Politechnika Warszawska
Wydział Elektroniki I Technik Informacyjnych
Instytut Telekomunikacji

Systemy Komputerowe

Laboratorium

Ćwiczenie 2
Temat: Tworzenie i integracja peryferii z wykorzystaniem języku opisu sprzętu Verilog dla układu SoC emulowanego przez program QEMU.

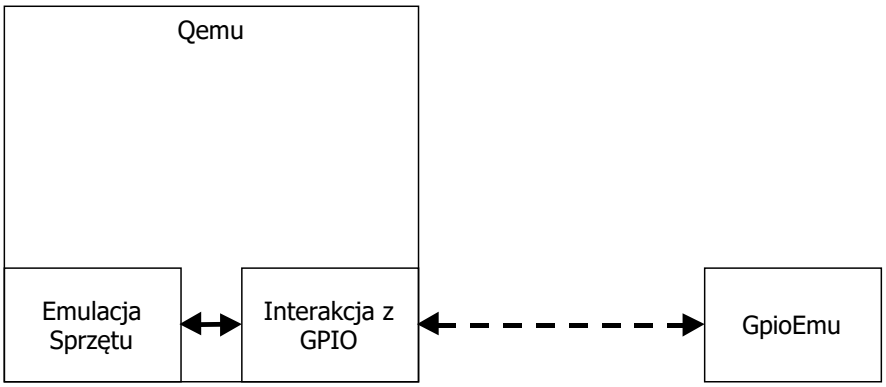
Warunki wstępne:
Zapoznać się z:
Językiem Verilog.
Budową i działaniem dekoderów adresów w systemach mikroprocesorywch.

1. Wprowadzenie

W obecnych czasach tworzenie nowych procesorów jak i systemów wielokrotnie realizowane jest z wykorzystaniem testowych środowisk zaopatrzonych w struktury FPGA. Te właśnie struktury rekonfiguruje się tak aby działały jak tworzony nowy procesor lub cały system (SoC),
W ramach tych zajęć laboratoryjnych, tworzony i badany będzie jeden z kluczowych elementów systemu jakim jest dekoder adresów. Posiadając zaprojektowany procesor konstruktor z reguły dołącza do niego pamięci i peryferia. Aby elementy te działały spójnie i z punktu widzenia procesora nie zakłócały swojego działania, wspierane przez nie przestrzenie muszą być rozmieszczone tak, aby się nie nakładały na siebie. O nie nakładanie się na siebie dba właśnie dekoder adresów.
Dla potrzeb tego ćwiczenia nie przygotowano dedykowanych zdalnych komputerów, jedyne co jest potrzebne to program Iverilog[http://iverilog.icarus.com], dla wygody można zatem całe ćwiczenie realizować na swoim prywatnym komputerze. A można go zainstalować na swoim komputerze po pobraniu paczki ze strony:
a)dla systemu Windows: plik iverilog-v11-20200824-x64_setup.exe do pobrania ze strony:
http://bleyer.org/icarus,
b)dla systemu Linux: zgodnie z instrukcją na stronie:
https://iverilog.fandom.com/wiki/Installation_Guide

2. Architektura tworzonego systemu

Zajęcia laboratoryjne i projektowe realizowane są z wykorzystaniem emulatora QEMU. To on wykonuje instrukcje procesora RISC-V oraz udostępnia mu pamięć i peryferia. Modyfikacja wbudowanego w niego dekodera adresów jest zadaniem dość skomplikowanym. Aby ułatwić zadanie laboratoryjne zmodyfikowano emulator QEMU tak, by jego podprzestrzeń pamięciowa była wyprowadzona aby możliwe było tworzenie dekoder adresów wyłącznie dla tego podzbioru adresowego. Podzbiór ten obejmuje 2^16 lokacji bajtowych.
W ramach prac laboratoryjnych tworzony będzie w języku opisu sprzętu: Verilog, kod tego właśnie dekodera i prostych peryferii. Utworzone pliki w formacie Verilog będą następnie kompilowane do postaci odpowiedniej dla potrzeb linkowania ich z resztą kodu emulatora QEMU. W wyniku tych operacji powstanie nowy nieco zmodyfikowany emulator QEMU.
Dekoder i peryferia mają jako całość zadanie:
-zrealizować nowe peryferia (tzw. tworząc zestaw rejestrów zatrzaskowych, emulować nowe GPIO procesora),
-zdekodować podawane adresy, tak aby emulowane CPU mogło do tych nowych peryferii się odwołać.



Realna komunikacja z użytkownikiem jest możliwa za pomocą programu *GpioEmu*. Jest to niezależny program, który nasłuchuje na porcie 9001 protokołu TCP i zgodnie z wewnętrznym protokołem wizualizuje stan peryferii GPIO wyjściowych lub udostępnia stan peryferii GPIO wejściowych.

