50,000,000,000 Instructions Per Second : Design and Implementation of a 256-Core BrainFuck Computer

Sang-Woo Jun

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
wjun@csail.mit.edu

Abstract

This paper describes the design and implementation of a machine for servicing a large number of BrainFuck execution instances. The design includes the BrainFuck manycore processor and the software daemon that manages the execution instances on the processor. The BrainFuck manycore processor is a 256 core processor implementing a slightly modified version of BrainFuck as its ISA. Each core is implemented with a two stage pipeline with a carefully designed ISA encoding operating on disjoint memory spaces. Each core achieves over 0.8 instructions per cycle. We have implemented our design with an x86 host server and a Xilix Vertex 7 FPGA, and achieved an agglomerate performance over 50 billion instructions per second, which is light years ahead of what any single general purpose processor is capable of achieving while executing BrainFuck code. The BrainFuck computer is an attractive solution for servicing high throughput BrainFuck cloud services, both in terms of performance and cost.

1. Introduction

As single core performance scaling levels out and power consumption of computing systems become a first order concern, the idea of using many simpler, or "wimpy", machines instead of a few powerful, or "brawny", ones is enjoying popularity. Many modern workloads, especially datacenter workloads are deemed fit for such a cluster of wimpy nodes, either because they are easily parallelized, or because most of the work is spent waiting for peripherals such as storage instead of intense computation. A large amount of research is focusing on what kind of wimpy machines are best fit for important workloads. In this paper, we present a rather extreme example of a wimpy processor, using the BrainFuck [22] esoteric programming language as its ISA.

Copyright © ACM [to be supplied]. . . $15.00
DOI: http://dx.doi.org/10.1145/(to come)

BrainFuck [22] is a Turing-complete, minimalistic and esoteric programming language with only eight commands. Due to its extreme simplicity, it is one of the most popular esoteric programming languages. Despite its simplicity, its Turing completeness assures that it can be used to perform any kind of computation that a programmable computer can. BrainFuck programs are usually executed using interpreters running on a general purpose computer. Because of the simplicity of the language, using a general purpose machine exclusively to run BrainFuck is not the best use of its resources.

In this paper, we describe the design and implementation of a BrainFuck computer, which includes a 256-core BrainFuck processor. We provide a reference design including a carefully designed ISA encoding and a high-performance two-stage pipelined microarchitecture. We also demonstrate the validity of the design and its performance using a prototype implementation on a Xilinx Vertex 7 FPGA coupled with a host x86 server. This work provides a valuable insight into an extreme instance of a wimpy manycore computer.

The rest of the paper is organized as follows: In Section 2 we present some related existing work regards to low-power manycore architectures and the BrainFuck language. In Section 3 we describe the detailed architecture of our system. In Section 4 we present some relevant details regarding our implementation of the architecture. In Section 5 we evaluate the performance of various parts of the implemented system. We conclude and suggest future work in Section 6.

2. Related Work

The BrainFuck [22] programming language is a minimalistic programming language with only eight commands. It is proven to be Turing complete, meaning it can be used to perform any kind of computation a programmable computer can. A BrainFuck program execution consists of a list of instructions, an instruction pointer, data memory, and a data pointer. The eight instructions are described in Table 1:

A great amount of developer effort was put into making use of the BrainFuck language, including various kind of interpreters, including online enviornments [9, 10, 21], IDEs for visual editing and debugging [4–7], variations of the language with various extensions including procedures, strings and thread forking [2, 12, 23], languages with syntax using Orangutal words [11] or fish words [3] or pixels [13], compilers to different languages including C# [18], and C [16], compilers from high level languages to BrainFuck, such as a subset of C to BrainFuck compiler [1, 14]. There has been various attempts to build dedicated computing machines to run Brain-

Fuck [8, 17]. One of the biggest performance limiting features of a dedicated BrainFuck computer is the structured branch instructions, [and]. Unless the program is pre-processed to discover matching brackets before execution, the computer must stop every time a new branch is encountered to scan the program and find the matching bracket. This problem can be alleviated by storing the locations of matching brackets nested in the bracket being scanned for. But even with such optimizations in place, high performance is hard to achieve without pre-processing the program.

