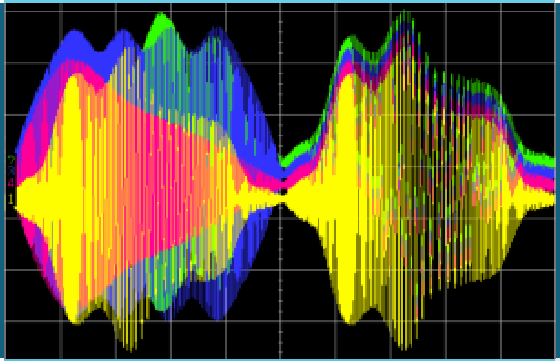
SAND2022-12341C

Performant coherent control: bridging the gap between high- and low-level operations on hardware



PRESENTED BY

Daniel Lobser, Jay Van Der Wall, and Joshua Goldberg

Requirements For A Self-Contained Control System

Requirements	Challenges
Complex pulse shaping	Can result in a lot of data. Classical computing resources to calculate pulse shapes.
Long gate sequences	Even more data.
Conditional sequences	Fast response based on mid-circuit measurements.
Tight-loop feedback on control parameters	Tight loop = low latency, must take network transfers out of the loop. Pulse-level gate definitions must be calculated and compiled on chip.

Approaches	Benefits	Drawbacks
Arbitrary waveform generator (AWG)	Maximum control over pulse shapes.	Requires lots of data, long upload times & limited circuit depth. Waveforms must be calculated externally
FPGA + Direct Digital Synthesizers (DDSs) or Digital-to-Analog Converters (DACs)	Highly customizable. Deterministic timing. Soft-core CPUs or state machines can be used for advanced control flow.	Soft-core CPUs have limited processing power and eat into FPGA resources. Design complexity.
System-on-Chip (SoC)	Hard-core CPUs and FPGA fabric. Some SoCs have real-time processors.	Design complexity.

Our Approach For A Self-Contained Control System

Xilinx RadioFrequency System on Chip (RFSoC)

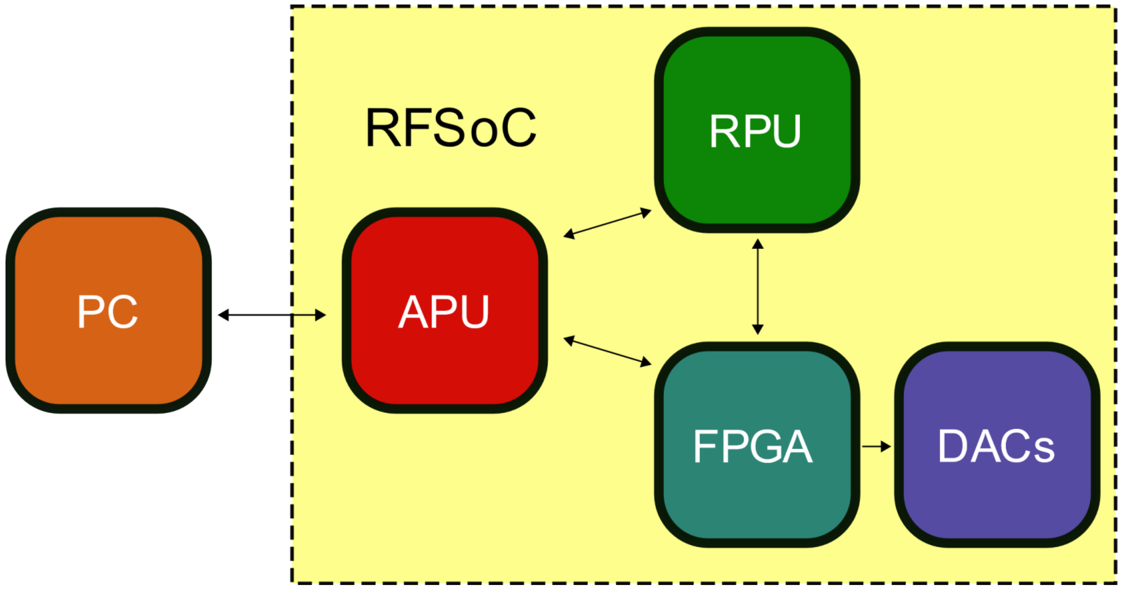
Quad-core ARM application processing unit (APU)
- Runs Linux
- Non-deterministic timing

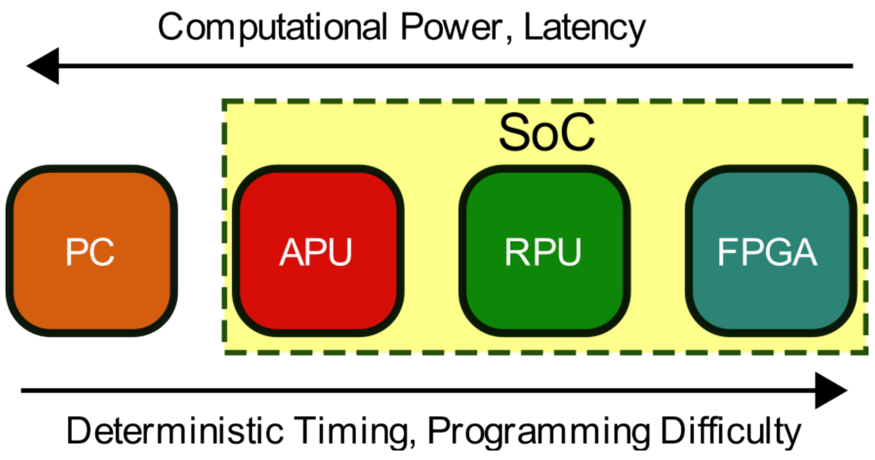
Dual-core ARM real-time processing unit (RPU)
- Runs bare-metal or real-time operating systems (RTOS)
- Deterministic timing (mostly)

APU and RPU can be independently operated via asymmetric multiprocessing (AMP)

Large FPGA fabric

Integrated 6.5 GSPS 14-bit DACs (8x) and 4 GSPS 12-bit ADCs (8x)




Computational Power, Latency
SoC
PC
APU
RPU
FPGA
Deterministic Timing, Programming Difficulty

Our Approach For A Self-Contained Control System

Custom gateware design (called "Octet"), tailored for target system

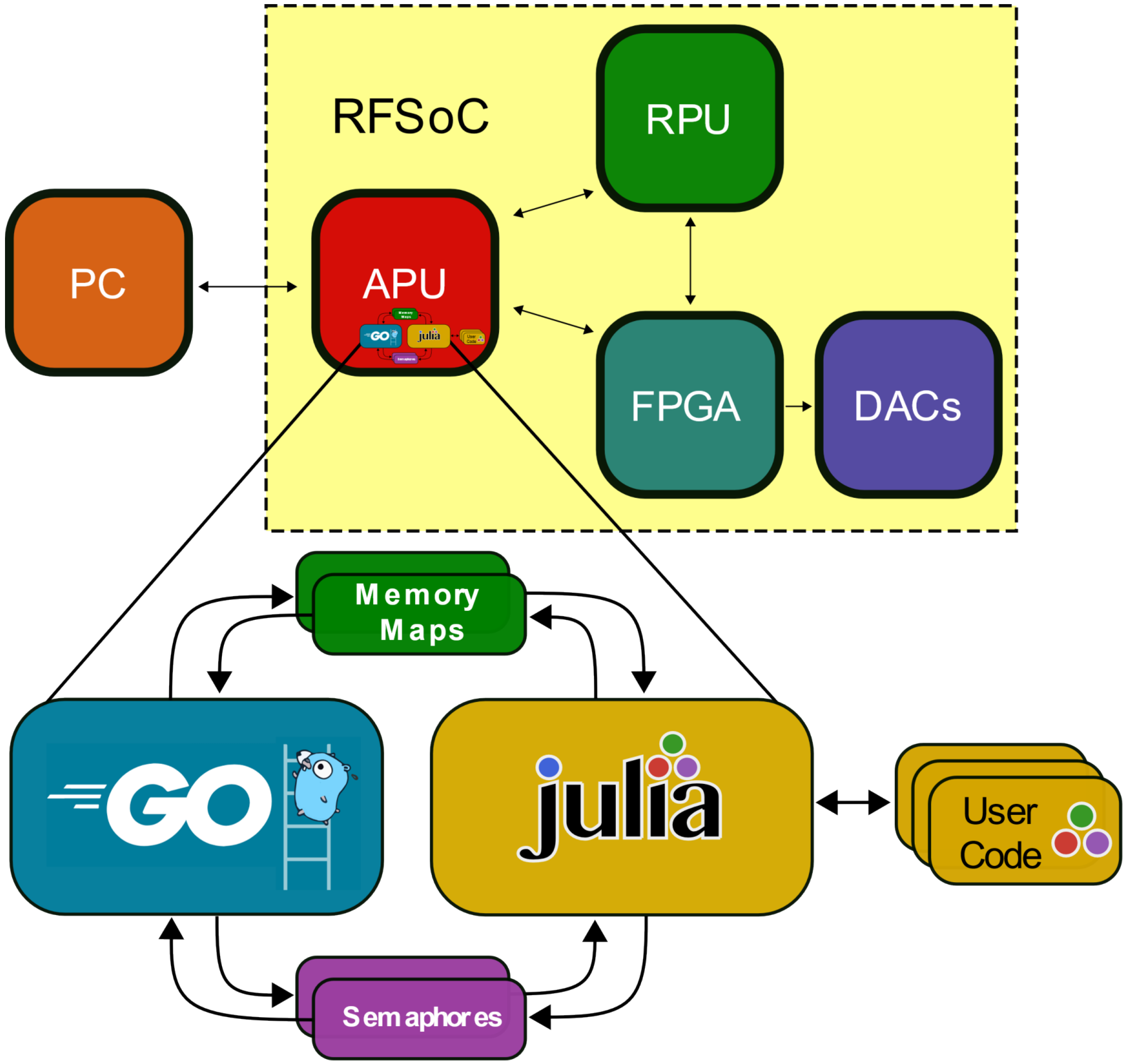
- Developed for QSCOUT (https://qscout.sandia.gov)
- Integrated cubic-spline interpolators (based on work from NIST)
- Custom DDS and gate sequencer modules
- Low-level error mitigation features for frequency stabilization and crosstalk compensation

Software interface written in Go, runs on APU

- Communication over ethernet (gRPC/Protobuf)

Circuits compiled on-chip from Jaqal ("Just another quantum assembly language")

Pulse-level gate definitions both on-chip (using Julia), and off-chip (using JaqalPaw, "Jaqal Pulses and Waveforms")



Realizing Quantum Gates

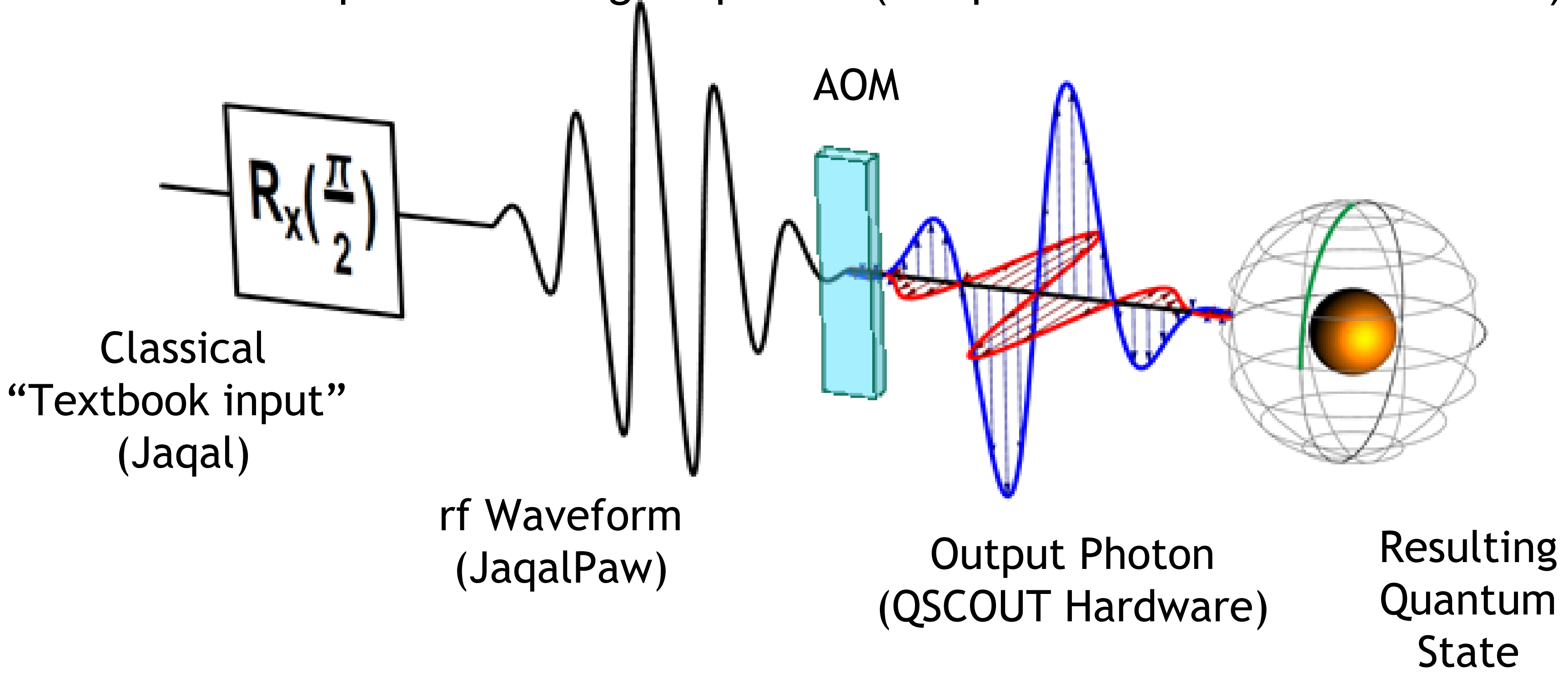
QSCOUT

Gates specified in Jaqal must be converted to a form that is experimentally realizable

The internal quantum states of individually-addressed ions are manipulated via laser light passed through a acousto-optic modulators (AOMs)

Each AOM is modulated with an rf waveform to precisely tune the frequency, phase, and amplitude of the light

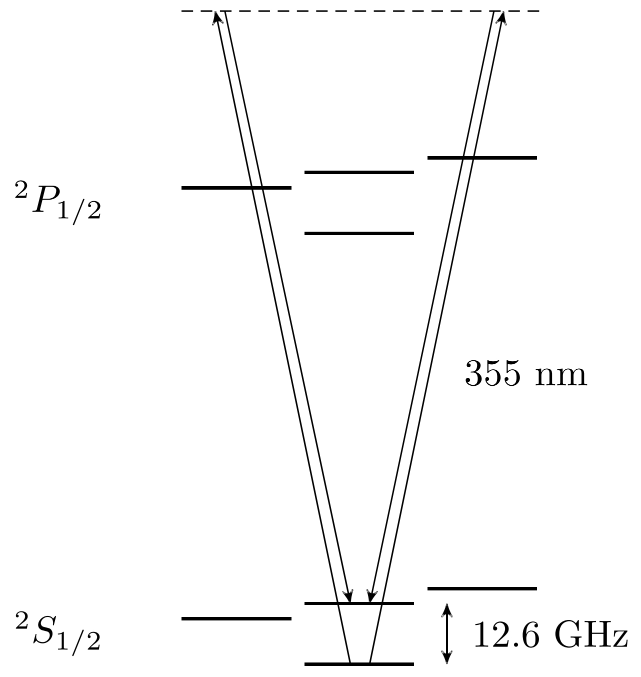
These waveforms are specified using JaqalPaw (“Jaqal Pulses and Waveforms”)



Target System

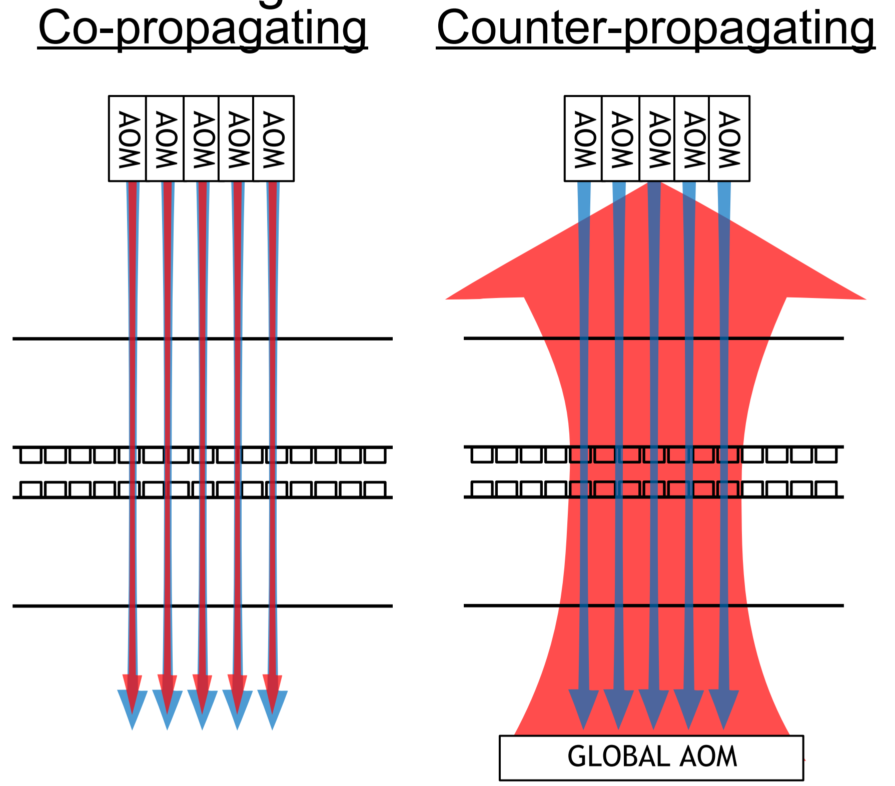
QSCOUT

^{171}Yb$^+$ qubit, clock state 12.6 GHz



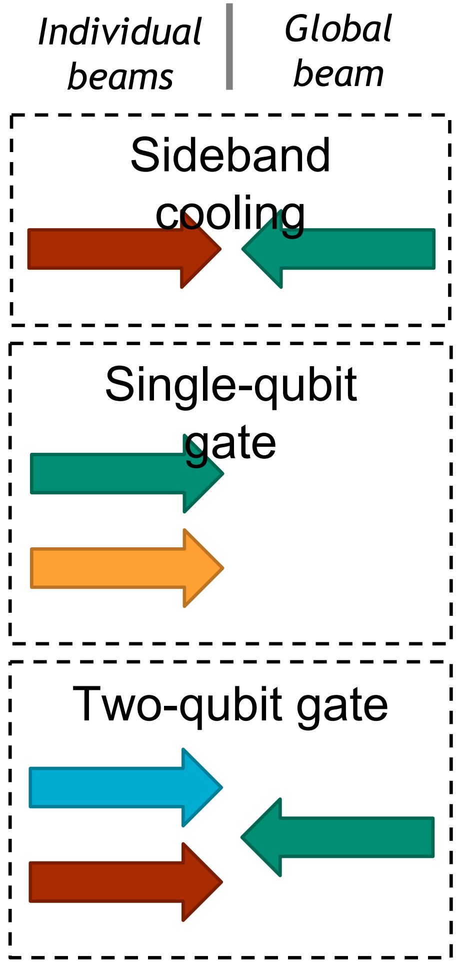
12.6 GHz driven via optical Raman transitions for individual qubit addressing

