Title

OpenSMART: Single-Cycle Multi-hop NoC Generator in BSV and Chisel



Authors

Kwon, Hyoukjun

Krishna, Tushar

Publication Date

2017

# *OpenSMART*: Single-Cycle Multi-hop NoC Generator in BSV and Chisel

Hyoukjun Kwon
School of Computer Science
Georgia Institute of Technology
hyoukjun@gatech.edu

Tushar Krishna
School of Electrical and Computer Engineering
Georgia Institute of Technology
tushar@ece.gatech.edu

***Abstract*—The chip industry faces two key challenges today – the impending end of Moore's Law and the rising costs of chip design and verification (millions of dollars today). Heterogeneous IPs - cores and domain-specific accelerators - are a promising answer to the first challenge, enabling performance and energy benefits no longer provided by technology scaling. IP-reuse with plug-and-play designs can help with the second challenge, amortizing NRE costs tremendously. A key requirement in a heterogeneous IP-based plug-and-play SoC environment is an interconnection fabric to connect these IPs together. This fabric needs to be scalable - low latency, low energy and low area - and yet be flexible/parametrizable for use across designs. The key scalability challenge in any Network-on-Chip (NoC) today is that the latency increases proportional to the number of hops.**

**In this work, we present a NoC generator called *OpenSMART*, which generates low-latency NoCs based on SMART[1]. SMART is a recently proposed NoC microarchitecture that enables multi-hop on-chip traversals within a single cycle, removing the dependence of latency on hops. SMART leverages wire delay of the underlying repeated wires, and augments each router with the ability to request and setup bypass paths. *OpenSMART* takes SMART from a NoC optimization to a design methodology for SoCs, enabling users to generate verified RTL for a class of user-specified network configurations, such as network size, topology, routing algorithm, number of VCs/buffers, router pipeline stages, and so on. *OpenSMART* also provides the ability to generate any heterogeneous topology with low and high-radix routers and optimized single-stage pipelines, leveraging fast logic delays in technology nodes today. *OpenSMART* v1.0 comes with both Bluespec System Verilog and Chisel implementations, and this paper also presents a case study of our experiences with both languages. *OpenSMART* is available for download[2] and is going to be a key addition to the emerging open-source hardware movement, providing a glue for interconnecting existing and emerging IPs .**

## I. Introduction

Networks-on-Chip (NoC) is a key component of almost all chips today. The domains vary from (i) many-core chips in HPC supercomputers and high-end servers with tens to hundreds of homogeneous cores [1], [2], [3], to (ii) mobile and embedded SoCs with tens of heterogeneous cores and controllers [4], [5], to (iii) GPUs with hundreds of SMs [6], to (iv) domain-specific accelerators, such as machine learning, with hundreds of processing elements [7], [8]. Without loss of generality, we refer to end point cores, accelerators, PEs, caches, etc. as "IPs" in this work. NoC is the interconnect backbone connecting IPs communicating each other and a critical IP block itself for plug-and-play designs. As the number of IPs in a hardware system increases, the communication fabric also needs to scale so that it does not become a performance or energy bottleneck.

Fundamentally, the latency of traversal between two IPs is proportional to the number of hops between them. Routers at each hop help to manage the multiplexing of different flows on the shared output links, but they add arbitration and switch delay to every message. This problem becomes worse as the number of IPs on a chip goes up; the number of hops each message takes to get from one end of the chip to the other goes up proportionally as well, increasing latency. Latency can have a direct impact on performance, as it affects the number of cycles the source core may have to stall while waiting for a response. This is especially a challenge in mobile SoCs where latency requirements are often much more stringent. Moreover, since heterogeneous IPs optimized for a certain operation are placed at design-time, the number of hops between communicating nodes cannot be reduced by thread/process migration. Apart from latency, NoC energy is another key challenge as systems scale, since the energy cost of data movement across a chip is often an order of magnitude more than the cost of computation [9].

The latency and energy of on-chip communication could be reduced by simpler routers or high-radix routers [10] that reduce the number of hops by adding dedicated links between distant nodes. Both of the design strategies come with their performance, area and energy trade-offs, which require careful design-space exploration to gauge performance benefits against overheads. With chip design already costing millions of dollars today, designing and verifying NoCs for every new architecture to consider various trade-offs aggravates the design cost problem.

NoC RTL generators [2], [11], [12] can ease the process of design-space exploration and verification. These generators parameterize router modules and links, and generate different topologies and microarchitectures that can be simulated for performance, and synthesized for area and power estimates. The key challenge with most open-source NoC generators today is that they rely on multi-stage textbook router implementations [13], which scale horribly in terms of latency and

[1]Single-cycle Multi-hop Asynchronous Repeated Traversal
[2]http://synergy.ece.gatech.edu/tools/OpenSMART/

energy as hop counts go up. Most of these generators also provide a *specific* pipeline implementation. There have been a lot of optimizations over the past decade of NoC research to reduce network delays by dynamic pipeline adjustments [14], [15], [16], but few of these have gone beyond hand optimized demonstrations into a parameterized tool flow.

A promising design optimization for scalable many-IP NoCs is SMART [17]. SMART leverages the fact that wires are fast enough to transmit signals 10+ mm within 1ns in process technologies today, and in future. The limiter to network latency today is the the conventional design philosophy of latching flits at every hop. SMART provides the performance of low-diameter high-radix topologies, without actually adding additional dedicated datapaths, by enabling flits to traverse multiple hops within a single cycle, up to the distance that the underlying wire can physically allow (known as maximum hops per cycle or $\text{HPC}_{max}$). This saves not only latency, but also energy since intermediate clocked latches are bypassed completely. More details about SMART are presented in Section II. As technology nodes shrink, and high-end cores get augmented with smaller dedicated accelerator IPs, the size of IP blocks is expected to go down. Thus, the same wire delay - which does not scale down with technology - can translate to higher $\text{HPC}_{max}$, making SMART even more attractive.

This work presents *OpenSMART*, an automated tool for generating SMART NoCs, hiding microarchitectural details of multi-hop path request, setup, and bypass from the designer. *OpenSMART* provides user-configurability and generates synthesizeable Verilog that can be plugged into any SoC. In addition, *OpenSMART* can also generate single and multi-cycle routers for any regular or irregular (heterogeneous) topology. Our experiments with the Nangate15nm open-cell library [18] demonstrate that our generated SMART NoCs provide 35% latency reduction and 39% EDP reduction over a 1-cycle optimized mesh router with random traffic.

To contribute to the emerging open-source hardware ecosystem [19], we release the source code of *OpenSMART* under BSD license. *OpenSMART* v1.0 has both a Bluespec System Verilog (BSV) [20] and Chisel [21] implementation, which provide higher-level abstractions of hardware. The implementation provides easier modularization (like object-oriented programming languages) and abundant libraries for frequently-used hardware logic. Such features will help *OpenSMART* users easily modify the source code for their specific purpose. Moreover, BSV and Chisel framework supports both C++ simulation and Verilog generation. This enables design-space exploration like software tools and also timing/area/power estimation though any ASIC/FPGA tool flow. We also present a case study demonstrating our experience with using both these HDLs and the optimizations afforded by each.

