CAST

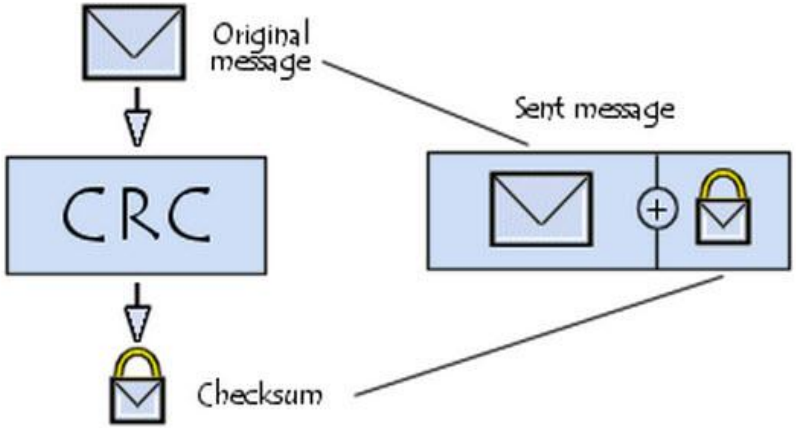
CRC Detection and Optimization

Mariam Arutunian <mariamarutunian@gmail.com>
Hayk Aslanyan <hayk.aslanian@gmail.com>

FOSDEM 2025
Belgium, Brussels

CRC

- CRC (Cyclic Redundancy Check) is used to detect errors in digital data transmission.
- CRCs generate a checksum based on the input data.



CRC implementation

There are various methods of CRC implementation in software:

- Bitwise method
- Carryless multiplication-based method
- Table-based method

Examples

Many real projects use bitwise CRC implementations.

```
uint8 t  crc_ibutton_update (uint8_t crc, uint8_t data) {
  uint8 t i;
  crc = crc ^ data;
  for (i = 0; i < 8; i++) {
    if (crc & 0x01)
      crc = (crc >> 1) ^ 0x8C;
    else
      crc >>= 1;
  }
  return crc;
}
```

```
__u32 nvme_mi_crc32_update(__u32 crc, void
*data, size_t len)
{
   int i;
   while (len--) {
       crc ^= *(unsigned char *)(data++);
       for (i = 0; i < 8; i++)
           crc = (crc >> 1) ^ ((crc & 1) ?
0x82F63B78 : 0);
   }
   return crc;
```

```
uint8_t crc8 (uint8_t data, uint8_t crc) {
  crc ^= data;
  for (int i = 0; i < 8; ++i) {
      uint8_t carry = crc & 0x80;
      crc <<= 1;
      if (carry)
       crc ^= 0x21;
    }
  return crc;
}
```

```
Guint32 crc32 (guchar * message) {
 int i, j;
 guint32 byte, crc;
 i = 0;
 crc = 0xFFFFFFFF;
 while (message[i] != 0)
   {
     byte = message[i];
     byte = bit_reverse (byte);
     for (j = 0; j <= 7; j++)
      {
        if ((int) (crc ^ byte) < 0)
          crc = (crc << 1) ^ 0x04C11DB7;
        else
          crc = crc << 1;
        byte = byte << 1;
      }
     i = i + 1;
   }
 return bit_reverse (~crc);
}
```

Problem statement

A. Detect bitwise CRC implementations in projects
B. Replace the detected CRC implementations with a faster CRC code:
    - using CRC instruction
    - using CLMUL instruction
    - using lookup table

A. Bitwise CRC detection

The algorithm consists of 2 phases:

1. **Initial checks** for potential CRC calculation detection.
2. **Verification** of detected potential CRC using symbolic execution.

1. Initial checks
(CRC detection)

1. Initial checks

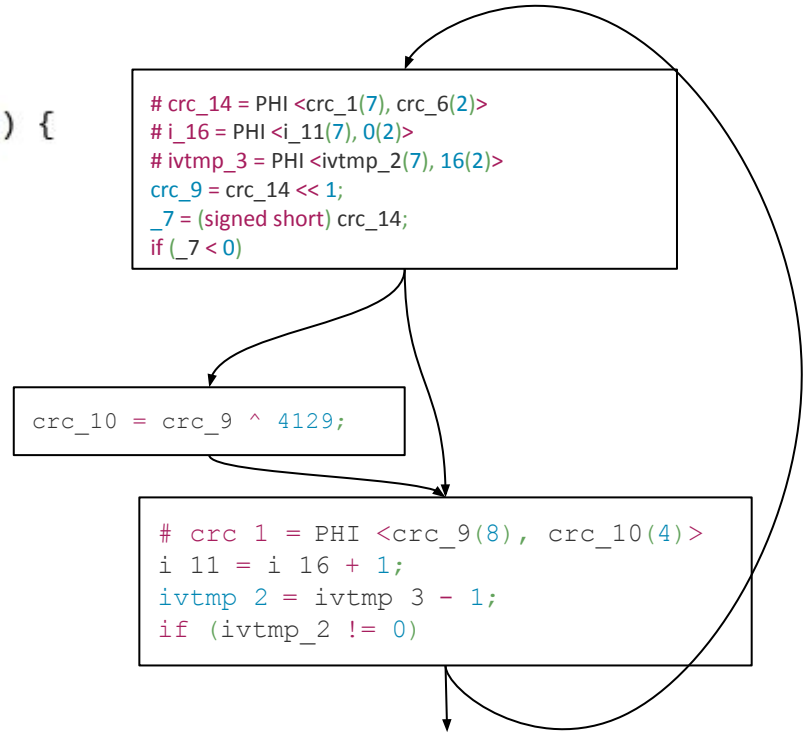
Detect a loop that meets the following criteria:

- Iteration number is 8, 16, 32, 64
- Is the innermost loop
- Contains a conditional operator (if)
    - If body contains an XOR operation
- Contains a shift operation

```
uint16_t crc16 (uint16_t data, uint16_t crc) {
    int carry;
    crc ^= data;
    for (int i = 0; i < 16; ++i) {
        carry = crc & 0x8000;
        crc <<= 1;
        if (carry) crc ^= 0x1021;
    }
    return crc;
}
```

GIMPLE

```
uint16_t crc16 (uint16_t data, uint16_t crc) {
    int carry;
    crc ^= data;
    for (int i = 0; i < 16; ++i) {
        carry = crc & 0x8000;
        crc <<= 1;
        if (carry) crc ^= 0x1021;
    }
    return crc;
}
```



2. Verification
(CRC detection)

2. Verification

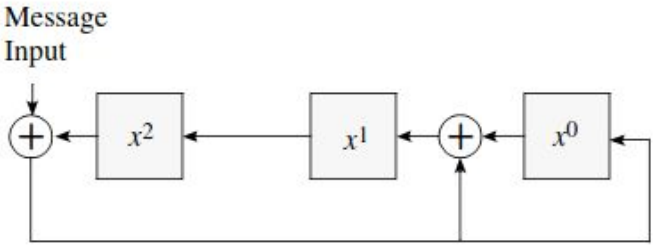
- Create LFSR for the polynomial
- Symbolically execute the loop
- Check that the resulting CRC values match the created LFSR after each iteration

CRC Polynomial

```
uint16_t crc16 (uint16_t data, uint16_t crc) {
    int carry;
    crc ^= data;
    for (int i = 0; i < 16; ++i) {
        carry = crc & 0x8000;
        crc <<= 1;
        if (carry) crc ^= 0x1021;
    }
    return crc;
}
```

- CRC is the remainder of a modulo-2 polynomial division of the data
  - $x^{16} + x^{12} + x^5 + 1$
- Polynomials are represented as binary numbers
  - $0x11021_{16} = (1\ 0001\ 0000\ 0010\ 0001)_2$
  - $0x\ 1021_{16} = (0001\ 0000\ 0010\ 0001)_2$