We use two beam configurations



Co-propagating is robust against phase uncertainty

Counter-propagating used for driving motional transitions



Need two tones per output

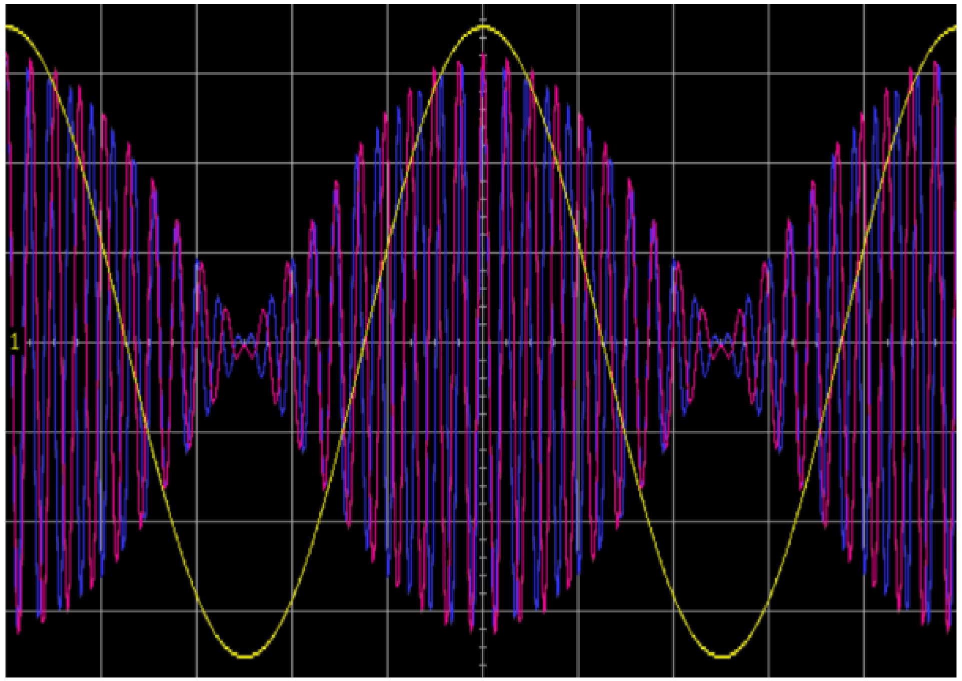
Different frequencies used in each configuration

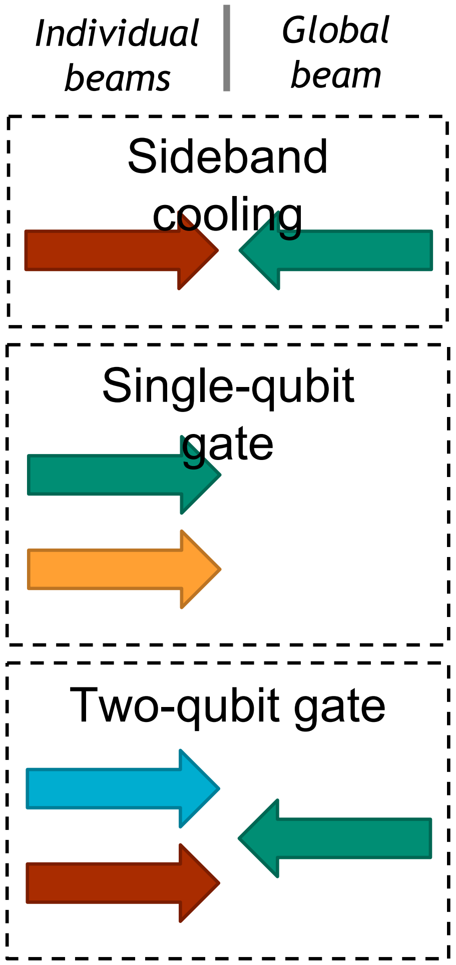
Target System

<u>*Phase control is imperative!*</u>

Each configuration requires different frequencies

Phase of beat note produced by red- and blue-sideband tones determines global phase of the two-qubit Mølmer-Sørensen gate





Need two tones per output

Different frequencies used in each configuration

Quantum Gates

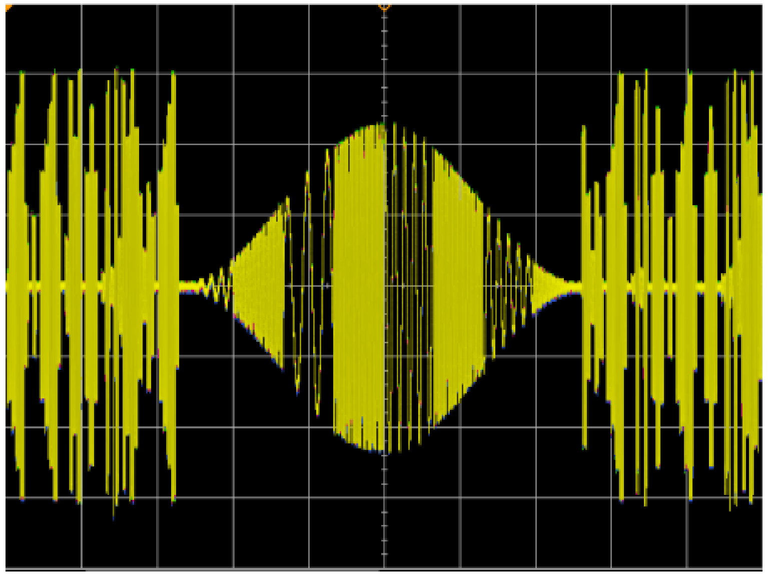
QSCOUT

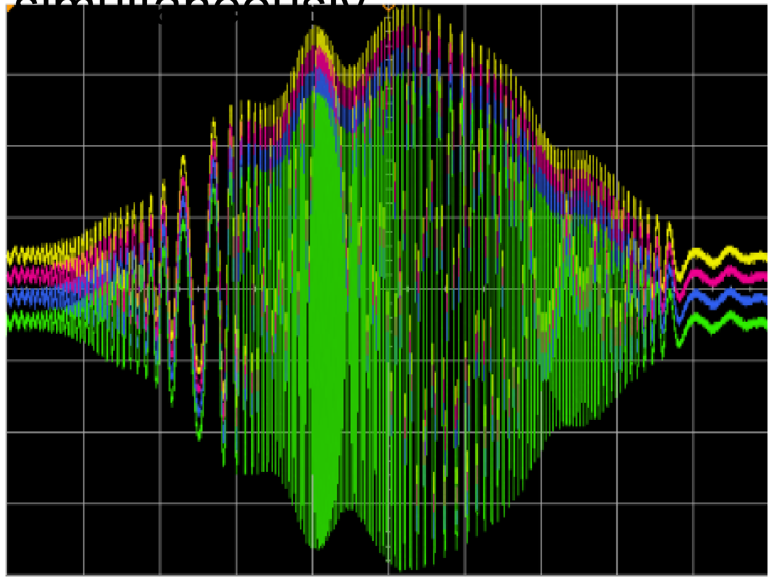
Gates must be defined as discrete “pulses” with precise timing and characteristics to achieve the desired results

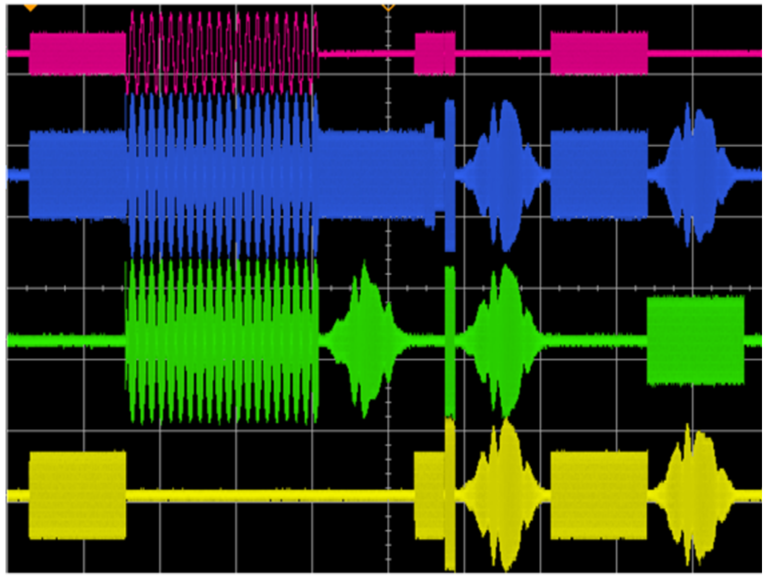
State of the art gate designs require discrete or continuous modulation of frequency, phase, and amplitude.

Gates must be synchronous across all channels and tones, with the ability to run all modulation types simultaneously

Long sequences can be necessary, so a compact representation is needed.







Gate times (1-200 us) are typically much slower than the period (5 ns) of the natural frequencies (200 MHz) needed to drive the AOMs

Instead of writing raw waveform data like an arbitrary waveform generator (AWG) we can take advantage of more compact representations

~~Compression, Compression, Compression~~ Compression^3

1. Minimizing samples: baseband frequencies of ~200 MHz can be generated by DDSs

Changing Frequency in a Direct Digital Synthesizer (DDS)

Conventional DDS consists of a phase accumulator and lookup table (LUT)

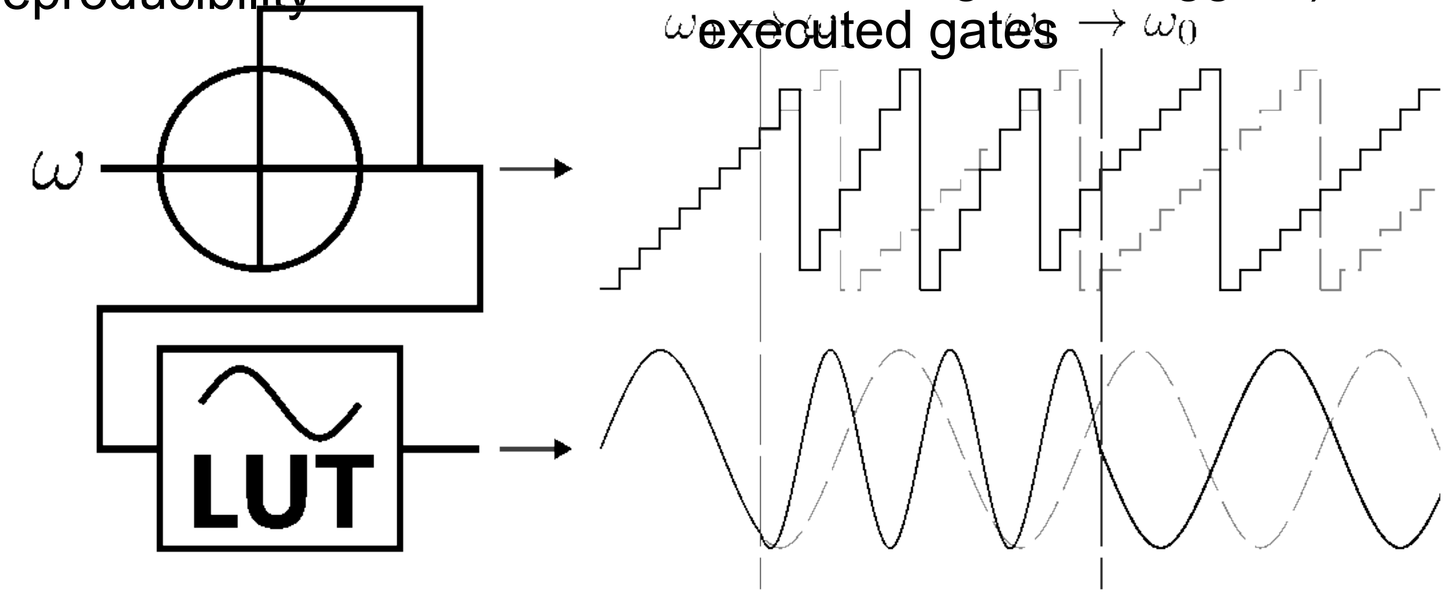
Changes to frequency are continuous

- Good for frequency modulation
- Bad for phase reproducibility

Multiple independent sources → Doesn't scale

Manual phase bookkeeping

- Context-dependent gate definitions
- Requires more data if conditionally-executed gates are used
- Not robust against timing variation (e.g. missed clocked edges for triggers) for conditionally-executed gates



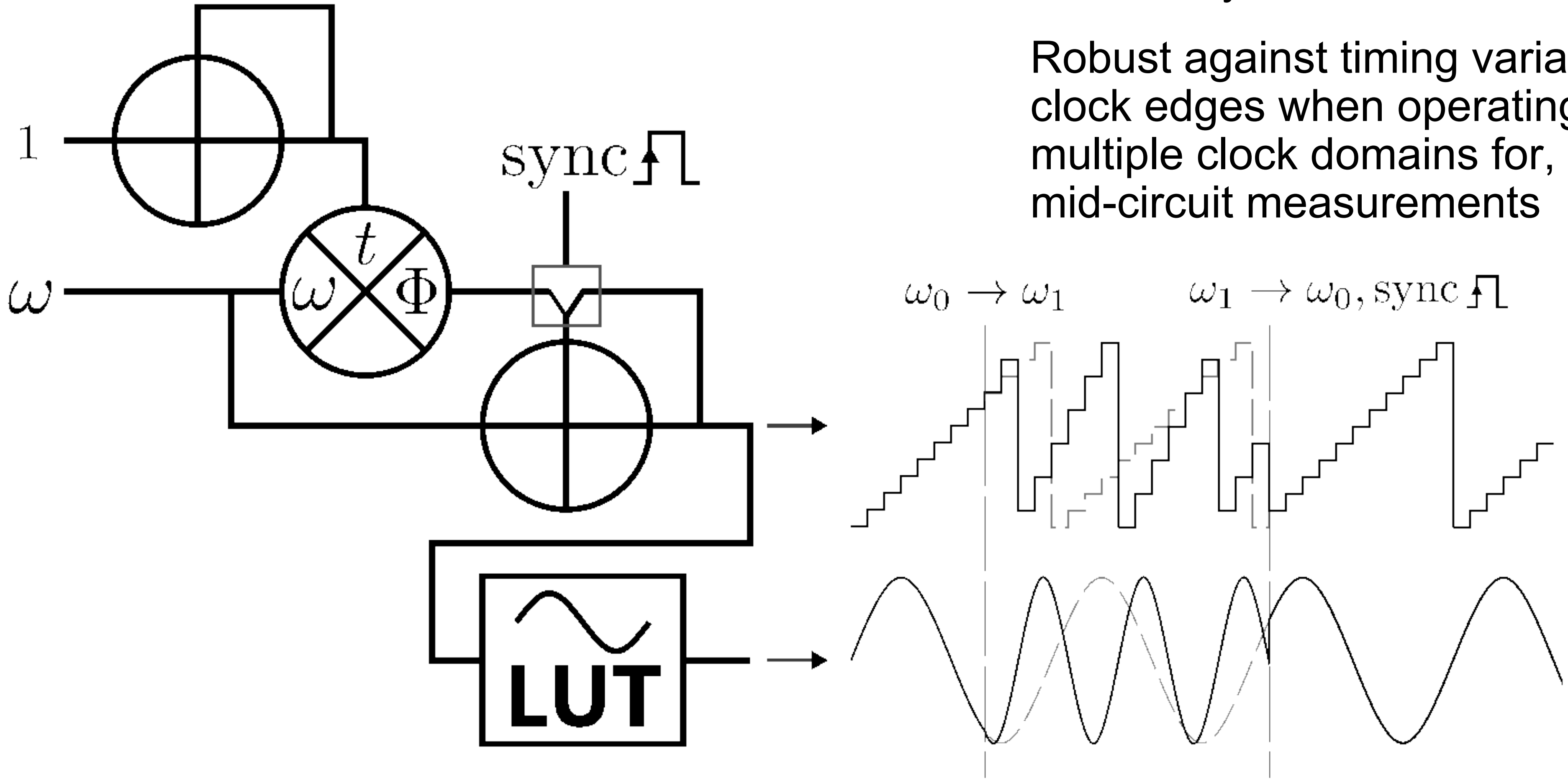
Hardware-Native Phase Bookkeeping: Global Phase Synchronization

Dedicated multiplier calculates global phase, $\Phi = \omega t$, which optionally overwrites accumulator when a synchronization trigger is applied

No manual bookkeeping required

Global counter fanned out to all channels: cross-channel synchronization built in

Robust against timing variation, e.g. missed clock edges when operating with hardware on multiple clock domains for, among other things, mid-circuit measurements


1
ω
t
ω
Φ
sync
LUT
$\omega_0 \rightarrow \omega_1$
$\omega_1 \rightarrow \omega_0$, sync

~~Compression, Compression, Compression~~ Compression^3

QSCOUT

1. Minimizing samples: baseband frequencies of ~200 MHz can be generated by DDSs
2. Hardware-native phase bookkeeping
   - Global phase synchronization handled by custom DDS

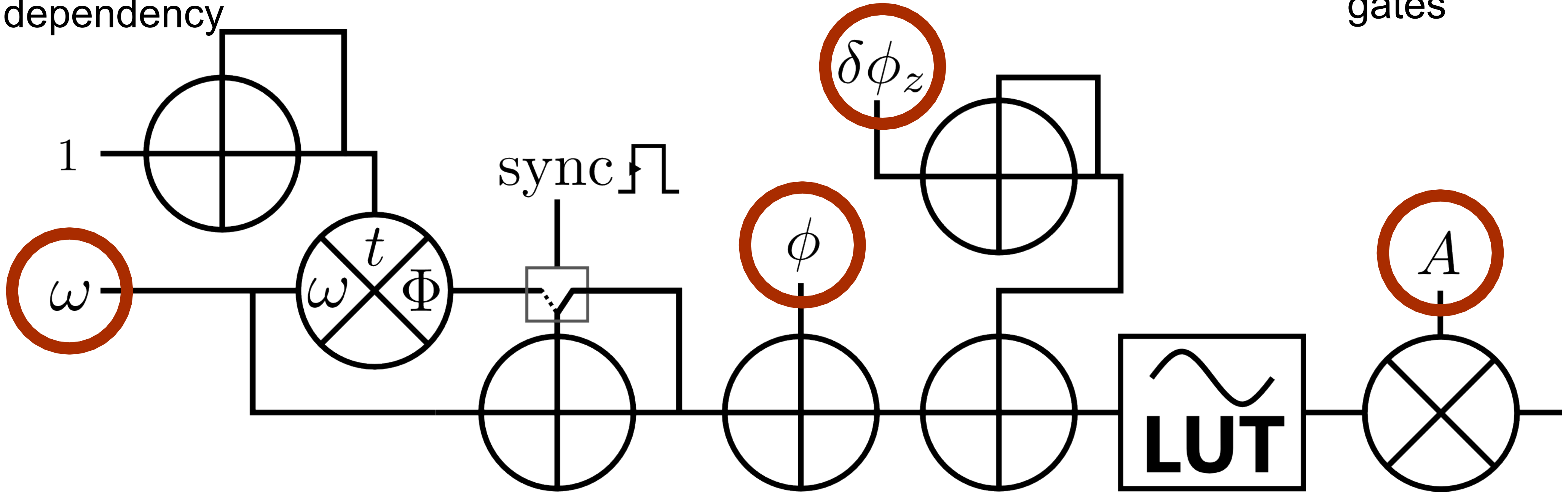
Hardware-Native Phase Bookkeeping: Virtual Z Gates aka “Frame Rotations”

