Bus Centric Synchronous Message Exchange for Hardware Designs

Conference Paper · August 2015

CITATIONS
4

READS
166

2 authors:

Brian Vinter
University of Copenhagen
113 PUBLICATIONS **580** CITATIONS
SEE PROFILE

Kenneth Skovhede
University of Copenhagen
22 PUBLICATIONS **51** CITATIONS
SEE PROFILE

Communicating Process Architectures 2015
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2015

Bus Centric Synchronous Message Exchange for Hardware Designs

Brian VINTER [a] and Kenneth SKOVHEDE [a],
[a] *University of Copenhagen, Niels Bohr Institute*

Abstract. In this work we present a new design and implementation of the Synchronous Message Exchange model. The new version uses explicit busses, which may include multiple fields, and where a components may use a bus for both reading and writing, whereas the original version allowed only reading from or writing to a bus, which triggered a need for some busses to exist in two versions for different directions. In addition to the new and improved bus-model, the new SME version also produces traces that may be used for validating a later VHDL implementation of the designed component, and can produce a graphical representation of a design to help with debugging.

Keywords. Scientific Byte Code, FPGA, PyCSP, Synchronous Messaging

Introduction

In [1] we introduced the synchronous message exchange, SME, model. The fundamental idea behind SME is to describe and execute synchronous models with an explicit message-passing paradigm, but still leverage the advantages of the synchronous nature of the model. Since then we have used SME for several investigations and prototypes and have ended up with a set of features that were missing and a design choice that, while correct, made the SME library tedious to work with in many cases.

In this work we introduce a completely new implementation of SME, the new implementation is marginally slower that the original SME library, but the new design makes using the library much easier, and the new features makes the use of the SME library far more extensive.

SME was originally derived from our initial work on modeling a new specialized vector processor, using PyCSP and the positive results we had using CSP for the initial hardware design [2].

Our work is exclusively driven by the need to provide more processing power, often with power consumption as an additional boundary condition. Where energy use is not an issue the use of General Purpose Graphical Processing Units, GPGPUs, is the more common technological choice, while for more energy sensitive applications Field Programmable Gate Arrays, FPGAs, are the common choice. Unfortunately designing hardware is a slow process, and we try to address the productivity issues associated with HDL languages by using SME.

1. SME

The overall motivation for SME came from the in-depth project on PyCSP[3] for hardware design was that CSP, and PyCSP, are not designed for globally synchronous execution. Not exactly new knowledge since the absence of global coordination is a defining characteristic

of CSP, but the consequences of forcing a globally synchronous model onto CSP are an explosion of channels and processes. This increases the complexity and reduces performance of the simulation. In addition, the external choice operation, which in many ways is the most powerful feature in CSP, is never used in a synchronous simulation environment. In other words, the CSP primitives are too primitive for hardware design purposes while the advanced features do not fill a need.

While point-to-point rendezvous message passing was not a useful technique for hardware design, other aspects of the CSP-like design approach were indeed convenient, shared-nothing, traces and equivalences were used extensively.

We decided to test these conclusions by implementing a new message-passing framework based on the above observations:

- Globally synchronous in nature
- Broadcasting channels
- Hidden clock
- Shared nothing
- Implicit latches

This new framework is in itself very simple and has a simple, although process and communication intensive, equivalence in CSP. Thus we are not proposing an entirely new and different message-passing approach, rather a special design pattern for CSP programs, which does not support external choice and communicates with the above listed features. However, this new model, Synchronous Message Exchange, SME, is easy to use for applications such as hardware design, and may be implemented to run much faster than PyCSP.

1.1. Shortcomings

The biggest shortcoming in the original SME model is the strict mapping between a port and a bus. This means that a connection between two components is often required to be made up from a set of busses, rather than one bus as a hardware designer would often consider it.