CRC calculation using LFSR



CRC using LFSR with $p(x) = x^3 + x + 1$ polynomial

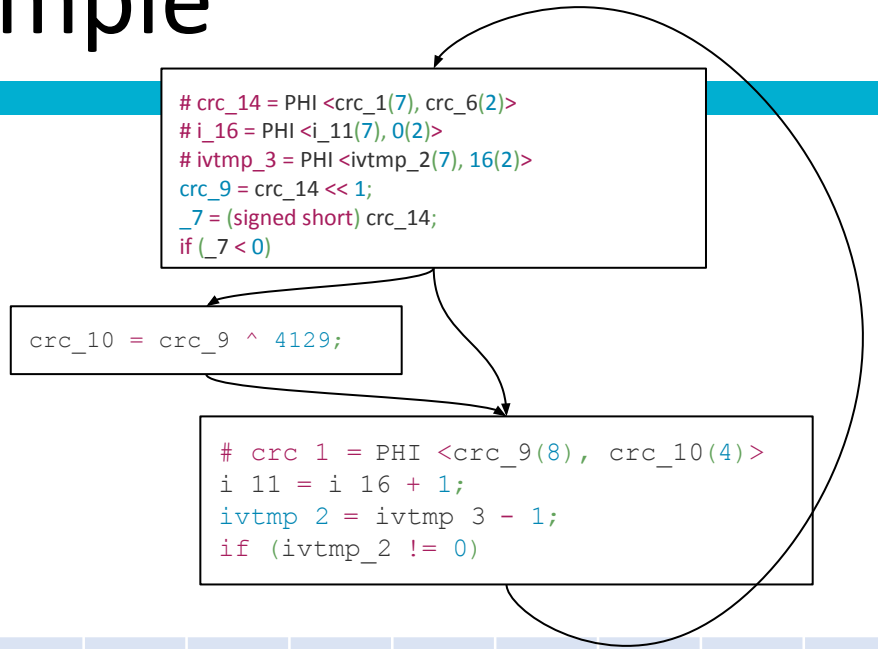
- LFSR stands for Linear Feedback Shift Register.
- LFSR is a shift register where the input bit depends on a linear function of its previous states.
- LFSR can be used to calculate CRC.

LFSR creation

- Execute the loop with specific values to extract the polynomial
- Create an LFSR using extracted information about the CRC, size, shift direction and the polynomial

LFSR creation example

Shift direction - left
Output crc - crc_1 (16_bit)



Polynomial (16_bit)

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

LFSR (16_bit)

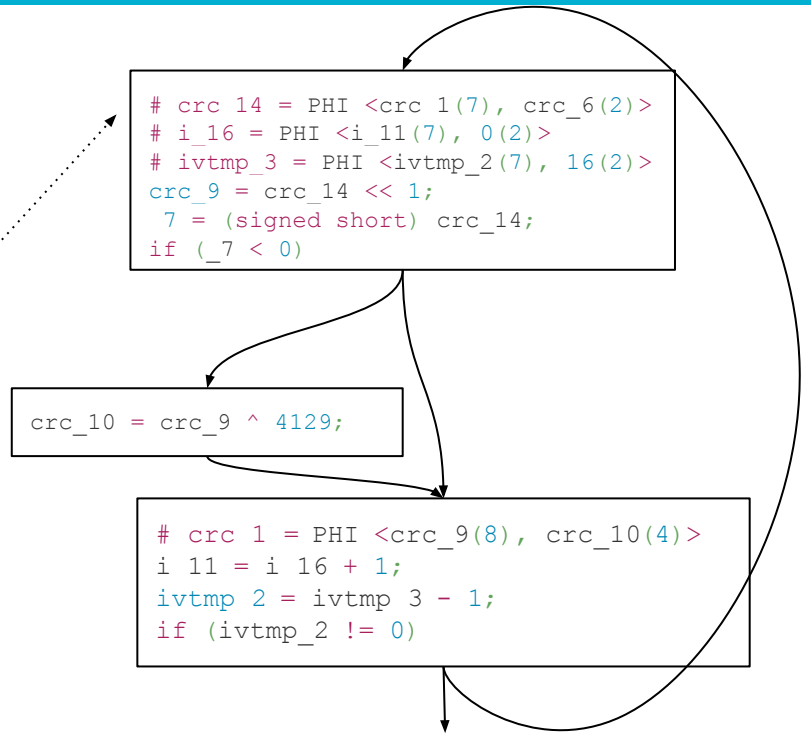
crc[14]	crc[13]	crc[12]	crc[11]^crc[15]	crc[10]	…	crc[4]^crc[15]	…	crc[0]	crc[15]

