PL in the Broader Research Community

EECS 590: Advanced Programming Languages
27. November 2017

Kevin Angstadt
angstadt@umich.edu

Who am I?

- Fourth-year PhD student (I did my first three years at UVA)
- Advised by Wes Weimer
- My Research: Programming Support for Emerging Hardware Technologies
  - I focus primarily on **automata processing**
  - Developed two languages for automata applications (RAPID and MNRL)
  - Automated techniques for adapting legacy code
  - Compiler techniques for more efficient execution
  - (You'll also find me tinkering with our group's autonomous vehicles)
- I also spend a decent amount of time teaching and advising undergraduates

Today's Agenda

- Debuggers for Traditional and Non-Traditional Architectures
  - High-Level Overview
  - Setting Breakpoints
  - Review of NFAs
  - Extending Breakpoints to Non-Traditional Architectures
  - Lifting Semantics to the Source Level
  - Evaluating the Effectiveness of Debuggers
- Discussion: Presenting PL in the Broader Research Community
  - Remember those three papers for today?

Debugging Support for Traditional and Non-Traditional Architectures

EECS 590: Advanced Programming Languages
27. November 2017

Kevin Angstadt
angstadt@umich.edu

COMPUTER SCIENCE
& ENGINEERING
UNIVERSITY OF MICHIGAN

“If debugging is the process of removing bugs, then programming must be the process of putting them in.”

Edsger Dijkstra

This lecture may feel very intuitive. That's quite alright! There are no real tricks with the basics of debugging; it's just something you may not have seen before.

Localizing/Understanding Bugs

- **Goal:** Print out information that helps us to identify problem points
- Two different approaches:
  - Compile the print-outs directly into the program source
  - Create a tool that can **attach** to an executing process, stop the process, and inspect program state
- What are pros/cons of each approach?
- Non-traditional architectures do not always support “printing out values”

"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 extract current program state from the processor

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

Machine Language Debugger (Traditional)

- Operates at the machine/assembly code level
  - Instructions are show vis **disassembly**
- Allows a user to inspect the values of registers and memory
- Supported features:
  - Attach to a process (requires some level of OS support; e.g., `ptrace`)
  - Set breakpoints (can be conditional) and watchpoints
  - Single-step through instructions

Breakpoints via Signals

- A signal is an asynchronous notification sent to a process about an event
  - User presses ctrl-c, Runtime exceptions, OS notifications
- Programs can implement signal handlers
  - A signal handler is a function that gets executed when a signal occurs during the execution of a program
- Add a breakpoint by changing the assembly instructions
  - We can't really insert instructions…why?

Setting a Breakpoint

- Identify location in program for breakpoint
- Store old instruction
- Replace with an exception (or something to generate a signal)
- Signal handler replaces instruction
- Prompt for debugging

Additional Features

- How do we inspect memory/registers?
- Conditional breakpoints?
- Watchpoints? (program execution stops when a memory location is read/written)
- Single-stepping?
- Hardware support for breakpoints/watchpoints?

What assumptions did we make?

- Ability to generate/handle signals
- Ability to dynamically poke memory
- Processor is executing assembly/machine instructions

These assumptions do not always hold for non-traditional architectures!

What if we can’t dynamically interact?

- Replay (time travel) debugging is our friend!
- How it works:
  - Run the program and record memory/registers every X cycles
  - ”Replay” the program using the snapshots and simulation (or possibly direct execution on the hardware)
- Replay can be particularly helpful for non-traditional architectures (e.g., wireless sensor networks, accelerators, etc.)
  - Also allows for stepping “backward” in a debugger

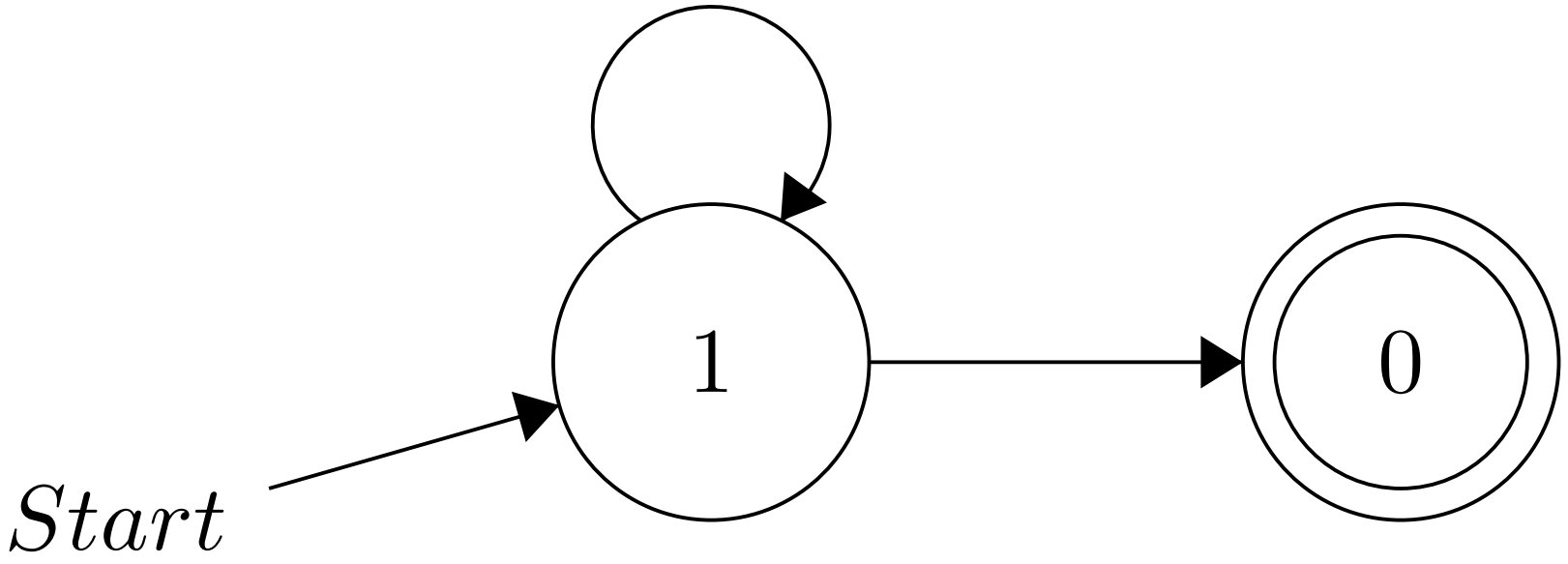
What if there are no assembly instructions?

