(0),
77     m_delimiters(delimiters) {}
78
79 bool Tokenizer::NextToken()
80 {
81     return NextToken(m_delimiters);
82 }
83
84 bool Tokenizer::NextToken(const std::string& delimiters)
85 {
86     size_t i = m_string.find_first_not_of(delimiters, m_offset);
87     if (string::npos == i)
88     {
89         m_offset = m_string.length();
90         return false;
91     }
92     size_t j = m_string.find_first_of(delimiters, i);
93     if (string::npos == j)
94     {
95         m_token = m_string.substr(i);
96         m_offset = m_string.length();
97         return true;
98     }
99
100     m_token = m_string.substr(i, j - i);
101     m_offset = j;
102     return true;
103 }
104
105 /*
106  * Language description
107 */

```

```

108
109 struct Instruction {
110     // binary opcode
111     std::string opcode;
112     // mnemonic
113     std::string mnemo;
114     // length in bytes
115     int length;
116     // instruction type
117     int type;
118 };
119
120 // list of language instructions
121 std::map<std::string, Instruction> lang;
122 // list of definitions (mapping registers to its numeric
    representations)
123 std::map<std::string, std::string> defs;
124 // list of possible instruction types
125 std::map<int, std::vector<std::string>> types;
126 // list of defined labels with corresponding addresses
127 std::map<std::string, int> labels;
128
129 // lightweight boost-like lexical cast
130 template<typename T2, typename T1>
131 inline T2 lexical_cast(const T1 &in) {
132     T2 out;
133     std::stringstream ss;
134     ss << in;
135     ss >> out;
136
137     if (ss.fail() || !ss.eof())
138         throw in + " is not a valid integer value";
139
140     return out;
141 }
142
143 // helper class converting hex numbers
144 template<typename ElemT>
145 struct HexTo {
146     ElemT value;
147     operator ElemT() const {return value;}
148     friend std::istream& operator>>(std::istream& in, HexTo& out) {
149         in >> std::hex >> out.value;
150         return in;
151     }
152 };
153
154 // convert string to integer
155 int str2int (const string &str) {
156     int n;
157
158     if ( (str.length() > 1) && (str.substr(0, 2) == "0x") ) {
159         n = lexical_cast< HexTo<int> >(str);
160     } else {

```

```

161         n = lexical_cast< int >(str);
162     }
163
164     return n;
165 }
166
167 // convert integer to its binary form (8 bit, U2 form)
168 std::string int2bin8(int n) {
169     std::string s;
170     for (int i = 0; i < 8; ++i) {
171         if (n & 1)
172             s = "1" + s;
173         else
174             s = "0" + s;
175
176         n >>= 1;
177     }
178
179     return s;
180 }
181
182 // convert string to its binary form (8 bit, U2 form)
183 std::string str2bin8(const std::string & str) {
184     int n;
185     if (labels.count(str)) {
186         n = labels[str];
187         std::cout << "-- Found label: " << str << "=" << n << "\n";
188     }
189     else
190         n = str2int(str);
191
192     if ( (n > 127) || (n < -128) ) {
193         throw str + ": out of range (should be [-128..127])";
194     }
195
196     return int2bin8(n);
197 }
198
199 // convert integer to its binary form (16 bit, U2 form)
200 std::string int2bin16(int n) {
201     std::string s;
202     for (int i = 0; i < 16; ++i) {
203         if (n & 1)
204             s = "1" + s;
205         else
206             s = "0" + s;
207
208         n >>= 1;
209     }
210
211     return s;
212 }
213
214 // convert string to its binary form (16 bit, U2 form)

```

```

215 std::string str2bin16(const std::string & str) {
216     int n;
217     if (labels.count(str)) {
218         n = labels[str];
219         std::cout << "-- Found label: " << str << "=" << n << "\n";
220     }
221     else
222         n = str2int(str);
223
224     if ( (n > 32767) || (n < -32768) ) {
225         throw str + ": out of range (should be [-32768..32767])";
226     }
227
228     return int2bin16(n);
229 }
230
231 // convert char to uppercase
232 struct upper {
233     int operator()(int c)
234     {
235         return std::toupper((unsigned char)c);
236     }
237 };
238
239 // convert integer to uppercase
240 std::string uppercase(std::string s) {
241     std::transform(s.begin(), s.end(), s.begin(), upper());
242     return s;
243 }
244
245 // strip whitespaces from begining and end of string
246 std::string strip(const std::string & line) {
247     int f = line.find_first_not_of(" \t\r\n");
248     int l = line.find_last_not_of(" \t\r\n");
249     //std::cout << f << "." << l << std::endl;
250     if (f >= l)
251         return "";
252
253     return line.substr(f, l-f+1);
254 }
255
256 // load language description from file
257 void loadLanguageDesc(const char * fname = NULL) {
258     // language description file
259     std::ifstream f;
260     // instruction count
261     int cnt;
262     // temporary
263     Instruction ins;
264
265     if (!fname)
266         f.open ("lang.txt", std::ifstream::in);
267     else
268         f.open (fname, std::ifstream::in);

```

