

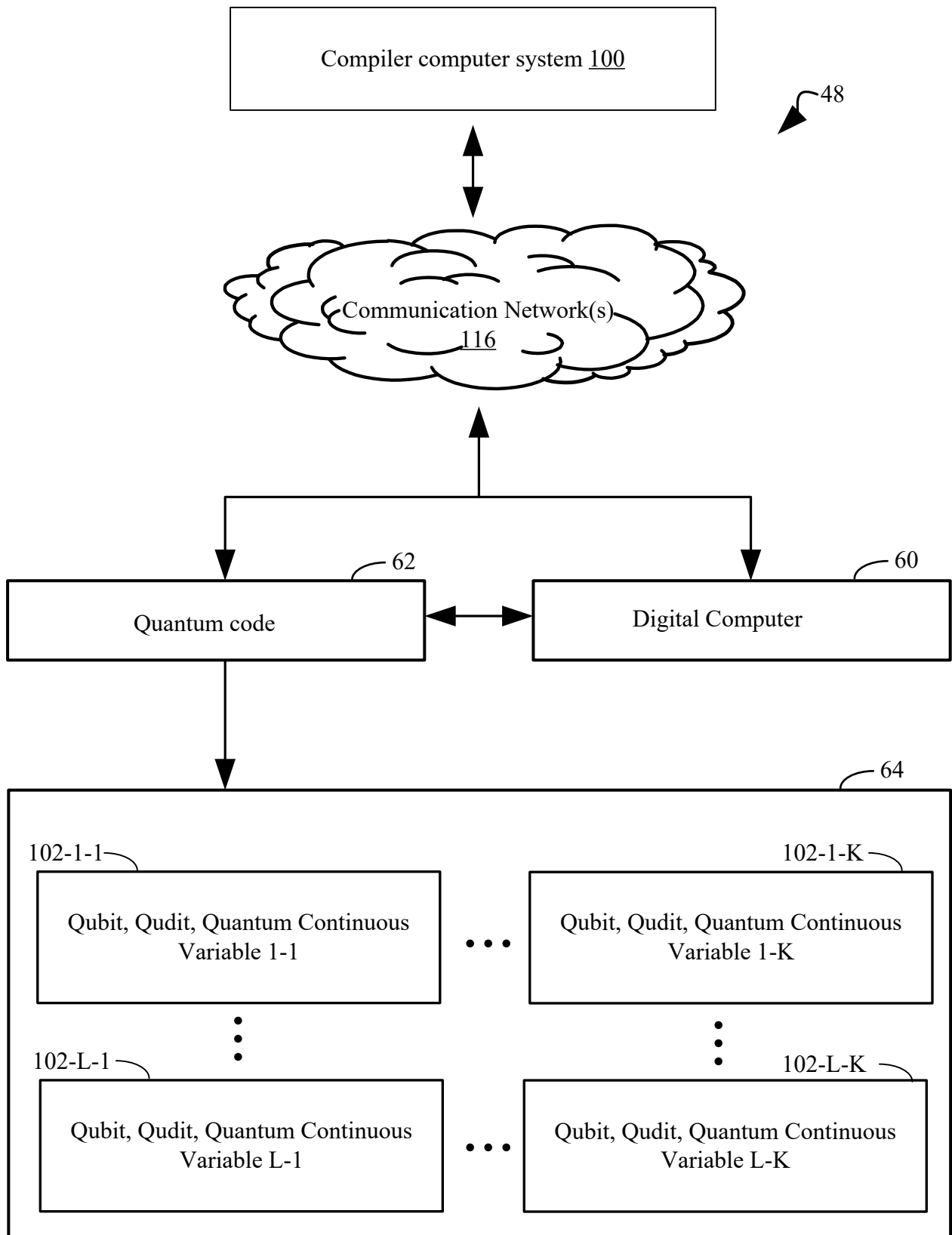


Figure 1

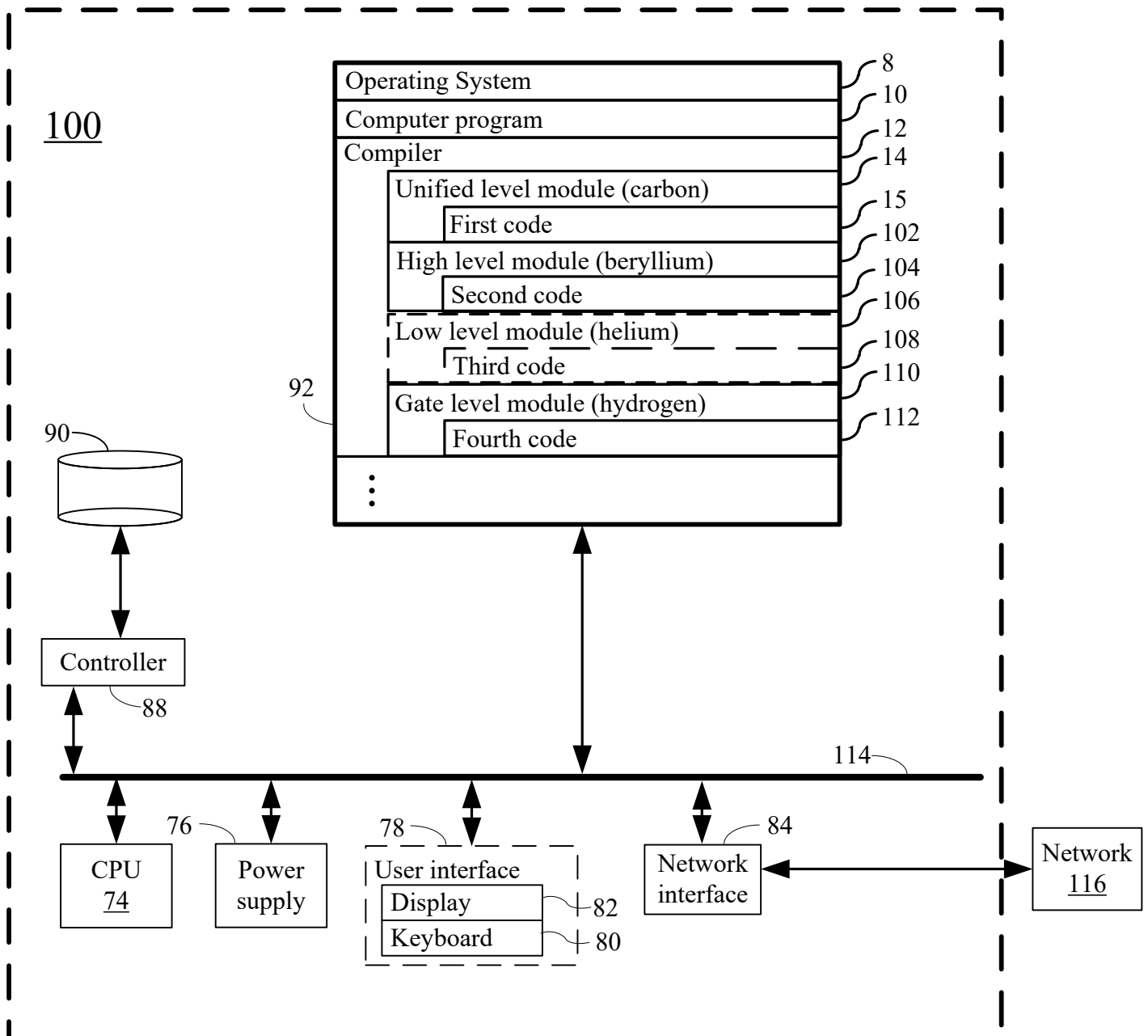


Figure 2

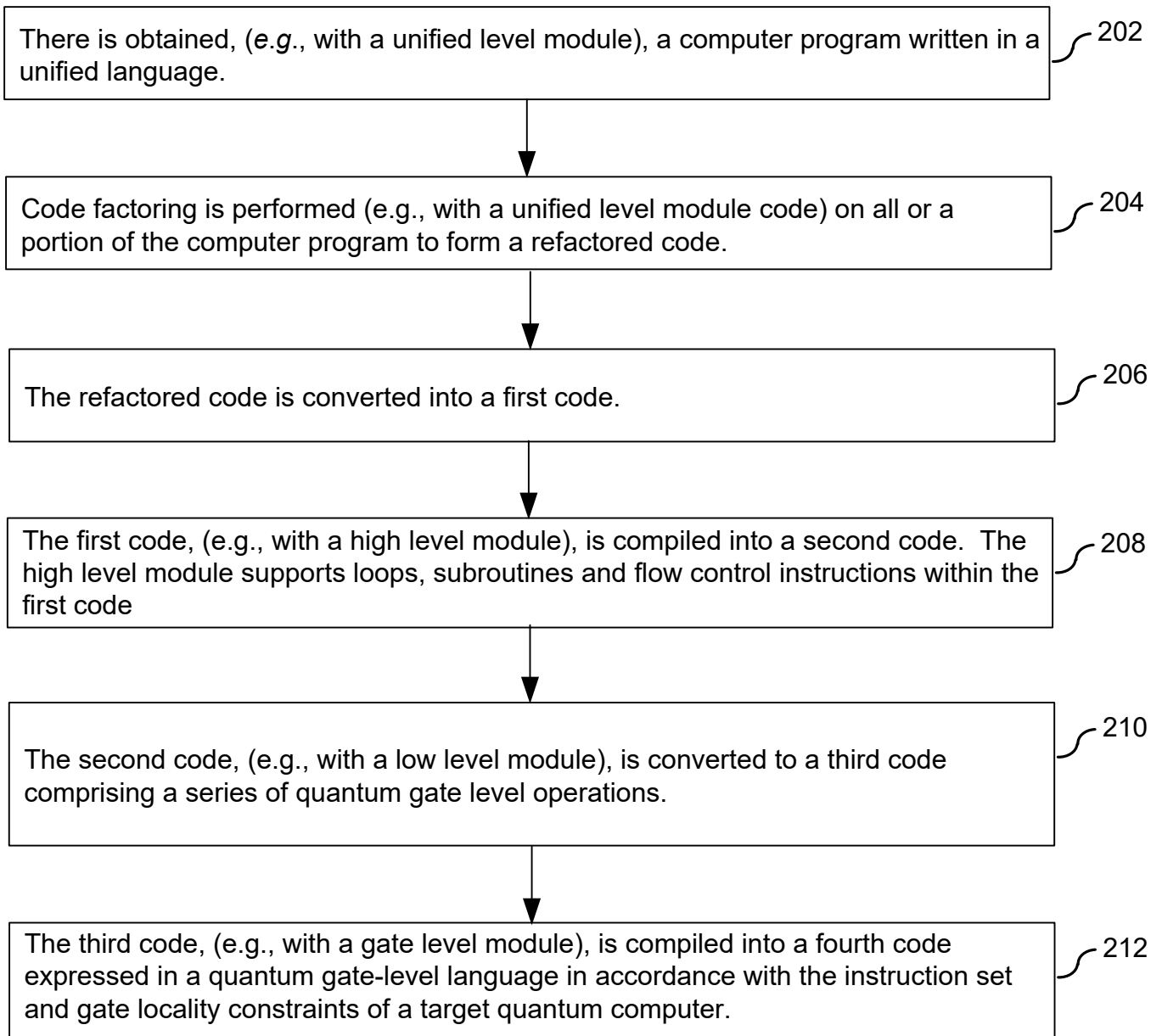
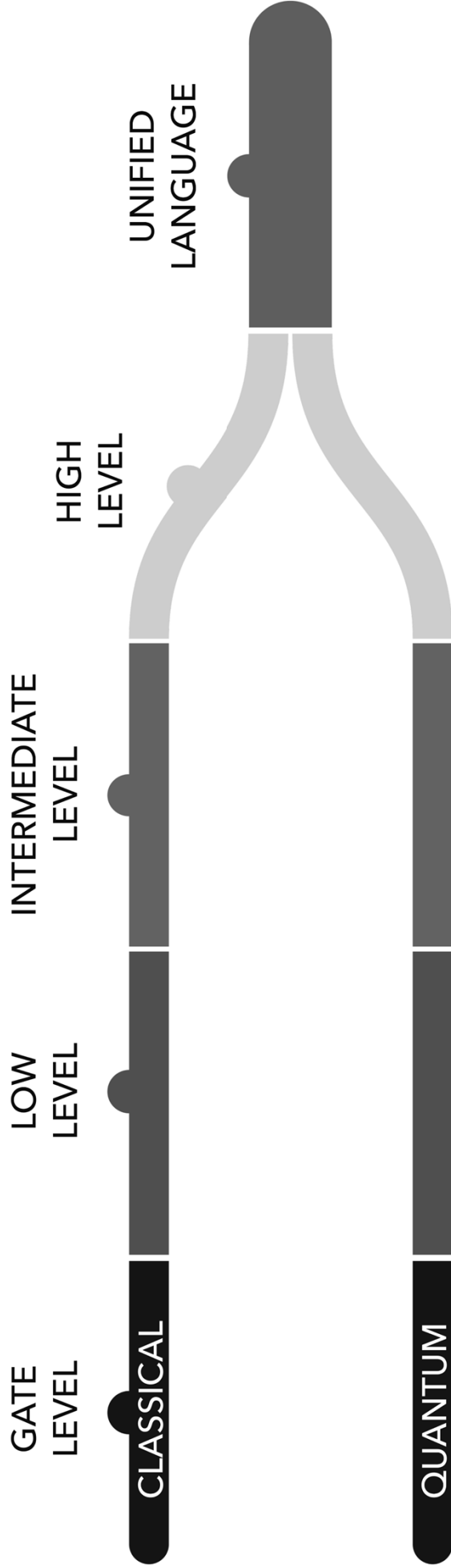


