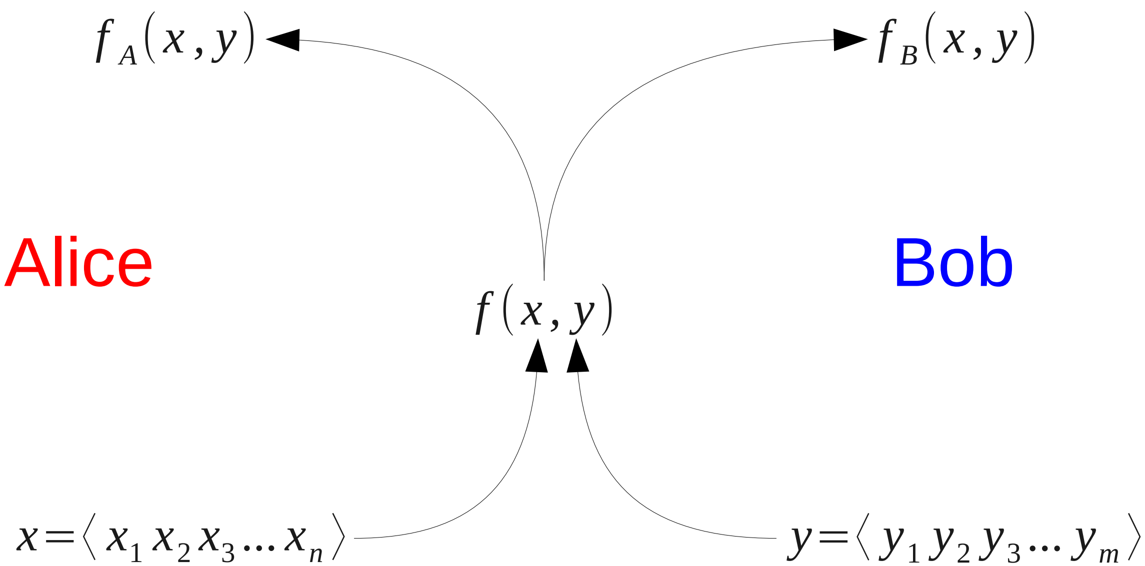
PCF: Scaling Secure Computation

Benjamin Kreuter, abhi shelat,
University of Virginia

Benjamin Mood, Kevin Butler
University of Oregon

Secure 2-Party Computation

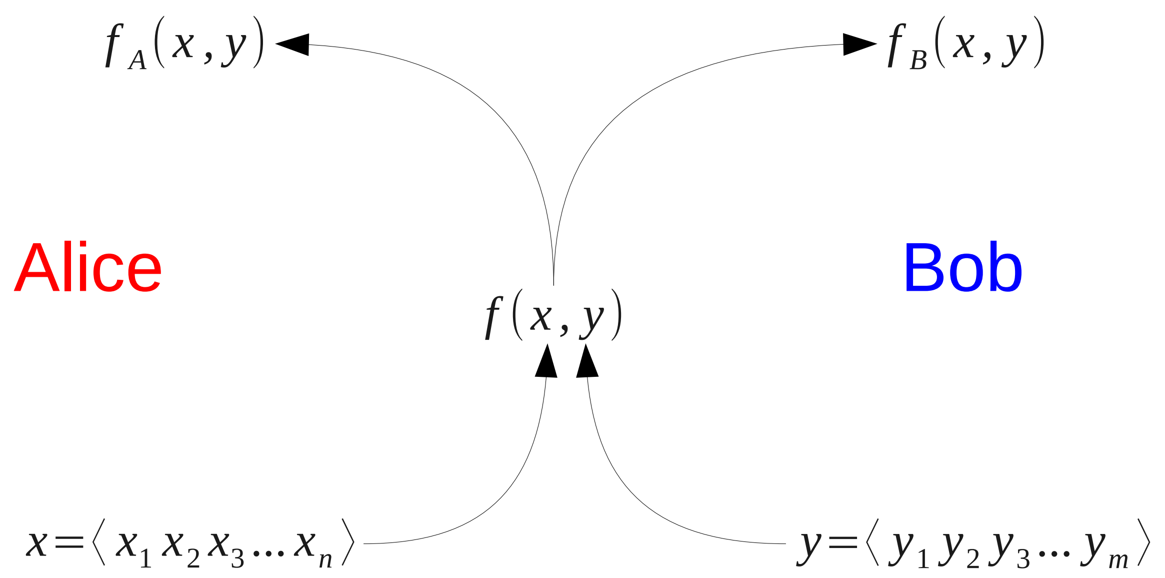


Guarantee: *x* and *y* remain private, outputs are correct

Key result [Yao82]:

Secure 2-party protocols exist for *any* computable function

Secure 2-Party Computation



Guarantee: *x* and *y* remain private, outputs are correct

Need *oblivious* representation of *f*

```
x = read_input();
if (x > 5) {
      y = 7;
} else {
      y = 12;
}
```

Leaks information about *x*

Oblivious Programs

- Control flow, memory access, etc. are independent of program inputs
- Key result: Pippenger and Fischer oblivious Turing machine construction
  - Logarithmic overhead
  - Also gives generic circuit family construction

Oblivious Programs

```
x = read_input();
if (x > 5) {
    y = 7;
} else {
    y = 12;
}
```

```
x = read_input();
c1 = x > 5;
y1 = 7;
c2 = !c1;
y2 = 12;
y = (y1 & c1) || (y2 & c2);
```

Multiplexer

Prior Work

Solved protocol scalability problem

2013 Tens of billions
This work

2012 Billions
[HEKM'11, KSS'12]

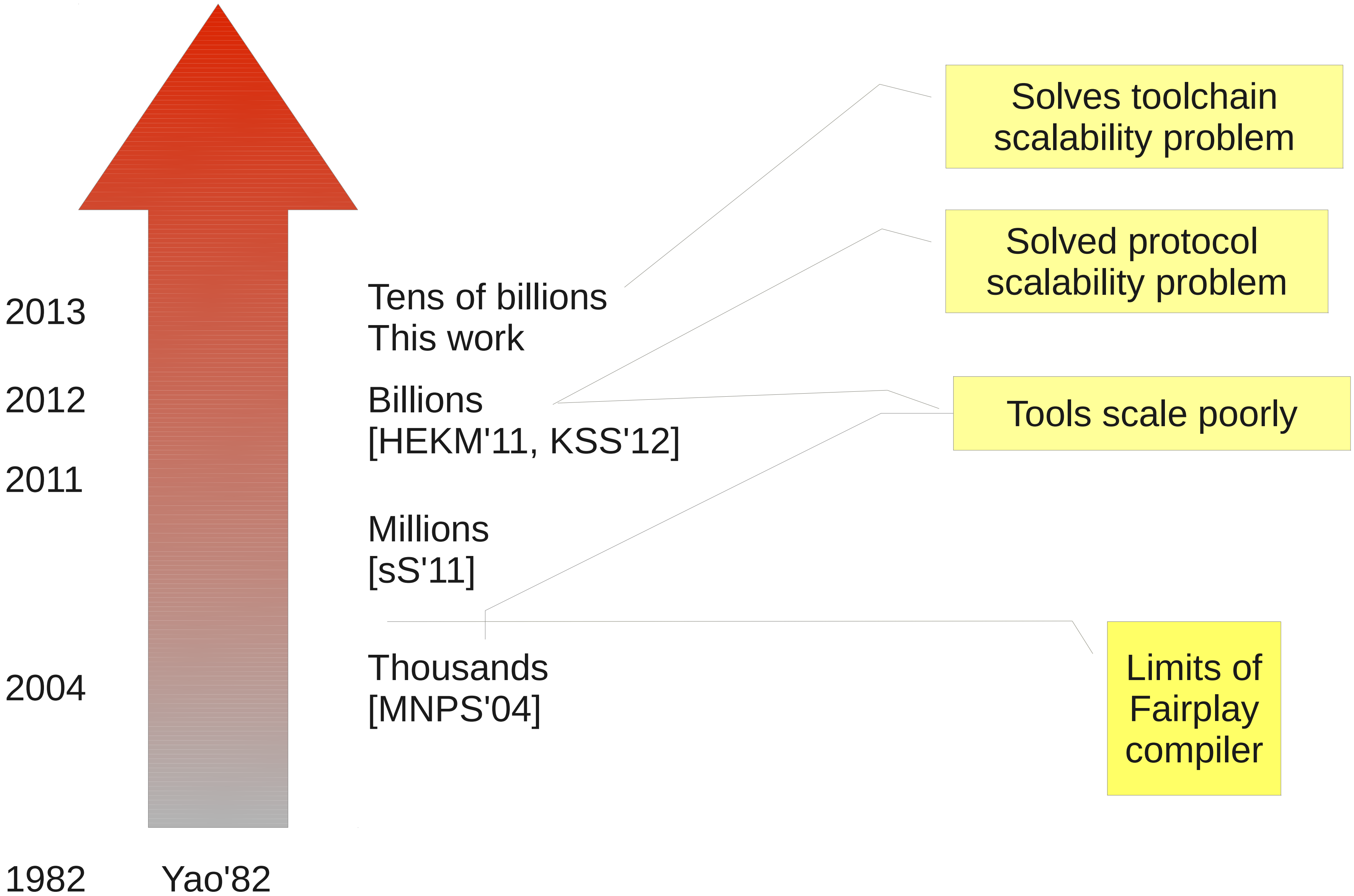
2011

Millions
[sS'11]

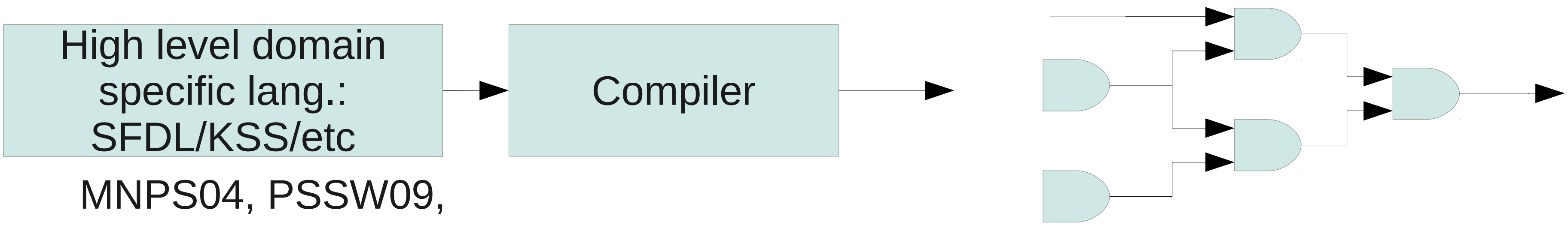
2004 Thousands
[MNPS'04]

1982 Yao'82

Prior Work


Solves toolchain scalability problem
Solved protocol scalability problem
2013
Tens of billions
This work
2012
Billions
[HEKM'11, KSS'12]
Tools scale poorly
2011
Millions
[sS'11]
2004
Thousands
[MNPS'04]
Limits of Fairplay compiler
1982
Yao'82

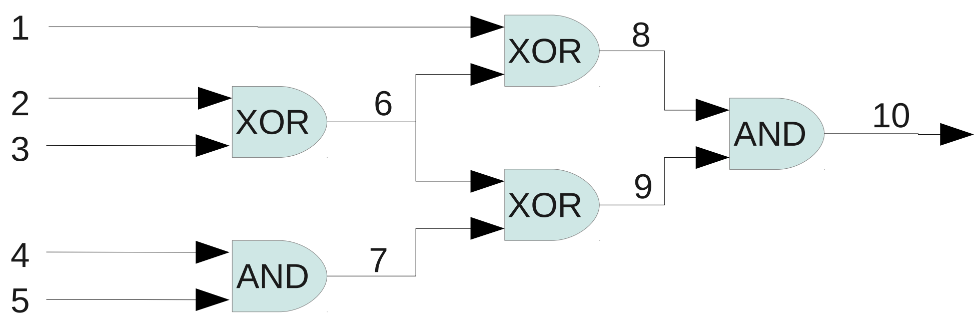
Previous Approaches



MNPS04, PSSW09, KSS12, HFKV12

- Circuits are described as lists of gates
  - Loops must be unrolled
  - Functions must be inlined
  - Conditionals must be flattened

Previous Approaches



1 = Input

2 = Input

3 = Input

4 = Input

5 = Input

6 = XOR(2,3)

7 = AND(4,5)

8 = XOR(1,6)

9 = XOR(6,7)

10 = AND(8,9)

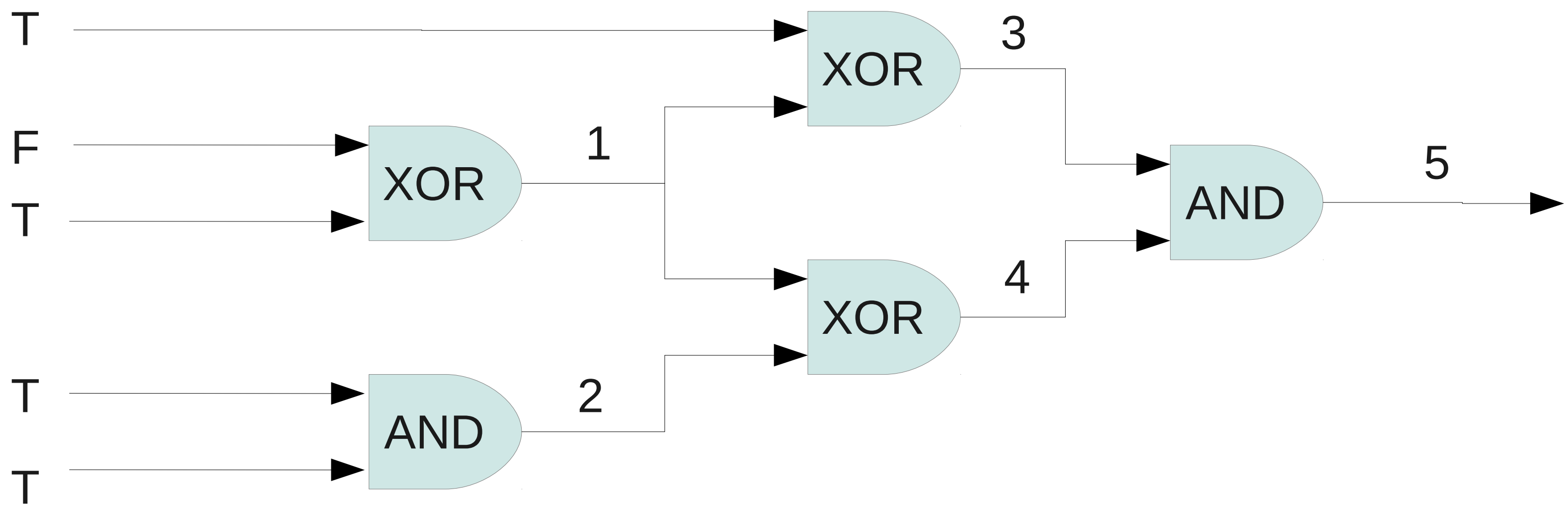
Problem:
Storage size = worst case running time

Previous Approaches

- Machine resources limit the size of circuits that can be optimized or stored
- Storage requirements grow with worst case running time
  - Millions of gates = many gigabytes

	KSS12		HFKV12	
Function	**Circuit Size**	**Compile Time**	**Circuit Size**	**Compile Time**
32-bit Integer Mult.	1.8MB	0.55s	785kB	6.43s
1024-bit Integer Mult.	112MB	430s	??	??
16x16 Matrix Mult.	432MB	2,200s	206MB	2,600
256-bit RSA	15GB	24,000s	-	-
1024-bit RSA	??	??	-	-

