Synchronous Message Exchange for Hardware Designs

Conference Paper · August 2014

CITATIONS
5

READS
99

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 2014
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2014

Synchronous Message Exchange for Hardware Designs

Brian VINTER and Kenneth SKOVHEDE
Niels Bohr Institute, University of Copenhagen, Denmark

Abstract. In our 2013 paper, we introduced the idea of modeling hardware with PyCSP. Encouraged by our initial success we started a masters project where two students continued our work towards a fully detailed processor built in PyCSP. The two students succeeded, but also identified a number of reasons why PyCSP is not well suited for modeling hardware. Their conclusion was that since the hardware is synchronous, communication is frequently based on broadcast and external choice is never used. This means that PyCSP does not provide the mechanisms that are needed, and the strength of PyCSP is never utilized. In this work we introduce a new messaging framework, Synchronous Message Exchange, SME, which greatly simplifies hardware modeling, and is also easily used for other strictly synchronous applications, such as a subset of computer games. We describe the SME framework, and show how it has a rather simple equivalence in CSP so that the properties that are associated with CSP based applications are maintained, except rendezvous message exchange.

Keywords. scientific byte code, FPGA, PyCSP, synchronous messaging

Introduction

A year ago we described our initial work on modeling a new specialized vector processor, using PyCSP [1] and the positive results we had using CSP for the initial hardware design [2]. Since then, two students have managed to refine the original high abstraction level simulation to a level where most of the components in the vector processor are synthesizable by hardware description languages. However, the overall conclusion from this work is that PyCSP does not meet the needs for hardware design nearly as closely as we concluded in the original work.

The introduction of enforced global synchrony, which was required, resulted in an explosion of channels for simulating clock-propagation and controlling progress, and in the introduction of latch processes to buffer channels and emulate the physical latches found in real-world implementations.

The students did find positive aspects of the CSP approach too. The consistent use of isolated processes that, through shared-nothing[3], were easily replicated to multiple instances was found to be a powerful approach. Such processes are more easily tested in isolation, similar to unit-testing in conventional software development. Equivalent implementations, i.e. a complex code implementation of one process and the same functionality implemented with multiple processes, were interchangeable and thus easily compared.

To model hardware we need a global agreement on progressed time, i.e. what may be called global synchrony. While the rendezvous semantics in CSP do indeed enforce strict synchrony between the communicating processes, no such semantic exist at the global level in CSP. Thus when we use the term synchronous for the remainder of this paper we refer to this kind of global synchrony, readers that are familiar with BSP[4], may think of our synchrony as a global super-step in BSP.

In the presented programming framework we focus on designing hardware with time constraint, namely explicitly clocked hardware. Since our target users are software developers, we would like to distance the user from the hardware details so they can focus on writing the program. We would also like to maintain several of the key features in a CSP based programming model, such as composition, and shared-nothing.

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.

In the perpetual quest for more computing power, the GPGPU has been extensively researched. This research and development has resulted in multiple programming environments, which allows programmers of different varieties to utilize the GPGPU's parallel processing capabilities. However, the GPGPUs are essentially co-processors and do not function without a main processor. GPGPUs are also not a panacea as many common computation problems are irregular and do not fit well into the massively parallel programming paradigm.

The *Field Programmable Gate Array* can achieve performance comparable to GPGPUs at competitive prices. These FPGA units can also run in stand-alone mode with no additional control processor. Furthermore, FPGAs can achieve the peak performance with twenty times less energy consumption than GPGPUs. Unfortunately, to achieve these attractive elements the programmer must essentially design an integrated circuit on the gate level, which is arguably extremely time consuming and difficult. Existing frameworks and tools exist, but have not yet been widely embraced [5].

In this work, we investigate a programming paradigm, which is targeted at software programmers rather than circuit engineers. Thus, our framework is based on a top-down iterative refinement of a program rather than composing in a bottom- up fashion. In the followin we will first revisit our previus work, based on PyCSP, and then move on to further motivate and introduce the synchronous message exchange.

