Logika cyfrowa

Wykład 5: układy kombinacyjne w SystemVerilogu

Marek Materzok

24 marca 2021

Abstrakcje kombinacyjne

Dlaczego abstrakcje kombinacyjne?

Specyfikowanie układów na poziomie bramek ma ograniczenia:

- Żmudne i czasochłonne
- Łatwo o pomyłkę
- Symulacja jest nieefektywna

Rozwiązanie: podniesienie poziomu abstrakcji!

Abstrakcje kombinacyjne

Poznane wcześniej abstrakcje:

- Wyrażenia algebry Boole'a
- Wektory bitowe i operacje wektorowe
- Wyrażenie warunkowe (multiplekser)
- Operatory arytmetyczne
- Funkcje

Wyrażenia algebry Boole'a

Opis operacji logicznych w formie algebraicznej, zamiast ręcznego specyfikowania pojedynczych bramek.

Przykładowo, zamiast poniższego kodu:

```
logic na, c1, c2;
not (na, a);
and (c1, a, b), (c2, na, c);
or (o, c1, c2);
```

Można napisać:

```
assign o = (a & b) | (~a & c);
```

Wektory bitowe i operacje wektorowe

Specyfikowanie operacji na grupach powiązanych bitów, zamiast na każdym bicie osobno.

Przykładowo, zamiast poniższego kodu:

```
assign o7 = i3, o6 = i2, o5 = i1, o4 = i0,
       o3 = i7, o2 = i6, o1 = i5, o0 = i4;
```

Można napisać:

```
assign o = {i[3:0], i[7:4]};
```

Albo:

```
assign o = {i[0+:4], i[4+:4]};
```

Wyrażenie warunkowe

Zwięzłe specyfikowanie multiplekserów.

Przykładowo, zamiast poniższego kodu:

```
assign o = (a && b) || (!a && c);
```

Można napisać:

```
assign o = a ? b : c;
```

Operatory arytmetyczne

Zwięzłe specyfikowanie układów arytmetycznych.

Przykładowo, zamiast poniższego kodu:

```
serialadder#(8) add(o, a, b[4+:4]);
```

Można napisać:

```
assign o = a + (b >> 4);
```

Uwaga – brak kontroli nad użytą implementacją! W praktyce unikać dzielenia, modulo i potęgowania.

Funkcje

Współdzielenie kodu w ramach modułu.

Przykładowo, zamiast poniższego kodu:

```
assign {t, c1} = {a^b, a&b};
assign {o, c2} = {t^c, t&c};
```

Można napisać:

```
function [1:0] ha(input a, b);
    ha = ({a^b, a&b});
endfunction
assign {t, c1} = ha(a, b);
assign {o, c2} = ha(t, c);
```

Specyfikacja proceduralna

Bloki always_comb

Specyfikowanie układów kombinacyjnych za pomocą *kodu*:

```
always_comb begin
    s  = a ^ b ^ c;
    co = ab | ac | bc;
end
```

Idea: kod w bloku `always_comb` opisuje, jak uaktualnić wyjścia *zawsze* gdy zmienią się wartości powiązanych wejść.

Bloki always_comb

Specyfikowanie układów kombinacyjnych za pomocą *kodu*:

```
always_comb begin
    s  = a ^ b ^ c;
    co = ab | ac | bc;
end
```

Idea: kod w bloku `always_comb` opisuje, jak uaktualnić wyjścia *zawsze* gdy zmienią się wartości powiązanych wejść.

Pułapka: nie każdy kod reprezentuje układ kombinacyjny!

Bloki always_comb

Specyfikowanie układów kombinacyjnych za pomocą *kodu*:

```
always_comb begin
    s  = a ^ b ^ c;
    co = ab | ac | bc;
end
```

Idea: kod w bloku `always_comb` opisuje, jak uaktualnić wyjścia *zawsze* gdy zmienią się wartości powiązanych wejść.

Pułapka: nie każdy kod reprezentuje układ kombinacyjny!

WIĘKSZA PUŁAPKA: nieprawidłowy kod **może** nie być odrzucony w syntezie, lecz wygenerować układ o **niespodziewanym zachowaniu**!