Only X and Y gates are directly driven

Z gates can be implemented as a virtual phase offset

This phase is persistent in order to affect all following gates → takes care of context-dependency

*Actual model slightly more complex: optional forwarding/inversion used to handle differential, or (anti)symmetric phase offsets needed for different configurations for single- and two-qubit gates



~~Compression, Compression, Compression~~ Compression^3

1. Minimizing samples: baseband frequencies of ~200 MHz can be generated by DDSs
2. Hardware-native phase bookkeeping
   - Global phase synchronization handled by custom DDS
   - Dedicated phase accumulators track virtual phase, eliminating issues with context dependency

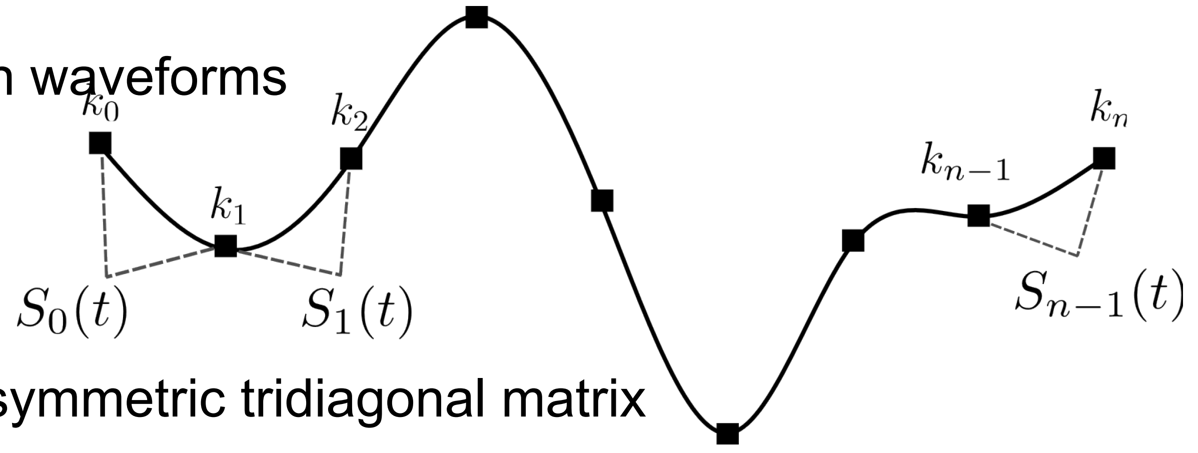
Cubic Splines

Parameter modulation described with cubic spline coefficients

Gateware interpolators use a lightweight model (developed by NIST) that relies only on addition

Provides compact representation for modulation waveforms

Piecewise Cubic Polynomial Segments
$$S_i(t) = \sum_{n=0}^{3} U_{n,i}(t - t_i)^n$$



Fitting natural cubic splines requires inverting a symmetric tridiagonal matrix

Can be represented with two arrays (memory efficient)

$$\begin{pmatrix} 2 & 1 & & & & & 0 \\ 1 & 4 & 1 & & & & \\ & 1 & 4 & & & & \\ & & & \ddots & & & \\ & & & & 4 & 1 & \\ & & & & 1 & 4 & 1 \\ 0 & & & & & 1 & 2 \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \\ \vdots \\ g_{n-2} \\ g_{n-1} \\ g_n \end{pmatrix} = \begin{pmatrix} 3(k_1 - k_0) \\ 3(k_2 - k_0) \\ 3(k_3 - k_1) \\ \vdots \\ 3(k_{n-1} - k_{n-3}) \\ 3(k_n - k_{n-2}) \\ 3(k_n - k_{n-1}) \end{pmatrix}$$

Mapping for gateware interpolators uses a custom floating point scheme and asymmetric register sizes to maintain resolution for slow modulation

$$v_0 \rightarrow \overbrace{01001101}^{40\text{ bits}}$$
$$v_1 \rightarrow \overbrace{00010011}^{40\text{ bits}}.\overbrace{0101}^{16\text{ bits}}$$
$$v_2 \rightarrow \overbrace{00000100}^{40\text{ bits}}.\overbrace{11010101}^{32\text{ bits}}$$
$$v_3 \rightarrow \overbrace{00000001}^{40\text{ bits}}.\overbrace{001101010100}^{48\text{ bits}}$$

~~Compression, Compression, Compression~~ Compression^3

QSCOUT

1. Minimizing samples: baseband frequencies of ~200 MHz can be generated by DDSs
2. Hardware-native phase bookkeeping: cuts down on amount of unique data needed
   - Global phase synchronization handled by custom DDS
   - Dedicated phase accumulators track virtual phase, eliminating issues with context dependency
3. Cubic spline interpolators offer 10^2 to 10^4 reduction in data on average

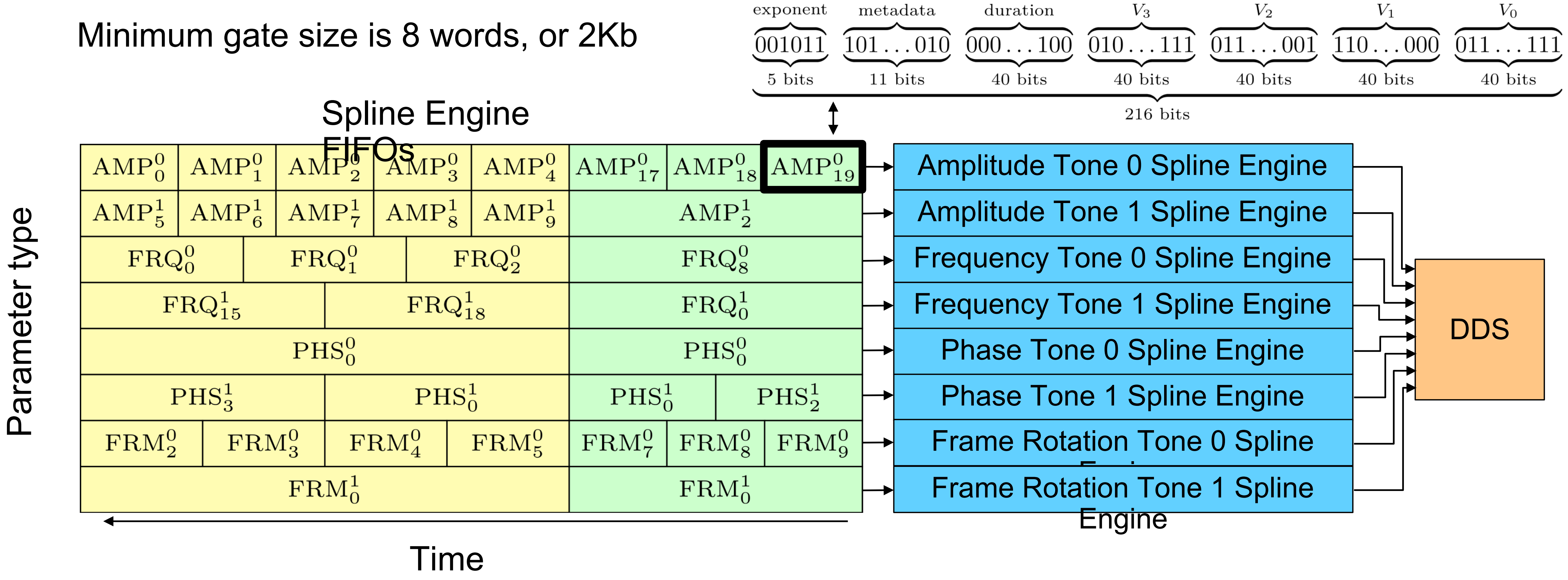
Sequencing Spline-Modulated Data

QSCOUT

Each spline segment is routed to the appropriate spline engine for subsequent control of a dual -tone DDS

Input words are really 256 bits and extra metadata is truncated during routing

Minimum gate size is 8 words, or 2Kb



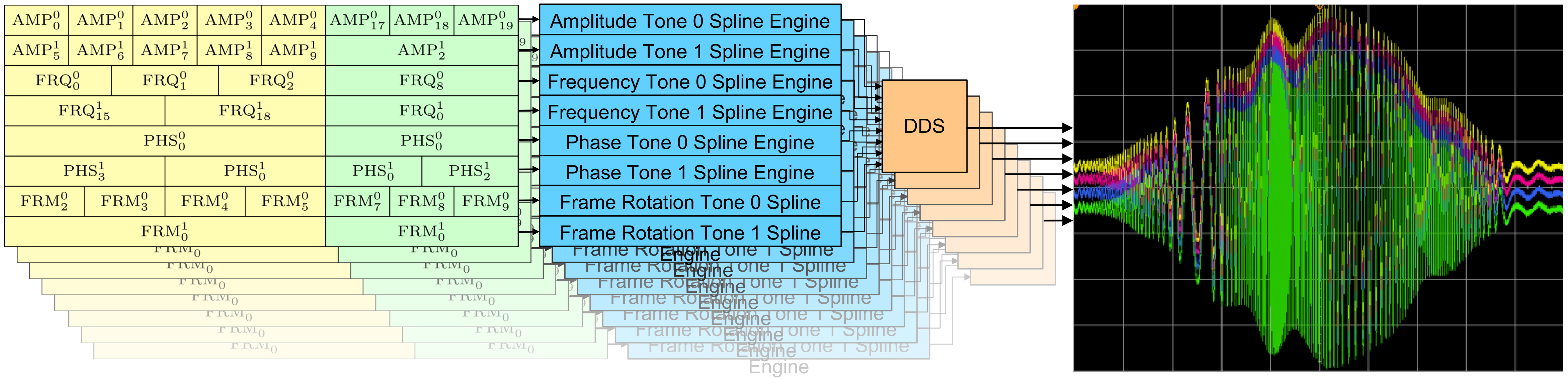
exponent
metadata
duration
V_3
V_2
V_1
V_0
001011
101 . . . 010
000 . . . 100
010 . . . 111
011 . . . 001
110 . . . 000
011 . . . 111
5 bits
11 bits
40 bits
40 bits
40 bits
40 bits
40 bits
216 bits
Spline Engine FIFOs
AMP^0_0 AMP^0_1 AMP^0_2 AMP^0_3 AMP^0_4 AMP^0_{17} AMP^0_{18} AMP^0_{19}
AMP^1_5 AMP^1_6 AMP^1_7 AMP^1_8 AMP^1_9 AMP^1_2
FRQ^0_0 FRQ^0_1 FRQ^0_2 FRQ^0_8
FRQ^1_{15} FRQ^1_{18} FRQ^1_0
PHS^0_0 PHS^0_0
PHS^1_3 PHS^1_0 PHS^1_0 PHS^1_2
FRM^0_2 FRM^0_3 FRM^0_4 FRM^0_5 FRM^0_7 FRM^0_8 FRM^0_9
FRM^1_0 FRM^1_0
Amplitude Tone 0 Spline Engine
Amplitude Tone 1 Spline Engine
Frequency Tone 0 Spline Engine
Frequency Tone 1 Spline Engine
Phase Tone 0 Spline Engine
Phase Tone 1 Spline Engine
Frame Rotation Tone 0 Spline Engine
Frame Rotation Tone 1 Spline Engine
DDS
Parameter type
Time

Data Handling for Concurrent Execution

All 64 spline engines must be run concurrently

Spline engines are fed by First-In-First-Out (FIFO) buffers

Unused channels are padded with NOPs to preserve timing downstream



AMP_0^0 AMP_1^0 AMP_2^0 AMP_3^0 AMP_4^0 AMP_{17}^0 AMP_{18}^0 AMP_{19}^0
AMP_5^1 AMP_6^1 AMP_7^1 AMP_8^1 AMP_9^1 AMP_2^1
FRQ_0^0 FRQ_1^0 FRQ_2^0 FRQ_8^0
FRQ_{15}^1 FRQ_{18}^1 FRQ_0^1
PHS_0^0 PHS_0^0
PHS_3^1 PHS_0^1 PHS_0^1 PHS_2^1
FRM_2^0 FRM_3^0 FRM_4^0 FRM_5^0 FRM_7^0 FRM_8^0 FRM_9^0
FRM_0^1 FRM_0^1
Amplitude Tone 0 Spline Engine
Amplitude Tone 1 Spline Engine
Frequency Tone 0 Spline Engine
Frequency Tone 1 Spline Engine
Phase Tone 0 Spline Engine
Phase Tone 1 Spline Engine
Frame Rotation Tone 0 Spline
Frame Rotation Tone 1 Spline Engine
DDS

Abstracting Pulse Information: Gate Slices

Gate data is broken up into “gate slices”, based on individual steps in a circuit

Gate slices separate data based on rf output channel

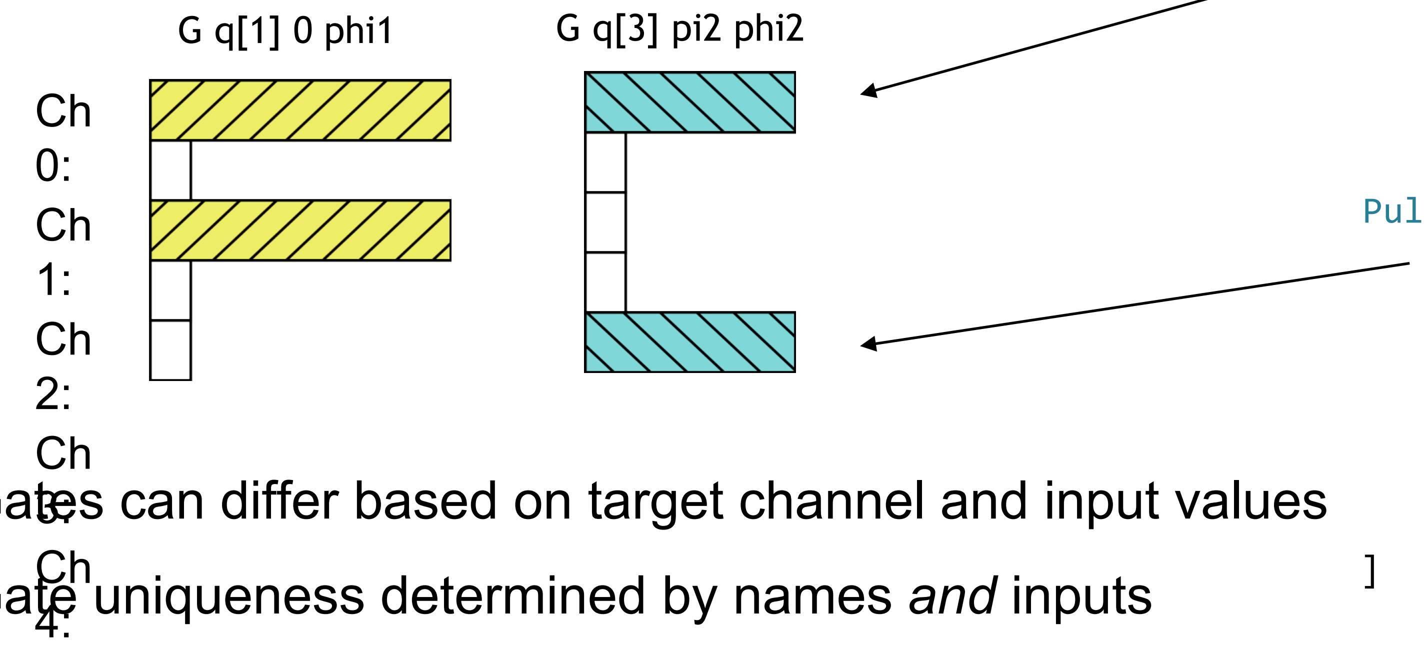
Jaqal (Just Another Quantum Assembly Language)

```
G q[1] 0 phi1
G q[3] pi2 phi2
```

JaqalPaw (Jaqal Pulses and Waveforms)

```
def gate_G(self, qubit, theta, phi):
    phase = (phi < 0) * 180 + theta / np.pi * 180
    duration = self.duration_from_rabi_angle(phi, qubit)
    return [PulseData(
            GLOBAL_BEAM,
            duration,
            amp0=self.gaussian(phi, qubit),
            freq0=self.upper_carrier_frequency,
            sync_mask=0b01,
            ...),
        PulseData(
            self.qubit_mapping[qubit],
            duration,
            amp0=self.gaussian(phi, qubit),
            freq0=self.lower_carrier_frequency,
            phase0=phase,
            sync_mask=0b01,
            ...),
    ]
```

Gate Slices



Gates can differ based on target channel and input values

Gate uniqueness determined by names *and* inputs

Maintaining Scheduling on Unused Channels: Padded Gate Slices

Gate slices are given a unique tag based on their call signature

G1 q[1] 0 phi1 $\rightarrow \mathfrak{S}_0$

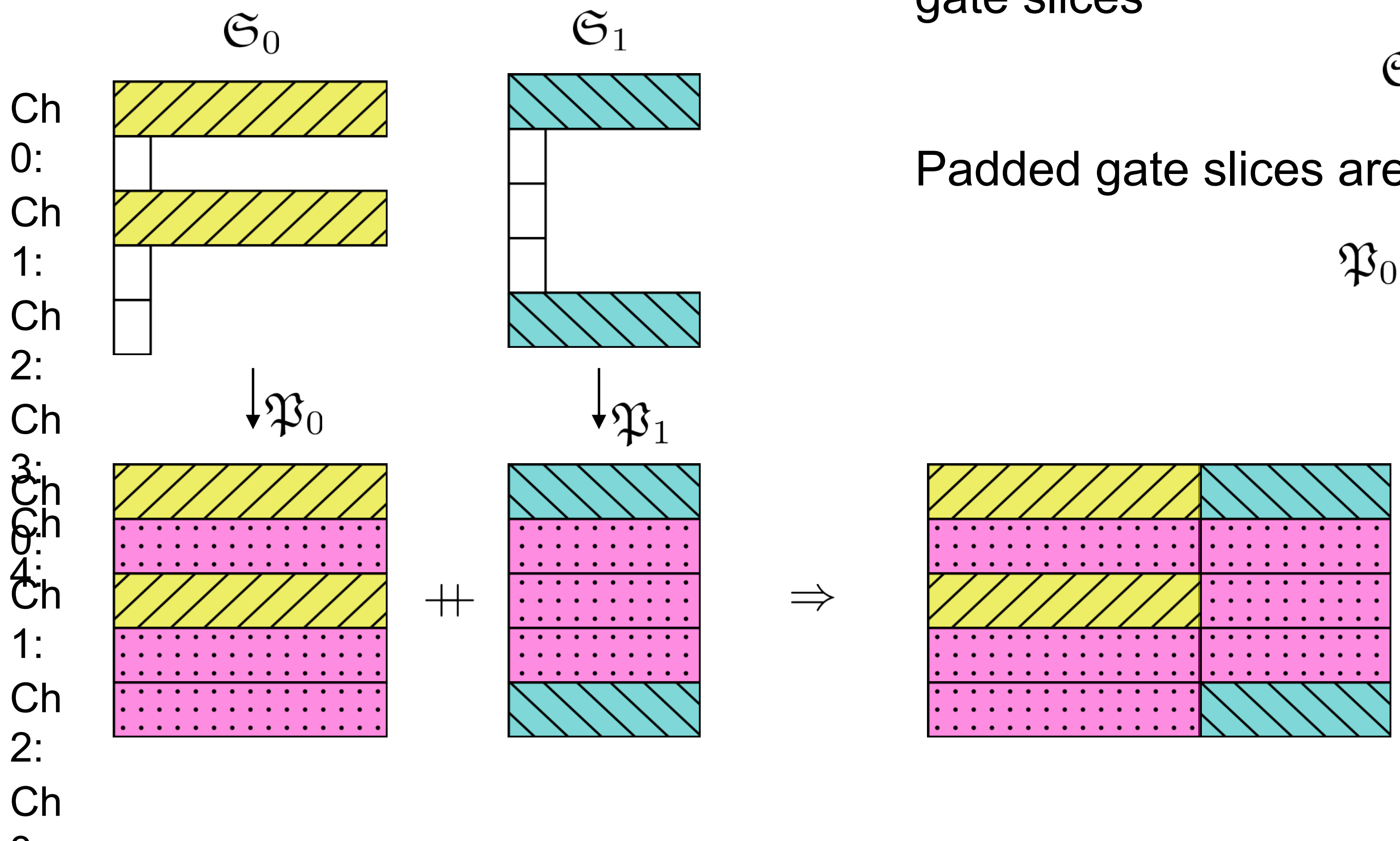
G2 q[3] pi2 phi2 $\rightarrow \mathfrak{S}_1$