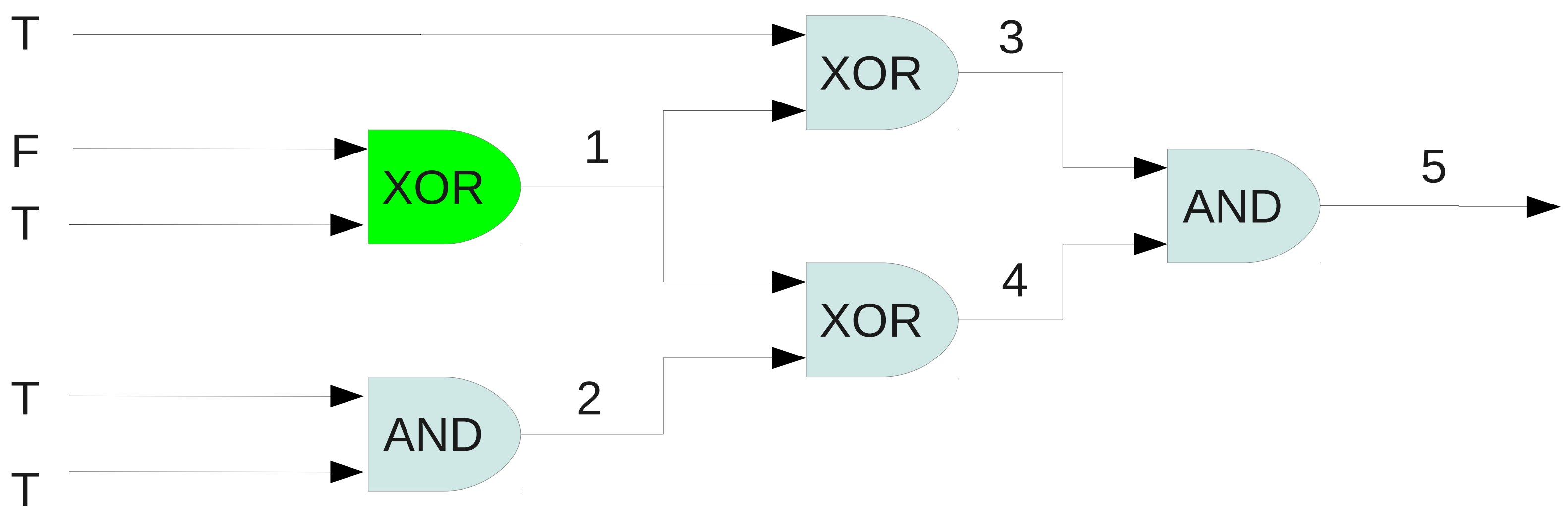
Related problem: Evaluating Circuits



Wire values

1	2	3	4	5

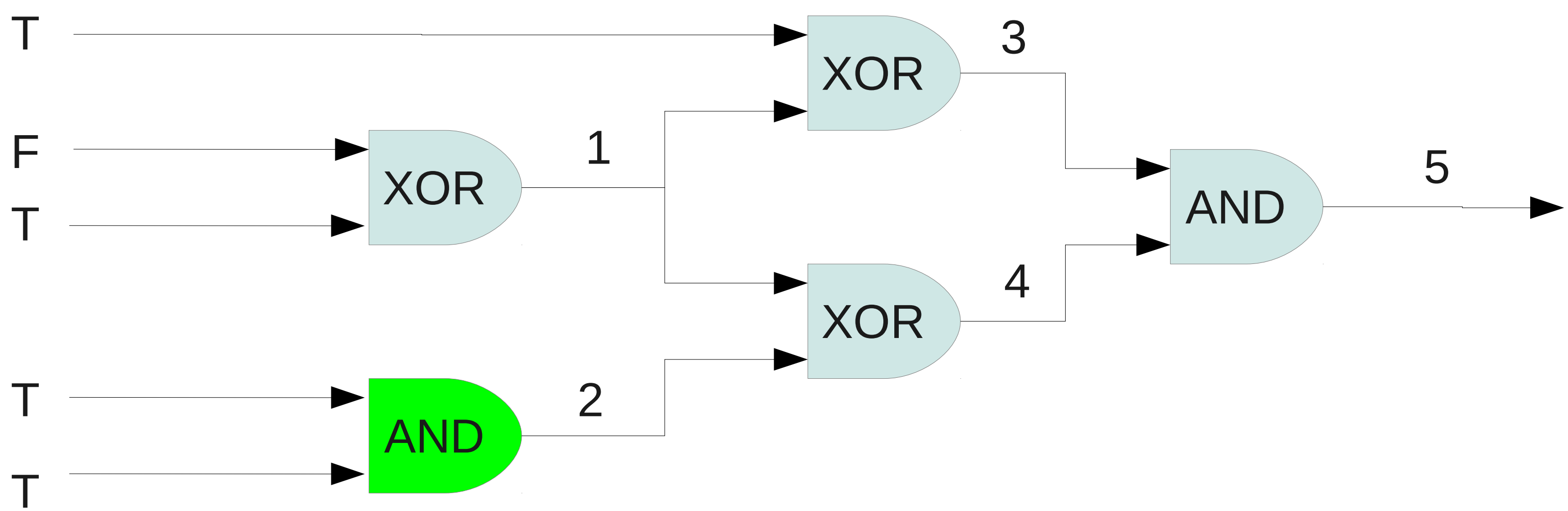
Related problem: Evaluating Circuits



Wire values

1	2	3	4	5
T				

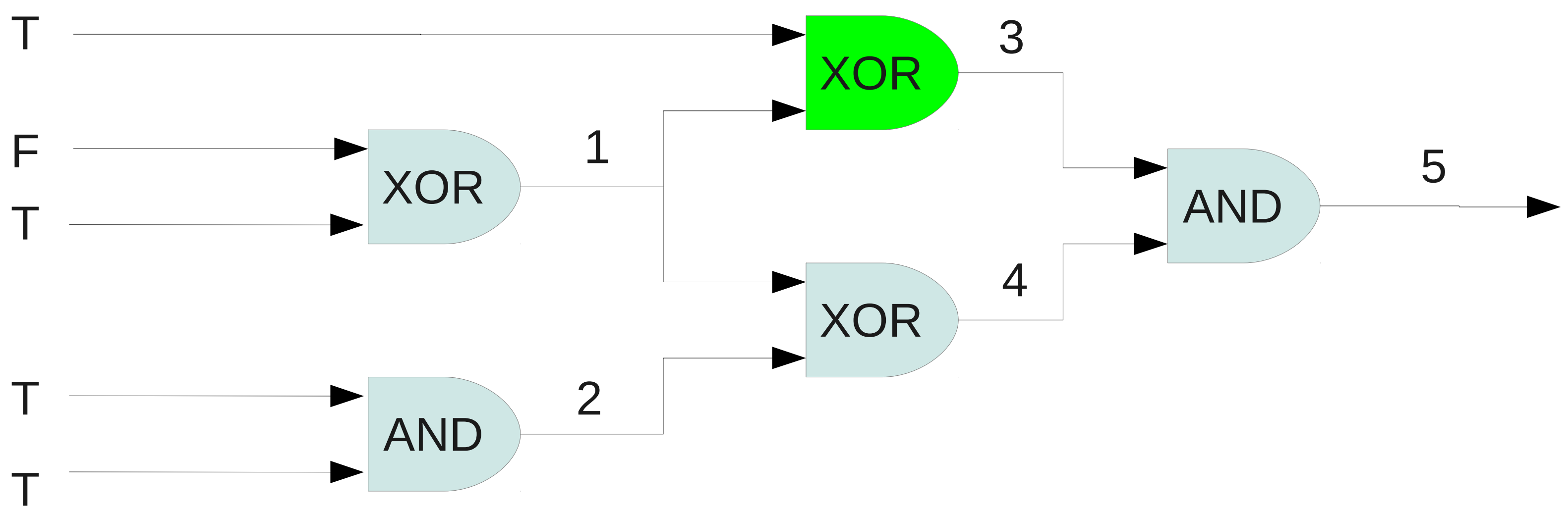
Related problem: Evaluating Circuits



Wire values

1	2	3	4	5
T	T			

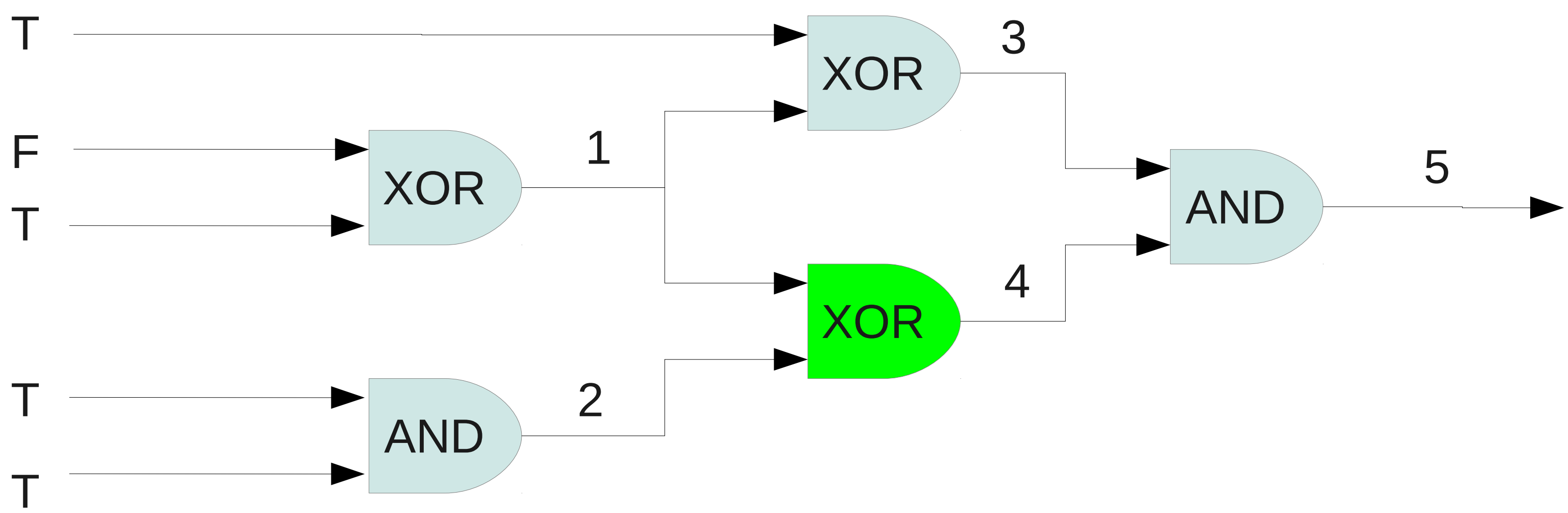
Related problem: Evaluating Circuits



Wire values

1	2	3	4	5
T	T	F		

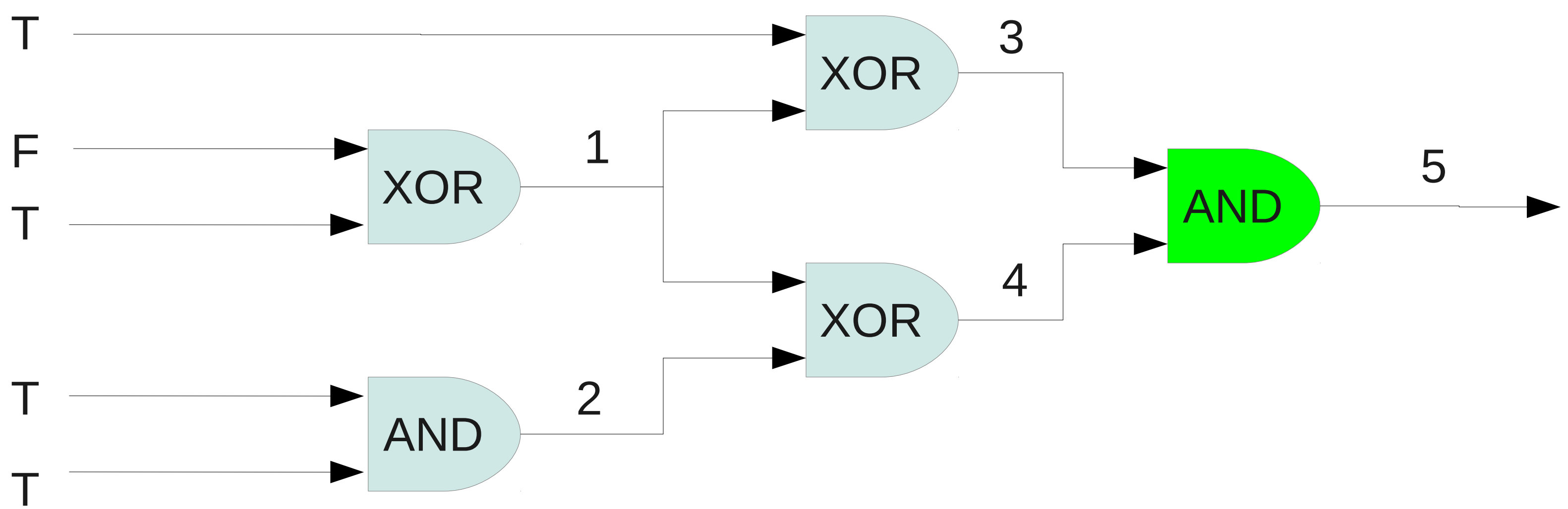
Related problem: Evaluating Circuits



Wire values

1	2	3	4	5
T	T	F	F	

Related problem: Evaluating Circuits



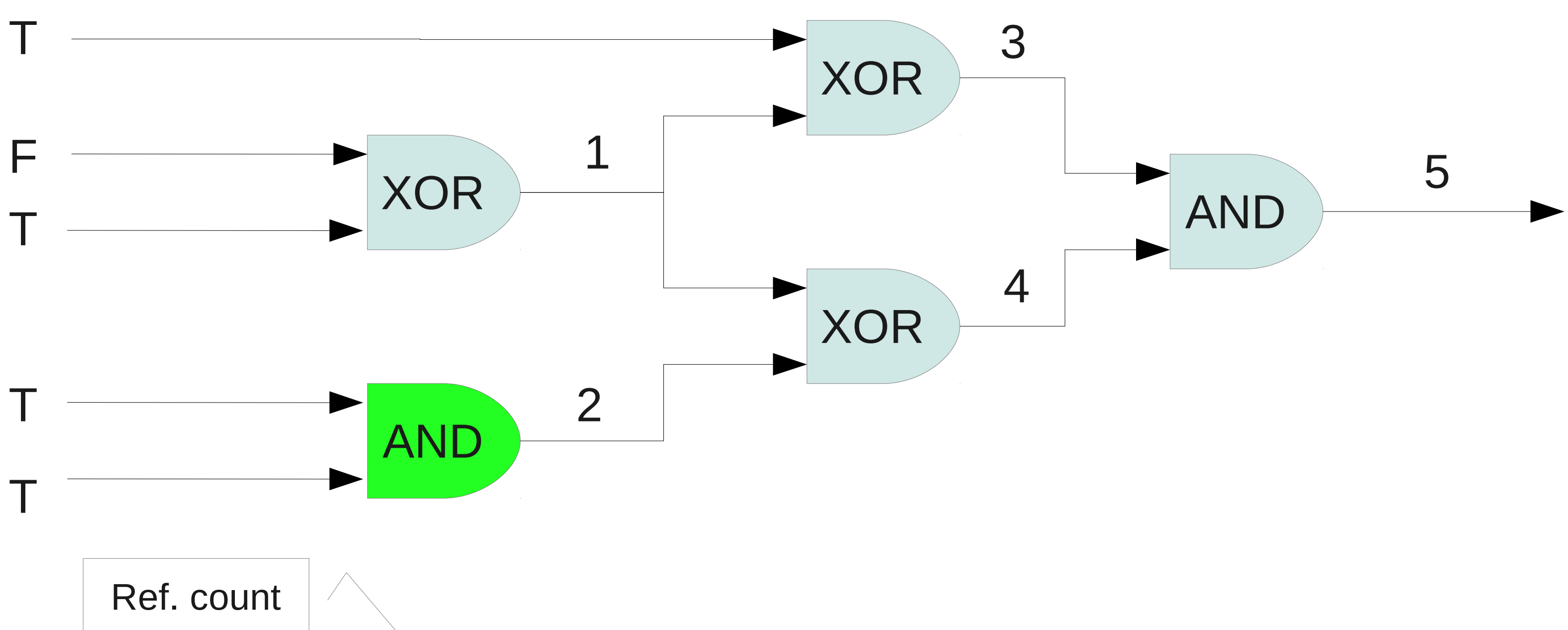
Wire values

1	2	3	4	5
T	T	F	F	F

Problem: Memory requirement grows with running time!

KSS12 Approach

- Observation: Wire values are not needed after their last use as an input to a gate

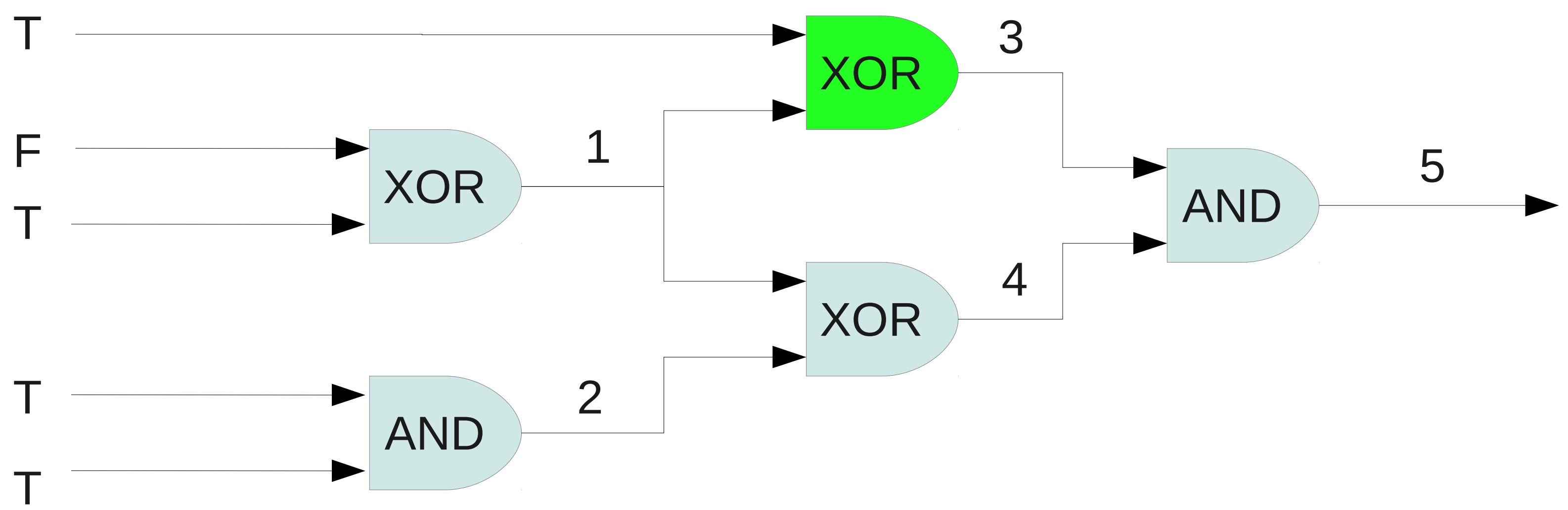


Wire values

1 **2**	2 **1**	3 **1**	4 **1**	5 **1**
T	T			

KSS12 Approach

- Observation: Wire values are not needed after their last use as an input to a gate

