OMNIS 2

Otwartość. Modernizacja. Nowoczesność. Integracja. Społeczność.

Podstawy Elektroniki
Ćwiczenia komputerowe

Wstęp do elektroniki cyfrowej

Autor: Dariusz Tefelski

Fundusze Europejskie dla Rozwoju Społecznego
Rzeczpospolita Polska
Dofinansowane przez Unię Europejską
Politechnika Warszawska

Plac Politechniki 1
00-661 Warszawa

www.pw.edu.pl

Spis treści

Fundusze Europejskie dla Rozwoju Społecznego
Rzeczpospolita Polska
Dofinansowane przez Unię Europejską
Politechnika Warszawska

Plac Politechniki 1
00-661 Warszawa
www.pw.edu.pl

1 Cel ćwiczenia

Celem ćwiczenia jest zapoznanie się z podstawami elektroniki cyfrowej. W tym poznanie pojęć takich jak stany logiczne, standardy napięciowe (5V TTL oraz 3.3V CMOS), kwestie dopasowania impedancji, obciążalności wyjść, czasów propagacji sygnałów oraz zagrożeń takich jak hazard i metastabilność. Ponadto poznanie algebry Boole'a, podstawowych bramek logicznych, przerzutników, układów sumujących i liczników, a także cyfrowe reprezentacje liczb (w tym kod U2 oraz standard IEEE 754 dla liczb zmiennoprzecinkowych).

2 Wprowadzenie teoretyczne

Współczesny świat w ogromnym stopniu opiera się na przetwarzaniu informacji w postaci cyfrowej. W przeciwieństwie do elektroniki analogowej, gdzie sygnały mogą przybierać dowolne wartości z ciągłego przedziału, elektronika cyfrowa operuje na wartościach dyskretnych. Informacja jest kodowana za pomocą stanów logicznych, najczęściej binarnych: logicznego zera (0) i logicznej jedynki (1). Jednak pod warstwą czystej abstrakcji matematycznej kryje się rzeczywisty świat fizyczny – prądy, napięcia, pojemności i rezystancje, które determinują działanie rzeczywistych układów scalonych.

Zrozumienie tej fizycznej natury sygnałów cyfrowych jest kluczowe dla każdego inżyniera. Błędy w projektowaniu obwodów cyfrowych często nie wynikają z niepoprawnej logiki matematycznej, lecz z ignorowania zjawisk fizycznych, takich jak opóźnienia propagacji, niedopasowanie impedancji czy stany metastabilne.

3 Stany logiczne i standardy napięciowe

W teorii cyfrowej stan logiczny jest wartością idealną (0 lub 1). W praktyce inżynierskiej stany te reprezentowane są przez przedziały napięć elektrycznych. Aby układ cyfrowy działał poprawnie i był odporny na zakłócenia (szumy), zdefiniowano precyzyjne standardy napięciowe.

Każda rodzina logiczna charakteryzuje się czterema podstawowymi parametrami napięciowymi:

- V_{OL} (*Voltage Output Low*) – maksymalne napięcie na wyjściu układu reprezentujące logiczne zero.
- V_{OH} (*Voltage Output High*) – minimalne napięcie na wyjściu układu reprezentujące logiczną jedynkę.
- V_{IL} (*Voltage Input Low*) – maksymalne napięcie na wejściu układu, które jest jeszcze interpretowane jako logiczne zero.
- V_{IH} (*Voltage Input High*) – minimalne napięcie na wejściu układu, które jest jeszcze interpretowane jako logiczna jedynka.

3.1 Standard 5V TTL

Klasyczna rodzina układów TTL (Transistor-Transistor Logic), zasilana napięciem $V_{CC} = 5$V, oparta jest na tranzystorach bipolarnych. Charakteryzuje się asymetrycznymi progami przełączania:

$$V_{OL,max} = 0.4 \text{ V} \qquad V_{IL,max} = 0.8 \text{ V}$$
$$V_{OH,min} = 2.4 \text{ V} \qquad V_{IH,min} = 2.0 \text{ V}$$

3.2 Standard 3.3V CMOS

Współczesne układy logiczne w technologii CMOS (Complementary Metal-Oxide-Semiconductor) są znacznie bardziej energooszczędne i zasilane niższymi napięciami, np. 3.3V. Czyste układy CMOS charakteryzują się symetrycznymi progami, zależnymi bezpośrednio od napięcia zasilania V_{DD} (najczęściej $V_{IL} \approx 0.3 \cdot V_{DD}$, $V_{IH} \approx 0.7 \cdot V_{DD}$):

$$V_{OL,max} \approx 0.1 \cdot V_{DD} = 0.33 \text{ V} \qquad V_{IL,max} \approx 0.3 \cdot V_{DD} = 0.99 \text{ V}$$
$$V_{OH,min} \approx 0.9 \cdot V_{DD} = 2.97 \text{ V} \qquad V_{IH,min} \approx 0.7 \cdot V_{DD} = 2.31 \text{ V}$$

3.3 Marginesy szumów

Różnica pomiędzy progami wyjściowymi a wejściowymi tworzy tzw. **margines szumów** (*Noise Margin*), czyli maksymalne napięcie zakłócające, które może nałożyć się na sygnał bez ryzyka błędnej interpretacji stanu logicznego:

- Margines szumów dla stanu niskiego: $NM_L = V_{IL,max} - V_{OL,max}$
- Margines szumów dla stanu wysokiego: $NM_H = V_{OH,min} - V_{IH,min}$

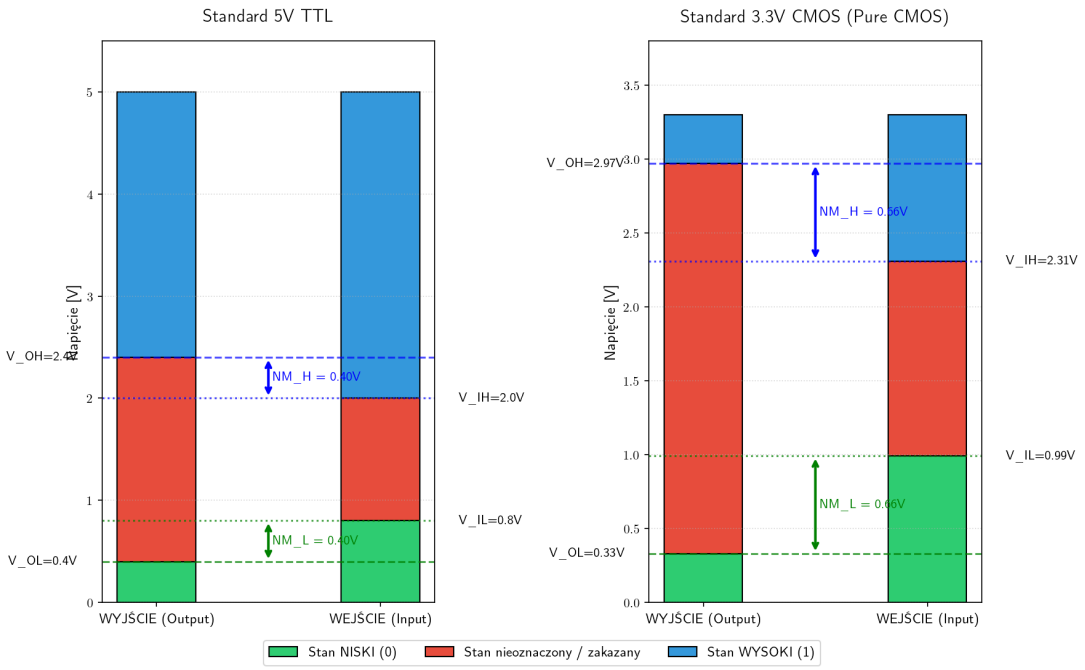
Dla standardu TTL marginesy te wynoszą $NM_L = NM_H = 0.4$V. Dla standardu CMOS 3.3V marginesy szumów są znacznie wyższe (odpowiednio 0.66V dla stanu niskiego i 0.66V dla stanu wysokiego), co czyni rodzinę CMOS bardziej odporną na zakłócenia elektromagnetyczne. Wizualizację tych poziomów przedstawia Rysunek 1.

4 Wejścia, wyjścia i dopasowanie impedancji

Poprawne łączenie ze sobą układów cyfrowych wymaga zrozumienia ich parametrów elektrycznych. Wyjście jednego układu pełni rolę **źródła sygnału**, a wejście drugiego – **odbiornika**.

4.1 Impedancja wejściowa i wyjściowa

- **Wejście układu cyfrowego** charakteryzuje się **bardzo wysoką impedancją wejściową** ($R_{in} \to \infty$, rzędu megaomów lub gigaomów w technologii CMOS). Oznacza to, że wejście pobiera minimalny prąd ze źródła sterującego.



Rysunek 1: Porównanie poziomów logicznych i marginesów szumów w standardach 5V TTL i 3.3V CMOS.

- **Wyjście układu cyfrowego** charakteryzuje się **bardzo niską impedancją wyjściową** ($R_{out} \approx 10\ \Omega - 100\ \Omega$). Umożliwia to szybkie ładowanie i rozładowywanie pojemności obwodu oraz minimalizuje spadek napięcia na samym wyjściu.

Zasada łączenia jest prosta: **zawsze łączymy wyjście układu sterującego z wejściem (lub wejściami) układów sterowanych**.

4.2 PRZESTROGA: Połączenie wyjść – konflikt na magistrali

Uwaga!

WAŻNE: NIGDY nie wolno łączyć bezpośrednio ze sobą dwóch lub więcej klasycznych wyjść układów logicznych (tzw. push-pull) na których mogą występować przeciwne stany logiczne!

Jeśli jedno wyjście zostanie ustawione w stan logicznej jedynki (V_{DD}, niska impedancja do zasilania), a drugie w stan logicznego zera (GND, niska impedancja do masy), dojdzie do bezpośredniego zwarcia linii zasilania z masą przez klucze wyjściowe układów. Płynący prąd zwarciowy:

$$I_{short} = \frac{V_{DD}}{R_{out1} + R_{out2}}$$

może łatwo przekroczyć dopuszczalne limity prądowe tranzystorów (często przekraczając 100mA), co prowadzi do wydzielenia ogromnej ilości ciepła, stopienia struktur krzemowych

i **nieodwracalnego uszkodzenia układów scalonych**. Do łączenia wyjść stosuje się wyłącznie specjalne wyjścia z otwartym kolektorem/drenem (Open Collector/Drain) z rezystorem podciągającym (Pull-up) lub wyjścia trójstanowe (Tri-state).

5 Obciążalność wyjść (Fan-out)

Obciążalność wyjścia (Fan-out) określa, ile wejść innych układów logicznych można podłączyć do jednego wyjścia, tak aby nie zostały przekroczone dopuszczalne poziomy napięć.

5.1 Fan-out w technologii TTL

W klasycznej technologii TTL przez wejścia płyną znaczące prądy stałe (szczególnie w stanie niskim, gdy prąd wpływa z wejścia do wyjścia sterującego, $I_{IL} \approx -1.6\text{mA}$). Wtedy obciążalność definiowana jest przez stosunek dopuszczalnych prądów wyjściowych do prądów wejściowych odbiorników:

$$N = \min\left(\frac{I_{OL,max}}{I_{IL,max}}, \frac{I_{OH,max}}{I_{IH,max}}\right)$$

Zazwyczaj dla TTL fan-out wynosi około 10. Podłączenie większej liczby wejść spowoduje, że napięcie V_{OL} wyjdzie poza dopuszczalną granicę 0.8V, co zakłóci działanie układu.

