1 Lezione del 10-12-24

1.1 Conversione digitale/analogico e analogico/digitale

Finora abbiamo assunto che le interfacce lavorino solo e soltanto con segnali digitali. In verità nel mondo esterno al computer le grandezze variano su un una scala continua. Occoronno appositi convertitori, detti convertitori digitale/analogico e analogico/digitale.

La grandezza analogica che consideriamo è un voltaggi appartenente alla scala FSR (Full Scale Range) $[5, 30]$V. Questa verrà convertita in un intero x rappresentato su N bit con $N \in \{8, 16\}$. A seconda dell'interpolazione scelta, potremo distinguere fra:

- **Conversione ubipolare:** $v \in [0, FSR]$, $x \in [0, 2^N - 1]$;
- **Conversione bipolare:** $v \in \left[-\frac{FSR}{2}, \frac{FSR}{2}\right]$, $x \in \left[-2^{N-1}, +2^{N-1} - 1\right]$

1.1.1 Errori di conversione

Dato $K = \frac{FSR}{2^N}$, nel caso ideale vorremmo $v = Kx$. In realtà, avremo che $|v - Kx| \leq \varepsilon$, con un ε dovuto a errori di conversione:

- **Errore di non linearità**;
- **Errore di quantizzazione**.

1.1.2 Tempi di risposta

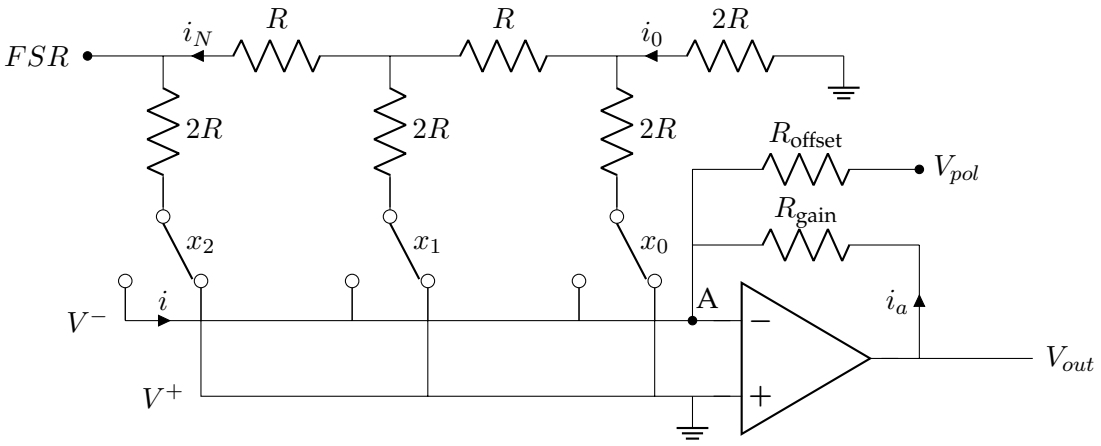
I convertitori D/A sono praticamente "combinatori", e quindi estrememente veloci (pochi nanosecondi). I convertitori A/D, di contro, hanno tempi di risposta variabili in base alle architetture. Noi vedremo i convertitori ad **approssimazioni successive** (**SAR**), che hanno tempi di risposta di qualche centinaio di nanosecondi.

1.1.3 Convertitori unipolari/bipolari

I convertitori D/A che lavorano con i numeri naturali usano la stessa rappresentazione in codice binario a cui siamo abituati. I convertitori bipolari che lavorano con interi usano invece rappresentazioni in traslazione (detta appunto anche *binaria bipolare*), anziché in complemento a 2. Il numero intero x viene quindi rappresentato dal naturale $X = x + 2^{N-1}$. In ogni caso, per riportare in complemento a 2 basterà complementare il MSB, o analogamente sommare all'intero in questione 2^{n-1}.

1.2 Convertitore D/A

Un convertitore D/A può essere schematizzato come segue:



La linea superiore è collegata a tensione FSR, cioè il valore massimo (fondo scala) del convertitore. Si ha che, per resistenze parallele, la resistenza a destra di ogni ramo verticale (con le resistenze da $2R$) vale $2R$, e quindi la resistenza a sinistra del ramo vale R. Assunte le due resistenze da $2R$ a destra collegate a massa (come sarà con gli interruttori commutati a destra, che per adesso assumiamo), si ha che la corrente che passa su di esse sarà i_0, quella che passa sulle due resistenze alla loro destra sarà $2i_0$, e quindi che la corrente che passa su ogni ramo verticale sarà il doppio della corrente che passa sul ramo verticale immediatamente a destra. Il valore di questa i_0 è dato prendendo la corrente che esce al nodo FSR, che sarà chiaramente:

