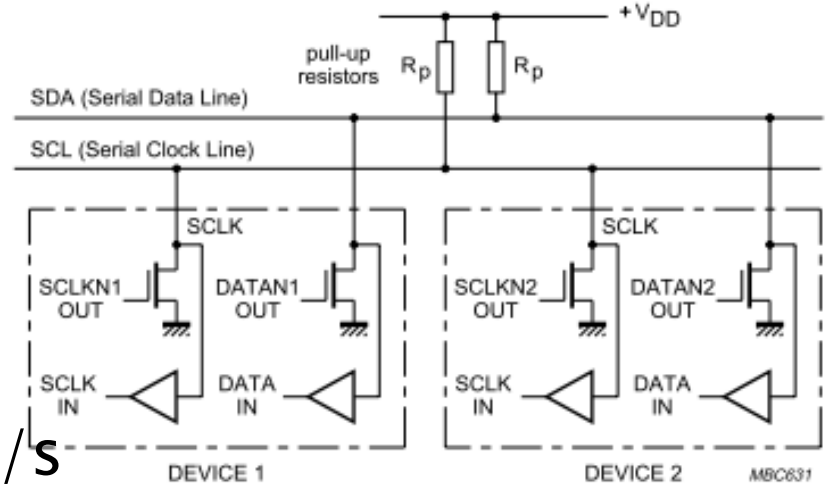
Verilog I2C

Commenti sul codice verilog per generare il protocollo I2C

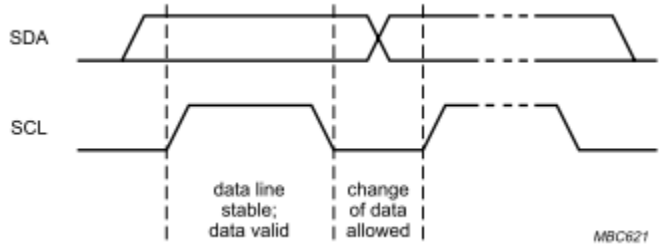
Protocollo I2C

- Protocollo seriale sviluppato da Philips
- Usa due linee (SDA, SCL) con 2 stati logici
  - Stato basso (forte)
  - Stato alto (debole) (Alta impedenza + resistenza di pullup)
- Può comunicare con diversi dispositivi (multimaster)
- Nella versione Standard funziona fino 100kb/s
- Esistono anche versioni Fast 400kb/s ed HS 3,4 Mb/s

