PERG-Rx: An FPGA-based Pattern-Matching Engine with Limited Regular Expression Support for Large Pattern Database

Johnny Ho

Supervisor: Guy Lemieux

Date: September 11, 2009

University of British Columbia

The Pattern-Matching Problem

5312c39392c33372c3131372c35312c35332c35352c35322c33372c3131372c34382c35312c3
5352c35362c33372c3131372c35332c35342c3130322c35332c33372c3131372c35352c3534
2c35362c39382c33372c3131372c34382c35312c35302c34382c33372c3131372c35312c3531
2c3130322c35332c33372c3131372c35322c35372c39392c35372c33372c3131372c39372c31
30302c35322c34392c33372c3131372c3130302c39382c35312c35312c33372c3131372c343
82c31530065006e00640020006b5312c39392c33372c3131372c35312c35332c35352c35322
c33372c3131372c34382c35312c35352c35362c33372c3131372c35332c35342c3130322c353
32c33372c3131372c35352c35342c35362c39382c33372c3131372c34382c35312c35302c343
82c33372c3131372c35312c35312c3130322c35332c33372c3131372c35322c35372c39392c3
5372c33372c3131372c39372c3130302c35322c34392c33372c3131372c3130302c39382c35

The Pattern-Matching Problem

5312c39392c33372c3131372c35312c35332c35352c35322c33372c3131372c34382c35312c3
5352c35362c33372c3131372c35332c35342c3130322c35332c33372c3131372c35352c3534
2c35362c39382c33372c3131372c34382c35312c35302c34382c33372c3131372c35312c3531
2c3130322c35332c33372c3131372c35322c35372c39392c35372c33372c3131372c39372c31
30302c35322c34392c33372c3131372c3130302c39382c35312c35312c33372c3131372c343
82c31530065006e00640020006b5312c39392c33372c3131372c35312c35332c35352c35322
c33372c3131372c34382c35312c35352c35362c33372c3131372c35332c35342c3130322c353
32c33372c3131372c35352c35342c35362c39382c33372c3131372c34382c35312c35302c343
82c33372c3131372c35312c35312c3130322c35332c33372c3131372c35322c35372c39392c3
5372c33372c3131372c39372c3130302c35322c34392c33372c3131372c3130302c39382c35

Pattern Database

234ab3200000383 Fixed string

The Pattern-Matching Problem

5312c39392c33372c3131372c35312c35332c35352c35322c33372c3131372c34382c35312c3
5352c35362c33372c3131372c35332c35342c3130322c35332c33372c3131372c35352c3534
2c35362c39382c33372c3131372c34382c35312c35302c34382c33372c3131372c35312c3531
2c3130322c35332c33372c3131372c35322c35372c39392c35372c33372c3131372c39372c31
30302c35322c34392c33372c3131372c3130302c39382c35312c35312c33372c3131372c343
82c31530065006e00640020006b5312c39392c33372c3131372c35312c35332c35352c35322
c33372c3131372c34382c35312c35352c35362c33372c3131372c35332c35342c3130322c353
32c33372c3131372c35352c35342c35362c39382c33372c3131372c34382c35312c35302c343
82c33372c3131372c35312c35312c3130322c35332c33372c3131372c35322c35372c39392c3
5372c33372c3131372c39372c3130302c35322c34392c33372c3131372c3130302c39382c35

Pattern Database

234ab3200000383	Fixed string
21372{8}ef00{2}17ad	Multiple strings with fixed gaps

The Pattern-Matching Problem

5312c39392c33372c3131372c35312c35332c35352c35322c33372c3131372c34382c35312c3
5352c35362c33372c3131372c35332c35342c3130322c35332c33372c3131372c35352c3534
2c35362c39382c33372c3131372c34382c35312c35302c34382c33372c3131372c35312c3531
2c3130322c35332c33372c3131372c35322c35372c39392c35372c33372c3131372c39372c31
30302c35322c34392c33372c3131372c3130302c39382c35312c35312c33372c3131372c343
82c31530065006e00640020006b5312c39392c33372c3131372c35312c35332c35352c35322
c33372c3131372c34382c35312c35352c35362c33372c3131372c35332c35342c3130322c353
32c33372c3131372c35352c35342c35362c39382c33372c3131372c34382c35312c35302c343
82c33372c3131372c35312c35312c3130322c35332c33372c3131372c35322c35372c39392c3
5372c33372c3131372c39372c3130302c35322c34392c33372c3131372c3130302c39382c35

Pattern Database

234ab3200000383	Fixed string
21372{8}ef00{2}17ad	Multiple strings with fixed gaps
234a*00000*df	Wildcards

The Pattern-Matching Problem

5312c39392c33372c3131372c35312c35332c35352c35322c33372c3131372c34382c35312c3
5352c35362c33372c3131372c35332c35342c3130322c35332c33372c3131372c35352c3534
2c35362c39382c33372c3131372c34382c35312c35302c34382c33372c3131372c35312c3531
2c3130322c35332c33372c3131372c35322c35372c39392c35372c33372c3131372c39372c31
30302c35322c34392c33372c3131372c3130302c39382c35312c35312c33372c3131372c343
82c31530065006e00640020006b5312c39392c33372c3131372c35312c35332c35352c35322
c33372c3131372c34382c35312c35352c35362c33372c3131372c35332c35342c3130322c353
32c33372c3131372c35352c35342c35362c39382c33372c3131372c34382c35312c35302c343
82c33372c3131372c35312c35312c3130322c35332c33372c3131372c35322c35372c39392c3
5372c33372c3131372c39372c3130302c35322c34392c33372c3131372c3130302c39382c35

Pattern Database

# of Patterns

234ab3200000383	Fixed string
21372{8}ef00{2}17ad	Multiple strings with fixed gaps
234a*00000*df	Wildcards

The Pattern-Matching Problem

5312c39392c33372c3131372c35312c35332c35352c35322c33372c3131372c34382c35312c3
5352c35362c33372c3131372c35332c35342c3130322c35332c33372c3131372c35352c3534
2c35362c39382c33372c3131372c34382c35312c35302c34382c33372c3131372c35312c3531
2c3130322c35332c33372c3131372c35322c35372c39392c35372c33372c3131372c39372c31
30302c35322c34392c33372c3131372c3130302c39382c35312c35312c33372c3131372c343
82c31530065006e00640020006b5312c39392c33372c3131372c35312c35332c35352c35322
c33372c3131372c34382c35312c35352c35362c33372c3131372c35332c35342c3130322c353
32c33372c3131372c35352c35342c35362c39382c33372c3131372c34382c35312c35302c343
82c33372c3131372c35312c35312c3130322c35332c33372c3131372c35322c35372c39392c3
5372c33372c3131372c39372c3130302c35322c34392c33372c3131372c3130302c39382c35

Pattern Database

234ab3200000383	Fixed string
21372{8}ef00{2}17ad	Multiple strings with fixed gaps
234a*00000*df	Wildcards

# of Patterns

Pattern Length

The Pattern-Matching Problem

- Applications:
  - Network intrusion detection systems (NIDS)
    - Deep packet inspection
    - Well-studied
    - Several thousands in number of patterns (*Snort* database)
  - Antivirus
    - Virus signature matching
    - PERG
    - Over 80,000 patterns in *ClamAV* database used

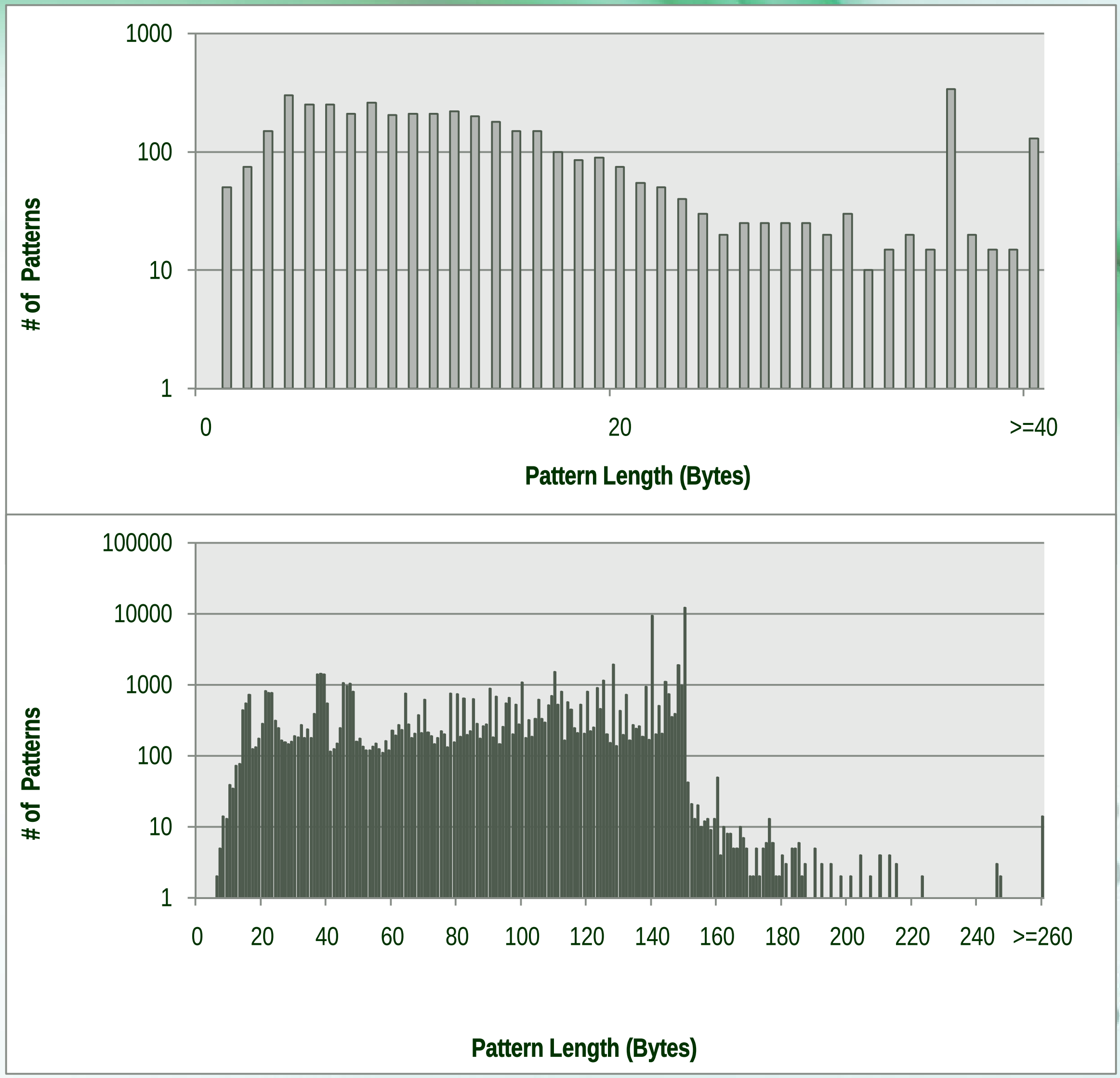
Motivation

- Antivirus is slow
  - Up to over 500% slowdown in I/O intensive process
  - Bottleneck: Pattern-Matching
  - Virus signature database
    - Large in number and range of lengths
    - Requires frequent update

Motivation

NIDS
(Snort)

Antivirus
(ClamAV)



Related Works

- **Existing approaches:**
  - FSM (Aho-Corsaik) , Bloom filter, Perfect/Cuckoo hash

	Regular Expression	**Dynamic Update**	**Resource Density**
FSM	Excellent	Poor	Poor
Bloom filter	Poor	Excellent	Excellent
Perfect/Cuckoo hash	Medium	Medium	Medium

Contributions

- **PERG : A FPGA-based pattern-matching engine for ClamAV**
  - Support limited regular expression
  - 24x better density than the next-best competitor (excluding Bloom filter)
  - 15x faster than software antivirus scanner

	Regular Expression	**Dynamic Update**	**Resource Density**
FSM	Excellent	Poor	Poor
Bloom filter	Poor	Excellent	Excellent
Perfect/Cuckoo hash	Medium	Medium	Medium
PERG	**Good**	**Good**	**Good**

Contributions

- **A Novel Hardware Architecture**
  - Handle pattern matching in a multi-staged manner without resorting to high-bandwidth off-chip memory requirement
- **A Novel Filter Consolidation Algorithm**
  - Reduce the hardware resources required by packing filter units into high capacity, thus reducing the number of filter units needed.
- **Circular State Buffer**
  - Support multiple traces of multi-segmented patterns with zero false negative probability
- **Limited Regular Expression Support**
  - Support for wildcard operators to detect polymorphic virus

Contributions

- Published in three conferences:

1. J. Ho, G. Lemieux, “PERG: A Scalable Pattern-matching Accelerator,” CMC Microsystems and Nanoelectronics Research Conference, Ottawa, pp. 29-32, October 2008.

2. J. Ho, G. Lemieux, “PERG: A Scalable FPGA-based Pattern-matching Engine with Consolidated Bloomier Filters,” IEEE International Conference on Field-Programmable Technology, Taipei, Taiwan, December 2008, pp. 73-80.

3. J. Ho, G. Lemieux, “PERG-Rx: A Hardware Pattern-matching Engine Supporting Limited Regular Expressions,” ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, California, pp. 257-260, February 2009.

Background: Bloom Filters

- Boolean hash table
  - False: Input **MUST** not be a pattern in database
    - Zero false negative probability
  - True: Input **MAY** be a pattern in database
    - False positive probability due to hash collision
    - Do not know which pattern is the potential match
      - Exact matching is needed and complex
- Use multiple hash functions to reduce false positive probability
  - All hash locations returned must be true for a match in a Bloom filter
- One Bloom filter is needed for each input (hash) length

Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table

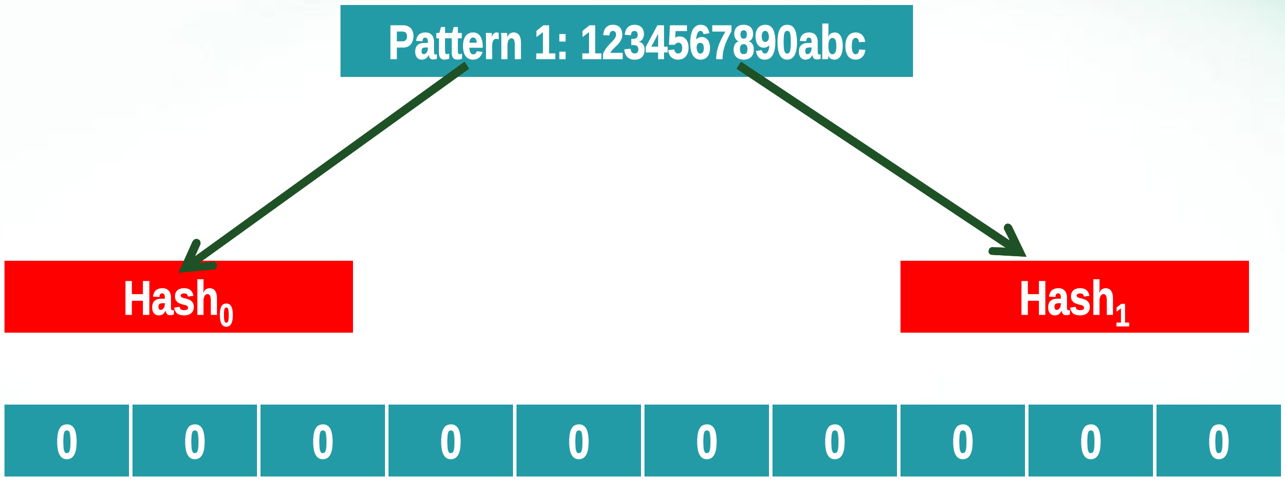
Pattern 1: 1234567890abc

$Hash_0$ **$Hash_1$**

0	0	0	0	0	0	0	0	0	0

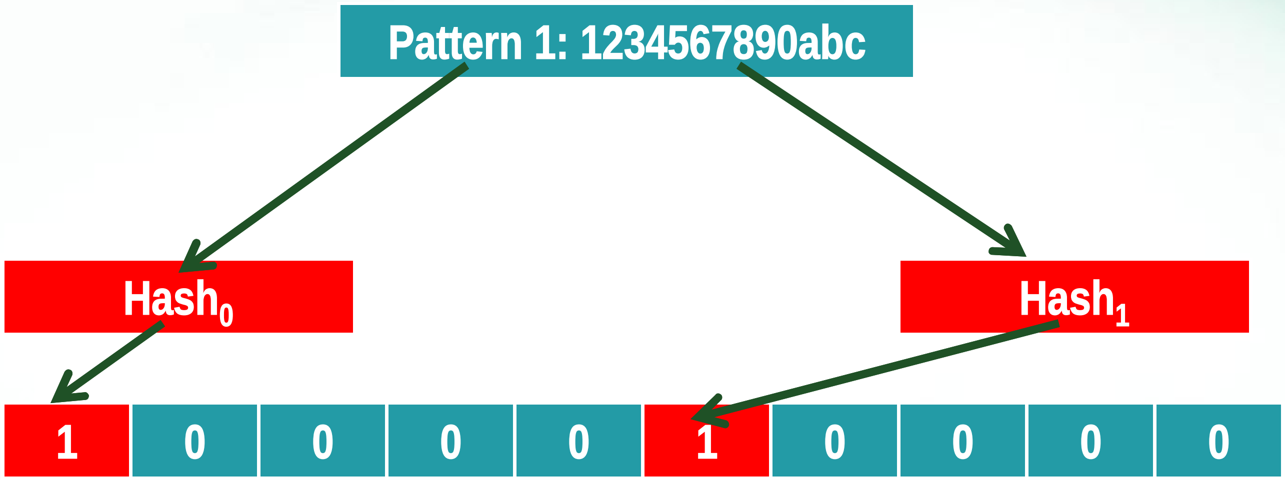
Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table



Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table



Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table

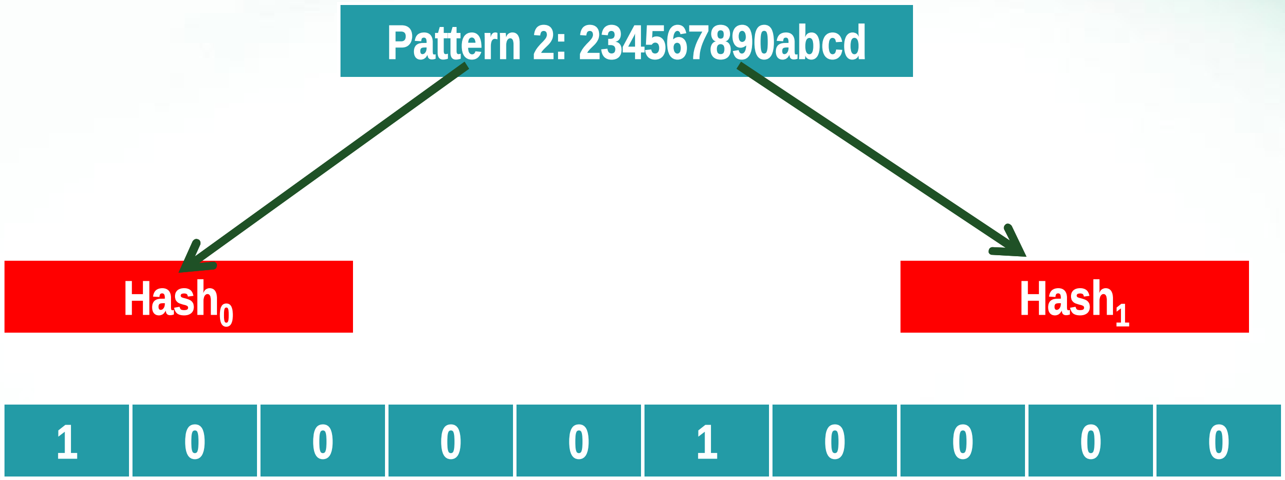
Pattern 2: 234567890abcd

$Hash_0$ **$Hash_1$**

1	0	0	0	0	1	0	0	0	0

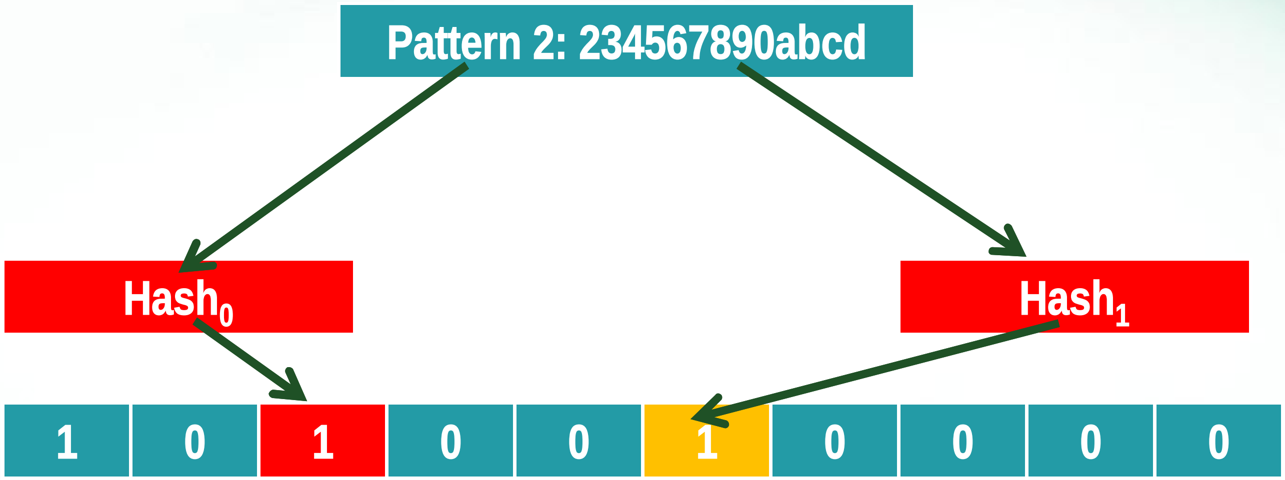
Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table



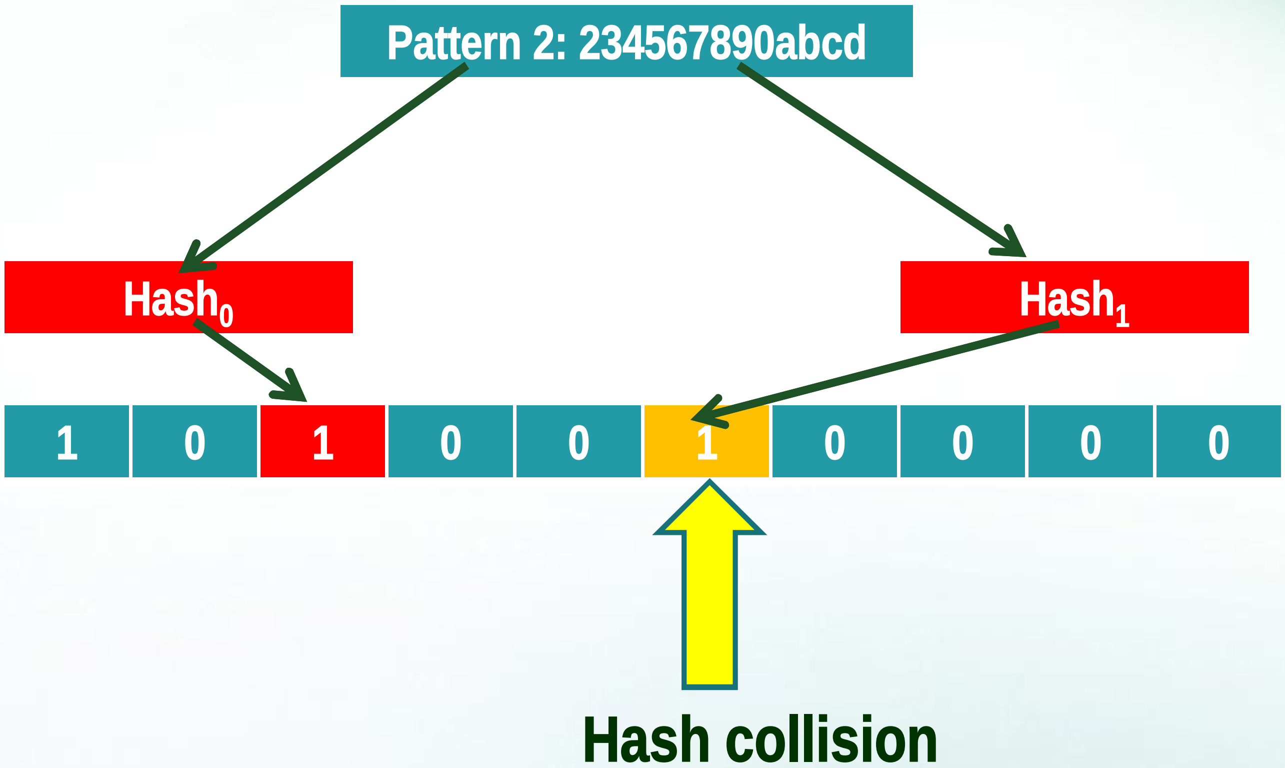
Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table



Background: Bloom Filters

- Construction: Hash patterns in database to the Boolean hash table



Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations

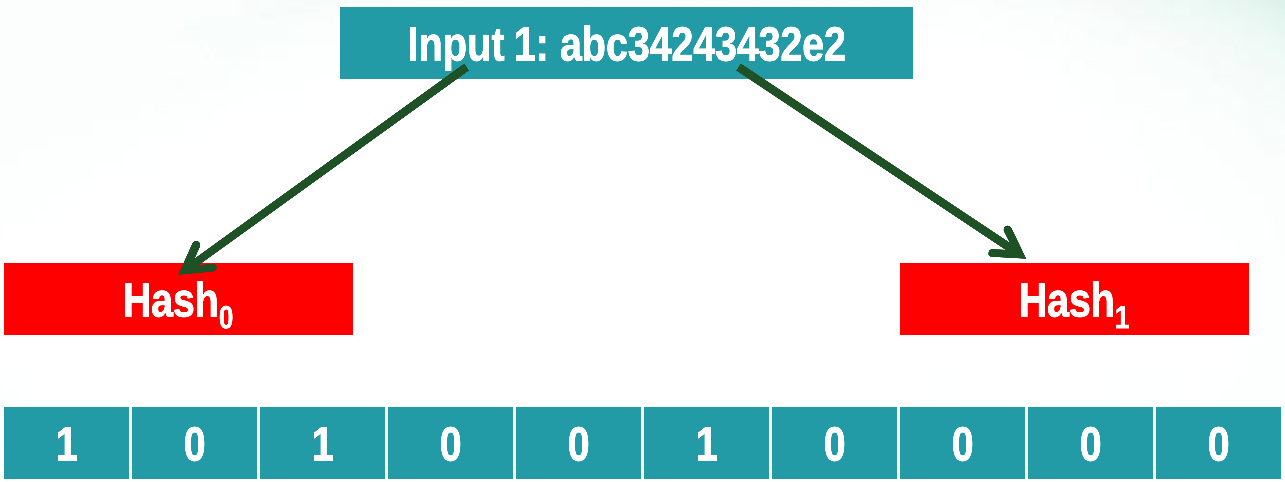
Input 1: abc34243432e2

Hash$_0$ **Hash$_1$**

1	0	1	0	0	1	0	0	0	0

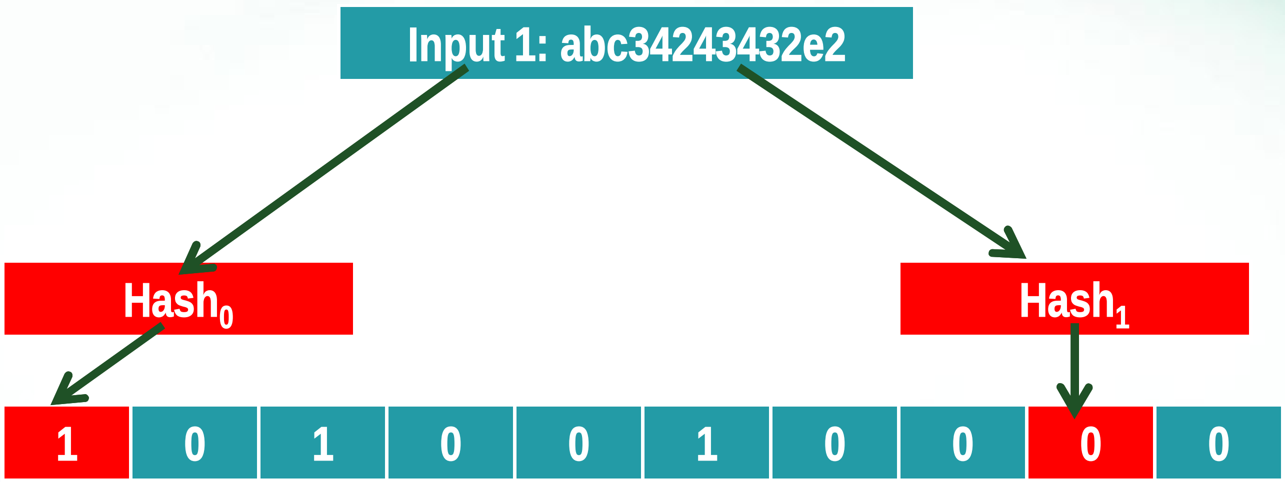
Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations



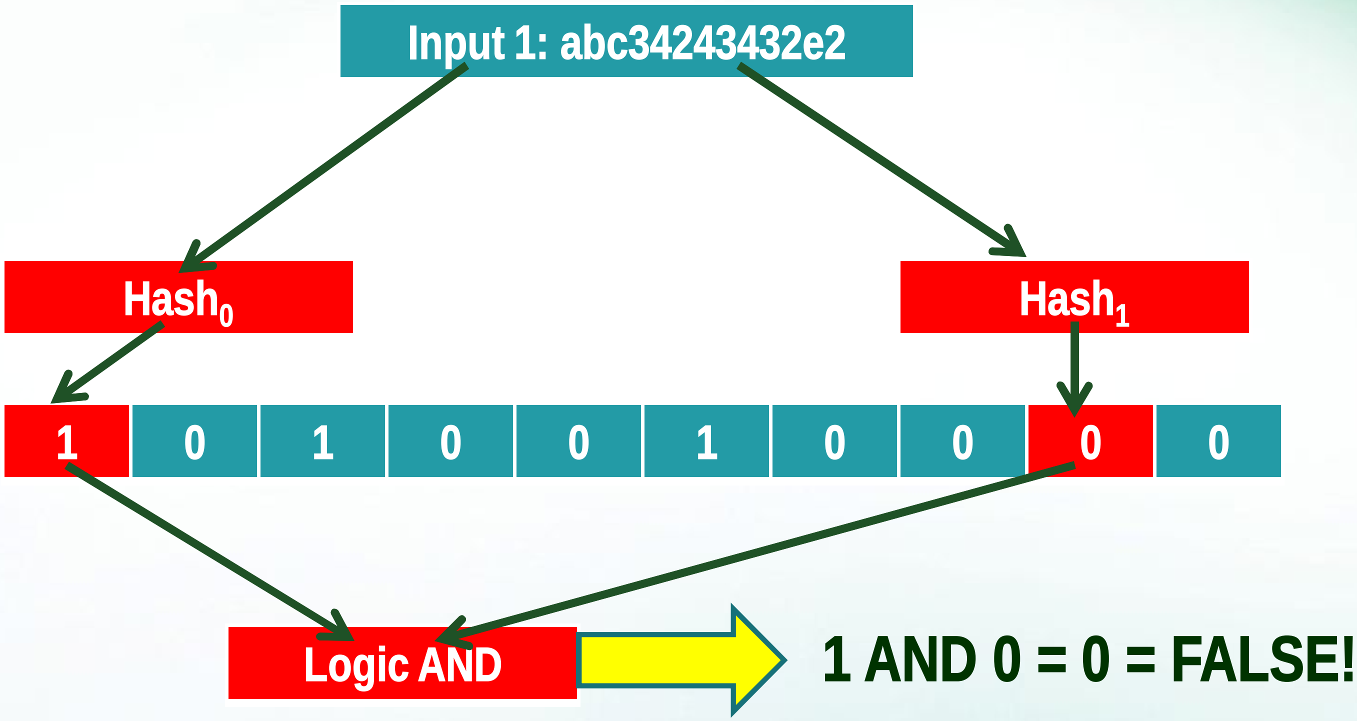
Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations



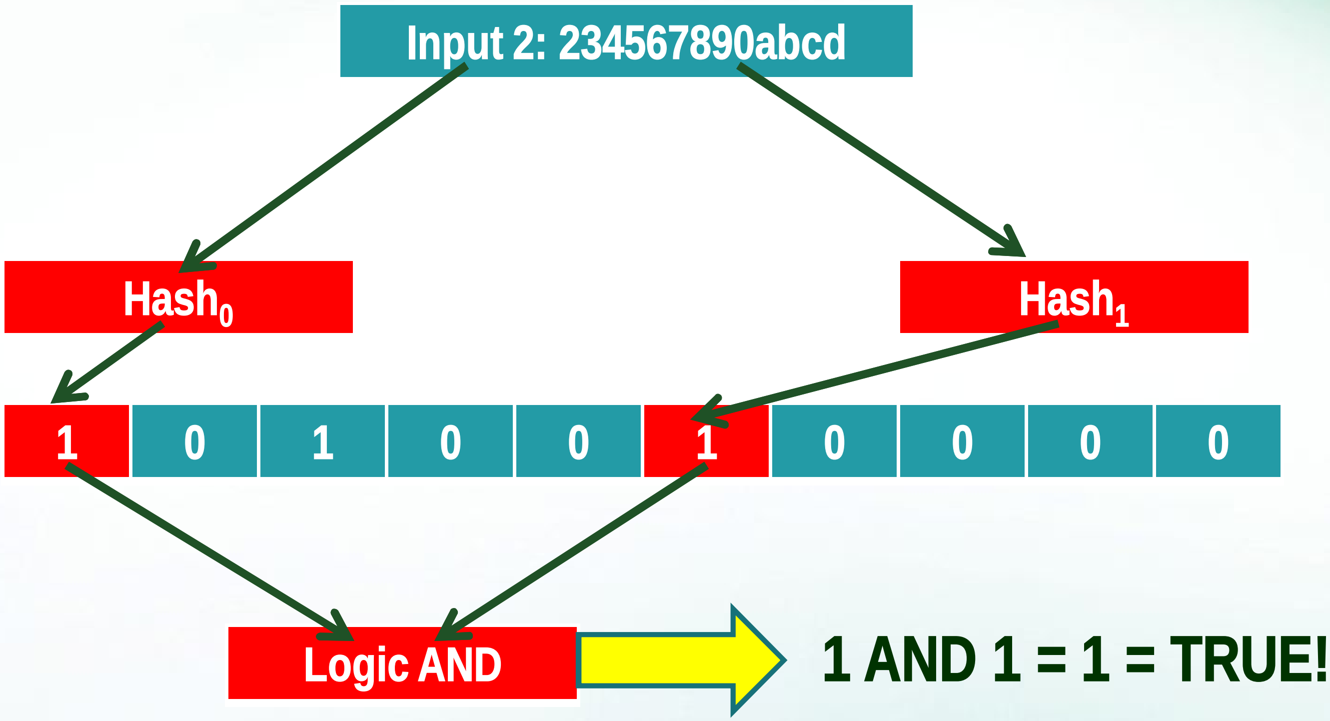
Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations



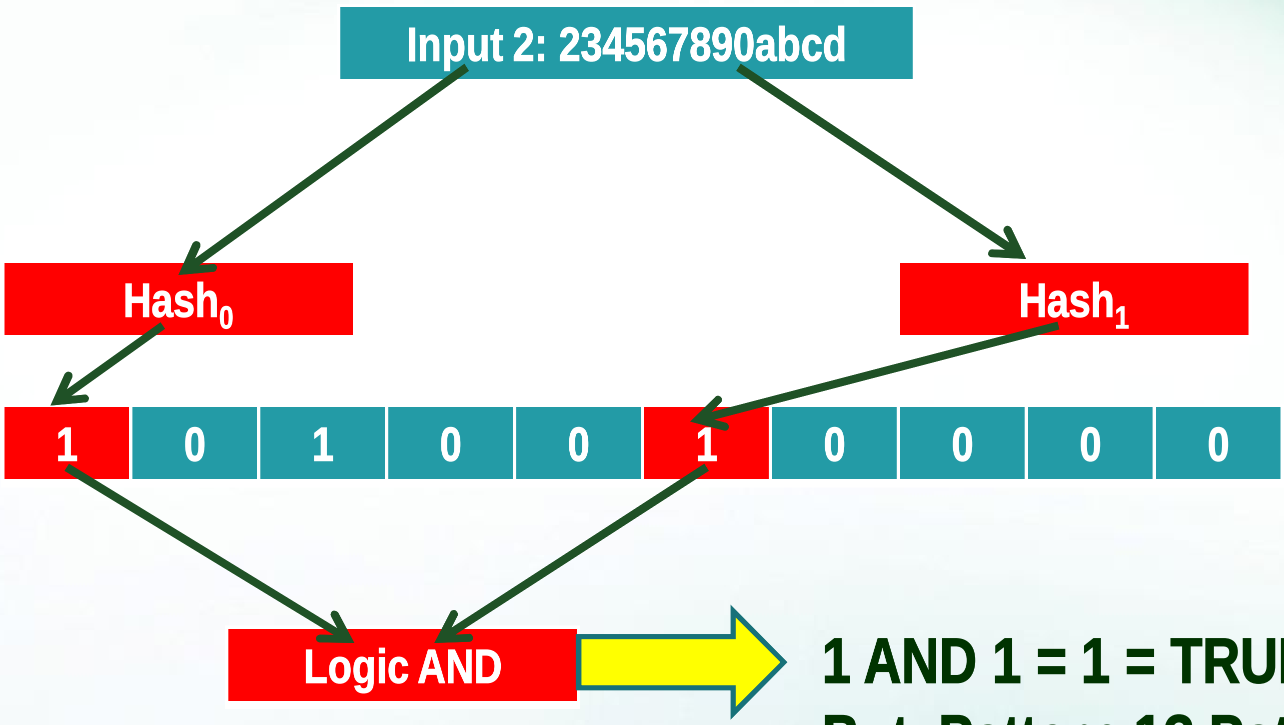
Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations



Background: Bloom Filters

- Usage: Hash input and logic-AND Boolean values at the hash locations



1 AND 1 = 1 = TRUE!

But..Pattern 1? Pattern 2?

