FSMD examples in Gezel and in VHDL

Tutorial 06 on Dedicated systems

Teacher: Giuseppe Scollo

University of Catania
Department of Mathematics and Computer Science
Graduate Course in Computer Science, 2016-17

Table of Contents

tutorial outline

this tutorial deals with:

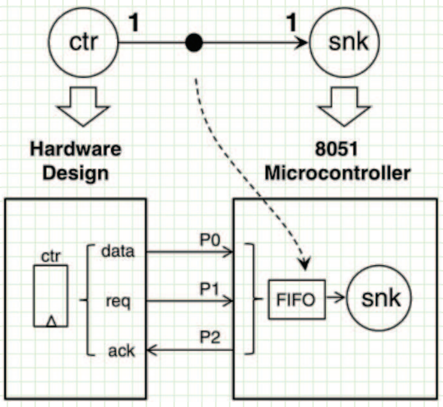
- codesign and cosimulation of FSMD models in Gezel
- hardware implementation methods for FSMD models
- FSMD design and implementation example
  - design of a median computation datapath
  - sequentialization by means of an FSMD
- FSM description in VHDL
- lab experience:
  - codesign FSMD/C and cosimulation of a testbench of the computation of the delay of Collatz trajectories

hardware/software implementation of dataflow models

problem: HW/SW partitioning the implementation of a dataflow system

a *communication protocol* is to be implemented at the HW/SW interface

here is a very simple example:



Schaumont, Figure 3.17 - Hybrid hardware/software implementation of a dataflow graph

- *handshake* protocol to synchronize token communication
- the hardware description includes actor ctr, the transmitter side of the protocol, and models of the microcontroller and of its hardware bus interfaces
- the software model includes actor snk, a FIFO queue and the protocol receiver side
- the hardware is described by an FSMD model
  - however, in this case the FSM is only given the role of observing the token communication events, to show the token value and their timing by means of a cosimulation directive

FSMD/C codesign example in Gezel

```
dp send_token( out dout : ns(8);
               out req : ns(1);
               in ack : ns(1)) {
  reg ctr : ns(8);
  reg rack : ns(1);
  reg rreq : ns(1);
  always {
    rack = ack;
    rreq = rack ? 0 : 1;
    ctr = (rack & rreq) ? ctr + 1 : ctr;
    dout = ctr;
    req = rreq;
  }
  sfg transfer {
    $display($cycle, " token ", ctr);
  }
  sfg idle {}
}

fsm ctl_send_token(send_token) {
  initial s0;
  @s0 if (rreq & rack) then (transfer) -> s0;
                       else (idle) -> s0;
}
```

```
ipblock my8051 {
iptype "i8051system";
    ipparm "exec=df.ihx";
    ipparm "verbose=1";
    ipparm "period=1"; }
ipblock my8051_data(in data : ns(8)) {
    iptype "i8051systemsink";
    ipparm "core=my8051";
    ipparm "port=P0"; }
ipblock my8051_req(in data : ns(8)) {
    iptype "i8051systemsink";
    ipparm "core=my8051";
    ipparm "port=P1"; }
ipblock my8051_ack(out data : ns(8)) {
    iptype "i8051systemsource";
    ipparm "core=my8051";
    ipparm "port=P2"; }

dp sys {
  sig data, req, ack : ns(8);
  use my8051;
  use my8051_data(data);
  use my8051_req (req);
  use my8051_ack (ack);
  use send_token (data, req, ack);
}

system S { sys; }
```

```
#include <8051.h>
#include "fifo.c"

void collect(fifo_t *F) {
if (P1) { // if hardware has data
  put_fifo(F, P0); // then accept it
  P2 = 1; // indicate data was taken
  while (P1 == 1); // wait until hardware ack
    P2 = 0; // and reset
  }
}

unsigned acc;
void snk(fifo_t *F) {
  if (fifo_size(F) >= 1)
    acc += get_fifo(F);
}

void main() {
  fifo_t F1;
  init_fifo(&F1);
  put_fifo(&F1, 0); // initial token
  acc = 0;
  while (1) {
    collect(&F1);
    snk(&F1);
  }
}
```