5.2 Fan-out w technologii CMOS

W technologii CMOS prądy wejściowe w stanie ustalonym są znikome ($I_{in} < 1\text{pA}$). Z perspektywy prądu stałego fan-out jest niemal nieskończony (można by podłączyć tysiące bramek). Jednak każde wejście CMOS stanowi obciążenie pojemnościowe ($C_{in} \approx 3\text{pF} - 10\text{pF}$). Podłączenie wielu wejść tworzy dużą pojemność sumaryczną $C_L = N \cdot C_{in} + C_{wire}$, która musi być ładowana i rozładowywana przez rezystancję wyjściową bramki R_{out}. Tworzy to filtr dolnoprzepustowy RC. Zwiększanie N powoduje drastyczne wydłużenie czasów narastania i opadania zboczy sygnałów, co wydłuża czas propagacji bramki i drastycznie ogranicza maksymalną częstotliwość pracy układu. W praktyce projektowej ogranicza się fan-out układów CMOS do 10-20. Zjawisko to zostało zobrazowane na wykresie w Zadaniu 3.

6 Propagacja sygnałów i konieczność pracy synchronicznej

Żaden układ fizyczny nie działa natychmiastowo. Zmiana stanu na wejściu bramki logicznej pojawia się na jej wyjściu dopiero po pewnym czasie, zwanym **czasem propagacji** (t_{PD} – *Propagation Delay*). Czas ten wynosi typowo od ułamka nanosekundy do kilkudziesięciu nanosekund, w zależności od technologii wykonania.

Wyróżniamy dwa podstawowe czasy propagacji:

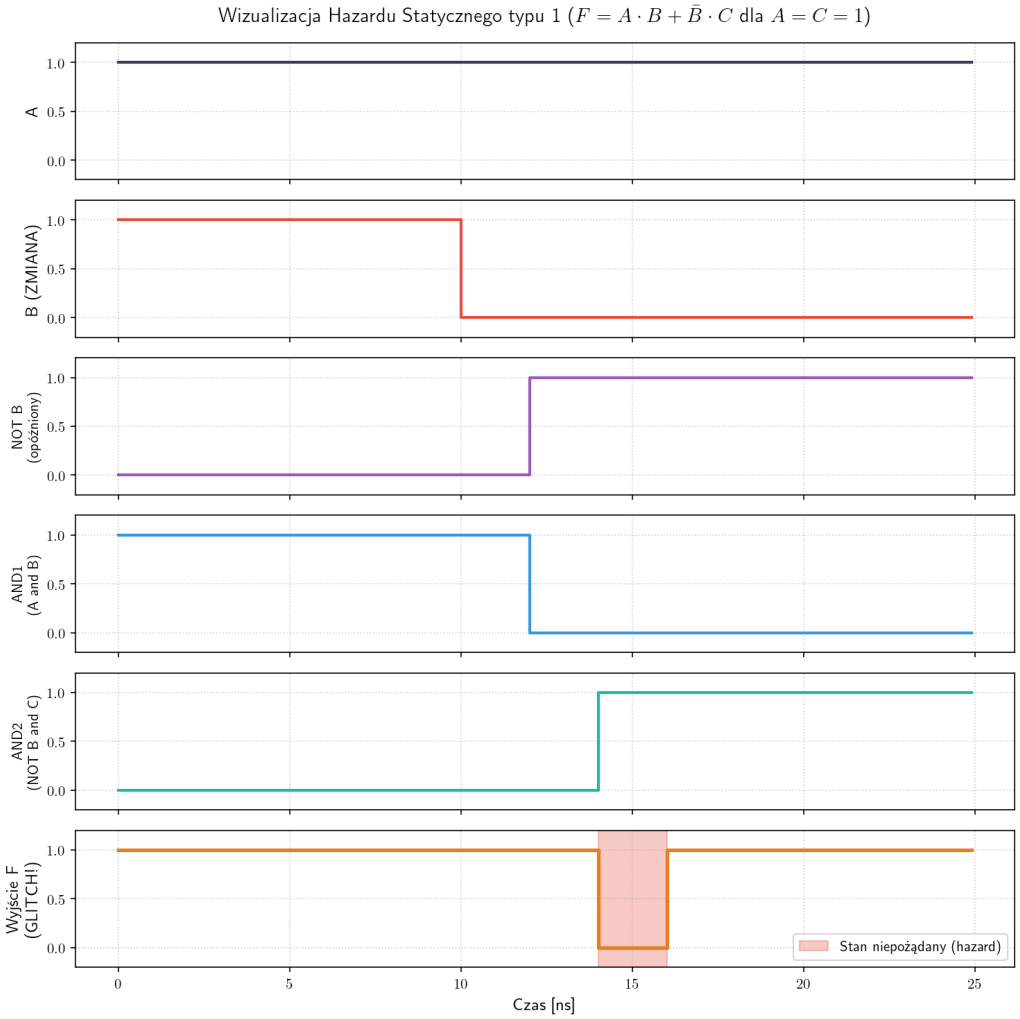
Fundusze Europejskie dla Rozwoju Społecznego | Rzeczpospolita Polska | Dofinansowane przez Unię Europejską | Politechnika Warszawska

Plac Politechniki 1
00-661 Warszawa
www.pw.edu.pl

- t_{PLH} – czas propagacji przy zmianie wyjścia ze stanu niskiego (Low) na wysoki (High).
- t_{PHL} – czas propagacji przy zmianie wyjścia ze stanu wysokiego (High) na niski (Low).

6.1 Hazard w układach asynchronicznych

W układach kombinacyjnych (asynchronicznych), gdzie sygnały biegną różnymi ścieżkami logicznymi o różnych opóźnieniach, dochodzi do zjawiska **hazardu** (*hazard*). Przejawia się ono powstawaniem chwilowych, niepożądanych impulsów na wyjściu (tzw. **szpilek** lub *glitches*).



Rysunek 2: Symulacja hazardu statycznego typu 1 wywołanego opóźnieniem sygnału na inwerterze.

Przykładem jest funkcja $F = A \cdot B + \bar{B} \cdot C$. Przy stałych wartościach $A = 1$ i $C = 1$, logicznie $F = B + \bar{B} = 1$. Jednak podczas przełączania B z 1 na 0, opóźnienie inwertera

generującego $\bar{B}$ sprawia, że przez krótki czas oba człony koniunkcji dają 0. W efekcie na wyjściu F pojawia się chwilowe zero (Rysunek 2), które może wyzwolić kolejne układy (np. liczniki), prowadząc do całkowitej destabilizacji pracy urządzenia.

6.2 Rozwiązanie: Synchroniczna praca układów

Aby wyeliminować negatywny wpływ hazardu i różnic w czasach propagacji ścieżek logicznych, w złożonych systemach stosuje się **pracę synchroniczną**. Wszystkie zmiany stanów w układzie są taktowane wspólnym sygnałem zegarowym (*Clock*), a stany pośrednie są zapamiętywane w elementach sekwencyjnych – **przerzutnikach typu D** na zboczu zegara. Dopóki czas trwania okresu zegara T_{clk} jest większy niż najdłuższa ścieżka krytyczna propagacji w układzie kombinacyjnym plus czas konfiguracji przerzutnika, hazard na wyjściach bramek kombinacyjnych zdąży zaniknąć przed kolejnym aktywnym zboczem zegara i układ będzie działał w 100% stabilnie i przewidywalnie.

7 Metastabilność

Metastabilność to zjawisko, w którym element pamiętający (np. przerzutnik) po aktywnym zboczu zegara nie jest w stanie szybko zdecydować, czy powinien przejść w stan logicznego zera, czy jedynki. Przerzutnik wchodzi wtedy w stan nieustalony – jego napięcie wyjściowe oscyluje wokół poziomu progowego (pomiędzy V_{IL} a V_{IH}) przez czas znacznie dłuższy niż standardowy czas propagacji.

7.1 Przyczyna: Czas setup i hold

Aby przerzutnik zapisał sygnał poprawnie, sygnał ten musi być stabilny na wejściu przez określony czas:

- **Czas konfiguracji** (t_{setup}) – minimalny czas przed aktywnym zboczem zegara, przez który sygnał wejściowy musi być stabilny.
- **Czas podtrzymania** (t_{hold}) – minimalny czas po aktywnym zboczu zegara, przez który sygnał wejściowy musi pozostać stabilny.

Naruszenie któregokolwiek z tych czasów (co często ma miejsce, gdy wprowadzamy do układu sygnały asynchroniczne, np. z przycisków lub innych domen zegarowych) prowadzi bezpośrednio do metastabilności.

7.2 Przeciwdziałanie

Zjawiska metastabilności nie da się całkowicie wyeliminować (jest to zjawisko statystyczne), ale można drastycznie zmniejszyć prawdopodobieństwo jego wystąpienia. W tym celu stosuje się **dwustopniowe synchronizatory** (szeregowo połączone dwa przerzutniki typu D taktowane tym samym zegarem). Pierwszy przerzutnik może wejść w stan metastabilny, ale ma niemal cały okres zegara na ustabilizowanie stanu przed nadejściem kolejnego zbocza, na którym drugi przerzutnik bezpiecznie przepisze już stabilną wartość logiczną.

Hazard w układach logicznych to niepożądane, przejściowe zakłócenie sygnału wyjściowego (krótki impuls, tzw. **glitch**), które pojawia się w wyniku różnych czasów propagacji sygnałów przez bramki logiczne.

Warto wiedzieć

Wyróżniamy trzy główne typy hazardu:

1. `Hazard Statyczny (Static Hazard)`
   Występuje, gdy wyjście powinno zachować stałą wartość (0 lub 1) podczas zmiany jednego z wejść, ale na chwilę zmienia stan na przeciwny.
   - **Hazard statyczny typu 1 (Static-1 Hazard):** Wyjście powinno być cały czas na poziomie wysokim (1), ale na chwilę spada do 0. Często spotykany w strukturach typu Sum of Products (SOP).
   - **Hazard statyczny typu 0 (Static-0 Hazard):** Wyjście powinno być cały czas na poziomie niskim (0), ale na chwilę skacze do 1. Często spotykany w strukturach typu Product of Sums (POS).

   Przyczyna: Sygnał dociera do bramki wyjściowej dwiema różnymi ścieżkami, z których jedna jest nieco opóźniona (np. przez inwerter).
2. `Hazard Dynamiczny (Dynamic Hazard)` Występuje, gdy wyjście powinno zmienić stan z 0 na 1 (lub z 1 na 0), ale zamiast jednej czystej zmiany, "oscyluje"kilka razy (np. $0 \longrightarrow 1 \longrightarrow 0 \longrightarrow 1$).
   Przyczyna: Sygnał wejściowy dociera do wyjścia co najmniej trzema różnymi drogami o różnych opóźnieniach. Występuje tylko w bardziej skomplikowanych, wielopoziomowych sieciach logicznych.
3. `Hazard Funkcjonalny (Functional Hazard)` Występuje, gdy **dwa lub więcej** wejść zmieniają się w tym samym czasie. Jeśli czasy przejścia sygnałów przez układ są różne, wyjście może na chwilę przyjąć stan, który nie wynika z tabeli prawdy dla żadnego ze stanów (początkowego ani końcowego).

Uwaga: Ten typ hazardu jest bardzo trudny do wyeliminowania metodami logicznymi. Rozwiązaniem jest zazwyczaj zapewnienie, by wejścia nie zmieniały się jednocześnie (np. przez synchronizację zegarem)

Informacje dodatkowe

Jak eliminować hazard?

1. `Metoda tablic Karnaugha (dla hazardu statycznego):` Należy dodać tzw. **redundantne grupy** (pętle), które „zakrywają" przejścia między sąsiednimi grupami w tabeli. Dzięki temu, gdy jedno wejście się zmienia, inna ścieżka logiczna podtrzymuje sygnał wyjściowy.
2. `Projektowanie synchroniczne (Najważniejsze w FPGA):` W układach FPGA nie walczy się z hazardem przez dodawanie bramek (co jest ryzykowne ze względu na optymalizację kompilatora). Zamiast tego, sygnały wyjściowe z logiki kombinacyjnej przepuszcza się przez **rejestry (flip-flopy)** sterowane zegarem. Rejestr „próbkuje" sygnał dopiero po ustaleniu się stanów nieustalonych, całkowicie ignorując hazard.
3. `Grey Coding:` Używanie kodu Graya (gdzie zmienia się tylko jeden bit na raz) w licznikach czy automatach skończonych (FSM) eliminuje hazard funkcjonalny.

