Lehrstuhl für Informatik 12
Cauerstraße 11
91058 Erlangen

Friedrich-Alexander-Universität
Technische Fakultät

Embedded Systems
1. Extended Exercise

Contents

1 Introduction **2**
1.1 Execution of the Experiment . . . 2
1.2 Numerical Approximation of Differential Equations . . . 3
1.3 Architectural Synthesis . . . 5
1.4 SystemC . . . 8
1.4.1 Modelling with SystemC . . . 8
1.4.2 Simulation of SystemC Models . . . 11

2 Experiment **15**
2.1 Functional Simulation . . . 15
2.1.1 Floating-point numbers . . . 15
2.1.2 Fixed-point numbers . . . 16
2.2 Structural Refinement . . . 16
2.2.1 Synthesis . . . 16
2.2.2 Allocation of Registers . . . 17
2.2.3 Implementation . . . 17

A References **19**

1 Introduction

Integrated digital circuits are becoming increasingly complex – the processor of an average PC already has several hundred million transistors, and the trend is strongly increasing. Current design flows for such complex systems usually start with a so-called high-level model of the system, which is implemented in C/C++, for example. After the exploration of the design space, i.e. after a certain architecture has been selected, it is clear which parts of the system are to be mapped to hardware and which to software. While in a pure C/C++ model the parts to be implemented in software can easily be mapped to the corresponding processors, for the parts to be implemented in hardware a manual translation into an HDL has to be done. Not only is this error-prone and time-consuming, there is also no longer any connection between the system model and the HDL model. To avoid this break in the design flow, SystemC provides the ability to create an executable specification and supports the complete design process during hardware-software co-design. In particular, the specification and design can be performed at different levels of abstraction, e.g. purely functional or cycle-accurate. In this experiment, you will use SystemC to describe an algorithm that solves a differential equation numerically and then simulate and refine your design in the computer.

1.1 Execution of the Experiment

The tools and libraries used in this experiment run under Linux on the Chair's workstations. We therefore assume that you are at least familiar with the basic principles of this operating system.

Please prepare for the exercise, by reading the script, understanding the basics (will be checked), preparing Chapter 1 at home, and reflecting the tasks of the two chapters, otherwise experience shows that time quickly runs out. We have tried to determine the degree of difficulty of the experiment in such a way that, on the one hand, the less experienced do not face insoluble problems, but, on the other hand, nobody has to be bored either. If you cannot solve a task independently, we will explain it to you.

1.2 Numerical Approximation of Differential Equations

The goal of this experiment is to implement a numerical algorithm to solve common differential equations [2] in SystemC. This requires some theory, which you will learn more about in this Section.

Many problems of applied mathematics lead to common differential equations, which e.g. describe the temporal change of the state variables of a physical system. Only in a few cases can the desired solution be specified in closed, analytical form. Therefore, numerical methods are often necessary, which provide sufficiently accurate approximations of the solution function.
The Euler method [4] finds solutions for differential equations of first order of the form

$$y'(x) = f(x, y(x)), \text{ initial condition } y(x_0) = y_0 \tag{1}$$

where x_0 and y_0 are known. It approximates the solution curve by linearizing the gradient tangent in equidistant control points

$$x_k = x_0 + dx * k \tag{2}$$

This is done according to the calculation rule:

$$\boxed{y_{k+1} = y_k + dx * y'_k = y_k + dx * f(x_k, y_k).} \tag{3}$$

In other words: Starting from the value of the graph y_k at the position x_k you get the value y_{k+1} at the position x_{k+1} by calculating the gradient in x_k and then simply drawing a straight line with this gradient to the next control point. Because of the descriptive geometric construction, the method is also called *polygon chain method*. It leads to relatively rough approximations and can only provide acceptable results for small increments dx.

Example We consider the differential equation

$$y' = -2xy^2, \text{ initial condition } y(0) = 1 \tag{4}$$

with the exact solution

$$y(x) = \frac{1}{x^2 + 1} \tag{5}$$

The solutions for the step size $dx = 0.1$ calculated with the Euler method can be seen together with the corresponding error in table 1.2.

Task 1.1 Perform the first two iterations for the Eq. 4 and complete the entries of the table 1.2 that contain question marks.

In this experiment we will not limit ourselves to differential equations of the first order as in Eq. (1), but will solve a differential equation of the second order

$$y''(x) = f(x, y(x), y'(x)) \tag{6}$$

Control Point x_k	Correct Value $y(x_k)$	Approximated Value $y_k(x_k)$	Error e_k
0.0	1.00000	1.00000	—
0.1	0.99010	?	?
0.2	0.96154	?	?
0.3	0.91743	0.94158	-0.02415
0.4	0.86207	0.88839	-0.02632
0.5	0.80000	0.82525	-0.02525
0.6	0.73529	0.75715	-0.02185

Table 1: Approximated solution for the differential equation given in Eq. (4)

This is possible by applying the above method twice, if in addition to x_0 there are also initial values for $y(x_0)$ and $y'(x_0)$. The algorithm, to determine the value of the function in point $x = a$, looks like this:

```
**for** $x := x_0$; $x < a$; $x := x + dx$ **do**
    Step 1: Calculate $y'(x+dx) := y'(x) + dx * y''(x)$
    Step 2: Calculate $y(x+dx) := y(x) + dx * y'(x)$
**end for**
```

Example Given are the differential equations (from [5])

$$y''(x) + 3xy'(x) + 3y(x) = 0 \tag{7}$$

as well as the initial condition $x_0 = 0$, $y(x_0) = 0$, $y'(x_0) = 1$ and the step size for the iterative numeric solution $dx = 0.1$.

- Solved for $y''(x)$, this results in: $y''(x) = -3xy'(x) - 3y(x)$.
- First Iteration:
  - Application of step 1 from the algorithm specified above: $y'(0.1) = y'(0) + 0.1 * y''(0) = 1$.
  - Step 2 results in: $y(0.1) = y(0) + dx * y'(0) = 0.1$. The value of the first control point is thus known.
- Second iteration building on the values found in the first iteration:
  - Step 1: $y'(0.2) = y'(0.1) + dx * y''(0.1) = 0.94$. Here $y''(0.1)$ is calculated again from the differential equation transformed above with the values from the last step.
  - Step 2: $y(0.2) = y(0.1) + dx * y'(0.1) = 0.2$.

It is clear that the error in the numerical solution can increase rapidly with the distance from the first control point x_0.

Commercial tools for the numerical solution of differential equations such as *Mathematica* [6] use more precise procedures. The solution of Eq. (7) found by Mathematica can be seen as a graph in Figure 1.