Due to the proliferation of massively scalable distributed processing platforms such as Hadoop [24], there has been an increased interest in using multiple weak, or "wimpy", computing nodes instead of fewer powerful, or "brawny" nodes [15]. For easily parallelizable workloads, a machine with numerous simpler compute nodes often have power/performance benefits compared to a smaller set of fast and complicated cores. As Moore's law continues to scale and provide more silicon resources in a chip, while the power law prohibits much faster and bigger cores to be built, devices such as General Purpose Graphic Processing Units (GPGPUs) which implement an immense number of very simple processing units on a single die are becoming an integral processing component. Implementing numerous simple cores on a reconfigurable fabric also has academic benefits, as they can be used to research processor infrastructure of future manycore processors [20].

3. Architecture

3.1 Architecture Overview

Figure 1 describes the overall architecture of the BrainFuck computer. The system consists of 256 cores coupled with an on-chip network access point, networked into a ring topology. All but one of these cores are BrainFuck cores, and node 0 is an virtual core implemented as a software daemon running on a host machine. Node 0 is a special core, as it can manage the execution of the other cores, including loading software, and routing input and output to and from the outside world. Each BrainFuck core has access to its private instruction and data memory space. Instruction memory is a list of 4-bit instructions, while the data memory is a list of 8-bit memory cells.

Ideally, the daemon should also be running on one of the BrainFuck cores, removing the need for a host machine and effectively becoming an Operating System for the machine. For now, we are using a x86 server to run the software daemon. Until an OS can be implemented in BrainFuck, the software daemon can also run on a simpler processor such as an ARM or a Microblaze to further lower operation cost.

3.2 Instruction Encoding

Our implementation of BrainFuck includes a carefully designed ISA encoding for the BrainFuck language, in order to simplify processor design and achieve high performance. Each instruction is encoded as a 4-bit word. Three of the four bits (bit 0, bit 2, bit 3) of a instruction word acts as flags, and one bit (bit 1) acts as the argument. The ISA encoding is described in Figure 2.

Command	Description
$\rangle$	Increment data pointer by one
$\langle$	Decrement data pointer by one
+	Increment data at the data pointer by one
-	Decrement data at the data pointer by one
[	If data at the data pointer is zero, jump to the matching]. Continue to the next instruction otherwise
]	If data at the data pointer is nonzero, jump to the matching [. Continue to the next instruction otherwise
.	Output data at the data pointer
,	Take input and store data at the data pointer

Table 1. List of BrainFuck instructions

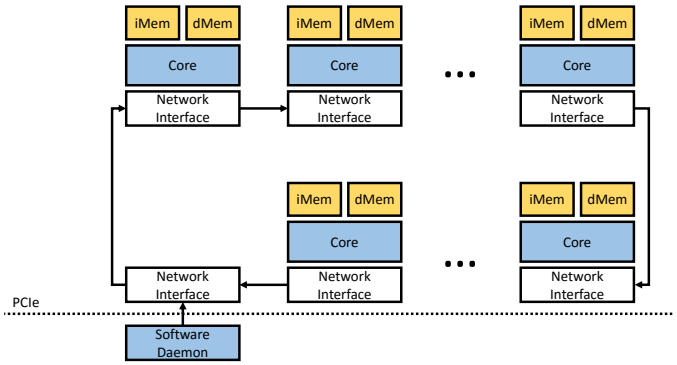

Figure 1. Architecture overview

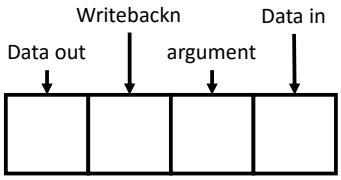

Figure 2. Instruction encoding

Each of the flags have the following meaning

- **DataOut** : Sends the data value under the data pointer to the data output queue.
- **WriteBackn** : Instruction does not involve writing back to memory.
- **DataIn** : Reads a byte of data from the data input queue.

Data bytes pushed into the data output queue is sent over the on-chip network to core 0, which can tag the data with its source node route the data to the outside world. Data input queue is populated by the on-chip network, with data received by node 0 from the outside world.

The argument bit determines the direction of each type of instruction. For example whether the data pointer is moved up($\rangle$) or down($\langle$), or if the data under the pointer is increased(+) or decreased(-), or whether the conditional branch instruction is a open bracket([) or closed bracket(]).

We had to make a slight modification to the conditional branch instructions in BrainFuck in order to achieve high performance. The conditional branch instructions [and] were changed to BZ and BNZ, which have the semantics of Branch if Zero and Branch if Not Zero, respectively. Because each instruction was encoded using only 4 bits, BZ and BNZ instructions were trailed by 4 more 4-bit cells encoding the 16 bits of address to jump to. This can be seen in Figure 3.