Wniosek dla inżyniera FPGA: Jeśli budujesz logikę w VHDL/Verilogu, zawsze staraj się, aby Twoja logika kombinacyjna była „zatrzaskiwana" w rejestrach. Hazard wewnątrz chmury logiki kombinacyjnej jest dopuszczalny, dopóki sygnał zdąży się ustabilizować przed kolejnym zboczem zegara.

8 Algebra Boole'a

Logika układów cyfrowych opiera się na algebrze Boole'a, w której zmienne mogą przybierać tylko dwie wartości: 0 (fałsz) i 1 (prawda).

8.1 Podstawowe operatory logiczne

- Koniunkcja (AND): $Y = A \cdot B$ (iloczyn logiczny)
- Alternatywa (OR): $Y = A + B$ (suma logiczna)
- Negacja (NOT): $Y = \bar{A}$ (zaprzeczenie)
- Alternatywa wykluczająca (XOR): $Y = A \oplus B = A\bar{B} + \bar{A}B$ (suma poprzeczna - daje 1, gdy wejścia się różnią)

8.2 Prawa Algebry Boole'a

Do najważniejszych praw należą:

$$A + 0 = A \qquad A \cdot 1 = A$$
$$A + \bar{A} = 1 \qquad A \cdot \bar{A} = 0$$
$$A + A = A \qquad A \cdot A = A$$
$$\overline{A + B} = \bar{A} \cdot \bar{B} \quad \text{(I prawo De Morgana)} \qquad \overline{A \cdot B} = \bar{A} + \bar{B} \quad \text{(II prawo De Morgana)}$$

Fundusze Europejskie dla Rozwoju Społecznego | Rzeczpospolita Polska | Dofinansowane przez Unię Europejską | Politechnika Warszawska

Plac Politechniki 1
00-661 Warszawa
www.pw.edu.pl

Twierdzenia De Morgana są kluczowe w praktyce, ponieważ pozwalają zastąpić dowolne bramki AND i OR za pomocą uniwersalnych bramek NAND lub NOR, co znacznie ułatwia produkcję i optymalizację układów scalonych.

9 Reprezentacje liczbowe i operacje arytmetyczne

W układach cyfrowych liczby zapisywane są w systemie dwójkowym (binarnym). Każda pozycja (bit) reprezentuje kolejną potęgę liczby 2.

9.1 Liczby całkowite bez znaku (NKB)

W Naturalnym Kodzie Binarnym liczba X reprezentowana przez n bitów $(a_{n-1}a_{n-2}\dots a_0)$ ma wartość:

$$X = \sum_{i=0}^{n-1} a_i \cdot 2^i$$

Przykładowo, $(1011)_2 = 1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = 11_{10}$.

9.2 Zapis liczb ujemnych: Kod uzupełnienia do dwóch (U2)

W systemach cyfrowych najpopularniejszym standardem zapisu liczb ze znakiem jest **kod uzupełnienia do dwóch (U2)**. Najstarszy bit (MSB) ma wagę ujemną -2^{n-1} i pełni rolę bitu znaku (0 – dodatnia, 1 – ujemna):

$$X = -a_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} a_i \cdot 2^i$$

Dzięki temu operacja odejmowania sprowadza się do dodawania liczby przeciwnej, co bardzo upraszcza budowę procesorów.

Negację liczby w kodzie U2 realizuje się poprzez operację: **zaneguj wszystkie bity i dodaj 1**. Na przykład, aby zapisać -5 na 4 bitach:

1. $5_{10} = (0101)_2$
2. Negacja bitów: 1010
3. Dodanie 1: 1011

Rzeczywiście: $-1 \cdot 8 + 0 \cdot 4 + 1 \cdot 2 + 1 \cdot 1 = -5$.

9.3 Operacje bitowe

Oprócz dodawania i odejmowania, procesory realizują szybkie operacje bitowe bezpośrednio na strukturach rejestrów:

- **AND** – maskowanie bitów (zerowanie wybranych pozycji).
- **OR** – ustawianie wybranych bitów na 1.

- **XOR** – negowanie wybranych bitów (klucz szyfrujący, generator parzystości).
- **Przesunięcie bitowe w lewo ($<< k$)** – przesunięcie bitów o k pozycji w lewo (odpowiednik mnożenia przez 2^k). Wolne miejsca z prawej strony wypełniane są zerami.
- **Przesunięcie bitowe w prawo ($>> k$)** – przesunięcie bitów o k pozycji w prawo (odpowiednik dzielenia całkowitego przez 2^k). Normalnie wypełniane zerami od lewej strony, ale przy przesunięciu arytmetycznym najstarszy bit znaku jest powielany.

9.4 Kolejność bajtów w pamięci – Big-Endian i Little-Endian

W systemach cyfrowych dane wielobitowe (np. słowa 16-, 32- lub 64-bitowe) są przechowywane w pamięci operacyjnej adresowanej bajtowo. Ponieważ pojedyncze słowo zajmuje więcej niż jeden bajt, kluczowym zagadnieniem staje się ustalenie kolejności, w jakiej poszczególne bajty tego słowa są zapisywane pod kolejnymi adresami pamięci.

Wyróżniamy dwa podstawowe standardy zapisu:

- **Big-Endian (grubokońcowość / zapis od najstarszego bajtu)** – najbardziej znaczący bajt (MSB – *Most Significant Byte*) słowa jest zapisywany pod najniższym adresem pamięci (najwcześniej), a kolejne bajty o mniejszym znaczeniu pod adresami wyższymi. Jest to intuicyjny sposób, zgodny z naturalnym kierunkiem pisania liczb (od lewej do prawej).
- **Little-Endian (małokońcowość / zapis od najmłodszego bajtu)** – najmniej znaczący bajt (LSB – *Least Significant Byte*) słowa jest zapisywany pod najniższym adresem pamięci, a bajty o większym znaczeniu pod adresami wyższymi.

Rozważmy jako przykład 32-bitową liczbę całkowitą zapisaną w formacie szesnastkowym jako $X = 0x12345678$. Składa się ona z czterech bajtów: 0x12 (MSB), 0x34, 0x56 oraz 0x78 (LSB). Jeżeli zechcemy zapisać tę liczbę w pamięci pod adresem startowym 0x1000, rozmieszczenie bajtów będzie wyglądało następująco (Tabela 1):

Tabela 1: Porównanie zapisów Big-Endian i Little-Endian dla słowa 0x12345678.

Adres pamięci	0x1000	0x1001	0x1002	0x1003
Zapis Big-Endian	0x12 (MSB)	0x34	0x56	0x78 (LSB)
Zapis Little-Endian	0x78 (LSB)	0x56	0x34	0x12 (MSB)

9.4.1 Kolejność bajtów w procesorach

- **Architektura x86 oraz x86-64 (Intel, AMD)** – stosuje wyłącznie standard **Little-Endian**. Jest to dominujący standard w komputerach osobistych i serwerach. Zaletą sprzętową tego rozwiązania jest łatwa konwersja typów danych (np. z rzutowania liczby 32-bitowej na 16-bitową – adres początku liczby nie ulega zmianie, po prostu odczytuje się mniej bajtów).

- **Procesory ARM** – są z natury obustronne (*Bi-endian*), co oznacza, że mogą sprzętowo pracować w obu trybach. Jednak w zdecydowanej większości systemów operacyjnych (np. Android, iOS, Linux) pracują domyślnie w trybie **Little-Endian**.
- **Starsze architektury (Motorola 68000, SPARC, IBM Mainframes)** – wykorzystywały standard **Big-Endian**.

9.4.2 Kolejność bajtów w standardzie sieciowym (Ethernet i IP)

W sieciach komputerowych, aby umożliwić wymianę danych pomiędzy maszynami o różnych architekturach procesorów, konieczne było zdefiniowanie jednolitego standardu przesyłania danych wielobajtowych.

- **Network Byte Order (Standard IP)** – protokoły rodziny TCP/IP (IP, TCP, UDP) definiują standard przesyłania jako **Big-Endian**. Oznacza to, że np. numery portów lub adresy IP przesyłane w nagłówkach pakietów muszą być zapisane w formacie Big-Endian. W systemach operacyjnych programiści korzystają ze specjalnych funkcji (np. htonl, ntohl, htons, ntohs) w celu konwersji pomiędzy kolejnością hosta (*Host Byte Order*) a sieciową (*Network Byte Order*).
- **Standard Ethernet (L2) i transmisja szeregowa** – na poziomie warstwy fizycznej i łącza danych (IEEE 802.3 Ethernet), dane są przesyłane przez medium szeregowo (bit po bicie). Tutaj napotykamy na bardzo ważną i subtelną różnicę pomiędzy kolejnością bajtów (*byte order*) a kolejnością bitów (*bit order*):
  - **Kolejność bajtów**: Pola wielobajtowe w ramce Ethernet (takie jak adresy MAC nadawcy i odbiorcy czy pole EtherType) są przesyłane i interpretowane w formacie **Big-Endian** (najpierw bajt najbardziej znaczący).
  - **Kolejność bitów**: W obrębie każdego pojedynczego bajtu, standard Ethernet 802.3 przesyła bity w kolejności **LSB-first (od najmniej znaczącego bitu)**! Jedynym wyjątkiem jest pole sumy kontrolnej CRC-32, które jest wysyłane od bitu najbardziej znaczącego (MSB-first).

Ta specyfika oznacza, że jeśli przesyłamy bajt o wartości 0xD4 (binarnie $1101\,0100_2$), na poziomie fizycznym medium jako pierwszy zostanie wysłany bit najmniej znaczący (czyli 0), a jako ostatni bit najbardziej znaczący (1).

9.5 Liczby zmiennoprzecinkowe (IEEE 754)

Liczby ułamkowe reprezentowane są w standardzie zmiennoprzecinkowym IEEE 754 jako przybliżenie liczb rzeczywistych. Liczba ma postać:

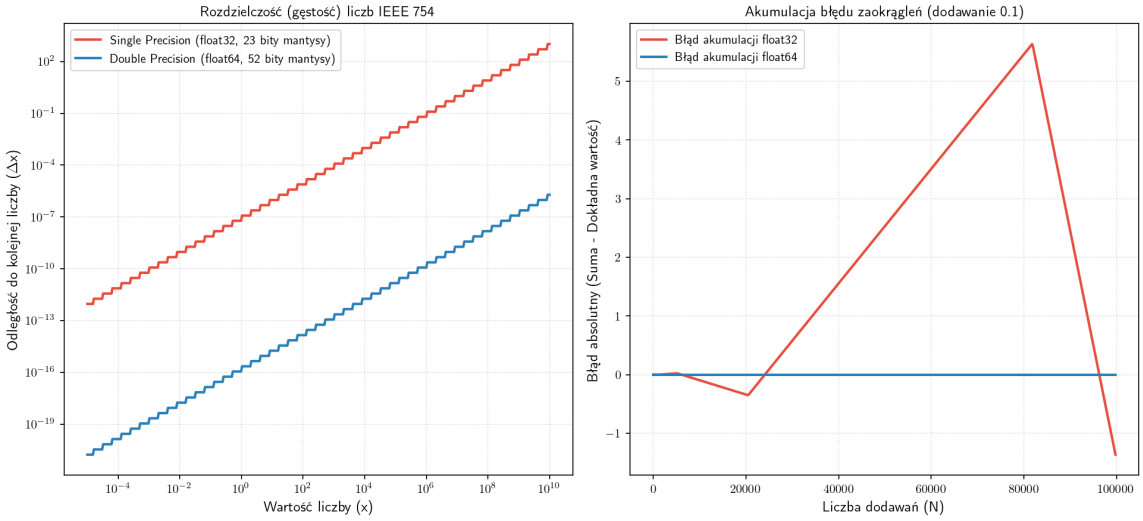
$$X = (-1)^s \cdot M \cdot 2^{E-bias}$$

Gdzie:

- s – bit znaku (1 bit).

- E – wykładnik (*exponent*, przesunięty o wartość *bias*, która dla pojedynczej precyzji wynosi 127, a dla podwójnej 1023).
- M – znormalizowana mantysa (*mantissa*) z ukrytą jedynką wiodącą ($1.f$).

