Connections Guide

Latency Insensitive Channel Library

MATCHLIB, NVIDIA

MatchLib communications methodology is based on high-level synthesis (HLS) on the latency-insensitive design (LID) paradigm. Systems are fully designed in synthesizable SystemC and C++, and components are connected through synthesizable SystemC latency-insensitive (LI) channels. The approach is based on a library and API of latency-insensitive channels called *Connections*, which is the subject of this guide.

1 INTRODUCTION

MatchLib's *Connections* is a library and API of latency-insensitive channels. It was presented for the first time at DAC 2018 as part of a new modular digital VLSI methodology [Khailany et al. 2018]. To know the motivation behind *Connections* refer to section 2.3 of [Khailany et al. 2018].

All components of this library are HLS-able and they are designed to be synthesized with Mentor Catapult. Table 1 shows an overview of the most relevant components and API in *Connections*.

Table 1. API of *Connections*, reflecting unified terminals (ports) and types of channels

Port	**Functions**
In<T>	Pop(), PopNB()
Out<T>	Push(), PushNB()
InBuffered<T>	Pop(), PopNB(), Empty(), Peek()
OutBuffered<T>	Push(), PushNB(), Full()

Channel	**Description**
Combinational<T>	Combinationally connects ports
Bypass<T>	Enables DEQ when empty
Pipeline<T>	Enables ENQ when full
Buffer<T>	FIFO channel
OutNetwork<T>, InNetwork<T>	Network channels: packetizer and de-packetizer

2 PORTS

A module's latency-insensitive (LI) interface is composed of ports, with which it connects to other modules through channels. The port is the element through which a module enforces the LI protocol. The communication invariant is that a transaction is valid when at the clock edge both *valid* and *ready* are true. Because of that, the behavior of the signals of the LI ports cannot be scheduled freely by the HLS tool. Therefore, Catapult provides a pragma called *modular IO*, used to enforce cycle accurate timing on the interface. See for example the use of the pragma in Listing 3.

There are mainly two types of ports: unbuffered and buffered. The *In* and *Out* ports are just regular ports, implemented in the traditional way that Catapult would expect. They are a generic C++ classes and they can be declared in any `sc_module` as member variables. The API for these ports includes *Pop()* and *PopNB()* for the *In* port, *Push()* and *PushNB()* for the *Out* port (see Table 1).

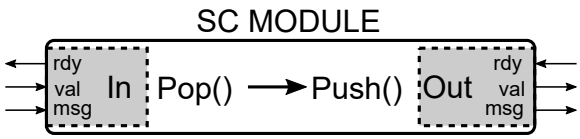

Fig. 1. Simple module with one input and out output ports.

For this section we will consider a simple module with one input and one output ports. Data is popped from the input port, some function is evaluated combinationally or through a multi-stage pipeline and the result is pushed into the output port. We aim at a pipelined module with initiation interval (II) of 1 and flush enabled, as that's almost always the requirement. The way to enforce that is to pass two directives to Catapult in the tcl file running the HLS. A good place to put them is between the `go assembly` and `go architect` commands. An example is in listing 1.

```
1
2 directive set /module_name/process_name/while -PIPELINE_INIT_INTERVAL 1
3
4 directive set /module_name/process_name/while -PIPELINE_STALL_MODE flush
```

Listing 1. Example of HLS directive to pipeline the main loop of a process

See in Figure 1 the simple module, there the API's used to access the ports are `Pop()` and `Push()`. The arrow connecting them can implement any function, but in Listing 2 we show that as a simple passthrough. That's the `while()` loop of the only process inside the module (an `SC_THREAD` in this case).

```
1
2 while(1) {
3
4   msg_t data = inport.Pop();
5
6   outport.Push(data);
7
8   wait();
9
10 }
```

Listing 2. Example of a blocking use of the API.

