Evolving Assemblers Using Disassembler Oracles

May 18th, 2018

Andrew Lamoureux

JAILBREAK BREWING COMPANY

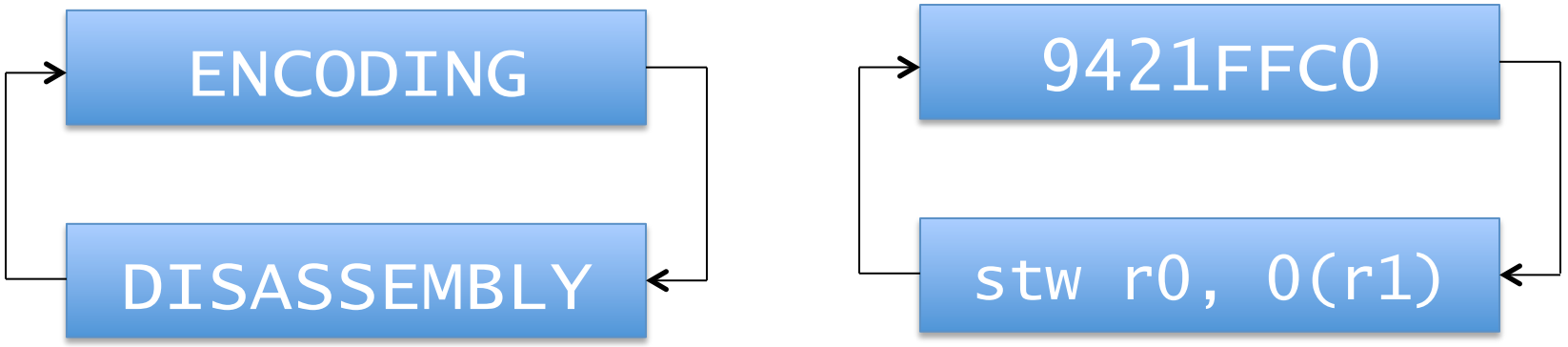
SECURITY SUMMIT

VECTOR 35

BINARYNINJA

Motivation: Binary Ninja Development

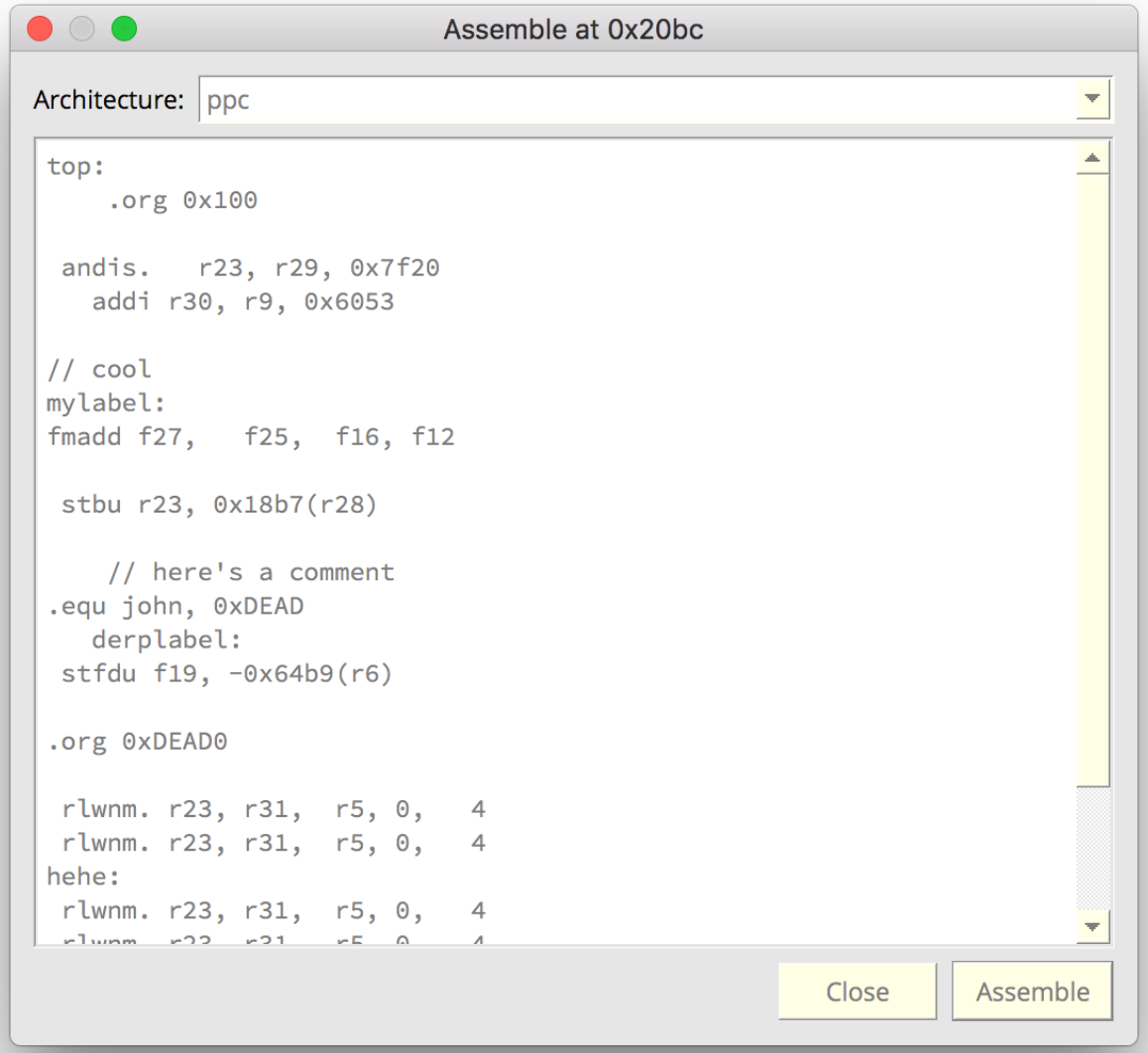
- “Round Tripping”




```
_start:
000020a8  7c3a0b78    mr       r26, r1 {arg7}
000020ac  3821fffc    addi     r1, r1, -0x4
000020b0  54210034    rlwinm   r1, r1, 0x0, 0x0, 0x1a
000020b4  38000000    li       r0, 0x0
000020b8  90010000    stw      r0, 0(r1) {var_20}
   stwu     r1, -64(r1)
000020c0  807a0000    lwz      r3, 0(r26) {arg7}
000020c4  389a0004    addi     r4, r26, 0x4 {arg_4}
000020c8  3b630001    addi     r27, r3, 0x1
000020cc  577b103a    slwi     r27, r27, 0x2
000020d0  7ca4da14    add      r5, r4, r27 {arg_4}
000020d4  48000009    bl       sub_20dc
000020d8  7fe00008    trap
```

Motivation: Binary Ninja Development

- Proper .org Support



Traditional Assembler

ASSEMBLER

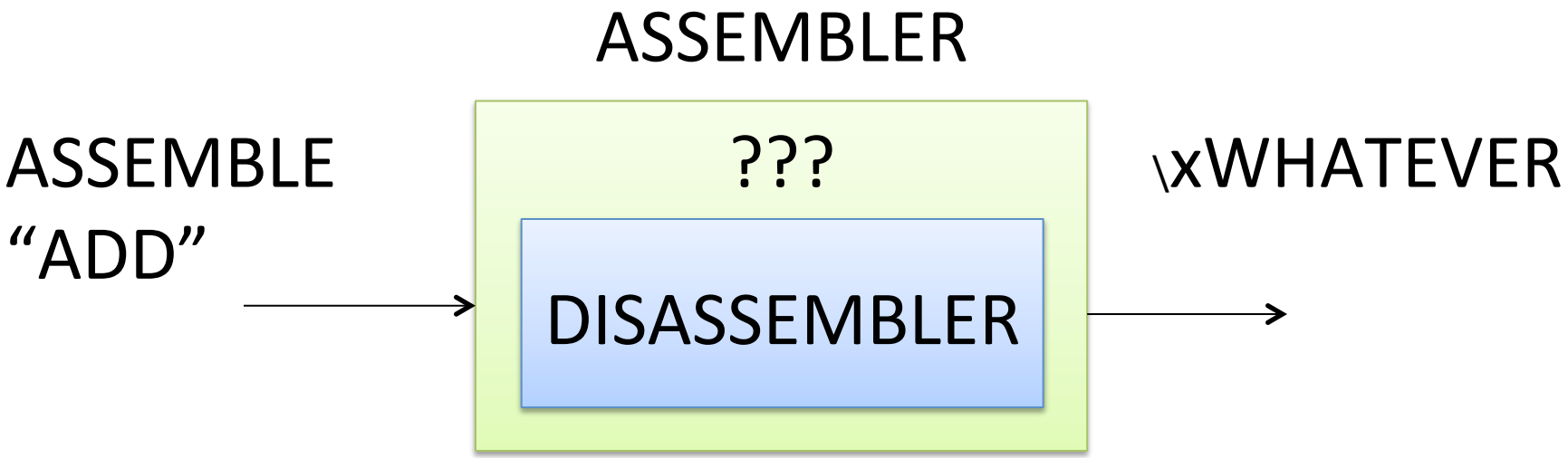
ASSEMBLE
“ADD”

```
case PUSH: …
case POP: …
case ADD: …
case SUB: …
```

\xWHATEVER

Minor tasks: parse directives, db statements, labels, obj files − major task: instruction encoding − what if it’s YOUR JOB?

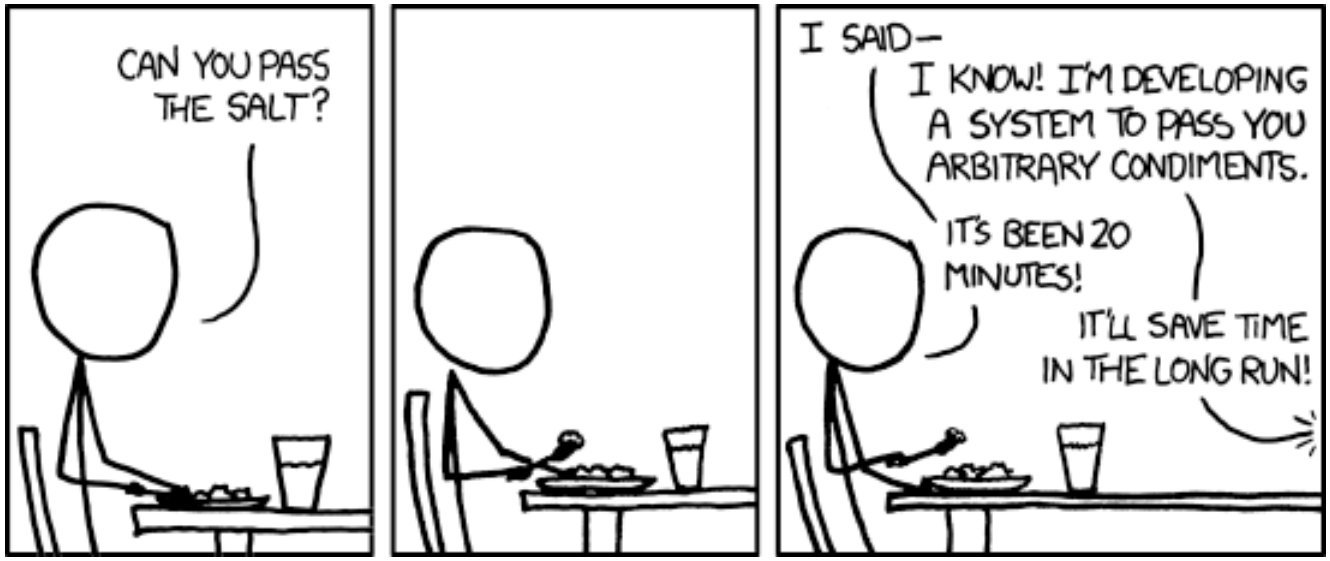
Oracle-Based Assembler (OBA)



Born from laziness, trying to avoid the 2week sentence of PDF reading

OBA Questions

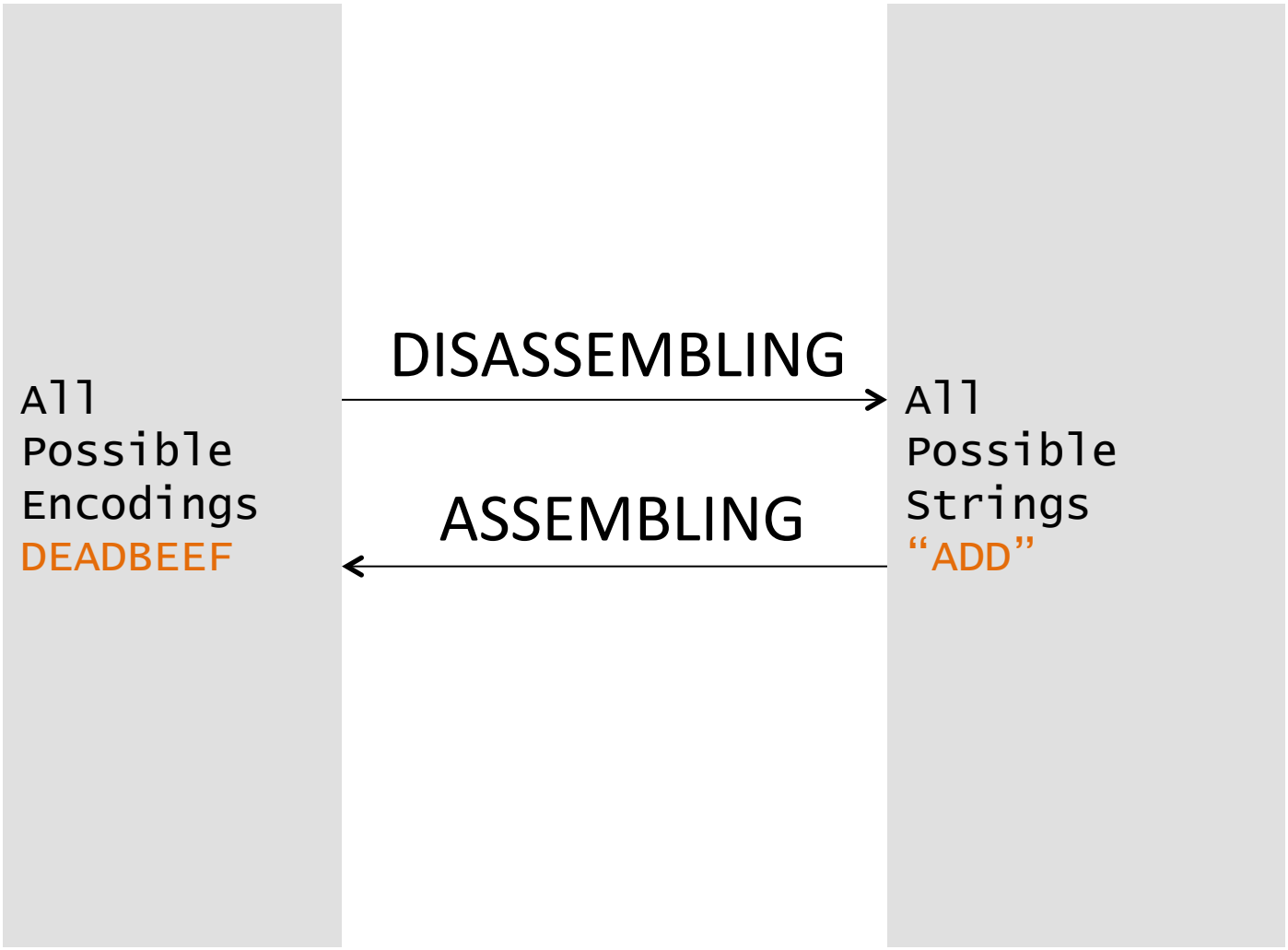
1. Is time actually saved?



https://xkcd.com/974/

2. Does the approach fulfill Binary Ninja's needs?

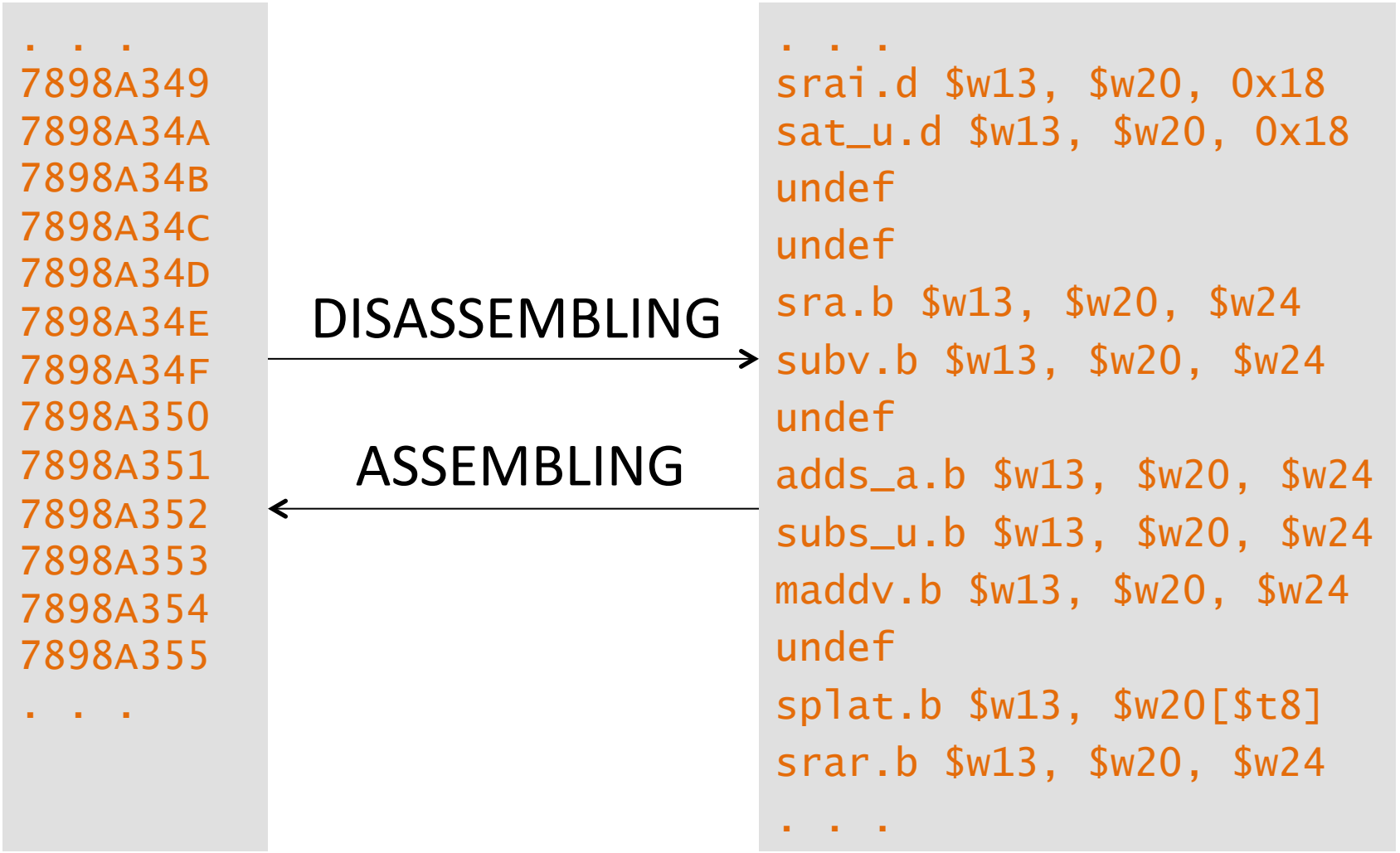
Lookup Table (LUT) View




```
disassemble(): uint32_t -> string
   assemble(): string -> uint32_t
```