- Not all architectures execution assembly instructions (e.g., FPGAs)
- We will focus on automata processing architectures
  - Rather than using assembly instructions, programs are executed as state machines (NFAs or DFAs), which process a sequence of data
  - Recent research has found that a wide range of applications can be translated to the automata processing paradigm (e.g., NIDS, part-of-speech tagging, random forests, association rule mining, high-speed physics particle tracking, etc.)
- First, the requested review of NFAs…

What is a **Finite Automaton?**

- A finite automaton consists of
  - An input alphabet, Σ
  - A set of states, Q
  - A starting state, q_0
  - A set of accepting states, $F \subseteq Q$
  - A transition function, $transition(\ input, q_i\ ) = q_j$

Finite Automaton Example



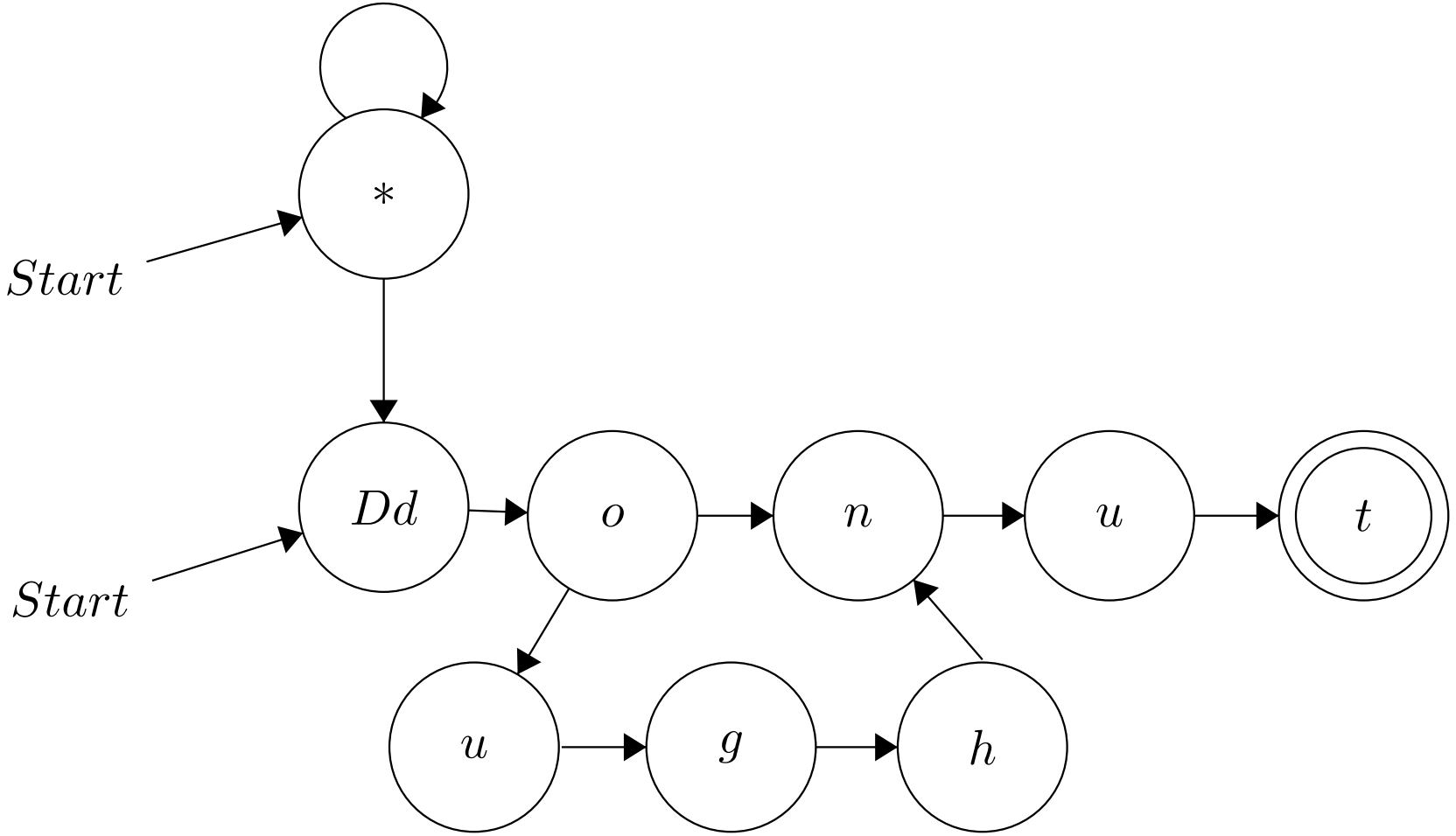
Check: 1110

Check: 101

Optimization: Non-determinism

- Can be in multiple states at one time
  - Can start in more than one state
  - Transition function returns a set of states
  - (ε-transitions: we'll ignore these, but you'll see them in literature)
- **Why?**
  - Exponentially more compact than a (D)FA
  - Often easier to generate
  - Allow for exploitation of parallelism in hardware

What does this find?