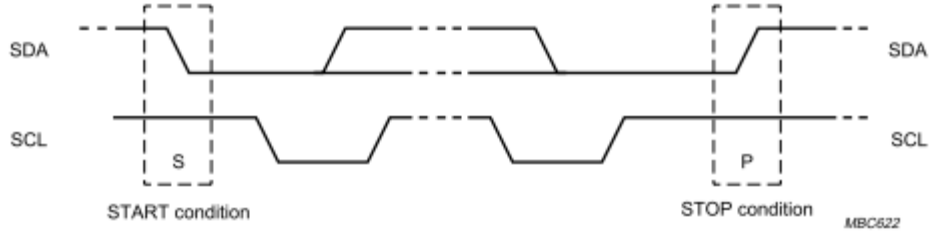


Protocollo I2C

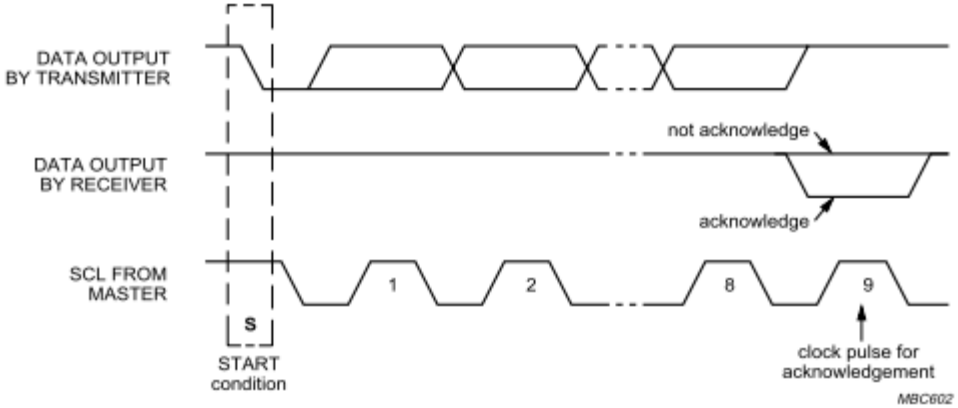
- Bit transmission



- Start-Stop transmission



- Acknowledge

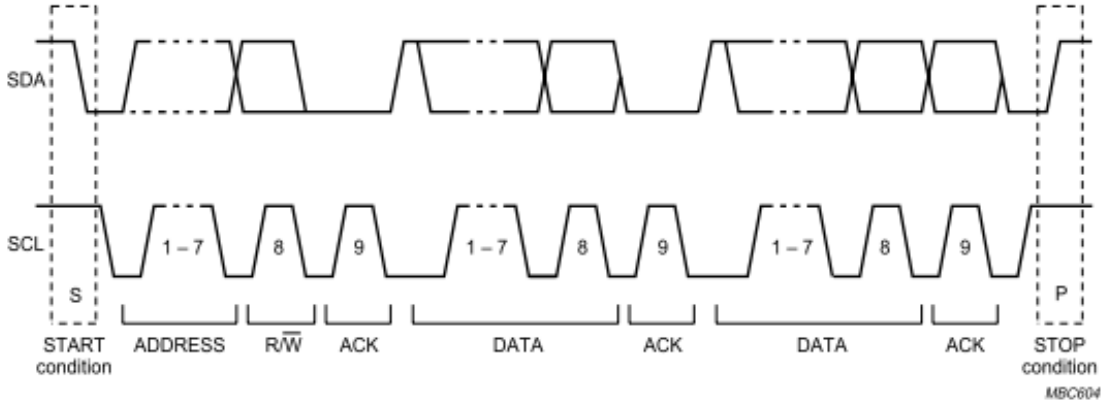


Protocollo I2C

Trasferimento con pacchetti di 8bits (+1 ack)

Tipicamente l'ultimo bit dell'indirizzo del dispositivo è il bit R/W

- Un dato alla volta
  - Start
  - Indirizzo dispositivo
  - Indirizzo del registro
  - Dato
  - Stop
- Più dati alla volta
  - Start
  - Indirizzo dispositivo
  - Indirizzo di Reg0
  - Dato Reg0
  - Dato Reg1
  - …..
  - Stop

