Politechnika Warszawska
Wydział Elektroniki I Technik Informacyjnych
Instytut Telekomunikacji

Systemy Komputerowe (SYKOM)

Laboratorium

Ćwiczenie 3
Temat: Wytworzenie specjalizowanego układu SoC emulowanego przez program QEMU z wykorzystaniem oprogramowania użytkowego bez systemu operacyjnego.

Warunki wstępne:
Zapoznać się z:
Materiałami dla laboratoriów 1 i 2.

1. Przygotowanie środowiska do pracy

Dla celów ćwiczeń laboratoryjnych przygotowano podobnie, jak dla lab.1 zestaw maszyn wirtualnych działających pod systemem Linux Debian. Ponownie przypomina się, że należy pamiętać, iż po zarezerwowanym dla siebie slocie czasowym, stan każdej z maszyn jest przywracany do stanu oryginalnego a wszystkie dane w niej składowane ulegają usunięciu. Aby jednak student miał możliwość pracy ze swoimi plikami – efektami swojej pracy, należy korzystać z przygotowanego indywidualnego i dedykowane każdemu studentowi repozytorium GIT.

2. Utworzenie personalizowanego emulatora QEMU

Emulator QEMU domyślnie wspiera rodzinę procesorów Risc-V, lecz nie potrafi emulować peryferii mający kontakt z użytkownikiem, poza konsolą. Dzięki dodaniu dla potrzeb tego przedmiotu, specjalnego mechanizmu, taka emulacja w określonym zakresie jest możliwa.
Podczas emulacji programów QEMU wykrywszy odwołania do przestrzeni GPIO (część obszaru adresów z przestrzeni obsługiwanej przez zasób `„sykt”` – opisanego w dokumentach dla lab.1) przekazuje to odwołania do modułu utworzonego w języku Verilog. Moduł ten odtąd będzie nazywać się `GpioEmu`, a podstawy jego budowy poznane były w ramach zajęć lab.2. W laboratorium 3 moduł ten po odpowiedniej obróbce stanie się integralną częścią emulatora QEMU.
Z punktu widzenia wnętrza emulatora QEMU moduł `GpioEmu` z jednej swojej strony styka się z emulowanym procesorem a z drugiej łączy się z tzw. zakończeniem emulacji – aplikacją `GpioConsole` (dostępna w domyślnych ścieżkach udostępnionych maszyn wirtualnych). Aplikacja ta zapewnia kontakt z otoczeniem i umożliwia jej użytkownikowi bezpośrednie ustawianie stanów pinów wejściowych i inspekcje stanu pinów wyjściowych modułu `GpioEmu`.
Aby dokonać obróbkę QEMU i scalić go z opisem sprzętu zapisanym w module `GpioEmu` w języku Verilog dla potrzeb laboratorium przygotowano specjalne narzędzie: `makeQemuGpioEmu`. Na podstawie źródeł emulatora oraz dostarczonego (np.: przez studenta) pliku w języku Verilog a opisującego modułu `GpioEmu`, potrafi on zbudować personalizowany emulator QEMU. Użycie tego narzędzia polega na wywołaniu:

```
makeQemuGpioEmu gpioemu.v
```

Gdzie:
`gpioemu.v` - plik opisu działania modułu GpioeEmu (uwaga: ze względu na składnie języka Verilog, plik tu podany musi mieć nazwę `gpioemu.v`)

W wyniku poprawnego działania programu, w katalogu gdzie wydano powyższe wywołanie, powinien pojawić się plik o nazwie, np.: `qemu-system-riscv32-sykt`. W tym ćwiczeniu laboratoryjnym zakładane jest m.in. wykonanie specjalizowanego modułu `GpioEmu` którego specyfikacja przesyłana jest każdemu studentowi niezależnie drogą elektroniczną.