Ponieważ mantysa ma skończoną długość (23 bity w float32, 52 bity w float64), większość ułamków dziesiętnych (np. 0.1) nie ma dokładnej reprezentacji binarnej i jest zaokrąglana. Prowadzi to do powstawania drobnych błędów zaokrągleń, które mogą drastycznie narastać w pętlach obliczeniowych (Rysunek 3).



Rysunek 3: Rozdzielczość liczb w standardzie IEEE 754 oraz wizualizacja błędu akumulacji przy wielokrotnym dodawaniu liczby 0.1.

10 Podstawowe komponenty i układy logiczne

10.1 Bramki logiczne i ich tabele prawdy

Bramki logiczne to elementarne układy realizujące funkcje algebry Boole'a. Poniżej przedstawiono zestawienie podstawowych bramek.

Tabela 2: Tabele prawdy podstawowych bramek logicznych.

A	B	AND $(A \cdot B)$	OR $(A + B)$	NAND $(\overline{A \cdot B})$	NOR $(\overline{A + B})$	XOR $(A \oplus B)$
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

10.2 Przerzutniki (Flip-Flops)

Przerzutniki to podstawowe komórki pamięci jednobitowej, reagujące na poziom lub zbocze sygnału taktującego:

- **Przerzutnik SR** – najprostszy asynchroniczny zatrzask z wejściem ustawiającym S (*Set*) i zerującym R (*Reset*). Stan $S = R = 1$ jest stanem zabronionym.
- **Przerzutnik D (*Delay*)** – przepisuje wartość z wejścia D na wyjście Q w momencie wystąpienia aktywnego zbocza zegarowego. Kluczowy w logice synchronicznej.
- **Przerzutnik JK** – ulepszony przerzutnik SR, w którym stan $J = K = 1$ powoduje zmianę wyjścia na przeciwne (negację stanu).
- **Przerzutnik T (*Toggle*)** – jeśli wejście $T = 1$, to każde aktywne zbocze zegara neguje stan wyjścia. Doskonały do budowy dzielników częstotliwości i liczników.

10.3 Rejestry (Registers)

Rejestr to układ sekwencyjny służący do przechowywania lub przekształcania wielobitowej informacji (słowa binarnego). Zbudowany jest z grupy powiązanych ze sobą przerzutników (najczęściej typu D) posiadających wspólny sygnał taktujący (zegar) oraz dodatkowe linie sterujące, takie jak zezwolenie na zapis (*Enable*) czy zerowanie (*Reset/Clear*).

W zależności od sposobu wprowadzania i wyprowadzania danych, rejestry dzielimy na cztery podstawowe typy:

- **SISO (*Serial-In Serial-Out*)** – rejestr z szeregowym wejściem i szeregowym wyjściem. Dane są wprowadzane bit po bicie z każdym taktem zegara i wysuwane na wyjście w ten sam sposób. Służy głównie jako linia opóźniająca.

- **SIPO (*Serial-In Parallel-Out*)** – rejestr z szeregowym wejściem i równoległym wyjściem. Umożliwia konwersję sygnału szeregowego na słowo równoległe (np. odbiór danych w protokołach UART czy SPI).
- **PISO (*Parallel-In Serial-Out*)** – rejestr z równoległym wejściem i szeregowym wyjściem. Pozwala na jednoczesne załadowanie całego słowa binarnego i wysyłanie go bit po bicie (np. transmisja danych UART).
- **PIPO (*Parallel-In Parallel-Out*)** – rejestr z równoległym wejściem i wyjściem. Służy do tymczasowego przechowywania (zatrzaśnięcia) danych wielobitowych w magistralach procesorów (rejestry buforowe i robocze).

10.3.1 Rejestry przesuwające (Shift Registers)

Szczególną klasą rejestrów są **rejestry przesuwające**, w których przerzutniki są połączone szeregowo w taki sposób, że wyjście Q jednego przerzutnika jest połączone bezpośrednio z wejściem D następnego. Z każdym aktywnym zboczem zegara stan każdego przerzutnika jest przepisywany do jego sąsiada z prawej (lub lewej) strony. Przesunięcie o jedną pozycję w lewo w rejestrze PIPO jest sprzętowym odpowiednikiem pomnożenia liczby przez 2, natomiast przesunięcie w prawo odpowiada dzieleniu całkowitemu przez 2 (dla liczb bez znaku). Rejestry te są fundamentalnymi blokami w budowie interfejsów komunikacyjnych oraz koprocesorów arytmetycznych.

10.4 Układy kombinacyjne i sekwencyjne

10.4.1 Układ sumujący (Sumator)

Sumator to układ kombinacyjny służący do dodawania liczb binarnych.

- **Półsumator (Half Adder)** – dodaje dwa bity A i B, dając sumę S oraz przeniesienie C (*Carry Out*):
$$S = A \oplus B, \quad C = A \cdot B$$
- **Sumator pełny (Full Adder)** – dodaje trzy bity: wejścia A, B oraz przeniesienie z poprzedniej pozycji C_{in}:
$$S = A \oplus B \oplus C_{in}, \quad C_{out} = A \cdot B + C_{in} \cdot (A \oplus B)$$

10.4.2 Układ licznika

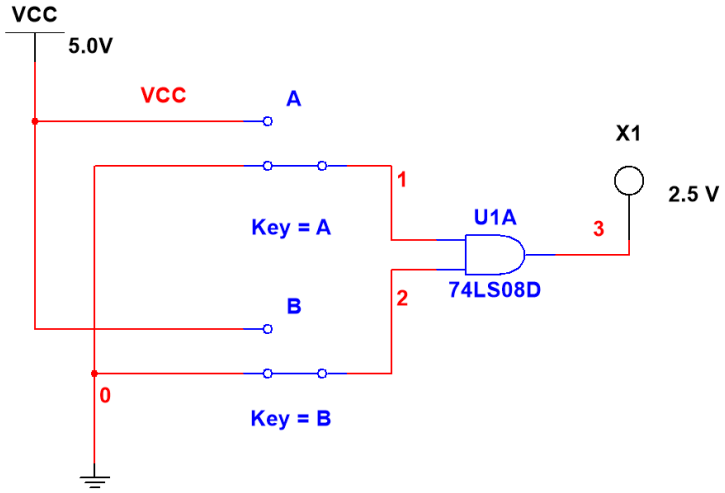
Liczniki to układy sekwencyjne zliczające impulsy zegarowe.

- **Licznik asynchroniczny** – zbudowany z szeregowo połączonych przerzutników T, gdzie wyjście Q jednego przerzutnika taktuje wejście zegarowe kolejnego. Wadą jest narastające opóźnienie propagacji (hazard stanów pośrednich).
- **Licznik synchroniczny** – wszystkie przerzutniki są taktowane tym samym sygnałem zegarowym jednocześnie, a przejścia stanów kontrolowane są przez sieć kombinacyjną podłączoną do wejść sterujących (np. J, K lub D). Brak hazardu stanów przejściowych.

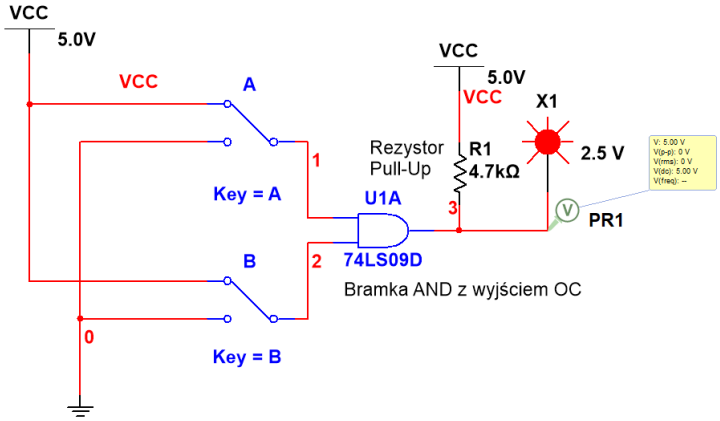
11 Zadania ćwiczeniowe w środowisku Multisim

Zadanie

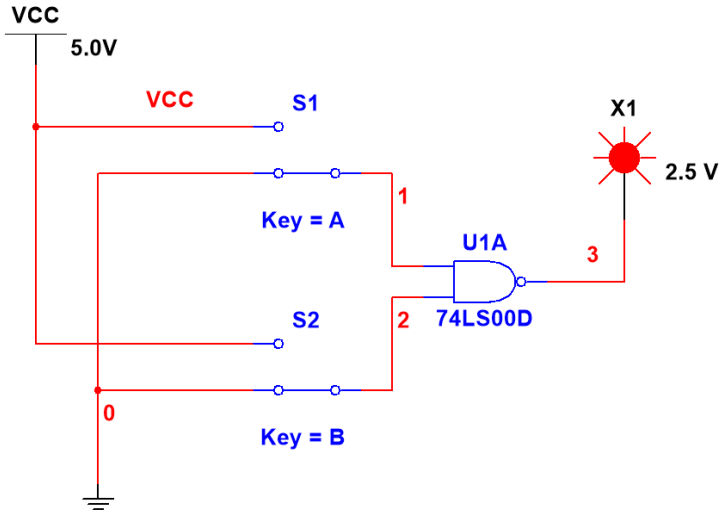
1. Narysuj w Multisim schemat testowego układu dla dwuwejściowych układów logicznych i przetestuj działanie różnych układów z serii 74LS.. Patrz rys. 4 do 9.
2. Przeanalizuj działanie układu 4-bitowego pełnego sumatora (z bitem przeniesienia Carry). Otwórz plik w Multisim: File ⟶ Open samples... ⟶ Educational Sample Circuits ⟶ Digital Circuits ⟶ **4BitFull**.ms14.
   Zapoznaj się z hierarchicznym schematem pojedynczej komórki sumującej. Rozpoznaj symbole wykorzystanych bramek.
3. Przeanalizuj działanie 4-bitowego układu sumującego i odejmującego (prosta jednostka ALU - Arithmatic Logic Unit). Otwórz plik w Multisim: File ⟶ Open samples... ⟶ Educational Sample Circuits ⟶ Digital Circuits ⟶ **AddSubtract**.ms14.
   Zwróć uwagę, jak przy operacji odejmowania bity są negowane przy pomocy bramek `XOR` i jak negowane są bity przy wyniku ujemnym też z wykorzystaniem bramek `XOR`
4. Przeanalizuj działanie demultipleksera 74138 (dekodera 8 z 3). Otwórz plik w Multisim: File ⟶ Open samples... ⟶ Educational Sample Circuits ⟶ Digital Circuits ⟶ **Demultiplexer**.ms14.
   Zauważ, że układ może posłużyć do wybierania - adresowania np. 8 różnych układów na magistrali SPI, sterując tylko 4 liniami.
5. Sprawdź działanie synchronicznego licznika w kodzie BCD (cyfry 0-9). Otwórz plik w Multisim: File ⟶ Open samples... ⟶ Educational Sample Circuits ⟶ Digital Circuits ⟶ **SyncCounter**.ms14.
   Zauważ wykorzystanie przerzutników typu JK oraz układu bramek do AND/OR do ograniczenia zliczeń do kodów 0-9.
6. Zbadaj działanie synchornicznego rewersyjnego licznika dzisiętnego BCD: 74190. Otwórz plik w Multisim: File ⟶ Open samples... ⟶ Educational Sample Circuits ⟶ Digital Circuits ⟶ **UpDwnCounter**.ms14.
   Zwróć uwagę na kaskadowe połączenie 2 układów odpowiadających za cyfry jedności i dziesiątek.
7. Przeanalizuj inne przykłady układów, takich jak rejestr przesuwny, licznik binarny, itp. Otwórz pliki w Multisim: File ⟶ Open samples... ⟶ Digital ⟶ ...



