IL2212 EMBEDDED SOFTWARE

Theoretical Homework 3 (Part C)

VERSION 1.0

Revision History

- Version 1.0: Initial version

Requirements

The total amount of points in the homework is 20 points. To pass the homework 14 points are required.

Homework Tasks

IMPORTANT NOTE: Please observe, that a successful implementation of the C-programs in Task 2 and Task 3 **will give also points for the practical homework**!

1. (3 POINTS) As part of the design space exploration phase, a number of candidate solutions have been generated and their performance figures are summarised in the following table. The performance numbers are normalised. In all cases (Execution time, power consumption, design cost) a lower number is better!

Candidate Solution	Execution Time (E)	Power Consumption (P)	Design Cost (C)
A	17	21	33
B	15	18	20
C	23	15	21
D	10	25	22
E	11	17	41
F	13	27	25
G	14	22	23
H	20	32	17

   (a) Assume that a cost function is used to identify the best solution. What is the best solution, if the following weights are used to emphasize

      i. a low power solution at reasonable costs, where the cost function is $Cost_1 = 0.1E + 0.5P + 0.4C$, or

ii. a high performance solution, where the cost function is $Cost_2 = 0.6E + 0.2P + 0.2C$

(b) Determine graphically the Pareto points, if only execution time and design costs are taken into account (power consumption is ignored).

(c) Assume now that there are hard design constraints, on all three design parameters: $E \leq 20$, $P \leq 30$, and $C \leq 35$. Give the Pareto points for the candidate solutions that fullfill all design constraints. Which candidate solutions are *Pareto points*?

2. (7 POINTS) Given is the following Lustre program `program_1`, which shall be compiled into a finite state machine implementation in C. As a first step, a finite state machine control structure shall be generated, before the C-code is generated based on this finite state machine.

```
1  node program_1 (a, b: bool) returns (x, y: bool);
2    var s: bool;
3  let
4    -- Forbidden: a and b shall not be active at the same time
5    assert(not (a and b));
6
7    s = false -> if ((not pre(s)) and a) or
8                    (pre(s) and (not a) and (not b))
9                 then
10                  true
11                else
12                  false;
13
14   x = b or (a and pre(s));
15   y = pre(s) and b;
16 tel.
```

(a) Choose suitable *state variables* as starting point to create the control structure based on a finite state machine (automaton).

(b) Create the control structure as a finite state machine by starting from the initial state and then simulating the behaviour of the state variables.

(c) Draw the state diagram of the resulting finite state machine.

(d) Give a short description of the functionality of `program_1` in your own words.

(e) Give the C-code for the Lustre program based on the generated finite state machine.

(f) Compare `program_1` with the Lustre `program_2`.

```
1  node program_2 (a, b: bool) returns (x, y: bool);
2    var s, t : bool;
3  let
4    -- Forbidden: a and b shall not be active at the same time
5    assert(not (a and b));
6
7    s = false -> if ((not pre(s)) and pre(t) and a) or
8                     b
9                 then
10                  true
11                else
12                  false;
13
14   t = false -> if ((not pre(t)) and a) or
15                    (pre(s) and a) or
```

```
16                      ((not (pre(s)) and pre(t) and b)) or
17                      ((not (pre(s)) and pre(t) and (not a)))
18                  then
19                    true
20                  else
21                    false;
22
23     x = s;
24     y = s and t;
25   tel.
```

The program `program_2` will generate the same output as `program_1`, but it has a different underlying concept. What are the main differences of these programs, and why do they still produce the same outputs. Compare both programs regarding an implementation in C following the techniques discussed in the course?

3. (10 POINTS)

NOTE: The SDF-model uses the same SDF-graph as in Task 4 of the theoretical homework 1. You can use the results of that task without motivation to solve this task!