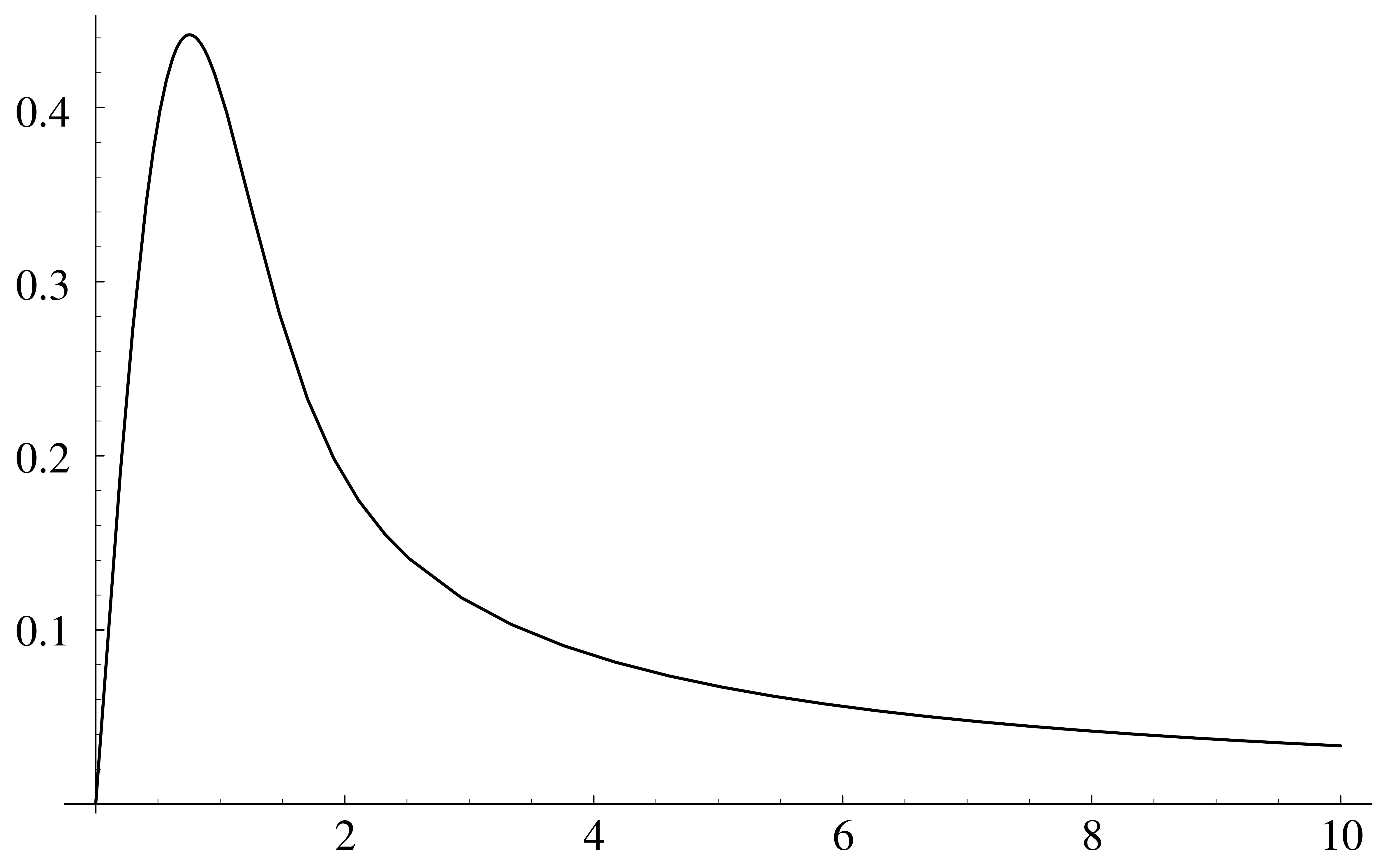


Figure 1: Numerische Lösung der DGL in Gl. (7) im Bereich von $x = 0$ bis $x = 10$

1.3 Architectural Synthesis

In addition to a functional description of the differential equation solver, a refinement of the model at the register transfer level is to be performed in this experiment.

The task of this high-level synthesis is to generate an operation unit (data path) and a control unit (control path) from a behavior description based on elementary arithmetic/logical functions (e.g. additions, multiplications, shift commands), which execute the algorithm described by the behavior description.

Example: The expression

$$result := (a+b)*(c+d) \tag{8}$$

is to be calculated in hardware. The design of the necessary system is done in the following steps:

- **Data flow graph**: From the given Eq. (8) the data flow graph shown in Figure 2 is generated. The nodes of the graph represent operations on data, the edges symbolize the flow of data between these operations.

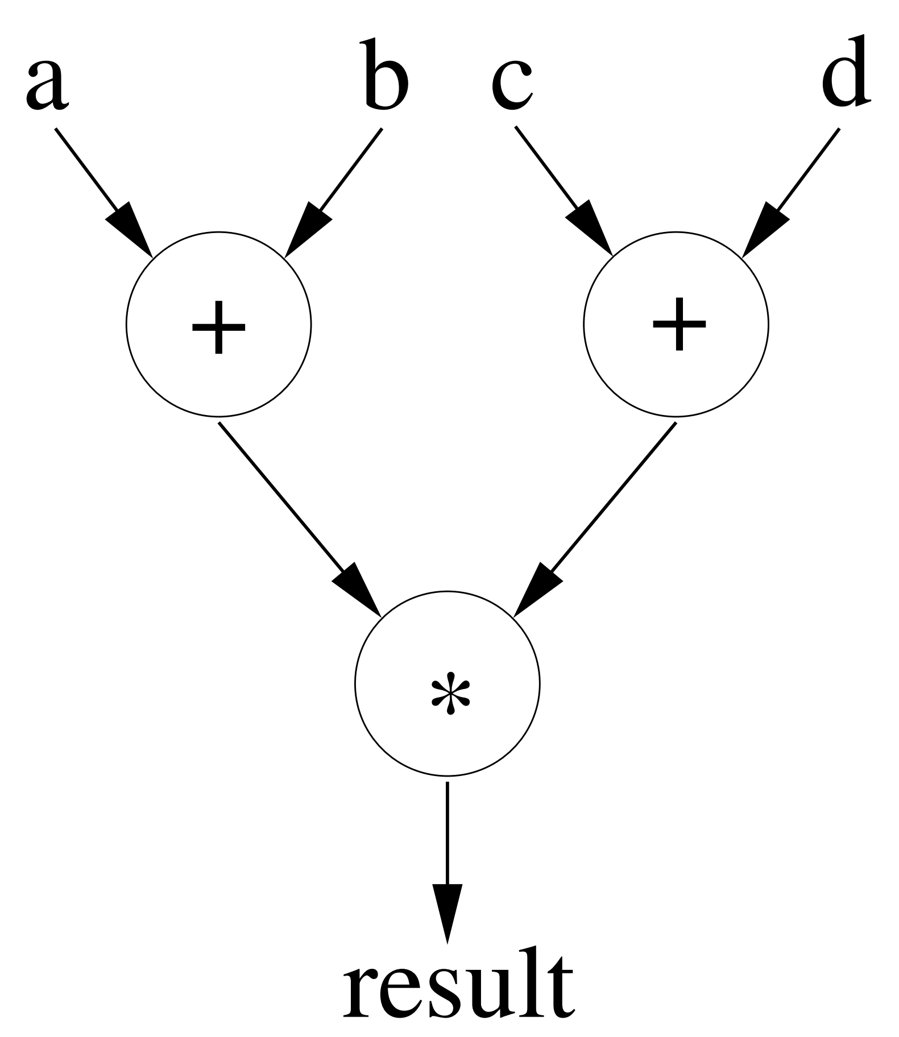


Figure 2: Eq. (8) as data flow graph.

- **Allocation**: The available functional units are to be determined. In this example, a multiplication and two additions must be calculated. This would be possible with the allocation of a universal ALU (Arithmethic Logic Unit), which can both add and subtract. Instead, however, we decide to use two functional units, namely

  - 1 Multiplier and
  - 1 Adder.

  This is done in the hope of being able to perform several operations in parallel and to speed up the calculation of the expression.

- **Scheduling**: Now it must be determined when which of the operations from the graph in Figure 2 should be executed. From the data dependencies it becomes clear that the multiplication (operation 3) may only start after the addition (operations 1 and 2) has been completed, since this requires the result of the addition operation for its own calculation. Since only one adder is available, the additions cannot be processed in parallel. The resulting schedule is shown in Figure 3; the horizontal lines symbolize clock boundaries.