Unused channels are padded with NOPs to preserve scheduling of later gate calls, and stored as "padded gate slices"

$$\mathfrak{S}_n \rightharpoonup \mathfrak{P}_n$$

Padded gate slices are concatenated for readout

$$\mathfrak{P}_0 + \mathfrak{P}_1 + \ldots$$

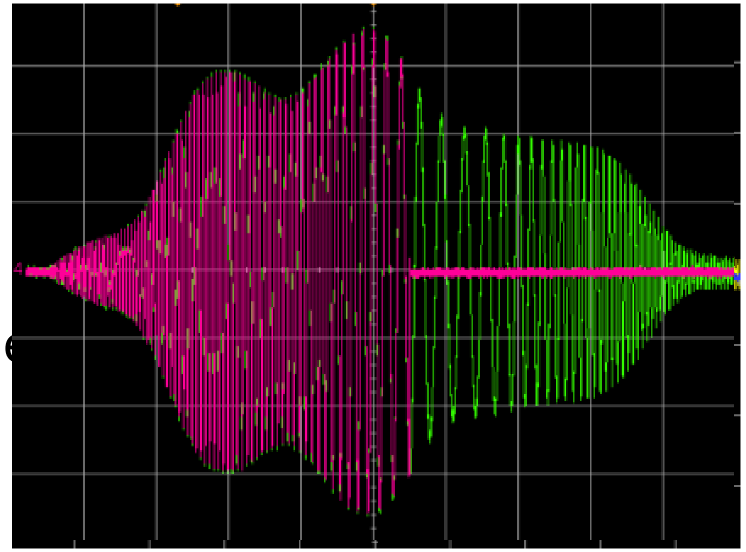


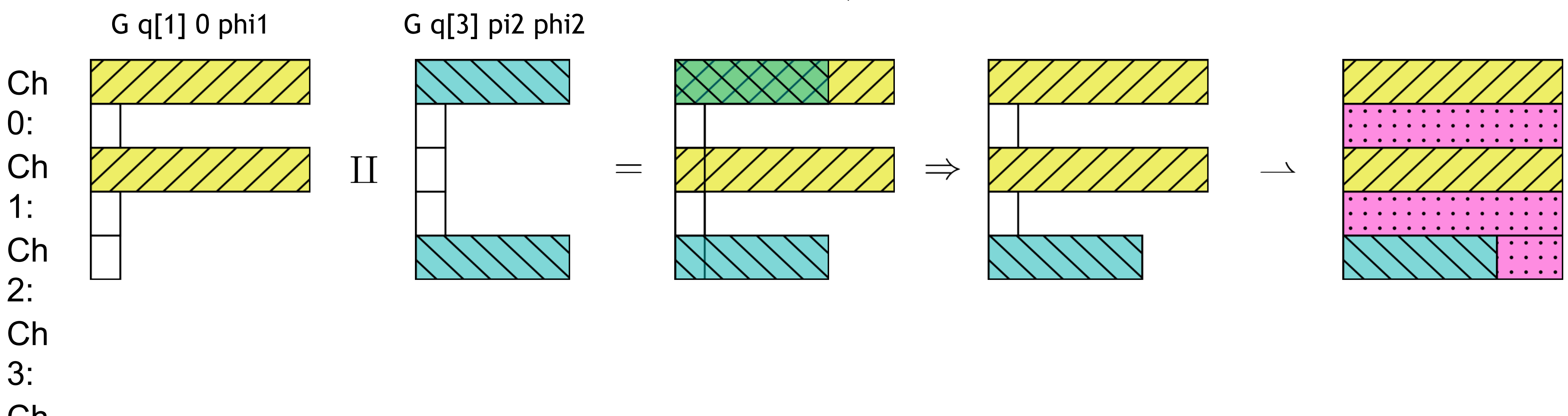
Handling Parallel Gate Execution

Jaqal supports parallel execution of gates:

```
< G q[1] 0 phi1 | G q[3] pi2 phi2 >
```

Must ensure compatibility of pulse information on shared channe



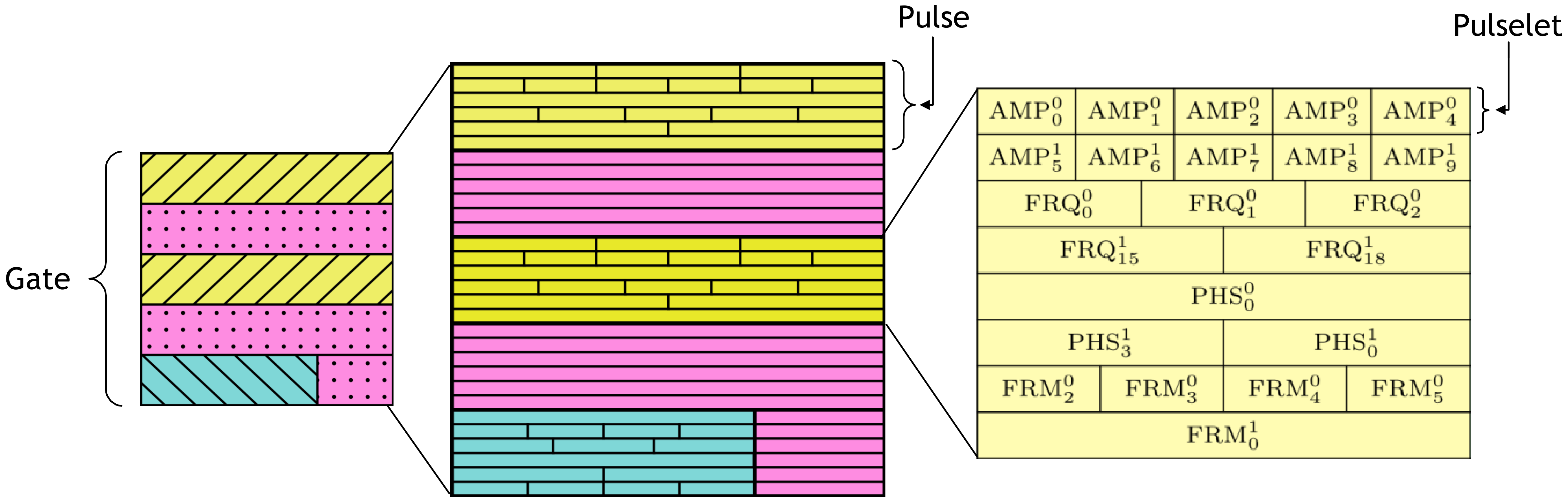


Breaking Up Data By Parameter

Each gate consists of a sequence of pulses, determined by a series of parameters: amplitude, frequency, phase, and frame rotation, for two tones per channel.

Each parameter supports modulation using spline interpolators.

Individual sequences of parameters are called pulselets

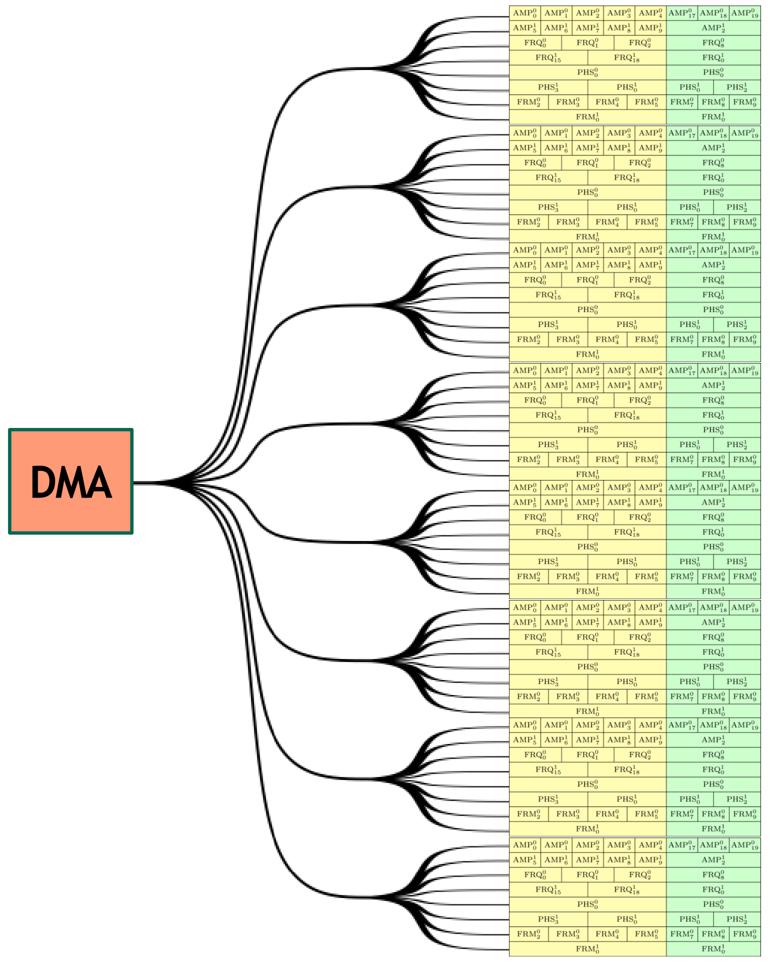

Gate
Pulse
Pulselet
AMP^0_0 AMP^0_1 AMP^0_2 AMP^0_3 AMP^0_4
AMP^1_5 AMP^1_6 AMP^1_7 AMP^1_8 AMP^1_9
FRQ^0_0 FRQ^0_1 FRQ^0_2
FRQ^1_{15} FRQ^1_{18}
PHS^0_0
PHS^1_3 PHS^1_0
FRM^0_2 FRM^0_3 FRM^0_4 FRM^0_5
FRM^1_0

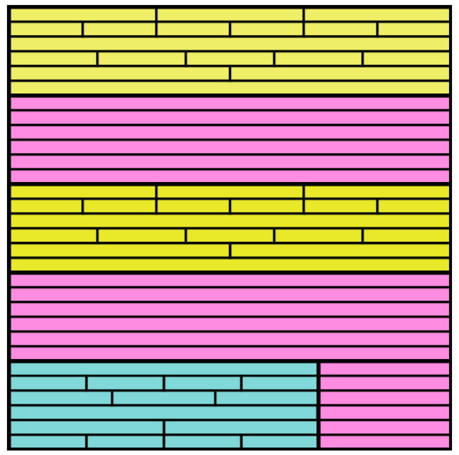
Data Ordering

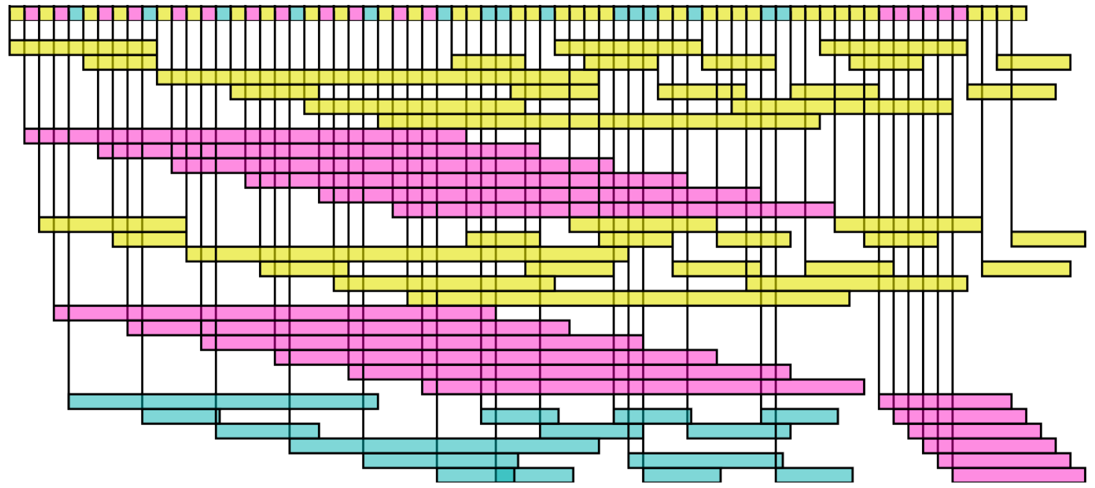
Data is fed to spline engines via DMA through a single bus and a switch network

Blocking conditions can lead to FIFO starvation, especially in cases of highly asymmetric data

Data need to be sorted based on time needed

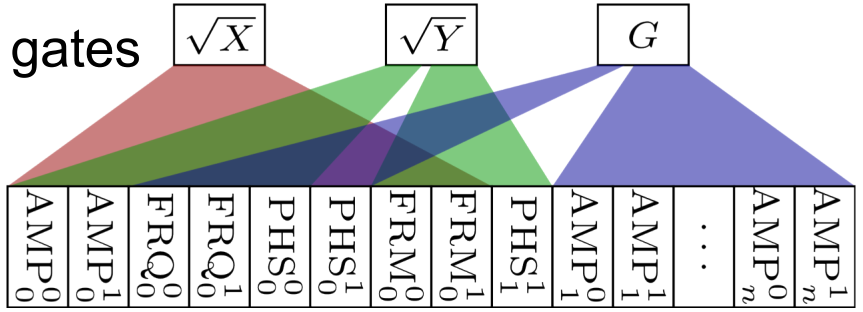






Reducing Overhead: Compressing Gate Data

In many cases, data can be reused, such as for X and Y gates



Only 2.25 Kb of raw data is needed to describe comparable X/Y gates (only one phase word differs)

→ Locally store data on chip and read out as necessary

Raw data stored in Pulse Lookup Table (PLUT)

- Gives a fixed factor of 12/216 in compression for raw data

PLUT

P_5	AMP_0^0
P_6	AMP_1^1
P_7	PHS_0^0
P_8	FRQ_3^1
P_9	FRM_0^0

Would be nice to iterate through address space, then only address boundaries are needed

- Data is not contiguous if more than 2 gates rely on the same spline segments
- Introduce a Mapping LUT (MLUT) to remap addresses so they are linearly ordered for each gate
- Iterator module is added to automatically step through addresses

Define gates with numeric ids and store in a Gate LUT (GLUT)

- Allows gates to be compressed to (currently) 11 bits, for a compression ratio < 0.00625
- Compression ratio is better for gates with a lot of modulation
- Restricts time ordering requirements to individual channels
- **More gates can be packed into a smaller memory footprint**
- **Allows for fast, measurement-based sequences (e.g. for**

~~Compression, Compression, Compression~~ Compression^3

QSCOUT

1. Minimizing samples: baseband frequencies of ~200 MHz can be generated by DDSs
2. Hardware-native phase bookkeeping: cuts down on amount of unique data needed

   - Global phase synchronization handled by custom DDS

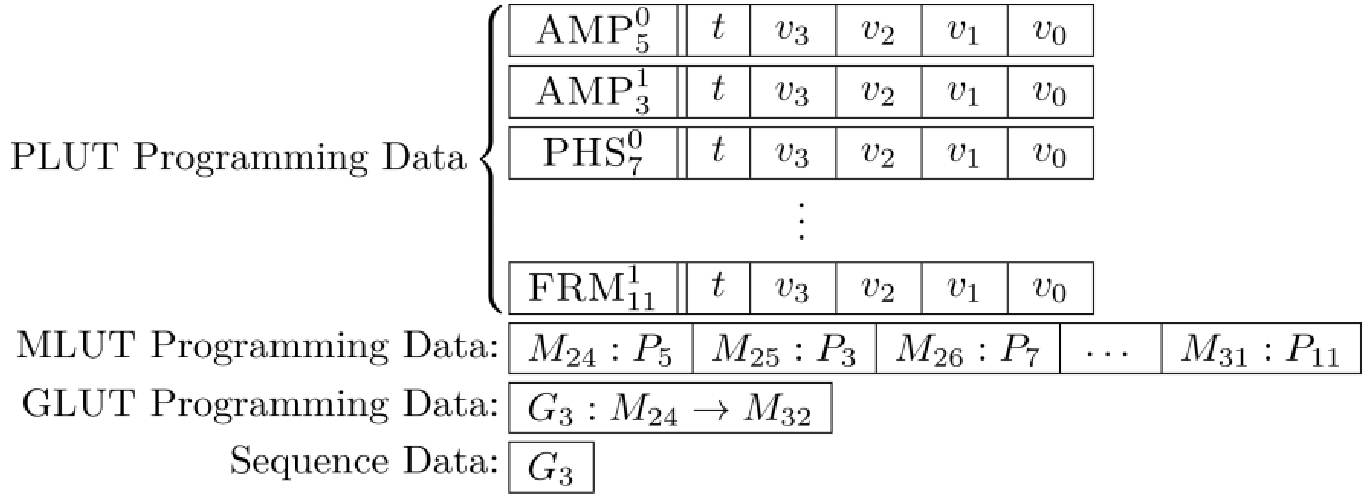
   - Dedicated phase accumulators track virtual phase, eliminating issues with context dependency
3. Cubic spline interpolators offer 10^2 to 10^4 reduction in data on average
4. Gate sequencer LUTs for storing pulse information locally

Data Flow

Initial programming data is uploaded before gates are sequenced

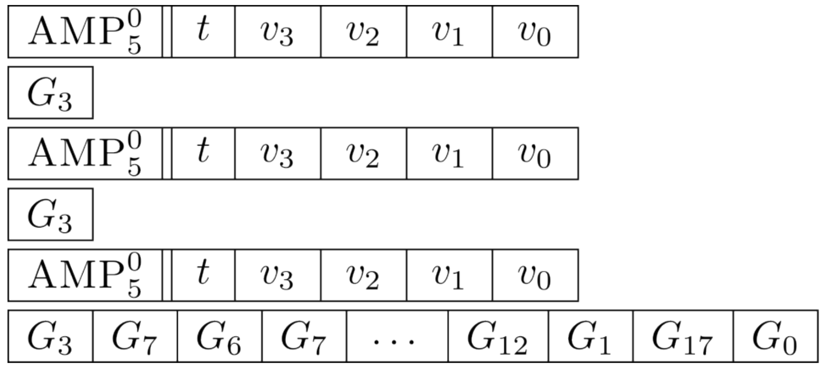
If a gate is called once, overhead is 3 extra words when compared to direct streaming

