

Logika cyfrowa

Wykład 10: pamięć, układy programowalne

Marek Materzok

5 maja 2021

Pamięć

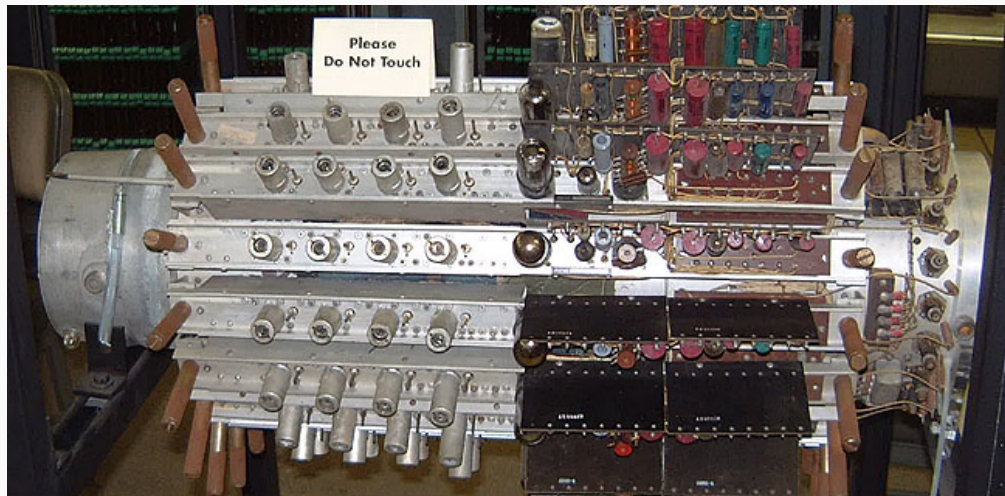
- Zbiór *komórek pamięci* indeksowanych *adresami*
- Tylko niewielka liczba komórek (typowo 1-3) obsługiwana równocześnie (w jednym cyklu układu synchronicznego)
- Układ używający pamięci musi „zlecać” dostępy w kolejnych cyklach

Rodzaje pamięci

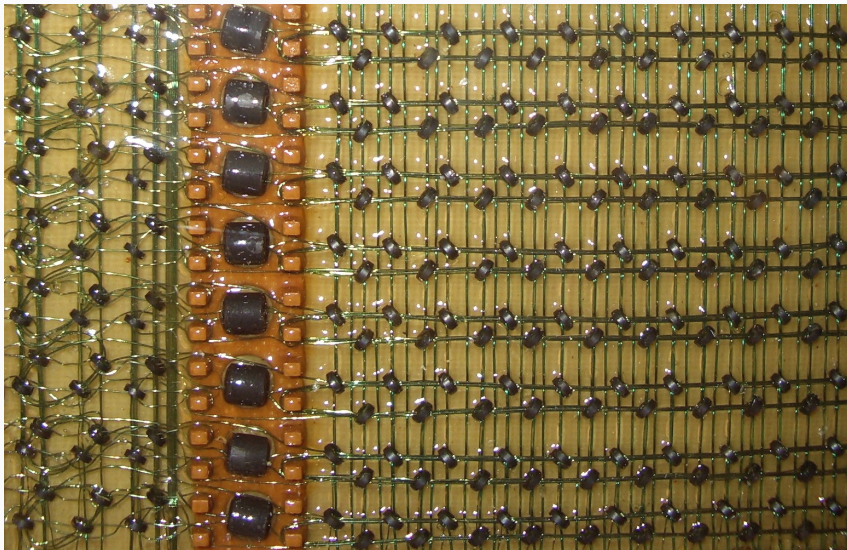
- Ulotne
 - SRAM (Static Random Access Memory)
 - DRAM (Dynamic Random Access Memory)
 - ...
- Nieulotne
 - ROM (Read Only Memory) – tylko do odczytu
 - PROM (Programmable Read Only Memory) – utrudniony, jednorazowy zapis
 - EEPROM (Electrically Erasable Programmable Read Only Memory)
 - MRAM (Magnetoresistive Random Access Memory)
 - PRAM (Phase-change Random Access Memory)
 - ...