2.1. Interfejsy interakcji z GPIO

Moduł interakcji z GPIO ma za zadanie w emulowanej przestrzeni pamięciowej odwzorować zestaw rejestrów GPIO. Zakładane jest, że przygotowana dla potrzeb tego laboratorium infrastruktura QEMU wydziela przestrzeń 2^16 adresów dla nowych peryferii. Dodatkowo tak dostosowany emulator QEMU definiuje specjalizowaną magistralę do komunikacji z nowymi peryferiami. W języku Verilog trzeba zdefiniować tę magistralę w następujący sposób:

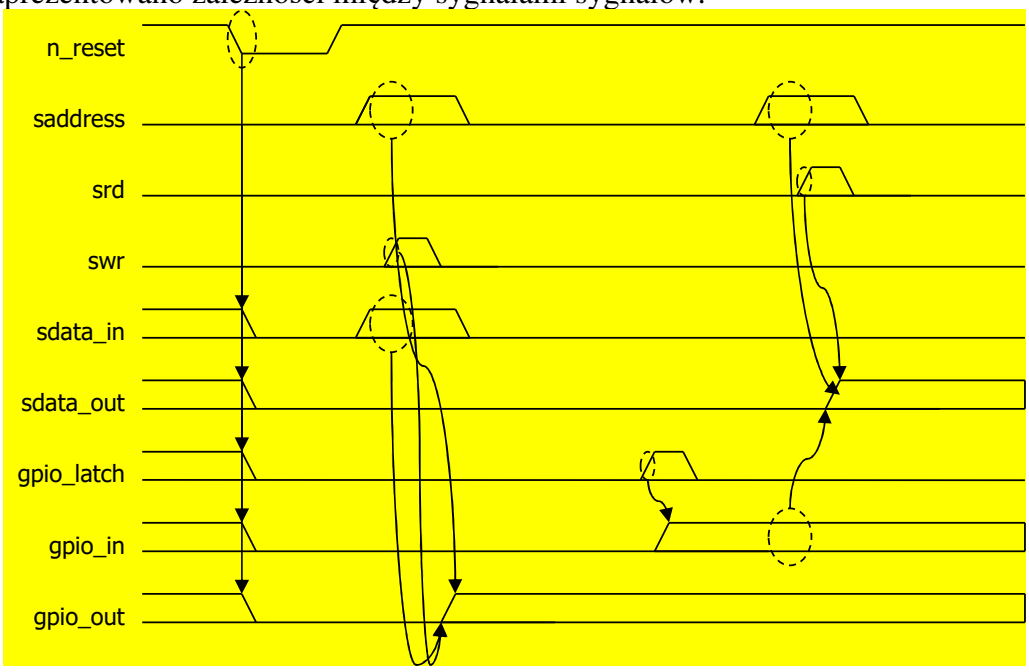
Nazwa	Opis
`n_reset`	Sygnał zerowania stanu peryferii
`saddress[15...0]`	Magistrala adresowa
`sdata_in[31...0]`	Magistrala danych – wejście
`sdata_out[31...0]`	Magistrala danych – wyjście
`srd`	Sygnał żądania wyprowadzenia informacji spod adresu wskazanego przez `saddress` na magistralę `sdata`
`swr`	Sygnał zapisu danych z magistrali `sdata` pod adres wskazany przez `saddress`
`gpio_in[31...0]`	Stan wejść GPIO (ustawianych przez `GpioEmu`)
`gpio_latch`	Sygnał zatrzaśnięcia przez peryferia (wejściowe) ich stanu w rej. wewnętrznym `GpioEmu` zawiązanym z `gpio_in`
`gpio_out[31...0]`	Stan wyjść GPIO (pokazywanych na `GpioEmu`)
`clk`	Zegar 1KHz (dla potrzeb operacji realizowanych przez GpioEmu), zegar nie jest zsynchronizowany z zegarem emulowanego CPU

Zatem szkielet pliku Verilog dla tworzonego komponentu powinien wyglądać w następujący sposób:

```
module gpioemu(n_reset, saddress[15:0], srd, swr,
               sdata_in[31:0], sdata_out[31:0],
               gpio_in[31:0], gpio_latch,
               gpio_out[31:0],
               clk,
               gpio_in_s_insp[31:0]);

      input             clk;
      input             n_reset;

      input [15:0]      saddress;
      input             srd;
      input             swr;
      input [31:0]      sdata_in;
      output[31:0]      sdata_out;
      reg [31:0]        sdata_out;
```

```
      input [31:0]      gpio_in;
      reg [31:0]        gpio_in_s;
      input             gpio_latch;
      output[31:0]      gpio_in_s_insp;

      output[31:0]      gpio_out;
      reg [31:0]        gpio_out;
      reg [31:0]        gpio_out_s;

//odpowiedz na reset (aktywowane przejściem 0-1)
      always @(negedge n_reset)
            gpio_in_s <= 0;
            ...
...
endmodule
```