Subsequent gate calls are cheap, up to 20 gates per word

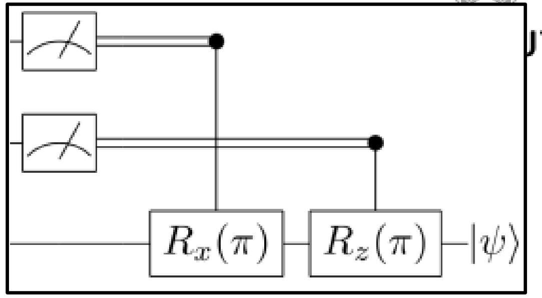

PLUT Programming Data
AMP_5^0 | t | v_3 | v_2 | v_1 | v_0
AMP_3^1 | t | v_3 | v_2 | v_1 | v_0
PHS_7^0 | t | v_3 | v_2 | v_1 | v_0
$\vdots$
FRM_{11}^1 | t | v_3 | v_2 | v_1 | v_0
MLUT Programming Data: $M_{24} : P_5$ | $M_{25} : P_3$ | $M_{26} : P_7$ | $\cdots$ | $M_{31} : P_{11}$
GLUT Programming Data: $G_3 : M_{24} \rightarrow M_{32}$
Sequence Data: G_3

Partial programming data can be interleaved with sequence data for circuits with a lot of unique gates

This is more efficient than direct streaming if fewer than N-1 parameters are updated for an N-parameter gate


AMP_5^0 | t | v_3 | v_2 | v_1 | v_0
G_3
AMP_5^0 | t | v_3 | v_2 | v_1 | v_0
G_3
AMP_5^0 | t | v_3 | v_2 | v_1 | v_0
G_3 | G_7 | G_6 | G_7 | $\cdots$ | G_{12} | G_1 | G_{17} | G_0

Fast Branching For Conditional Gate Sequences



When mid-circuit measurements require a conditional sequence of gates to be run, the hardware must be able to react quickly

For situations where these gates are known in advance, they can be passed to the hardware with a partial gate identifier (i.e. address for the gate LUT)

Given a gate identifier of **0b001010** and a measurement result of **0b0011**, the lookup value of the gate address is converted to **0b0011001010** using a matrix-style bitmask

0	0	1	1	0	0	1	0	1	0

Once a measurement result is complete, a secondary trigger is sent to the gate sequencer such that the additional latency only depends on the latency imposed by the measurement process and extra trigger

Moreover, depending on how the gate LUT is programmed, one can optionally and dynamically configure the aspect ratio of the matrix lookup, since measurement result masks are simply OR'd with the input gate identifier

1	1	0	0	0	0	1	0	1	0

1	1	0	0	0	0	1	0	1	0

1	1	0	0	0	0	1	0	1	0

Since gate identifiers are packed into 256-bit words, multiple gates can be applied based on a single measurement result and chained together across multiple 256-bit words to realize long measurement-based sequences

Pulse Managers

QSCOUT

The compiler has a dedicated Pulse Manager for each rf channel

Pulse Managers are responsible for organizing spline data associated with different gates

They mirror the structure of the gate sequencer LUTs for tracking associated data across gates

Extra tables are used for memoization of spline inputs to avoid refitting

Makes use of Array of Structs (AoS) and Struct of Array (SoA) schemes for fast lookup

Pulses:	[0: {P[0], P[1], P[4], P[9], …}, 1: {P[12], P[1], P[4], P[9], …}]
Pulselets:	[0: {Spl[5], Spl[6], Spl[7], …}, 1: {Spl[0]}, …}]
Spline Segments:	[0: {Ch: 1, Type: Amplitude, Duration(clk): 132, U0: 235125, U1: 23523..}, …]

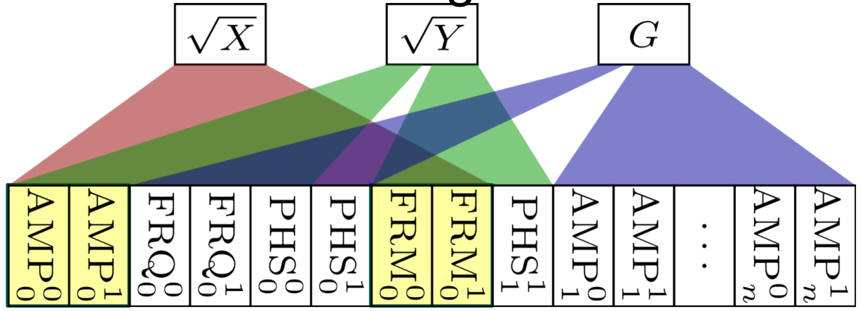
Feedback on Gate Definitions

Feedback that requires complex algorithms or advanced gate designs is not always as trivial as incrementing/decrementing a parameter

Example: Gaussian amplitude modulation in the presence of AOM distortion

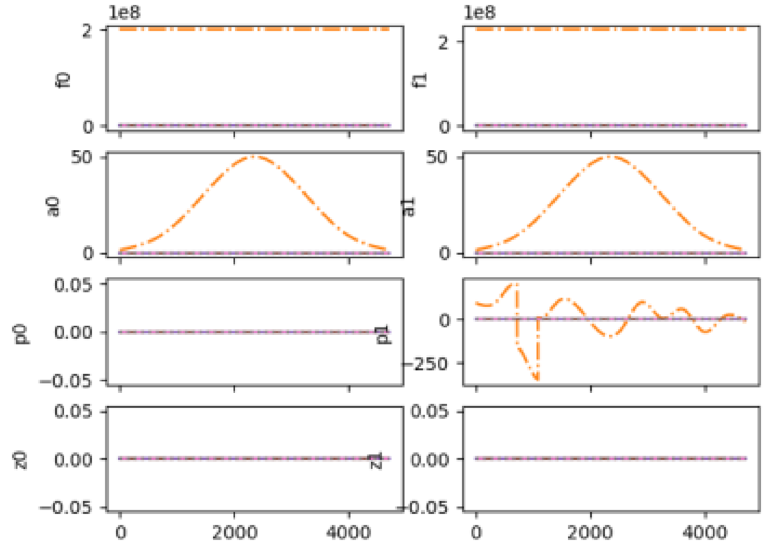
- Gaussian needs to be calculated and mapped to account for the distortion profile
- Splines need to be recalculated/encoded for the gate sequencers

Approach: Use in-situ mutations of gate data to minimize reprogramming



Mutated data can be shared among different “classes” of gates

Optional “mutation ids” can strictly tag similar classes of gates to avoid undesired overlap among gates



Compiler Performance Case Study: On-Chip Gate Mutations

QSCOUT

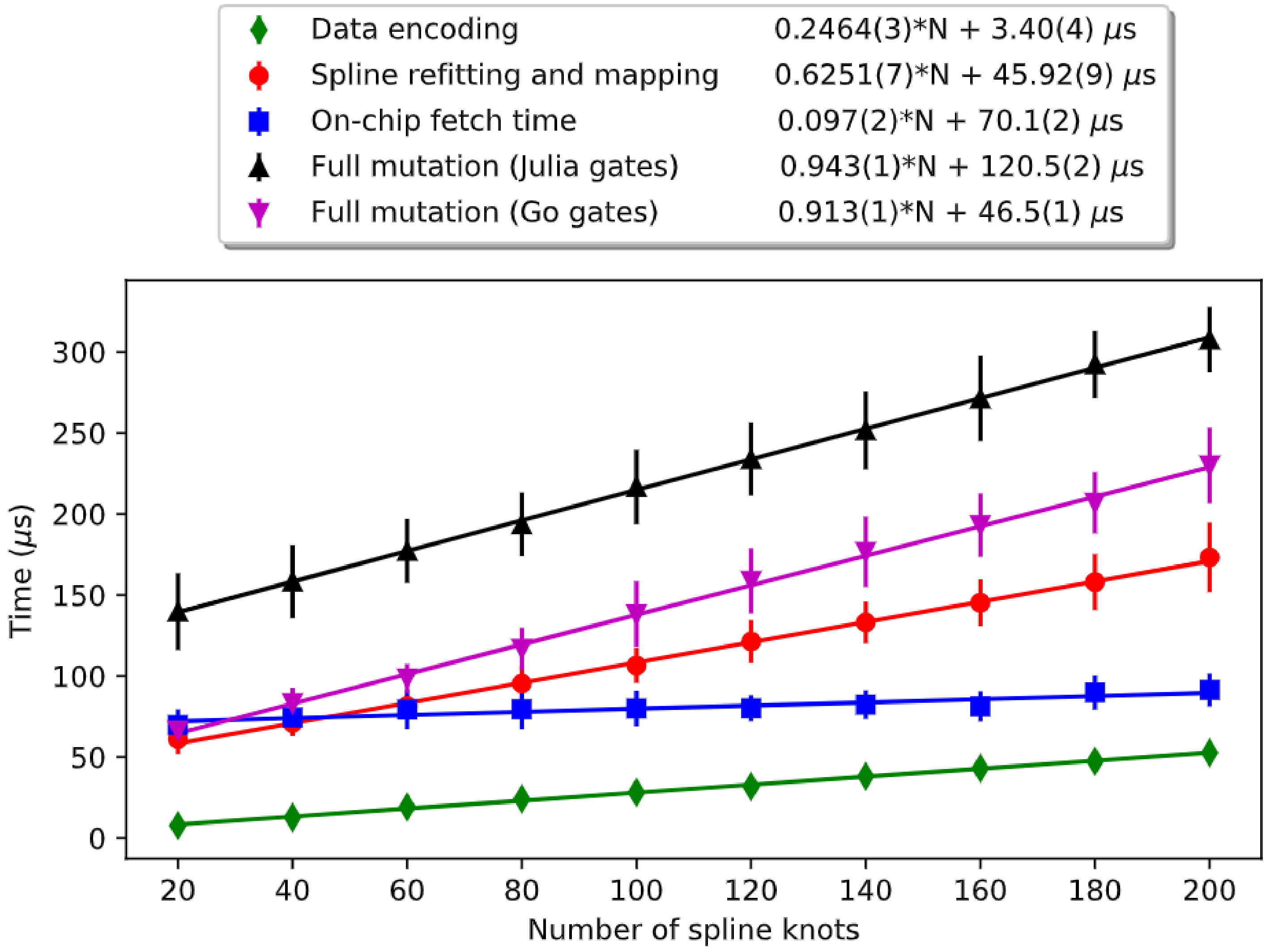
Gates tested dominantly for Gaussian amplitude modulation, symmetric across tones

All parameters updated during mutation

Times are shorter than the 1 ms Doppler cooling stage used for repreparing ions after detection

Off-chip fetch times are ~1 ms, with ~170-200 us for Protobuf serialization alone

On-chip mutation is faster than off-chip mutation when accounting for upload times



Parser Performance

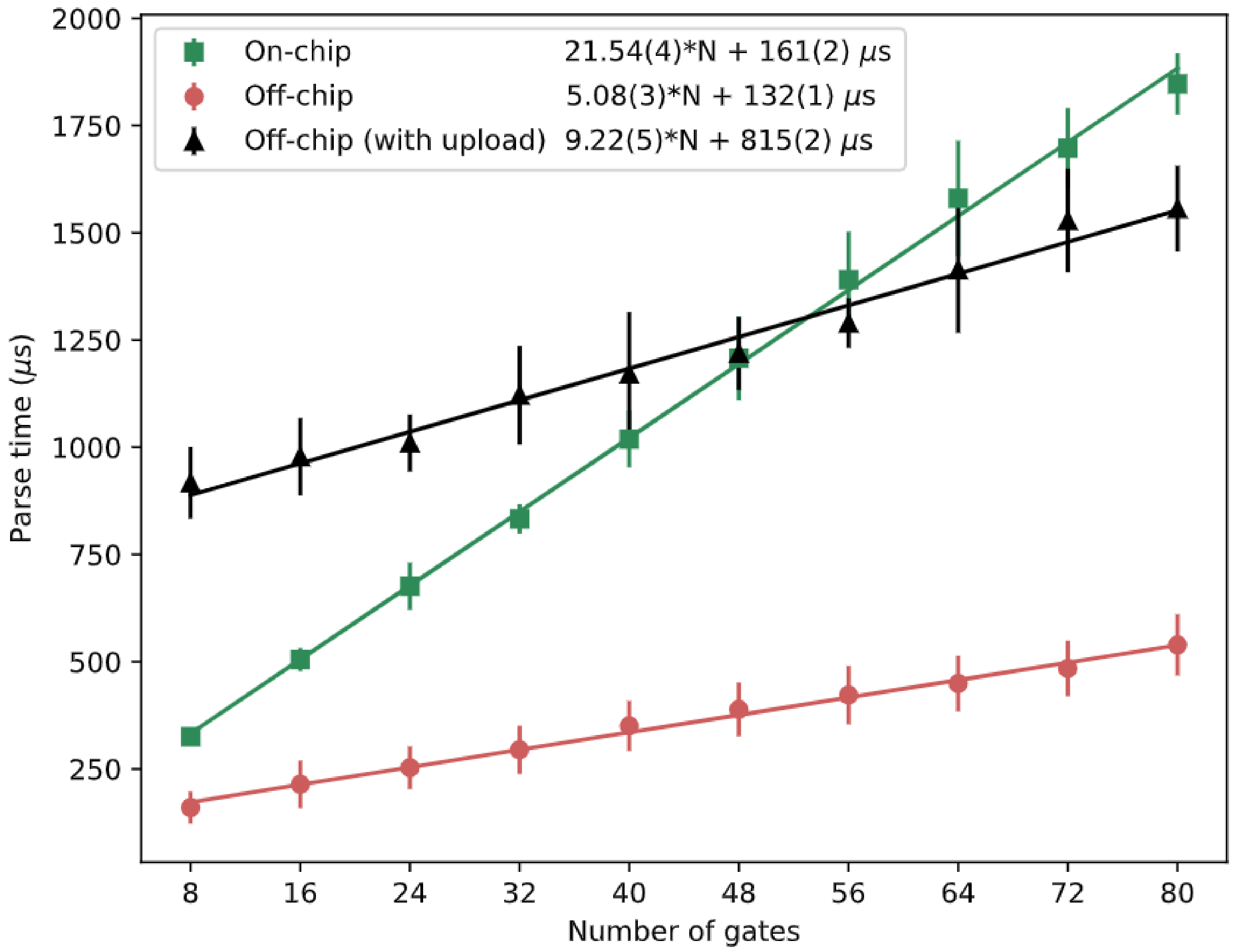
Recursive-descent parser

- Well-suited to Jaqal due to its lack of left-branching grammar rules

Parsed outputs are recast into a set of tables, referred to as a “tabulated” intermediate representation (TIR)

The TIR stores unique gate calls (gate name and inputs), macros, loops, and other elements into distinct tables once, and subsequently refers to them by index, thus removing redundancies

This provides the first stage of compression for the final representation of gates on hardware



Tabulated Intermediate Representation

Jaqal

```
from qscout.v1.std usepulses *

let pi 3.141592653589793
let pi2 1.5707963267948966
let phi1 1.234
let phi2 2.456

register q[8]

prepare_all
G q[1] 0 phi1
G q[3] pi2 phi2
< G q[1] 0 phi1 | G q[3] pi2 phi2 >
loop 10 {
    G q[1] 0 phi1
    < G q[1] 0 phi1 | G q[3] pi2 phi2 >
}
G q[3] pi2 phi2
measure_all
```

Tabulated Intermediate Representation

```
circuit {
  constants [{ name: "pi"   value: 3.141592653589793 }
             { name: "pi2"  value: 1.5707963267948966 }
             { name: "phi1" value: 1.234 }
             { name: "phi2" value: 2.456 }]
  registers { name: "q" size: 8 }
  imports { source: "qscout.v1.std" }
  gate_table [{ index: 0
                name: "G"
                args [{ type: QUBIT string_value: "q" arguments { value: 1 } }
                      { type: INTEGER value: 0}
                      { type: CONSTANT string_value: "phi1" }]
              { index: 1
                name: "G"
                args [{ type: QUBIT string_value: "q" arguments { value: 3 } }
                      { type: CONSTANT string_value: "pi2" }
                      { type: CONSTANT string_value: "phi2" }] }]
  block_table [{ index: 2
                 block_type: PARALLEL
                 statements: [ 0 1 ] }
               { index: 3
                 block_type: LOOP
                 argument { typ: INTEGER value: 10 }
                 statements: [ 1 2 ] }
               { index: 4
                 block_type: SUBCIRCUIT
                 statements: [ 0 1 2 3 1 ] }]
  body: 4
}
```

Tabulated Intermediate Representation

QSCOUT

Jaqal

```
from qscout.v1.std usepulses *

let pi 3.141592653589793
let pi2 1.5707963267948966
let phi1 1.234
let phi2 2.456

register q[8]

prepare_all
G q[1] 0 phi1
G q[3] pi2 phi2
< G q[1] 0 phi1 | G q[3] pi2 phi2 >
loop 10 {
    G q[1] 0 phi1
    < G q[1] 0 phi1 | G q[3] pi2 phi2 >
}
G q[3] pi2 phi2
G q[3] pi2 phi2
G q[1] 0 phi1
G q[3] pi2 phi2
< G q[1] 0 phi1 | G q[3] pi2 phi2 >
G q[3] pi2 phi2
G q[1] 0 phi1
< G q[1] 0 phi1 | G q[3] pi2 phi2 >
G q[1] 0 phi1
G q[1] 0 phi1
< G q[1] 0 phi1 | G q[3] pi2 phi2 >
< G q[1] 0 phi1 | G q[3] pi2 phi2 >
G q[1] 0 phi1
< G q[1] 0 phi1 | G q[3] pi2 phi2 >
G q[1] 0 phi1
measure_all
```

Tabulated Intermediate Representation

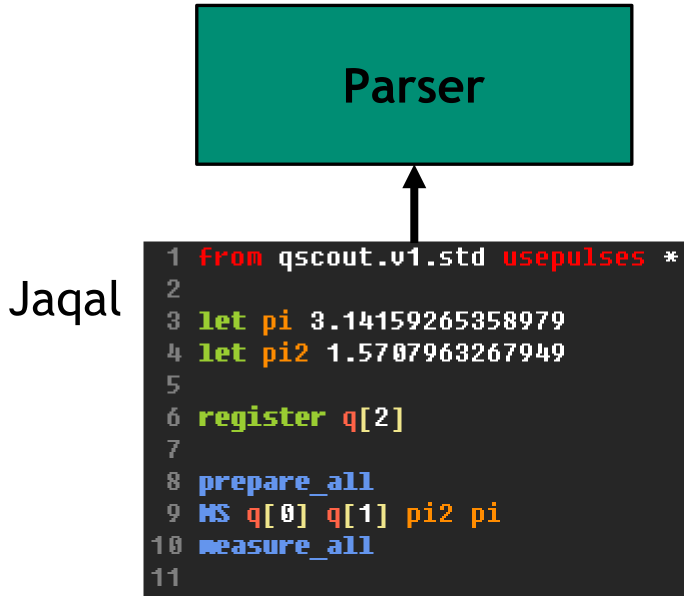
```
circuit {
  constants [{ name: "pi"   value: 3.141592653589793 }
             { name: "pi2"  value: 1.5707963267948966 }
             { name: "phi1" value: 1.234 }
             { name: "phi2" value: 2.456 }]
  registers { name: "q" size: 8 }
  imports { source: "qscout.v1.std" }
  gate_table [{ index: 0
                name: "G"
                args [{ type: QUBIT string_value: "q" arguments { value: 1 } }
                      { type: INTEGER value: 0}
                      { type: CONSTANT string_value: "phi1" }]
              { index: 1
                name: "G"
                args [{ type: QUBIT string_value: "q" arguments { value: 3 } }
                      { type: CONSTANT string_value: "pi2" }
                      { type: CONSTANT string_value: "phi2" }] }]
  block_table [{ index: 2
                 block_type: PARALLEL
                 statements: [ 0 1 ] }
               { index: 3
                 block_type: LOOP
                 argument { typ: INTEGER value: 10 }
                 statements: [ 1 2 ] }
               { index: 4
                 block_type: SUBCIRCUIT
                 statements: [ 0 1 2 3 1 1 0 1 2 1 0 2 0 0 2 2 0 2 0 ] }]
  body: 4
}
```