Różne zasady działania, podobny interfejs

Pamięć rtęciowa (używana w latach 1947-53)



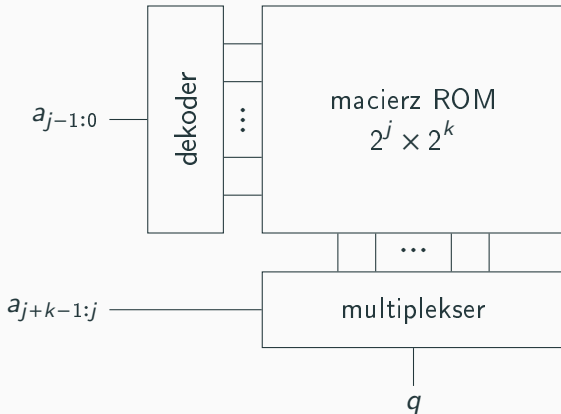
Pamięć ferrytowa (core memory) używana w latach 1955-75



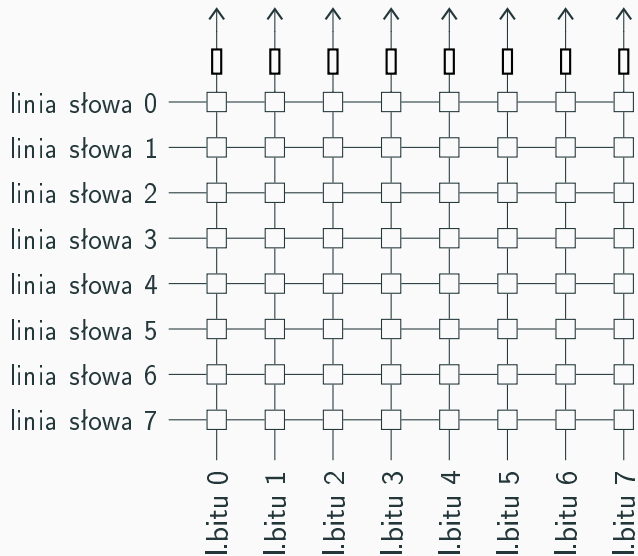
- Odwzorowanie n -bitowych adresów na m -bitowe wartości zakodowane w sposób nieulotny
- Funkcja boolowska $\mathbb{B}^n \rightarrow \mathbb{B}^m$!
- Możliwa implementacja – układ kombinacyjny
 - Wada – konieczność zaprojektowania nowego układu dla zmienionej zawartości
- Inne implementacje podyktowane redukcją kosztów

Organizacja pamięci ROM

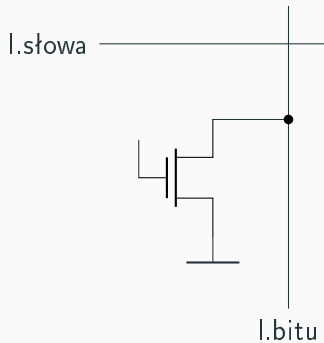
- Bity adresu – dzielone między dekodery i multiplexery
- Dekoder wybiera wiersz
- Multiplexer wybiera kolumnę
- Wyjście q multiplexera – odczytany bit



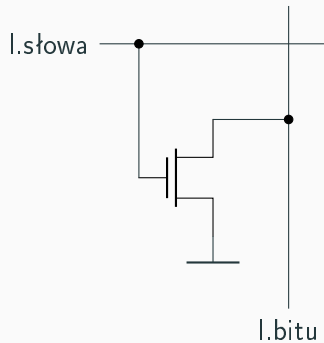
Macierz ROM



Mask ROM



bit 1

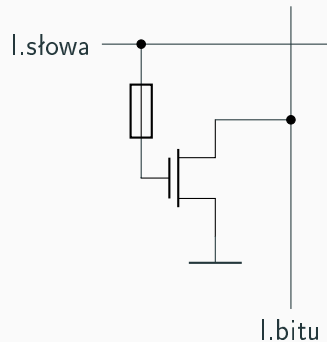


bit 0

Programowane w momencie produkcji układu

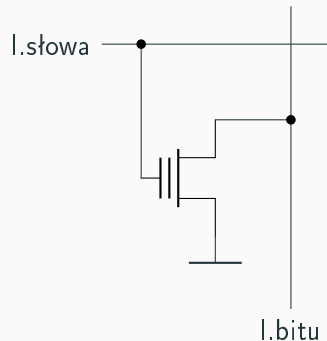
Programowalny ROM (PROM)

- W komórce element przepalany prądem
- Programowanie po wyprodukowaniu, przed montażem (np. podwyższonym napięciem)



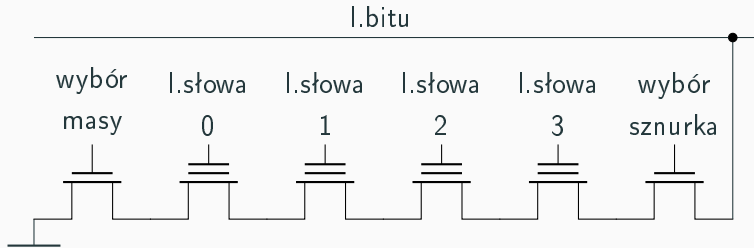
Kasowalny PROM (EPROM, EEPROM)

