Extending CPU-less Parallel Execution of Lambda Calculus in Digital Logic with Arrays and Stacks

Authors: Jasmine Ritchie, Harry Fitchett, Charles Fox

Abstract

Moore's law has ended for clock speed, but continues for transistor density, motivating the search for new ways to use more transistors to compute more in parallel. GPUs have provided one approach, but must be programmed in specialist parallel programming styles. Programming would be easier if we could execute normal-looking programs with parallelization done automatically. Functional programming has been proposed as a basis for this due to its lack of state enabling execution in different orders. Previous work showed how to do this for a basic lambda calculus language, but is not practical for real world programming as many everyday constructs would have to be build up via slow recursive definitions. It is more usual to extend lambda calculus into larger languages with additional primitives to do common tasks such as arrays and stacks. We do this here and implement and test our design using LogiSim Evolution.

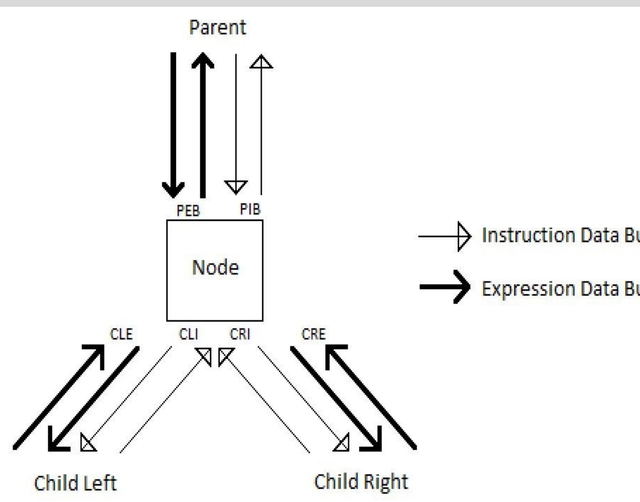


Figure 1 Abstract Visual Representation of a Node and its Connections

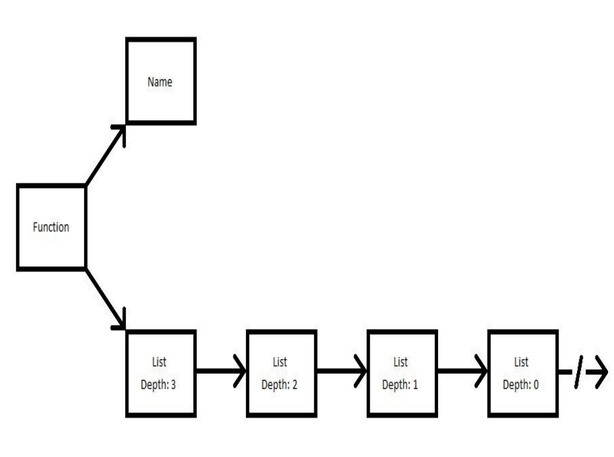


Figure 2 Abstract Visual Representation of List Chain compared to Name

Project Background

This project supports a corresponding paper titled “CPU-less Parallel Execution of Lambda Calculus in Digital Logic”. that introduces a new computational model inspired by lambda calculus, adopting a functional rather than processor-centric approach. Instead of relying on a single powerful processor, this model distributes computation across many small, simple processors called nodes.

In this framework, programs are divided into expressions, with each node representing one expression. Expressions can take different forms such as names, functions, or applications and while single node cannot compute anything on its own, when connected through a set of routing patterns, the aggregate is able to reduce programs.

Every node connects to three others: a parent, a left child, and a right child. Because of this branching structure, programs are typically represented as tree-like graphs. Each node has three registers: an expression register, which the expression type, and two child pointers, which hold the location of its left and right children. Nodes also communicate with one another through shared buses, exchanging two main types of information: expression and instruction data. Expression data includes the contents of a node’s registers along with flags that help determine when computation is complete. Instruction data, by contrast, consists of a binary key that identifies a particular instruction, along with the ID of the node the instruction targets (when applicable).