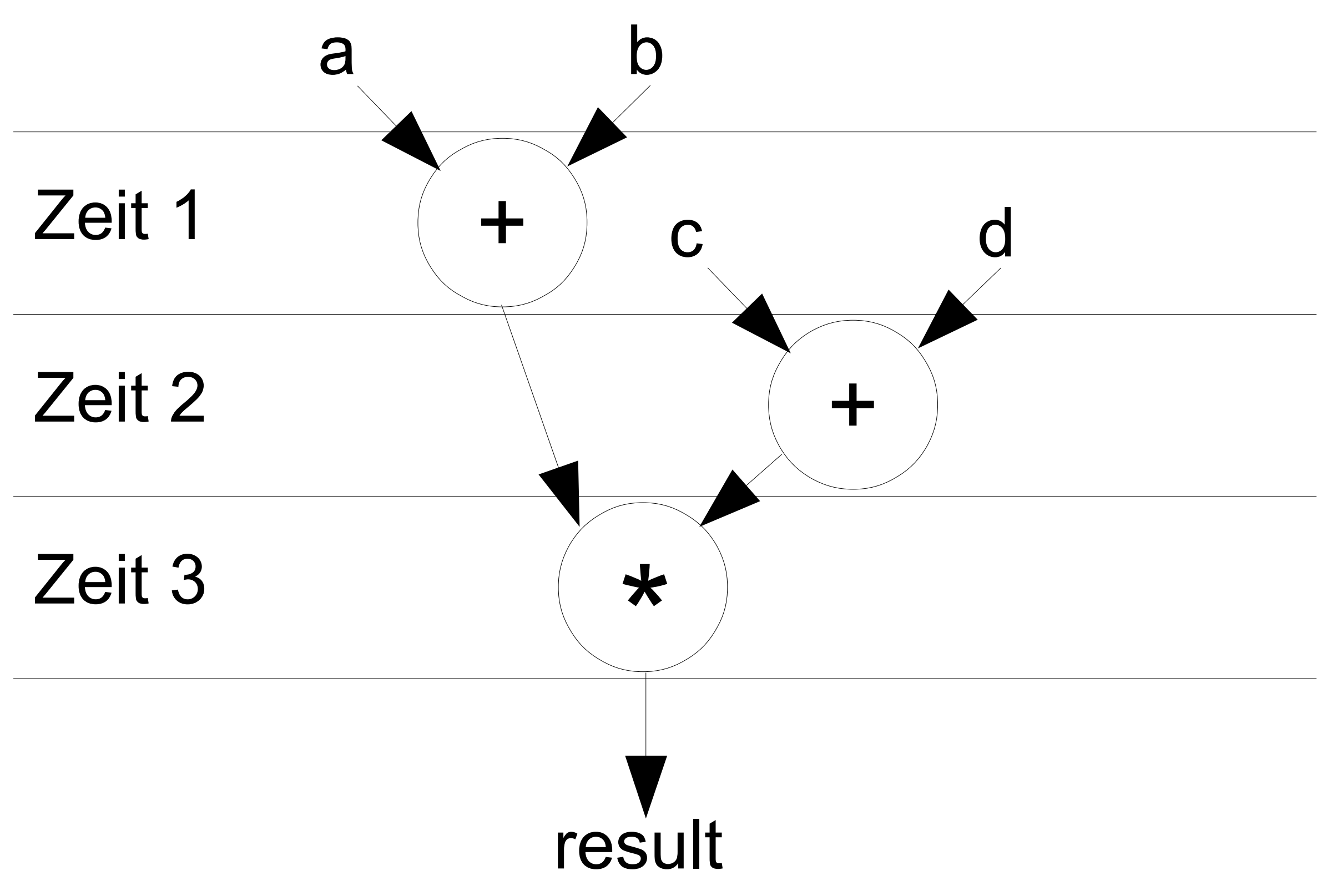


Figure 3: Data flow graph (DFG) for the Eq. (8) with the schedule.

- **Binding**: While the allocation determines which hardware units are available, the binding determines which operation of the data flow graph is performed on which instance of a functional unit.

  In this example it is simple: Operation 3 (the multiplication) is bound to the multiplier, for the two additions the binding is to the only existing adder. If more than one adder were available, a decision would have to be made here as to which of the available adder instances the additions would be bound to.

- **Generation of the arithmetic unit**: The hardware consists of an arithmetic unit.

  To build this for the differential equation from Figure 3, a mapping to a structure of multiplexers, registers and functional units as shown in Figure 4 is necessary.

  Inputs and outputs of the functional units ($e1$, $e2$) are connected with the existing registers ($r1$, ..., $r6$).

  Multiplexers ($m1$, $m2$, $m3$) enable the result of a calculation stored in a register to be direct to another functional unit ($m1$, $m2$) in the next calculation step or the results of different clock cycles and different functional units to be stored in the same register ($m3$).

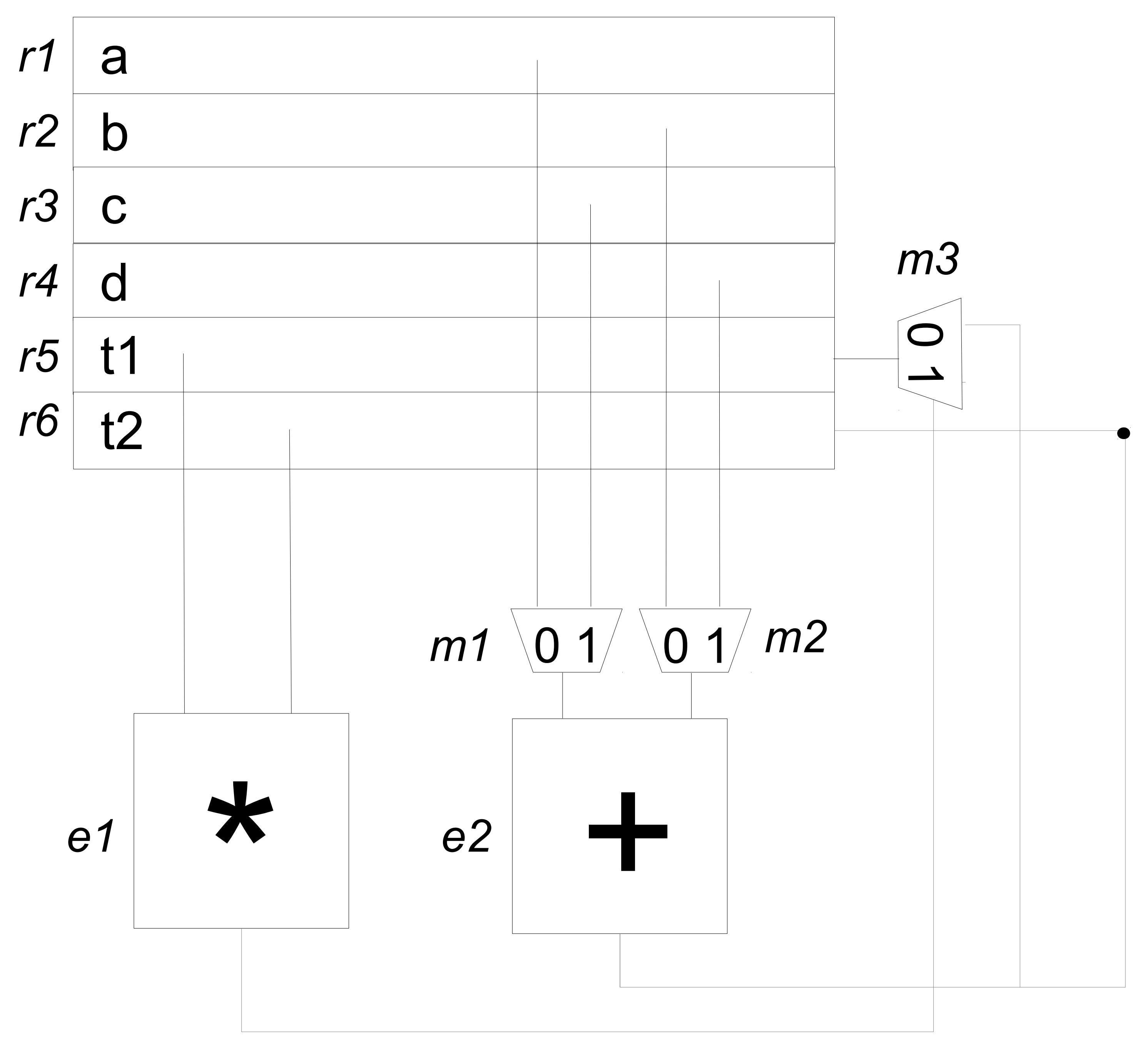


Figure 4: Basic hardware structure of the arithmetic unit generated by architectural synthesis from the differential equation.

- **Generation of the control unit**: The hardware also consists of a control unit. The control unit controls the functional units, the registers, and the multiplexers of the arithmetic unit and determines in which intervals the individual actions are activated.

  The control unit can be represented as a finite state machine (FSM), as shown in Figure 5: A state change takes place in each cycle. The FSM is a Moore machine whose output always depends on the current state.

  The output represents the control signal for the resources of the execution unit, that is 0, 1 or a - (*don't care*).