1. Related Work

Programming FPGAs with high level abstractions has been attempted in many variations, in both industry and academia. The current industry standard appears to be the low-level languages *VHDL* and *Verilog*. Higher level approaches are offered by the two key FPGA manufacturers *Xilinx* and *Altera* in the form of a C like language called *Vivado C* from the former and an OpenCL implementation from the latter.

The CSP inspired language Handel-C [6] was also in use but is no longer available.

The IBM framework named *Liquid Metal*, or simply *Lime* is a Java task-based framework that can convert Java bytecode to either OpenCL or Verilog and execute these through a special virtual machine implementation on either GPGPU or FPGA hardware.

A number of other solutions are in use, such as the NI LabView, which allows designing programs with a graphical interface, and can execute these on multiple platforms, including FPGAs. The Matlab HDL Coder is another environment that allows synthesis of Matlab functions into VHDL.

Even with all these on-going approaches to synthesis from high(er) level languages, there is a general understanding that FPGAs will not see more general usage without a solid high level framework to build on [5]. The reasons for this are general fragmentation and interoperability issues with other tools.

2. PyCSP for Hardware Design

Our two students made a heroic effort to implement the vector processor using PyCSP [7]. The initial approach was to use a purely self-timed approach that fits nicely with the implicit coordination found in CSP. The initial work on this self-timed approach was eventually halted, when it became apparent that the logic required to handle coordinated reads by a single process, from multiple writers with a single reader was prohibitively complex. This complexity arises because an external choice only offers reading from multiple writers, but in the hardware logic we require that the item read is from *a particular* writer. While PyCSP offers external choice both on input and output channels, and indeed a combination of the two, in the full Bohrium processor implementation with more than 500 non-trivial processes, i.e. not counting latches and other simple management processes, external choice was not used even once, in all cases it was deterministic which channel a process would write to or read from at any given point.

The next approach was to redesign the logic, and instead implement a clock process that drives the circuit. This greatly reduced the complexity of the logic, but introduced a significant increase in the number of processes. Each process must now explicitly read the clock signal, but to avoid race conditions between reads and writes, there must be two distinct clock signals, the so called *tick* and *tock* signals. This approach then requires the introduction of channels with a single buffer element to prevent deadlocks in the PyCSP processes when one process is in the read phase, and another is attempting to write to that process. Figure 1 shows a simple reader-writer network in CSP and the equivalent network with tick-tock and a latch added, the added complexity is evident.

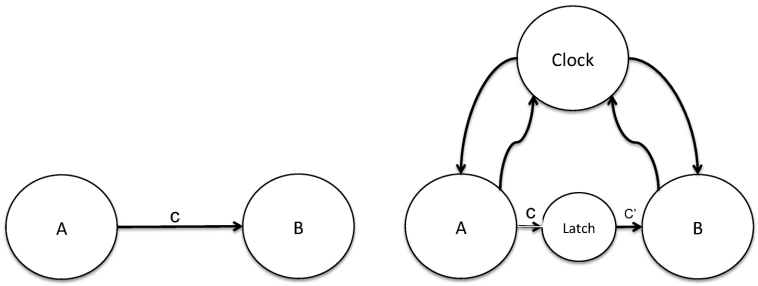


Figure 1. A simple reader-writer network in CSP and with clocks added.

This explosion in channels required development of new tools to manage the amount of channels. These tools include a static verifier, which traverses the full PyCSP network, as well as a graphical tool for visualizing the processes. Since the processes cannot be fully anonymous in this model, a new design scheme was added, which allows processes to use named channels. These channels are automatically created and connected at runtime, with a few basic checks to ensure that all channel ends are connected.

Despite these obstacles, it was possible to synthesize some smaller PyCSP networks entirely into VHDL and Vivado-C.

