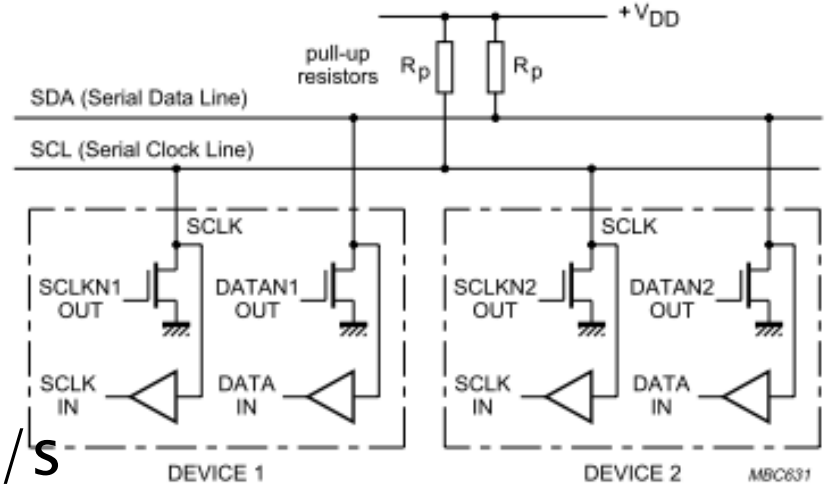
Verilog I2C

Commenti sul codice verilog per generare il protocollo I2C

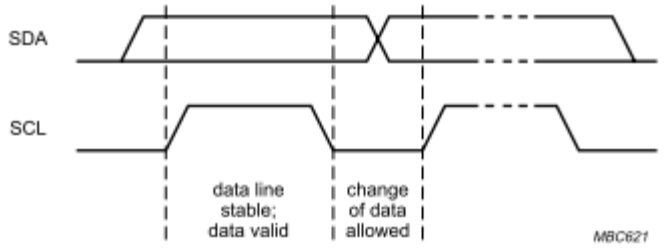
Protocollo I2C

- 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

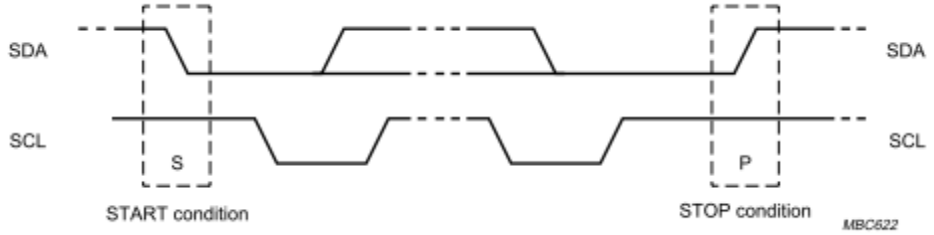


Protocollo I2C

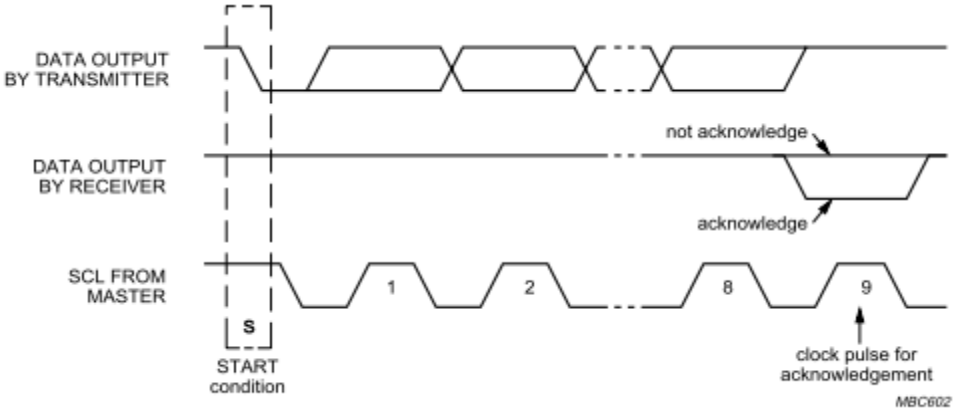
- Bit transmission



- Start-Stop transmission



- Acknowledge

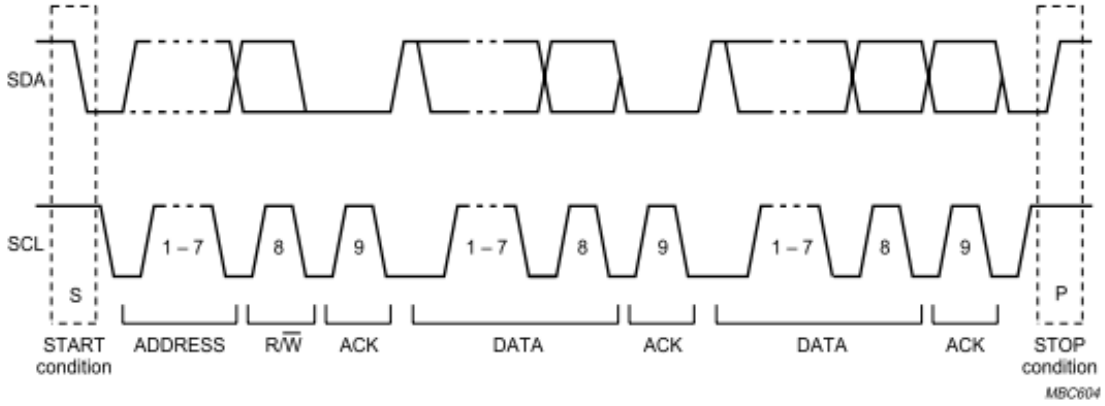


Protocollo I2C

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

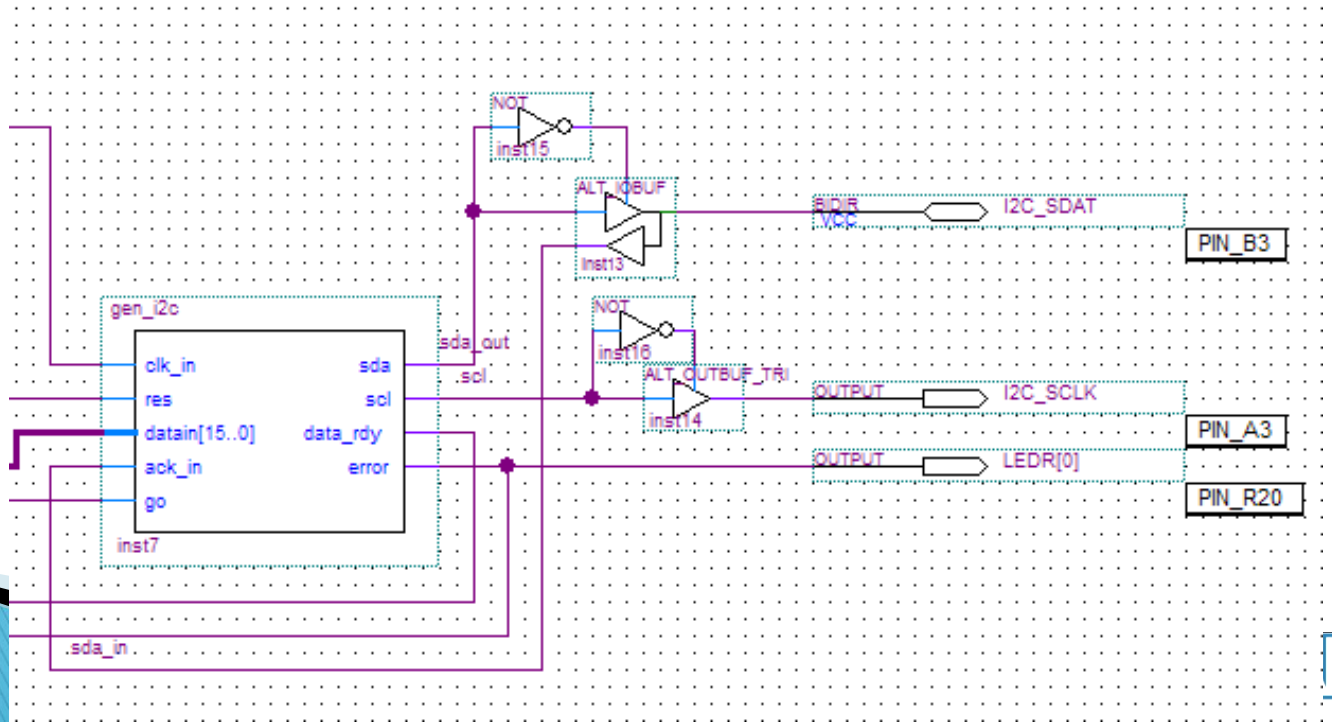


Disclaimer

- 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)

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

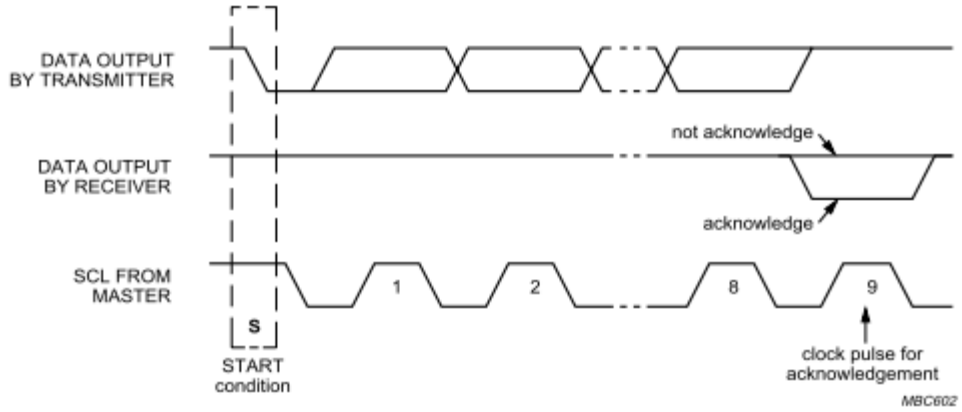
```
module gen_i2c (clk_in,res,datain,ack_in,valid,sda,scl,data_rdy,error);

