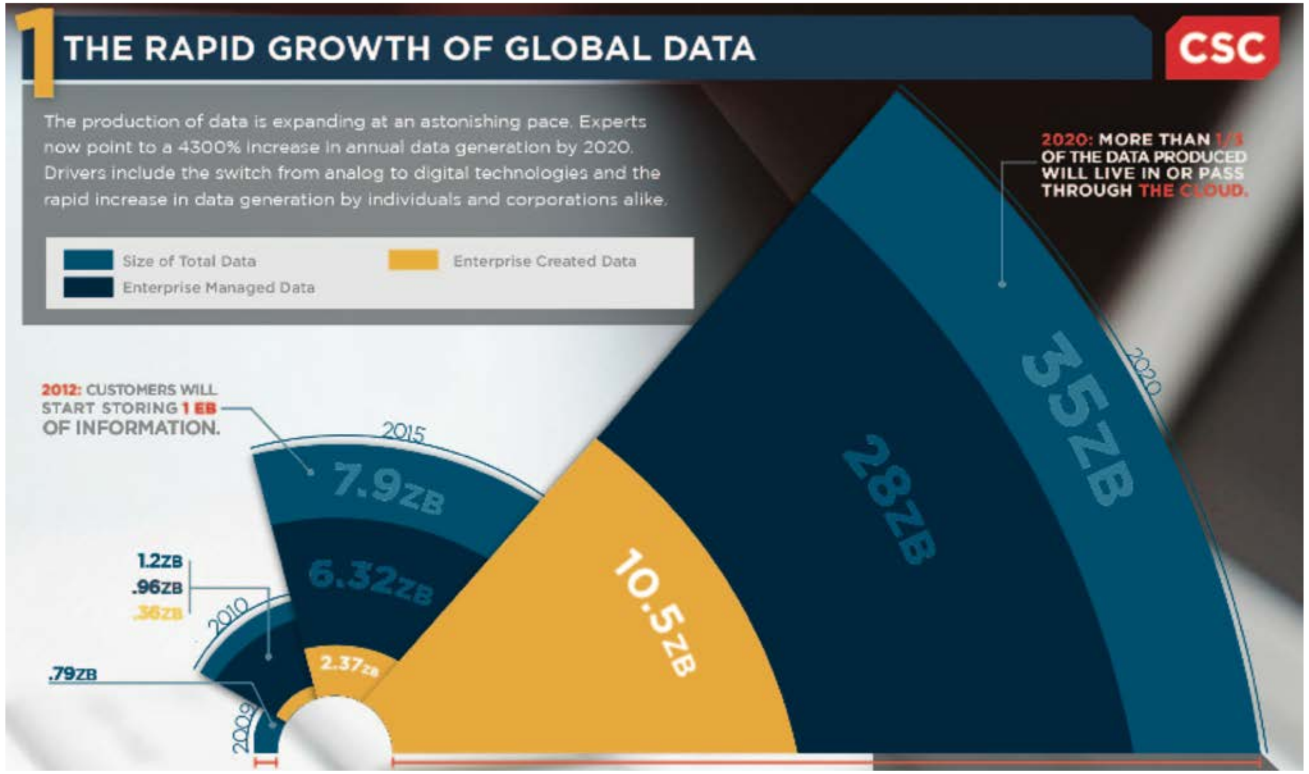
Improving Programming Support for Hardware Accelerators Through Automata Processing Abstractions

PhD Dissertation Proposal

Kevin Angstadt
angstadt@umich.edu

18. December 2018

COMPUTER SCIENCE
& ENGINEERING
UNIVERSITY OF MICHIGAN



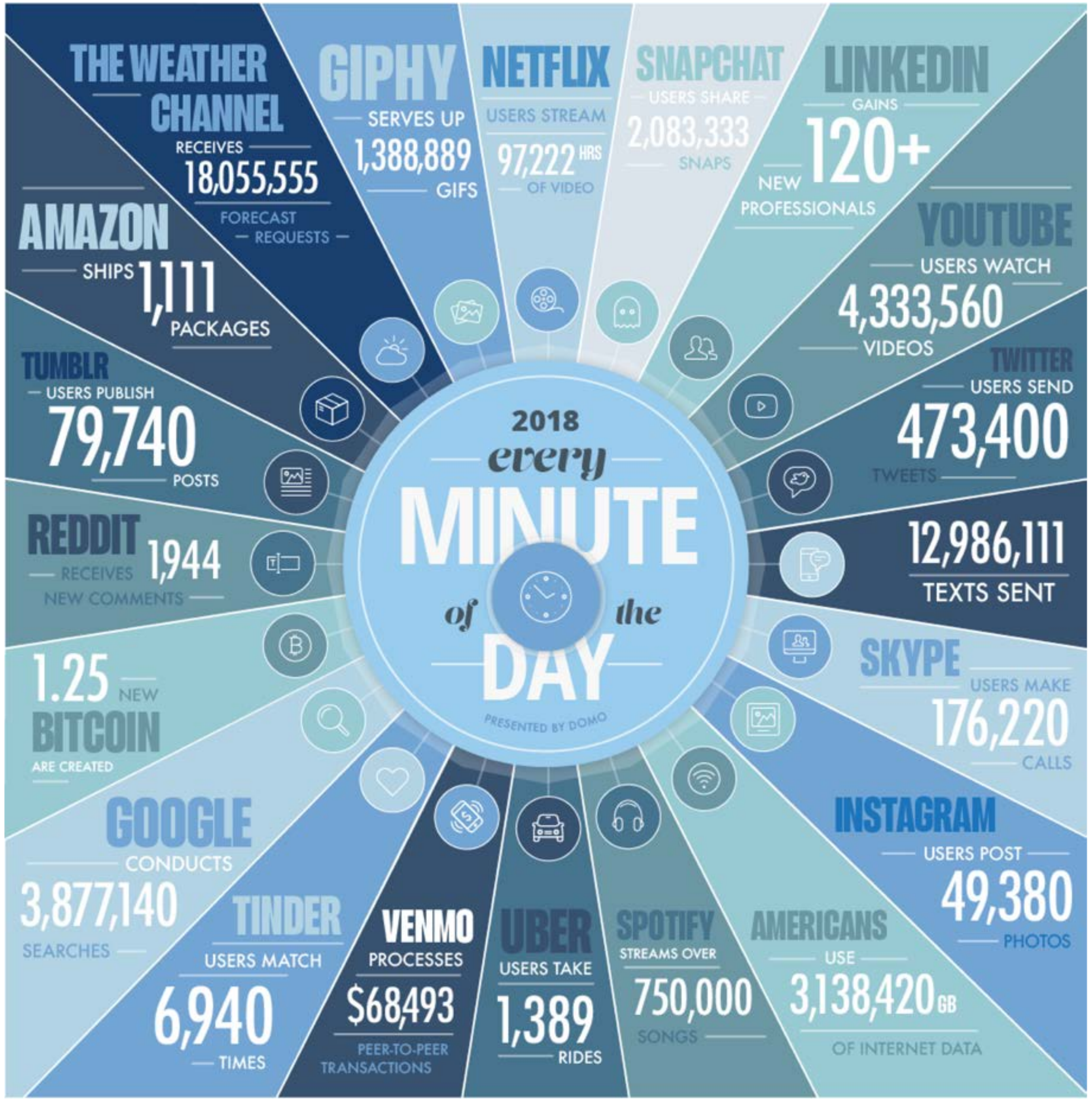
The production of data is expanding at an astonishing pace. Experts now point to a 4300% increase in annual data generation by 2020. Drivers include the switch from analog to digital technologies and the rapid increase in data generation by individuals and corporations alike.

2020: MORE THAN 1/3 OF THE DATA PRODUCED WILL LIVE IN OR PASS THROUGH THE CLOUD.

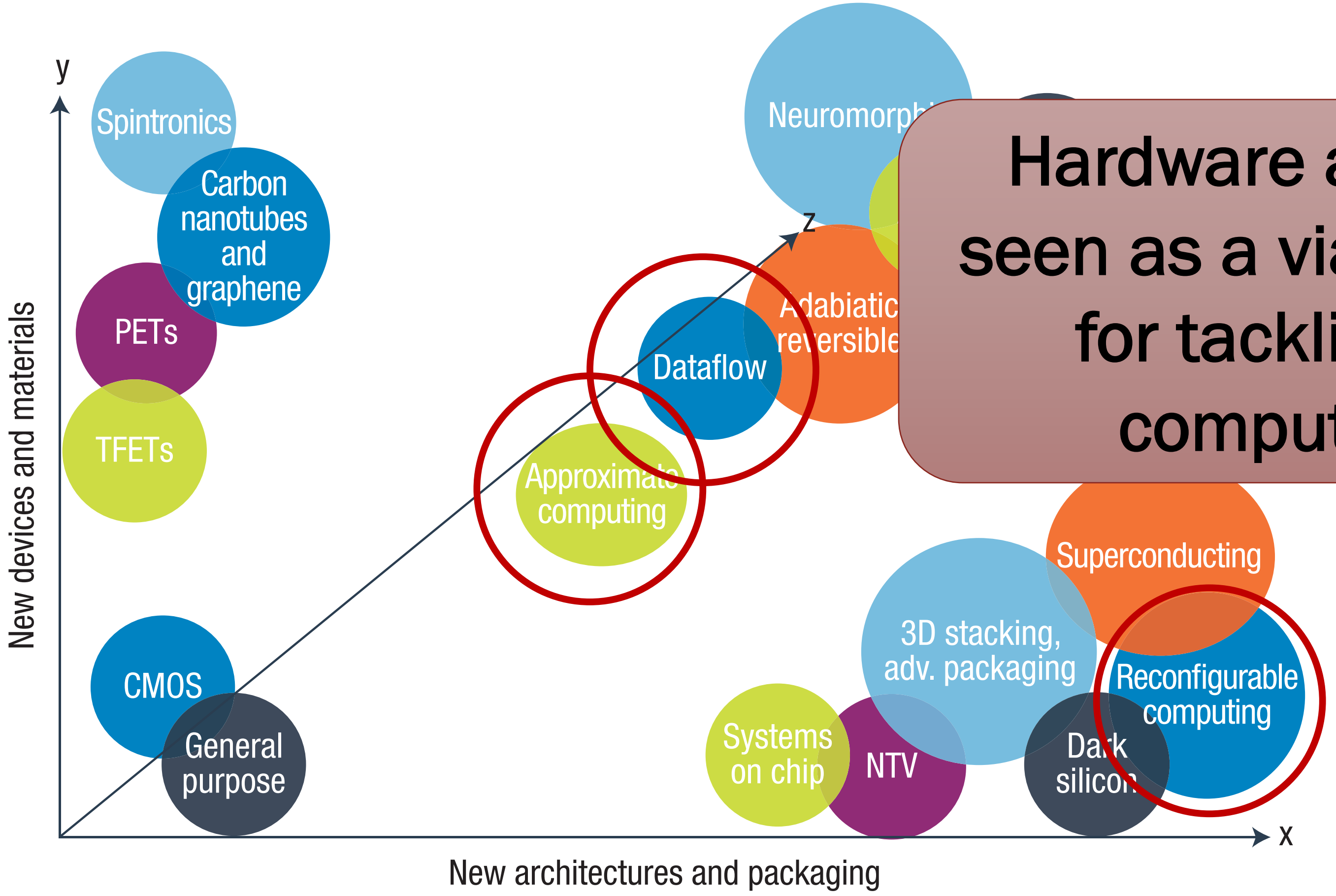
2012: CUSTOMERS WILL START STORING 1 EB OF INFORMATION.

> “By 2020, it’s estimated that for every person on earth, 1.7MB of data will be created every second.”

DOMO, “Data Never Sleeps 6.0”. 2018



Physical Limits Spark Creativity



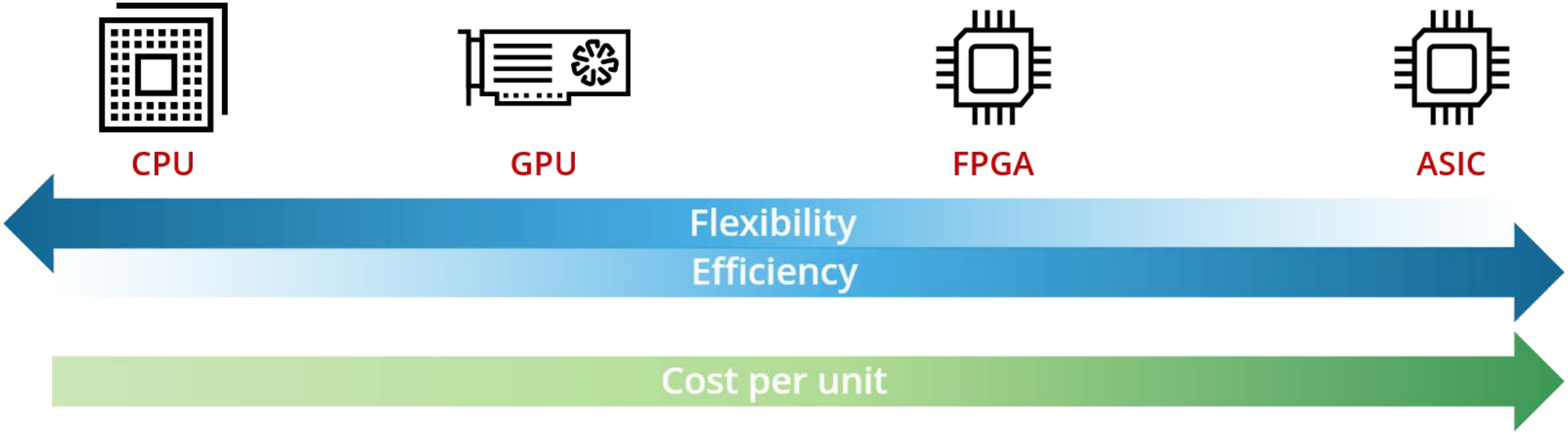
J. M. Shalf and R. Leland, "Computing Beyond Moore's Law". IEEE Computer, 2015.

Hardware accelerators are seen as a viable path forward for tackling increasing compute demands.

Source: Dazeinfo

Source: MIT Technology Review

How Accelerators Help



CPU GPU FPGA ASIC

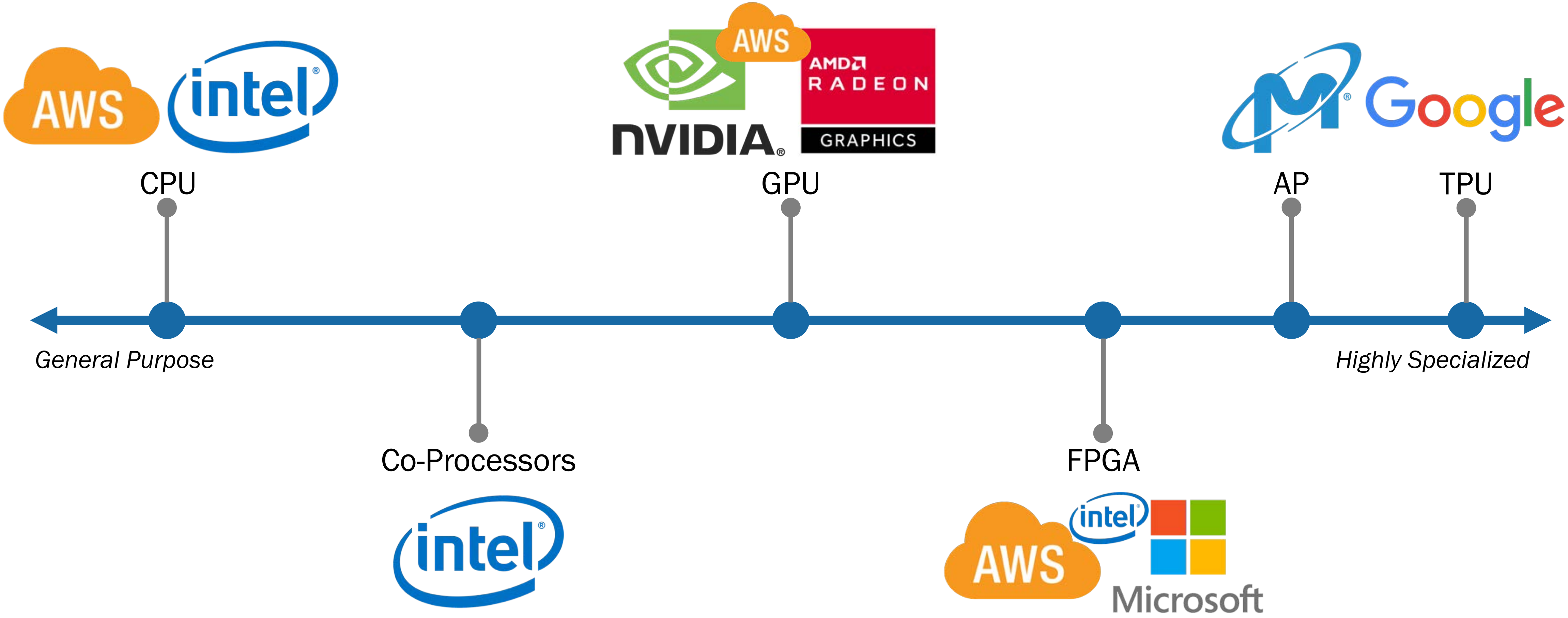
Flexibility

Efficiency

Cost per unit

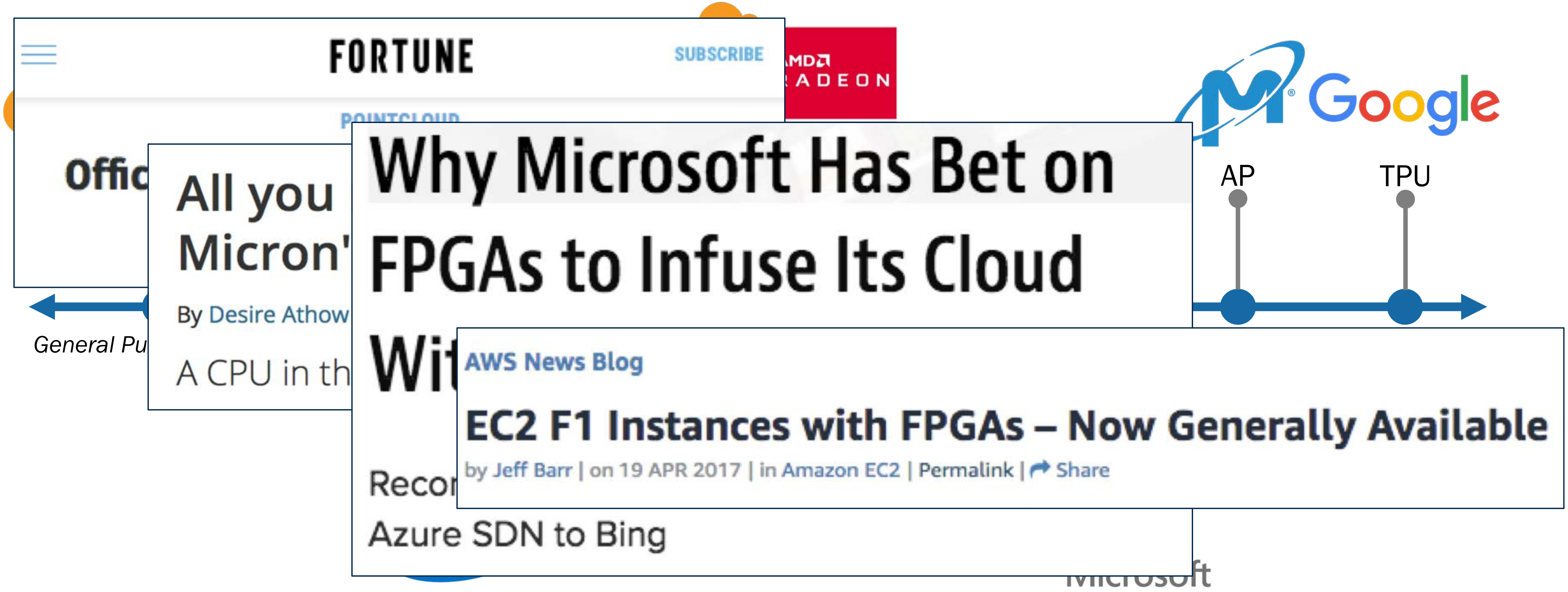
A. Shimoni, “A gentle introduction to hardware accelerated data processing”. Medium, 2018

Significant Interest from Industry



Adapted from: J. Poort, “Cloud 3.0: The Rise of Big Compute”. Rescale, 2017.

Significant Interest from Industry



FORTUNE

SUBSCRIBE

POINTCLOUD

Offic

All you

Micron'

By Desire Athow

A CPU in th

Why Microsoft Has Bet on FPGAs to Infuse Its Cloud Wit

Recor

Azure SDN to Bing

AWS News Blog

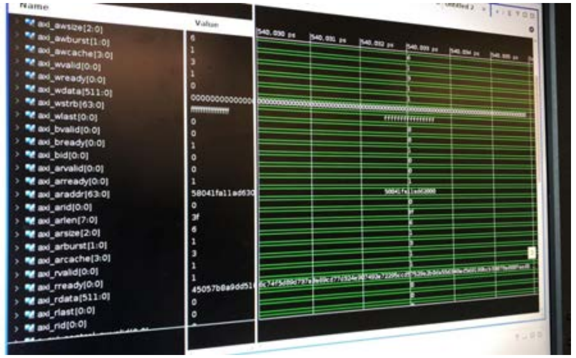
EC2 F1 Instances with FPGAs – Now Generally Available

by Jeff Barr | on 19 APR 2017 | in Amazon EC2 | Permalink | Share

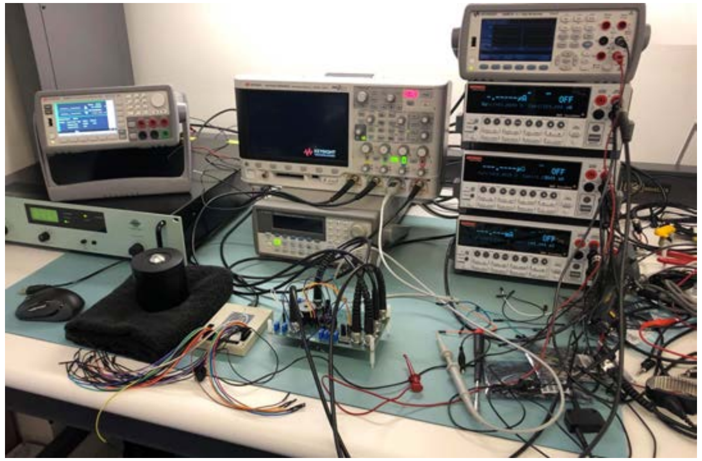
Adapted from: J. Poort, “Cloud 3.0: The Rise of Big Compute”. Rescale, 2017.

Lack of [Good] Programming Models

- Akin to “assembly-level” programming on CPU architectures
  - Are circuit design and digital logic concepts in CS curricula?
- Require **low-level knowledge** of architectural design to produce performant code
- Difficult to **debug** and **maintain:** oscilloscopes and logic analyzers
- Many efforts to improve
  - OpenCL, Xilinx SDAccel, etc.
  - High-level language + annotations + decent performance
  - **Non-intuitive impact** of high-level implementation on performance



Name	Value
axi_awsize[2:0]	6
axi_awburst[1:0]	1
axi_awcache[3:0]	3
axi_wvalid[0:0]	1
axi_wready[0:0]	0
axi_wdata[511:0]	0000000000000
axi_wstrb[63:0]	ffffffffffffff
axi_wlast[0:0]	0
axi_bvalid[0:0]	0
axi_bready[0:0]	1
axi_bid[0:0]	0
axi_arvalid[0:0]	0
axi_arready[0:0]	1
axi_araddr[63:0]	58041fa11ad630
axi_arid[0:0]	0
axi_arlen[7:0]	3f
axi_arsize[2:0]	6
axi_arburst[1:0]	1
axi_arcache[3:0]	3
axi_rvalid[0:0]	1
axi_rready[0:0]	1
axi_rdata[511:0]	45057b8a9dd51
axi_rlast[0:0]	0
axi_rid[0:0]	0



Successful Programming Models

- **Performance and Scalability:** minimize overhead introduced by high-level programming models and tools.
- **Ease of Use:** provide familiar abstractions and a shallow learning curve.
- **Expressive Power:** support the applications that developers wish to accelerate with dedicated hardware.
- **Legacy Support:** support the adaptation of existing software to execute efficiently on hardware accelerators while placing a minimal burden on developers.

Finite automata provide a suitable abstraction for bridging the gap between high-level programming models and maintenance tools familiar to developers and the low-level representations that execute efficiently on hardware accelerators.

Proposal Thesis

Automata Processing in the Big Data World

Detecting Intrusion Attempts in Network Packets

Learning Association Rules with an *a priori* approach

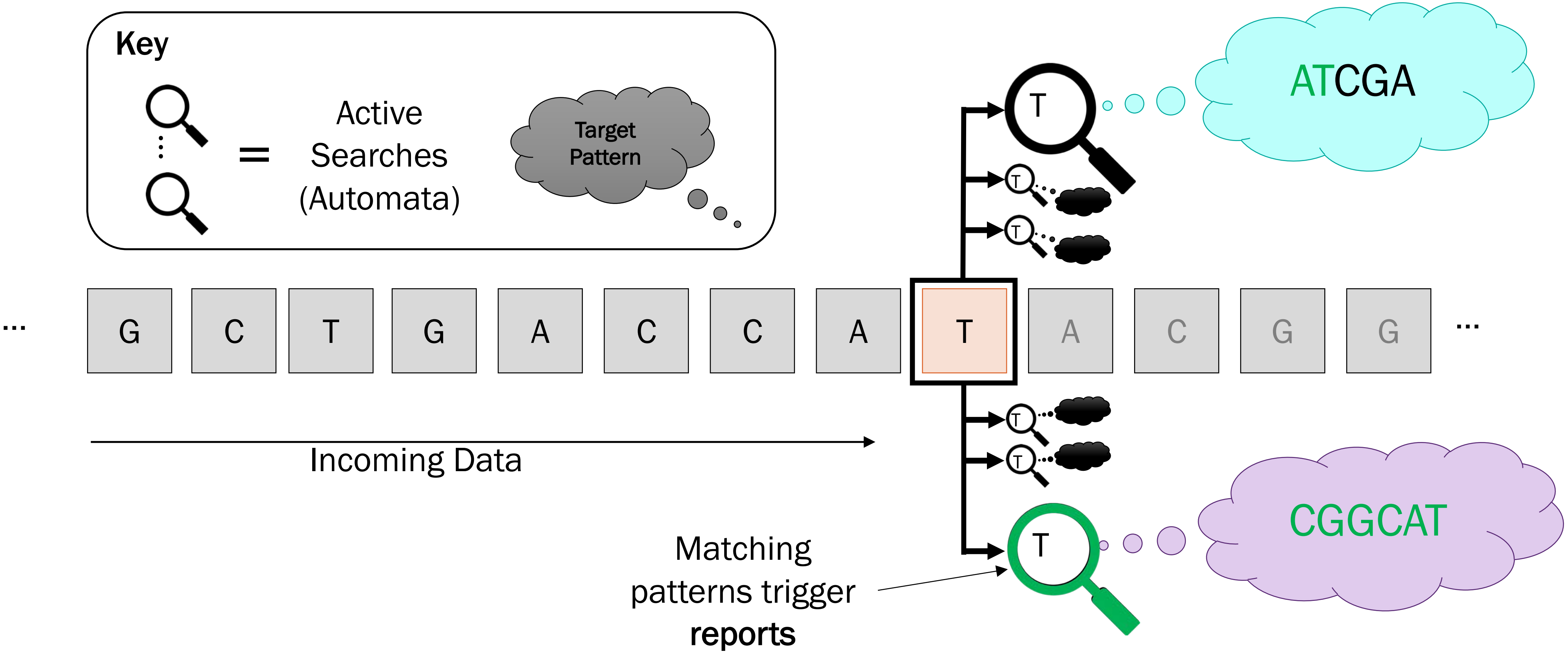
Detecting incorrect POS tags in NLP

Looking for Virus Signatures in Binary Data

Detecting Higgs Events in Particle Collider Data

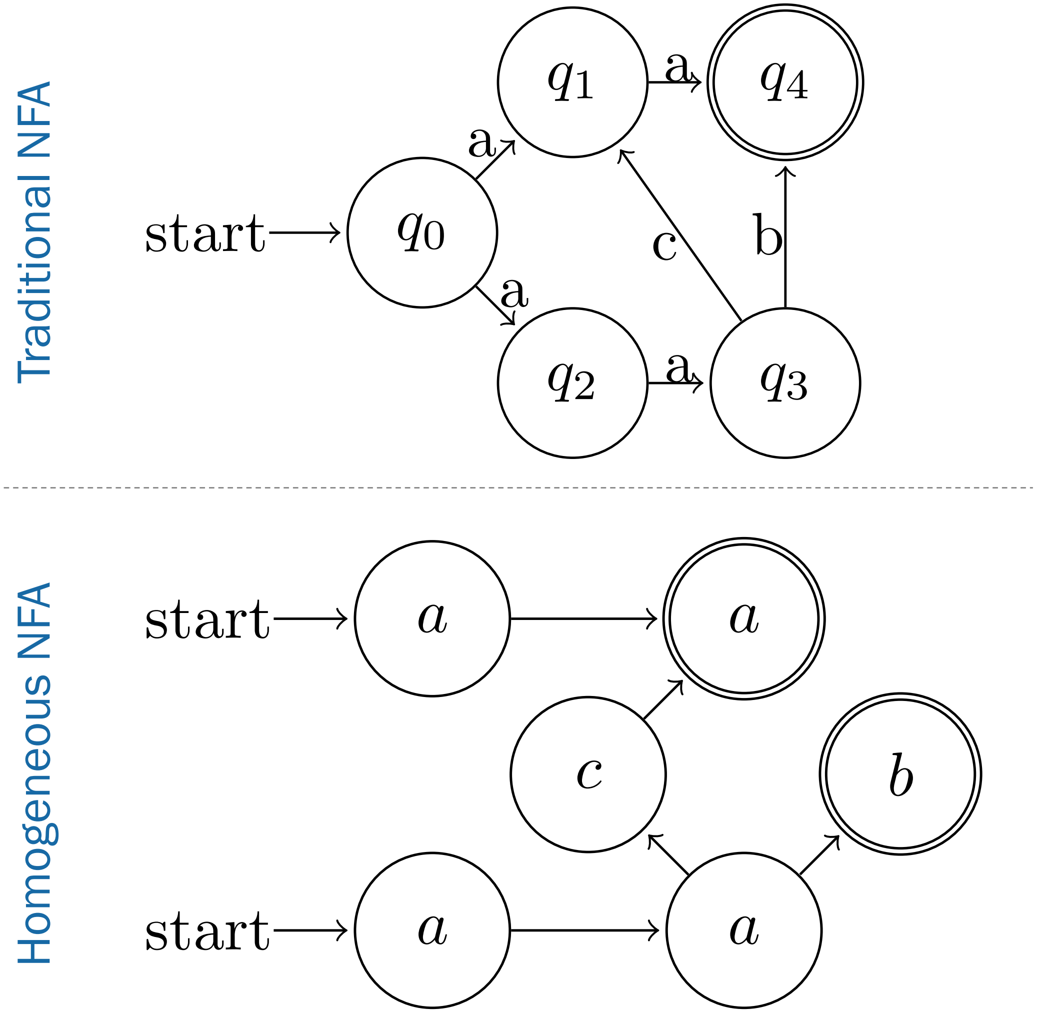
Aligning DNA Fragments to the Human Genome

