Systemy wbudowane - lista 2 (lab)

Błażej Wróbel, 250070, 3. rok, W11, informatyka

Zadanie 1.

- ➔ Opis działania bramek:

  W pliku *lab2-gates.vhd* opisujemy następujące bramki logiczne : **AND**, **OR**, **XOR** i **NOT**. Za pomocą słów kluczowych entity oraz port specyfikujemy wejścia i wyjścia bramki (podajemy czy dany 'port' jest wejściem lub wyjściem oraz określamy jaki typ wartości będzie się tam pojawiać). Następnie za pomocą słowa kluczowego architecture tworzymy opis działania bramki np. na wyjście **Z** podajemy koniunkcje wartości logicznych pojawiających się na wejściach **A** i **B** (definicja zachowania bramki **AND**. Dla innych bramek postępujemy analogicznie).
- ➔ Opis działania multipleksera:

  W tym przypadku również specyfikujemy wejścia i wyjścia (na początku). Z definicji zawartej w ciele konstrukcji entity widać, że multiplekser będzie mieć 4 wejścia, 2 selektory i jedno wyjście. Następnie za pomocą słów kluczowych architecture oraz process odpowiednio opisujemy zachowanie układu i uruchamiamy jego symulacje. Nasz multiplekser działa następująco:
  - ★ Jeśli **s0** = 0 i **s1** = 0 , to **sel** = 0 i wtedy wartość na **a** jest kierowana na wyjście **x.**
  - ★ Jeśli **s0** = 1 i **s1** = 1 , to **sel** = 1 i wtedy wartość na **b** jest kierowana na wyjście **x**.
  - ★ Jeśli **s0** = 0 i **s1** = 1 , to **sel** = 2 i wtedy wartość na **c** jest kierowana na wyjście **x**.
  - ★ Jeśli **s0** = 1 i **s1** = 0 , to **sel** = 3 i wtedy wartość na **d** jest kierowana na wyjście **x**.

  Całe działanie multipleksera w zależności od wartości **sel** opisujemy za pomocą wyrażenia case.

W programie wykorzystano bibliotekę IEEE.STD_LOGIC_1164 - zawiera ona definicje podstawowych typów logicznych/boolowskich (np. std_logic).

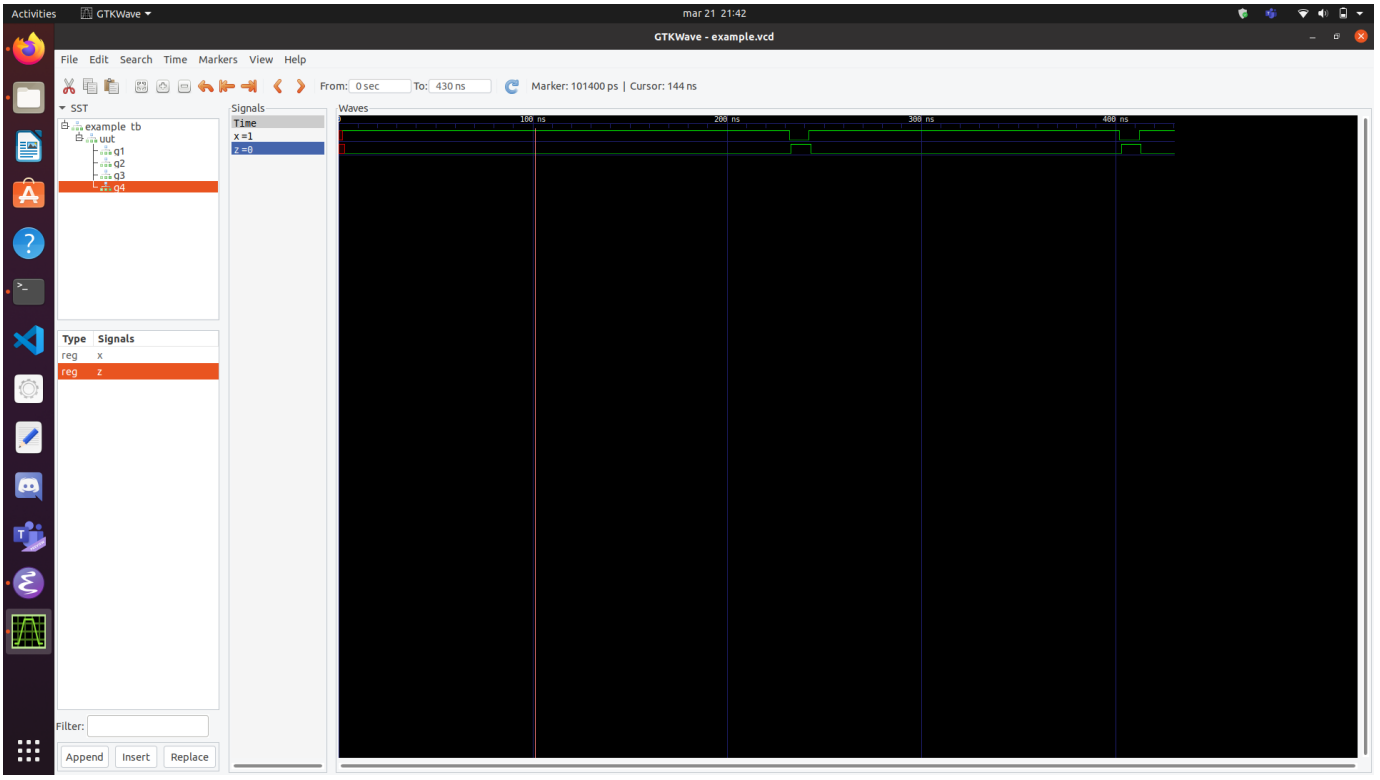
Zadanie 2.