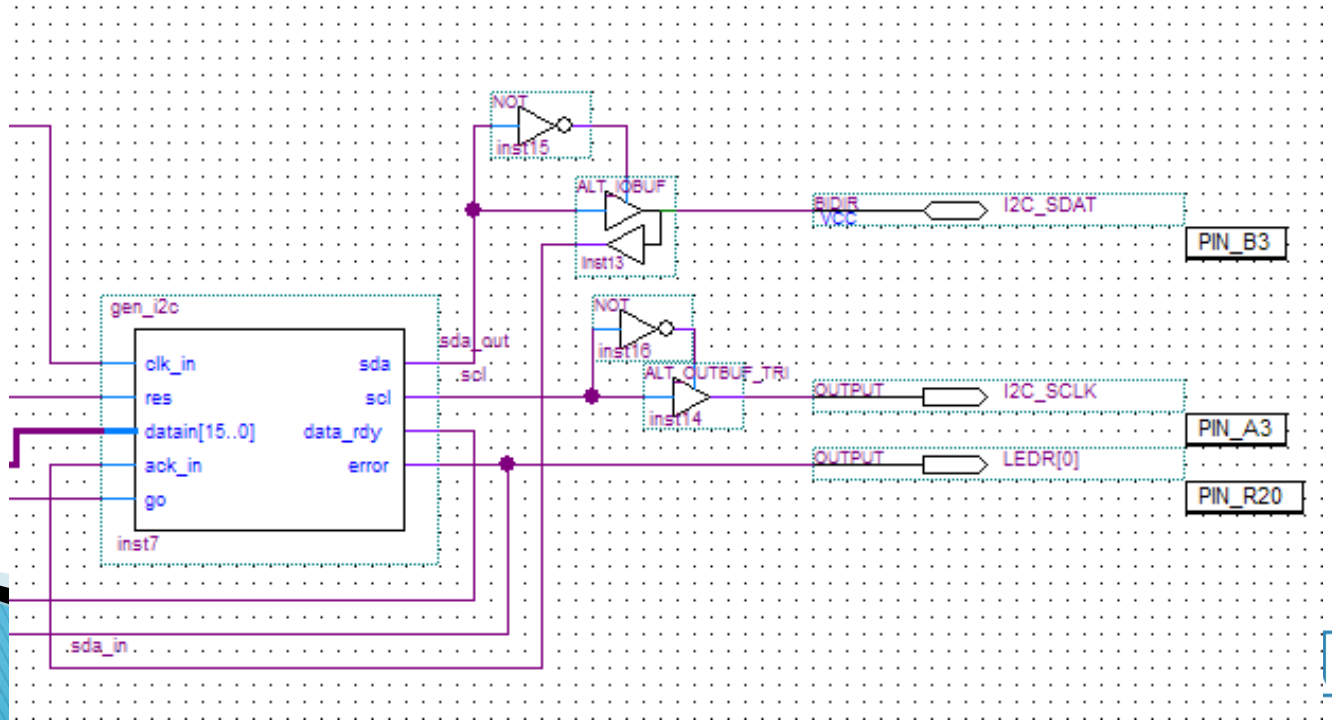


Disclaimer

- Il codice proposto
  - È sicuramente perfettibile
  - Vuole essere solo un esempio su come si può scrivere un sistema sintetizzabile in Verilog
  - Ha funzionalità limitate (1 master ed 1 slave)
  - Presenta alcuni malfunzionamenti che sono oggetto di analisi e di perfezionamento.

Bit Generation

- Realizzazione di un blocco per generare i tre byte (device, indirizzo,dato)
- Ingressi: clk, reset, dato [15..0], ack_in, go)
- Uscite: sda, scl, data_rdy, error



Modulo, porte e segnali interni

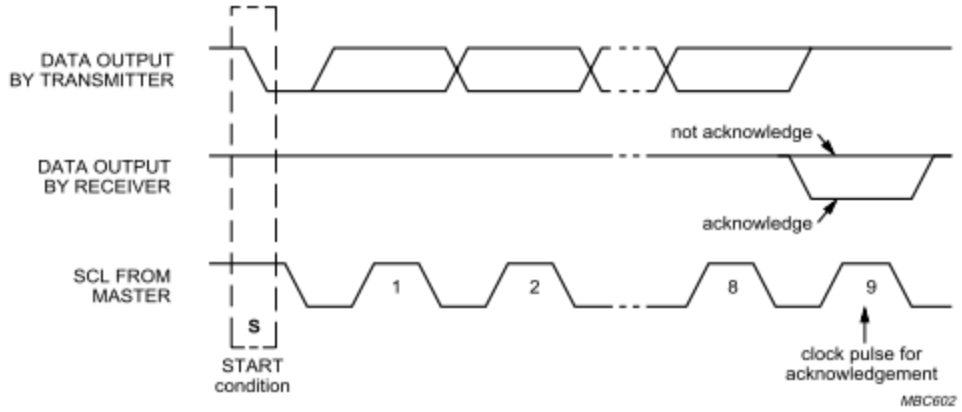
```
module gen_i2c (clk_in,res,datain,ack_in,go,sda,scl,data_rdy,error);

input   clk_in;
input   res;
input   [15:0] datain;
input   go;
input   ack_in;
output  reg sda=1;
output  reg scl=1;
output  reg data_rdy;
output  reg error;

reg [8:0] counter;
reg [3:0] cmd_sda;
reg [3:0] cmd_scl;
reg ACK1,ACK2,ACK3;

wire [1:0] counter_4=counter[1:0];
wire [6:0] counter_bit=counter[8:2];

reg Trans_ON;
wire [23:0] bits={8'h34,datain};
```