The rest of the paper is organized as follows: Section 2 introduces previous research related to the NoC generators and high-level HDLs. Section 3 describes the microarchitecture design of *OpenSMART* and presents the characteristics of BSV and Chisel implementations. Section 4 discusses the evaluation results of *OpenSMART* network. Section 5 concludes.

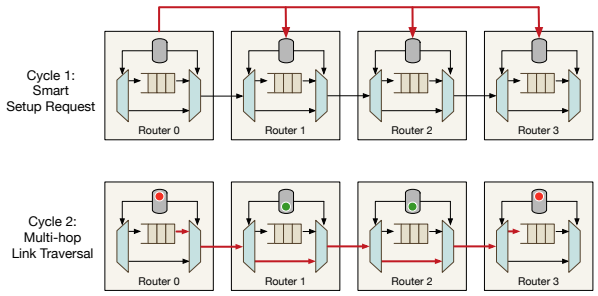


Fig. 1: An example of single-cycle multi-hop traversal in SMART.

## II. Background and Related Work

**SMART NoC.** SMART [17] is a single-cycle multi-hop traversal network design that reduces average flit hop counts in mesh-based networks. SMART requires two stages for flit traversal; one for setting up a multi-hop path, and then next for the multi-hop link traversal. Figure 1 shows an example of a flit in router 0 traversing till router 3 in one cycle. After winning local arbitration in Cycle 0, the flit at router 0 sends a smart setup request (SSR) in Cycle 1 to the intermediate routers, router 1 and 2, via dedicated control wires (per direction), indicating an intent to bypass. The intermediate routers arbitrate among the received SSRs by using a simple policy called Prio=Local that prioritizes local (buffered) flits over bypass requests. Router 0 sends its flit in Cycle 2. If there are no local requests for the same output port at routers 1 and 2, the incoming flit is directly sent to the output link without getting latched, all the way till router 3. If any of the intermediate routers had a locally buffered flit, the flit from router 0 would have stopped at that router, prioritizing the local flit to use the output link instead.

At low-loads, SMART enables most flits to bypass all routers. At high-loads, it performs like a conventional design with hop by hop traversal.

*$\text{HPC}_{max}$.* The maximum number of bypass hops, or maximum hops-per-cycle ($\text{HPC}_{max}$), is a design-time parameter, constrained by the clock period of system, tile size, and the wire delay of data links between routers.

In this work, we implement SMART_1D [17] that only allows multi-hop bypass along a dimension, not at turns, since that reduces the number of SSR wires. In addition, we implement the Prio=Local arbitration policy between SSRs, which has been shown to be higher performing and more fair than prioritizing bypass flits over local flits [17].

**NoC Generators.** Table I contrasts some existing NoC generators with *OpenSMART* . Flexnoc [22] is a commercial NoC generator by Arteris which generates a customized topology for each SoC, but is not open. Connect [12] is a NoC generator optimized for FPGAs. Connect uses Bluespec System Verilog (BSV) for the implementation and generates synthesizable Verilog of the network design specified by user parameters. Only the verilog is available for users, not the BSV source code. It supports a web-based graphical user interface so that users can obtain network designs with various topologies easily. Open SoC Fabric [11] provides an NoC

| | Connect [12] | OpenSoC Fabric [11] | OpenPiton [2] | OpenSMART (this work) |
|---|---|---|---|---|
| Language | Verilog from BSV | Chisel | Verilog | BSV and Chisel |
| Topology | Arbitrary topologies | Mesh, Flattened butterfly | Mesh | Mesh, Arbitrary topologies |
| Flow control | VC, input/output-queued | VC | Wormhole + Priority | VC, SMART |
| Buffer Management | Credit, Peek-flow | Credit | Credit | Credit |
| Router Microarchitecture | 1-cycle | 4-cycle | 1-2 cycle | 1-cycle, 2-cycle, SMART |

TABLE I: Comparison of Open NoC generators.

generator written in Chisel. This generator supports 2-D mesh and flattened butterfly networks of arbitrary size. Open SoC Fabric discloses their source code, and users can freely edit the source code and re-compile it because Chisel is an open-source language. NoC System Generator [23] receives a specification of NoCs in XML file format and produces VHDL codes which satisfy the specification. It only supports 2D or 3D mesh topology thus the available network configuration is limited. OpenPiton [2] is an open-source manycore system generator, and it has presented fabricated ASIC chips as well as FPGA implementation that runs full-stack Linux. OpenPiton contains a NoC structure to support the cache coherence, memory, and inter-core interrupt traffic of the SPARC cores it employs. The NoC does not have virtual channels but ensures deadlock freedom using separate physical networks.

**High-level HDLs.** The design effort challenge from complex designs and non-intuitive semantics of traditional HDLs has inspired research in high-level HDLs. The advantage of these HDLs is the ability to do design-space exploration via C++ simulation like software simulators, but also generate actual Verilog to pass through an ASIC or FPGA flow. BSV [20] supports System Verilog style module interface and type systems. BSV adopts the concept of guarded atomic actions [24] to describe behavior inside hardware modules. The guarantee of the atomicity efficiently reduces design efforts by increasing the granularity of parallelism. BSV generates C++ source code for software simulations, and synthesizable System C and Verilog codes. Chisel [21] is based on Scala [25]. Scala is an object-oriented and functional language thus it provides high-level features based on Scala. A recent opensource project, the RISC-V processor [26], is implementing using this language. Lava [27] attempted to design hardware in Haskell [28] which is one of the major functional language for software. It contains high-level features of functional languages such as polymorphism and high-order functions. Such features enable more abstract and general descriptions of hardware. ArchHDL [29] is a high-level HDL built upon C++. It models registers as variables and wires as lambda functions using new features introduced in C++'11. As registers store elements and wires carry some values in Verilog, this style facilitates Verilog-like design. MYHDL [30] is a Python-based high-level HDL. It generates synthesizable Verilog or VHDL source codes. Both ArchHDL and MYHDL claim to provide orders of magnitude faster simulation time than native Verilog simulation.

In this work, we pick BSV and Chisel as target languages for implementing *OpenSMART* and provide characteristics of generated RTL from each version.

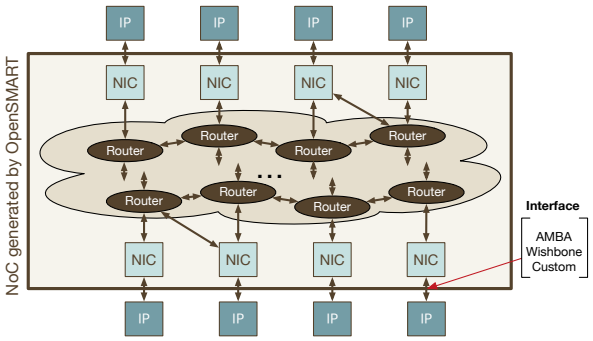


Fig. 2: *OpenSMART* overview. *OpenSMART* is an N×N switch implemented as a NoC connecting N IPs (cores/accelerators/other compute or memory units).

```
#Network
Num_Nodes 16
Topology_File Mesh.dot
Routing_algorithm XY
Flow_Control VC
Link_Width 128

#Router
Pipeline_Stages 1
Num_VCs 4
VC_Depth 1
```

(a) Configuration File

```
//Mesh.dot
graph mesh16 {
  r0--r1--r2--r3;
  r4--r5--r6--r7;
  r8--r9--r10--r11;
  r12--r13--r14--r15;
  r0--r4--r8--r12;
  r1--r4--r8--r13;
  r2--r6--r10--r14;
  r3--r7--r11--r15;
}
```

(b) Topology File

| Topology | Mesh | Arbitrary |
|---|---|---|
| Routing | Source, XY, YX | Source, Spanning Tree |
| Flow Control | VC, VC+SMART | VC |
| Buffer Management | Credit | Credit |
| Router Microarchitecture | 1-cycle, 2-cycle, SMART | 1-cycle, 2-cycle |

(c) Configuration Parameters

Fig. 3: An example of *OpenSMART* configuration file, topology file and configuration parameters.