Finite Automata: 10,000ft View



Homogeneous Finite Automata

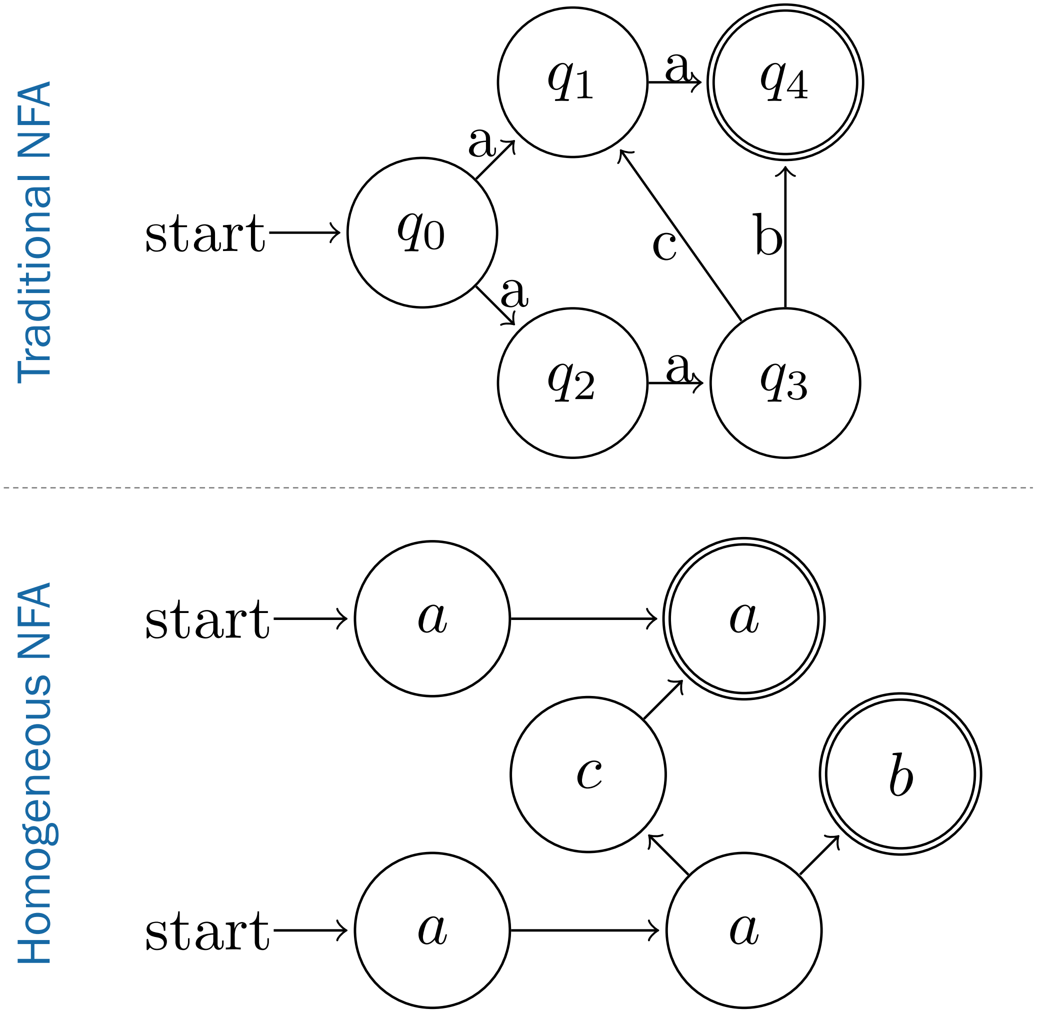
- Finite set of states with transitions operating over a finite alphabet
- Input data processed by repeatedly applying transition rules
- Non-determinism: multiple transitions on single input
- Homogeneity: all incoming transitions occur on the same input character



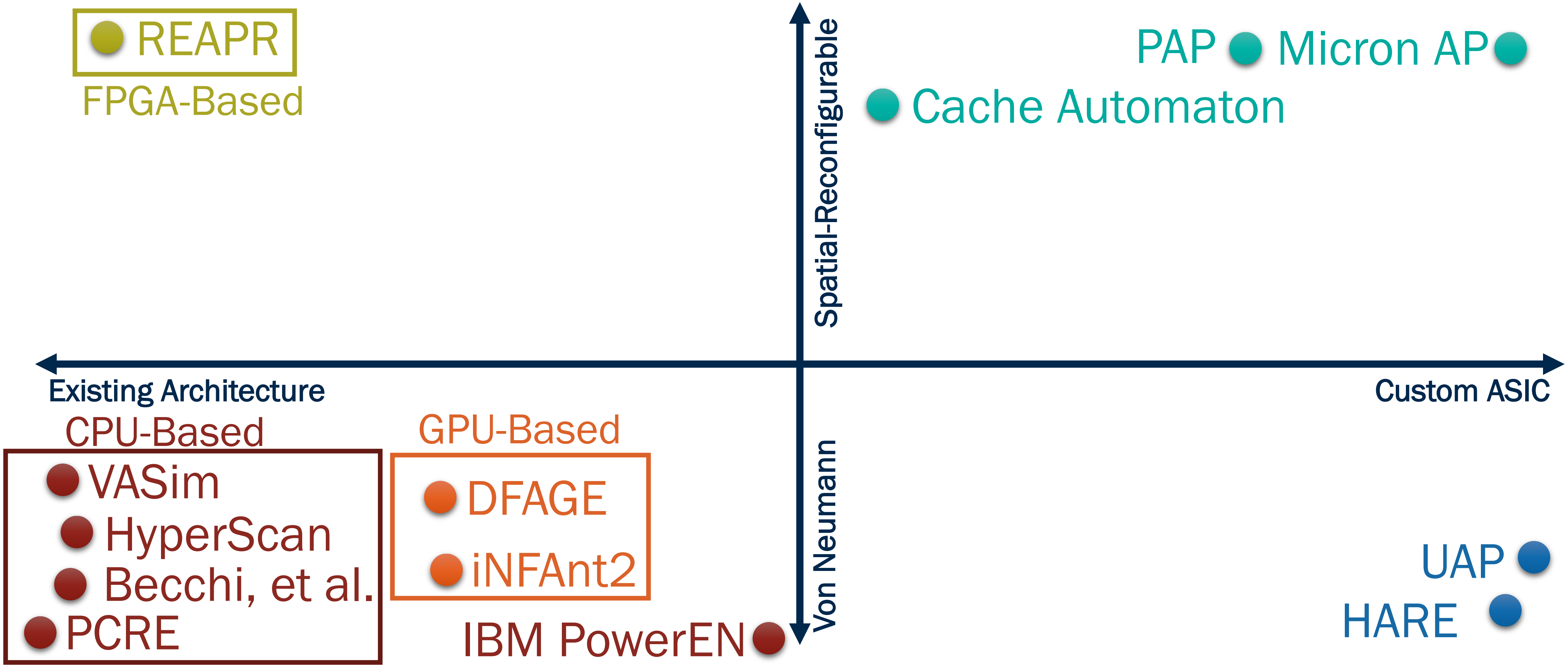
Homogeneous Finite Automata

State Transition Element (STE): a state in a homogeneous NFA

- **Non-determinism**: multiple transitions on single input
- **Homogeneity**: all incoming transitions occur on the same input character



Automata/RegEx Processing Platforms



Proposal Overview

- Proposed Research Efforts
  - High-Level Programming Language: RAPID
  - High-Speed, Interactive Debugger for Hardware Accelerators
  - In-Cache Accelerator for Pushdown Automata
  - Adapting Legacy Code for Execution on Hardware Accelerators
- Candidate Schedule
- Conclusion/Discussion

High-Level Languages for Automata Processing

Research Effort 1

Research Effort 1

RE1

Goal: establish the feasibility of compiling an imperative, high-level programming language to a set of finite automata for execution on hardware accelerators

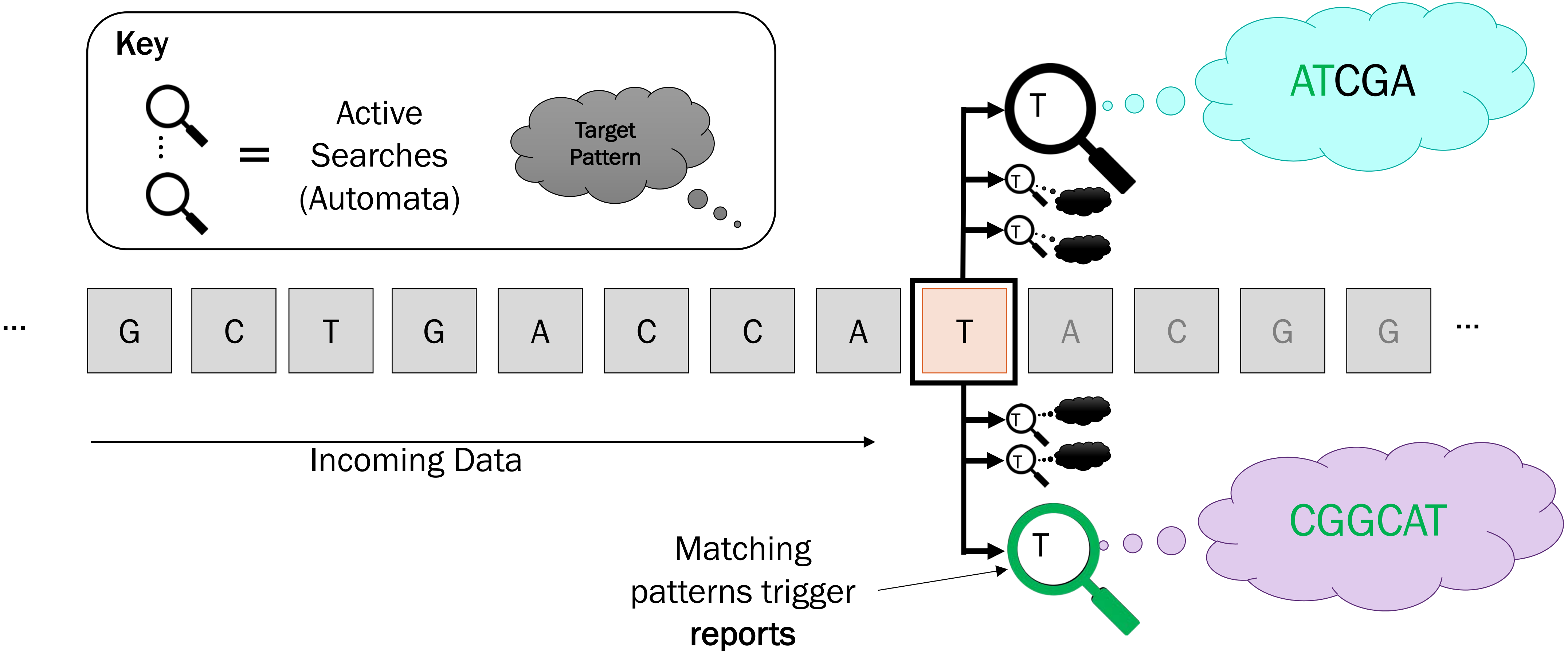
Hypothesis: A high-level programming language will improve the conciseness of representing an algorithm while maintaining the performance of hand-crafted applications for hardware accelerators

RAPID at a Glance

RE1

- Provides concise, maintainable, and efficient representations for pattern-identification algorithms
- Conventional, C- or Java-style language with domain-specific parallel control structures
- Excels in applications where patterns are best represented as a combination of text and computation
- Compilation strategy supports execution on AP, FPGAs, CPUs, and GPUs

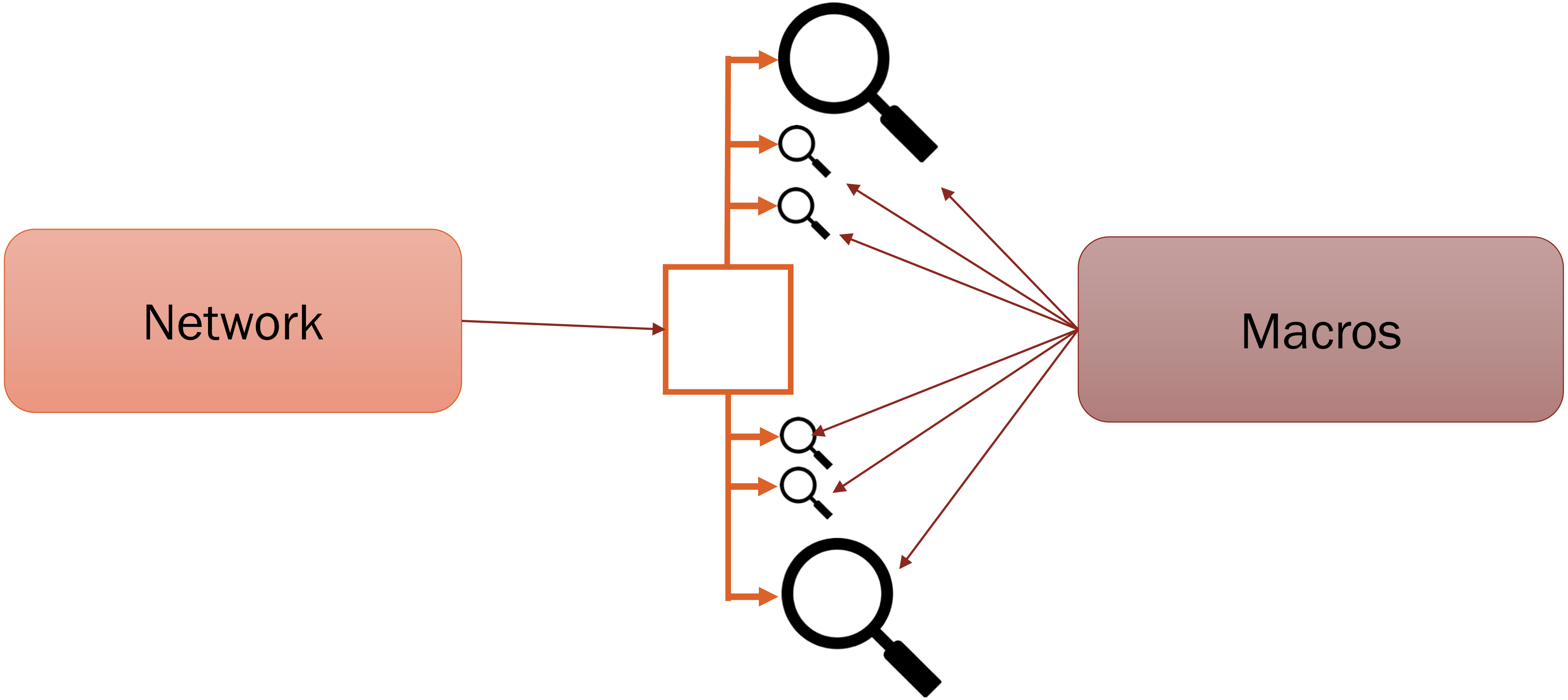
Domain-Specific Code Abstraction



Key
=
Active Searches (Automata)
Target Pattern
T
ATCGA
CGGCAT
…
G
C
T
G
A
C
C
A
T
A
C
G
G
…
Incoming Data
Matching patterns trigger **reports**

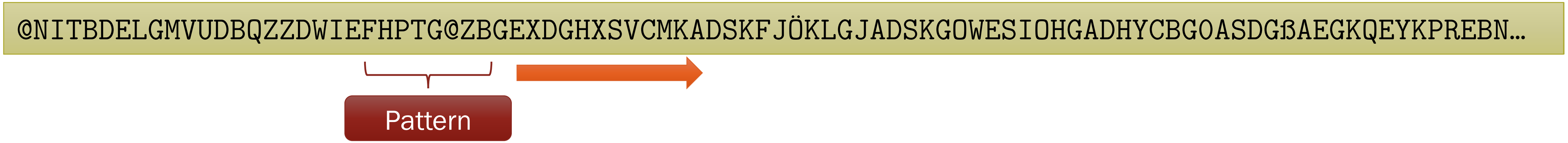
Domain-Specific Code Abstraction

RE1



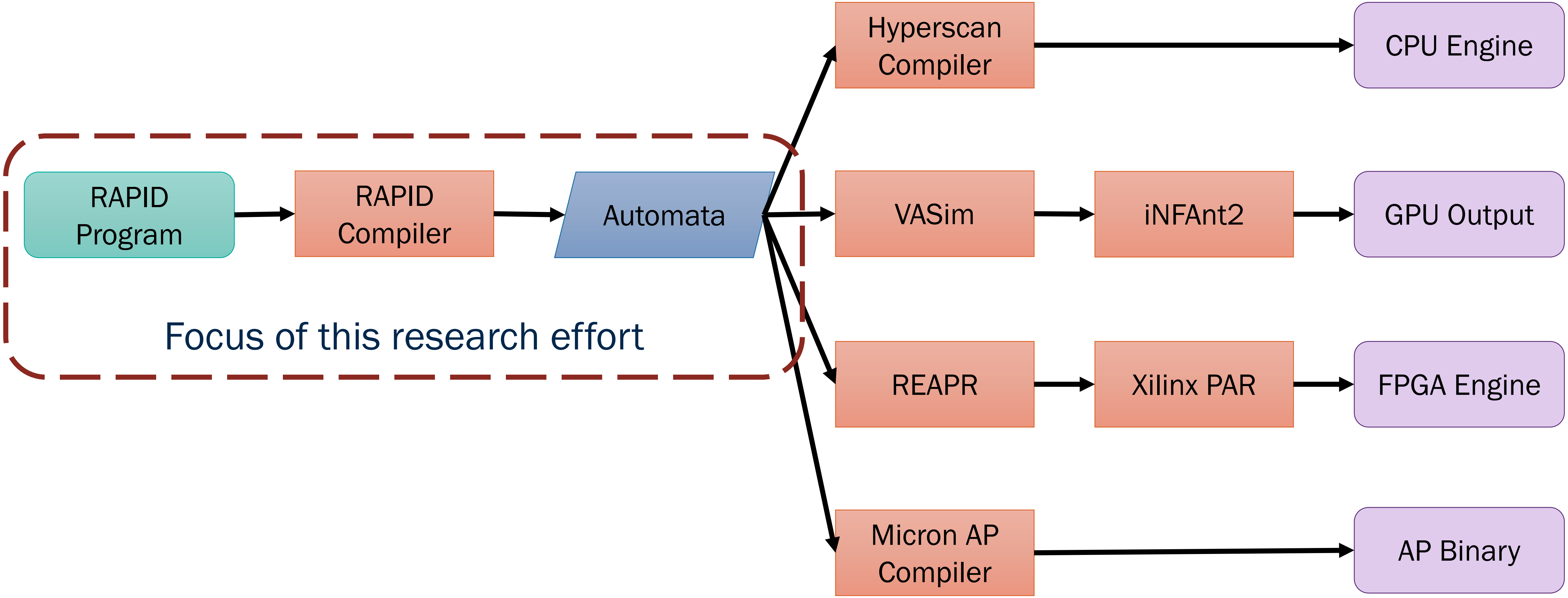
Parallel Control Structures

- Concise specification of multiple, simultaneous comparisons against a single data stream
- Support common pattern search paradigms
- Static and dynamic thread spawning for massive parallelism support
- Explicit support for sliding window computations



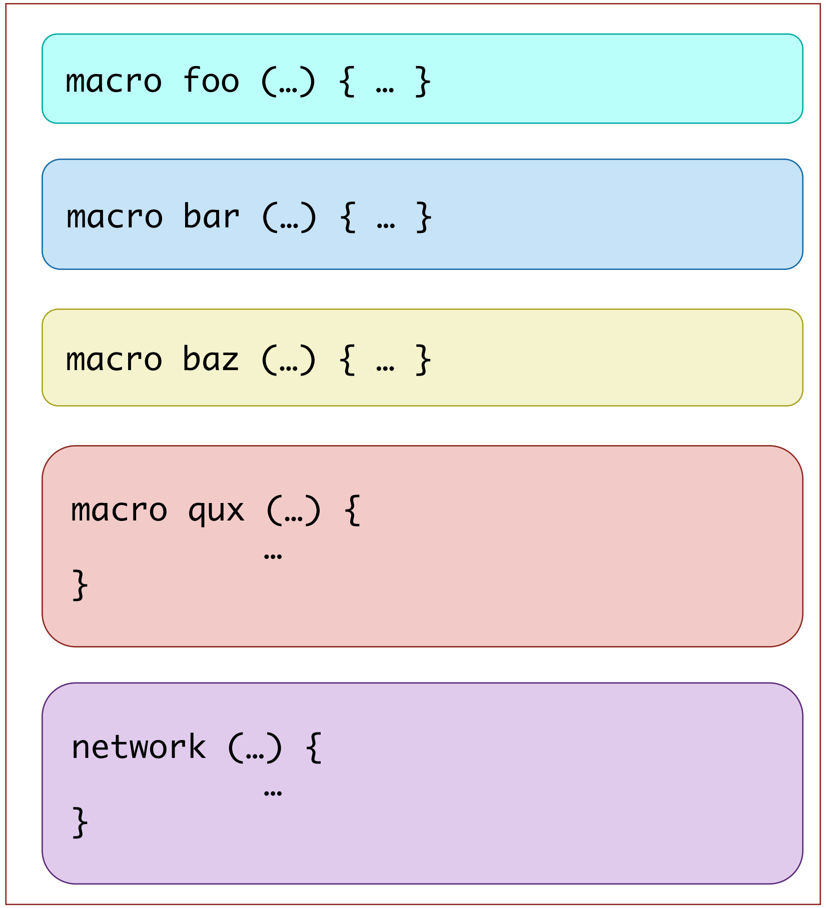
@NITBDELGMVUDBQZZDWIEFHPTG@ZBGEXDGHXSVCMKADSKFJÖKLGJADSKGOWESIOHGADHYCBG0ASDGßAEGKQEYKPREBN…

Multi-Architecture System Overview



Code Generation

RAPID Program



```
macro foo (…) { … }

macro bar (…) { … }

macro baz (…) { … }

macro qux (…) {
              …
}

network (…) {
              …
}
```

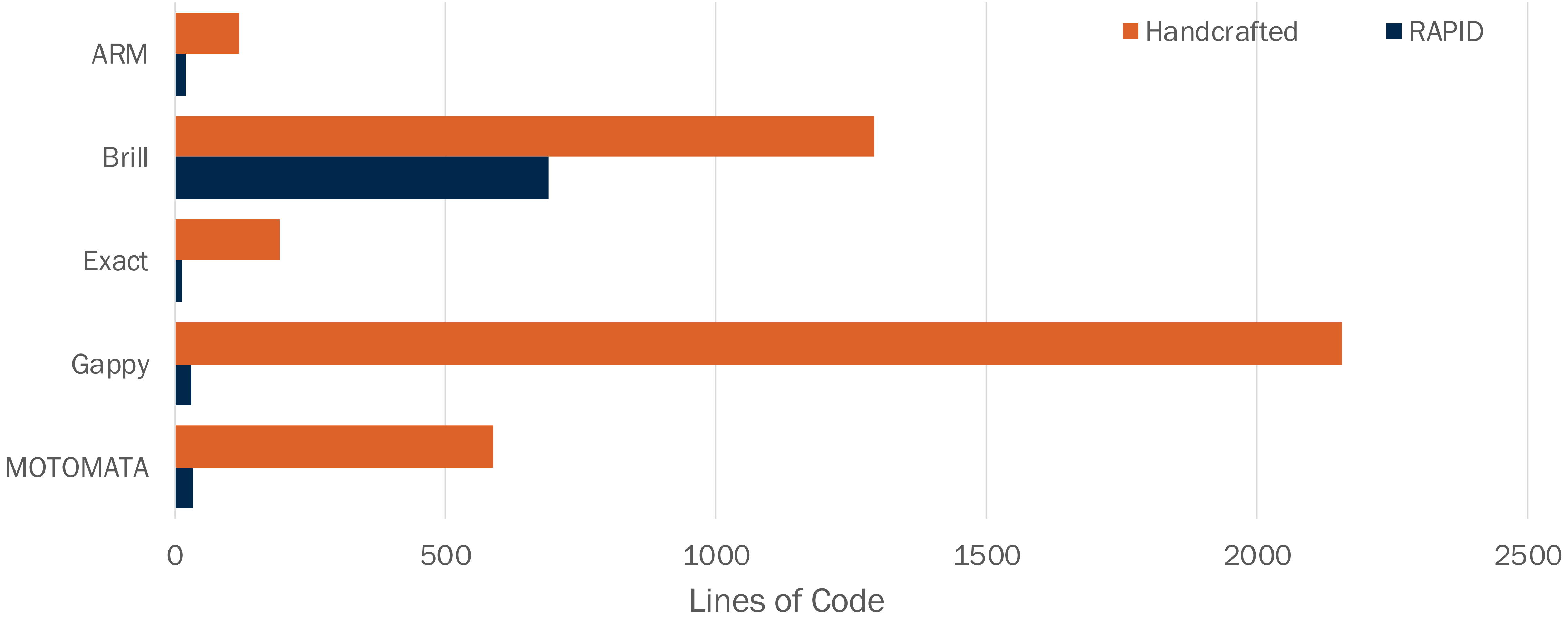
- Recursive transformation of RAPID program
- Similar to RegEx → NFA transformation
- Adapt strategy from **Staged Computation**
  - Imperative statements: evaluate at compile time
  - Declarative interaction with input: evaluate at runtime

Experimental Methodology

RE1

- Five benchmarks from real-world applications (**expressiveness**)
  - **Success:** Applications can be implemented in RAPID
- Compare LOC in RAPID and hand-crafted baseline automata (**scalability**)
  - **Success:** RAPID size remains constant or grows sublinearly with application instance size
- Measure required hardware resources for AP and FPGA (**performance**)
  - **Success:** overheads within 15% of baseline

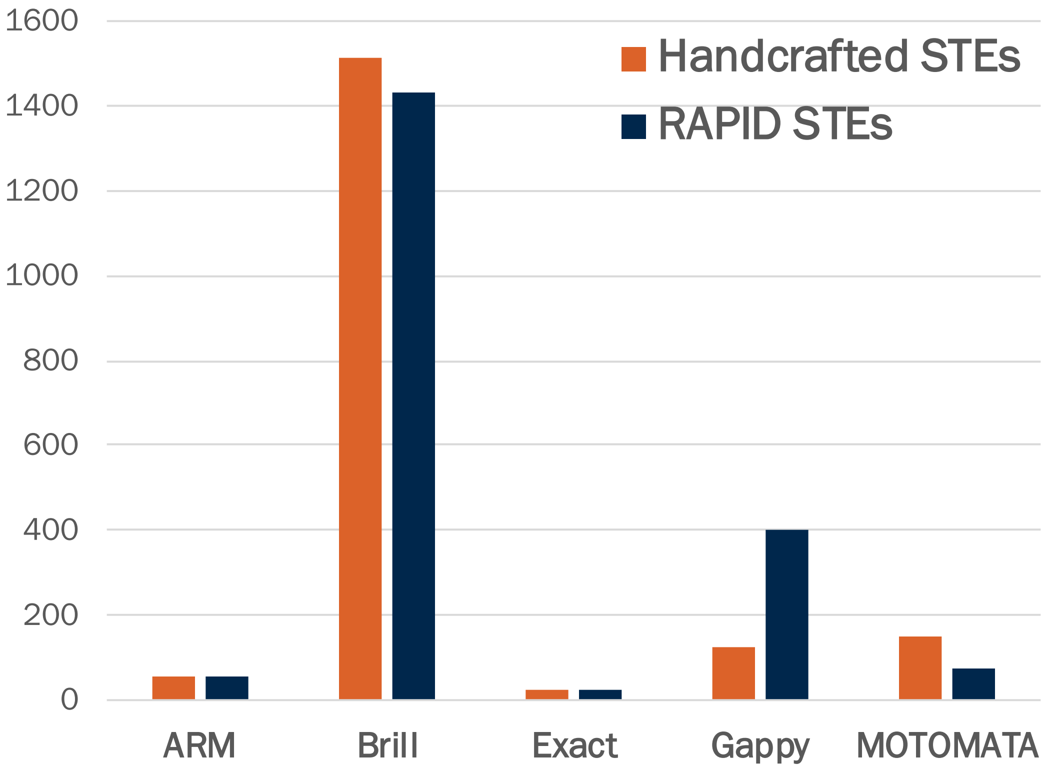
RAPID Lines of Code



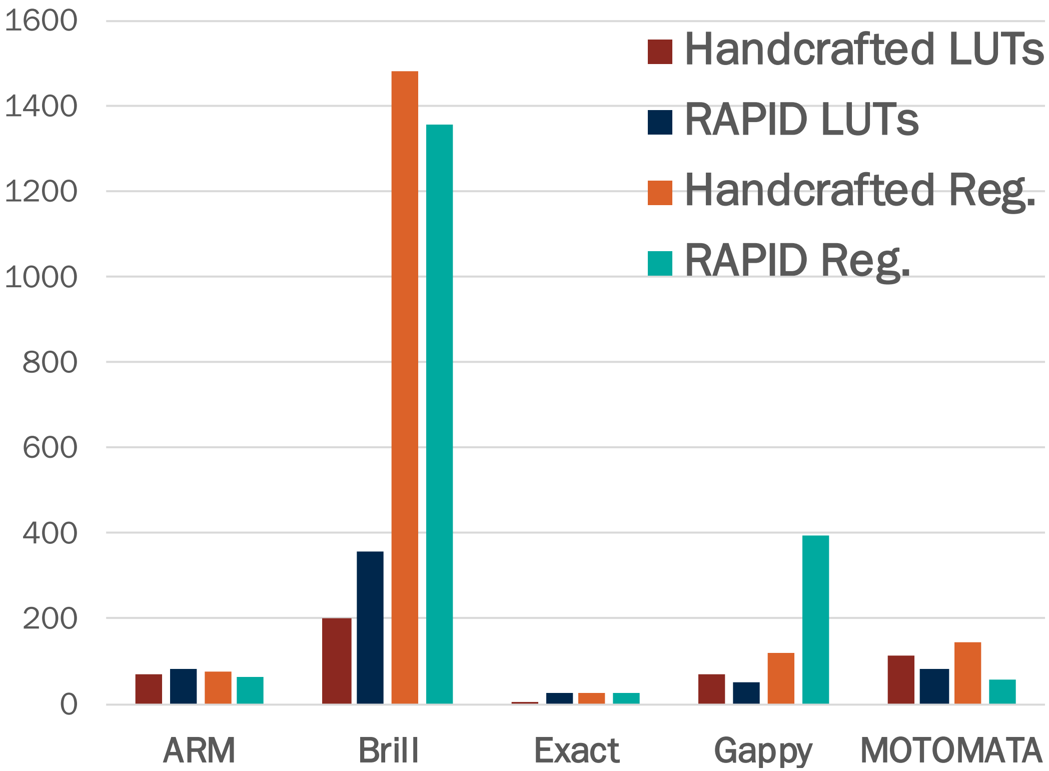
RAPID Hardware Utilization

RE1

Automata Processor



FPGA



Research Effort 1 Summary

RE1

- Feasibility of compiling an imperative, high-level programming language to a set of finite automata for execution on hardware accelerators
- RAPID extends C- or Java-like language with domain-specific parallel control structures and code abstraction
- Preliminary results demonstrate low overheads on AP and FPGA; reduced program size

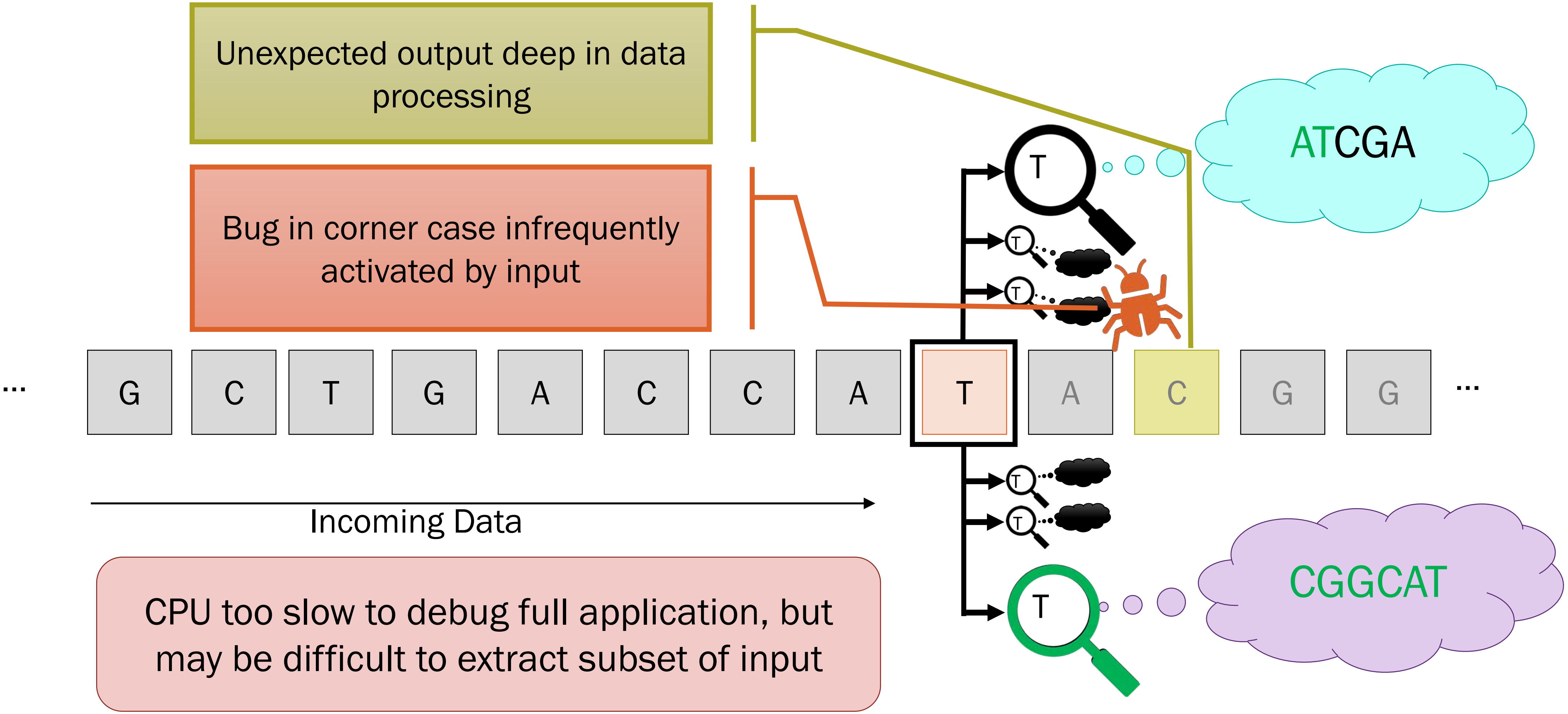
Interactive Debugging for High-Level Languages and Accelerators

Research Effort 2

Houston, we have a problem!