$$i_n = \frac{FSR}{R}$$

e osservando che la corrente che passa sulla prima resistenza da $2R$ (quella collegata a massa) dovrà essere:

$$i_0 = \frac{FSR}{R} \cdot \frac{1}{2^n} = \frac{K}{R}$$

Vediamo velocemente che la cosa si mantiene anche commutando gli interruttori a sinistra. L'amplificatore operazionale mantiene l'uscita V_{out} a tensione:

$$V_{out} = \alpha \cdot (V^+ - V^-) = \alpha \cdot V^-$$

visto che che la linea V^+ è collegata a massa. Allo stesso tempo, dal lato sinistro dell'amplificatore, la resistenza R_{gain} (che vedremo in seguito ha valenza, assieme alla resistenza R_{offset}, per la correzione dell'errore di non linearità del convertitore), ci fornisce:

$$V_{out} = R_{gain} \cdot i_a + V^-$$

da cui:

$$R_{gain} \cdot i_a + V^- = \alpha \cdot V^-, \quad R_{gain} \cdot i_a = V^-(\alpha - 1) \implies V^- = \frac{R_{gain} \cdot i_a}{\alpha - 1}$$

da cui, con $\alpha >> 1$, $V^- \approx 0$, cioè molto vicina a massa.

Si ha quindi che la corrente che passa sulla linea V^- vale:

$$i = i_0 \cdot x_0 + 2i_0 \cdot x_1 + 4i_0 \cdot x_2$$

che è esattamente il naturale codificato da $X = \{x_2 x_1 x_0\}$, cioè su 3 cifre. Potremo infatti generalizzare a un numero n di cifre arbitrario:

$$i = i_0 \cdot x_0 + 2i_0 \cdot x_1 + ... + 2^{n-1} i_0 \cdot x_{n-1} = i_0 \sum_{i=0}^{n-1} 2^i \cdot x_i = i_0 \cdot X$$

Infine, sostituendo il valore ricavato prima per i_0, otteniamo:

$$i = \frac{K}{R} \sum_{i=0}^{n-1} 2^i \cdot x_i = \frac{K}{R} \cdot X$$

Vediamo allora come i resistori R_{gain} e R_{offset} influenzano il segnale V_{out} in uscita, e come la tensione V_{pol} trasforma il comportamento del convertitore da unipolare a bipolare. Sostituendo la formula trovata prima per V^- nell'equazione dell'amplificatore operazionale, si trova immediatamente:

$$V_{out} = \alpha \cdot \frac{R_{gain} \cdot i_a}{\alpha - 1}$$

che assumendo come prima $\alpha >> 1$ restituisce:

$$V_{out} = R_{gain} \cdot i_a$$

cioè la resistenza R_{gain} funge da fattore di scala per il voltaggio V_{out} in uscita.

Impostando poi il bilancio delle correnti al nodo A, si ottiene:

$$i = \frac{K}{R} \cdot X = \frac{V_{out}}{R_{\text{gain}}} + \frac{V_{pol}}{R_{\text{offset}}} \implies V_{out} = R_{\text{gain}} \left(\frac{K}{R} \cdot X - \frac{V_{pol}}{R_{\text{offset}}} \right)$$

da cui notiamo che la resistenza R_{offset} regola l'intercetta, e la resistenza R_{gain} la pendenza della retta che lega X a V_{out}. Inoltre, impostando $V_{pol} = 0$ si ottiene un **converitore unipolare**, mentre impostando $V_{pol} = \frac{FSR}{2}$ si ottiene un **converitore bipolare**.

Anche se non si considerano resistori e amplificatori operazionali come componenti combinatori, il circuito è effettivamente "combinatorio" nel senso che ha tempi di risposta estremamente veloci. Il problema è però quello delle transizioni multiple dello stato di uscita: questo si risolve attraverso un filtro *passa-basso* in uscita.

In Verilog, un implementazione semplificata di un convertitore D/A può essere la seguente, che emula un segnale analogico usando variabili di tipo `real`:

```
// un convertitore digitale-analogico a 3 bit di controllo e fondo
// scala a 10 volts
module digital_analog_converter(x2_x0, a_out);
  input[2:0] x2_x0;
  output real a_out;

  parameter real FSR = 10;
  parameter real K = FSR / (2 ** 3);

  always @(*) begin
    a_out = x2_x0 * K;
  end
endmodule
```