The primary reason for building processor simulators in Python is the convenience and flexibility that comes from using a high productivity language such as Python. Python allows us to easily experiment with new ideas and new components are easily prototypes and evaluated for feasibility. Thus the added complexity that comes from channels that explicitly propagate clock-signals and the latch processes that are inherent with the hardware design process, works against the perception of Python as a high productivity language. This was of course remedied to a certain extend from special tools; such as automatic connection to the clock-process and special visualizers that offers to abstract away the clock channels and latch processes. An example of just how complex the full processor becomes is shown in figure 2,

note that the full processor is too big for readability in this format and thus it is only included to give an impression of the challenges that arise with modelling using PyCSP.

The conclusion was that PyCSP alone was not a viable tool for describing timed hardware, which is logical since it is essentially forcing a globally synchronous environment onto CSP model. Another important lesson is that the powerful *external choice* concept in CSP could not be utilized in this globally synchronous approach.

However, many of the other concepts in CSP were found to be very valuable. Using a shared-nothing approach with strict message passing is a good match to the hardware concept where communication happens through ports. The compositionality of CSP processes was also very beneficial as it allows both bottom-up and top-down approaches as well as maintaining strict encapsulation.

3. Synchronous Message Exchange

The overall conclusion from the in-depth project on PyCSP 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.

Figure 3 shows a graphical equivalent setup of one process broadcasting to three other processes in a synchronous setup. The reduced complexity of the SME approach for this scenario is clear even from this simple setup. Note that the channels with arrows in both ends are used to represent a channel pair, so these are in fact two channels in CSP. Since a pure CSP implementation is indeed possible all of the properties of CSP, including reasoning on deadlocks etc should be maintained.

The driving idea behind the Synchronous Message Exchange model is the emulation of a clock signal to better model hardware. Processes are connected to broadcasting channels, here called busses to avoid confusion with conventional CSP channels. To further support the hardware analogy, the SME approach also works directly with latches, in hardware description languages usually called ports. The execution flow for a process is then simple: before a clock-cycle is executed all input-ports are read into from the bus they are connected to,