```

269
270     f >> cnt;
271
272     for (int i = 0; i < cnt; ++i) {
273         f >> ins.opcode >> ins.mnemo >> ins.length >> ins.type;
274         lang[ins.mnemo] = ins;
275     }
276
277     f >> cnt;
278
279     string s1, s2;
280     for (int i = 0; i < cnt; ++i) {
281         f >> s1 >> s2;
282         defs[s1] = s2;
283     }
284
285     f >> cnt;
286     int t;
287     std::vector<std::string> tokens;
288     for (int i = 0; i < cnt; ++i) {
289         f >> t;
290         tokens.clear();
291         getline(f, s1);
292         Tokenizer s(s1, " \t,");
293         //std::cout << "Type: " << t << "\n";
294         while (s.NextToken()) {
295             tokens.push_back(s.GetToken());
296             //std::cout << "\t" << s.GetToken() << "\n";
297         }
298         types[t] = tokens;
299     }
300 }
301
302 /*
303  * Convert each type of instruction to its binary form
304  */
305 std::vector<std::string> type0(Instruction ins,
306     std::vector<std::string> tokens) {
307     if (tokens.size() != 1)
308         throw tokens[0] + " should have no arguments";
309
310     std::vector<std::string> ret;
311     std::string s;
312     s = ins.opcode;
313     s += "000";
314     ret.push_back(s);
315     return ret;
316 }
317 std::vector<std::string> type1(Instruction ins,
318     std::vector<std::string> tokens) {
319     if (tokens.size() != 2)
320         throw tokens[0] + " should have one argument";

```

```

321     std::vector<std::string> ret;
322     std::string s;
323     s = ins.opcode;
324     s += "000";
325     ret.push_back(s);
326     s = str2bin8(tokens[1]);
327     ret.push_back(s);
328     return ret;
329 }
330
331 std::vector<std::string> type2(Instruction ins,
    std::vector<std::string> tokens) {
332     if (tokens.size() != 3)
333         throw tokens[0] + " should have two arguments";
334
335     if (defs.count(tokens[1]) < 1)
336         throw tokens[1] + " unknown. Should be register name R0..R7";
337
338     std::vector<std::string> ret;
339     std::string s;
340     s = ins.opcode;
341     s += defs[tokens[1]];
342     ret.push_back(s);
343     s = str2bin8(tokens[2]);
344     ret.push_back(s);
345     return ret;
346 }
347
348 std::vector<std::string> type3(Instruction ins,
    std::vector<std::string> tokens) {
349     if (tokens.size() != 4)
350         throw tokens[0] + " should have three arguments";
351
352     if (defs.count(tokens[1]) < 1)
353         throw tokens[1] + " unknown. Should be register name R0..R7";
354
355     if (defs.count(tokens[2]) < 1)
356         throw tokens[2] + " unknown. Should be register name R0..R7";
357
358     if (defs.count(tokens[3]) < 1)
359         throw tokens[3] + " unknown. Should be register name R0..R7";
360
361     std::vector<std::string> ret;
362     std::string s;
363     s = ins.opcode;
364     s += defs[tokens[1]];
365     ret.push_back(s);
366     s = "0" + defs[tokens[2]] + "0" + defs[tokens[3]];
367     ret.push_back(s);
368     return ret;
369 }
370
371 std::vector<std::string> type4(Instruction ins,
    std::vector<std::string> tokens) {

```

```

372     if (tokens.size() != 3)
373         throw tokens[0] + " should have three arguments";
374
375     if (defs.count(tokens[1]) < 1)
376         throw tokens[1] + " unknown. Should be register name R0..R7";
377
378     std::vector<std::string> ret;
379     std::string s;
380     s = ins.opcode;
381     s += defs[tokens[1]];
382     ret.push_back(s);
383     s = str2bin16(tokens[2]);
384     ret.push_back(s.substr(0, 8));
385     ret.push_back(s.substr(8, 8));
386     return ret;
387 }
388
389 /*
390  * Assembly given tokens into instruction.
391  */
392 std::vector<std::string> assemblyLine(std::vector<std::string> tokens) {
393     Instruction ins;
394
395     if (tokens.size() < 1)
396         throw "Empty line";
397
398     std::string mnemo = uppercase(tokens[0]);
399
400     std::vector<std::string> ret;
401     std::string s;
402
403     if (lang.count(mnemo) < 1) {
404         throw mnemo + " - unknown instruction";
405     }
406
407     ins = lang[mnemo];
408
409     for (size_t i = 0; i < tokens.size(); ++i) {
410         //std::cout << "\t-- " << tokens[i] << "\n";
411     }
412
413     if (types.count(ins.type) < 1) {
414         throw mnemo + " - unknown instruction type (check language
415             definition file)";
416     }
417
418     tokens[0] = ins.opcode;
419
420     size_t cnt = 0;
421     for (size_t i = 0; i < types[ins.type].size(); ++i) {
422         if (types[ins.type][i] == "0") {
423             s += "0";
424             continue;

```