Figure 3



A language that can be compiled and run on classical or quantum processors

Figure 4

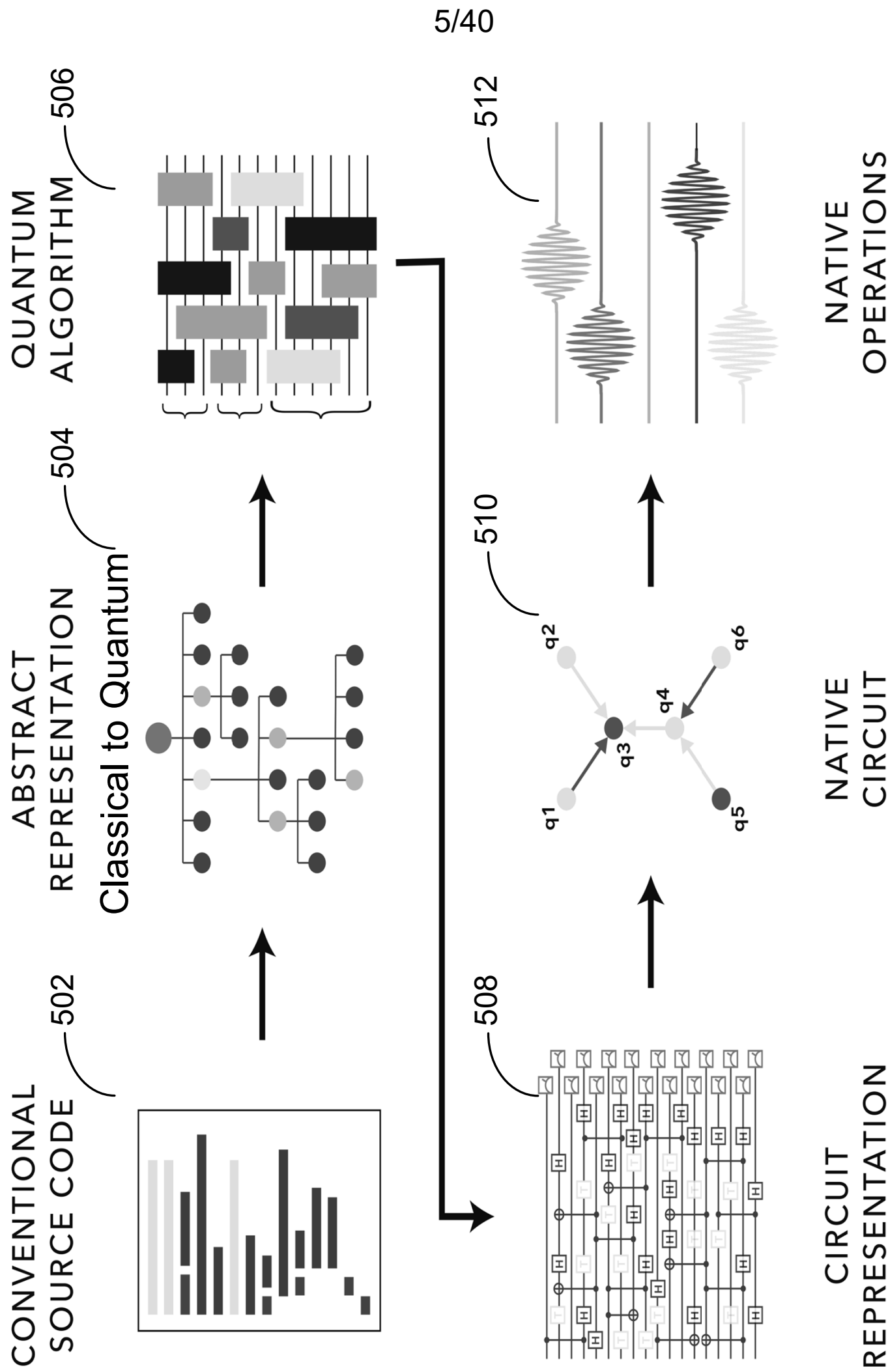


Figure 5

COMPILER STRUCTURE

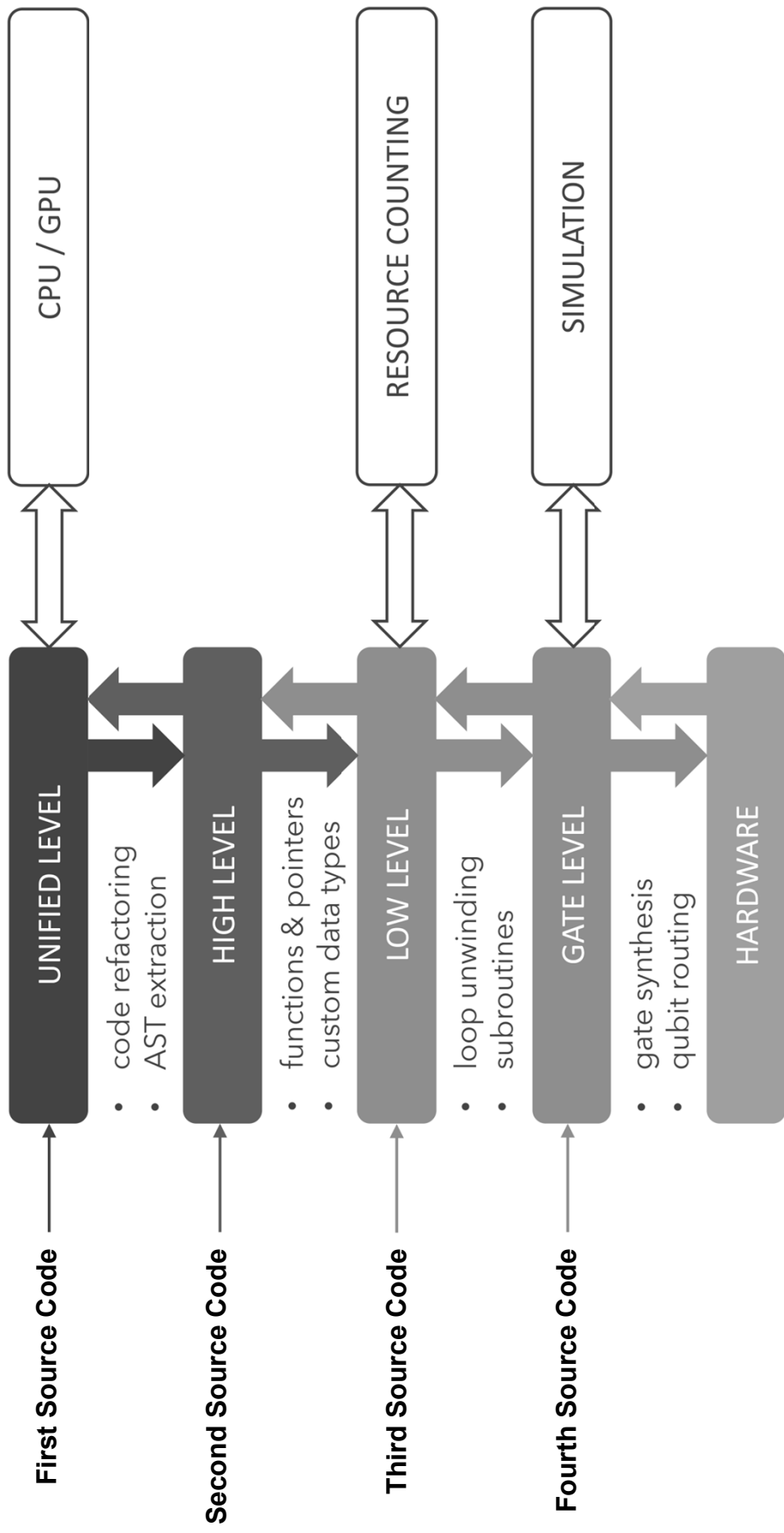


Figure 6

Compiler Levels

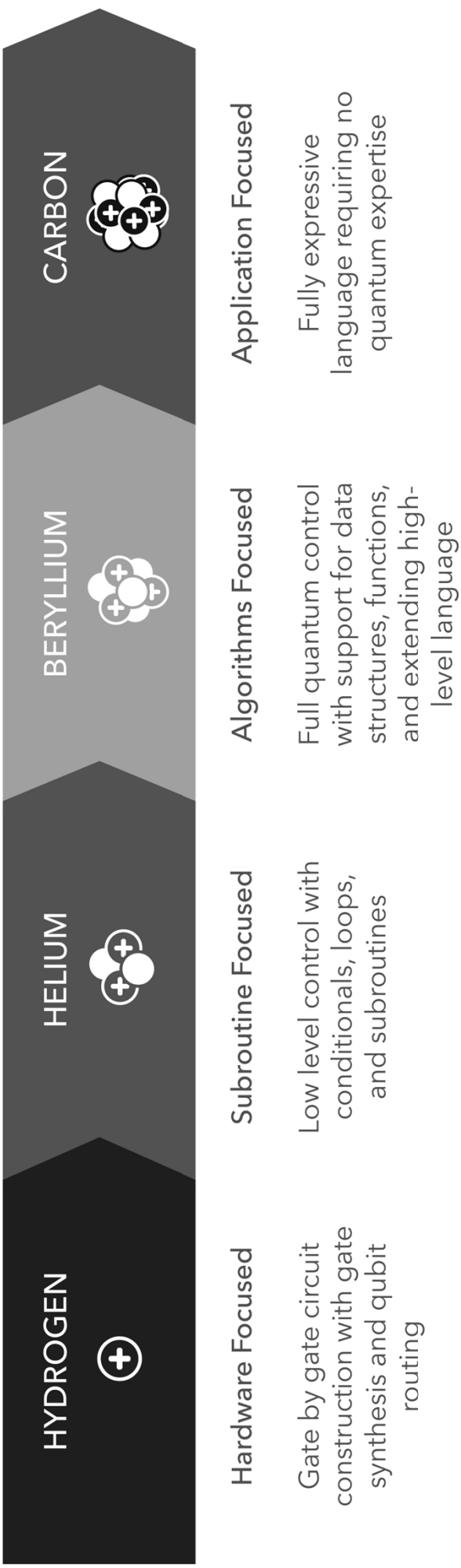


Figure 7

HYDROGEN CODE



Fully customizable gate-level language



Focused specifically on quantum processing



Gates and operations defined by structured text



Compiler handles qubit routing and gate bridging



Inbuilt gate synthesis to convert between gate sets



Designed to integrate custom characterization to maximize performance

```
1 # Prepare two qubits
2 qubit -> a
3 qubit -> b
4
5 # Entangle qubits
6 h[a]
7 cnot[a,b]
8
9 # Measure first qubit
10 h[a]
11 measure[a] -> u
12
13 # Measure second qubit
14 rz(0.3927)[b]
15 measure[b] -> v
```