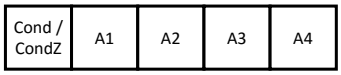

Figure 3. Branch instruction encoding

Using this design, each of the eight BrainFuck instructions were encoded into the following forms:

```
+ = 4'b0001
- = 4'b0011
, = 4'b0010
. = 4'b1100
< = 4'b0100
> = 4'b0110
[ = 4'b0101
] = 4'b0111
```

There is also one more command, "Stop", which is encoded as `4'b1111`, which indicates that program execution should stop there.

3.3 Microarchitecture

Figure 4 describes the details of the microarchitecture. A BrainFuck core is a 2-stage pipelined Harvard architecture, with separate instruction and data memories. There are only two stages, fetch and execute. Because data access in BrainFuck can only happen where the data pointer is pointing to, the fetch stage fetches both instruction and data memories.

The use of Harvard architecture was for design simplicity, especially since the instruction and data memories had different bit widths. (4 and 8, respectively) However, there is nothing prohibiting the use of a single memory space for both intruction and data memories.

An epoch register was used to tag each instruction and data reads, so that when a branch instruction (changing instruction pointer, [and]), or a data pointer instruction (¡ and ¿) is processed, the epoch is increased and instructions in the pipeline tagged with a previous epoch is discarded.

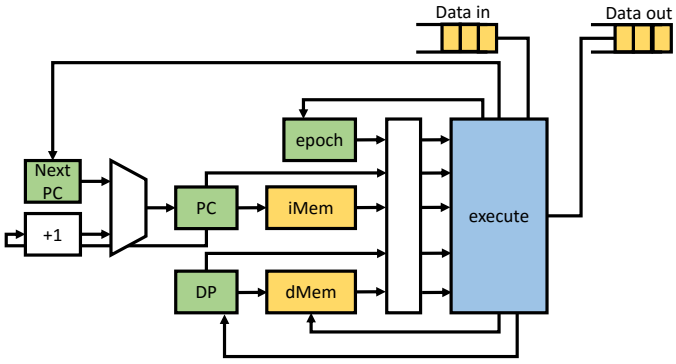

Figure 4. Microarchitecture of a BrainFuck core

Data forwarding is also implemented, so that even after data modification instructions (+ and -) invalidate the data already read from the same address, the fetched instruction does not have to be invalidated, causing an epoch change. Each instruction stores the data pointer address and its value after the instruction operates on it, to a pair of registers. The forwarding operation is described in Figure 5. Such a simple forwarding mechanism is very effective in BrainFuck, because instructions can only operate on the address pointed to by the data pointer.

3.4 On-Chip Network

Our design of the BrainFuck processor implements a ring topology for its 256 cores. This decision was made in favor of simplicity of design, especially since the routing logic of a ring topology is very simple to implement. Future modification of the BrainFuck processor will most likely include more complex and effective topologies.

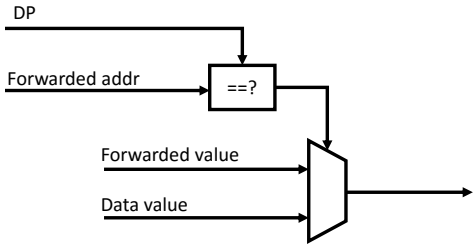

Figure 5. Data forwarding in the execution stage

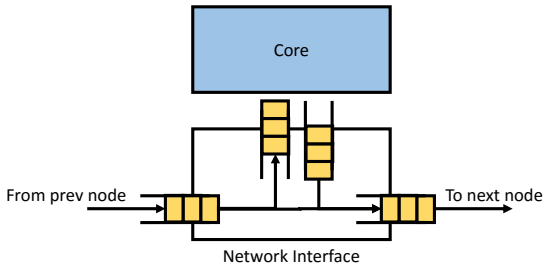

Figure 6. On-chip network node

Figure 6 describes the architecture of a network node. Each core is coupled with a network node to be incorporated into the on-chip network. The on-chip network is a packet-switched mesh network, in that each node is responsible of routing packets instead of having dedicated switches. A packet is 62 bits wide, and is composed of the following fields:

Field	Size(Bits)	Description
src	11	Source node idx
dst	11	Destination node idx
ptype	8	Packet type
data	32	Payload

Table 2. On-chip network packet structure

3.5 Core Zero

In our BrainFuck processor, the first core in the on-chip network, or core zero, is special, as it manages the execution of all other cores. Core zero is capable of sending special packets onto the on-chip network, targeted at specific nodes. In BrainFuck, the two commands `.` and `,` are used to communicate with the outside world. Core zero is responsible for relaying data in and out of the outside world for these commands. The following is the list of packet types that core zero can inject into the network.