~~Compression, Compression, Compression~~ Compression^3

1. Minimizing samples: baseband frequencies of ~200 MHz can be generated by DDSs
2. Hardware-native phase bookkeeping: cuts down on amount of unique data needed
   - Global phase synchronization handled by custom DDS
   - Dedicated phase accumulators track virtual phase, eliminating issues with context dependency
3. Cubic spline interpolators offer 10^2 to 10^4 reduction in data on average
4. Gate sequencer LUTs store unique pulse information locally
5. Parser distills unique gate calls resulting in a compressed intermediate representation

Non-local Gate Definitions

Jaqal is first parsed, then converted to a compressed intermediate representation (IR)

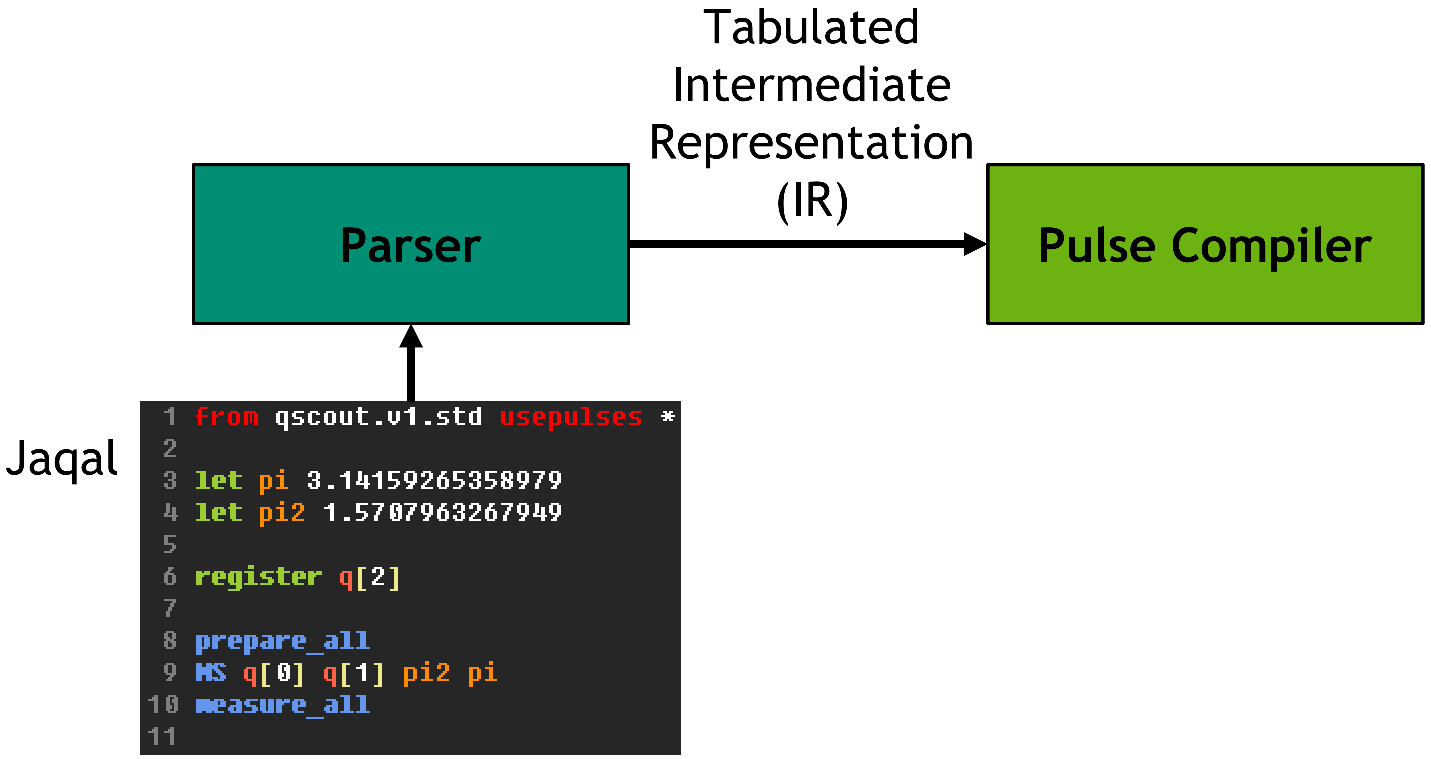


```
from qscout.v1.std usepulses *

let pi 3.14159265358979
let pi2 1.5707963267949

register q[2]

prepare_all
MS q[0] q[1] pi2 pi
measure_all

```

Non-local Gate Definitions

Jaqal is first parsed, then converted to a compressed intermediate representation (IR)

The IR is sent to the pulse compiler, which looks for existing gate definitions



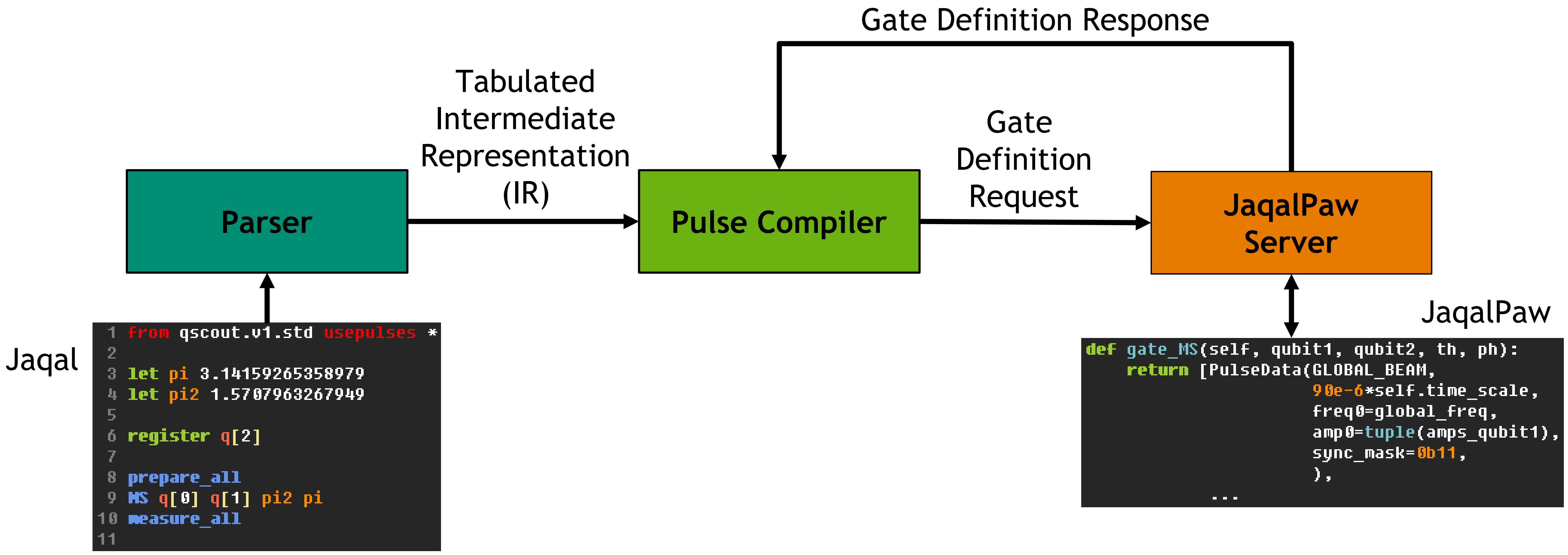
```
from qscout.v1.std usepulses *

let pi 3.14159265358979
let pi2 1.5707963267949

register q[2]

prepare_all
MS q[0] q[1] pi2 pi
measure_all

```

Non-local Gate Definitions

Jaqal is first parsed, then converted to a compressed intermediate representation (IR)

The IR is sent to the pulse compiler, which looks for existing gate definitions

If a gate isn't defined, or needs an updated definition, the compiler requests a new one from a JaqalPaw server



```
from qscout.v1.std usepulses *

let pi 3.14159265358979
let pi2 1.5707963267949

register q[2]

prepare_all
MS q[0] q[1] pi2 pi
measure_all

```

```
def gate_MS(self, qubit1, qubit2, th, ph):
    return [PulseData(GLOBAL_BEAM,
                      90e-6*self.time_scale,
                      freq0=global_freq,
                      amp0=tuple(amps_qubit1),
                      sync_mask=0b11,
                      ),
            ...
```

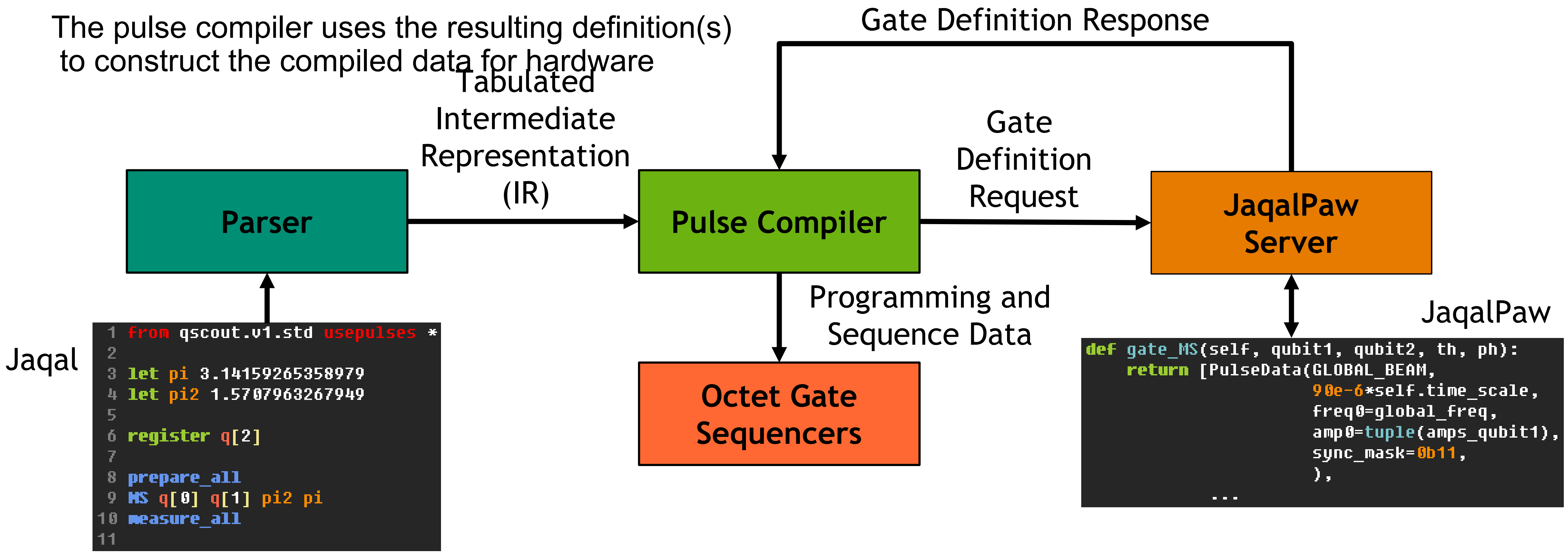
Non-local Gate Definitions

Jaqal is first parsed, then converted to a compressed intermediate representation (IR)

The IR is sent to the pulse compiler, which looks for existing gate definitions

If a gate isn't defined, or needs an updated definition, the compiler requests a new one from a JaqalPaw server

The pulse compiler uses the resulting definition(s) to construct the compiled data for hardware



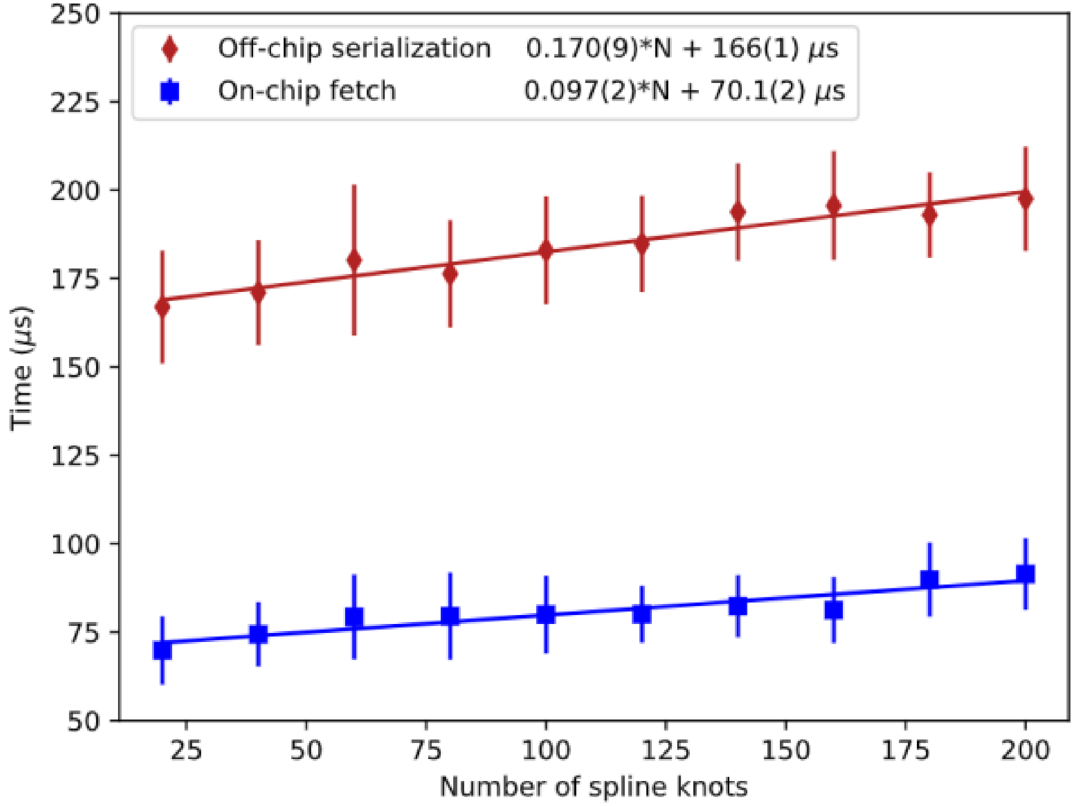
```
from qscout.v1.std usepulses *

let pi 3.14159265358979
let pi2 1.5707963267949

register q[2]

prepare_all
MS q[0] q[1] pi2 pi
measure_all
```

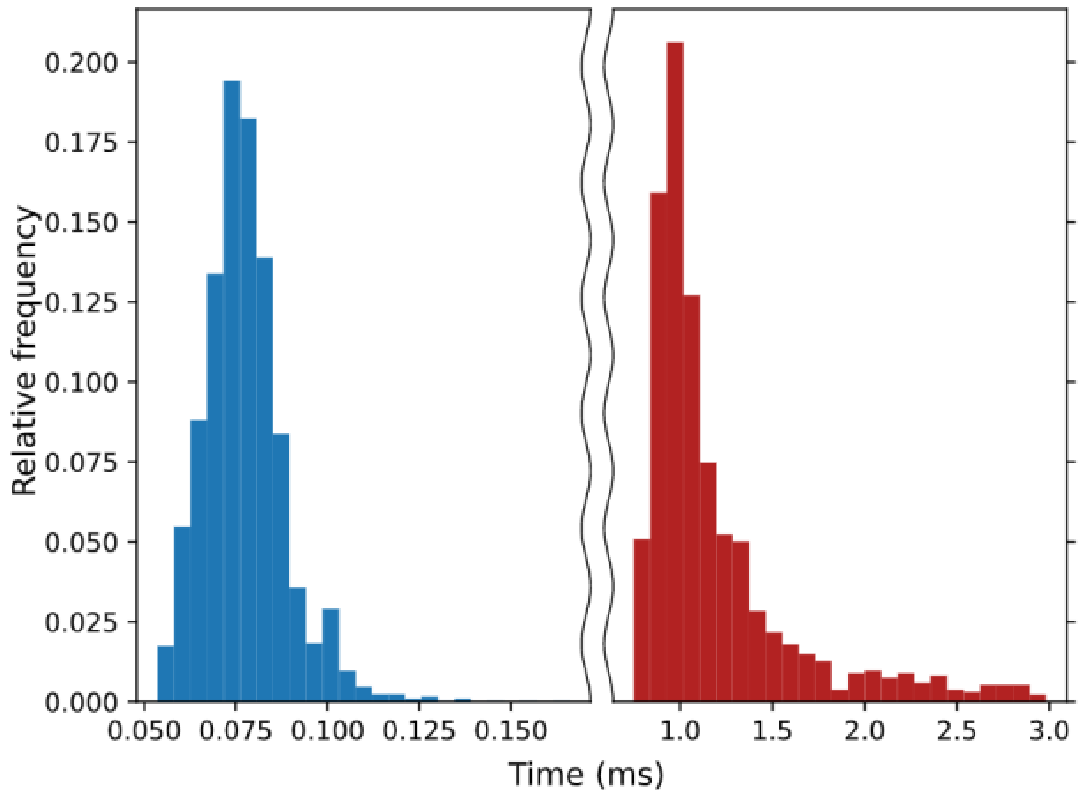
```
def gate_MS(self, qubit1, qubit2, th, ph):
    return [PulseData(GLOBAL_BEAM,
                      90e-6*self.time_scale,
                      freq0=global_freq,
                      amp0=tuple(amps_qubit1),
                      sync_mask=0b11,
                      ),
            ...
```

On-Chip Gate Definition Performance

Protobuf serialization time on an external machine is longer than generating on-chip gates (left)

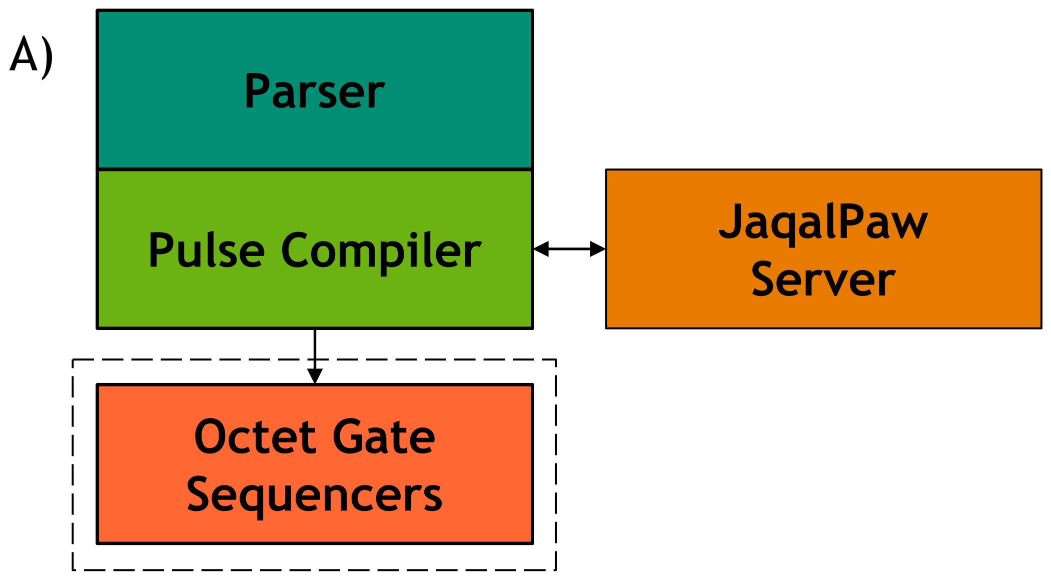
Taking upload times into account, the on-chip speedup is ~10-15x





Compiler Topologies

A) Parser and Pulse Compiler are on one machine, JaqalPaw Server is (optionally) on another machine. Octet Gate Sequencer handling is on chip.