Figure 8

902

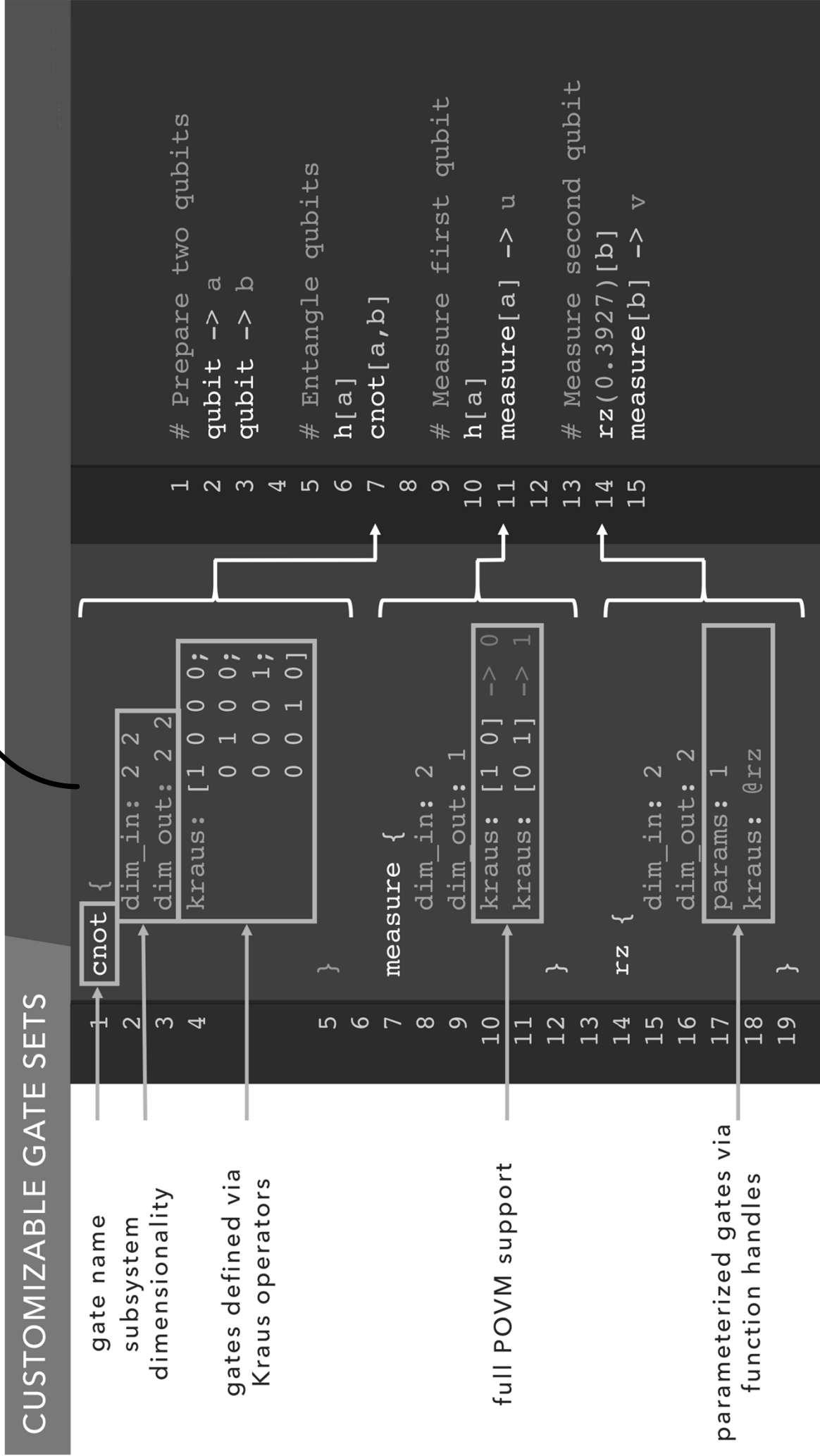


Figure 9

GATE SYNTHESIS

Gate synthesis allows for swapping between gate sets, and approximating one gate set with another.

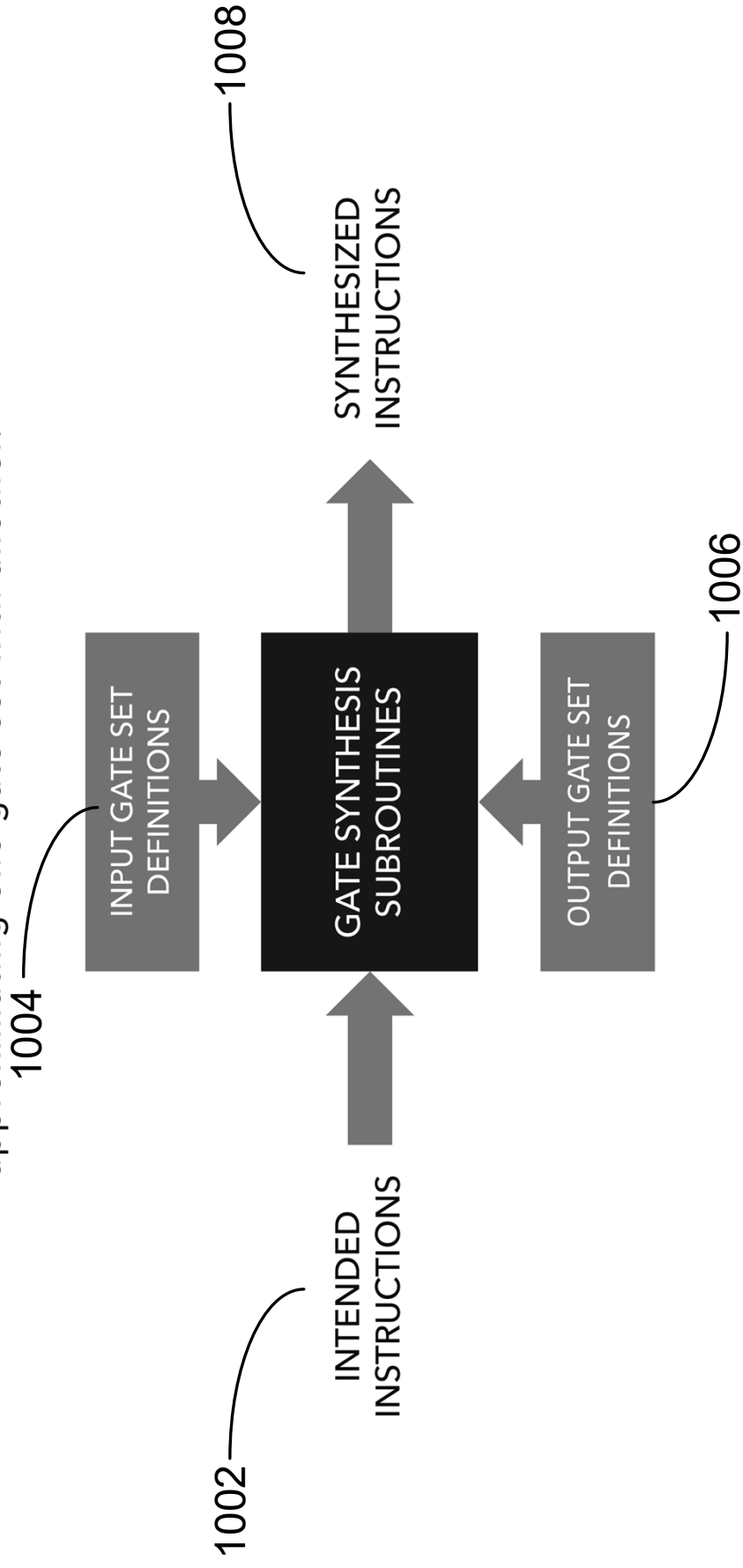
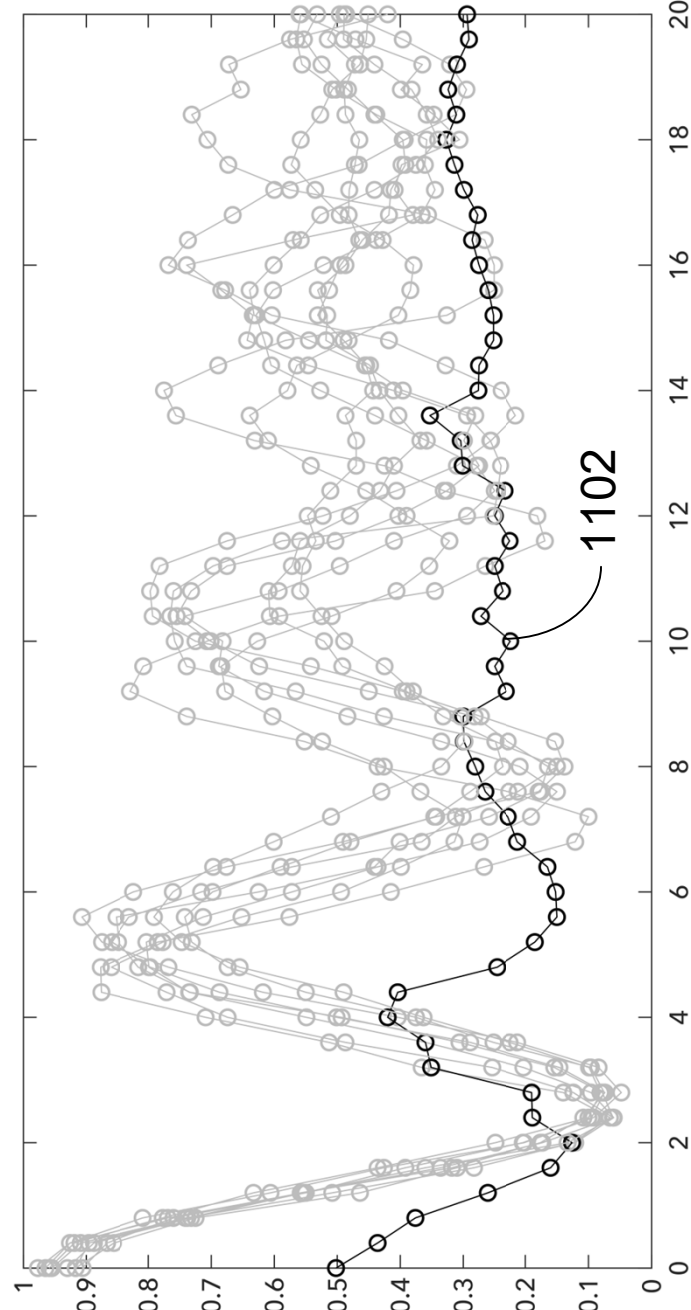


Figure 10

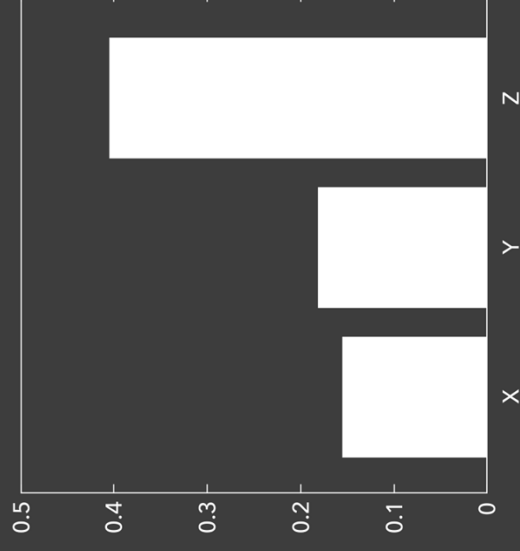
CALIBRATION

Bad qubits?



Poor performance in benchmarks can be caused by correctable errors.

ANATOMY OF A BAD QUBIT



PURITY : 0.94

* data from Rigetti Q32
Aspen-7 QPU

Figure 11

MEASURING STRAY COUPLINGS

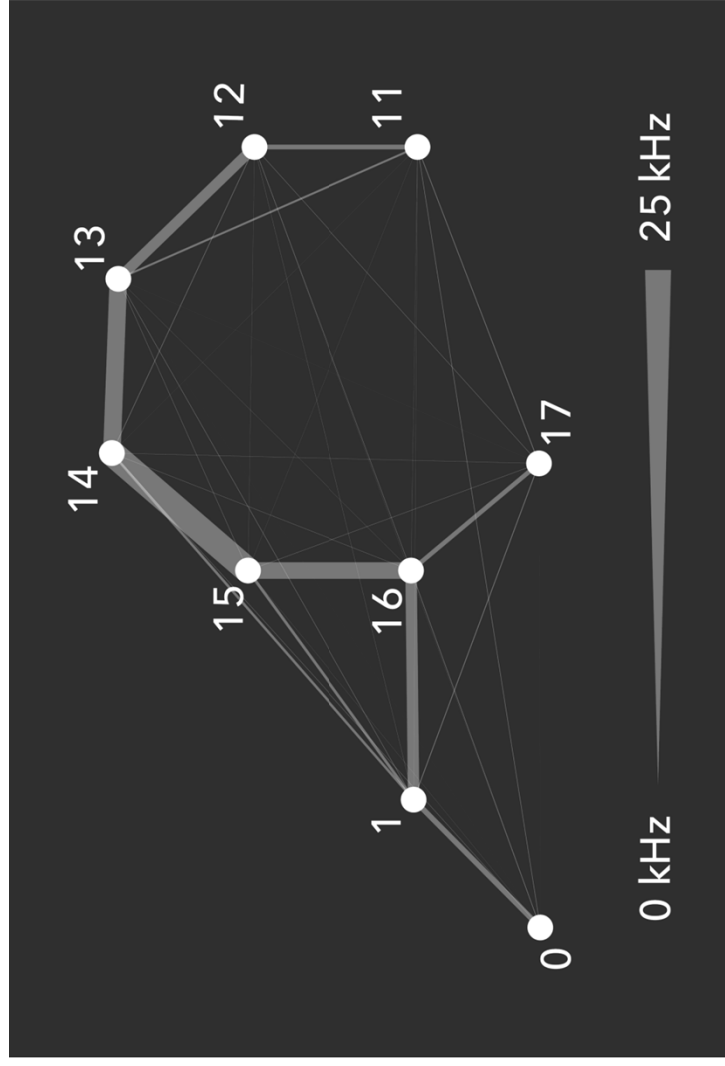
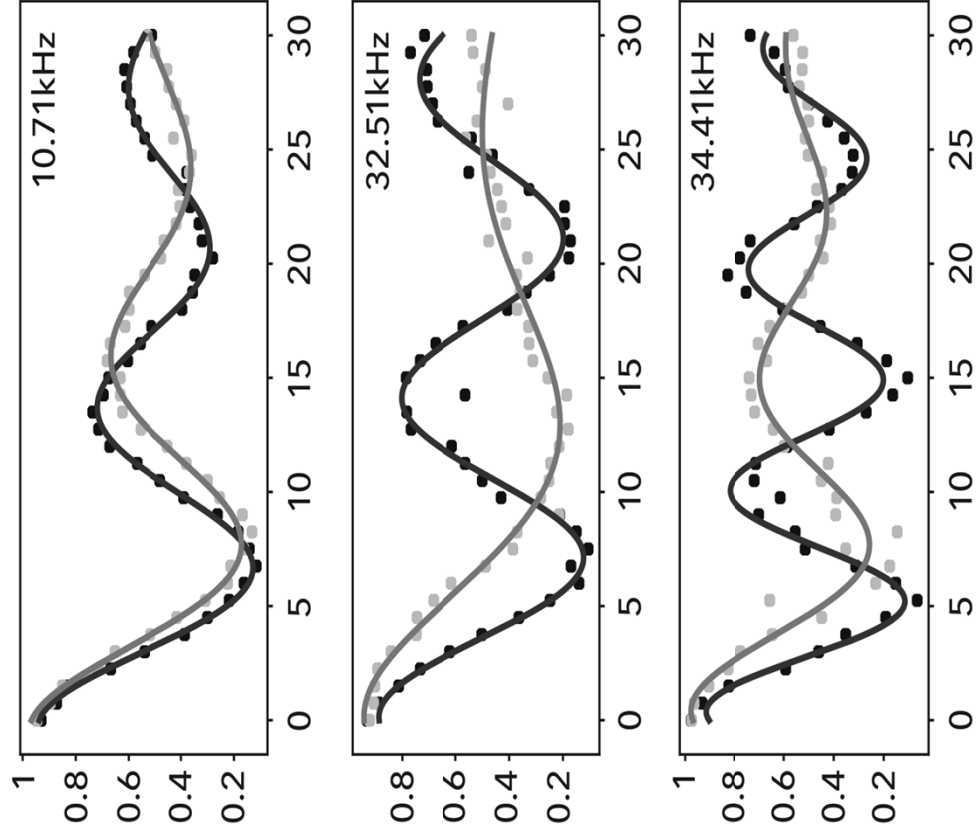