Uwaga! (najważniejszy slajd tego wykładu)

- Pisanie kodu proceduralnego wygląda jak programowanie, ale nim **NIE JEST**.
- Kod opisuje sposób konstrukcji **schematu** w syntezie. Pisząc kod, należy myśleć o wynikowym schemacie.
- Należy pamiętać o zasadzie **modularności**. Schemat należy podzielić na **bloki funkcjonalne**, i implementować je niezależnie.
- Należy unikać pokusy opisywania całego układu jednym blokiem kodu!
- Jeśli zamiast konstrukcji proceduralnych łatwo zastosować np. blok `assign`, należy tak zrobić!


© 2014 Middle Earth Enterprises © 2014 FFG
WWW.JCBARQUET.COM

Instrukcja przypisania

Nadaje wartość sygnałowi:

```
o = e;
```

Instrukcja przypisania

Nadaje wartość sygnałowi:

```
o = e;
```

Pułapka: to jest **instrukcja** przypisania, a nie **wyrażenie**, jak w C!

Instrukcja przypisania

Nadaje wartość sygnałowi:

```
o = e;
```

Pułapka: to jest **instrukcja** przypisania, a nie **wyrażenie**, jak w C!

Pułapka: zależność cykliczna **nie** specyfikuje układu kombinacyjnego!

Przykłady – instrukcja przypisania

Dobrze:

```
always_comb o = a & b;
```

Źle:

```
always_comb o = o & b;
```

Źle:

```
always_comb o = p = a & b;
```

Instrukcja złożona

Podinstrukcje są wykonywane **w kolejności**!

```
begin
    i1;
    i2;
    i3;
end
```

Instrukcja złożona

Podinstrukcje są wykonywane **w kolejności**!

```
begin
    i1;
    i2;
    i3;
end
```

Pułapka: przy wielokrotnym przypisaniu wartości sygnału, wygrywa **ostatnie**.

Instrukcja złożona

Podinstrukcje są wykonywane **w kolejności**!

```
begin
    i1;
    i2;
    i3;
end
```

Pułapka: przy wielokrotnym przypisaniu wartości sygnału, wygrywa **ostatnie**.

Pułapka: w przypadku zależności, kolejność przypisań musi być **zgodna** z zależnościami.

Przykłady – instrukcja złożona

Dobrze:

```
always_comb begin
    o = a | b;
    p = o | c;
end
```

Dobrze, ale unikać:

```
always_comb begin
    o = a | b;
    p = o;
    p = p | c;
end
```

Przykłady – instrukcja złożona

Dobrze:

```
always_comb begin
    o = a | b;
    p = o | c;
end
```

Źle:

```
always_comb begin
    p = o | c;
    o = a | b;
end
```

Instrukcja warunkowa

W zależności od wartości wyrażenia, zredukowanej do jednego bitu, wybierana jest jedna z dwóch podinstrukcji:

```
if (b) i1;
else i2;
```

Instrukcja warunkowa specyfikuje multiplekser.

Instrukcja warunkowa

W zależności od wartości wyrażenia, zredukowanej do jednego bitu, wybierana jest jedna z dwóch podinstrukcji:

```
if (b) i1;
else i2;
```

Instrukcja warunkowa specyfikuje multiplekser.

Pułapka: każdy sygnał przypisany w jednej z gałęzi **musi** być też przypisany w drugiej!

Przykłady – instrukcja warunkowa

Dobrze:

```
always_comb
    if (b) o = a;
    else o = c;
```

Źle:

```
always_comb
    if (b) o = a;
    else p = c;
```

Zasada: jedno wyjście, jeden blok

Każdy sygnał musi być przypisany w **co najwyżej jednym** bloku.

Dobrze:

```
always_comb
    if (b) o = a;
    else o = c;
```

Źle:

```
always_comb
    if (b) o = a;
always_comb
    if (!b) o = c;
```

Instrukcja wyboru

Wykonuje instrukcję, której etykieta jest równa testowanej wartości. Przypadki są sprawdzane **w kolejności**.

```
case (v)
    e1: i1;
    e2: i2;
    default: id;
endcase
```

Instrukcja wyboru specyfikuje multiplekser (być może o dużej liczbie wejść).

Instrukcja wyboru

Wykonuje instrukcję, której etykieta jest równa testowanej wartości. Przypadki są sprawdzane **w kolejności**.

```
case (v)
    e1: i1;
    e2: i2;
    default: id;
endcase
```

Instrukcja wyboru specyfikuje multiplekser (być może o dużej liczbie wejść).