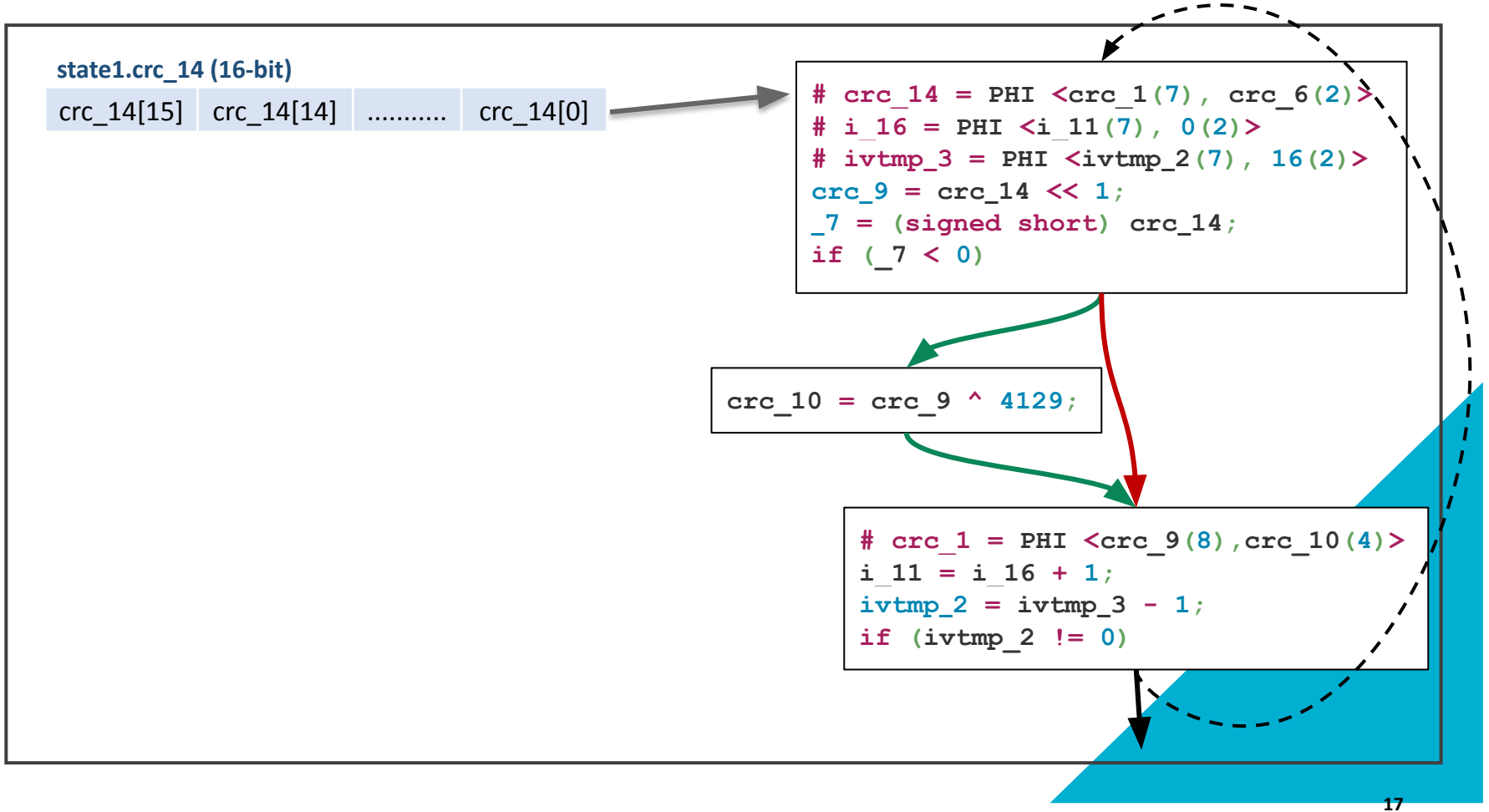
Loop execution

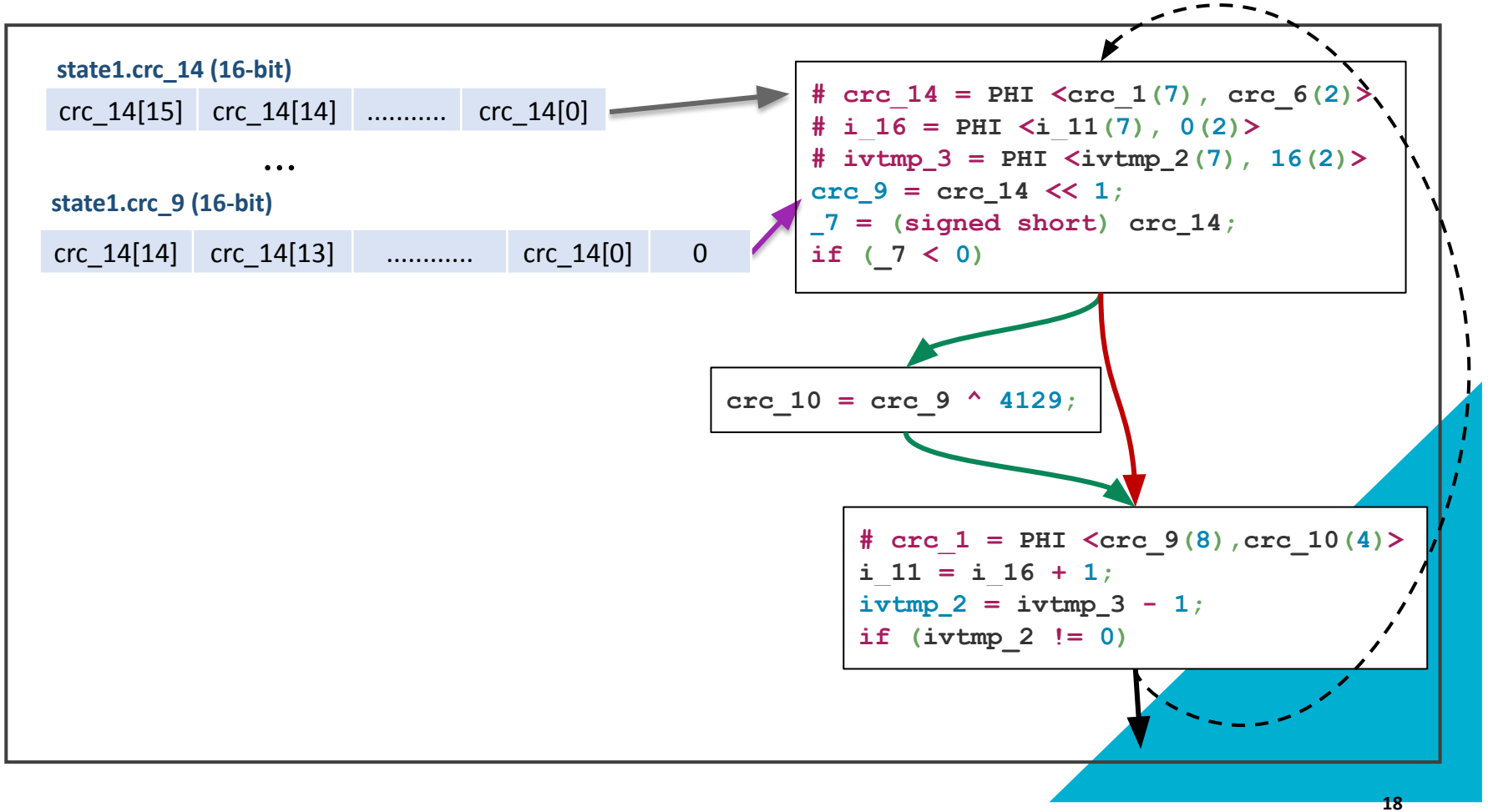
- Assign symbolic values to uninitialized loop header variables

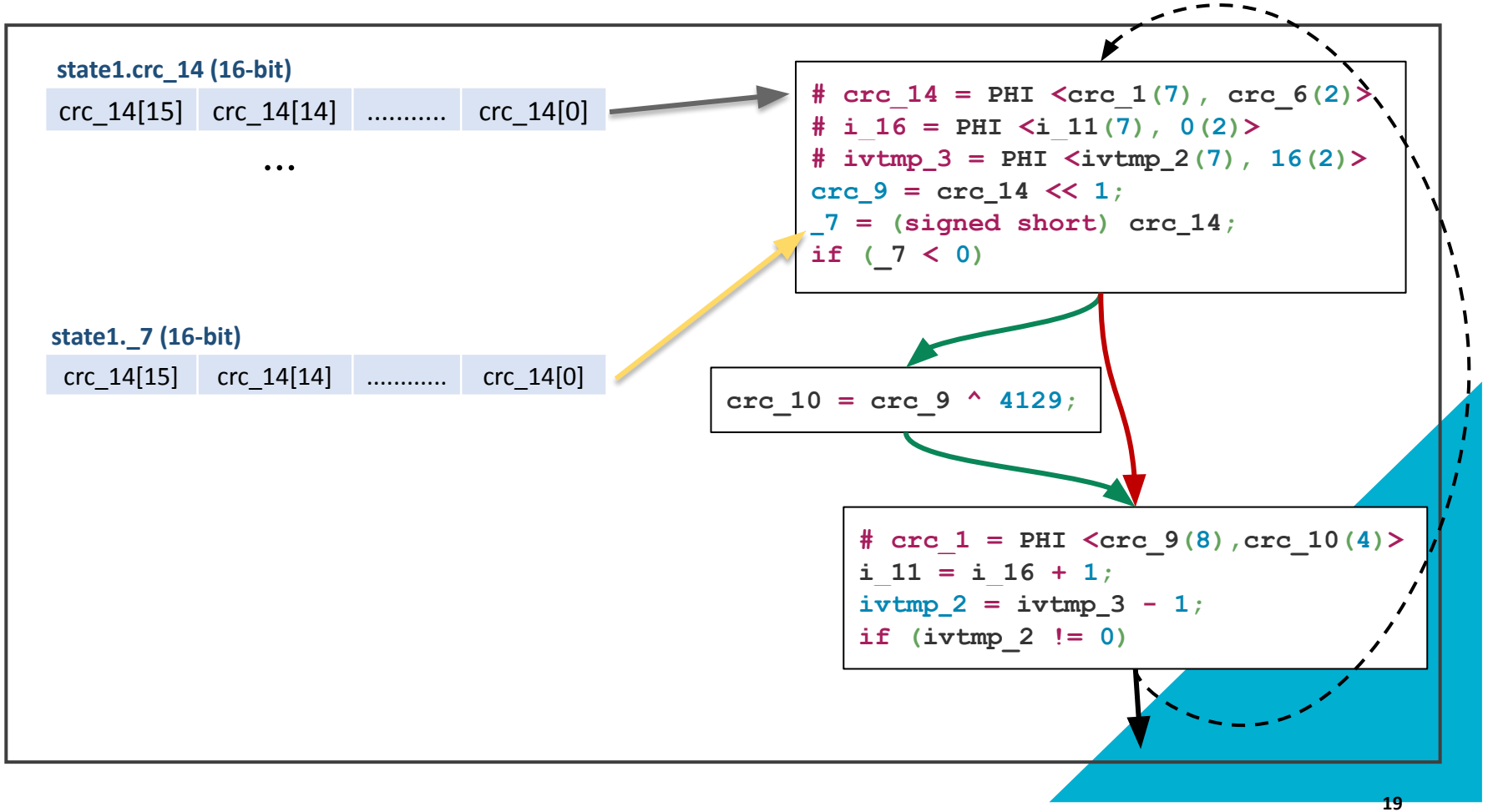
crc_14[15]	crc_14[14]	.....	crc_14[0]