Figure 12

HELIUM CODE

```
1  qubit -> a
2  qubits(8) -> b
3  qubit -> c
4
5  if [a]
6      cnot[a,b]
7  end
8
9  repeat
10     gosub entangle
11 until [a]
12
13 sub entangle
14     increment k to 8
15     h[b(k)]
16     cnot[b(k),a]
17 next
18 endsub
```

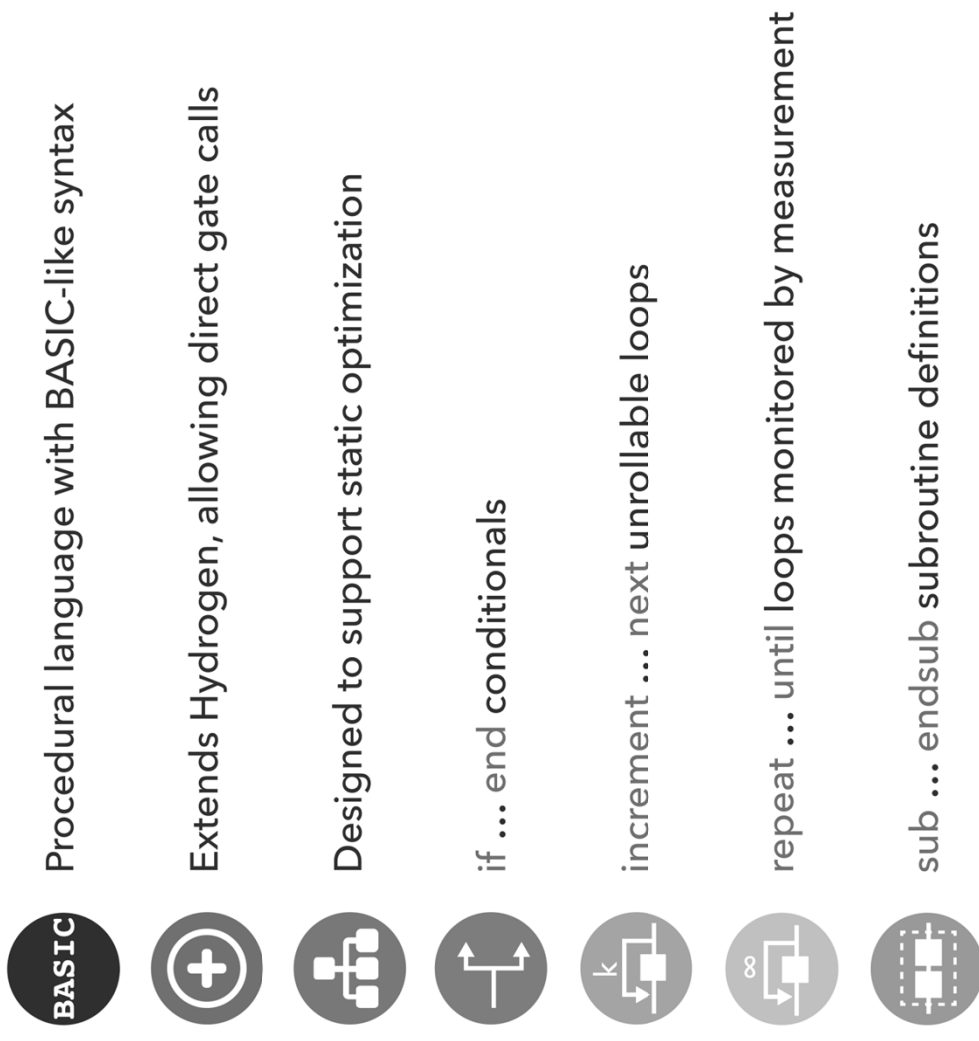
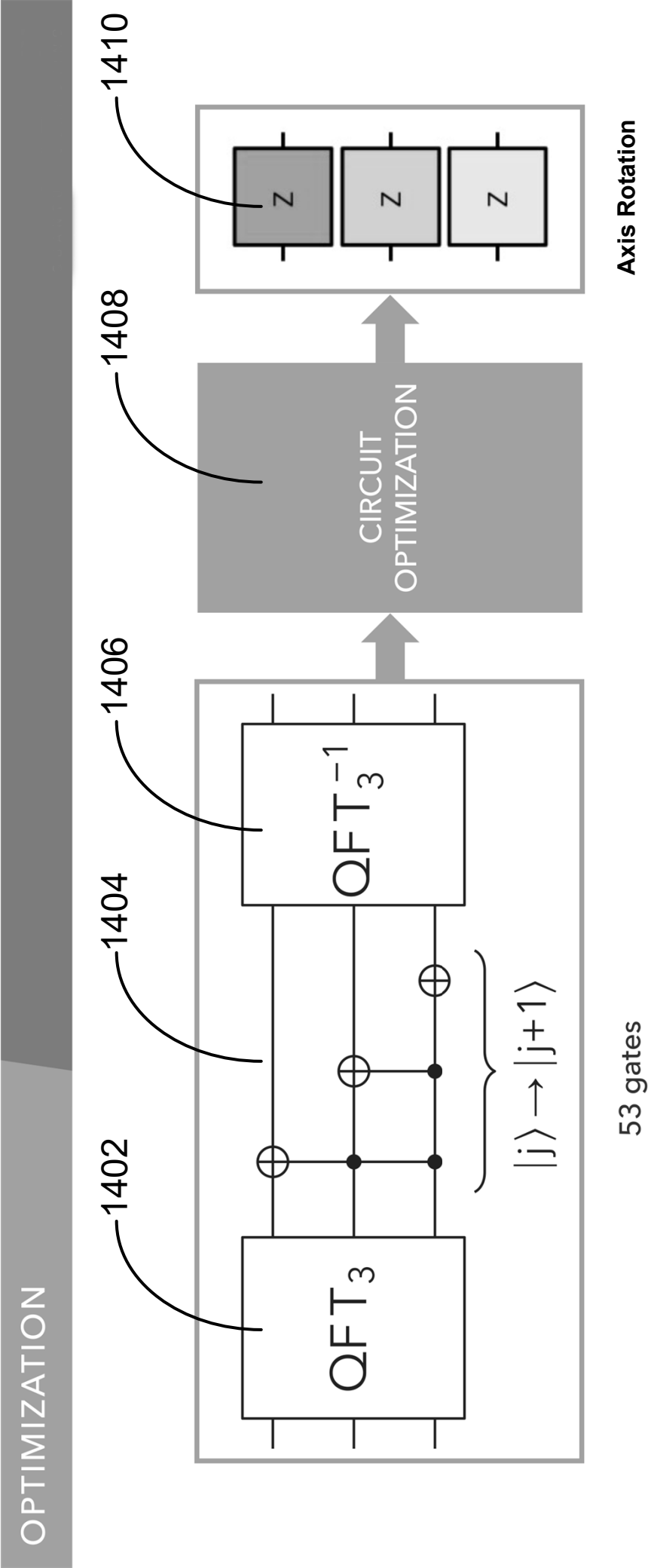


Figure 13



* real results from optimization subroutines

Figure 14

BERYLLIUM CODE

```
1 # A simple function to measure qubits
2
3 function Meas()[reg,angle] ->
4   (output) {
5
6     qubits(reg.length) -> ancillas
7
8     for k in 1:reg.length {
9       [ancilla(k)] <- 0
10      rz([angle(k)])[reg(k)]
11      cnot[reg(k), ancillas(k)]
12      measure[ancillas(k)] -> output(k)
13    }
14 }
```

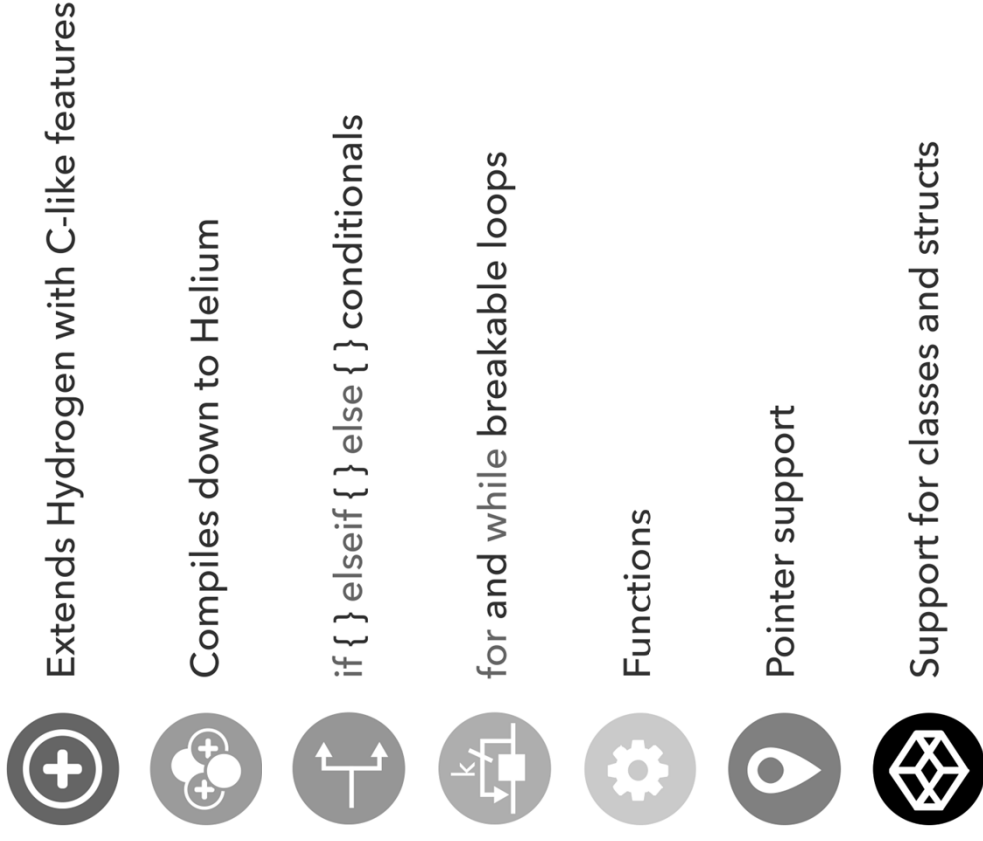


Figure 15

POINTERS

Memory locations can be passed directly as parameters, with the bit string at the memory location interpreted as an input parameter for the gate.



