Logika cyfrowa

Wykład 15: potokowa implementacja RISC V

Marek Materzok

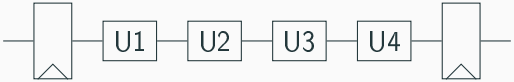
10 czerwca 2024

Przetwarzanie potokowe (pipelining)

Przetwarzanie potokowe – motywacja

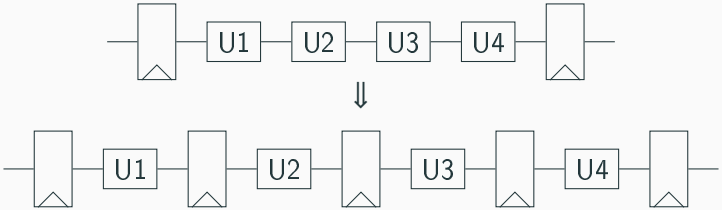
- W układach synchronicznych długość ścieżki krytycznej musi być krótsza niż okres sygnału zegara
- Gdy ścieżka krytyczna jest długa, nie można szybko taktować układu
- Duża część układu jest w danej chwili czasu bezczynna

Uproszczenie – wiele podukładów kombinacyjnych połączonych w szereg:



Przetwarzanie potokowe

Można poprawić wydajność stosując **przetwarzanie potokowe** (pipelining):



Analiza wydajności

Załóżmy że t_p to czas propagacji ścieżki krytycznej układu U, a t_u to czas ustalania rejestru.

W układzie potokowym opóźnienie wydłuża się:

- $t_1 = 4t_p + t_u$
- $t_2 = 4t_p + 4t_u$

Ale rośnie przepustowość – układ potokowy może przetwarzać *wiele danych w tym samym czasie*:

- $f_1 = \frac{1}{t_1} = \frac{1}{4t_p + t_u}$
- $f_2 = \frac{4}{t_2} = \frac{4}{4t_p + 4t_u} = \frac{1}{t_p + t_u}$

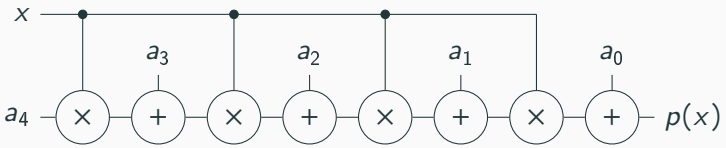
Analiza wydajności – ilustracja

Przykład – schemat Hornera

Schemat Hornera to algorytm obliczania wartości wielomianu:

$$\begin{aligned} p(x) =& a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_n x^n \\ =& a_0 + x(a_1 + x(a_2 + x(a_3 + \cdots + x(a_{n-1} + x a_n)\cdots))) \end{aligned}$$

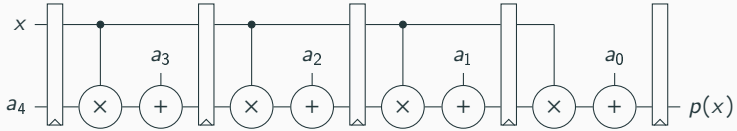
Schemat Hornera można zaimplementować w sprzęcie przez wielokrotne zastosowanie układu mnożąco-dodającego:



Gotowe układy mnożąco-dodające są dostępne na wielu FPGA.

Przykład – schemat Hornera z potokiem

Schemat Hornera można obliczać potokowo w poniższy sposób:



Wartość x musi być przechowywana w rejestrach potoku, aby możliwe było obliczanie kilku wartości w tym samym czasie.

Przykład – schemat Hornera w SystemVerilogu

Aby zasymulować opóźnienia występujące w układzie, dla potrzeb przykładu zastosujemy modele bramkowe układów arytmetycznych.

```
module halfadder(
    output o, c,
    input a, b,
);
    assign o = a ^ b;
    assign c = a & b;
endmodule
```

```
module fulladder(
    output o, co,
    input a, b, c
);
    logic t, c1, c2;
    halfadder ha1(t, c1, a, b);
    halfadder ha2(o, c2, t, c);
    assign co = c1 | c2;
endmodule
```

Przykład – schemat Hornera w SystemVerilogu

```
module adder#(parameter W = 12) (
    output [W-1:0] o,    output cn,
    input  [W-1:0] a, b, input  c0
);
    logic [W:0] c;
    assign c[0] = c0;
    assign cn = c[W];
    genvar i;
    for (i = 0; i < W; i = i+1)
        fulladder fa1(o[i], c[i+1], a[i], b[i], c[i]);
endmodule
```

