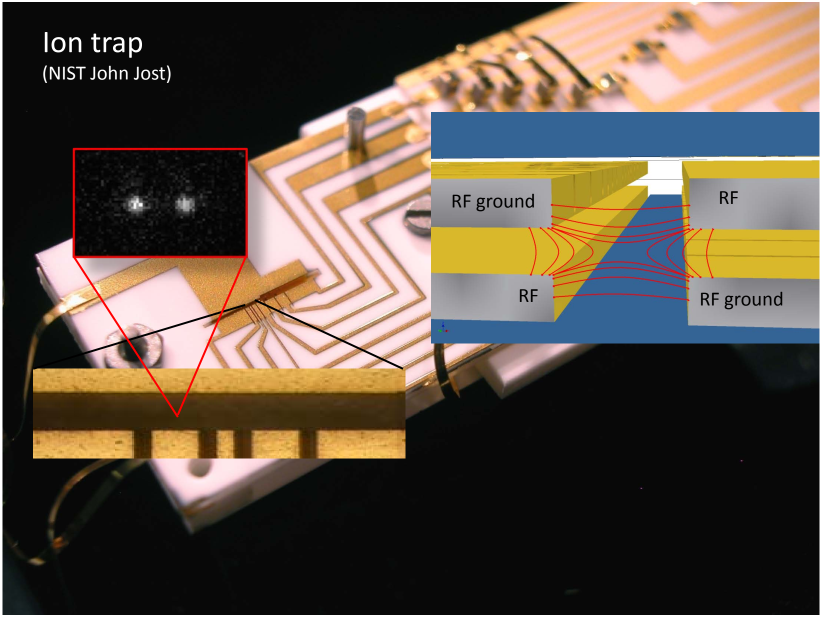
Real-time experiment control for quantum physics

Robert Jördens

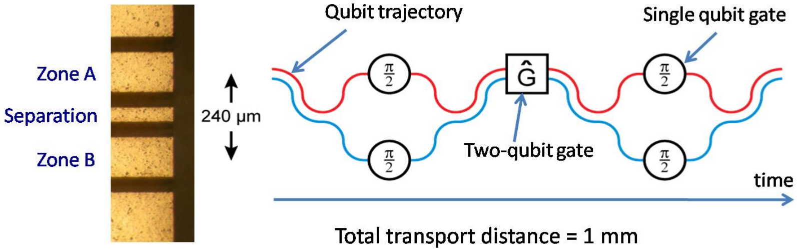
Ion Storage Group, Time and Frequency, NIST, Boulder, CO
`rjordens@nist.gov`

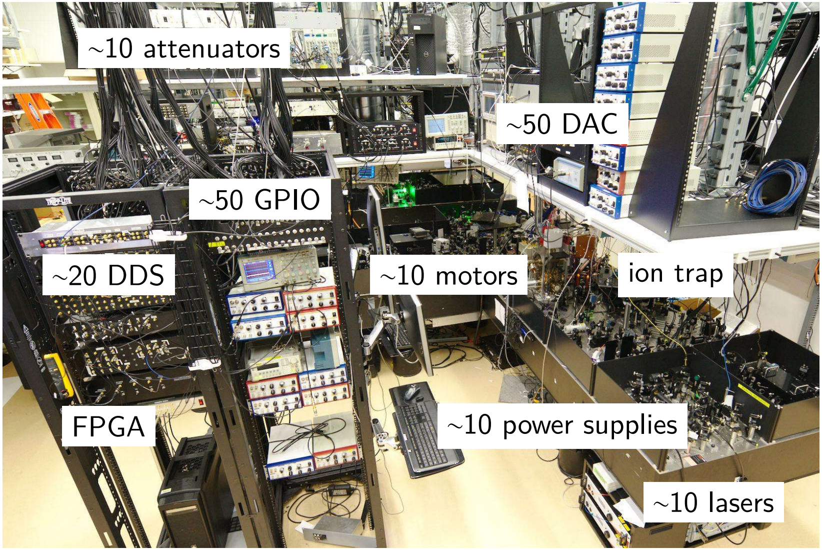
Ion trap

(NIST John Jost)