## III. The OpenSMART NoC Generator

### A. Overview

The goal of *OpenSMART* is to generate a soft IP of an N×N *switch* that can be plugged into an SoC with N IPs, as shown in Figure 2. The underlying implementation of this switch is a NoC. The design parameters of the NoC are taken as an input configuration file, and the RTL of the NoC is outputted in BSV, Chisel, and Verilog[3].

**External Interface.** The default interface of input/output port of the switch to each IP is the following: <*message_class, message_payload*>, padded with <*ready, enable*> signals for

[3]The Verilog is generated from the BSV version, thus not easy to read.

managing flow control. Messages across different message classes are guaranteed to be non-blocking. Messages within a message class may block each other. In addition, we also provide wrappers for AMBA and Wishbone interfaces. All IPs connect to a unique network interface (NIC) in our design.

**Network Configuration.** The *OpenSMART* tool takes the number of nodes, topology, routing algorithm, flow control, router pipeline delay, and router configuration as user inputs as part of a configuration file. Figure 3 shows an example config file and the options we currently support. Arbitrary topologies (including a Mesh) can be specified in DOT [31]. A $N \times N$ crossbar, for instance, can be generated by simply declaring one router in the topology file.

### *B. Library of Building Blocks*

We define a library of modular building blocks that our tool uses to implement various router microarchitectures.

**Network Interface.** Network interfaces break incoming message packets into multiple flits (flow control unit - basic unit of operation in each router).

*Flit.* The size of each flit is equal to the bandwidth of each link in the topology. A flit contains the virtual channel (VC) ID (**vc**), its type (**flitType**), which indicates whether the flit is the head, the body, or the tail flit of a message, route information (**routeInfo**), which contains the next output port and the number of remaining hops in each direction, data (**flitData**), which contains information carried by the flit, and a statistic for benchmarks (**stat**). The vc and routeInfo is only carried by the head flit. The statistics field is used for debug and verification in the BSV/Chisel version, and the compiler automatically removes the field when *OpenSMART* synthesizes the network designs into Verilog code.

**Arbiter.** *OpenSMART* implements both round-robin and matrix arbiters. In an N:1 matrix arbiter, N one-bit registers are used to encode priorities among the requesters and are updated upon each grant. We found matrix arbiters providing better performance, without noticeable critical path or area overhead compared to round-robin for mesh routers with 5-ports, and less than eight VCs per port. The arbiters are used in both the input and output units, to implement *separable switch allocation* [32] as described below.

**Input Unit.** The Input unit contains virtual channel buffers and an input VC arbiter as Figure 4(a) illustrates.

*Input Buffers.* For each VC, we use a separate set of registers for the routeInfo (i.e., output port) - that is fanned out/in to the arbiters - and a FIFO queue for the flit. This lets us perform reads of the routeInfo and flit, by the switch arbiters and crossbar respectively in parallel to reduce the critical path.

*Input VC Arbiter.* The input VC arbiter selects one VC as a winner among the flits at that port. A flit arriving at an empty input port automatically wins the input VC arbitration, without having to wait an additional cycle. Thus the Input unit abstracts the arbitration process from the rest of the router, and outputs the output port request and data for the flit that is ready to be sent out. This flit proceeds for output port arbitration.

**Output Unit.** The Output unit contains an output port arbiter, and a VC selector, as Figure 4(b) illustrates.

*Output Port Arbiter.* An arbiter at each output port arbitrates among requests from multiple input ports. The grant from the output port is used to trigger VC selection and set the select lines for the crossbar muxes.

*Virtual Channel Selector. OpenSMART* implements an extremely light-weight queue-based dynamic VC selection [33]. At each output port, a queue tracks the free VCs at the next router, and generates a *hasVC* signal if it is non-empty. The head of the queue is stored in a separate register called *nextVC*. When a flit wins the switch and is being sent out, it replaces its VC field with the *nextVC* register value and the dynamic VC queue pops the *nextVC*. The signals, *nextVC* and *hasVC*, are decoupled so that they have no dependences.

This design removes the need for a separate VC allocation stage like other NoC router generators [11], [12], [23], and is is appropriate for bypass flits in SMART that need to perform VC selection while bypassing multiple hops within a single-cycle (as explained later in Section III-C).

When a tail flit leaves a router, the router sends the free VC ID in a *credit signal* to its upstream router, which pushes it into its free VC queue. The pushed VC is available in the next cycle to prevent possible combinational loops.

**Routing Unit.** *OpenSMART* supports two dimension-order routing (DOR) algorithms - XY or YX - for a mesh topology, and a source-routing algorithm for irregular topologies in heterogeneous SoCs. These routing algorithms are selectable in the user configuration.

The XY/YX modules are combinational logic implementations and perform **lookahead routing**, i.e., the routing is performed one-hop in advance, enabling switch allocation (for this router) and routing (for the next router) to occur in parallel [13]. For SMART routers, we encode the $x\_hops$ and $y\_hops$ as one-hot values, so that route computation just involves a 1-bit right-shift at every hop during a single-cycle multi-hop traversal.

In source-routing, the source NI embeds the entire route as a set of turns, where each turn at a router is a $log_2(num\_ports)$-bit value that uniquely identifies an output port. The source-routing logic simply right-shifts the route by $log_2(num\_ports)$-bits at each hop. To avoid routing deadlocks in arbitrary irregular topologies, we provide support for spaning-tree based non-minimal routing [34] by encoding the source routing such that all flits route via a root node [34].

**Crossbar Switch.** The crossbar implements the fundamental switching functionality of routers, forwarding flits from input ports to their designated output ports. The crossbar is implemented using demultiplexers and multiplexers, that are driven by the the grant signals from the output port arbiters.

**SMART Unit.** The SMART unit is instantiated by SMART routers and adds functionality for a single-cycle multi-hop traversal over the baseline router functionality. It comprises of a SMART Setup Request (SSR) Generator, SSR Links, and a SMART Arbiter, as shown in Figure 4(c).

