Quantum Computing Simulation with FPGA

Mirko Mariotti [1,2] Giulio Bianchini [1] Loriano Storchi [3,2] Daniele Spiga [2]
Diego Ciangottini [2] Giuseppe Prudente [2]

[1]Dipartimento di Fisica e Geologia, Universitá degli Studi di Perugia

[2]INFN sezione di Perugia

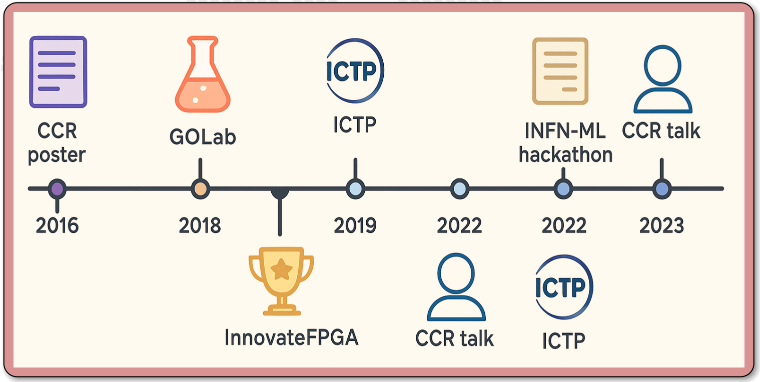
[3]Dipartimento di Farmacia, Universitá degli Studi G. D'Annunzio

Outline

Background

- FPGA
- Firmware
- Quantum Computing

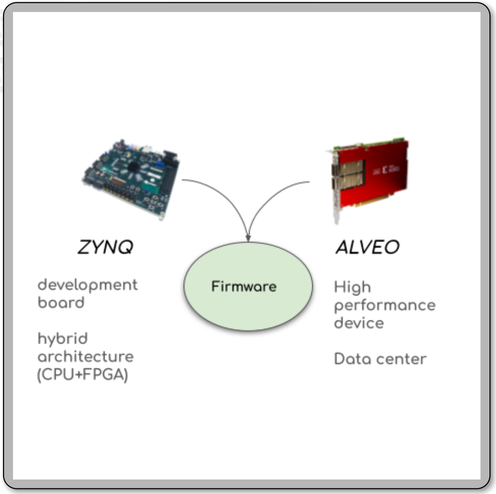
Timeline



Spoke 2 use cases - Ultra-fast algorithms running on FPGA
Development of a Customizable Framework for Multi-FPGA Accelerator Generation via architectures

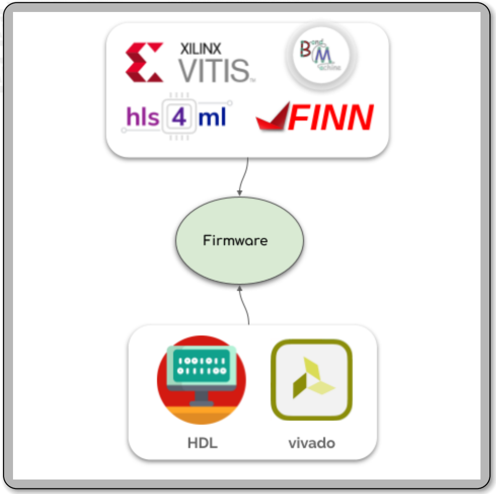
FPGA R&D

- We mainly focus on using FPGA as a hardware accelerator for scientific computing.
- We are interested in both low-level programming and high-level synthesis.
- We are also proposing a new architecture called BondMachine (BM) that is designed to be used as a hardware accelerator.



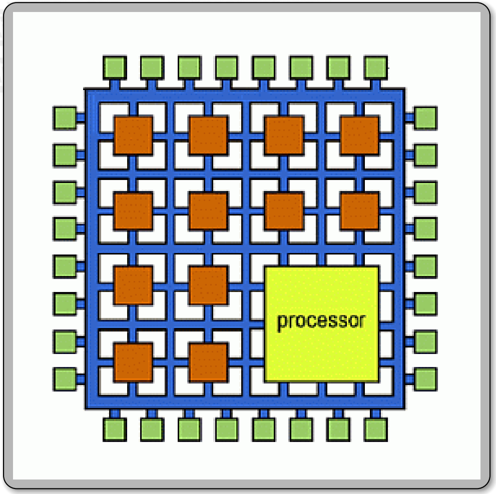
FPGA R&D

- We mainly focus on using FPGA as a hardware accelerator for scientific computing.
- We are interested in both low-level programming and high-level synthesis.
- We are also proposing a new architecture called BondMachine (BM) that is designed to be used as a hardware accelerator.