input    clk_in;
input   res;
input   [15:0] datain;
input   valid;
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];

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

Segnali I/O

- Segnali “classici” `clk, reset`
- Segnali in I/O desiderati `datain[15..0],sda,sck`.
- Segnali di controllo:
  - `valid`: il dato 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_in:` 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 negedge res) begin
if (!res)
    data_rdy=1;
else if (valid)
    data_rdy=0;
else if (counter_bit ==7'd33)
    data_rdy=1;
end
```

La trasmissione è attiva quando data_rdy è allo stato logico basso

Generazione dati parte 1

```
// generazione dei segnali
always @(posedge clk_in or negedge res or posedge valid) begin
if (!res|valid)
begin
counter=0;sda=1;scl=1;
end
else if (!data_rdy)
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={4{bits[23]}}; cmd_scl=4'b0110; end
   6'd3  : begin cmd_sda={4{bits[22]}}; cmd_scl=4'b0110; end
   6'd4  : begin cmd_sda={4{bits[21]}}; cmd_scl=4'b0110; end
   6'd5  : begin cmd_sda={4{bits[20]}}; cmd_scl=4'b0110; end
   6'd6  : begin cmd_sda={4{bits[19]}}; cmd_scl=4'b0110; end
   6'd7  : begin cmd_sda={4{bits[18]}}; cmd_scl=4'b0110; end
   6'd8  : begin cmd_sda={4{bits[17]}}; cmd_scl=4'b0110; end
   6'd9  : begin cmd_sda={4{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={4{bits[15]}}; cmd_scl=4'b0110; end
   6'd12  : begin cmd_sda={4{bits[14]}}; cmd_scl=4'b0110; end
   6'd13  : begin cmd_sda={4{bits[13]}}; cmd_scl=4'b0110; end
   6'd14  : begin cmd_sda={4{bits[12]}}; cmd_scl=4'b0110; end
```