Rysunek 4: Multisim: Model bramki typu AND

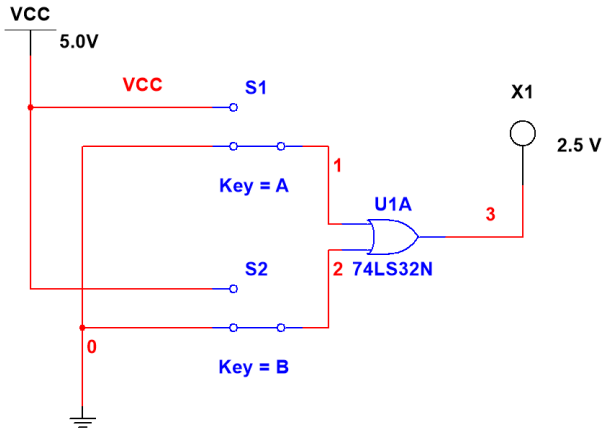


Rysunek 5: Multisim: Model bramki typu AND z wyjściem typu OC (Open Collector)

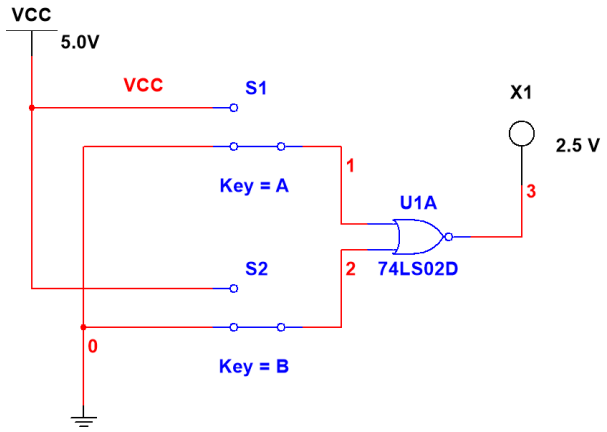


Rysunek 6: Multisim: Model bramki typu NAND

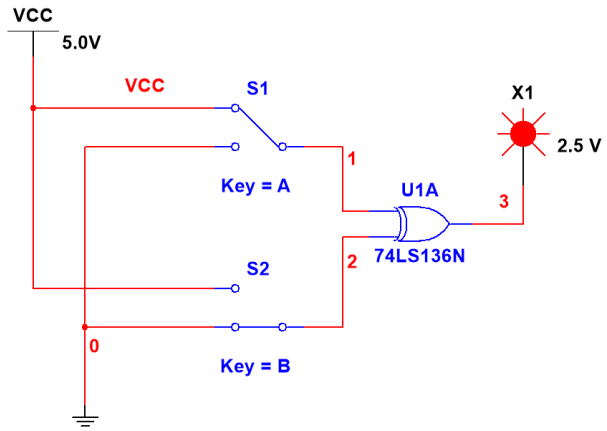
Plac Politechniki 1
00-661 Warszawa
www.pw.edu.pl



Rysunek 7: Multisim: Model bramki typu OR



Rysunek 8: Multisim: Model bramki typu NOR



Rysunek 9: Multisim: Model bramki typu XOR

12 Zadania ćwieczeniowe w środowisku Python

W tej sekcji przedstawiono cztery zadania laboratoryjne, które pozwalają na symulację i wizualizację kluczowych zjawisk z zakresu elektroniki cyfrowej. Kody są przystosowane do uruchomienia bezpośrednio w środowisku Jupyter-lab.

12.1 Zadanie 1: Poziomy napięć i marginesy szumów

Cel zadania: Zapoznanie się ze standardami napięciowymi 5V TTL oraz 3.3V CMOS poprzez wizualizację ich zakresów wejściowych/wyjściowych i obliczenie marginesów szumów.

Listing 12.1: Skrypt wizualizujący poziomy napięć.

```
"""
Zadanie 1: Poziomy napięć i marginesy szumów w standardach 5V TTL i 3.3V
↪  CMOS.
Skrypt generuje wykres przedstawiający dopuszczalne zakresy napięć
↪  wejściowych i wyjściowych.
"""

import matplotlib.pyplot as plt
import numpy as np


def rysuj_poziomy():
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 7), sharey=False)

    # Dane dla TTL 5V
    ttl_levels = {
        "V_OL": 0.4,
        "V_IL": 0.8,
        "V_IH": 2.0,
        "V_OH": 2.4,
        "V_CC": 5.0,
    }

    # Dane dla CMOS 3.3V (standardowe LVCMOS 3.3V)
    cmos_levels = {
        "V_OL": 0.4,
        "V_IL": 0.8,
        "V_IH": 2.0,
        "V_OH": 2.4,
        "V_CC": 3.3,
    }

    # Uwaga: Dla czystego CMOS 3.3V często przyjmuje się V_IL = 0.3*VDD =
    ↪  0.99V, V_IH = 0.7*VDD = 2.31V,
```

Listing 12.1: Skrypt wizualizujący poziomy napięć. (c.d.)

```
    # V_OL = 0.1V, V_OH = VDD-0.1V = 3.2V. Pokażemy te bardziej
    ↪ restrykcyjne poziomy CMOS.
    cmos_pure_levels = {
        "V_OL": 0.33,  # 0.1 * VDD
        "V_IL": 0.99,  # 0.3 * VDD
        "V_IH": 2.31,  # 0.7 * VDD
        "V_OH": 2.97,  # 0.9 * VDD
        "V_CC": 3.3,
    }

    def rysuj_rodzine(ax, levels, tytul):
        # Szerokość słupków
        w = 0.35
        x = np.array([0, 1])

        # Wyjście (Output) - słupek po lewej
        ax.bar(
            x[0],
            levels["V_OL"],
            width=w,
            color="#2ecc71",
            edgecolor="black",
            label="Stan NISKI (0)",
        )
        ax.bar(
            x[0],
            levels["V_OH"] - levels["V_OL"],
            bottom=levels["V_OL"],
            width=w,
            color="#e74c3c",
            edgecolor="black",
            label="Stan nieoznaczony / zakazany",
        )
        ax.bar(
            x[0],
            levels["V_CC"] - levels["V_OH"],
            bottom=levels["V_OH"],
            width=w,
            color="#3498db",
            edgecolor="black",
            label="Stan WYSOKI (1)",
        )

        # Wejście (Input) - słupek po prawej
        ax.bar(
            x[1], levels["V_IL"], width=w, color="#2ecc71",
            ↪ edgecolor="black"
```

Listing 12.1: Skrypt wizualizujący poziomy napięć. (c.d.)

```
        )
        ax.bar(
            x[1],
            levels["V_IH"] - levels["V_IL"],
            bottom=levels["V_IL"],
            width=w,
            color="#e74c3c",
            edgecolor="black",
        )
        ax.bar(
            x[1],
            levels["V_CC"] - levels["V_IH"],
            bottom=levels["V_IH"],
            width=w,
            color="#3498db",
            edgecolor="black",
        )

        # Linie pomocnicze i opisy poziomów wyjściowych
        ax.axhline(levels["V_OL"], color="green", linestyle="--",
        ↪  alpha=0.7)
        ax.axhline(levels["V_OH"], color="blue", linestyle="--",
        ↪  alpha=0.7)
        ax.text(
            -0.5,
            levels["V_OL"],
            f"V_OL={levels['V_OL']}V",
            va="center",
            ha="right",
            fontweight="bold",
        )
        ax.text(
            -0.5,
            levels["V_OH"],
            f"V_OH={levels['V_OH']}V",
            va="center",
            ha="right",
            fontweight="bold",
        )

        # Linie pomocnicze i opisy poziomów wejściowych
        ax.axhline(levels["V_IL"], color="green", linestyle=":",
        ↪  alpha=0.7)
        ax.axhline(levels["V_IH"], color="blue", linestyle=":", alpha=0.7)
        ax.text(
            1.35,
            levels["V_IL"],
```

Listing 12.1: Skrypt wizualizujący poziomy napięć. (c.d.)

```
        f"V_IL={levels['V_IL']}V",
        va="center",
        ha="left",
        fontweight="bold",
    )
    ax.text(
        1.35,
        levels["V_IH"],
        f"V_IH={levels['V_IH']}V",
        va="center",
        ha="left",
        fontweight="bold",
    )

    # Marginesy szumów
    nm_l = levels["V_IL"] - levels["V_OL"]
    nm_h = levels["V_OH"] - levels["V_IH"]

    # Rysowanie strzałek marginesów szumów
    ax.annotate(
        "",
        xy=(0.5, levels["V_OL"]),
        xytext=(0.5, levels["V_IL"]),
        arrowprops=dict(arrowstyle="<->", color="green", lw=2),
    )
    ax.text(
        0.52,
        (levels["V_OL"] + levels["V_IL"]) / 2,
        f"NM_L = {nm_l:.2f}V",
        color="green",
        fontweight="bold",
        va="center",
    )

    ax.annotate(
        "",
        xy=(0.5, levels["V_IH"]),
        xytext=(0.5, levels["V_OH"]),
        arrowprops=dict(arrowstyle="<->", color="blue", lw=2),
    )
    ax.text(
        0.52,
        (levels["V_IH"] + levels["V_OH"]) / 2,
        f"NM_H = {nm_h:.2f}V",
        color="blue",
        fontweight="bold",
        va="center",
```

Listing 12.1: Skrypt wizualizujący poziomy napięć. (c.d.)

```
        )

        ax.set_xticks(x)
        ax.set_xticklabels(
            ["WYJŚCIE (Output)", "WEJŚCIE (Input)"],
            fontsize=11,
            fontweight="bold",
        )
        ax.set_ylabel("Napięcie [V]", fontsize=12)
        ax.set_title(tytul, fontsize=14, pad=15, fontweight="bold")
        ax.set_ylim(0, levels["V_CC"] + 0.5)
        ax.grid(axis="y", linestyle=":", alpha=0.5)

    rysuj_rodzine(ax1, ttl_levels, "Standard 5V TTL")
    rysuj_rodzine(ax2, cmos_pure_levels, "Standard 3.3V CMOS (Pure CMOS)")

    # Legend
    handles, labels = ax1.get_legend_handles_labels()
    fig.legend(
        handles,
        labels,
        loc="lower center",
        ncol=3,
        bbox_to_anchor=(0.5, -0.05),
        fontsize=11,
    )

    plt.tight_layout()
    plt.show()


rysuj_poziomy()
```

Interpretacja wyników: Wykres wygenerowany przez skrypt (patrz Rysunek 1) przedstawia różnice w filozofii obu rodzin. Rodzina TTL ma węższe i niesymetryczne progi logiczne, natomiast standard CMOS oferuje znacznie większe marginesy szumów (NM_L, NM_H), co przekłada się na wyższą odporność na zakłócenia zewnętrzne.

12.2 Zadanie 2: Hazard w układach asynchronicznych

Cel zadania: Zrozumienie mechanizmu powstawania hazardu statycznego w układzie kombinacyjnym wywołanego skończonym czasem propagacji bramek (inwertera).