An example would be the simple, textbook style, bus between a CPU and a memory unit. Such a connection consists of a databus, an adressbus, and a rw-flag. The SME implementation of a processor-memory connection is shown in listing 1

Listing 1: A processor-memory system using the original SME system

```
1 from Sync import sync_process
2
3 class cpu(sync_process):
4    def __init__(self, name, inputs, outputs):
5       self.connect(name, inputs, outputs)
6       self.state = 3
7
8    def run(self):
9       #The implementation of the pseudo processor
10
11 class memory(sync_process):
12    def __init__(self, name, inputs, outputs):
13       self.connect(name, inputs, outputs)
14       self.memory = [0] * 1024
15       self.has_result = False
16
17    def run(self):
18       #The implementation of the pseudo memory
19
20 from Sync import sync_component
```

```
21
22 system = sync_component('CPUMEM', {},{}, 12)
23 system.add(cpu('cpu', {'databus_in':'data_from_memory'}, {'databus_out
     ':'data_to_memory', 'rw': 'memory_rw', 'addressbus': 'data_address'
     }))
24 system.add(memory('memory', {'databus_in':'data_to_memory', 'rw': '
     memory_rw', 'addressbus': 'data_address'}, {'databus_out':'
     data_from_memory'}))
25 system.run()
```

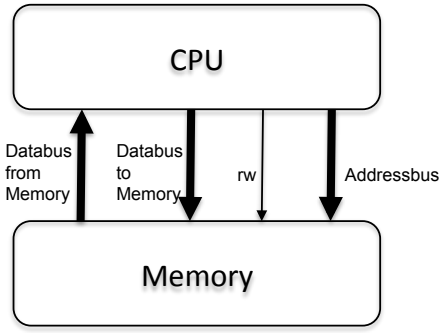


Figure 1. A simple processor-memory setup, using the original SME model.

The original SME approach requires us to model the databus as two different busses as shown in figure 1, one going to the memory and one coming from the memory. This is of no consequence with respect to a final implementation in hardware, but is still inconvenient for the programmer. Two way busses are mostly relevant for modeling off-chip interfaces, but another use, for internal chip modeling is the use of multiple potential writers, where only one may selected for writing within a single clockcycle.

Another shortcoming was the lack of multiple clock speeds in one simulation, it would also be convenient to clock components at different speeds to better model real-world systems.

Finally we have realized that while the idea of producing VHDL code directly from the model is alluring, in reality translating a model that is already implemented and tested in SME is quite simple and require little work. On the other hand writing the VHDL code that is used to verify the VHDL implementation is a time consuming and complex task since VHDL is actually poorly suited for writing test codes. Thus it would be very efficient for SME to support generating test vectors, rather than target code.

2. New SME

2.1. Busses

The biggest change from the original SME model is that busses now are active objects, a bus has a name, and a set of fields. This approach allows us to model the processor-memory bus from 1.1 as one processor memory bus object with databus, addressbus, and rw-flag as the relevant fields. Listing 2 shows the same structure as 1, but with the use of active bus objects.

Listing 2: A processor-memory system using the new SME system

```
1 from SME import Bus, Network, External
2
3 class cpu(External):
```

```
4     def setup(self, args):
5         self.bus = args
6         self.state = 4
7         self.bus['rwp'] = 3
8         self.bus['addressbus'] = 0
9
10    def run(self):
11        #The implementation of the pseudo processor
12
13 class memory(External):
14    def setup(self, args):
15        self.bus = args
16        self.memory = [0] * 1024
17
18    def run(self):
19        #The implementation of the pseudo memory
20
21 class system(Network):
22   def wire(self, args):
23     self.bus = Bus('PM', ['addressbus', 'databus', 'rwp'])
24     self.memory = memory('Mem', self.bus)
25     self.cpu = cpu('CPU', self.bus)
26
27 test = system('Membus')
28 test.clock(13)
```

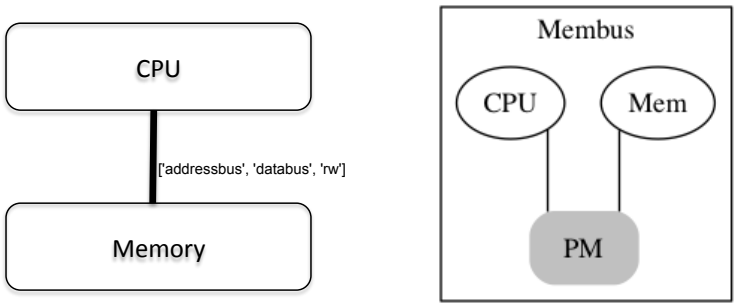


Figure 2. A simple processor-memory setup, sketch and rendered SME model.