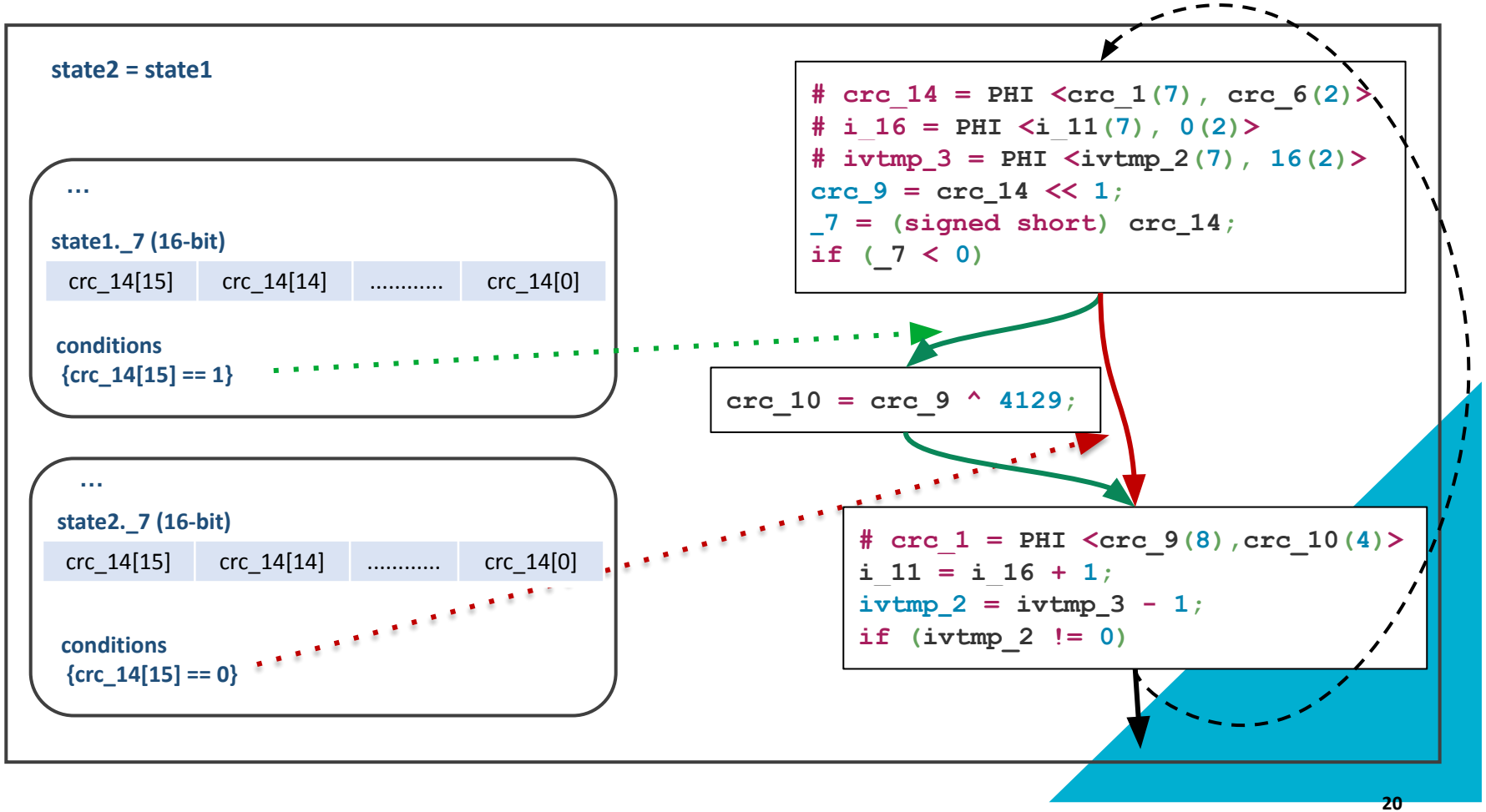
- Execute the loop on a bit level
- Track the state of each path