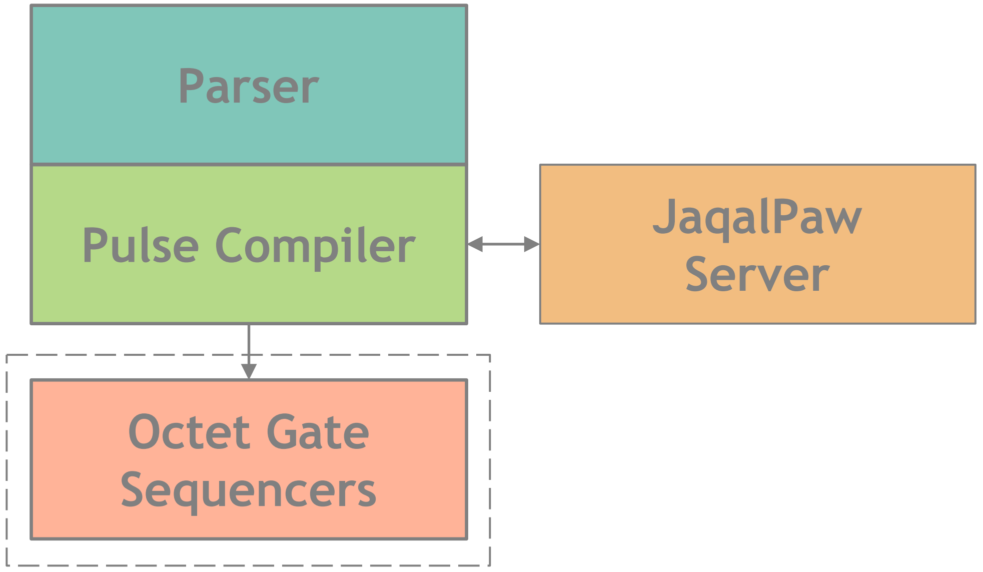


Compiler Topologies

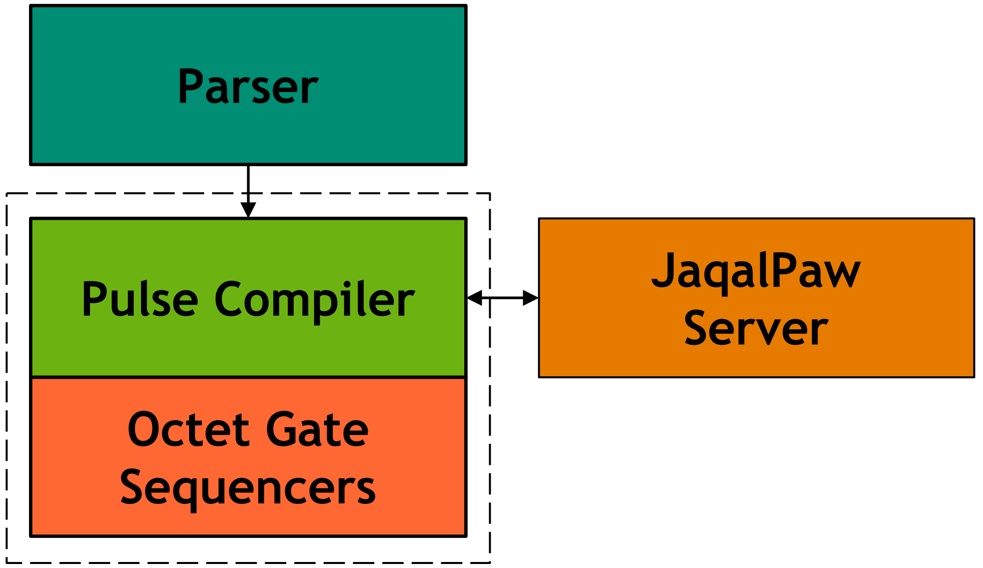
A) Parser and Pulse Compiler are on one machine, JaqalPaw Server is (optionally) on another machine. Octet Gate Sequencer handling is on chip.

B) Parser is on one machine, JaqalPaw Server is (optionally) on another machine. Pulse Compiler and Octet Gate Sequencer handling is on chip.

A)



B)

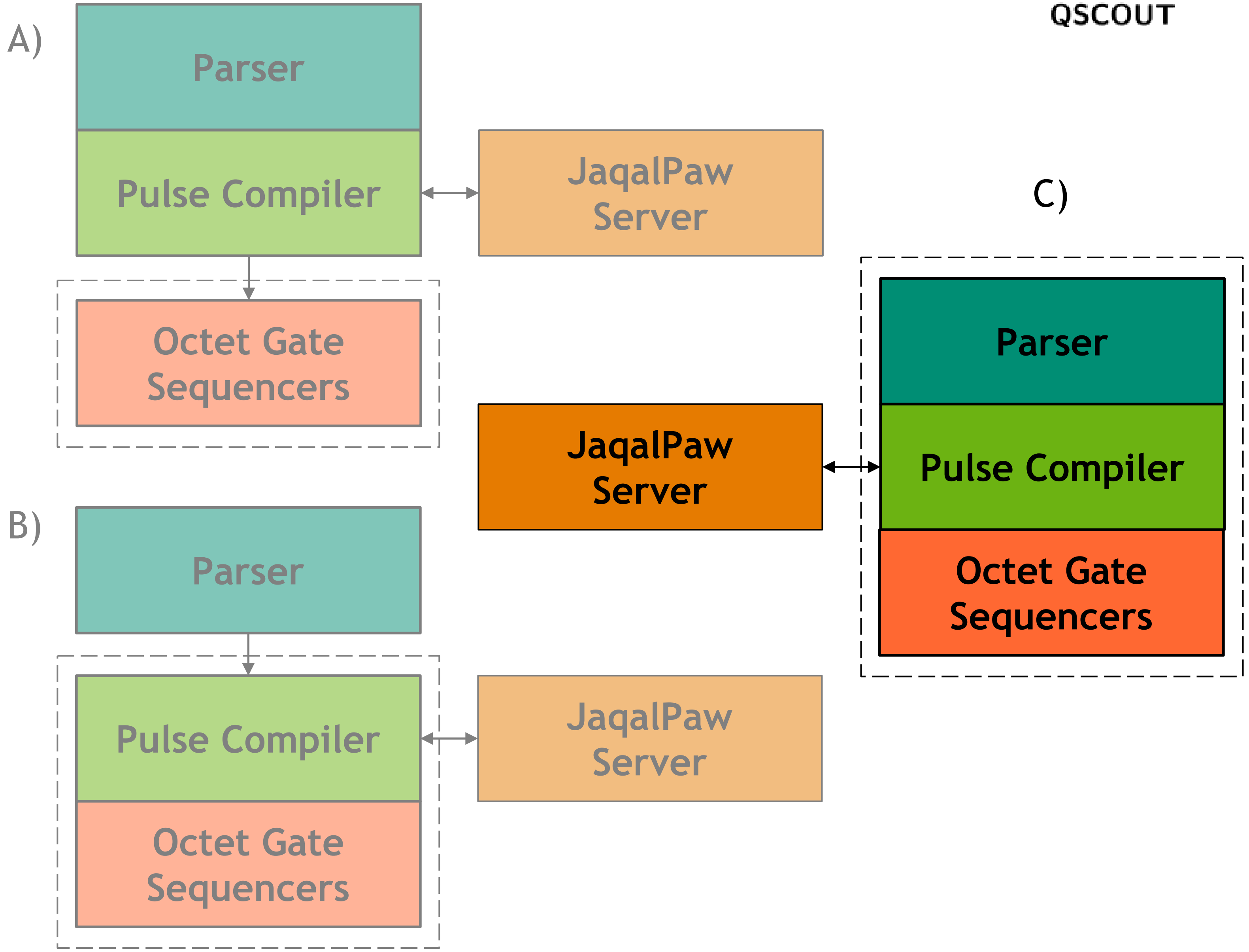


Compiler Topologies

A) Parser and Pulse Compiler are on one machine, JaqalPaw Server is (optionally) on another machine. Octet Gate Sequencer handling is on chip.

B) Parser is on one machine, JaqalPaw Server is (optionally) on another machine. Pulse Compiler and Octet Gate Sequencer handling is on chip.

C) JaqalPaw Server is on one machine. Parser, Pulse Compiler, and Octet Gate Sequencer handling is on chip.



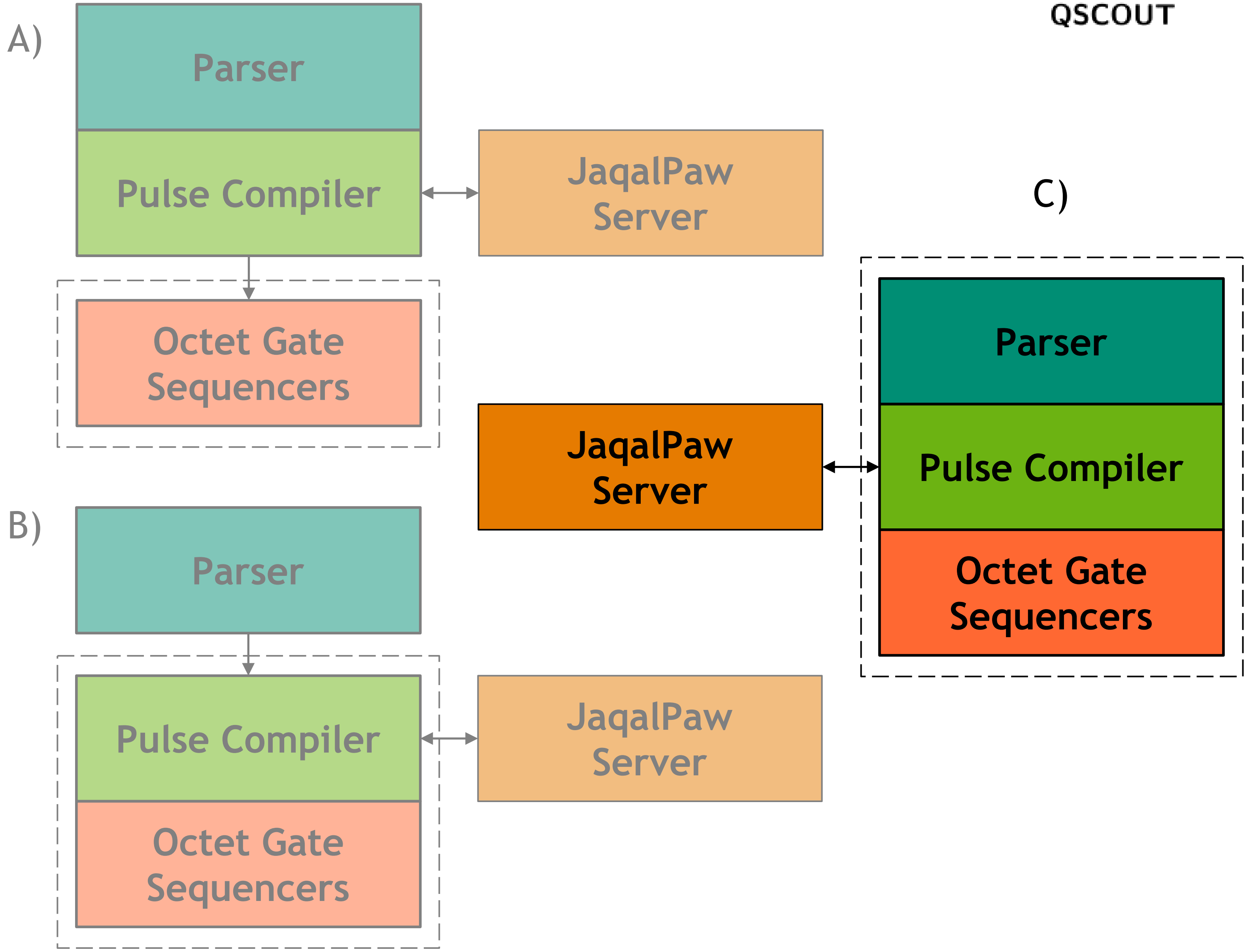
Compiler Topologies

A) Parser and Pulse Compiler are on one machine, JaqalPaw Server is (optionally) on another machine. Octet Gate Sequencer handling is on chip.

B) Parser is on one machine, JaqalPaw Server is (optionally) on another machine. Pulse Compiler and Octet Gate Sequencer handling is on chip.

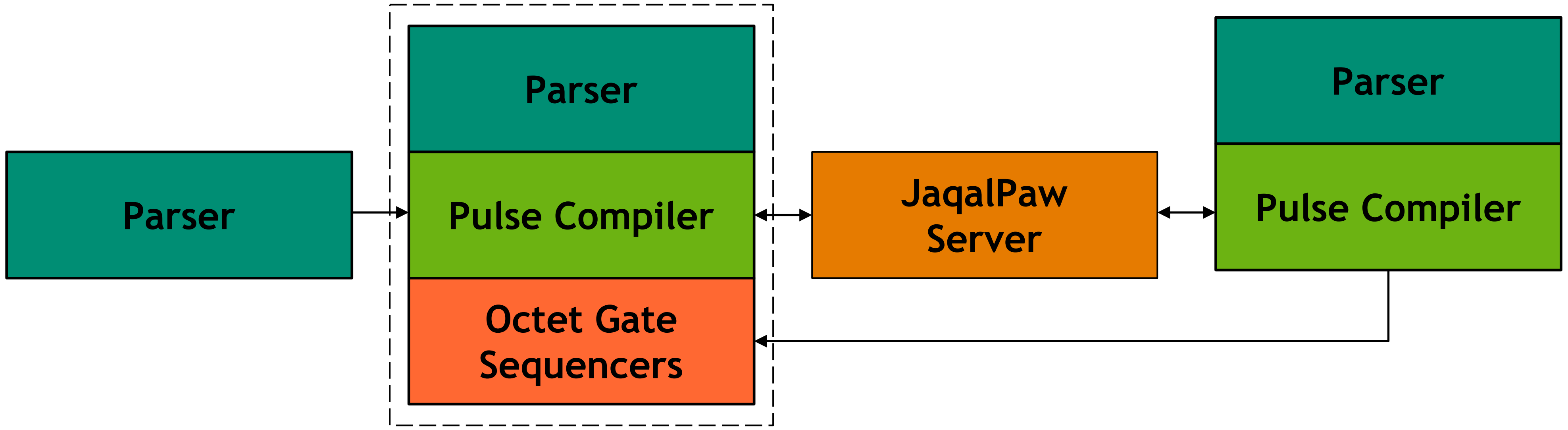
C) JaqalPaw Server is on one machine. Parser, Pulse Compiler, and Octet Gate Sequencer handling is on chip.

D) (Not shown). Everything on chip, but still need to send relevant calibration data



Simultaneous Support for All Topologies

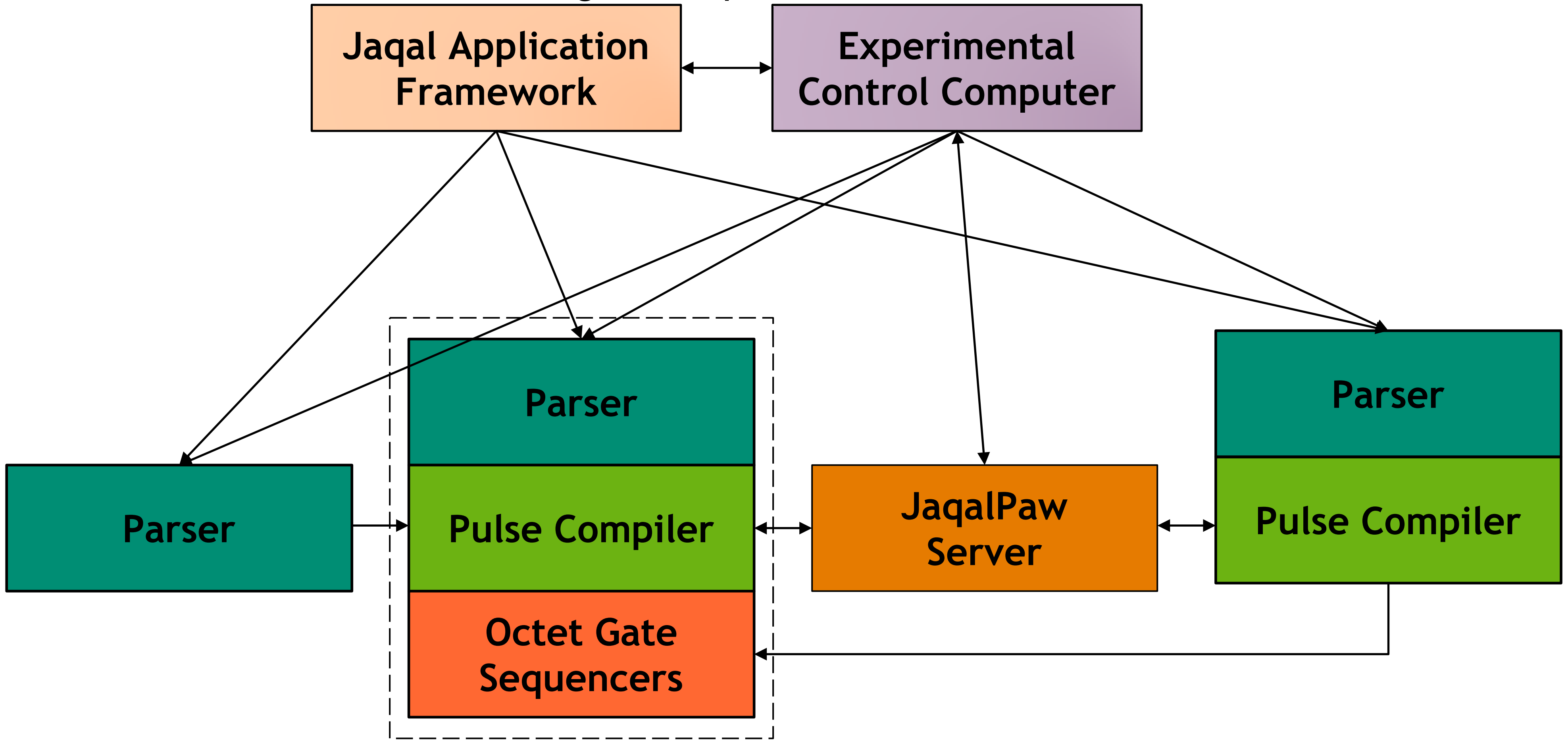
We can run all topologies simultaneously.



Simultaneous Support for All Topologies

We can run all topologies simultaneously.

Can use the configuration that makes the most sense on a case-by-case basis. For example, massive Jaqal files will parse more efficiently on a normal PC, but smaller files and mutative algorithms are better suited for running on chip.