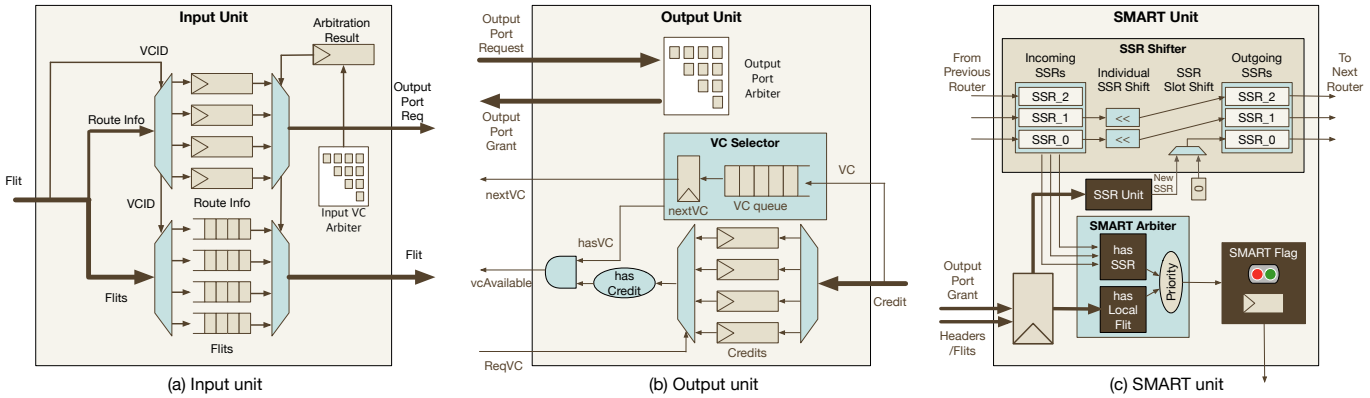


Fig. 4: The building blocks of *OpenSMART* routers.

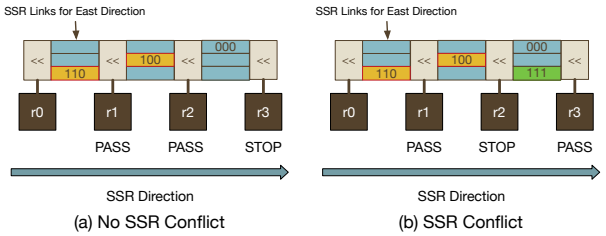


Fig. 5: An example of SSR propagation with HPC$_{max}$= 4. The highlighted SSRs are active SSRs. In both cases r0 sends SSR=110 to bypass r1 and r2. In case (a) it is successful. In case (b), r2 sends its own SSR, so r2 sets its SMART flag to stop.

*SSR Unit.* SSR units, one per output port, generate SSR signals for every winner of output port arbitration. We implement SSRs as (HPC$_{max}$-1)-bit signals to represent all the routers each flit requests to bypass along the current direction (X or Y, in XY routing). The number of bypasses is the min($remaining\_hops\_in\_dimension$, HPC$_{max}$)- 1.

*SSR Links.* The SSR signals use dedicated control links which span from each router (SSR sender) up to HPC$_{max}$-1 receiver routers, in all four directions; the routers HPC$_{max}$ hops away from the SSR sender always latch the flit from the sender. The SSR signal needs to traverse HPC$_{max}$-1 hops within one cycle. Each SSR link is HPC$_{max}$-1 bits wide[4].

At any cross-section, there are HPC$_{max}$-1 SSR links, carrying SSRs from senders 1-hop to HPC$_{max}$-1 hops away, as shown in Figure 5. At each hop, all SSRs shift up by one slot; this removes the furthest SSR (which has reached HPC$_{max}$-1 hops) and the bottom slot is occupied by the SSR from that router. In addition to this shift, the SSR signals on all links shift left by 1-bit to decrease remaining bypass hops. *Thus the MSB of any SSR at a router indicates its intent to request a bypass at that router or not.*

*SMART Arbiter.* The SMART arbiter reads the MSB from all SSR signals entering it; if any of these bits, it indicates a bypass request. SMART arbiters set the SMART flag to bypass only if it (a) receives a bypass request, (b) the next router has a free VC, and (c) no local flit is requesting the same output port as the bypass flit that sent the SSR. If a local flit also requests the output port, SMART arbiter prioritizes the local flit over a bypass flit. This policy implements the *prio=local* of SMART [16]. There is one SMART arbiter for every IO pair per dimension (i.e., W→E, E→W, N→S, and S→N).

[4]We chose to implement SSRs as (HPC$_{max}$-1)-bit signals instead of log$_2$(HPC$_{max}$) like the original SMART design [16], to remove a decoder from the SSR arbiter and correspondingly increase HPC$_{max}$.

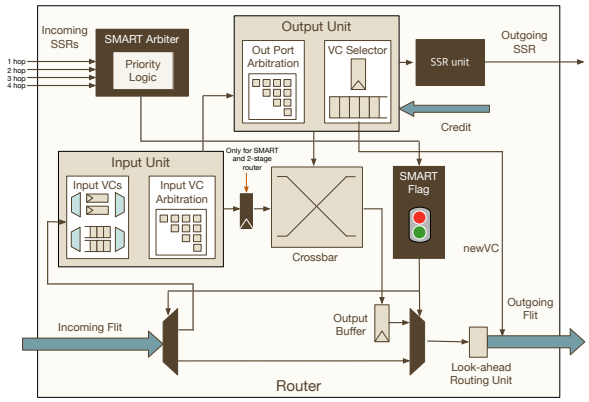


Fig. 6: *OpenSMART* Router Microarchitecture built using library modules. Modules in light/dark color represent the default 1-stage router and additional modules for SMART, respectively.

## C. Router Microarchitectures

Using the library of modules described above, myriad router microarchitectures can be generated by *OpenSMART*, as shown in Figure 6. The Input unit and Output unit together create a **separable switch allocator**. We place the Routing unit after the crossbar switch in all our designs to update the route in outgoing flit as it did not increase our critical path, and helps when instantiating a SMART router. It can also be placed in the Input unit for every incoming flit [13].

**2-Stage Router.** The two-stage pipelined router separates the two major critical paths in a router - switch allocation (i.e., Input unit → Output unit) and switch traversal (i.e., Crossbar switch) using a pipeline latch. This design can increase the clock frequency, but may also increase flit latencies because of the extra pipeline stage. We envision using this design for high-radix routers, where the crossbar traversal can take

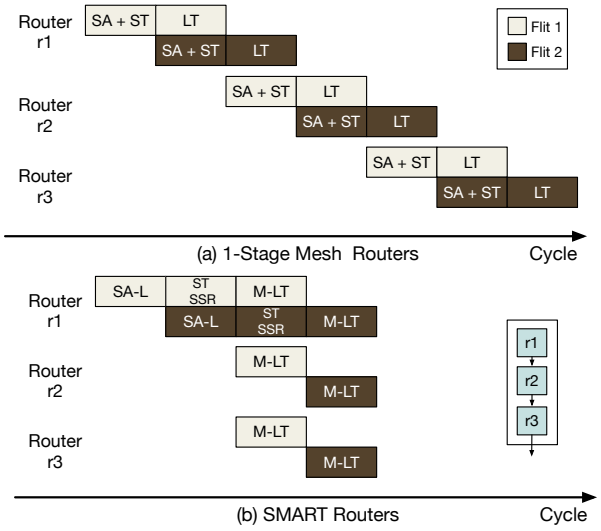


Fig. 7: Pipeline stages in an *OpenSMART* SMART router. SA-L, SSR, ST, and M-LT indicate output port arbitration, SSR communication, switch traversal, and multi-hop link traversal.

significant delay, or for higher clock rates in mesh designs. At no contention, flits take 3-cycles per-hop (2-cycle router + 1-cycle link) with this design.