Background: Bloomier Filters

- Structurally similar to Bloom filter
  - Resource efficient
  - Zero false negative probability
  - False positive probability
- Perfect-hash capability
  - Associate hash location with ONE single pattern
- Use multiple hash functions
  - Higher theoretical setup success rate than traditional perfect hash

Background: Bloomier Filters

- Construction: Start off similarly to Bloom filter; hash each pattern in database one by one into a hash table

Hash$_0$ **Hash$_1$**

0	0	0	0	0	0	0	0	0	0

Background: Bloomier Filters

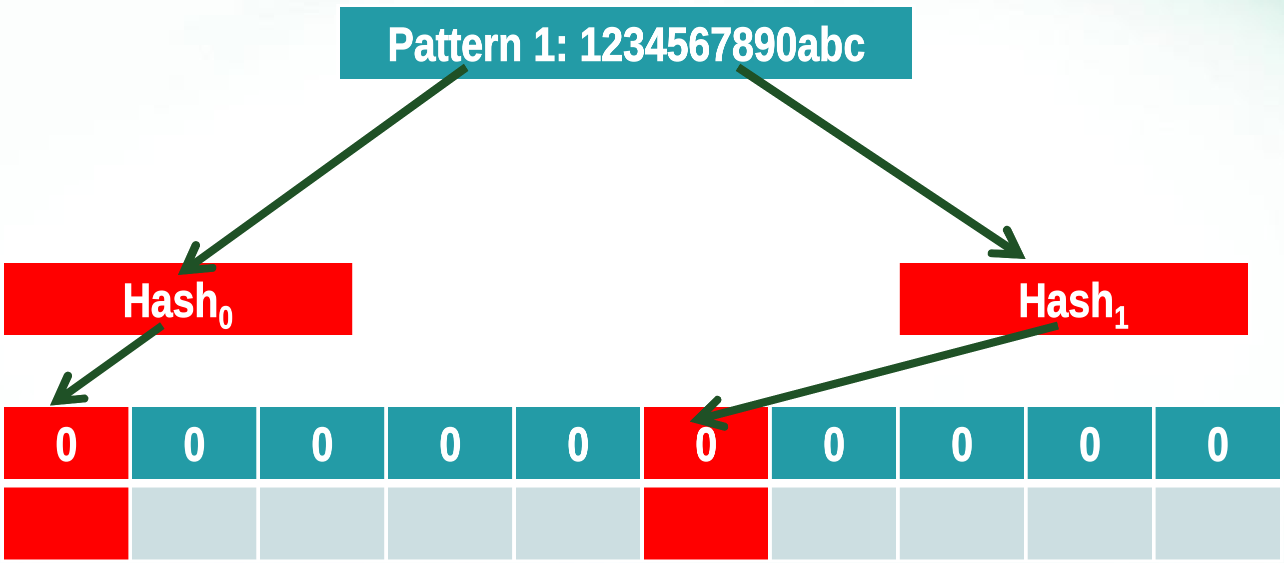
- Instead of storing Boolean membership information, stores two attributes: a *hash select* and the pattern itself

Hash_0 **Hash_1**

0	0	0	0	0	0	0	0	0	0	**Hash Select**
										Pattern

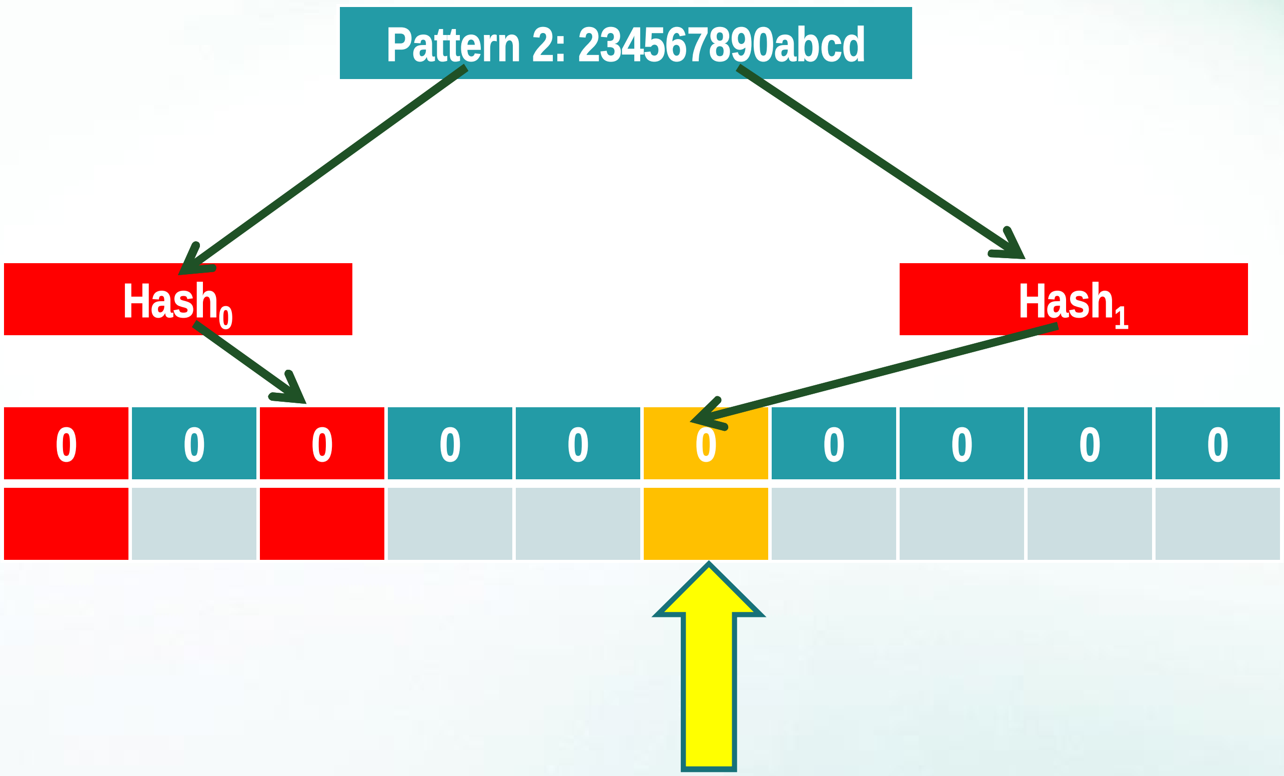
Background: Bloomier Filters

- Construction: Hash patterns in database to the hash table



Background: Bloomier Filters

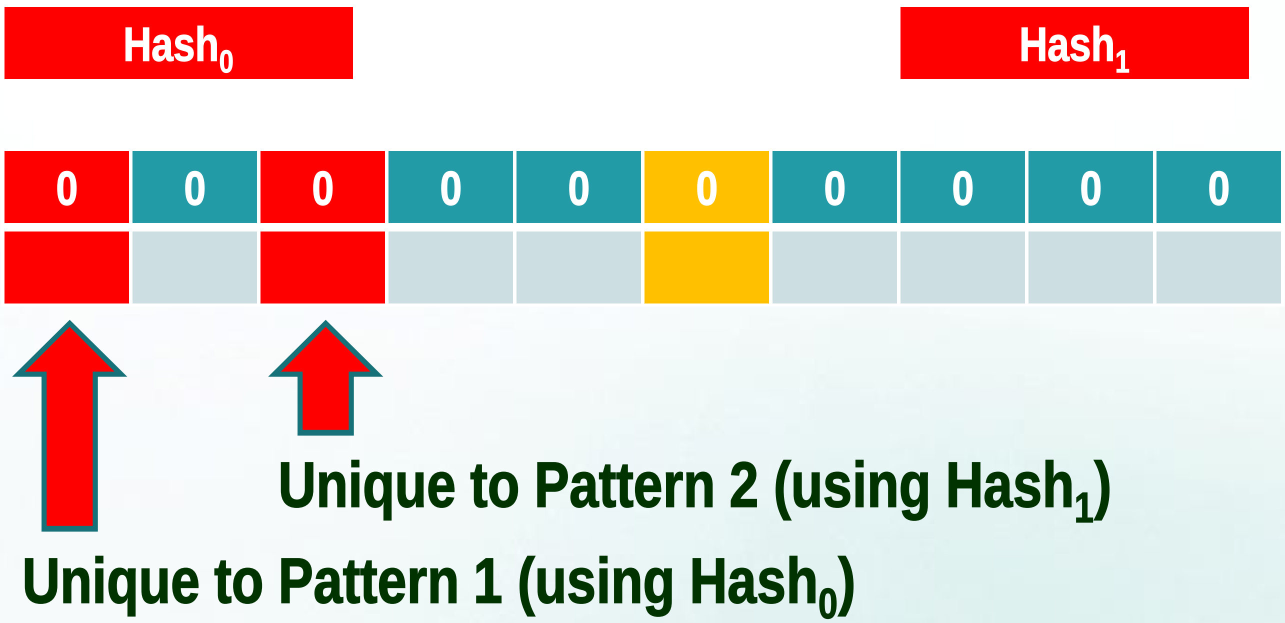
- As with Bloom filter and any other hash-based scheme, collision is inevitable



Hash collision between Pattern 1 and 2

Background: Bloomier Filters

- Identify hash location that is uniquely occupied by a pattern
  - If *N* hash functions are used, only one out of the *N* hash locations need to be unique



Background: Bloomier Filters

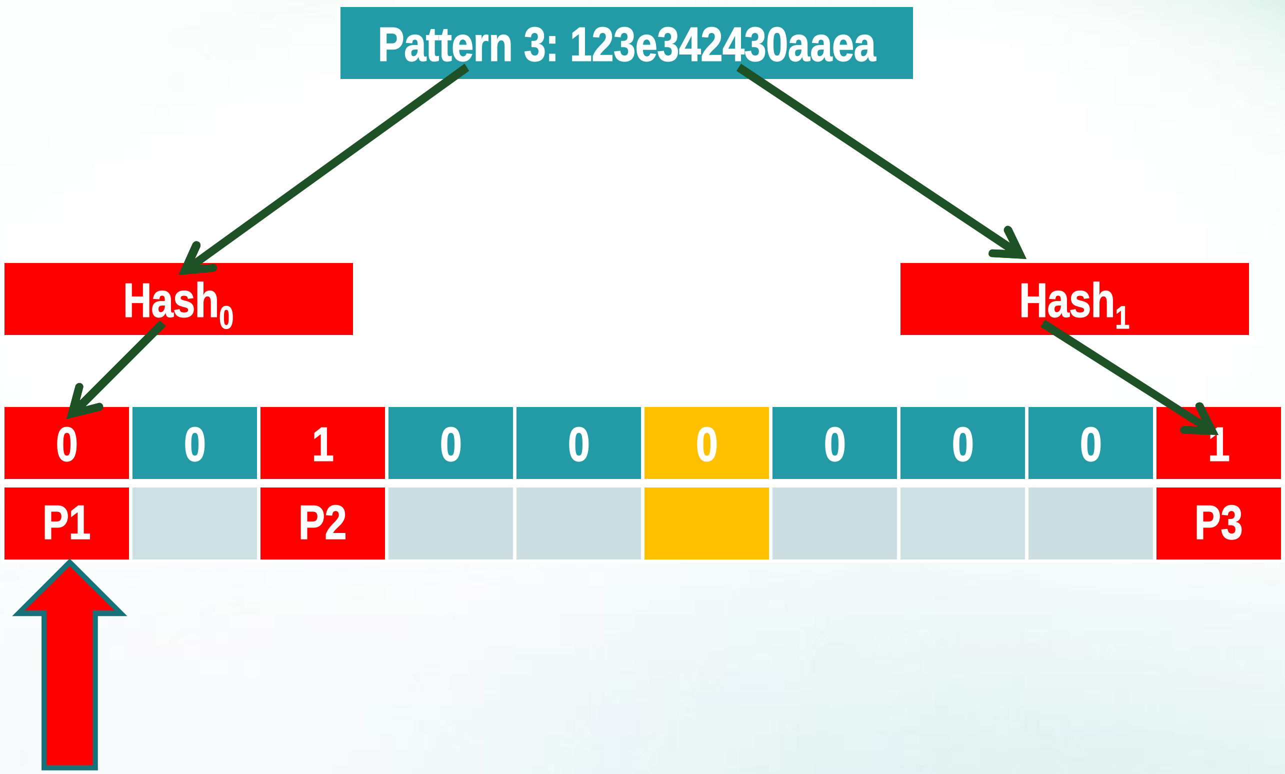
- Store the pattern at its uniquely associated location
- Store a hash select value to identify which of the *N* hash function will point to this unique location

$Hash_0$ $Hash_1$

0	0	1	0	0	0	0	0	0	0
P1		P2							

Background: Bloomier Filters

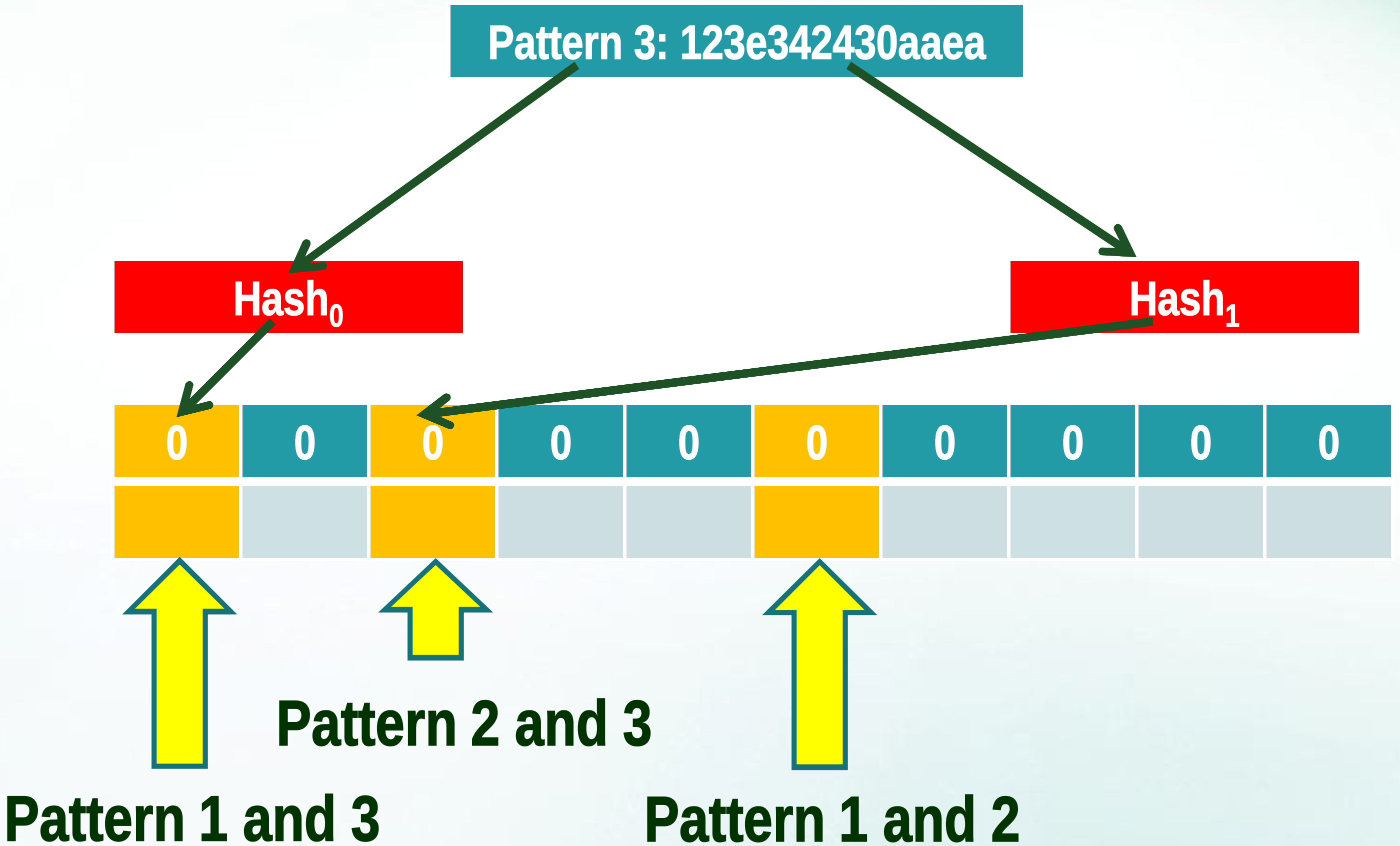
- A location can be uniquely associated with a pattern even if it exists in multiple pattern *neighborhoods*