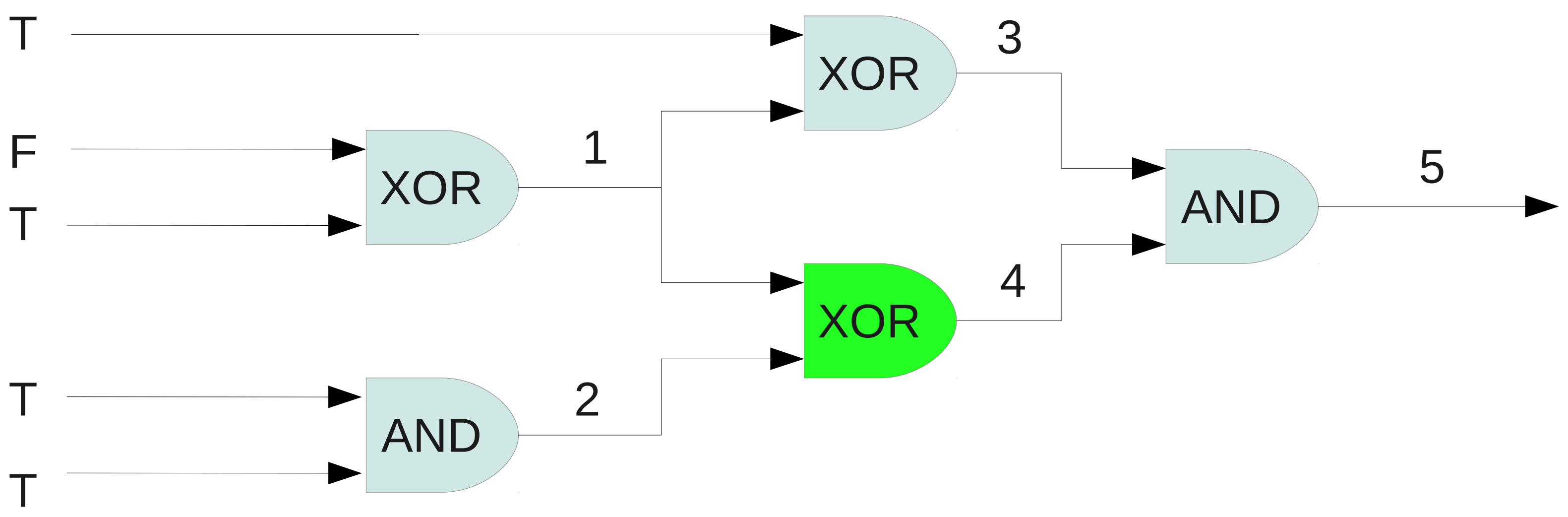


Wire values

1 1	2 1	3 1	4 1	5 1
T	T	F		

KSS12 Approach

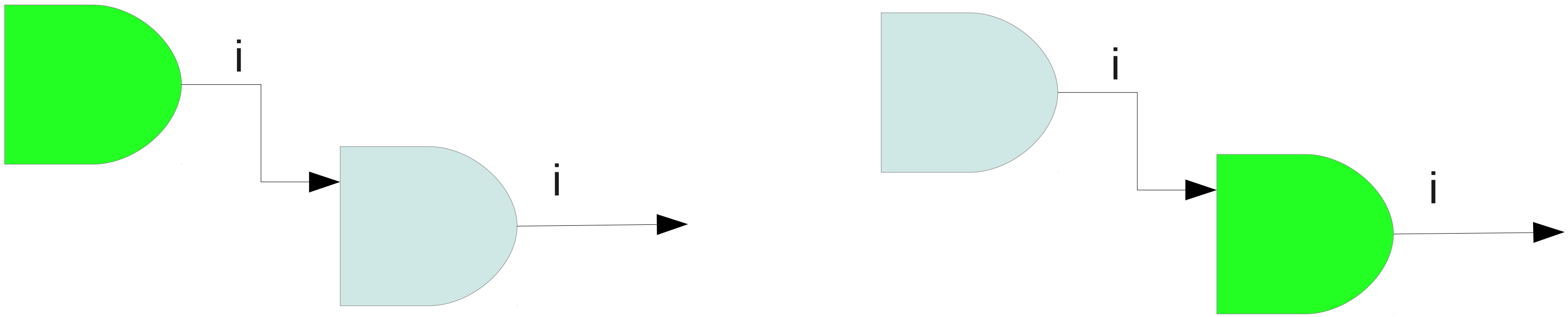
- Observation: Wire values are not needed after their last use as an input to a gate



Wire values

1 0	2 0	3 1	4 1	5 1
T	T	F	F	

Key Insight (1): Overwriting

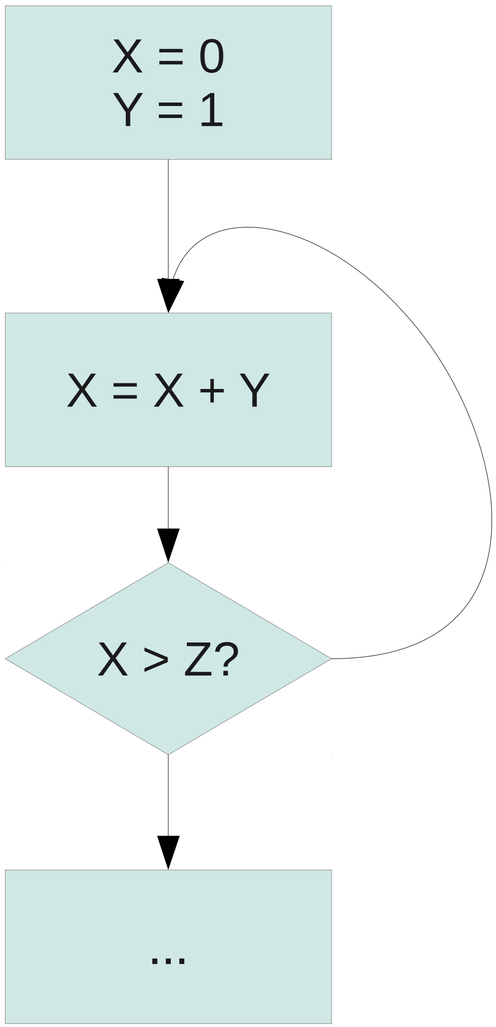


	i	
...	x	...

	i	
...	y	...

- When wire values are not needed, just overwrite them
- Simpler than reference counting
- The compiler can use high-level information (scope, assignments, etc.) to determine when a wire can be overwritten
- Removes need for unique wire IDs – just need the index in the table

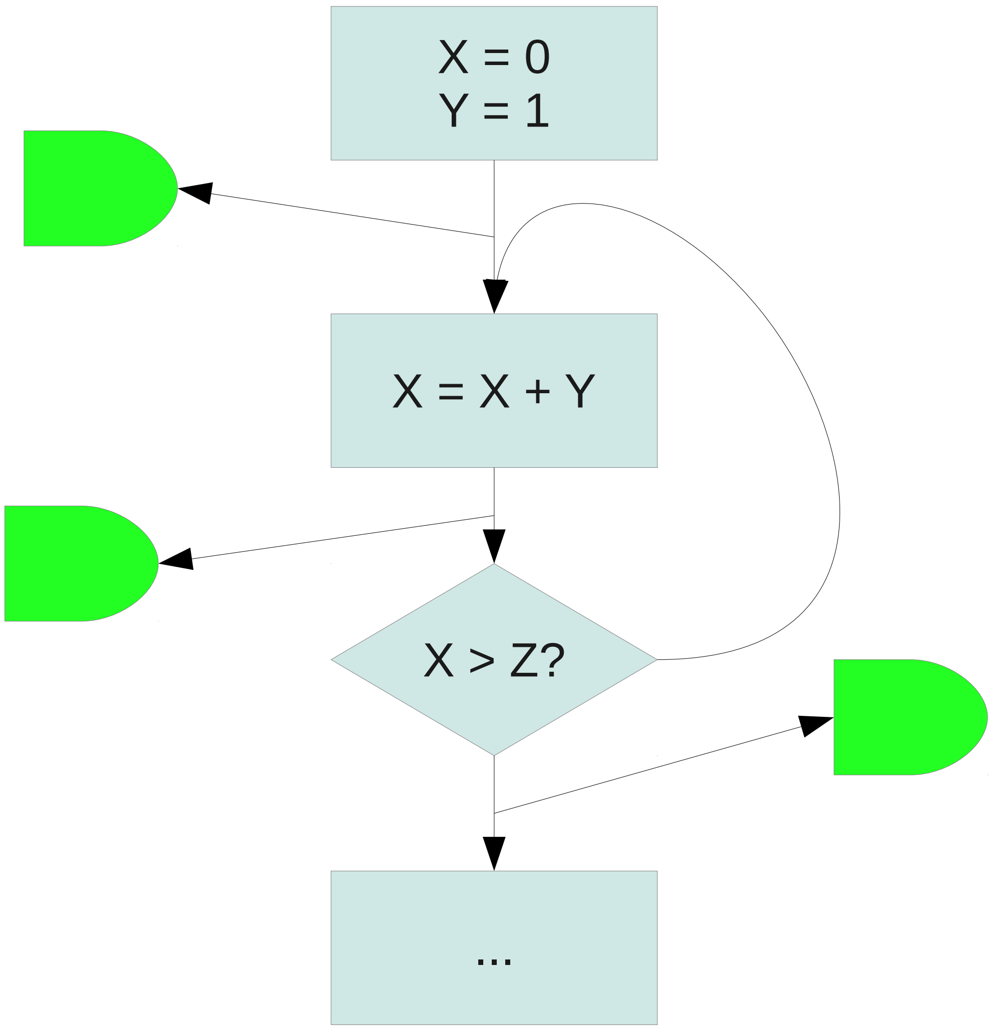
Key Insight (2): JIT Gate Generation



- Unrolling loops carries a heavy cost: the loop body is repeated many times
- Control-flow graphs are more compact than circuits
  - CFGs are also useful for optimization

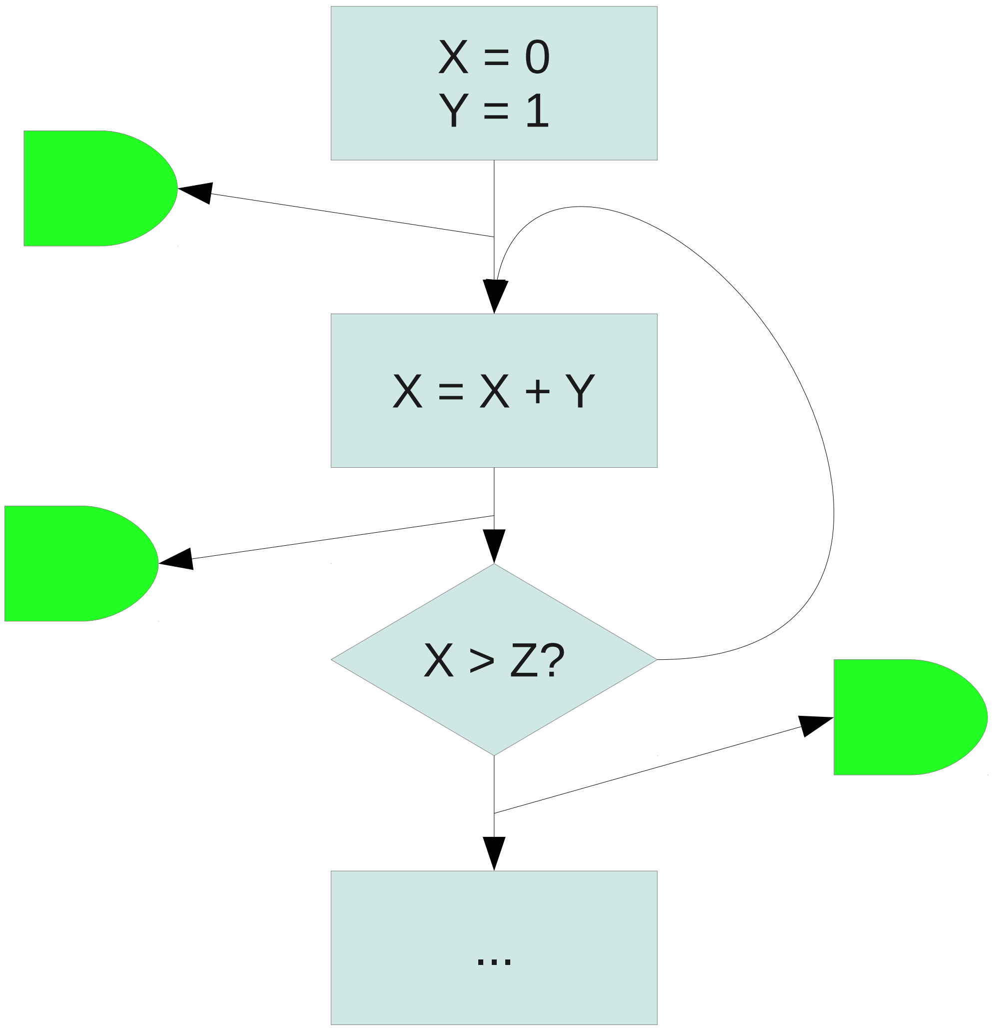
Key Insight (2): JIT Gate Generation

- Loops do not need to be unrolled immediately
- Just-in-time gate generation



Key Insight (2): JIT Gate Generation

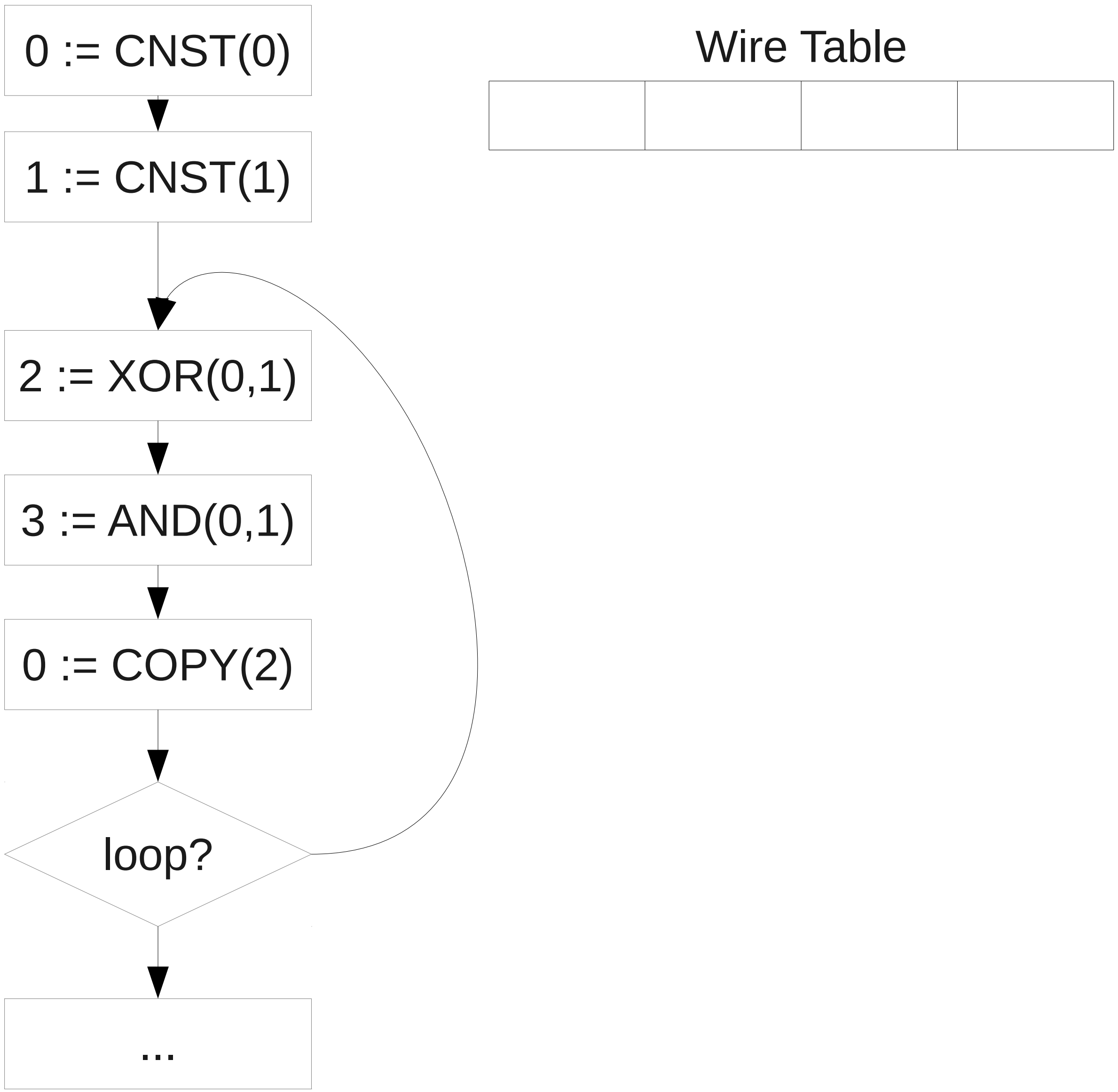
- Loops do not need to be unrolled immediately
- Gates emitted as side effects of state transitions
- Now we deal with <u>programs,</u> not circuits



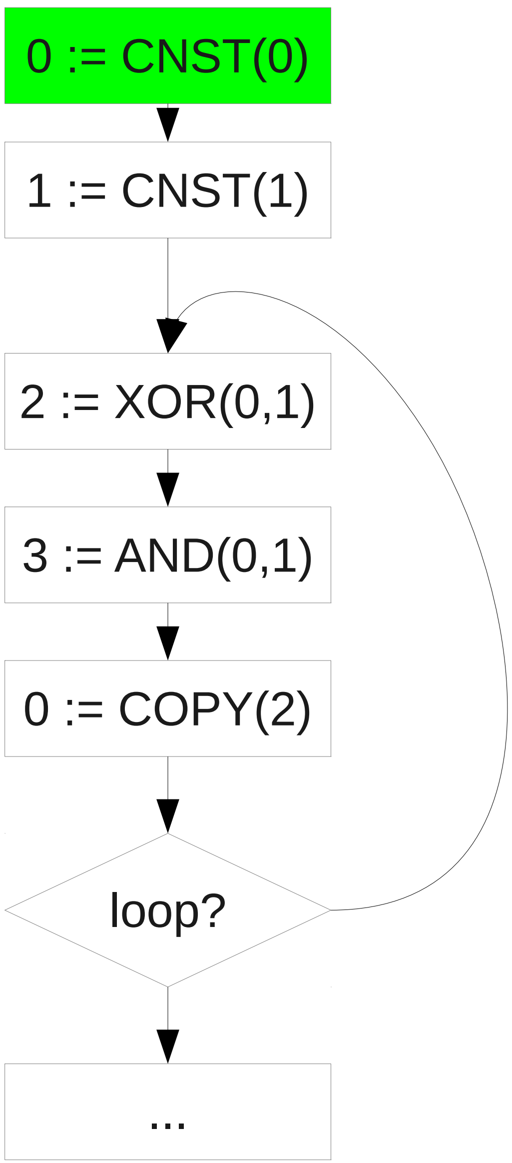
PCF

- PCF combines these two ideas
- This requires no changes to secure 2-party computation protocols
  - The PCF runtime emits a stream of gates – this can be used like any other description
  - No compromises on security