**1-Stage Router.** The single-stage router requires only one cycle to pass the entire router logic (Input unit → Output unit → Crossbar switch → Routing unit) unless there is congestion. *This is the most optimized version of the router pipeline.* At no contention, flits take 2-cycles per-hop (1-cycle router + 1-cycle link) with this design as shown in Figure 7.

**SMART Router.** The SMART router implements a 2-stage router pipeline, followed by a 1-cycle traversal across multiple links. The SMART design only works over a mesh topology, as it performs multi-hop bypasses along a dimension. The microarchitecture is shown in Figure 6. The first-stage is switch allocation (Input unit → Output unit). The second-stage is switch traversal (Crossbar switch) and routing. In parallel, the SSR generator sends out SSRs up to HPCmax hops and the SMART arbiter at each intermediate router sets the SMART flag, as described earlier in Section III-B.

*Single-cycle Multi-hop Traversal.* In the next-cycle, the flit performs a multi-link traversal along the X or Y dimension. We do not allow bypasses at turns. During the single-cycle multi-link traversal, the VCid in the flit is updated at every hop by replacing with *nextVC* register at that router, as described in Section III-B[5]. The router sends a credit back for the VCid the flit came with. Similarly, the routeInfo is updated at every hop by a 1-bit right shift in the $x\_hops/y\_hops$ fields since the SMART router uses XY routing.

The *SMART flag* at every router, set at the end of the previous cycle by the SSR arbiters, implicitly determines whether the flit continues to bypass or stops, without the flit having to do anything actively. A multi-hop traversal terminates either at the turning router, or the destination router, or at an intermediate router $\text{HPC}_{max}$ hops away, or prematurely at an intermediate router if has contention for the same output port by a higher priority (local) flit.

At no contention, flits take 3-cycles per-dimension (2-cycle router + 1-cycle multi-link traversal) with this design as shown in Figure 7. In the worst case (contention at every router), this design takes 3-cycles per-hop.

The critical path for SMART can be one of the following: local switch allocation (i.e., Input unit → Output unit), SSR multi-hop traversal and arbitration, or flit multi-hop traversal. We pick the critical period based on switch allocation, and then choose the highest $\text{HPC}_{max}$ such that both SSR traversal and flit traversal meet timing (through an iterative process).

### D. BSV vs. Chisel Implementations

BSV and Chisel offer two alternate paradigms for describing hardware and we describe our experience with using both for this work. We present a case study of the matrix arbiter implementation describing its interface and logic.

```
interface MatrixArbiter#(numeric type numReq);
  method ActionValue#(Bit#(numReq))
    getArbit(Bit#(numReq) reqBit);
endinterface
```

Code 1: Matrix Arbiter interface in BSV

```
class MatrixArbiter(numReq: Int)
 extends Module {
   val io = new Bundle {
    val enable = UInt(width=1,dir=INPUT)
    val requests = UInt(width=numReq,dir=INPUT)
    val grants = UInt(width=numReq,dir=OUTPUT)
  }
```

Code 2: Matrix Arbiter interface in Chisel

**Interface.** Code 1 and Code 2 show the interface definition of a matrix arbiter in BSV and Chisel respectively. Both of the interfaces are parameterized with an integer **numReq** that represents the maximum number of arbitration requesters. The parameter defines the width of input ports, reqBit (BSV) and requests (Chisel), and output ports, the return value of method getArbit (BSV) and grants (Chisel).

Chisel requires users to manage the communication between modules explicitly. For instance, an **enable** control signal is used to represent the validity of the input **requests** and activate the arbitration logic accordingly.

BSV on the other hand uses a *method* which is a function that returns a value from a logic inside a module. The Bluespec compiler creates a hardware scheduler that generates **RDY** and **EN** signal for every method that indicates if the method can fire and will fire in the cycle respectively. For example, **RDY_getArbit** and **EN_getArbit** are generated for the example in Code 1. Although such implicit communication protocol may involve extra hardware logic, we found that it provides better abstractions of the intermodule communication compared to traditional HDLs, simplifying code.

[5]This design enables us to use the same VC selector library module for both the regular and SMART routers, unlike the original SMART design [16] which requires a special VC allocator at the destination router.

**Logic.** When a matrix arbiter grants a requester, the arbiter updates priority registers, which are one-bit registers that represent priority between requesters. Because arbitration logic needs to provide a total order of requesters for its functionality, the arbiter requires $n^2/2 - n$ priority registers, which form a triangular array of registers, and need to maintain their values in a way that ensures fair arbitration. For the maintenance of priority registers, when a matrix arbiter grants a requester, the arbiter resets the row that has the same row index as the winner to zero and sets the column that has the same column index as the winner. This logic is described in the code snippets in Code 3 and Code 4.

```
rule updatePriorityBits(hasRequester);
  let targetIdx = winnerIdx;
  /* 1. Clear the row */
  for(Integer j=0; j<numReq; j=j+1) begin
    priorityBits[targetIdx][j] <= 0;
  end

  /* 2. Set the column */
  for(Integer i=0; i<numReq; i=i+1) begin
    if(i != targetIdx) begin
      priorityBits[i][targetIdx] <= 1;
    end
  end
endrule
```

Code 3: Priority register update logic in BSV

```
when (io.enable === UInt(1)) {
  for (i <- 0 until n) {
    // when req(i) is granted,
    // set all p(x,i) to 1, and p(i,y) to 0.
    when (grants(i)) {
      for (j <- 0 until n) {
        if (j > i) {
          priority(j)(i) := Bool(true)
        } else if (j < i) {
          priority(i)(j) := Bool(false) }}}}}
```

Code 4: Priority register update logic in Chisel

Both languages provide a for loop that represents parallel value updates and reduces the lines of codes. The Chisel implementation describes all logic within the module body while BSV describes logic in *rules*, which represents a guarded atomic action [35] block that groups hardware logic and guarantees the atomic execution of the actions described in the block. The rules contain rule guards such as **hasRequester** in Code 3 and the scheduler we mentioned in the **interface** paragraph executes the rule only if the rule guard is true. This feature enables users to define the *behavior* of hardware they are implementing and the granularity of parallelism. Chisel in contrast allows users to define hardware explicitly.

Due to these key differences, we found that the BSV version of *OpenSMART* has much fewer lines of code than Chisel, but sometimes requires more hardware to implement the same functionality as we show later in our evaluations.

## IV. Evaluation

### A. Methodology

We tested the *OpenSMART* generated NoCs using testbenches that model external IPs and inject flits from every IP port to user-specified destinations at user-specified rates. We compiled the testbenches to generate software simulation executables using BSV and Chisel C++ simulation frameworks.

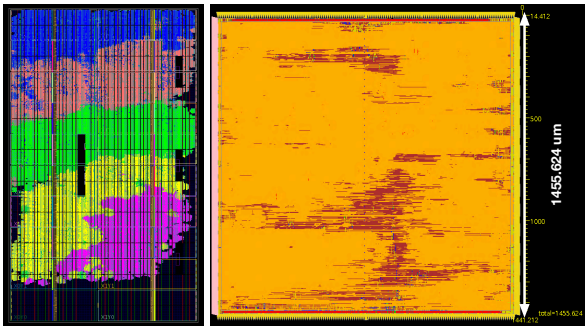


Fig. 8: The layout results of 5x5 network in FPGA (left) and ASIC (right). In FPGA layout, we represent each row of the mesh network in different color.