Unexpected output deep in data processing

Bug in corner case infrequently activated by input



CPU too slow to debug full application, but may be difficult to extract subset of input

Research Effort 2

RE2

Goal: Design a high-speed interactive debugger for a high-level language (RAPID) executing on a hardware accelerator

Hypothesis: The automata abstraction reduces the program state that must be monitored at the signal level on hardware accelerators while still allowing for program semantics to be lifted to higher levels of abstraction and meaningfully supporting debugging

“A [debugger is a] program designed to help detect, locate, and correct errors in another program. It allows the developer to **step through the execution** of the process and its threads, **monitoring memory, variables, and other elements** of process and thread context.”

MSDN Windows Dev Center

(https://msdn.microsoft.com/en-us/library/windows/desktop/ms679306(v=vs.85).aspx)

Multi-Step Process

1. First, we must halt execution and extract current program state from the processor

   Insight: repurpose existing hardware/signal monitoring

2. Then, we lift the extracted state to the semantics of the source language

   Insight: generate mapping from expressions/statements to hardware elements during compilation

Where do we stop?

- **Breakpoints** annotate expressions/statements to specify locations to pause execution for inspection
  - Traditional notion relies on instructions stream
  - Mechanism does not apply directly to architectures with no instructions (e.g., FPGAs, AP)
- **Key Insight:** Automata computation driven by input
  - Set breakpoints on input data, not instructions
  - Supports use case of stopping computation at abnormal behavior
  - Can also provide abstraction of traditional breakpoints

Capturing State

- Process input data up to breakpoint
- State of automata is **compact**
  - O(n) in the number of states of the NFA
  - **State vector** captures relevant execution information
- Repurpose existing hardware to capture
  - AP: State vector cache already stores active states
  - FPGA: Integrated Logic Analyzers (ILAs) and Virtual I/O pins (VIOs) allow for probing of activation bits

Challenge: Space overhead on FPGAs

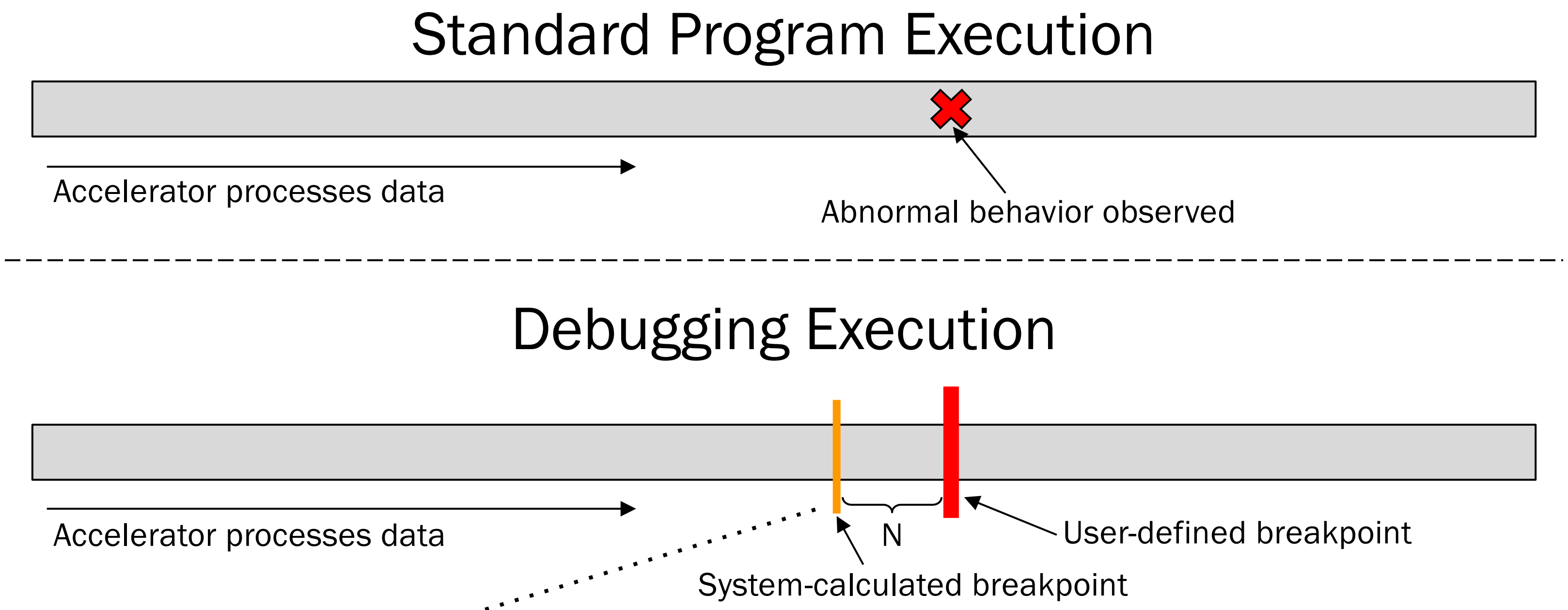
ILAs can be dynamically reconfigured to probe different signals, but support additional features, causing bloat

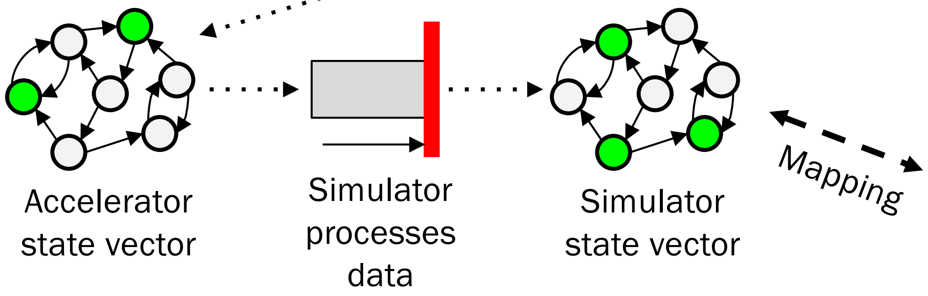
VIOs are tied to the same clock as the design slow down the design

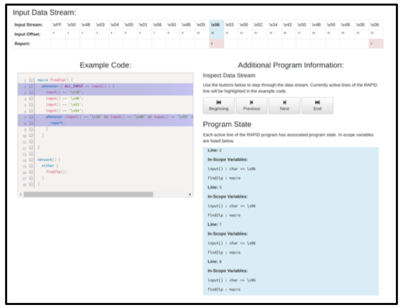
Lifting Hardware State to Source-Level

- Modify the RAPID compiler to generate debugging tables
  - Approach for the RAPID compiler is similar to traditional compilation
  - For every line, which NFA states does it map to?
  - For every line, what variables are in scope and what are their values (or which hardware resources hold their values)?
- At breakpoint, apply mappings in reverse
- Now have:
  - Expressions currently executing
  - Values of in-scope variables

Putting it all together


Standard Program Execution
Accelerator processes data
Abnormal behavior observed
Debugging Execution
Accelerator processes data
N
User-defined breakpoint
System-calculated breakpoint




Input Data Stream:

Input Stream:	\xFF	\x50	\x4B	\x03	\x04	\x00	\x01	\x06	\x50	\x4B	\x05	\x06	\x53	\x00	\x5C	\x34	\x42	\x50	\x4B	\x50	\x4B	\x05	\x06
Input Offset:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Report:												r											r

Example Code:

```
1  macro findZip() {
2    whenever ( ALL_INPUT == input() ) {
3      input() == '\x50';
4      input() == '\x4B';
5      input() == '\x03';
6      input() == '\x04';
7      whenever (input() == '\x50' && input() == '\x4B' && input() == '\x05' &
8        report;
9      }
10   }
11
12 }
13
14 network() {
15   either {
16     findZip();
17   }
18 }
```

Additional Program Information:

Inspect Data Stream

Use the buttons below to step through the data stream. Currently active lines of the RAPID line will be highlighted in the example code.

Program State

Each active line of the RAPID program has associated program state. In-scope variables are listed below.

Line: 2

In-Scope Variables:

```
input() : char => \x06
findZip : macro
```

Line: 3

In-Scope Variables:

```
input() : char => \x06
findZip : macro
```

Line: 7

In-Scope Variables:

```
input() : char => \x06
findZip : macro
```

Line: 8

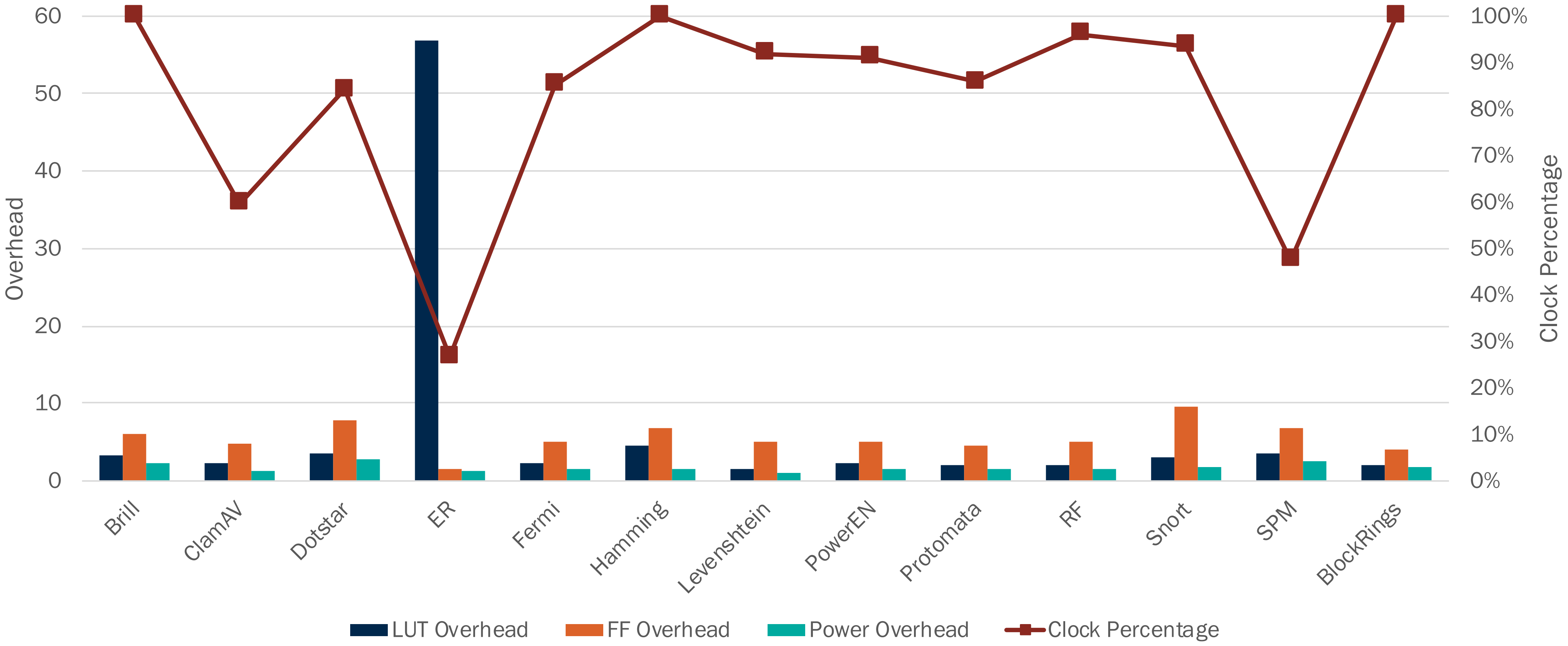
In-Scope Variables:

```
input() : char => \x06
findZip : macro
```

Experimental Methodology

- Measure **performance** and **scalability** of FPGA-based automata debugging
  - ANMLZoo benchmark Suite (14 real-world applications)
  - Measure additional resources used and relative clock frequency
  - **Success:** fit on commercial FPGA and exceed performance of baseline CPU engine
- Measure **ease of use** with a human study
  - Participants given **fault localization** task
  - Ten RAPID programs with indicative bugs
  - Collect implicated lines and measure time taken to answer
  - **Success:** statistically significant improvement in accuracy or time

Preliminary FPGA Results



Human Study Results

- N=61 participants (predominantly UVA students)
- Our debugging tool improves a user's fault localization accuracy for RAPID programs in a statistically significant manner (p = 0.013)
- No statistically significant impact on the time needed to localize faults in RAPID programs
- Debugging information for RAPID programs helps novices and experts alike (there is no interaction between developer experience and the ability to interpret debugging information)

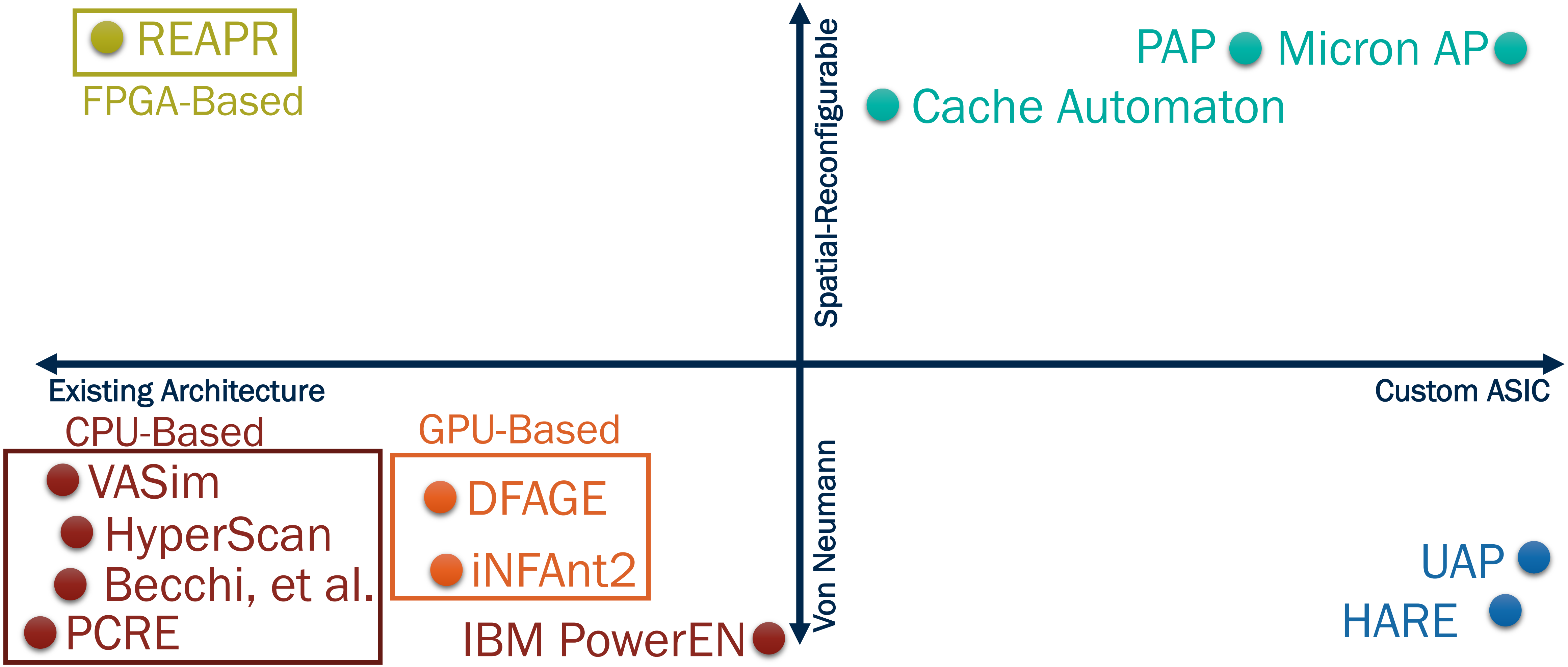
Research Effort 2 Summary

- Design a high-speed interactive debugger for RAPID on hardware accelerators
- Repurpose existing hardware to extract runtime state from accelerator and bridge semantic gap with high-level language
- Preliminary FPGA results demonstrate viability
  - High overhead remain a challenge
- Human Study results demonstrate ease of use

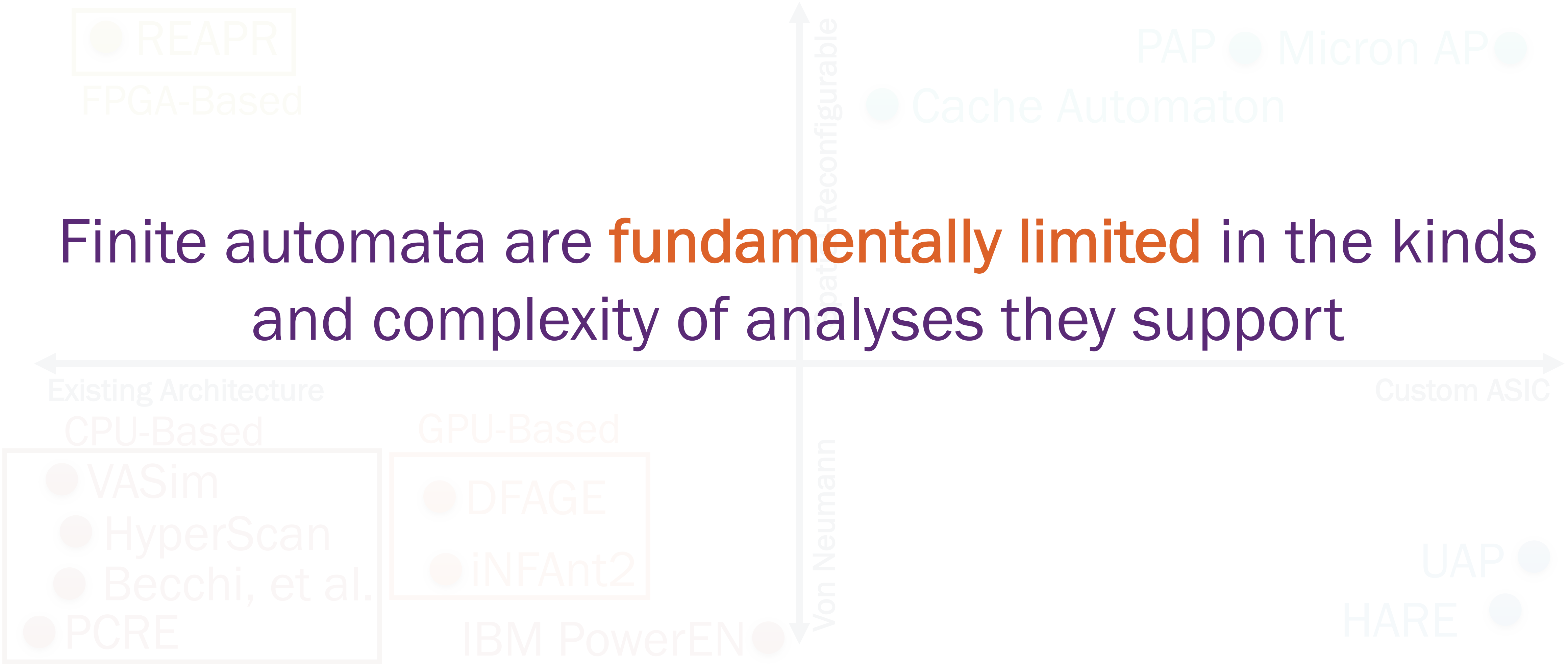
Expressive Power of In-Memory Automata Accelerators

Research Effort 3

Automata/RegEx Processing Platforms



Automata/RegEx Processing Platforms



REAPR

FPGA-Based

PAP

Micron AP

Cache Automaton

Spatial Reconfigurable

Existing Architecture

Custom ASIC

CPU-Based

GPU-Based

VASim

HyperScan

Becchi, et al.

PCRE

DFAGE

iNFAnt2

IBM PowerEN

Von Neumann

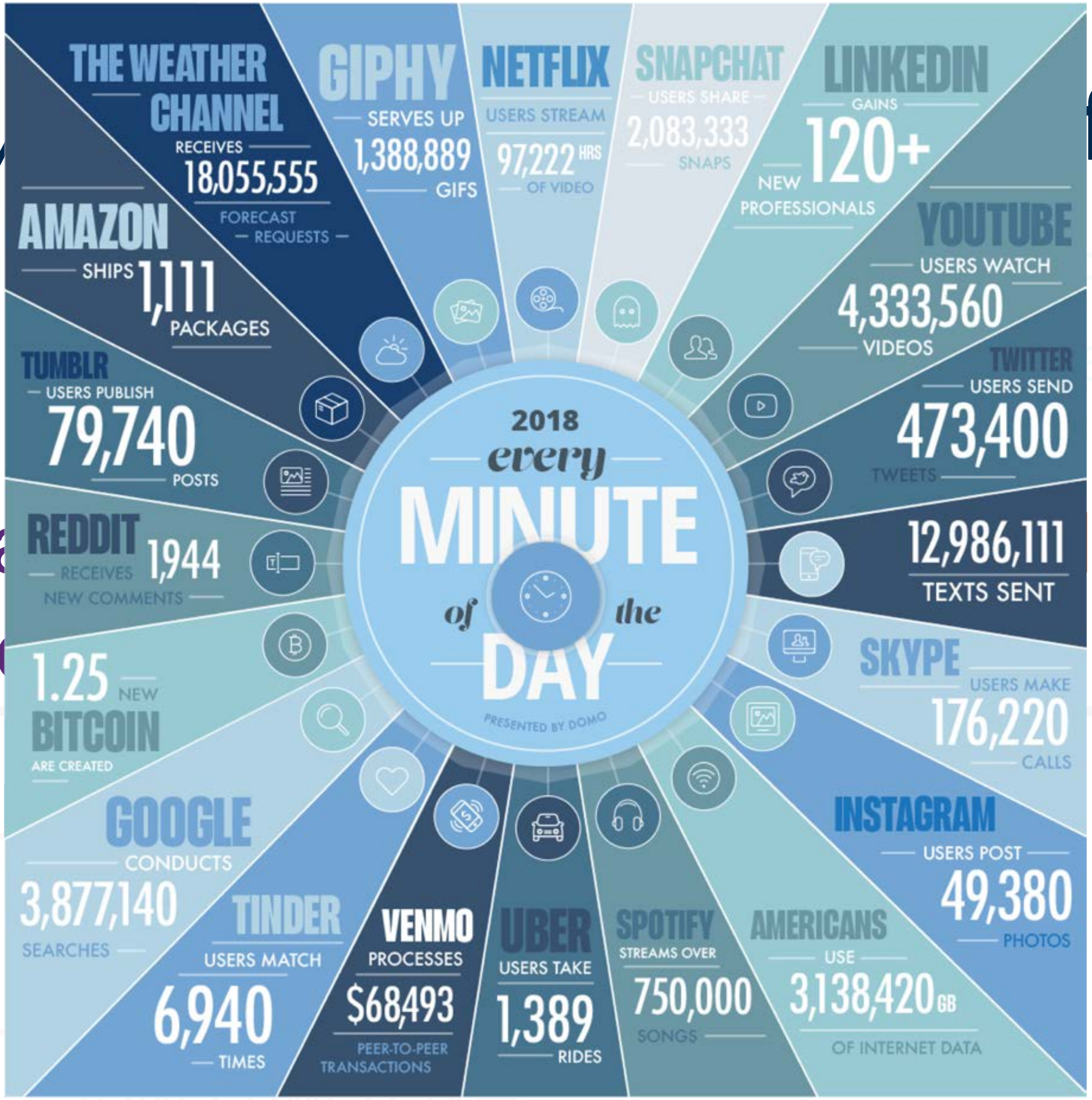
UAP

HARE

Finite automata are fundamentally limited in the kinds and complexity of analyses they support

Automata, forms

Finite automa in the kinds and c pport



REAPR

FPGA-Based

P ● Micron AP ●

aton

Existing Architecture

Custom ASIC

CPU-Based

- VASim
- HyperScan
- Becchi, et al.
- PCRE

IBM PowerEN

UAP

HARE

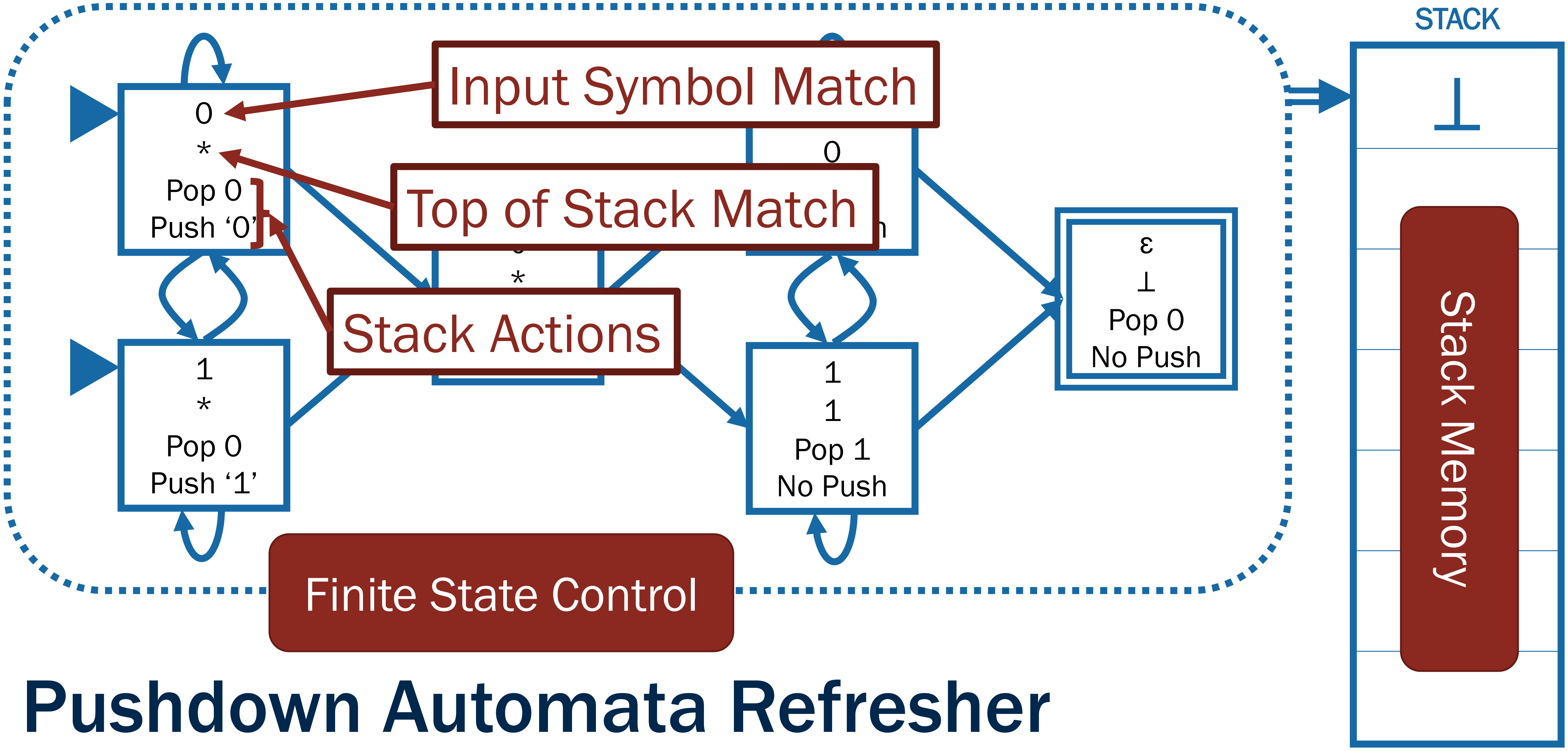
Research Effort 3

Goal: Extend the expressive power of automata-based accelerators to support more common data processing tasks, including the parsing of recursively-nested structures.