W powyższym kodzie najistotniejsze jest zachowanie nazewnictwa i znaczenia połączeń interfejsowych (tabela powyżej). Zgodność ta umożliwi systemowi integracji poprawnie połączenie opisu zapisanego w języku Verilog z implementacją emulatora QEMU.
Aby dokładniej zrozumieć działanie komponentu, może być konieczne zapoznanie się z przebiegami czasowymi na tej magistrali. Warto tu jednak nadmienić, iż emulator QEMU nie wymaga czasowo krytycznych odpowiedzi – co wynika z metody integracji tworzonego tu pliku w języku Verilog, która upraszcza te zagadnienia. Jednakże gdyby chcieć powstały plik opisu komponentu wykorzystać do tworzenia tzw. SoC takie zależności należałoby uwzględnić.
Poniżej zaprezentowano zależności między sygnałami sygnałów.



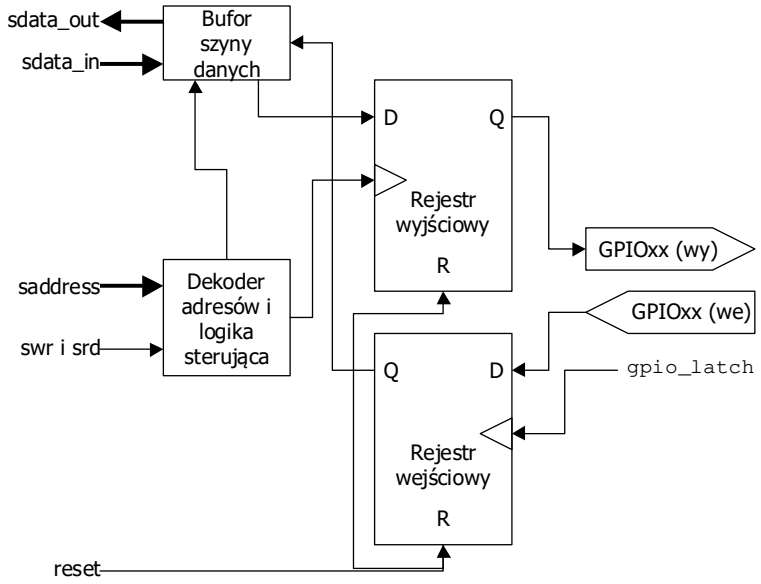
Widać na powyższym rysunku, że zapis do rejestru `gpio_out` jest związany z adresem przekazanym na `saddress`, oraz sygnałem zatrzaśnięcia `swr`. Zapis do rejestru `gpio_in` jest związany wyłącznie z sygnalem `gpio_latch` – geenrowany przez zmianę linii wejściowych. Stan linni `sdata_out` zależy od podanego adresu `saddress` i uaktywnienia sygnału `srd`. Zauważyć należy, że sygnał `sdata_out` utrzymuje swoją wartość niemal w nieskończoność – dokładnie: do następnego żadania odczytu danych. Wynika to z faktu, iż w rozpatrywanym tu modelu nie uwzględniono konieczności zwolnienia magistrali danych po zakończeniu każdej z transakcji wymiany danych między `GpioEmu` a CPU.

2.2. Wymagane interfejsy – emulacji nowych GPIO

Budowany układ peryferyjny może wpływać na otoczenie. Finalnie jest to realizowane przez tzw. GPIO (ang. General - Purpose Input / Output). Elementy GPIO w typowym mikroprocesorze / mikrokontrolerze są z reguły bezpośrednie połączenie z końcówkami tego elementu. Nie wchodząc jednak w realizację fizyczną tego aspektu, w środowisku emulatora QEMU nie przewidziano bezpośrednio takowej możliwości. Dla potrzeb tego przedmiotu utworzono wersję tego emulatora tak, aby możliwa była zarówno wizualizacja stanu elementów GPIO jak również możliwe były zmiany stanów wejść GPIO niejako z zewnątrz. Obie te funkcjonalności realizuje specjalnie utworzony graficzny program `GpioEmuConsole`. Program ten przedstawia użytkownikowi jedno okienko z szeregiem elementów GPIO. Stan każdego z elementów GPIO można zmieniać za pomocą tzw. chechbox, a za pomocą okręgu podświetlanego kolorem zielonym użytkownik jest informowany o faktycznym jego stanie. Z programem tym podczas swojej pracy emulator QEMU może się połączyć, a następnie replikować stan elementów GPIO tak, aby emulowana przez QEMU aplikacja miała namiastkę współpracy z peryferiami typu GPIO.