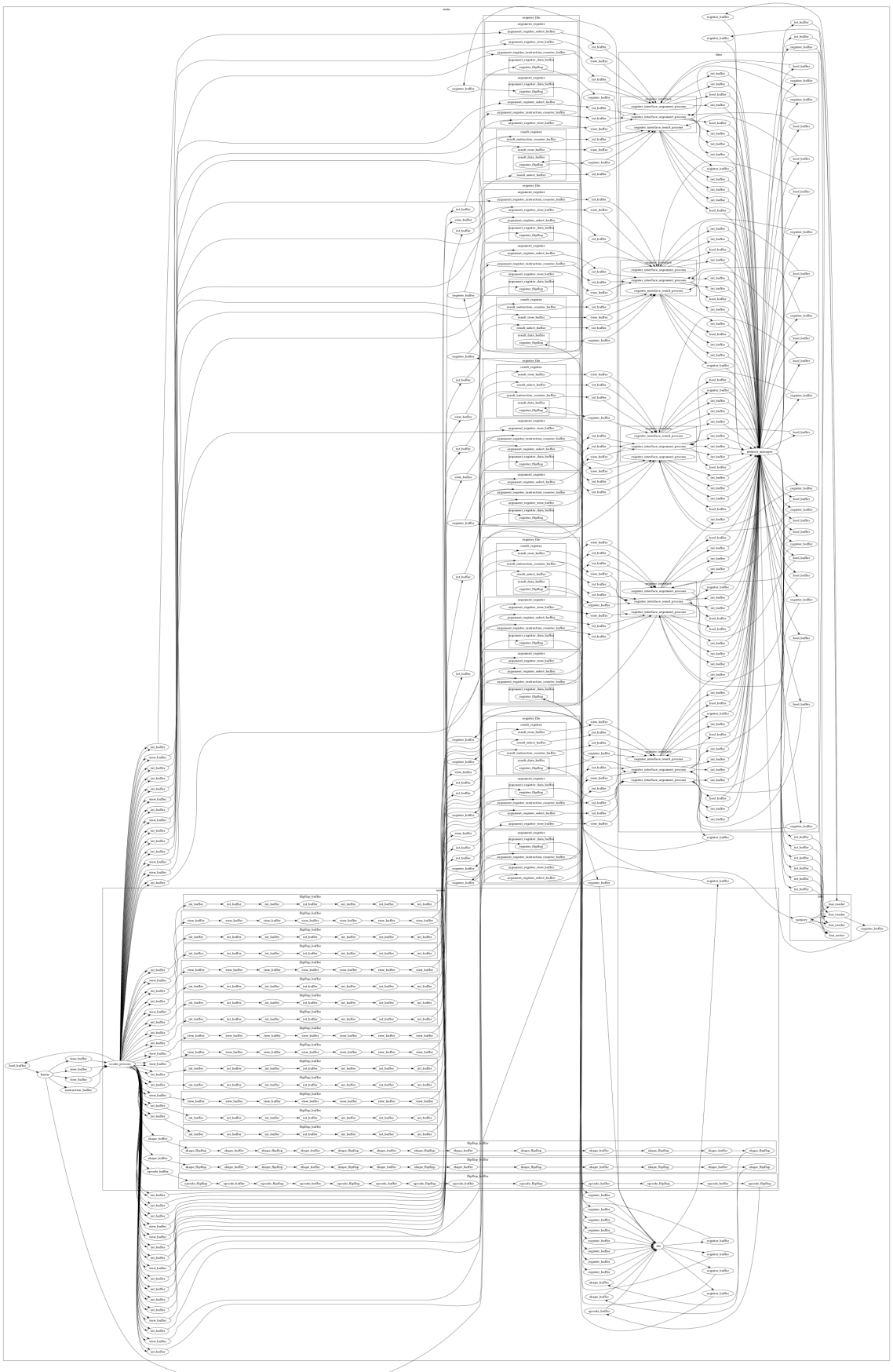


Figure 2. A full Bohrium processor from the student project. A full version can be seen at: http://goo.gl/j1w7Da

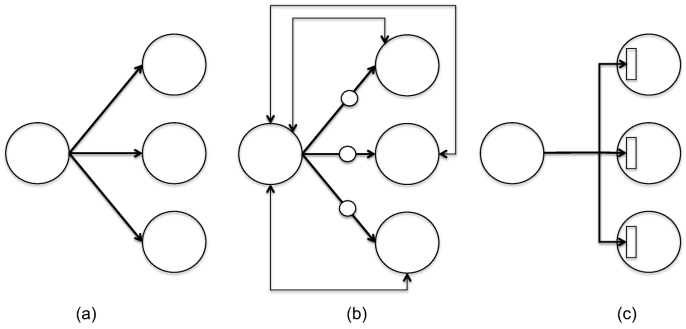


Figure 3. A motivating example, (a) is the simple synchronous network we wish to model, (b) the actual implementation in PyCSP and (c) the solution with the proposed synchronous message exchange. In the latter the squares inside the processes signifies the implicit messaging slots.

then the process is executed, and finally output ports are written to the bus. The bus model strictly maintains a hardware analogy, i.e. exactly one process may write to the bus in any one clock-cycle. If a bus is not written to in a cycle, or if more than one process writes to the bus an exception is raised. The analogy to the hardware clock is sketched in figure 4.

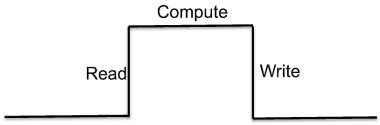
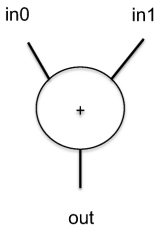


Figure 4. A schetch to link a physical clock to the SME flow.

Processes are described as Python objects that inherit from a process-class and must implement a run method, and the process constructor must call a parent method to connect its local ports to the assigned busses. The run method is different from the solution in PyCSP and similar libraries in that the run method starts and terminates in every clock-cycle, i.e. the while loop that usually repeats the functionality of a process is eliminated. Figure 5 shows a trivial process that adds two numbers. This is central to the clocked nature of the SME model and makes modeling easier for the programmer since the body of a process only needs to express what the process does semantically, not the communication nor the iterations.




```
class add(sync_process):
  #Inputs: {in0, in1} Outputs: {out}
  def __init__(self, name, inputs, outputs):
    self.connect(name, inputs, outputs)

  def run(self):
    self.out = self.in0 + self.in1
```

Figure 5. A simple process that adds two inputs and outputs the result.