- *Load program word* : Load program onto iMem
- *Init* : Clear program memory, reset instruction and data pointers
- *Start* : Start execution
- *Input Data* : Data that can be read with the `,` command

Core zero also receives all data out packets from all nodes and relays it into the outside world.

We implemented our core zero using a virtual core implemented in software of a host machine, in the form of a software daemon. The software daemon reads compiled programs from files, loads

it into idle cores and starts execution. It also reads input from a different set of files, or takes interactive input from a user. Output data is printed onto the screen.

Because our processor's ISA is different from raw ASCII representation of a BrainFuck program, we also implemented a rudimentary compiler that takes an ASCII encoded BrainFuck program and translates it into our ISA encoding.

4. Implementation

We implemented a prototype of our BrainFuck computer using a Xilinx VC707 FPGA development board coupled with am x86 host server. 256 BrainFuck cores were implemented on the FPGA, each running at 250MHz. The zero core was implemented as a software daemon. The software daemon and the network node of core zero communicated with each other over PCIe. The instruction and data memory were implemented as disjoint BRAM blocks on the FPGA. Figure 7 shows our development hardware platform.

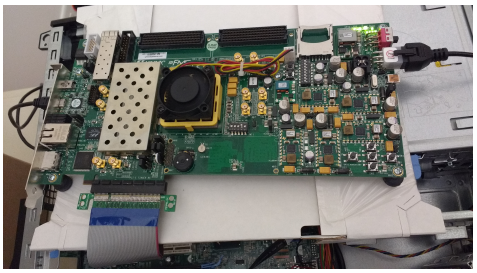

Figure 7. Development platform

The FPGA code development was done in the Bluespec hardware description language, and the software daemon was implemented in C++. The communication between them over PCIe was done using an open-source PCIe interface library called Bluespecpcie [19]. Bluespecpcie implements Gen 2 PCIe 8 lanes on the VC707 and abstracts it into the form of a FIFO. When high throughput is required, it also provides a memcpy-like DMA interface.

For example, the software program can send and receive 128-bit data to and from the hardware using the following syntax in C++.

```
Word d;
// Send
pcie->sendWord(d);

// Receive
pcie->recvWord(&d);
```

The hardware side can send and receive data to and from the C++ software using the following Bluespec syntax:

```
// Send
pcie.enq(d);

// Receive
let d <- pcie.first;
pcie.deq;
```

Using these features, the software daemon implements the following functions to manage the BrainFuck cores:

- *loadProgram(node, data, length)* : Loads a program string in *data* onto *node*.
- *init(node)* : Initializes *node*.
- *start(node)* : Start execution at *node*.
- *sendData(node, data)* : Send *data* to be read at *node*.

4.1 Resource Utilization

Table 3 shows the resource utilization of the BrainFuck processor on the Vertex 7 FPGA. It can be seen that we are already reaching the resource limitations of the FPGA, and 512 BrainFuck cores will probably not fit on the FPGA.

	Slice	LUTs as logic	BRAM
Available	75900	303600	1030
Used	55726	166654	391.5
	(73.42%)	(54.89%)	(38.01%)

Table 3. Vertex 7 Resource Utilization

5. Evaluation

We evaluated the performance of the BrainFuck cores using various microbenchmarks such as hello world, fibonacci and sqare number generation. We also benchmarked various iterations of the core design to show the benefits of different microarchitectural design modifications. We tested three different designs: (1) Naive, with no pipelining, (2) 2 Stage Pipeline, and (3) Data Forwarding, 2 Stage Pipeline with data forwarding. We observed the following Instructions per Second (IPC) with various designs of the Core:

Design	IPC
Naive	0.5
2 Stage Pipeline	0.6
Data Forwarding	0.84

Table 4. IPCs of various core designs

Figure 8 shows the same data put in a graph format with the bars going up in the right. 2Pipe plots the performance of a 2 stage pipeline processor, and Forwarding plots the performance of a processor with data forwarding.