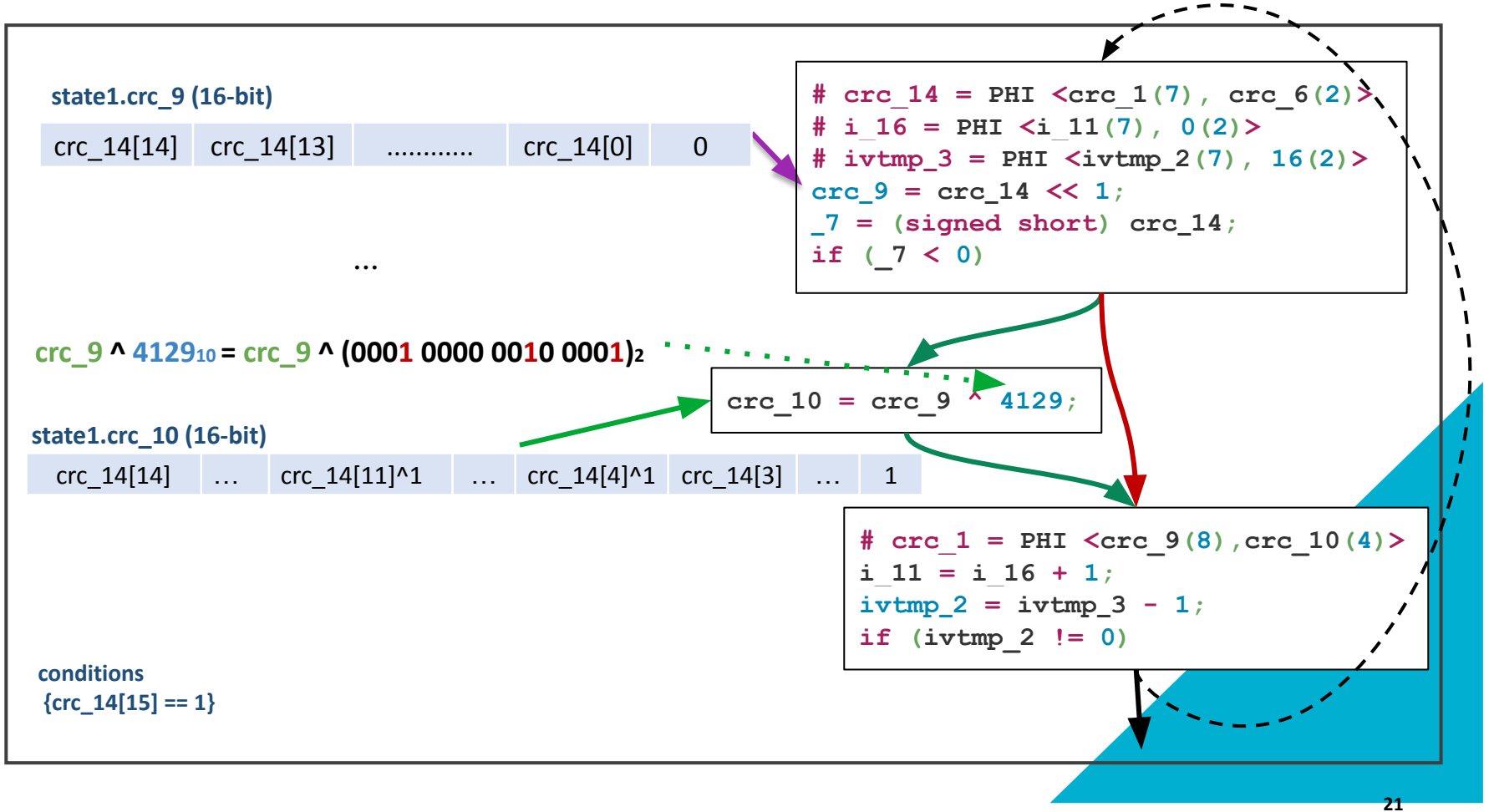


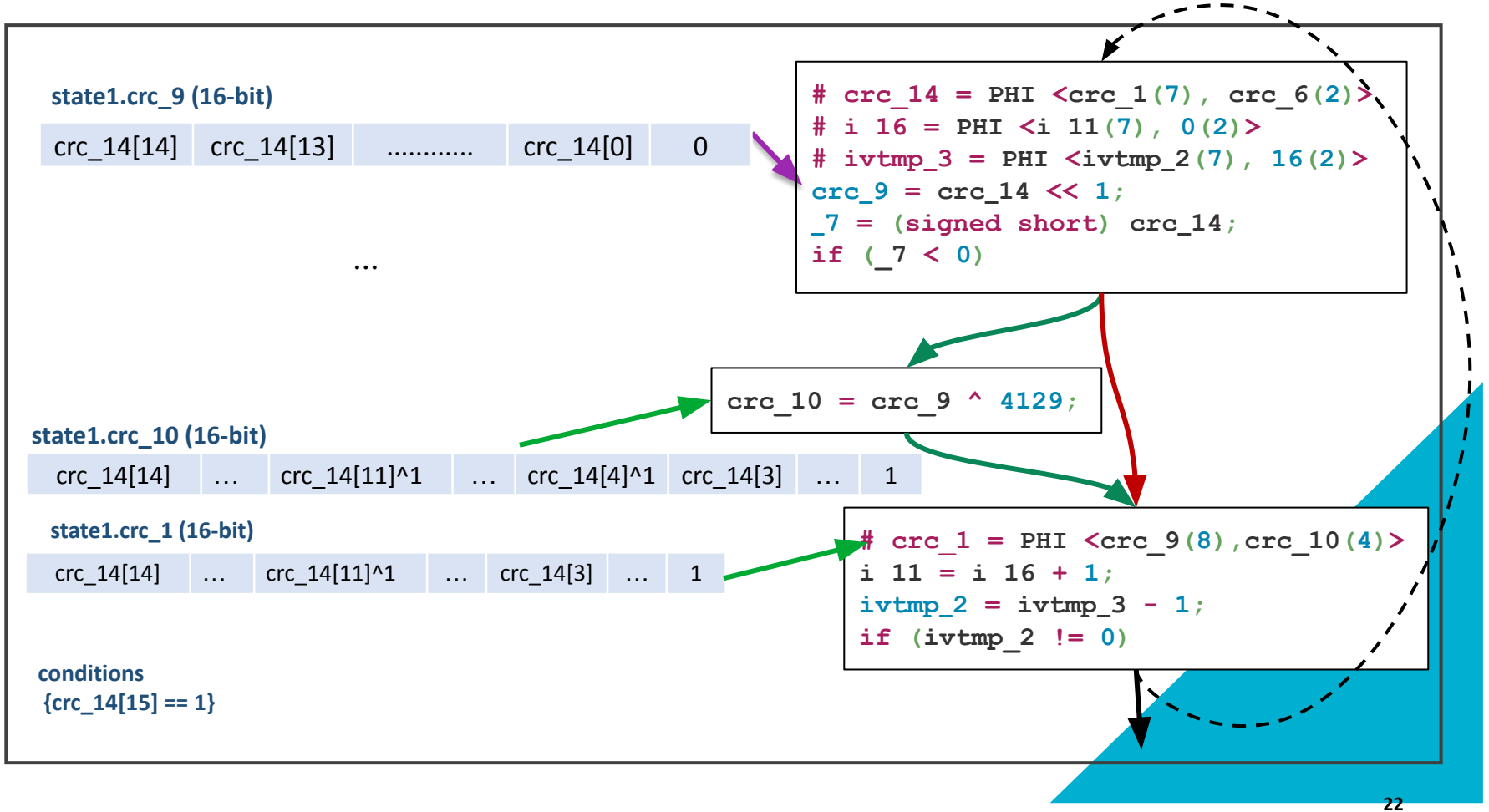

state1.crc_14 (16-bit)
crc_14[15]
crc_14[14]
.....
crc_14[0]
# crc_14 = PHI <crc_1(7), crc_6(2)>
# i_16 = PHI <i_11(7), 0(2)>
# ivtmp_3 = PHI <ivtmp_2(7), 16(2)>
crc_9 = crc_14 << 1;
_7 = (signed short) crc_14;
if (_7 < 0)
crc_10 = crc_9 ^ 4129;
# crc_1 = PHI <crc_9(8),crc_10(4)>
i_11 = i_16 + 1;
ivtmp_2 = ivtmp_3 - 1;
if (ivtmp_2 != 0)


state1.crc_14 (16-bit)
crc_14[15]
crc_14[14]
.....
crc_14[0]
...
state1.crc_9 (16-bit)
crc_14[14]
crc_14[13]
.....
crc_14[0]
0
# crc_14 = PHI <crc_1(7), crc_6(2)>
# i_16 = PHI <i_11(7), 0(2)>
# ivtmp_3 = PHI <ivtmp_2(7), 16(2)>
crc_9 = crc_14 << 1;
_7 = (signed short) crc_14;
if (_7 < 0)
crc_10 = crc_9 ^ 4129;
# crc_1 = PHI <crc_9(8),crc_10(4)>
i_11 = i_16 + 1;
ivtmp_2 = ivtmp_3 - 1;
if (ivtmp_2 != 0)


state1.crc_14 (16-bit)
crc_14[15]
crc_14[14]
.....
crc_14[0]
...
state1._7 (16-bit)
crc_14[15]
crc_14[14]
.....
crc_14[0]
# crc_14 = PHI <crc_1(7), crc_6(2)>
# i_16 = PHI <i_11(7), 0(2)>
# ivtmp_3 = PHI <ivtmp_2(7), 16(2)>
crc_9 = crc_14 << 1;
_7 = (signed short) crc_14;
if (_7 < 0)
crc_10 = crc_9 ^ 4129;
# crc_1 = PHI <crc_9(8),crc_10(4)>
i_11 = i_16 + 1;
ivtmp_2 = ivtmp_3 - 1;
if (ivtmp_2 != 0)


state2 = state1
...
state1._7 (16-bit)
crc_14[15]
crc_14[14]
.....
crc_14[0]
conditions
{crc_14[15] == 1}
...
state2._7 (16-bit)
crc_14[15]
crc_14[14]
.....
crc_14[0]
conditions
{crc_14[15] == 0}
# crc_14 = PHI <crc_1(7), crc_6(2)>
# i_16 = PHI <i_11(7), 0(2)>
# ivtmp_3 = PHI <ivtmp_2(7), 16(2)>
crc_9 = crc_14 << 1;
_7 = (signed short) crc_14;
if (_7 < 0)
crc_10 = crc_9 ^ 4129;
# crc_1 = PHI <crc_9(8),crc_10(4)>
i_11 = i_16 + 1;
ivtmp_2 = ivtmp_3 - 1;
if (ivtmp_2 != 0)