Pattern 1 and 3 both map to this location

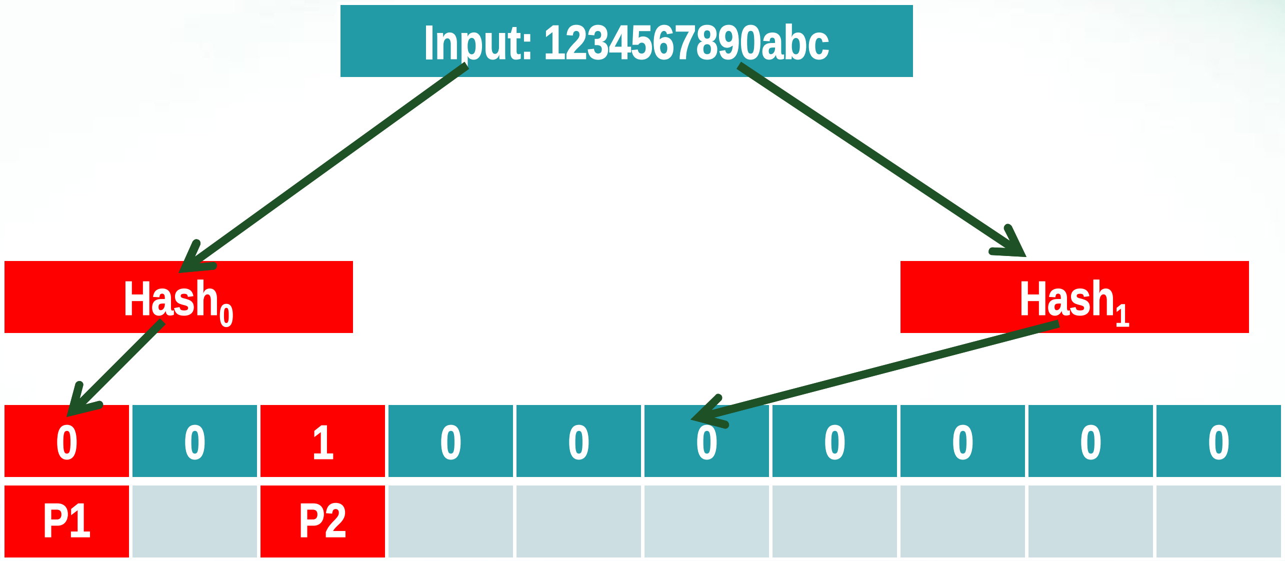
Background: Bloomier Filters

- Construction may fail if unique association between hash location and input pattern cannot be achieved



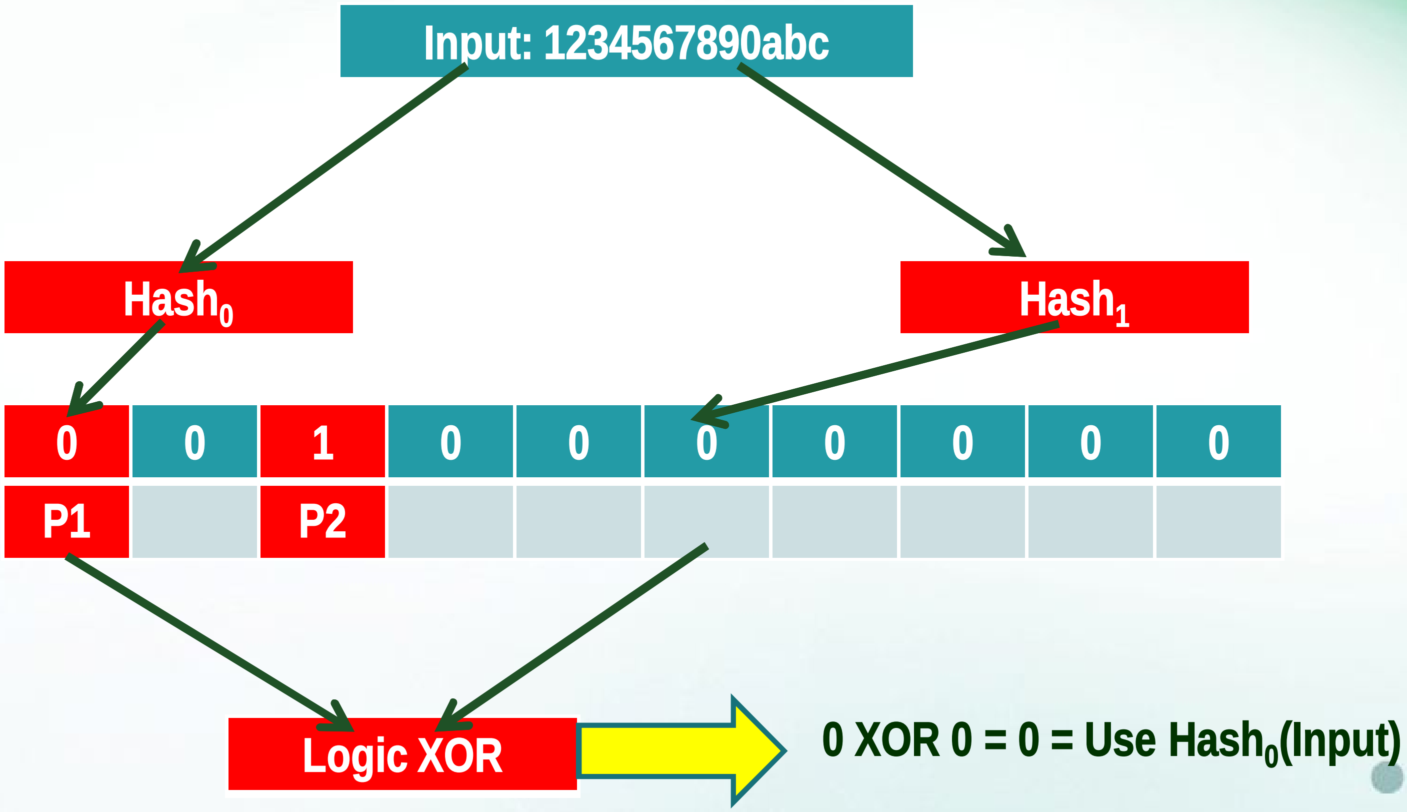
Background: Bloomier Filters

- Usage: Similar to Bloom filter, hash the input with the *N* hash functions



Background: Bloomier Filters

- Logic-XOR *hash select* values at the *N* hash locations to determine which hash location stores the unique pattern

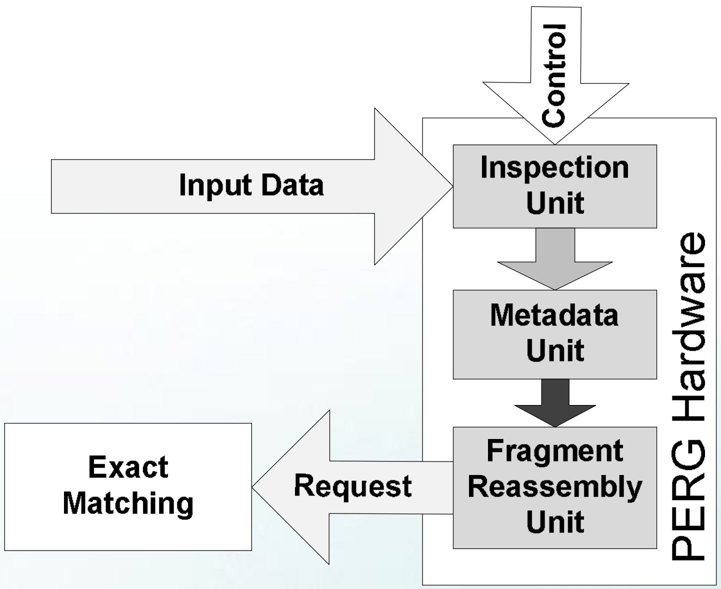


PERG System Overview

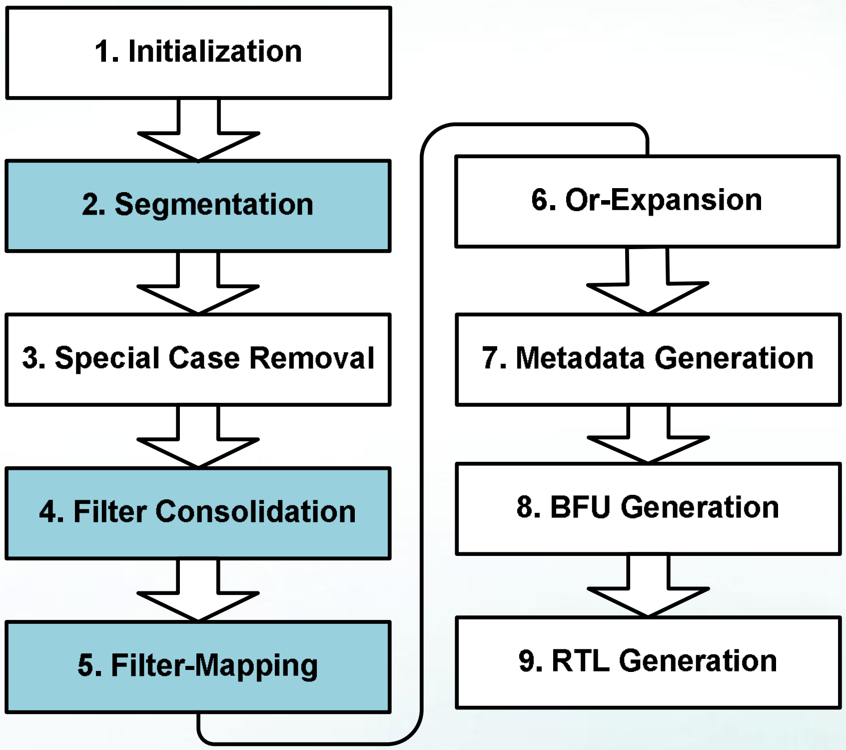
- Performs virus pattern matching on hardware
  - Rely on host to perform exact-matching
  - Communicate with host system through PCI bus
- Contains two parts
  - Pattern Compiler
    - Input: pattern database
    - Output: HDL and memory initialization file
    - Breaking up patterns into segments for optimization and regular-expression support purpose
  - Configurable Hardware Architecture
    - Virtex II-Pro FPGA + 4 MB SRAM

PERG System Overview

- Hardware contains three units:
  - Inspection Unit
    - Contains Bloomier Filter Units (BFU) to filtering input for patterns
  - Metadata Unit
    - Stores *Metadata* that contains information on how to link segments of patterns back together
  - Fragment Reassembly Unit (FRU)
    - Keep track of traces of multi-segmented patterns and link them back accordingly



Pattern Compiler



Pattern Compiler: Segmentation

ABCD**{4}**EFG

Pattern Compiler: Segmentation

ABCD**{4}**EFG

ABCD**{4}** EFG

1. Split at displacement/wildcard

Pattern Compiler: Segmentation

ABCD{4}EFG

ABCD{7}   EFG

1. Split at displacement/wildcard
2. Adjust offset

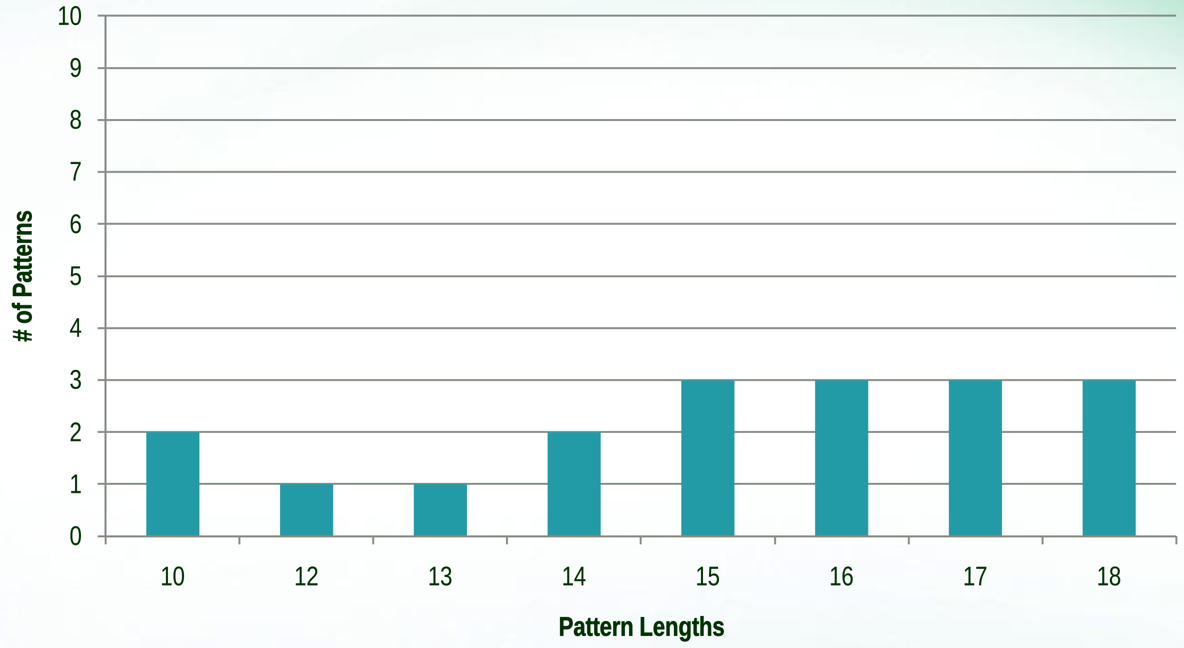
Pattern Compiler: Filter Consolidation

- Patterns in ClamAV comes in a wide range of lengths
  - Each pattern length would require its own BFU
  - Pattern length range is not evenly distributed

- Filter consolidation reduces the number of pattern lengths by packing patterns at different length together
  - Packing begins at the longest pattern length
  - When the utilization threshold of a BFU hash table is met, assign this length as a BFU length
  - Segments whose lengths do not match any BFU length are split into two overlapping segments of equal length
    - Length of the new overlapping segments is equal to the length of the nearest shorter BFU length
    - Splitting is done in *filter-mapping* stage

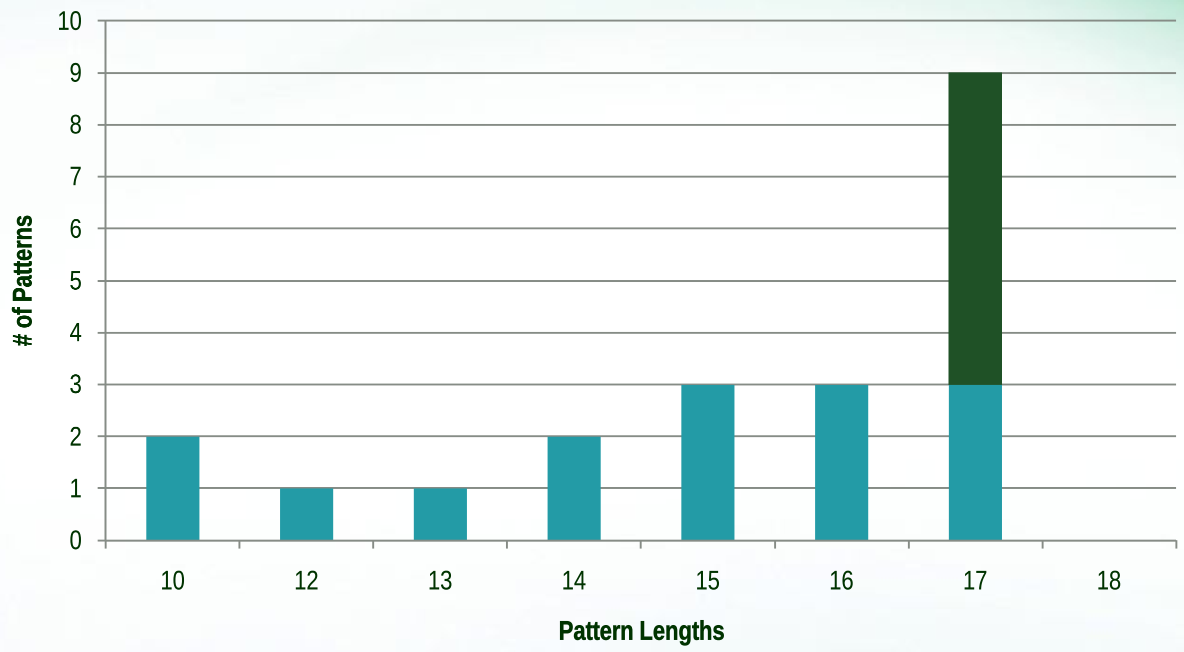
Pattern Compiler: Filter Consolidation

- Assume threshold is set to be 9 patterns



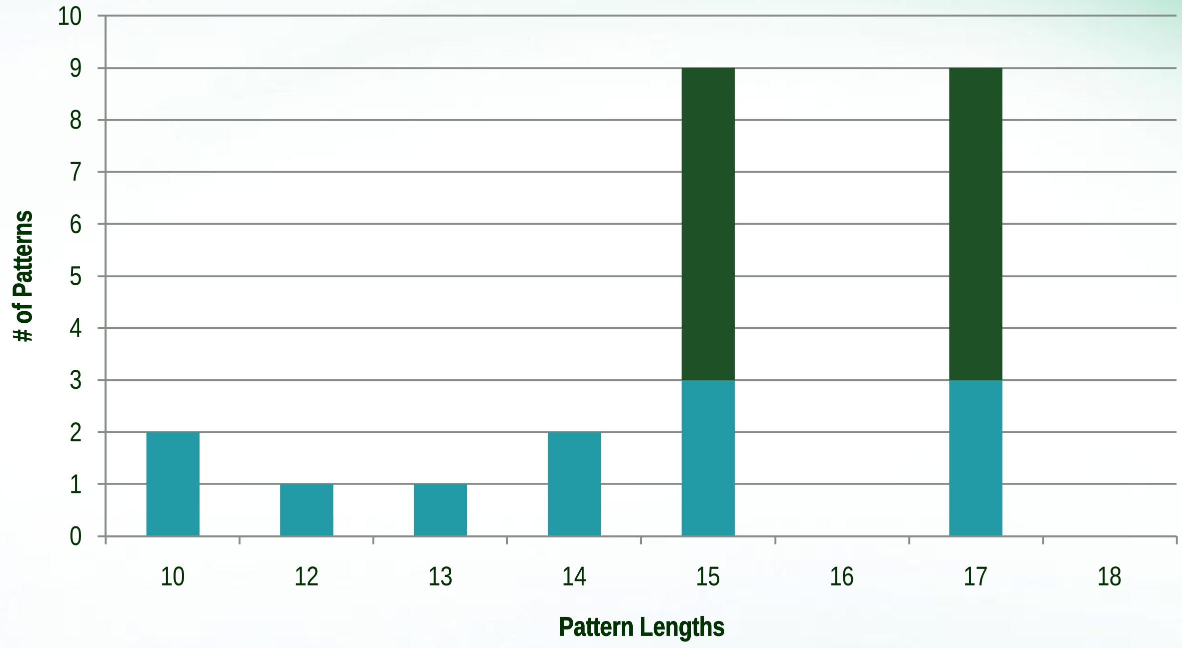
Pattern Compiler: Filter Consolidation

- If the current length is below threshold, decrement the length



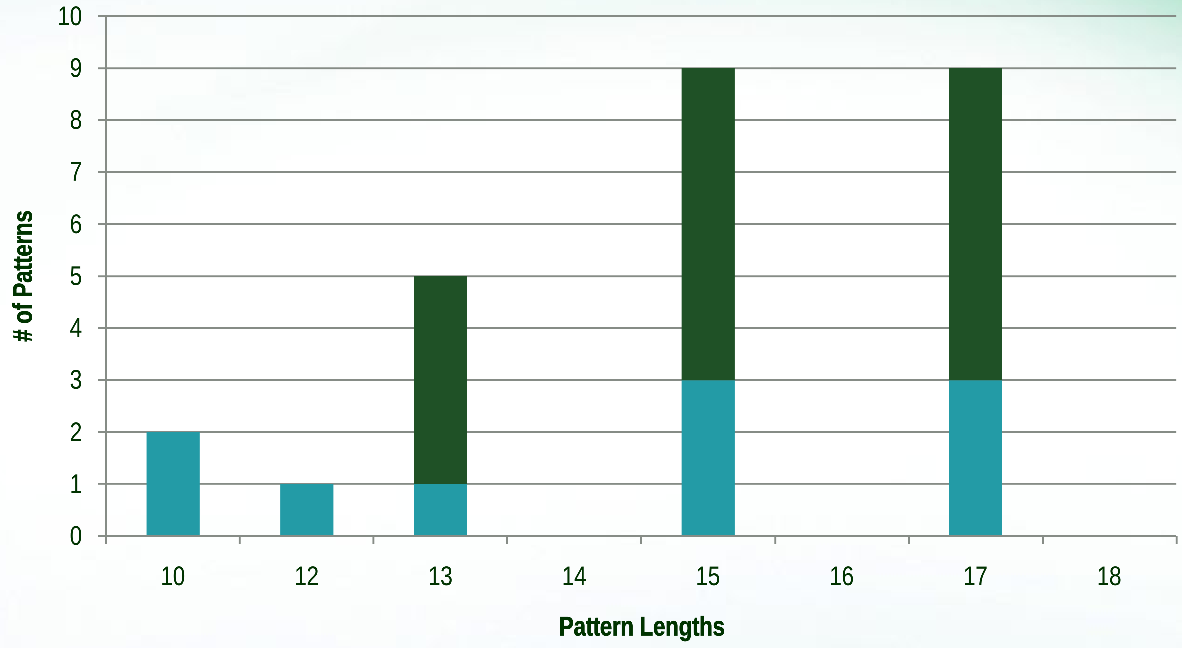
Pattern Compiler: Filter Consolidation

- If a length is skipped, patterns at the skipped length are divided to two overlapping segments