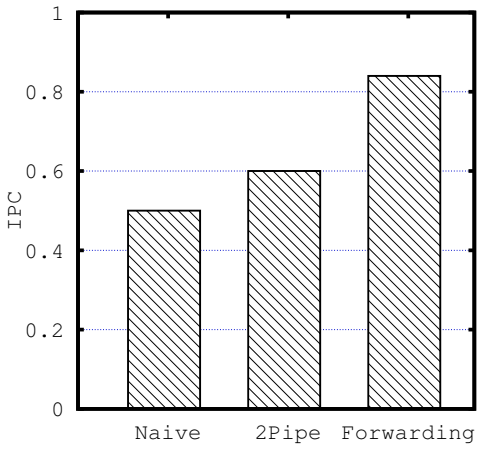

Figure 8. IPCs of varioud core designs

0.84 instructions per second on a core running at 250MHz, each core is capable of processing over 200M instructions per second. With 256 cores on the processor, the entire processor is capable of processing over 50,000M instructions per second.

5.1 Comparison Against General Purpose Hardware

The most high performance way of executing BrainFuck programs on a general purpose computing machine would be to translate it into native machine code. Considering each BrainFuck command on average takes 5 or more assembly instructions to implement, even assuming a perfect 1 instructions per second on a 3GHz processor, it would require almost one hundred cores to compete with this performance.

Figure 9 shows the performance of a BrainFuck core against a core of a 3GHz processor, *assuming a perfect 1 instruction per second on the general purpose processor*.

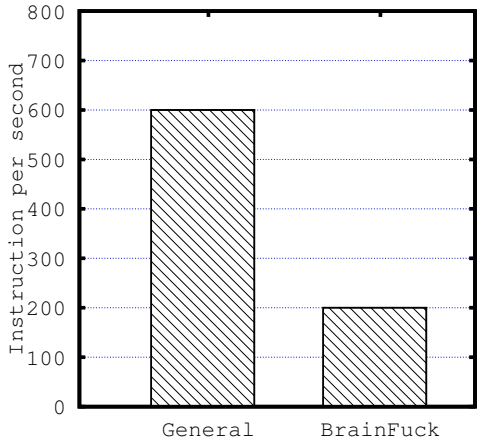

Figure 9. Performance of a BrainFuck core against a 3GHz processor

Figure 10 shows the total performance of the BrainFuck processor, compared against a 16-core 3GHz processor, *assuming a perfect 1 instruction per second on the general purpose processor*.

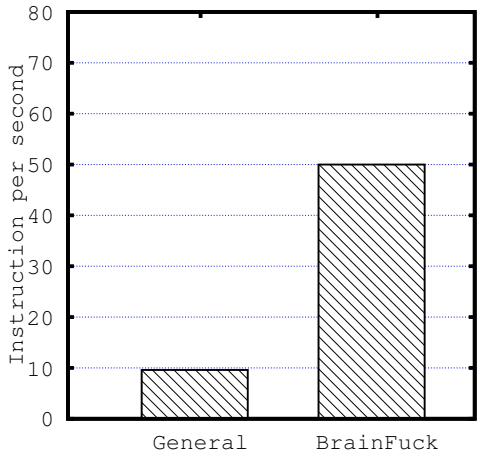

Figure 10. Performance of the BrainFuck processor against a 16-core processor

5.2 Power Consumption

A VC707 board is designed to consume an upwards of 30W under load. A modern Core i7 processor with 6 cores consume 80W of power under high load. Assuming a perfect 1 instruction per cycle on the i7, it would take 83 cores to achieve 50 billion instructions per second. 83 cores would consume over 1,000W of power. Compared to an FPGA coupled with a low-power host server, this is an immense amount of power consumption.

Figure 11 shows this difference. It should be noted that for the general purpose case, we have only considered the power consumption of the processor itself, and no other components of the computer. Whereas for the BrainFuck processor, we have done an end-to-end power measurement, which would include the hard disk, motherboard, fan and leds on the chassis.

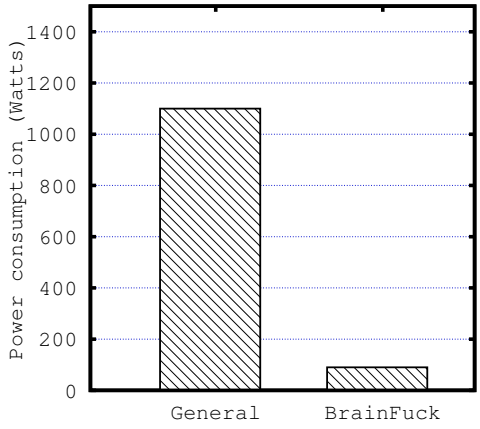

Figure 11. Power consumption of 50 billion instructions per second.