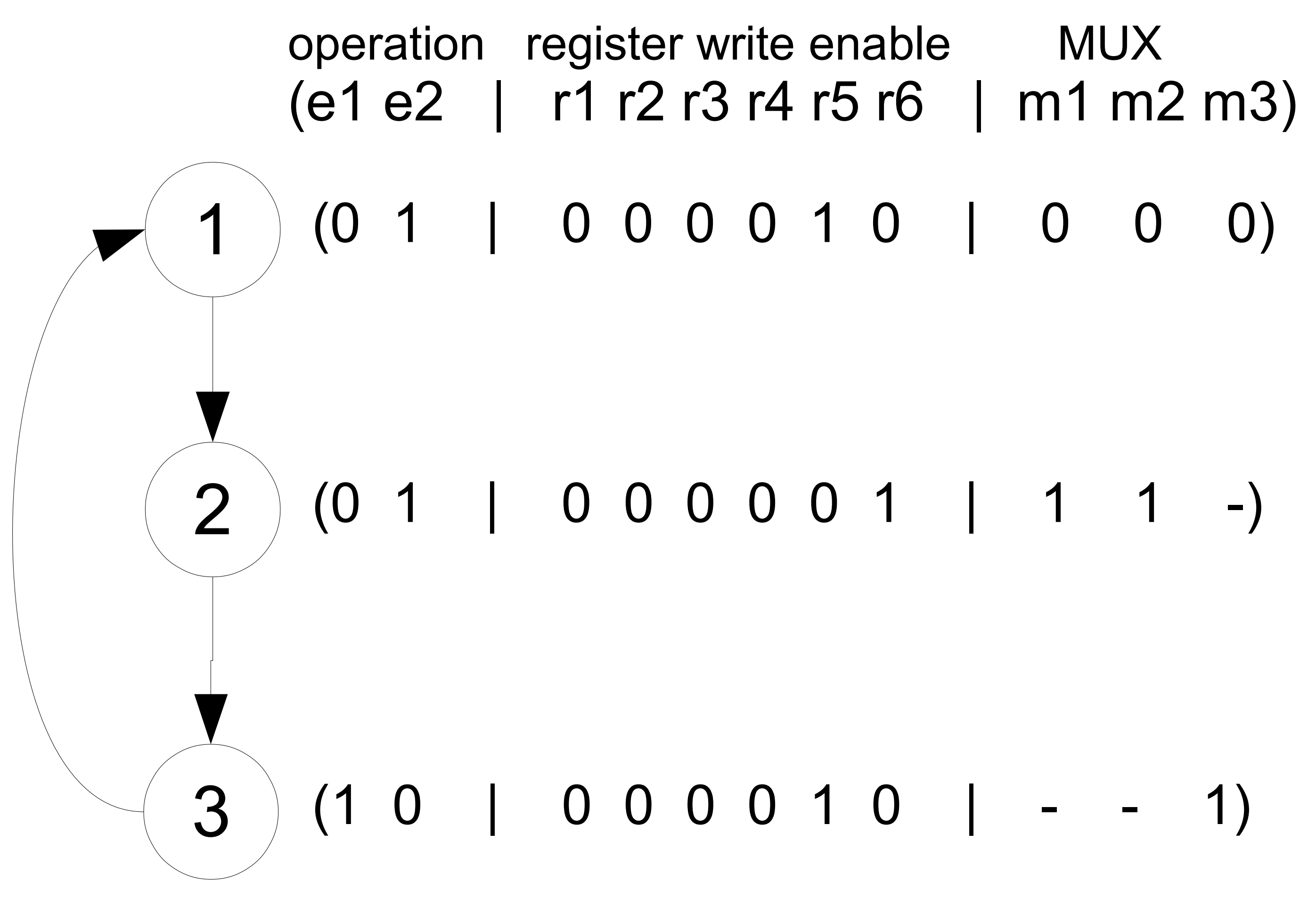


Figure 5: Moore machine, that realizes the schedule from Figure 3 and send appropriate control signals to the execution unit (Figure 4).

The above example illustrates the processes involved in architectural synthesis only very roughly. Real designs result in much more complex problems than shown here. For example, both allocation and binding have an effect on the result of the scheduling, so they cannot usually be treated as separate problems. Furthermore, the used registers and multiplexers contribute to a large extent to the area consumption of the generated hardware, so that one goal is to keep the use of these units as small as possible. Through these and some other points, the resulting Synthesis problems quickly become very complex, so that you can not try all possible solutions or by „guessing“ a schedule as in this example. Instead, optimization techniques are often used, which usually may not not produce the best possible hardware architecture, but can produce a very good one.

Task 1.2 The schedule shown in Figure 3 requires three cycles to process the Eq. (8). Can it be done faster? What disadvantages does a faster execution have in this case?

1.4 SystemC

SystemC is a system description language (and de facto standard) for modeling hardware/software systems [3]. It supports different levels of abstraction in modeling. Technically, SystemC is a C++ class library and extends C++ by the aspects:

- Concurrency
- Event-driven simulation
- Timing annotation
- Special data types for hardware modeling

The SystemC language set is standardized in IEEE Std 1666 [1]. A reference implementation of SystemC is offered by the Open SystemC Initiative (OSCI) at `www.systemc.org`.

1.4.1 Modelling with SystemC

In the following, some important modelling concepts will be presented.

SystemC-Modules As in hardware description languages, e.g. VHDL or SystemVerilog, SystemC provides language constructs for the declaration of *entities*. A *SystemC module* thus encapsulates the description in a hardware or software component. Each SystemC module must be derived from the base class `sc_module`. In SystemC, modules should only exchange data via channels. Access to a connected channel is done by so-called *Ports*. The behavior of the module is described either by composition from other modules or by concurrent processes.

Example 1.1 The following SystemC module implements a simple module to be verified (*Device Under Verification, DUV*) that only reads data from an input port and copies it unchanged to the output port.

```
class DUV: public sc_module {
  public:
    sc_fifo_in<double>  in;
    sc_fifo_out<double> out;
    double data;

  private:
    void pass() {
      while(true) {
```

```
10            data = in.read();
11            std::cout << "duv:  " << data << std::endl;
12            out.write(data);
13          }
14        }
15
16      public:
17        SC_HAS_PROCESS(DUV);
18
19        DUV(sc_module_name name) : sc_module(name) {
20          SC_THREAD(pass);
21        }
22    };
```

In line 1, the SystemC module `DUV` is declared and derived from the base class `sc_module`. Lines 3-5 contain the declaration of ports and variables. The input and output ports `in` and `out` are later connected to FIFO channels.

The actual behavior is defined in lines 8-14 of the `pass` method. Here a date is read from input port `in`, output on the standard output for debugging purposes and then written unchanged to the output port `out`.

The fact that the module `DUV` is not a purely structural description, but the behavior is implemented in the form of processes, is indicated by the macro call in line 17. Which process exactly is shown in the constructor in line 20 by the macro call `SC_THREAD`. This also indicates that it is a thread. When defining the constructor, it is expected that the constructor of the base class is called with an argument that represents the name of the module as a string.

The process type thread used in the example is comparable to Posix threads in C++ and JAVA threads. In general, a SystemC module will contain multiple concurrent processes.

Ports and Interfaces As shown in example 1.1, *Ports* are defined as objects in a class. By using the C++ keyword `public` they can be used by other objects outside the class. In the example, a predefined port type was used. It can be connected to *SystemC-FIFOs*. Besides the port types `sc_fifo_in` and `sc_fifo_out` SystemC provides further predefined port types. In the field of hardware modeling, ports for *Signals* are of particular importance. A distinction is made between input, output and input/output ports. The corresponding classes are `sc_in`, `sc_out` and `sc_inout`. All port types are implemented as template classes, where the template parameter determines the data type to be transported. This can be any C++, SystemC, or C++ user-defined data type. In the example 1.1, `double` was used.

In addition to using predefined port types, it is also possible to define your own port types. This is done by deriving it from the base class `sc_port`. This class is again a template class which receives an interface of the type `sc_interface` as template parameter. This interface only declares the methods for channel access (z. B. `read()` and `write()`). The actual implementation of these methods happens in the *channel*.

Example 1.2 The following port definition is equivalent to `sc_in`:

$$\texttt{sc\_port} < \texttt{sc\_signal\_in\_if} < \texttt{bool} >>$$

Here, `sc_signal_in_if` is a predefined interface that is derived from `sc_interface`.

Processes Essentially, SystemC distinguishes between two types of *processes*: *Threads* and *Methods*. The common features of these two process types are that both are described by sequential instructions and have their own execution thread. This means that they can be executed concurrently.

Processes are activated in the same way as VHDL using sensitivity lists. Both threads and methods can have static and dynamic sensitivity lists. The main difference between the two process types is that methods of the type `SC_METHOD` must not be interrupted, i.e. they always run from start to finish. Then, the method is terminated and possibly reactivated in the future according to the sensitivity list.

In contrast, SystemC threads of type `SC_THREAD` must not be terminated. In the example 1.1 a `while(true)` loop is used. In order for a thread to not claim the entire simulation time, it needs to be interrupted. For this, the thread blocks itself on events such as a clock signal or as in the example 1.1 on the read accesses on a FIFO channel. If this FIFO channel is empty, the thread is blocked until a new date is written into the channel.

SystemC methods are mainly used to model hardware components. In system level modeling, where software and hardware components interact, SystemC threads are preferred despite their speed disadvantage.

Channels *channels* are used for communication between modules. Channels can be as primitive as a signal or as complex as a whole bus or a so-called *Network on Chip* (*NoC*). For example, SystemC provides predefined channel types for signals (`sc_signal`), FIFO channels (`sc_fifo`), semaphores (`sc_semaphore`), etc. SystemC also allows you to define your own channel types. For this, the channel must implement the abstract methods of the interface class (derived from `sc_interface`).