Micron's Automata Processor

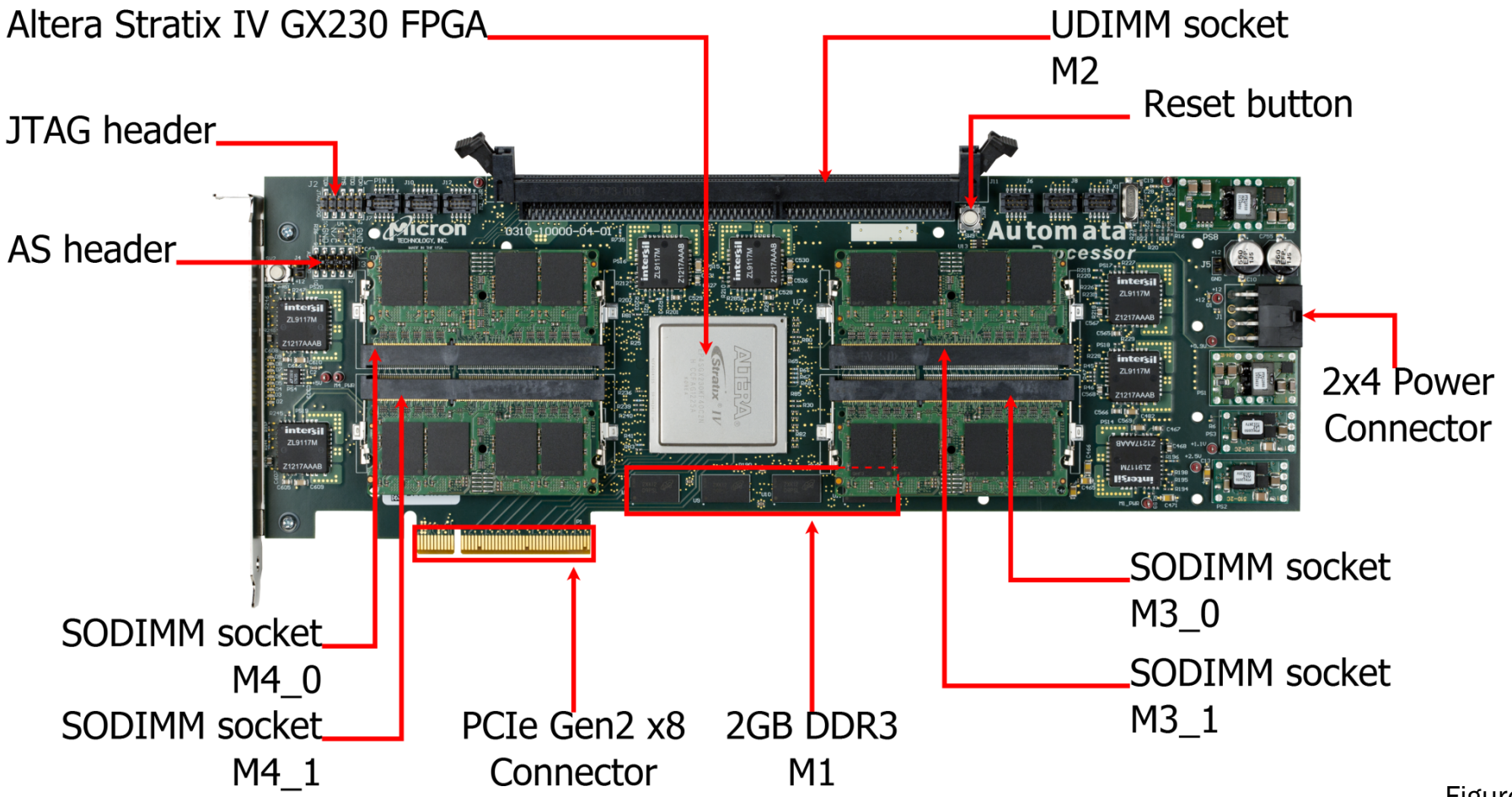
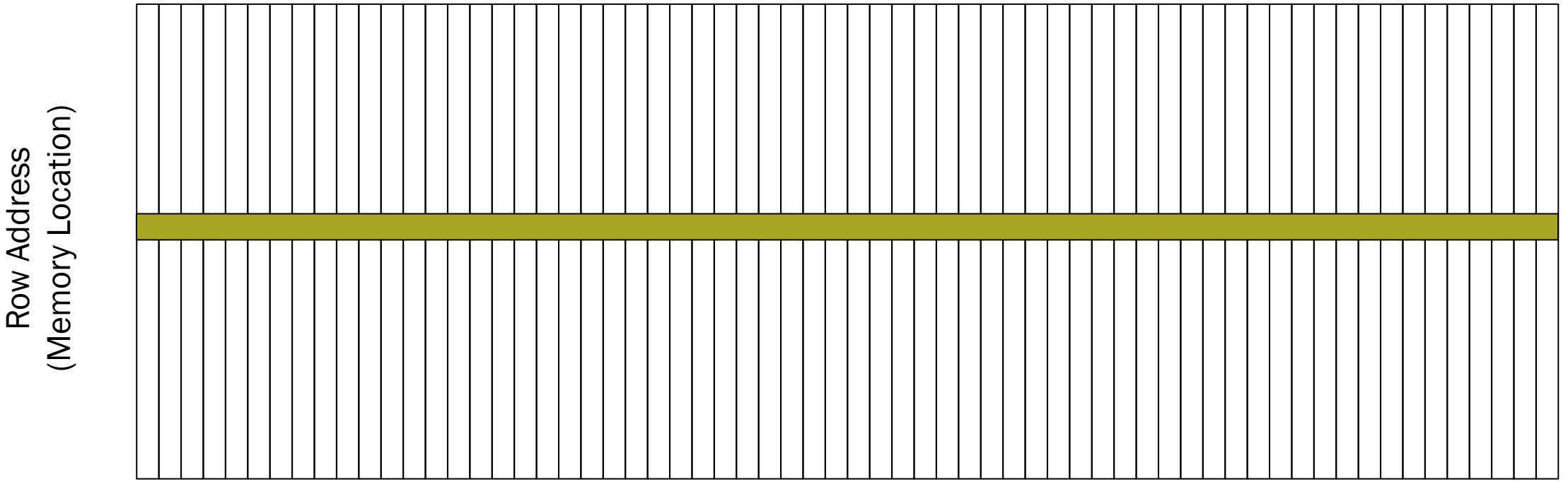