Generazione dati parte 2

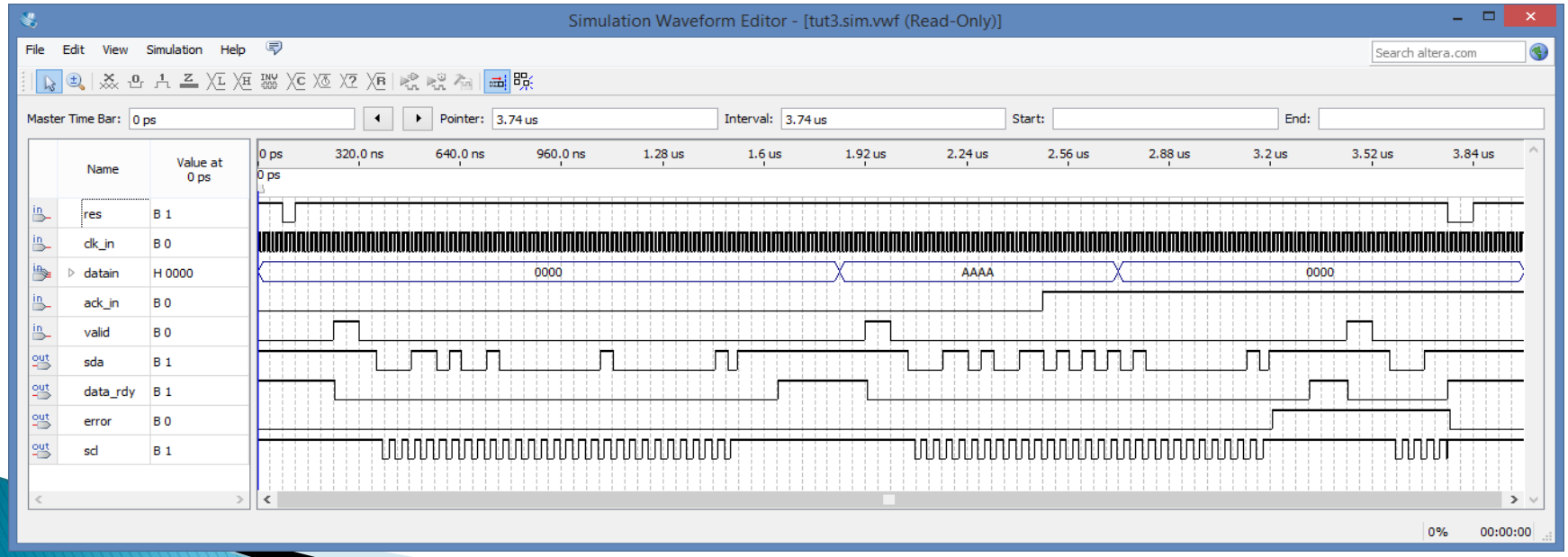
```
    6'd24  : begin cmd_sda={4{bits[3]}}; cmd_scl=4'b0110; end
    6'd25  : begin cmd_sda={4{bits[2]}}; cmd_scl=4'b0110; end
    6'd26  : begin cmd_sda={4{bits[1]}}; cmd_scl=4'b0110; end
    6'd27  : begin cmd_sda={4{bits[0]}}; cmd_scl=4'b0110; end
    // ACK3
    6'd28  : begin cmd_sda=4'b1111; cmd_scl=4'b0110; 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
endmodule
```