The *Pop()-Push()* use of the API is blocking: in the data path from input to output there's at least one place where the code would stall until the input interface becomes valid or the output interface becomes ready. `Pop()` and `Push()` a blocking behavior, unlike their non-blocking counterpart: `PopNB()` and `PushNB()`. The implementation of these four functions is shown in Listing 3 . The blocking operations have an internal loop that polls on the interface by trying the transaction until it's successful. Differently, the non-blocking ones, try the transaction only once and return a boolean which indicates whether the transaction was successful or not. Thus, it's possible to call the non-blocking APIs on multiple interfaces within the a single clock cycle even if they are not successful.

```
1
```

```
// Pop

#pragma design modulario < in >

msg_t Pop() {

  do {

    rdy.write(true);

    wait();

  } while (val.read() != true);

  rdy.write(false);

  return msg.read();

}



// PopNB

#pragma design modulario < in >

bool PopNB(msg_t& m) {

  rdy.write(true);

  wait();

  rdy.write(false);

  m = msg.read();

  return val.read();

}



// Push

#pragma design modulario < out >

void Push(const msg_t& m) {

  do {

    val.write(true);

    msg.write(m);

    wait();

```

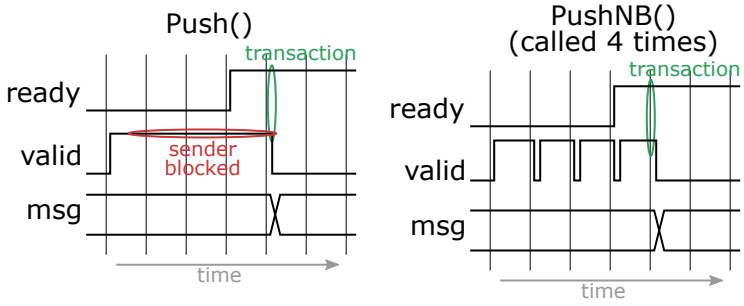

Fig. 2. Simple timing diagram of the Push() API stalling by the output port.

```
58   } while (rdy.read() != true);
59
60   val.write(false);
61
62 }
63
64
65
66 // PushNB
67
68 #pragma design modulario < out >
69
70 bool PushNB(const msg_t& m) {
71
72   val.write(true);
73
74   msg.write(m);
75
76   wait();
77
78   val.write(false);
79
80   return rdy.read();
81
82 }
```

Listing 3. Simplified version fo the API for non-buffered ports.

Looking closely at `Pop()`, we see that as soon as it's called the `valid` signal is set to true. At the next clock edge (i.e. after the wait()), if the incoming ready signal is true, the transaction is considered completed, otherwise the ready is raised again. Basically the valid is kept true until the ready is also true at the clock edge. In Figure 2 on the left you can observe the case of a `Push()` while the ready signal is false.

All the ways to have a blocking behavior will be presented in the next section. If there's something blocking in a process then it will not be possible to reach parallelism, like to have multiple in-to-out links that work in parallel. For that we need the non-blocking APIs: `PopNB()` and `PushNB()`. Additionally, either `Full()` or `Empty()` to check in advance if the output is ready to receive or if the input has incoming valid data, respectively. Listing 4 shows one example of non-blocking behavior. The `Full()` API simply returns the ready signal of the output port.

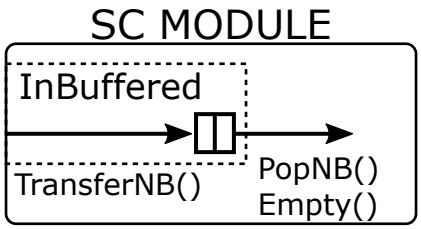

Fig. 3. OutBuffered port API's.

```
1
2  while(1) {
3
4    if (!outport.Full())
5
6      if (inport.PopNB(data))
7
8        outport.Push(data);
9
10   wait();
11
12 }
```

Listing 4. Examples of a non-blocking use of the API.

Due to Catapult's limitation, `Full()` and `Empty()` do not work on the regular `Out` and `In` ports of *Connections*, as the tool does not allow a combinational path between *ready* and *valid* . For instance, in this case in Listing 4 *valid* is set by `PushNB()` if `Full()` returns true, that is the output port's ready signal is true. `Empty()` simply returns the negation of the `val` input signal. `Peek()` simply returns the input `msg` field. *Full*, *Empty* and *Peek* are not defined with the *modular IO* pragma as they are combinational, there's no cycle accurate timing to be enforced.