Segnali I/O

- Segnali “classici” `clk, reset`
- Segnali in I/O desiderati `dato[15..0],sda,sck`.
- Segnali di controllo:
  - `go`: il dati in ingresso è stabile e si può trasmettere
  - `data_rdy` : sono pronto a ricevere nuovi dati, se `data_rdy` non è alto non si modifichi il dato in ingresso
  - `ack`: viene controllato nel preciso istante in cui il dispositivo accusa il ricevuto
  - `error`: viene segnalato un eventuale errore

Generazione dei bits

- Ogni bit sfrutti 4 cicli consecutivi di clock
  - Idle: `sda =1111, scl=1111`
  - Start: `sda =1000, scl=1110`
  - Dato: `sda =xxxx, scl=0110`
  - ack: `sda =1111, scl=0110`
  - Stop `sda =0001, scl=0111`



Contatori per i bits

- Vengono introdotti 2 contatori
  - Contatore delle 4 fasi (counter_4) del bit
  - Contatore dei bit

Trasmissione attiva

```
// Trans_ON segnale per indicare il periodo di trasmissione dati attiva
always @(posedge clk_in or posedge res) begin
if (res)
    begin; Trans_ON=0; data_rdy=1; end
else if (go)
    begin; Trans_ON=1; data_rdy=0; end
else if (counter_bit ==7'd33)
    begin; Trans_ON=0; data_rdy=1; end
end
```