Przykład – schemat Hornera w SystemVerilogu

```
module multiplier#(parameter W = 12) (
    output [W-1:0] o,
    input [W-1:0] a, b
);
    logic [W-1:0] ab[W-1:0];
    logic [W-1:0] ps[W-1:0];
    genvar i;
    for (i = 0; i < W; i = i+1)
        assign ab[i] = a & {W{b[i]}};
    assign ps[0] = ab[0];
    assign o = ps[W-1];
    for (i = 1; i < W; i = i+1)
      adder#(12-i) a(ps[i][W-1:i], , ps[i-1][W-1:i], ab[i], 0);
    for (i = 1; i < W; i = i+1)
      assign ps[i][i-1:0] = ps[i-1][i-1:0];
endmodule
```

Przykład – schemat Hornera w SystemVerilogu

```
module adder_multiplier#(parameter W = 12) (
    output [W-1:0] o,
    input [W-1:0] a, b, c
);
    logic [W-1:0] h;
    multiplier#(W) m(h, a, b);
    adder#(W) s(o, , h, c, 0);
endmodule
```

Przykład – schemat Hornera w SystemVerilogu

```
module horner_comb
#(parameter W = 12, A0 = 12, A1 = 3, A2 = 1, A3 = 1, A4 = 1) (
    input clk,
    output logic [W-1:0] o,
    input [W-1:0] x
);
    logic [W-1:0] t[0:4];
    logic [W-1:0] a[0:4];
    logic [W-1:0] xx;
    assign a[0] = A0, a[1] = A1, a[2] = A2, a[3] = A3, a[4] = A4;
    assign t[4] = a[4];
    always_ff @(posedge clk) xx <= x;
    always_ff @(posedge clk) o <= t[0];
    genvar i;
    for (i = 0; i < 4; i = i+1)
        adder_multiplier#(W) am(t[i], t[i+1], xx, a[i]);
endmodule
```

Przykład – schemat Hornera z potokiem w SystemVerilogu

```
module horner_pipeline
#(parameter W = 12, A0 = 12, A1 = 3, A2 = 1, A3 = 1, A4 = 1) (
    input clk,
    output [W-1:0] o,
    input [W-1:0] x
);
    logic [W-1:0] t[0:4], a[0:4];
    (* mem2reg *) logic [W-1:0] tp[0:4], xp[0:3];
    assign a[0] = A0, a[1] = A1, a[2] = A2, a[3] = A3, a[4] = A4;
    assign t[4] = a[4];
    always_ff @(posedge clk) xp[3] <= x;
    assign o = tp[0];
    genvar i;
    for (i = 0; i <= 4; i = i+1)
      always_ff @(posedge clk) tp[i] <= t[i];
    for (i = 0; i < 3; i = i+1)
      always_ff @(posedge clk) xp[i] <= xp[i+1];
    for (i = 0; i < 4; i = i+1)
      adder_multiplier#(W) am(t[i], tp[i+1], xp[i], a[i]);
endmodule
```

Implementacja potokowa RISC V

Implementacja potokowa – idea

- Podobnie jak w implementacji wielocyklowej, wykonanie jednej instrukcji zostanie rozbite na wiele cykli
- Potokowa budowa ścieżki danych umożliwi przetwarzanie wielu instrukcji w tym samym czasie
- Ze względu na możliwe zależności między instrukcjami ścieżka sterowania będzie zarządzać pracą potoku
- Architektura harwardzka – jak w implementacji jednocyklowej

Budowa potoku

Pięć stopni potoku, wykonujących różne funkcje:

- Fetch – odczyt instrukcji z pamięci kodu,
- Decode – dekodowanie instrukcji i odczyt rejestrów,
- Execute – wykonanie operacji w ALU,
- Memory – odczyt lub zapis w pamięci danych,
- Writeback – zapis w rejestrze.

Każdy stopień używa innej części układu.

Potok a instrukcje

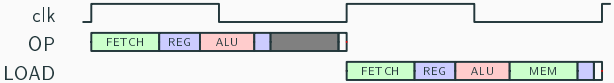
Większość instrukcji nie używa części stopni potoku:

instr	F	D	E	M	W
OP	✓	✓	✓		✓
OP-IMM	✓	✓	✓		✓
BRANCH	✓	✓	✓		
JAL	✓	✓			✓
JALR	✓	✓	✓		✓
LOAD	✓	✓	✓	✓	✓
STORE	✓	✓	✓	✓	
LUI	✓	✓			✓
AUIPC	✓	✓	✓		✓

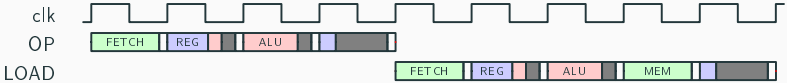
Nieużywany stopień jest dla takiej instrukcji bezczynny.