The compiler generates a unitary quantum circuit to realize this operation.

```

1  rz(-0.98175)[1]
2  cnot[2,1]
3  rz(-0.58905)[1]
4  cnot[3,1]
5  rz(0.19635)[1]
6  cnot[2,1]
7  rz(-0.19635)[1]
8  cnot[3,1]
9  rz(-0.19635)[2]
10 cnot[3,2]
11 rz(0.19635)[2]
12 cnot[3,2]
13 rz(0.19635)[3]
14 h[3]
15 rz(0.7854)[3]
16 h[3]
17 cnot[2,1]
18 cnot[3,1]
19 rz(0.5236)[1]
20 cnot[2,1]
21 cnot[3,1]
22 rz(-0.5236)[3]
23 h[3]
```

Figure 16

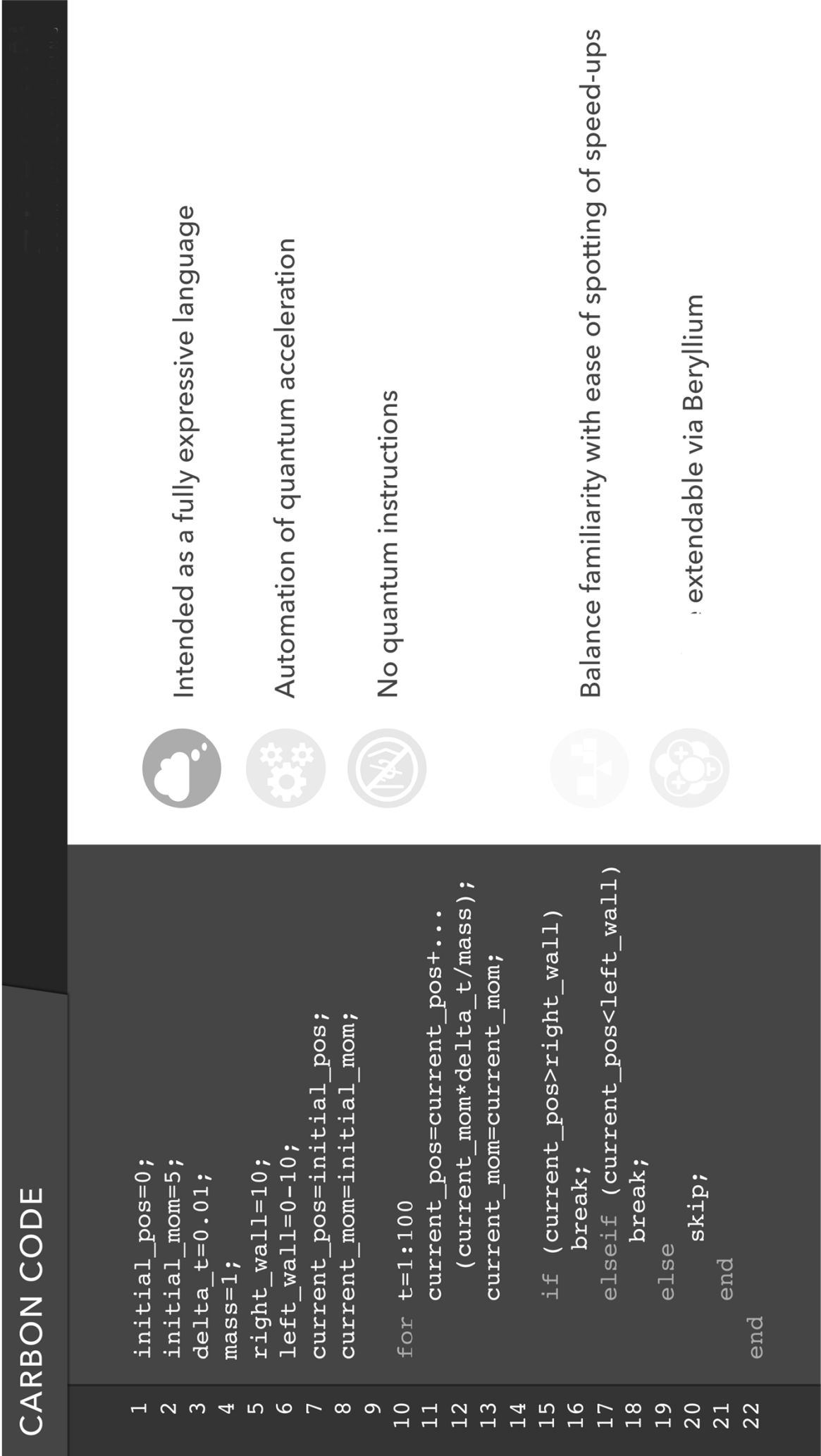
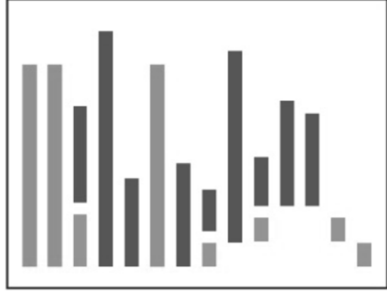


Figure 17

SOURCES OF COMPLEXITY

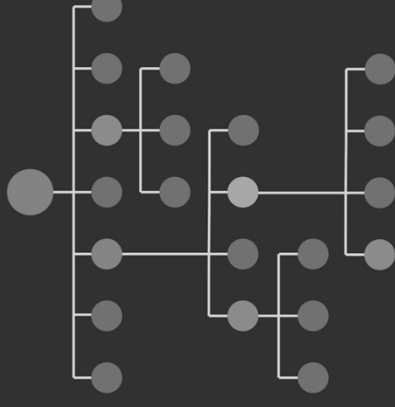
EXPLICIT COMPLEXITY



Emerges from the explicit flow-control instructions entered by the programmer.

REFACTOR to take advantage of quantum algorithms for specific tasks.

IMPLICIT COMPLEXITY



Emerges from the use of data types that carry non-trivial cost to manipulate.

Implement data types using QUANTUM REPRESENTATIONS (using Beryllium).

Figure 18

CODE REFACTORING

```
1 heat_map=zeros(10000,10000);
2
3 for i=2:9999
4     for j=2:9999
5         heat_map(i,j)=(heat_map(i-1,j)/4)+(heat_map(i,j-1)/4)+(heat_map(i,j+1)/4);
6     end
7 end
8
9 extrema=0;
10 where_hottest=0;
11 row_to_search=5000;
12
13 for i=1:10000
14     heat_map_value=heat_map(row_to_search,i);
15     if heat_map_value>extrema
16         extrema=heat_map_value;
17         where_hottest=i;
18     end
19 end
20
21
22
23
24 for i=1:10000
25     if heat_map(i,i)>5
26         break;
27     end
28 end
```

Figure 19

CODE REFACTORING

```
1  heat_map=zeros(10000,10000);
2
3  for i=2:9999
4      for j=2:9999
5          heat_map(i,j)=(heat_map(i-1,j)/4)+(heat_map(i+1,j)/4)+(heat_map(i,j-1)/4)+(heat_map(i,j+1)/4);
6      end
7  end
8
9  extrema=0;
10 where_hottest=0;
11 row_to_search=5000;
12
13 for i=1:10000
14     heat_map_value=heat_map(row_to_search,i);
15 end
16
17 for i=1:10000
18     if extrema<heat_map(row_to_search,i);
19         extrema=heat_map(row_to_search,i);
20         where_hottest=i;
21     end
22 end
23
24 for i=1:10000
25     if 5<heat_map(i,i)
26         break;
27     end
28 end
```

Loop split and simplified

Reordered comparisons

Figure 20

CODE REFACTORING

1 heat_map=zeros(10000,10000);

2
3 for i=2:9999
4 for j=2:9999
5 heat_map(i,j)=(heat_map(i-1,j)/4)+(heat_map(i+1,j)/4)+(heat_map(i,j-1)/4)+(heat_map(i,j+1)/4);
6 end
7 end

Exponential speed-up

8
9 extrema=0;
10 where_hottest=0;
11 row_to_search=5000;

12
13 for i=1:10000
14 heat_map_value=heat_map(row_to_search,i);
15 end

16
17 for i=1:10000
18 if extrema<heat_map(row_to_search,i);
19 extrema=heat_map(row_to_search,i);
20 where_hottest=i;
21 end
22 end

23
24 for i=1:10000
25 if 5<heat_map(i,i)
26 break;
27 end
28 end

2102

2104

2106

2108

Analytically solved
Exponential speed-up

Max value search
Polynomial speed-up

Sequential search
Polynomial speed-up

Figure 21

EVOLUTION OF LANGUAGES

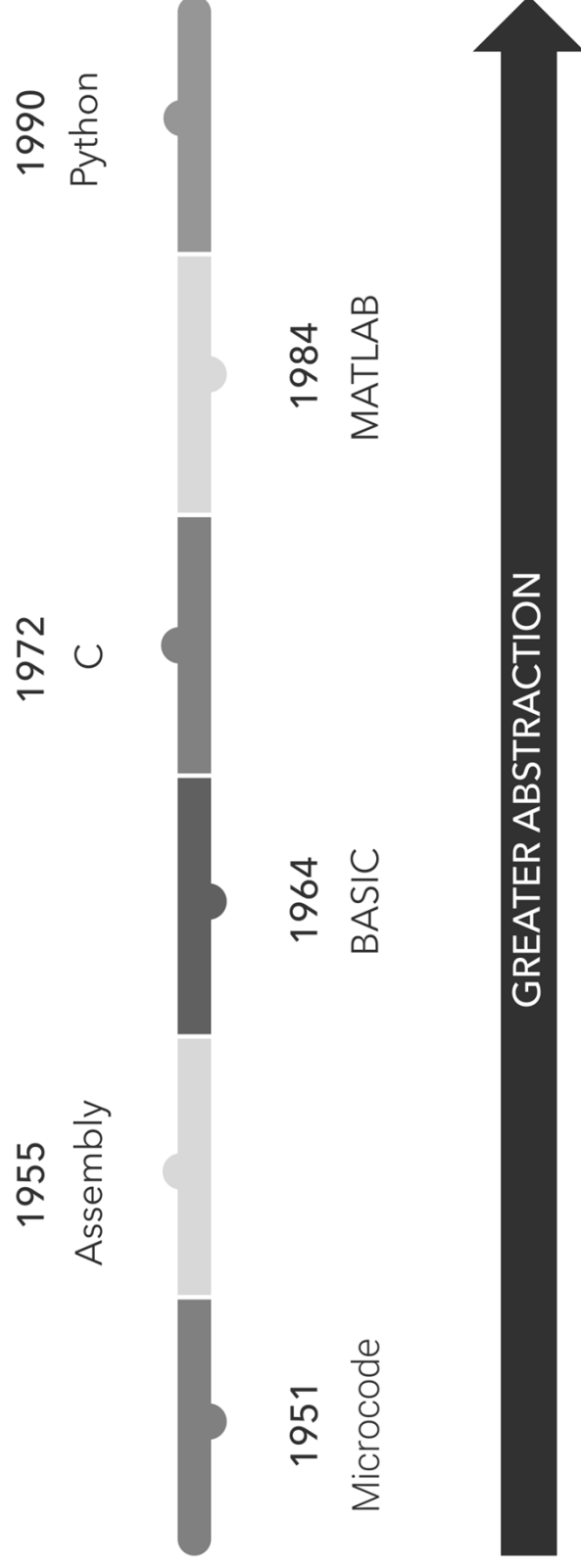


Figure 22
(Prior Art)