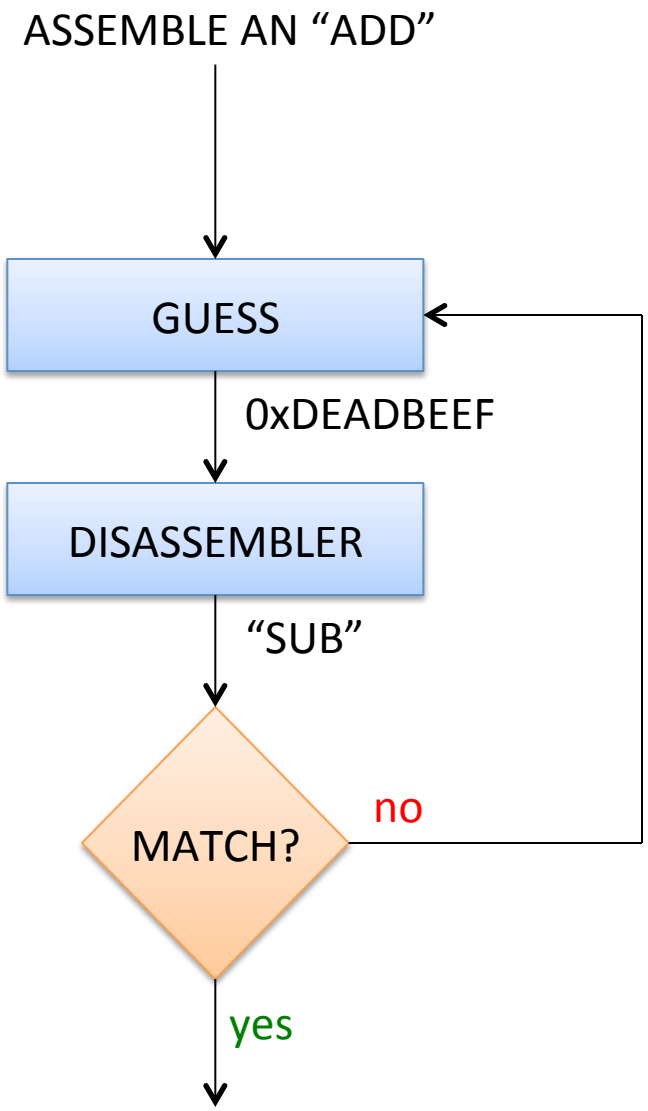
Started thinking in terms of a table – a disassembler compresses this table – gut feeling of inversion – cryptography says

Lookup Table (LUT) View



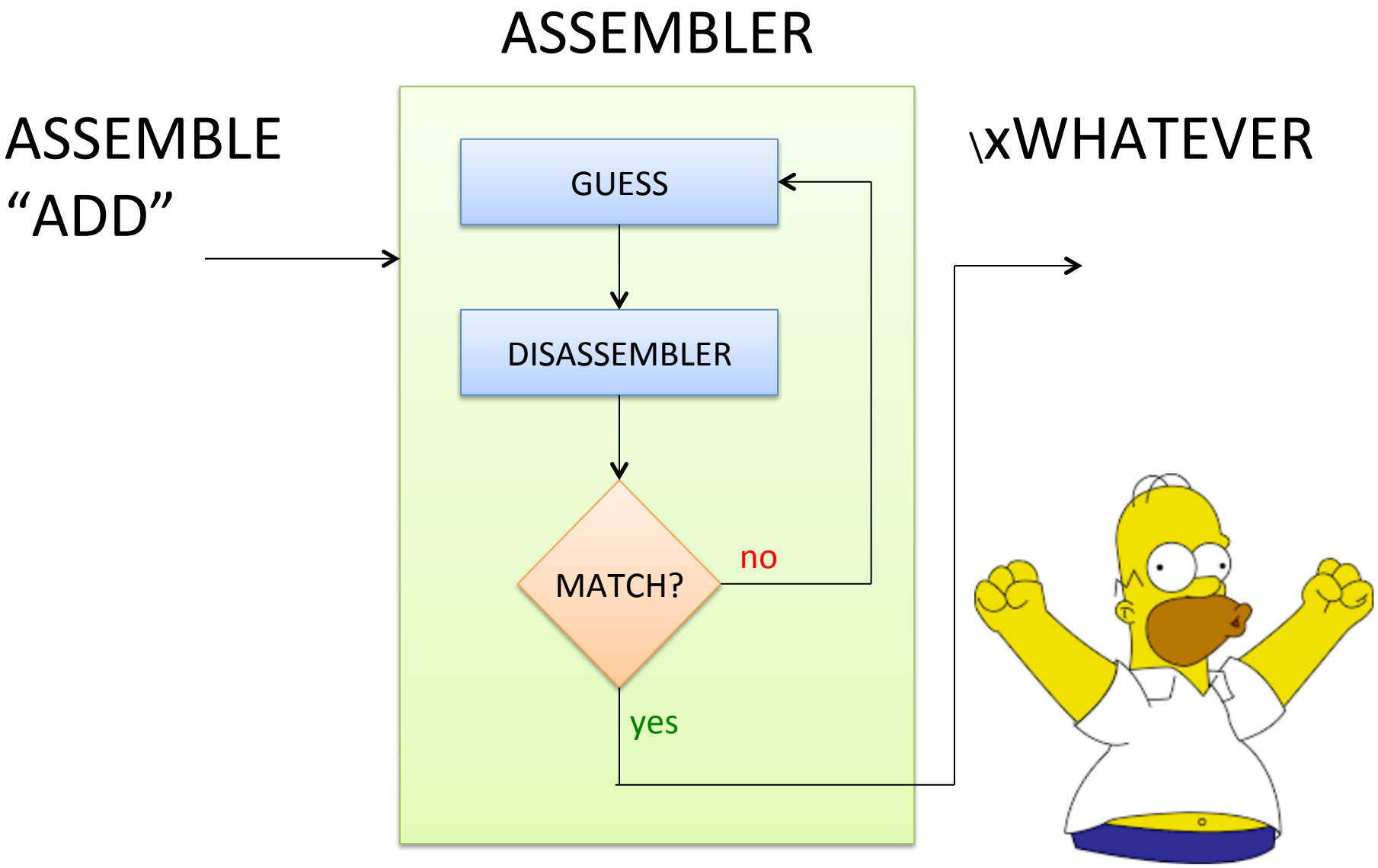
If we could see the entire table at one time, problem is solved – but instead we have a “flashlight”

Spoiler: Guess and Check



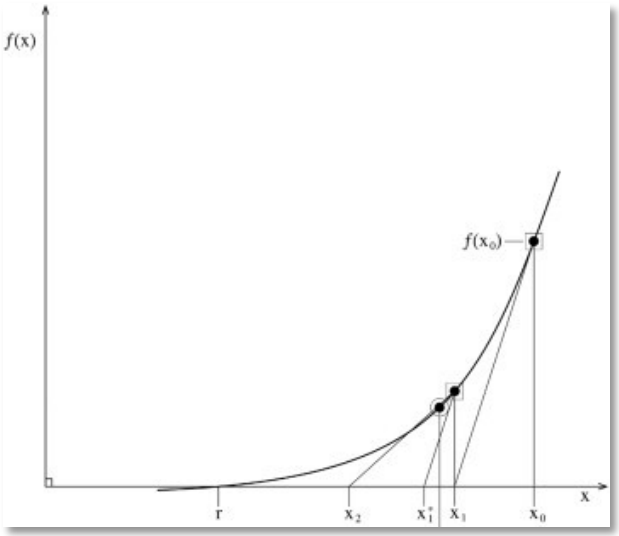
1. Guess
2. Check
3. Modify

Oracle-Based Assembler



If we substitute the “Guess, Check, Modify” into the diagram with Homer in it, here’s what we get…

Guess’n’Check In Different Fields



- no matter, just need actionable feedback

“Guess’n’Check” doesn’t sound very elegant, but it’s found in tons of different fields

Problems with Vanilla Genetic Algorithm

1. Where to start?
2. What fitness function?
3. How to converge?

The remainder of this talk solves these problems.

The “isolated opcode problem”

Instruction Space as Number Line

```
7898A349 srai.d    $w13, $w20, 0x18
7898A34A sat_u.d   $w13, $w20, 0x18
7898A34B undef
7898A34C undef
7898A34D sra.b     $w13, $w20, $w24
7898A34E subv.b    $w13, $w20, $w24
7898A34F undef
7898A350 adds_a.b  $w13, $w20, $w24
7898A351 subs_u.b  $w13, $w20, $w24
7898A352 maddv.b   $w13, $w20, $w24
7898A353 undef
7898A354 splat.b   $w13, $w20[$t8]
7898A355 srar.b    $w13, $w20, $w24
```

…but this is a very human view

also try 0x11223344

Choosing a Disassembler

Capstone Engine!

```
from capstone import *
md = Cs(CS_ARCH_PPC, 0)
for I in md.disasm("\xDE\xAD\xBe\xEF", 0):
   print i.mnemonic + ' ' + i.op_str
```

Pros/Cons Capstone

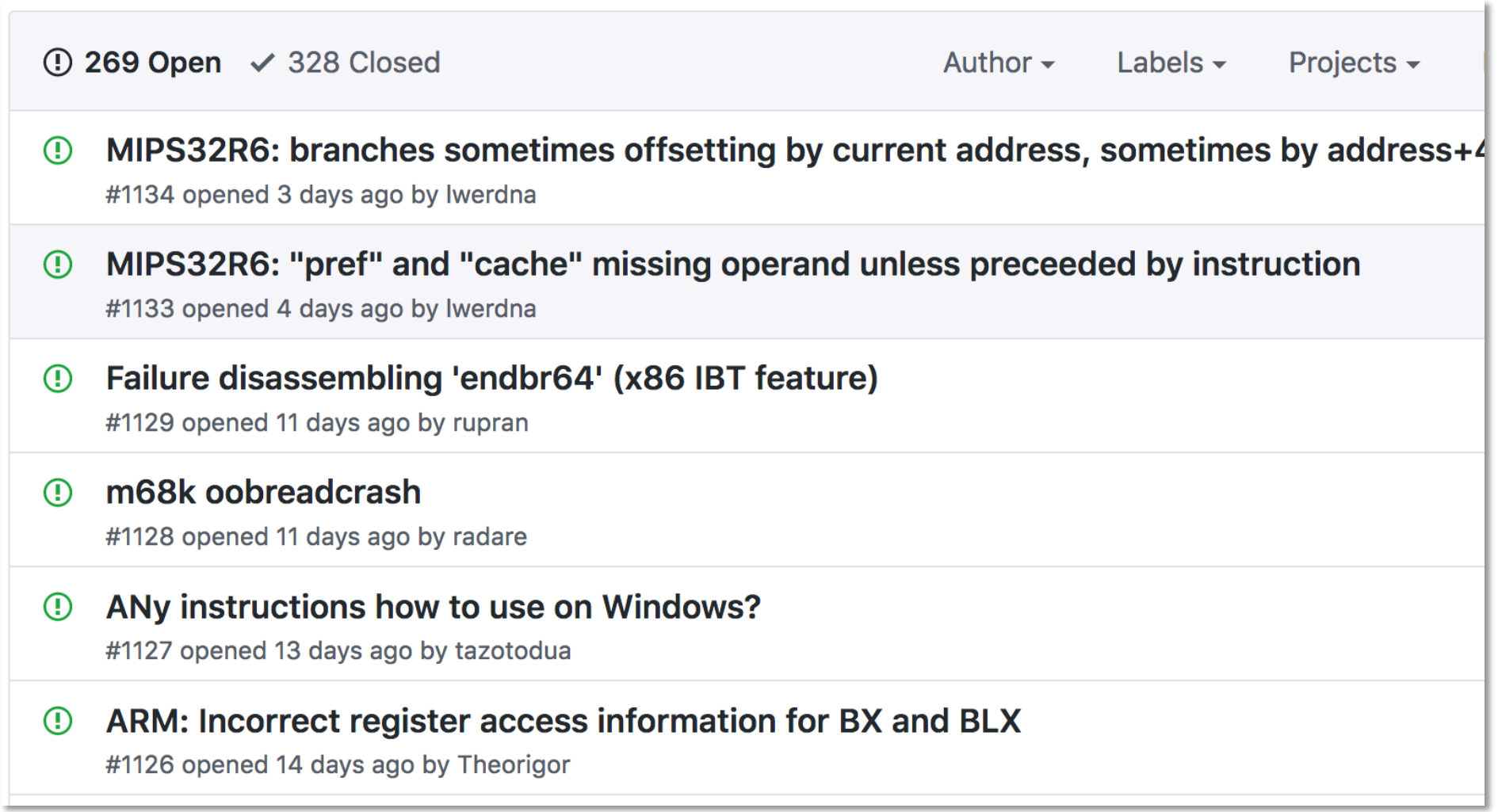
Pros

- Free: BSD License
- Multiple Architectures
- Easy to Install (brew, pip, apt-get)
- Widespread Adoption

Cons

- Not performance oriented
- The cs_detail isn’t always accurate
- Some bugs

Capstone Ain't Perfect



Capstone Speed

How long to go thru instruction space? About 40 minutes

[Timing Tests]

Capstone
The Ultimate Disassembler

Capstone Speed

file	method	time	rate
method0.cpp	binary cs_disasm()	01.2s	1,700,000 instrs/sec
method1.cpp	binary cs_disasm_iter()	01.0s	2,000,000 instrs/sec
method0.py	ctypes to gofer.cpp	03.3s	630,000 instrs/sec
method1.py	bindings	19.3s	100,000 instrs/sec

How long to go thru instruction space? About 40 minutes

[Timing Tests]

Capstone
The Ultimate Disassembler

Exploring The Number Line

```
examples = {}
count = {}

for insword in xrange(0,2**32):

   opcode = disassemble(insword).split()[0]

   if not opcode in examples:
      examples[opcode] = insword
      count[opcode] = 1
   else:
      count[opcode] += 1
```

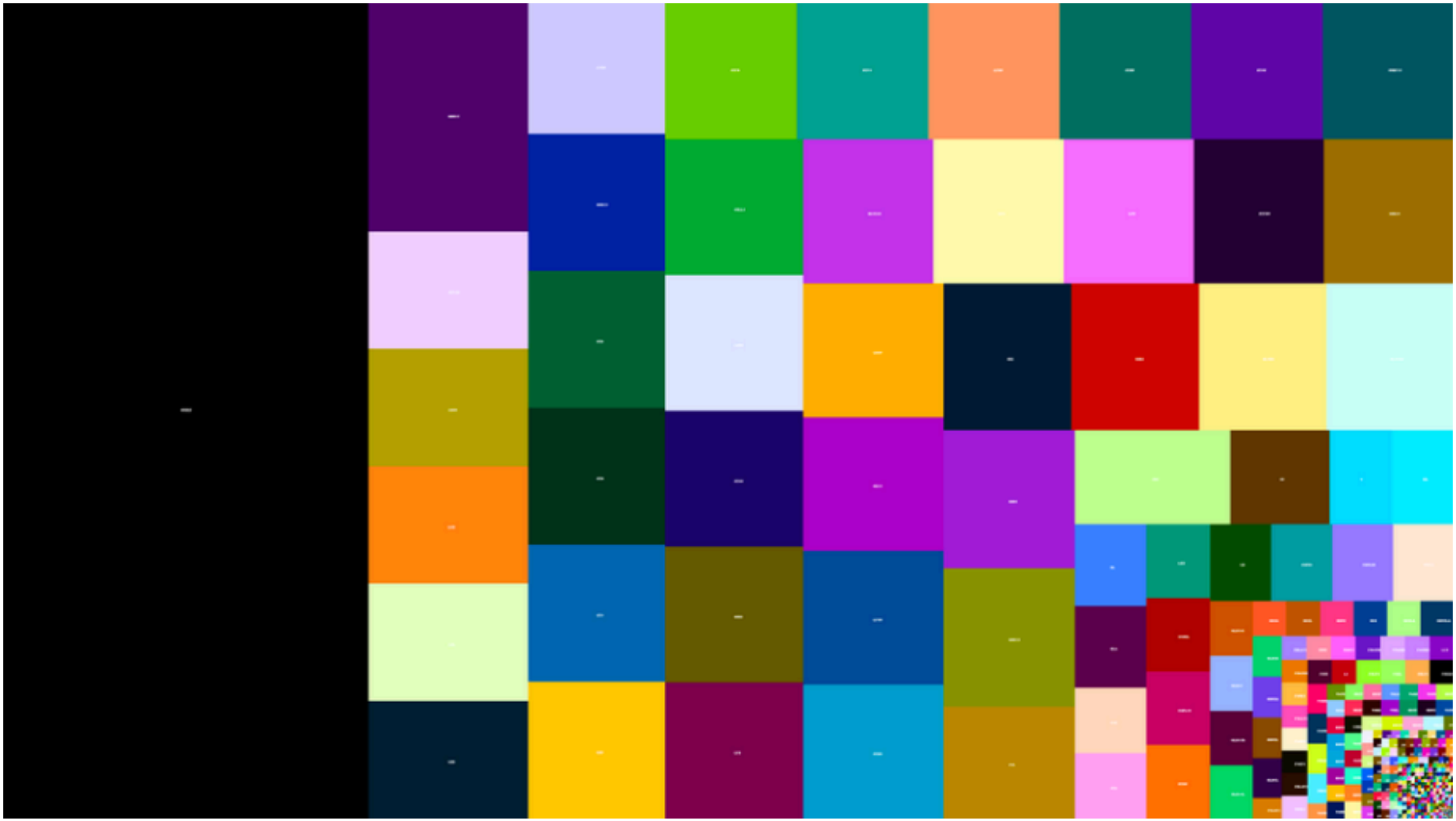
show survey_opcs.py, opc_counts.txt, opc_examples.txt

Problems with Vanilla Genetic Algorithm

1. ~~Where to start?~~
2. What fitness function?
3. How to converge?

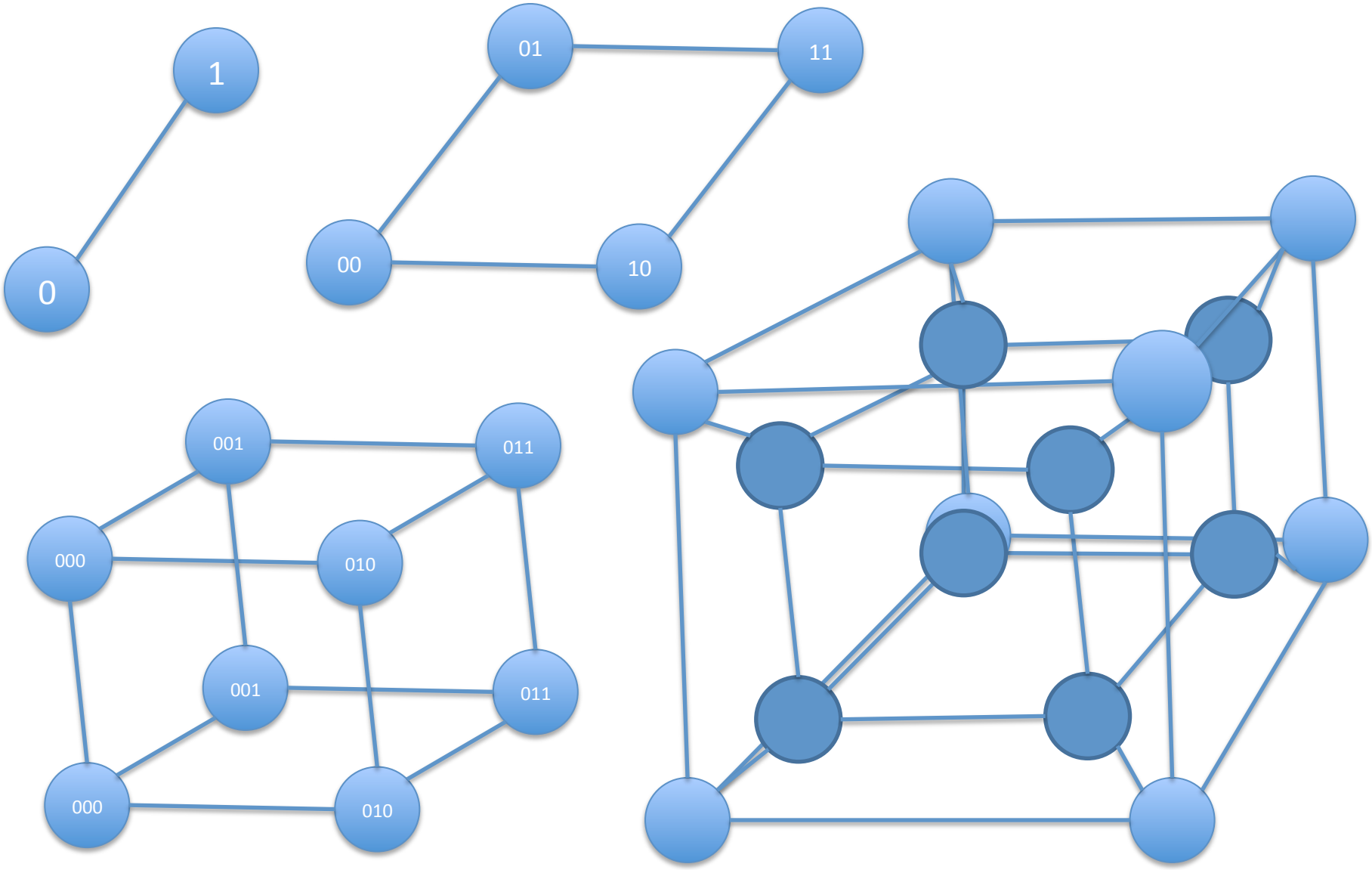
The remainder of this talk solves these problems.