Aspekty wydajnościowe

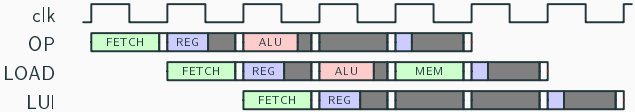
Procesor jednocyklowy:



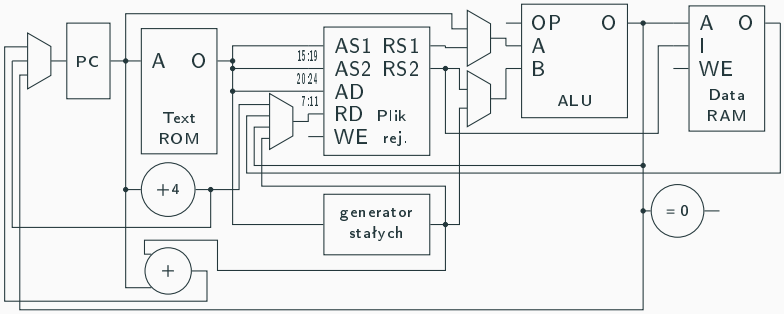
Procesor wielocyklowy:



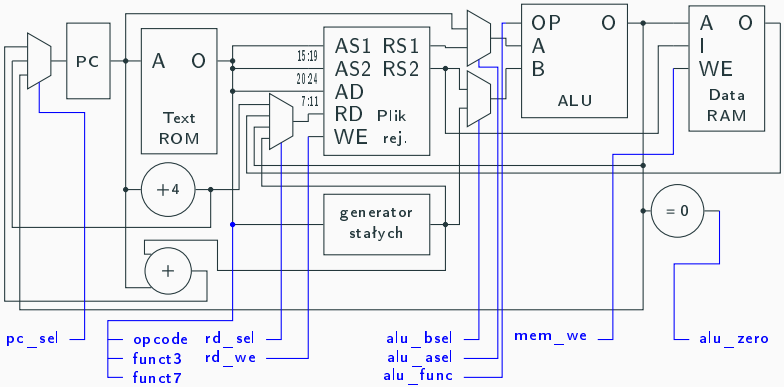
Procesor potokowy:



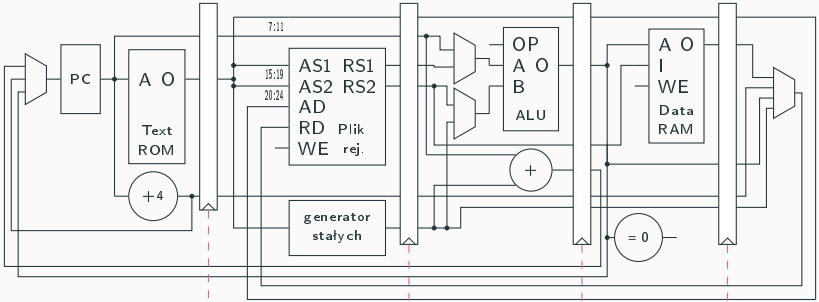
Ścieżka danych – procesor jednocyklowy



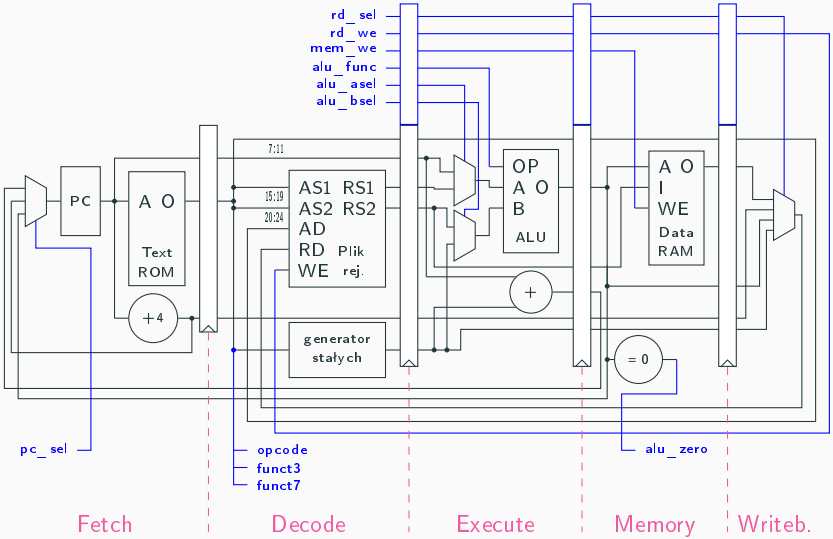
Ścieżka danych – procesor jednocyklowy



Ścieżka danych – z potokiem



Ścieżka danych – z potokiem

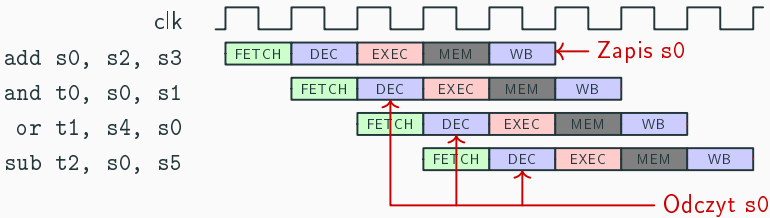


Hazardy w potoku

- Sytuacje, gdy instrukcja zależy od wyniku, który jeszcze nie został obliczony
- Rodzaje:
  - Hazard danych – wartość jeszcze nie zapisana w pliku rejestrów
  - Hazard sterowania – jeszcze nie wiadomo, którą instrukcję wykonać jako kolejną
- Wymagają interwencji, aby instrukcje wykonały się poprawnie