The following synchronous data flow (SDF) model shall be implemented in C.

```
1   module SDF_Application where
2
3   import ForSyDe.Shallow
4
5   system :: Signal Int -> Signal Int -> Signal Int
6   system s_in1 s_in2 = s_out where
7     s_1 = actor_a s_in1
8     s_2 = actor_b s_in2
9     s_3 = actor_c s_1 s_4_delayed
10    (s_4, s_out) = actor_d s_2 s_3
11    s_4_delayed = delaySDF [0] s_4
12
13  actor_a :: Signal Int -> Signal Int
14  actor_a = actor11SDF 2 1 f_1 where
15    f_1 [x, y] = [x + y]
16
17  actor_b :: Signal Int -> Signal Int
18  actor_b = actor11SDF 1 2 f_2 where
19    f_2 [x] = [x, x+1]
20
21  actor_c :: Signal Int -> Signal Int -> Signal Int
22  actor_c = actor21SDF (2,1) 1 f_3 where
23    f_3 [x, y] [z] = [x + y + z]
24
25  actor_d :: Signal Int -> Signal Int -> (Signal Int, Signal Int)
26  actor_d = actor22SDF (2,1) (1,2) f_4 where
27    f_4 [x, y] [z] = ([x + y + z], [x + y, x + y + z])
```

The following functions shall be used to create FIFOs and access them:

- `channel createFIFO(token* buffer, size_t size)`: creates a FIFO and returns a channel id of type `channel`.
  In order to create a FIFO, first buffer space needs to be allocated as in the following statement, where the space for a buffer of size 2 is allocated.

```
token* buffer = malloc(2 * sizeof(token));
```

- **void** `readToken(channel ch, token* data)`: reads a token from channel `ch`. The read token can be accessed via the variable `data`.
- **void** `writeToken(channel ch, token data)`: writes a token `data` to channel `ch`.

Furthermore, the following functions shall be used to create SDF-actors.

```
void actor11SDF(int consum, int prod,
                channel* ch_in, channel* ch_out,
                void (*f) (token*, token*))
{
  token input[consum], output[prod];
  int i;

  for(i = 0; i < consum; i++) {
    readToken(*ch_in, &input[i]);
  }
  f(input, output);
  for(i = 0; i< prod; i++) {
    writeToken(*ch_out, output[i]);
  }
}

void actor12SDF(int consum, int prod1, int prod2,
                channel* ch_in, channel* ch_out1, channel* ch_out2,
                void (*f) (token*, token*, token*))
{
  token input[consum], output1[prod1], output2[prod2];
  int i;

  for(i = 0; i < consum; i++) {
    readToken(*ch_in, &input[i]);
  }
  f(input, output1, output2);
  for(i = 0; i< prod1; i++) {
    writeToken(*ch_out1, output1[i]);
  }
  for(i = 0; i< prod2; i++) {
    writeToken(*ch_out2, output2[i]);
  }
}

void actor21SDF(int consum1, int consum2, int prod,
                channel* ch_in1, channel* ch_in2, channel* ch_out,
                void (*f) (token*, token*, token*))
{
  token input1[consum1], input2[consum2], output[prod];
  int i;

  for(i = 0; i < consum1; i++) {
    readToken(*ch_in1, &input1[i]);
  }
  for(i = 0; i < consum2; i++) {
    readToken(*ch_in2, &input2[i]);
  }
  f(input1, input2, output);
```

```
50      for(i = 0; i< prod; i++) {
51        writeToken(*ch_out, output[i]);
52      }
53    }
```

(a) Write the missing C-function `actor22SDF`.

(b) Create a C-program that uses the `actorXYSDF`-functions and correctly implements the functionality of the SDF-model. The program shall continuously process input tokens from the input channels (supplied by the user via the terminal window) and write to the output channels (the output shall be visible in the terminal window). The input and output channels and all internal channel shall be modelled as FIFO buffers.

The program shall include

i. the creation of the necessary FIFO-buffers and initial tokens

ii. an endless loop, which schedules the actors in the correct order

iii. the correct function arguments to execute the actors

iv. the C-code for the functions that the actors will run when they are executed

v. the function that read inputs from a terminal window and write outputs to a terminal window.

(c) Can the C-program from the previous task be converted into a more efficient program? Explain how this can be done. You do not have to give the full code for the optimised program, but you can use code fragments to explain your solution.