Instruction Space as a TreeMap



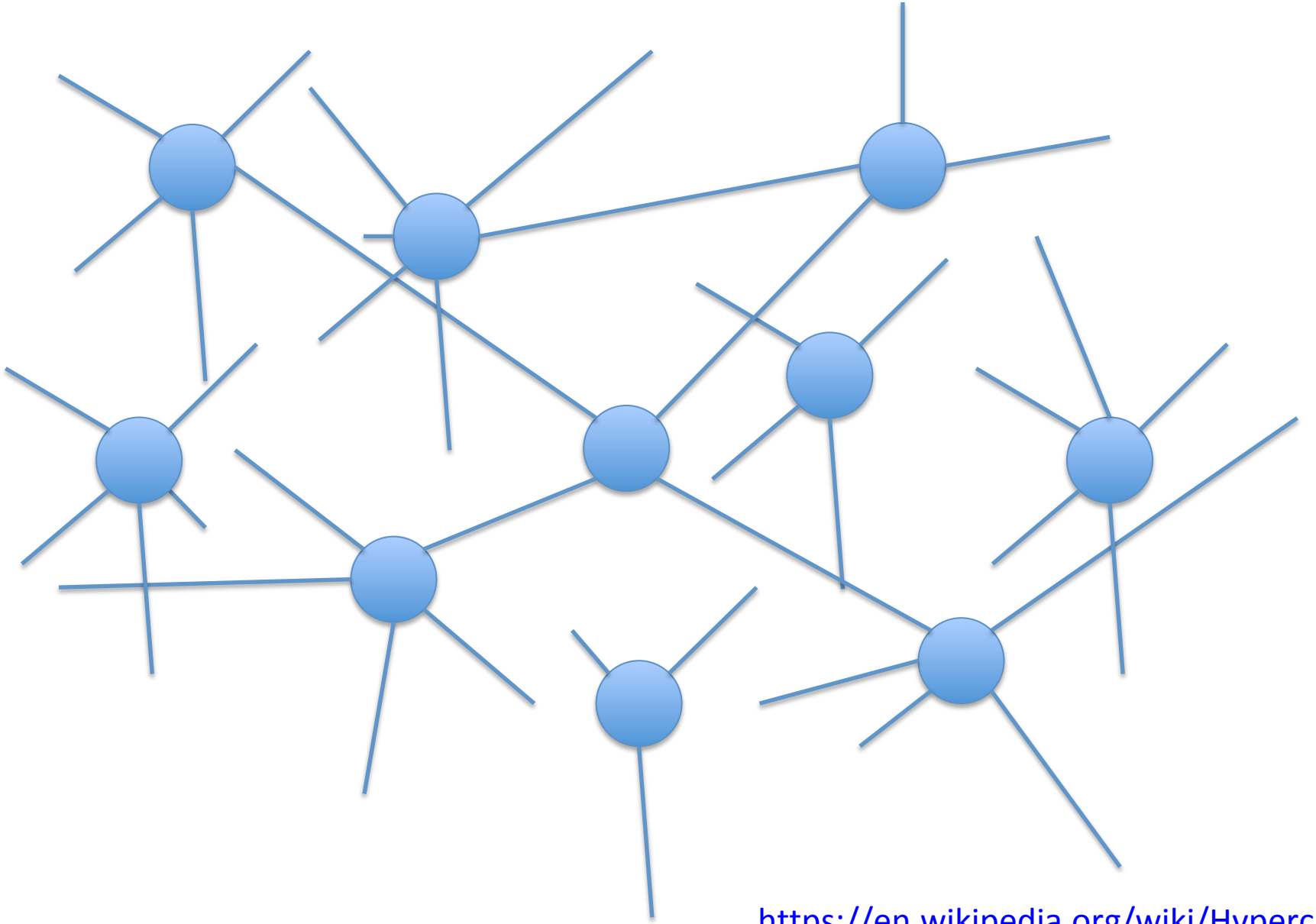
- mips treemap
- ppc treemap

Instruction Space as a Hypercube



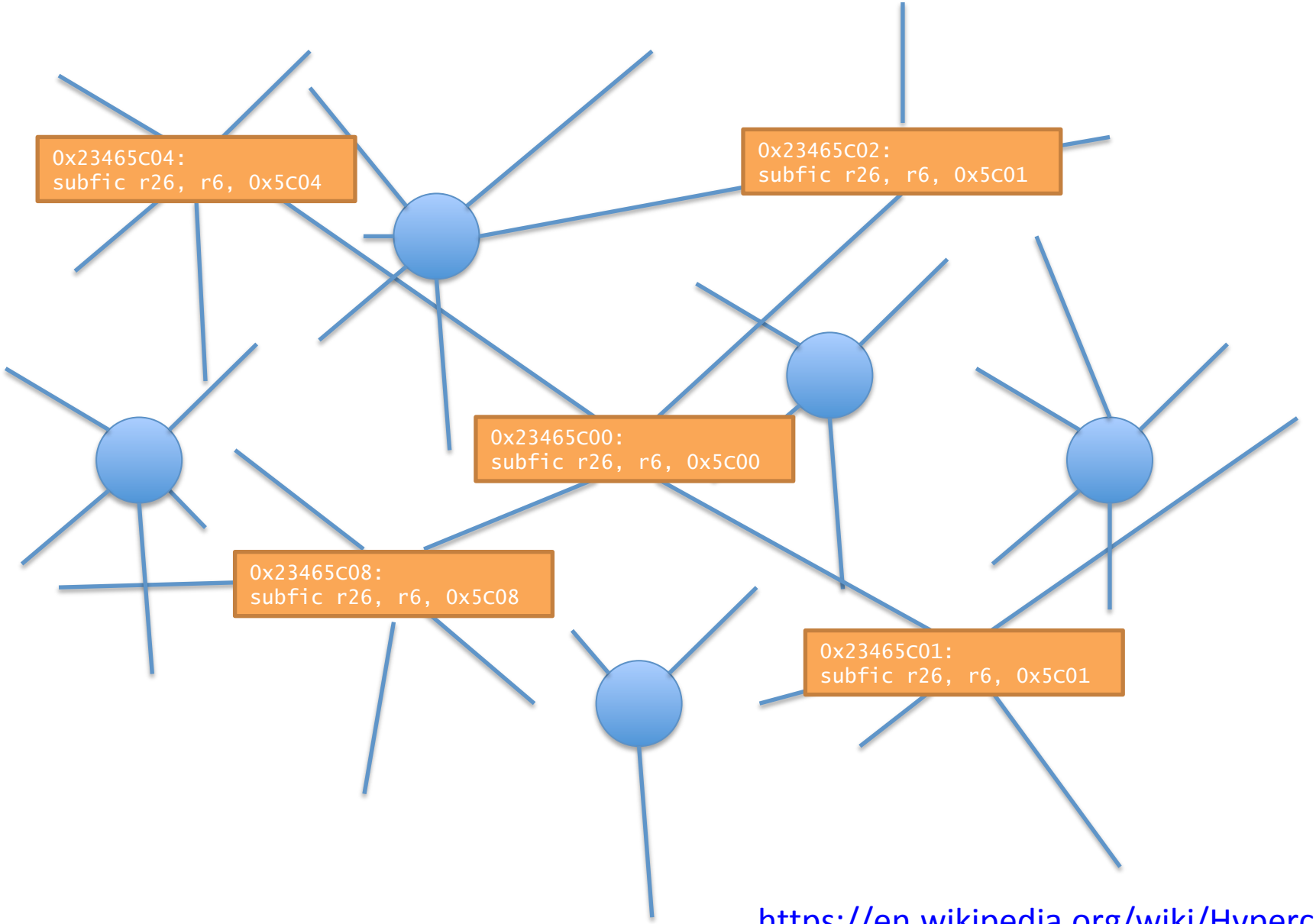
https://en.wikipedia.org/wiki/Hypercube

Instruction Space as a Graph



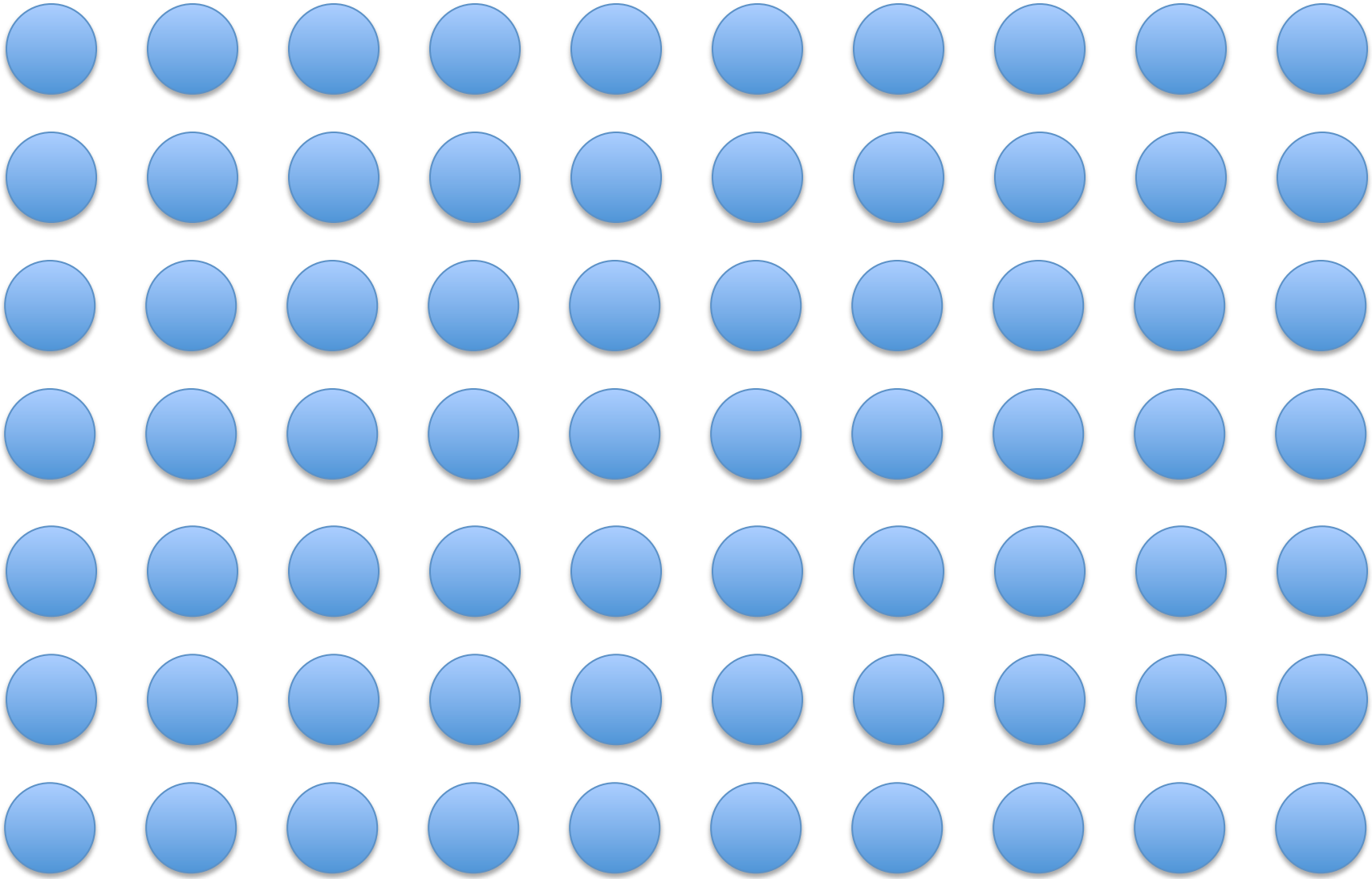
https://en.wikipedia.org/wiki/Hypercube

Instruction Space as a Graph

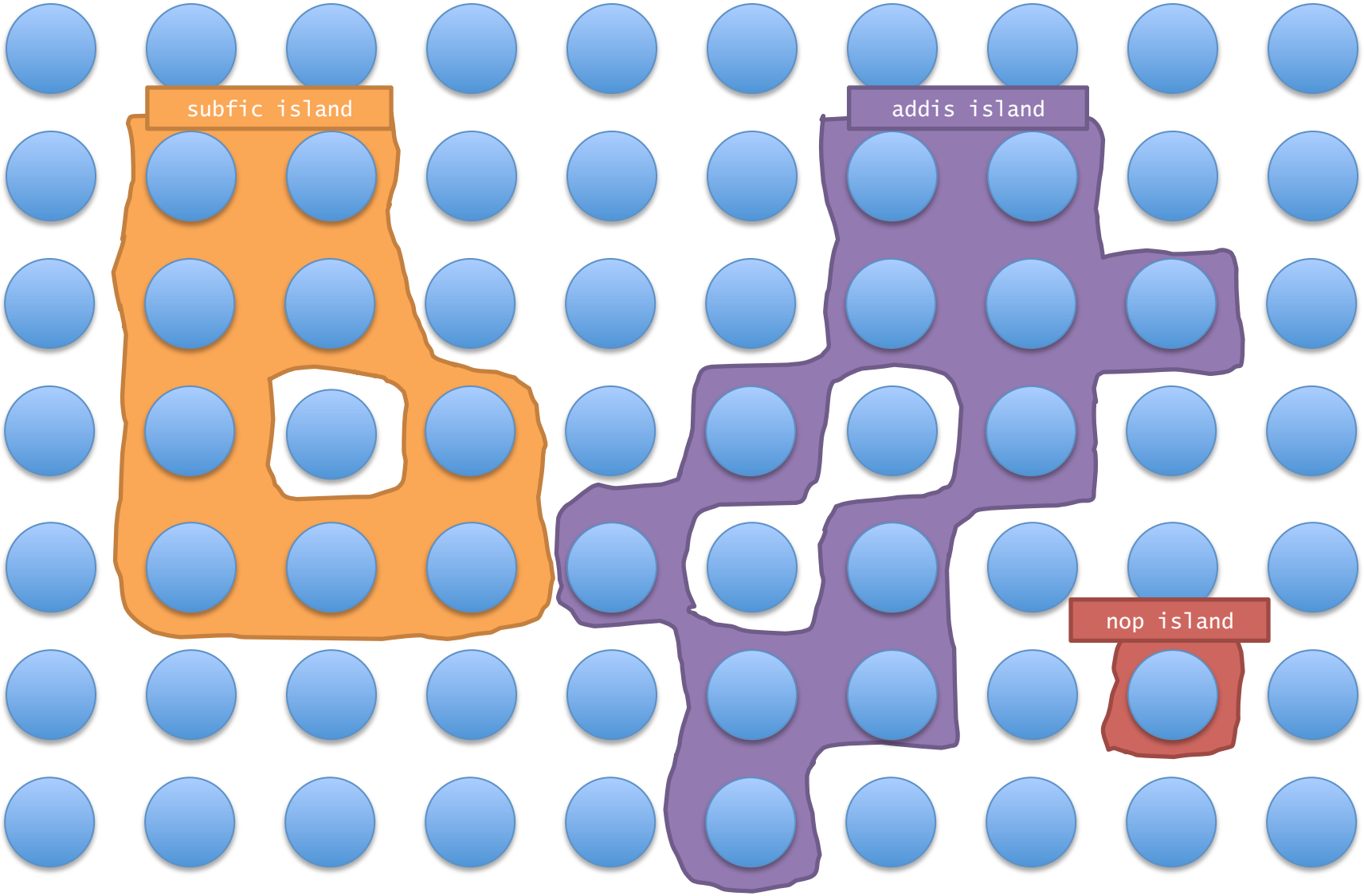


https://en.wikipedia.org/wiki/Hypercube

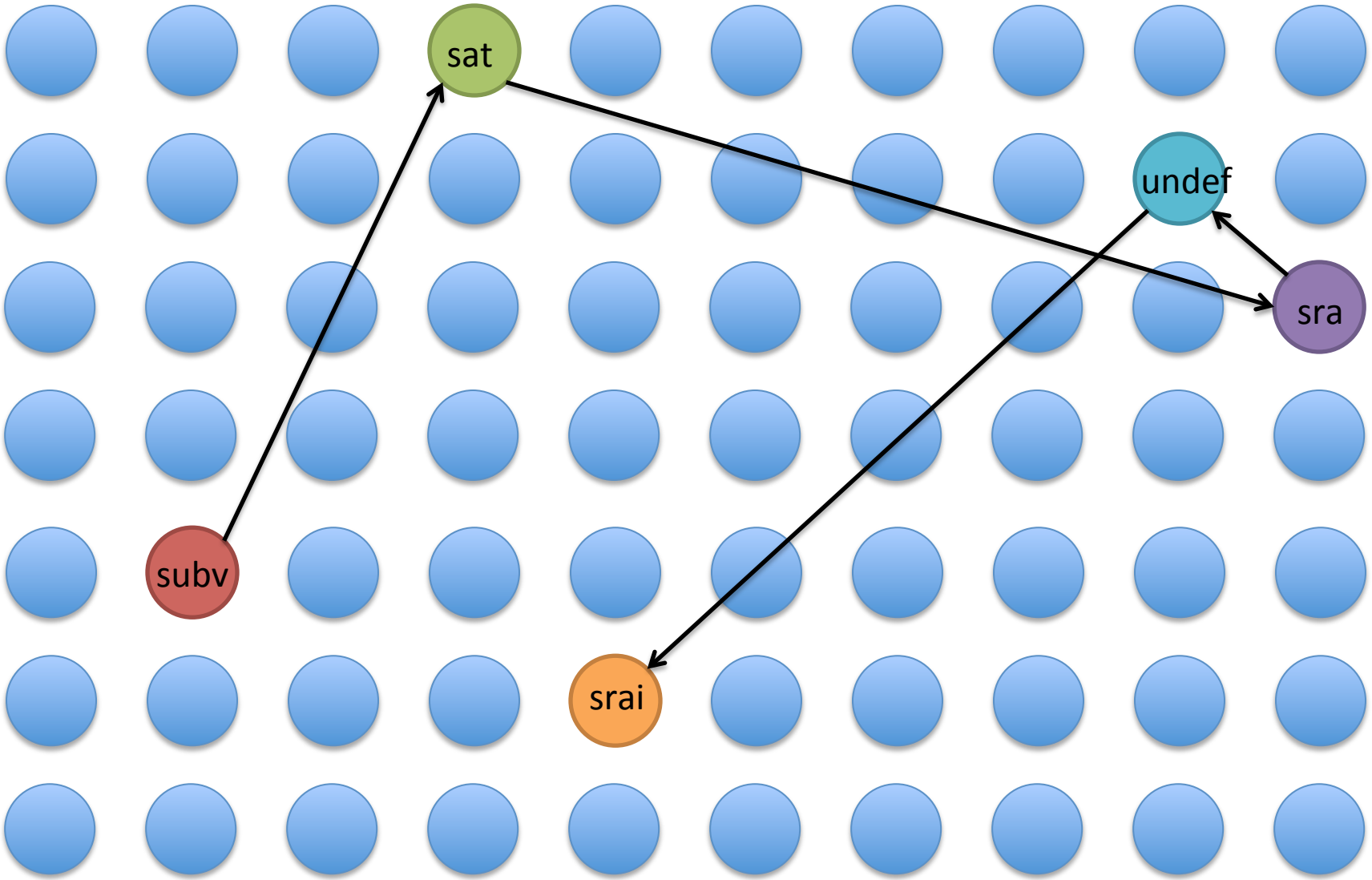