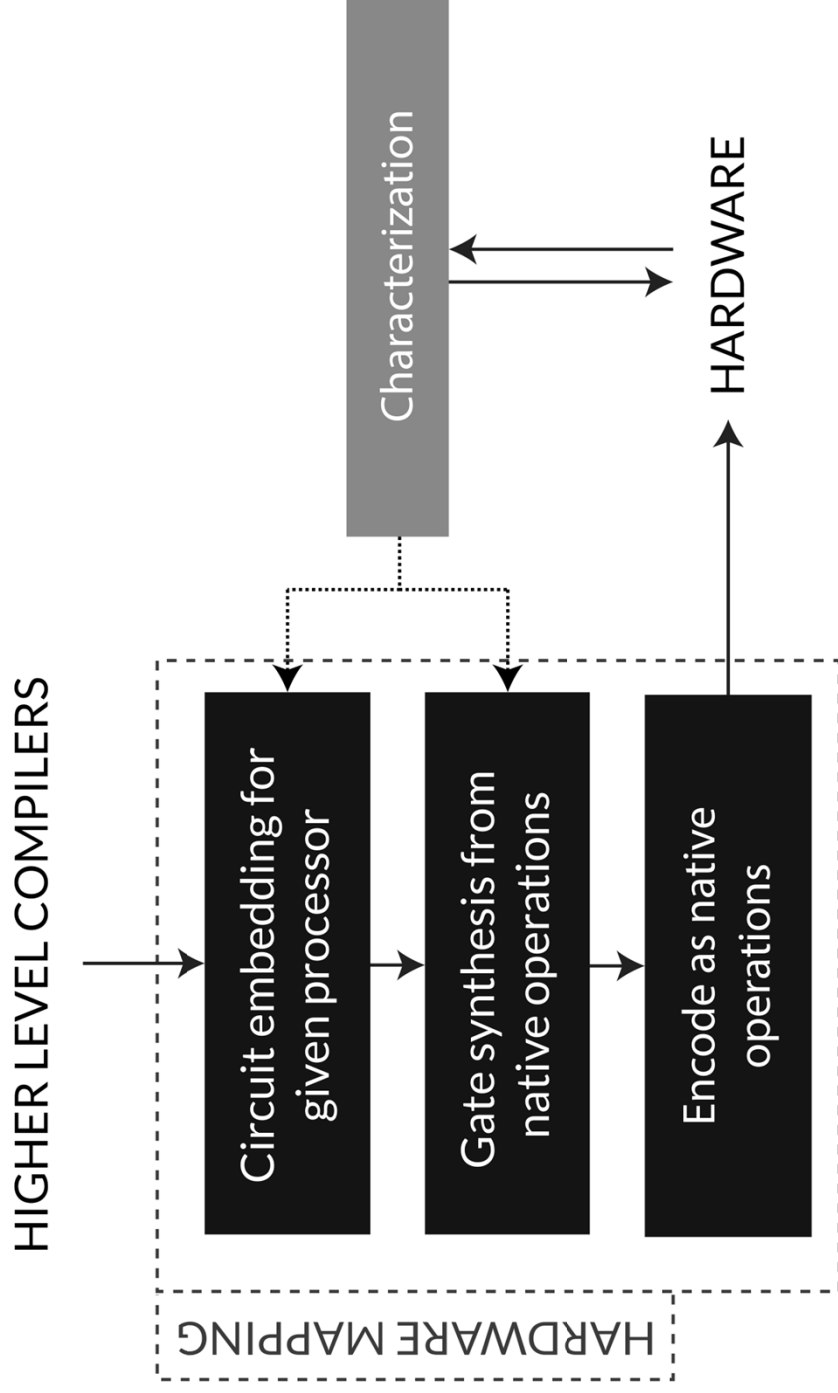


Figure 23

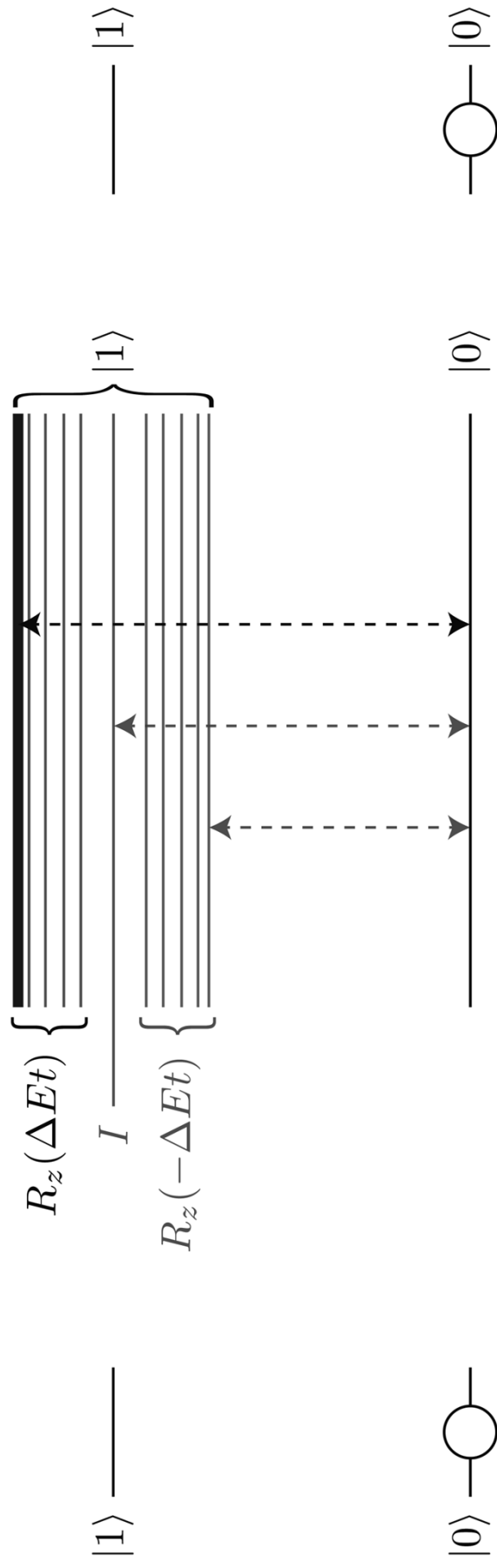


Figure 24

MITIGATING ERRORS

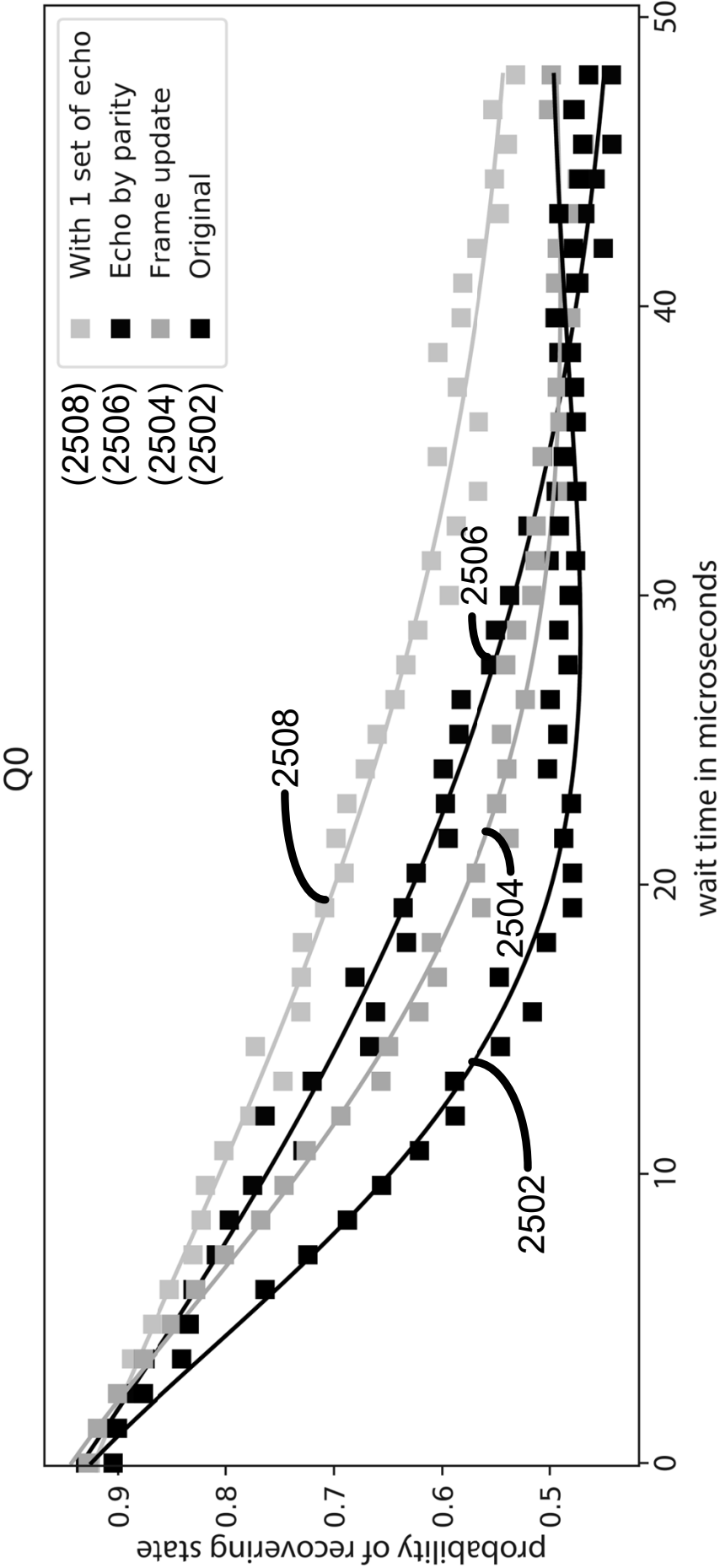


Figure 25

MITIGATING ERRORS

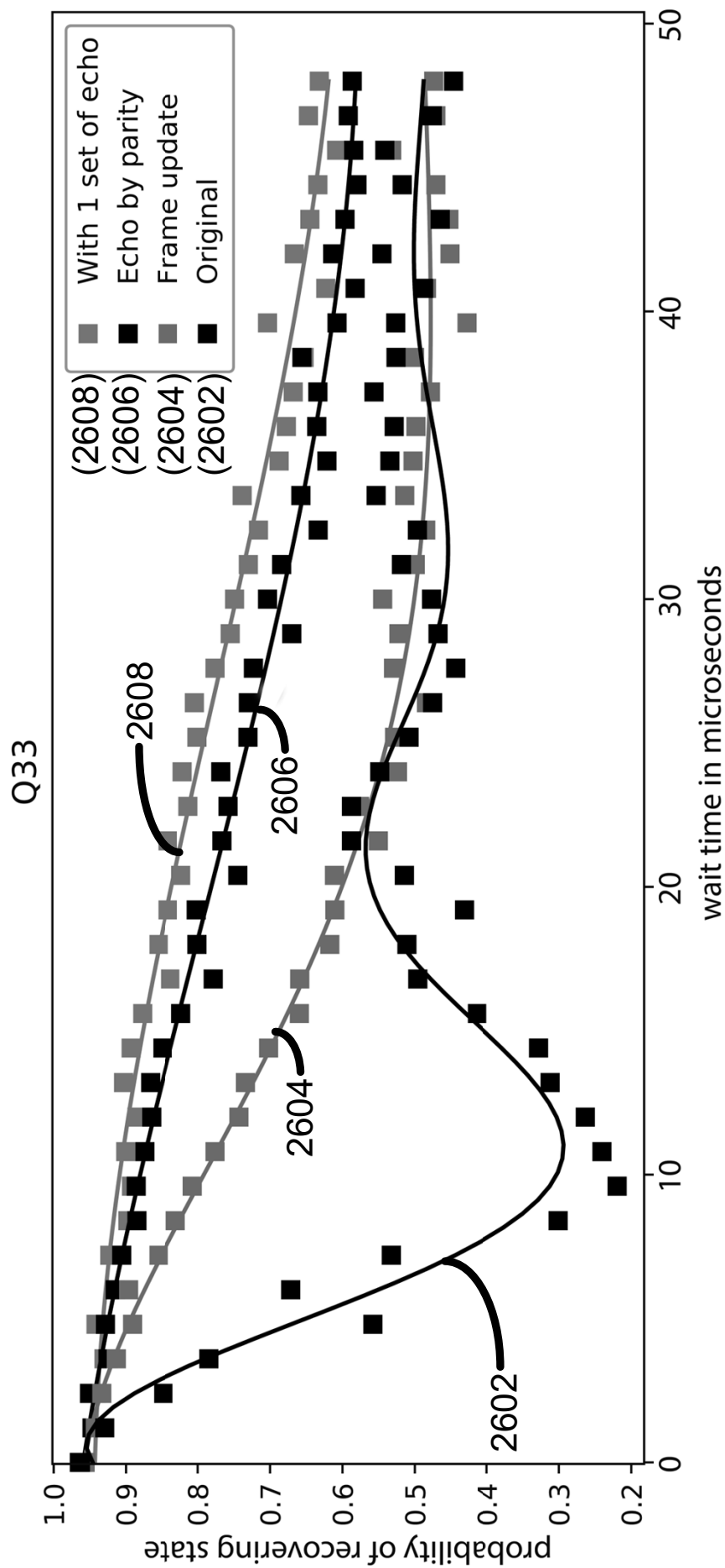


Figure 26

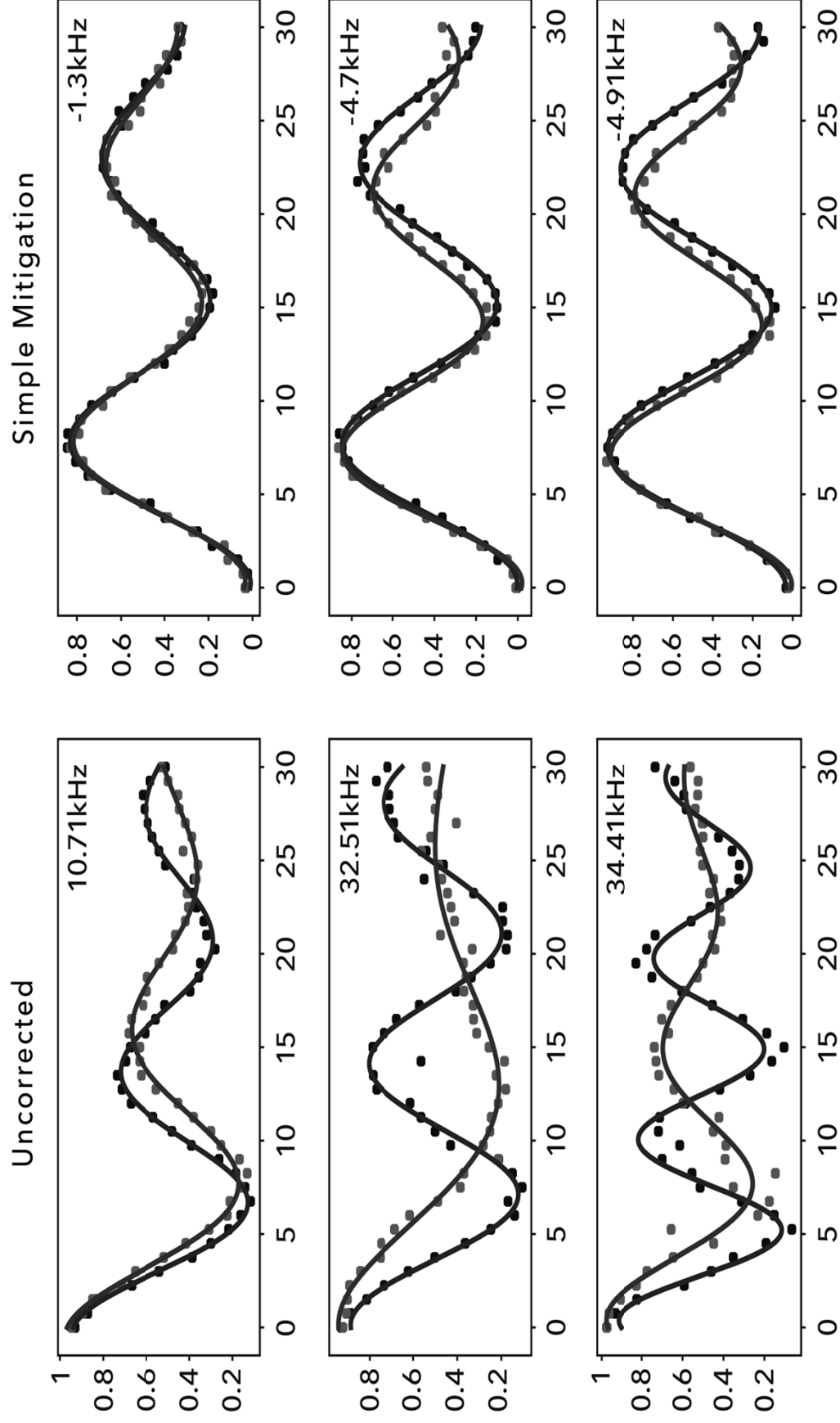


Figure 27

MAPPING STRAY COUPLINGS

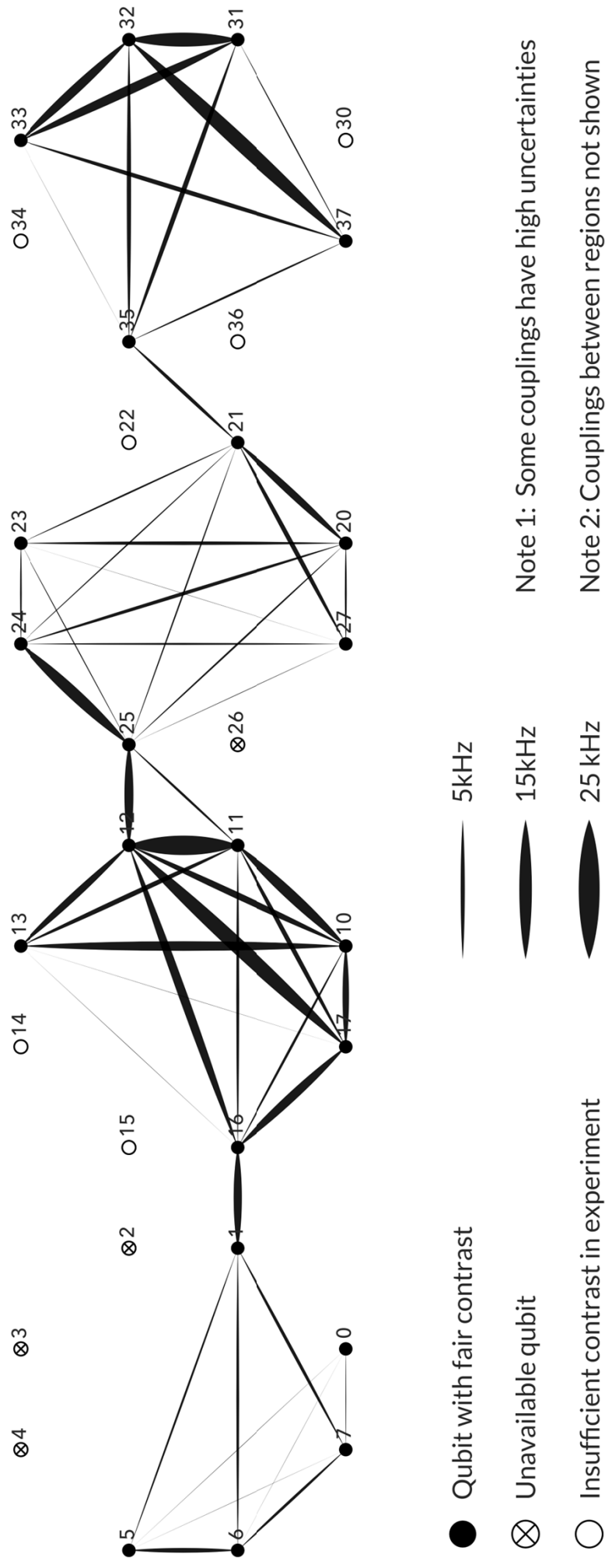