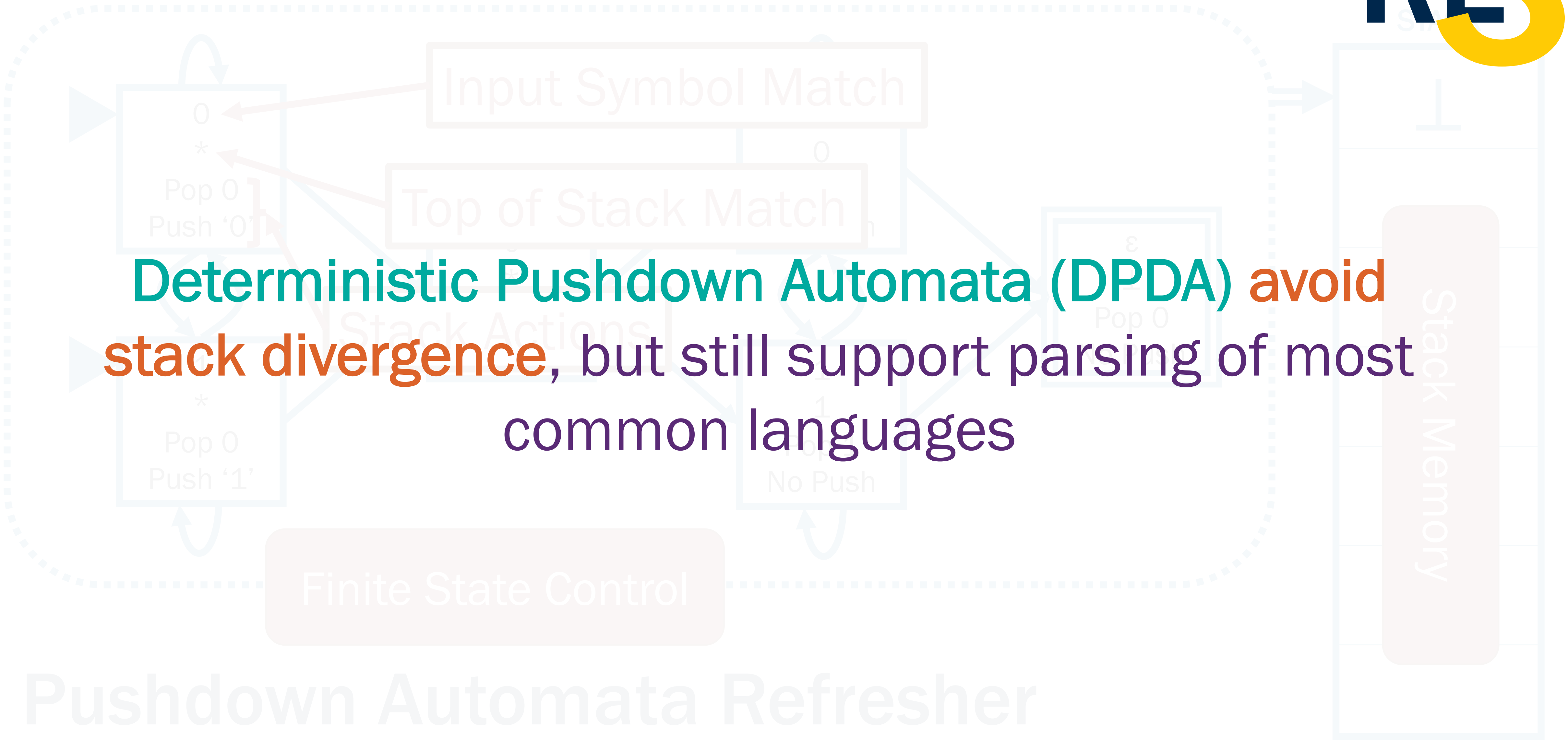
Hypothesis: An in-cache accelerator architecture supporting pushdown automata computation will support a rich class of applications, and allow for improved performance over state-of-the-art baselines.

ASPEN Will Support Richer Analyses

- Accelerated in-SRAM Pushdown ENgine
- Scalable processing engine that uses LLC slices to accelerate Pushdown Automata computation
- Custom five-stage datapath using SRAM lookups can process up to one byte per cycle
- Optimizing compiler supports existing grammars, packs states efficiently, and reduces the number processing stalls

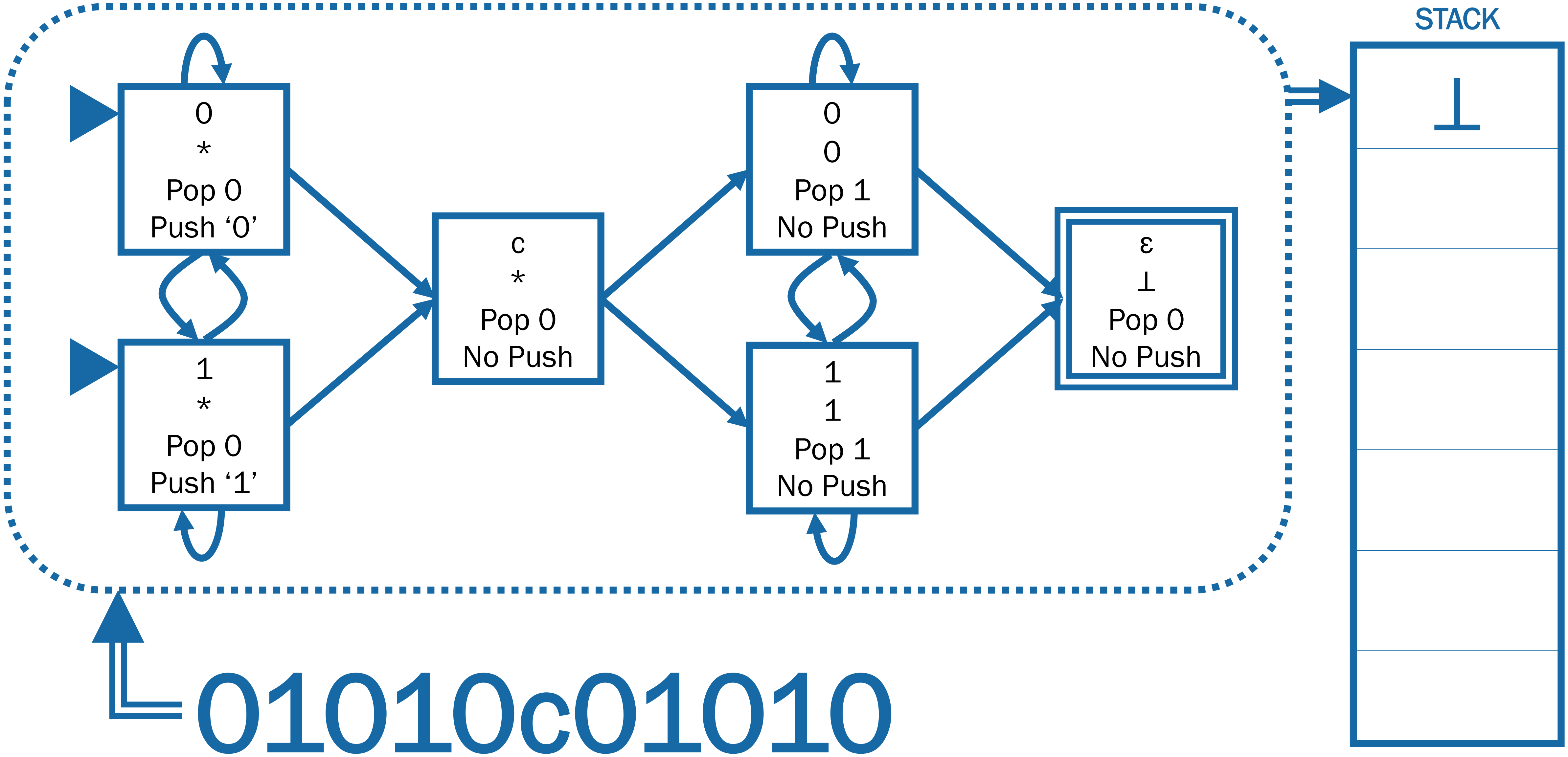


Pushdown Automata Refresher

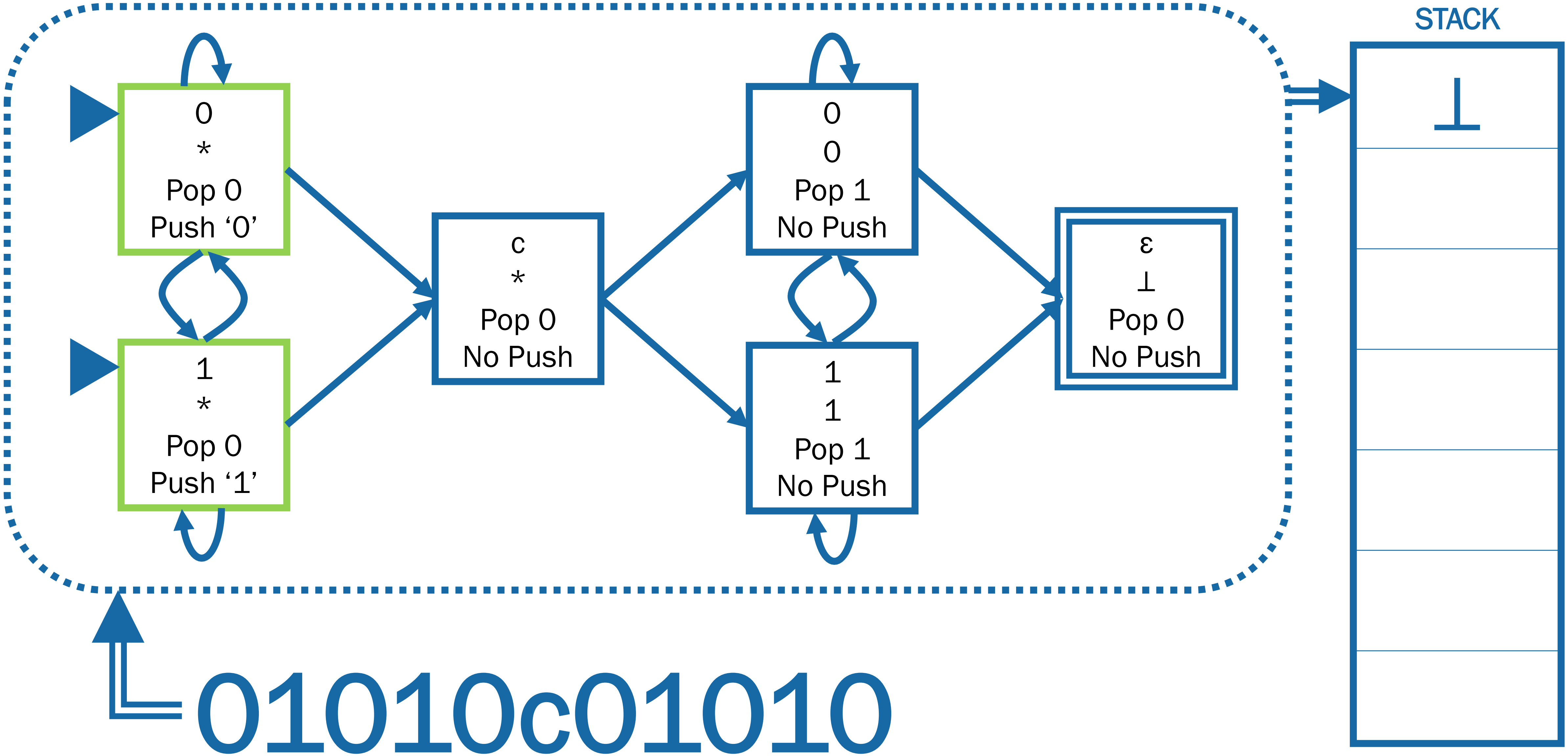


Deterministic Pushdown Automata (DPDA) avoid stack divergence, but still support parsing of most common languages

Recognizing Palindromes with a Middle Character

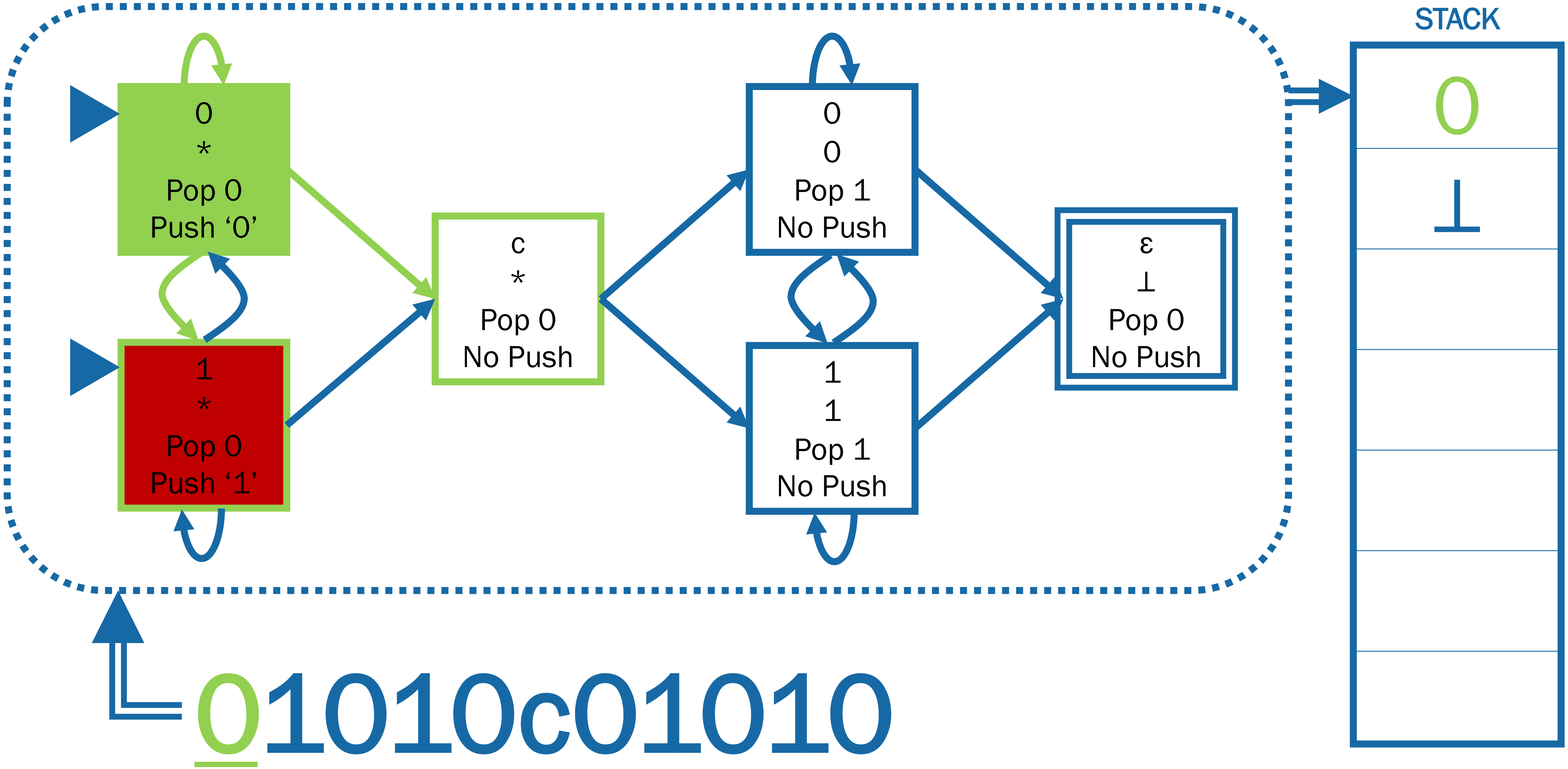


Recognizing Palindromes with a Middle Character

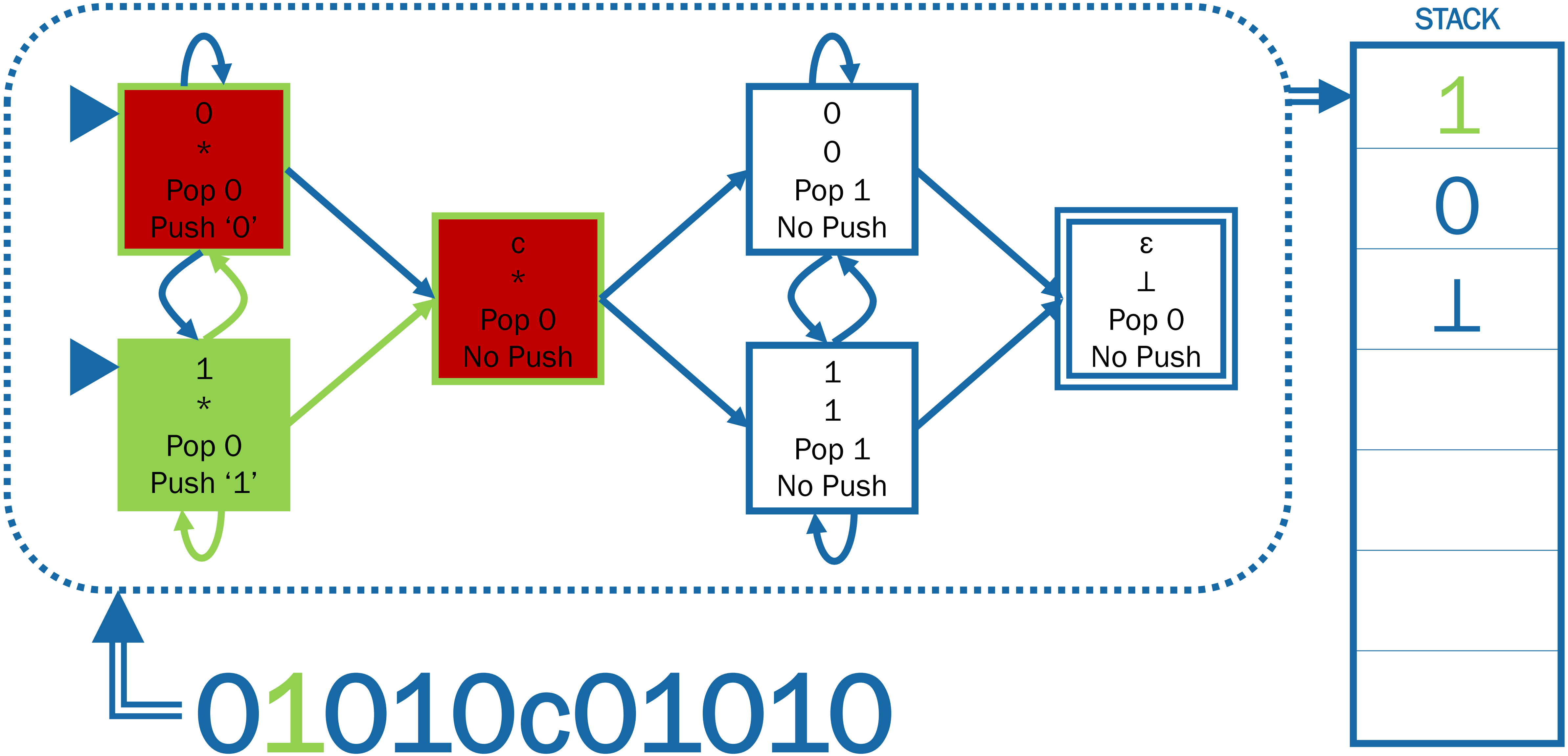


01010c01010

Recognizing Palindromes with a Middle Character

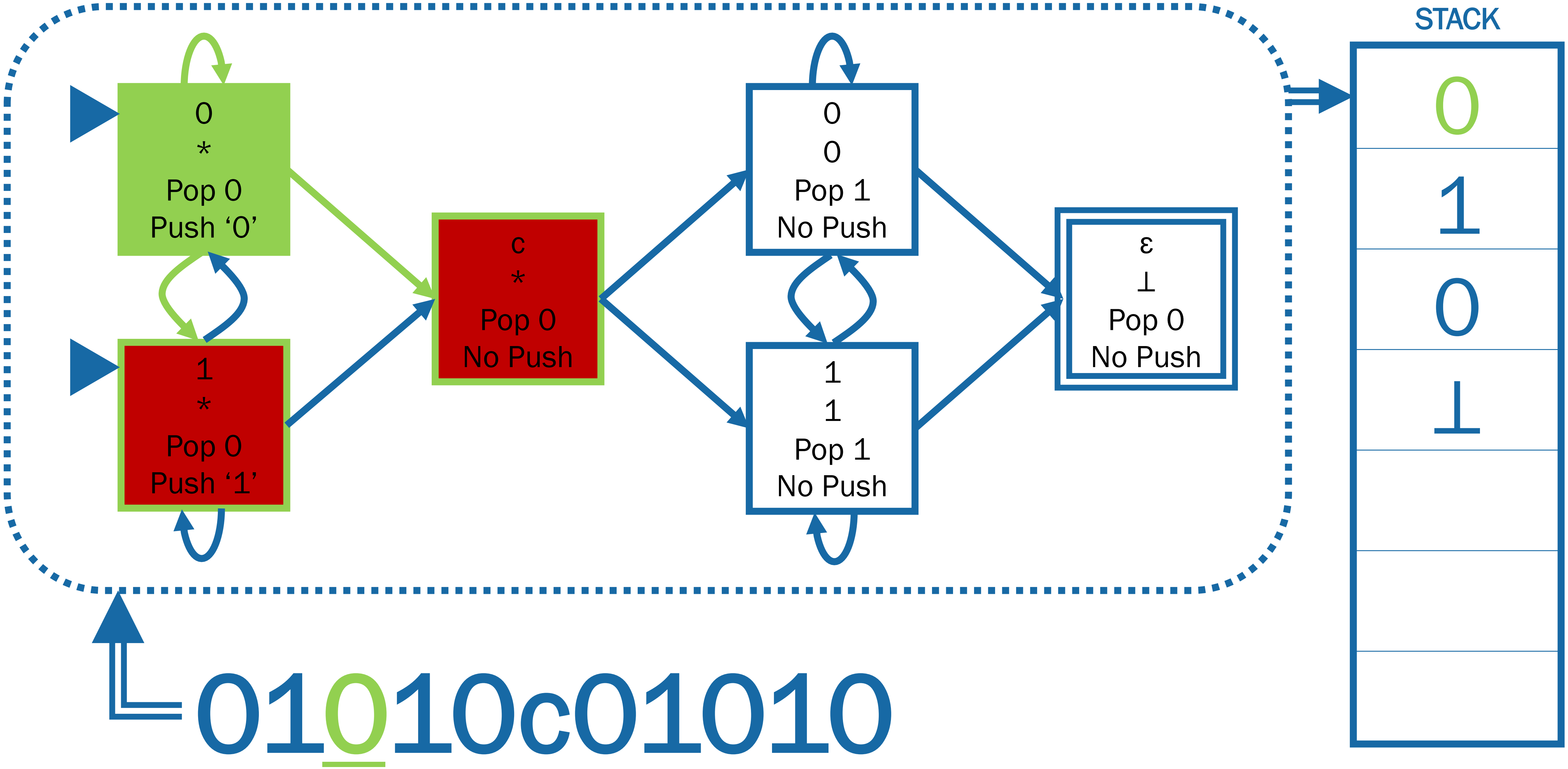


Recognizing Palindromes with a Middle Character

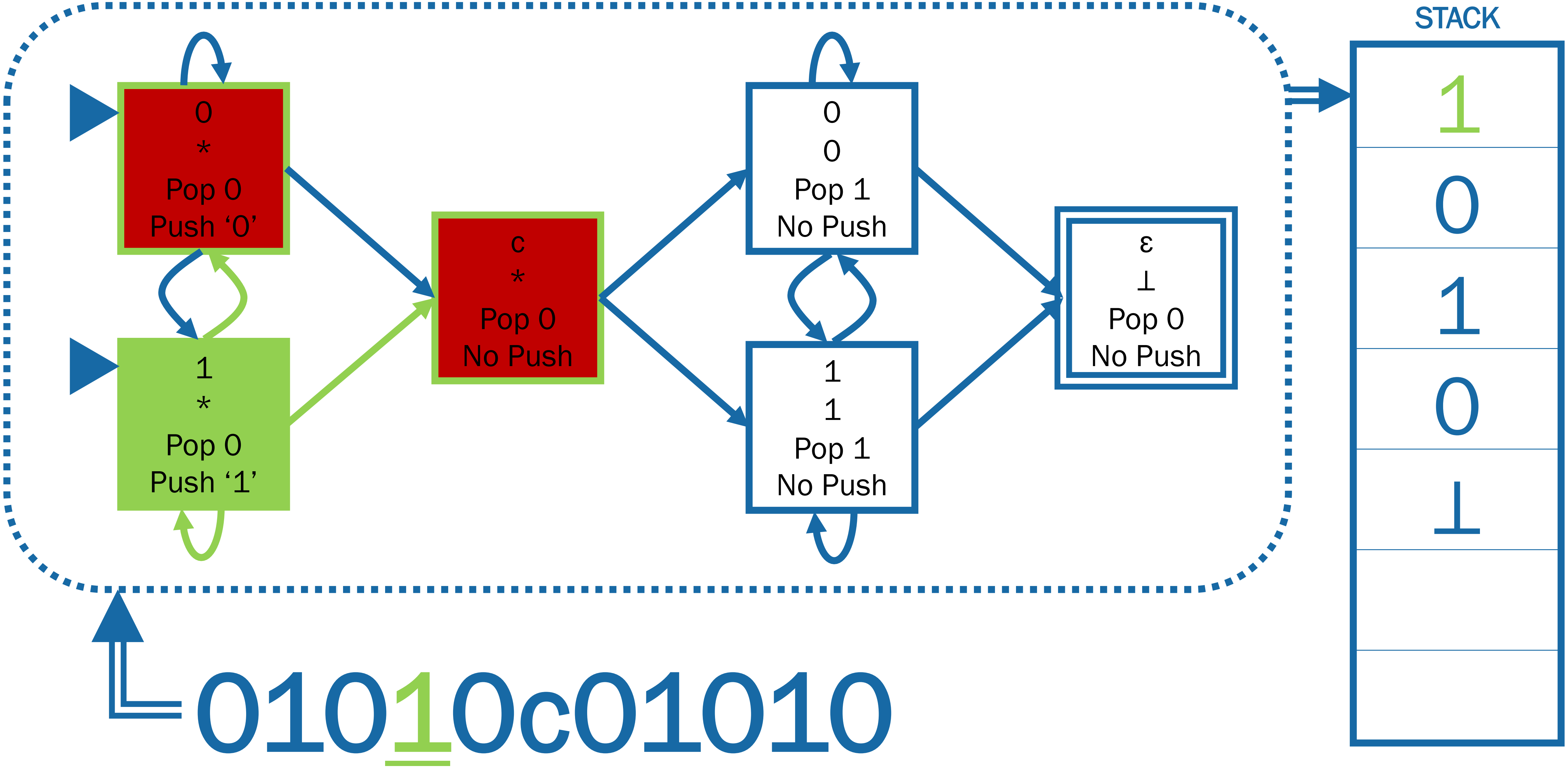


01010c01010

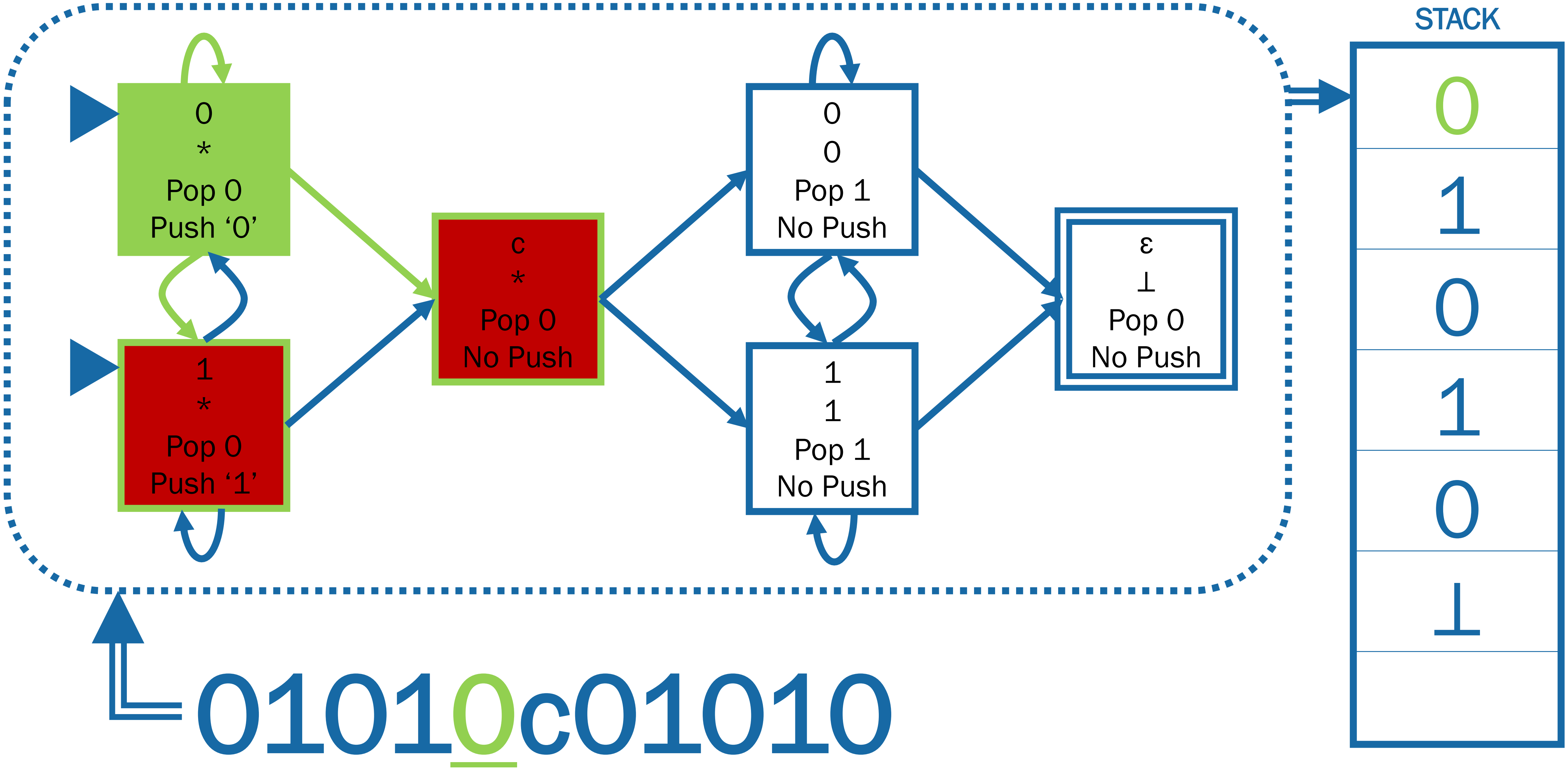
Recognizing Palindromes with a Middle Character