- Tranzystor z pływającą bramką
- Programowanie po wyprodukowaniu podwyższonym napięciem (tunelowanie)
- Kasowanie:
 - Ultrafioletem (EPROM)
 - Podwyższonym napięciem (EEPROM)



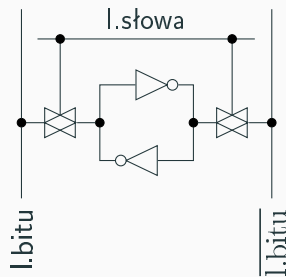
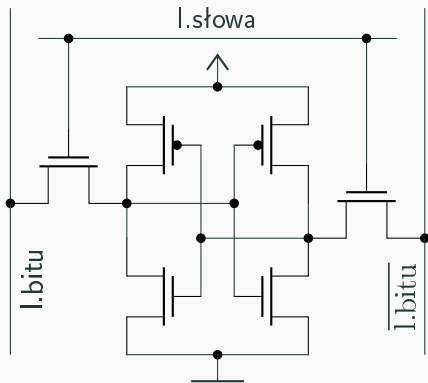
Pamięć flash

- NOR flash – EEPROM kasowany blokami (np. 4k)
- NAND flash – zwiększona gęstość przez połączenie tranzystorów w „sznurki”



Pamięć SRAM (statyczna)

Komórka – uproszczony przerzutnik asynchroniczny SR:



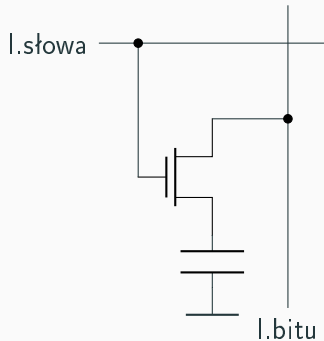
Pamięć DRAM (dynamiczna)

- Bit zapisany jako ładunek kondensatora
- Odczyt destruktywny (rozładowuje kondensator)
- Samouptywność, wymagane regularne odświeżanie

Prezentacja działania SRAM i DRAM:

<https://www.falstad.com/circuit/>

Schematy / Układy sekwencyjne



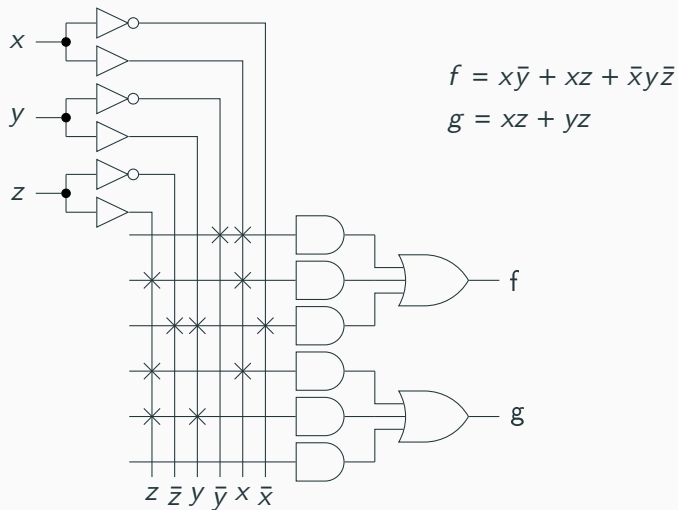
Układy programowalne

Rodzaje układów programowalnych (PLD)