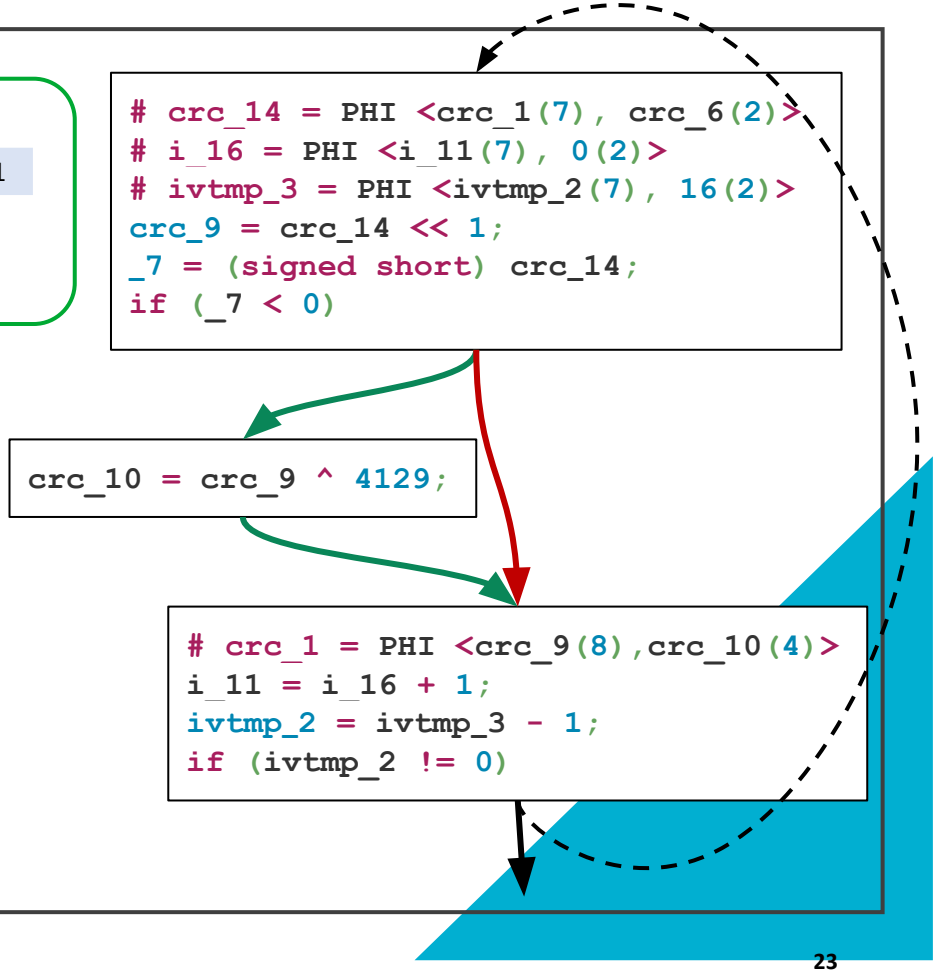

state1.crc_9 (16-bit)
crc_14[14]
crc_14[13]
.....
crc_14[0]
0
...
# crc_14 = PHI <crc_1(7), crc_6(2)>
# i_16 = PHI <i_11(7), 0(2)>
# ivtmp_3 = PHI <ivtmp_2(7), 16(2)>
crc_9 = crc_14 << 1;
_7 = (signed short) crc_14;
if (_7 < 0)
crc_9 ^ 4129₁₀ = crc_9 ^ (0001 0000 0010 0001)₂
crc_10 = crc_9 ^ 4129;
state1.crc_10 (16-bit)
crc_14[14]
…
crc_14[11]^1
…
crc_14[4]^1
crc_14[3]
…
1
# crc_1 = PHI <crc_9(8),crc_10(4)>
i_11 = i_16 + 1;
ivtmp_2 = ivtmp_3 - 1;
if (ivtmp_2 != 0)
conditions
{crc_14[15] == 1}

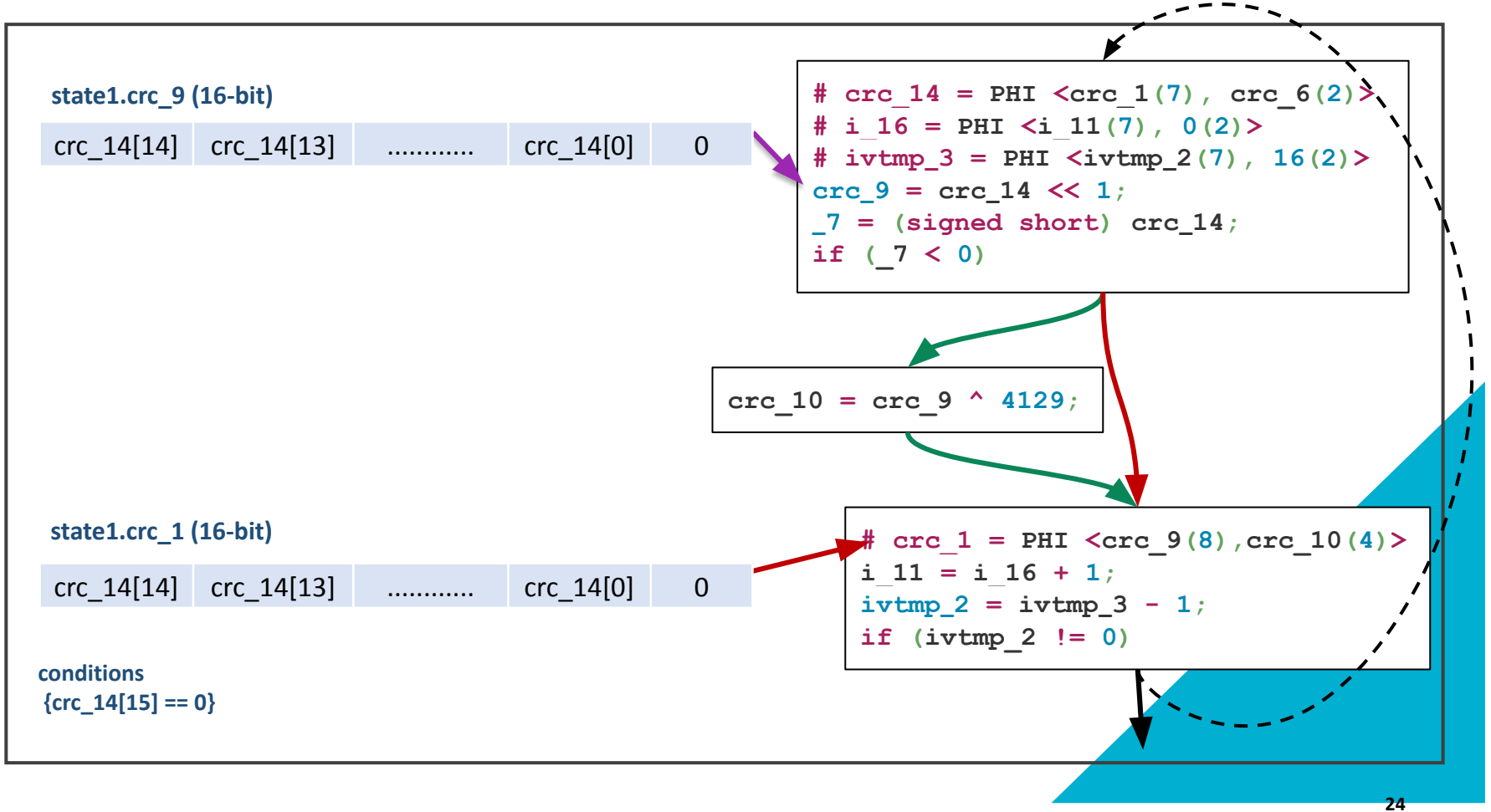

state1.crc_9 (16-bit)
crc_14[14]
crc_14[13]
.....
crc_14[0]
0
...
# crc_14 = PHI <crc_1(7), crc_6(2)>
# i_16 = PHI <i_11(7), 0(2)>
# ivtmp_3 = PHI <ivtmp_2(7), 16(2)>
crc_9 = crc_14 << 1;
_7 = (signed short) crc_14;
if (_7 < 0)
crc_10 = crc_9 ^ 4129;
state1.crc_10 (16-bit)
crc_14[14]
…
crc_14[11]^1
…
crc_14[4]^1
crc_14[3]
…
1
state1.crc_1 (16-bit)
crc_14[14]
…
crc_14[11]^1
…
crc_14[3]
…
1
# crc_1 = PHI <crc_9(8),crc_10(4)>
i_11 = i_16 + 1;
ivtmp_2 = ivtmp_3 - 1;
if (ivtmp_2 != 0)
conditions
{crc_14[15] == 1}