W tym ćwiczeniu jak już wspomniano należy zbudować dekoder, bufor szyny danych oraz moduł peryferyjnych rejestrów. Za pomocą tych rejestrów możliwa będzie realna komunikacja z użytkownikiem poprzez program `GpioEmuConsole`.

Z punktu widzenia emulacji pojedynczych elementów GPIO wyjścia Q rejestrów będą zgodnie z podaną indywidualnie specyfikacją połączone z emulowanymi elementami GPIO – przypadek kierunku „wyjście”, lub z wejściami D drugiego zestawu rejestrów zatrzaskujących – przypadek kierunek „wejściowy”. Całość pokazuje rysunek:



Bufor szyny danych ma za zadanie odseparować wyjście Q rejestru wyjściowego tak, aby stan tego wyjścia pojawiał się na magistrali danych tylko wtedy gdy CPU chce odczytać stan peryferii, których stan utrzymuje ten rejestr. Bufor szyny danych pomaga także doprowadzić odpowiednie informacje z tej szyny tylko gdy CPU chce zapisać coś do rejestru wyjściowego. Przypisanie GPIOxx do właściwego bitu szyny danych, będzie podane w specyfikacji przekazanej danej grupie laboratoryjnej.

Drugim elementem do zdefiniowania to dekoder adresów i logika sterująca. Jest to najbardziej skomplikowana część do zaprojektowania. Jej podstawowym zadaniem jest generowanie sygnału zapisu do rejestrów.

W przypadku zapisu do regestru wyjściowego – na podstawie stanu magistrali adresowej, który to odpowiada adresowi przydzielonemu modułowi GPIO (w przestrzeni 2^16) oraz sygnałowi /WR, generowany jest sygnał zatrzaśnięcia dla tego rejestru. W przypadku zapisu do regestru wejściowego – na podstawie stanu magistrali adresowej, który to odpowiada adresowi

przydzielonemu modułowi GPIO oraz sygnałowi /RD, generowany jest sygnał zatrzaśnięcia dla tego rejestru.
Przypisanie adresów pod jakimi mają być powyższe operacje wykonywane, będzie podane w specyfikacji przekazanej danej grupie laboratoryjnej.

3. Tworzenie i weryfikacja działania komponentu współpracy z peryferiami

Tworzony komponent należy zapisać w pliku Verilog np.: *GpioEmu.v*. Po utworzeniu można go poddać kompilacji (z linii poleceń):

```
iverilog –o GpioEmu.vvp GpioEmu.v GpioEmu_tb.v
```

Gdzie:

```
GpioEmu_tb.v      – plik z opisem testowania
GpioEmu.vvp       – plik wynikowy
```

W powyższych operacjach brakuje jednak pliku `GpioEmu_tb.v`. Jest on odpowiedzialny z wykonanie właściwych testów i jego treść także powinna być zapisana w języku Verilog. Pracę można zacząć od następującego szkieletu:

```
module GpioEmu_tb;
initial begin
    $dumpfile("GpioEmu.vcd");
    $dumpvars(0, test);
end
initial begin
      # 5   reset = 1;
      # 10  S_Adddress[]= ...;
      # 11  S_Data[]= ...;
      # 12  S_WR=...;
      ...
      # 1000 $finish;
end
...
endmodule
```

Mając wszystkie niezbędne plik wejściowe po przeprowadzeniu poprawnej kompilacji (powyżej), aby przetestować działanie modelu opisanego w *GpioEmu.v* trzeba wydać polecenie:

```
vvp GpioEmu.vvp
```

Plik ten jest wewnętrzną formą zapisu działania generowanego przez tzw. *Icarus Verilog runtime engine*. Aby obejrzeć wynik jej działania (w szczególności wykresy czasowe) można skorzystać z programu GtkWave:

```
gtkwave.exe GpioEmu.vcd gtk_conf.gtkw
```

4. Zadanie finalne

Wymagane w tym ćwiczeniu laboratoryjnym jest aby przeprowadzić testowanie utworzonego pliku *GpioEmu.v*, zarówno pod względem testów fundamentalnych, jak i funkcjonalnych. Weryfikując czy zbudowany tak komponent działa poprawnie.
Podczas testów załóż, że:
-adres nowych peryferii będzie w zakresie: 0xXXXX-0xYYYY,
-GPIO_OUT są podłączone finalnie do bitów: 1, 2, 4, 31 magistrali danych,
-GPIO_IN są podłączone finalnie do bitów: 2, 3, 7, 31 magistrali danych.