The possibility to define own channels allows to support different levels of abstraction in SystemC. This distinguishes SystemC clearly from hardware description languages like VHDL.

Composition A SystemC module can be defined as concurrent processes or as the composition of other SystemC modules. This is illustrated by an example of a simulative verification environment.

Example 1.3 The verification environment is defined by the SystemC module `TestBench`:

```
class TestBench : public sc_module {
  private:
    Generator generator;
    DUV       duv;
    Monitor   monitor;

  public:
    sc_fifo<double> f1;
    sc_fifo<double> f2;

    TestBench(sc_module_name name )
      : sc_module(name),
        generator("generator", 5),
        duv("duv"),
        monitor("monitor"),
        f1(2),
        f2(2) {
      generator.out(f1);
      duv.in(f1);
20          duv.out(f2);
21          monitor.in(f2);
22        }
23    };
```

The verification environment contains the module DUV, which is to be verified, as well as the modules Generator and Monitor. The Generator module is responsible for providing simulation stimuli, while the Monitor module records and evaluates the output of the DUV. The three modules are instantiated in lines 3-5. In addition to these three modules, the verification environment also contains two FIFO channels to transport the data between the modules (lines 8 and 9).

The modules and channels are initialized in the constructor of the verification environment (lines 13-17). Each module is initialized with a name. The generator module also receives the number of stimuli to be generated as constructor parameters. The FIFO channels receive as constructor parameters the size of the channel, i.e. each FIFO channel f1 and f2 can store two dates of type double. Furthermore, the modules and channels are connected in the constructor of the verification environment. For example, the Generator module writes out to channel f1 (line 18) via its port. The DUV module reads from the same channel via the input port in.

1.4.2 Simulation of SystemC Models

The simulation requires the SystemC model to be compiled and executed with a C++ compiler. For the simulation of the verification environment from example 1.3 the modules Generator and Monitor have to be implemented first. The following source code shows the two implementations:

```
class Generator: public sc_module {
  public:
    sc_fifo_out<double> out;
    int i;

  private:
    int max;
    void produce() {
      for (i=1; i<=max; i++) {
        out.write(i);
        cout << "generator: " << i << std::endl;
      }
    }

  public:
    SC_HAS_PROCESS(Generator);

    Generator(sc_module_name name, int m)
      : sc_module(name), max(m) {
      SC_THREAD(produce);
    }
};
```

The Generator module generates a total of max stimuli, which it writes to the output port out. To keep the implementation as simple as possible, a for loop was used.

```
 class Monitor: public sc_module {
  public:
    sc_fifo_in<double> in;
    double data;

  private:
    void consume(void) {
      while(true) {
        data = in.read();
        cout << "monitor: " << data << std::endl;
      }
    }

  public:
    SC_HAS_PROCESS(Monitor);

    Monitor(sc_module_name name) : sc_module(name) {
      SC_THREAD(consume);
    }
};
```

The `Monitor` module simply reads the data from the input port `in` and outputs it to the standard output.

Example 1.4 The main program for the simulation of five stimuli is:

```
1    int sc_main (int argc, char **argv) {
2      TestBench testbench("testbench");
3      sc_start();
4      return 0;
5    }
```

First the verification environment is instantiated in line 2. `sc_start()` starts the simulation. If there are no further events in the simulation, the simulation is terminated.

The output of the simulation could look like this:

```
generator: 1
generator: 2
duv:       1
duv:       2
generator: 3
generator: 4
monitor:   1
monitor:   2
duv:       3
duv:       4
generator: 5
monitor:   3
monitor:   4
duv:       5
monitor:   5
```

It can be seen that the event-driven simulation kernel of the SystemC reference implementation attempts to execute a module for as long as possible. Here, this means that the only executable module, the `Generator` module, is executed twice in a row at the beginning. After that, the FIFO `f1` is full, which is why the simulation kernel selects the next executable module. In this case only the `DUV` module is executable. After this module has also been executed twice, the FIFO `f2` is filled, which blocks the further execution of the `DUV` module.

For this reason, the SystemC simulator switches to another executable module. In this example it is again the `Generator` module. Alternatively, the simulator could have selected the `Monitor` module. Again, the simulator executes the `Generator` module twice. Then only the `Monitor` module is executable, which now consumes the two dates produced first, and so on.

Finally, after all five stimuli have been produced, passed on and consumed, the simulation ends. Note that the order of execution could have been different, since the SystemC standard [1] does not specify a scheduling strategy.

Time Modeling in SystemC SystemC provides primitives for time modeling. The central construct for this is the `wait()` statement. The `wait()` statement can be passed as argument either a *SystemC event* or a time specification. For example, the `wait(10, SC_NS)` statement causes the current process to be blocked and to continue only after the simulation of 10*ns*.

Example 1.5 For example, the SystemC model from example 1.3 can be modelled with time so that the thread `produce` is extended by a `wait` statement:

```
void produce() {
  for (int i=1; i<=max; i++) {
    out.write(i);
    wait(10, SC_NS);
  }
}
```

This means that after each writing of a stimulus, the system waits for 10*ns* until the next stimulus is generated. Furthermore, the thread `pass` can be modeled with a delay time of 5*ns*:

```
void pass(void) {
  while(true) {
    data = in.read();
    wait(5, SC_NS);
    out.write(data);
  }
}
```

In order to record the temporal course of the data, it is a good idea to use a so-called *trace* of the simulation. This can be done, for example, by adding the following instructions to the constructor of the `DUV` module:

```
sc_set_time_resolution(1, SC_NS);
sc_trace_file *tf;
tf = sc_create_vcd_trace_file("trace");
sc_trace(tf, generator.i, "stimuli");
sc_trace(tf, duv.data, "data");
sc_trace(tf, monitor.data, "response");
```

The statement in line 1 determines the temporal granularity with which data is recorded in the trace. In lines 2 and 3, the file for storing the trace is declared and initialized. Lines 4 to 6 determine which data is to be recorded in the trace.

The result of the simulation can be seen in Fig. 6.