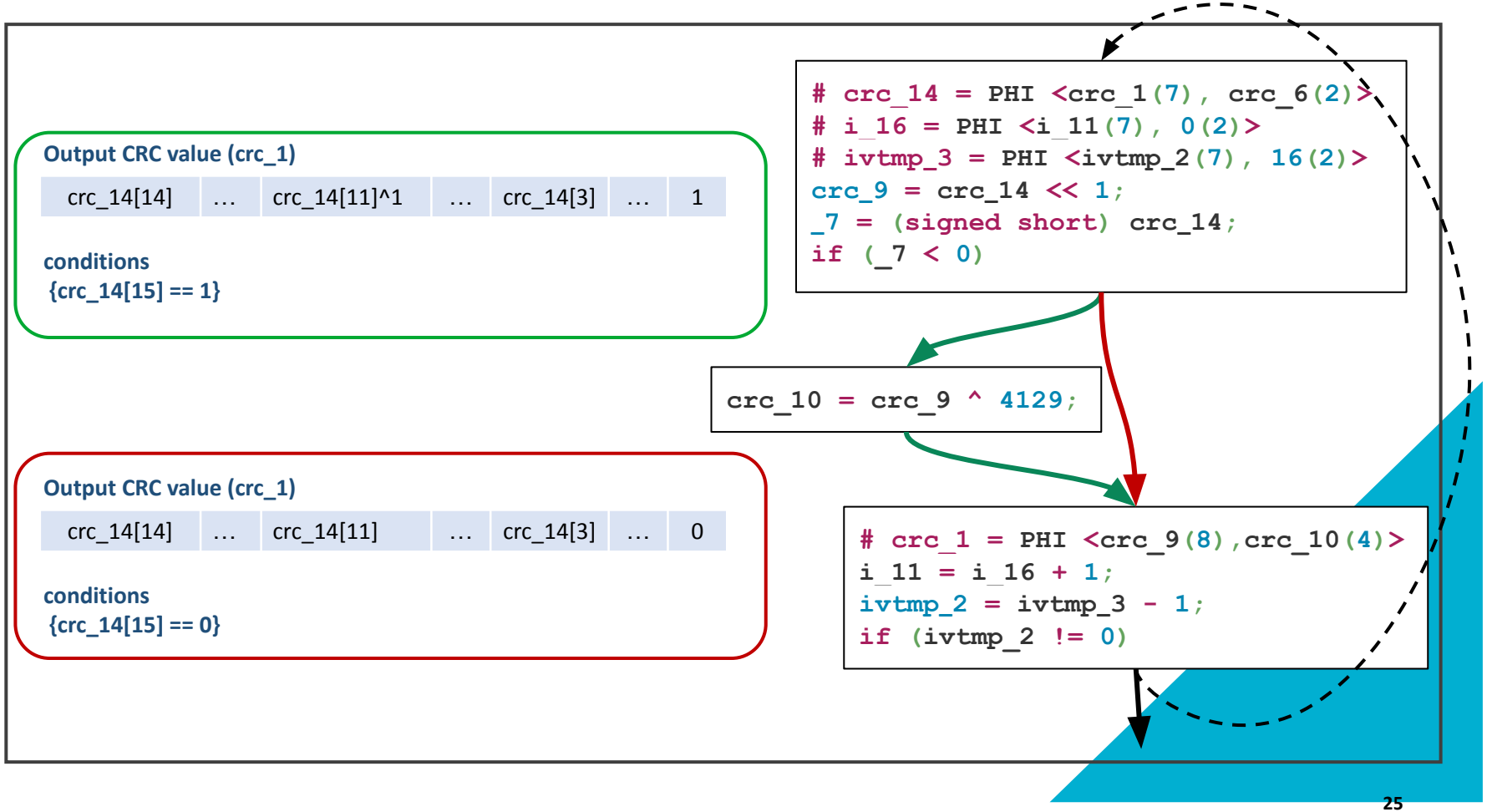
Output CRC value (crc_1)

crc_14[14]	…	crc_14[11]^1	…	crc_14[3]	…	1

conditions
{crc_14[15] == 1}




state1.crc_9 (16-bit)
crc_14[14]
crc_14[13]
.....
crc_14[0]
0
# crc_14 = PHI <crc_1(7), crc_6(2)>
# i_16 = PHI <i_11(7), 0(2)>
# ivtmp_3 = PHI <ivtmp_2(7), 16(2)>
crc_9 = crc_14 << 1;
_7 = (signed short) crc_14;
if (_7 < 0)
crc_10 = crc_9 ^ 4129;
state1.crc_1 (16-bit)
crc_14[14]
crc_14[13]
.....
crc_14[0]
0
# crc_1 = PHI <crc_9(8),crc_10(4)>
i_11 = i_16 + 1;
ivtmp_2 = ivtmp_3 - 1;
if (ivtmp_2 != 0)
conditions
{crc_14[15] == 0}


Output CRC value (crc_1)
crc_14[14] … crc_14[11]^1 … crc_14[3] … 1
conditions
{crc_14[15] == 1}
Output CRC value (crc_1)
crc_14[14] … crc_14[11] … crc_14[3] … 0
conditions
{crc_14[15] == 0}
# crc_14 = PHI <crc_1(7), crc_6(2)>
# i_16 = PHI <i_11(7), 0(2)>
# ivtmp_3 = PHI <ivtmp_2(7), 16(2)>
crc_9 = crc_14 << 1;
_7 = (signed short) crc_14;
if (_7 < 0)
crc_10 = crc_9 ^ 4129;
# crc_1 = PHI <crc_9(8),crc_10(4)>
i_11 = i_16 + 1;
ivtmp_2 = ivtmp_3 - 1;
if (ivtmp_2 != 0)

LFSR matching

LFSR (16_bit)

crc[14]	crc[13]	crc[12]	crc[11]^crc[15]	crc[10]	…	crc[4]^crc[15]	…	crc[0]	crc[15]

Final state1 (16_bit), Conditions {crc[15] == 1}

crc[14]	crc[13]	crc[12]	crc[11]^1	crc[10]	…	crc[4]^1	…	crc[0]	1

Final state2 (16_bit), Conditions {crc[15] == 0}

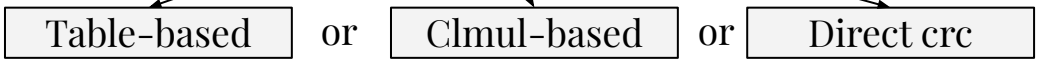
crc[14]	crc[13]	crc[12]	crc[11]	crc[10]	…	crc[4]	…	crc[0]	0

B. Faster CRC generation

B. Faster CRC generation

```
uint16_t crc16 (uint16_t data, uint16_t crc) {
   int carry;
   crc ^= data;
   for (int i = 0; i < 16; ++i) {
       carry = crc & 0x8000;
       crc <<= 1;
       if (carry) crc ^= 0x1021;
   }
   return crc;
}
```

```
uint16_t crc16 (uint16_t data, uint16_t crc)
{
 crc ^= data;
 crc = CRC_IFN (crc, data, polynomial);
 return crc;
}
```



.CRC IFN (CLMUL-based)

For an crc_size-bit CRC in forward/backward bit order

- Compute q+ = x^{2n}/P using long division
- Assign P' field polynomial P with highest term excluded
  - crc = clmul(clmul(data^crc,q+) >>/<< crc_size, P') & mask(crc_size)
- Additional optimization

.CRC IFN (Table-based)

For an crc_size bit CRC and data_size bit data

Generate

```
data_size_type crc_table [256] = {crc0, crc1, …};
```

```
for (int i = 0; i < data_size / 8; i++)
 {
   int index = ((crc >> (crc_size - 8))
             ^ (data >> (data_size - (i + 1) *8))
             & 0xFF);
   crc = (crc << 8) ^ crc_table[index];
 }
```

Results

```
28    #define POLY    (0x1070U << 3)
29    static u8 crc8(u16 data)
30    {
31            int i;
32
33            for (i = 0; i < 8; i++) {
34                    if (data & 0x8000)
35                            data = data ^ POLY;
36                    data = data << 1;
37            }
38            return (u8)(data >> 8);
39    }
```