Generazione dati

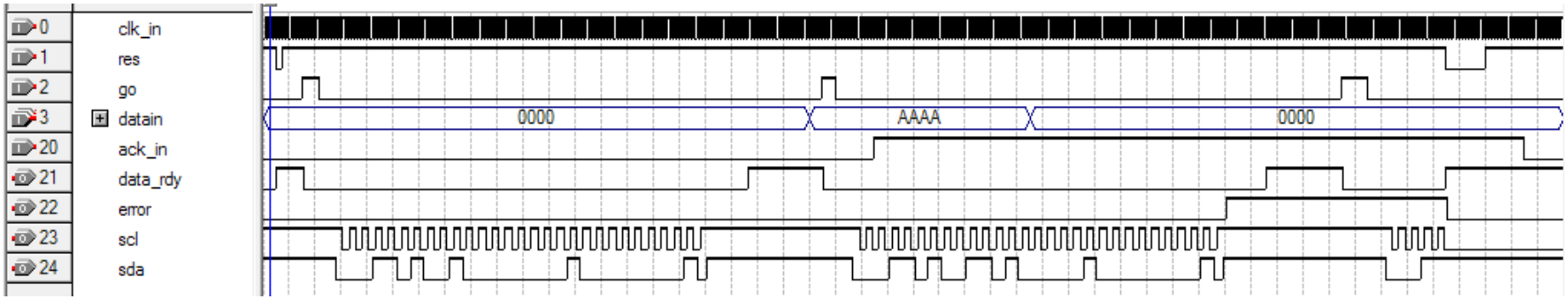
```
// generazione dei segnali
always @(posedge clk_in or posedge res or posedge go) begin
if (res|go) counter=0;
else if (Trans_ON)
begin
  counter=counter+1;
  case (counter_bit)
    6'd0  : begin cmd_sda=4'b1111; cmd_scl=4'b1111; end
    // start
    6'd1  : begin cmd_sda=4'b0001; cmd_scl=4'b0111; end
    // device
    6'd2  : begin cmd_sda={bits[23],bits[23],bits[23],bits[23]}; cmd_scl=4'b0110; end
    6'd3  : begin cmd_sda={bits[22],bits[22],bits[22],bits[22]}; cmd_scl=4'b0110; end
    6'd4  : begin cmd_sda={bits[21],bits[21],bits[21],bits[21]}; cmd_scl=4'b0110; end
    6'd5  : begin cmd_sda={bits[20],bits[20],bits[20],bits[20]}; cmd_scl=4'b0110; end
    6'd6  : begin cmd_sda={bits[19],bits[19],bits[19],bits[19]}; cmd_scl=4'b0110; end
    6'd7  : begin cmd_sda={bits[18],bits[18],bits[18],bits[18]}; cmd_scl=4'b0110; end
    6'd8  : begin cmd_sda={bits[17],bits[17],bits[17],bits[17]}; cmd_scl=4'b0110; end
    6'd9  : begin cmd_sda={bits[16],bits[16],bits[16],bits[16]}; cmd_scl=4'b0110; end
    // ACK1
    6'd10  : begin cmd_sda=4'b1111; cmd_scl=4'b0110; end
    // address
    6'd11  : begin cmd_sda={bits[15],bits[15],bits[15],bits[15]}; cmd_scl=4'b0110; end
    6'd12  : begin cmd_sda={bits[14],bits[14],bits[14],bits[14]}; cmd_scl=4'b0110; end
    6'd13  : begin cmd_sda={bits[13],bits[13],bits[13],bits[13]}; cmd_scl=4'b0110; end
    ...
    6'd25  : begin cmd_sda={bits[2],bits[2],bits[2],bits[2]}; cmd_scl=4'b0110; end
    6'd26  : begin cmd_sda={bits[1],bits[1],bits[1],bits[1]}; cmd_scl=4'b0110; end
    6'd27  : begin cmd_sda={bits[0],bits[0],bits[0],bits[0]}; cmd_scl=4'b0110; end
    // ACK3
    6'd28  : begin cmd_sda=4'b1111; cmd_scl=4'b0110; end
    // stop
    6'd29  : begin cmd_sda=4'b1000; cmd_scl=4'b1110; end
    // end
    6'd30  : begin cmd_sda=4'b1111; cmd_scl=4'b1111; end
  endcase
  sda=cmd_sda[counter_4];
  scl=cmd_scl[counter_4];
end
end
```

ACK

```
// verifica ACK
always @(posedge clk_in)
if (res) begin ACK1=1;ACK2=1;ACK3=1;error=0;end
else
begin
if (counter_bit==6'd10 & counter_4==3) ACK1=ack_in;
if (counter_bit==6'd19 & counter_4==3) ACK2=ack_in;
if (counter_bit==6'd28 & counter_4==3) ACK3=ack_in;
if (counter_bit > 6'd29) error=ACK1&ACK2&ACK3;
end
```

Simulazione

- La trasmissione parte dopo il “go”
- Il dato in ingresso non deve cambiare se si sta trasmettendo
- I primi 3 bit sono l’indirizzo del dispositivo (h34)
- Se non c’è ACK (simulato) viene segnalato errore



Segnali di controllo

- Idea: quando si riceve il `data_rdy` si legge da una ROM il `dato`, lo si fornisce esternamente e si genera il segnale `go`

```
module gen_code (
    clk,
    res,
    rdy_in,
    data,
    go
);
    input  clk;
    input  res;
    input  rdy_in;
    output reg [15:0] data;
    output reg go;

    reg [3:0] addr;
    reg [15:0]ROM[`rom_size:0];
```