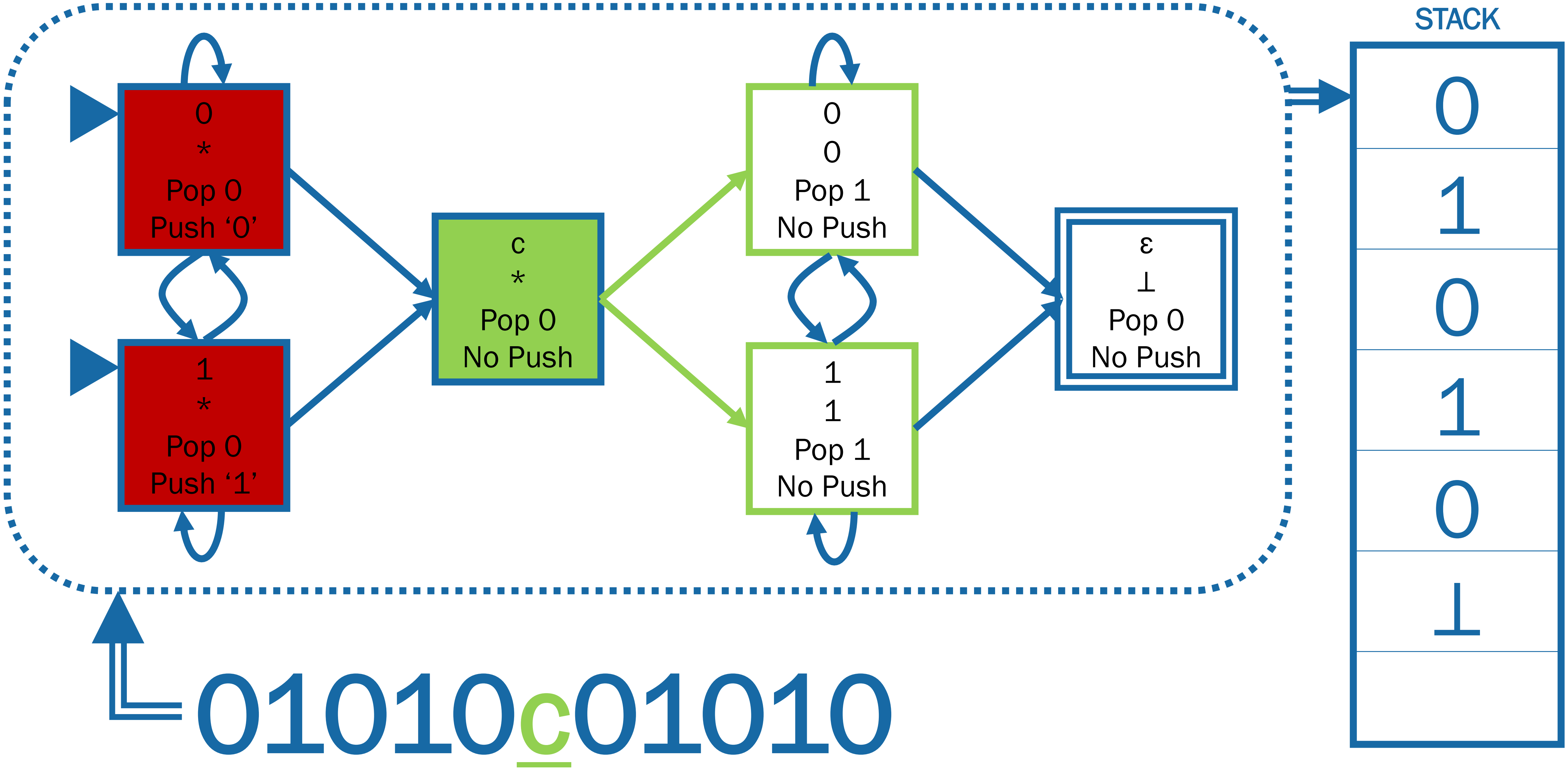
Recognizing Palindromes with a Middle Character



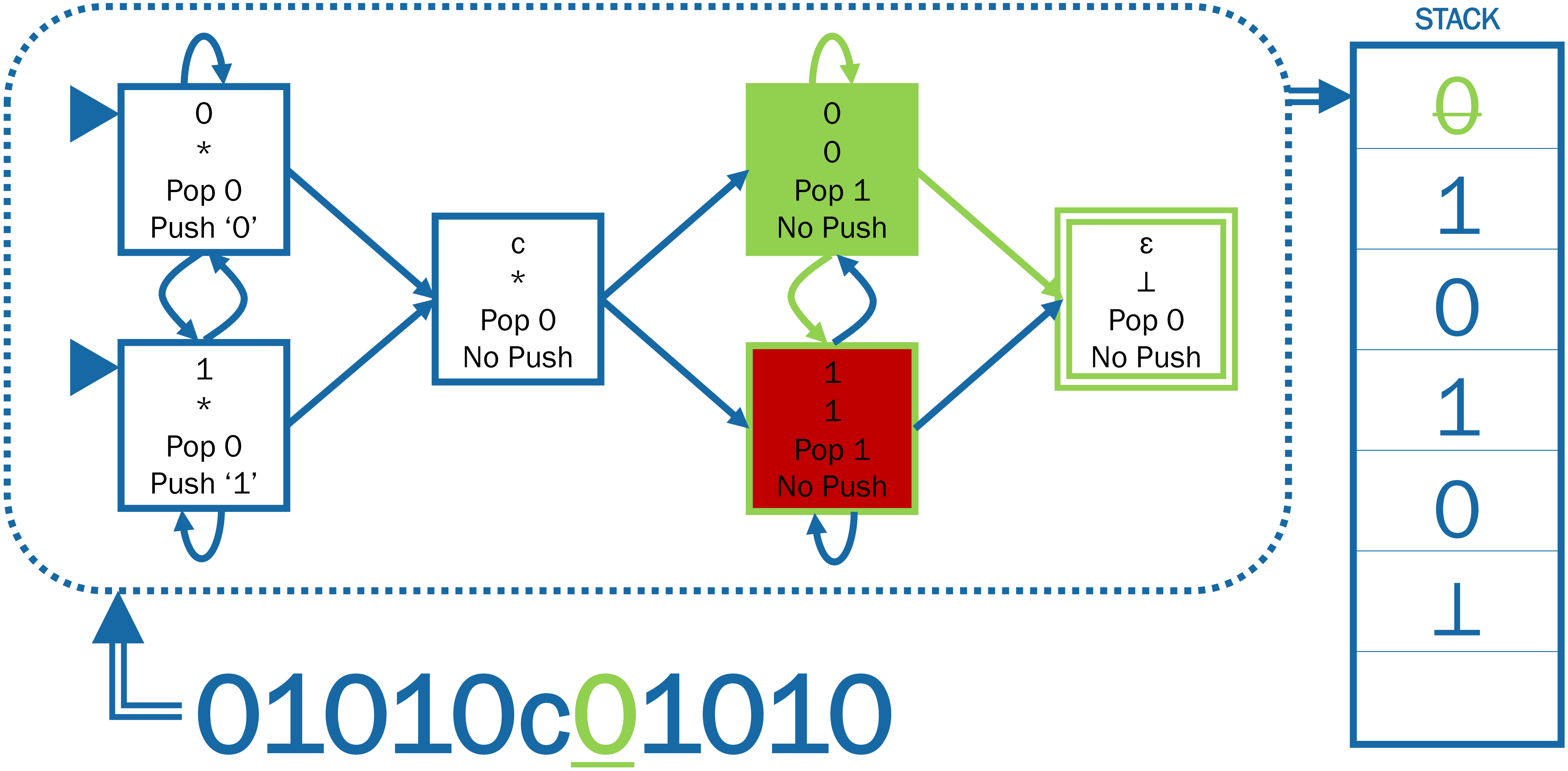
Recognizing Palindromes with a Middle Character



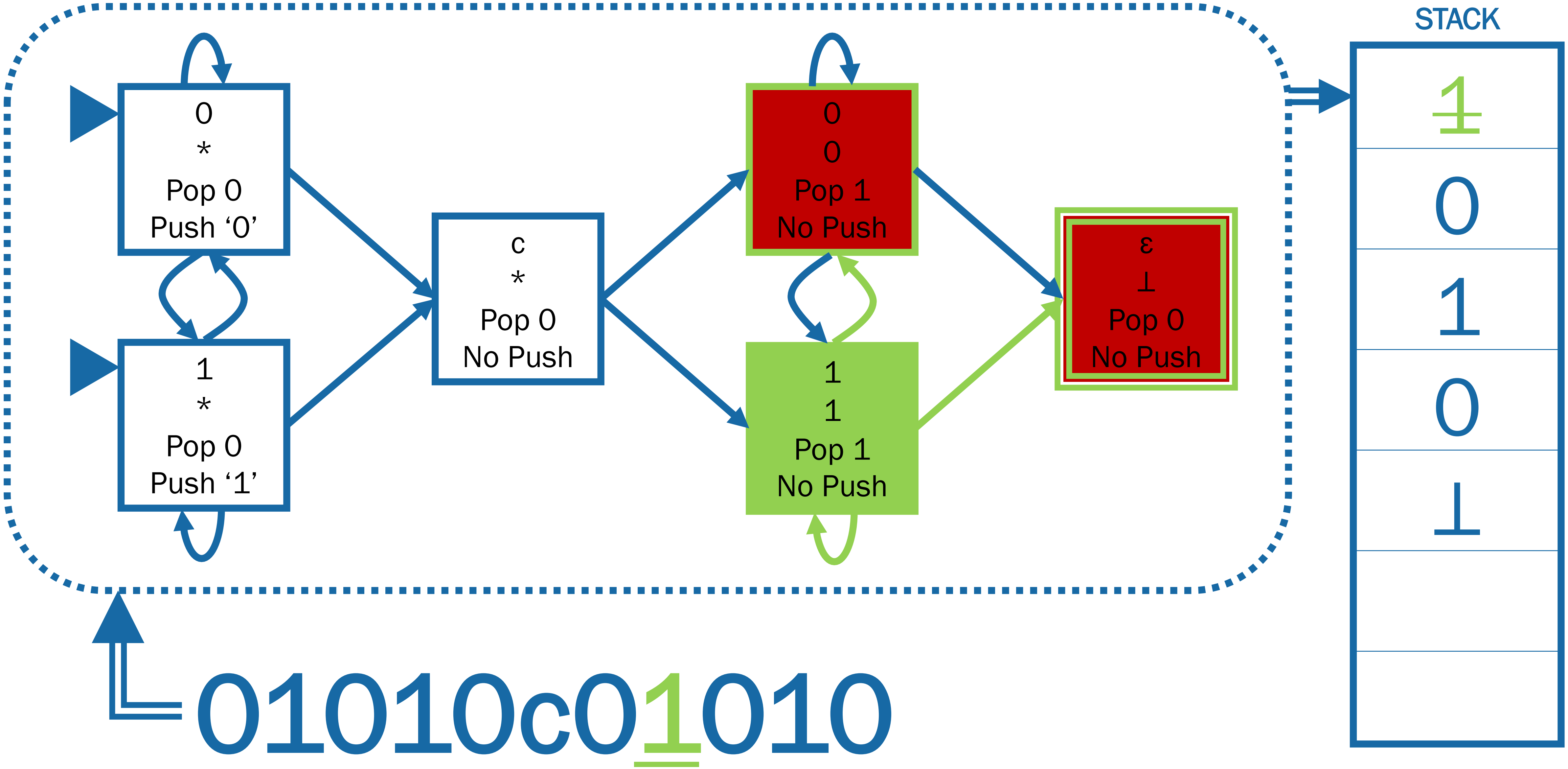
Recognizing Palindromes with a Middle Character



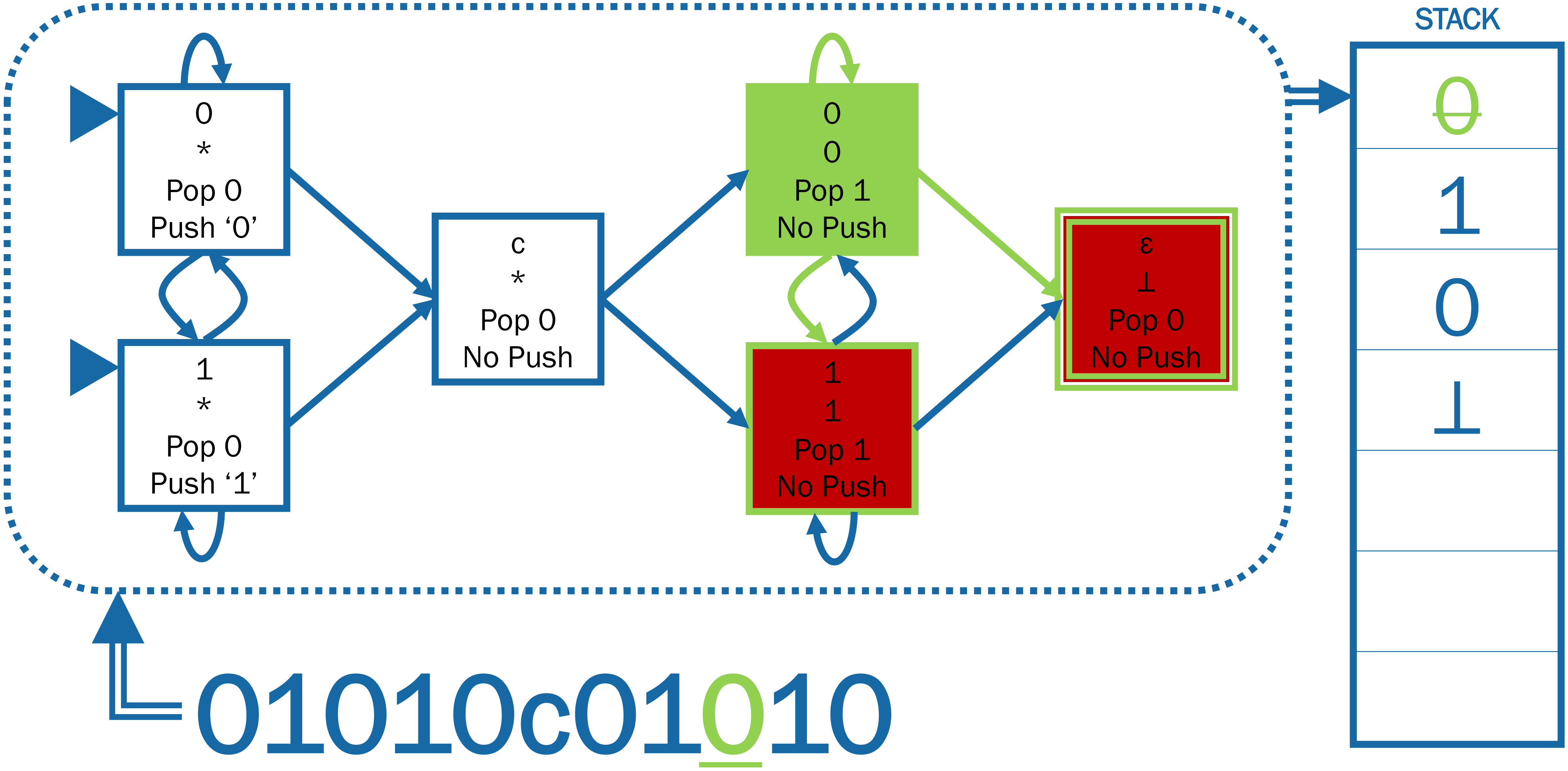
Recognizing Palindromes with a Middle Character



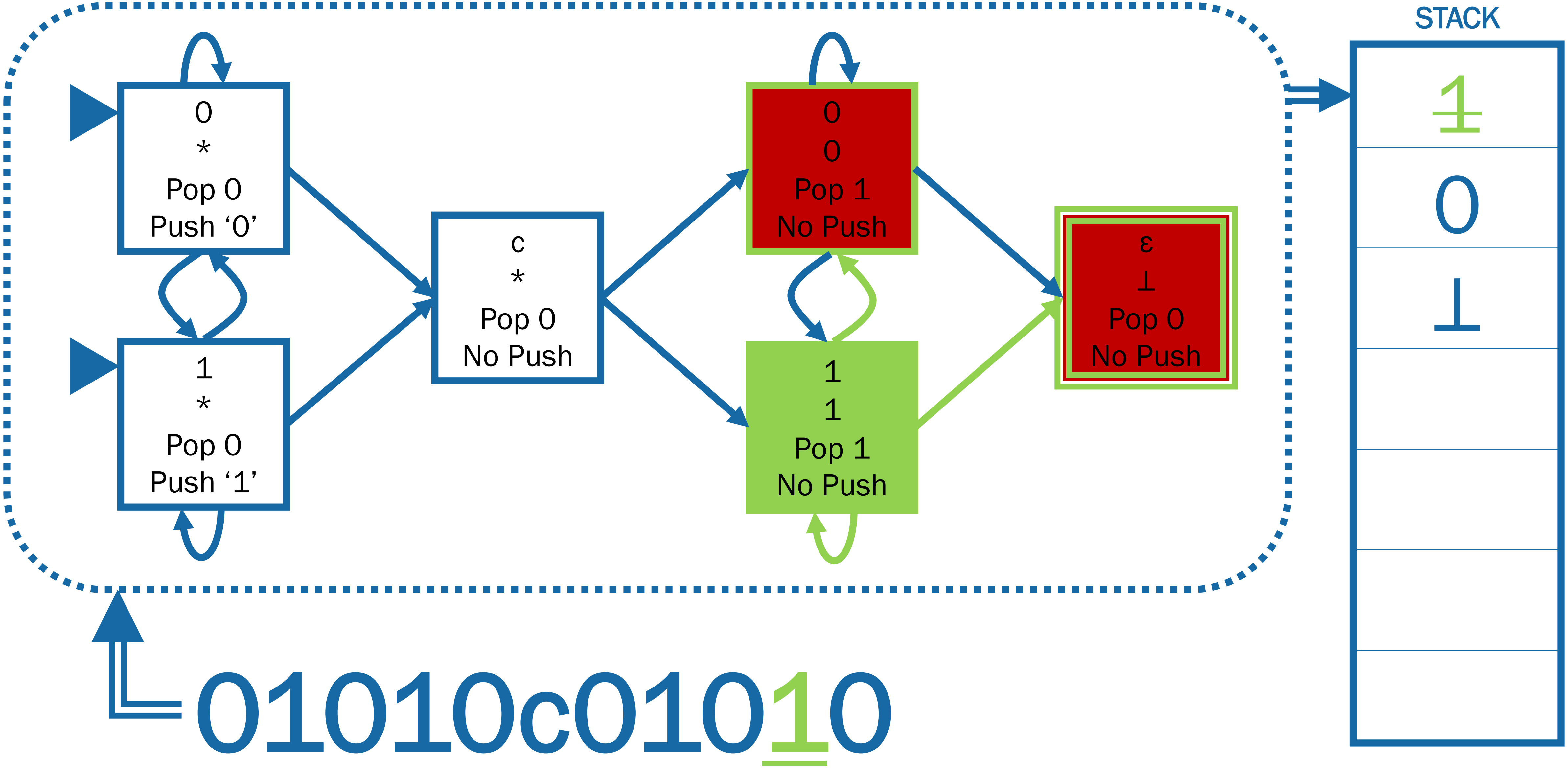
Recognizing Palindromes with a Middle Character



Recognizing Palindromes with a Middle Character



Recognizing Palindromes with a Middle Character



0
*
Pop 0
Push ‘0’

1
*
Pop 0
Push ‘1’

c
*
Pop 0
No Push

0
0
Pop 1
No Push

1
1
Pop 1
No Push

ε
⊥
Pop 0
No Push

STACK

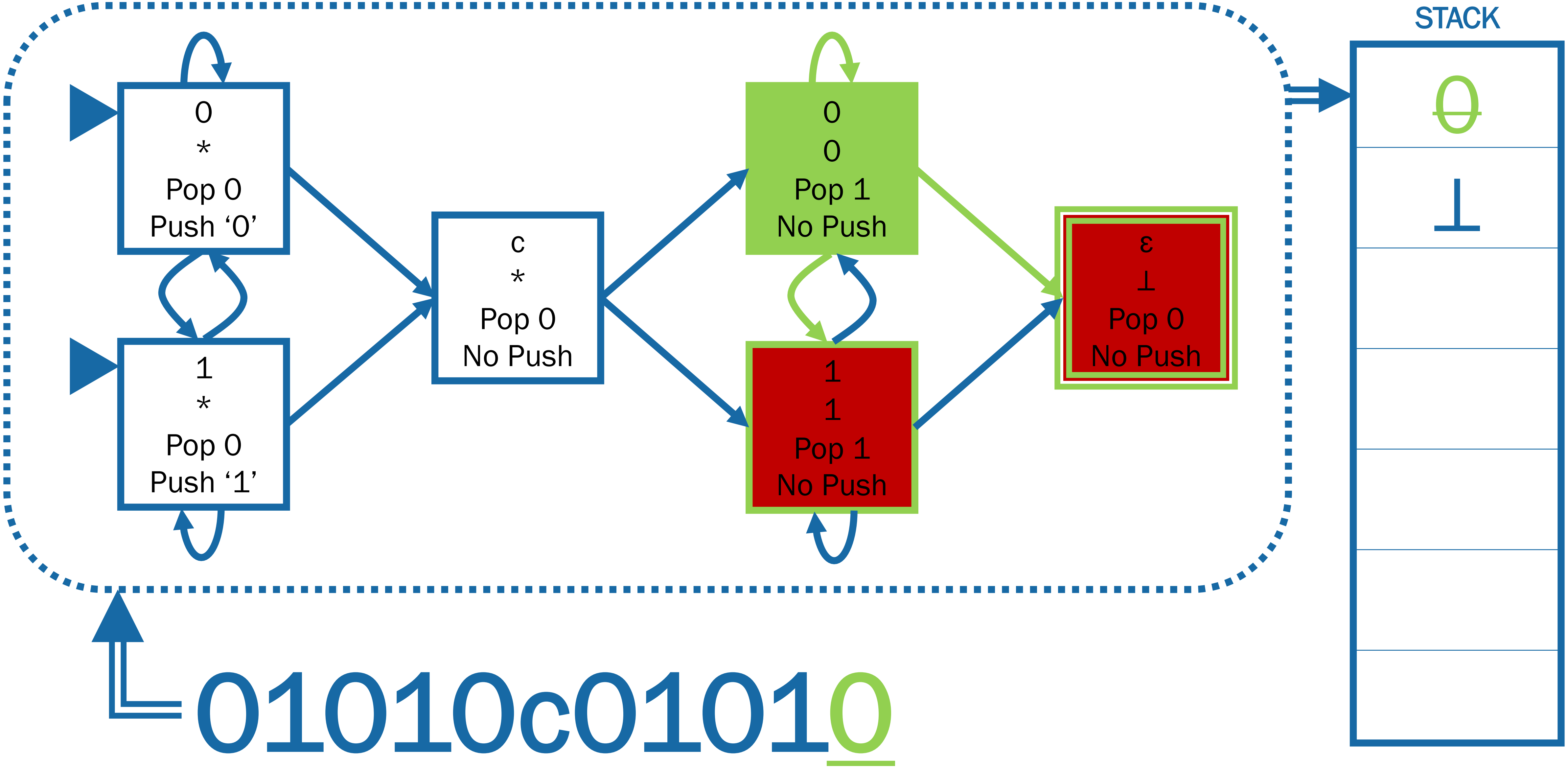
1

0

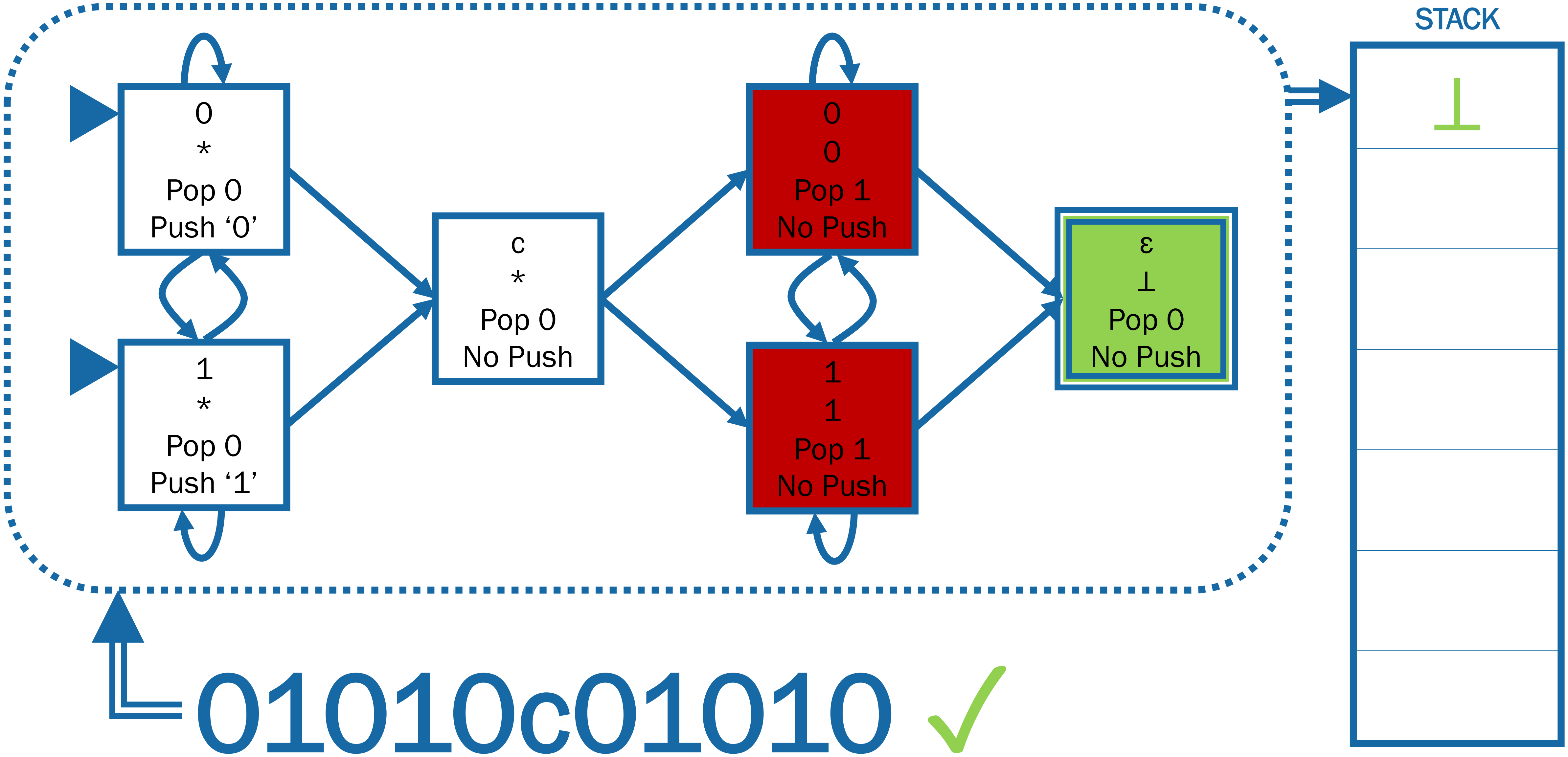
⊥

01010c01010

Recognizing Palindromes with a Middle Character



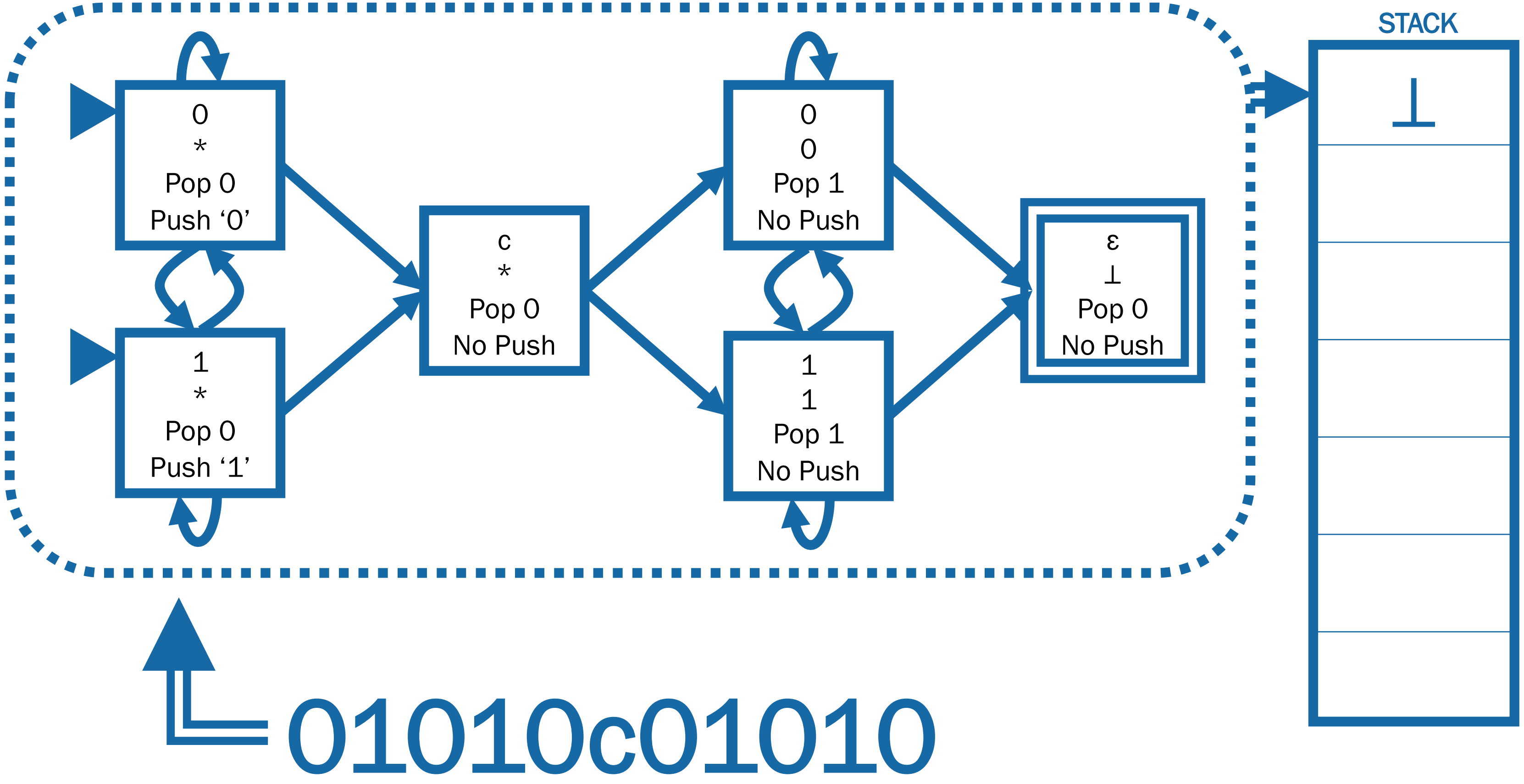
Recognizing Palindromes with a Middle Character