Simulazione

- La trasmissione parte solo quando “valid” si abbassa
- Il dato in ingresso non deve cambiare se si sta trasmettendo
- I primi 8 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 `valid`

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

    reg [2:0] fasi;
    reg [3:0] addr;
    reg [15:0]ROM[`rom_size:0];
```

Segnali di controllo (problemi)

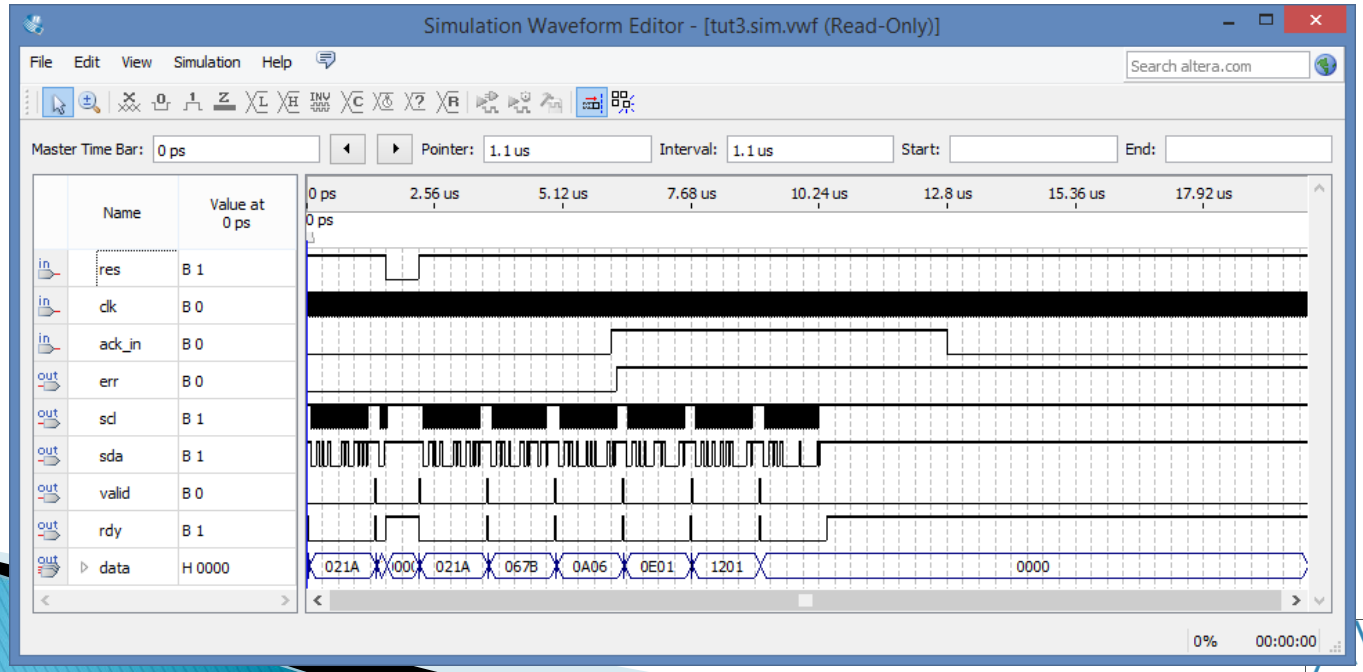
- Soluzione 1

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


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

Azioni sconsigliate

- Generare nuovi segnali si clock (anche impliciti)