To overcome this limitation, *Connections* includes another type of ports: `InBuffered` and `Outbuffered`. These ports contain a FIFO, as shown in Figure 3. The main loop can now use `Full()`, `Empty()` and `Peek()` on the ports because those are applied to the FIFO instead of directly to the output interface, so that the problematic combinational path is internal to the module. Additionally, another API called `TransferNB()` manages the communication between the FIFO and the output interface. `TransferNB` needs to be called at each iteration of the main loop to avoid deadlocks or poor performance, however we will see that there is a specific position for calling this API for a correct and efficient design. Listing 5 shows the `TransferNB()` implementation for the `InBuffered` port. Instead, Listing 6 shows the main loop of a simple module with an `InBuffered` port.

```
1
2  // TransferNB inside InBuffered port
3
4  void TransferNB() {
5
6    if (!fifo.isFull()) {
7
8      msg_t msg;
```

```
9
10      if (this->PopNB(msg))
11
12        fifo.push(msg);
13
14    }
15
16  }
```

Listing 5. TransferNB implementation for input port.

```
1
2  while (1) {
3
4    inport->TransferNB();
5
6
7
8    if (!inport->Empty()) {
9
10     data = inport->Peek();
11
12    if(outport->PushNB(data))
13
14       inport->Pop();
15
16   }
17
18 }
```

Listing 6. Example of TransferNB use for an `InBuffered` port.

Buffered ports have a configurable depth. Let's see an example of instantiation of the 4 different types of port, with buffered ports with depth 2: Listing 7.

```
1
2  // Ports definition
3
4  Connections::In<msg_t> inport;
5
6  Connections::Out<msg_t> outport;
7
8  Connections::InBuffered<msg_t, 2> inport_buf; // 2 is the depth
9
10 Connections::OutBuffered<msg_t, 2> outport_buf; // 2 is the depth
```

Listing 7. Definition of all port types.

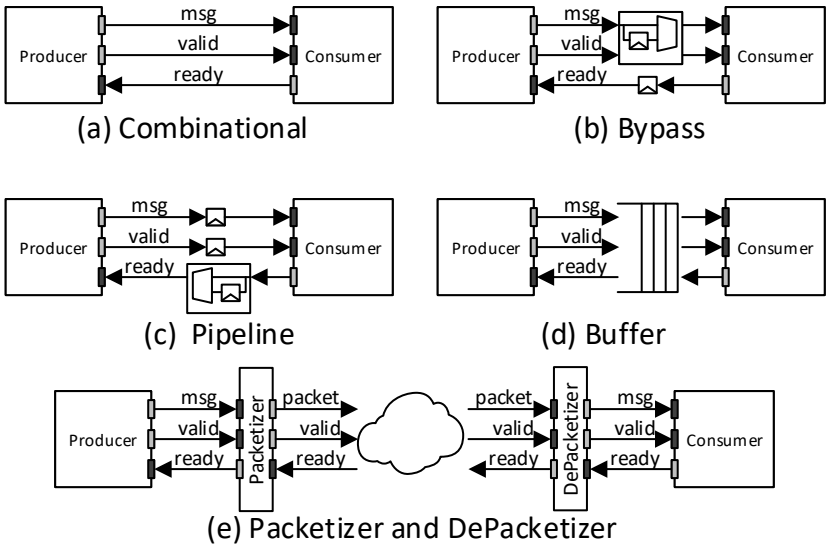

Fig. 4. *Connections* channel implementations.

3 CHANNELS

Connections includes four main types of channels: *Combinational*, *Bypass*, *Pipeline*, *Buffer*. There are two additional channels that are not addressed by this work, they are called *InNetwork* and *OutNetwork* and they packetize a message into and extract from a network, but they are currently not recommended for use in this release. Figure 4 depicts the channels, which are also listed in Table 1.

3.1 *Combinational* Channel

The *Combinational* channel is simply a C++ class that defines three `sc_signal`, which in other words are three wires: the booleans `rdy` (i.e. ready), and `val` (i.e. valid), and a bit vector `msg` (i.e. message). The *Combinational* class receives the data type of the message as a template parameter used to determine the bit width of `msg`. See the first few lines of the class in Listing 8.

Combinational channels are used to connect two modules (`sc_modules`), and in hardware is nothing but the bundle of three signals, there's no need for any member function or variable. However, in software the *Combinational* channel class contains a large number of member functions; most of them are only there for simulation purposes. The use cases for these functions see a *Combinational* channel with only one end attached to an `sc_module`'s I/O port. This is a use case that we do not explore in this document. Notice that some of those additional functions appear to not be in use anymore and they could be removed.

```
1
2 template <typename Message>
3
4 class Combinational {
5
6  public:
```

```
7
8
9
10  // Interface
11
12  typedef Wrapped<Message> WMessage;
13
14  static const unsigned int width = WMessage::width;
15
16  typedef sc_lv<WMessage::width> MsgBits;
17
18
19
20  sc_signal<MsgBits> msg;
21
22  sc_signal<bool> val;
23
24  sc_signal<bool> rdy;
```

Listing 8. First lines of the *Combinational* class.