Processes are connected in a network through the construct of a component. We obtain compositionality by defining components as consisting of processes and/or components. By allowing components to consist of components hierarchy, we allow libraries of components to the formed for later reuse, an example could be an ALU as a component which in itself consist of smaller components, all of which can be passed arguments from the higher level definition, i.e. the bit-width of the ALU. Busses are explicitly named and connections to busses are through that namespace, i.e. busses are not passed to processes as in most CSP libraries, but linked through the namespace. A link to a bus is defined as a pair of names, the name of the local port in the process that is connected to the bus, and the name of the

bus. Links are made directional, i.e. a link to a bus is either input or output. Technically the network is passed as Python dictionaries, and the port variables are created by the process connection method, i.e. the programmer does not create the actual variables that model the port, the SME system does that. Figure 6 shows a small component that connects the adder from figure 5 to two constants as inputs and a printing process as the output.

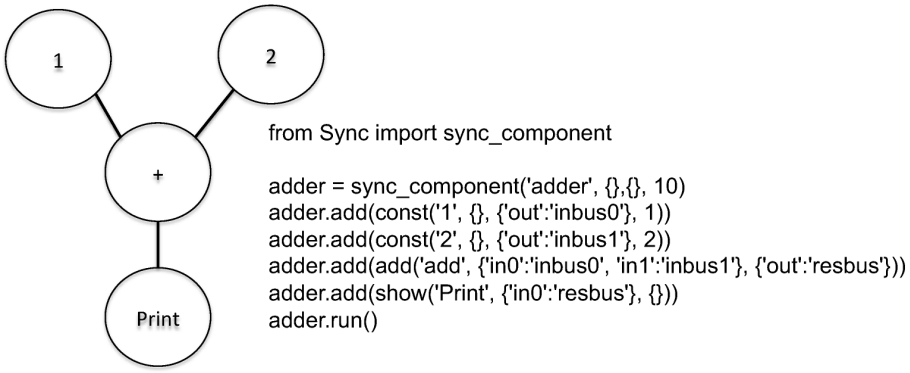


Figure 6. A simple component that adds two constants and prints the result.

In essence what the SME implements is equivalent to what would be a Any2All channel, where all channels have an overwriting buffer size of one as known from JCSP[8]. The broadcasting channel sends unique copies of the message to all recipients, however since Python is inherently object oriented, if the message is a reference to an object the unique references will still point to the same, then shared, object. This challenge is well documented in JCSP, and while it has been shown that this can be eliminated in PyCSP[9] we have chosen not to enforce immutable objects for now, partly because of the added complexity and increased runtime, partly because, if used properly, shared objects may indeed be useful. The Any2All channel is fundamentally different from other kinds of channel-construct, even though it too has a CSP equivalence. For now we do not consider merging the Any2All with the existing PyCSP package as its very nature is so different that it is not obvious whether the Any2All model can even interoperate with the standard Any2Any channel in PyCSP. It should be noted that conventional CSP does infact allow for broadcasting:

$$c!I \rightarrow SKIP [\![\{\!| c |\!\}]\!] X,$$

where X is a collection of parallel processes with c in their alphabet, so that passing I along c can only happen if all processes in X input from c. This is, to the knowledge of the authors, not possible in any of the available CSP-like libraries, including the above mentioned.

3.1. Examples of SME

3.1.1. Simple CPU

Since SME is designed with hardware simulation in mind, it makes sense to demonstrate through the use of a hardware example. We considered reusing the vector processor from [2] but the more precise implementation we have built with SME consists of more than 130 synchronous busses connecting 18 components and thus becomes too complex for presentation here, the vector processor simulator will be made available online for those interested. Note that the SME is sufficiently different from PyCSP, in particular the transparent communication and lack og while-true loops in functions, that nothing from the previous, or the student, vector processor has been reused.