This project builds on that work by extending the expression set to include a common and practical data type in hardware, lists.

Update Depth Instruction

The `UpdateDepth` instruction tells a node at a specific depth to update its internally stored value. Similar to changing an element in a list using its index. In the first line, each node passes the instruction it received from its parent (via the PIB) down to its child (via the CRI), causing every node within the list torceive this instruction. Next, each node checks if the instructions target depth matches its own depth. If it does, that node updates its value (stored in the CLP) using the new value provided by the PEB.

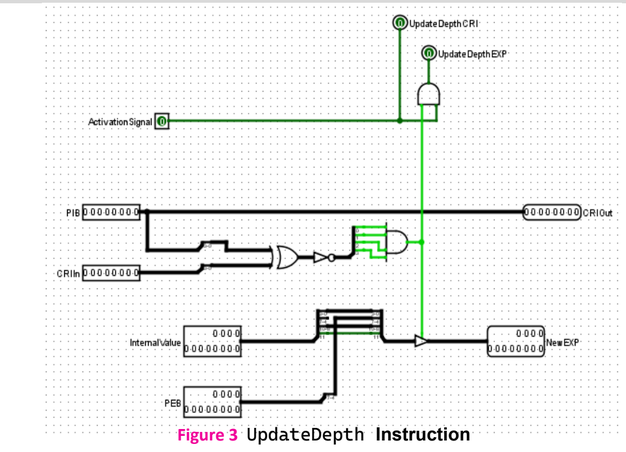


Figure 3 `UpdateDepth` **Instruction**

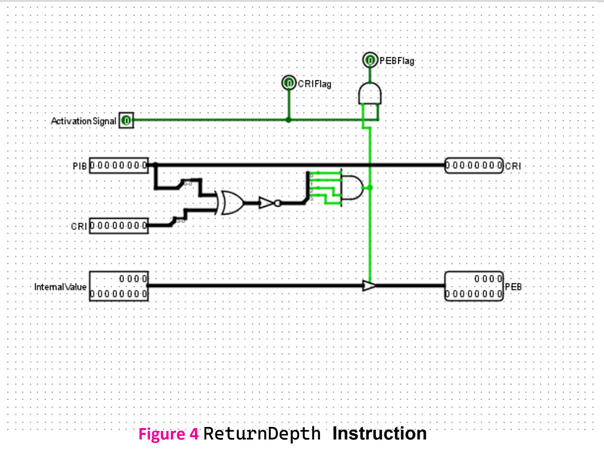


Figure 4 `ReturnDepth` **Instruction**

Remove Bottom Node Instruction

The `RemoveBottomNode` instruction removes the bottom node from a list, taking multiple clock pulses to complete this instruction targets a node of depth one. During the first clock pulse, each node, other than the target, passes this instruction downwards, ensuring that every node in the list receives it. The target begins removal by sending a `Nullification` instruction to its child. During the second clock pulse, non-targeted nodes continue to pass the instruction and the target sets its CRP to zero, removing its child from the list.