Intuitively, a *Combinational* channel has zero clock cycles of latency, full throughput and no storage capacity. This channel serves the purpose of connecting two modules, because binding directly the modules' ports is not allowed, basically because they have an opposite directionality. *Combinational* instead is made of `sc_signals` so it can be bound to the `sc_module`'s ports. See how to bind a *Combinational* channel in Listing 9. The binding works regardless of the ports type.

```
1
2  // Member variables: sc_modules and channel
3
4  sender_module_t sender;
5
6  receiver_module_t receiver;
7
8  Connections::Combinational<msg_t> channel;
9
10
11
12 // Binding inside constructor
13
14 sender.inport(channel);
15
16 receiver.outport(channel);
```

Listing 9. Example of classic Combinational channel binding.

This is the most common type of channel. Ideally all channels should be combinational, so that they don't add latency or area. However there are cases where we might either want to break a channel that is physically too long, therefore too much timing delay, or cases in which the system-level composition requires channels with a depth of one or more to balance out converging paths with a mismatched amount of pipeline stages. The latter is done to avoid deadlock and

stuttering, solving a problem called re-convergence; there's a good description of it in *Mentor*'s *HLS Bluebook* [Mentor 2017] in Section 8.3: *"Reconvergence: Balancing the Latency Between Blocks"*.

Because of the above reasons, we may have situations where we want channels with storage capabilities and either by-passable or with no combinational paths through them. Hence the need for three types of channels presented next.

3.2 *Bypass* channel

The idea behind the *Bypass* channel is to have a by-passable queue for the *message* and *valid* signals and a registered *ready* signal going in the opposite direction, as shown in Figure 4. Why registered? Basically for this channel, the *ready* signal received by the sender is true when the queue is not full. The *full* status is updated with a finite state machine, whose inputs include the ready signal coming from the receiver. Hence, it takes one clock cycle for the ready signal to go from receiver to sender. Instead, the valid signal on the receiver's side is the logic or of the valid coming directly from the sender and the inverse of the *empty* status of the queue. Thus, here we have a combinational path for the *valid*. The *message* behaves similarly.

3.3 *Pipeline* channel

The Pipeline channel buffers *valid* and *message*, but it has a combinational path from receiver to sender for the *ready* signal. Basically, in the case of a full queue, a valid=true from the sender and a ready=true from the receiver, it is possible to dequeue and enqueue the message simultaneously. This is basically the behavior of a pipeline with draining.

This channel has a minimum latency of 1 and a capacity of 1. When the Pipeline channel with configurable size will be available, the capacity will be equal to the channel's depth. This still has a combinational path, but it's only 1 bit, much less than for the *Bypass* channel.

3.4 *Buffer* channel

The *Buffer* channel breaks all combinational paths, it's not possible to both enqueue and dequeue when the queue is full. As a result, this module can only have full throughput with a depth of at least two and in fact the depth of 1 is not even supported. Trying to declare a *Buffer* channel with depth of 1 raises an assert. For any other depth, just like for the other channels, the maximum throughput is always one. None of these channels will ever be the cause for a reduced throughput. The minimum latency for the *Buffer* channel is one clock cycle and the capacity is equal to the channel's depth.

3.5 Binding with *ChannelBinder*

The *Bypass* channel, as well as the *Pipeline*and *Buffer* ones, in *Connections* is an `sc_module`. Mainly made of `SC_METHOD` processes, these channels are designed in a RTL-like manner. They are then different from the *Combinational* channel, which is a generic class.