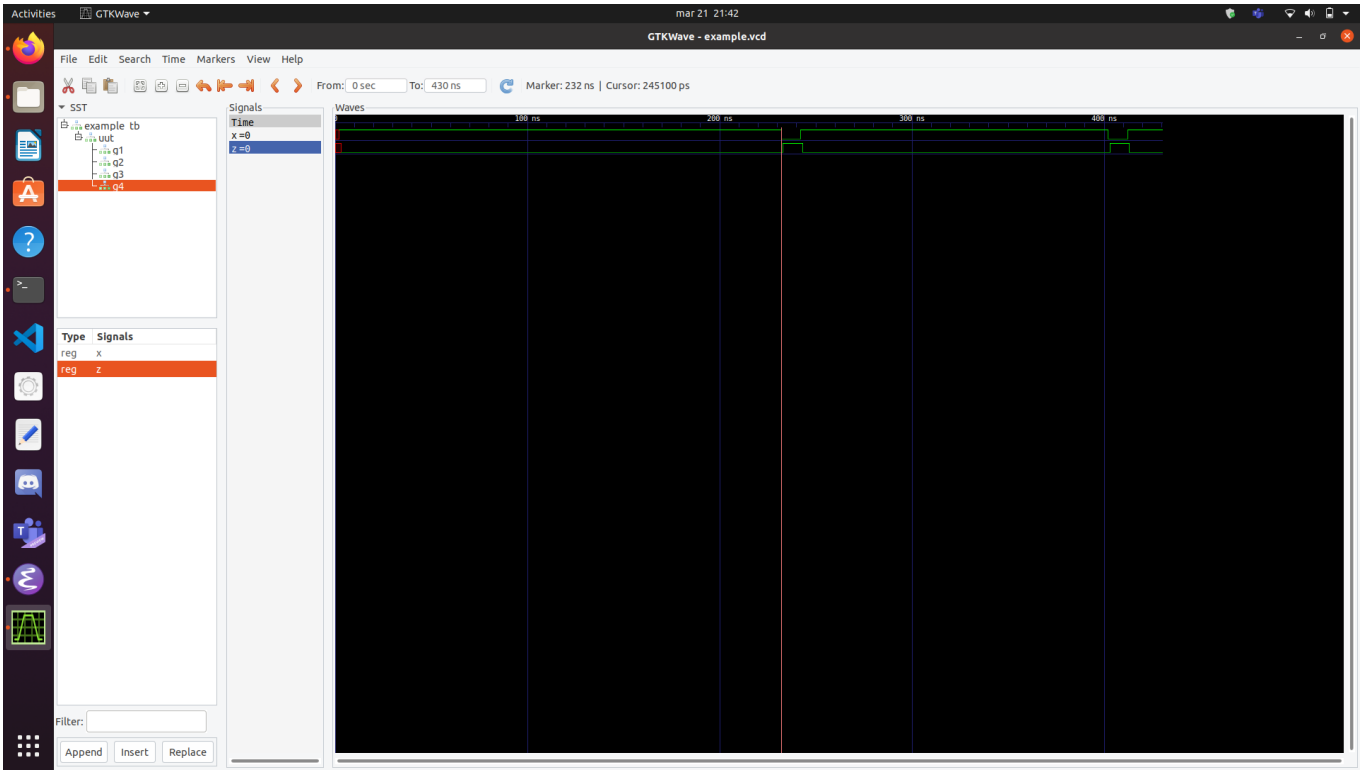
W pliku *example.vhd* używamy bramek **NOT** i **OR** jako komponentów, z których tworzymy większy układ logiczny. Najpierw (przed wykorzystaniem), deklarujemy potrzebne nam komponenty i sygnały występujące w układzie (one są potrzebne, ponieważ będą pojawiały się jako wejście dla innych bramek). Deklaracje bramek w tym programie odpowiadają rysunkowi z listy, ponieważ:

- ★ Bramka **G1** jest bramką **NOT**. Jej wejście to **a**, a wyjście to sygnał **NOT_OR**.
- ★ Bramka **G2** jest bramką **OR**. Jej wejście to **b** oraz **c**, a wyjście to sygnał **OR_OR**.
- ★ Bramka **G3** jest bramką **OR**. Jej wejście to sygnały **OR_OR** i **NOT_OR**, a wyjście to sygnał **OR_NOT**.
- ★ Bramka **G4** jest bramką **NOT**. Jej wejście to sygnał **OR_NOT**, a wyjście to **x**.

Po analizie przebiegów poszczególnych sygnałów (plik *example.vcd*) można stwierdzić, że ich przebieg pokrywa się z przebiegiem odpowiadającym im sygnałów z układu na rysunku. Na przykład, na poniższych zdjęciach znajduje się przebieg sygnału dla bramki **G4**. Z przebiegu sygnału widać, że ta bramka realizuje funkcję **NOT** (zdjęcie numer 1). Jednak są

momenty, gdy bramka nie neguje sygnału wejściowego (zdarza się to, gdy obydwa sygnały zmieniają wartość jednocześnie - zdjęcie numer 2).





Inne przebiegi (np. dla bramek **G1, G2** i całego układu) znajdują się w pliku *example.vcd*

Zadanie 3.

Nowości:

- ➢ Użycie nowej biblioteki : numeric_std, która definiuje operacje arytmetyczne na wektorach.
- ➢ Deklaracja i tworzenie komponentu **UUT** (przy deklaracji specyfikujemy wejścia/wyjścia, a później (przed utworzeniem) nadajemy portom nazwy).
- ➢ Pojawienie się typów std_logic_vector i unsigned, które są wektorami bitów z tą różnicą, że na std_logic_vector nie da się przeprowadzić żadnych operacji arytmetycznych, a na typie unsigned jest to możliwe (co uzasadnia użycie biblioteki numeric_std, w której takie operacje są zdefiniowane).

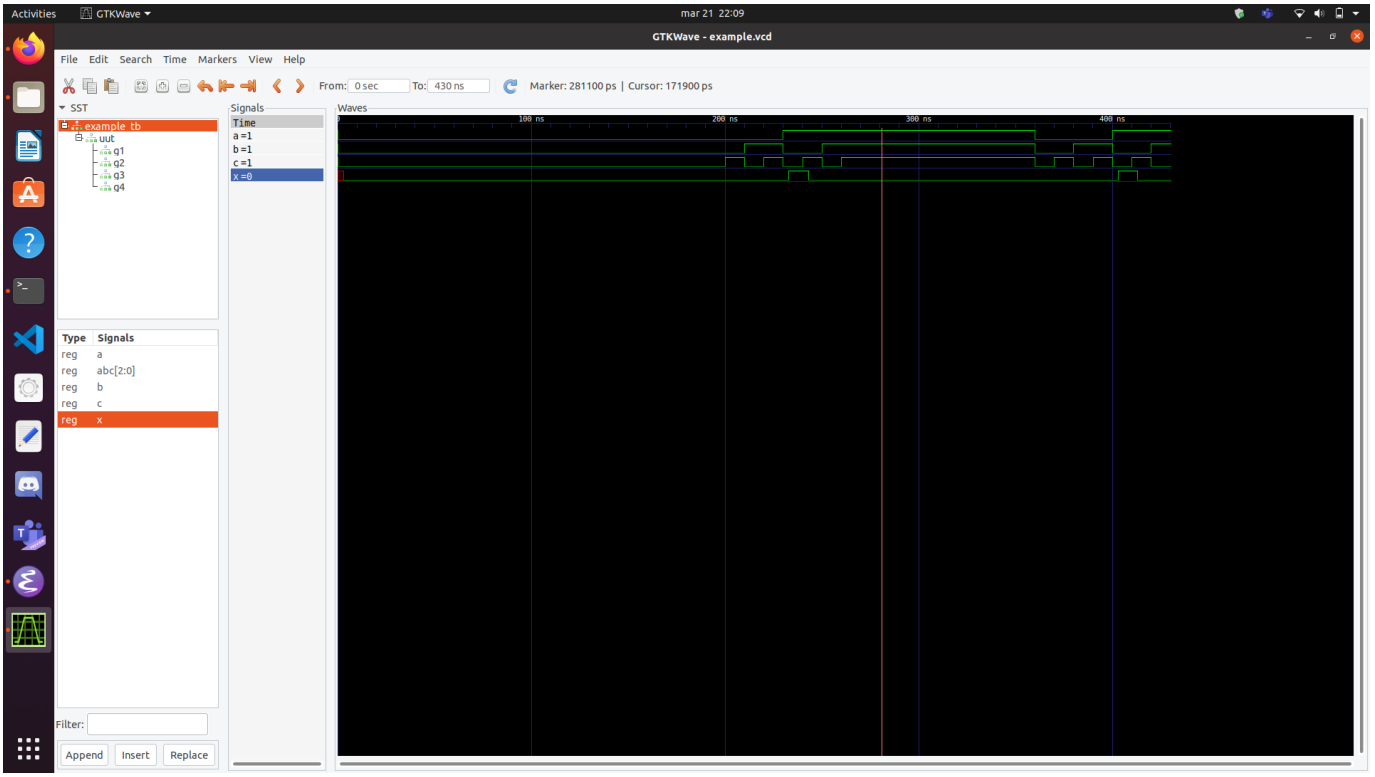
std_logic_vector (i unsigned) używa się jako alternatywnej deklaracji sygnałów wejściowych (sygnał wejściowy traktujemy jako 3-elementowy wektor bitów), a potem wykorzystujemy go do 'sprytnego' zmieniania stanów wejść (możemy to robić ręcznie poprzez definiowanie wartości każdego wejścia lub z użyciem pętli - wtedy musimy wykorzystać któryś z naszych wektorów bitów (I w tym przypadku za każdym razem zwiększamy go o 1. To jest inny, może prostszy sposób pisania testów dla układów)). Program testowy działa dla obu użytych typów, z tym, że gdy chcemy zmienić stan wejścia to musimy:

- Dla typu std_logic_vector:
  Najpierw jawnie zrzutować go na typ unsigned, dodać 1 i potem ponownie (jawnie) zrzutować na typ std_logic_vector.
- Dla typu unsigned:
  Po prostu dodać jedynkę.

Układ z rysunku realizuje następującą funkcję logiczną:

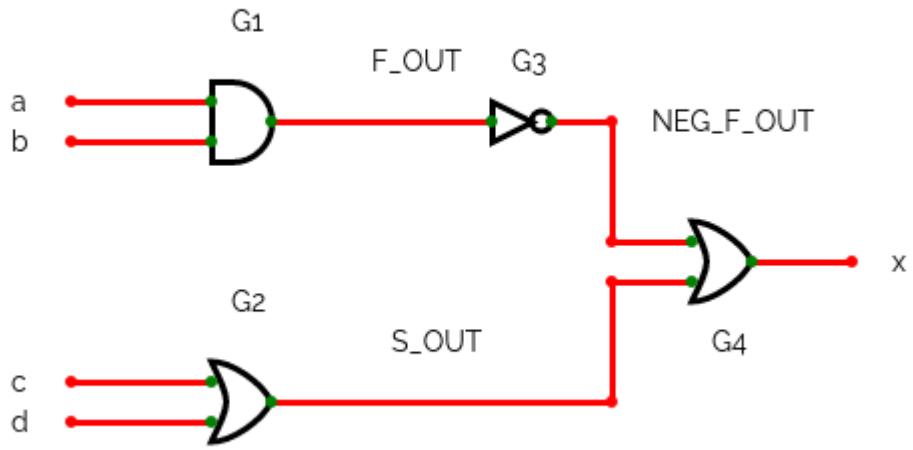
F(a, b, c) = a **AND** (**NOT**(b)) **AND** (**NOT**(c))

Można wyciągnąć taki wniosek po analizie przebiegu sygnału wyjściowego w zależności od sygnałów wejściowych:



Zatem ten układ realizuje taką samą funkcję logiczną jak układ z rysunku (mają takie same tabele prawdy).