6. Conclusion and Future Work

In this paper, we described our design and implementation of a 256-core BrainFuck computer and demonstrated its performance. We think this work provides valuable insight into high-speed physical implementation of BrainFuck processors, and its future applications in the datacenter.

There are three areas of future work that we have under consideration:

- *Shared memory architecture between cores*, allowing us to explore cache coherence protocols and other memory system techniques optimized for the BrainFuck architecture
- *Alternate network topologies*, to improve the I/O latency of BrainFuck programs
- *Physical Core Zero*, implementing core Zero in the FPGA, so that the host server is no longer necessary. Core zero will connect to the internet by itself to communicate with users. Core Zero can be implemented using simple architecture such as Microblaze, or ideally, be just another BrainFuck core with specially mapped memory regions.

References

[1] *BrainFix, the language that translates to fluent Brainfuck*, 2013, (Accessed Feb 24, 2016). URL http://www.codeproject.com/Articles/558979/BrainFix-the-language-that-translates-to-fluent-Br.

[2] *Brainfork*, Accessed Feb 24, 2016. URL https://esolangs.org/wiki/Brainfork.

[3] *Blub*, Accessed Feb 24, 2016. URL https://esolangs.org/wiki/Blub.

[4] *Brainfuck Developer*, Accessed Feb 24, 2016. URL http://4mhz.de/bfdev.html.

[5] *Brainfuck Machine*, Accessed Feb 24, 2016. URL http://www.kacper.kwapisz.eu/index.php?i=19.

[6] *Visual Brainfuck*, Accessed Feb 24, 2016. URL http://kuashio.blogspot.com/2011/08/visual-brainfuck.html.

[7] *bfbuilder.el*, Accessed Feb 24, 2016. URL https://github.com/zk-phi/bfbuilder.

[8] *The Brainf*ck CPU Project*, Accessed Feb 24, 2016. URL http://www.clifford.at/bfcpu/bfcpu.html.

[9] *brainfuck interpreter*, Accessed Feb 24, 2016. URL http://esoteric.sange.fi/brainfuck/impl/interp/i.html.

[10] *brainfuck.tk*, Accessed Feb 24, 2016. URL http://brainfuck.tk/.

[11] *Ook!*, Accessed Feb 24, 2016. URL http://www.dangermouse.net/esoteric/ook.html.

[12] *RUM*, Accessed Feb 24, 2016. URL https://github.com/irskep/rum.

[13] *Brainloller*, Accessed Feb 24, 2016. URL https://esolangs.org/wiki/Brainloller.

[14] *C2BF*, Accessed Feb 24, 2016. URL https://github.com/benjojo/c2bf.

[15] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. Fawn: A fast array of wimpy nodes. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 1–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629577. URL http://doi.acm.org/10.1145/1629575.1629577.

[16] R. Carter. *Brainfuck to C translator*, 2011, (Accessed Feb 24, 2016). URL https://gist.github.com/Ricket/939687.

[17] G. Erdi. *A Brainfuck CPU in FPGA*, January 19, 2013 (Accessed Feb 24, 2016).

[18] M. Jankovi. *Brainf*ck Compiler*, 2011, (Accessed Feb 24, 2016). URL http://www.codeproject.com/Articles/283053/Brainf-ck-Compiler.

[19] S.-W. Jun. *BluespecPCIe*, 2016 (Accessed Feb 18, 2016). URL https://github.com/sangwoojun/bluespecpcie.

[20] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P.-Y. Droz. Ramp blue: A message-passing manycore system in fpgas. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 54–61, Aug 2007. doi: 10.1109/FPL.2007.4380625.

[21] O. Mazonka. *Fast Brainfuck interpreter bff4.c*, 2011 (Accessed Feb 24, 2016). URL http://mazonka.com/brainf/.

[22] U. Müller. Brainfuck–an eight-instruction turing-complete programming language. *Available at the Internet address http://en. wikipedia. org/wiki/Brainfuck*, 1993.

[23] P. M. Parks. *pBrain*, Accessed Feb 24, 2016. URL http://www.parkscomputing.com/applications/pbrain/.

[24] T. White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.