1.2.1 Interfaccia per la conversione D/A

Vediamo un'interfaccia parallela per l'operazione di un convertitore D/A. Sul lato di uscita non si avranno handshake, in quanto il convertitore è in sé più veloce del clock del processore.

```
// un'interfaccia di conversione digitale-analogico
module digital_analog_interface(s_, iow_, d7_d0, a_out);
  input s_, iow_;
  input[7:0] d7_d0;
  output real a_out;

  wire[7:0] byte_out;

  parallel_out p_out(
    .s_(s_), .iow_(iow_),
    .d7_d0(d7_d0),
    .byte_out(byte_out)
  );

  digital_analog_converter dac (
    .x7_x0(byte_out),
    .a_out(a_out)
  );
endmodule
```

1.3 Convertitore A/D

Descriviamo un particolare tipo di convertitori A/D detto convertitore ad **approssimazioni successive** (alternative potrebbero essere i convertitori *paralleli* o i convertitori *a rampa*, anch'essi basati su comparatori). Il cuore di un convertitore di questo tipo è una rete sequenziale detta **SAR** (Successive Approximation Register). L'uscita del SAR viene fatta passare attraverso un convertitore D/A dello stesso tipo dell'A/D, e confrontata attraverso un **comparatore** con l'ingresso corrente in modo da migliorare la previsione, in quella che è effettivamente una **ricerca logaritmica** (o *binaria* o *dicotomica*). In particolare, ad ogni iterazione della ricerca si ricava il valore di un singolo bit, per cui n bit richiedono n iterazioni. Lato processore, il SAR dovrà implementare inoltre un handshake, che scegliamo `soc/eoc`.

Una descrizione in linguaggio Verilog della SAR potrebbe essere la seguente. Si noti che si presentano due versioni: il problema della prima è che abbiamo bisogno di un nuovo stato per ogni iterazione di aggiornamento di RBR; si introducono quindi nella seconda versione un registro COUNT e una rete combinatoria per il calcolo dei bit di RBR.

```
// un convertitore analogico-digitale ad approssimazioni successive
// a 8 bit e fondo scala a 10 volts
module analog_digital_converter(reset_,
                                v, x7_x0, digit
                                soc, eoc);
  input reset_;
  input real v;
  output[7:0] x7_x0;
  input soc;
  output eoc;

  parameter real FSR = 10;
  parameter real K = FSR / (2 ** 8);
```

```
14
15    wire i7_i0;
16    assign x7_x0 = i7_i0;
17
18    wire a_out;
19
20    digital_analog_converter #(.FSR(FSR)) dac (
21      .x7_x0(i7_i0), .a_out(a_out)
22    );
23
24    wire digit;
25    assign digit = v > (a_out - K / 2) ? 1'B1 : 0'B0;
26
27    wire sar_clock;
28    initial sar_clock = 0;
29    always @(*) #1 sar_clock = ~sar_clock;
30
31    successive_approximation_register sar (
32      .clock(sar_clock), .reset_(reset_),
33      .x7_x0(i7_i0), .digit(digit),
34      .soc(soc), .eoc(eoc)
35    );
36  endmodule
37
38  // prima implementazione registro SAR
39  module successive_approximation_register(clock, reset_,
40                                           x7_x0, digit,
41                                           soc, eoc);
42    input clock, reset_;
43    input real v;
44    output[7:0] x7_x0;
45    input digit;
46    input soc;
47    output eoc;
48
49    reg[7:0] RBR;
50    assign x7_x0 = RBR;
51
52    reg EOC;
53    assign eoc = EOC;
54
55    reg[3:0] STAR;
56    localparam
57      s0 = 0,
58      s1 = 1,
59      s2 = 2,
60      s3 = 3,
61      s4 = 4,
62      s5 = 5,
63      s6 = 6,
64      s7 = 7,
65      s8 = 8,
66      s9 = 9,
67      s10 = 10;
68
69    always @(reset_ == 0) #1 begin
70      STAR <= s0;
71      EOC <= 1;
72    end
73
```

```
74    always @(posedge clock) if(reset_ == 1) #3 begin
75      casex(STAR)
76        s0 : begin
77          EOC <= 1;
78          STAR <= (soc == 1) ? s1 : s0;
79        end
80        s1 : begin
81          RBR <= 8'B1000_0000;
82          EOC <= 0;
83          STAR <= s2;
84        end
85        s2 : begin
86          RBR <= {alpha, 'B100_0000};
87          STAR <= s3;
88        end
89        s3 : begin
90          RBR <= {RBR[7], alpha, 'B10_0000};
91          STAR <= s4;
92        end
93        s4 : begin
94          RBR <= {RBR[7:6], alpha, 'B1_0000};
95          STAR <= s5;
96        end
97        s5 : begin
98          RBR <= {RBR[7:5], alpha, 'B1000};
99          STAR <= s6;
100       end
101       s6 : begin
102         RBR <= {RBR[7:4], alpha, 'B100};
103         STAR <= s7;
104       end
105       s7 : begin
106         RBR <= {RBR[7:3], alpha, 'B10};
107         STAR <= s8;
108       end
109       s8 : begin
110         RBR <= {RBR[7:2], alpha, 'B1};
111         STAR <= s9;
112       end
113       s9 : begin
114         RBR <= {RBR[7,1], alpha};
115         STAR <= s10;
116       end
117       s10 : begin
118         STAR <= (soc == 0) ? s0: s10;
119       end
120     endcase
121   end
122 endmodule
123
124 // seconda implementazione registro SAR, piu' compatta
125 module successive_approximation_register(clock, reset_,
126                                          x7_x0, digit,
127                                          soc, eoc);
128   input clock, reset_;
129   input real v;
130   output[7:0] x7_x0;
131   input digit;
132   input soc;
133   output eoc;
```

```
134
135 reg[7:0] RBR;
136 assign x7_x0 = RBR;
137
138 reg EOC;
139 assign eoc = EOC;
140
141 reg[2:0] COUNT;
142
143 reg[1:0] STAR;
144 localparam
145   s0 = 0,
146   s1 = 1,
147   s2 = 2,
148   s3 = 3;
149
150 always @(reset_ == 0) #1 begin
151   STAR <= s0;
152   COUNT <= 7;
153   EOC <= 1;
154 end
155
156 function[7:0] newbyte;
157   input[7:0] rbr;
158   input digit;
159   input[2:0] count;
160   casex(count)
161     7: newbyte = {          digit, 'B100_0000};
162     6: newbyte = {rbr[7],   digit, 'B10_0000};
163     5: newbyte = {rbr[7:6], digit, 'B1_0000};
164     4: newbyte = {rbr[7:5], digit, 'B1000};
165     3: newbyte = {rbr[7:4], digit, 'B100};
166     2: newbyte = {rbr[7:3], digit, 'B10};
167     1: newbyte = {rbr[7:2], digit, 'B1};
168     0: newbyte = {rbr[7:1], digit};
169   endcase
170 endfunction
171
172 always @(posedge clock) if(reset_ == 1) #3 begin
173   casex(STAR)
174     s0 : begin
175       EOC <= 1;
176       COUNT <= 7;
177       STAR <= (soc == 1) ? s1 : s0;
178     end
179     s1 : begin
180       RBR <= 8'B1000_0000;
181       EOC <= 0;
182       STAR <= s2;
183     end
184     s2 : begin
185       RBR <= newbyte(RBR, digit, COUNT);
186       COUNT <= COUNT - 1;
187       STAR <= (COUNT == 0) ? s3 : s2;
188     end
189     s3 : begin
190       STAR <= (soc == 0) ? s0 : s3;
191     end
192   endcase
193 end
```

```
endmodule
```