PCF Runtime



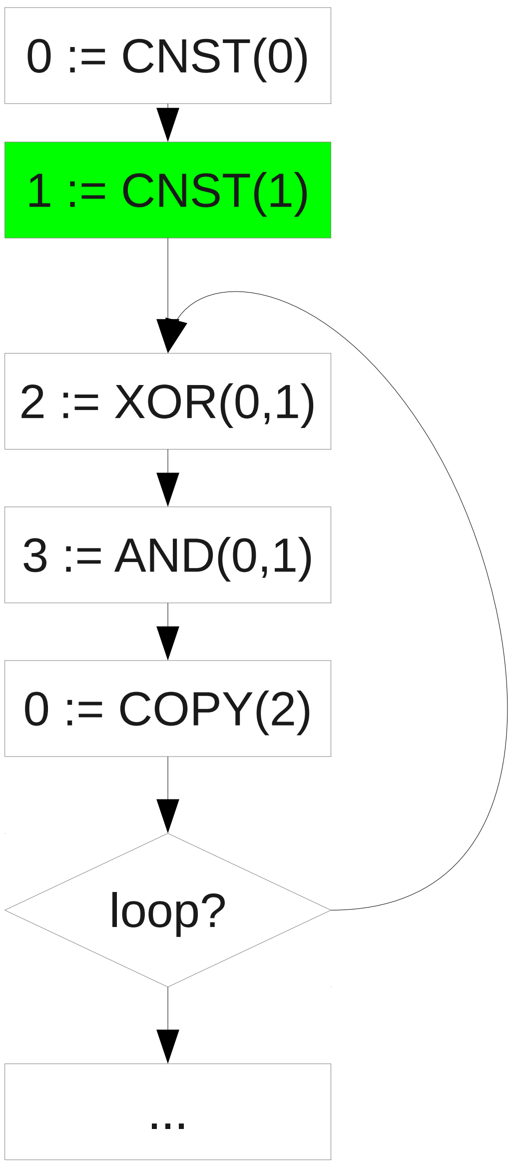
PCF Runtime



Wire Table

0			

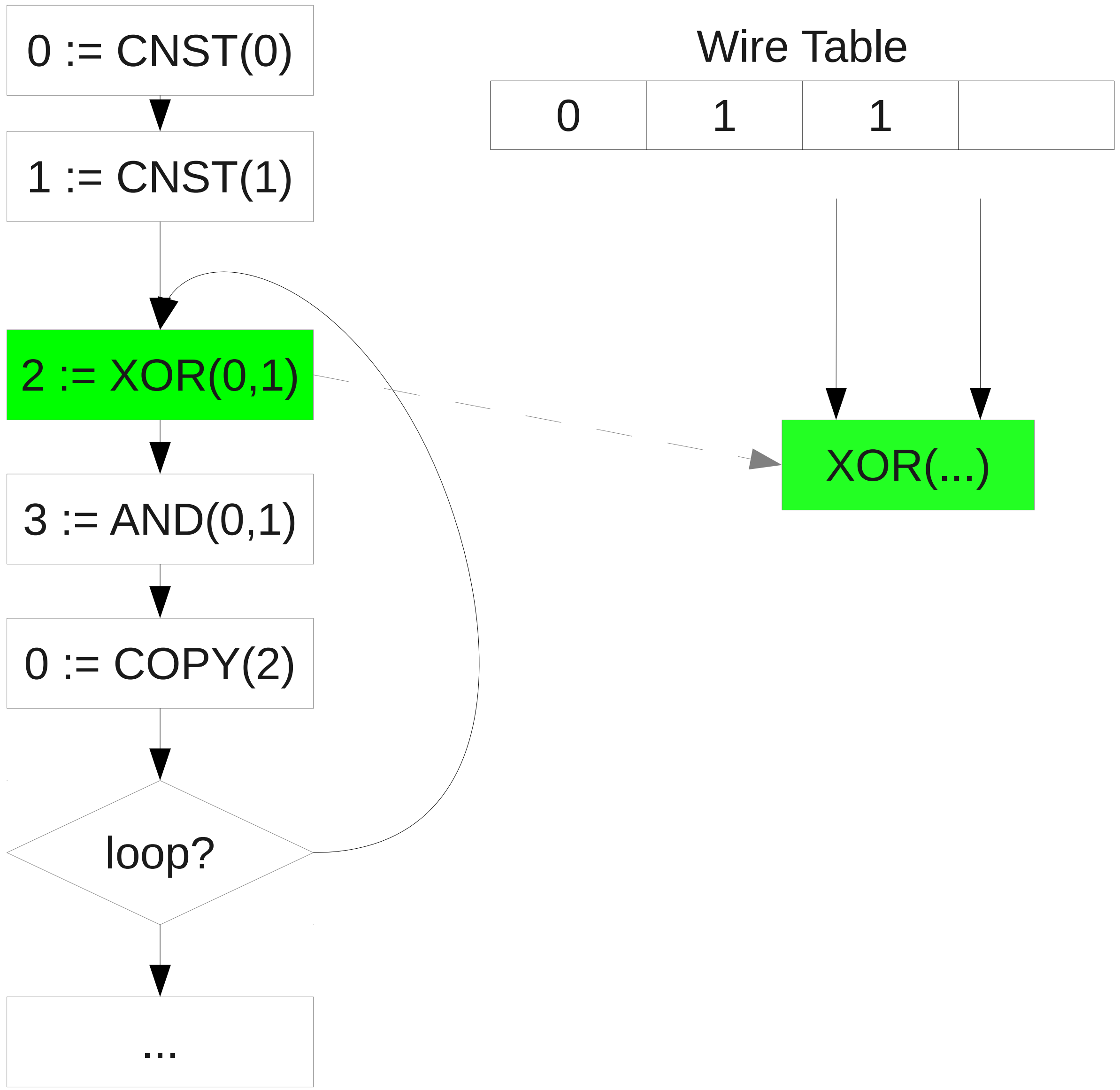
PCF Runtime



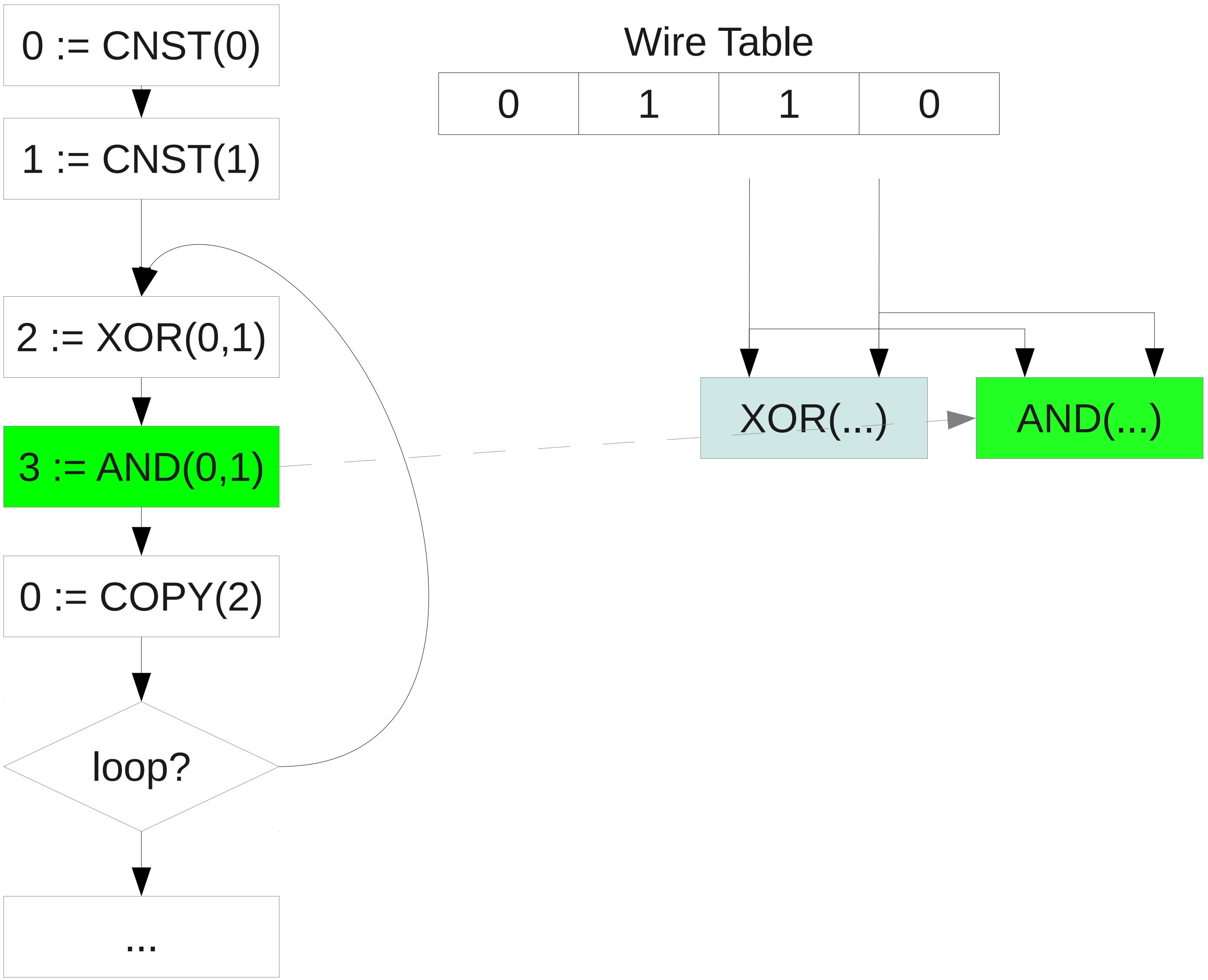
Wire Table

0	1		

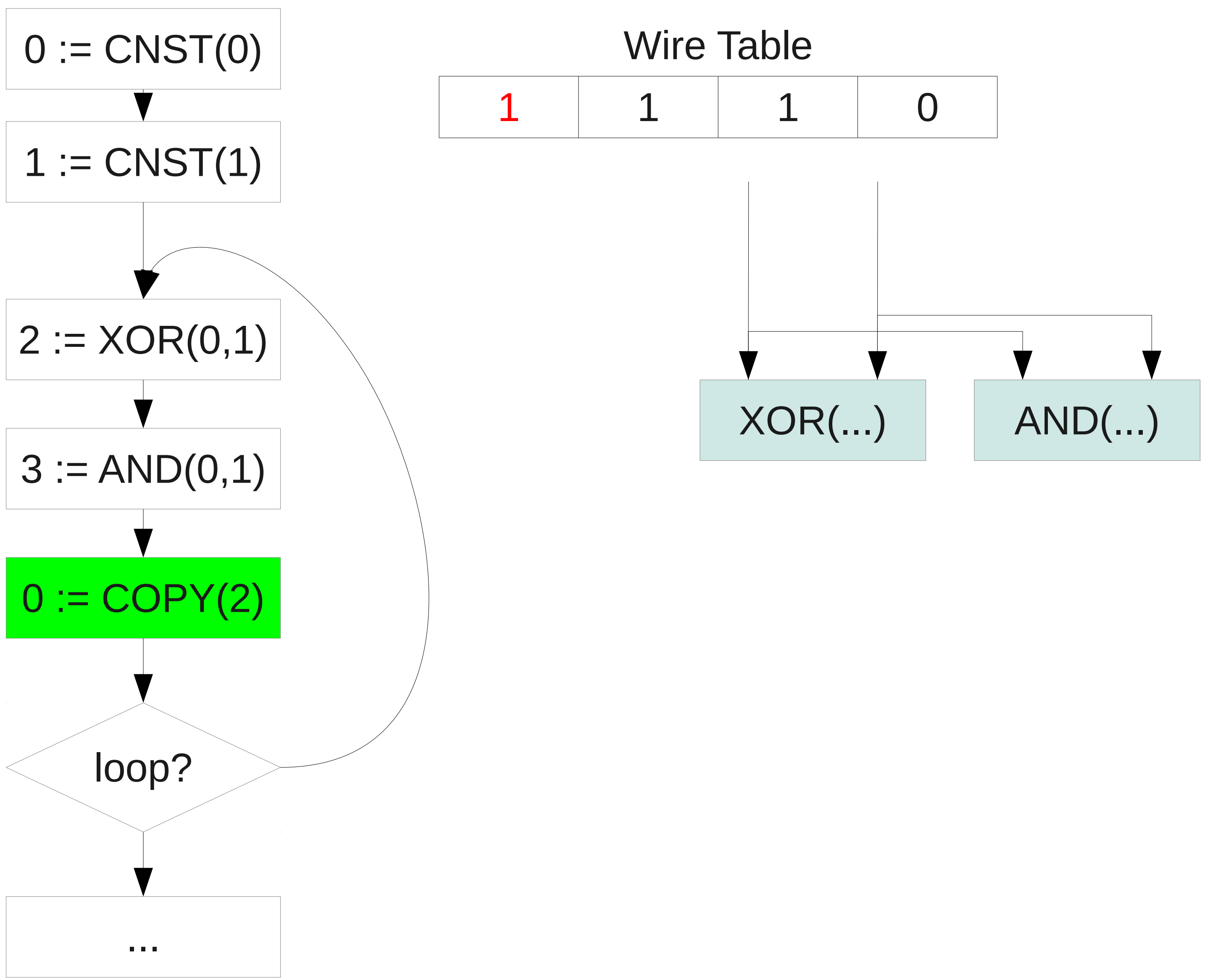
PCF Runtime



PCF Runtime

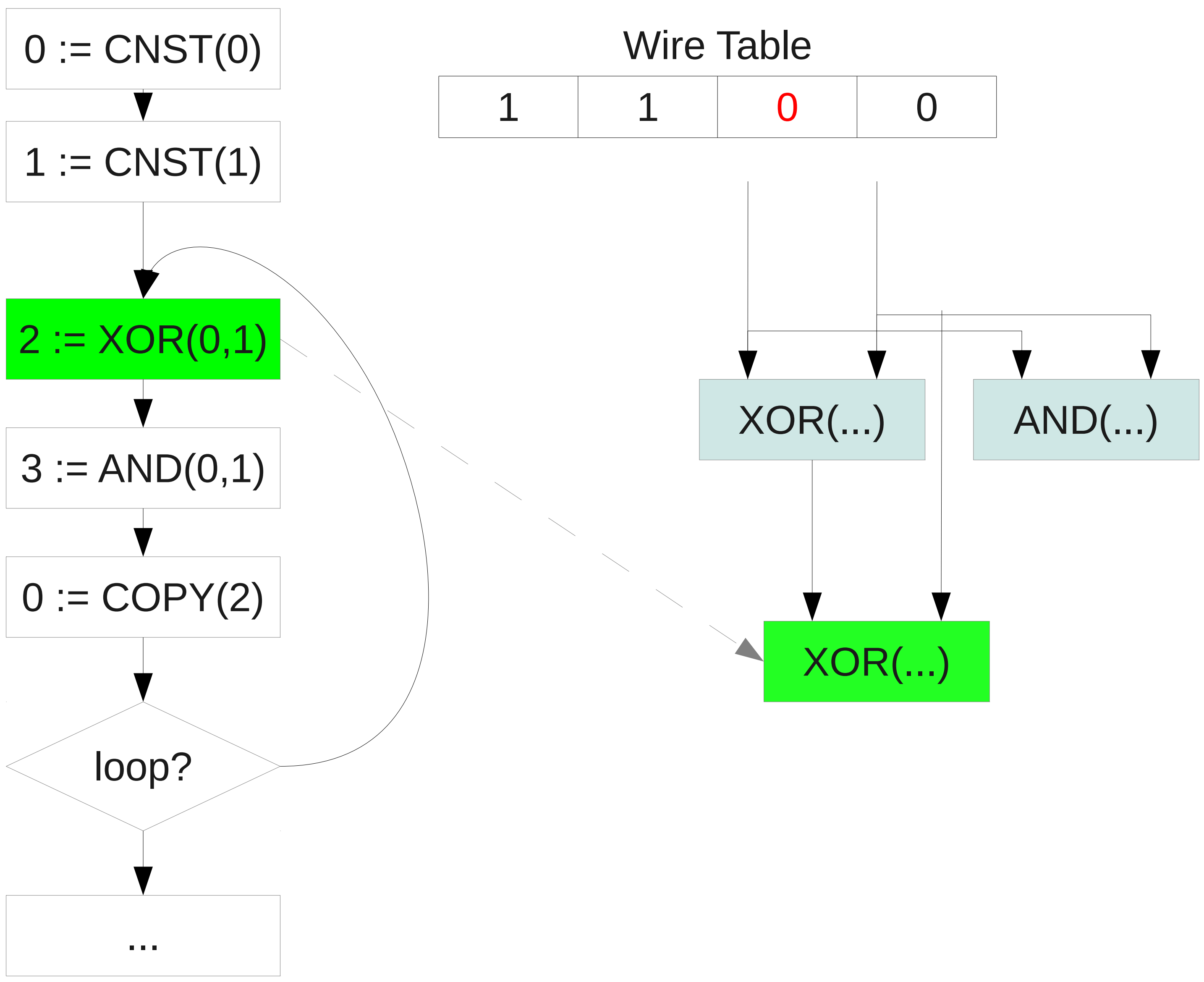

0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)
loop?
...
Wire Table
0
1
1
0
XOR(...)
AND(...)

PCF Runtime

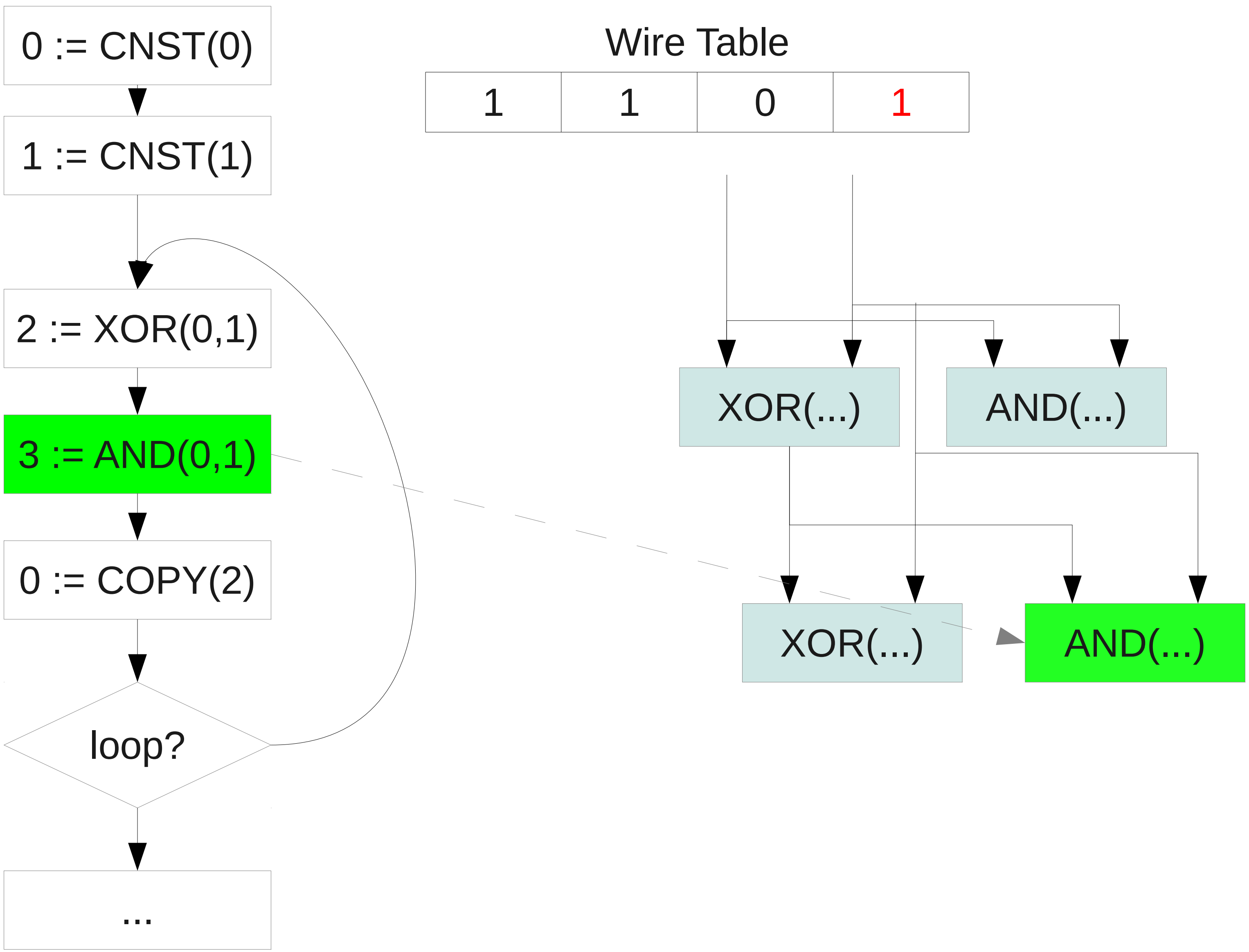


0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)
loop?
...
Wire Table
1
1
1
0
XOR(...)
AND(...)