Collaborators

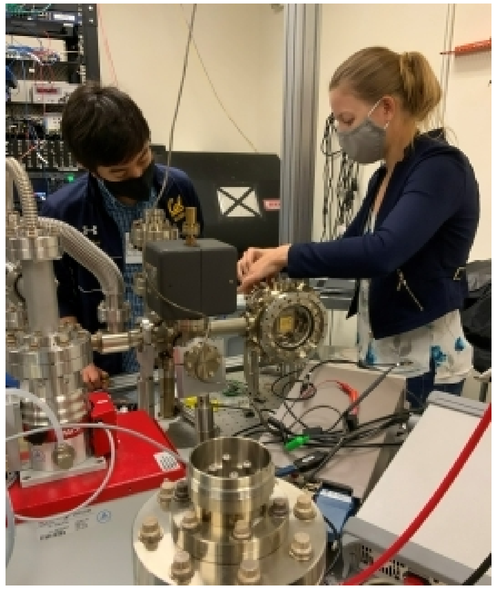
Embedded
Dan Lobser
Jay Van Der Wall
Josh Goldberg

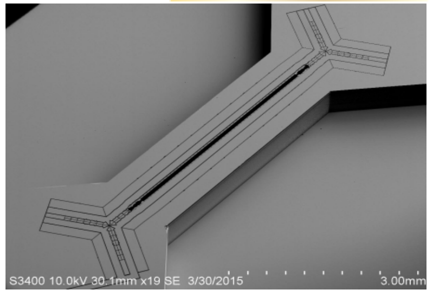
Theory & Software
Andrew Landahl
Ben Morrison
Kenny Rudinger
Antonio Russo
Brandon Ruzic
Jay Van Der Wall
Josh Goldberg
Tim Proctor
Kevin Young

Experimental
Susan Clark, PI
Christopher Yale
Dan Lobser
Melissa Revelle
Matt Chow
Ashlyn Burch
Megan Ivory
Theala Redhouse
Josh Wilson
Craig Hogle
Dan Stick

Collaborators
Alan Bell (AOSense)
Ken Brown (Duke)
Marko Cetina (Duke)
Nafis Irtija (UNM)
Jungsang Kim (Duke)
Chris Monroe (Duke)
Jim Plusquellic (UNM)
Eirini Tsiropolou (UNM)

2021 R&D 100 WINNER





Email: qscout@sandia.gov (mailing list)
Web: https://qscout.sandia.gov
Jaqal: https://gitlab.com/jaqal/jaqalpaq

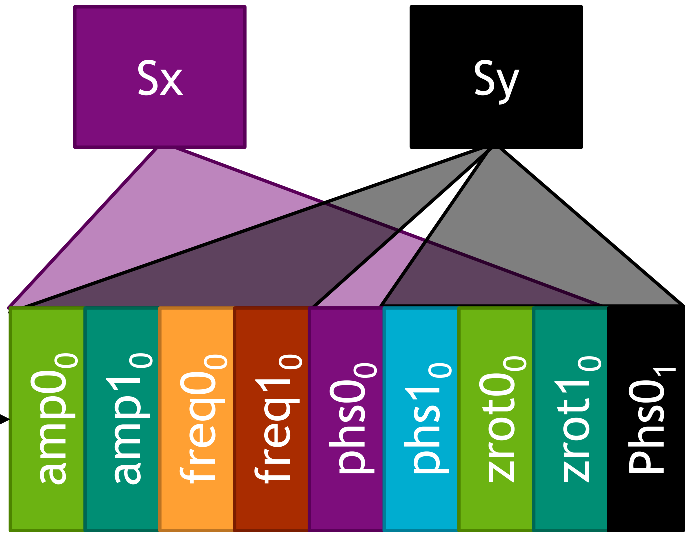
Funding
US Department of Energy, Office of Science, Office of Advanced Scientific
Computing Research Quantum Testbed Program and National

Gate Mutations

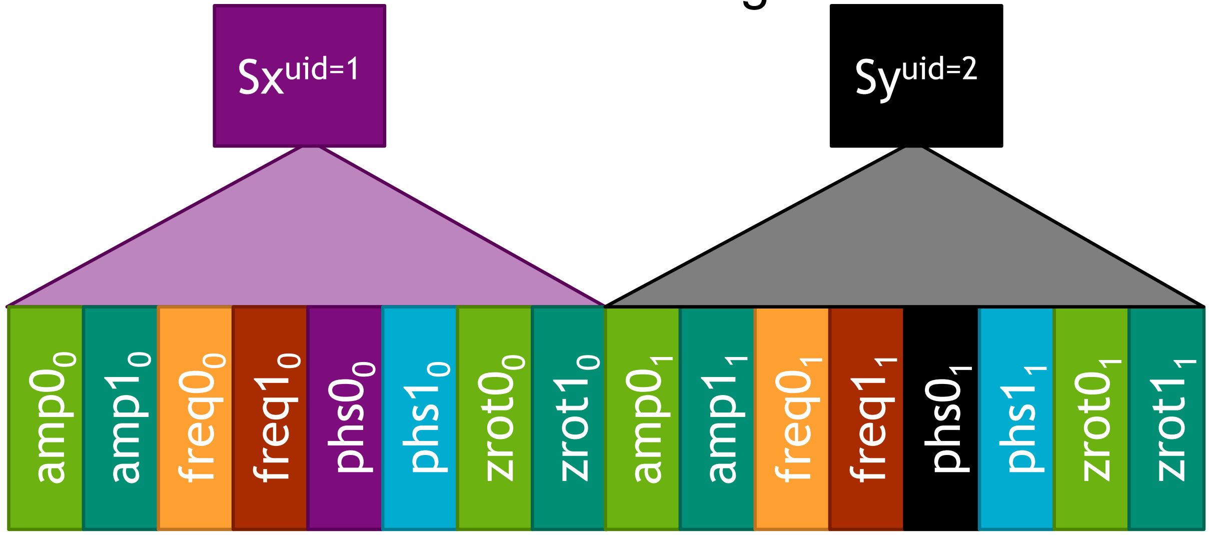
Gates are broken up into low level pulse information and duplicate information is shared across gates

If a gate needs to be updated, adding new information to the lookup tables can cause fragmentation and eats up precious memory in firmware lookup tables

Instead, gate data can be overwritten in place by mutating pulse information at the lowest level



Mutated data can affect other gates!



Gates can be assigned a “uniqueness” identifier, that can be used to prevent data de-duplication for a single gate, or a class of gates which are expected to share common data

Non-local Gate Mutations

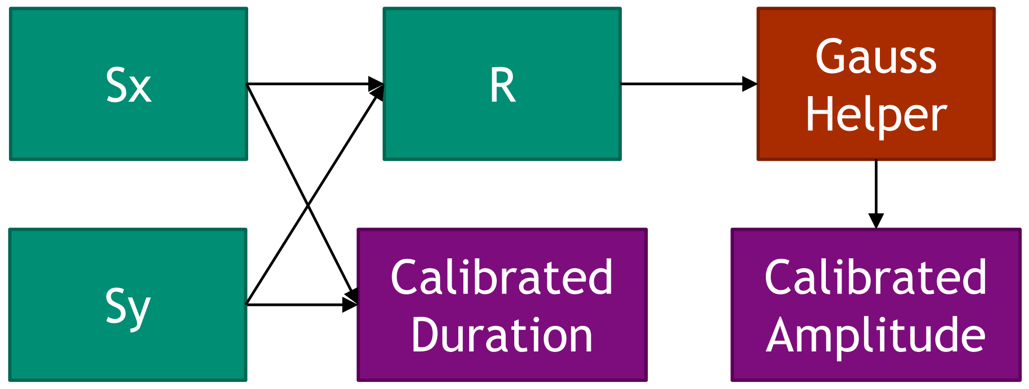
Gates typically rely on calibration parameters, and mutations typically target these parameters

Gates can call other functions, including from external libraries, as well as other gates, all of which can rely on calibration parameters

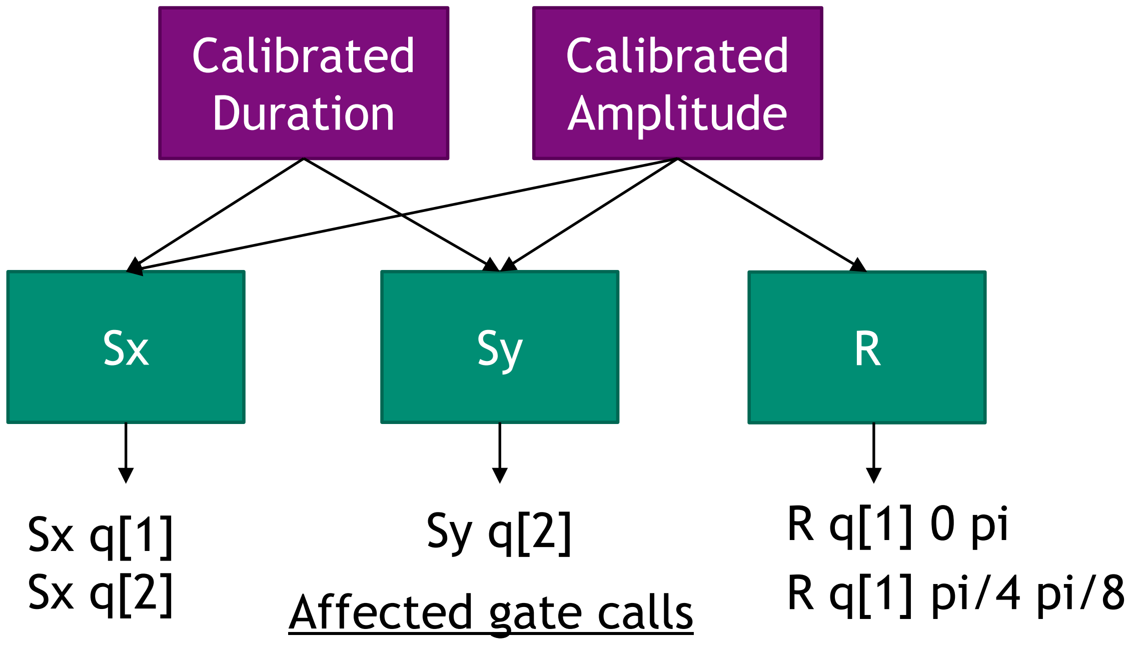
We back out a dependency graph for all calibration parameters and all affected gates

The gate requests from the gate compiler are collected for a particular circuit to re-construct gate definitions for each gate and the associated inputs

Full dependency graph



Inverted calibration/gate dependency graph

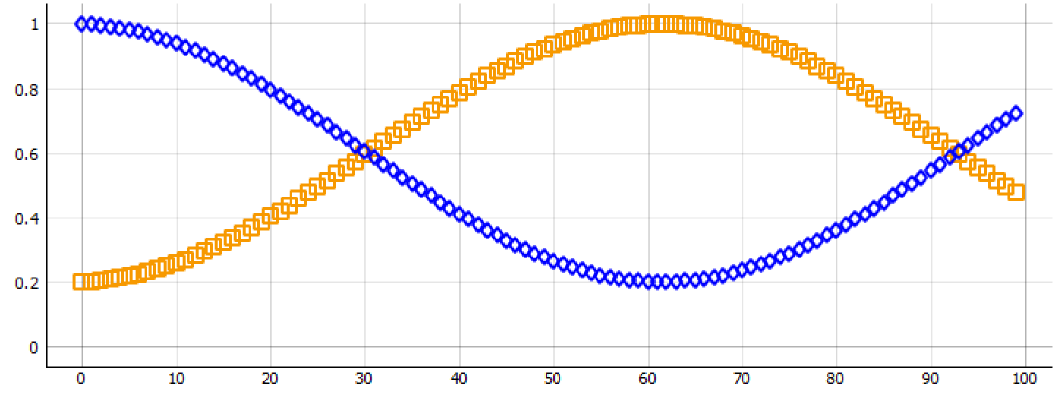


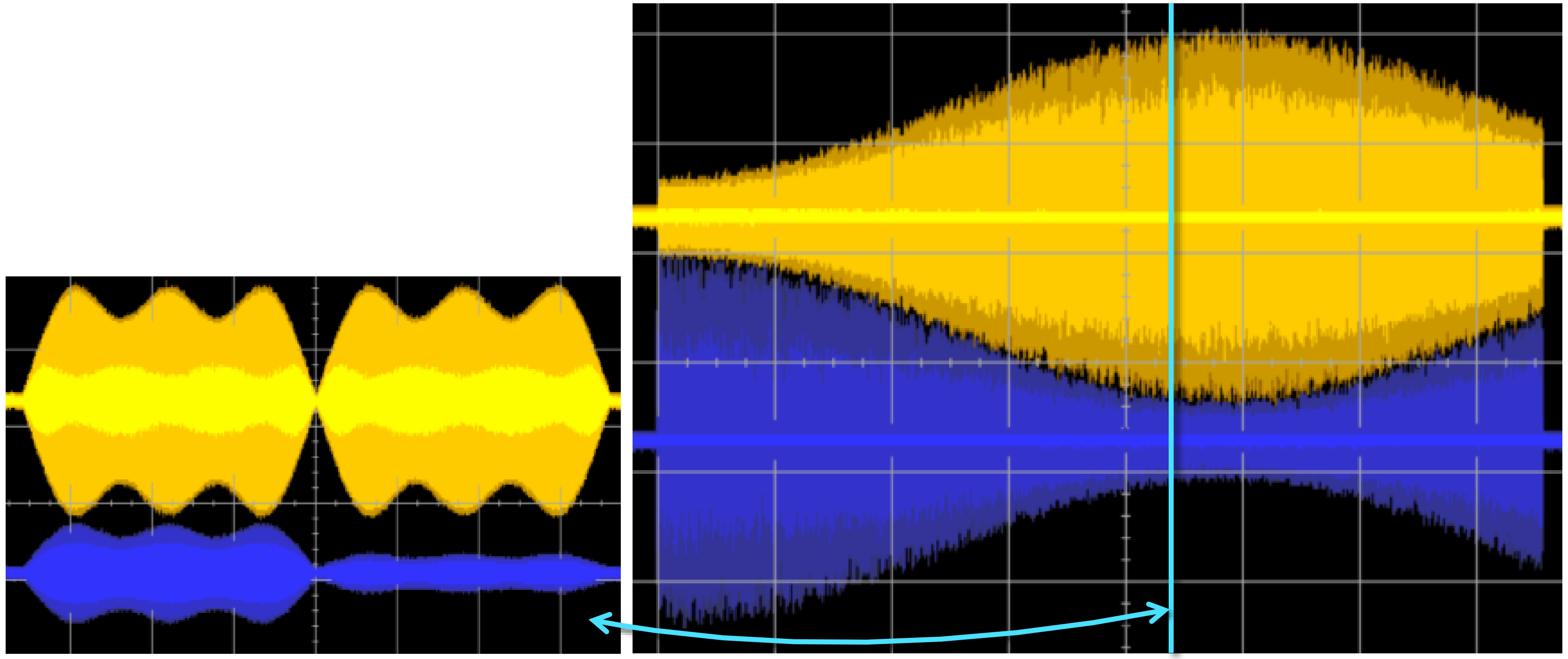
Drift Control

QSCOUT

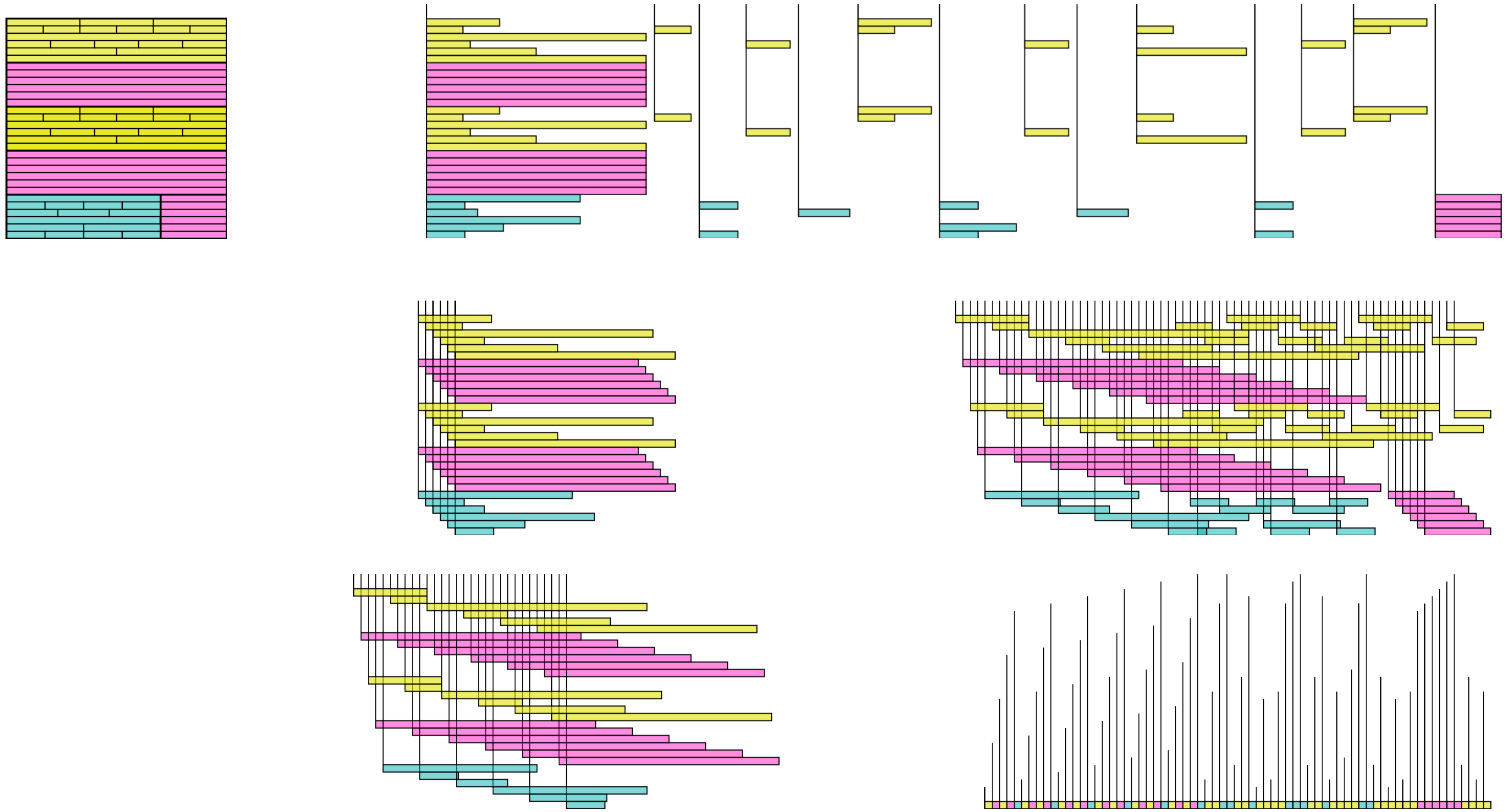
Experimental result is intercepted and used to modify one or more calibration parameters based on a user-defined function

The updated parameters trigger necessary mutations for the affected gates used in the current circuit to generate the next point





Data Ordering



RPU-Driven Sequences

Subcircuits cherry picked based on results of Collatz sequences calculated with the RPU

Static registers used to update overall amplitude scaling before each circuit, amplitudes determined from trig functions on the RPU