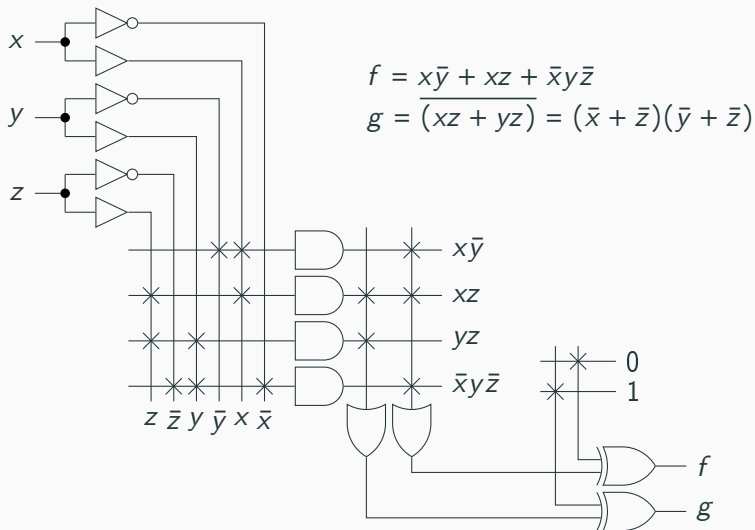
PLD – Programmable Logic Devices

- PROM – zapis funkcji boolowskiej w formie tabelki
- PAL – zapis funkcji w DNF, układ AND-OR o stałej strukturze
- PLA – zapis funkcji w DNF, programowalny układ AND-OR
- CPLD – układ programowalny z wbudowanymi przerzutnikami o prostej, gruboziarnistej strukturze
- FPGA – układ programowalny z wbudowanymi przerzutnikami, o strukturze drobnoziarnistej siatki, logiką kombinacyjną w postaci LUT (Look-Up Table) RAM

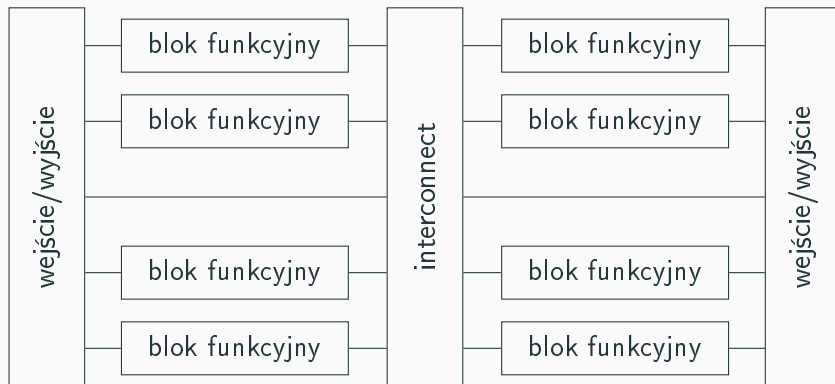
PAL – Programmable Array Logic



PLA – Programmable Logic Array

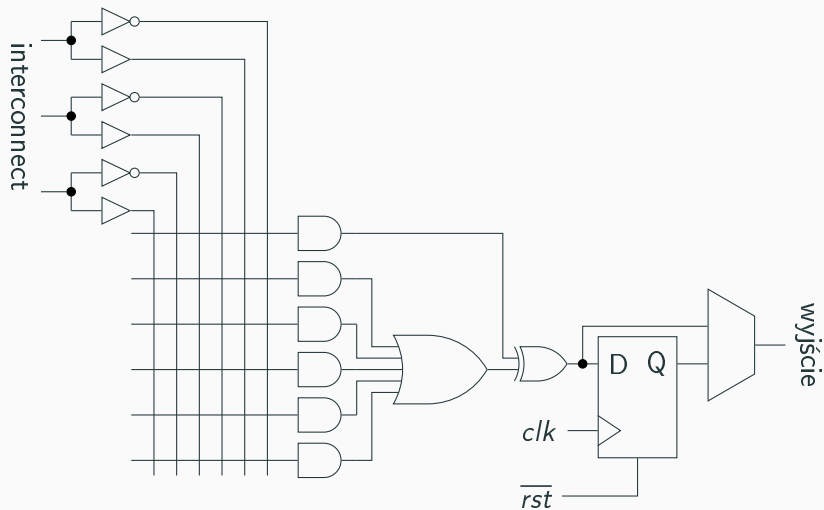


CPLD – Complex Programmable Logic Device

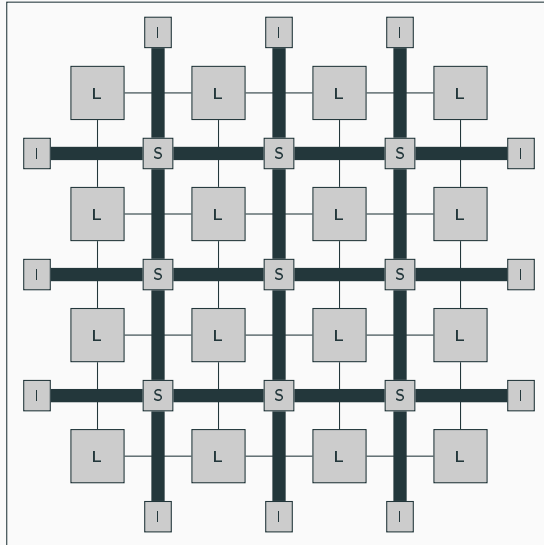


Bloki funkcyjne typowo zawierają kilkanaście makrokomórek.