A bus retains the properties of the previous SME implementation in that we check that two or more components do not write to a bus in the same cycle, and, symmetrical, that no component reads from a bus to which nothing has been written. The much simpler design is also visible from figure 2.

2.2. Multiclock

The new SME model also supports clock-multipliers. A network clocked at one rate can mark components inside the network as being clocked at a integer multiplier of its own clock. This means that there is no need for a global clock coordination, since any components is clocked relatively to its parent network. The limitations are that relative frequencies can only be expressed as integers and that inner componenets can only run faster than their context processor.

2.3. Tracing

A consequence of making busses active components is that logging all data that are written to a bus can be logged for each clock-cycle. These logs can then be exported as comma-

separated-values (CSV) which may be imported into the VHDL tool and used for validating the VHDL implementation, thus eliminating the need for writing test-code in VHDL. This very simple tool may replace very large amounts of VHDL code for test purposes, and thus contributes to significant productivity improvements.

2.4. Typing

Another consequence of declaring busses explicitly is that we can easily add type-checking. We have chosen to add types to bus-fields, either at declaration time or at first write to the bus-field. Thus an integer bus can be declared in one of two ways:

```
1 a = Bus('a', ['data'], dtype=int)
```

or

```
1 a = Bus('a', ['data'])
2 a['data'] = 42
```

2.5. Visualization

Finally we have added simple diagram tool to SME, this means that a SME model can be drawn graphically, which sometimes helps debugging. Busses are shown as components and a components connection to a bus is shown as a connection between the two. The visualization of the processor-memory bus is shown in fig 1, right side.

Since an SME network may in fact be represented as a graph we utilize this fact to generate visualizations with the Graphviz toolkit[1]. The visualization is generated but the SME script calling a method in a network named plot. The visualization produces the Graphviz script for the network, by adding processes, and traversing any networks recursively while keeping track of the busses that are hosted within the networks. Once the network is traced to the end the busses are added this is a requirement from Graphviz for the busses to be places at the correct recursion level. While the visualization will typically be called on the outer most network this is not necessary, and a programmer may choose to visualize any sub-level of the design.

3. Implementation

The new SME model is implemented through four classes, Network, Function, Bus, and External.

A Bus is the most complex of the classes. It is based on the standard Python dictionary class, but every entry exists in a read and a write copy. If a write operation targets a field that has another value than None, an exception is thrown to mark that a double-write happened within the same clock-cycle. The exception announces both the name of the bus and the field that got the conflicting write. Writes are also type-checked, if no type exists on the filed then is is assigned on first write. Likewise, all reads are tested for None values, and if a read would return None an exception is raised on the violation of the rule that a component tries to read a value that was not written. From a programmers perspective a Bus is never called directly, but fields within a bus may be accessed like any other dictionary.

Functions are end-functionalities, these are similar to processes in CSP and differ little from the old SME version, however they are now not created with variables that repre-

[1] http://www.graphviz.org/

sents the interfaces to other components, but rather busses may be passed to the functions as parameters. Externals are identical to Functions, truly identical, i.e. the External is just another name for Function, they exist only for programmers convenience, i.e. if an object is a Function the programmer should only use simple types that can be translated into VHDL, while Externals are meant as support functions that should not be translated into VHDL, thus lists, dictionaries and print-statements makes sense in Externals while they do not in Functions. Functions and Externals must implement two methods; setup and run. Setup is called exactly once, on creation time, and are passed the parameters the constructor includes. The run method is called exactly once per clock-cycle, and must be non-blocking, just as in the original SME model.

Networks are meant to structure the SME model. Network has only one method called wire, which is called only once, at creation time. As with Functions and Externals, parameters that are provided to the constructor is passed along to the wire method.

4. Examples

SME is a very simple piece of software and rather than describing the use in tedious detail we prefer to show some examples of the use of SME. Some of these examples are purely for demonstration purposes while others are, scaled down, examples of what we use SME for internally.

4.1. AllOps

AllOps is the SME version of hello world. The network is very simple, two support functions interface with four simple functions. One support function produces numbers, and post these on two busses and the four functions then operates on these two numbers, producing their sum, difference, product and integer-division. The last support function reads the simple arithmetic results and prints them. Listing 3 shows the code, though only one of the arithmetic functions, the others are identical except for the operator. Figure 3 shows two graphics representations of the network, including the version the SME can generate by itself.