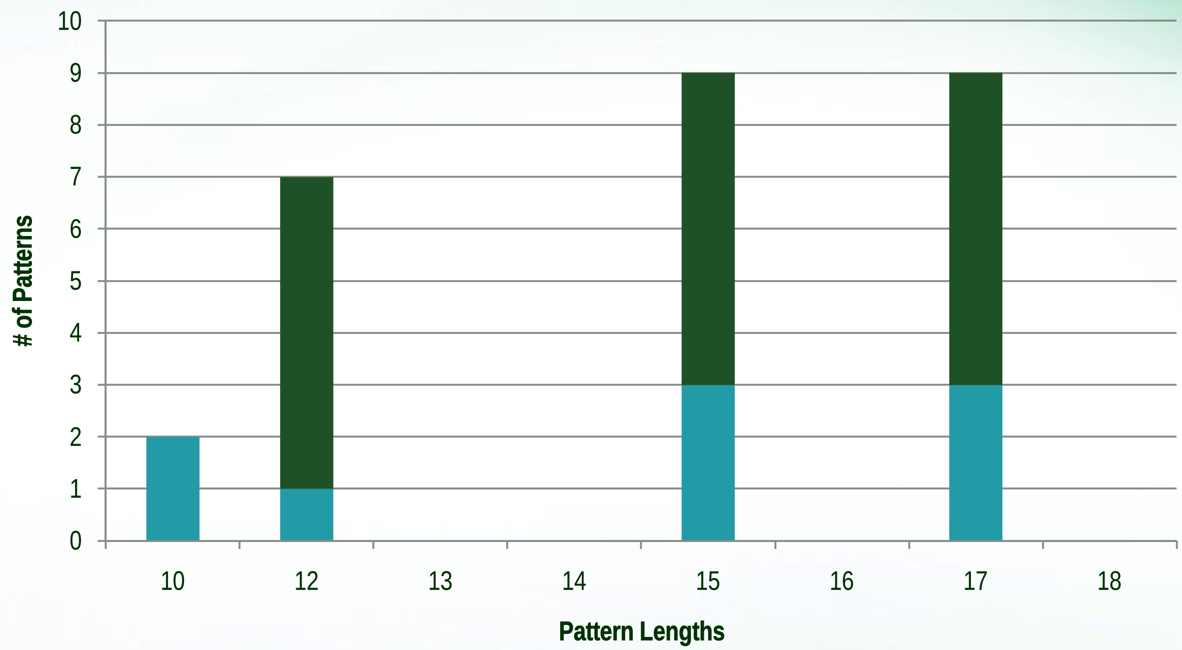
Pattern Compiler: Filter Consolidation

- Since the # of segments doubled, its cost contribution also doubles



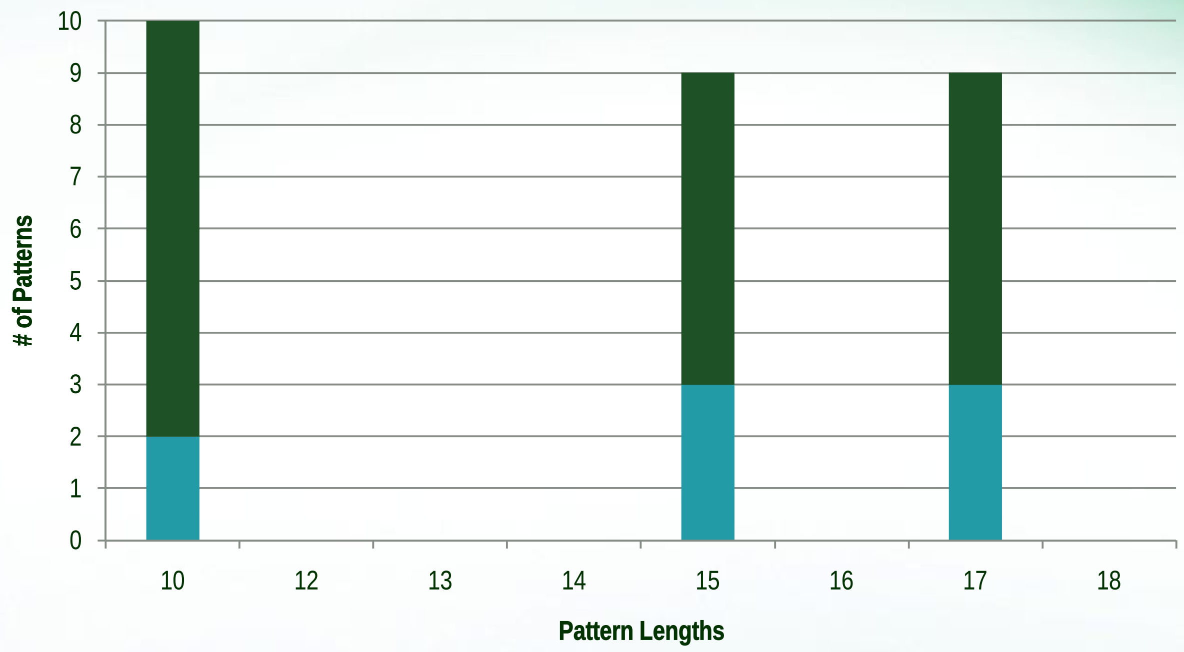
Pattern Compiler: Filter Consolidation

- The contribution however only needs to be doubled once



Pattern Compiler: Filter Consolidation

- Consolidation completes at user-defined minimum length



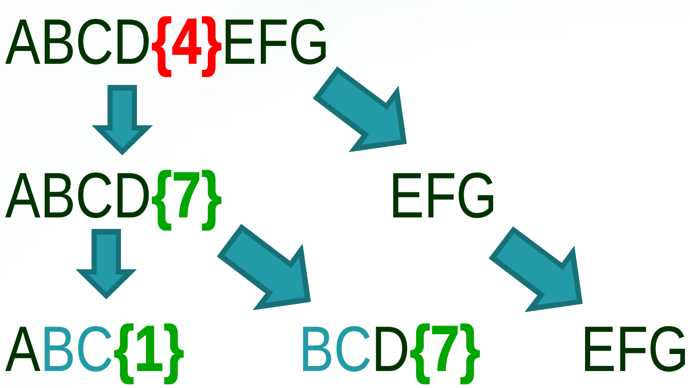
Pattern Compiler: Filter Mapping

ABCD**{4}**EFG

ABCD**{7}** EFG

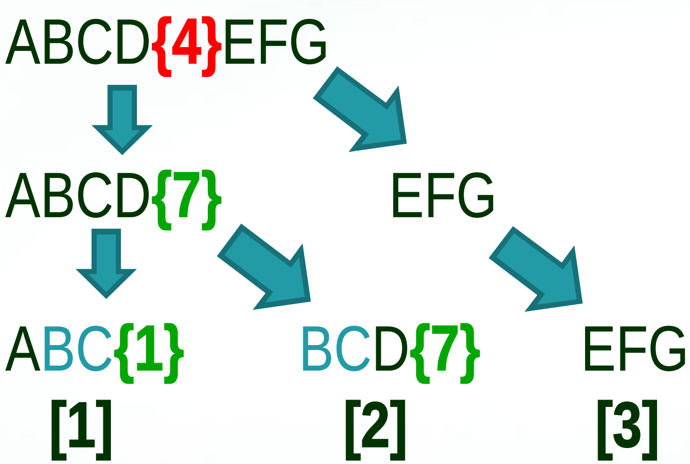
1. Split at displacement/wildcard
2. Adjust offset

Pattern Compiler: Filter Mapping



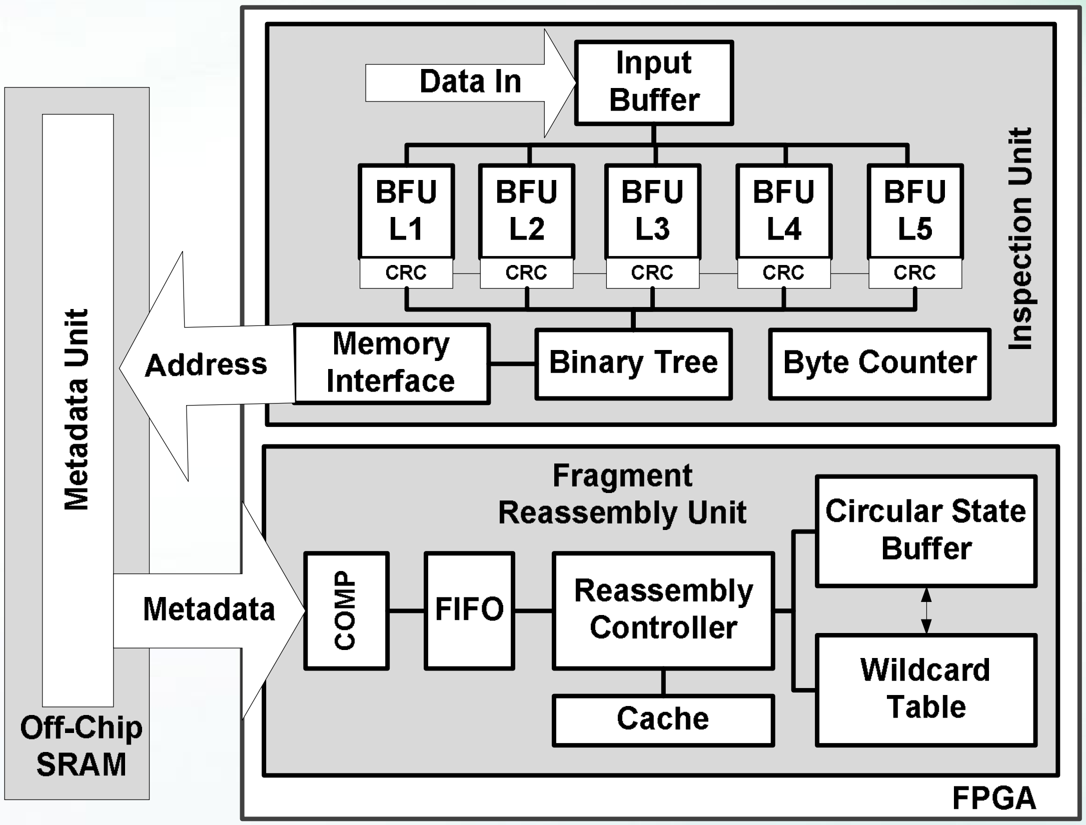
1. Split at displacement/wildcard
2. Adjust offset
3. Assume BFU length = 3 character, split the unfit segment into two overlapping segments

Pattern Compiler: Filter Mapping

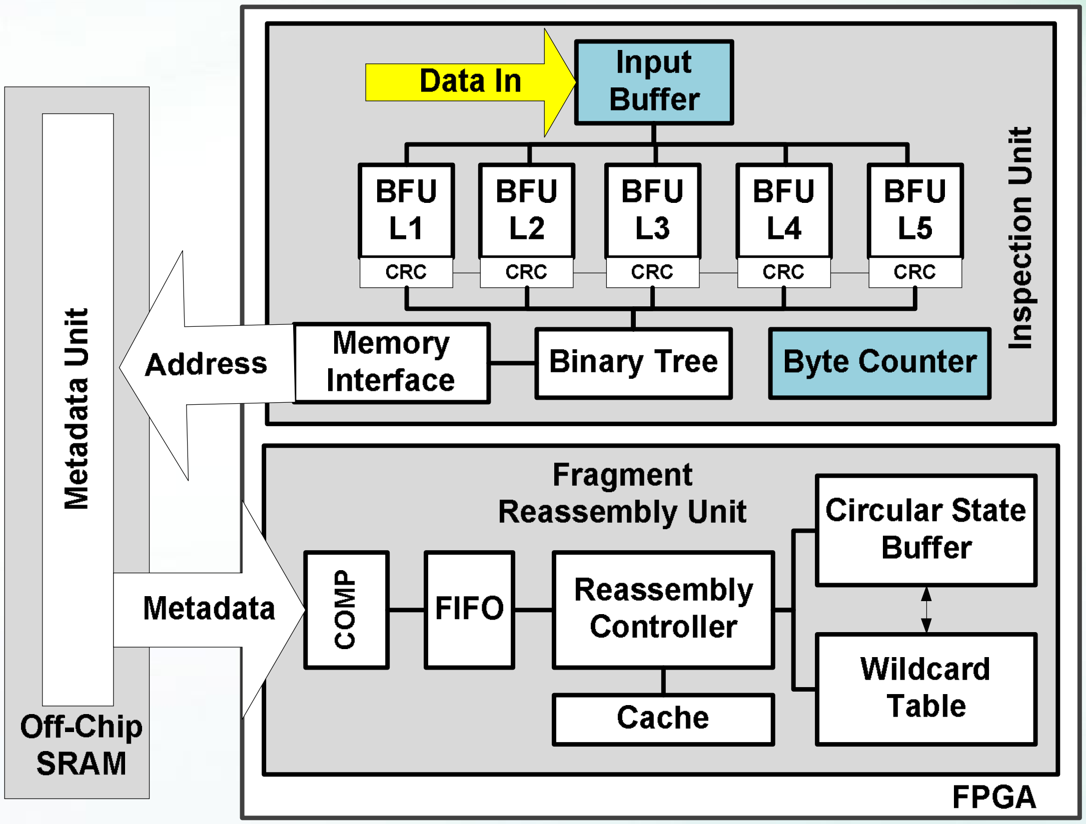


1. Split at displacement/wildcard
2. Adjust offset
3. Assume BFU length = 3 character, split the unfit segment into two overlapping segments
4. Assign Link #