Instead, we demonstrate using a tiny CPU, which consists of only a micro-code controller, an ALU and a Register file. A diagram of the simple CPU is shown in figure 7 and the entire implementation in figure 8.

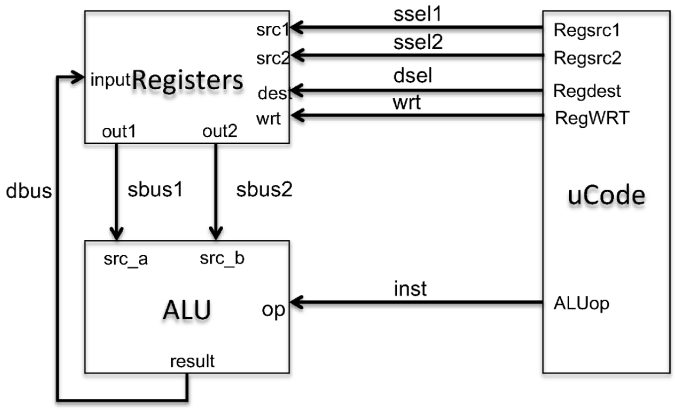


Figure 7. A simple CPU.

```
from Sync import sync_process, sync_component

class uCode(sync_process):
    def __init__(self, name, inputs, outputs):
        self.connect(name, inputs, outputs)
        self.program =[(0,1,2,1,0,0)]
        self.ip = 0

    def run(s):
        s.Regsrc1, s.Regsrc2, s.Regdest, s.RegWRT, \
                                s.ALUop, s.ip =
s.program[s.ip]

class ALU(sync_process):
    def __init__(self, name, inputs, outputs):
        self.connect(name, inputs, outputs)
        self.ops =['+', '-', '*', '/']

    def run(self):
        self.result = eval('%s %s %s'%\
                    (self.src_a, self.ops[self.op],
self.src_b))
```

```
class Register(sync_process):
    def __init__(self, name, inputs, outputs, entries):
        self.connect(name, inputs, outputs)
        self.regs = [0]*entries

    def run(self):
        self.out1 = self.regs[self.src1]
        self.out2 = self.regs[self.src2]
        if self.wrt:
            self.regs[self.dest]=self.input

cpu = sync_component('CPU', {}, {}, 100)
cpu.add(Register('Register',
            {'src1':'ssel1','src2':'ssel2','dest':'dsel','wrt':'wrt',\
             'input':'dbus'},
            {'out1':'sbus1','out2':'sbus2'}, 32))
cpu.add(ALU('ALU',
        {'src_a':'sbus1','src_b':'sbus2','op':'inst'},
        {'result':'dbus'}))

cpu.add(uCode('uCode',
          {},
          {'Regsrc1':'ssel1','Regsrc2':'ssel2','Regdest':'dsel', \
           'RegWRT':'wrt','ALUop':'inst'}))

cpu.run()
```

Figure 8. Full implementation of the simple CPU.

3.1.2. Frogger

The synchronous progression that hardware simulations require are also common in many games, thus we decided to see if the SME library was also suitable for game development. To keep the focus on the actual task, and not the game itself, we implemented a subset of a very old game, Frogger [10]. The gameplay in our version is exceedingly simple, a frog, which may only move one step along the Y-axis, is connected to a bus to which it, in every step of the game, broadcasts its position. Cars, which may only move along the X-axis, read the frog position and determine if there is a collision. Both the Frog and the Cars are connected to a point-checking bus to which the Frog send a positive number if it reaches the top or bottom of the game-area, while the Cars sends a negative number upon collision with the Frog. Figure 9 shows both the layout of the processes in the game and a snapshot of the running game.