Listing 3: AllOps demo application

```
1 from SME import Bus, Network, Function, External
2
3 class producer(External):
4    def setup(self, args):
5       self.dbus, self.keys = args
6       for key in self.keys:
7          self.dbus[key] =  1
8
9    def run(self):
10      import random
11      for key in self.keys:
12         self.dbus[key] =  random.randint(1, 100)
13
14 class add(Function):
15   def setup(self, args):
16      self.input, self.output = args
17      self.output['add'] = 0
18
19   def run(self):
20      self.output['add'] = self.input['data1'] + self.input['data2']
21
```

```
22 #sub, mul and div are similar
23
24 class printer(External):
25    def setup(self, args):
26       self.dbus, self.keys = args
27       for key in self.keys:
28          print key,'\t',
29       print ''
30
31    def run(self):
32       for key in self.keys:
33          print self.dbus[key],'\t',
34       print ''
35
36 class all_ops(Network):
37    def wire(self, args):
38       self.input = Bus('Input',['data1','data2'])
39       self.output = Bus('Output', ['add','sub', 'mul', 'div'])
40
41       self.producer = producer('Producer', [self.input, self.input.
            fields()])
42       self.add = add('Add', [self.input, self.output])
43       self.sub = sub('Sub', [self.input, self.output])
44       self.mul = mul('Mul', [self.input, self.output])
45       self.div = div('Div', [self.input, self.output])
46       self.printer = printer('Printer', [self.output, self.output.
            fields()])
47
48 chip = all_ops('AllOps')
49 chip.clock(10)
```

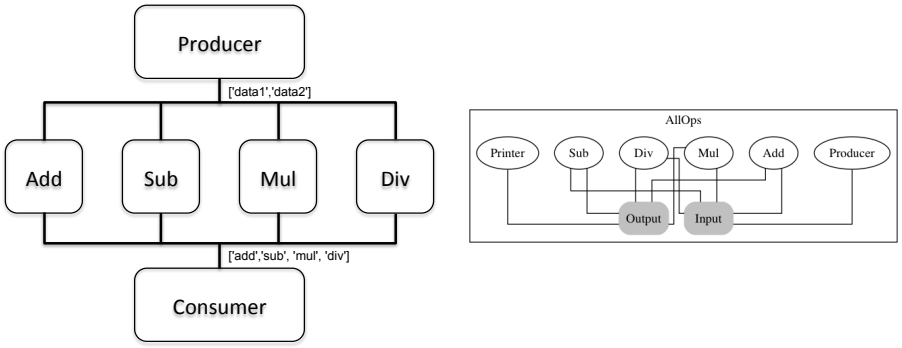


Figure 3. A simple newtork four operations, sketch and rendered SME model.

4.2. Line-detector

The line detector is a scaled down solution from a real project. A line detector in this case is an xray camera[4] with width 1 and n-pixels high. The actual line-detector is more than one pixel wide, but interleaves the pixels over time to produce a more reliable image. In this example we have eliminated the interleaving and just focused on triggering the camera and the parallel to serial readout. The code it too big to include put can be found here [5]. However the structure can seen on listing 4, the output of the simulation is shown in figure 5, which is an actual xray scan of a wooden plank, which is also used as input for the simulation. The line detector is scalable, and the overall wrapper object, System, receives parameteres that specify, how many pixels long the camera is, how much buffer space the camera has for

images, the clock-rate at which the camera is triggered for new readings. The final parameter, simdata, is purely simulation and used to simulate the xray signal.

Listing 4: Structure of the xray Line Detector

```
1 class System(Network):
2     def wire(self, args):
3         self.pixels, self.buffer, self.rate, self.simdata = args
4         self.controlbus = Bus('Control', ['readout', 'selector', 'data'
              ])
5         self.databus = Bus('DataBus', 'data')
6
7         self.controller = Controller('Controller', [self.controlbus,
              self.rate, self.pixels])
8         self.reader = Reader('Reader', [self.databus, self.controlbus,
              self.buffer])
9         self.elements = Array('PixelArray', [self.controlbus, self.
              databus, self.pixels, self.simdata]
```

