Logika cyfrowa

Praktyczna lista zadań nr 14

Termin: 12 czerwca 2024 godzina 30:00

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 skutkować przyznaniem 0 punktów.**

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

   Program w tym języku jest dowolnej długości ciągiem 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
`+`	`mem[hd]++`	dodaj 1 do komórki pod głowicą
`-`	`mem[hd]--`	odejmij 1 od komórki pod głowicą
`>`	`hd++`	przesuń głowicę w prawo
`<`	`hd--`	przesuń głowicę w lewo
`[`	`while(mem[hd]) {`	jeśli 0 pod głowicą, skocz do pasującego `]`
`]`	`}`	jeśli nie 0 pod głowicą, skocz do pasującego `[`
`.`	`putchar(mem[hd])`	wypisz znak
`,`	`mem[hd] = getchar()`	wczytaj znak

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

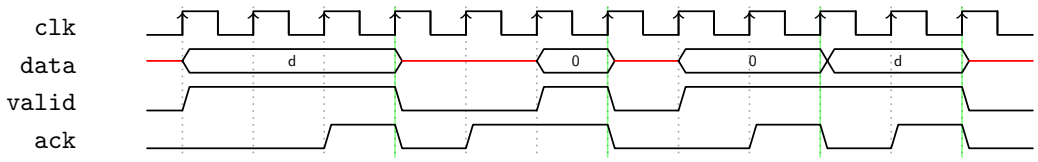
   Celem zadania jest realizacja układu cyfrowego, który będzie wykonywał programy w języku Brainf**k. Układ powinien mieć następujące wejścia i wyjścia:

   - `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.



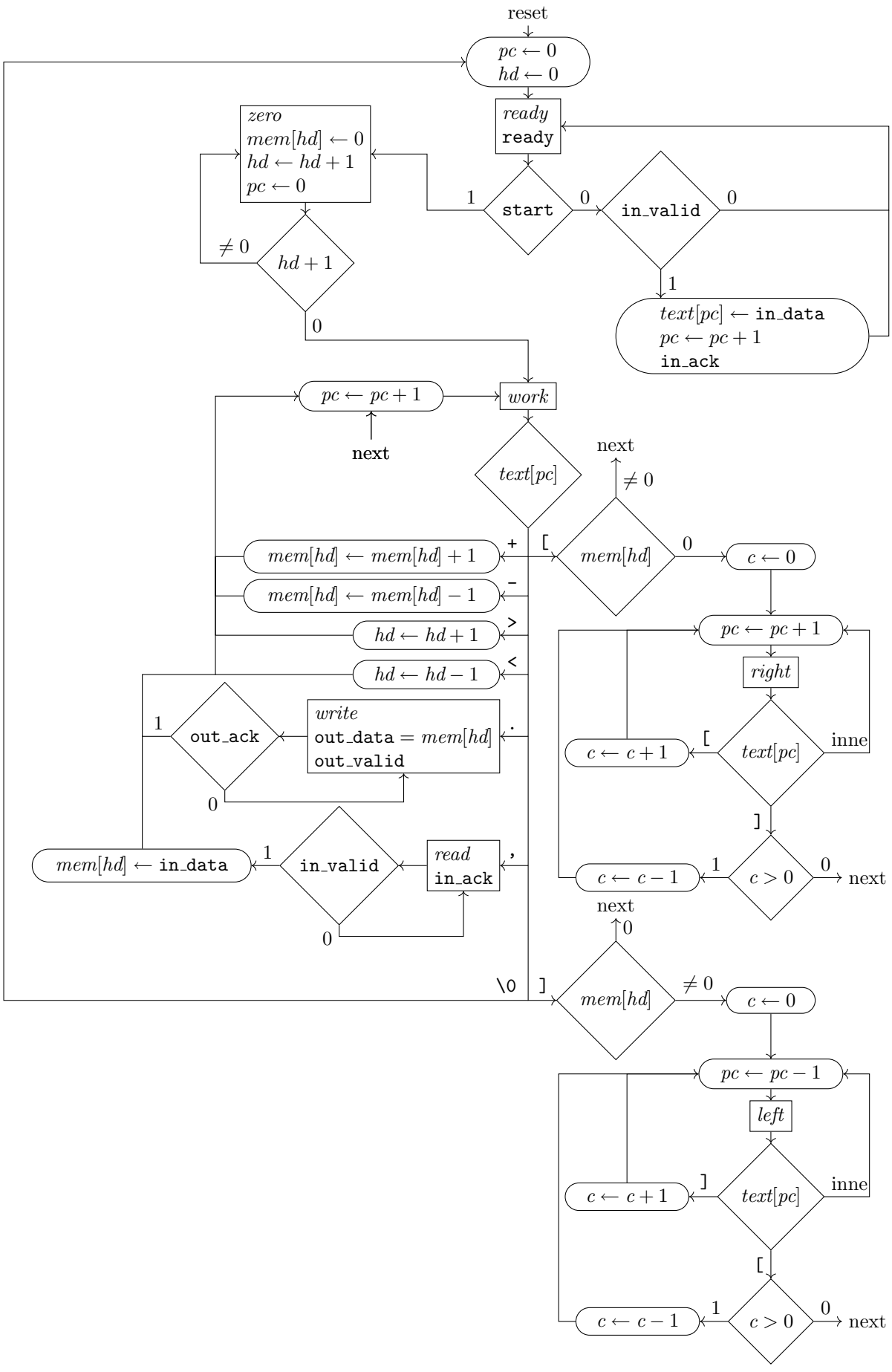
Rozwiązanie powinno zawierać co najmniej 256 bajtów pamięci kodu i 256 bajtów pamięci danych. Jeśli wskaźnik instrukcji lub głowica wyjdą poza dostępny obszar pamięci, zachowanie układu może być dowolne. Każda z dwóch pamięci powinna być pamięcią RAM z portem do odczytu (asynchronicznego) i portem do 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 wykorzystywać metodę opisaną powyżej. Aby wygenerować obraz pamięci w formacie zrozumiałym dla SystemVerilogа, 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.


reset
$pc \leftarrow 0$
$hd \leftarrow 0$
ready
`ready`
zero
$mem[hd] \leftarrow 0$
$hd \leftarrow hd + 1$
$pc \leftarrow 0$
1
`start`
0
`in_valid`
0
1
$\neq 0$
$hd + 1$
0
$text[pc] \leftarrow$ `in_data`
$pc \leftarrow pc + 1$
`in_ack`
$pc \leftarrow pc + 1$
work
next
$text[pc]$
next
$\neq 0$
+
[
$mem[hd] \leftarrow mem[hd] + 1$
$mem[hd]$
0
$c \leftarrow 0$
-
$mem[hd] \leftarrow mem[hd] - 1$
>
$hd \leftarrow hd + 1$
$pc \leftarrow pc + 1$
<
$hd \leftarrow hd - 1$
right
1
`out_ack`
write
`out_data` $= mem[hd]$
`out_valid`
.
$c \leftarrow c + 1$
[
$text[pc]$
inne
0
]
$mem[hd] \leftarrow$ `in_data`
1
`in_valid`
read
`in_ack`
,
$c \leftarrow c - 1$
1
$c > 0$
0
next
0
next
\0
]
$mem[hd]$
$\neq 0$
$c \leftarrow 0$
$pc \leftarrow pc - 1$
left
$c \leftarrow c + 1$
]
$text[pc]$
inne
[
$c \leftarrow c - 1$
1
$c > 0$
0
next