Edges Omitted



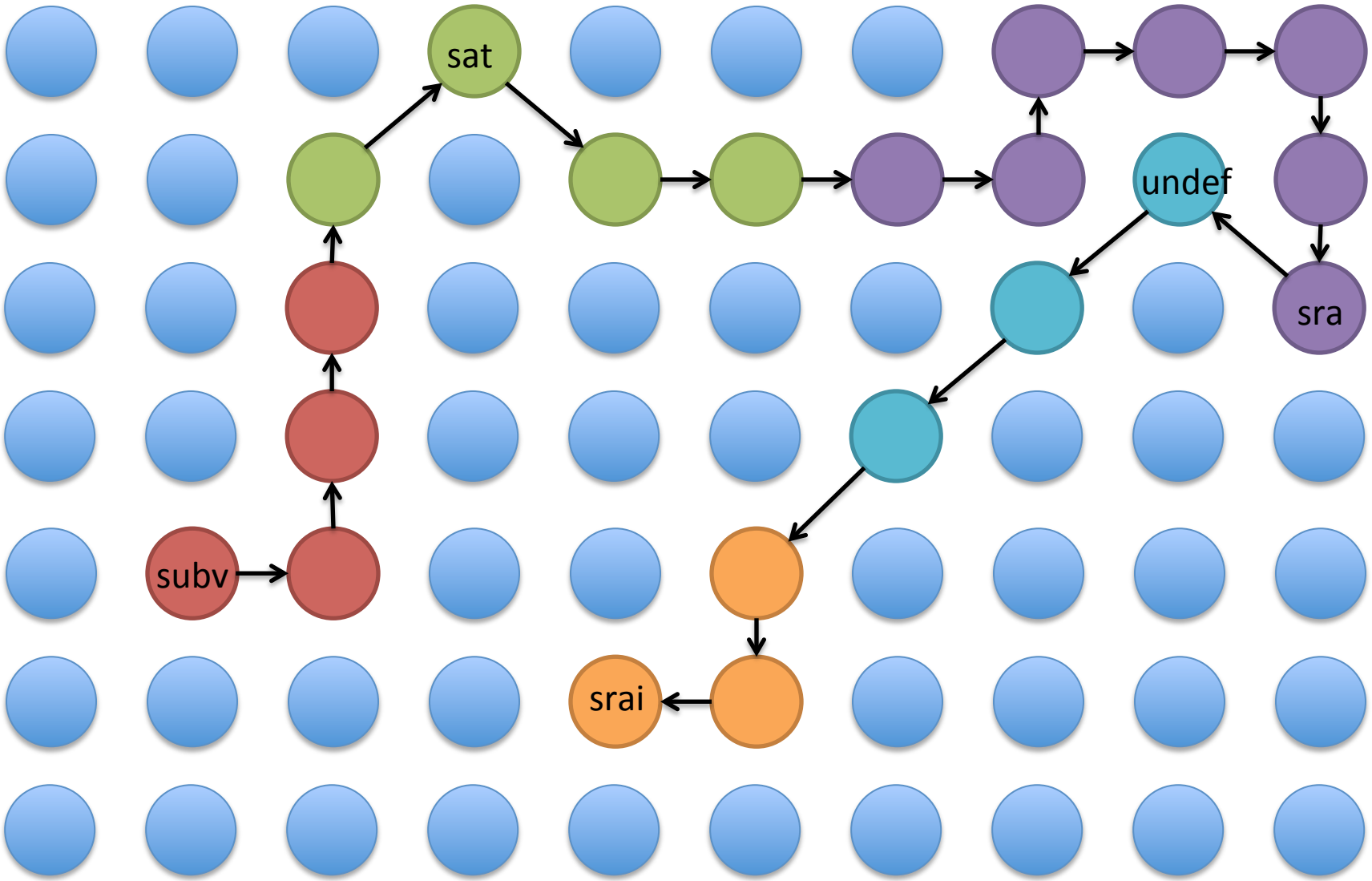
Opcode Regions, or “Islands”



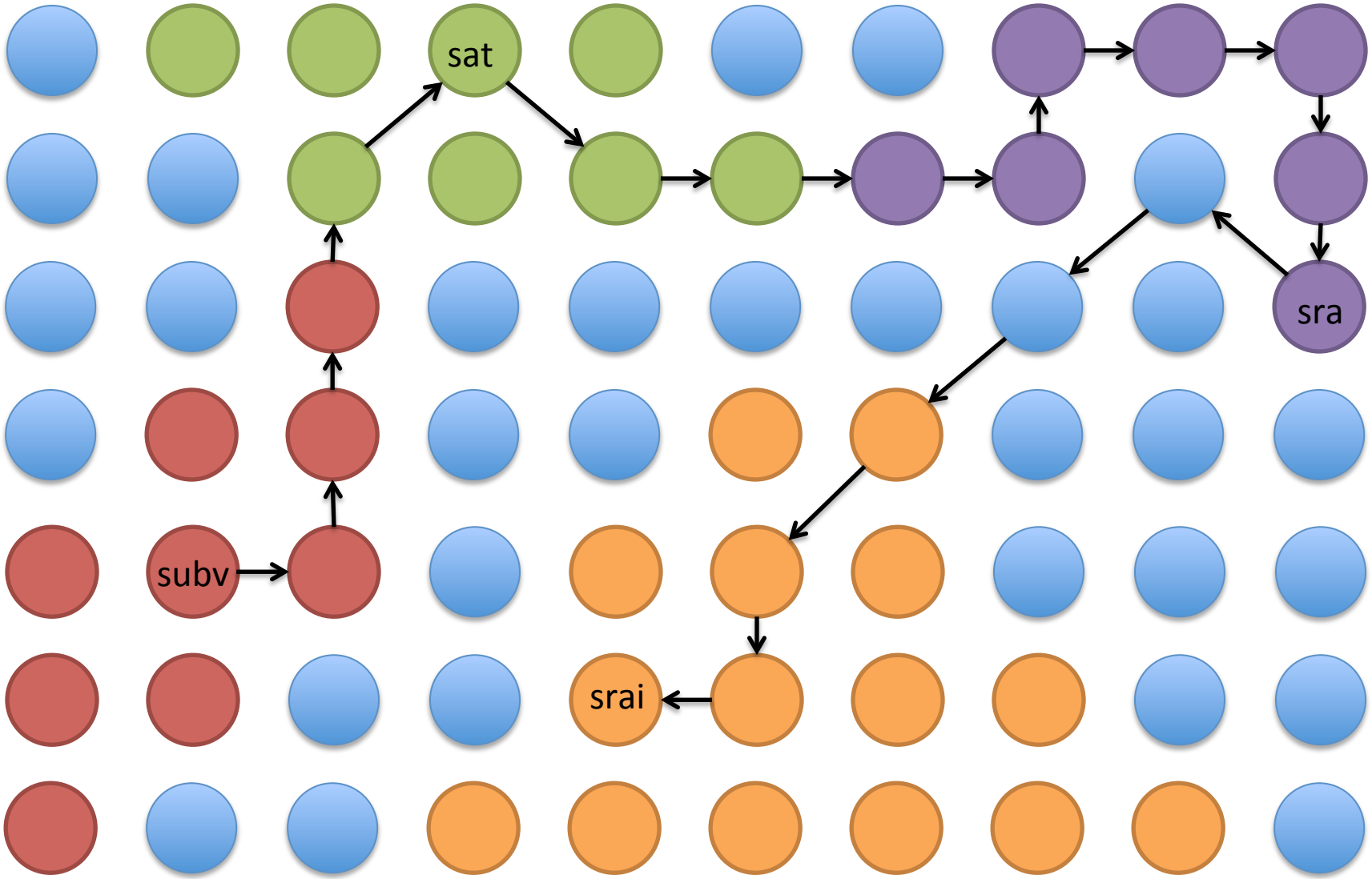
Number Line vs. Hamming=1 Travel



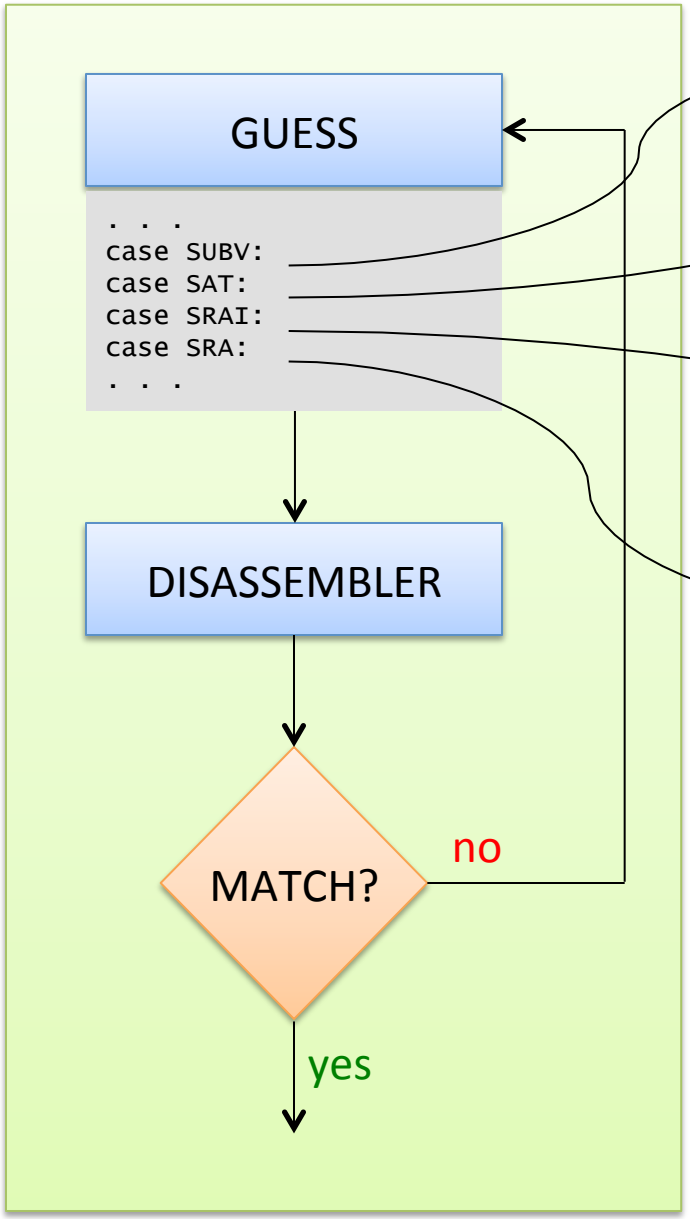
Number Line vs. Hamming=1 Travel

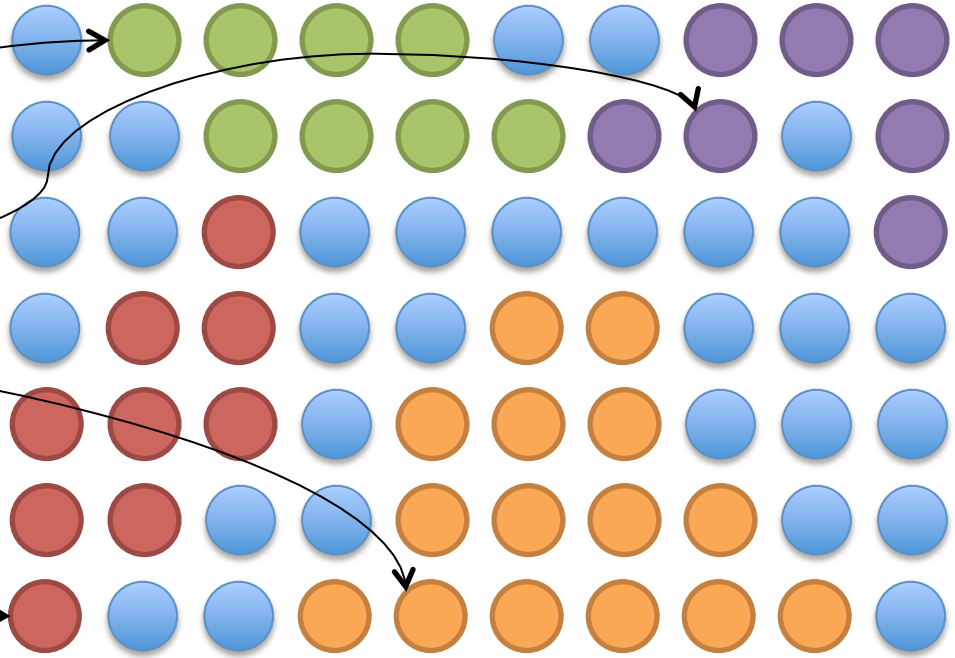


Number Line vs. Hamming=1 Travel



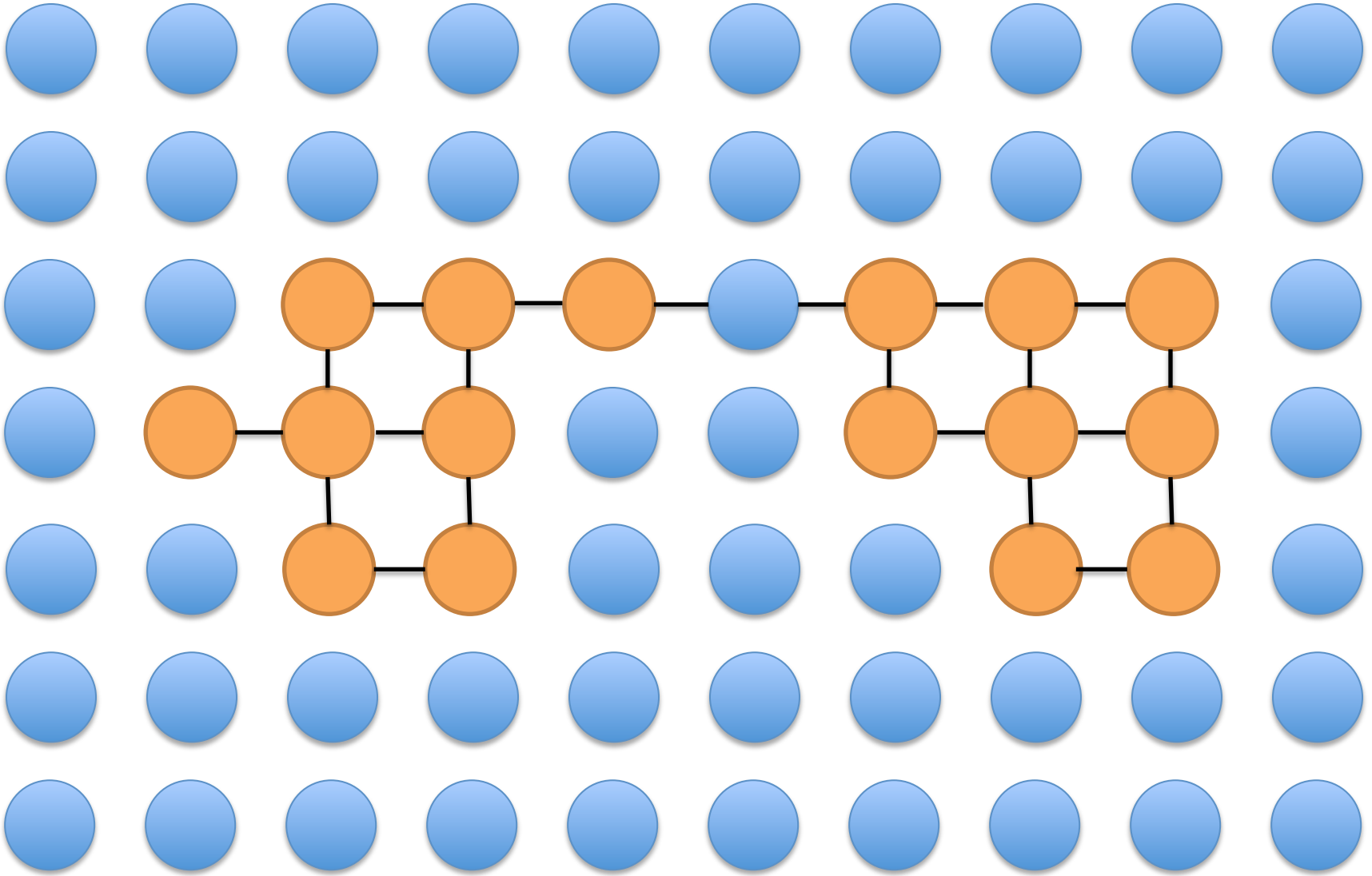
Islands Are The Guesser's Starting Points





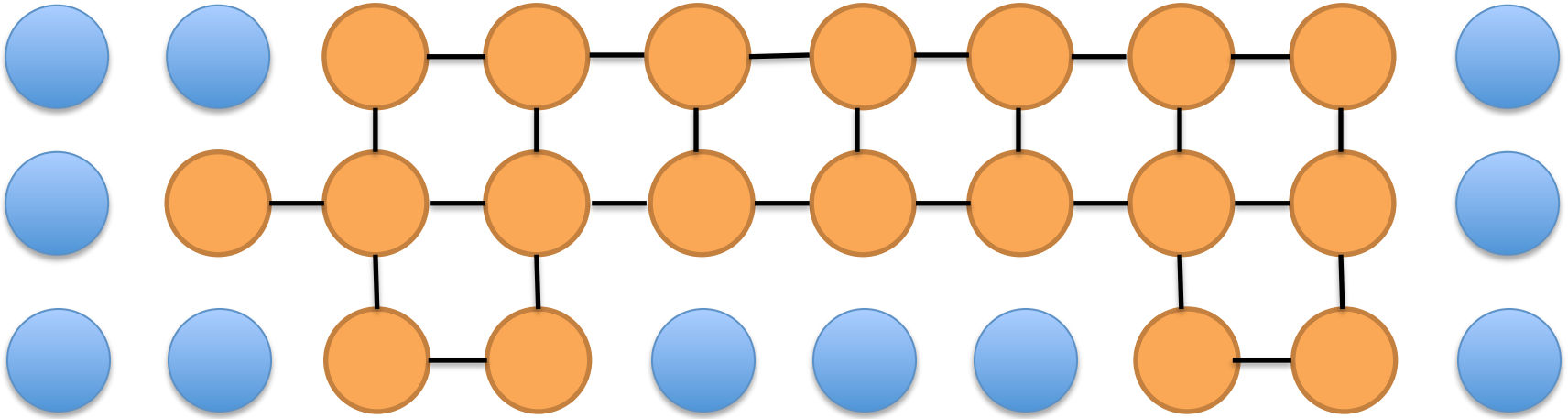
PPC:
~1000 opcodes
~75% defined
~3 million instrs/island