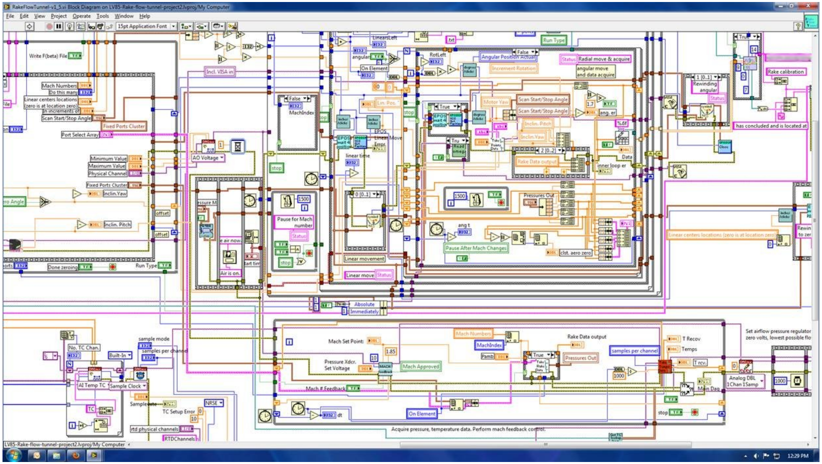


Quantum gate sequences



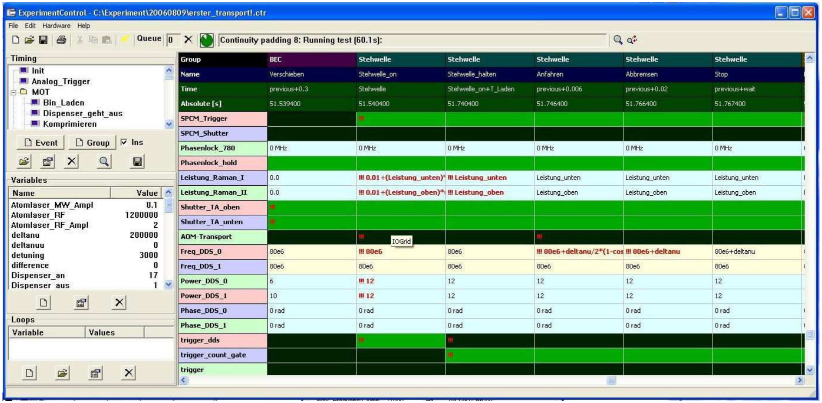

~10 attenuators
~50 DAC
~50 GPIO
~20 DDS
~10 motors
ion trap
FPGA
~10 power supplies
~10 lasers

Physicists are not programmers:



LabVIEW: a “visual programming language” (a.k.a. “high viscosity language”)

Physicists are not programmers:


ExperimentControl - C:\Experiment\20060809\erster_transport!.ctr
File
Edit
Hardware
Help
Queue
0
Event
Group
Ins
IOGrid

Continuity padding 8: Running test (60.1s):

Timing

- Init
- Analog_Trigger
- MOT
  - Bin_Laden
  - Dispenser_geht_aus
  - Komprimieren

Variables

Name	Value
Atomlaser_MW_Ampl	0.1
Atomlaser_RF	1200000
Atomlaser_RF_Ampl	2
deltanu	200000
deltanuu	0
detuning	3000
difference	0
Dispenser_an	17
Dispenser_aus	1