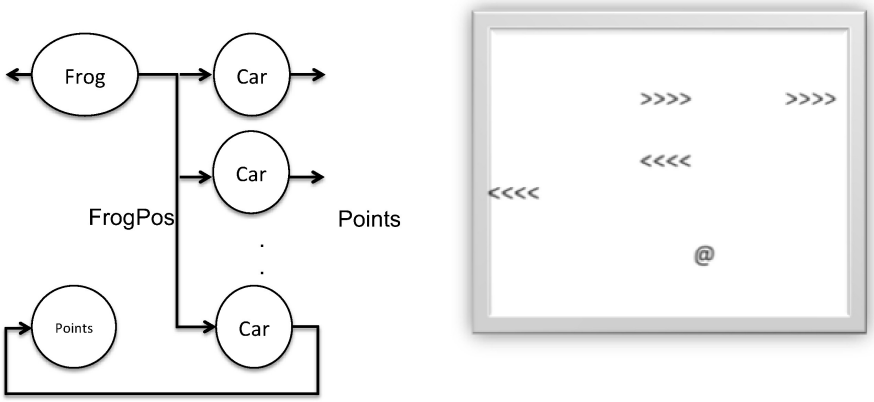


Figure 9. Structure and snapshot of the frogger game. The frog is the `@` while the `<<<` and `>>>` are cars in different directions.

The full frogger-game is implemented, with no consideration to size, in only 91 lines of code. Adding more cars is, as it would be in a conventional CSP library, simply a matter of adding more car processes. The overall progress in the gameplay is coordinated by the global synchronization in SME which simplifies the implementation a lot.

3.1.3. Commstime

Tradition dictates that any new CSP orientated programming library should demonstrate the commstime benchmark. While this is a seemingly simple network to build, also with SME, the synchronous nature of SME makes commstime quite different since every bus exchanges a message every cycle. This means that where the original network output a sequence of unique integers, the same network, run with SME, will produce the same number replicated as many times as there are processes in the path. The full implementation of commstime is shown in figure 11, and the layout in figure 10.

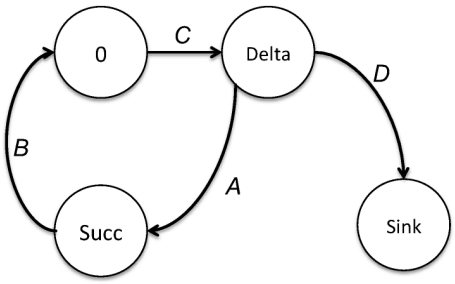


Figure 10. Diagram of commstime in SME.

The SME model makes it trivial to attach a probe to a bus to monitor the activity, thus adding four lines helps us understand how commstime behaves in the SME version. The added code and the result in a formatted version is shown in figure 12.

SME is a much simpler model than PyCSP. For commstime in particular we can see that the fact that scheduling is not needed is visible in performance. On a older iMac 2.8GHz Core i5 processor, PyCSP manages approximately 13.1 us per communication while SME requires only 2.9 us. This is not surprising since SME is so much simpler and does not involve an evaluation of which processes are ready to run. While SME is primarily designed to make hardware modeling simpler, it is beneficial to run app 4x faster for long CPU simulations, once we evolve to simulating a full CPU in the SME approach as well.

```
from Sync import sync_process, sync_component

class prefix(sync_process):
  def __init__(self, name, inputs, outputs, value):
    self.connect(name, inputs, outputs)
    self.out = value

  def run(self):
     self.out = self.in0

class delta2(sync_process):
  def __init__(self, name, inputs, outputs):
    self.connect(name, inputs, outputs)

  def run(self):
     self.out0 = self.in0
     self.out1 = self.in0

class succ(sync_process):
  def __init__(self, name, inputs, outputs):
    self.connect(name, inputs, outputs)
  def run(self):
      self.out0 = self.in0 + 1


class sink(sync_process):
  def __init__(self, name, inputs, outputs):
    self.connect(name, inputs, outputs)

  def run(self):
     pass

comms = sync_component('CommsTime', {},{}, 50000)
comms.add(prefix('Prefix', {'in0':'c'}, {'out':'a'}, 0))
comms.add(delta2('Delta2', {'in0':'a'}, {'out0':'b', 'out1':'d'}))
comms.add(succ('Succ', {'in0':'b'}, {'out0':'c'}))
comms.add(sink('Sink', {'in0':'d'}, {}))
import time
start = time.time()
comms.run()
stop = time.time()
print 'dt = ',stop-start
```