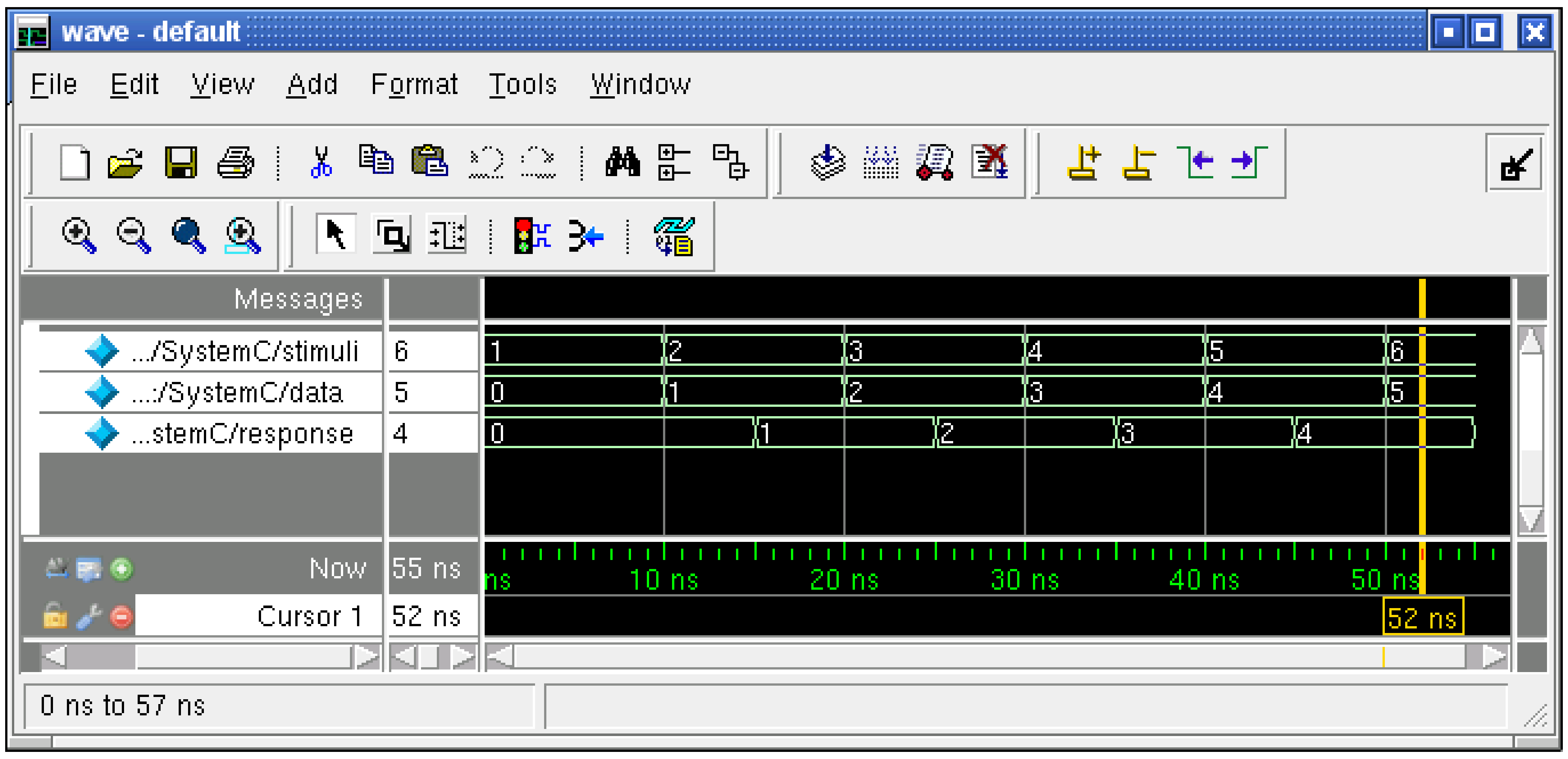


Figure 6: Trace of the SystemC simulation

2 Experiment

Now it's your turn! Use the knowledge you gained in the tutorial of the last chapter to implement a differential equation solver in SystemC that solves the differential equation from Eq. (7) numerically for arbitrary initial conditions.

Login to a lab computer. To perform the trial you will need a *Shell* (command line interpreter) and an editor (e.g.`kate` or `vi`) to edit the source files. Commands, that you have to enter into the *Shell*, are marked in the following by a ☞ symbol. The directory `ueb01` contains all necessary files for the attempt. Change to this working directory.

☞
```
cd ~/ueb01
```

2.1 Functional Simulation

In the following part, you will first implement and simulate your algorithm on a functional level. You can find the necessary files in the directory `versuch/funktional`:

- `EquationSolver.hpp` – Contains the class `EquationSolver` that provides the basic framework for the equation solver. Add your algorithm to this file.
- `EquationSolverTest.hpp` – Contains the class `EquationSolverTest`, that provides a test bench for the equation solver. Generates the necessary input signals for the equation solver.
- `EquationSolverTop.hpp` – Contains the class `EquationSolverTop`, that provides the top module for the application. Instantiates the above 2 classes and connects their ports.
- `sim.cpp` – Instantiates the top module and starts the simulation.
- `datatypes.hpp` – Defines the datatype used in the data path of the differential equation solver (in this case floating and fixed point numbers).

The functionalities of the (relevant) ports of the class `EquationSolver` are explained in table 2.

Name	Typ	Bedeutung
`x0_in`	`sc_in<T>`	X-Value of the first control point(x_0)
`y0_in`	`sc_in<T>`	Initial value $y(x_0)$
`u0_in`	`sc_in<T>`	Initial value$y'(x_0)$
`dx_in`	`sc_in<T>`	Step size (distance of control points)
`a_in`	`sc_in<T>`	Upper bound of the calculation interval
`y_out`	`sc_out<T>`	Result of the calculation

Table 2: Ports of the class `EquationSolver`

Task 2.1 Implement the algorithm described in section 1.2 in SystemC by completing the `EquationSolver` class.

2.1.1 Floating-point numbers

Task 2.2 First the functional simulation with floating point numbers shall be carried out. Adjust the file `datatypes.hpp` accordingly. Then test your implementation by typing ☞ `make`. This compiles and executes the simulation, solving the differential equation in the range $[0, 10[$ and writes the data to the file `sim.data`. Display the generated output with gnuplot by typing ☞ `make plot`.

2.1.2 Fixed-point numbers

SystemC provides the fixed-point datatype `sc_fixed<WL,IWL>`. The template parameter `WL` specifies the total number of bits (e.g. 32) and the template parameter `IWL` specifies the number of bits used for the integer part (e.g. 16) of the number.

Task 2.3 Adjust the file `datatypes.hpp` so that fixed-point numbers with 32-bit word width and 16-bit integer parts are used instead of floating point numbers. Then test your implementation by typing ☞ `make`. Display the generated output with gnuplot by typing ☞ `make plot`. Compare the two generated curves. What do you notice? Why are the two curves different?

16 bits not enough

2.2 Structural Refinement

After the *functional simulation* was successful, the synthesis is started, i.e. allocation, binding and scheduling with subsequent implementation as SystemC model on register transfer level. The necessary files can be found in the directory `versuch/strukturell`:

- `Resources.hpp` – Contains the resources that can be used for implementation (adders, subtractors, multipliers, comparators, registers and multiplexers).
- `EquationSolver.hpp` – Contains the class `EquationSolver` that provides the basic framework for the equation solver. Add the required components and signals to this file and connect them accordingly.
- `EquationSolverTest.hpp` – Contains the class `EquationSolverTest`, which provides a test bench for the equation solver (see functional simulation).
- `EquationSolverTop.hpp` – Contains the `EquationSolverTop` class, which is the top module of the application (inherited from the functional implementation).
- `sim.cpp` – Instantiates the top module and starts the simulation.
- `datatypes.hpp` – Defines the data type used in the data path of the differential equation solver.

2.2.1 Synthesis

Task 2.4 The data flow graph corresponding to Eq. 7 is shown in Figure 7. Determine a schedule with minimum latency L_{min} using the ASAP algorithm. The binding information can be taken from the resource graph in Figure 8.

The execution times of the resource types (obtained from logic synthesis) can be found in table 8. These take into account, among other things, the delay time of the upstream and downstream multiplexers, which is approx. $1ns$. The targeted clock period is $T = 5ns$. The following formula can be used for a resource type r_k to be able to work with integer, dimensionless values d as usual during scheduling: $d(r_k) = \lceil \frac{z(r_k)}{T} \rceil$.