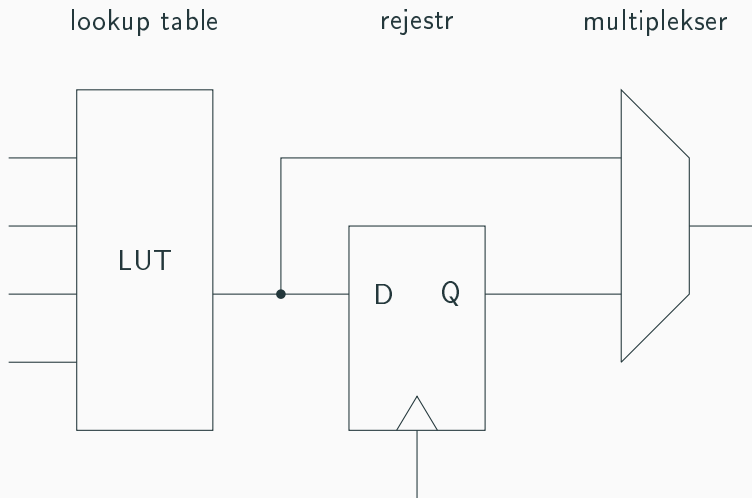
Makrokomórka CPLD – przykład



FPGA – Field Programmable Grid Array



Blok logiczny FPGA – przykład



Pamięć w SystemVerilogu

- Zakres napisany za identyfikatorem definiuje tablicę:

```
logic [7:0] mem [0:255]; // 256 bajtów
```

- Tablice w Verilogu są jednowymiarowe.
SystemVerilog dopuszcza tablice wielowymiarowe, ale Yosys ich nie obsługuje ☹
- Tablice w SystemVerilogu wykorzystuje się do opisywania pamięci **zgodnie z zaprezentowanymi później idiomami**.

Elementy pamięciowe można inicjalizować w bloku `initial`:

```
logic x;  
initial x = 0;
```

Elementy pamięciowe można inicjalizować w bloku `initial`:

```
logic x;  
initial x = 0;
```

- Inicjalizacja dla przerzutników (typowo) **nie jest syntezywalna**. Należy wprowadzić sygnał *reset*, gdy inicjalizacja jest konieczna.

Elementy pamięciowe można inicjalizować w bloku `initial`:

```
logic x;  
initial x = 0;
```

- Inicjalizacja dla przerzutników (typowo) **nie jest syntezywalna**. Należy wprowadzić sygnał *reset*, gdy inicjalizacja jest konieczna.
- Inicjalizacja dla pamięci ROM/RAM dla niektórych technologii (np. niektóre FPGA) **jest syntezywalna**.

Elementy pamięciowe można inicjalizować w bloku `initial`:

```
logic x;  
initial x = 0;
```

- Inicjalizacja dla przerzutników (typowo) **nie jest syntezywalna**. Należy wprowadzić sygnał *reset*, gdy inicjalizacja jest konieczna.
- Inicjalizacja dla pamięci ROM/RAM dla niektórych technologii (np. niektóre FPGA) **jest** syntezywalna.
- Pamięci **nie można** resetować! Użycie sygnału *reset* do czyszczenia pamięci spowoduje wygenerowanie osobnego rejestru dla każdej komórki pamięci.

Przykład – zerowanie:

```
integer i;  
logic [7:0] mem [0:255];  
initial  
    for (i = 0; i < 256; i = i+1)  
        mem[i] = 0;
```

Inicjalizacja plikiem obrazu

```
logic [7:0] mem [0:255];  
initial $readmemh("image.hex", mem); // szesnastkowy  
// albo:  
initial $readmemb("image.bin", mem); // binarny
```

Format pliku:

- Liczby szesnastkowe/binarne, jedna liczba na komórkę
- Oddzielone białymi znakami lub komentarzami
- Adresy postaci @hh...h – liczba szesnastkowa poprzedzona małpą

Dostęp do pamięci

- Każda operacja dostępu do pamięci jest rozumiana jako dodatkowy **port** (do odczytu lub zapisu).
 - SystemVerilog nie zabrania wprowadzić dowolnie wielu portów, ALE rzeczywiste pamięci mają zwykle **jeden do dwóch**.
- Odczyty w blokach kombinacyjnych definiują **odczyt asynchroniczny** (niezależny od zegara).
`assign out = mem[addr];`
- Odczyty i zapisy w blokach `always_ff` definiują **odczyt lub zapis synchroniczny** (odbywający się po zboczu zegarowym).
`always_ff @(posedge clk) mem[addr] <= in;`


```
module memory(  
    input [5:0] addr,  
    output [3:0] out  
);  
    logic [3:0] mem [0:63];  
    assign out = mem[addr];  
    initial $readmemh("image.hex", mem);  
endmodule
```