Since it's not possible to bind ports of different `sc_modules` directly, the binding of *Bypass*, *Pipeline* and *Buffer* channels was previously very verbose. Therefore now *Connections* contains a new templated class called `ChannelBinder`. It allows for a much more compact binding of modules and channels of any type. Listing 10 shows how to define the channel and how to bind it to the sender and receiver's ports inside the constructor. This example is valid also for all the other channels, for which of course the channel declaration would change, whereas the binding would be the same. Notice that clock and reset are not actually needed for the *Combinational* channel, but one can decide to pass them anyways as arguments so that the binding one-liner is the same for all channels types.

```
1
2  // Types
3
4  typedef Connections::ChannelBinder<data_t> binder_t;
5
6
7  // Member variables: sc_modules and channel
8
9  sender_module_t sender;
10
11 receiver_module_t receiver;
12
13 Connections::Bypass<msg_t, 2> channel;
14
15
16 // Binding inside constructor
17
18 binder_t *binder = new binder_t(receiver.inport, sender.outport, channel);
```

Listing 10. Example of channel binding with ChannelBinder. This applies to all types of channels.

As you can see in Listing 10, the *Bypass* channel takes a second template parameter, which is its depth. The same goes for the *Buffer* channel. *Pipeline* channels as of now can only have a depth of 1.

3.6 Channel Summary

Connections channels' implementation is really solid, because the scheduling that Catapult does on them is almost none. The only drawback that should be known is that the channels do not get optimized together with the modules they connect. They are modules themselves, therefore they are scheduled on their own. For example, if a module requires a bypass register at its input to have full throughput, connecting a bypass channel at the input is not expected to yield full throughput. Catapult doesn't know about the channel while scheduling the component in which the bypass register is needed. If a certain degree of buffering is required by a module for functioning, it is suggested to put the buffering inside the module. Instead, if the buffering is needed at the system level and the module can work properly without it when tested alone, then it is suggested to put the buffering outside. This guarantees modules with minimum buffering and that can be reused more easily. In summary, only embed the extra buffering inside a module if that's strictly needed for the module to work properly in stand alone.

When should each type of channel be used? When composing modules there are system level requirements which might require extra storage on a channel to avoid stuttering, deadlock or to break the combinational paths. The latter happens either because modules are placed physically far away or because the modules connected to the channel have a combinational path between the ready and valid signals. For these case the designer can look into using *Bypass*, *Pipeline* or *Buffer* channels. Each of them provides different features in terms of latency, throughput, capacity and combinational paths. The designer shall choose the most appropriate one, based on those features.

The performance specifications of all channels are laid out in Table 2. To close this section, Listing 11 shows an example of a simple module called *top*, that instantiates two submodules (assuming they are defined elsewhere) and then links them with a channel.

Table 2. *Connections* channels features.

Channel Type	**Min Latency** (cycles)	**Max Throughput**	**Storage Capacity** (# of messages)	**Combinational Paths**
Combinational	0	1	0	message + ready + valid
Bypass	0	1	buffer depth	message + valid
Pipeline	1	1	1 (potentially buffer depth)	ready
Buffer	1	1	buffer depth	none

```
1
2  class top : public match::Module {
3   public:
4    sender_module_t sender;
5    receiver_module_t receiver;
6
7    Connections::In<data_t> inport;
8    Connections::Out<data_t> outport;
9    Connections::Combinational<data_t> channel;
10
11   top(sc_module_name nm)
12   : match::Module(nm),
13     inport("inport"),
14     outport("outport"),
15     channel("channel") {
16
17
18     // Binding
19     sender.clk(clk);
20     sender.rst(rst);
21     sender.inport(inport);
22     sender.outport(channel);
23
24     receiver.clk(clk);
25     receiver.rst(rst);
26     receiver.outport(outport);
27     receiver.inport(channel);
28
29   }
30
31 };
```

Listing 11. Example of simple module with two submodules connected by a channel.

4 BEST PRACTICES

Now that the basics on Connections are laid out, let's look at all the possible use cases of the API. Here we will provide best practices to achieve successful HLS, full throughput, minimum latency and design correctness with very few design iterations.

POP INPUT				
ID	CODE	BLOCKING	OPTIMAL	COMMENTS
1	`indata = inport.Pop();`	B	Y	
2	`while(!inport.PopNB(indata))` `{}`	B	N	
3	`if(inport.PopNB(indata))` `{}`	NB	Y	

4	`if(!inport.Empty()) {`
	`  indata = inport.Pop();`
	`}`
		NB	Y	Combinational ready-valid path. Ready set combinationally w.r.t to valid.