Listing 12.2: Skrypt symulujący zjawisko hazardu.

```
"""
Zadanie 2: Symulacja hazardu statycznego typu 1 w układzie asynchronicznym.
Funkcja logiczna: F = (A AND B) OR (NOT(B) AND C)
Dla A = 1, C = 1 funkcja powinna stale dawać 1, ale podczas zmiany B z 1 na 0
ze względu na opóźnienie inwertera pojawia się szpilka (glitch) o wartości 0.
"""

import matplotlib.pyplot as plt
import numpy as np


def symuluj_hazard():
    # Krok czasowy w nanosekundach
    dt = 0.1
    t = np.arange(0, 25, dt)

    # Wejścia
    A = np.ones_like(t)  # A = 1 stale
    C = np.ones_like(t)  # C = 1 stale

    # B przełącza się z 1 na 0 w t = 10 ns
    B = np.where(t < 10.0, 1.0, 0.0)

    # Definiujemy opóźnienia bramek (w nanosekundach)
    delay_not = 2.0
    delay_and1 = 2.0
    delay_and2 = 2.0
    delay_or = 2.0

    # Funkcja pomocnicza do opóźniania sygnału o zadany czas
    def delay_signal(sig, delay_ns, dt):
        steps = int(delay_ns / dt)
        if steps == 0:
            return sig.copy()
        delayed = np.zeros_like(sig)
        # Wypełniamy początkowe kroki wartością początkową sygnału
        delayed[:steps] = sig[0]
        delayed[steps:] = sig[:-steps]
        return delayed

    # 1. NOT B
    not_B_ideal = 1.0 - B
    not_B = delay_signal(not_B_ideal, delay_not, dt)
```

Listing 12.2: Skrypt symulujący zjawisko hazardu. (c.d.)

```
# 2. AND1 = A AND B
and1_ideal = A * B
and1 = delay_signal(and1_ideal, delay_and1, dt)

# 3. AND2 = NOT_B AND C
and2_ideal = not_B * C
and2 = delay_signal(and2_ideal, delay_and2, dt)

# 4. OR = AND1 OR AND2
or_ideal = np.clip(and1 + and2, 0.0, 1.0)
F = delay_signal(or_ideal, delay_or, dt)

# Wykresy
fig, axes = plt.subplots(6, 1, figsize=(10, 10), sharex=True)

# Style linii i kolorów
style = {"lw": 2, "color": "#2c3e50"}

axes[0].step(t, A, where="post", **style)
axes[0].set_ylabel("A, C", fontsize=12, fontweight="bold")
axes[0].set_ylim(-0.2, 1.2)
axes[0].grid(True, linestyle=":", alpha=0.6)
axes[0].set_title(
    "Wizualizacja Hazardu Statycznego typu 1 ($F = A\\cdot B +
    ↪  \\bar{B}\\cdot C$ dla $A=C=1$)",
    fontsize=14,
    fontweight="bold",
    pad=15,
)

axes[1].step(t, B, where="post", color="#e74c3c", lw=2)
axes[1].set_ylabel("B (ZMIANA)", fontsize=12, fontweight="bold")
axes[1].set_ylim(-0.2, 1.2)
axes[1].grid(True, linestyle=":", alpha=0.6)

axes[2].step(t, not_B, where="post", color="#9b59b6", lw=2)
axes[2].set_ylabel("NOT B\n(opóźniony)", fontsize=10,
↪  fontweight="bold")
axes[2].set_ylim(-0.2, 1.2)
axes[2].grid(True, linestyle=":", alpha=0.6)

axes[3].step(t, and1, where="post", color="#3498db", lw=2)
axes[3].set_ylabel("AND1\n(A and B)", fontsize=10, fontweight="bold")
axes[3].set_ylim(-0.2, 1.2)
axes[3].grid(True, linestyle=":", alpha=0.6)

axes[4].step(t, and2, where="post", color="#1abc9c", lw=2)
```

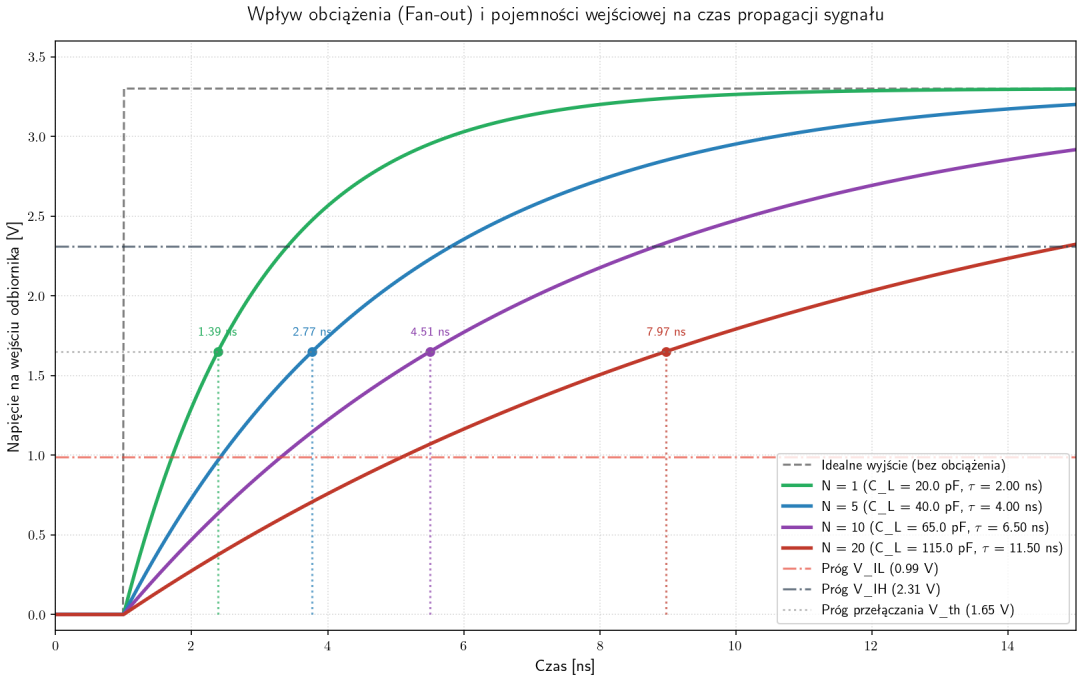
Listing 12.2: Skrypt symulujący zjawisko hazardu. (c.d.)

```
    axes[4].set_ylabel("AND2\n(NOT B and C)", fontsize=10,
    ↪  fontweight="bold")
    axes[4].set_ylim(-0.2, 1.2)
    axes[4].grid(True, linestyle=":", alpha=0.6)

    # Wyjście F z zaznaczonym hazardem
    axes[5].step(t, F, where="post", color="#e67e22", lw=2.5)
    axes[5].set_ylabel("Wyjście F\n(GLITCH!)", fontsize=12,
    ↪  fontweight="bold")
    axes[5].set_ylim(-0.2, 1.2)
    axes[5].set_xlabel("Czas [ns]", fontsize=12, fontweight="bold")
    axes[5].grid(True, linestyle=":", alpha=0.6)

    # Zaznaczenie obszaru hazardu na wyjściu F
    hazard_start = 10.0 + delay_and1 + delay_or
    hazard_end = 10.0 + delay_not + delay_and2 + delay_or
    axes[5].axvspan(
        hazard_start,
        hazard_end,
        color="#e74c3c",
        alpha=0.3,
        label="Stan niepożądany (hazard)",
    )
    axes[5].legend(loc="lower right")

    # Zmiana B następuje w t=10ns.
    # AND1 reaguje w t=12ns.
    # NOT B reaguje w t=12ns.
    # AND2 reaguje w t=14ns.
    # OR reaguje w t=14ns (spadek) i t=16ns (wzrost).

    plt.tight_layout()
    plt.show()


symuluj_hazard()
```

Interpretacja wyników: Na wygenerowanym wykresie (Rysunek 2) widać, że w momencie zmiany stanu wejściowego B z 1 na 0, na wyjściu teoretycznie stałej funkcji $F = 1$ pojawia się krótkotrwałe zakłócenie (szpilka o wartości 0) trwające 2 ns. Wynika to z opóźnienia inwertera NOT, przez co przez chwilę bramka AND1 już wyłączyła stan wysoki, a bramka AND2 jeszcze go nie włączyła.

12.3 Zadanie 3: Fizyczna natura wejść/wyjść – ładowanie obwodu RC

Cel zadania: Zrozumienie, w jaki sposób pojemność wejściowa tranzystorów MOSFET (C_{in}) w bramkach CMOS oraz rezystancja wyjściowa stopnia sterującego (R_{out}) tworzą obwód RC spowalniający sygnał i jak wpływa to na czas propagacji t_{PLH} przy różnym obciążeniu (Fan-out N).



Listing 12.3: Skrypt symulujący ładowanie pojemności wejściowych CMOS.

```
"""
Zadanie 3: Symulacja ładowania pojemności wejściowej (obwód RC) a czas
↪ propagacji.
Skrypt pokazuje jak rezystancja wyjściowa bramki sterującej (R_out) oraz
↪ pojemność
wejściowa bramki odbiornika (C_in) wpływają na kształt sygnału oraz czas
↪ propagacji
t_PLH przy różnej liczbie podłączonych wejść (obciążalności fan-out N = 1,
↪ 5, 10, 20).
"""

import matplotlib.pyplot as plt
import numpy as np


def symuluj_ladowanie_rc():
    # Parametry fizyczne
```

Listing 12.3: Skrypt symulujący ładowanie pojemności wejściowych CMOS. (c.d.)

```
R_out = 100.0  # Rezystancja wyjściowa bramki sterującej [Ohm]
C_in = 5e-12  # Pojemność wejściowa pojedynczego odbiornika [Farad] (5
↪  pF)
C_wire = 15e-12  # Pojemność ścieżki na PCB [Farad] (15 pF)
V_DD = 3.3  # Napięcie zasilania CMOS [V]

# Progi przełączania dla CMOS 3.3V
V_IL = 0.99  # 0.3 * V_DD [V]
V_IH = 2.31  # 0.7 * V_DD [V]
V_th = V_DD / 2  # Próg przełączania idealny (1.65 V)

# Krok czasowy i czas symulacji (w sekundach)
t = np.linspace(0, 15e-9, 1000)  # 0 - 15 ns

# Sygnał wejściowy (idealny skok jednostkowy w t = 1 ns)
t_switch = 1e-9
V_in = np.where(t < t_switch, 0.0, V_DD)

# Liczba obciążeń (Fan-out N)
fan_out_values = [1, 5, 10, 20]
colors = ["#27ae60", "#2980b9", "#8e44ad", "#c0392b"]

plt.figure(figsize=(11, 7))

# Rysujemy idealny sygnał sterujący
plt.plot(
    t * 1e9,
    V_in,
    "k--",
    label="Idealne wyjście (bez obciążenia)",
    alpha=0.5,
    lw=1.5,
)

for N, col in zip(fan_out_values, colors):
    # Całkowita pojemność obciążenia
    C_load = N * C_in + C_wire
    # Stała czasowa tau = R * C
    tau = R_out * C_load

    # Odpowiedź układu RC na skok napięcia:
    # V(t) = 0 dla t < t_switch
    # V(t) = V_DD * (1 - exp(-(t - t_switch)/tau)) dla t >= t_switch
    V_out = np.zeros_like(t)
    active_idx = t >= t_switch
    V_out[active_idx] = V_DD * (
```

Listing 12.3: Skrypt symulujący ładowanie pojemności wejściowych CMOS. (c.d.)

```
            1.0 - np.exp(-(t[active_idx] - t_switch) / tau)
        )

        # Obliczenie czasu propagacji t_PLH (czas od t_switch do
        ↪  osiągnięcia progu V_th = V_DD/2)
        # 1 - exp(-t_prop/tau) = 0.5 => t_prop = tau * ln(2)
        t_prop = tau * np.log(2)
        t_PLH = t_switch + t_prop

        # Wykres napięcia
        plt.plot(
            t * 1e9,
            V_out,
            color=col,
            lw=2.5,
            label=f"N = {N} (C_L = {C_load * 1e12:.1f} pF, $\\tau$ = {tau
            ↪  * 1e9:.2f} ns)",
        )

        # Zaznaczenie punktu propagacji (V_out = V_th)
        plt.plot(t_PLH * 1e9, V_th, "o", color=col, ms=6)
        # Pionowa linia do osi X
        plt.vlines(t_PLH * 1e9, 0, V_th, colors=col, linestyles=":",
        ↪  alpha=0.7)
        plt.text(
            t_PLH * 1e9,
            V_th + 0.1,
            f"{t_prop * 1e9:.2f} ns",
            color=col,
            fontsize=9,
            ha="center",
        )

    # Rysujemy linie progów logicznych
    plt.axhline(
        V_IL,
        color="#e74c3c",
        linestyle="-.",
        alpha=0.7,
        label="Próg V_IL (0.99 V)",
    )
    plt.axhline(
        V_IH,
        color="#2c3e50",
        linestyle="-.",
        alpha=0.7,
```

Listing 12.3: Skrypt symulujący ładowanie pojemności wejściowych CMOS. (c.d.)

```
        label="Próg V_IH (2.31 V)",
    )
    plt.axhline(
        V_th,
        color="gray",
        linestyle=":",
        alpha=0.5,
        label="Próg przełączania V_th (1.65 V)",
    )

    plt.title(
        "Wpływ obciążenia (Fan-out) i pojemności wejściowej na czas
        ↪ propagacji sygnału",
        fontsize=14,
        fontweight="bold",
        pad=15,
    )
    plt.xlabel("Czas [ns]", fontsize=12, fontweight="bold")
    plt.ylabel(
        "Napięcie na wejściu odbiornika [V]", fontsize=12,
        ↪ fontweight="bold"
    )
    plt.xlim(0, 15)
    plt.ylim(-0.1, V_DD + 0.3)
    plt.grid(True, linestyle=":", alpha=0.5)
    plt.legend(loc="lower right", fontsize=10)

    plt.tight_layout()
    plt.show()
    # plt.savefig('ladowanie_pojemnosci_rc.png', bbox_inches='tight',
    ↪ dpi=150)
    # print("Wykres ladowanie_pojemnosci_rc.png został pomyślnie
    ↪ wygenerowany.")
    # plt.close()


symuluj_ladowanie_rc()
```

Interpretacja wyników: Z wykresu widać, że wraz ze wzrostem liczby podłączonych odbiorników ($N = 1, 5, 10, 20$) rośnie pojemność zastępcza obciążenia C_L. Skutkuje to wolniejszym narastaniem napięcia na wejściach odbiorników. Czas potrzebny do przekroczenia progu przełączania V_{th} (czas propagacji t_{PLH}) rośnie liniowo od ułamka nanosekundy do ponad 1.5 ns przy $N = 20$. Pokazuje to, dlaczego zbyt wysoki parametr fan-out ogranicza maksymalną częstotliwość taktowania układów CMOS.