Figure 11. Implementation of commstime in SME.

```
comms.add(dump('A',{'input':'a'}, {}, 'A'))
comms.add(dump('B',{'input':'b'}, {}, 'B'))
comms.add(dump('C',{'input':'c'}, {}, 'C'))
comms.add(dump('D',{'input':'d'}, {}, 'D'))
```

Cycle	A	B	C	D
1	0	0	0	0
2	0	0	1	0
3	1	0	1	0
4	1	1	1	1
5	1	1	2	1
6	2	1	2	1
7	2	2	2	2
8	2	2	3	2
9	3	2	3	2
10	3	3	3	3

Figure 12. Logging every bus in commstime.

Conclusions

From a courageous effort from two students, we have learned that PyCSP is not nearly as well suited for modeling hardware circuits as we had previously concluded. In this work we have introduced a much simpler solution for synchronous messaging networks, SME. We have sketched a very simple CSP equivalence for the SME model, however. While the Any2All channel with one overwriting buffer per receiver, can easily be implemented in PyCSP, it is not self-evident that the channel could be merged with the existing PyCSP package, since the nature of the synchronous channels is very different. An indication of this difference is that the SME library is more than four times faster than PyCSP, even with the current, rather nave, implementation.

We have demonstrated how the synchronous model makes modeling hardware much easier than with PyCSP, and also how the same model may be used for other, inherently synchronous, applications.

Future Work

We remain undecided whether it is formally possible to include the new Any2All channel type with the existing PyCSP, and will seek to investigate this at a later time. In addition,

an obvious extension is to allow different synchronous components to run at different speed, relative to the virtual clock. Technically, we already allow for this, but the correctness has not been validated in all corner cases.

The student project demonstrated that large portions of their code could automatically be translated to Vivaldo-C for actual synthetization. We believe that this will be even simpler for SME based simulations, but this remains a future task.

Acknowledgements

We owe a large gratitude to Esben Skaarup and Andreas Frisch, two former students who managed to extend PyCSP with tools that enabled them to actually model a processor in its full complexity. Their work and conclusions enabled us to conclude that a much simpler solution than PyCSP was called for.

We owe a great gratitude to the anonymous reviewers for valuable comments on our original, rather different, version of this paper. Especially we would like to thank the reviewer that provided us with the formal description of broadcast in CSP.

References

[1] 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, November 2009.

[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, November 2013.

[3] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[4] Thomas Cheatham, Amr Fahmy, Dan Stefanescu, and Leslie Valiant. Bulk synchronous parallel computing: a paradigm for transportable software. In *Tools and Environments for Parallel and Distributed Systems*, pages 61–76. Springer, 1996.

[5] David F Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[6] Handel-C Synthesis Methodology. `http://www.mentor.com/products/fpga/handel-c/`. [Online; accessed 26 July 2013].

[7] Esben Skaarup and Andreas Frisch. Generation of FPGA hardware specifications from PyCSP networks. Master's thesis, University of Copenhagen, Niels Bohr Institute, 2014.

[8] Nan C. Schaller, Gerald H. Hilderink, and Peter H. Welch. Using Java for Parallel Computing - JCSP versus CTJ. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Communicating Process Architectures 2000*, pages 205–226, September 2000.

[9] Mads Ohm Larsen and Brian Vinter. Exception Handling and Checkpointing in CSP. In Peter H. Welch, Frederick R. M. Barnes, Kevin Chalmers, Jan B. Pedersen, and Adam T. Sampson, editors, *Communicating Process Architectures 2012*, pages 201–212, August 2012.

[10] Frogger. `http://www.arcade-museum.com/game_detail.php?game_id=7857`. [Online; accessed May 30th 2014].