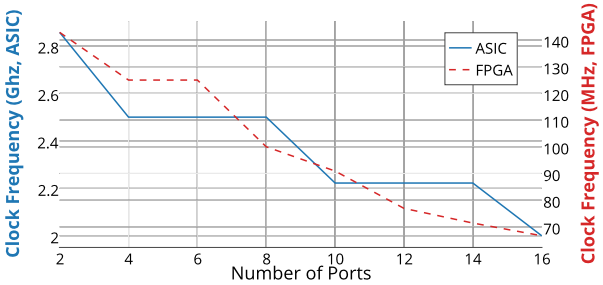


Fig. 9: The maximum clock frequency of a 2-stage router in ASIC and FPGA flow as the function of the number of ports

We also validated our design with hardware synthesis tools for ASIC and FPGA design flows. We use Synopsys Design Compiler and Cadence Encounter with the NanGate 15nm open cell library [18] for synthesis and place-route respectively for the ASIC flow, and Xilinx Vivado Design Suites with VC709 evaluation board for FPGA flow. The hardware synthesis tools not only show that the *OpenSMART* networks are synthesizeable for both ASIC and FPGA but also provide area, power, and timing closure information. Figure 8 shows the layouts of a 5×5 NoC using both flows.

We demonstrate the strength and flexibility of the *OpenSMART* framework by generating myriad NoC microarchitectures as case studies, and comparing them from a performance, timing, area and power perspective. Unless specified, we present results using the BSV version We compare the BSV and Chisel versions later in Section IV-D.

### B. Case Study I: Low-Radix vs. High-Radix Routers

A standard technique to reduce network delay and improve throughput is to introduce high-radix routers. At an extreme, a $N \times N$ crossbar provides a non-blocking connection between any pair of communicating IPs, at pure wire delay. Crossbars and heterogeneous routers with different number of ports are common in NoCs in many SoCs today [4], [5], [22]. We use *OpenSMART* to sweep and compare the hardware cost of

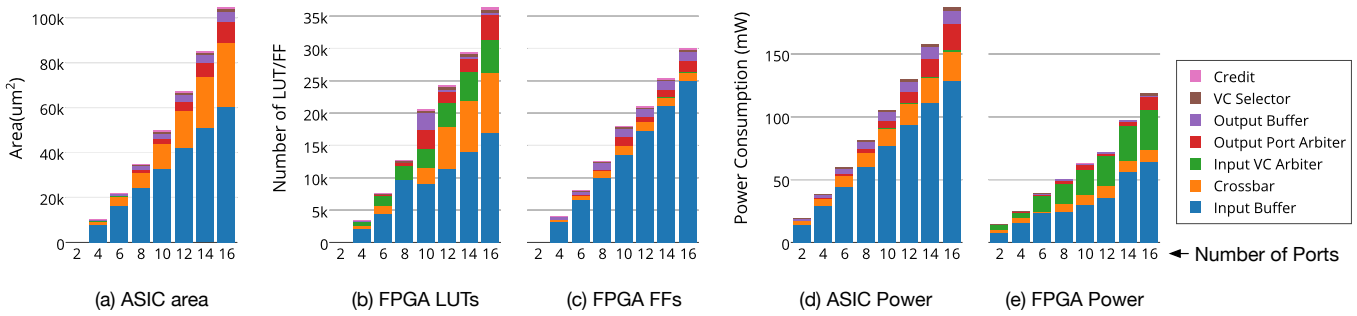


Fig. 10: Area and power breakdown of components in ASIC and FPGA design as the function of the number of ports

routers as a function of router radix. All designs have 128-bit links, 2-stage routers, and 4 VCs per port - each 1-flit deep.

**Timing.** Figure 9 plots the maximum achievable frequency of a 2-stage router on the 15nm ASIC flow and the VC709 FPGA as a function of increasing number of router ports. On an ASIC, a simple 2×2 switch can achieve close to 2.9GHz and a 5x5 mesh router can achieve 2.5GHz. We see the frequency dropping at 8, and 14 ports to 2.2 and 2GHz respectively[6]. It is important to note that the goal of the synthesis tools (both ASIC and FPGA) is to meet timing, which comes at the cost of larger cells and more buffers. Thus a flat frequency with increase in ports translate to a large area and power penalty as we show next. On the FPGA, we observe a somewhat linear drop from 140MHz to 70MHz as the number of ports increase.

**Area.** Figure 10(a) plots the area breakdown of the routers on an ASIC. A 5-port mesh router fits in a tiny area of 160um×160um. As the number of ports increase, we see the crossbar area increasing significantly, as expected with high-radix routers. Correspondingly, the output port arbiter size also increases from less than 1% at low radix to close to 10% at 14 and 16 ports. On the FPGA (Figure 10(b) and (c)) we observe most of the FFs going towards implementing the input buffers, and the LUTs implementing most of the crossbar.

**Power.** Figure 10(d) and (e) plots the power consumption of our generated routers at 1GHz on ASIC and 67 MHz on FPGA respectively. For the ASIC flow a 5-stage mesh router consumes about 50mW. We see a bulk of the power consumption occurring in the input buffers, consistent with prior studies [17], [33].

We see a similar trend with the FPGA, with the buffers consuming most of the power. These observations point to the attractiveness of SMART to potentially bypass buffering across multiple routers. On the FPGA, beyond six ports, the input arbiters start consuming noticeable power.

### C. Case Study II: Mesh vs. SMART

We compare a Mesh NoC with 1-cycle routers and a SMART NoC overlaid on this Mesh. We enable destination bypass [16] in the SMART NoC. We evaluate both NoCs under the following configuration: 8x8 network, 128 bit links, and 4 VCs per port.

**Performance.** We use the software simulation framework afforded by BSV to study the latency and throughput characteristics of the NoCs. The testbench injects 1-flit packets from each IP at increasing injection rates with user-specified traffic patterns. It also calculates the performance (throughput and latency) characteristics of the networks. Throughput is estimated by collecting the number of injected and received flits at each host IP port. Total delay is calculated by estimating the average end-to-end cycles from the ingress NIC, through the NoC, to the egress NI.

For comparison and validation purposes, we also implemented the same Mesh and SMART NoCs in Garnet [36] which is a pure software simulation framework, and plot the results from Garnet running with the same traffic patterns. Figure 11 plots our results for Uniform Random and Bit Complement traffic.

**Mesh vs. SMART**: Figure 11(a) and (b) show that the SMART NoC provides a 35% and 43% latency reduction at low-loads for uniform random and bit-complement traffic, respectively. As we show later, both the mesh and SMART NoCs meet timing with the same clock frequency; thus the latency plot (in cycles) actually represents absolute wall-clock time. Figure 11(c) and (d) demonstrate a 19% throughput improvement. While SMART by design is aimed at latency improvement, not throughput, a faster recycling of flits through the NoC reduces credit round-trip delay, which by Little's Law helps improve throughput. For bit-complement, the network throughput drops beyond saturation - this is a well-known behavior due to heavy congestion at the center of the network that back-pressures other routers and throttles injection[13].