Worst Case: Separated Distant Regions

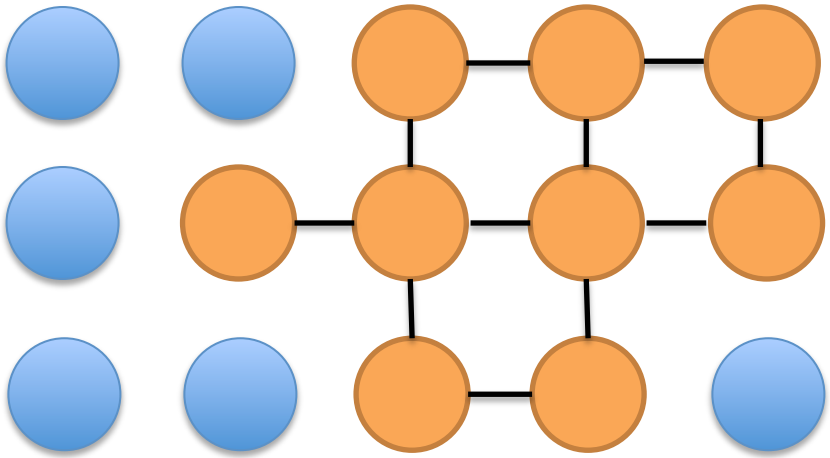


More Regions Are Better

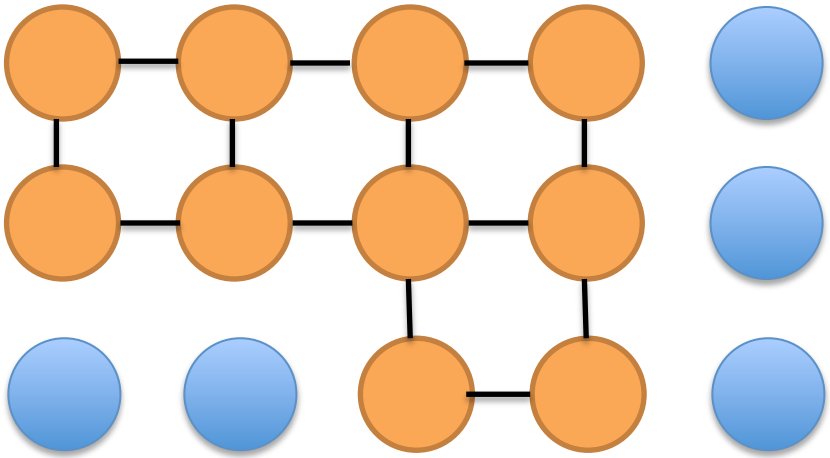
PPC: stmw



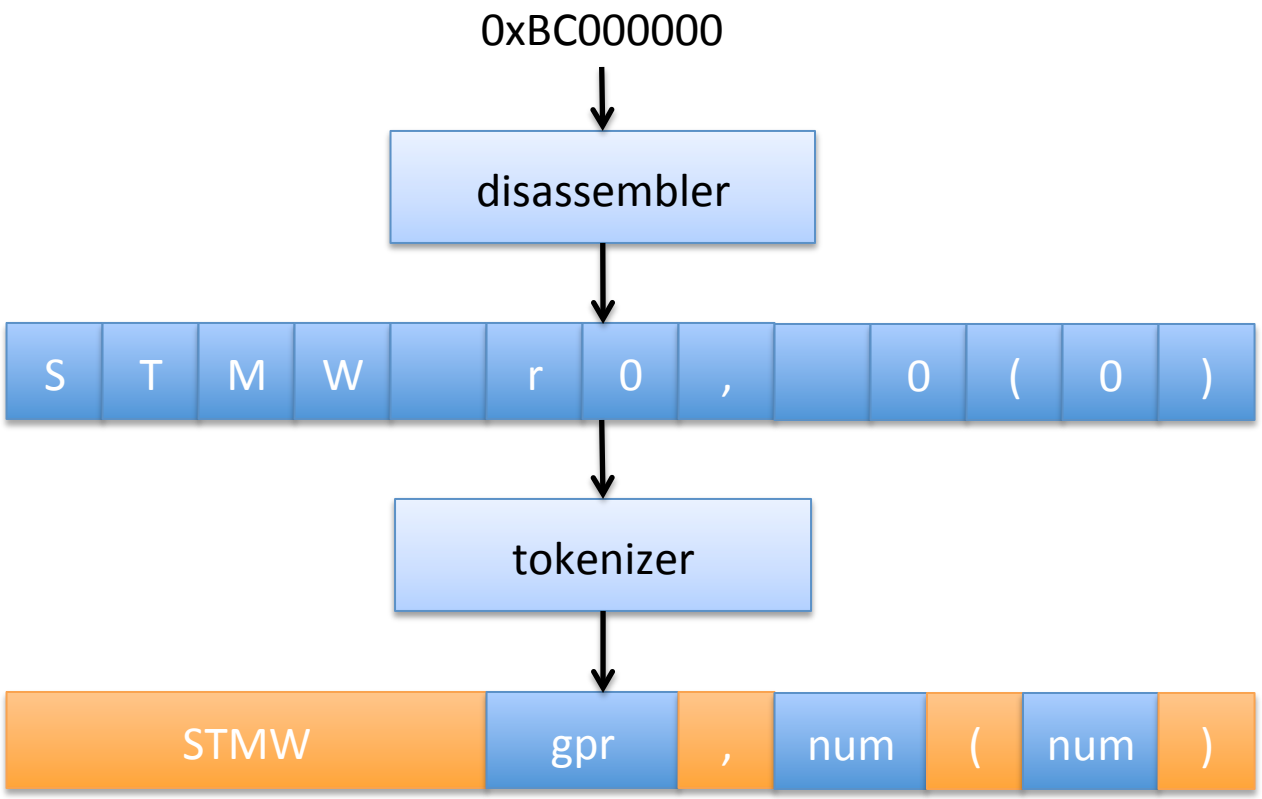
stmw <gpr>, <num> (num)



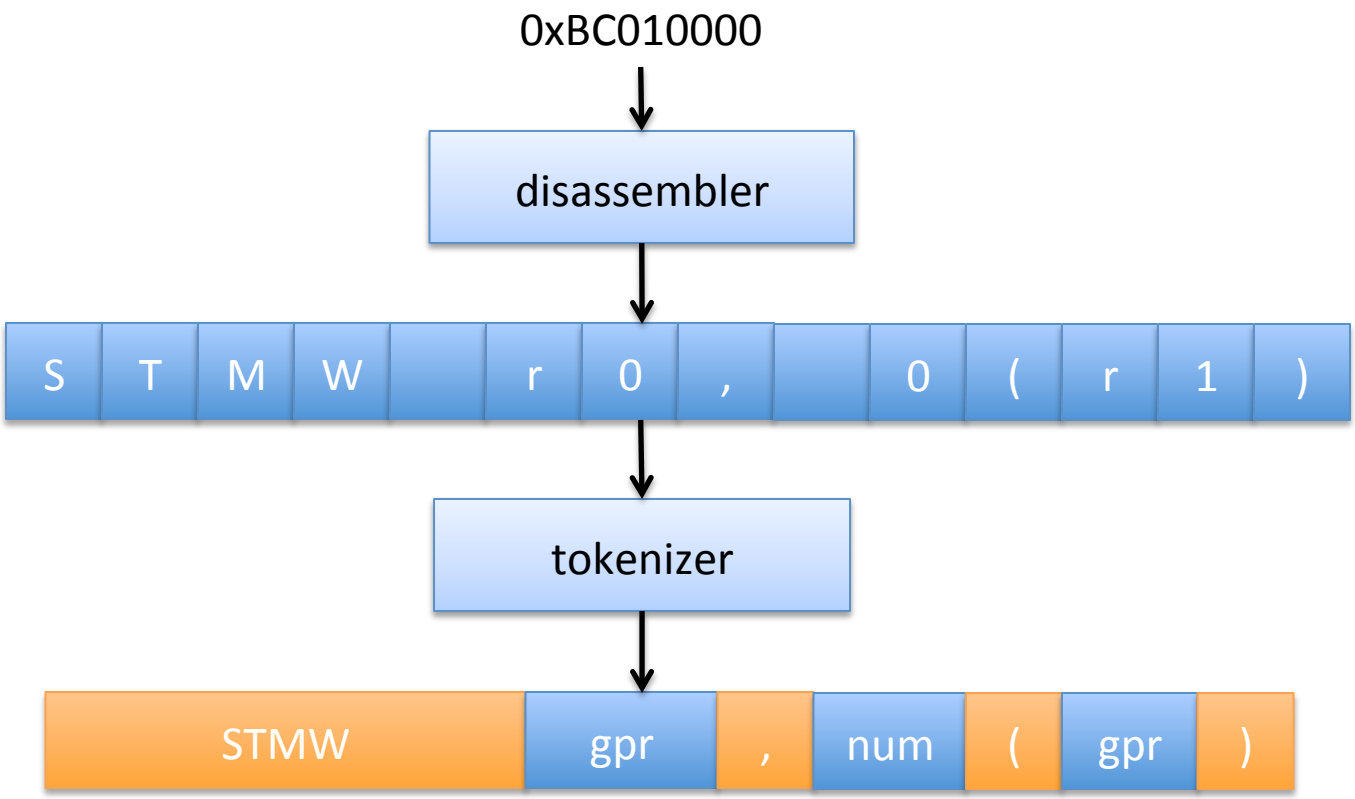
stmw <gpr>, <num> (GPR)



More Islands Via Tokenization



More Islands Via Tokenization



What Possible Tokens?

PPC:

```
GPR
VREG
FLAG
CREG
VSREG
FREG
NUM
list('(),.+-')
```

MIPS:

```
FREG
WREG
ACREG
GPREG
NUM
CASH
OFFS
list('[](),.*+-')
```

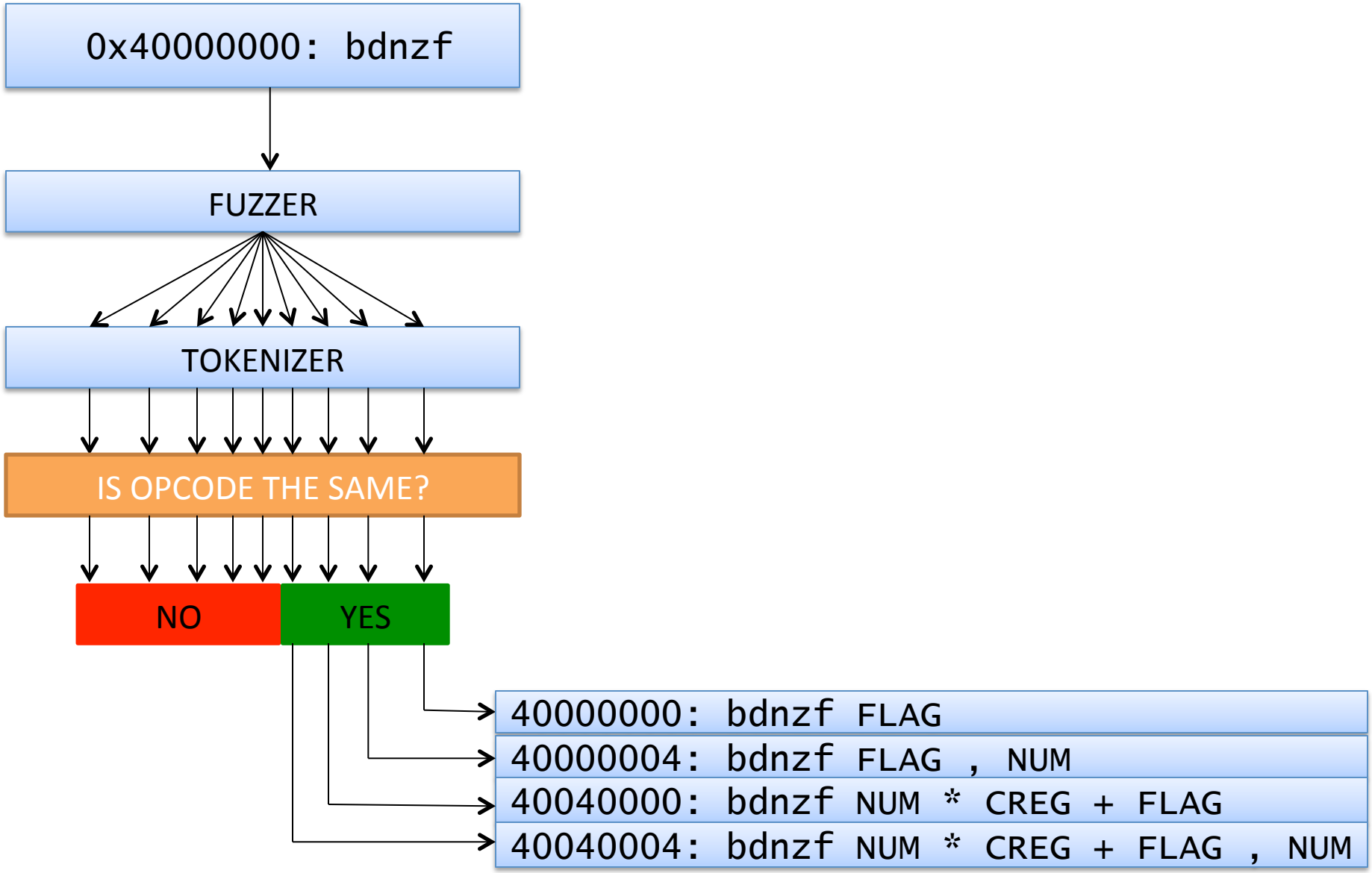
show mips/common.py

How do we find syntax examples?

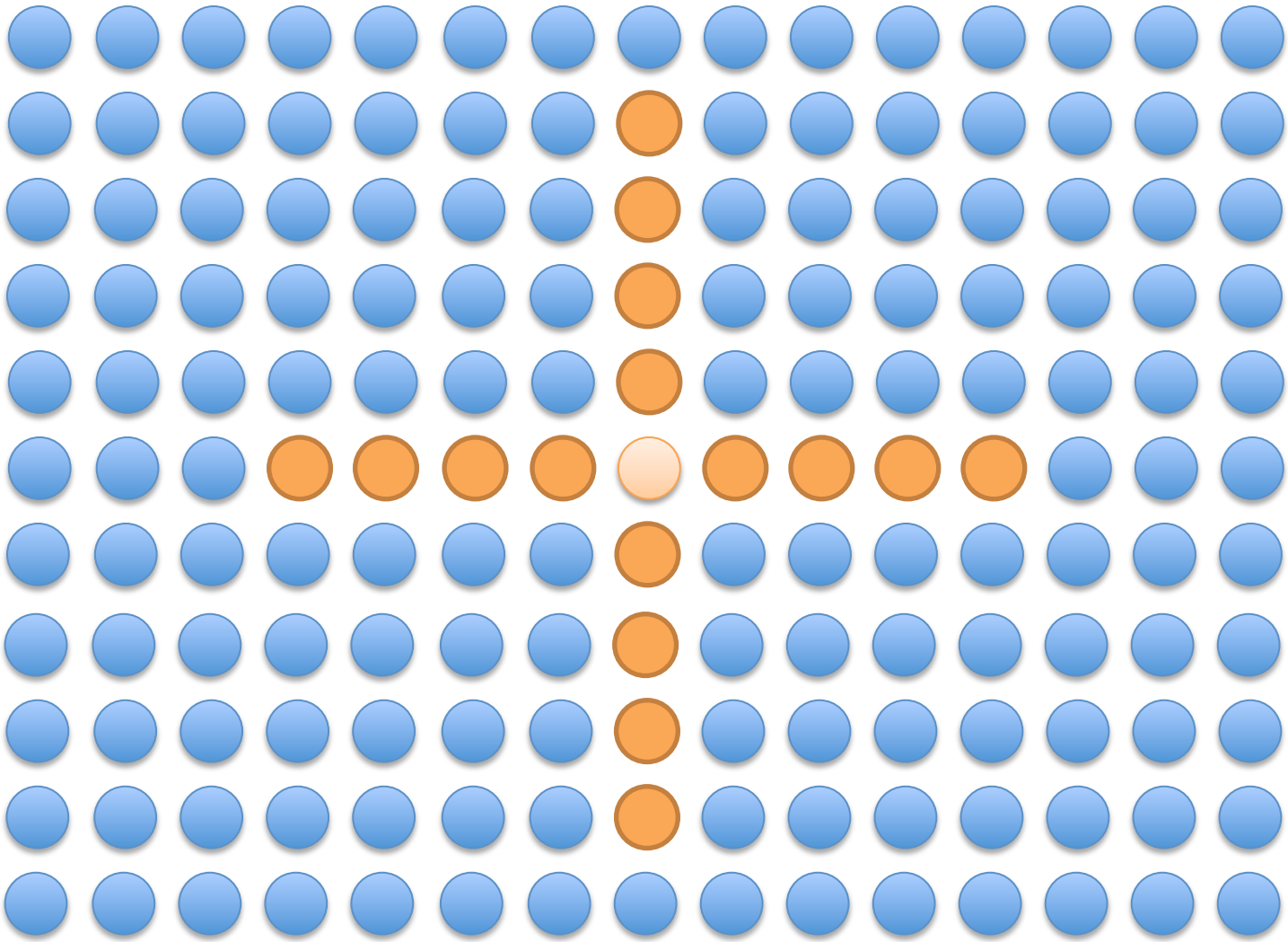
```
for insword in examples:
   for ss in EverySubsetOf4Bits():
      for val in range(16):
         insword2 = apply(insword, ss, val)

         tokens = tokenize(disassemble(insword2)))
         syntax = ' '.join(tokens)
         if not syntax in seen:
            seen[syntax] = insword2
```

- nCr(32,4) * 16 = 575,360
- nCr(32,5) * 32 = 6,444,032
- syn_examples.txt

How do we find syntax examples?