Figure 28

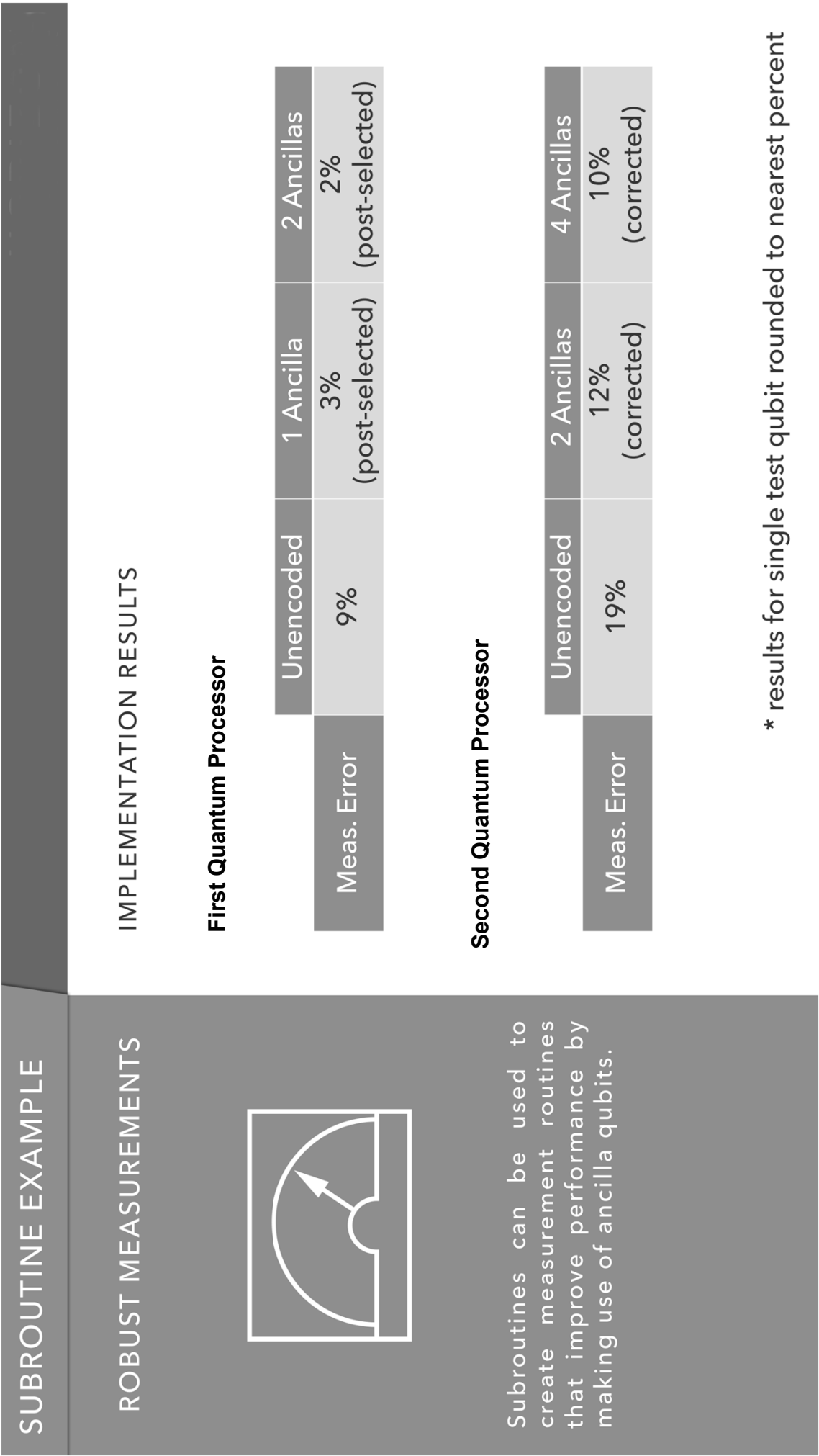
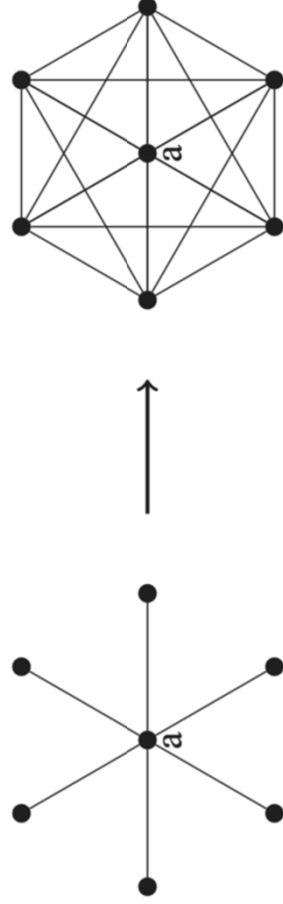


Figure 29

QUANTUM DATA STRUCTURES EXAMPLE

GRAPHS STATES AS GRAPHS



Extremely simple data structure that outperforms any possible classical data structure for certain combinations of tasks.

e.g. vertex comparison $O(1)$ and local edge complementation $O(1)$.