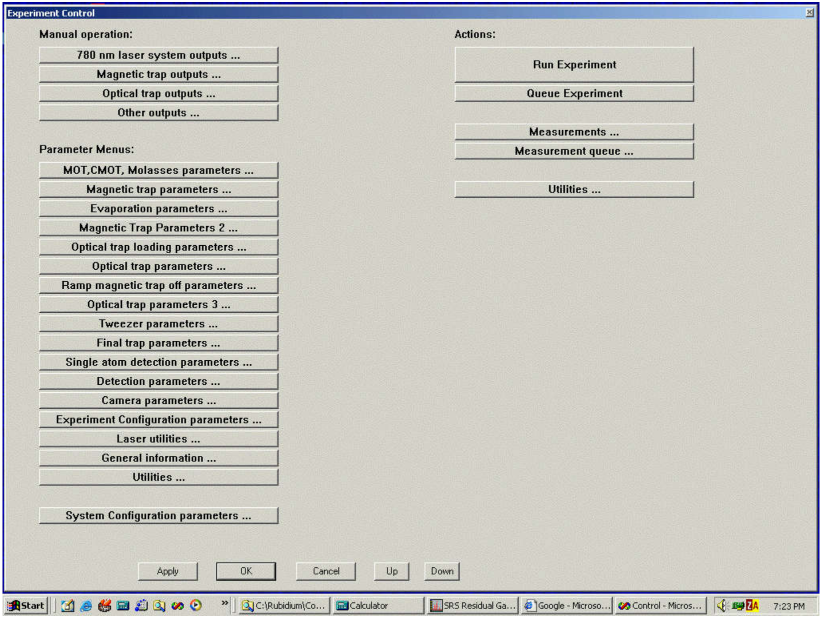
Loops

Variable	Values

Group	BEC	Stehwelle	Stehwelle	Stehwelle	Stehwelle	Stehwelle
Name	Verschieben	Stehwelle_on	Stehwelle_halten	Anfahren	Abbremsen	Stop
Time	previous+0.3	Stehwelle	Stehwelle_on+T_Laden	previous+0.006	previous+0.02	previous+wait
Absolute [s]	51.539400	51.540400	51.740400	51.746400	51.766400	51.767400
SPCM_Trigger	⬛	🟩 !!!	🟩	🟩	🟩	🟩
SPCM_Shutter	⬛	⬛	⬛	⬛	⬛	⬛
Phasenlock_780	0 MHz	0 MHz	0 MHz	0 MHz	0 MHz	0 MHz
Phasenlock_hold	🟩	🟩	🟩	🟩	🟩	🟩
Leistung_Raman_I	0.0	!!! 0.01+(Leistung_unten)*	!!! Leistung_unten	Leistung_unten	Leistung_unten	Leistung_unten
Leistung_Raman_II	0.0	!!! 0.01+(Leistung_oben)*	!!! Leistung_oben	Leistung_oben	Leistung_oben	Leistung_oben
Shutter_TA_oben	🟩 !!!	🟩	🟩	🟩	🟩	🟩
Shutter_TA_unten	🟩 !!!	🟩	🟩	🟩	🟩	🟩
AOM-Transport	⬛	⬛ !!!	⬛	⬛ !!!	⬛	⬛
Freq_DDS_0	80e6	!!! 80e6	80e6	!!! 80e6+deltanu/2*(1-cos	!!! 80e6+deltanu	80e6+deltanu
Freq_DDS_1	80e6	80e6	80e6	80e6	80e6	80e6
Power_DDS_0	6	!!! 12	12	12	12	12
Power_DDS_1	10	!!! 12	12	12	12	12
Phase_DDS_0	0 rad	0 rad	0 rad	0 rad	0 rad	0 rad
Phase_DDS_1	0 rad	0 rad	0 rad	0 rad	0 rad	0 rad
trigger_dds	⬛	🟩 !!!	⬛ !!!	⬛	⬛	⬛
trigger_count_gate	⬛	⬛	🟩 !!!	🟩	🟩	🟩
trigger	⬛	⬛	⬛	⬛	⬛	⬛

Rigid time-versus-channel matrix: inflexible (loops, conditionals?)

Physicists are not programmers:


Experiment Control
Manual operation:
780 nm laser system outputs ...
Magnetic trap outputs ...
Optical trap outputs ...
Other outputs ...
Parameter Menus:
MOT,CMOT, Molasses parameters ...
Magnetic trap parameters ...
Evaporation parameters ...
Magnetic Trap Parameters 2 ...
Optical trap loading parameters ...
Optical trap parameters ...
Ramp magnetic trap off parameters ...
Optical trap parameters 3 ...
Tweezer parameters ...
Final trap parameters ...
Single atom detection parameters ...
Detection parameters ...
Camera parameters ...
Experiment Configuration parameters ...
Laser utilities ...
General information ...
Utilities ...
System Configuration parameters ...
Actions:
Run Experiment
Queue Experiment
Measurements ...
Measurement queue ...
Utilities ...
Apply
OK
Cancel
Up
Down
Start
C:\Rubidium\Co...
Calculator
SRS Residual Ga...
Google - Microso...
Control - Micros...
7:23 PM