Na tym etapie warto zauważyć, że moduł GpioEmu do swojej pracy może potrzebować sygnał zegarowy. W obecnej wersji modyfikowanego emulatora QEMU sygnał zegarowy, nie jest związany ani w zakresie częstotliwości ani w zakresie fazy z zegarem CPU emulowanym przez QEMU. Stosowany przez moduł sygnał zegarowy charakteryzuje jedynie częstotliwość równa w przybliżeniu 100Hz. Poleganie na założeniu że pewne operacje moduł GpioEmu wykona w określonym czasie jest zatem błędne i może kończyć się problemami. W pewnych przypadkach gdy plik_opisu.v opisujący ten moduł będzie zawierał bardziej skomplikowany automat, wymagający do wypracowania wyniku swojej pracy wielu cykli sygnału zegarowego (np.: algorytm znajdowania liczb pierwszych) może stać się problematycznym określenie przez emulowany CPU (czyli wykonywany przez niego program) kiedy operacje zlecone powyższemu automatowi zostaną zakończone.
Można jednak powyższą uciążliwość obejść - wystarczy, że konstruowany moduł GpioEmu zawierać będzie odpowiednie wsparcie dla badania stanu jego wewnętrznych operacji. Wsparcie takie musi zapewnić możliwość emulowanemu programowi „wyciągnięcia" z ustalonych rejestrów wewnętrznych tego modułu wartości określającej stan wewnętrznych operacji. Technika ta jest dobrze znana z wielu systemów komputerowych. Zakłada się tam, że wśród tzw. rejestrów statusowych, stan jednego z bitów – z reguły o nazwie BUSY – będzie stanowił informacje dla CPU czy zlecony proces został zakończony i czy można przejść do pobierania wyników zleconej operacji i traktować je jako poprawne. Dla przykładu realizacja takich funkcji od strony emulowanego CPU w języku C mogła być następująca:

```
1. ...

2. #define SYKT_GPIO_ADDR_SPACE         (0xYYYYYYYY)

3. #define SYKT_GPIO_CTRL_ADDR          (SYKT_GPIO_ADDR_SPACE+0xCCCCCCCC)
4. #define SYKT_GPIO_ARG1_ADDR          (SYKT_GPIO_ADDR_SPACE+0xAAAAAAAA)
5. ...
6. #define SYKT_GPIO_RESULT_ADDR        (SYKT_GPIO_ADDR_SPACE+0xRRRRRRRR)
7. #define SYKT_GPIO_STATUS_ADDR        (SYKT_GPIO_ADDR_SPACE+0xSSSSSSSS)

8. #define BUSY_BIT                     (0x00000001)
9. #define MY_GET_PRIME_START_COMMAND   (0x80000000)

10.uint32_t my_component_get_prime_numbers(uint32_t n){
11.    RAW_SPACE(SYKT_GPIO_ARG1_ADDR)=(uint32_t) n;    //numer szukanej liczby pierwszej
12.    RAW_SPACE(SYKT_GPIO_CTRL_ADDR)=MY_GET_PRIME_START_COMMAND;
13.    for(;;){                                        //czekamy na wynik
14.          if((RAW_SPACE(SYKT_GPIO_STATUS_ADDR) & BUSY_BIT)==0) //Czy BUSY równe 0
15.                break;
16.    }
17.    return RAW_SPACE(SYKT_GPIO_RESULT_ADDR);
18. }
```

W powyższym przykładzie zakłada się, że argumenty przekazywane są za pomocą makra RAW_SPACE() do modułu GpioEmu, podobnie wyniki pobierane w identyczny sposób, natomiast całą pracę wykonuje moduł. Dla przypomnienia RAW_SPACE zostało opisane w materiałach do lab.1.
Na powyższym listingu widać przykład współdziałania z modułem GpioEmu który wewnętrznie oblicza liczbę pierwszą o numerze przekazanym w argumencie 'n' w wywołaniu funkcji my_component_get_prime_numbers(). W liniach 10-18 widzimy implementację funkcji łączącej komponent Verilog z programem w C. W liniach 11-12 mamy funkcje przekazania argumentu 'n' i uruchomienia procesu wewnętrznego tego modułu (założono tu, że wpis do SYKT_GPIO_CTRL_ADDR wartości MY_GET_PRIME_START_COMMAND – uruchamia pracę tego modułu).
Gdy moduł GpioEmu rozpocznie swoją pracę, będzie ją wykonywał przez pewien czas - proces ten może zająć pewien czas wymagając określonej liczby cykli zegarowych. Z natury algorytmu szukania liczb pierwszych wynika realizowanego przez ten moduł, że nie da się zatem przewidzieć kiedy proces zostanie zakończony. Jednak dzięki liniom 13-16 funkcja zostanie zablokowana do czasu gdy moduł GpioEmu zakończy swoje wewnętrzne operacje. Implementacja linii 13-16 to nieskończona pętla która zostanie przerwana gdy warunek w linii 14 zostanie spełniony. Warunek

w tej linii bada stan rejestru `SYKT_GPIO_STATUS_ADDR` a dokładniej jego bitu `BUSY` i gdy bit ten stanie się równy 0 - zakończy się działanie pętli.
Proszę zauważyć, że w powyższym kodzie adresy rejestrów zostały zapisane symbolicznie jako: `0xYYYYYYYY` oraz modyfikatory: `0xCCCCCCCC`, `0xAAAAAAAA`, `0xRRRRRRRR`, `0xSSSSSSSS`. Nie jest to błąd, dokładna wartość `0xYYYYYYYY` wynika z inspekcji narzędziem DTC związanej z zasobem 'sykt'. Natomiast pozostałe wartości zostaną przekazane w ramach indywidualnej specyfikacji zadania dla laboratorium 3.
Proszę także pamiętać, że powyższy kod nie jest pełen – na co wskazuje linia 1 (wypełniona za pomocą znaków „…") i trzeba w jej miejsce wstawić odpowiedni kod, zgodnie z informacjami podanym w lab.1.