Hazard danych

Problem – rejestr s0 odczytywany, zanim aktualna wartość zostaje zapisana:

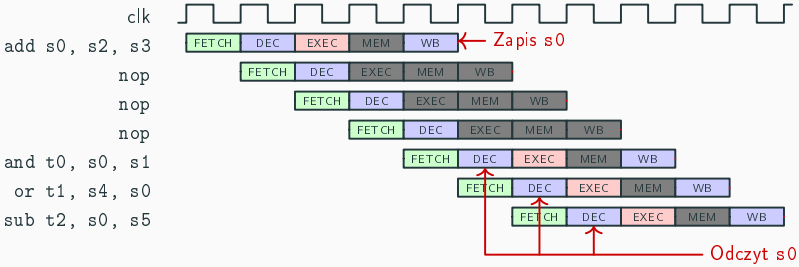


Obsługa hazardów danych

- Dodanie pustych instrukcji `nop` przez kompilator
- Zmiana kolejności instrukcji przez kompilator
  wymaga kompilatora znającego mikroarchitekturę!
- *Stalling* – zatrzymanie potoku w czasie pracy
- *Forwarding* – przekazanie danych z pominięciem fragmentów potoku

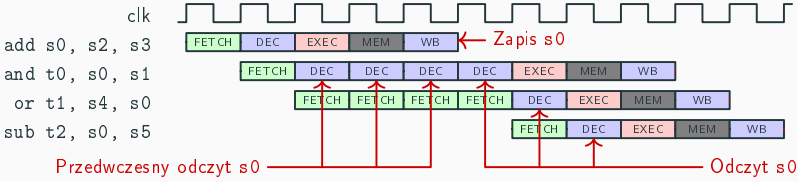
Dodanie instrukcji `nop`

Zamiast instrukcji `nop` kompilator może wstawić inne, potrzebne instrukcje, o ile nie wywołują hazardów:



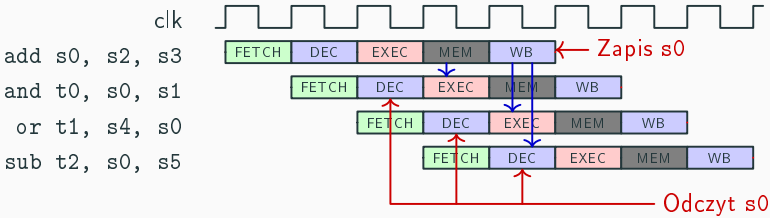
Stalling

- Potok do stopnia Decode jest zatrzymywany
- Pozostałe stopnie pracują normalnie



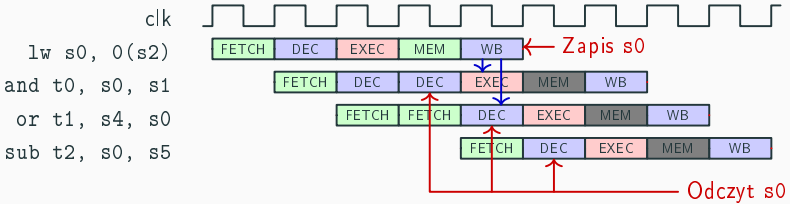
Forwarding

- Po stopniu Execute wynik jest już dostępny w rejestrach potoku
- Modyfikując ścieżkę danych, można przekazać wynik bezpośrednio do potrzebnego stopnia



Sam forwarding nie wystarcza

- Instrukcja LOAD generuje wynik dopiero w stopniu Memory
- Dla niektórych kombinacji instrukcji stalling jest konieczny:



Hazard sterowania

- Instrukcja BRANCH
  wynik porównania i adres skoku znany dopiero w MEM
- Instrukcje JAL, JALR
  adres skoku znany dopiero w MEM
- Obsługa hazardów sterowania:
  - Stalling
  - Wykonywanie spekulatywne
    - w razie pomyłki, potok musi być wyczyszczony
    - predyktory skoków zmniejszają ryzyko pomyłki

Wykonywanie spekulatywne – przykład

- Instrukcje pod adresami 24, 28, 2C załadowane zbędnie
- Muszą być usunięte z potoku zanim zmienią stan architekturalny (rejestry lub pamięć)