Hard-coded components: not generic and opaque implementation

Enter ARTIQ

Advanced Real-Time Infrastructure for Quantum physics

- High performance — nanosecond resolution, hundreds of ns latency
- Expressive — describe algorithms with few lines of code
- Portable — treat hardware, especially FPGA boards, as commodity
- Modular — separate components as much as possible
- Flexible — hard-code as little as possible

Define a simple timing language

```
trigger.sync()                # wait for trigger input
start = now()                 # capture trigger time
for i in range(3):
    delay(5*us)
    dds.pulse(900*MHz, 7*us)  # first pulse 5 µs after trigger
at(start + 1*ms)              # re-reference time-line
dds.pulse(200*MHz, 11*us)     # exactly 1 ms after trigger
```

- Written in a subset of Python
- Executed on a CPU embedded on a FPGA (the *core device*)
- `now()`, `at()`, `delay()` describe time-line of an experiment
- Exact time is kept in an internal variable
- That variable only loosely tracks the execution time of CPU instructions
- The value of that variable is exchanged with the RTIO fabric that does precise timing

Convenient syntax additions

```
with sequential:
    with parallel:
        a.pulse(100*MHz, 10*us)
        b.pulse(200*MHz, 20*us)
    with parallel:
        c.pulse(300*MHz, 30*us)
        d.pulse(400*MHz, 20*us)
```

- Experiments are inherently parallel: simultaneous laser pulses, parallel cooling of ions in different trap zones
- `parallel` and `sequential` contexts with arbitrary nesting
- `a` and `b` pulses both start at the same time
- `c` and `d` pulses both start when `a` and `b` are both done (after 20 μs)
- Implemented by inlining, loop-unrolling, and interleaving

Physical quantities, hardware granularity

```
n = 1000
dt = 1.2345*ns
f = 345*MHz

dds.on(f, phase=0)                # must round to integer tuning word
for i in range(n):
    delay(dt)                     # must round to native cycles

dt_raw = time_to_cycles(dt)       # integer number of cycles
f_raw = dds.frequency_to_ftw(f) # integer frequency tuning word

# determine correct phase despite accumulation of rounding errors
phi = n*cycles_to_time(dt_raw)*dds.ftw_to_frequency(f_raw)
```

- Need well defined conversion and rounding of physical quantities (time, frequency, phase, etc.) to hardware granularity and back
- Complicated because of calibration, offsets, cable delays, non-linearities
- No generic way to do it automatically and correctly
- $\rightarrow$ need to do it explicitly where it matters

Invite organizing experiment components and code reuse

```
class Experiment:
    def build(self):
        self.ion1 = Ion(...)
        self.ion2 = Ion(...)
        self.transporter = Transporter(...)

    @kernel
    def run(self):
        with parallel:
          self.ion1.cool(duration=10*us)
          self.ion2.cool(frequency=...)
        self.transporter.move(speed=...)
        delay(100*ms)
        self.ion1.detect(duration=...)
```

RPC to handle distributed non-RT hardware

```
class Experiment:
    def prepare(self):                  # runs on the host
        self.motor.move_to(20*mm)       # slow RS232 motor controller

    @kernel
    def run(self):                      # runs on the RT core device
        self.prepare()                  # converted into an RPC
```

- When a kernel function calls a non-kernel function, it generates a RPC
- The callee is executed on the host
- Mechanism to report results and control slow devices
- The kernel must have a loose real-time constraint (a long `delay`) or means of re-synchronization to cover communication, host, and device delays

Kernel deployment to the core device

- RPC and exception mappings are generated
- Constants and small kernels are inlined
- Small loops are unrolled
- Statements in parallel blocks are interleaved
- Time is converted to RTIO clock cycles
- The Python AST is converted to LLVM IR
- The LLVM IR is compiled to OpenRISC machine code
- The OpenRISC binary is sent to the core device
- The runtime in the core device links and runs the kernel
- The kernel calls the runtime for communication (RPC) and interfacing with core device peripherals (RTIO, DDS)

`https://github.com/m-labs/artiq`

- Fully open-source, BSD licensed
- Ported and running on two different FPGA boards
- Design applicable beyond ion trapping (superconducting qubits, neutral atoms...)
- Fastest open-source DDR3 SODIMM controller as a sub-project: 64 Gbps
- Interfacing with lab hardware
- Hardware-in-the-loop unittests
- Self-contained simulator
- Currently ~1 μs latency and ~1 MHz event rate
- DMA should improve that dramatically