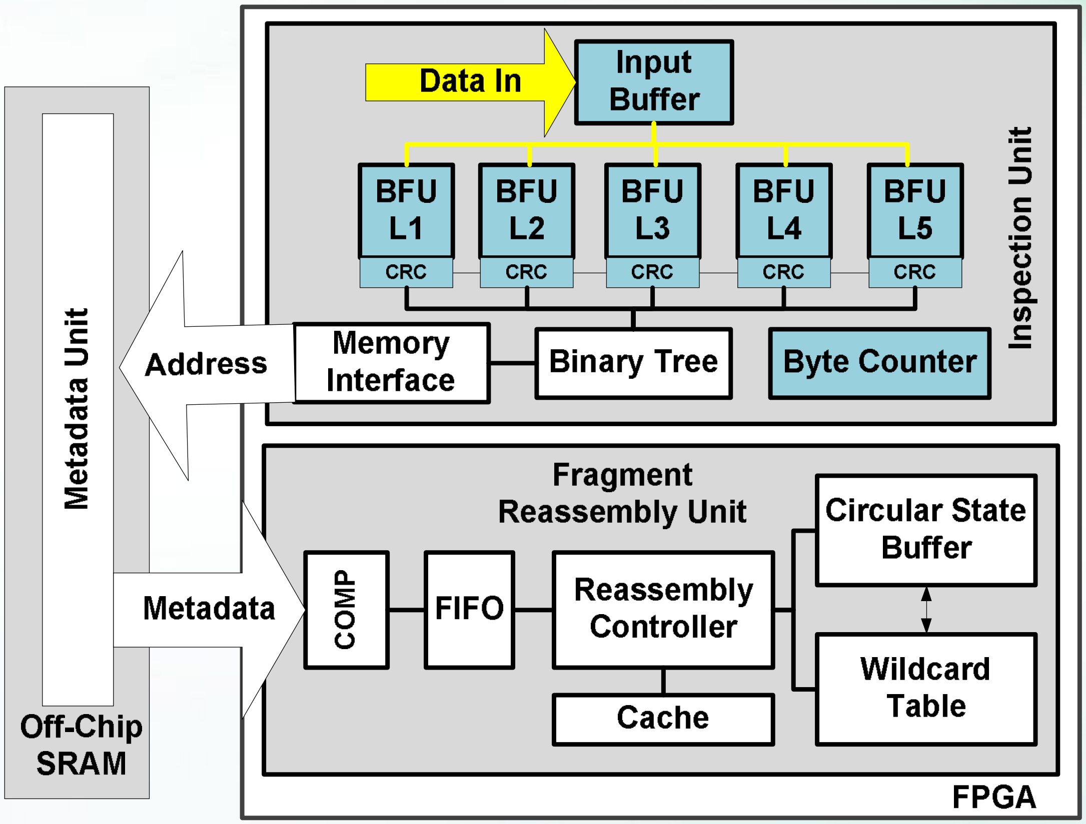
Hardware Architecture



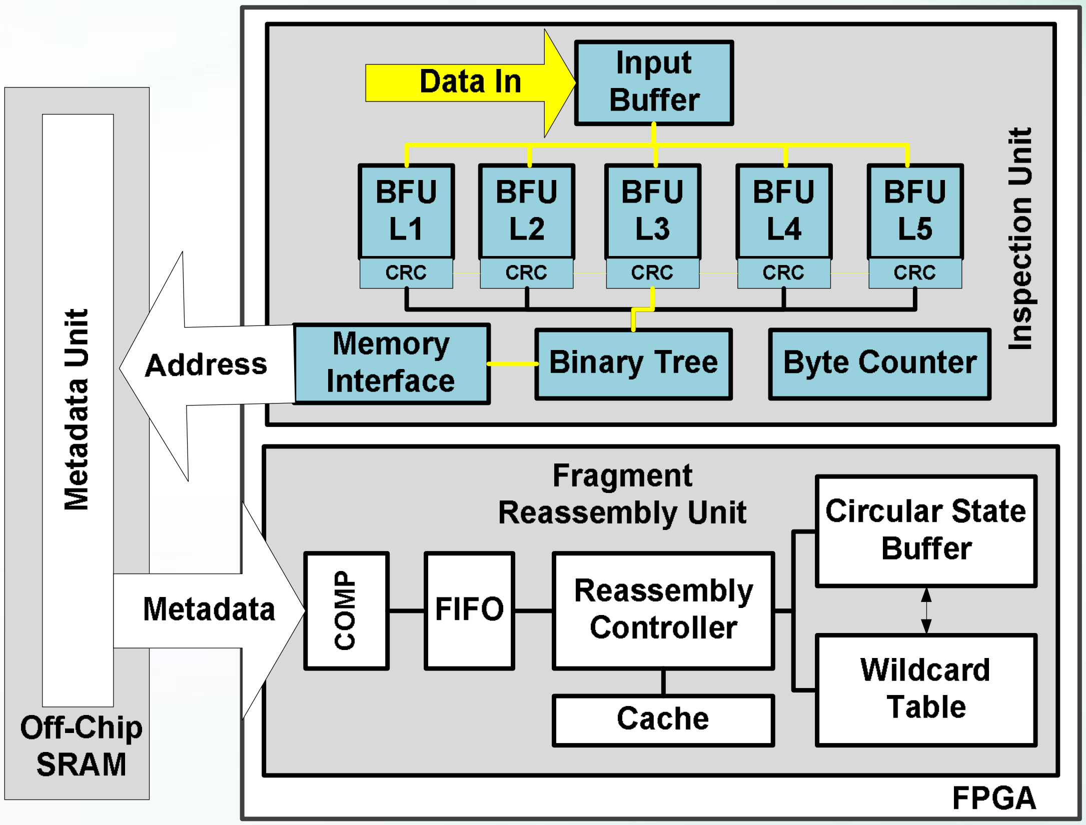
Hardware Architecture



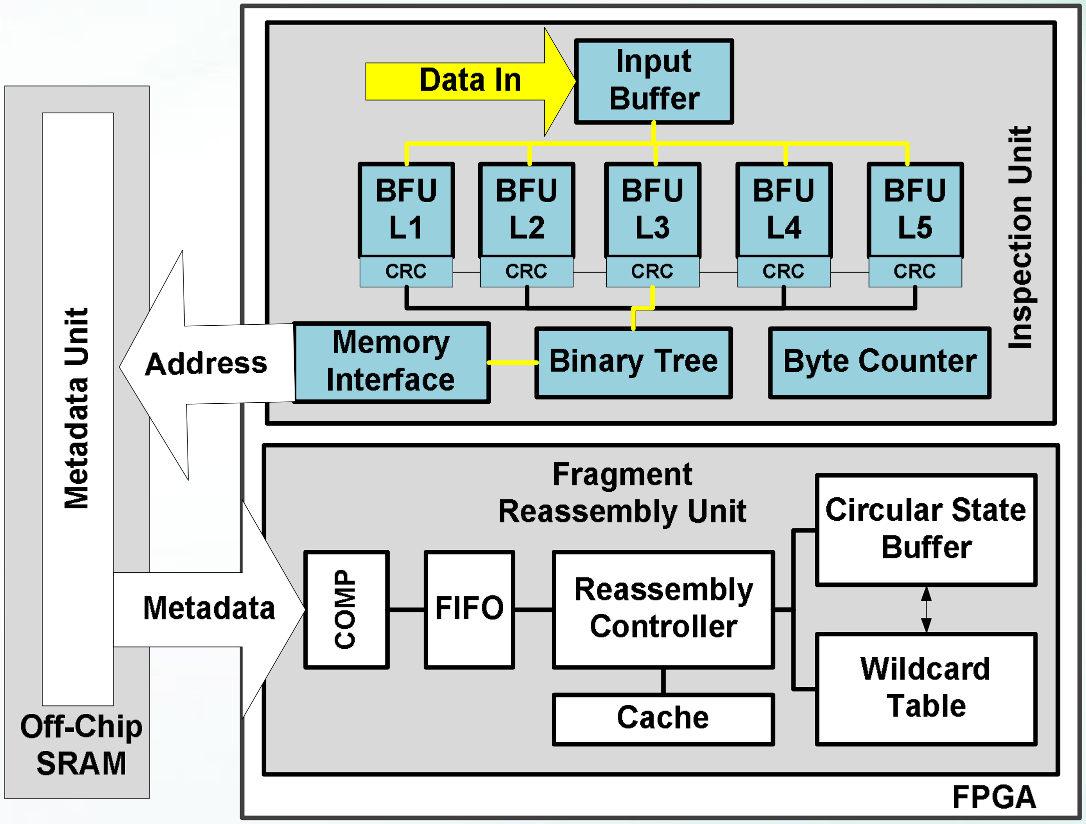
Hardware Architecture



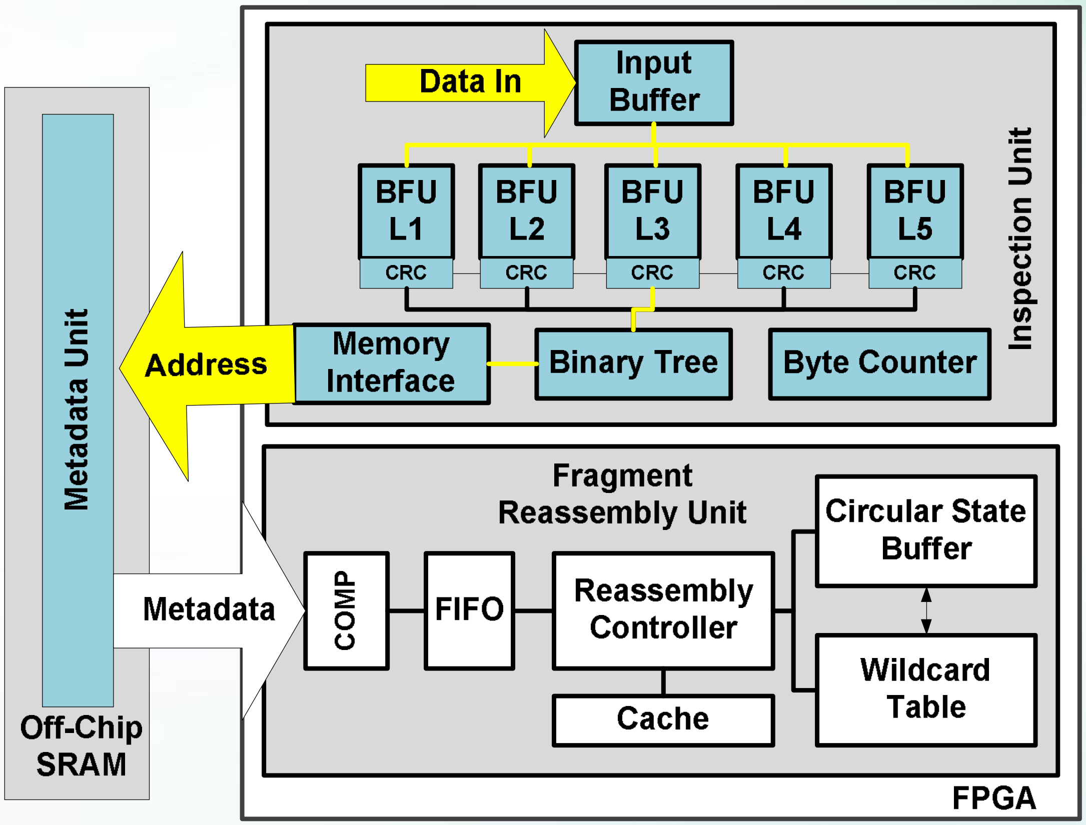
Hardware Architecture



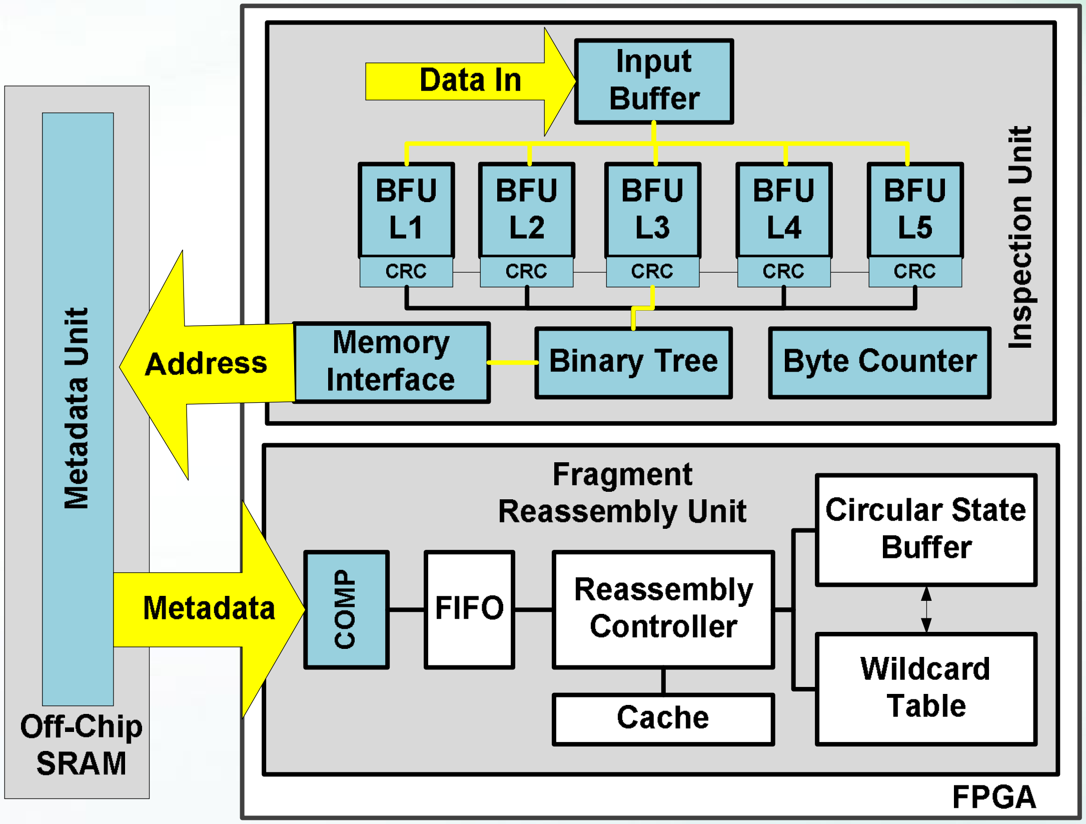
Hardware Architecture



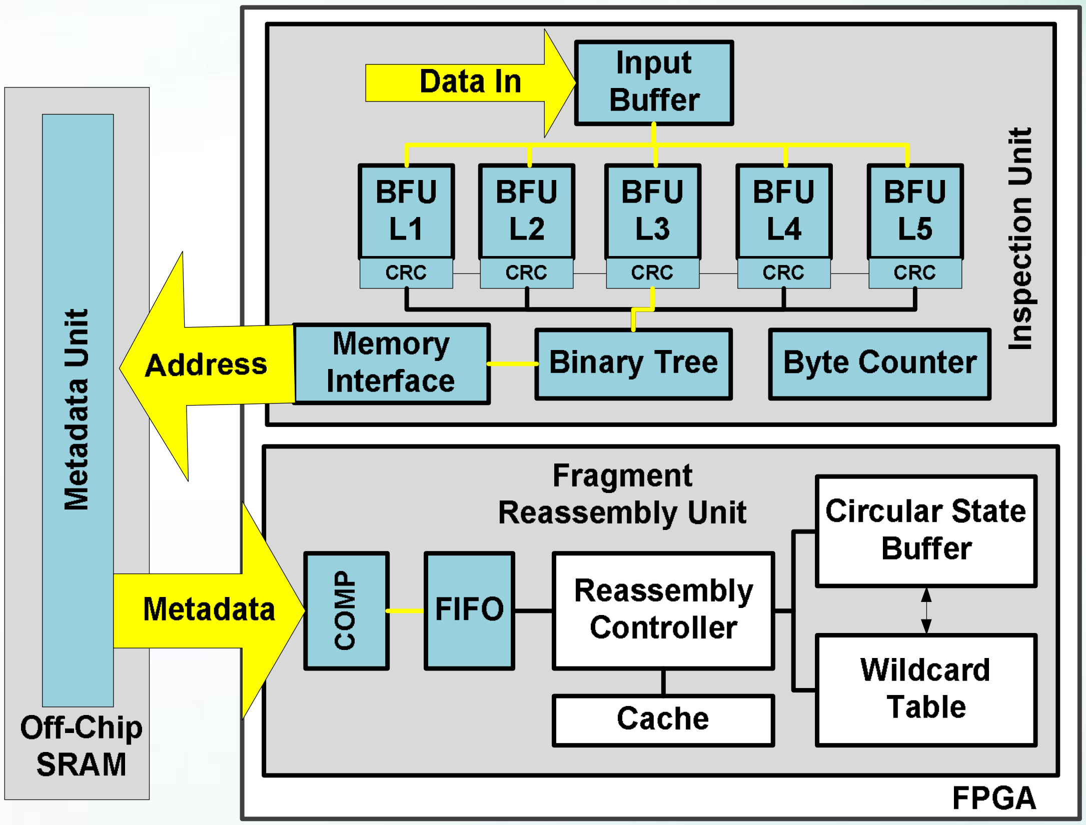
Hardware Architecture



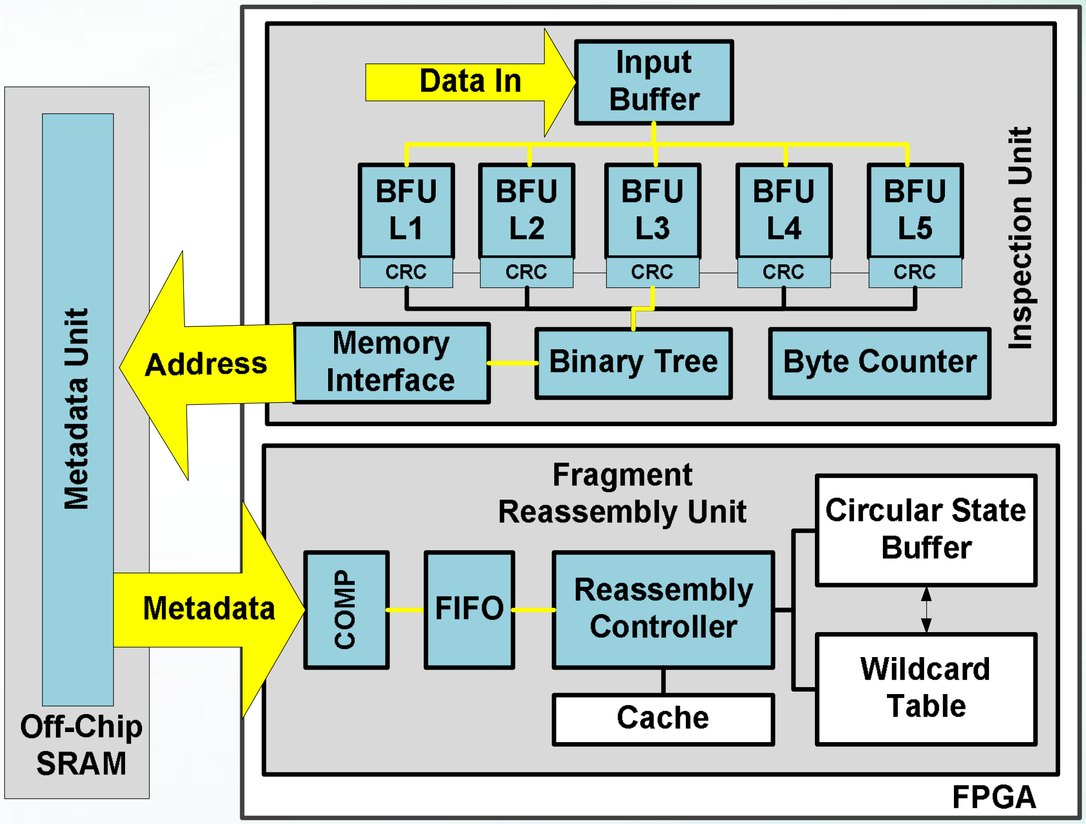
Hardware Architecture



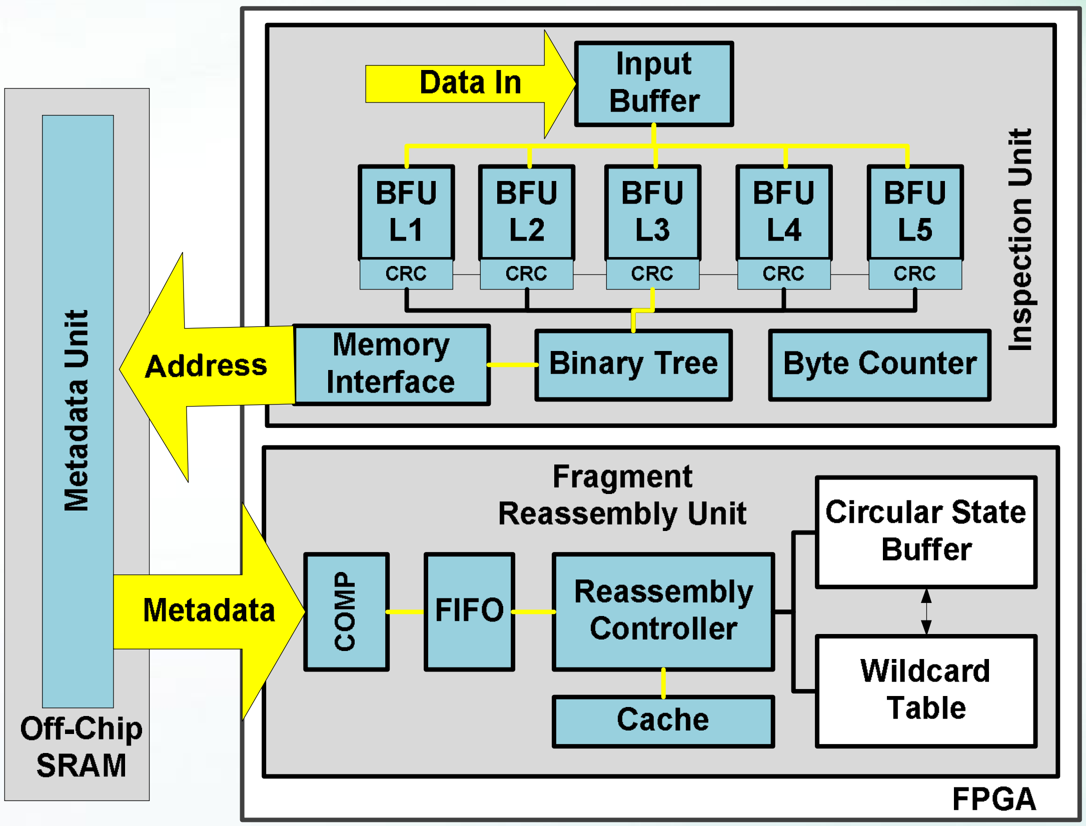
Hardware Architecture



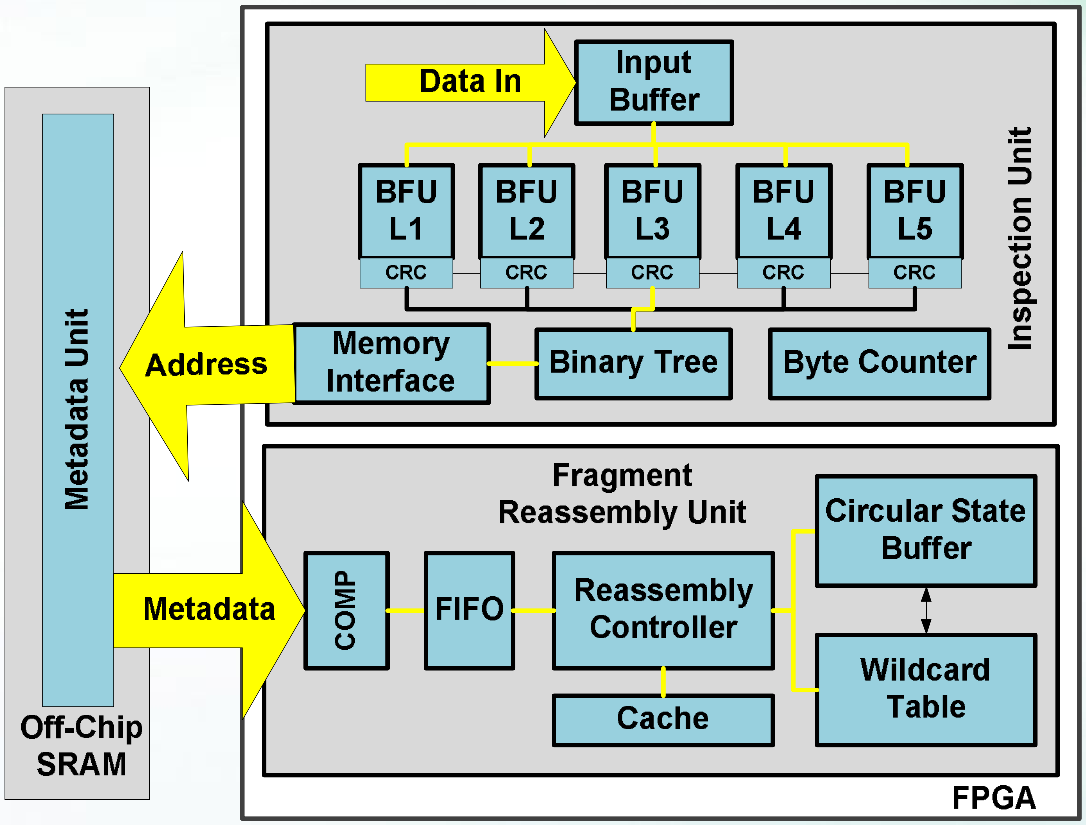
Hardware Architecture



Hardware Architecture



Hardware Architecture

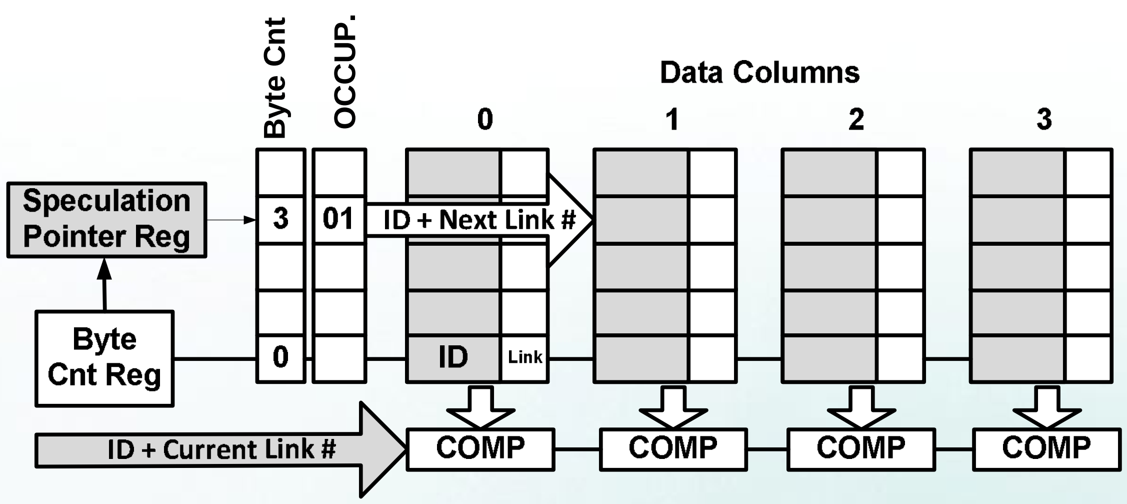


Circular Speculative Buffer

- Purpose
  - Detect patterns spanned across multiple segments and separated by fixed byte lengths
- Advantages
  - Support Multiple Traces
  - Guarantee No False Negative
  - Low Hardware Usage
    - Aliasing allows Design Trade-off between
    - Hardware and False Positive Probability

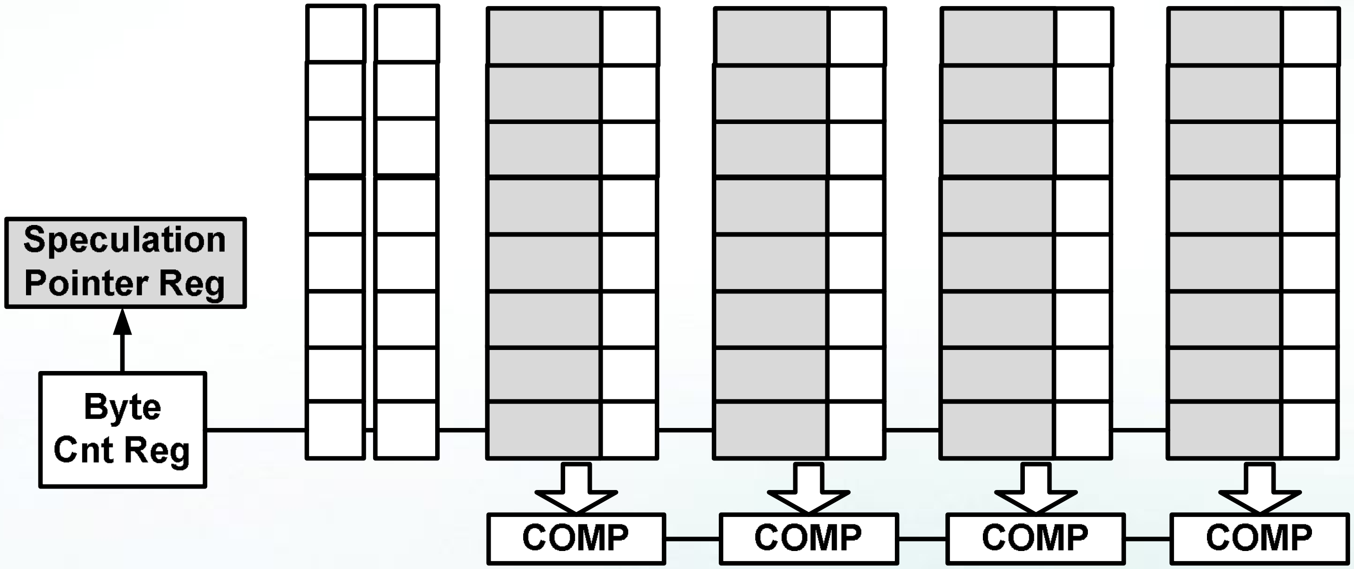