Figure courtesy of Micron

Exploiting DRAM



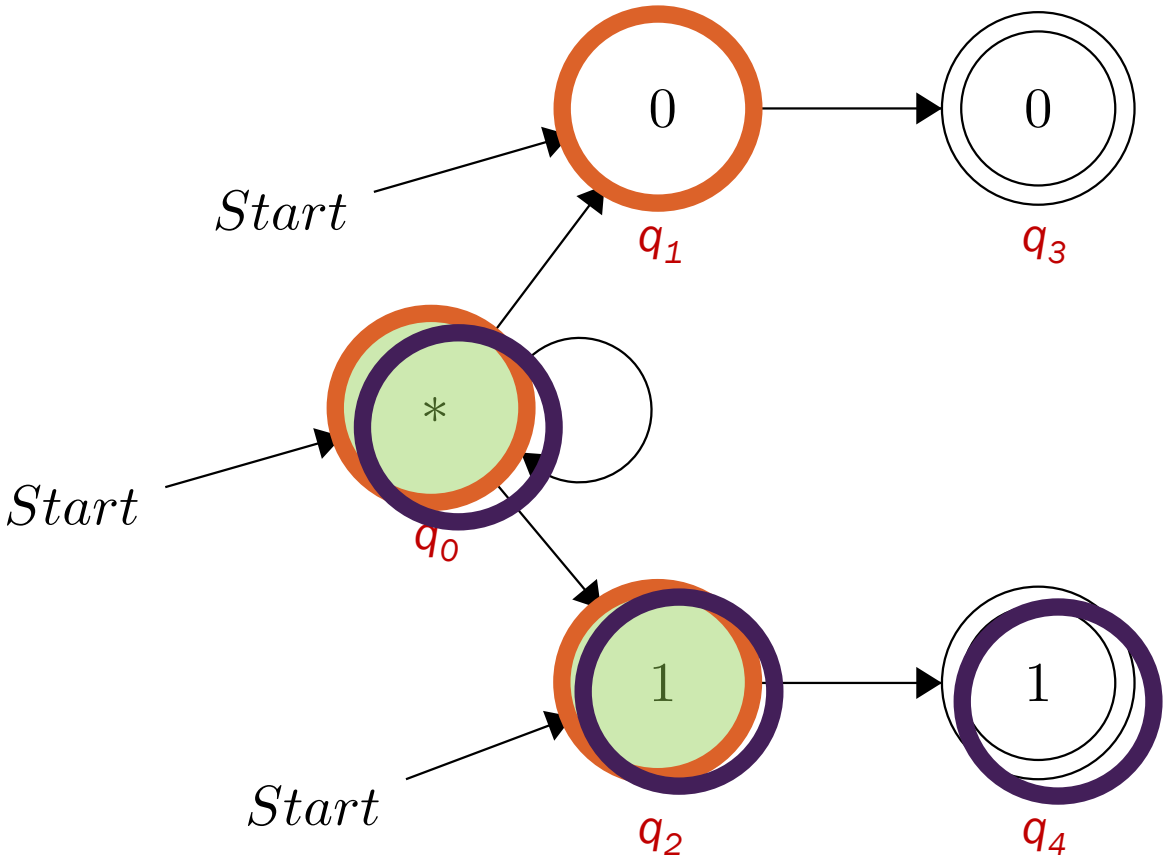
Row Access results in **one** word being retrieved from memory

Using Bit-Parallelism with NFAs

- Bit-parallel closure: given a set of states, what is the set of reachable states?
- Symbol closure: given an input symbol, what states accept this symbol?
- Transition function: intersection (bitwise AND) of these two closures

Bit-Parallel Execution

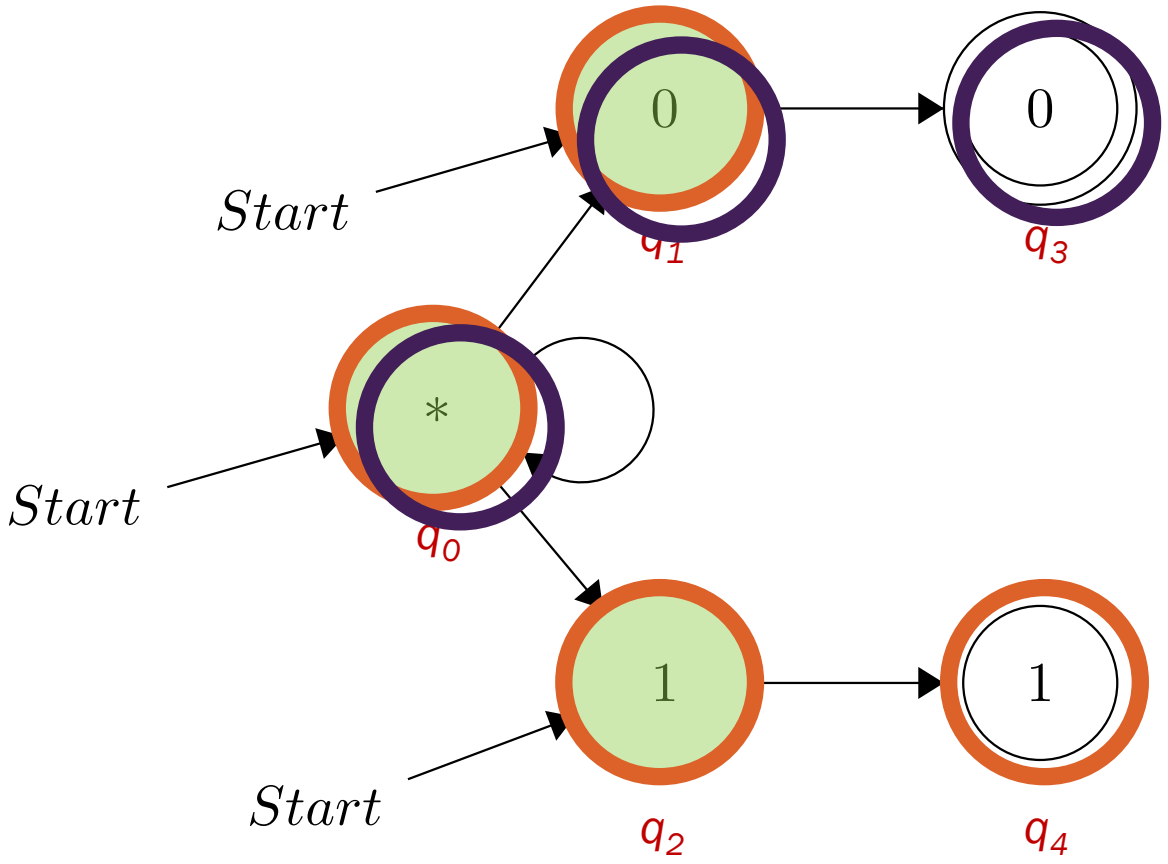
Input Stream: 1011



- Bit-parallel closure({*start*}):
  - $\{q_0, q_1, q_2\}$
- Symbol closure(1):
  - $\{q_0, q_2, q_4\}$
- Bit & Symbol:
  - $\{q_0, q_2\}$