  - Es: `always @ (posedge valid)`

  comporta un disallineamento (valid è utilizzato come clk)
- Sincronizzare il conteggio sullo stato di “valid” comporta un conteggio erroneo (valid dura 2 cicli di clock, per cui incrementa due volte



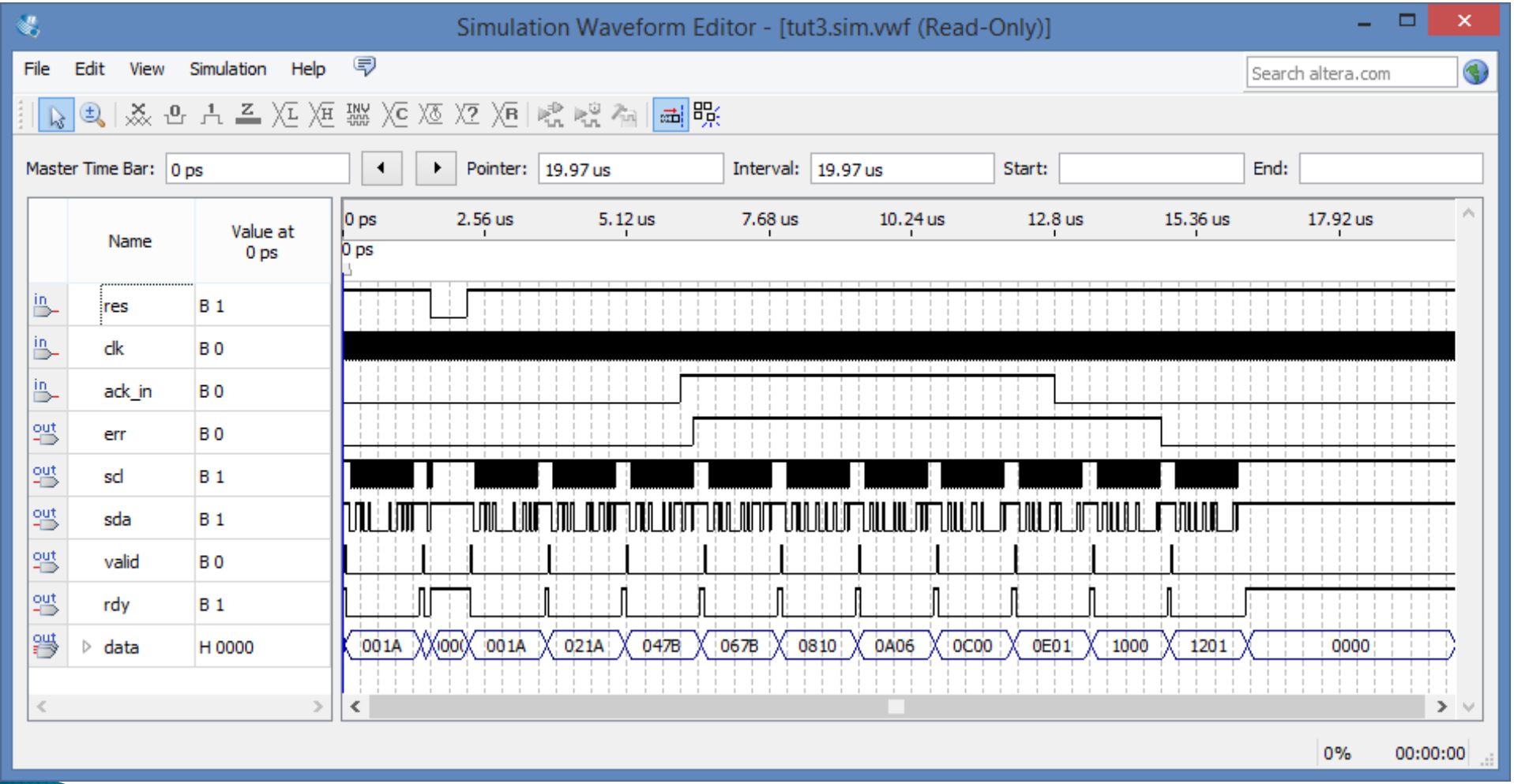
Possibile Soluzione

Usare un contatore delle fasi (a saturazione) per scandire le diverse fasi di trasmissione

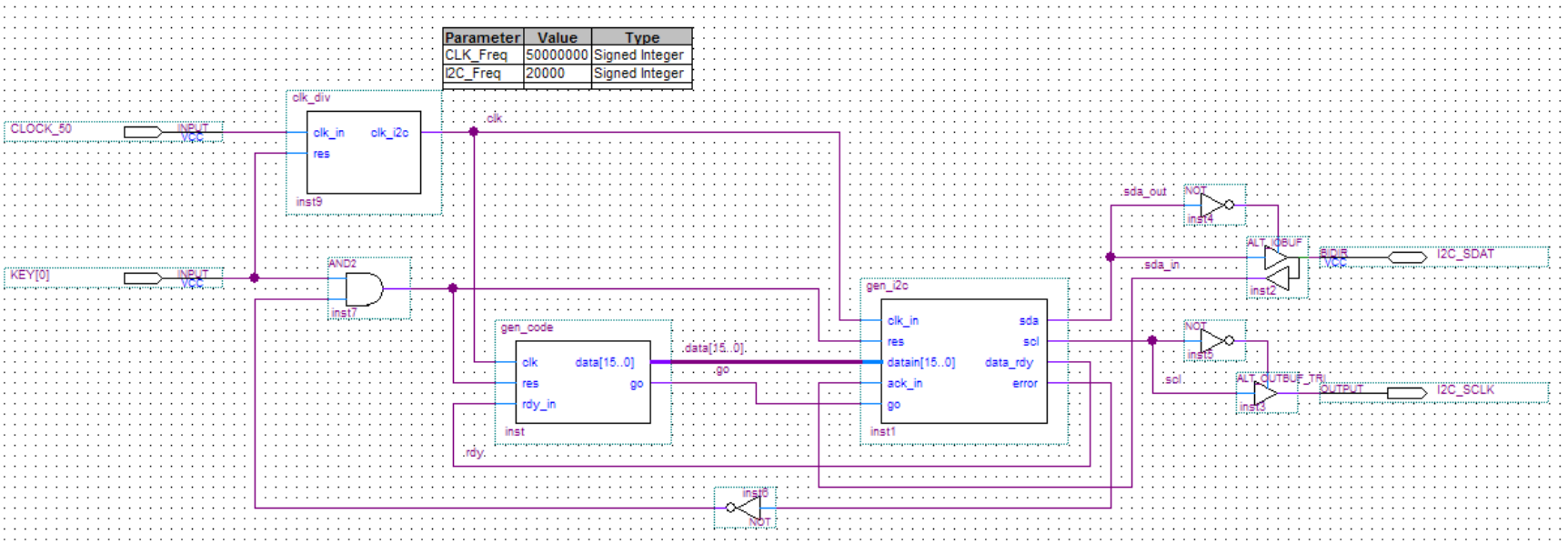
- Es: all'istante 1 incrementa il conteggio, all'istante 5 alza il segnale valid e lo abbassa all'istante 6

```
// contatore delle fasi
always @(posedge clk or negedge res) begin
    if (!res)  fasi<=3'b000;
    else if (!rdy_in) fasi<=3'b000;
    else if ((rdy_in)&(fasi < 3'b111)) fasi<=fasi+3'b001;
    end

// generazione segnale "valid"
always @(posedge clk or negedge res) begin
    if (!res)  valid=0;
   else if ((fasi==3'b101)&(addr < `rom_size)) valid=1;
   else valid=0;
end

// incremento dell'indirizzo
always @(posedge clk or negedge res) begin
    if (!res) addr = 0;
    else if ((addr < `rom_size)&(fasi==3'b001)) addr=addr+1; end
```

Simulazione



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
.sda_out
.sda_in
.scl
NOT
inst4
inst5
inst6
ALT_IOBUF
inst2
ALT_OUTBUF_TRI
inst3
BIDIR
I2C_SDAT
OUTPUT
I2C_SCLK