```

425     }
426
427     if (tokens.size() <= cnt)
428         throw mnemo + " - to few arguments";
429
430     if (types[ins.type][i] == "OPCODE") {
431         s += tokens[cnt];
432         cnt++;
433         continue;
434     }
435
436     if (types[ins.type][i] == "Rd") {
437         if (defs.count(tokens[cnt]) < 1)
438             throw tokens[cnt] + " unknown. Should be register name
439                 R0..R7";
439         s += defs[tokens[cnt]];
440         cnt++;
441         continue;
442     }
443
444     if (types[ins.type][i] == "IM8") {
445         s += str2bin8(tokens[cnt]);
446         cnt++;
447         continue;
448     }
449
450     if (types[ins.type][i] == "IM16") {
451         s += str2bin16(tokens[cnt]);
452         cnt++;
453         continue;
454     }
455 }
456
457 if (tokens.size() > cnt)
458     throw mnemo + " - to much arguments";
459
460 for (size_t i = 0; i < s.length() / 8; ++i) {
461     ret.push_back(s.substr(i*8, 8));
462 }
463
464 return ret;
465 }
466
467 /*
468  * Assembly given file.
469  */
470 void assembly(const char * fname) {
471     std::ifstream f(fname);
472     std::string line;
473     int cnt = 0;
474     int wrd = 0;
475     std::vector<std::string> tokens;
476     std::vector<std::string> codes;
477     bool first = true;

```



```

478
479     if (!f.good()) {
480         throw "No such file";
481     }
482
483     try {
484         while(!f.eof()) {
485             cnt++;
486             tokens.clear();
487             getline(f, line);
488             //std::cout << "[" << line << "]\n";
489             line = strip(line);
490             //std::cout << "[" << line << "]\n";
491
492             if (line.length() < 1) {
493                 if (first) {
494                     std::cout << "WIDTH = 8;\n"
495                                 "DEPTH = 32;\n"
496                                 "ADDRESS_RADIX = DEC;\n"
497                                 "DATA_RADIX = BIN;\n"
498                                 "\n"
499                                 "CONTENT BEGIN\n";
500
501                     first = false;
502                 }
503                 continue;
504             }
505
506             // comment
507             if (line[0] == ';' ) {
508                 line[0] = '-';
509                 line = "-" + line;
510                 std::cout << line << std::endl;
511                 continue;
512             }
513
514             if (first) {
515                 std::cout << "WIDTH = 8;\n"
516                             "DEPTH = 32;\n"
517                             "ADDRESS_RADIX = DEC;\n"
518                             "DATA_RADIX = BIN;\n"
519                             "\n"
520                             "CONTENT BEGIN\n";
521
522                 first = false;
523             }
524
525             // label
526             if (line[line.length()-1] == ':') {
527                 std::string label = strip(line.substr(0,
528                                     line.length()-1));
529                 cout << "-- New label: " << label << " at " << wrd <<
530                     "\n";
531                 std::cout << "-- " << cnt << ": " << line << "\n";

```

```

530         labels[label] = wrd;
531         continue;
532     }
533
534     Tokenizer s(line, " \t,");
535     while (s.NextToken()) {
536         tokens.push_back(s.GetToken());
537     }
538     if (tokens.size()) {
539         std::cout << "-- " << cnt << ": " << line << "\n";
540         codes = assemblyLine(tokens);
541         for (size_t i = 0; i < codes.size(); ++i) {
542             std::cout << "\t" << wrd << " :\t" << codes[i] <<
543                 "\n";
544             wrd++;
545         }
546     }
547
548     std::cout << "END;\n";
549
550 }
551 catch (string s) {
552     std::cout << "Error: " << cnt << ": " << s << std::endl;
553 }
554 catch (const char * s) {
555     std::cout << "Error: " << cnt << ": " << s << std::endl;
556 }
557 catch (...) {
558     std::cout << "Error: " << cnt << ": " << "Unknown error\n";
559 }
560 }
561
562 int main(int argc, char * argv[]) {
563     if (argc < 2) {
564         std::cout << "Usage: " << argv[0] << " file\n";
565         return 0;
566     }
567
568     try {
569         loadLanguageDesc();
570         assembly(argv[1]);
571     }
572     catch (string s) {
573         std::cout << "Error: " << s << std::endl;
574     }
575     catch (const char * s) {
576         std::cout << "Error: " << s << std::endl;
577     }
578     catch (...) {
579         std::cout << "Unknown error\n";
580     }
581
582     return 0;

```