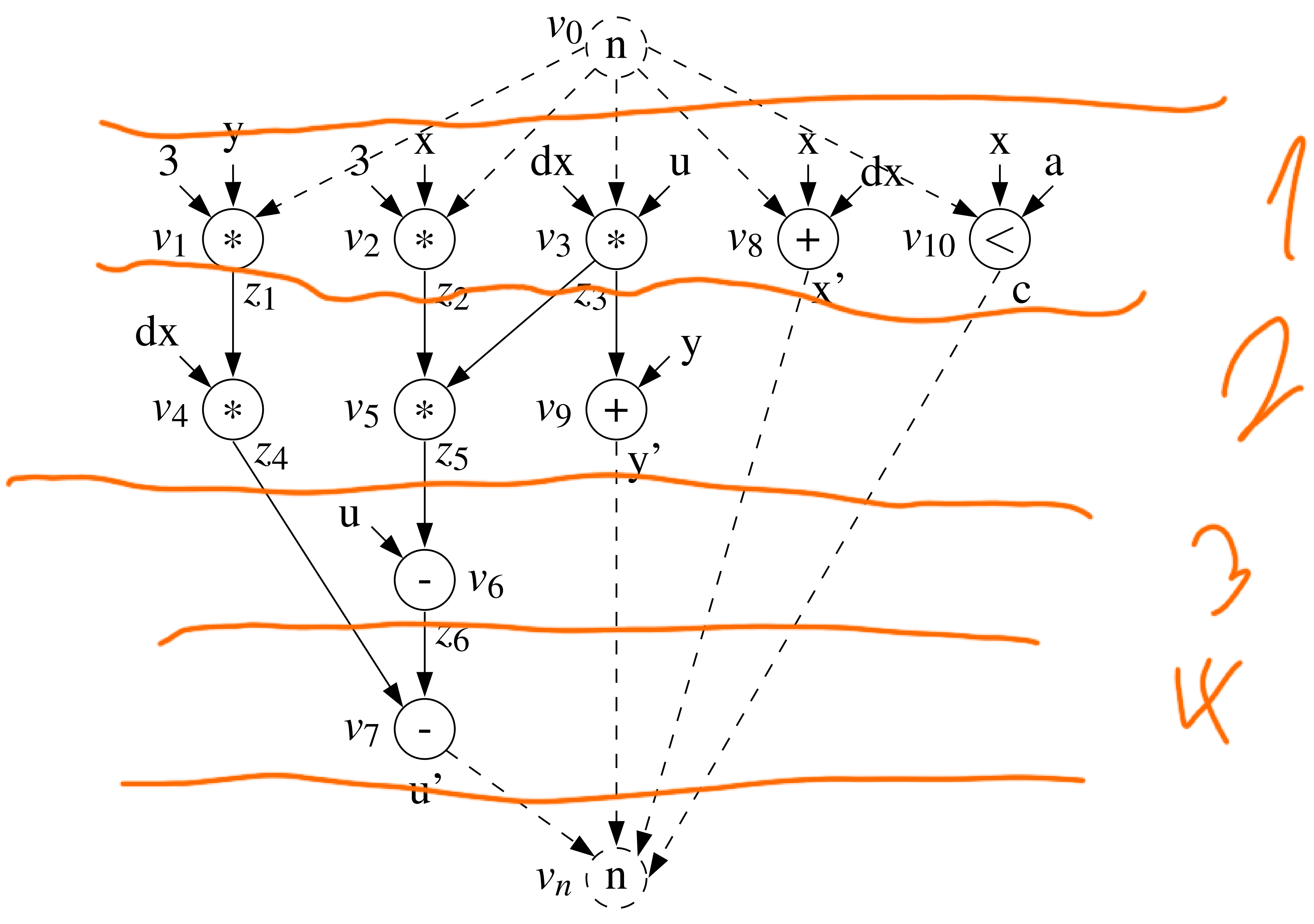
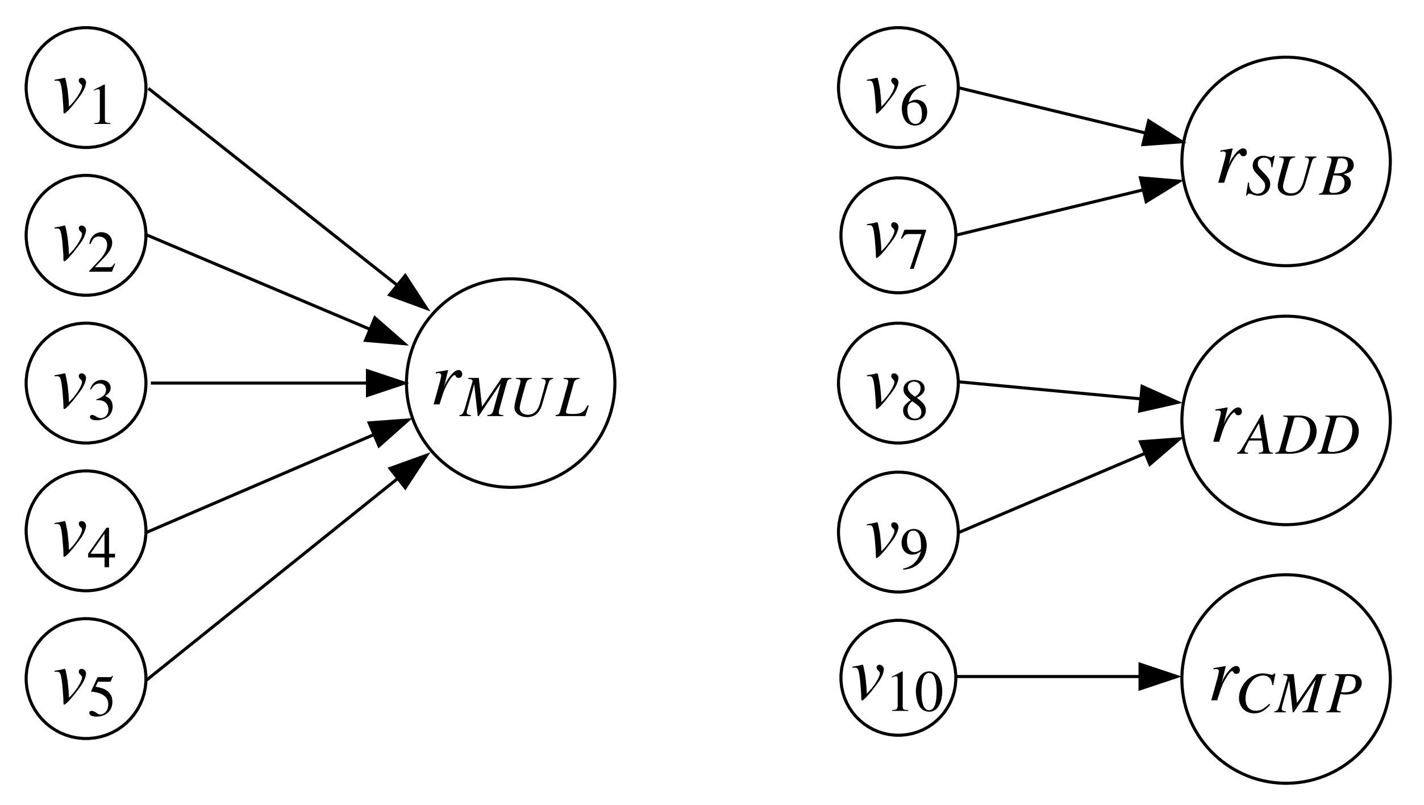


Figure 7: Data flow graph



r_k	$z(r_k)$	$d(r_k)$
r_{MUL}	$10ns + 2 * 1ns = 12ns$	3
r_{SUB}	$2ns + 2 * 1ns = 4ns$	1
r_{ADD}	$2ns + 2 * 1ns = 4ns$	1
r_{CMP}	$1.5ns$	1

Figure 8: Resource graph and execution times

2.2.2 Allocation of Registers

The input signals `x0_in`, `y0_in`, `u0_in`, `dx_in` and `a_in` are stored in registers as soon as they are valid. The respective registers are referred to as r_x, r_y, r_u, r_{dx} and r_a. In addition, the calculated intermediate results z_1 - z_6 of the individual operations must be stored in registers. Those registers are referred to as r_1 - r_6.

Task 2.5 The contents of a register may not be changed until all operations that use the variable currently stored in this register have been completed. Check your schedule to see if there are any operations that violate this property. If so, you might be able to move them backwards without increasing the latency.

2.2.3 Implementation

To obtain a SystemC model at register transfer level, you need to perform the following subtasks. You only need to edit `EquationSolver.hpp`, the remaining files are identical to the previous task.

Task 2.6 Instantiate the required components and signals according to allocation, binding, and register allocation. Note: In order to write the initial values for the variables x, y and u, which are made available via the ports `x0_in`, `y0_in`, and `u0_in`, respectively, into the designated registers, additional multiplexers are required before the inputs of the registers r_x, r_y and r_u.

Task 2.7 Connect the ports of the instantiated components to the corresponding signals.

Task 2.8 The FSM shall be described *functionally* in this experiment. Therefore the function `main` must be modified so that the approximation algorithm is no longer executed functionally in a loop, but the structural implementation of the algorithm is used instead. The following tasks must be performed:

- Set the multiplexer control signals to select the input signals of the functional units or registers.
- Activating or deactivating the clock enable signals of the registers in order to store a new value at the next clock edge or to preserve the stored value.

Note: The FSM is modeled as `SC_CTHREAD` which is sensitive to the positive clock edge. To wait for a clock cycle — which corresponds to a state transition in the FSM — the SystemC function `wait()` must be called.

To monitor signal values, SystemC provides a tracing infrastructure (see chapter 1.4.2). A trace file can be opened using the functions `sc_create_vcd_trace_file` or closed using `sc_close_vcd_trace_file`. Using the function `sc_trace`, the values of a signal can be written to an open trace file during simulation.

Task 2.9 Test your implementation by typing ☞ `make`. This compiles and executes the simulation, solving the differential equation in the range $[0, 10[$ and writes the data to the file `sim.data`. Display the generated output with gnuplot by typing ☞ `make plot`. To verify the correctness of the implementation, you can enter ☞ `make check`. This command compares the output data with the reference data of the functional implementation. The trace file also created by the simulation `sim.trace` can be viewed with *ModelSim* by entering the command ☞ `make trace`.

Task 2.10 Optional: Adapt the file `datatypes.hpp` so that fixed-point numbers with 32-bit word width and 16-bit integer parts are used instead of floating point numbers. Then test your implementation by entering ☞ `make`. Display the generated output of gnuplot graphically with ☞ `make plot`. Compare the generated fixed-point number curve with the fixed-point number curve from the previous (functional) implementation. What do you notice? Why do the two curves differ?

A References

References

[1] Mike Baird, editor. *IEEE Standard 1666-2005 SystemC Language Reference Manual*. IEEE Standards Association, New Jersey, USA, 2005.

[2] Bronstein and Semendjajew. *Taschenbuch der Mathematik (***Der** *Bronstein)*. Verlag Nauka Moskau, 1991.