Bit-Parallel Execution

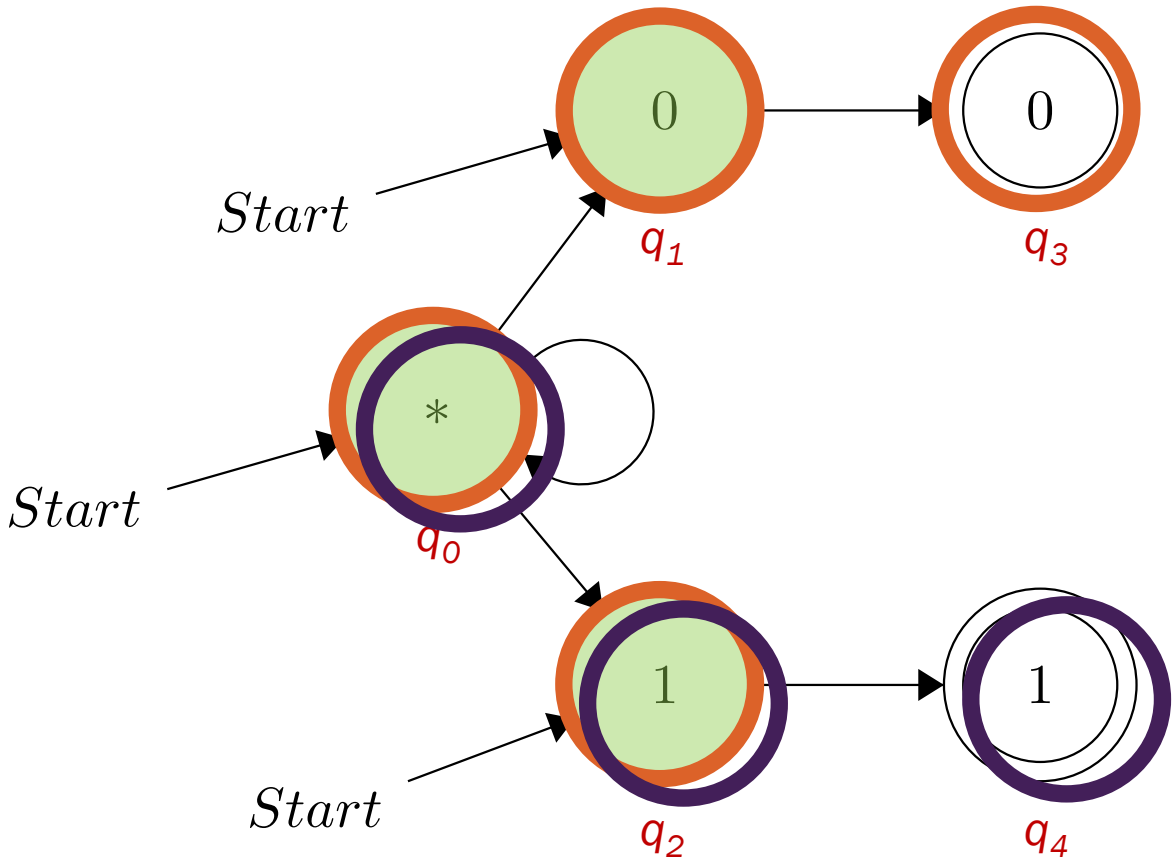
Input Stream: 1011



- Bit-parallel closure($\{q_0, q_2\}$):
  - $\{q_0, q_1, q_2, q_4\}$
- Symbol closure(0):
  - $\{q_0, q_1, q_3\}$
- Bit & Symbol:
  - $\{q_0, q_1\}$

Bit-Parallel Execution

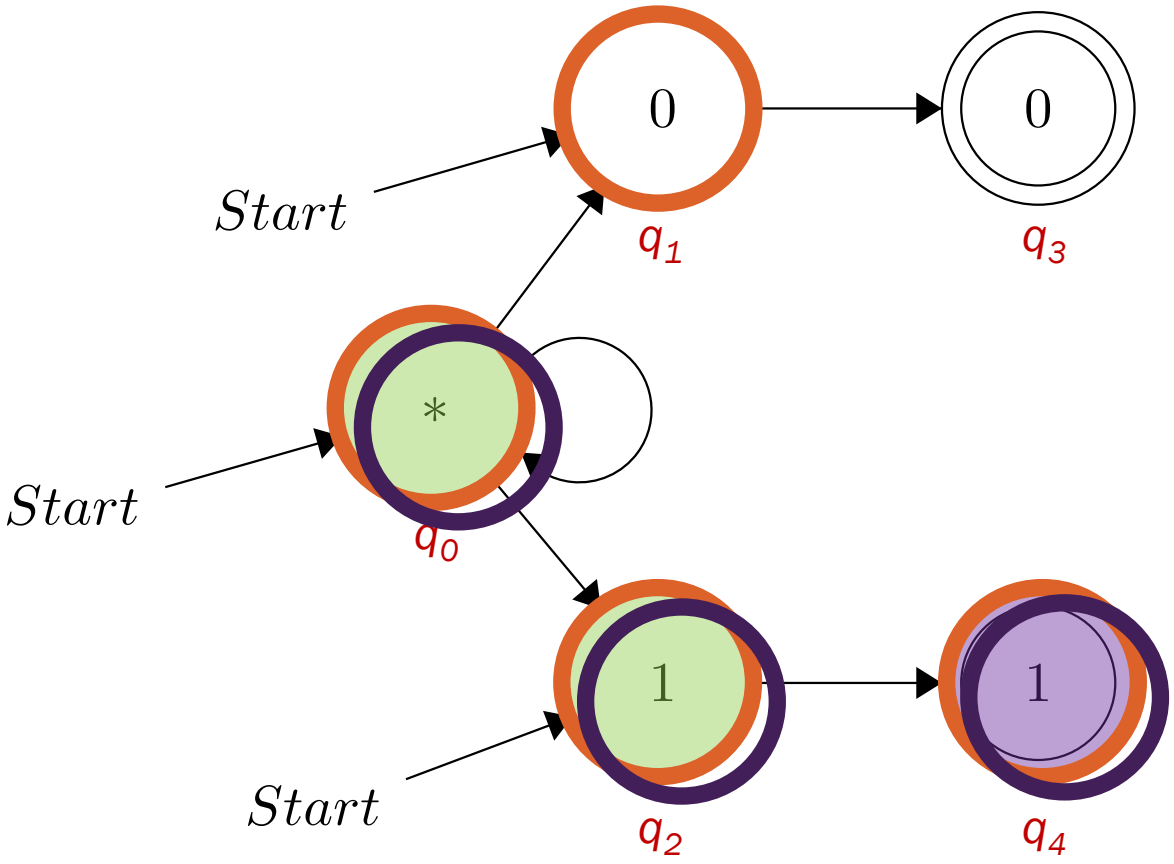
Input Stream: 1011



- Bit-parallel closure($\{q_0, q_2\}$):
  - $\{q_0, q_1, q_2, q_3\}$
- Symbol closure(1):
  - $\{q_0, q_2, q_4\}$
- Bit & Symbol:
  - $\{q_0, q_2, q_4\}$