Pułapka: Nie używa się `break` jak w C!

Instrukcja wyboru

Wykonuje instrukcję, której etykieta jest równa testowanej wartości. Przypadki są sprawdzane **w kolejności**.

```
case (v)
    e1: i1;
    e2: i2;
    default: id;
endcase
```

Instrukcja wyboru specyfikuje multiplekser (być może o dużej liczbie wejść).

Pułapka: Nie używa się `break` jak w C!

Pułapka: Wartości `x` (don't know) nie należy używać w etykietach!

Przykłady – instrukcja wyboru

Dobrze:

```
always_comb
    case ({a, b})
        2'b01: o = 1'b1;
        2'b10: o = 1'b1;
        default: o = 1'b0;
    endcase
```

Też dobrze:

```
always_comb
    case ({a, b})
        2'b01, 2'b10: o = 1'b1;
        default: o = 1'b0;
    endcase
```

Przykłady – instrukcja wyboru

Dobrze:

```
always_comb
    case (a)
        1'b1: o = 1'b0;
        1'b0: o = 1'b1;
    endcase
```

Źle:

```
always_comb
    case (a)
        1'b1, 1'b0: o = 1'b1;
        1'bx: o = 1'b0;
    endcase
```

Przykłady – instrukcja wyboru

```
always_comb begin
    {o, p} = 2'b00;
    case ({a, b})
        2'b10: {o, p} = 2'b11;
        2'b01: {o, p} = 2'b01;
    endcase
end
```

Źle:

```
always_comb
    case ({a, b})
        2'b10: {o, p} = 2'b11;
        2'b01: {o, p} = 2'b01;
    endcase
```

Instrukcja wyboru – na odwrót

Dobrze, ale używać z rozwagą.

```
always_comb
    case (1'b1)
        a: {o, p} = 2'b01;
        b: {o, p} = 2'b10;
        default: {o, p} = 2'b00;
    endcase
```

Instrukcja wyboru z wildcardami

Wykonuje instrukcję, której etykieta **dopasowuje się do** testowanej wartości. Znak ? dopasowuje się do dowolnego bitu. Przypadki są sprawdzane **w kolejności**.

```
casez (v)
    e1: i1;
    e2: i2;
    default: id;
endcase
```

Instrukcja wyboru z wildcardami

Wykonuje instrukcję, której etykieta **dopasowuje się do** testowanej wartości. Znak ? dopasowuje się do dowolnego bitu. Przypadki są sprawdzane **w kolejności**.

```
casez (v)
    e1: i1;
    e2: i2;
    default: id;
endcase
```

Pułapka: Znak ? działa zarówno w etykietach, jak i w testowanej wartości!

Przykłady – instrukcja wyboru z wildcardami

```
always_comb
    casez ({a, b})
        2'b1?: {o, p} = 2'b10;
        2'b01: {o, p} = 2'b01;
        default: {o, p} = 2'bxx;
    endcase
```

Źle:

```
always_comb
    case ({a, b})
        2'b1?: {o, p} = 2'b10;
        2'b01: {o, p} = 2'b01;
        default: {o, p} = 2'bxx;
    endcase
```

Instrukcja wyboru z wildcardami – na odwrót

Źle – narzędzia do syntezy mogą nie interpetowac właściwie:

```
always_comb
    casez (2'b1?)
        a: {o, p} = 2'b01;
        b: {o, p} = 2'b10;
        default: {o, p} = 2'b00;
    endcase
```

Idiomy

Idiomy

- W języku opisu sprzętu (typowo) nie programuje się, tylko **opisuje sprzęt**.
- Przed rozpoczęciem pisania należy mieć (w głowie lub na papierze) **schemat** implementowanych bloków i ich wzajemnych połączeń
- Abstrakcje sprzętowe implementowane za pomocą **idiomów**.

Idiomy

- W języku opisu sprzętu (typowo) nie programuje się, tylko **opisuje sprzęt**.
- Przed rozpoczęciem pisania należy mieć (w głowie lub na papierze) **schemat** implementowanych bloków i ich wzajemnych połączeń
- Abstrakcje sprzętowe implementowane za pomocą **idiomów**.