Circular Speculative Buffer

- Works like a time-wheel
- Operation is divided into *Verification* and *Speculation* phases
- Number of rows = Maximum displacement supported
  - Indexed by lower bits of Byte Count
- Three types of columns
  - Upper bits of Byte Count
  - Data (Rule ID + Link #)
  - Occupancy
- Reset upon a new file stream



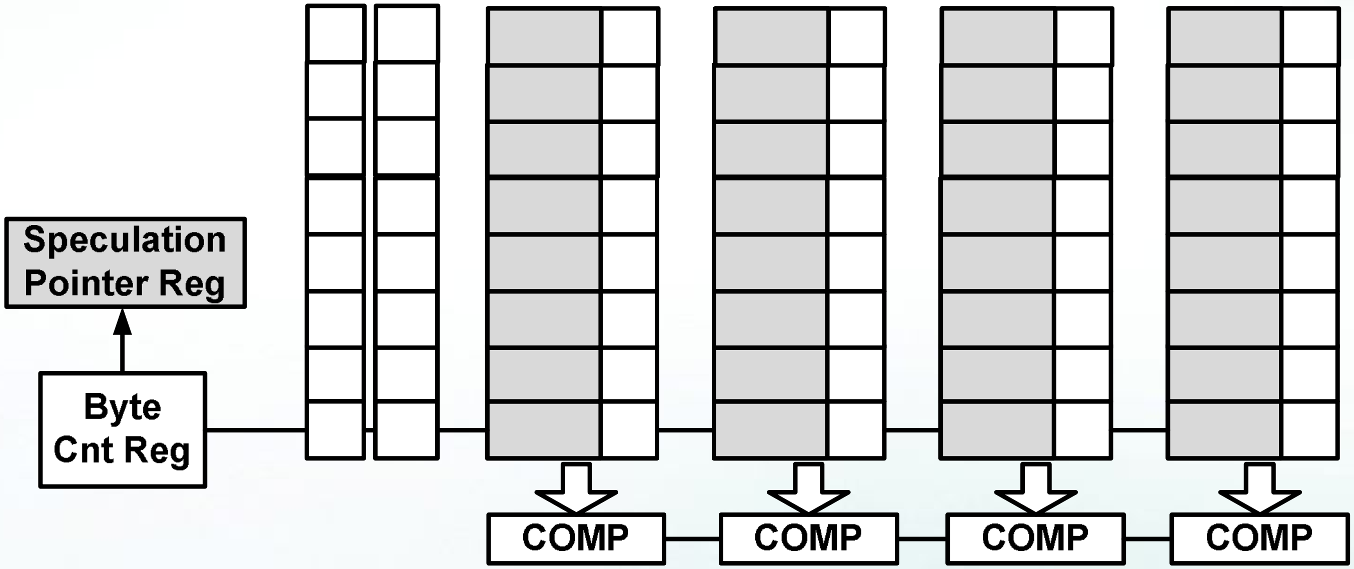
Circular Speculative Buffer

- Example
  - Pattern A: ABC{1} BCD{7}EFG
  - Input: ABCD1234EFG



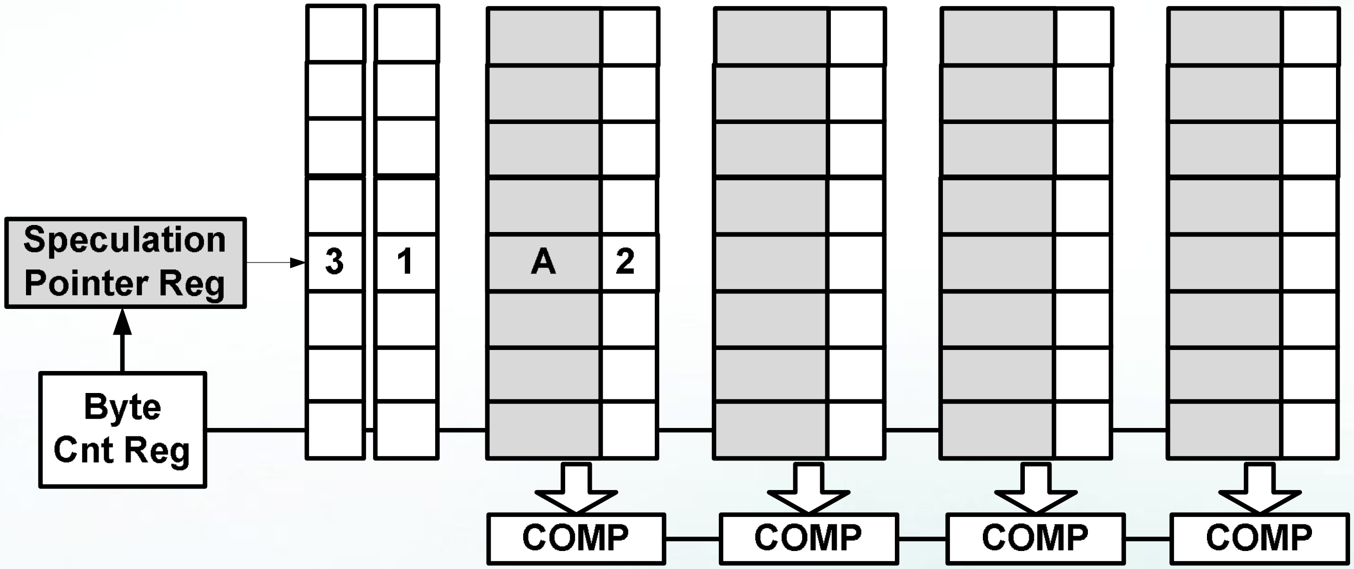
Circular Speculative Buffer

- At Byte Count = 2, C arrives and Segment ABC is detected
- Verification:
  - ABC is the first segment of Pattern A, so no previous state is needed to progress



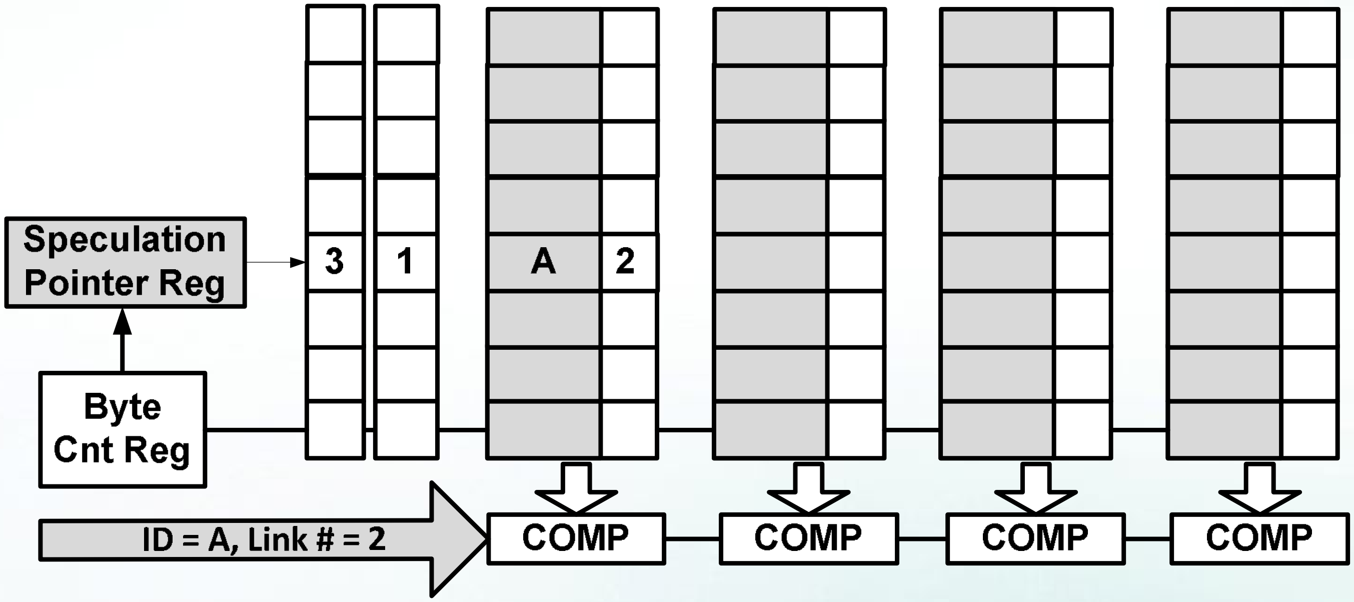
Circular Speculative Buffer

- Speculation:
  - Record the next segment (BCD) expect to follow ABC at the expected Byte Count location (**Speculation Pointer**)
    - Byte Count + Displacement =2+1= 3
- Increment **Occupy Column** pointed by Speculation Pointer by 1