Schaumont, Listing 3.5 - GEZEL hardware description of data flow example of Fig. 3.17

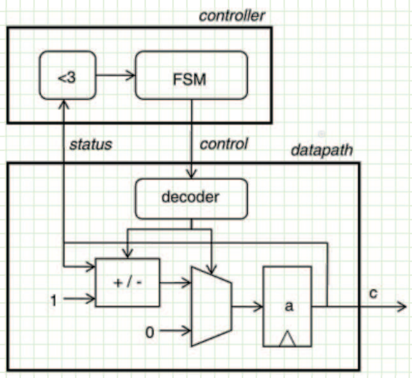
hardware implementation of FSMD models

hardware mapping of datapath expressions: same basic rules as with always blocks, however:

hardware datapath structure also depends on instruction schedule by FSM...

why?

consider the up-down counter example from the previous lecture:



Schaumont, Figure 5.8 - Implementation of the up-down counter FSMD

- by the FSM model, different instructions will always execute in different clock cycles
- instructions which are never concurrent may *share* operators' hardware
  such as the adder/subtractor in this case
- a local decoder converts the instruction encoding into control signals for the datapath, to enable the proper execution path
- a local decoder in the controller extracts datapath state information for the FSM conditions

FSMD design example: median computation

function specification: computation of the median in a list of five numbers

problem: design a *stream*-processing filter that produces a new output upon each new input

such filters find application in image processing, for noise reduction

for a fast algorithm, with no list sorting:

in a $(2n+1)$-element list, with pairwise distinct elements, the median has n smaller elements

```
dp median(in a1, a2, a3, a4, a5 : ns(32); out q1 : ns(32)) {
   sig z1, z2, z3, z4, z5, z6, z7, z8, z9, z10 : ns(3);
   sig s1, s2, s3, s4, s5 : ns(1);
   always {
      z1 = (a1 < a2);
      z2 = (a1 < a3);
      z3 = (a1 < a4);
      z4 = (a1 < a5);
      z5 = (a2 < a3);
      z6 = (a2 < a4);
      z7 = (a2 < a5);
      z8 = (a3 < a4);
      z9 = (a3 < a5);
      z10 = (a4 < a5);
      s1 = ((      z1 +       z2 +       z3 +       z4) == 2);
      s2 = (((1-z1) +       z5 +       z6 +       z7) == 2);
      s3 = (((1-z2) + (1-z5) +       z8 +       z9) == 2);
      s4 = (((1-z3) + (1-z6) + (1-z8) +     z10) == 2);
      q1 = s1 ? a1 : s2 ? a2 : s3 ? a3 : s4 ? a4 : a5;
   }
}
```

Schaumont, Listing 5.19 - GEZEL Datapath of a median calculation of five numbers

the algorithm presented by this datapath also works when some elements are equal

- the description is almost identical to that of a C function for the same algorithm
- but with a substantial, practical difference:

sequential execution of assignments in C vs. parallel execution in the datapath

where each output production requires just one clock-cycle!

provided all input values are simultaneously available ...

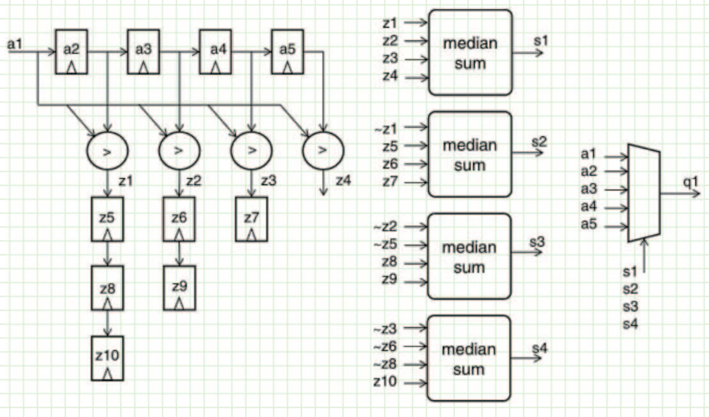
an FSMD may allow a big space saving, as we are going to see

datapath of a median computation datapath

the presented algorithm requires 192 I/O lines and 10 comparators

a stream processing filter may reduce these requirements to 64 I/O lines and 4 comparators

to this end the hardware stores the four data preceding the last one from the input stream and at every iteration with a new input element reuses the stored results of six comparisons from the previous three iterations