Idiom: wyrażenie językowe, którego znaczenie jest swoiste, odmienne od znaczenia, jakie należałoby mu przypisać, biorąc pod uwagę poszczególne części składowe oraz reguły składni. (Wikisłownik)

Idiomy

- W języku opisu sprzętu (typowo) nie programuje się, tylko **opisuje sprzęt**.
- Przed rozpoczęciem pisania należy mieć (w głowie lub na papierze) **schemat** implementowanych bloków i ich wzajemnych połączeń
- Abstrakcje sprzętowe implementowane za pomocą **idiomów**.

Idiom: wyrażenie językowe, którego znaczenie jest swoiste, odmienne od znaczenia, jakie należałoby mu przypisać, biorąc pod uwagę poszczególne części składowe oraz reguły składni. (Wikisłownik)

Uwaga: w razie wątpliwości, osobne elementy na schemacie należy zaimplementować **osobnymi** blokami, połączonymi za pomocą nazwanych przewodów. (Patrz: najważniejszy slajd wykładu.)

Sumator

```
module binary_adder(
    output [3:0] s,
    output co,
    input [3:0] a, b,
    input ci
);
    assign {co, s} = a + b + ci;
endmodule
```

Dobór implementacji sumatora jest zostawiany narzędziu do syntezy.

Komparator

```
module mag_compare(
    output lt, eq, gt,
    input [3:0] a, b
);
    assign lt = a < b;
    assign gt = a > b;
    assign eq = a == b;
endmodule
```

Multiplekser 2-wejściowy

```
module mux_2(
    output o,
    input a, b,
    input s
);
    assign o = s ? a : b;
endmodule
```

Multiplekser 2-wejściowy (behawioralny)

```
module mux_2(
    output o,
    input a, b,
    input s
);
    always_comb
        if (s) o = a;
        else o = b;
endmodule
```

Multiplekser 4-wejściowy

```
module mux_4(
    output o,
    input a, b, c, d,
    input [1:0] s
);
    always_comb
        case(s)
            2'd0: o = a;
            2'd1: o = b;
            2'd2: o = c;
            2'd3: o = d;
        endcase
endmodule
```

Dekoder 1 do 2

```
module decoder_1_to_2(
    output [1:0] o,
    input i
);
    assign o = i ? 2'b10 : 2'b01;
endmodule
```

Dekoder 2 do 4

```
module decoder_2_to_4(
    output [3:0] o,
    input [1:0] i
);
    always_comb
        case(i)
            2'd0: o = 4'b0001;
            2'd1: o = 4'b0010;
            2'd2: o = 4'b0100;
            2'd3: o = 4'b1000;
        endcase
endmodule
```

Dekoder 2 do 4 – inaczej

```
module decoder_2_to_4(
    output [3:0] o,
    input [1:0] i
);
    assign o = 4'b1 << i;
endmodule
```

Enkoder priorytetowy 2 do 1

```
module prio_encoder_2_to_1(
    output o,
    input [1:0] i
);
    always_comb
        casez(i)
            2'b01: o = 1'b0;
            2'b1?: o = 1'b1;
            default: o = 1'bx;
        endcase
endmodule
```

Enkoder priorytetowy 4 do 2

```
module prio_encoder_4_to_2(
    output [1:0] o,
    input [3:0] i
);
    always_comb
        casez(i)
            4'b0001: o = 2'd0;
            4'b001?: o = 2'd1;
            4'b01??: o = 2'd2;
            4'b1???: o = 2'd3;
            default: o = 2'bx;
        endcase
endmodule
```

Enkoder priorytetowy 4 do 2

```
module prio_encoder_4_to_2(
    output [1:0] o,
    input [3:0] i
);
    always_comb
        case(1'b1)
            i[3]: o = 2'd3;
            i[2]: o = 2'd2;
            i[1]: o = 2'd1;
            i[0]: o = 2'd0;
            default: o = 2'bx;
        endcase
endmodule
```

Enkoder 2 do 1 – nieefektywny

Nieefektywna implementacja – nie korzysta z 1-hot:

```
module encoder_2_to_1(
    output o,
    input [1:0] i
);
    always_comb
        case(i)
            2'b01: o = 1'b0;
            2'b10: o = 1'b1;
            default: o = 1'bx;
        endcase
endmodule
```