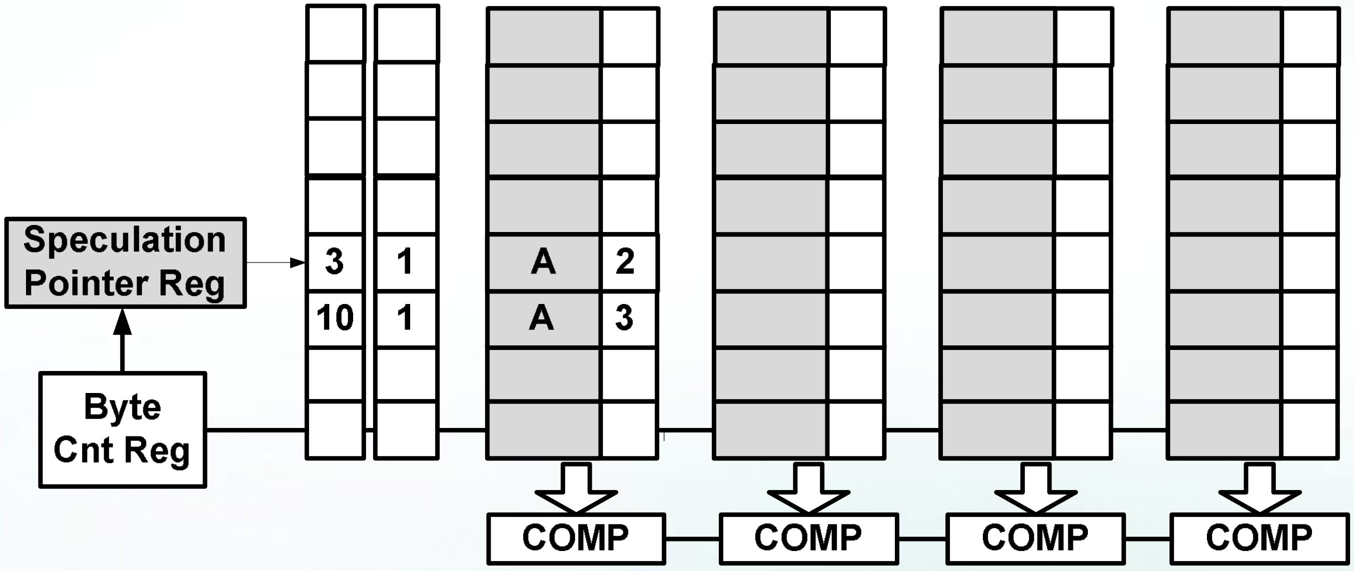
Circular Speculative Buffer

- At Byte Count = 3, D arrives and Segment BCD is detected
- Verification:
  - Is BCD expected by an ongoing trace at the current Byte Count row? Yes, as set previously by Segment ABC



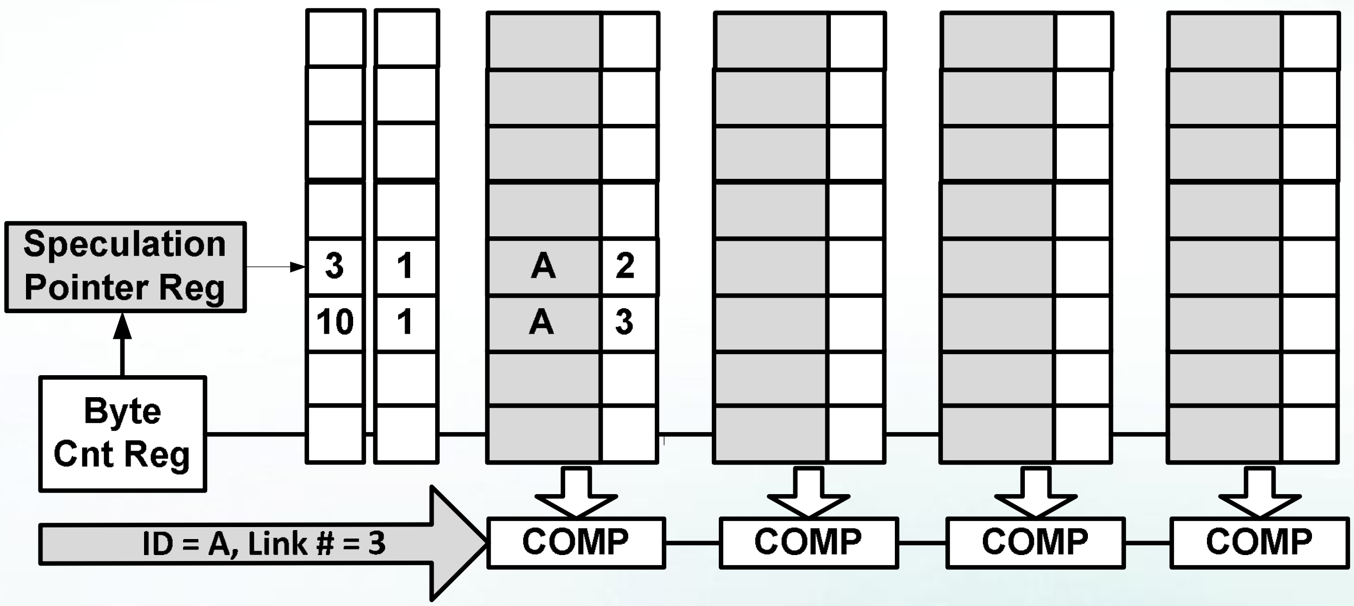
Circular Speculative Buffer

- Speculation:
  - Record the next segment (EFG) expect to follow BCD at the expected Byte Count location
  - Speculation Pointer = 3 + 7 =10
  - If Speculation Pointer > # of rows, the value wraps over
- Increment Occupy Column by 1



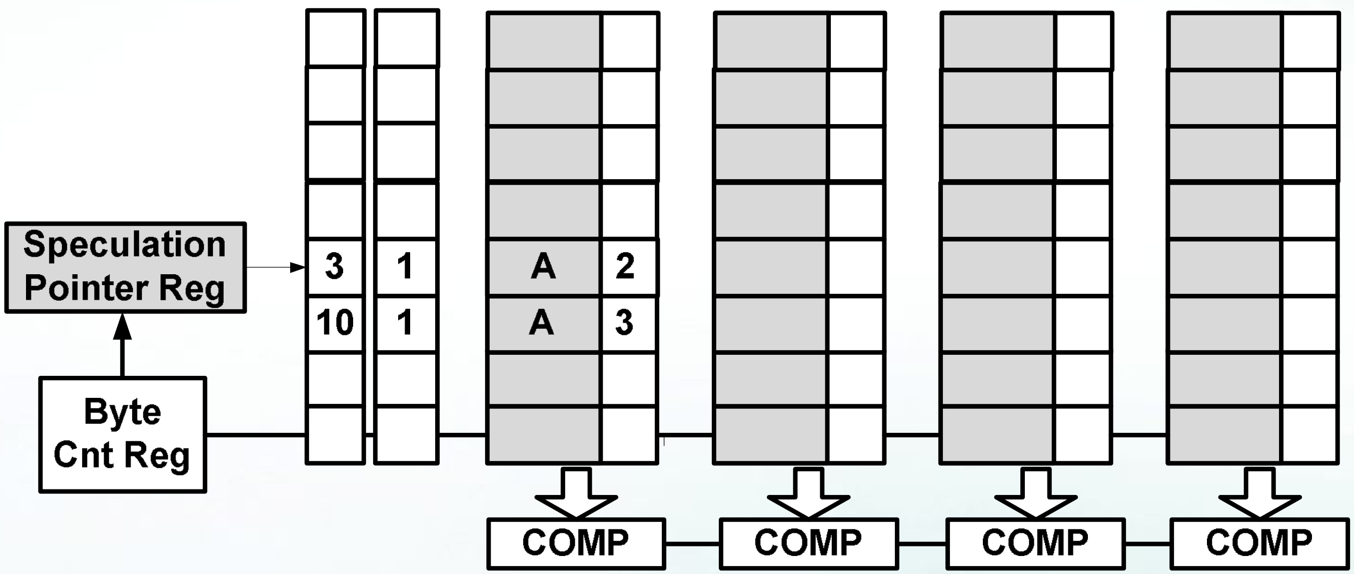
Circular Speculative Buffer

- At Byte Count = 10, G arrives and Segment EFG is detected
- Verification:
  - Returns true as EFG is indeed expected by an ongoing trace as set previously by Segment ABC



Circular Speculative Buffer

- Since EFG is the last segment of the pattern
  - The full pattern has been reconstructed from the input
  - Request for exact-matching is sent
- Trace of Pattern A remains in CSB until overwritten or reset when new file stream arrives



Wildcard Support

- Wildcards can be generated to two types
  - At-least wildcard
  - Within wildcard (Lossy)

Symbol	Original	After Conversion
??	Single-Byte Wildcard	Displacement
*	(Any-Number-of-Byte) Wildcard	At-Least Wildcard
{n-}	At-Least (n-Byte) Wildcard	At-Least Wildcard
{-N}	Within (n-Byte) Wildcard	Within Wildcard
{n-N}	Range wildcard	Within Wildcard

Wildcard Support

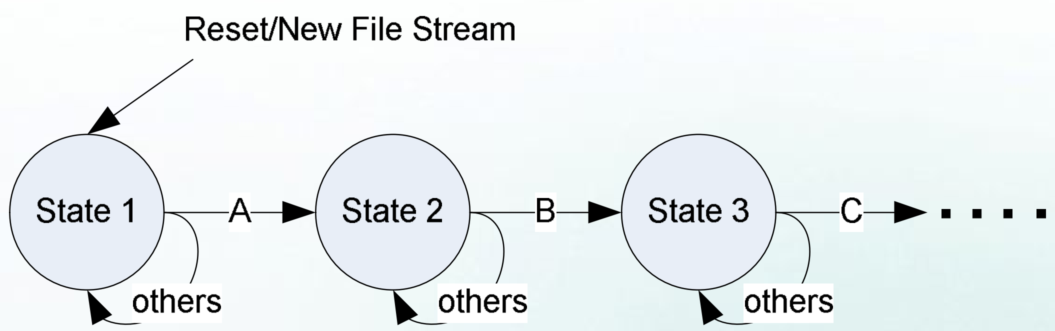
- Wildcard Table
  - Indexed directly by Rule ID of the pattern
  - Contains a Byte Range attribute in each entry to keep track of within/at-least conditions
  - State (progress of trace) is maintained through Link # similar to CSB
    - Reset at start of a new file stream

- Different traces of the same pattern is mapped to the same table entry to reduce resource usage

- Lossy but resource efficient
  - Increase false positive probability
  - Zero false negative probability

Wildcard Support

- At-least Wildcard
  - State only progress forwards (Link # only increases)
  - If state remains the same until the expected segment arrives after its Byte Range is satisfied
    - For an At-least Wildcard of *n* bytes ({n-}), once *n* bytes has passed in the file stream, the range condition is always satisfied



Wildcard Support

- Within Wildcard
  - State only progress forwards (Link # only increases)
  - If state remains the same until the expected segment arrives after its Byte Range is satisfied
    - For an At-least Wildcard of *n* bytes ({n-}), once *n* bytes has passed in the file stream, the range condition is always satisfied
  - Exception
    - If incoming segment contains the same Link # as the previous segment (which indicate it is followed by a Within Wildcard), Byte Range is refreshed (updated)

Wildcard Support

- Example
  - Pattern A: ABC**{3-}**DEF**{-7}**GHI **{-8}**JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL

Wildcard Support

- Example
  - Pattern A: ABC**{3-}**DEF**{-7}**GHI **{-8}**JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL
- Wildcard Table Entry After the first ABC has arrived at Byte Count =0

Link #	Byte Range	Wildcard Type
2	3	At-Least

Wildcard Support

- Example
  - Pattern A: ABC**{3-}**DEF**{-7}**GHI **{-8}**JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL
- Wildcard Table Entry After the first DEF has arrived at Byte Count =4

Link #	Byte Range	Wildcard Type
3	11	Within

Wildcard Support

- Example
  - Pattern A: ABC**{3-}**DEF**{-7}**GHI **{-8}**JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL
- Wildcard Table Entry After the first GHI has arrived at Byte Count =10

Link #	Byte Range	Wildcard Type
4	18	Within

Wildcard Support

- Example
  - Pattern A: ABC{3-}DEF{-7}GHI {-8}JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL
- Wildcard Table Entry After the first JKL has arrived at Byte Count =20
  - Byte Range condition is NOT satisfied; no action taken

Link #	Byte Range	Wildcard Type
4	18	Within

Wildcard Support

- Example
  - Pattern A: ABC{3-}DEF{-7}GHI {-8}JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL
- Wildcard Table Entry After the second DEF has arrived at Byte Count =23
  - Incoming Link # < Link # in Table Entry; no action taken

Link #	Byte Range	Wildcard Type
4	18	Within

Wildcard Support

- Example
  - Pattern A: ABC{3-}DEF{-7}GHI {-8}JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL
- Wildcard Table Entry After the second GHI has arrived at Byte Count =26
  - Incoming Link # < Link # in Table Entry, BUT
    - Wildcard Type = Within
    - Incoming Link # = Link # - 1
  - Updated Byte Range: 26+8 = 34

Link #	Byte Range	Wildcard Type
4	34	Within

Wildcard Support

- Example
  - Pattern A: ABC**{3-}**DEF**{-7}**GHI **{-8}**JKL
  - Input: ABC…DEF….GHI…JKL…DEF…GHI…JKL
- Wildcard Table Entry After the second JKL has arrived at Byte Count =30
  - Incoming Link # = Link # in Table Entry
  - Metadata indicates JKL is the final segment of the pattern
    - Request of exact-matching is sent
    - Wildcard Table entry unchanged

Link #	Byte Range	Wildcard Type
4	34	Within

Experimental Results

- Resource usage is determined by synthesizable Verilog model
- Performance is determined by cycle-accurate simulator written in C, normalized to the frequency reported by synthesis tool
  - SRAM is assumed to operate at ¼ of core frequency
- Based on ClamAV 0.93.1 main
  - # of patterns remained after special-case removal stage= 84,387
- Use Ubuntu-7.10-i386.iso sample input
  - Two tests: iso and extracted

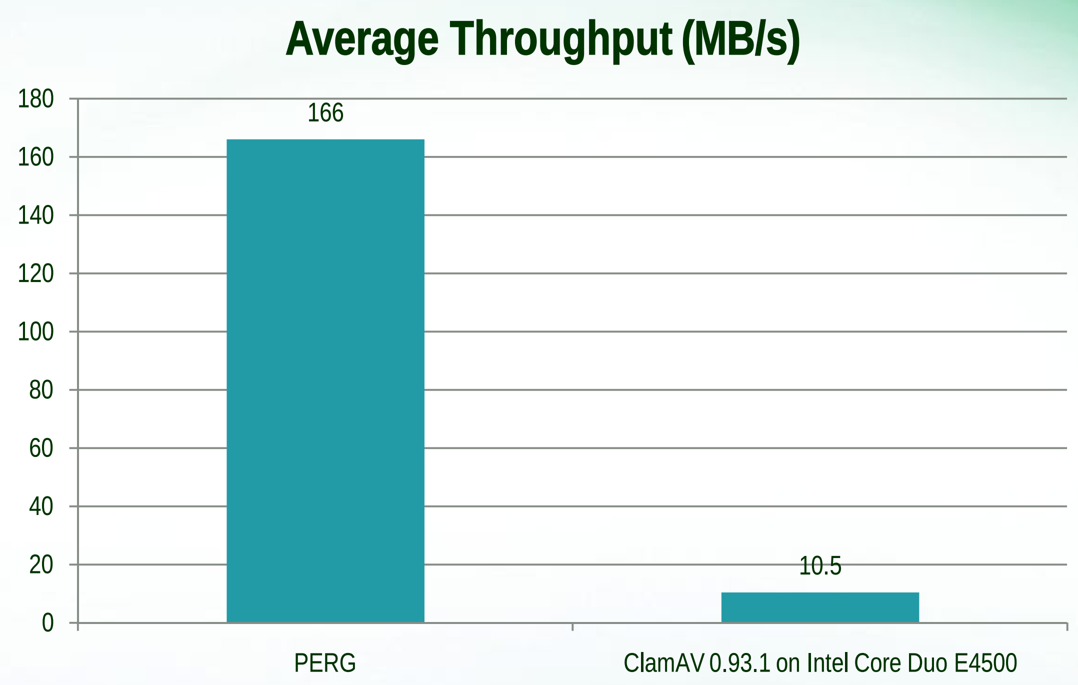
Performance

	Single File (Ubuntu7_10_x86.iso)	Extracted Files (274)
# of Bytes Scanned	729,608,192	727,677,929
# of False Positives	4	4
False Positive Probability for Each Byte Scanned	0.0000005%	0.0000005%
# of Off-chip Memory Requests	82,499,591	80,500,329
Probability of Off-chip Memory Request for Each Byte Scanned	11.31%	11.07%
Off-chip Memory Throughput	19.4 MB/s	19.0 MB/s
Average Throughput	166 MB/s (0.922 B/cycle)	168 MB/s (0.933 B/cycle)
Modeled Frequency	180 MHz	180 MHz

Comparison

System	Patterns mapped	LC per Pattern	Memory per Pattern (kb/pattern)	TLP	TMP	Throughput (Gbps)
PERG	**84,387**	**0.5073**	**0.0358**	2.56	**36.28**	1.3
Cuckoo Hashing	5,026	0.5933	0.2220	**3.84**	10.27	2.28
HashMem	1,474	1.7436	0.4410	1.55	6.12	**2.70**
PH-Mem	2,200	2.8509	0.1309	0.74	16.12	2.11
ROM+Coproc	2,031	4.1753	0.1359	0.50	15.31	2.08

Comparison



Average Throughput (MB/s)
180
160
140
120
100
80
60
40
20
0
166
10.5
PERG
ClamAV 0.93.1 on Intel Core Duo E4500

Effectiveness of Filter Consolidation

	Without Filter Consolidation	With Filter Consolidation
Total # of Segments Mapped to BFUs	89,423	141,147
Total # of BFUs	220	26
Total # of BRAMs used by BFUs	256	168
# of Cache Entries	132	3823

Scalability and Dynamic Updatability

Number of BFUs	16	16	16	16	16
Total number of patterns	1440000	1440000	1440000	1440000	1440000
Utilization	90%	90%	90%	90%	90%
Change %	10%	25%	50%	75%	100%
Average number of rehashes	13.98	11.36	15	16.56	15.72
Number of setup failures (out of 50)	32	36	31	30	29

Scalability and Dynamic Updatability

Number of BFUs	16	16	16	16	16
Total number of patterns	80000	96000	112000	128000	144000
Utilization	50%	60%	70%	80%	90%
Average number of patterns inserted	37466	27774	17892	3892	48
Average number of insertions until failure	749.32	555.48	357.84	77.84	0.96
% of theoretical max reached	73.41625	77.35875	81.1825	82.4325	90.03

Conclusions

- PERG excels in pattern-per-resource density
- Lags behind in throughput
  - Still significantly faster than software
- Bloomier filters, checksum, and FRU together ensure false positives stay low despite lossy wildcard support
- A highly-utilized BFU is desirable
  - Filter consolidation is necessary
- To allow dynamic update, hash function must become more programmable

Future Works

- Support for interleaving file stream
- Integration with antivirus software
- Alternative database
- Update and Expansion Option
- Eliminate special-case removal stage

Contributions

- **A Novel Hardware Architecture**
  - Handle pattern matching in a multi-staged manner without resorting to high-bandwidth off-chip memory requirement
- **A Novel Filter Consolidation Algorithm**
  - Reduce the hardware resources required by packing filter units into high capacity, thus reducing the number of filter units needed.
- **Circular State Buffer**
  - Support multiple traces of multi-segmented patterns with zero false negative probability
- **Limited Regular Expression Support**
  - Support for wildcard operators to detect polymorphic virus