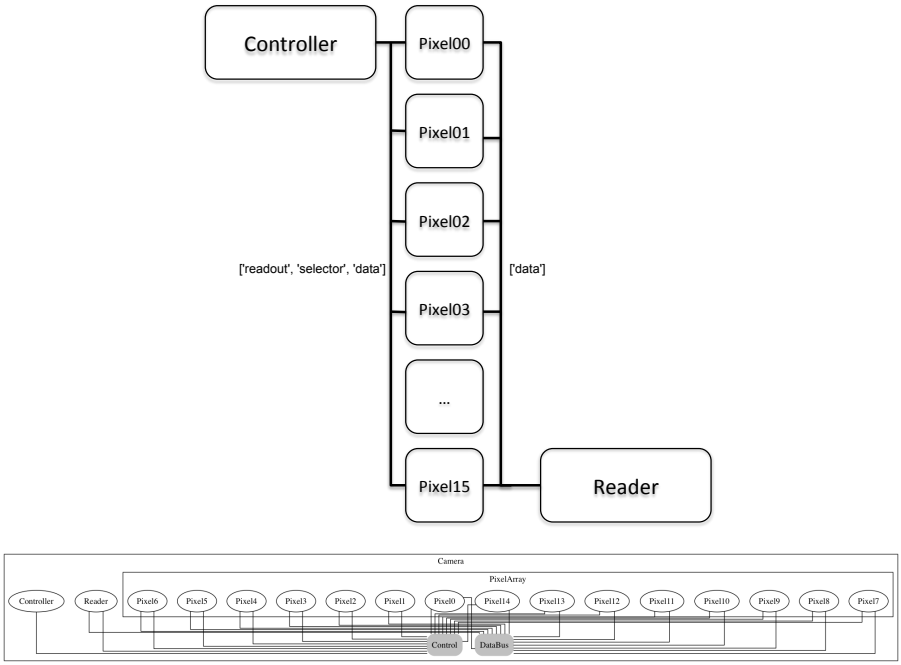


Figure 4. A xray linedetector, sketch and rendered SME model.

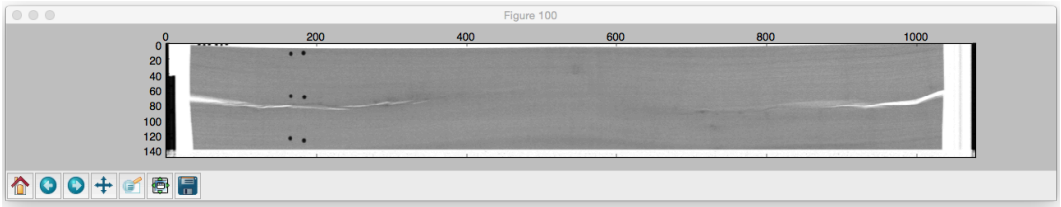


Figure 5. Result of the xray linedetector simulation.

4.3. Vector processor

The vector processor is another scaled down version of an actual project. The key is that a vector processor has exactly one control unit and then a large number of cores that execute the

program in strict SIMD style[6]. The memory is modeled as an External, since memory is an external component in our design, which will be interfaced by the developed processor. Since the memory must necessarily have two memory ports per processor-core the SME model uses a Python loop to generate the busses and cores, however similar techniques are available in VHDL. Below listing 5 show the SME code that describes the network, but not the functions, these can be found on [5]. The Vector procesor is an prime example of the shortcomings of the automatic network rendering, as figure 6 clearly shows the hand-drawn diagram is far more readable. The outer-most wrapper object takes as parameters; how many pipelines should be instantiated, how many registers each pipeline has, the size of the simulated memory, and a static test-program to be run by the vector processor in the simulation.

Listing 5: Structure of the Vector Processor

```
1  class Vector(Network):
2     def wire(self, args):
3        pipelines, regsize, memsize, program = args
4        from_memory_entries = ['from_memory%d'%i for i in xrange(
            pipelines)]
5        to_memory_entries = ['to_memory%d'%i for i in xrange(pipelines)]
6        print from_memory_entries+to_memory_entries
7        self.controlbus = Bus('controlbus', ['inst', 'src1', 'src2', '
            dest'])
8        self.databus = Bus('databus', from_memory_entries+
            to_memory_entries)
9        self.addressbus = Bus('addressbus', ['rd_addr', 'wr_addr', 'rd',
             'wr'])
10
11       self.controller = Controller('Controller', (self.controlbus,
            self.addressbus, program))
12       for i in xrange(pipelines):
13          name = 'pipeline%d'%i
14          self.__dict__[name] = Pipeline(name, (self.controlbus, self.
               databus, regsize, from_memory_entries[i],
               to_memory_entries[i]))
15       self.memory = Memory('Memory', (self.databus, self.addressbus,
            pipelines, memsize))
```