PCF Runtime

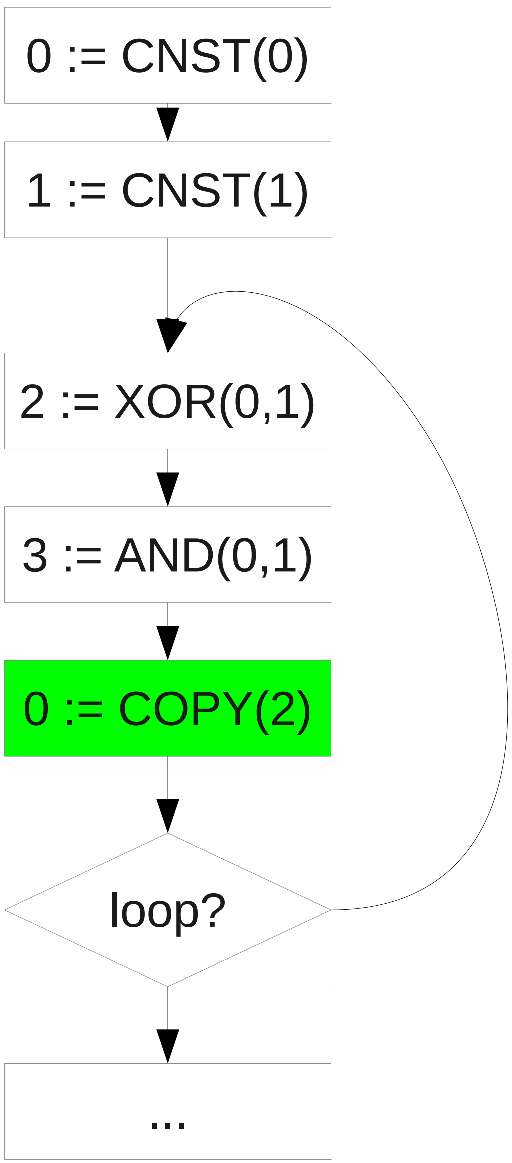

0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)
loop?
...
Wire Table
1
1
0
0
XOR(...)
AND(...)
XOR(...)

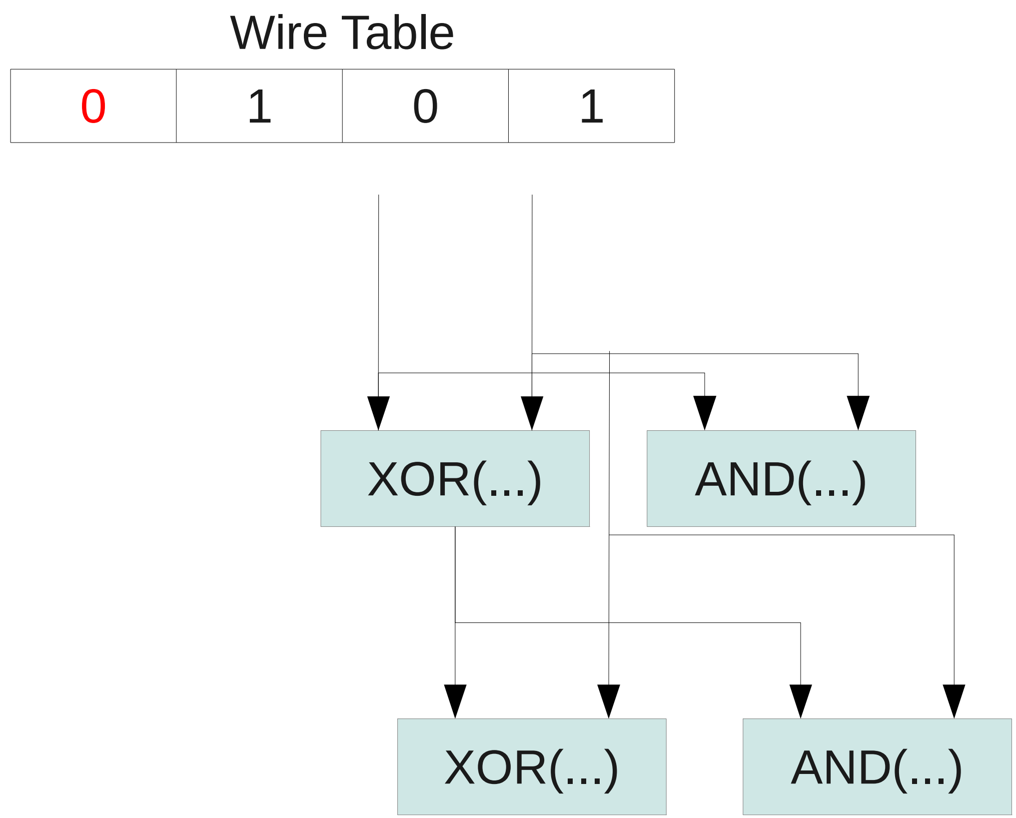
PCF Runtime



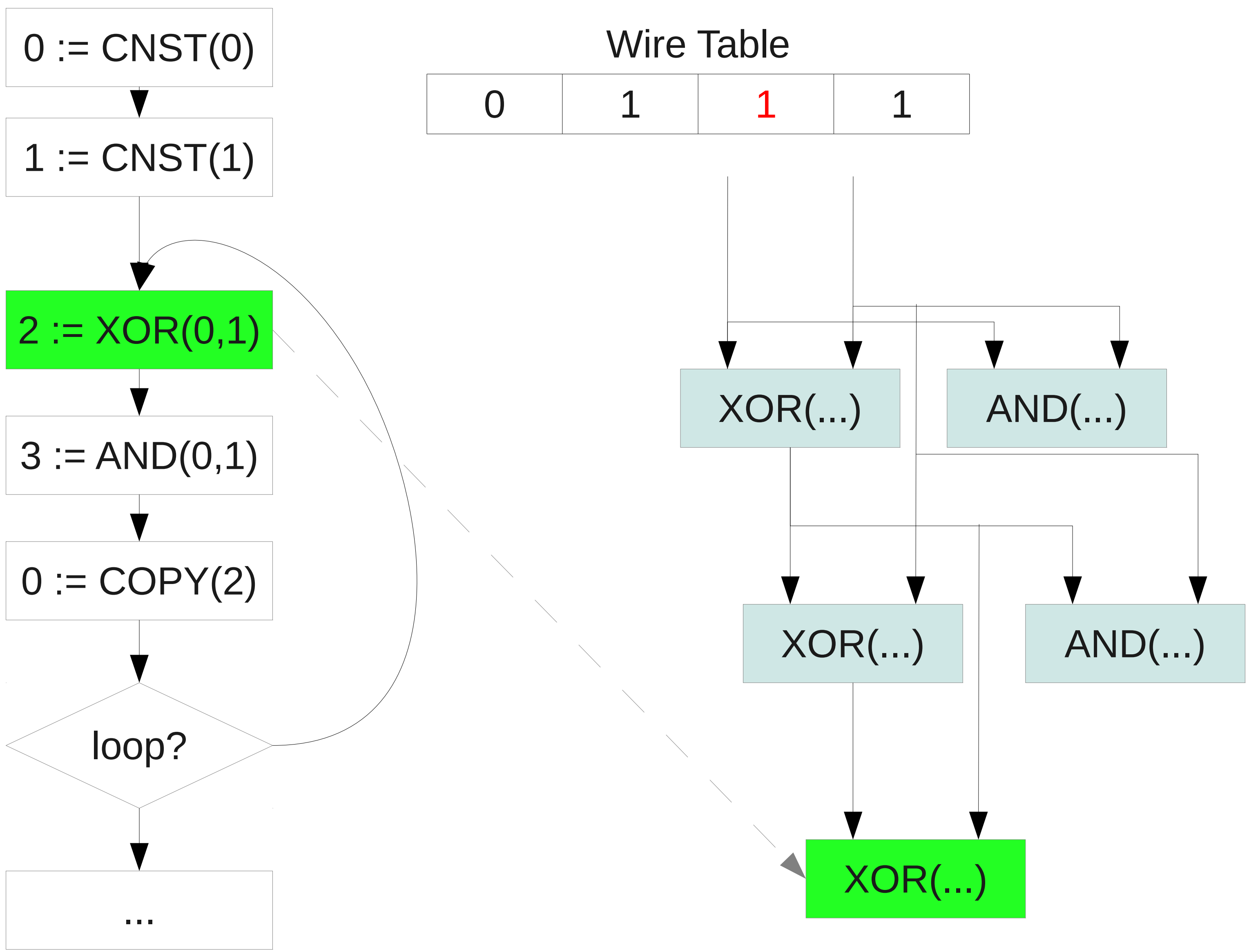
0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)
loop?
...
Wire Table
1
1
0
1
XOR(...)
AND(...)
XOR(...)
AND(...)

PCF Runtime


0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)
loop?
...


Wire Table
0
1
0
1
XOR(...)
AND(...)
XOR(...)
AND(...)

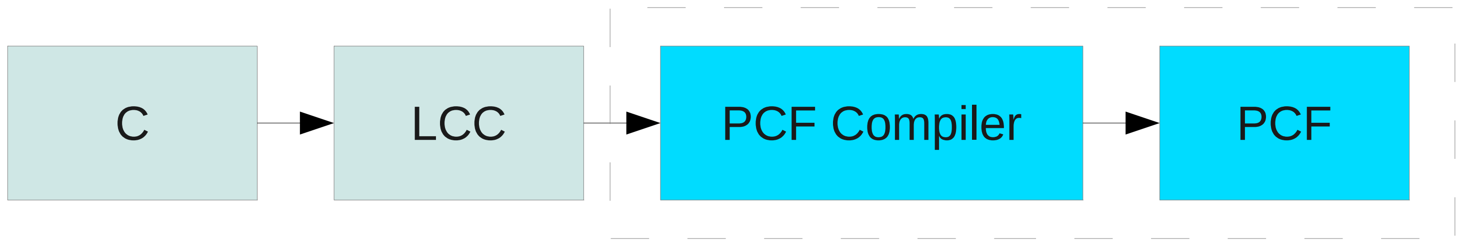
PCF Runtime


0 := CNST(0)
1 := CNST(1)
2 := XOR(0,1)
3 := AND(0,1)
0 := COPY(2)
loop?
...
Wire Table
0
1
1
1
XOR(...)
AND(...)
XOR(...)
AND(...)
XOR(...)

Key Insight (3): Start with Bytecode

- Previous systems support one language
  - Fairplay, KSS'12 use a domain specific language
  - HFKV'12 support C (suggest LLVM bytecode as future work)
- Our system reads a bytecode format as input
  - Any language can be compiled to bytecode; any language can be supported

Our System



- No changes to C, just restrictions on what programs can be executed (e.g. loop termination must not depend on input values, no pointers to functions)
- Nothing special about LCC – could use JVM, LLVM, etc. to support other languages
- Can handle big functions, tens of billions of gates or more, using just a laptop computer

k-width Millionaire's

```
unsigned int alice(unsigned int,unsigned int);
unsigned int bob(unsigned int,unsigned int);
void output_alice(unsigned int);

void main(void)
{
  unsigned int res = 0x00000001, x=0;
  unsigned int i;
  unsigned int borrow;

  for(i = 0; i < 128;)
    {
      unsigned int a1 = alice(i,0);
      unsigned int b1 = bob(i,0);
      unsigned int b = borrow;
      borrow = 0;

      if(a1 < (b1 + b))
        borrow = 1;

      i += 32;
    }
  if(borrow == 0) x= 0x1;
  else x = 0xffffffff;

  output_alice(x);
}
```

Regex Matching

```
unsigned int alice(unsigned int,unsigned int);
unsigned int bob(unsigned int,unsigned int);
void output_alice(unsigned int);
void output_bob(unsigned int);

#define N 4
#define M 1024

unsigned int transZ[16*N];
unsigned int transO[16*N];

void read_table(void)
{
  unsigned int i = 0, inp = 0;

  for (i = 0; i < N; i++)
    {
      inp = alice(32*i, 0);
      transZ[2*i] = inp & 0xFF;

      transO[2*i] = (inp >> 8) & 0xFF;

      transZ[2*i+1] = (inp >> 16) & 0xFF;

      transO[2*i+1] = (inp >> 24) & 0xFF;
    }
}
```

```
void main(void)
{
  unsigned int i = 0, j = 0, k=0, z = 0, inp = 0;
  unsigned int state;

  read_table();

  state = 0;
  for(z = 0; z < M; z++)
    {
      inp = bob(32*z,0);
      for(i = 0; i < 32; i++)
        {
          for(j = 0; j < 16*N; j++)
            {
              unsigned int xstate = 0;
              if((inp & 0x01) != 0)
                xstate = transO[j] & 0xFF;

              if((inp & 0x01) == 0)
                xstate = transZ[j] & 0xFF;

              if((j == state) && (k == 0))
                {
                  k = 1;
                  state = xstate;
                }
            }

          inp = inp >> 1;
          k = 0;
        }
    }
  output_alice(state);
}
```