PUSH OUTPUT				
ID	CODE	BLOCKING	OPTIMAL	COMMENTS
A	`outport.Push(outdata);`	B	Y	
B	`while(!outport.PushNB(outdata))` `{}`	B	N	
C	`if(outport.PushNB(outdata))` `{}`	NB	Y	

D	`if(!outport.Full()) {`
	`  outport.Push(outdata);`
	`}`
		NB	Y	Combinational ready-valid path. Valid set combinationally w.r.t to ready.

Fig. 5. All possible ways to access the ports in *Connections*.

4.1 All API uses

Figure 5 lists all the possible ways that *Connections* provides to access input and output ports. These code snippets would live in the main while loop of an SC_THREAD or SC_CTHREAD of a module. You can imagine them being used in complex modules, but all the uses fall into the categories in the table.

In most cases the data popped from an input port goes through a certain function or pipeline, and some data will come out and be pushed to the output port. We can almost always identify pop-push links throughout a module. Remember that if there are links that have to be fully concurrent we must use the APIs in non-blocking manner. In those cases we are forced to have at least one buffered port on that link, because we need to use either Full() or Empty(). Moreover, when we use buffered ports we always have at least 1 additional concurrent link added by the call to `TransferNB`, therefore, with buffered ports we can only use APIs in a non-blocking way.

POP INPUT & PUSH OUTPUT			
ID	CODE	BLOCKING	COMMENTS

1A	`indata = inport.Pop();`
	`outdata = func(indata);`
	`outport.Push(outdata);`
		B	

1B	`indata = inport.Pop();`
	`outdata = func(indata);`
	`while (!outport.PushNB(outdata)) {}`
		B	

1C			Data loss!

1D	`if(!outport.Full()) {`
	`  indata = inport.Pop();`
	`  outdata = func(indata);`
	`  outport.Push(outdata);`
	`}`
		B	

2A	`while (!inport.PopNB(indata)) {}`
	`outdata = func(indata);`
	`outport.Push(outdata);`
		B	

2B	`while (!inport.PopNB(indata)) {}`
	`outdata = func(indata);`
	`while (!outport.PushNB(outdata)) {}`
		B	

2C			Data loss!

2D	`if(!outport.Full()) {`
	` while(!inport.PopNB(indata)) {}`
	` outdata = func(indata);`
	` ouport.Push(outdata);`
	`}`
		B	

3A	`if(inport.PopNB(indata)) {`
	`  outdata = func(indata);`
	`  outport.Push(outdata);`
	`}`
		B	

3B	`if(inport.PopNB(indata)) {`
	`  outdata = func(indata);`
	`  while(!outport.PushNB(outdata)) {}`
	`}`
		B	

3C			Data loss!

Fig. 6. All possible pop-push links using *Connections* API (part 1).

Figures 6 and 7 show all the possible in-out links, by combining all the access types for input and output ports presented above. The missing entries would not function correctly, for example by losing data. The three colored entries are the only 3 possibilities for fully non-blocking links.

A few comments on the code snippets:

- *Pop* or *Push* in the APIs above are not necessarily done on an I/O interface. They can be done as well on some internal C++ components like a FIFO that have an LI-like interface, but are not `sc_modules`.
- `func()` can be any function, even a function scheduled with multiple pipeline stages.
- Every `{...}` code block can be filled with some computation or communication as needed.

3D	if(!outport.Full()) {
	  if(inport.PopNB(indata)) {
	  outdata = func(indata);
	  outport.Push(outdata);
	  }
	}
		NB	This is TransferNB() in InBuffered() Preferred if pop from input interface and push to internal component.

4A	if(!inport.Empty()) {
	indata = inport.Pop();
	outdata = func(indata);
	outport.Push(outdata);
	}
		B	

4B	if(!inport.Empty()) {
	indata = inport.Pop();
	outdata = func(indata);
	while(!outport.PushNB(outdata)) {}
	}
		B	

4C	if(!inport.Empty()) {
	  indata = inport.Peek();
	  outdata = func(indata);
	  if(outport.PushNB(outdata))
	  inport.IncrHead();
	}
		NB	This is TransferNB() in OutBuffered(). Preferred if pop from internal component and push to output interface.

4D	if(!inport.Empty() && !outport.Full()) {
	  indata = inport.Pop();
	  outdata = func(indata);
	  outport.Push(outdata);
	}
		NB	If both pop and push are done on I/O interfaces 3D and 4C are preferrable, because they create a combinational ready-valid only on 1 interface, not both.