Bit-Parallel Execution

Input Stream: 1011



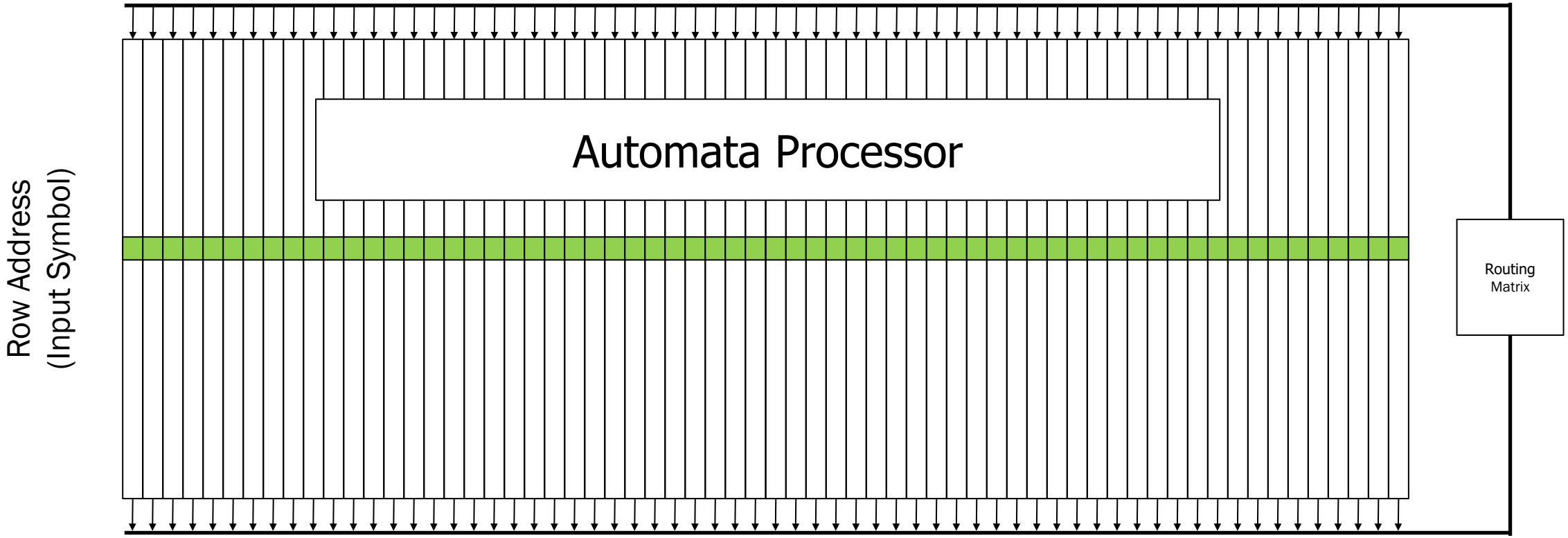
- Bit-parallel closure($\{q_0, q_2\}$):
  - $\{q_0, q_1, q_2, q_4\}$
- Symbol closure(1):
  - $\{q_0, q_2, q_4\}$
- Bit & Symbol:
  - $\{q_0, q_4\}$

REPORT!

Architectural Design

- Memory Array (Symbol Closure)
  - 256 Rows (i.e. 256 Input Characters)
  - 49,152 Columns (i.e. 49,152 States)
- Routing Matrix (Bit-Parallel Closure)
  - Saturating Counters
  - Boolean Gates

The AP at a High Level

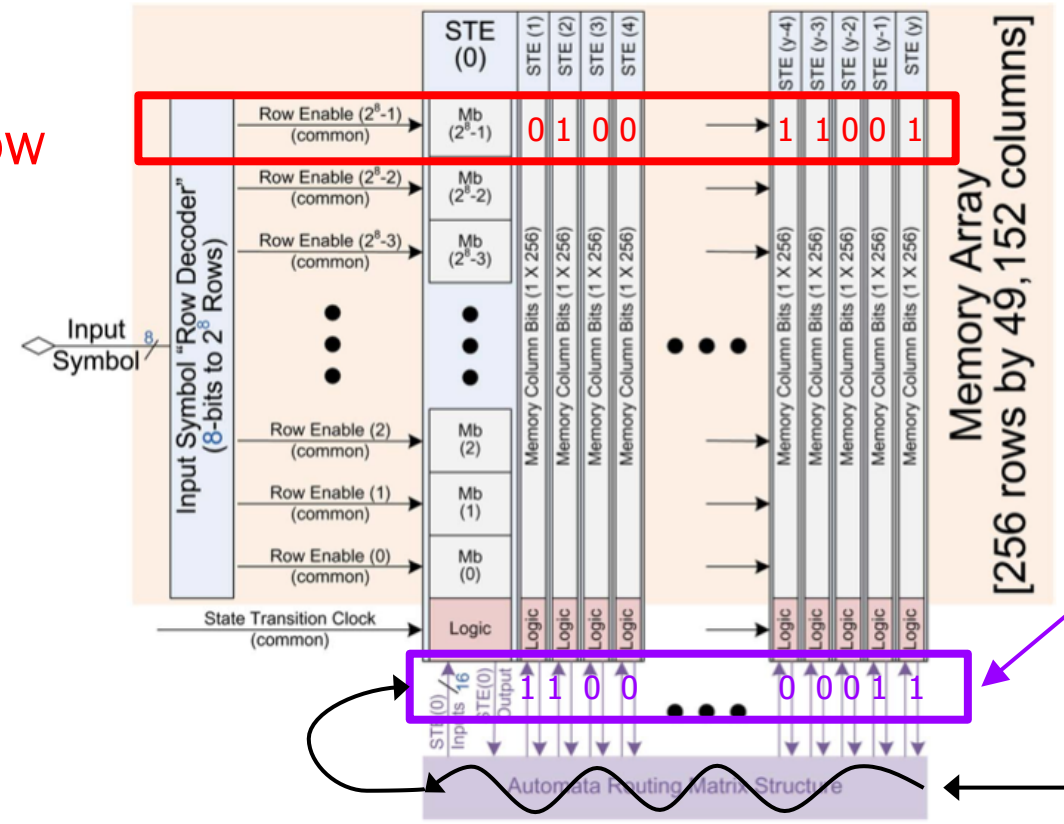


Row Access results in **49,152** match & route operations

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

Meanwhile, Back at the Ranch: Multi-Step Process

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

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

What are we trying to do?

- We want to set breakpoints in NFAs and inspect their state
- In this context, what is a breakpoint?
  - CPU: every clock cycle (conceptually) an instruction is executed. Breakpoints are set to trigger when an instruction is executed
  - AP: every clock cycle (conceptually) an input data symbol is consumed by the NFAs. Breakpoints are set to trigger after a given number of symbols are consumed.
- What is program state on an automata processor?