Mapping DPDA Efficiently to Hardware

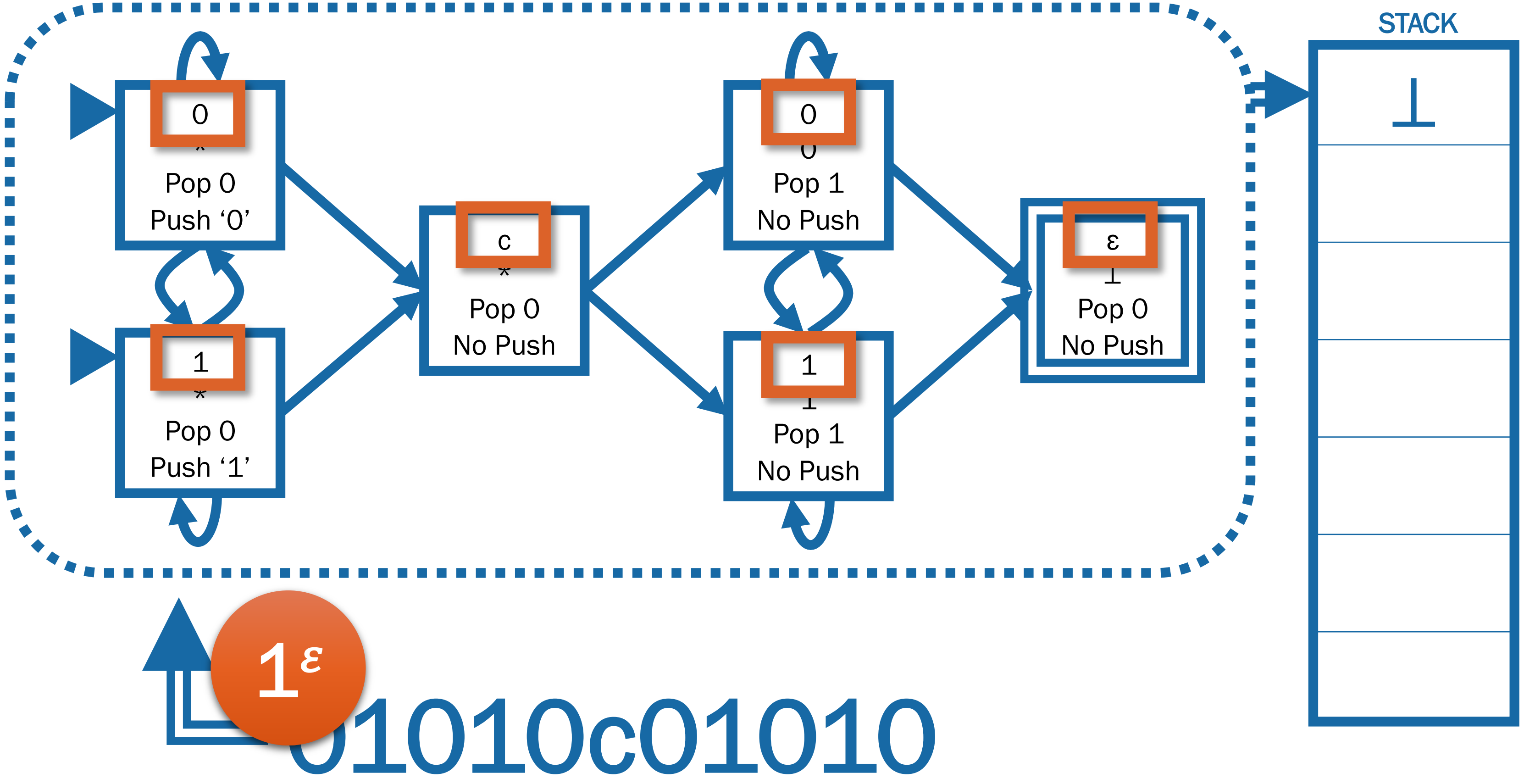
- ASPEN supports homogeneous DPDA
  - All transitions to a state occur on the same input character, top of stack comparison, and stack operation
  - Similar in nature to homogeneous NFAs
- Equal expressive power as standard DPDA
- State increase is quadratic in the worst case with a fixed alphabet
- Allows for efficient mapping to hardware resources
  - Transitions decoupled from input/stack matches

Five Steps of DPDA Execution Per Cycle



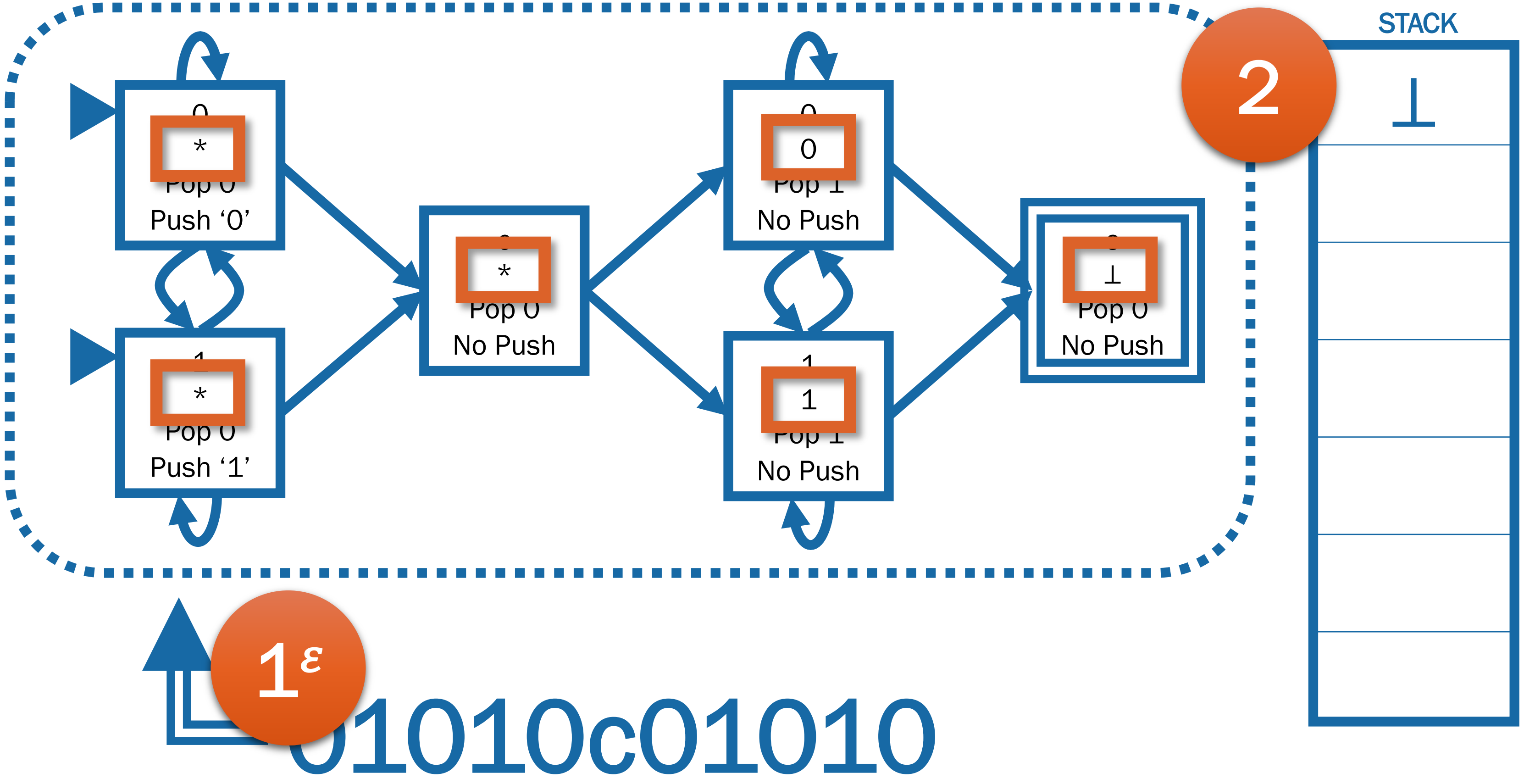
Five Steps of DPDA Execution Per Cycle

1. Input Match$^{\varepsilon}$
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition



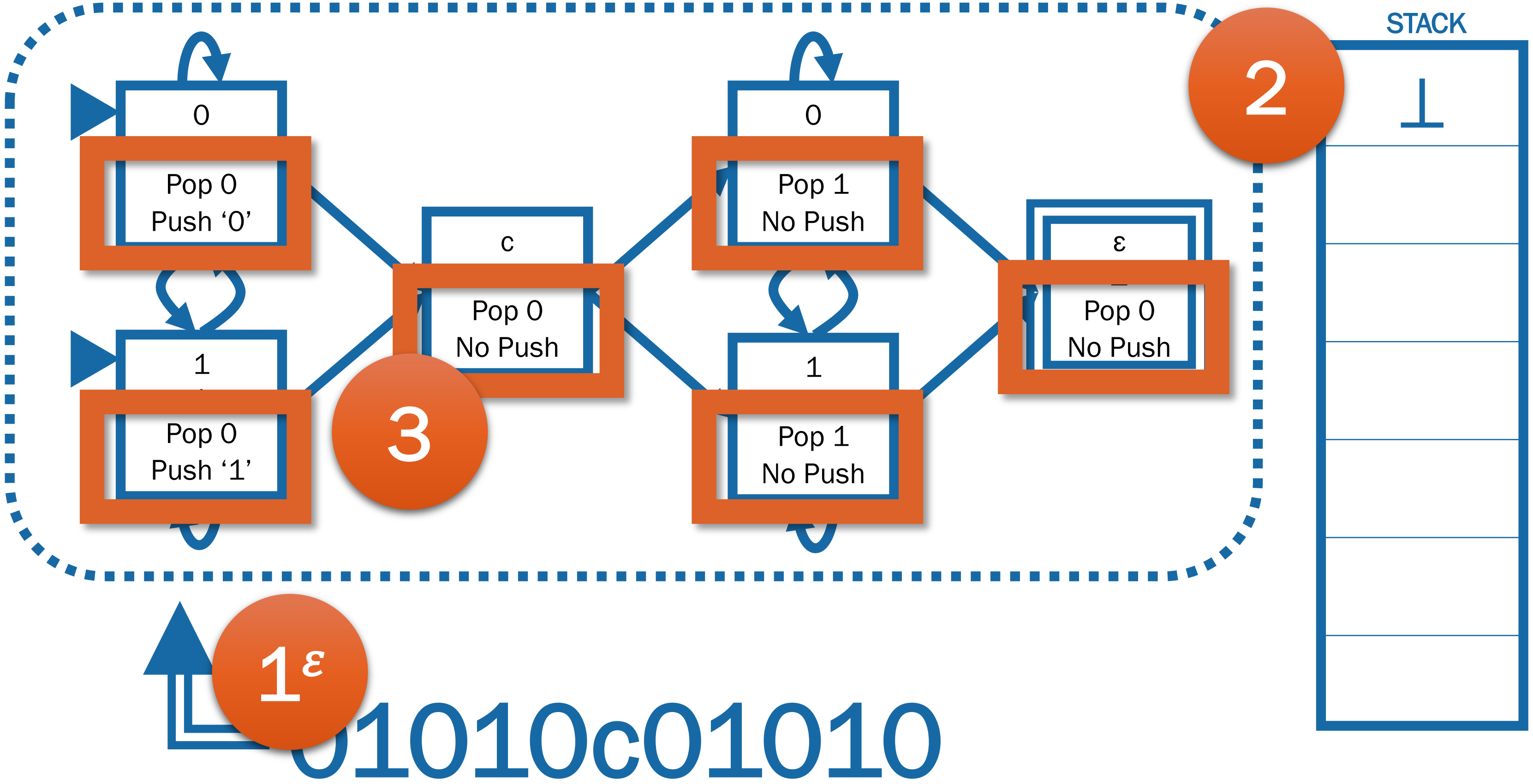
Five Steps of DPDA Execution Per Cycle

1. Input Match$^{\varepsilon}$
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition



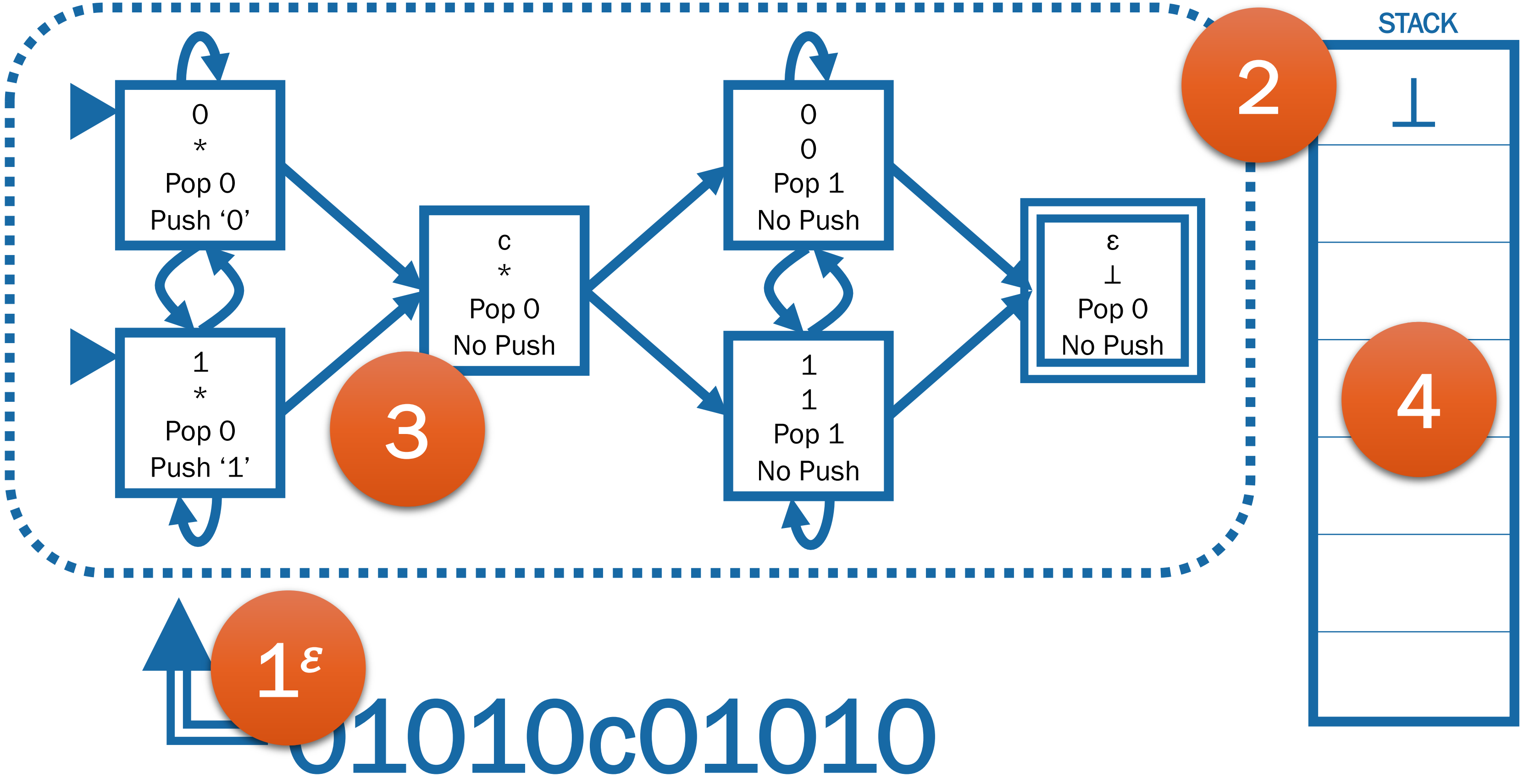
Five Steps of DPDA Execution Per Cycle

1. Input Match$^{\varepsilon}$
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition



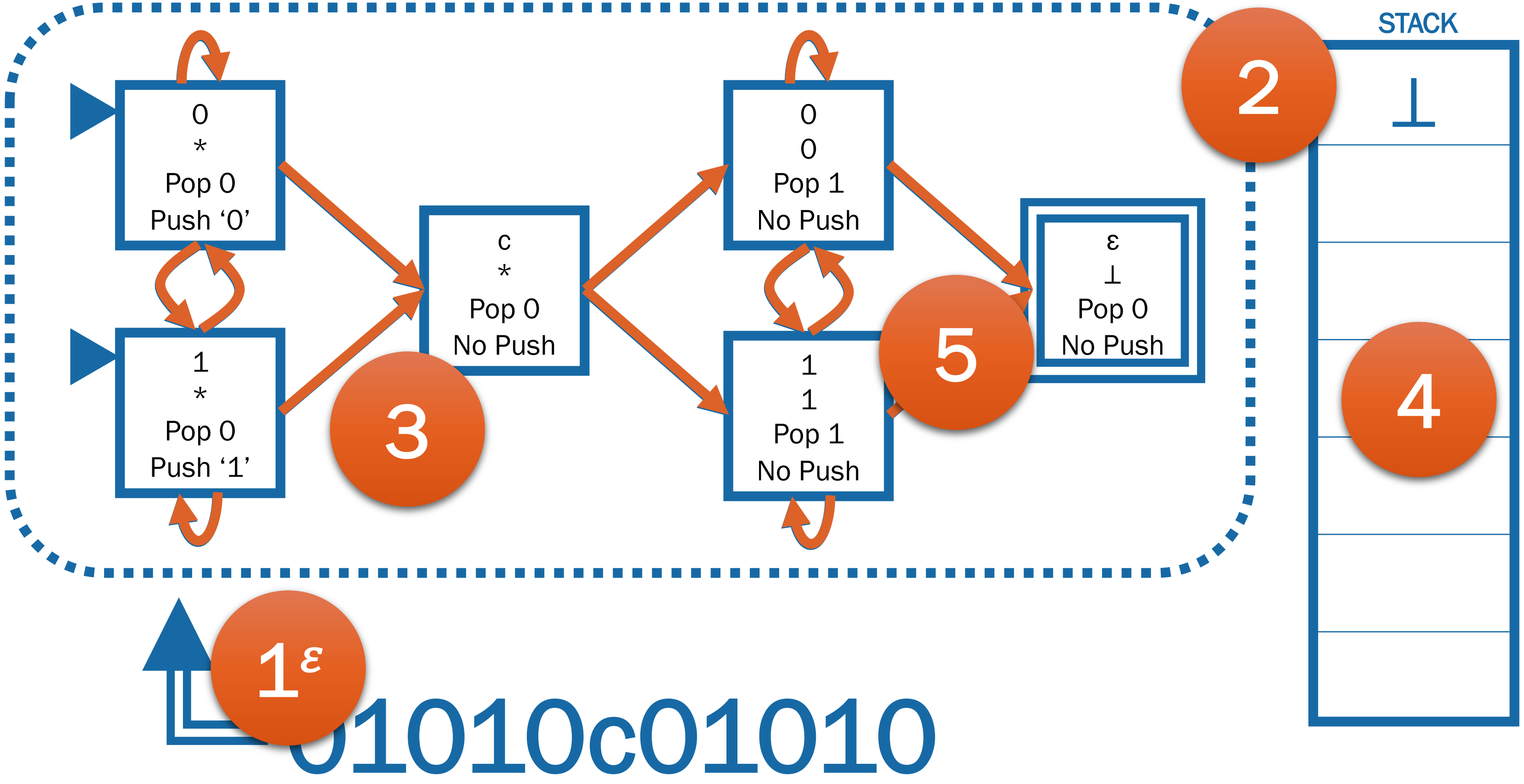
Five Steps of DPDA Execution Per Cycle

1. Input Match$^{\varepsilon}$
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition

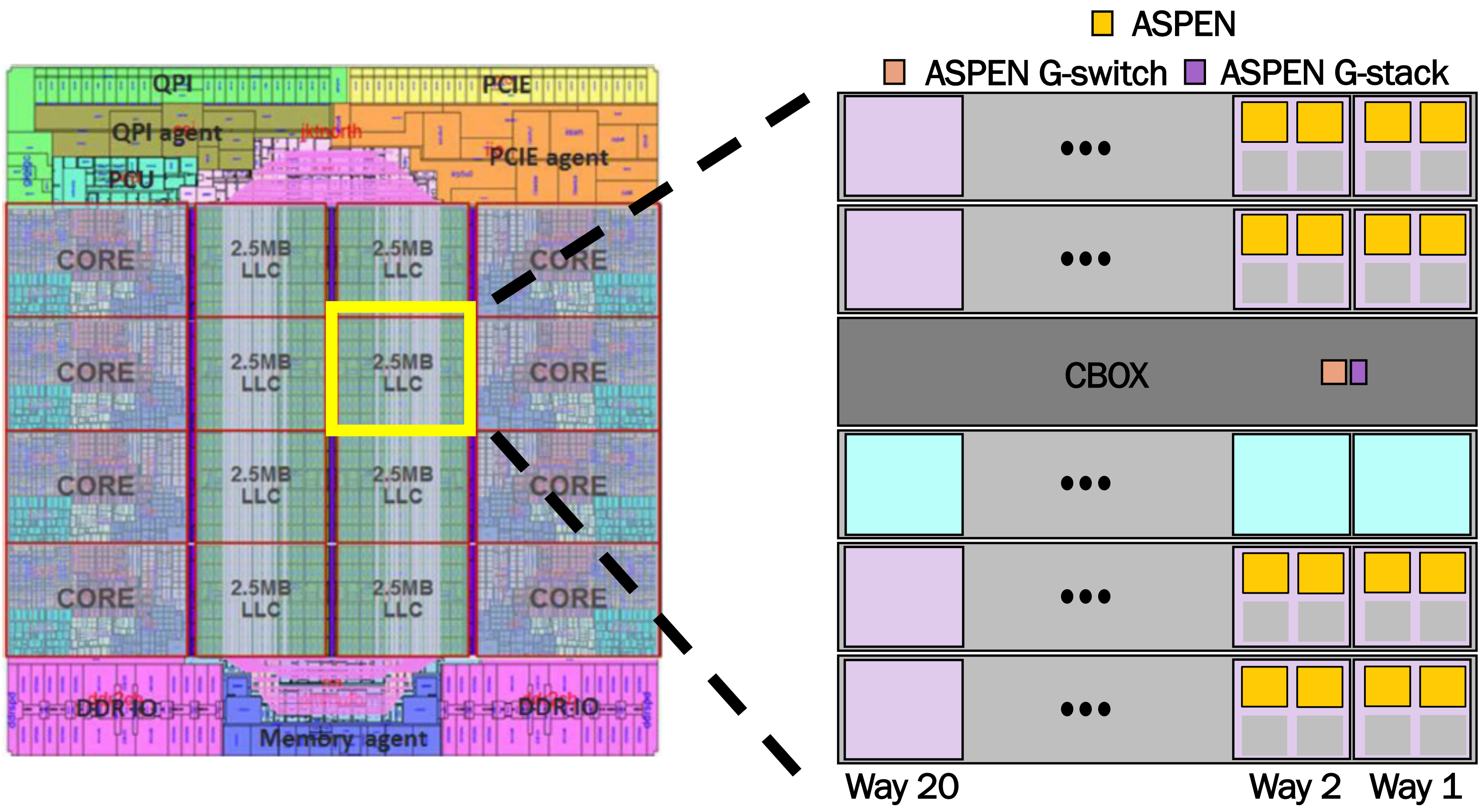


Five Steps of DPDA Execution Per Cycle

1. Input Match$^{\varepsilon}$
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition



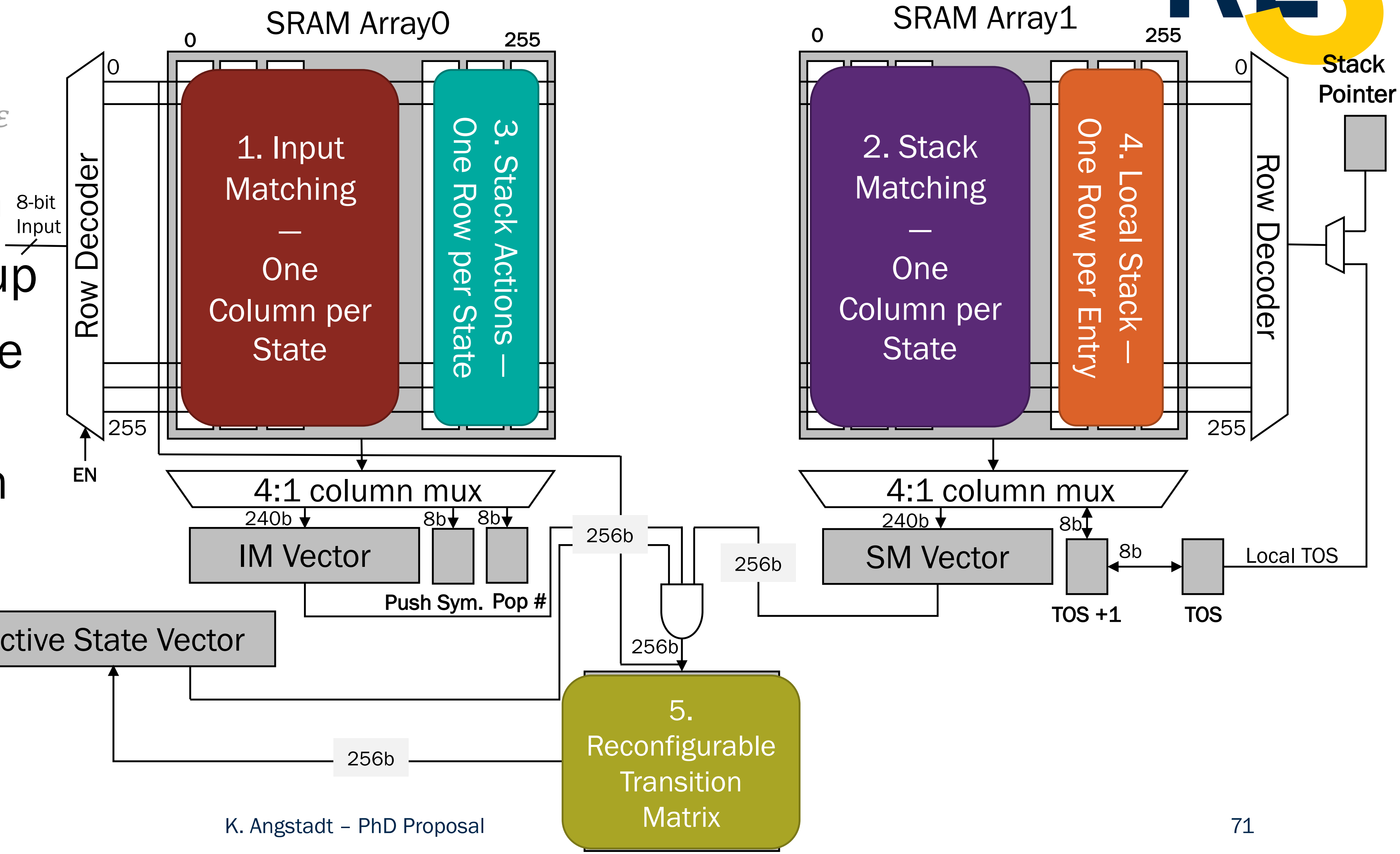
Where is ASPEN?



- ASPEN uses 2 arrays per bank
- 240 states per bank
- Full connectivity within bank
- Global switch and stack in CBOX for large DPDA

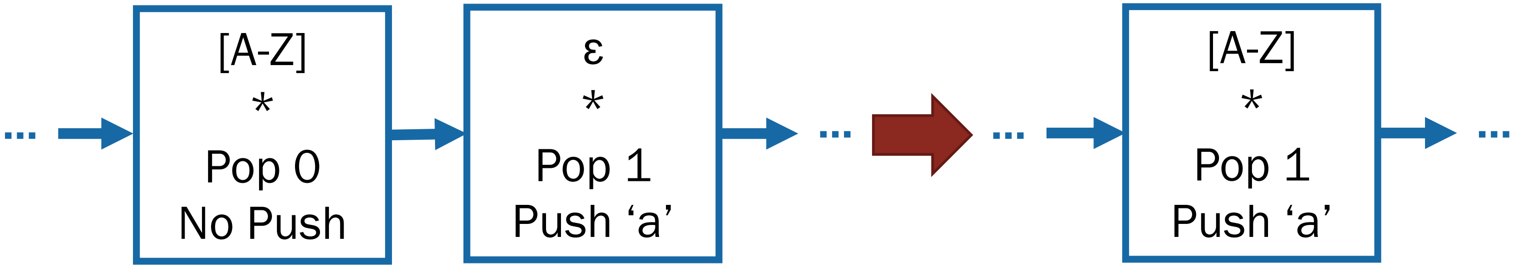
ASPEN Datapath — 240 States in 2 SRAM Arrays

1. Input Match$^{\varepsilon}$
2. Stack Match
3. Action Lookup
4. Stack Update
5. State Transition



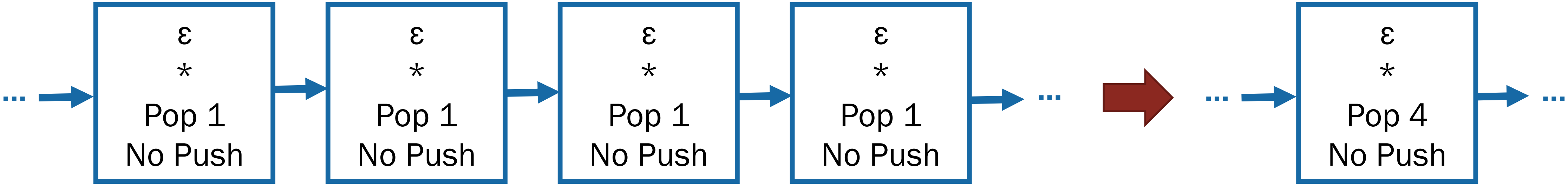
Optimizations

Epsilon Merging



Goal: Reduce the number of stalls while processing input

Multipop

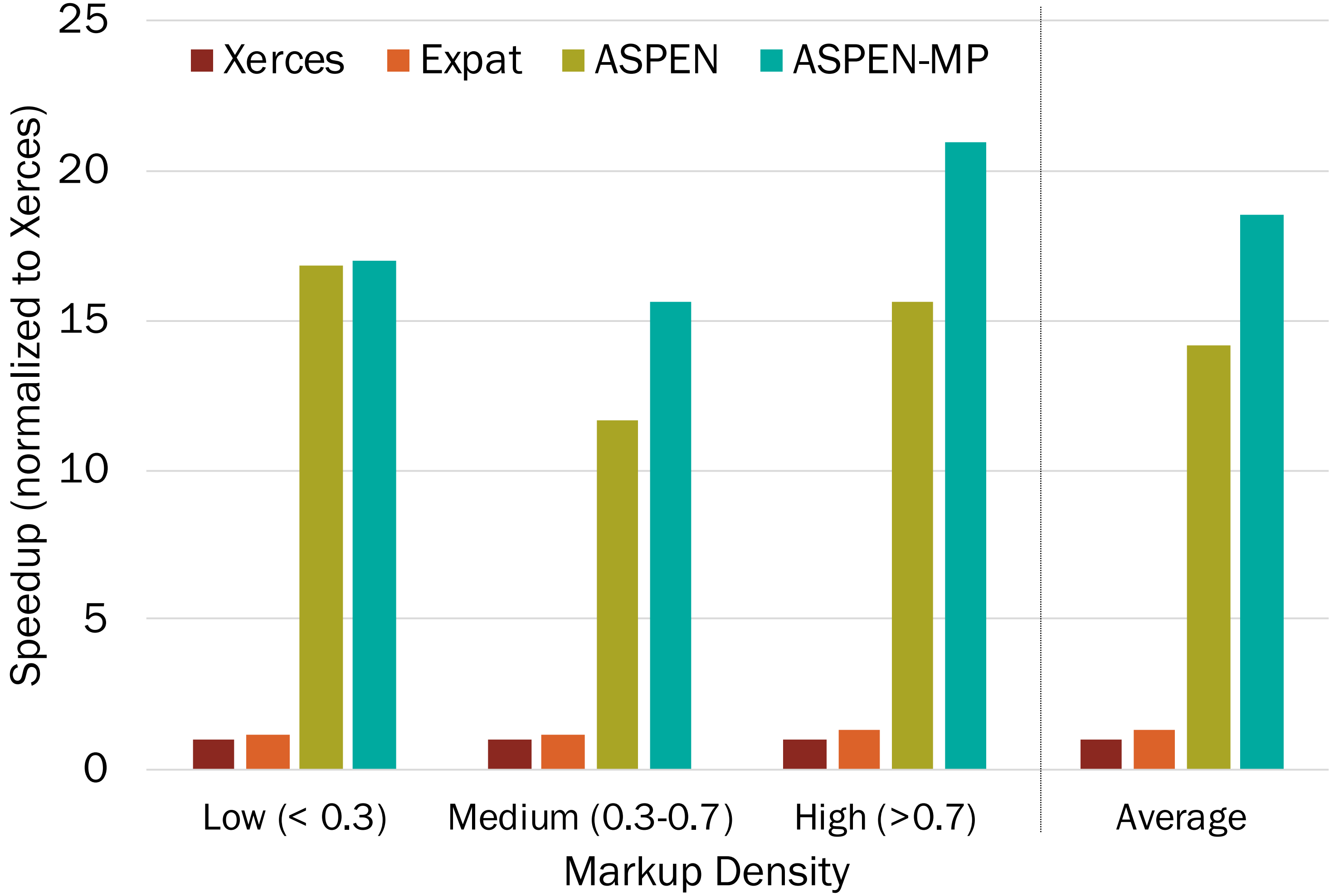


- Average of 65% reduction in epsilon states

Experimental Methodology

- Compile existing grammars (**expressive power** and **legacy support**)
  - Cool, JSON, XML, Dot, etc.
  - Measure hardware utilization, including optimization improvement
- Real-world application case studies (**performance and scalability**)
  - Stress different aspects of architecture
  - Large DPDA with Global Stack (high connectivity)
  - Small DPDA with Local Stacks (sparse connectivity)
  - Compare runtime performance to state-of-the-art baselines
- **Success:** Outperform state-of-the-art for indicative applications while not exceeding power thresholds for modern CPUs