Fig. 7. All possible pop-push links using *Connections* API (part 2).

4.2 Blocking behavior

When the ports are unbuffered (`In` and `Out`), it is only possible to have pop-push links with a blocking behavior. These are only useful for very simple use cases, where it is only required to have at most one pop-push link working at the same time. More complex modules with parallel pop-push links require a non-blocking behavior.

In Figure 5, 6, 7, all the options with a `PopNB` or `PushNB` in a while loop likely lead to incorrect or suboptimal designs. All the other options available for non-blocking links are reliable and perform well.

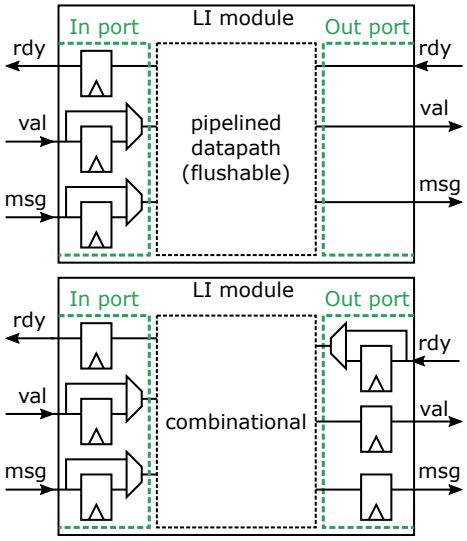

Fig. 8. Ideal LI modules.

4.3 Non-blocking behavior

As already explained, there are only three possible fully non-blocking uses of the API, the ones highlighted in green in Figure 6. We also mentioned that `InBuffered` and `OutBuffered` ports are required to be able to use `Empty()`, `Full()` and `Peek()` on the interface of an `sc_module`.

The following items try to describe what are the always-correct ways of using the ports and API of the current *Connections* library. On the other hand we also provide insight on what is the impact of using buffered ports rather than unbuffered ones.

First of all, it is relevant to understand what an ideal (i.e. minimal) latency-insensitive module would look like. A LI module does not have combinational paths through it. Figure 8 shows a module with one input and one output port and some function at its core. There will always be an input bypass register for the inputs and if the core is combinational or non-flushable an non by-passable register for the outputs. Therefore a minimal LI module would have a latency of 1 clock cycle, throughput of 1 and storage capacity of 2. This consideration shows that having input buffered port of depth 1 is not an overhead, it's actually the minimum for the module to function properly. In fact, when using the regular `In` and `Out` ports, either Catapult infers and input bypass register or the module cannot work with full throughput. For the output port instead, an `OutBuffered` port with depth of one is not an overhead only if the core of the module is combinational. Whenever the core of the module is pipelined, ideally a buffered output port could be merged by Catapult with the last pipeline stage. However as of now we cannot count of this, because it is hard to predict and it only works with very simple modules.

One current problem with *Connections* is a redundant layer of registers that Catapult adds at the output interface. This is probably the reason why `Empty`, `Full` and `Peek` don't work properly. Figure 9 illustrates the extra set of redundant registers.

To overcome this problem, the `OutBuffered` port used in *Connections* is by-passable. As a results the designs are functionally correct, but the area is not optimal because of the double buffering at the output. Although there is a

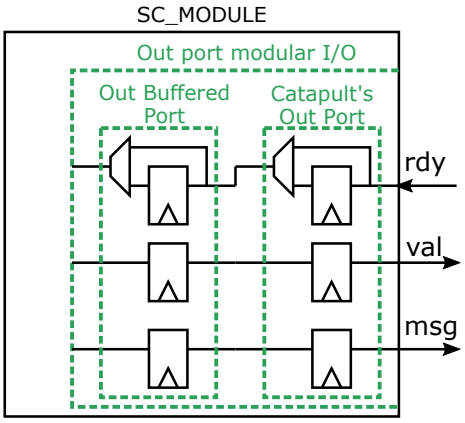

Fig. 9. Simple schematic of redundant registers added by Catapult at the output interface.

solution in the works, this document considers the current state of *Connections*, which means that we still want the `OutBuffered` port to be by-passable.