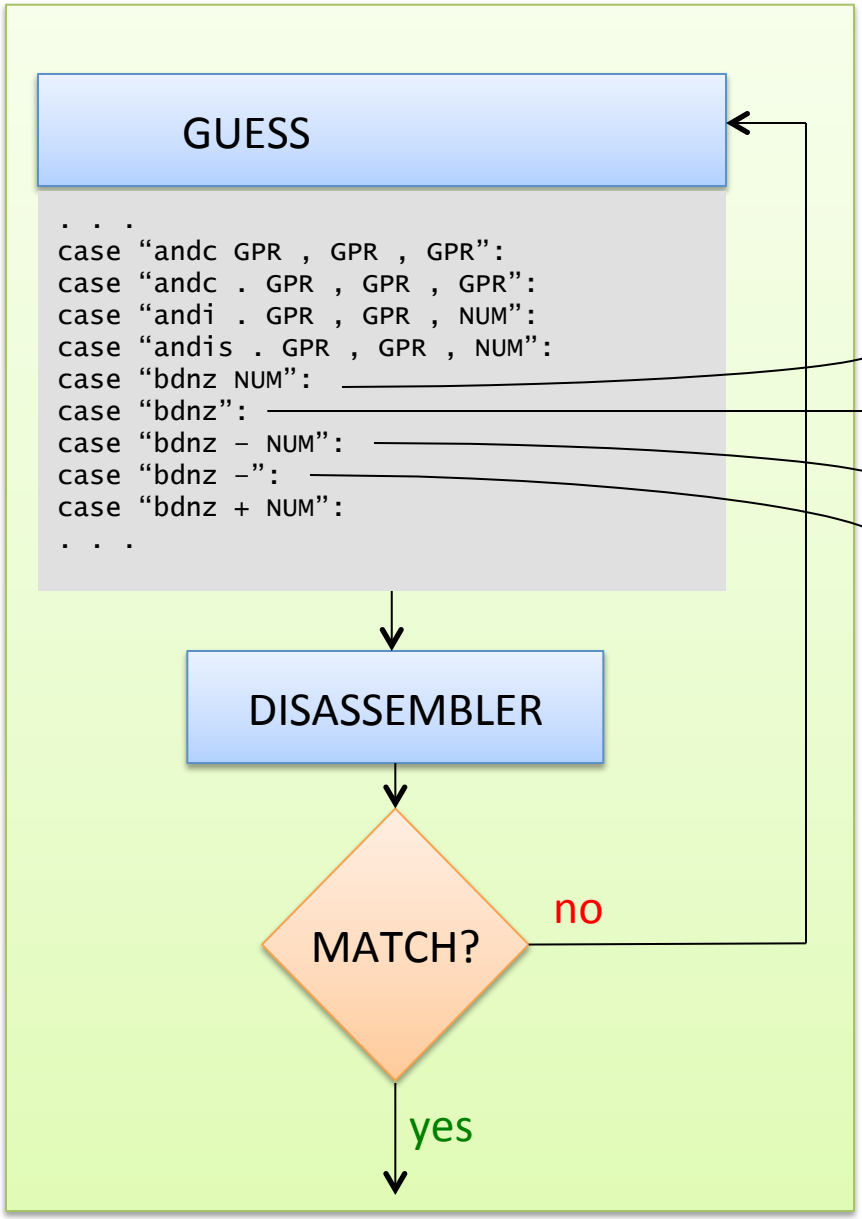


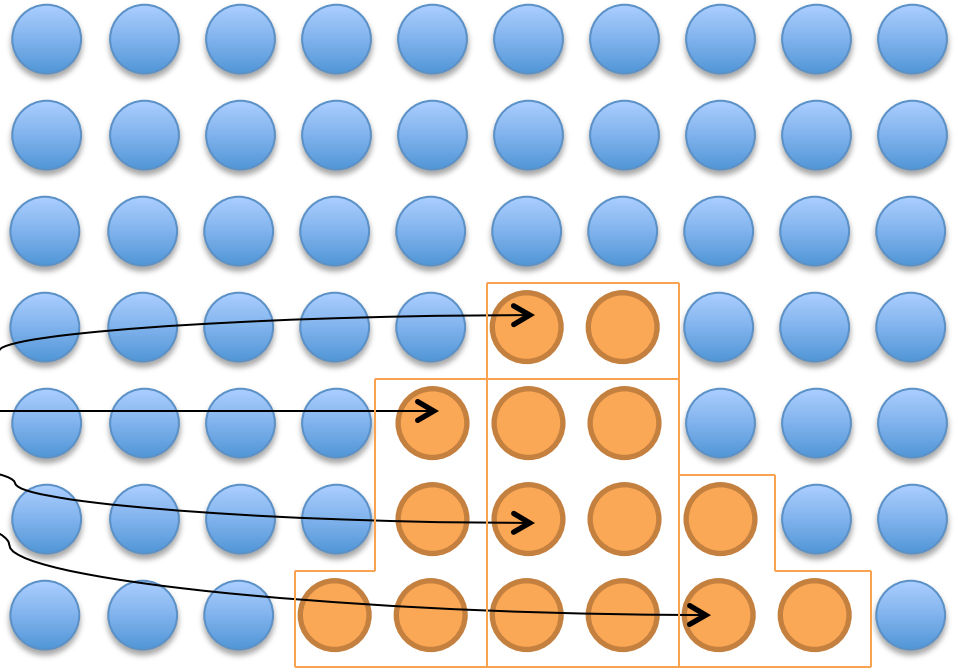
Toggling 4-bits At a Time



2098 Island Now
1.5 million instrs/island

Guesser Deals In Syntaxes Now





PPC:
2098 syntaxes
~75% defined
1.5 million instrs/island

Recap

1. Didn’t know where to initiate guesses
2. Scanned instruction space to find examples for each opcode (this solved the “Where to start?” problem)
3. Too broad, split opcodes into their syntaxes

Problems with Vanilla Genetic Algorithm

1. ~~Where to start?~~
2. What fitness function?
3. How to converge?

The remainder of this talk solves these problems.

Comparing Assembly, Fitness Function

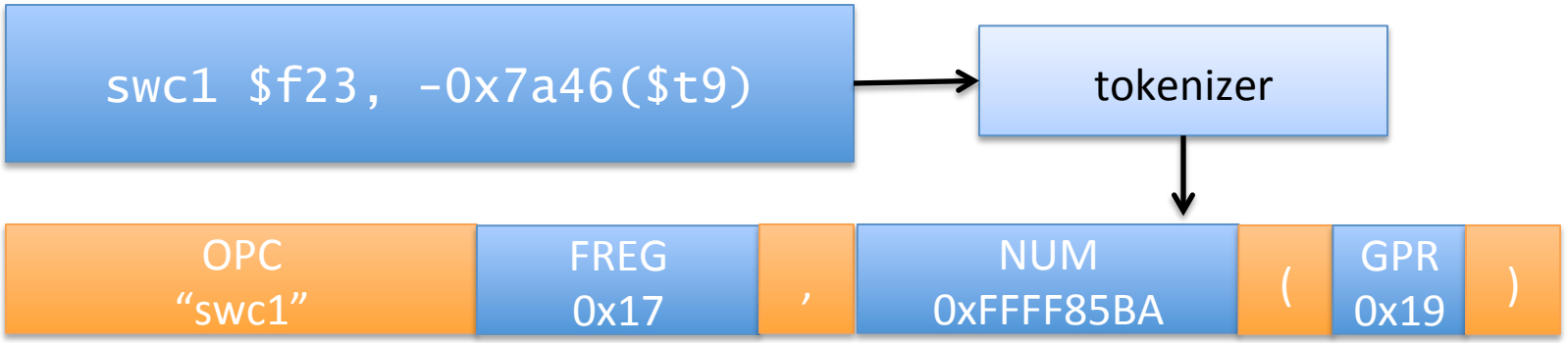
```
swc1 $f23, -0x7a46($t9)
```

Which one is closest?

- swc1 $f0, 0x101($zero)
- sdc1 $f23, 0x7a46($t9)
- swc1 $f0, 0x106($zero)
- swc1 $f15, -0x7a46($zero)
- swc1 $f23, -0x7a46($at)
- swc1 $f23, -0x7a46($t1)

https://en.wikipedia.org/wiki/Category:String_similarity_measures

Comparing Assembly, Fitness Function



Which one is closest?

OPC	FREG	,	NUM	(	GPR	)
“swc1”	0x00	,	0x101	(	0x0	)
“sdc1”	0x17	,	0xFFFF85BA	(	0x19	)
“swc1”	0x00	,	0x106	(	0x00	)
“swc1”	0x15	,	0xFFFF85BA	(	0x00	)

Comparing Assembly, Fitness Function

Score zero if:

- Opcode mismatch
- Token quantity mismatch
- Token type mismatch

For STRING tokens:

- 0% if mismatch, 100% otherwise

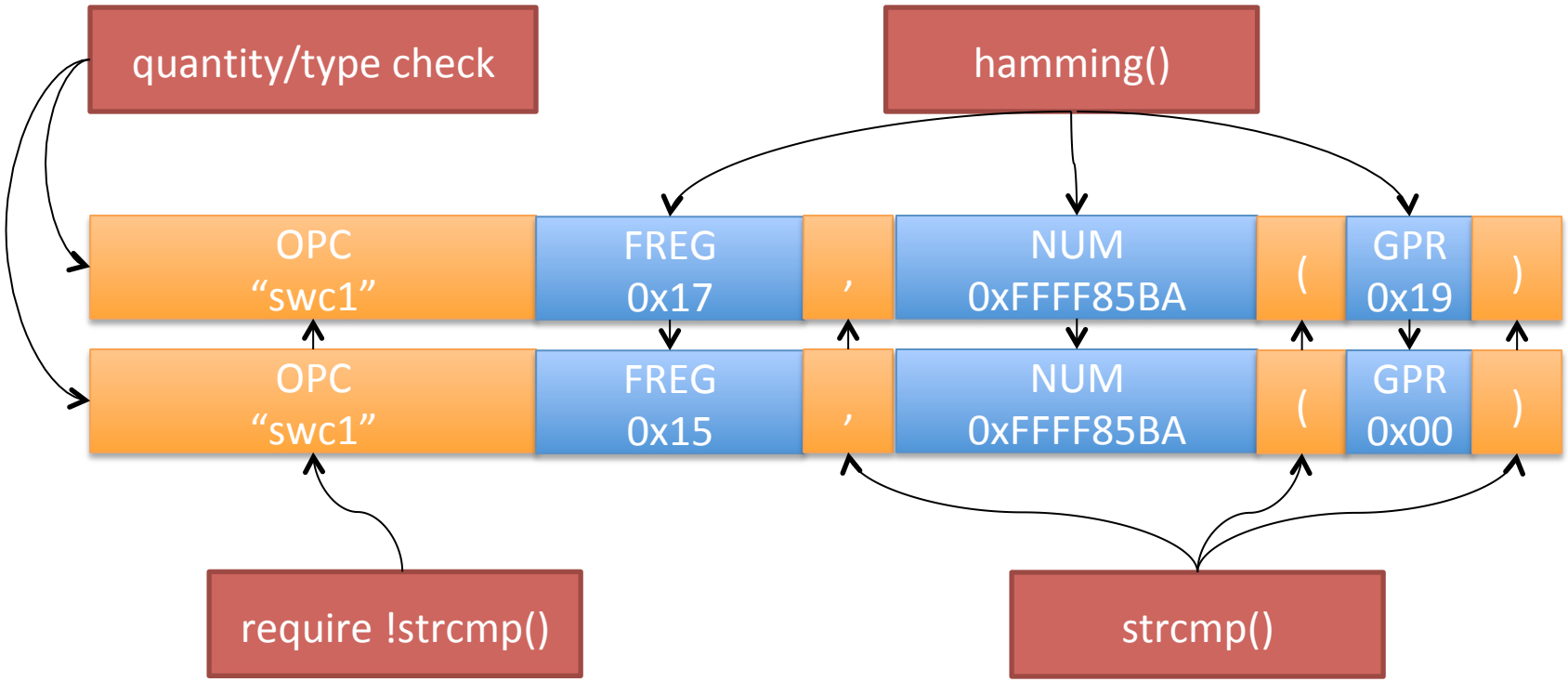
For NUM tokens, registers:

- hamming distance

For OFFS tokens:

- hamming distance of encoded address

Comparing Assembly, Fitness Function



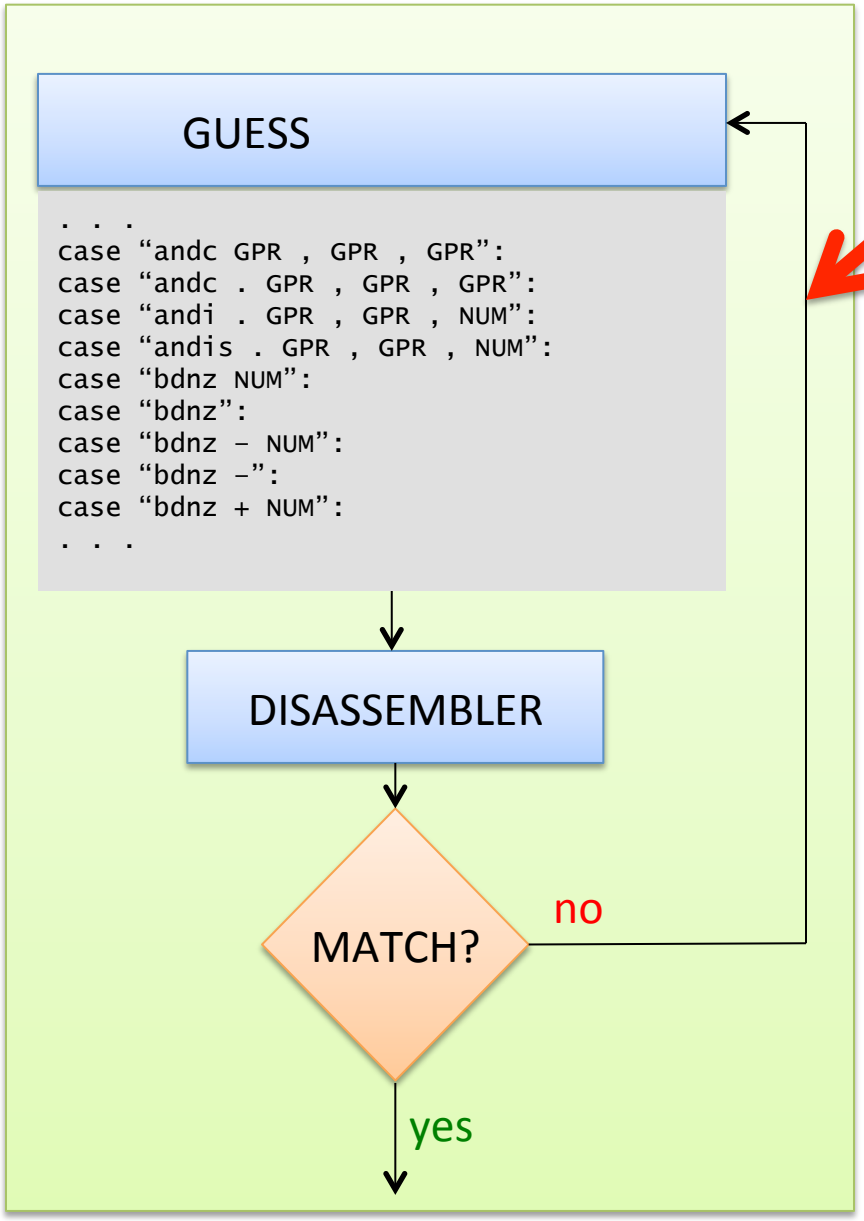
Assemble.cpp PPC
Assemble.cpp MIPS

Problems with Vanilla Genetic Algorithm

1. ~~Where to start?~~
2. ~~What fitness function?~~
3. How to converge?

The remainder of this talk solves these problems.

After failure, what direction?



What do we do on guess #2?
…on guess #3?
…on guess #4?

Randomly toggle bits?

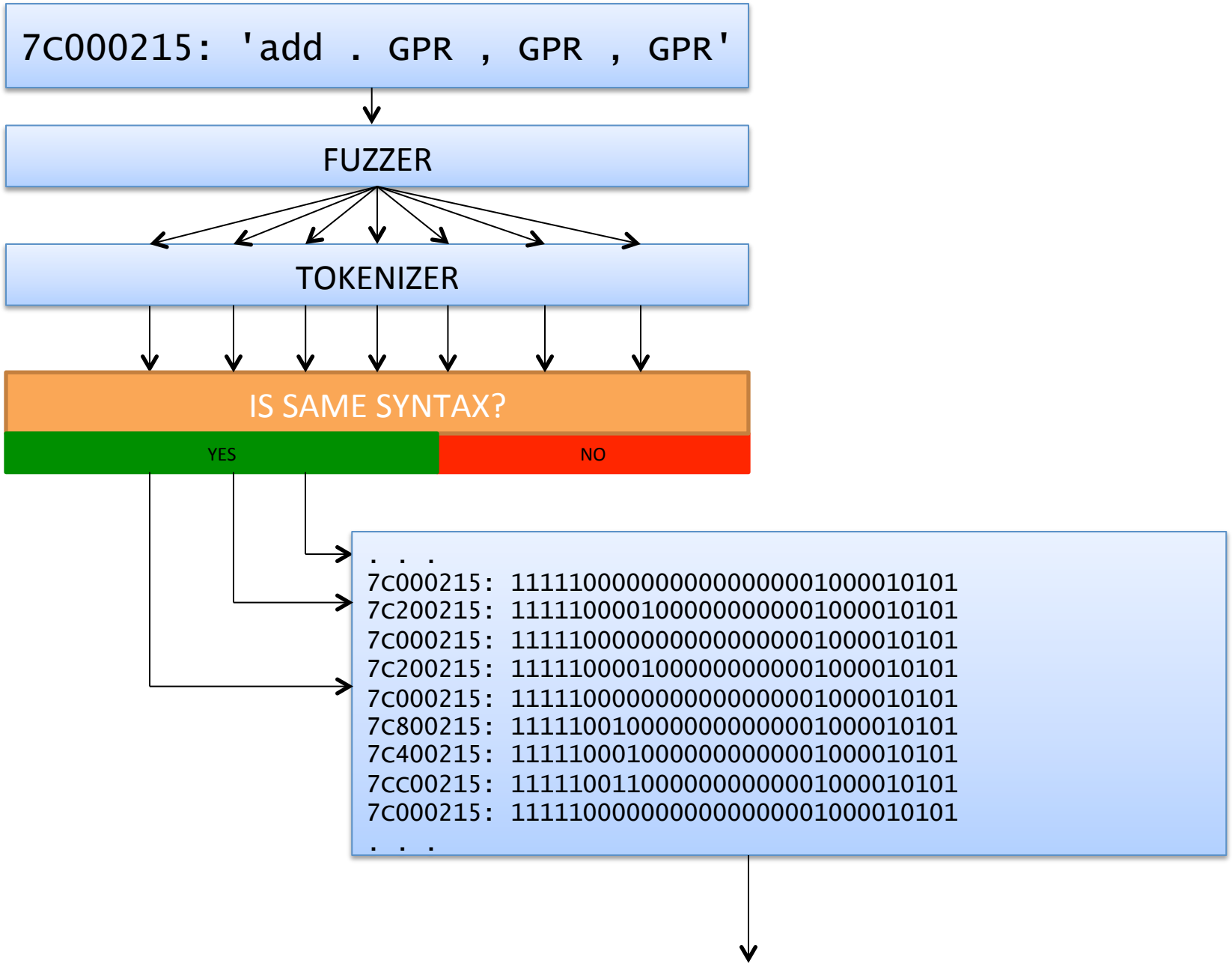
Fuzzing Encodings

- Precompute which bits are worth toggling

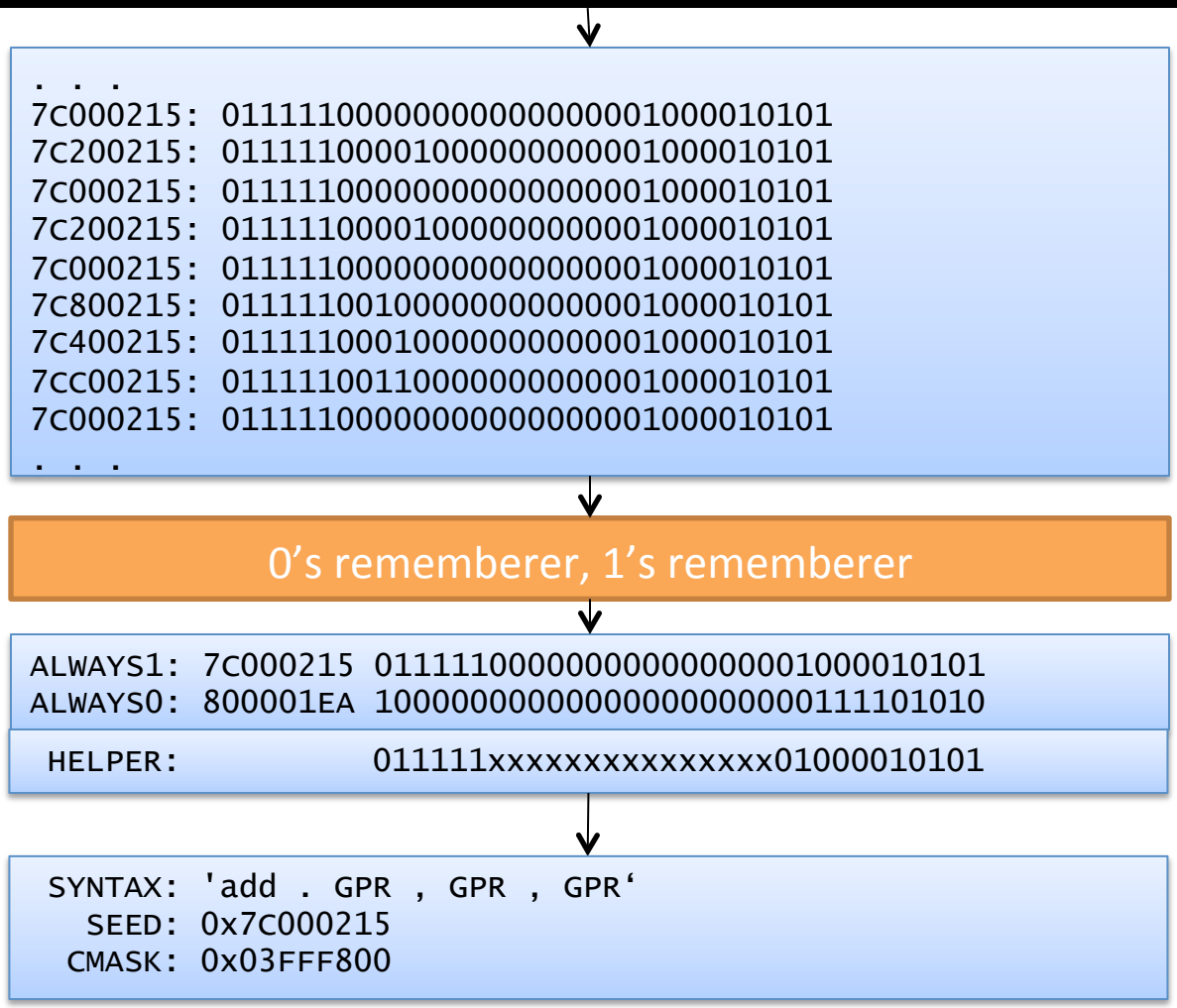
```
# given opc with encoding insword

always1 = insword
always0 = ~insword
for ss in EverySubsetOf4Bits():
   for val in range(16):
      insword2 = apply(insword, ss, val)
      # continue if different syntax
      always1 &= insword2
      always0 &= ~insword2
```

mips_sandbox/syn_bitpatterns.py ppc_sandbox/syn_bitpatterns.py

In search of bit patterns…



In search of bit patterns…



"ADD . GPR , GPR , GPR"