Pamięć RAM z asynchronicznym odczytem

```
module memory(  
    input wr, clk,  
    input [5:0] addr,  
    input [3:0] in,  
    output [3:0] out  
);  
    logic [3:0] mem [0:63];  
    assign out = mem[addr];  
    always_ff @(posedge clk)  
        if (wr) mem[addr] <= in;  
endmodule
```

Pamięć RAM z synchronicznym odczytem (jeden port odczyt/zapis)

```
module memory(  
    input wr, clk,  
    input [5:0] addr,  
    input [3:0] in,  
    output [3:0] out  
);  
    logic [3:0] mem [0:63];  
    always_ff @(posedge clk)  
        if (!wr) out <= mem[addr];  
    always_ff @(posedge clk)  
        if (wr) mem[addr] <= in;  
endmodule
```

Pamięć RAM z s.o. dwuportowa (R+W)

```
module memory(  
    input rd, wr, clk,  
    input [5:0] rdaddr, wraddr,  
    input [3:0] in,  
    output [3:0] out  
);  
    logic [3:0] mem [0:63];  
    always_ff @(posedge clk)  
        if (rd) out <= mem[rdaddr];  
    always_ff @(posedge clk)  
        if (wr) mem[wraddr] <= in;  
endmodule
```

Pamięć RAM asynchroniczna trójportowa (R+R+W)

```
module memory(  
    input wr, clk,  
    input [5:0] rdaddr1, rdaddr2, wraddr,  
    input [3:0] in,  
    output [3:0] out1, out2  
);  
    logic [3:0] mem [0:63];  
    assign out1 = mem[rdaddr1];  
    assign out2 = mem[rdaddr2];  
    always_ff @(posedge clk)  
        if (wr) mem[wraddr] <= in;  
endmodule
```

Pamięć RAM asynchroniczna trójportowa (R+W+W)

```
module memory(  
    input wr1, wr2, clk,  
    input [5:0] rdaddr, wraddr1, wraddr2,  
    input [3:0] in1, in2,  
    output [3:0] out  
);  
    logic [3:0] mem [0:63];  
    assign out = mem[rdaddr];  
    always_ff @(posedge clk) begin  
        if (wr1) mem[wraddr1] <= in1;  
        if (wr2) mem[wraddr2] <= in2;  
    end  
endmodule
```

Uwaga – niezdefiniowane zachowanie gdy wraddr1 == wraddr2

- Odczyt może być *asynchroniczny* (w bloku `assign` lub `always_comb`) lub *synchroniczny* (`always_ff`).
- Zapisy tylko synchroniczne!
- Każdy odczyt i zapis z tablicy oznacza *wprowadzenie dodatkowego portu pamięci*.
 - Uwaga: w przypadku wspólnego adresu do odczytu i zapisu, mówi się o jednym porcie do odczytu lub zapisu.
- Pamięci o więcej niż dwóch portach są w praktyce **rzadko spotykane**. Typowa pamięć jest jednoportowa.
- **Zalecam** specyfikowanie pamięci w osobnym module.

Projektowanie układów z pamięcią

Przykład 1 – układ sumujący zawartość pamięci

- Będziemy sumować zawartość pamięci ROM zawierającej 16 liczb 12-bitowych
- 1 port do odczytu (asynchronicznego): 4 bity adresu, 12 bitów wyjścia
- Ponieważ możliwy jest odczyt jednej komórki w cyklu, potrzebny jest stan, aby:
 - Generować kolejne adresy (4 bity)
 - Wiedzieć, że skończyliśmy (1 bit)
 - Pamiętać wynik pośredni (12 bitów)
- Pamięć ROM z asynchronicznym odczytem można traktować jak układ kombinacyjny
- Wejścia i wyjścia układu:
 - Wejście 1-bitowe `start` – rozpoczęcie sumowania
 - Wyjście 1-bitowe `fin` – koniec sumowania (wynik poprawny)
 - Wyjście 12-bitowe `out` – wynik

Projektowanie układu dla przykładu 1 – transfer rejestrów

- Potrzebne rejestry:
 - Akumulator 12-bitowy acc – oblicza sumę
 - Licznik 4-bitowy cnt – odlicza adresy ROM
 - Bit końca fin
- Model odczytu asynchronicznego pamięci:
 - $val_t = mem[addr_t]$

Projektowanie układu dla przykładu 1 – transfer rejestrów

Odczyt pamięci: $val_t = mem[cnt_t]$ (w module pamięci)

Kiedy $start_{t-1} = 1$:

- $cnt_t \leftarrow 0$
- $acc_t \leftarrow 0$
- $fin_t \leftarrow 0$

Kiedy $start_{t-1} = 0$ i $fin_{t-1} = 0$:

- $cnt_t \leftarrow cnt_{t-1} + 1$
- $acc_t \leftarrow acc_{t-1} + val_{t-1}$
- $fin_t \leftarrow cnt_{t-1} = 15$

W przeciwnym wypadku brak zmian wartości rejestrów.