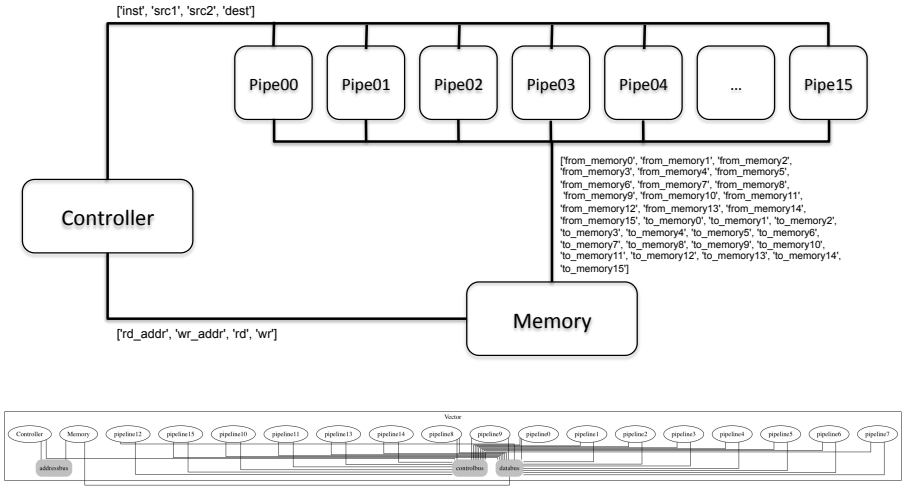


Figure 6. A small vector-processor.

4.4. High Frequency Trading Chip

The final example is a naıve version of a chip for automatic high frequency trading, diagrammed in figure 7. A simple strategy for trading stocks is to compare the average price of stock, either against itself but using two different window sizes[7], i.e. a short and a long average, or against an index of similar stocks. The actual trading strategies can be quite different, buy when the stock over-performs and sell when it drops under, or vice versa. The hardware implementation is essential since the window sizes can be down to sub milliseconds, thus no human intervention is possible. The SME implementation supports both matching a stock against itself and against an index set. The implementation is very simple, however this is required as the target performance is less than 100 ns from input to output.

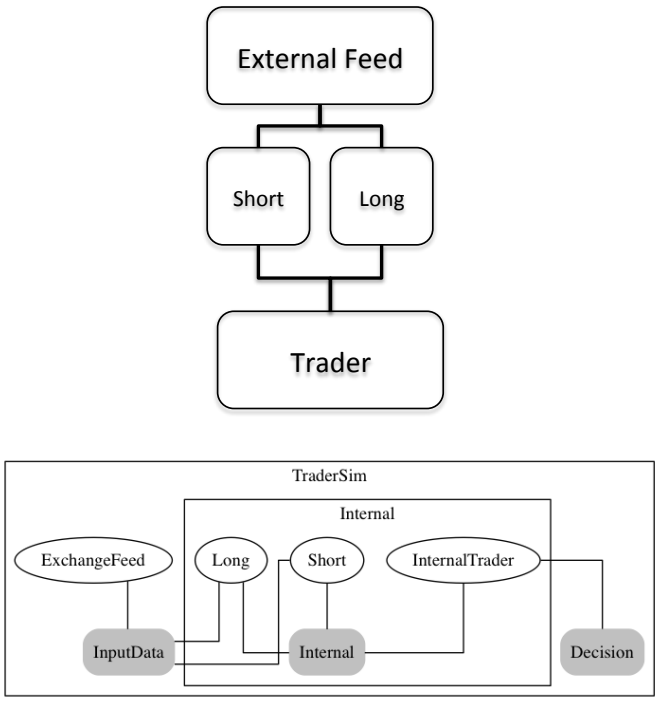


Figure 7. A High Frequency Trading Chip

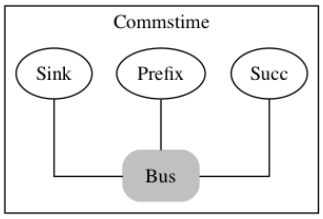
4.5. Benchmark

Performance is not essential to SME, but compared to the alternative VHDL tools SME is much faster! However, since we are replacing the original SME version with this new code it is of interest to know how the relative performance is. To this end we have implemented the traditional commstime network, as we did for the original SME version. Below table 1 shows the relative performance to the original SME model, both with and without tracing for VHDL purposes. The experiment is run on an iMac with 2.7GHz Intel Core i7 and 16GB ram.

