

Logika cyfrowa

Programistyczna lista zadań nr 14

Termin: 14 czerwca 2020

Uwaga! Poniższe zadania należy rozwiązać przy użyciu języka SystemVerilog, sprawdzić w DigitalJS oraz wysłać w systemie Web-CAT na SKOS. Należy pamiętać, aby nazwy portów nadesłanego modułu zgadzały się z podanymi w treści zadania. Wysłany plik powinien mieć nazwę `toplevel.sv`. **Nie przestrzeganie tych zasad będzie skutkowało przyznaniem 0 punktów.**

1. Brainf**k¹ to bardzo prosty język programowania, w którym – pomimo jego ekstremalnej prostoty – można wyrazić dowolny algorytm.

Program w tym języku jest ciągiem jednego z ośmiu wyróżnionych znaków ASCII, gdzie każdy z nich oznacza inną instrukcję do wykonania. Podobnie jak w konwencjonalnych językach programowania, program jest standardowo wykonywany po jednej instrukcji, po kolei, od lewej do prawej; jedynie wyróżnione instrukcje sterujące mogą zmieniać kolejność wykonania.

Pamięć w języku Brainf**k nie jest adresowana bezpośrednio. Zamiast tego programista ma do dyspozycji „głowicę”, wskazującą na jedną z komórek pamięci rozmiaru bajtu. Operacje mogą być wykonywane tylko na komórce wskazywanej przez głowicę, zaś sama głowica może być w jednym kroku przesunięta tylko o jedną komórkę w lewo lub prawo.

Osiem instrukcji języka Brainf**k to:

	odpowiednik w C	opis słowny
+	<code>mem[hd]++</code>	dodaj 1 do komórki pod głowicą
-	<code>mem[hd]--</code>	odejmij 1 od komórki pod głowicą
>	<code>hd++</code>	przesuń głowicę w prawo
<	<code>hd--</code>	przesuń głowicę w lewo
[<code>while(mem[hd]) {</code>	jeśli 0 pod głowicą, skocz do pasującego]
]	<code>}</code>	jeśli nie 0 pod głowicą, skocz do pasującego [
.	<code>putchar(mem[hd])</code>	wypisz znak
,	<code>mem[hd] = getchar()</code>	wczytaj znak

Bajt zerowy `\0` (nie mylić ze znakiem ASCII 0 o kodzie 48) będziemy uważać za znak końca programu.

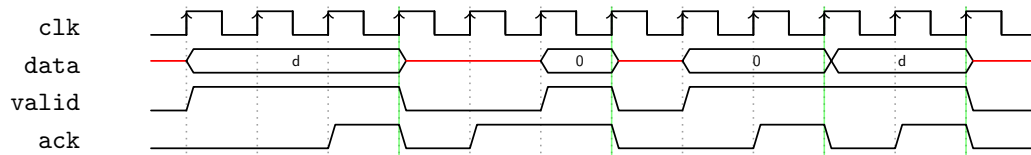
Celem zadania jest realizacja układu cyfrowego, który będzie wykonywał programy w języku Brainf**k. Układ powinien mieć następujące wejścia i wyjścia:

- `clk` – wejście zegara,
- `nrst` – zanegowane wejście resetu asynchronicznego,
- `in_data` – 8-bitowe wejście,
- `in_valid` – jednobitowe wejście oznaczające poprawność danych w `in_data`,
- `in_ack` – jednobitowe wyjście potwierdzające otrzymanie danych w `in_data`,
- `out_data` – 8-bitowe wyjście,
- `out_valid` – jednobitowe wyjście oznaczające poprawność danych w `out_data`,
- `out_ack` – jednobitowe wejście potwierdzające otrzymanie danych w `out_data`,
- `start` – 1-bitowe wejście uruchamiające obliczenia,
- `ready` – 1-bitowe wyjście sygnalizujące gotowość układu do rozpoczęcia pracy.

¹<https://en.wikipedia.org/wiki/Brainfuck>

Trójki sygnałów **data**, **valid** i **ack** tworzą *magistrale* komunikujące procesor z wejściem/wyjściem. Sygnałem **valid** nadawca sygnalizuje pojawienie się nowych danych w **data**, zaś sygnałem **ack** odbiorca sygnalizuje odebranie danych. Sygnał **valid** musi się pojawić w tym samym lub późniejszym cyklu niż dane w **data**. Protokół zakłada, że jeśli w momencie pojawienia się zbocza narastającego zarówno **valid** i **ack** były w stanie wysokim, transfer danych powiódł się. Nadawca może w takiej sytuacji albo wstawić nowe dane do **data** i utrzymać stan wysoki **valid**, albo zmienić stan **valid** na niski.

Przykładowa komunikacja jest opisana poniższym diagramem przebiegów, pokazującym przesłanie ciągu d00d. Sygnał **ack** w poniższym diagramie pokazuje różne poprawne sposoby działania odbiorcy danych. Zielonymi liniami oznaczone są momenty, w których następuje transfer danych.



Rozwiązanie powinno zawierać co najmniej 256 bajtów pamięci kodu i 256 bajtów pamięci danych. Jeśli wskaźnik instrukcji lub głowica wyjdą poza dostępny obszar pamięci, zachowanie układu może być dowolne. Każda z dwóch pamięci powinna być pamięcią RAM z portem do odczytu (asynchronicznego) i portem do zapisu.

Kiedy układ jest w stanie gotowości, wejście danych będzie służyć do ładowania programów. Kolejne bajty przesłane przy użyciu sygnałów **in_data**, **in_valid** oraz **in_ack** powinny zostać załadowane do kolejnych adresów w pamięci kodu, zaczynając od 0. Po rozpoczęciu pracy układu, zanim zacznie się wykonywanie kodu, pamięć danych powinna zostać wyzerowana.

Można wykorzystać `$readmemh` do załadowania programu na etapie testów, jednak skrypt sprawdzający będzie wykorzystywał metodę opisaną powyżej. Aby wygenerować obraz pamięci w formacie zrozumiałym dla SystemVeriloga, można użyć następującego skryptu w Pythonie:

```
t = "+[-[<<[+--->]-[<<<]]>>>-]>-.---.>..>.<<<<-.<+.>>>>>.>.<<.-.-."
f = open("program.vh", "w")
for c in t: f.write("%x " % ord(c));
f.close()
```

Przy implementacji można wspomóc się poniższym diagramem algorytmicznym. Inne implementacje są akceptowalne, o ile przestrzegają zasad wymienionych powyżej.