Segnali di controllo (problemi)

```
always @(posedge clk) begin
    if ((rdy_in==1) & (addr <= `rom_size)) go=1;
    else go=0;
end

always @(posedge go or posedge res) begin
    if (res) addr = 0;
    else if (addr <= `rom_size) addr=addr+1; end

always @(posedge go) begin

    ROM[0]= 16'h0000;        // linvol
    ROM[1]= 16'h021A;        // rinvol
    ROM[2]= 16'h047B;        // lhpvol
    ROM[3]= 16'h067B;        // rhpvol
    ROM[4]= 16'h08F8;        // audiopath
    ROM[5]= 16'h0a06;        // digitalpath
    ROM[6]= 16'h0c00;        // power down disable
    ROM[7]= 16'h0e01;        // format and master
    ROM[8]= 16'h1002;        // control
    ROM[9]= 16'h1201;        // active
    //ROM[8]= 16'h1e00;                  // reset

    data=ROM[addr];

end
endmodule
```

Azioni sconsigliate

- Generare nuovi segnali si clock (anche impliciti)
  - Es:

```
always @ (posedge go)
wire clk_div4= !(counter_4[1] || counter_4[0]) ;
```

- Ridefinire sotto altre condizioni ed in altre procedure segnali che siano già assegnati.
  - Es: modificare il segnale “go” attraverso procedure diverse

Altra versione

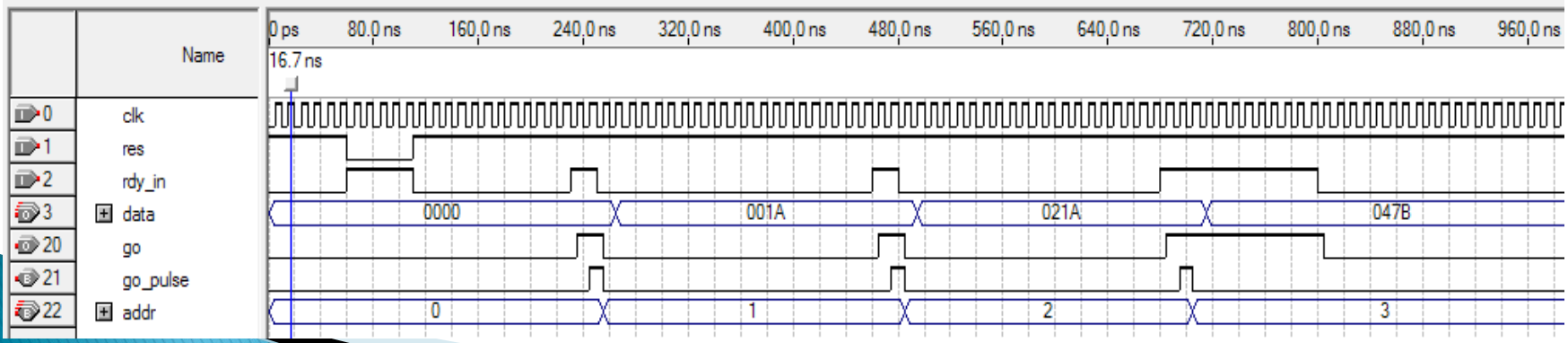
Generazione di un'impulso unitario

```
module gen_code (clk,res,rdy_in,data,go );
    input  clk,res,rdy_in;
    output reg [15:0] data;
    output reg go;

    reg [3:0] addr;
        reg [15:0]ROM[`rom_size:0];
        reg go1,go_pulse;

// generazione segnale "go"
always @(posedge clk or negedge res) begin
    if (!res)  go=0;
    else if ((rdy_in) & (addr < `rom_size)) go=1;
    else go=0;
end

// generazione di un impulso unitario in corrispondenza al fronte di salita
always @(posedge clk)  go1=go;
always @(posedge clk) begin if (!go1 & go) go_pulse=1; else go_pulse=0; end

// incremento dell'indirizzo
always @(posedge clk or negedge res) begin
    if (!res) addr = 0;
    else if (addr < `rom_size & go_pulse) addr=addr+1; end
```