Local edge complementation



Intraset edge complementation



Interset edge complementation



Graph comparison



Vertex comparison and automorphism testing



Efficient classical readout from multiple copies

Figure 30

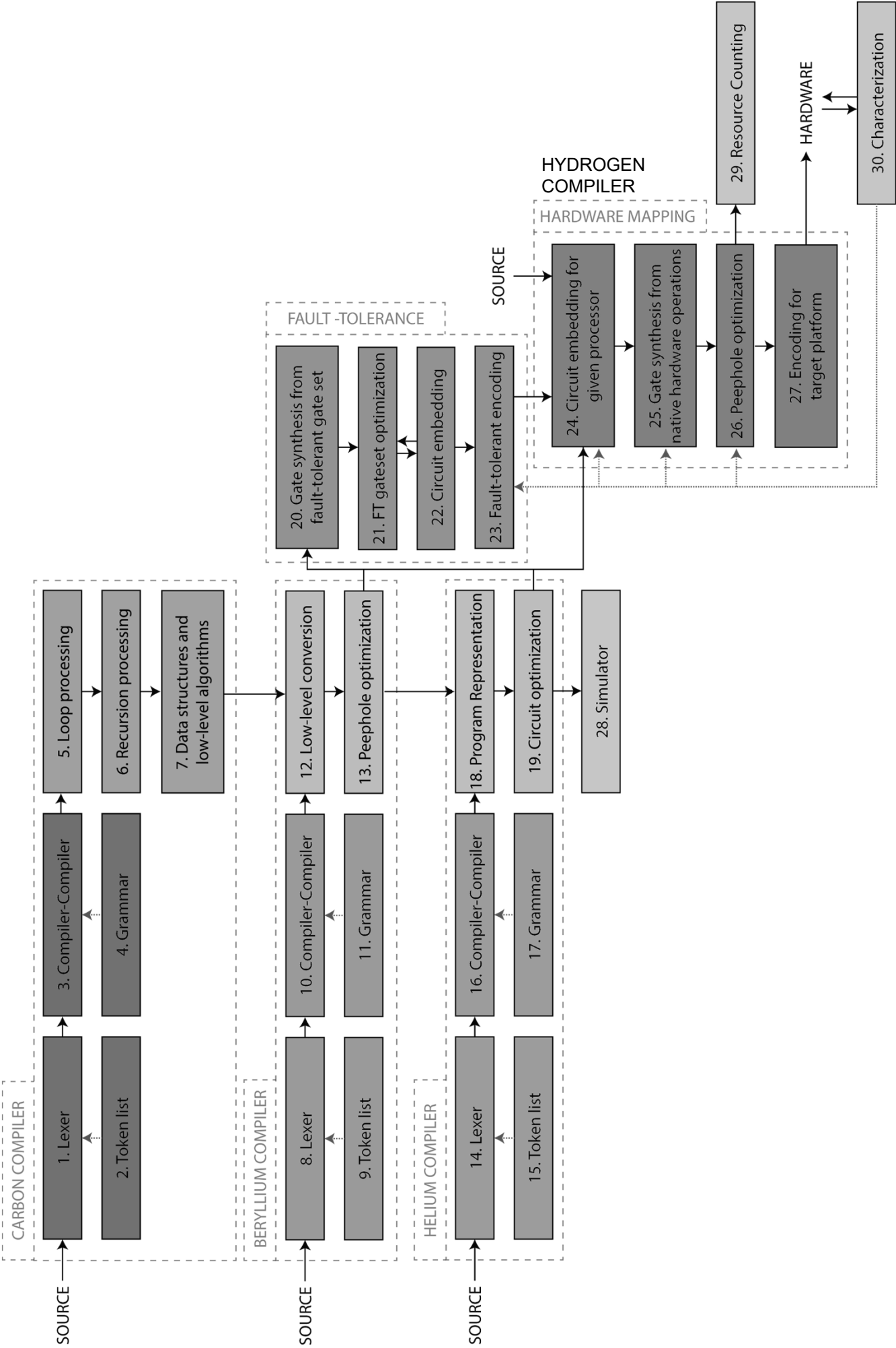


Figure 31

```

td {
  dim_in: 2
  dim_out: 2

  kraus: [1 0; 0 0.7071067811865475-0.7071067811865475i]
}

s {
  dim_in: 2
  dim_out: 2

  kraus: [1 0; 0 i]
}

sd {
  dim_in: 2
  dim_out: 2

  kraus: [1 0; 0 -i]
}

h {
  dim_in: 2
  dim_out: 2

  kraus: [0.7071067811865475 0.7071067811865475; 0.7071067811865475 -0.7071067811865475]
}

cnot {
  dim_in: 2 2
  dim_out: 2 2

  kraus: [1 0 0 0; 0 1 0 0; 0 0 0 1; 0 0 1 0]
}

```

Figure 32A

```

cz {
  dim_in: 2 2
  dim_out: 2 2
  kraus: [1 0 0 0; 0 1 0 0; 0 0 1 0; 0 0 0 -1]
}

swap {
  dim_in: 2 2
  dim_out: 2 2
  kraus: [1 0 0 0; 0 0 1 0; 0 1 0 0; 0 0 0 1]
}

toffoli {
  dim_in: 2 2 2
  dim_out: 2 2 2
  kraus: [1 0 0 0 0 0 0; 0 1 0 0 0 0 0; 0 0 1 0 0 0 0; 0 0 0 1 0 0 0; 0 0 0 0 1 0 0; 0 0 0 0 0 0 1; 0 0 0 0 0 0 0 1 0]
}

fredkin {
  dim_in: 2 2 2
  dim_out: 2 2 2
  kraus: [1 0 0 0 0 0 0; 0 1 0 0 0 0 0; 0 0 1 0 0 0 0; 0 0 0 1 0 0 0; 0 0 0 0 1 0 0; 0 0 0 0 0 1 0; 0 0 0 0 0 0 0 1]
}

```

Figure 32B

```

qubit {
  dim_in: 1
  dim_out: 2

  kraus: [1; 0]

  static: generate_qubit_label
}

measure {
  dim_in: 2
  dim_out: 1

  kraus: [1 0] -> 0
  kraus: [0 1] -> 1
}

vnmeasure {
  dim_in: 2
  dim_out: 2

  kraus: [1 0; 0 0] -> 0
  kraus: [0 0; 0 1] -> 1
}

```

Figure 32C

```

# HELIUM_web_test_prog0.q
qubit -> a
qubit -> b
qubit -> c

increment k to 12
    h[k]
    cnot[k,6]
    t[k]
    rz(k)[k]
next

9 -> u

measure[a] -> v

```

Figure 33A

```

# HELIUM_web_test_prog1.q
qubit -> a
qubit -> b
qubit -> c

9 -> u

increment k to u
    h[k]
next

measure[a] -> v

gosub test

gosub test2

sub test
    h[b]
    rz(u)[c]
    h[c]
Endsub

sub test2
    cnot[2,3]
    h[4]
endsub

```

Figure 33B

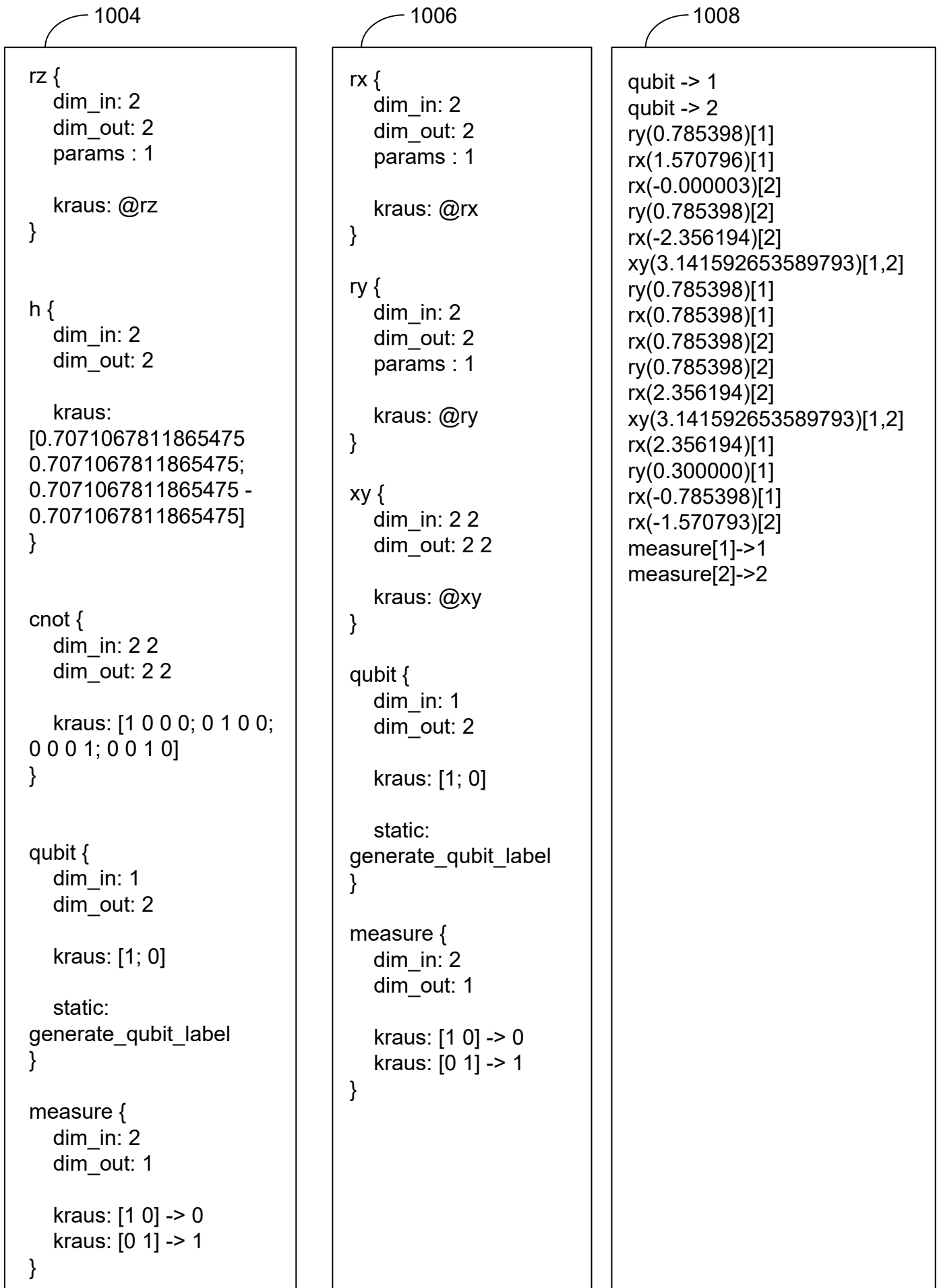


Figure 34

1004

```

rz {
  dim_in: 2
  dim_out: 2
  params : 1

  kraus: @rz
}

h {
  dim_in: 2
  dim_out: 2

  kraus:
[0.7071067811865475
0.7071067811865475;
0.7071067811865475 -
0.7071067811865475]
}

cnot {
  dim_in: 2 2
  dim_out: 2 2

  kraus: [1 0 0 0; 0 1 0 0;
0 0 0 1; 0 0 1 0]
}

qubit {
  dim_in: 1
  dim_out: 2

  kraus: [1; 0]

  static:
generate_qubit_label
}

measure {
  dim_in: 2
  dim_out: 1

  kraus: [1 0] -> 0
  kraus: [0 1] -> 1
}

```

1006

```

rx {
  dim_in: 2
  dim_out: 2
  params : 1

  kraus: @rx
}

ry {
  dim_in: 2
  dim_out: 2
  params : 1

  kraus: @ry
}

iswap {
  dim_in: 2 2
  dim_out: 2 2

  kraus: [1 0 0 0; 0 0 (1j)
0; 0 (1j) 0 0; 0 0 0 1]
}

qubit {
  dim_in: 1
  dim_out: 2

  kraus: [1; 0]

  static:
generate_qubit_label
}

measure {
  dim_in: 2
  dim_out: 1

  kraus: [1 0] -> 0
  kraus: [0 1] -> 1
}

```

1008

```

qubit -> 1
qubit -> 2
ry(0.785398)[1]
rx(1.570796)[1]
rx(-0.000003)[2]
ry(0.785398)[2]
rx(-2.356194)[2]
iswap[1,2]
ry(0.785398)[1]
rx(0.785398)[1]
rx(0.785398)[2]
ry(0.785398)[2]
rx(2.356194)[2]
iswap[1,2]
rx(2.356194)[1]
ry(0.300000)[1]
rx(-0.785398)[1]
rx(-1.570793)[2]
measure[1]->1
measure[2]->2

```

Figure 35

One set of echo for qubits of Aspen 7

qubit -> 1
qubit -> 2
qubit -> 3
qubit -> 4
qubit -> 5
qubit -> 6
qubit -> 7
qubit -> 8
qubit -> 9
qubit -> 10
qubit -> 11
qubit -> 12
qubit -> 13
qubit -> 14
qubit -> 15
qubit -> 16
qubit -> 17
qubit -> 18
qubit -> 19
qubit -> 20
qubit -> 21
qubit -> 22
qubit -> 23
qubit -> 24
qubit -> 25
qubit -> 26
qubit -> 27
qubit -> 28
qubit -> 29
qubit -> 30
qubit -> 31
qubit -> 32
qubit -> 33
qubit -> 34
qubit -> 35
qubit -> 36
qubit -> 37

0.5 -> Thalf

wait(Thalf)[33]
x[33]
wait(Thalf)[33]
x[33]

measure[33] -> 33

Echo by parity for qubits of Aspen 7

qubit -> 1
qubit -> 2
qubit -> 3
qubit -> 4
qubit -> 5
qubit -> 6
qubit -> 7
qubit -> 8
qubit -> 9
qubit -> 10
qubit -> 11
qubit -> 12
qubit -> 13
qubit -> 14
qubit -> 15
qubit -> 16
qubit -> 17
qubit -> 18
qubit -> 19
qubit -> 20
qubit -> 21
qubit -> 22
qubit -> 23
qubit -> 24
qubit -> 25
qubit -> 26
qubit -> 27
qubit -> 28
qubit -> 29
qubit -> 30
qubit -> 31
qubit -> 32
qubit -> 33
qubit -> 34
qubit -> 35
qubit -> 36
qubit -> 37

Figure 37A

```
1 -> Tfull
0.5 -> Thalf

wait(Thalf)
# Apply decoupling sequence on odd qubits
x[1]
x[3]
x[5]
x[7]
x[9]
x[11]
x[13]
x[15]
x[17]
x[19]
x[21]
x[23]
x[25]
x[27]
x[29]
x[31]
x[33]
x[35]
x[37]
wait(Thalf)
# Apply decoupling sequence on odd qubits
x[1]
x[3]
x[5]
x[7]
x[9]
x[11]
x[13]
x[15]
x[17]
x[19]
x[21]
x[23]
x[25]
x[27]
x[29]
x[31]
x[33]
x[35]
x[37]

measure[33] -> 33
```

Figure 37B