011111xxxxxxxxxxxxxxx01000010101

BitPatterns Found!

```
"tdi NUM , GPR , NUM",{0x0800000A,0x03FFFFFF}}, //000010xxxxxxxxxxxxxxxxxxxxxxxx tdi 0, r0, 0xa
   "tdlgti GPR , NUM",{0x08200000,0x001FFFFF}}, //00001000001xxxxxxxxxxxxxxxxxxx tdlgti r0, 0
   "tdllti GPR , NUM",{0x08400000,0x001FFFFF}}, //00001000010xxxxxxxxxxxxxxxxxxx tdllti r0, 0
    "tdeqi GPR , NUM",{0x08800000,0x001FFFFF}}, //00001000100xxxxxxxxxxxxxxxxxxx tdeqi r0, 0
    "tdgti GPR , NUM",{0x09000000,0x001FFFFF}}, //00001001000xxxxxxxxxxxxxxxxxxx tdgti r0, 0
    "tdlti GPR , NUM",{0x0A000000,0x001FFFFF}}, //00001010000xxxxxxxxxxxxxxxxxxx tdlti r0, 0
    "tdnei GPR , NUM",{0x0B000000,0x001FFFFF}}, //00001011000xxxxxxxxxxxxxxxxxxx tdnei r0, 0
     "tdui GPR , NUM",{0x0BE00000,0x001FFFFF}}, //00001011111xxxxxxxxxxxxxxxxxxx tdui r0, 0
"twi NUM , GPR , NUM",{0x0C000000,0x03FFFFFF}}, //000011xxxxxxxxxxxxxxxxxxxxxxxx twi 0, r0, 0
   "twlgti GPR , NUM",{0x0C200000,0x001FFFFF}}, //00001100001xxxxxxxxxxxxxxxxxxx twlgti r0, 0
   "twllti GPR , NUM",{0x0C400000,0x001FFFFF}}, //00001100010xxxxxxxxxxxxxxxxxxx twllti r0, 0
    "tweqi GPR , NUM",{0x0C800000,0x001FFFFF}}, //00001100100xxxxxxxxxxxxxxxxxxx tweqi r0, 0
    "twgti GPR , NUM",{0x0D000000,0x001FFFFF}}, //00001101000xxxxxxxxxxxxxxxxxxx twgti r0, 0
    "twlti GPR , NUM",{0x0E000000,0x001FFFFF}}, //00001110000xxxxxxxxxxxxxxxxxxx twlti r0, 0
    "twnei GPR , NUM",{0x0F000000,0x001FFFFF}}, //00001111000xxxxxxxxxxxxxxxxxxx twnei r0, 0
     "twui GPR , NUM",{0x0FE00000,0x001FFFFF}}, //00001111111xxxxxxxxxxxxxxxxxxx twui r0, 0
. . .
```

Pitfall: Fuzzer Needs 4 Bits!

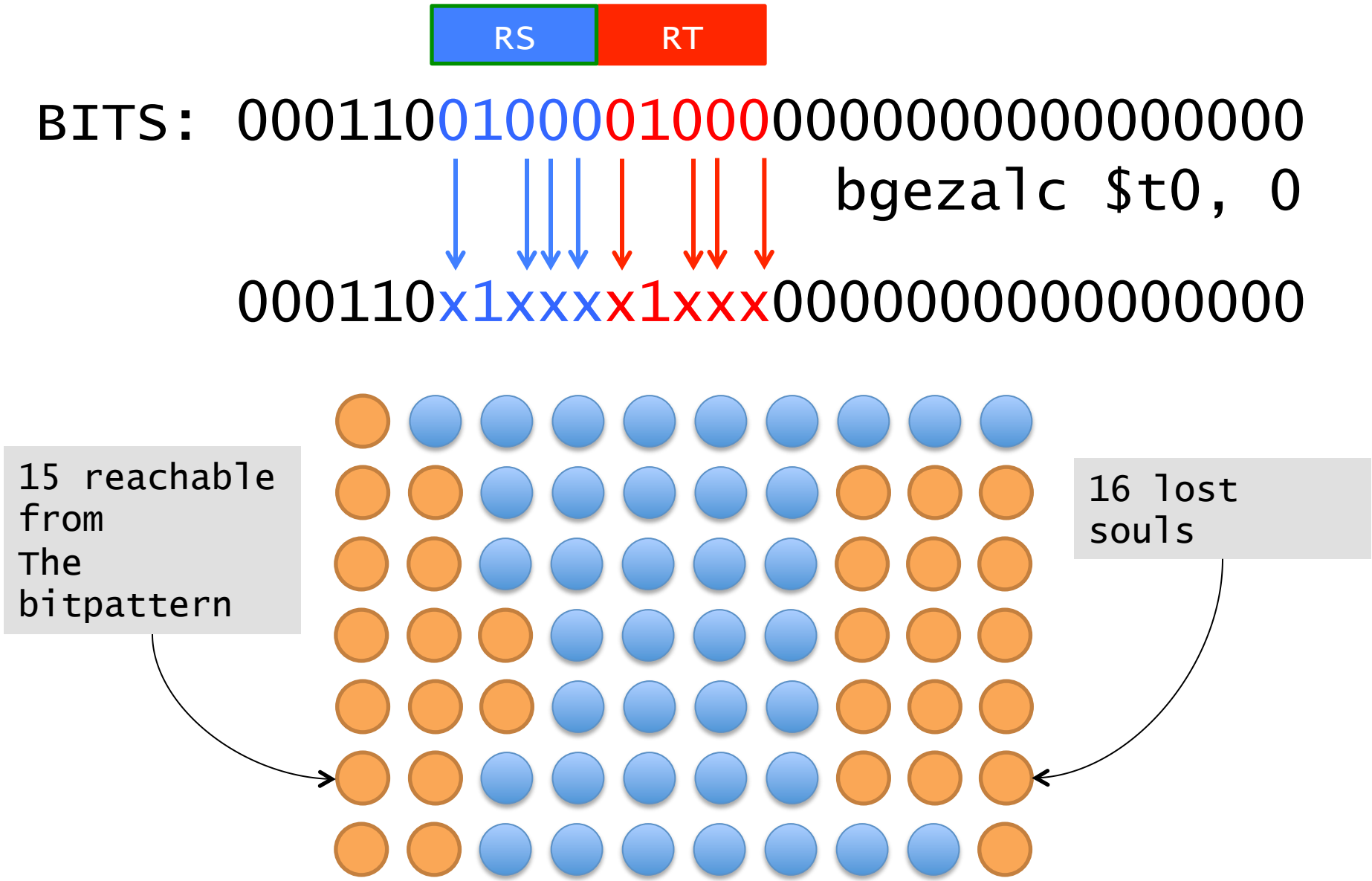
SYNTAX: 'bgezalc GPREG , NUM': 0x19080000

BITS: 00011001000010000000000000000000
bgezalc $t0, 0

BITS: 00011000000000000000000000000000
blez $zero, 4

BITS: 00011001000001000000000000000000
bgezalc $a0, 0

Pitfall: Fuzzer needs 4 Bits!



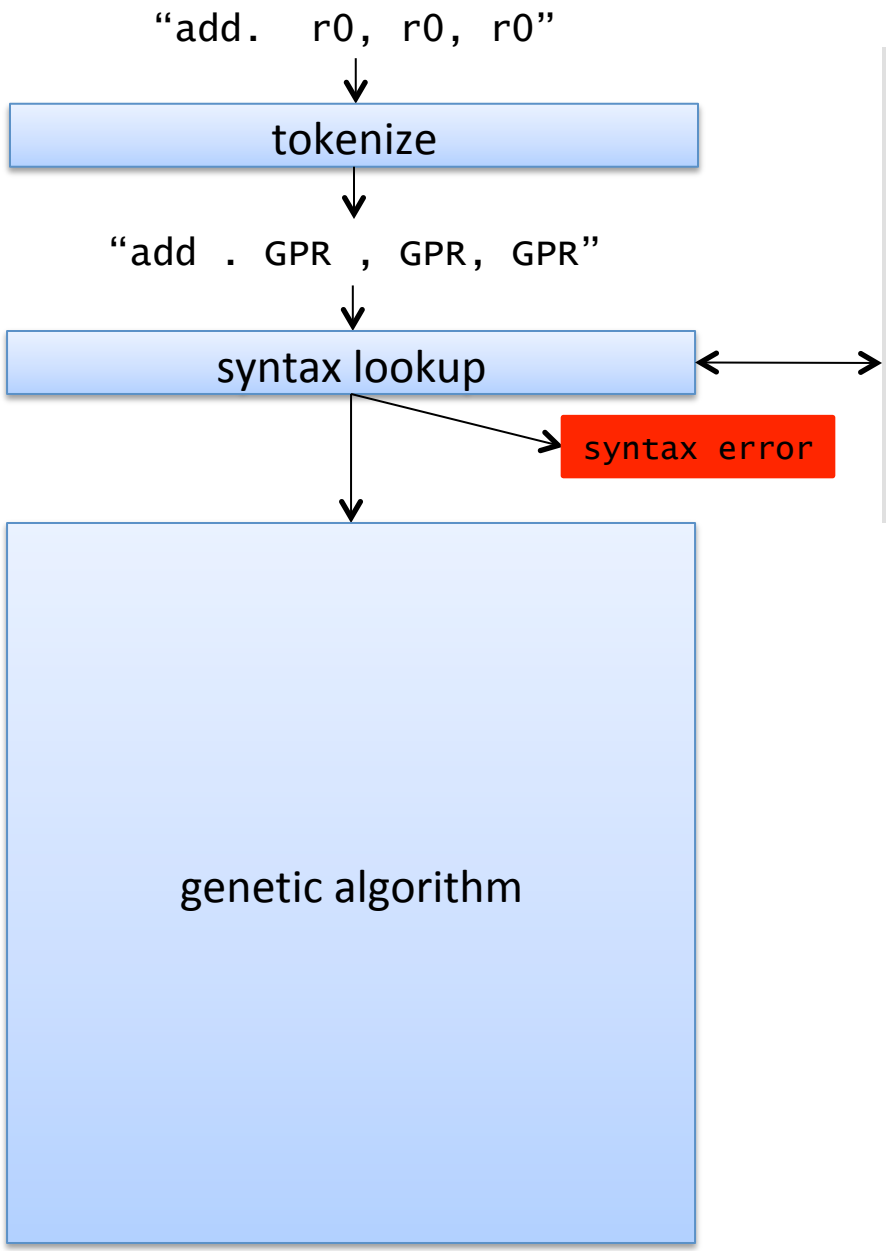
Recap

Wanted to use genetic algorithm, but..

- collected seeds by searching the entire instruction space
- tokenizing diversified the starting points, allowing us to think in terms of instruction syntaxes
- syntaxes yielded a good fitness function that takes into account field types
- fuzzed for bitpatterns which store what bits are worth twiddling

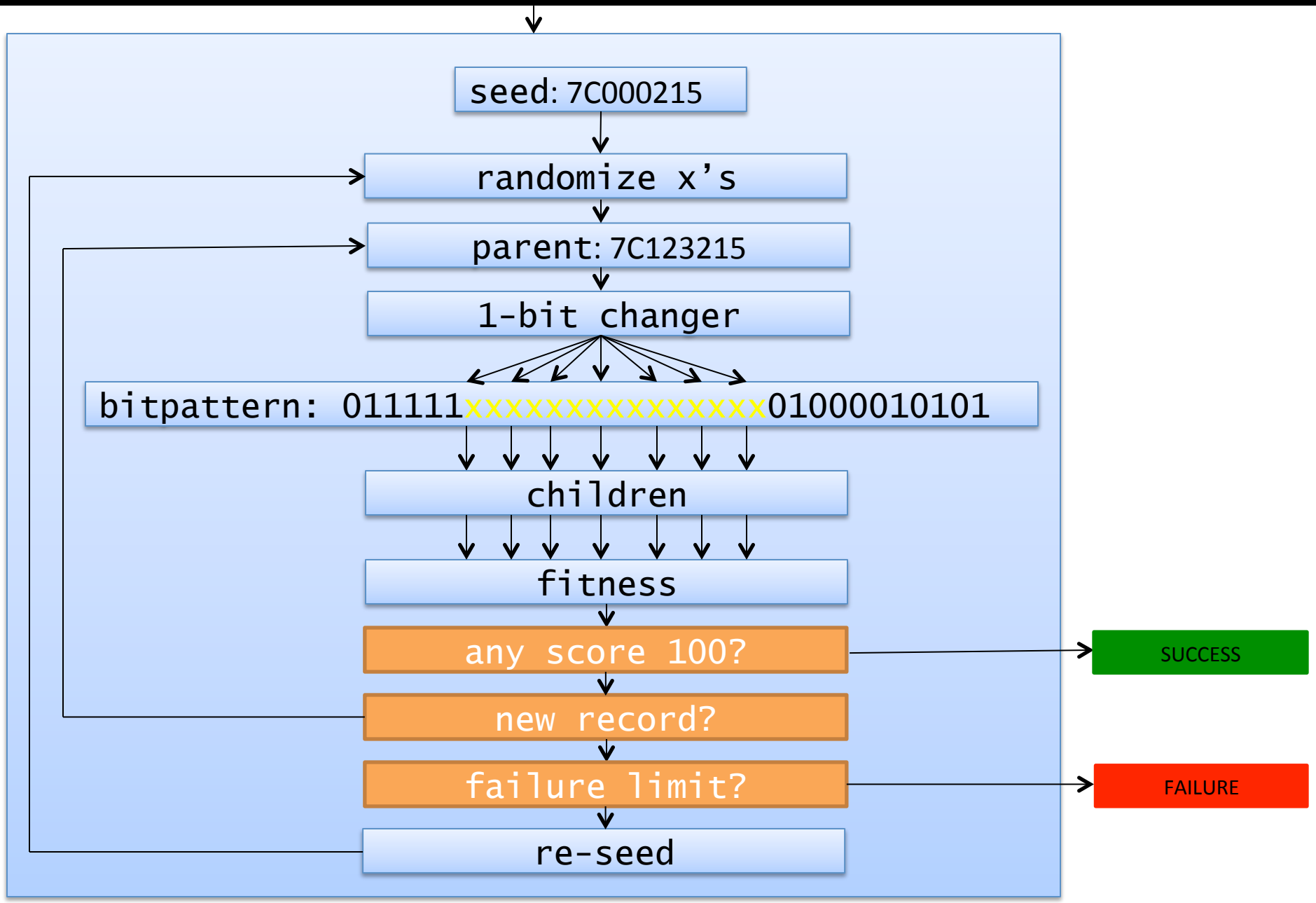
WHAT NOW?