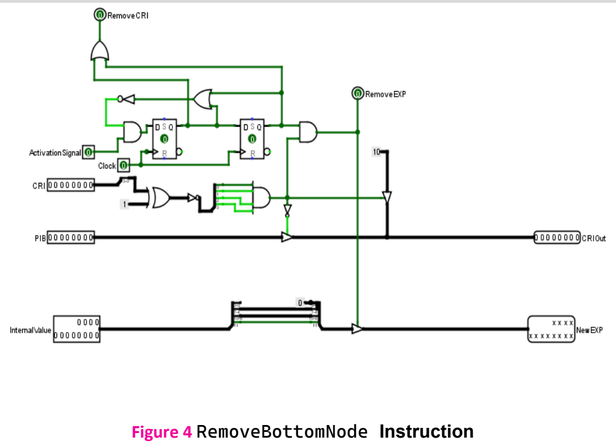


Figure 4 `RemoveBottomNode` **Instruction**

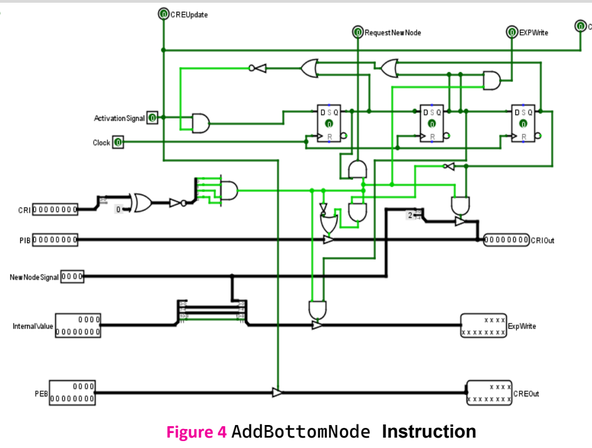


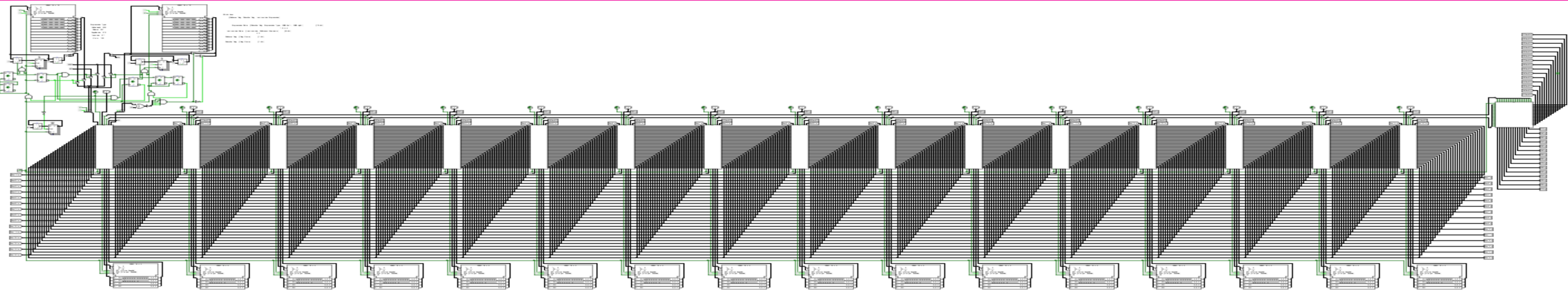
Figure 4 `AddBottomNode` **Instruction**

Update Depth Instruction

The `ReturnDepth` instruction retrieves the value of a node at a target depth. Similar to accessing an list item in conventional programming by querying its index. Like the `UpdateDepth` instruction, the first line ensures that the instruction is propagated to all nodes within the list. Each node then checks if its own depth matches the instructions target depth. If there's a match, the node returns its value to its parent using the PEB. Since nodes automatically pass expression data upwards, the requested value will be returned to the top of the list.

Add Bottom Node Instruction

The `AddBottomNode` instruction adds a node to the end of a list. When used together with `RemoveBottomNode` it enables the list to emulate stack-like behavior, where the bottom node is the top of the stack. This instruction targets the node with a depth of zero. When a non targeted node receives this instruction it passes data downwards, eventually reaching the target node. This instruction takes multiple clock cycles to complete. The first clock pulse causes the target node to request a new ID, so it can prepare to link to a new node. During the second clock cycle the target node updates its CRP to point to the new node. Finally, the target node sends a `UpdateExpression` instruction to its child, ensuring it adopts the expression characteristics defined on its PEB.



Student Researcher: Jasmine Ritchie

Supervisor: Harry Fitchett, Charles Fox

UNIVERSITY OF LINCOLN
Lincoln Academy of Learning and Teaching

Undergraduate Research Opportunities Scheme

DISCOVER_
Find out more about UROS and this research project at:
LNCN.AC/UROS