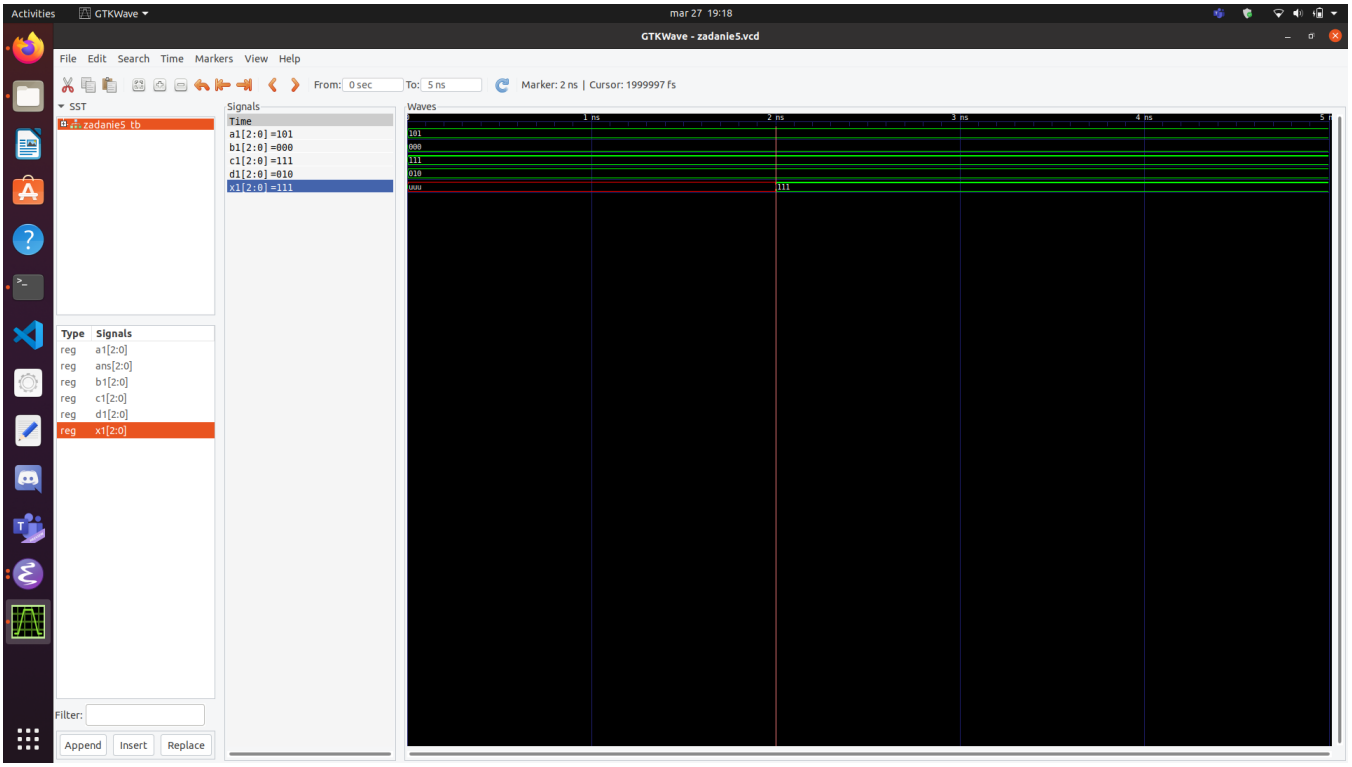
Zadanie 4.
Pragma generic służy do zdefiniowania układu z 16 wejściami i 8 wyjściami (wejścia **A** i **B** traktujemy jako wektory 8 bitowe, tak samo z wyjściem **C**). Za pomocą tej pragmy możemy 'sparametryzować' nasze modele - jedna zmienna jest parametrem i w razie konieczności zmiany wartości tego parametru, musimy zmienić wartość tylko tej zmiennej. Test znajduje się w pliku *Xand_tb.vhd*. Po działaniu tego testu widać, że możemy tworzyć "wersje" tego samego układu, ale z różną liczbą wejść i wyjść.
Zadanie 5.
Rozwiązanie znajduje się w plikach *zadanie5.vhd* i *zadanie5_tb.vhd*. W pliku *vectorgates.vhd* są zdefiniowane bramki, które mogą działać na wektorach bitów.
W tym zadaniu realizowałem następujący układ logiczny:



Po analizie tego układu można stwierdzić, że realizuje taką funkcję logiczną:

F(a, b, c, d) = **NOT** a **OR NOT** b **OR** c **OR** d

Układ (oraz bramki) jest tak zaprojektowany, że na wejściu może przyjmować wektory bitów, a na wyjściu zwracać wektor bitów (a nie tylko pojedyncze bity). Dodatkowo, te wektory mogą mieć zmienną długość (użycie *generic*).
Otrzymane przebiegi sygnałów (plik *zadanie5.vcd*):



Jak widać, układ działa (przynajmniej dla tego przypadku testowego). Oczywiście, żeby mieć całkowitą pewność, że układ działa prawidłowo, należałoby sprawdzić wszystkie możliwe wartości, które mogą pojawić się na wejściu (a ich jest 4096).