1.3.1 Interfaccia per la conversione A/D

Vediamo un'interfaccia parallela per l'operazione di un convertitore A/D. Lato processore si implementerà, come abbiamo visto, un handshake `soc/eoc`.

Un'implementazione in Verilog può essere la seguente:

```
// un'interfaccia di conversione analogico-digitale
module analog_digital_converter(clock, reset_,
                                s_, ior_, iow_, a0, d7_d0,
                                v);
  input clock, reset_;
  input reset_;
  input s_, ior_, iow_;
  input a0;
  inout d7_d0;
  input real v;

  reg SOC;

  wire e_x, e_s, e_e;
  hs_parallel_in_comb comb (
    .s_(s_), .ior_(ior_), .iow_(iow_), .a0(a0),
    .e_x(e_x), .e_s(e_s), .e_e(e_e)
  );

  wire[7:0] rbr;
  wire eoc;

  analog_digital_converter adc (
    .reset_(reset_),
    .v(v), .x7_x0(rbr),
    .soc(SOC), .eoc(eoc)
  )

  always @(posedge e_s) #1
    SOC <= d7_d0[1];
  assign d7_d0[0] = e_e ? eoc : 'HZ;
  assign d7_d0 = e_x ? rbr : 'HZ;

endmodule

module analog_digital_comb(s_, ior_, iow_, a0, e_x, e_s, e_e);
  input s_, ior_, iow_, a0;
  output e_x, e_s, e_e;

  assign {e_x, e_s, e_e} = ({s_, ior_, iow_, a0} == 3'B0010) ? 'B001:
                           {{s_, ior_, iow_, a0} == 3'B0100) ? 'B010:
                           {{s_, ior_, iow_, a0} == 3'B0011) ? 'B100:
                           /*           don't care          */ 'B000;
endmodule
```