Lifting the Semantics

- We now have a mechanism for stopping program execution and inspecting the state of the underlying processor
- Many (most?) applications are written using a high-level programming language that is compiled to target processor's instruction set/computational representation
  - What is a high-level language for automata processing?
- What is the relationship between processor state and the source-level program?

Debugging Tables (Traditional)

- Compiler emits tables mapping source-level expressions to assembly instructions
  - For every line, what assembly range does it map to?
  - For every line, what variables are in scope and where do they live?
- Breakpoint insertion: look up location in the debugging table and set a corresponding breakpoint in the assembly

- What happens if we optimize code?

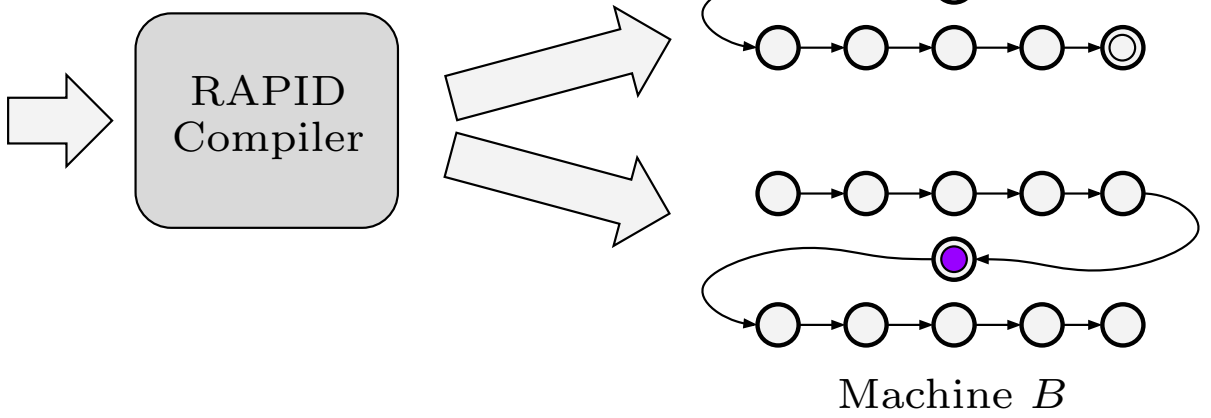
Debugging Tables for Automata Processing

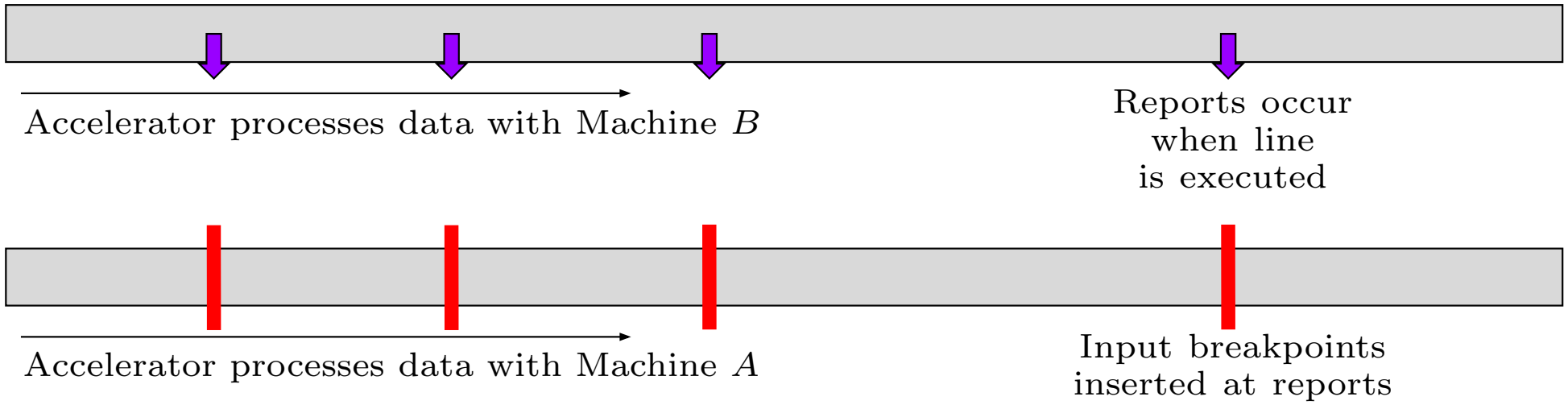
- Very similar to the CPU approach…
  - For every line, which NFA states does it map to?
  - For every line, what variables are in scope and what are their values (or which hardware resources hold their values)?
- Breakpoints on the input data remain the same for high-level languages
- We can set breakpoints on lines of code by leveraging reporting signals to identify offsets in the input stream where the line of code is executing