Schaumont, Figure 5.9 - Median-calculation datapath for a stream of values

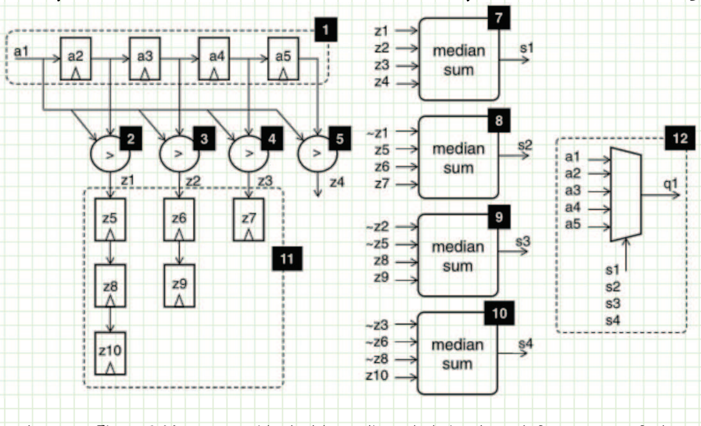
```
dp median(in a1 : ns(32); out q1 : ns(32)) {
   reg a2, a3, a4, a5 : ns(32);
   sig z1, z2, z3, z4;
   reg z5, z6, z7, z8, z9, z10 : ns(3);
   sig s1, s2, s3, s4, s5 : ns(1);
   always {
      a2 = a1;
      a3 = a2;
      a4 = a3;
      a5 = a4;
      z1 = (a1 < a2);
      z2 = (a1 < a3);
      z3 = (a1 < a4);
      z4 = (a1 < a5);
      z5 = z1;
      z6 = z2;
      z7 = z3;
      z8 = z5;
      z9 = z6;
      z10 = z8;
      s1 = ((      z1 +       z2 +       z3 +       z4) == 2);
      s2 = (((1-z1) +       z5 +       z6 +       z7) == 2);
      s3 = (((1-z2) + (1-z5) +       z8 +       z9) == 2);
      s4 = (((1-z3) + (1-z6) + (1-z8) +     z10) == 2);
      q1 = s1 ? a1 : s2 ? a2 : s3 ? a3 : s4 ? a4 : a5;
   }
}
```

sequentialization by means of an FSMD

the filter in fig 5.9 accepts a new input and produces a new output at every clock cycle

the number of computing components may be further reduced by distributing the computation over multiple clock cycles under a sequential schedule, so that a single comparator and a single module to compute s1, s2, s3, s4 are reused, at the cost of adding a few multiplexers and registers for the internal signals

the figure shows the schedule, the FSMD which implements this idea is in Schaumont Sect. 5.5.4, which also presents the testbench FSMD here reproduced aside the figure

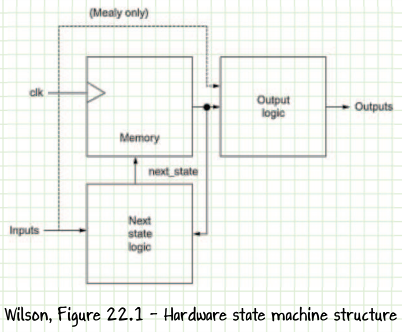


Schaumont, Figure 5.10 - Sequential schedule median-calculation datapath for a stream of values