***OpenSMART* vs. Garnet.** Synthetic traffic injectors in Garnet, like other software simulators, model infinite queues at the source to maintain injection at the user-specified rate without stalling. The queueing delay shows up as a latency spike when the network saturates. In *OpenSMART*, however, the injector stalls once the network cannot accept more packets, like a real IP. Thus for a fair comparison, we plot the latency within the network (source to destination NIC) from *OpenSMART* and Garnet. The curves are almost identical for low-load latencies. The throughput values differ slightly due to differences in implementations of the arbiters in software and BSV.

[6]We caution from using these exact timing numbers at face value since Nangate15nm is an open-cell library, not a commercial one, and thus probably uses optimistic assumptions in its cells with regards to the process.

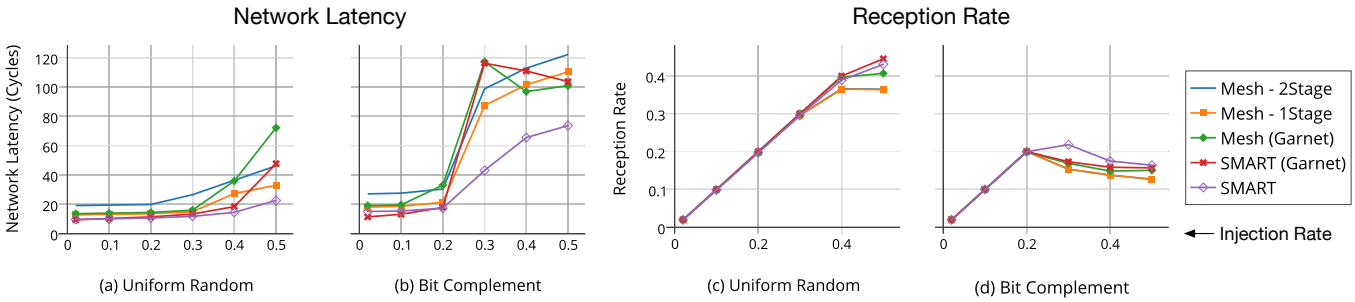


Fig. 11: Network latency and reception rate of *OpenSMART* network with uniform-random and bit-complement traffic.

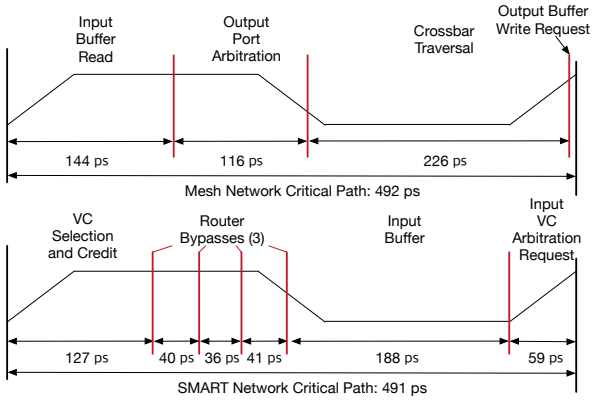


Fig. 12: Critical paths of *OpenSMART* mesh and SMART network

**Timing.** Figure 12 compares the critical paths of the Mesh with 1-stage routers and SMART NoCs in the ASIC flow. Both designs met post-synthesis timing at 2 GHz. The critical path for the mesh is essentially the 1-stage router. For SMART, the critical path within the router is shorter, since crossbar traversal occurs in the second stage, as described earlier in Section III-C. We found the critical path to be the multi-link traversal. At 2GHz, the router can perform three router bypasses (i.e., $\text{HPC}_{max}$) within a cycle.

The observed critical path of the SMART network has four key parts: (i) vc selection and credit generation at the starting router, (ii) intermediate router bypasses, (iii) buffer write at the final router (demuxing into the appropriate VC) and (iv) input VC arbitration. Step (iv) is an optimization that performs input VC arbitration right after link traversal and buffering, in the same cycle. This enables incoming flits to directly start output port arbitration at the beginning of the next cycle. This optimization helps lower the critical path inside the router for a 1-stage design, as Figure 12 shows: input buffer read, output port arbitration, and crossbar traversal are completed in 492 ps, since input VC arbitration was done the cycle before when the flit arrived and was getting latched. However, for SMART, this optimization can increase the critical path of the multi-hop link traversal stage, limiting $\text{HPC}_{max}$. We did not use separate pipeline partitions for Mesh and SMART in order to honor *OpenSMART*'s automated plug and play approach for building routers using the same set of library components. However, we are working on re-timing optimizations to push some of this delay to the next stage without affecting the target clock period, to enable SMART to get a larger $\text{HPC}_{max}$.

**Traversal Energy.** Figure 13(a) plots the energy of traversal for a flit as a function of hops in both a mesh and the SMART NoC. SMART assumes an $\text{HPC}_{max}$ of 3 (i.e., stop every 3 hops) and a successful bypass at the intermediate routers, paying buffering costs only at the $\text{HPC}_{max}$ boundaries. We see the energy benefits compared to a mesh increasing close to 2× as the number of hops go up.

**Achievable $\text{HPC}_{max}$.** We ran an entire k×k SMART NoC with increasing $k$ through synthesis and place-and-route as a function of clock frequency and observed the critical paths reported by the ASIC (and FPGA) tools for each run. Since the critical path in SMART is the multi-hop traversal, the critical path report showed the number of routers being bypassed before the signal is latched to meet timing. This number is nothing but the achievable $\text{HPC}_{max}$ (Figure 13(b) and (c)). In an ASIC, at 1GHz, we see an $\text{HPC}_{max}$ of 14 post-synthesis, which goes down to 7 post-layout due to wire delays. We believe this number can be optimized further by optimizing the multi-hop traversal critical path as described earlier. We also ran the $\text{HPC}_{max}$ study on the FPGA and observe it going down from 17 at 50MHz to 1 at 100MHz.

**Area and Power.** In the ASIC flow, SMART network increases the area and power consumption by 15% and 3%, respectively, compared to a mesh network with the same number of routers. However, the area and power overhead of SMART is much smaller than that of high-radix routers, which are alternatives to SMART to provide single-cycle traversal between distant routers. For example, a 5x5 flattened butterfly router requires nine ports and consumes almost double the power and area than a mesh router based on our results presented in Figure 10.

This case study shows that *OpenSMART* can enable researchers to use one framework to perform both design-space exploration and get real timing, area, and power numbers. This is unlike pure software simulators which can often end up modeling unrealistic hardware, and pure RTL models which limit design-space exploration.