SME	SMEv2	SMEv2 no log
1.616	1.967	1.704

Table 1. Full iteration of commstime using the original SME, the new SME and the new SME with no logging, times are in us.

The new SME version is clearly slower, 21%, however if we turn off logging of all messages, which is typically done during development and debugging, the added overhead




```
class prefix(Function):
  def setup(self, args):
    self.bus, self.input, self.output, self.value = args
    self.bus[self.output] = self.value

  def run(self):
     self.bus[self.output] = self.bus[self.input]

class succ(Function):
  def setup(self, args):
    self.bus, self.input, self.output = args
    self.bus[self.output] = 0

  def run(self):
     self.bus[self.output] = self.bus[self.input] + 1


class sink(Function):
  def setup(self, args):
    self.bus, self.input = args

  def run(self):
    pass

class network(Network):
 def wire(self, args):
  self.bus = Bus('Bus',['a','c'])

   self.prefix = prefix('Prefix', [self.bus, 'c', 'a', 0])
   self.succ = succ('Succ',[self.bus, 'a', 'c'])
   self.sink = sink('Sink', [self.bus, 'a'])
```

Figure 8. Commstime, layout and code

is down to 5%, which is negliable. This means that the SME model is fully usable for real size systems. As an example the linedetector from 4.2 is run with 151 detector pixels, an update frequency of 200 cycles and a simualation time of 216.000 clockcycles, on the same hardware in 53 seconds.

Conclusions

In the year that has passed since we published the original SME work we have used the approach, and the library intensely ourselves for several projects. The experiences with SME triggered us to redesign the whole system. The new SME library makes use of explicitly instantiated busses, and each bus can include several named items. The new approach eliminates the need for multiple busses between the same components, and this simplifies the description of a new synchronous system. The explicit use of busses also allows us to log all values that are written to a bus, and thus produce a cycle-accurate trace of the whole system. This trace can be saved in CSV format, and thus be fed into a VHDL description of the same system, which saves us significant time at that stage of development since writing test-codes in VHDL is not efficient and often takes much more time than the VHDL description of the system itself.

References

[1] Brian Vinter and Kenneth Skovhede. Synchronous message exchange for hardware designs. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan Bækgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2012*, pages 201–212, aug 2014.

[2] Martin Rehr, Kenneth Skovhede, and Brian Vinter. BPU Simulator. In Peter H. Welch, Frederick R. M. Barnes, Jan F. Broenink, Kevin Chalmers, Jan Bkgaard Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2013*, pages 233–248, Nov 2013.

[3] Brian Vinter, John Markus Bjørndalen, and Rune Møllegaard Friborg. PyCSP Revisited. In Peter H. Welch, Herman Roebbers, Jan F. Broenink, Frederick R. M. Barnes, Carl G. Ritson, Adam T. Sampson, Gardiner S. Stiles, and Brian Vinter, editors, *Communicating Process Architectures 2009*, pages 263–276, Nov 2009.

[4] T Lohse, P Krüger, H Heuer, M Oppermann, H Torlee, and N Meyendorf. Counting x-ray line detector with monolithically integrated readout circuits. In *SPIE Microtechnologies*, pages 87632Q–87632Q. International Society for Optics and Photonics, 2013.

[5] Brian Vinter. Synchronous message exchange. `http://www.erda.dk/public/archives/YXJjaGl2ZS1vWDRBRjI=/published-archive.html`.

[6] Wilfried Oed and Otto Lange. On the effective bandwidth of interleaved memories in vector processor systems. *Computers, IEEE Transactions on*, 100(10):949–957, 1985.

[7] Mark Bagnoli and Naveen Khanna. Insider trading in financial signaling models. *The Journal of Finance*, 47(5):1905–1934, 1992.