Technical Issues

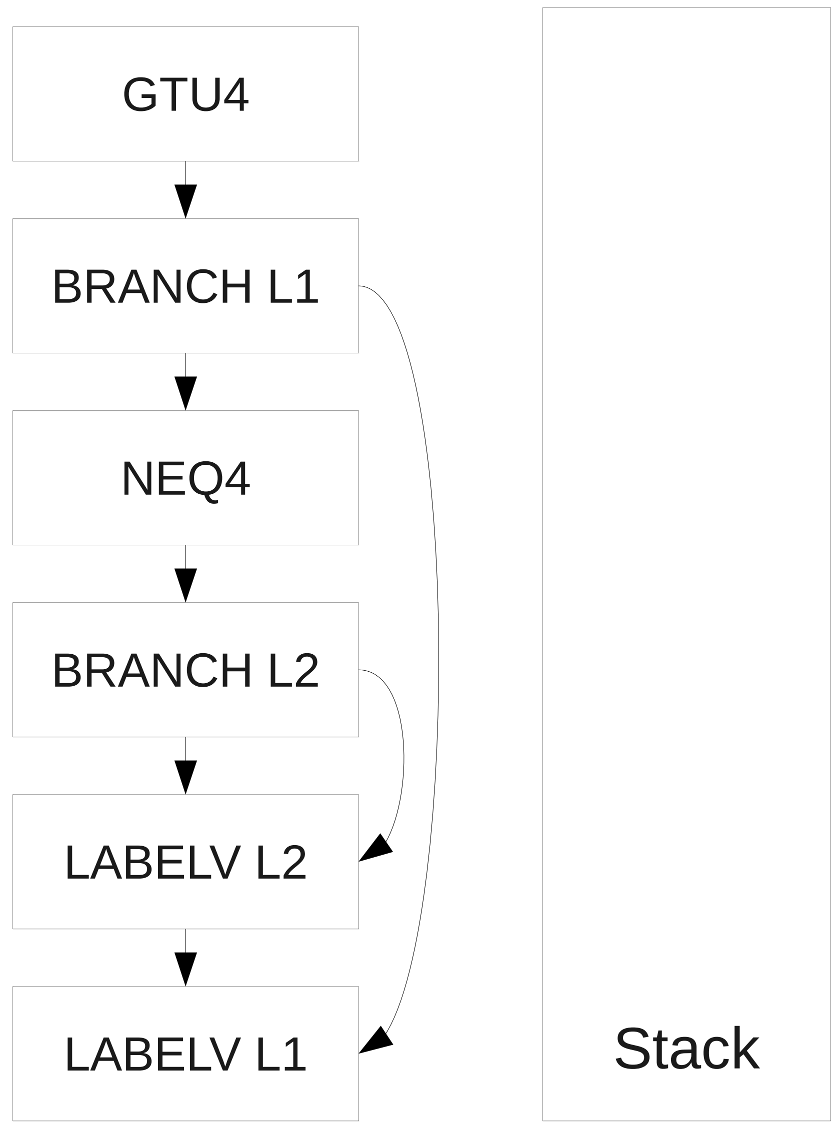
Handling Branches

- Branch statements require special treatment
  - Must evaluate all possible control paths
  - Multiplexers must be used for assignments
- In bytecode formats, branches are not as structured as in HLLs

```
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```

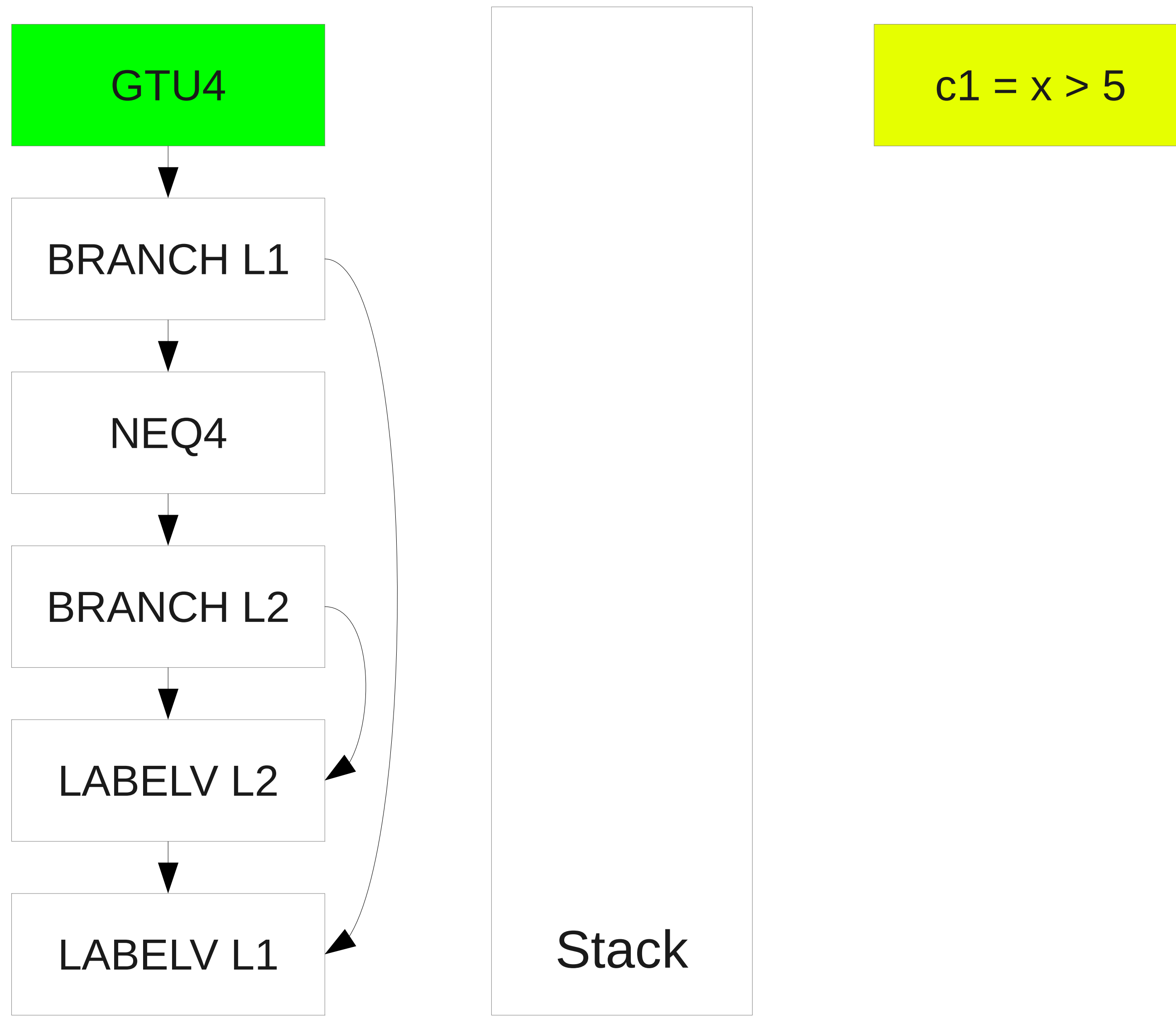
Handling Branches (Fairplay/KSS'12)

```
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```



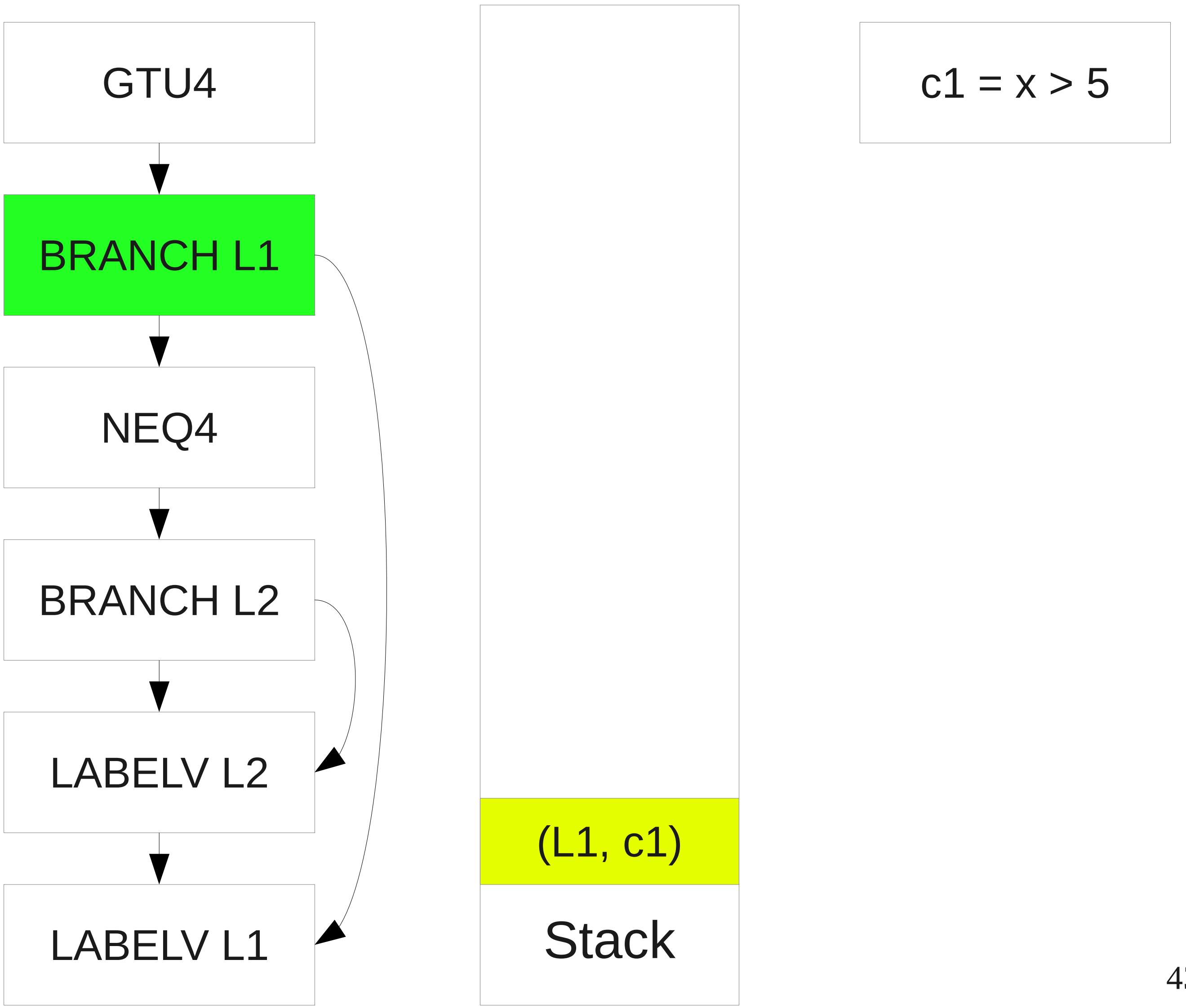
Handling Branches (Fairplay/KSS'12)

```
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```



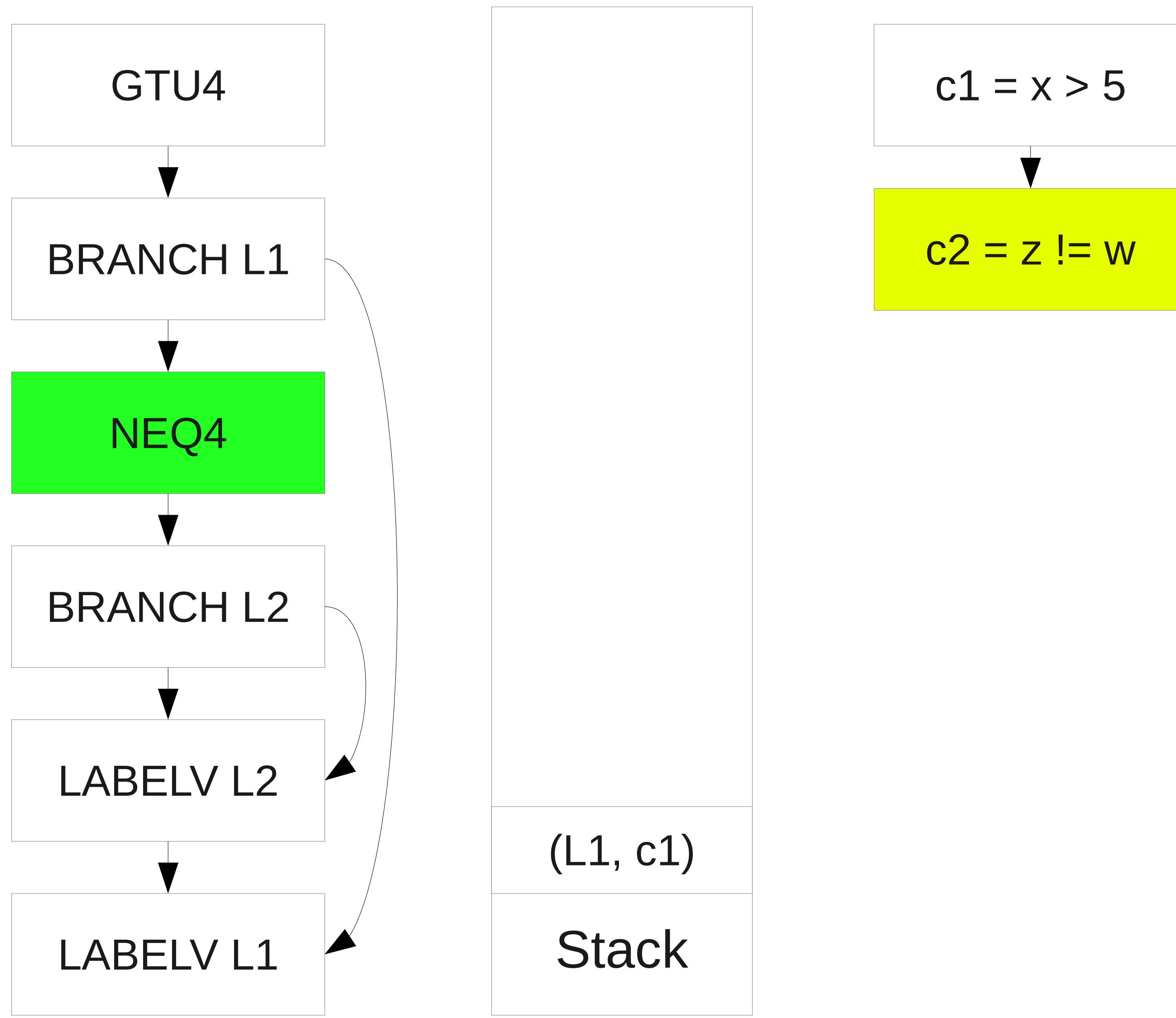
Handling Branches (Fairplay/KSS'12)

```
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```



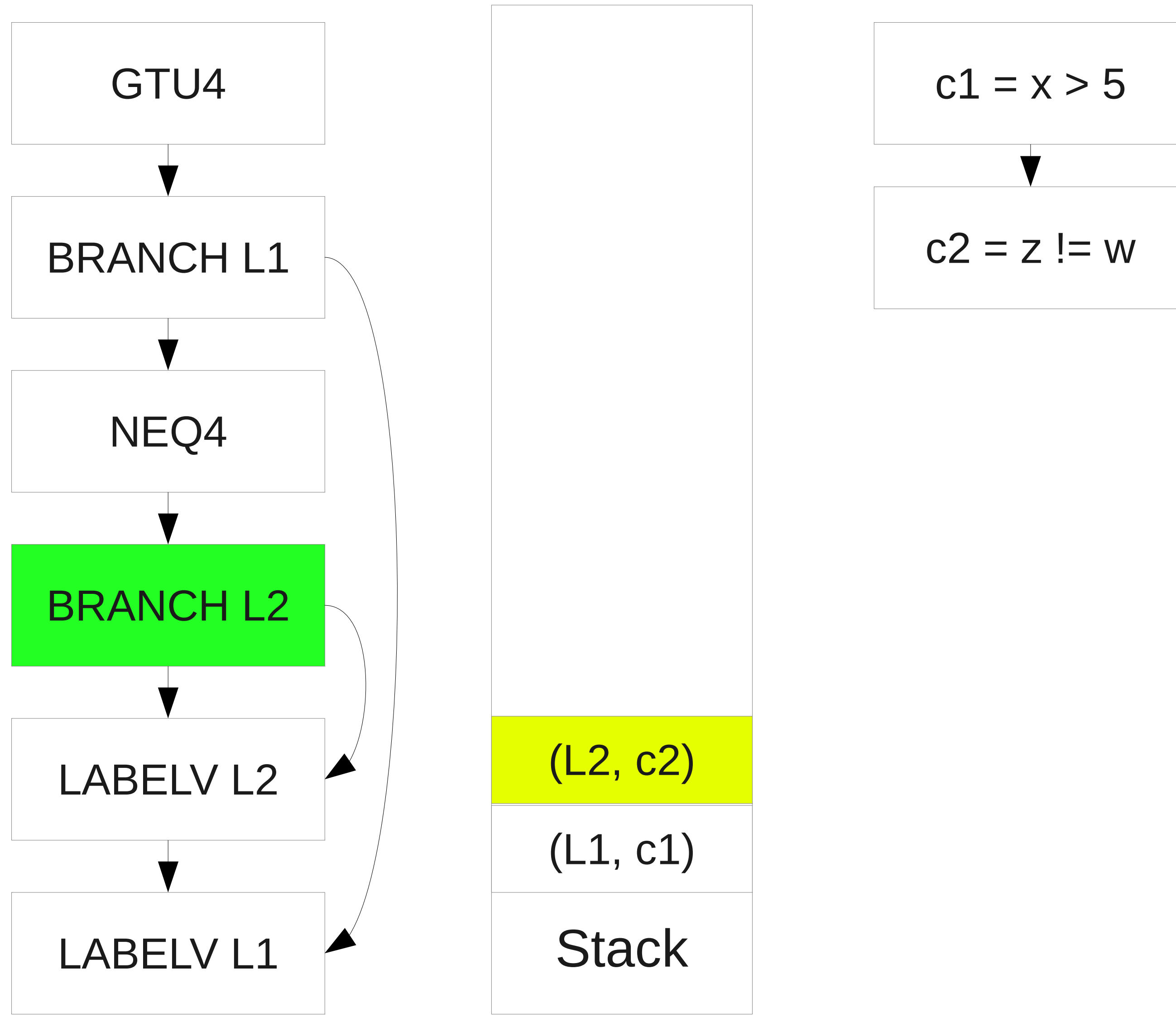
Handling Branches (Fairplay/KSS'12)

```
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```



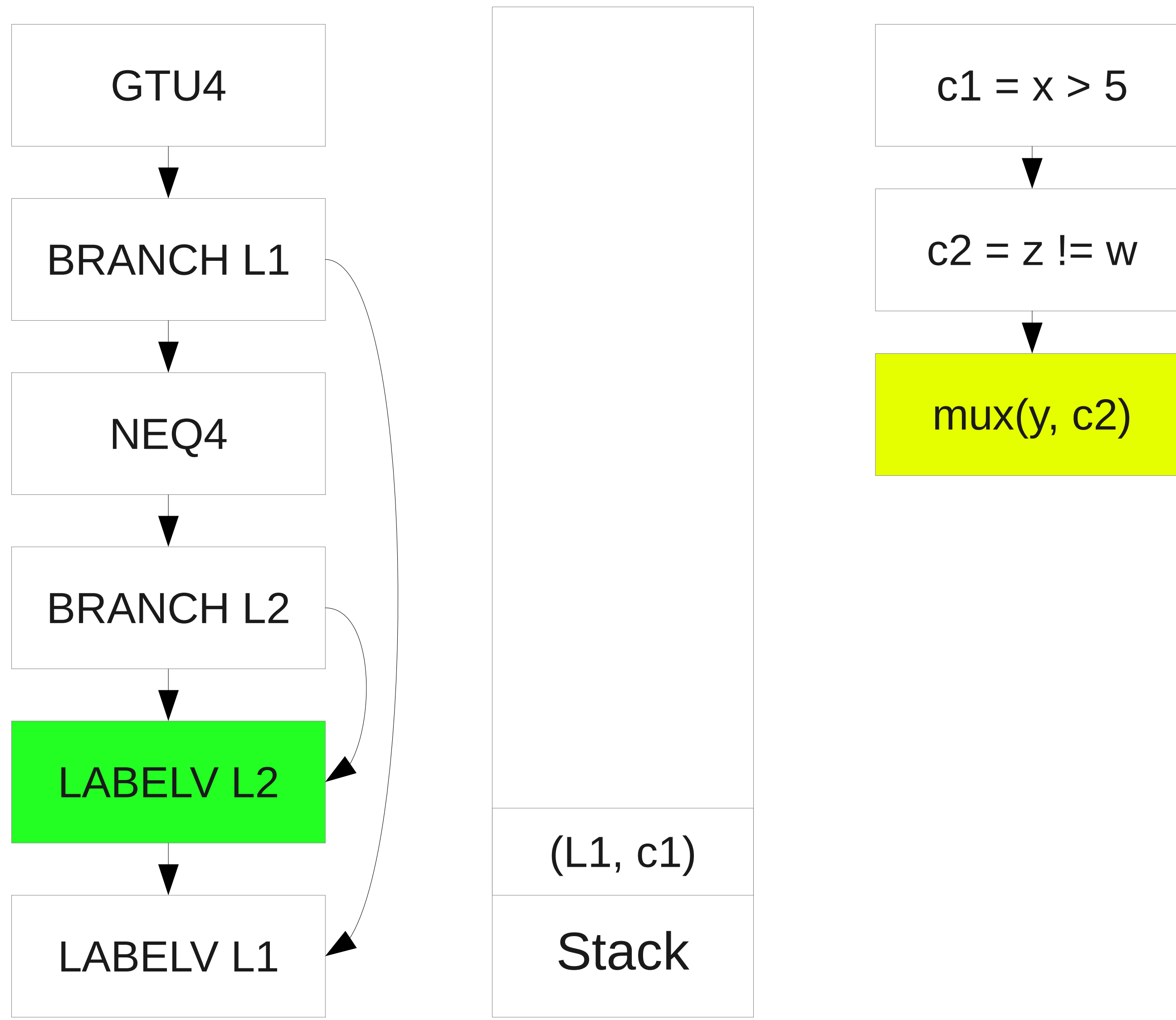
Handling Branches (Fairplay/KSS'12)