- Verified 3 bitwise CRC implementations in **Linux Kernel**

Linux/drivers/i2c/i2c-core-smbus.c

Results

- Verified 19 bitwise implementations in **Fedora packages**

```
256  static unsigned short calc_crc(unsigned short crc, unsigned char data)
257  {
258          unsigned int i, j, org, dst;
259          org = data;
260          dst = 0;
261
262          for (i=0; i < CHAR_BIT; i++) {
263                  org <<= 1;
264                  dst >>= 1;
265                  if (org & 0x100) {
266                          dst |= 0x80;
267                  }
268          }
269          data = (unsigned char)dst;
270          crc ^= (unsigned int)data << (16 - CHAR_BIT);
271          for ( j=0; j<CHAR_BIT; j++ ) {
272                  if ( crc & 0x8000U )
273                          crc = (crc << 1) ^ 0x1021U ;
274                  else
275                          crc <<= 1 ;
276          }
277          return crc;
278  }
```

Asterisk/asterisk/main/callerid.c

```
2362  static unsigned char
2363  crc8(unsigned char value)
2364  {
2365      for (int i = 0; i < 8; ++i) {
2366          value = (value & 0x80) ? ((value << 1) ^ 0x31) : (value << 1);
2367      }
2368
2369      return value;
2370  }
2371
```

CROSS-LIB/tools/generic/CC1541/cc1541.c

Tests

Tests that include
- CRC (24)
- Not CRC (12)

implementations

True positive	True negative	False positive	False negative
21	12	0	3

True positive rate	True negative rate	False positive rate	False negative rate
0.875	1	0	0.125

Speed tests on X86-64

Test	Polynomial	Bitwise	Table-based		CLMUL-based	
		Time	Time	Improvement	Time	Improvement
bf-crc8-data8	0x121	44s	11s	75%	38s	14%
bf-crc16-data8	0x11021	54s	10s	81%	37s	31%
bf-crc16-data16		1m18s	20s	74%	34s	56%
bf-crc32-data8	0x182F63B78	48s	12s	75%	37s	23%
bf-crc32-data16		1m51s	22s	80%	37s	67%
bf-crc32-data32		2m39s	40s	75%	33s	79%
bf-crc64-data32	0x142F0E1EBA9EA3693	7m10s	37s	91%	-	-
…						
Average improvement			**54%**		**43%**	

Intel Core i7-3770 @ 3.40GHz, CRC function called 0xFFFFFFFF times

Speed tests on RISC-V

Test	Polynomial	Bitwise	Table-based	
		Time	Time	Improvement
bf-crc8-data8	0x121	4m31s	20s	93%
bf-crc16-data8	0x11021	2m37s	26s	83%
bf-crc16-data16		8m52s	51s	90%
bf-crc32-data8	0x182F63B78	2m46s	26s	84%
bf-crc32-data16		5m31s	51s	85%
bf-crc32-data32		8m5s	1m40s	79%
bf-crc64-data32	0x142F0E1EBA9EA3693	11m5s	1m31s	86%
…				
Average improvement			**66%**	

StarFive VisionFive 2 v1.38, CRC function called 0xFFFFFFFF times

Speed tests on AARCH64

Test	Polynomial	Bitwise	Table-based	
		Time	Time	Improvement
bf-crc8-data8	0x121	1m21s	19s	77%
bf-crc16-data8	0x11021	1m16s	21s	72%
bf-crc16-data16		2m29s	41s	72%
bf-crc32-data8	0x182F63B78	1m16s	24s	68%
bf-crc32-data16		2m32s	43s	72%
bf-crc32-data32		4m32s	1m24s	69%
bf-crc64-data32	0x142F0E1EBA9EA3693	5m5s	1m25s	72%
…				
Average improvement			**50%**	

Raspberry Pi 4 Model B Rev 1.5, CRC function called 0xFFFFFFFF times

Comparing crc32 Instruction Efficiency

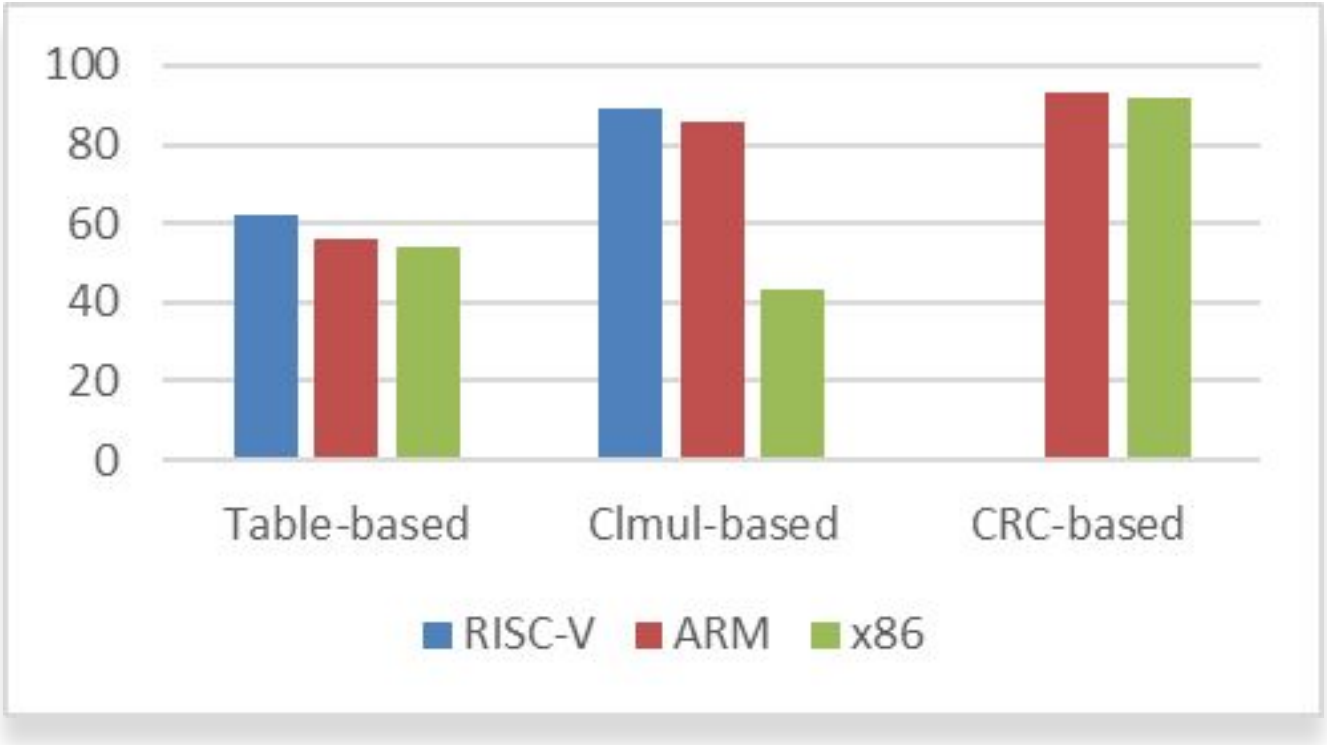
Test	Bitwise	Table-based	CLMUL-based	crc32 instruction
crc32_data32*	2m35s	1m18s	38s	4.4s
crc32-data16*	39.6s	54s	40s	4.5s
crc32-data8*	39.5s	53s	40s	4.4s
Average improvement		**-7%**	**24%**	**92%**

X86-64 architecture

Test	Bitwise	Table-based	crc32 instruction
crc32_data32*	5m7s	1m40s	7.2s
crc32-data16*	1m19s	1m57s	7.1s
crc32-data8*	1m18s	1m57s	7.2s
Average improvement		**-10%**	**93%**

AARCH64 architecture

Average improvement after bitwise CRC replacement



Published Article

M. Arutunian, S. Sargsyan, M. Mehrabyan, L. Bareghamyan and H. Aslanyan, «**Automatic Recognition and Replacement of Cyclic Redundancy Checks for Program Optimization**,» *IEEE Access,* v. 12, pp. 192146 - 192158, 2024.

CAST

Thank you!

mariamarutunian@gmail.com
Researcher at CAST
(Center of Advanced Software Technologies - castech.am)