12.4 Zadanie 4: Reprezentacja liczb zmiennoprzecinkowych i błędy precyzji

Cel zadania: Zrozumienie, jak liczby rzeczywiste są reprezentowane w standardzie IEEE 754, jak wygląda dekompozycja bitowa liczby float oraz jak błędy zaokrągleń akumulują się podczas prostych obliczeń.

Listing 12.4: Skrypt badający reprezentację IEEE 754.

```
"""
Zadanie 4: Analiza reprezentacji zmiennoprzecinkowej IEEE 754 i błędu
↪ zaokrągleń.
Skrypt dekomponuje liczbę typu float (podwójnej precyzji - 64 bity) na
↪ znak,
wykładnik i mantysę oraz wizualizuje zjawisko utraty precyzji (odległość
↪ między
kolejnymi liczbami) oraz akumulację błędu przy wielokrotnym dodawaniu
↪ 0.1.
"""

import struct
import matplotlib.pyplot as plt
import numpy as np


def float_to_bin64(f):
    """Konwertuje liczbę float (double) na reprezentację binarną w
    ↪ formacie IEEE 754 (64 bity)."""
    # Spakuj float jako 8-bajtową liczbę podwójnej precyzji (double)
    packed = struct.pack(">d", f)
    # Rozpakuj jako 64-bitową liczbę całkowitą bez znaku
    integ = struct.unpack(">Q", packed)[0]
    # Sformatuj jako 64-znakowy ciąg bitów
    b = f"{integ:064b}"

    sign = b[0]
    exponent = b[1:12]
    mantissa = b[12:]
    return sign, exponent, mantissa


def analizuj_float():
    print("--- Dekompozycja IEEE 754 (64-bit Double Precision) ---")
    proby = [-1, 0, 1.0, 0.1, 0.5, -2.5, 1e-9]
    for p in proby:
        sign, exp, mant = float_to_bin64(p)
        val_exp = int(exp, 2) - 1023  # Przesunięcie wykładnika (bias)
        print(f"Liczba: {p:12}")
        print(f"  Znak:      {sign} (0 = +, 1 = -)")
```

Listing 12.4: Skrypt badający reprezentację IEEE 754. (c.d.)

```
        print(
            f"  Wykładnik: {exp} (bin) = {int(exp, 2)} (dec, po odjęciu
            ↪  biasu 1023: {val_exp})"
        )
        print(f"  Mantysa:   {mant[:52]} 52 bity")
        print("-" * 50)

    # Wizualizacja 1: Krok kwantyzacji (odległość między liczbami) w
    ↪  funkcji wartości
    # Dla IEEE 754 odległość ta wynosi eps = 2^(wykładnik - 52) dla 64
    ↪  bitów (double)
    # i eps = 2^(wykładnik - 23) dla 32 bitów (single). Zademonstrujmy to
    ↪  dla 32 bitów (float32).
    wartosci = np.logspace(-5, 10, 500)
    eps_32 = np.spacing(
        np.float32(wartosci)
    )  # Odległość do następnej liczby w float32
    eps_64 = np.spacing(
        np.float64(wartosci)
    )  # Odległość do następnej liczby w float64

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(13, 6))

    # Wykres odległości między sąsiednimi liczbami zmiennoprzecinkowymi
    ax1.loglog(
        wartosci,
        eps_32,
        label="Single Precision (float32, 23 bity mantysy)",
        color="#e74c3c",
        lw=2,
    )
    ax1.loglog(
        wartosci,
        eps_64,
        label="Double Precision (float64, 52 bity mantysy)",
        color="#2980b9",
        lw=2,
    )
    ax1.set_xlabel("Wartość liczby (x)", fontsize=12, fontweight="bold")
    ax1.set_ylabel(
        "Odległość do kolejnej liczby ($\\Delta$x)",
        fontsize=12,
        fontweight="bold",
    )
    ax1.set_title(
        "Rozdzielczość (gęstość) liczb IEEE 754",
        fontsize=13,
```

Listing 12.4: Skrypt badający reprezentację IEEE 754. (c.d.)

```
        fontweight="bold",
    )
    ax1.grid(True, which="both", ls=":", alpha=0.5)
    ax1.legend()

    # Wizualizacja 2: Akumulacja błędu w pętli
    # Klasyczny przykład: dodawanie 0.1 wielokrotnie. 0.1 nie ma
    ↪  skończonej reprezentacji w binarnym systemie pozycyjnym!
    N = 100000
    wartosc_dodawana = 0.1

    # Symulacja dodawania w float32 (wymuszamy typ float32)
    suma_float32 = np.float32(0.0)
    krok_32 = np.float32(wartosc_dodawana)
    historia_err_32 = []

    # Symulacja dodawania w float64 (domyślny w Pythonie)
    suma_float64 = np.float64(0.0)
    krok_64 = np.float64(wartosc_dodawana)
    historia_err_64 = []

    kroki = np.arange(1, N + 1)

    # Liczymy błąd co 500 kroków, żeby przyspieszyć symulację
    punkty_pomiaru = np.arange(0, N, 200)
    err_32 = []
    err_64 = []

    for i in range(N):
        suma_float32 += krok_32
        suma_float64 += krok_64
        if i in punkty_pomiaru:
            dokladna_wartosc = (i + 1) * wartosc_dodawana
            err_32.append(float(suma_float32) - dokladna_wartosc)
            err_64.append(float(suma_float64) - dokladna_wartosc)

    ax2.plot(
        punkty_pomiaru,
        err_32,
        label="Błąd akumulacji float32",
        color="#e74c3c",
        lw=2,
    )
    ax2.plot(
        punkty_pomiaru,
        err_64,
        label="Błąd akumulacji float64",
```

Listing 12.4: Skrypt badający reprezentację IEEE 754. (c.d.)

```
        color="#2980b9",
        lw=2,
    )
    ax2.set_xlabel("Liczba dodawań (N)", fontsize=12, fontweight="bold")
    ax2.set_ylabel(
        "Błąd absolutny (Suma - Dokładna wartość)",
        fontsize=12,
        fontweight="bold",
    )
    ax2.set_title(
        "Akumulacja błędu zaokrągleń (dodawanie 0.1)",
        fontsize=13,
        fontweight="bold",
    )
    ax2.grid(True, linestyle=":", alpha=0.5)
    ax2.legend()

    plt.tight_layout()
    plt.show()


analizuj_float()
```

Interpretacja wyników: Wykres po lewej stronie na Rysunku 3 pokazuje, że odległość między kolejnymi sąsiednimi reprezentowalnymi liczbami zmiennoprzecinkowymi (Δx) rośnie wykładniczo wraz ze wzrostem samej liczby x. Wynika to ze skończonej liczby bitów mantysy. Wykres po prawej stronie ilustruje poważne zjawisko: dodawanie liczby 0.1 (która nie ma dokładnej reprezentacji binarnej) w pętli 100 000 razy generuje narastający błąd absolutny sumy, który dla pojedynczej precyzji (float32) osiąga aż ponad 0.015, podczas gdy dla float64 pozostaje znikomy. Uświadamia to studentom wagę doboru odpowiedniej precyzji w obliczeniach numerycznych i sterowaniu cyfrowym.

Warto wiedzieć

W Numpy istnieje funkcja: `np.spacing(x)`, która pokazuje o ile różni się następna reprezentowalna liczba zmiennoprzecinkowa. Można to sprawdzić uruchamiając kod: 12.5

Listing 12.5: Skrypt wskazujący następną rozróżnialną liczbę zmiennoprzecinkową reprezentowaną przez float64 (double) wg. standardu IEEE 754.

```
import numpy as np

x = 1.5

print(
```

Listing 12.5: Skrypt wskazujący następną rozróżnialną liczbę zmiennoprzecinkową reprezentowaną przez float64 (double) wg. standardu IEEE 754. (c.d.)

```
    f"ULP dla {x}: {np.spacing(x)}"
)  # np.spacing pozwala na poznanie o ile minimalnie różni się następna
↪  zapisana liczba
print(f"Dla {x} next: {(x + np.spacing(x))}")
```

13 Pytania kontrolne

W ramach przygotowania do zajęć laboratoryjnych i kolokwium, studenci powinni odpowiedzieć na następujące pytania:

1. `Dlaczego margines szumów dla standardu 3.3V CMOS jest większy niż dla 5V TTL? Odpowiedź uzasadnij odpowiednimi progami napięciowymi.`

   Odpowiedź: Margines szumów (*Noise Margin*) to różnica pomiędzy dopuszczalnymi progami wyjściowymi a wejściowymi dla poszczególnych stanów logicznych. Stanowi on maksymalną wartość napięcia zakłócającego, jakie może nałożyć się na sygnał bez ryzyka błędnej interpretacji stanu logicznego.

   - **Dla standardu 5V TTL**:

     $$NM_L = V_{IL,max} - V_{OL,max} = 0.8\ \text{V} - 0.4\ \text{V} = 0.4\ \text{V}$$
     $$NM_H = V_{OH,min} - V_{IH,min} = 2.4\ \text{V} - 2.0\ \text{V} = 0.4\ \text{V}$$

     Margines szumów dla obu stanów wynosi 0.4 V.

   - **Dla standardu 3.3V CMOS (czyste progi CMOS)**:

     $$NM_L = V_{IL,max} - V_{OL,max} = 0.99\ \text{V} - 0.33\ \text{V} = 0.66\ \text{V}$$
     $$NM_H = V_{OH,min} - V_{IH,min} = 2.97\ \text{V} - 2.31\ \text{V} = 0.66\ \text{V}$$

     Margines szumów dla obu stanów wynosi 0.66 V.

   Mimo niższego napięcia zasilania (3.3V w porównaniu do 5V), rodzina CMOS posiada znacznie szerzej i symetryczniej rozstawione progi przełączania ($V_{IL} \approx 0.3 \cdot V_{DD}$, $V_{IH} \approx 0.7 \cdot V_{DD}$). Przekłada się to bezpośrednio na wyższy margines szumów (0.66 V w stosunku do 0.4 V), przez co układy CMOS są istotnie bardziej odporne na zakłócenia elektromagnetyczne.