RAPID Program

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

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

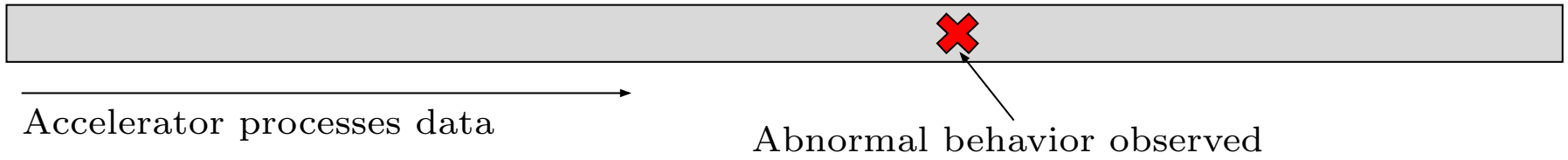




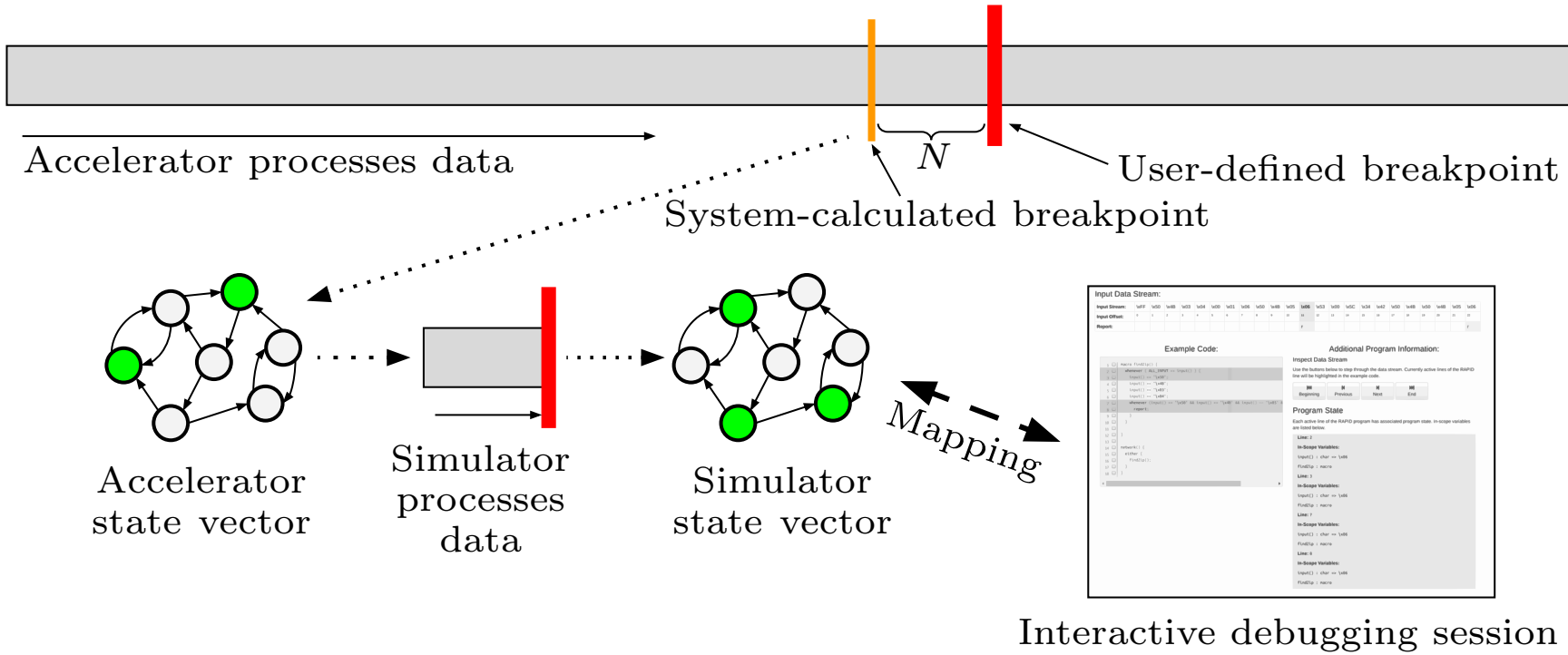
Putting it all together

- Developer writes program in high-level language (e.g., RAPID) for automata processing
- The compiler generates NFAs for execution and debugging tables with mappings between source lines and NFA states
- During program execution, the developer observes abnormal behavior after executing X bytes
- The debugging system processes X bytes and extracts the state vector
- Control is transferred to an interactive debugger where the recorded state and debugging tables are used to inspect the state of the program

Standard Program Execution

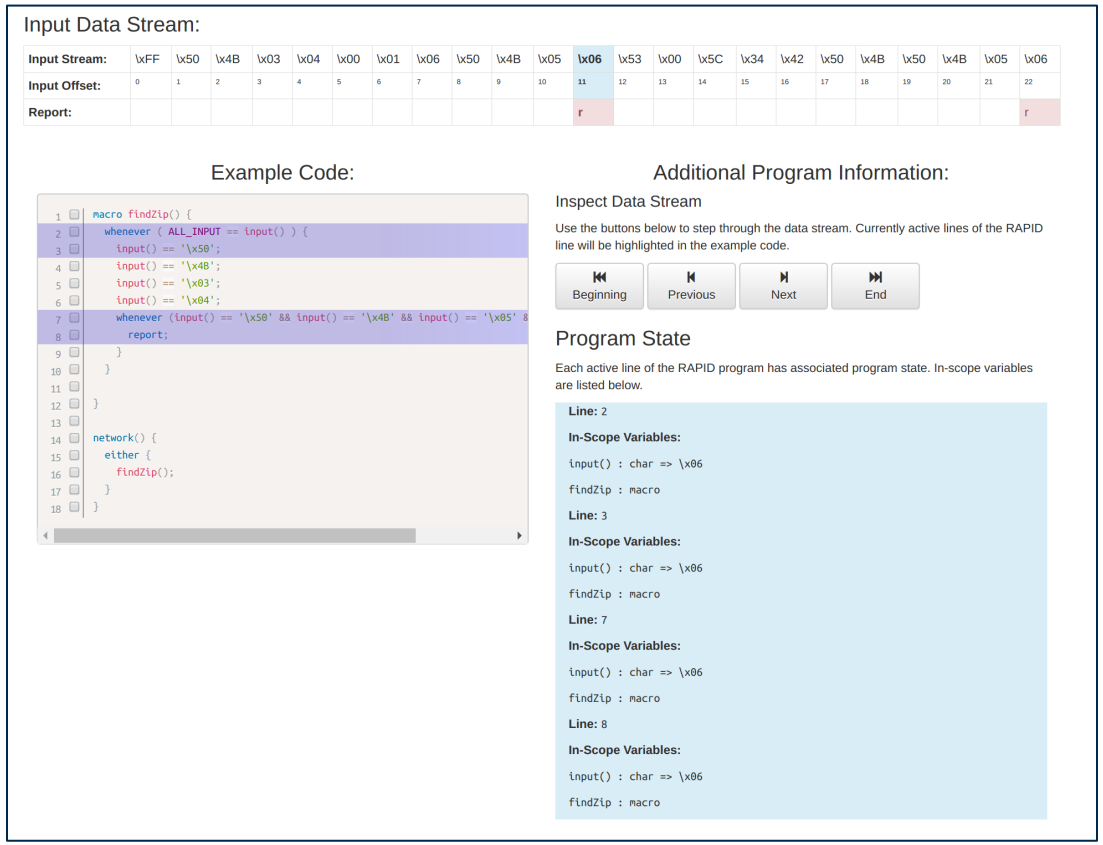


Debugging Execution



Interactive debugging session

Do these tools actually help?



- ~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

Human Study Results

- Our tool did not (statistically significantly) decrease the time needed to localize faults in RAPID programs
- Our debugging tool improves a user's fault localization accuracy for RAPID programs in a statistically significant manner ($p = 0.013$)
- Debugging information for RAPID programs helps novices and experts alike (there is no interaction between developer experience and the ability to interpret debugging information)

What about other non-traditional archs?

- **Macrodebugging**: replay debugging of event traces in wireless sensor networks
- **GPU stream programs**: record program state on the GPU and inspect after program execution
- **Non-Intrusive FPGA Debugger (NIFD)**: GDB interface to FPGA application that interprets configure information to provide breakpoints and interactive debugging