3. Przygotowanie aplikacji testowej

Aby mieć pewność, że nowy system emulowany przez QEMU działa poprawnie, należy zbadać go przygotowując aplikację lub wręcz zestaw testujący go. W przykładzie tu omawianym, będzie to tylko jedna aplikacja, której zadaniem będzie uruchomienie `my_component_get_prime_numbers()`. Odpowiedni kod mógłby wyglądać w następujący sposób:

```
1. #define uint32_t   unsigned long
2. ...
3. int main() {
4.     uint32_t n=12;            //numer szukanej liczby pierwszej
5.     uint32_t prime;
6.     prime=my_component_get_prime_numbers(n);
7.     my_put_uint32(prime);
8.     my_simulation_exit(0);
9.     return 0;
10. }
```

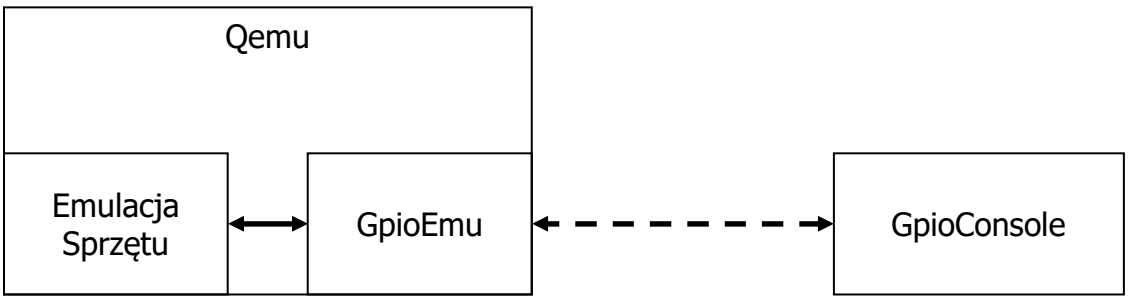
Implementacja powyższego kodu nie jest skomplikowana, wyjaśnienia wymaga linia 7. Widać w niej wywołanie funkcji `my_put_uint32()`, jest to prosta funkcja której zadaniem jest wypisać na konsoli QEMU w postaci tekstowej - na podstawie określonej w specyfikacji zadania notacji (HEX, DEC, OCT, BIN) - wartość przekazaną jako argument o typie uint32_t a której implementacja korzysta z implementacji `my_putchar()` omówionej w ramach laboratorium numer 1.
Po utworzeniu implementacji aplikacji testowej należy zgodnie z opisem przedstawionym dla laboratorium 1 dokonać jej kompilacji. W dalszej części tego dokumentu zakłada się, że po kompilacji aplikacja testowa będzie miała także nazwę: sykt.

4. Weryfikacja działania utworzonego systemu

Na tym etapie zakładane jest, iż przygotowana w pkt.3 aplikacja testowa potwierdzi poprawność działania systemu. Aby sprawdzić jej działanie trzeba uruchomić ją w nowo utworzonym (w pkt.2) emulatorze. Oto właściwe wywołanie emulatora:

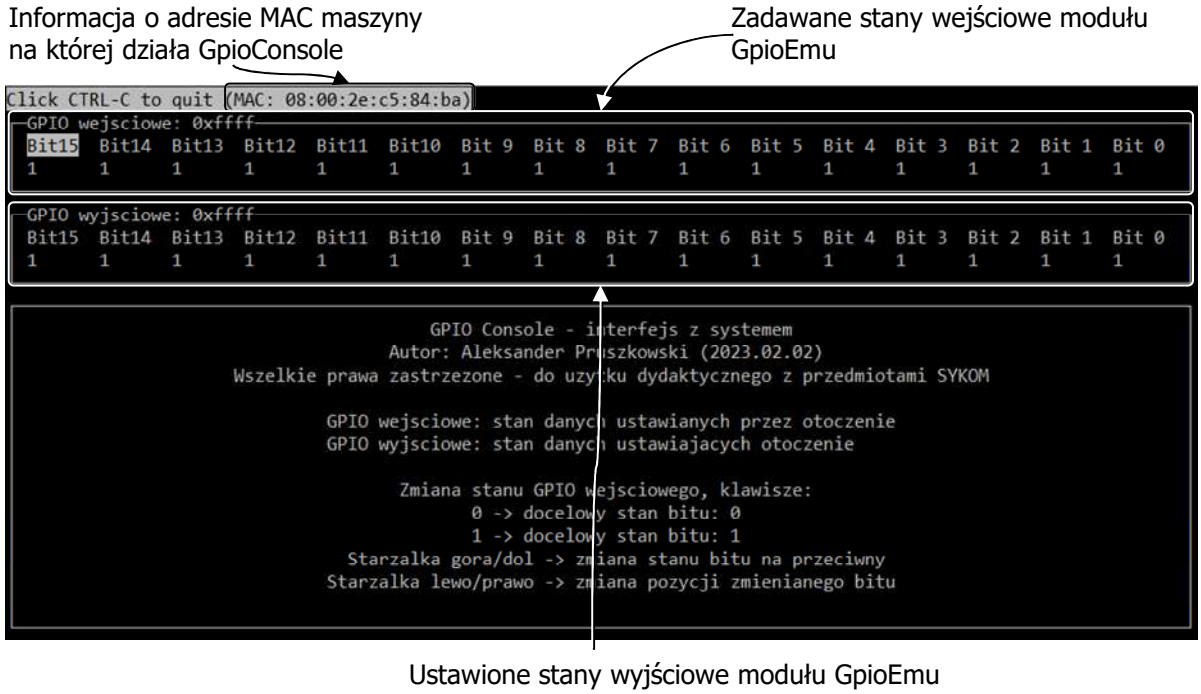
```
./qemu-system-riscv32-sykt -machine sykt -bios none -m 128M -kernel ./sykt
```

Zakłada się w powyższym wywołaniu, że nie korzystamy z programu debugger GDB – gdyby była taka potrzeba, odpowiednie informacje można znaleźć w dokumencie wprowadzającym do laboratorium 1.
Jak wcześniej wspominano zmodyfikowany emulator QEMU może komunikować się bezpośrednio z użytkownikiem. Realizuje się to poprzez program użytkowy o nazwie `GpioConsole`. Program ten jest aplikacją uruchamianą z linii poleceń na przydzielonej maszynie wirtualnej, łącząc się z nią w osobnym terminalu (np.: z pomocą PuTTY lub ssh). Dla przypomnienia `GpioConsole` podczas swojego działania łączy się z emulatorem QEMU w następujący sposób:



Połączenie między QEMU i `GpioConsole` realizuje lokalna sieć IP za pomocą specjalnie utworzonego protokołu aplikacyjnego bazującego na datagramach UDP. Protokół pozwala przekazywać stan emulowanych peryferii styku z użytkownikiem do emulatora QEMU. Należy jednak pamiętać, że na poziom dokładności przekazywania tych stanów mogą negatywnie wpływać: szybkość połączenia między emulatorem QEMU a aplikacją `GpioConsole` oraz skończona wydajność komputera na jakich oba te elementy działają.

Uruchomienie tej aplikacji nie wymaga żadnych argumentów wywołania, wystarczy wydać polecenie: `GpioConsole` i program pojawi się na ekranie terminala w którym ten program uruchomiono. Rysunek poniżej pokazuje co powinno się ukazać:



Obsługa tej aplikacji jest opisana w jej największym okienku. Relacja czasowa momentów uruchomienia programów `GpioConsole` i QEMU nie jest krytyczna. Ważne jest jednak że dopiero po uruchomieniu `GpioConsole`, dane w polu „GPIO wyjściowe" zostaną pokazane – program ten nie daje także możliwości obejrzenia danych historycznych.

Aby jednak łatwiej analizować stan modułu `GpioEmu` w programie `GpioConsole` dodano funkcjonalność logowania stanu wyjść i wejść tego modułu. Stan ten jest zapisywany do pliku `gpio_log.txt` który jest tworzony w katalogu uruchomienia programu `GpioConsole`. Format danych zapisanych w tym pliku jest dość prosty:

```
[znacznik czasu] gpio_in: wartość, gpio_out: wartość
```

Dla przykładu treść tego pliku mogłaby wyglądać następująco:

```
...
[3665953657] gpio_in: 0x00, gpio_out: 0x7f
[3665953763] gpio_in: 0x00, gpio_out: 0xff
[3665954294] gpio_in: 0x00, gpio_out: 0x7f
[3665954335] gpio_in: 0x00, gpio_out: 0xff
...
```

Podsumowując program `GpioConsole` daje możliwość wpływania na moduł `GpioEmu` jednak trzeba pamiętać, że interakcja ta wymaga przesyłania odpowiedniej treści poprzez datagramy UDP i co więcej - nie ma gwarancji, że takowe informacje (w szczególności gdy będzie ich zbyt dużo) nie zostaną zgubione.