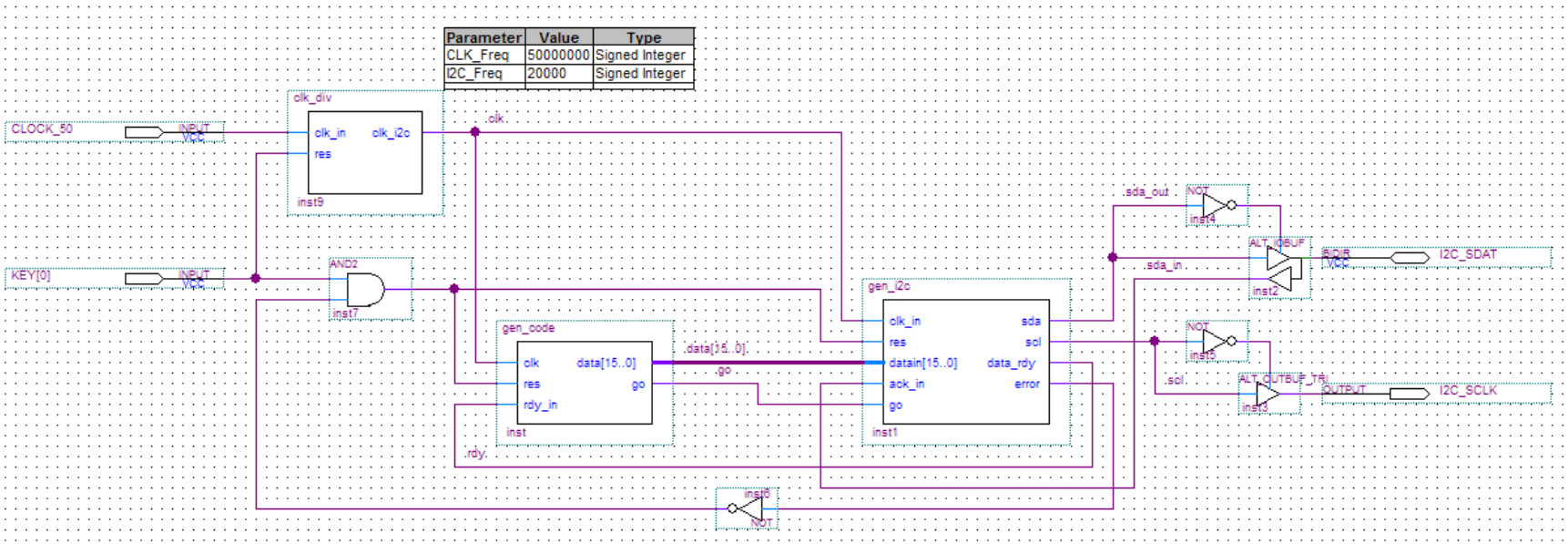
Simulazione

- Sincronismo sul fronte: clock skew
- Sincronismo sullo stato: corsa del contatore

- Il dato all'indirizzo 0 non viene mai letto! (dummy)



Schematico


Parameter Value Type
CLK_Freq 50000000 Signed Integer
I2C_Freq 20000 Signed Integer
clk_div
clk_in
clk_i2c
res
inst9
CLOCK_50
KEY[0]
AND2
inst7
.clk
gen_code
clk
data[15..0]
res
go
rdy_in
inst
.data[15..0]
.go
.rdy
gen_i2c
clk_in
sda
res
scl
datain[15..0]
data_rdy
ack_in
error
go
inst1
inst10
NOT
.sda_out
inst14
.sda_in
ALT_IOBUF
inst2
BIDIR
I2C_SDAT
inst15
.scl
ALT_OUTBUF_TRI
inst3
OUTPUT
I2C_SCLK