```
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```



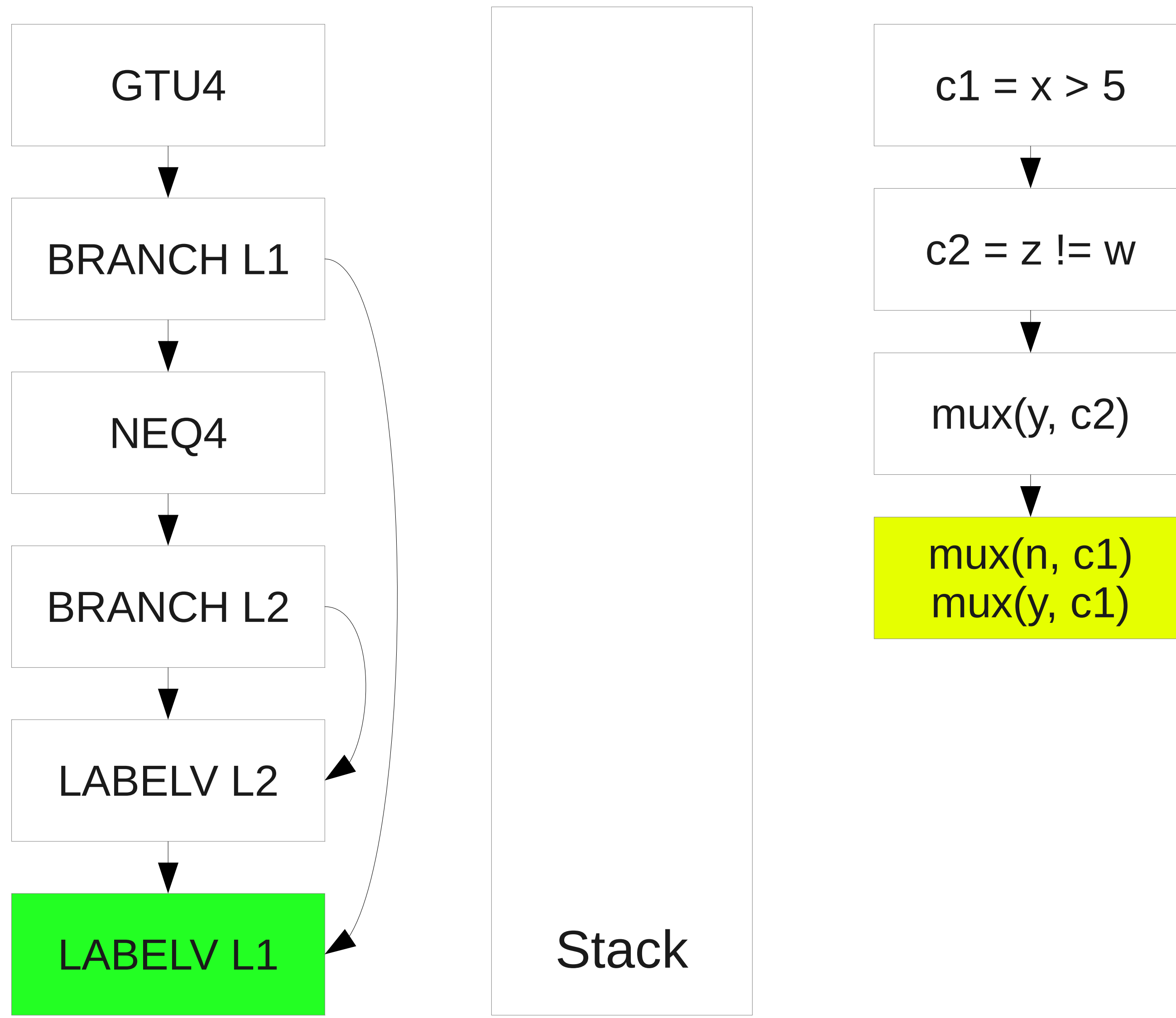
Handling Branches (Fairplay/KSS'12)

```
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```

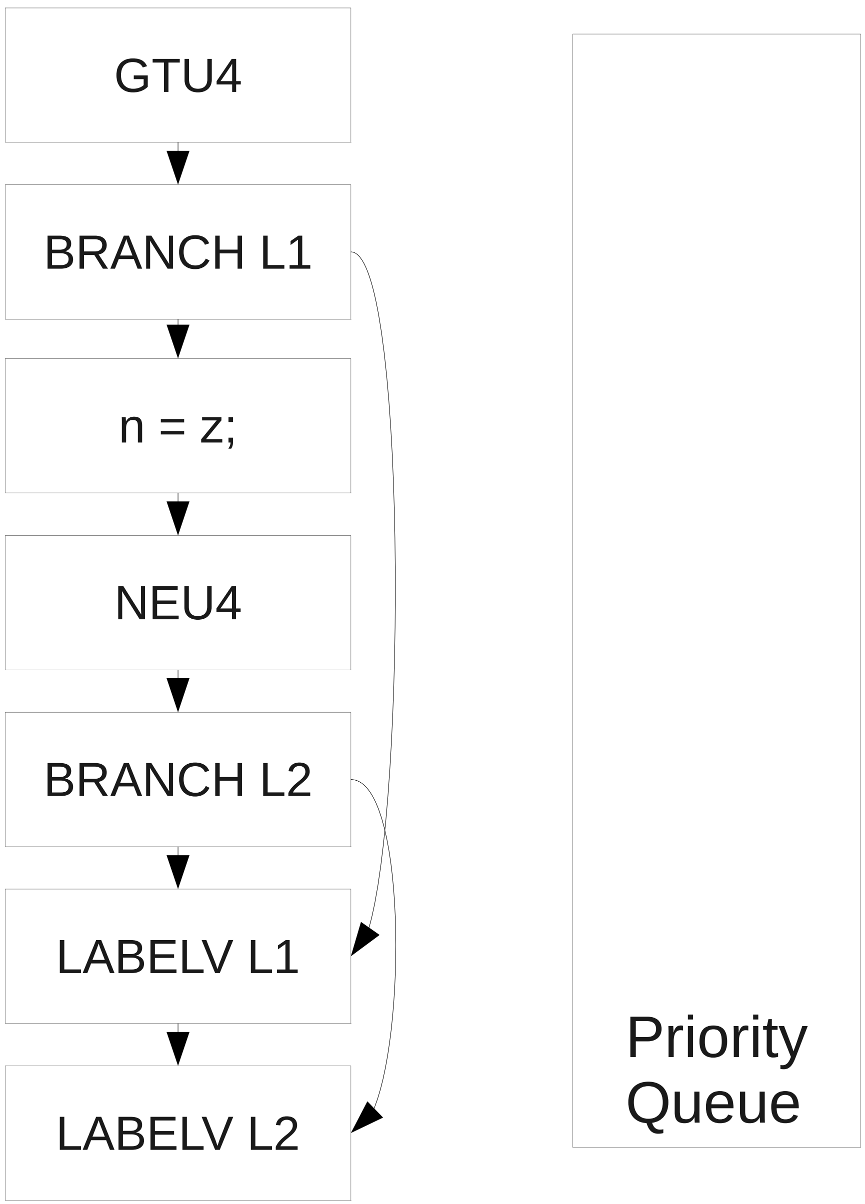


Handling Branches (Fairplay/KSS'12)