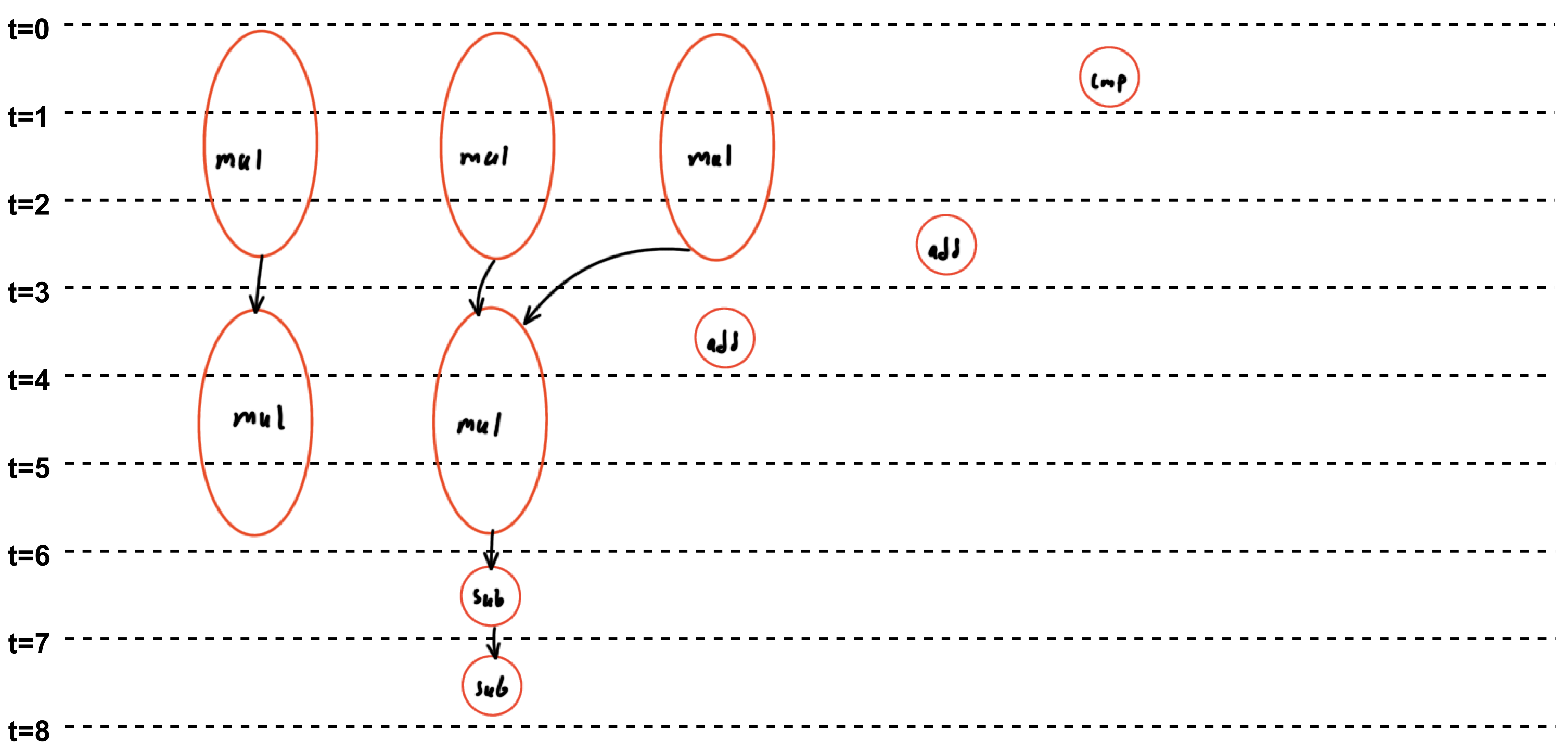
[3] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[4] H.R. Schwarz. *Numerische Mathematik*. B.G. Teubner Stuttgart, 1997.

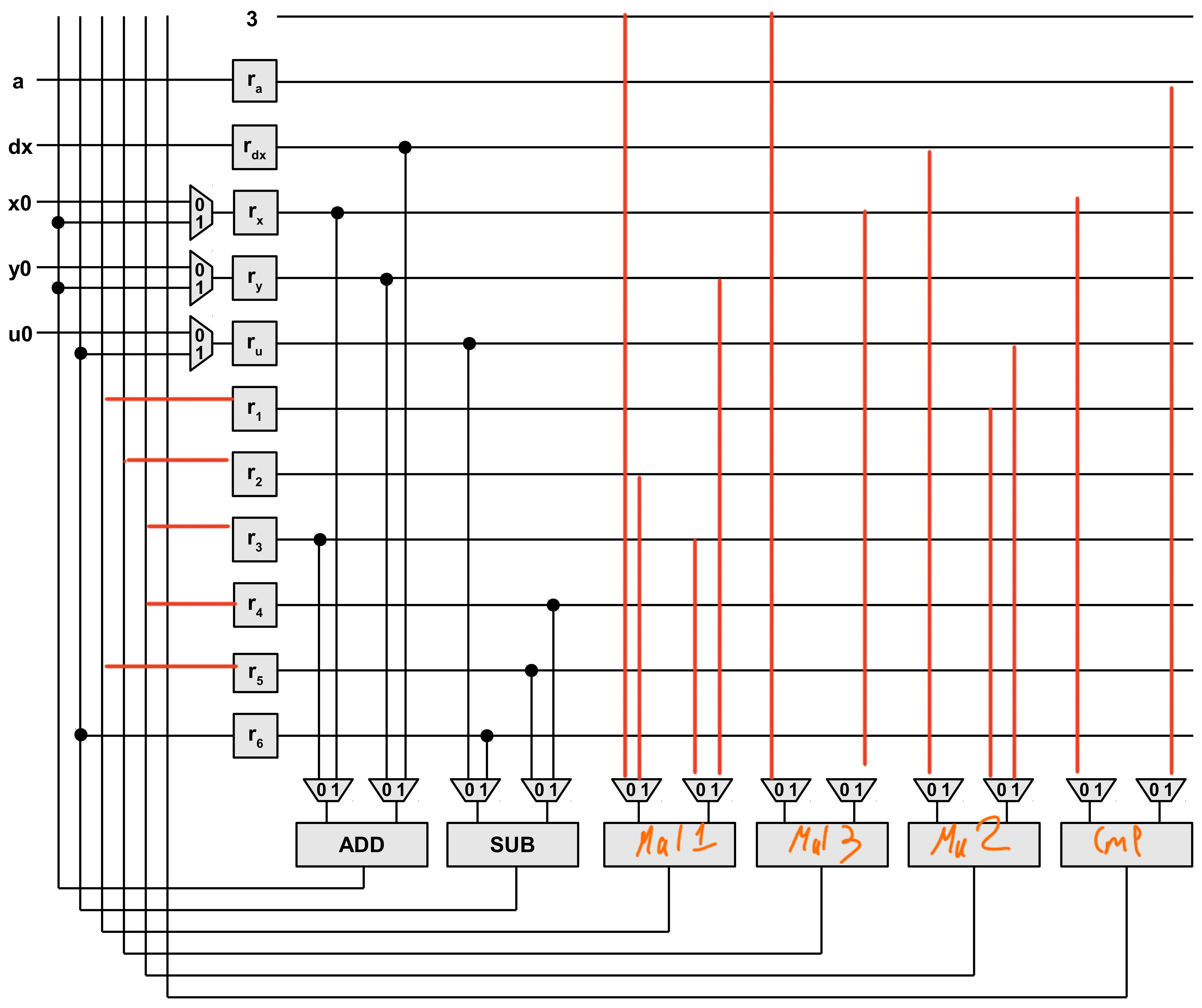
[5] Jürgen Teich and Christian Haubelt. *Digitale Hardware/Software-Systeme: Synthese und Optimierung, 2. Auflage*. eXamen.press. Springer, 2007.

[6] Stephen Wolfram. *Das Mathematica Buch*. Addison-Wesley Verlag, 1997.

Arbeitsblatt zur 1. Erweiterten Übung zur Vorlesung Eingebettete Systeme



t=0
t=1
t=2
t=3
t=4
t=5
t=6
t=7
t=8
mul
mul
mul
mul
mul
cmp
add
add
sub
sub



3
a
dx
x0
y0
u0
r_a
r_{dx}
r_x
r_y
r_u
r_1
r_2
r_3
r_4
r_5
r_6
0 1
ADD
SUB
Mul 1
Mul 3
Mu 2
Cmp