Here are some tips on buffered ports depth:

- Buffered port with depth 1 almost always lead to throughput of 0.5. Therefore, it is safer to avoid using them. Theoretically, there are cases where simply adding an input bypass register (InBuffered of depth 1) should enable full throughput. That is the case of every pipeline module, with flushable pipeline and at least 2 pipeline stages, that is the core function of the module should not be combinational. Instead, output ports of depth 1 are expected to lead to suboptimal throughput.
- The safest design choice seems to be using either an InBuffered port of depth 2 paired with a regular Out port or a pair of In and OutBuffered with depth of 2.

Finally, we describe how to use the `TransferNB()` API function properly and optimally, as an improper usage can easily lead to HLS failure, incorrect design or suboptimal performance.

Listing 12 shows the implementation of both the input and output `TransferNB()`.

```
1
2  // InBuffered port
3
4  void TransferNB() {
5
6    if (!fifo.isFull()) {
7
8      Message msg;
9
10     if (this->PopNB(msg))
11
12       fifo.push(msg);
13
14   }
15
16 }
```

```
17
18
19
20 bool TransferNB_p1() {
21
22   return !fifo.isFull();
23
24 }
25
26
27
28 void TransferNB_p2(bool is_not_full) {
29
30   if (is_not_full) {
31
32     Message msg;
33
34     if (this->PopNB(msg))
35
36       fifo.push(msg);
37
38   }
39
40 }
41
42
43
44 // OutBuffered port
45
46 void TransferNB() {
47
48   if (!fifo.isEmpty()) {
49
50     Message msg = fifo.peek();
51
52       if (this->PushNB(msg))
53
54         fifo.pop();
55
56   }
57
58 }
```

Listing 12. Input and output `TransferNB()` implementation.

Both implementations are composed of two parts. First they check the status of the FIFO in the port. Then if there is space to push into it (or there is data to pop from it) they proceed with the pop-push operations. `TrasferNB()` operated between the I/O interface and the FIFO in the buffered port. The criticality lies in what is happens on the other end of the FIFO. On the other side, similarly the core process checks the state of the FIFO and potentially enqueue or dequeues elements. Therefore at every clock cycles there are potentially four operations acting on the FIFO: two checks on its state, a potential enqueue and a potential dequeue.

Listing 13 we show three correct implementations of a module with one input port and one output port, with an internal process that simply pops from the input port and immediately pushes into the output port. These implementations have the three possible port setups: `InBuffered - Out, In - OutBuffered, InBuffered - OutBuffered`. The functions with `p1` and `p2` subfix are simply the `TransferNB()` divided in two parts. This split is needed to ensure an always correct implementation.

```
1
2  // InBuffered - Out
3
4  bool is_not_full = inport->TransferNB_p1();
5
6
7
8  if (!inport->Empty()) {
9
10   indata = inport->Peek();
11
12   outdata = func(indata);
13
14
15
16   if(outport->PushNB(outdata))
17
18     inport->IncrHead();
19
20 }
21
22
23
24 inport->TransferNB_p2(is_not_full);
25
26
27
28 // In - OutBuffered
29
30 inport->TransferNB();
31
32
33
34 if (!inport->Empty() && !outport->Full()) {
35
36   indata = inport->Pop();
37
38   outdata = func(indata);
39
40   outport->Push(outdata);
41
42 }
43
44
45
46 outport->TransferNB();
47
```

```
48
49
50 // InBuffered - OutBuffered
51
52 bool is_outport_not_full = !outport->Full();
53
54 outport->TransferNB();
55
56
57
58 if (is_outport_not_full) {
59
60   if (inport->PopNB(indata)) {
61
62     outdata = func(indata);
63
64     outport->Push(outdata);
65
66   }
67
68 }
```

Listing 13. Input and output `TransferNB()` implementation.

REFERENCES

Brucek Khailany, Evgeni Khmer, Rangharajan Venkatesan, Jason Clemons, Joel S. Emer, Matthew Fojtik, Alicia Klinefelter, Michael Pellauer, Nathaniel Pinckney, Yakun Sophia Shao, Shreesha Srinath, Christopher Torng, Sam (Likun) Xi, Yanqing Zhang, and Brian Zimmer. 2018. A Modular Digital VLSI Flow for High-productivity SoC Design. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. ACM, New York, NY, USA, Article 72, 6 pages. https://doi.org/10.1145/3195970.3199846

Mentor. 2017. *HLS Bluebook*. Mentor.