### *D. BSV and Chisel Implementations*

Although both BSV and Chisel versions of *OpenSMART* implement the same functionality, the ASIC synthesis results

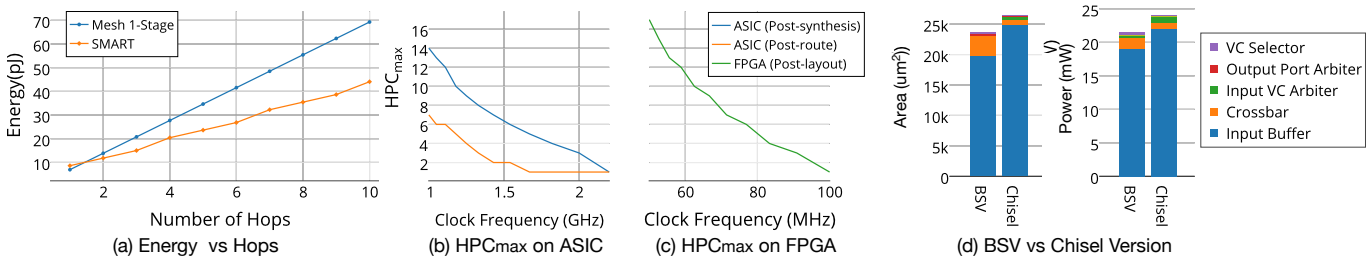


Fig. 13: *OpenSMART* Design Features

in Figure 13(d) show interesting differences in the area and power consumption for a 1-stage router. This difference is because of the compiler of each language as well as design choices. The compilers affected the area and power estimation results of input buffers. Input buffers have the same design in both implementations, four queues for each VC. The queue implementation in Chisel requires approximately 20% more area and power than that in BSV. In contrast, crossbar implementation in BSV requires approximately four times as much area as that in Chisel. This is because we implemented the BSV crossbar using concurrent registers, which is one of the BSV library modules. Concurrent registers employ a priority logic implemented with multiplexers that selects an input data to be written on the register from multiple inputs. Because output port arbiter ensures that at most one flit is granted each output port of the crossbar, we exploited the priority logic as a large multiplexer that automatically selects the incoming flit, thus simplifying the crossbar implementation. But this adds a dummy register for each output port. The Chisel crossbar, in contrast, consists of only multiplexers.

## V. Conclusion

This work presents *openSMART*, an open-source NoC generator in BSV and Chisel. A library of building blocks within a router and the NoC allow the design to support arbitrary topologies, routing schemes and router pipelines (2-stage, 1-stage, and SMART). Single-cycle multi-hop traversals using generated SMART NoCs demonstrate performance and energy benefits over single-cycle mesh routers.

## References


[1] G. Chrysos, "Intel® xeon phi coprocessor-the architecture," *Intel Whitepaper*, 2014.

[2] J. Balkind *et al.*, "Openpiton: An open source manycore research framework," in *ASPLOS*, 2016, pp. 217–232.

[3] D. Shaw *et al.*, "Anton 2: raising the bar for performance and programmability in a special-purpose molecular dynamics supercomputer," in *SC*, 2014, pp. 41–53.

[4] Qualcomm, "Snapdragon s4 processors: System on chip solutions for a new mobile age," https://www.qualcomm.com/documents/snapdragon-s4-processors-system-chip-solutions-new-mobile-age.

[5] Samsung, "Exynos5 white paper," http://www.samsung.com/semiconductor/minisite/Exynos/data/Enjoy_the_Ultimate_WQXGA_Solution_with_Exynos_5_Dual_WP.pdf.

[6] C. Wittenbrink *et al.*, "Fermi gf100 gpu architecture," *Micro*, vol. 31, no. 2, pp. 50–59, 2011.

[7] Y. Chen *et al.*, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," in *ISSCC*, 2016.

[8] T. Chen *et al.*, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014, pp. 269–284.

[9] S. Keckler *et al.*, "Gpus and the future of parallel computing," *Micro*, vol. 31, no. 5, pp. 7–17, 2011.

[10] J. Kim *et al.*, "Flattened butterfly topology for on-chip networks," in *Micro*, 2007, pp. 172–182.

[11] F. Fatollahi-Fard *et al.*, "OpenSoC Fabric: On-chip network generator," in *ISPASS*, 2016, pp. 194–203.

[12] M. Papamichael *et al.*, "Connect: re-examining conventional wisdom for designing nocs in the context of fpgas," in *FPGA*, 2012, pp. 37–46.

[13] N. Jerger *et al.*, "On-chip networks," *Synthesis Lectures on Computer Architecture*, vol. 4, no. 1, pp. 1–141, 2009.

[14] A. Kumar *et al.*, "Express virtual channels: towards the ideal interconnection fabric," in *ISCA*, 2007, pp. 150–161.

[15] H. Matsutani *et al.*, "Prediction router: Yet another low latency on-chip router architecture," in *HPCA*, 2009, pp. 367–378.

[16] T. Krishna *et al.*, "Breaking the on-chip latency barrier using smart," in *HPCA*, 2013, pp. 378–389.

[17] T. Krishna et al., "SMART: single-cycle multihop traversals over a shared network on chip," *Micro*, vol. 34, no. 3, pp. 43–56, 2014.

[18] M. Martins *et al.*, "Open cell library in 15nm freepdk technology," in *ISPD*, 2015, pp. 171–178.

[19] G. Gupta *et al.*, "Open-source hardware: Opportunities and challenges," *arXiv preprint arXiv:1606.01980*, 2016.

[20] R. Nikhil, "Bluespec system verilog: efficient, correct rtl from high level specifications," in *MEMOCODE*, 2004, pp. 69–70.

[21] J. Bachrach *et al.*, "Chisel: Constructing hardware in a scala embedded language," in *DAC*, 2012, pp. 1212–1221.

[22] Arteris, "Flexnoc," http://www.arteris.com/flexnoc.

[23] J. Chan *et al.*, "Nocgen: A template based reuse methodology for networks on chip architecture," in *VLSI*, 2004, pp. 717–720.

[24] M. Pellauer *et al.*, "Synthesis of synchronous assertions with guarded atomic actions," in *MEMOCODE*, 2005.

[25] M. Odersky *et al.*, "The scala language specification," http://www.scala-lang.org/docu/files/ScalaReference.pdf, 2004.

[26] A. Waterman *et al.*, "The risc-v instruction set manual, volume i: Base user-level isa," *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 2011.

[27] P. Bjesse *et al.*, "Lava: hardware design in haskell," in *ICFP*, 1998, pp. 174–184.

[28] "Haskell language," https://www.haskell.org.

[29] S. Sato *et al.*, "Archhdl: a new hardware description language for high-speed architectural evaluation," in *MCSoC*, 2013, pp. 107–112.

[30] J. Decaluwe, "Myhdl: a python-based hardware description language," *Linux journal*, vol. 2004, no. 127, p. 5, 2004.

[31] E. Gansner *et al.*, "Drawing graphs with dot," http://www.graphviz.org/Documentation/dotguide.pdf, 2006.

[32] L. Peh *et al.*, "A delay model and speculative architecture for pipelined routers," in *HPCA*, 2001, pp. 255–266.

[33] S. Park *et al.*, "Approaching the theoretical limits of a mesh noc with a 16-node chip prototype in 45nm soi," in *DAC*, 2012.

[34] M. Schroeder *et al.*, "Autonet: A high-speed, self-configuring local area network using point-to-point links," *JSAC*, vol. 9, no. 8, 1991.

[35] D. Rosenband *et al.*, "Hardware synthesis from guarded atomic actions with performance specifications," in *ICCAD*, 2005, pp. 784–791.

[36] N. Agarwal *et al.*, "Garnet: A detailed on-chip network model inside a full-system simulator," in *ISPASS*. IEEE, 2009, pp. 33–42.