2. `Wyjaśnij, dlaczego zwarcie dwóch wyjść logicznych typu push-pull o przeciwnych stanach może doprowadzić do fizycznego uszkodzenia układu scalonego. Jakie prądy mogą wtedy popłynąć?`

   Odpowiedź: Klasyczne wyjście komplementarne (typu *push-pull*) składa się z dwóch tranzystorów połączonych szeregowo pomiędzy linią zasilania (V_{DD}) a masą (GND). Stan wysoki realizowany jest poprzez otwarcie tranzystora górnego (PMOS) przy zamkniętym dolnym, natomiast stan niski przez otwarcie tranzystora dolnego (NMOS)

przy zamkniętym górnym. Otwarte klucze wyjściowe posiadają bardzo niską impedancję ($R_{out} \approx 10\ \Omega - 100\ \Omega$).

W przypadku połączenia dwóch takich wyjść, z których jedno próbuje wymusić stan wysoki (V_{DD}), a drugie stan niski (GND), powstaje zamknięta ścieżka zwarcia o znikomej rezystancji. Prąd zwarcia (I_{short}) ograniczany jest wyłącznie przez sumę małych rezystancji kanałów tranzystorów:

$$I_{short} = \frac{V_{DD}}{R_{out1} + R_{out2}}$$

Dla typowych wartości $V_{DD} = 3.3$ V oraz $R_{out1} = R_{out2} = 15\ \Omega$, natężenie prądu wynosi:

$$I_{short} = \frac{3.3\text{ V}}{30\ \Omega} = 110\text{ mA}$$

Maksymalny dopuszczalny prąd ciągły wyjścia pojedynczej bramki logicznej wynosi przeważnie od 4 mA do 20 mA. Płynący prąd zwarciowy przekracza te dopuszczalne limity od kilku- do kilkunastokrotnie. Zgodnie z prawem Joule'a na tranzystorach wyjściowych wydziela się olbrzymia moc w postaci ciepła ($P = I^2 \cdot R_{out}$). Prowadzi to do błyskawicznego przegrzania krzemu, stopienia mikroskopijnych struktur półprzewodnikowych (zniszczenia termicznego złączy) i nieodwracalnego uszkodzenia struktury wewnętrznej układu scalonego.

3. `Co to jest ścieżka krytyczna w układzie cyfrowym i jak wpływa ona na maksymalną częstotliwość pracy zegara w projektach synchronicznych?`

Odpowiedź: Ścieżka krytyczna (*critical path*) to ścieżka w układzie kombinacyjnym charakteryzująca się najdłuższym sumarycznym czasem propagacji sygnału pomiędzy wyjściem dowolnego elementu sekwencyjnego (np. przerzutnika D startowego) a wejściem kolejnego elementu sekwencyjnego (przerzutnika odbiorczego). Na czas ten składa się suma czasów propagacji wszystkich bramek logicznych leżących na tej trasie oraz opóźnienia interkonektów (rezystancja i pojemność połączeń fizycznych).

W układach synchronicznych wszystkie przerzutniki zatrzaskują dane równocześnie na aktywnym zboczu sygnału zegarowego. Aby system działał poprawnie, sygnał zmieniony na wyjściu pierwszego przerzutnika musi przebyć całą drogę przez logikę kombinacyjną (w tym najdłuższą ścieżkę krytyczną) i ustabilizować się na wejściu kolejnego przerzutnika przed nadejściem następnego zbocza zegara, spełniając przy tym wymóg czasu konfiguracji (t_{setup}).

Minimalny dopuszczalny okres zegara $T_{clk,min}$ określa zależność:

$$T_{clk,min} = t_{CQ} + t_{logic,max} + t_{setup}$$

gdzie:

- t_{CQ} – opóźnienie propagacji od zbocza zegara do ustalenia stanu na wyjściu przerzutnika startowego,
- $t_{logic,max}$ – czas propagacji najdłuższej drogi kombinacyjnej (ścieżki krytycznej),

- t_{setup} – minimalny czas konfiguracji przerzutnika odbiorczego.

Maksymalna częstotliwość pracy zegara $f_{clk,max}$ jest odwrotnością tego okresu:

$$f_{clk,max} = \frac{1}{T_{clk,min}}$$

Przekroczenie tej częstotliwości (czyli skrócenie okresu zegara poniżej czasu przejścia przez ścieżkę krytyczną) spowoduje zatrzaśnięcie niestabilnego sygnału, co naruszy czas t_{setup}, wywoła zjawisko metastabilności i doprowadzi do całkowicie błędnego funkcjonowania układu.

4. `Załóżmy, że podłączamy przycisk mechaniczny bezpośrednio do wejścia zegarowego przerzutnika D. Dlaczego układ nie będzie działał stabilnie? Jakie zjawiska fizyczne i logiczne tutaj zachodzą i jak im zapobiec?`

   Odpowiedź: Układ nie będzie działał stabilnie ze względu na dwa zjawiska fizyczno-logiczne:

   (a) **Drgania styków (*contact bounce*)**: Z powodu sprężystości metalu, w momencie wciskania lub puszczania przycisku jego styki nie łączą się natychmiastowo na stałe, lecz zderzają się ze sobą wielokrotnie, powodując serię szybkich zwarć i rozwarć trwającą od kilku do kilkudziesięciu milisekund. Szybki układ cyfrowy (zdolny reagować na sygnały o częstotliwościach megahercowych) zinterpretuje te drgania jako kilkadziesiąt niezależnych zboczy zegarowych. Przerzutnik zmieni więc stan losową liczbę razy podczas jednego fizycznego naciśnięcia.

   (b) **Powolne zbocza i szum**: Przycisk mechaniczny generuje sygnały o łagodnych zboczach. W strefie przejściowej (między V_{IL} a V_{IH}) drobne szumy elektryczne nakładające się na sygnał mogą powodować wielokrotne, fałszywe przełączanie wejścia przerzutnika.

   Metody zapobiegania:

   - **Sprzętowe**:

     (a) Filtrowanie dolnoprzepustowe RC (kondensator eliminuje szybkie szpilki napięcia z drgań styków) połączone z wejściem posiadającym histerezę (np. przerzutnik Schmitta), które formuje strome i czyste zbocza logiczne.

     (b) Zastosowanie przełącznika przełącznego (SPDT) połączonego z asynchronicznym zatrzaskiem SR (zbudowanym z bramek NAND/NOR) – zderzenia styków następują tylko w obrębie jednej pozycji, co zatrzask całkowicie ignoruje po pierwszym kontakcie.

   - **Programowe (*debouncing*)**: W systemach mikroprocesorowych lub FPGA próbkuje się stan przycisku z niską częstotliwością (np. co 10–20 ms) bądź też po wykryciu pierwszego zbocza blokuje się odczyt wejścia na czas wygaszenia drgań styków. Dodatkowo, w strukturach FPGA sygnały zewnętrzne z przycisków powinny być najpierw zsynchronizowane z zegarem systemowym za

pomocą dedykowanych synchronizatorów, a do wyzwalania akcji należy używać jednotaktowych impulsów zezwolenia (*Enable*), zamiast taktować układ bezpośrednio z linii przycisku.

5. `Wykonaj negację liczby` $X = -6$ `zapisanej na 8 bitach w formacie U2. Podaj wynik w postaci binarnej.`

   Odpowiedź: Negacja liczby w kodzie U2 (*Two's Complement*) polega na wyznaczeniu liczby o przeciwnej wartości logicznej ($Y = -X = -(-6) = +6$). Operację tę realizuje się poprzez algorytm: **zaneguj wszystkie bity liczby wejściowej i dodaj 1**.

   Wyznaczmy najpierw postać binarną liczby $X = -6_{10}$ na 8 bitach w U2:

   (a) Liczba $+6_{10}$ w zapisie 8-bitowym: $0000\,0110_2$

   (b) Negacja bitów liczby $+6_{10}$: $1111\,1001_2$

   (c) Dodanie 1: $1111\,1001_2 + 1 = 1111\,1010_2$ (jest to reprezentacja liczby -6_{10} w U2)

   Teraz wykonujemy operację negacji liczby $X = -6_{10} = 1111\,1010_2$:

   (a) Negacja wszystkich bitów liczby X:

   $$\overline{1111\,1010_2} = 0000\,0101_2$$

   (b) Dodanie wartości 1 do zanegowanego ciągu:

   $$0000\,0101_2 + 1 = 0000\,0110_2$$

   Wynik końcowy operacji negacji liczby -6 w formacie U2 wynosi $0000\,0110_2$ (co odpowiada dokładnie wartości $+6_{10}$).

6. `Wyjaśnij pojęcie metastabilności. W jakich sytuacjach w projektach FPGA/ASIC najczęściej dochodzi do jej wystąpienia i jak inżynierowie zabezpieczają przed nią swoje układy?`

   Odpowiedź: Metastabilność to stan przejściowy elementu pamiętającego (np. przerzutnika D), w którym po aktywnym zboczu zegara jego napięcie wyjściowe nie potrafi szybko ustalić się na żadnym z dopuszczalnych poziomów logicznych (0 lub 1). Wyjście wchodzi w stan równowagi chwiejnej (oscyluje wokół wartości progowej $V_{DD}/2$), utrzymując się w nim przez czas losowy, wielokrotnie przekraczający standardowy czas propagacji tranzystorów (t_{CQ}), zanim losowe fluktuacje termiczne nie sprowadzą go do jednego ze stanów stabilnych.

   Sytuacje występowania w FPGA/ASIC: Metastabilność jest konsekwencją naruszenia czasu konfiguracji (t_{setup}) lub podtrzymania (t_{hold}) przerzutnika – czyli sytuacji, gdy sygnał wejściowy zmienia się zbyt blisko aktywnego zbocza zegara taktującego. Najczęstsze przykłady to:

(a) **Przejście sygnałów między różnymi domenami zegarowymi (*Clock Domain Crossing – CDC*)**: Kiedy dane przesyłane są między częściami układu taktowanymi różnymi, asynchronicznymi zegarami. Zbocze zegara czytającego może pojawić się dokładnie w trakcie przełączania sygnału wejściowego.

(b) **Odczyt sygnałów asynchronicznych z otoczenia**: Wprowadzanie sygnałów zewnętrznych (np. przycisków, danych z magistral zewnętrznych bez wspólnego zegara).

Zabezpieczanie przed skutkami metastabilności: Metastabilności nie da się całkowicie wyeliminować (jest to zjawisko stochastyczne), ale można zminimalizować jej prawdopodobieństwo do wartości pomijalnych w praktyce (np. średni czas między awariami MTBF liczony w milionach lat):

- **Dwustopniowe (lub trzystopniowe) synchronizatory**: Najpopularniejsza metoda polegająca na szeregowym połączeniu dwóch przerzutników typu D taktowanych zegarem domeny docelowej. Pierwszy przerzutnik odbiera sygnał asynchroniczny i może wejść w stan metastabilny, ale ma niemal cały okres zegara na ustabilizowanie swojego stanu wyjściowego. Drugi przerzutnik zatrzaskuje już w pełni stabilny i określony stan logiczny.
- **Kolejki FIFO z kodowaniem Graya**: Przy przesyłaniu danych wielobitowych między domenami zegarowymi stosuje się asynchroniczne bufory FIFO, w których wskaźniki adresu zapisu i odczytu są synchronizowane z użyciem kodu Graya. W kodzie tym przy zmianie o jeden zmienia się zawsze tylko jeden bit słowa, co zapobiega interpretacji niepoprawnych wartości przejściowych w przypadku metastabilności pojedynczej linii.
- **Wymuszenie restrykcji czasowych i analiza CDC**: Narzędzia syntezy (np. Vivado, Quartus) analizują ścieżki przechodzenia sygnałów i automatycznie nakładają odpowiednie ograniczenia czasowe na fizyczne rozmieszczenie synchronizatorów w strukturze krzemowej.

13.1 Wymagane oprogramowanie

Do wykonania ćwiczenia wymagany jest NI Multisim, Jupyter-lab i Python 3 z zainstalowanymi modułami:

- `Matplotlib` — do wizualizacji wyników,
- `Numpy` — do podstawowych struktur danych i obliczeń,

14 Autorzy i historia opracowania

- dr inż. Dariusz Tefelski - wersja z 2026r.