Application 1: XML Parsing



- Benchmarks: Parabix, Ximpleware, UW XML
- ASPEN is 13-18x faster (on average) than popular CPU Parsers
- Performance did not vary significantly with complexity of XML
- Optimizations and tokenization hide ε-stalls

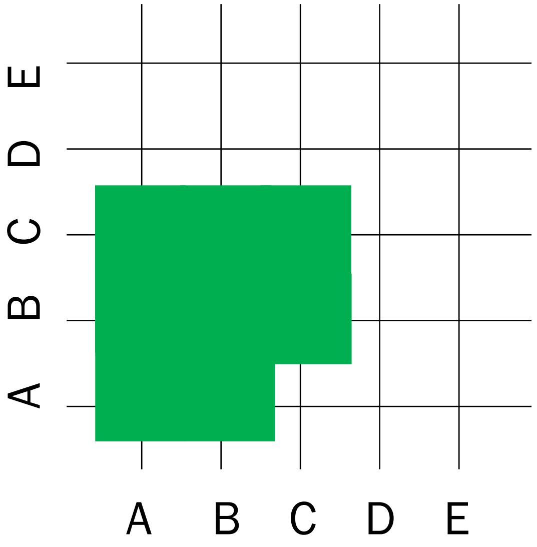
Application 2: Anomaly Intrusion Detection

- Identify abnormal program executions by monitoring memory access patterns
- Hypothesis: normal executions will exhibit consistent, similar memory access patterns
- Cache is perfect!
  - Transparently snoop on memory access
  - Automata accelerators repurpose address lines for data input
- Revisits “a sense of self” work by Forrest et al. using system calls for signatures of normal behavior

MemIDS Basic Approach: Sliding Windows

- Successful approaches (syscalls) use simple automata
- Build dictionary of observed behavior
- Sliding window of sequences to determine normal/abnormal

Training Memory AABACCBBC

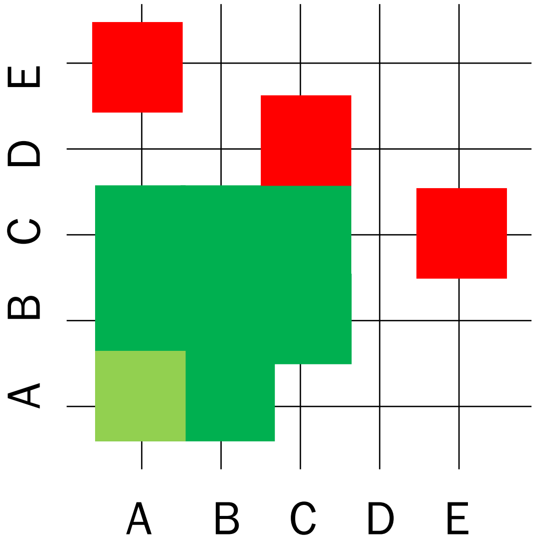


MemIDS Basic Approach: Sliding Windows

- Successful approaches (syscalls) use simple automata
- Build dictionary of observed behavior
- Sliding window of sequences to determine normal/abnormal

Training Memory AABACCBBC

Testing Memory AAECD



MemIDS Research Questions

- Can we successfully detect the execution of different programs?
- How many bits of address are needed for a meaningful dictionary?
- What classes of abnormal behavior can we detect?
  - Speculative execution-based attacks
  - Stealthy malware
  - Information leaks
- What do we do with this information?
  - Block access
  - Delay access
  - Return bogus values

Research Effort 3 Summary

- Expand expressive power of automata accelerators with new in-cache architecture for DPDA
- New homogeneous DPDA variant for efficient hardware implementation
- Optimizing compiler to support existing grammars
- Preliminary results demonstrate improved performance for XML parsing
- Ongoing work to leverage accelerator for intrusion detection applications

Adapting Legacy Code for Execution on Hardware Accelerators

Research Effort 4

Legacy Code in the Age of Hardware Accelerators

- Legacy code typically cannot be directly compiled for accelerators
- Learning a new programming model is costly and slows rate of adoption of new accelerators
- May want to “try out” new hardware with existing software
  - No training on new hardware
  - Limited time or resources to allocate

Research Effort 4

RE4

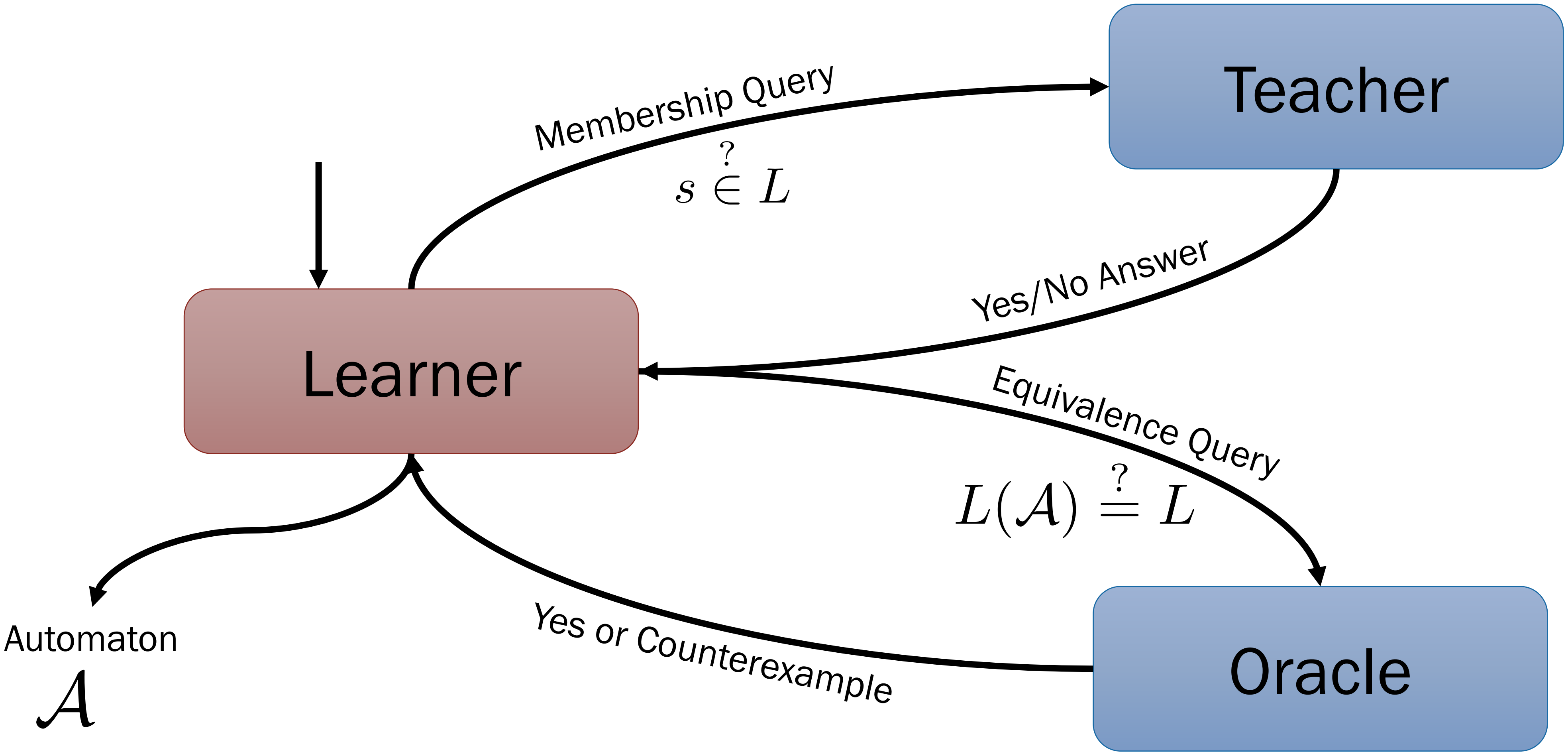
Goal: Reduce the burden on developers tasked with porting legacy code to execute on hardware accelerators

Hypothesis: An algorithm that learns a set of finite automata from a legacy source code kernel, using a combination of automata learning and formal methods, can correctly synthesize a functionally-equivalent kernel computation, reduce the manual annotation and refactoring efforts of human developers, and efficiently represent real-world applications.

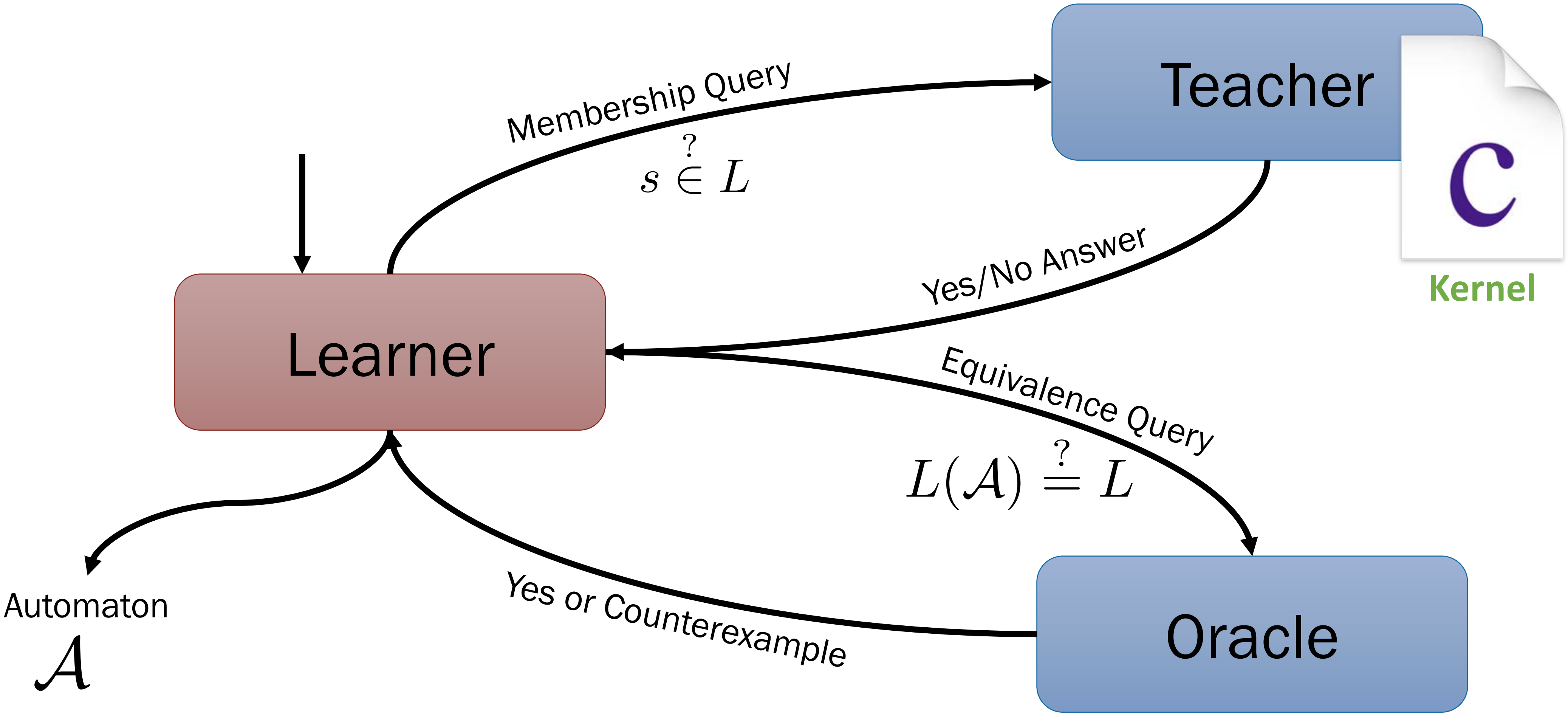
Problem Statement

- Input: function kernel : string -> bool
- Assumptions:
  - Function decides a regular language
  - Source code for function is available
- Output: finite automaton with the same behavior on “all” inputs as kernel

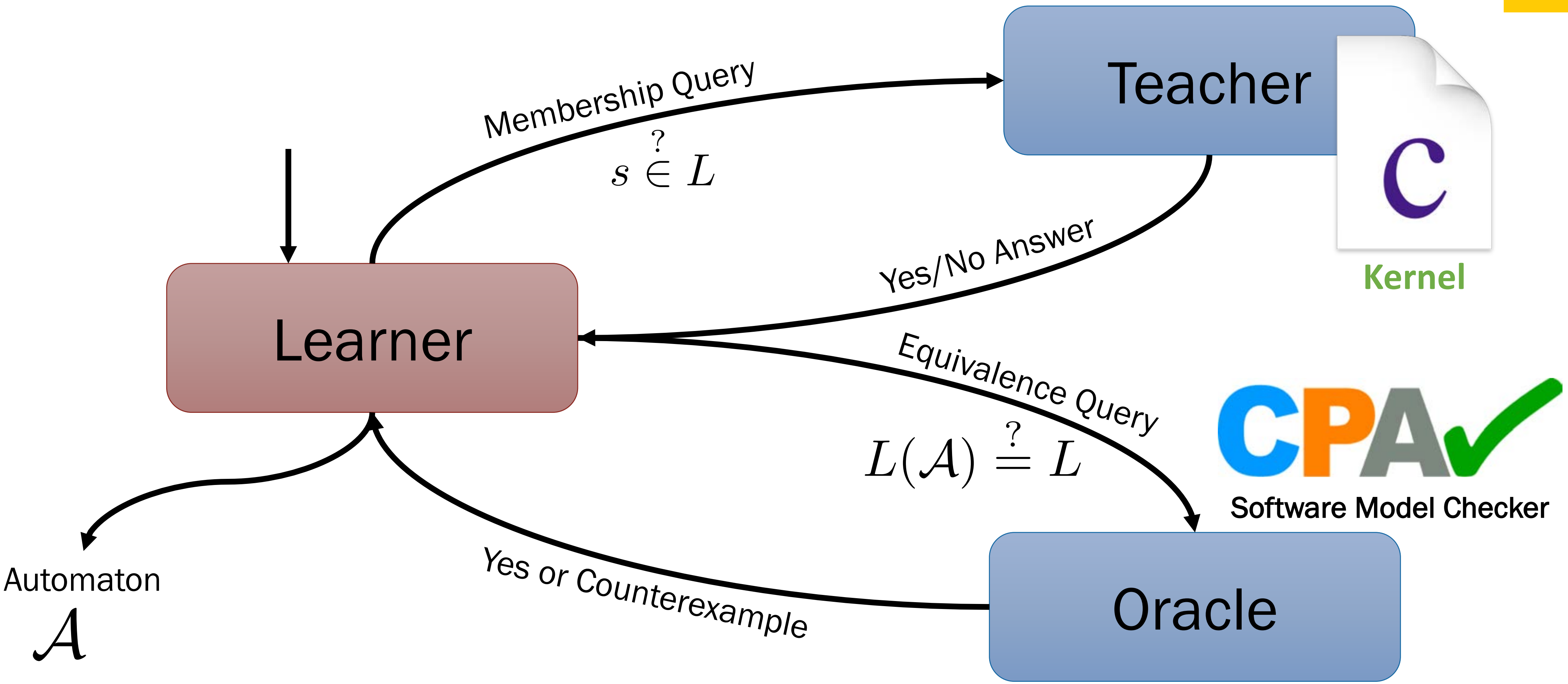
Angluin-Style Learning (L*)



Angluin-Style Learning (L*)



Angluin-Style Learning (L*)



Equality Checking as Software Verification

- Explores control flow graph looking for property violations
  - Success finding variety of bugs (e.g., double-free, locking violations, etc.)
  - Used in industry for driver verification
- Properties specified as automata
  - **Research:** How do we leverage this?
  - Adapt L*-learned candidate automata to specifications
  - Verify source kernel: violations are counterexamples
- **Research:** identify appropriate verification strategies

Mitigating Risk: Speculative Research

- Restrict kernel functions
  - Decide regular languages
  - Streaming access to input data
  - Time permitting: relax these assumptions
- Allow for approximate solutions
  - Measure accuracy with respect to time
  - Reduce impact of error with examples
- User-provided annotations
  - Guide software verification
  - Note “transitions” in kernel

Experimental Methodology

- Seek benchmark suite of existing kernel functions
- Is it possible for our algorithm to adapt kernels? (**legacy support**)
- Measure time needed to learn automata and their size in states/hardware resources (**scalability**)
  - **Success:** Run over the weekend and fit on commercial FPGA
- Measure accuracy of approximate solutions
  - Use provided test suites and augment with test input generation
  - **Success:** use of example inputs improves accuracy

Research Effort 4 Summary

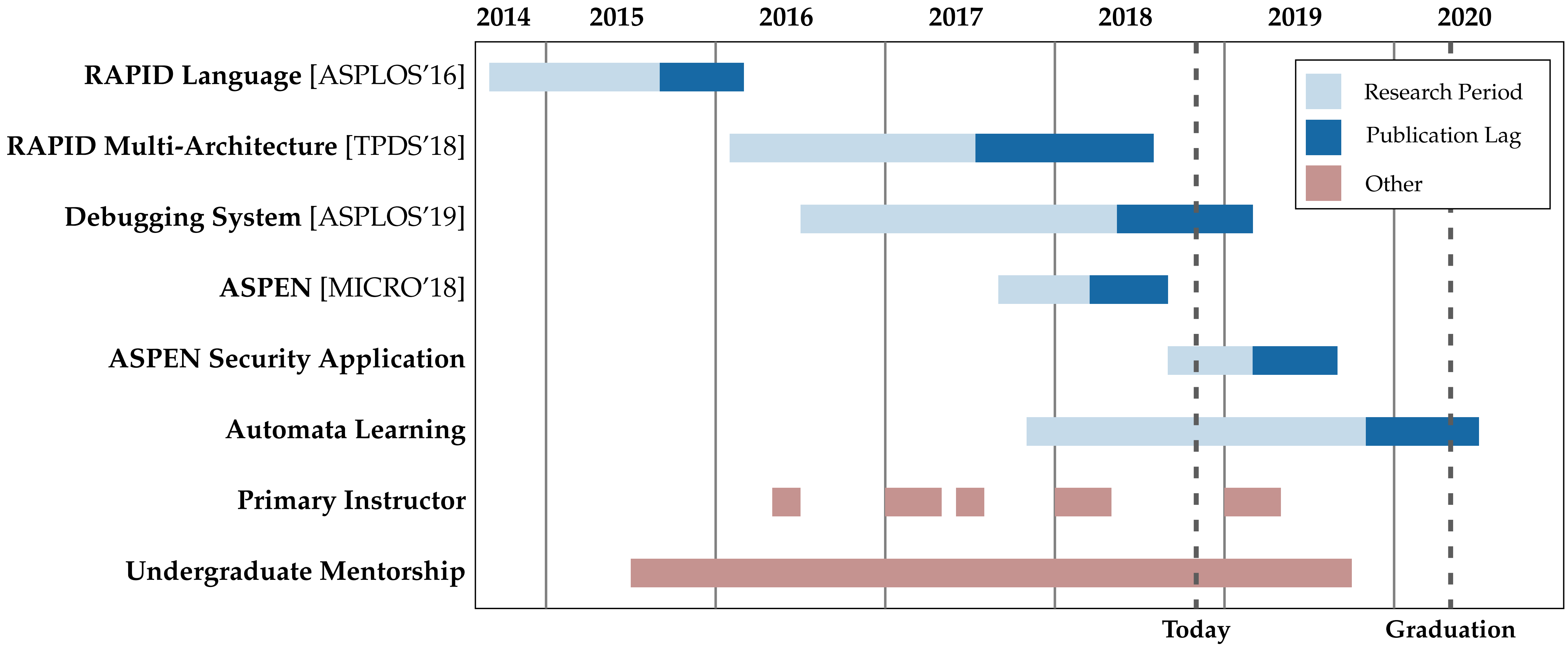
RE4

- Design algorithm to adapt legacy kernels for execution on hardware accelerators
- Adopt Angluin-style learning approach
- Convert equivalence queries to software verification tasks
- Speculative research
  - Limit kernels to decide regular languages
  - Annotations to guide software verification
  - Approximate solutions and guiding example inputs

Proposal Overview

- Proposed Research Efforts
  - High-Level Programming Language: RAPID
  - High-Speed, Interactive Debugger for Hardware Accelerators
  - In-Cache Accelerator for Pushdown Automata
  - Adapting Legacy Code for Execution on Hardware Accelerators
- Candidate Schedule
- Conclusion/Discussion

Proposed Research Schedule



Typical Venues

Computer Architecture

- MICRO (March/April)
- HPCA (July/August)
- ASPLOS (August)
- TPDS (Journal)

PL and Software Engineering

- ASE (April)
- POPL (July)
- ICSE (August/September)

Publications Supporting Proposed Research

1. Matthew Casias, **Kevin Angstadt**, Tommy Tracy II, Kevin Skadron, Westley Weimer. Debugging Support for Pattern-Matching Languages and Accelerators. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, Providence, Rhode Island, 2019. ACM, to appear. (21% acceptance rate)
2. **Kevin Angstadt**, Jack Wadden, Westley Weimer, and Kevin Skadron. Portable Programming with RAPID. In *Transactions on Parallel and Distributed Systems*, to appear. IEEE. (4.181 journal impact factor)
3. **Kevin Angstadt**, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, Reetuparna Das. ASPEN: A Scalable In-SRAM Architecture for Pushdown Automata. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, Fukuoka, Japan. 2018. IEEE. (21% acceptance rate)
4. **Kevin Angstadt**, Jack Wadden, Vinh Dang, Ted Xie, Dan Kramp, Westley Weimer, Mircea Stan, and Kevin Skadron. MNCaRT: An Open-Source, Multi-Architecture Automata-Processing Research and Execution Ecosystem. In *Computer Architecture Letters*, vol. 17, no. 1, pp. 84-87, Jan.-June 1 2018. IEEE. (~24% acceptance rate)
5. Jack Wadden, **Kevin Angstadt**, and Kevin Skadron. Characterizing and Mitigating Output Reporting Bottlenecks in Spatial Automata Processing Architectures. In *Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture*, Vienna, Austria, 2018. IEEE. (21% acceptance rate)
6. **Kevin Angstadt**, Westley Weimer, and Kevin Skadron. RAPID Programming of Pattern-Recognition Processors. In *Proceedings of the 21st International Conference on Architectural Support for programming Languages and Operating Systems*, Atlanta, Georgia, 2016. ACM. (22% acceptance rate)

Additional Publications

7. **Kevin Angstadt** and Ed Harcourt. A Virtual Machine Model for Accelerating Relational Database Joins using a General Purpose GPU. In *Proceedings of the High Performance Computing Symposium*, Alexandria, VA, 2015. Society for Computer Simulation International
8. Sihang Liu, **Kevin Angstadt**, Mike Ferdman, Samira Khan. ARMOR: Towards Restricted Approximation with a Worst-Case Guarantee. In: *Proceedings of the 2018 Workshop on Approximate Computing Across the Stack*, Williamsburg, VA, 2018.
9. Kate Highnam, **Kevin Angstadt**, Kevin Leach, Westley Weimer, Aaron Paulos, and Patrick Hurley. An Uncrewed Aerial Vehicle Attack Scenario and Trustworthy Repair Architecture. In *Proceedings of the 46th International Conference on Dependable Systems and Networks*, Industrial Track, Toulouse, France, 2016. IEEE.

Invited Papers and Tech Reports

10. Ke Wang, **Kevin Angstadt**, Chunkun Bo, Nathan Brunelle, Elaheh Sadredini, Tommy Tracy, II, Jack Wadden, Mircea Stan, and Kevin Skadron. An overview of Micron's Automata Processor. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, Pittsburgh, PA, 2016. ACM.
11. **Kevin Angstadt**, Jack Wadden, Westley Weimer, and Kevin Skadron. MNRL and MNCaRT: An Open-Source, Multi-Architecture State Machine Research and Execution Ecosystem. Technical Report CS-2017-01, Department of Computer Science, University of Virginia, May 2017.

Proposal Summary

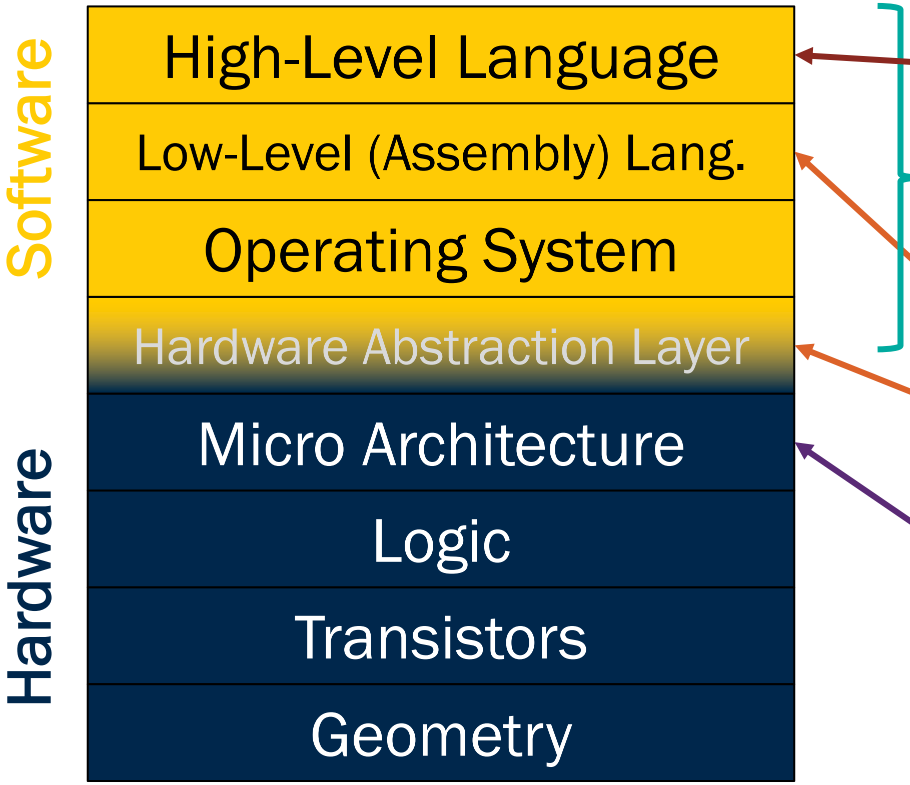
- Hardware accelerators more commonplace—need for programming models and maintenance tools
- Four components to improve programming support for hardware accelerators using automata abstractions
  - High-level programming language (RAPID)
  - High-speed, interactive debugging for RAPID on AP and FPGA
  - In-cache accelerator for pushdown automata (ASPEN)
  - Adapt legacy kernels for execution on hardware accelerators
- Evaluation w.r.t. Performance & scalability, ease of use, expressive power, and legacy support

Discussion Cache

- Space overheads for FPGA debugging probes
  - Dynamic probing or decoupling clock signals or some other approach?
- Memory-based intrusion detection evaluations and applications
  - What classes of abnormal behavior can we detect?
  - What do we do with this information?
- Automata synthesis approach and evaluation
  - Thoughts on representing equivalence queries?
  - What will computer architects want to know?
- Potential interest overlap and collaboration?
- Integrating with pedagogy…undergraduate mentorship?