```
if(x <= 5) {
    n = z;
    if(z == w) {
        y += n;
    }
}
```



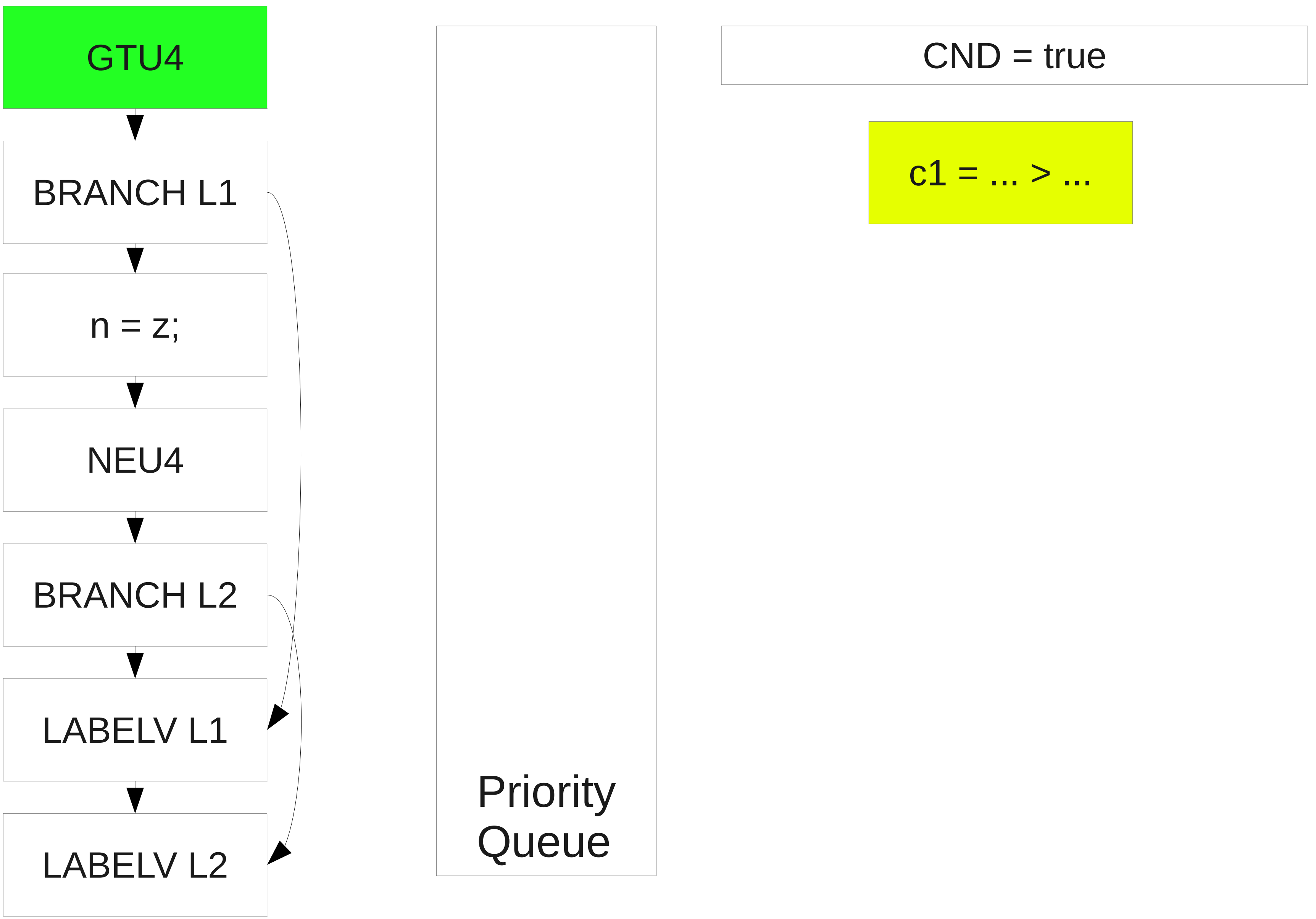
Handling (Complex) Branches



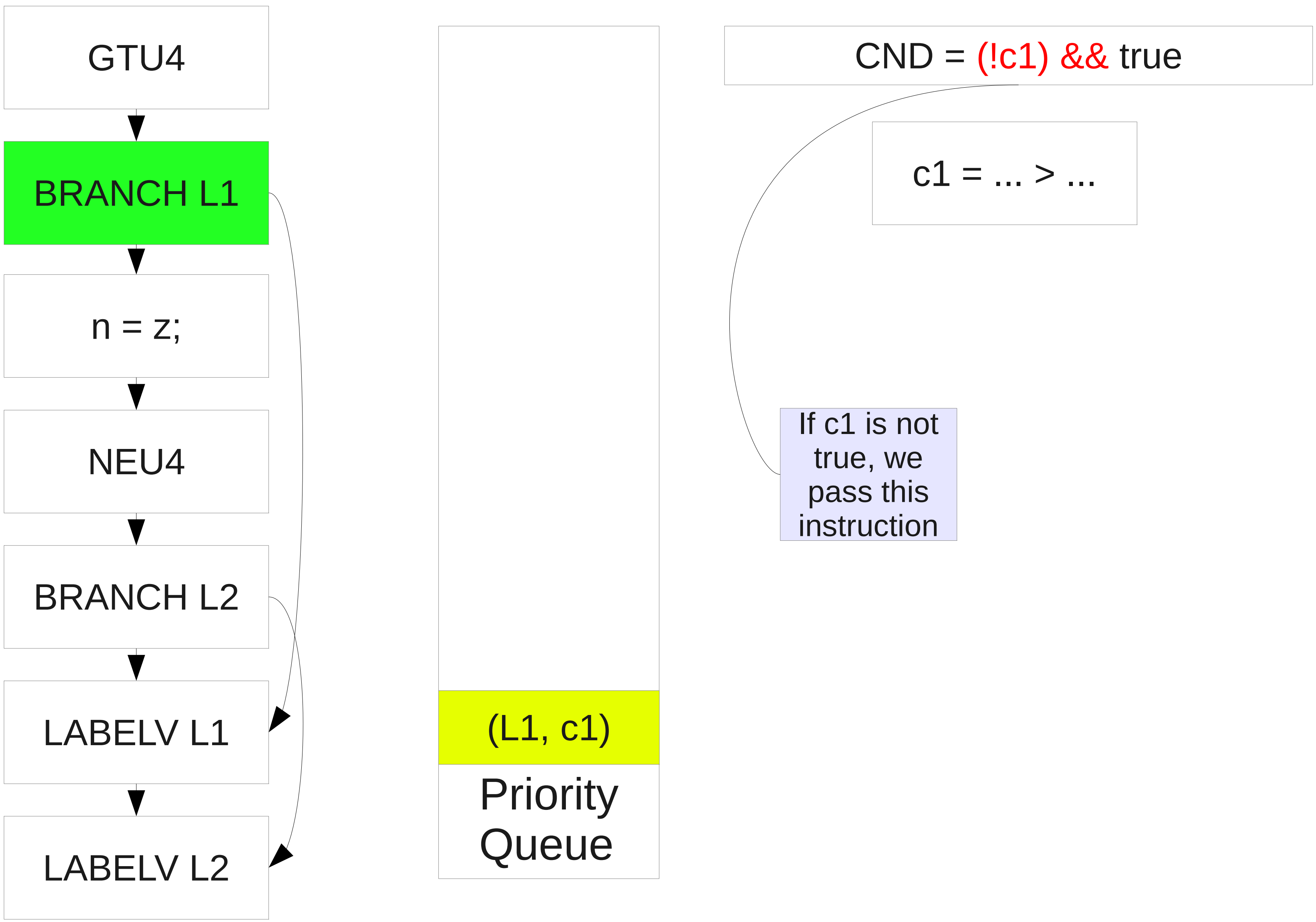
CND = true

Strategy:
- Maintain global condition wire
- Emit muxes for each assignment
- Use priority queue to keep track of next branch target

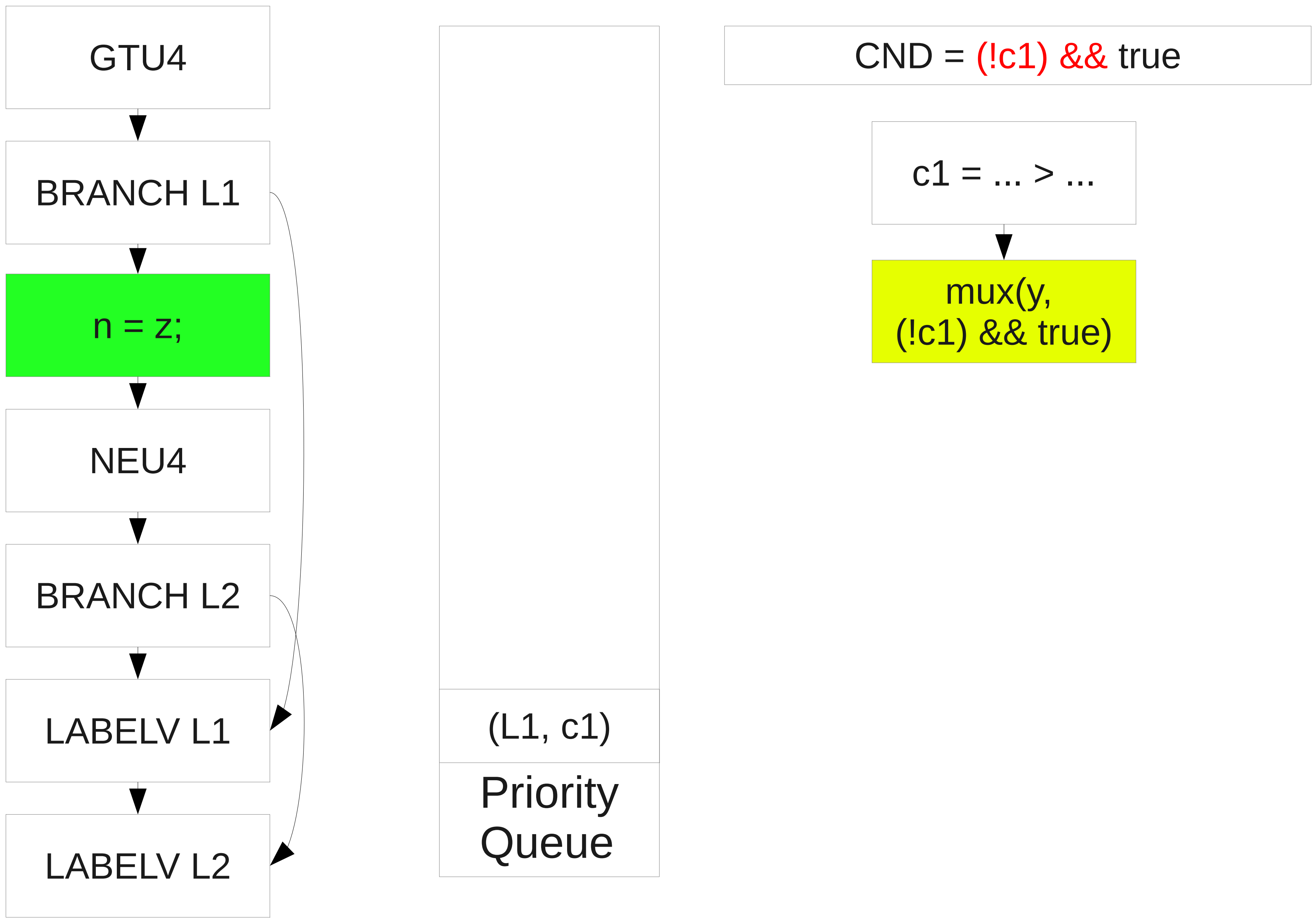
Handling (Complex) Branches



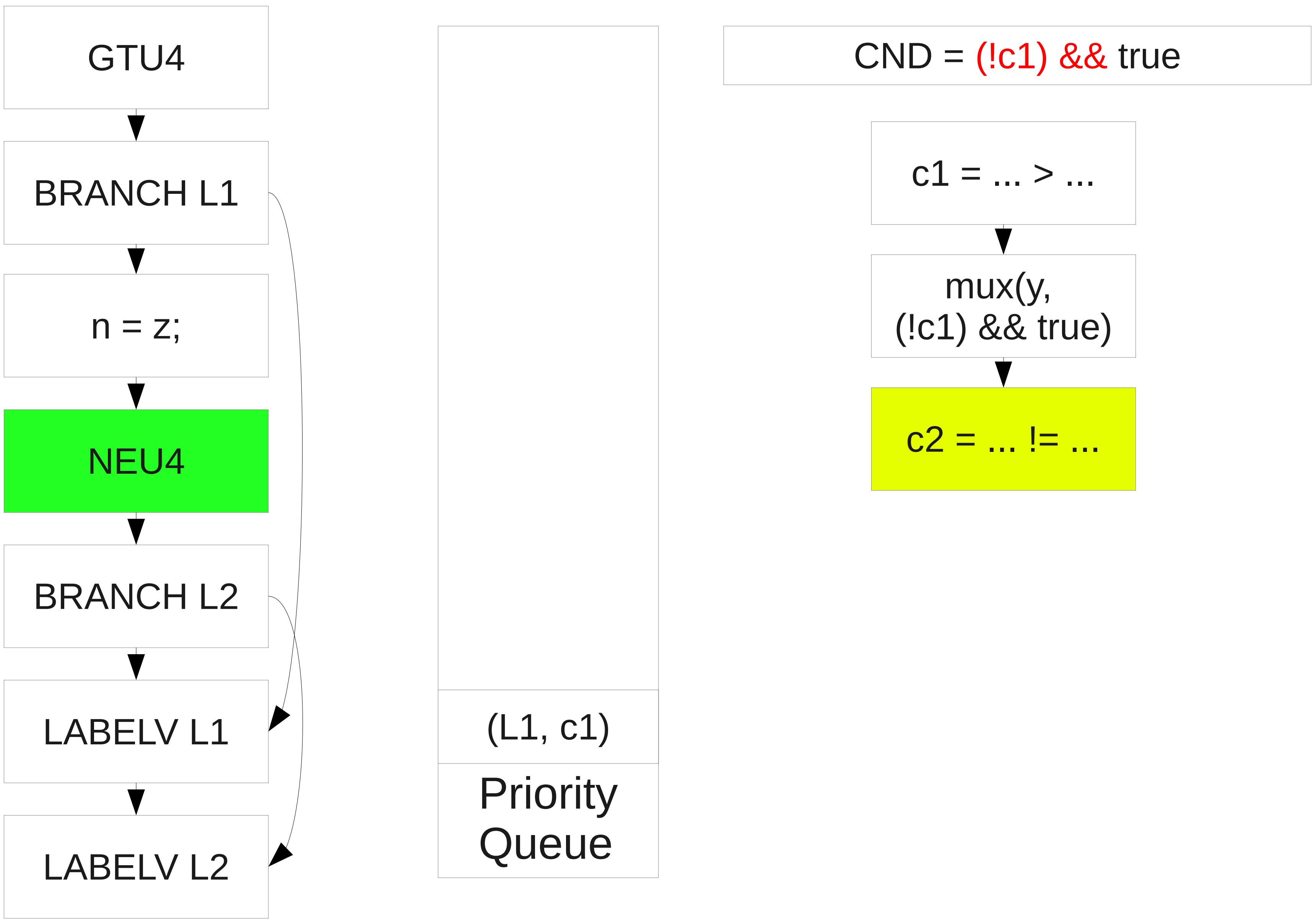
Handling (Complex) Branches



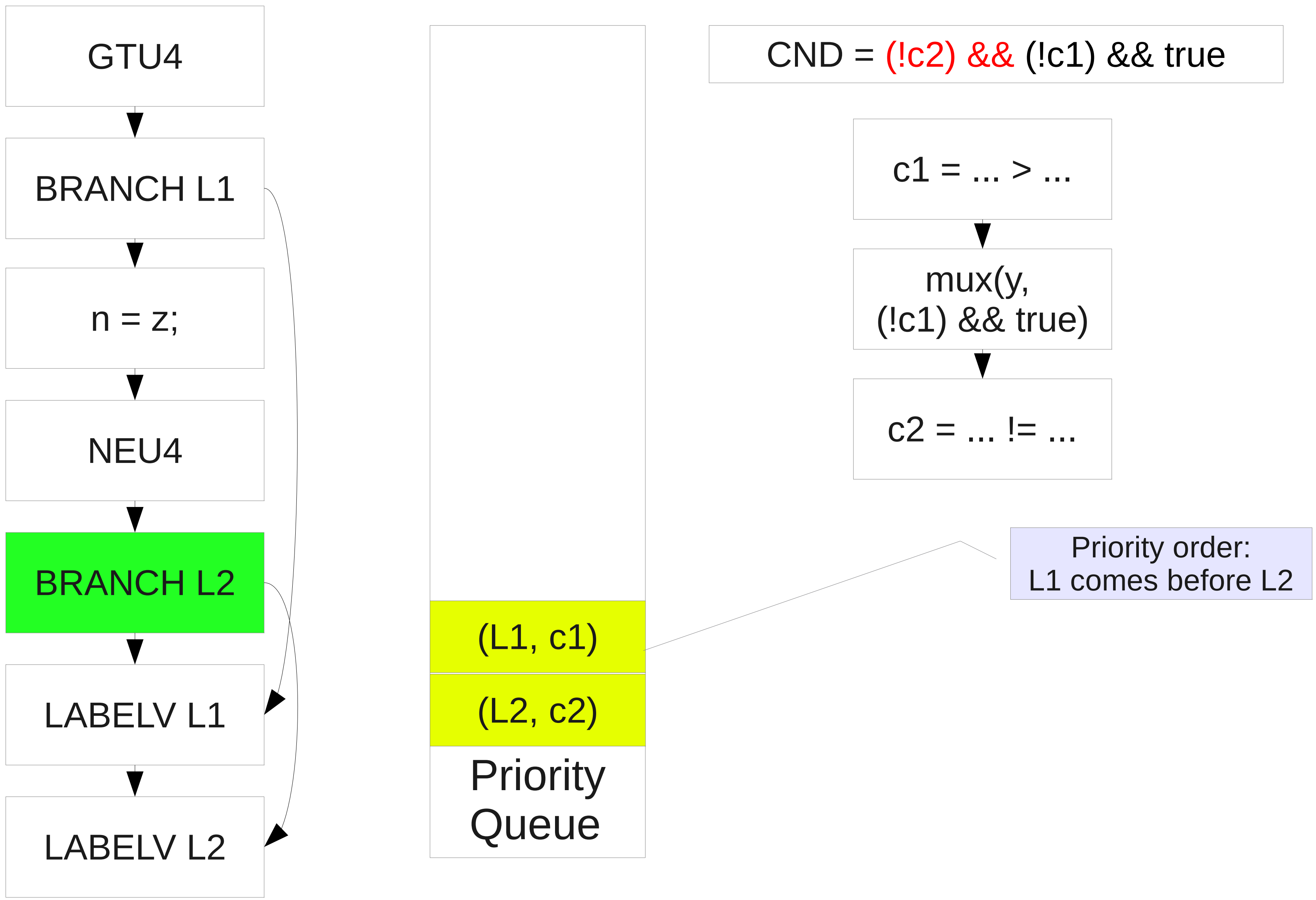
Handling (Complex) Branches



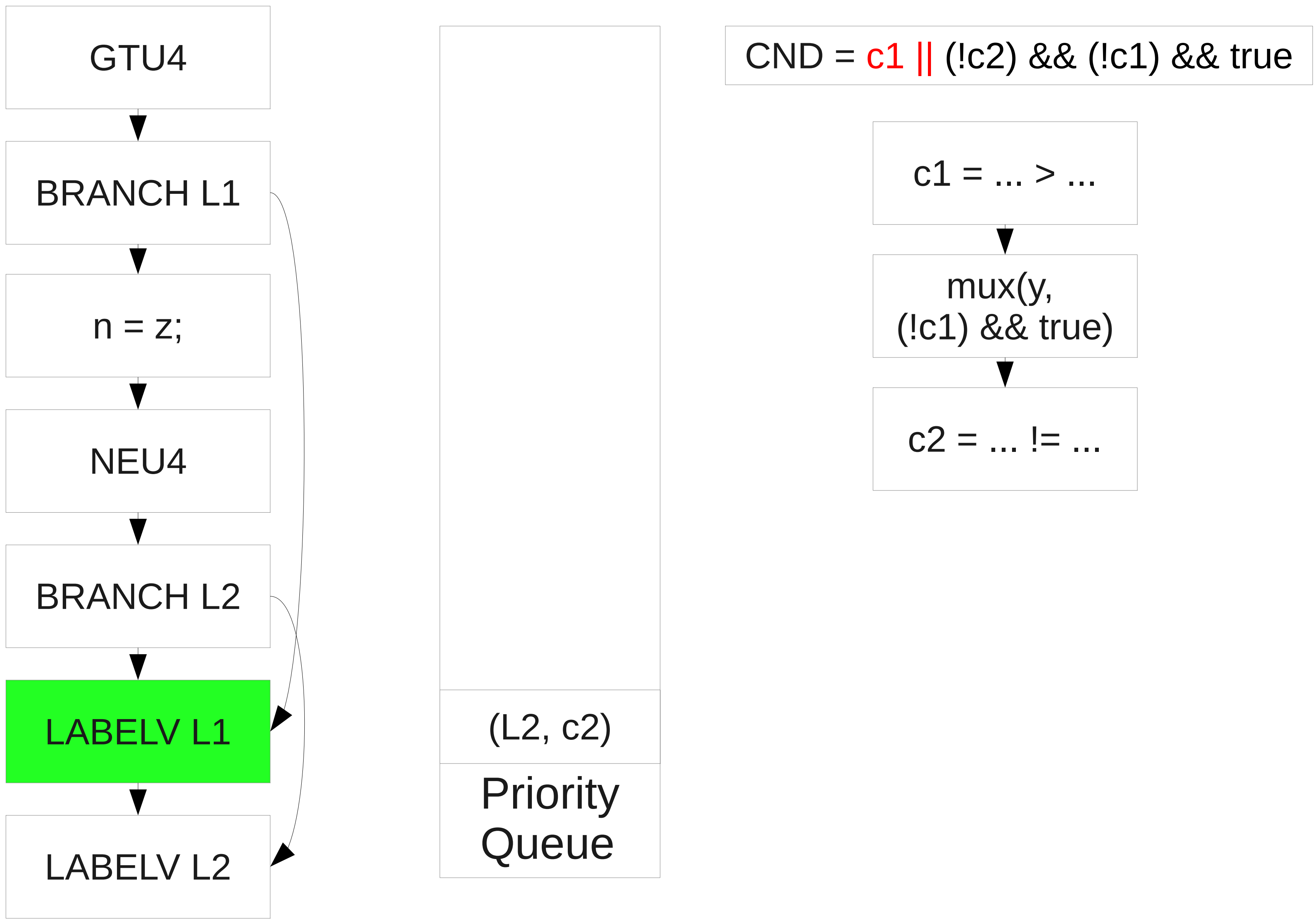
Handling (Complex) Branches



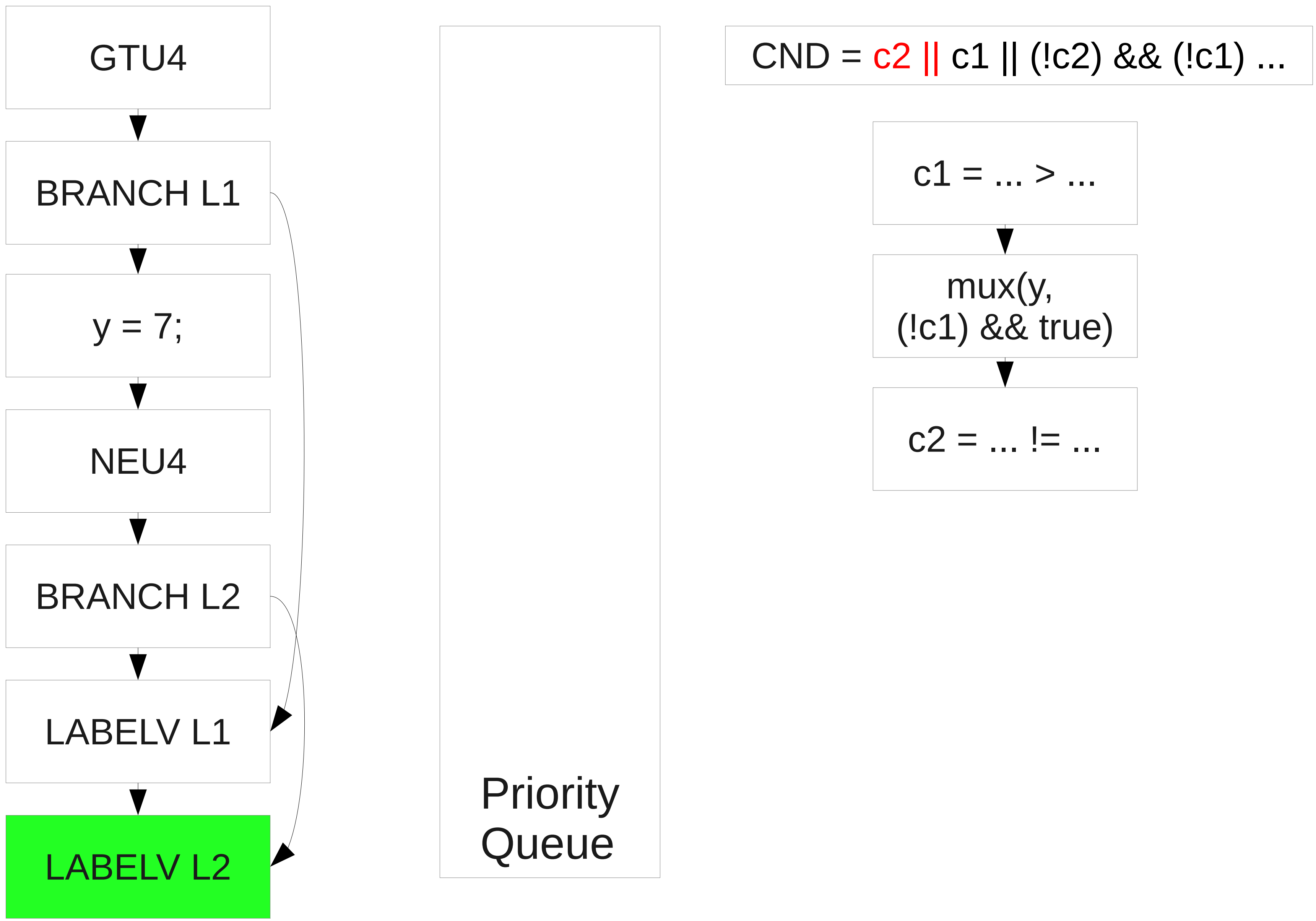
Handling (Complex) Branches



Handling (Complex) Branches



Handling (Complex) Branches



Handling Loops

- Common assumption: backwards branches are used to build loops
- Only one rule: such branches must not be dependent on input values (enforced by runtime system)
- Preventing infinite loops is the user's responsibility

Circuit Optimization Strategy

- Two stages – compile time and run time
- At compile time, use techniques based on dataflow analysis.
  - Circuit sizes are reduced indirectly by reducing program run time
- At run time, check gates for constant outputs

Circuit Optimization Strategy

Function	KSS12		HFKV12		This Work	
	Total	Non-XOR	Total	Non-XOR	Total	Non-XOR
16,384-bit Millionaire's	98,303	49,154	330,784	131,103	97,733	32,229
32-bit Multiplication	15,935	5,983	65,121	26,624	21,742	6,517
64-bit Multiplication	64,639	24,384	321,665	126,529	105,880	24,766
8x8 Matrix Multiplication	8,067,458	3,058,754	3,267,585	907,776	1,782,656	522,304
16x16 Matrix Multiplication	64,570,969	24,502,530	24,140,673	7,262,208	14,308,864	4,186,368

File Sizes and Compile Times

Function	KSS12		HFKV12		PCF	
	Circuit Size	Compile Time (s)	Circuit Size	Compile Time (s)	Circuit Size	Compile Time (s)
16384-bit Millionaire's	1.9MB	4.66	3.0MB	105.	98kB	3.40
16000-bit Hamming Distance	1.9MB	9.75	9.0MB	309.	130kB	10.8
1024-bit Multiplication	112MB	430.	??	??	494kB	74.0
16x16 Matrix Multiplication	432MB	2,200	206MB	2,600	528kB	109.
256-bit RSA	15GB	24,000	-	-	1.2MB	109.
1024-bit RSA	??	??	-	-	1.3MB	564.

~1000x improvement

Comparison with “circuit libraries”

	Circuit Libraries [HEKM'11, MAL'11]	**PCF**
Scalability	Good – Circuit not stored anywhere	Good – Circuit is compressed
Building Circuits	Ad-hoc – separate gadgets composed by user	Automatic – gadgets composed automatically by compiler
Optimization	Per-gadget, user can be clever	Automatic, can cross gadget boundaries

Comparison with “circuit libraries”

- Our approach *subsumes* circuit libraries
- New gadgets can be added for new bytecode instructions

Using PCF

- A library for interpreting PCF files
- Simple interface – two functions
- Compiler and library are available upon request, and posted to github shortly

We are happy to help integrate PCF into your secure computation project

Conclusion

We have scalable <u>protocols</u> for secure 2-party computation

and...

We have scalable <u>tools</u> for secure 2-party computation

Future Work

- Other settings
  - Verifiable computation – arithmetic circuits / QAPs
  - FHE – arithmetic circuits + SIMD
  - Multiparty computation (more than 2 parties)
- Other computation models
- New optimization techniques

Questions?