Przykład 1 – SystemVerilog

```
module sum_rom(  
    input clk, start,  
    output logic fin,  
    output [11:0] out  
);  
    logic [3:0] cnt;  
    logic [11:0] acc, val;  
    // ROM asynchroniczny  
    rom mem(cnt, val);  
    assign out = acc;  
  
    always_ff @(posedge clk)  
        if (start) begin  
            cnt <= 4'b0;  
            acc <= 12'b0;  
            fin <= 1'b0;  
        end else if (!fin) begin  
            cnt <= cnt + 1;  
            acc <= acc + val;  
            fin <= cnt == 15;  
        end  
endmodule
```

Przykład 1 – wykonanie

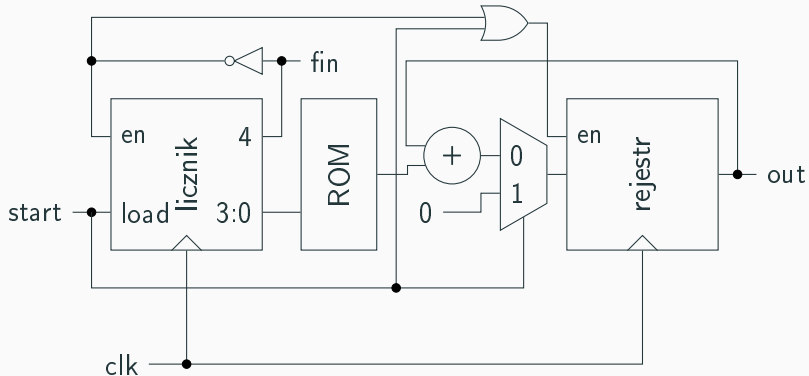
Przyjmijmy, że pamięć zawiera pierwsze 16 elementów ciągu Fibonacciego.

cykl	start	cnt	val	acc	fin
1	1	x	x	x	x
2	0	0	0	0	0
3	0	1	1	0	0
4	0	2	1	1	0
5	0	3	2	2	0
⋮	⋮	⋮	⋮	⋮	⋮
16	0	14	377	609	0
17	0	15	610	986	0
18	0	0	0	1596	1
19	0	0	0	1596	1

Projektowanie układu dla przykładu 1 – inne podejście „układowe”

- Adresy: licznik 5-bitowy
 - najstarszy bit oznacza koniec sumowania
- Wynik pośredni: sumator wielocyklowy
 - Inicjalizacja synchroniczna na 0

Przykład 1 – układ sumujący zawartość pamięci



Przykład 1 – SystemVerilog, podejście „układowe”

```
module sum_rom(  
    input clk, start,  
    output fin,  
    output [11:0] out  
);  
    logic [3:0] addr;  
    logic [11:0] val;  
    logic en;  
    assign en = !fin;  
    counter cnt(clk, en, start, {fin, addr});  
    register re(clk, start || en,  
        start ? 12'b0 : out + val, out);  
    rom mem(addr, val);  
endmodule
```


Przykład 2 – układ sumujący ROM z odczytem synchronicznym

- Pamięć synchroniczna jest **taktowana zegarem**
- Wynik dostępu do pamięci dostępny w **kolejnym** cyklu zegara (po zboczu zegarowym)
- Model: dodatkowy rejestr za pamięcią z odczytem asynchronicznym
 - $val_t \leftarrow mem[addr_{t-1}]$
- Należy skonstruować układ, aby „odebrał” wartość cykl po wykonaniu dostępu
- **Spostrzeżenie:** jeśli wyślemy adres cykl wcześniej, logika sterujące może pozostać bez zmian

Projektowanie układu dla przykładu 2 – transfer rejestrów

Nową wartość adresu możemy obliczyć kombinacyjnie:

- $\text{newcnt}_t = \text{start}_t \vee \text{fin}_t ? 0 : \text{cnt}_t + 1$

Pamięć adresujemy nowym adresem:

- $\text{val}_t \leftarrow \text{mem}[\text{newcnt}_{t-1}]$ (w module pamięci)

Aktualizujemy licznik nowym adresem:

- $\text{cnt}_t \leftarrow \text{newcnt}_{t-1}$

Logika dla acc i fin bez zmian

Przykład 2 – SystemVerilog

```
module sum_rom_sync(  
    input clk, start,  
    output logic fin,  
    output [11:0] out  
);  
    logic [3:0] cnt, newcnt;  
    logic [11:0] acc, val;  
    // ROM synchroniczny  
    rom_sync mem(clk, newcnt, val);  
    assign newcnt =  
        start || fin ? 0 : cnt + 1;  
    assign out = acc;
```

```
    always_ff @(posedge clk)  
        cnt <= newcnt;  
    always_ff @(posedge clk)  
        if (start) begin  
            acc <= 12'b0;  
            fin <= 1'b0;  
        end else if (!fin) begin  
            acc <= acc + val;  
            fin <= cnt == 15;  
        end  
endmodule
```

Przykład 2 – wykonanie

Zauważmy, że `val` uzyskuje wartość dla `newcnt` po cyklu.

cykl	start	cnt	newcnt	val	acc	fin
1	1	x	0	x	x	x
2	0	0	1	0	0	0
3	0	1	2	1	0	0
4	0	2	3	1	1	0
5	0	3	4	2	2	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮
16	0	14	15	377	609	0
17	0	15	0	610	986	0
18	0	0	0	0	1596	1
19	0	0	0	0	1596	1

Przykład 3 – układ obliczający wartości bezwzględne

- Będziemy zamieniać wszystkie wartości zapisane w RAM (16 liczb 8-bitowych ze znakiem) na ich wartość bezwzględną.
- 1 port do odczytu asynchronicznego: 4 bity adresu, 8 bitów wyjścia
- 1 port do zapisu: 8 bitów wejścia, 1 bit enable, adres wspólny z odczytem
- Zapis jest synchroniczny: następuje na zboczu zegarowym (tak jak zapisy do rejestrów)
- Wejścia i wyjścia układu:
 - Wejście 1-bitowe `start` – rozpoczęcie sumowania
 - Wyjście 1-bitowe `fin` – koniec sumowania (wynik poprawny)

Projektowanie układu dla przykładu 3 – transfer rejestrów

- Potrzebne rejestry:
 - Licznik 4-bitowy cnt – odlicza adresy RAM
 - Bit końca fin
- Wyniki – wartości bezwzględne – będą zapisywane w pamięci w kolejnym cyklu po ich odczytaniu. Model zapisów synchronicznych:
 - $mem_t[addr_{t-1}] \leftarrow data_{t-1}$

Projektowanie układu dla przykładu 3 – transfer rejestrów

Odczyt pamięci: $val_t = mem_t[cnt_t]$ (w module pamięci)

Zapisujemy pamięć tylko, gdy odczytana liczba jest ujemna:

- $wr_t = val_t < 0$

Zapisywana wartość jest przeciwna do odczytanej:

- $data_t = -val_t$

Zapis pamięci: $mem_t[cnt_{t-1}] \leftarrow data_{t-1}$ (w module pamięci)

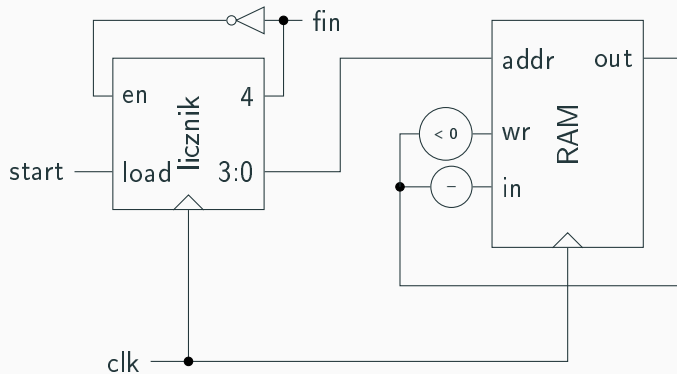
Logika dla cnt i fin jak w przykładzie 1.

Przykład 3 – SystemVerilog

```
module abs_ram(  
    input clk, start,  
    output logic fin,  
    output signed [7:0] val  
);  
    logic [3:0] cnt;  
    logic [7:0] data = -val;  
    logic wr = val < 0;  
    ram mem(clk, wr, cnt,  
        data, val);
```

```
    always_ff @(posedge clk)  
        if (start) begin  
            cnt <= 4'b0;  
            fin <= 1'b0;  
        end else if (!fin) begin  
            cnt <= cnt + 1;  
            fin <= cnt == 15;  
        end  
endmodule
```


Przykład 3 – podejście „układowe”



Przykład 3 – SystemVerilog, podejście „układowe”

```
module abs_ram(  
    input clk, start,  
    output fin,  
    output signed [7:0] val  
);  
    logic [3:0] addr;  
    logic en;  
    assign en = !fin;  
    counter cnt(clk, en, start, {fin, addr});  
    ram mem(clk, val < 0, addr, -val, val);  
endmodule
```