Bonus Slides

Programming Support



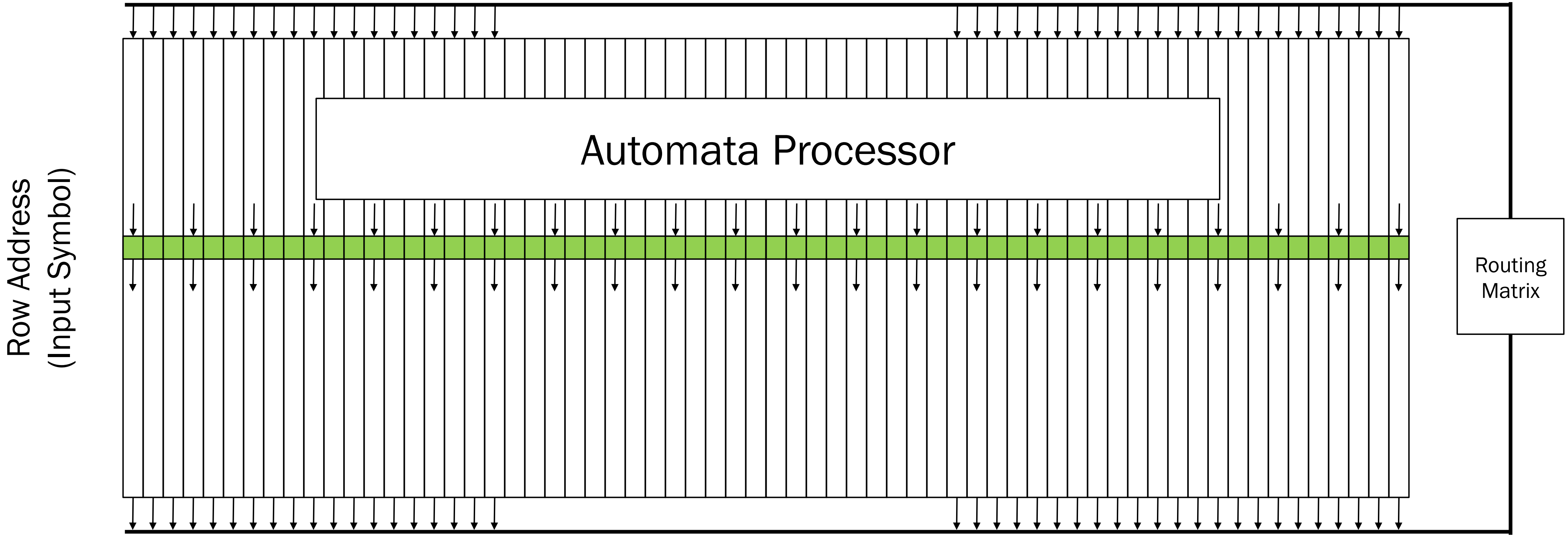
New programming languages better-suited for specific application domains/hardware

Software systems to adapt/transform/improve existing (legacy) code

Low-level representations to bridge the gap between software and hardware

Architectural modifications to support applications and features

The AP at a High Level

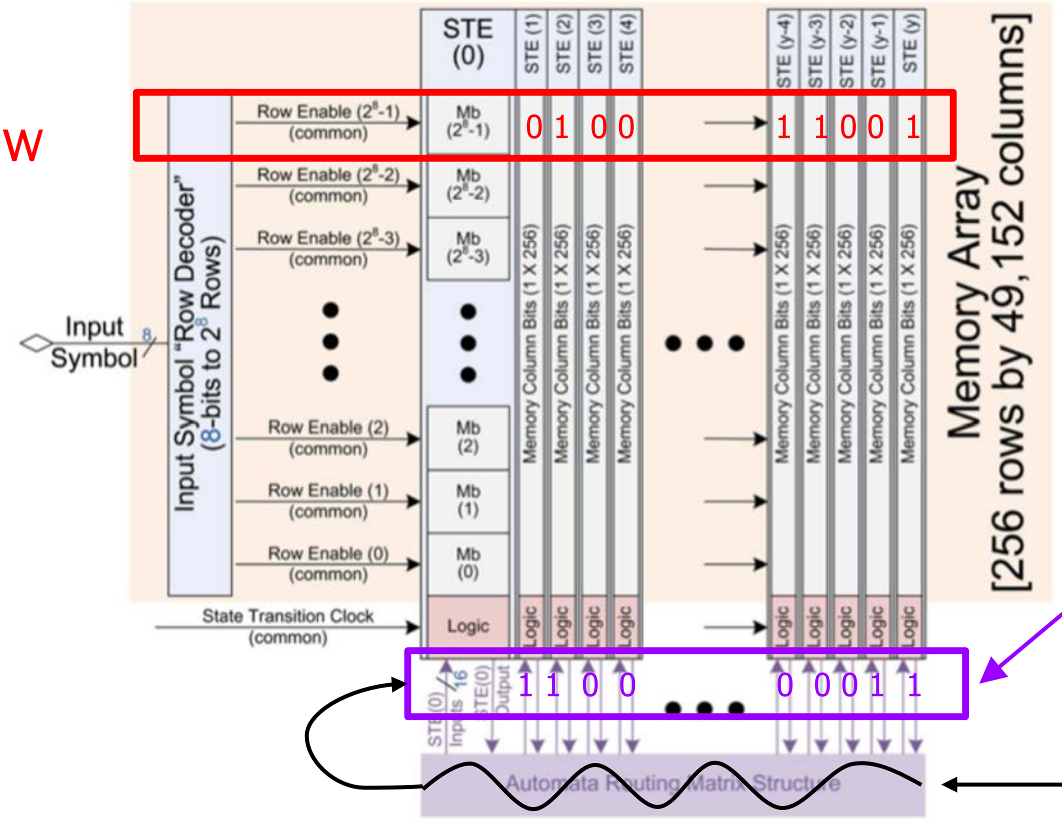


Row Access results in **49,152** match & route operations

Executing NFA in DRAM

- Columns in DRAM store STE labels (Each STE is a single column)
- Reconfigurable routing matrix connects the STEs

Input:
Drives a Row



Columns with “1”:
STEs that accept
input symbol

&

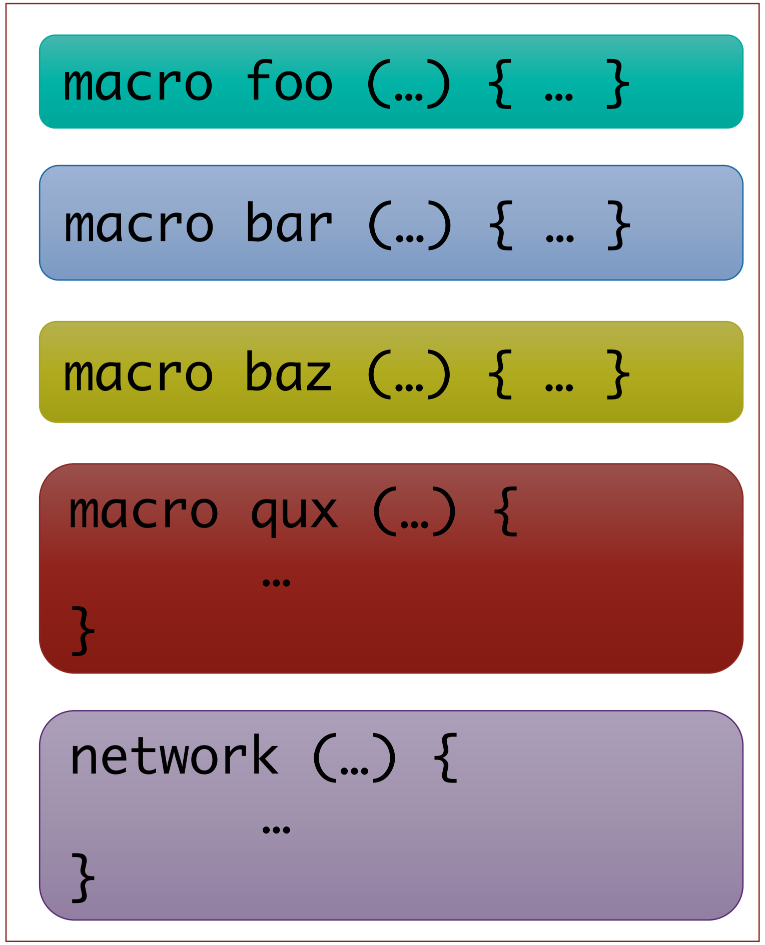
Active States

=

Active States for Next
Clock Cycle

Program Structure

- Macro
  - Basic unit of computation
  - Sequential control flow
  - Boolean expressions as statements for terminating threads of computation
- Network
  - High-level pattern matching
  - Parallel control flow
  - Parameters to set run-time values



```
macro foo (…) { … }

macro bar (…) { … }

macro baz (…) { … }

macro qux (…) {
        …
}

network (…) {
        …
}
```

Either/Orelse Statements

```
either {
    hamming_distance(s,d); //hamming distance
    'y' == input();        //next input is 'y'
    report;                //report candidate
} orelse {
    while('y' != input()); //consume until 'y'
}
```

- Perform parallel exploration of input data
- Static number of parallel operations

Some Statements

```
network (String[] comparisons) {
    some(String s : comparisons)
        hamming_distance(s,5);
}
```

- Parallel exploration may depend on candidate patterns
- Iterates over items, dynamically spawn computation

Whenever Statements

```
whenever( ALL_INPUT == input() ) {
    foreach(char c : "rapid")
        c == input();
    report;
}
```

- Body triggered whenever guard becomes true
- `ALL_INPUT`: any symbol in the input stream

Parallel Control Structures

Sequential Structure	Parallel Structure
if…else	either…orelse
foreach	some
while	whenever

Example RAPID Program

Association Rule Mining

Identify items from a database that frequently occur together

Example RAPID Program

If all symbols in item set match, increment counter

Spawn parallel computation for each item set

Sliding window search calls *frequent* on every input

Trigger *report* if threshold reached

```
macro frequent (String set, Counter cnt) {
        foreach(char c : set) {
            while(input() != c);
        }
        cnt.count();
}

network (String[] set) {
    some(String s : set) {
        Counter cnt;
        whenever(START_OF_INPUT == input())
            frequent(s,cnt);
        if (cnt > 128)
            report;
    }
}
```

Houston, we have a problem!

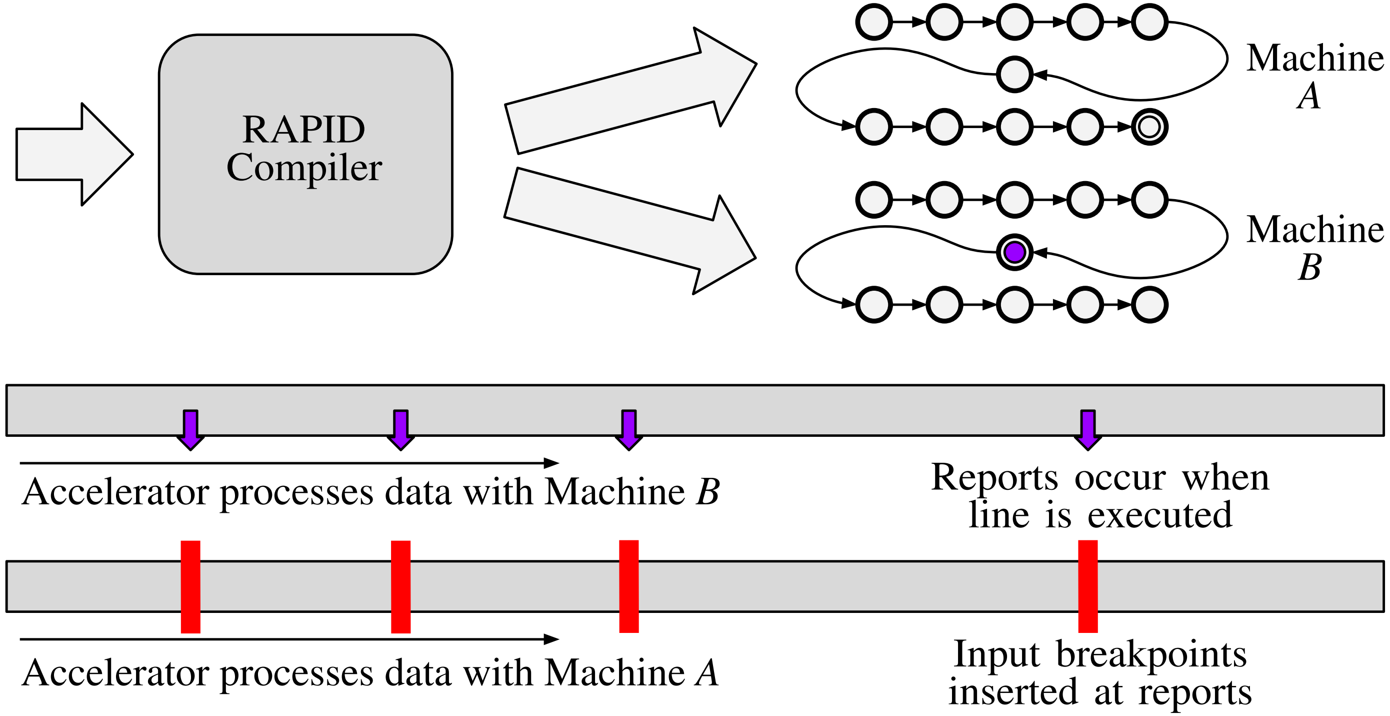
- Accelerator applications often target large datasets
- CPUs typically too slow to debug full applications
- Abnormal behavior may not manifest itself in testing inputs
  - Low quality/coverage test suites
  - Ex. ANMLZoo (automata processing benchmark suite) contains two inputs per application and no gold standard output!
- May be difficult to extract subset of input for debugging
- Low-level debugging support exists—tedious to use and abstraction mismatch

Traditional Breakpoints

RAPID Program

```
macro helloWorld() {
  whenever(  ALL_INPUT == input() ) {
    foreach(char c : "Hello") {
      c == input();
    }
    input() == ' ';
    foreach(char c : "world") {
      c == input();
    }
    report;
  }
}

network() {
  helloWorld();
}
```



RAPID Compiler

Machine *A*

Machine *B*

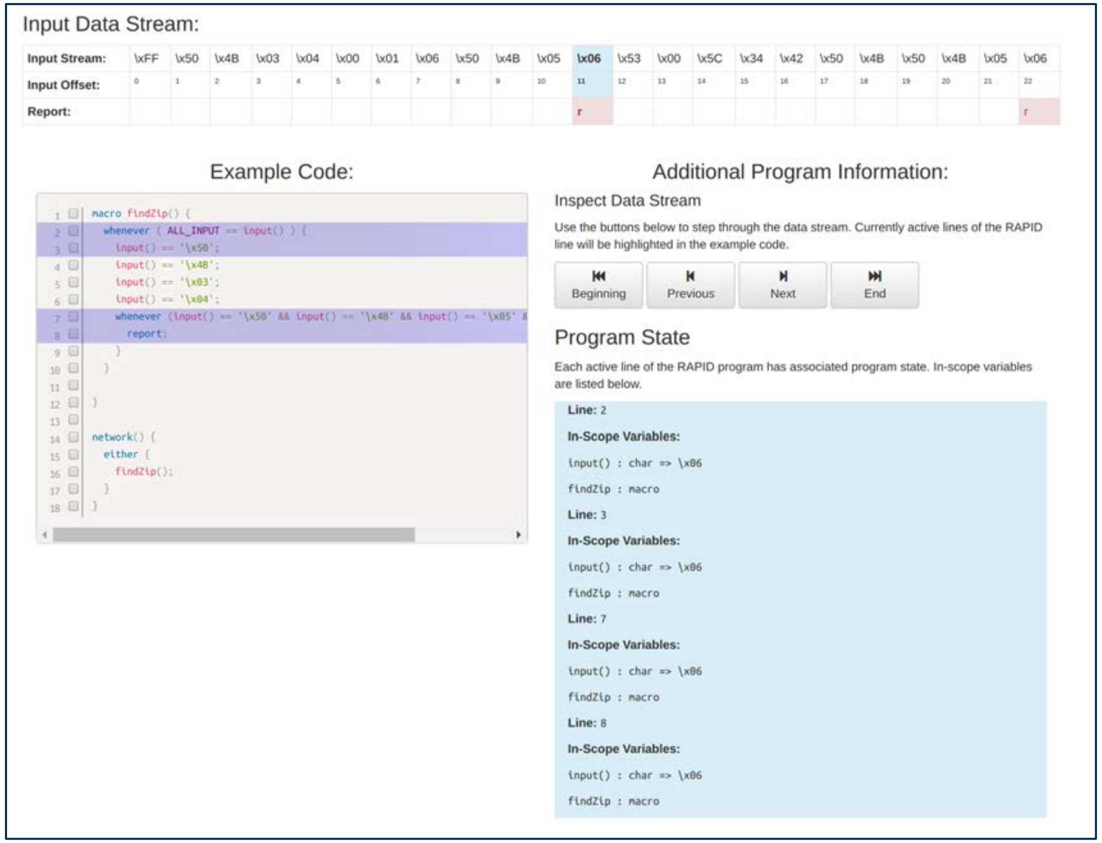
Accelerator processes data with Machine *B*

Reports occur when line is executed

Accelerator processes data with Machine *A*

Input breakpoints inserted at reports

Human Study Results



Input Data Stream:

Input Stream:	\xFF	\x50	\x4B	\x03	\x04	\x00	\x01	\x06	\x50	\x4B	\x05	**\x06**	\x53	\x00	\x5C	\x34	\x42	\x50	\x4B	\x50	\x4B	\x05	\x06
Input Offset:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
Report:												r											r

Example Code:

```
1   macro findZip() {
2     whenever ( ALL_INPUT == input() ) {
3       input() == '\x50';
4       input() == '\x4B';
5       input() == '\x03';
6       input() == '\x04';
7       whenever (input() == '\x50' && input() == '\x4B' && input() == '\x05' &
8         report;
9       }
10    }
11
12  }
13
14  network() {
15    either {
16      findZip();
17    }
18  }
```

Additional Program Information:

Inspect Data Stream

Use the buttons below to step through the data stream. Currently active lines of the RAPID line will be highlighted in the example code.

Program State

Each active line of the RAPID program has associated program state. In-scope variables are listed below.

Line: 2

In-Scope Variables:

`input() : char => \x06`

`findZip : macro`

Line: 3

In-Scope Variables:

`input() : char => \x06`

`findZip : macro`

Line: 7

In-Scope Variables:

`input() : char => \x06`

`findZip : macro`

Line: 8

In-Scope Variables:

`input() : char => \x06`

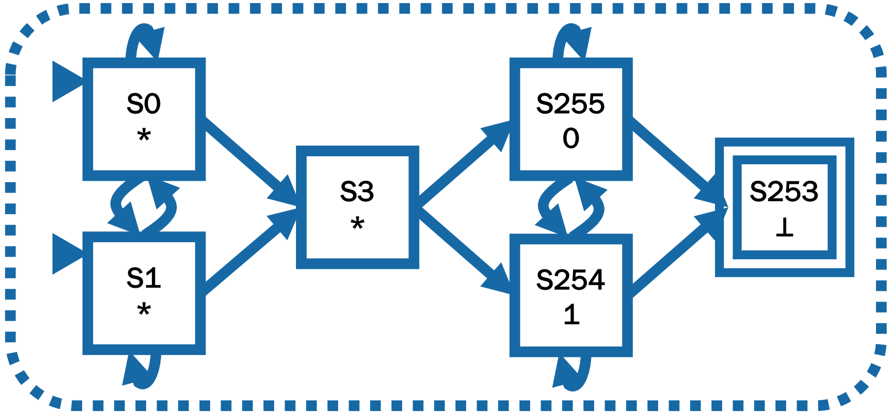
`findZip : macro`

- ~60 participants (students and professional developers)
- Participants shown 10 RAPID programs with seeded defects
  - 5 have interactive debugger
  - 5 have no debugging information
- Asked to identify location of bug in source code and describe
- Recorded time needed to perform each task

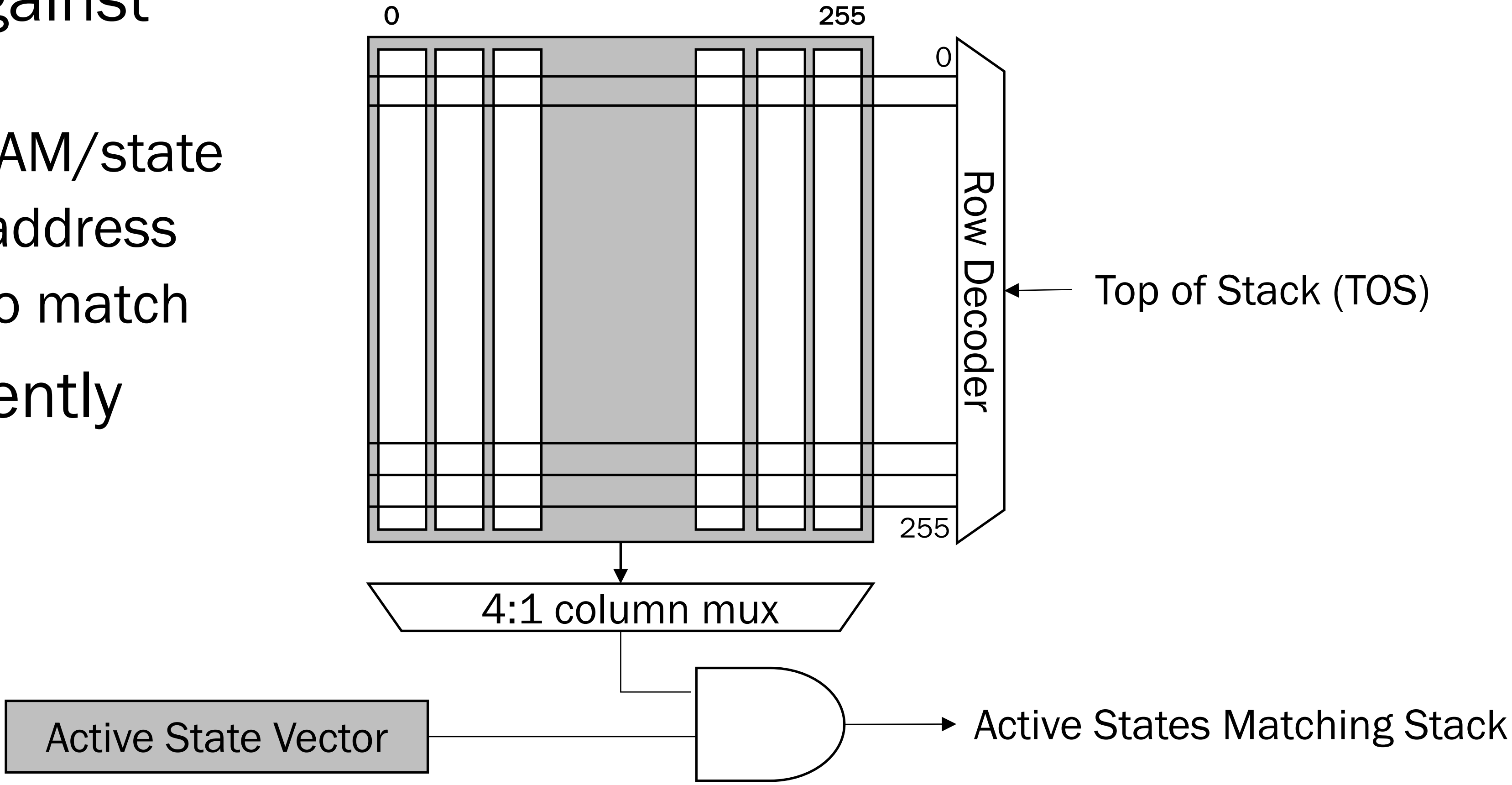
Implementing ASPEN in LLC

- ASPEN repurposes LLC slices for pushdown automata computation
- Location in LLC supports tighter coupling with CPU operations than dedicated accelerator
  - PDA often part of a larger workflow
  - ASPEN similar to auxiliary functional unit in CPU (similar to FPU or vector unit)
- SRAM arrays in LLC already support necessary operations for DPDA execution

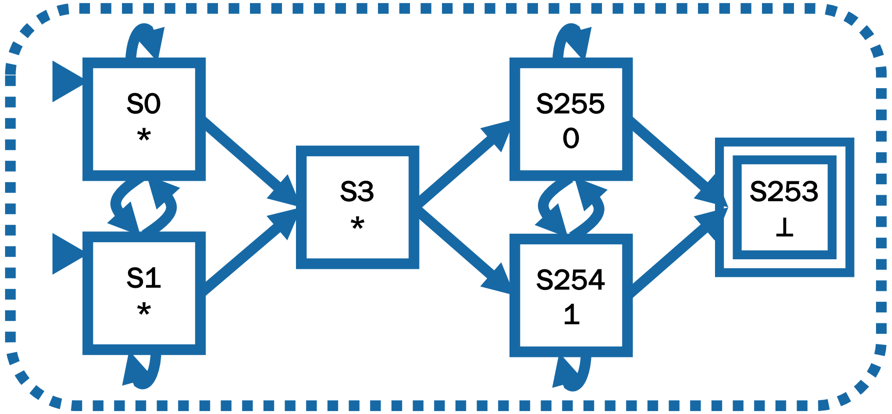
Stack Match in SRAM



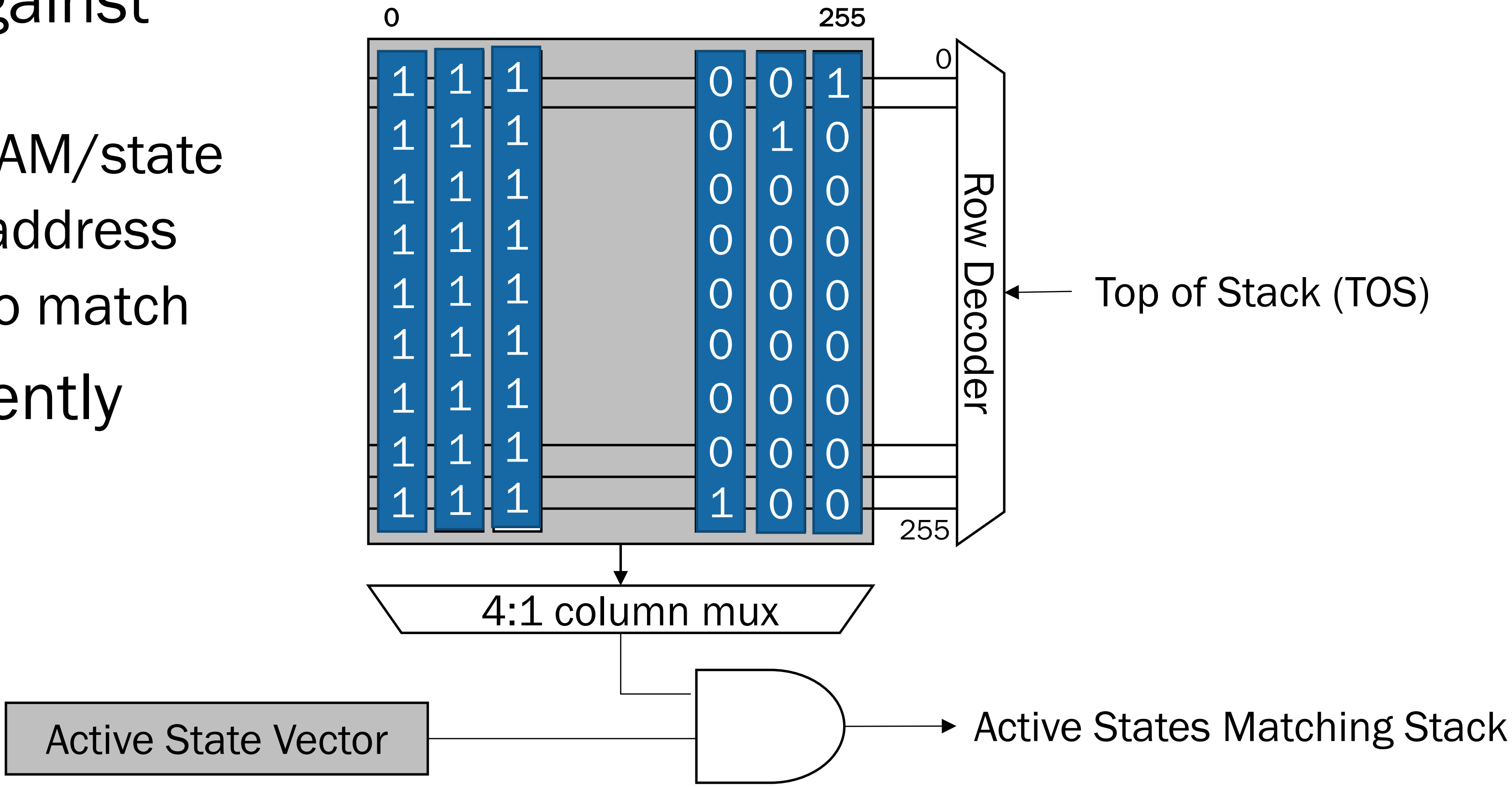
- Check **all states** against top of stack
  - One column of SRAM/state
  - Input TOS as row address
  - “1”: match; ”0”: no match
- **Intersect** with currently active states



Stack Match in SRAM



- Check **all states** against top of stack
    - One column of SRAM/state
    - Input TOS as row address
    - “1”: match; ”0”: no match
- **Intersect** with currently active states



1	1	1	0	0	1
1	1	1	0	1	0
1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	0	0	0
1	1	1	1	0	0

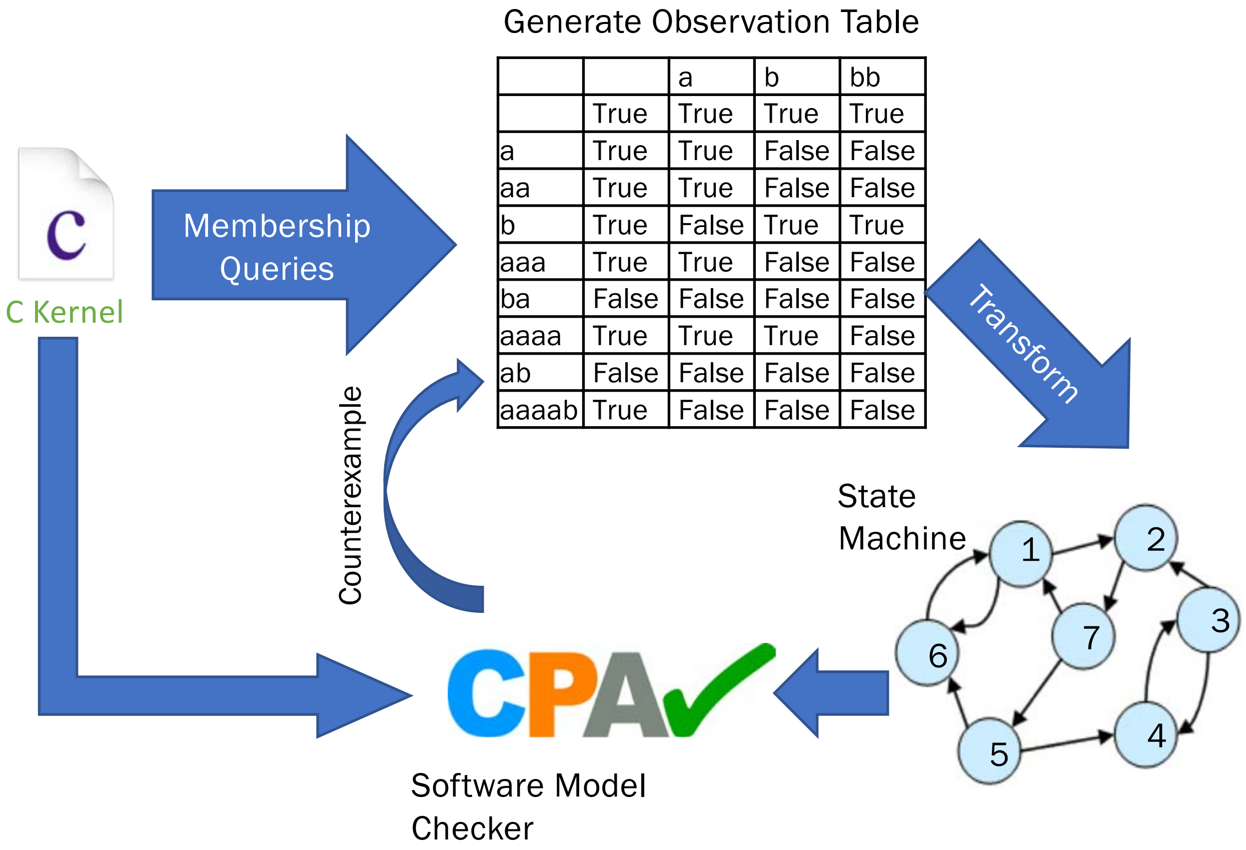
Preliminary Results: XML Parsing

Component	Max Frequency	Operating Frequency
ASPEN	880 MHz	850 MHz
Cache Automaton	4 GHz	3.4 GHz

- **Baseline Evaluation**
  - **CPU:** 2.6 GHz dual-socket Intel Xeon E5-2697-v3 (28 cores total)
- **Performance and Power:** PAPI, Intel RAPL
- **ASPEN Simulation:**
  - METIS graph partitioning framework
  - VASim modified for cycle-accurate DPDA simulation

Angluin-Style Learning

- Attempts to learn a finite automaton from a held-out model
- Requires set membership queries (Is string in language? Returns yes/no answer)
    - *Run legacy code for answer*
- Requires equivalence queries (Is the automaton equivalent to model? Returns yes/counterexample)
    - *Software model checking to find differences*



		a	b	bb
	True	True	True	True
a	True	True	False	False
aa	True	True	False	False
b	True	False	True	True
aaa	True	True	False	False
ba	False	False	False	False
aaaa	True	True	True	False
ab	False	False	False	False
aaaab	True	False	False	False