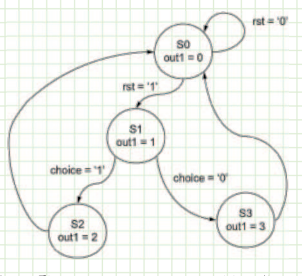
```
dp t_median {
    sig istr, ostr : ns(1);
    sig a1_in, q1 : ns(32);
    use median(istr, a1_in, ostr, q1);
    reg r : ns(1);
    reg c : ns(16);
    always { r = ostr; }
    sfg init { c = 0x1234; }
    sfg sendin { a1_in = c;
                    c = (c[0] ^ c[2] ^ c[3] ^ c[5]) # c[15:1];
                    istr = 1; }
    sfg noin { a1_in = 0;
                istr = 0; }
}
fsm ctl_t_median(t_median) {
    initial s0;
    state s1, s2;
    @s0 (init, noin) -> s1;
    @s1 (sendin) -> s2;
    @s2 if (r) then (noin) -> s1;
                else (noin) -> s2;
}
```

FSM description in VHDL

general hardware structure of an FSM and example of VHDL description of a Moore FSM:



Wilson, Figure 22.1 - Hardware state machine structure



Wilson, Figure 22.2 - State transition diagram

the VHDL description of the example is taken from the textbook by Wilson, with several amendments

what about the sensitivity list of the second process? the author refers to an implicit clock... the simulation agrees with this, yet the compilation issues a warning in this respect

```
library ieee;
use ieee.std_logic_1164.all;
entity fsm is
    port(
        clk, rst, choice : in std_logic;
        out1 : out std_logic_vector(1 downto 0)
    );
end entity fsm;

architecture simple of fsm is
        type state_type is ( s0, s1, s2, s3 );
        signal current, next_state : state_type;
begin
    process ( clk )
    begin
        if ( clk = '1' ) then
            current <= next_state;
        end if;
    end process;
```

```
    process ( current )
    begin
        case current is
            when s0 =>
                out1 <= "00";
                if ( rst = '1') then
                    next_state <= s1;
                else
                    next_state <= s0;
                end if;
            when s1=>
                out1 <= "01";
                if ( choice = '0') then
                    next_state <= s3;
                else
                    next_state <= s2;
                end if;
            when s2=>
                out1 <= "10";
                next_state <= s0;
            when s3=>
                out1 <= "11";
                next_state <= s0;
        end case;
    end process;
end;
```

lab experience

target of this experience is the HW/SW codesign and the cosimulation, in the Gezel platform gplatform, of an FSMD model for the computation of the delay of Collatz trajectories

the functionality of the hardware datapath is similar to that of the circuit for the delay of Collatz trajectories presented in the second lecture and worked out in the second lab experience, yet with a few differences due to the communication protocol with the software and to design decisions relating to cosimulation; in particular:

- the hardware configuration includes an 8051 microcontroller with the 8-bit parallel ports P0, P1, P2, P3
- the size of interface signals and registers for data exchange is thus halved to 8 bit, and as well it is halved to 16 bit the size of internal signals and registers for trajectory computation
- the handshake protocol for data exchange is similar to that shown in the FSDM/C codesign example presented, but with an important difference: in this case there are two distinct handshakes, one to send the trajectory start point from the software to the datapath, the other to send the result from the datapath to the software
- the software plays the same role as the tester in the testbench developed in the third lab experience, viz. to drive the datapath with the numbers ranging from 1 to 255 as trajectory start points
- the hardware component is to be described by an FSMD where, for each double data exchange with the software, the controller FSM goes through a four-state sequence: initial, receiver-handshake for the trajectory start point, delay computation, sender-handshake for the result; then back to the initial state
- by proper usage of the $display cosimulation directive, the clock cycle is to be shown at every beginning and ending of the trajectory computation (entry and exit into/from the third aforementioned state), together with the values of the trajectory start point and of the computed delay

references

recommended readings:

Schaumont (2012) Ch. 3, Sect. 3.3; Ch 5, Sect. 5.4.4-5.5

Wilson (2015) Ch. 22

readings for further consultation:

Vahid (2006) Cap. 9

Zwolinski (2004) Sect. 5.5-6, 7.1-2