Enkoder 2 do 1

```
module encoder_2_to_1(
    output o,
    input [1:0] i
);
    always_comb
        unique casez(i)
            2'b?1: o = 1'b0;
            2'b1?: o = 1'b1;
            default: o = 1'bx;
        endcase
endmodule
```

Enkoder 4 do 2

```
module encoder_4_to_2(
    output [1:0] o,
    input [3:0] i
);
    always_comb
        unique casez(i)
            4'b???1: o = 2'd0;
            4'b??1?: o = 2'd1;
            4'b?1??: o = 2'd2;
            4'b1???: o = 2'd3;
            default: o = 2'bx;
        endcase
endmodule
```

Enkoder 4 do 2 – inna wersja

```
module encoder_4_to_2(
    output [1:0] o,
    input [3:0] i
);
    always_comb
        unique case(1'b1)
            i[3]: o = 2'd3;
            i[2]: o = 2'd2;
            i[1]: o = 2'd1;
            i[0]: o = 2'd0;
            default: o = 2'bx;
        endcase
endmodule
```

Multiplekser one-hot 2-wejściowy

```
module mux_2_1hot(
    output o,
    input a, b,
    input [1:0] s
);
    always_comb
        unique casez(s)
            2'b?1: o = a;
            2'b1?: o = b;
            default: o = 1'bx;
        endcase
endmodule
```

Multiplekser one-hot 4-wejściowy

```
module mux_2_1hot(
    output o,
    input a, b, c, d,
    input [3:0] s
);
    always_comb
        unique casez(s)
            4'b???1: o = a;
            4'b??1?: o = b;
            4'b?1??: o = c;
            4'b1???: o = d;
            default: o = 1'bx;
        endcase
endmodule
```

Demultiplekser 2-wyjściowy

```
module demux_2(
    output a, b,
    input i,
    input s
);
    always_comb begin
        a = 0; b = 0;
        if (s) b = i;
        else a = i;
    end
endmodule
```

Dekoder wyświetlacza 7-segm.

```
module seg7(
    input [3:0] hex,
    output [1:7] leds
);
    always_comb
        case(hex)           //abcdefg
            4'd0:  leds = 7'b1111110;
            4'd1:  leds = 7'b0110000;
            4'd2:  leds = 7'b1101101;
            // ...
        endcase
endmodule
```

ALU 74381

```
module alu_74381(
    output [3:0] o,
    input [2:0] s,
    input [3:0] a, b
);
    always_comb
        case (s)
            3'd0: o = 4'b0000;
            3'd1: o = b - a;
            3'd2: o = a - b;
            3'd3: o = a + b;
            3'd4: o = a ^ b;
            3'd5: o = a | b;
            3'd6: o = a & b;
            3'd7: o = 4'b1111;
        endcase
endmodule
```

Pętle for

Instrukcja pętli

Podobna składniowo do C, ale bardziej ograniczona:

```
for(x = e1; e2; x = e3) i;
```

- `x` jest zmienną typu `integer`
- Wyrażenia e1, e2, e3 nie używają sygnałów (w tym wejść, wyjść, `logic`)
- Ale mogą używać parametrów, zmiennych `integer` i stałych

Instrukcja pętli

Podobna składniowo do C, ale bardziej ograniczona:

```
for(x = e1; e2; x = e3) i;
```

- `x` jest zmienną typu `integer`
- Wyrażenia e1, e2, e3 nie używają sygnałów (w tym wejść, wyjść, `logic`)
- Ale mogą używać parametrów, zmiennych `integer` i stałych

Pułapka: Iteracja w **przestrzeni**, a nie w czasie!

Pętla jest konstrukcją **zaawansowaną**. W razie wątpliwości – unikać.

Instrukcja pętli – przykład

Enkoder priorytetowy 4 do 2 używający instrukcji pętli:

```
module prio_encoder_4_to_2(
    output [1:0] o,
    input [3:0] i
);
    integer k;
    always_comb begin
        o = 2'bx;
        for (k = 0; k < 4; k = k + 1)
            if (i[k]) o = k;
    end
endmodule
```

Instrukcja pętli – przykład

Dekoder 2 do 4 używający instrukcji pętli:

```
module decoder_2_to_4(
    output [3:0] o,
    input [1:0] i
);
    integer k;
    always_comb begin
        o = 4'b0;
        for (k = 0; k < 4; k = k + 1)
            if (i == k) o[k] = 1;
    end
endmodule
```