Putting it all together

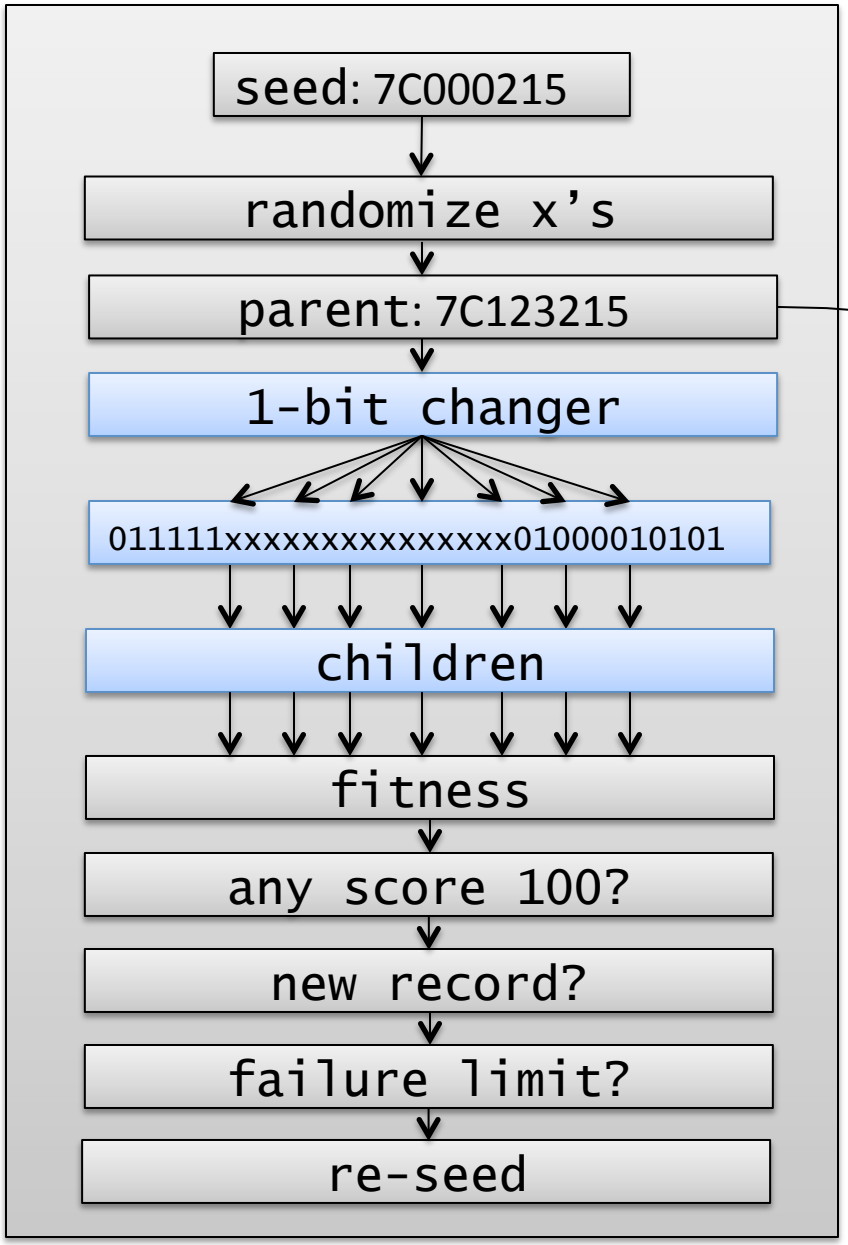


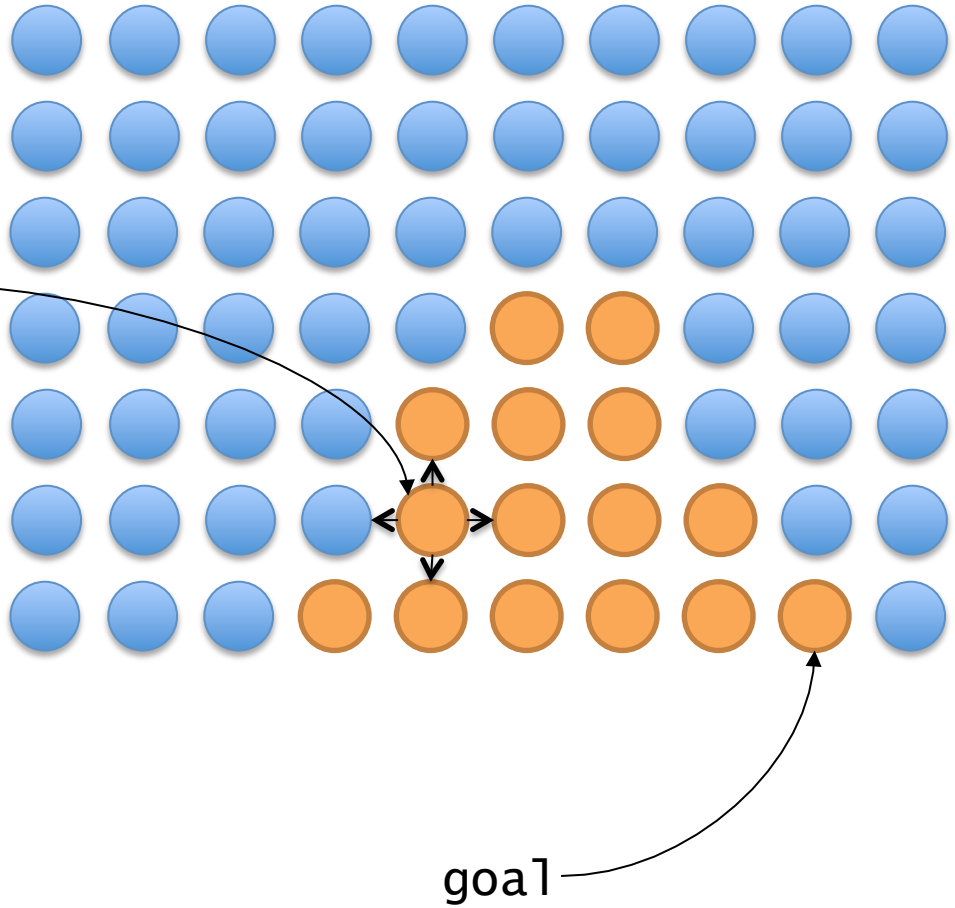

```
 "addic GPR , GPR , NUM",{0x30000000,0x03FFFFFF}},
"addic . GPR , GPR , NUM",{0x34000000,0x03FFFFFF}},
          "li GPR , NUM",{0x38000000,0x03E0FFFF}},
   "addi GPR , GPR , NUM",{0x38010000,0x03FFFFFF}},
         "lis GPR , NUM",{0x3C000000,0x03E0FFFF}},
  "addis GPR , GPR , NUM",{0x3C010000,0x03FFFFFF}},
            "bdnzf FLAG",{0x40000000,0x00230000}},
           "bdnzfl FLAG",{0x40000001,0x00230000}},
           "bdnzfa FLAG",{0x40000002,0x00230000}},
          "bdnzfla FLAG",{0x40000003,0x00230000}},
       "bdnzf FLAG , NUM",{0x40000004,0x0023FFFC}},
…
```

Putting it all together

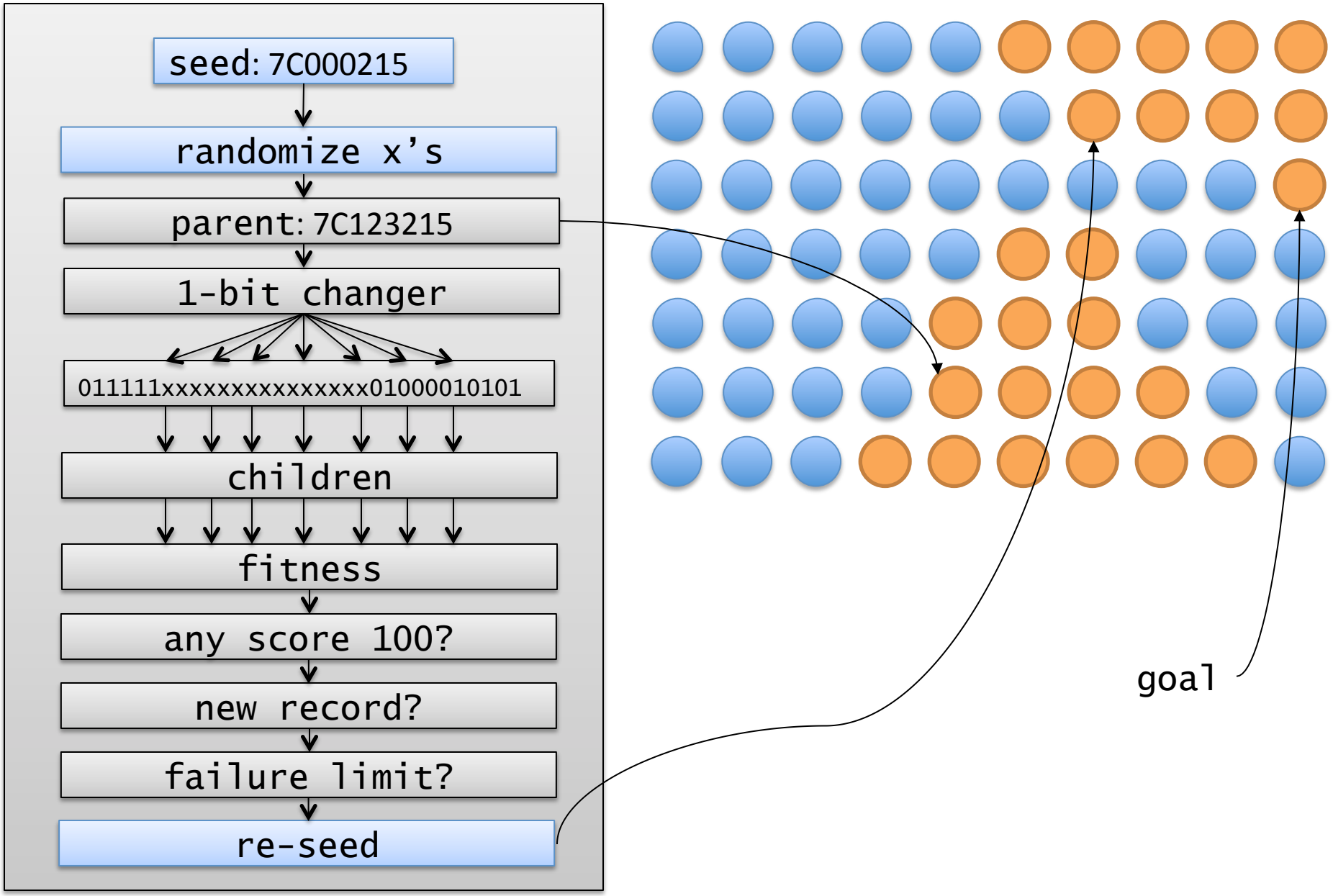


Putting it all together

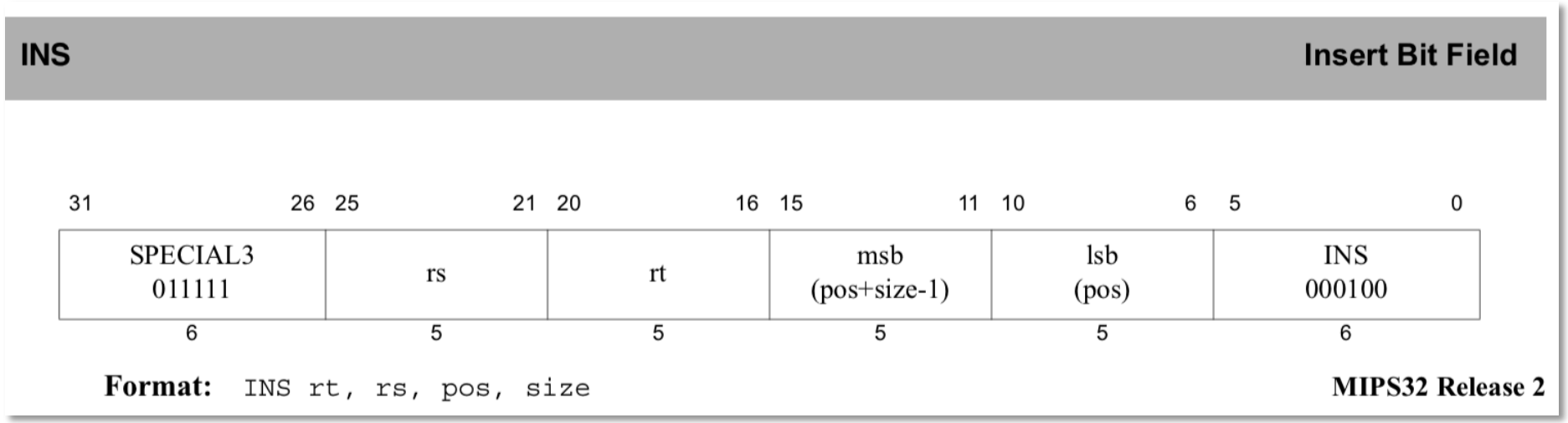




Why is re-seeding needed?



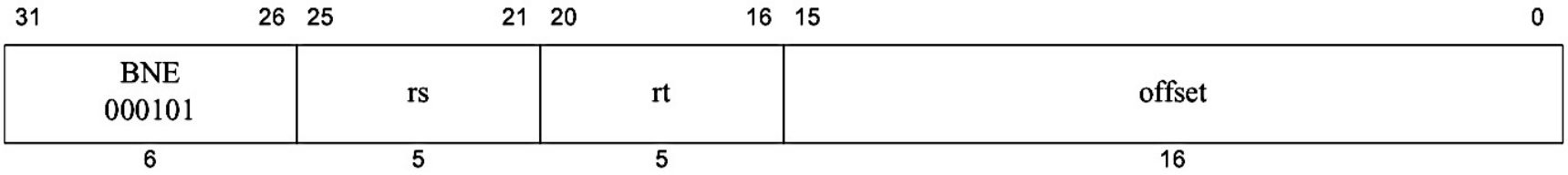
Genetic Hazards: Inter-field Dependencies




```
case 0x7C000004:                                        // INS
{
    insword = 0x7C000004;
    insword |= toks[3].ival << 21;                      // rs
    insword |= toks[1].ival << 16;                      // rt
    insword |= (toks[7].ival + toks[5].ival - 1) << 11; // msb
    insword |= toks[5].ival << 6;                       // lsb
    break;
}
```

Genetic Hazards: Math Encodings

BNE — **Branch on Not Equal**

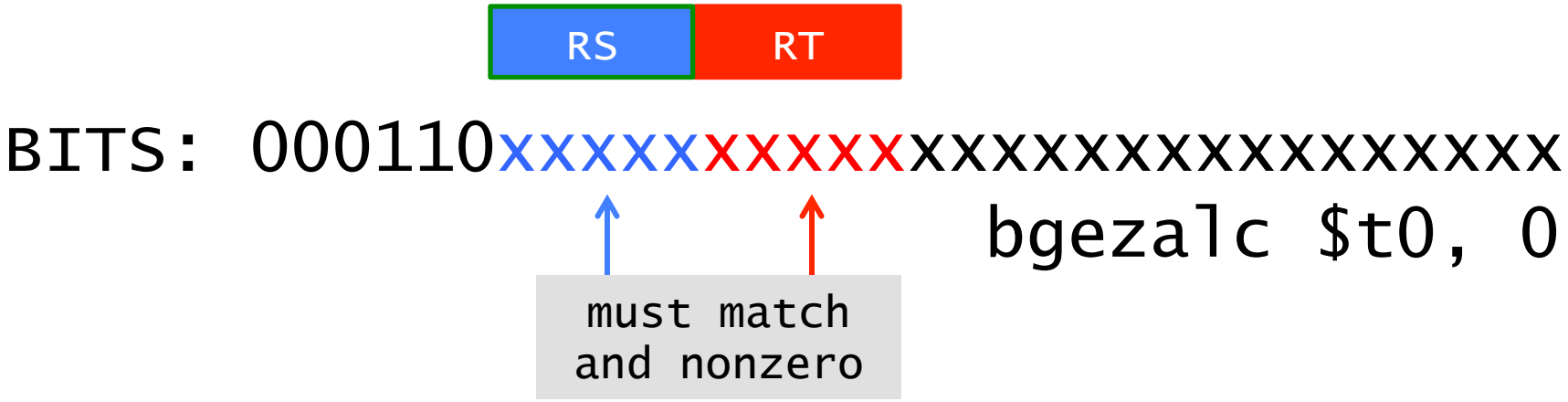


Format: `BNE rs, rt, offset` **MIPS32**

```
EffectiveAddress = CurrentAddress + (offs + 1) << 4;
```

Offset	address
0000	000100 (4)
0001	001000 (8)
0011	010000 (16)
0010	001100 (12)
0110	011100 (28)
0111	100100 (36)
0101	011000 (24)
0100	010100 (20)

Genetic Hazards: Inter-field Dependencies

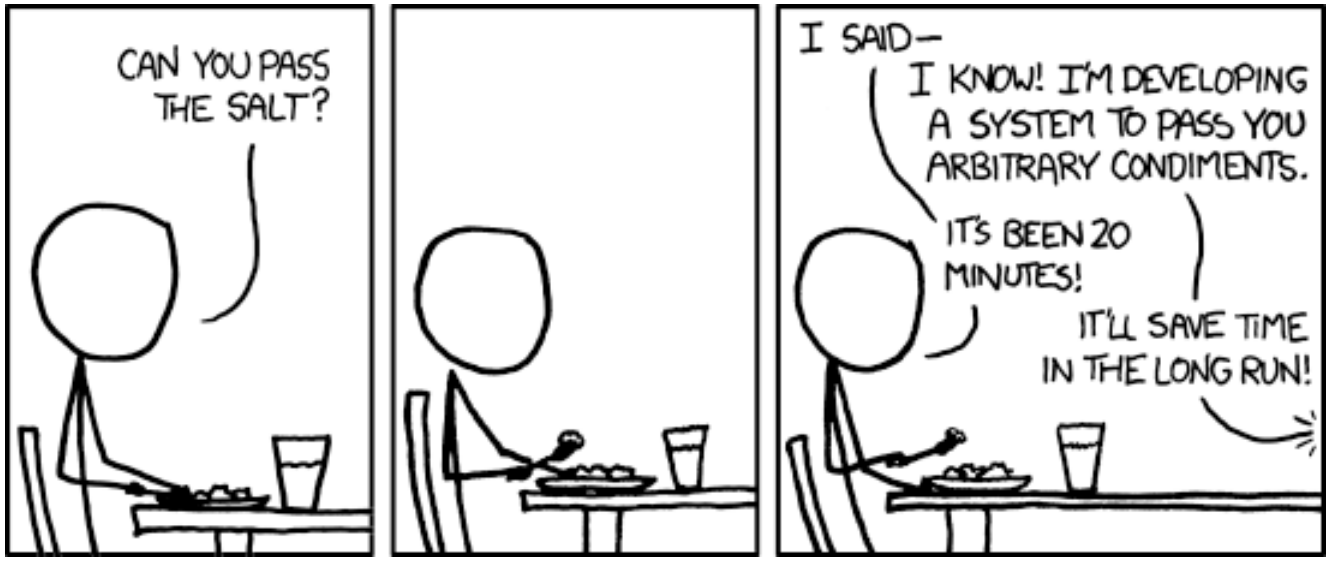


PPC Special Cases:
sldi
srwi
slwi
crclr
xxspltd
xxswapd
xvmovdp
xvmovsp
mr
tdi
twi

MIPS Special Cases:
ins
ext
bgezc
bgezalc
bltzalc
bltz
any bxx 16-bit

OBA Questions

1. Is time actually saved?



https://xkcd.com/974/

2. Does the approach fulfill Binary Ninja's needs?

Pros/Cons of this approach

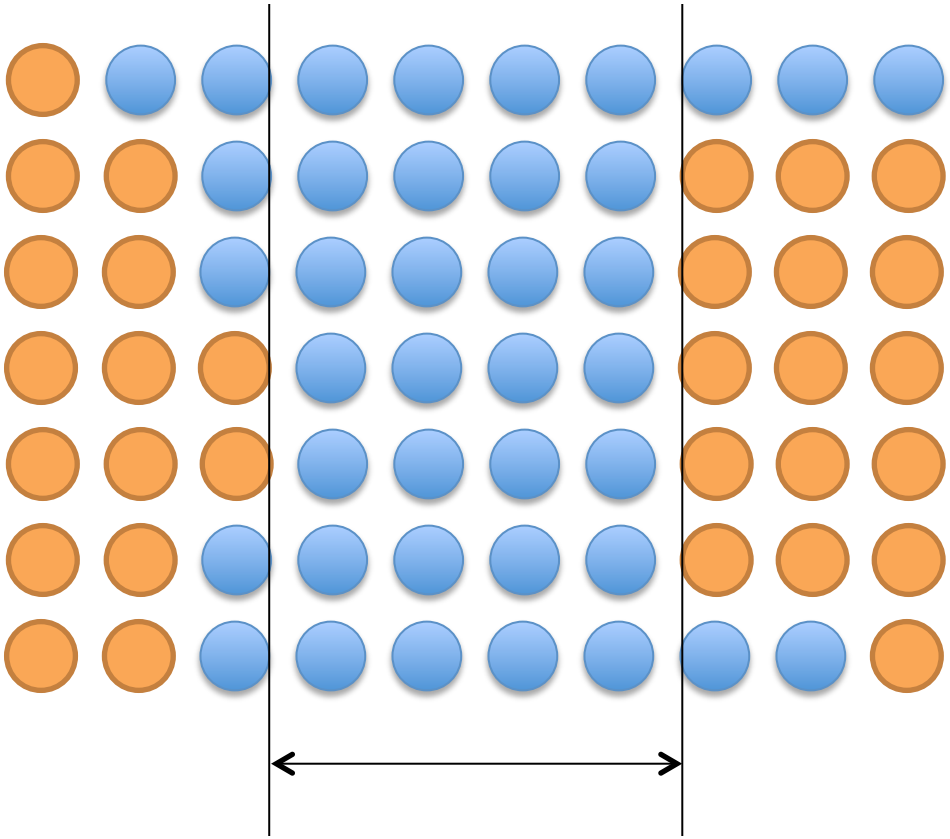
Pros

- easier, faster, and more interesting to write
- deniability: “it’s the disassembler’s fault!”
- provable for small instruction spaces

Cons

- slower
- less feedback on errors
- more difficult to maintain

Generalize where this will work?



One Guess: The suitability of an architecture to this approach related to:

- How many single instructions split into multiple islands
- By how much distance are those islands separated?

Works On What Architectures?

Another Guess: Suitability is inversely related to the “avalanche effect” present in the ISA’s encodings

- What percentage of instructions improve with the 1-bit guesser?
- With the 2-bit guesser?

What does the future hold?

- tackle more difficult spaces: ARM, THUMB, Intel
- unify the PPC and MIPS versions
- pure python version, submit to Python Package Index (PyPI)
- the full exhaustive test of all MIPS
- Automate the entire task! Evolve an assembler that uses an evolutionary algorithm given a disassembler.

The End!

Questions?

https://github.com/lwerdna/generate_assembler