FPGA R&D

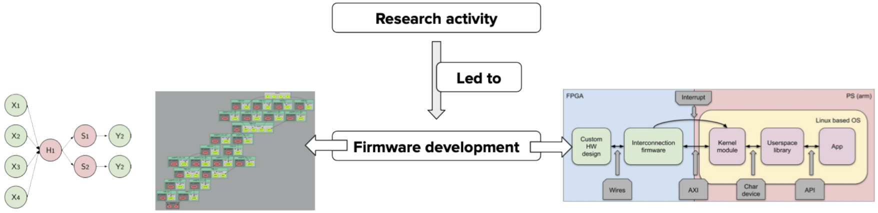
- We mainly focus on using FPGA as a hardware accelerator for scientific computing.
- We are interested in both low-level programming and high-level synthesis.
- We are also proposing a new architecture called BondMachine (BM) that is designed to be used as a hardware accelerator.



The BondMachine Framework

The BondMachine is an open source software ecosystem for the dynamical generation of computer architectures that can be synthesized on FPGAs.

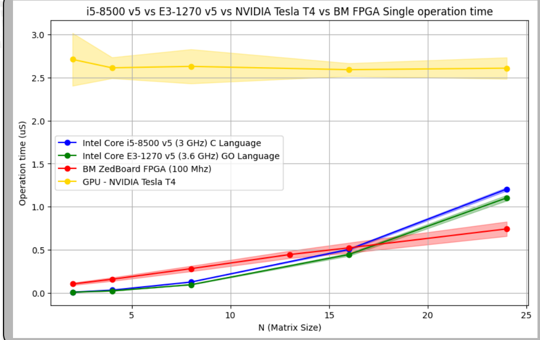
- High level programming language (Golang) for both the hardware and software
- Functional style programming
- Architecture generating compiler
- Computational graph and Machine Learning Models



The BondMachine, a moldable computer architecture - doi.org/10.1016/j.parco.2021.102873 - https://www.bondmachine.it

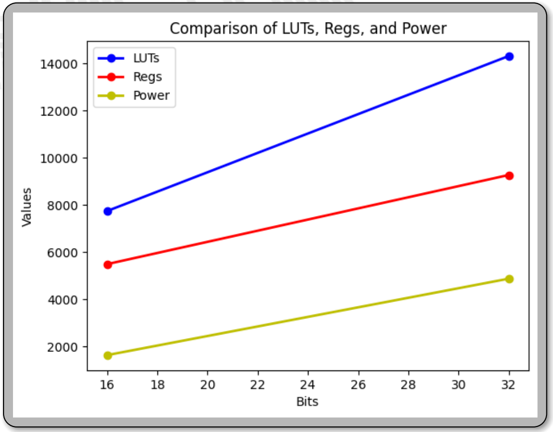
R&D: Analysis

- Latency and throughput analysis
- occupancy analysis
- Energy efficiency analysis
- Comparison with other architectures
- Numerical precision analysis
- Data type and/ or instruction set Analysis



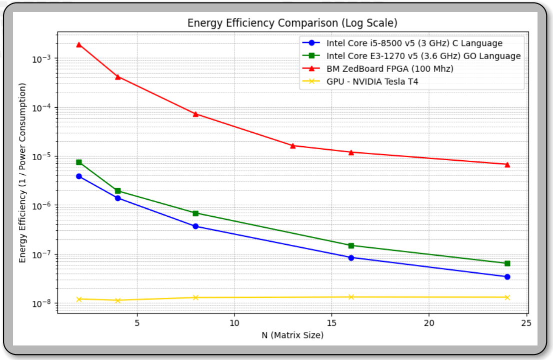
R&D: Analysis

- Latency and throughput analysis
- occupancy analysis
- Energy efficiency analysis
- Comparison with other architectures
- Numerical precision analysis
- Data type and/ or instruction set Analysis



R&D: Analysis

- Latency and throughput analysis
- occupancy analysis
- Energy efficiency analysis
- Comparison with other architectures
- Numerical precision analysis
- Data type and/ or instruction set Analysis



R&D: Analysis

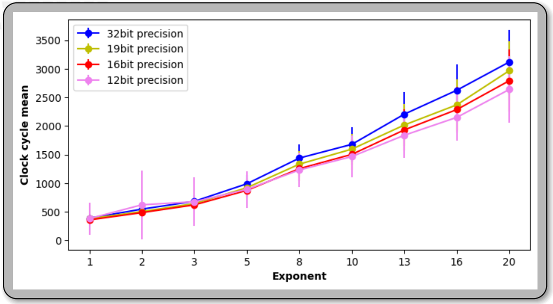
- Latency and throughput analysis
- occupancy analysis
- Energy efficiency analysis
- Comparison with other architectures
- Numerical precision analysis
- Data type and/ or instruction set Analysis

Solution	LUTs	REGs	Time / Inf (µs)
HLS4ML	10.31%	6.89%	~ 0.71
BondMachine	15.73%	7.94%	~ 1.4

CPU	Time / Inf (s)	En. / Inf (J)
ARM Cortex A9	10E-02	10E-06
Intel i7-1260P	10E-06	10E-04
NVIDIA Tesla T4	10E-04	10E-03
ZedBoard BM	10E-06	10E-08

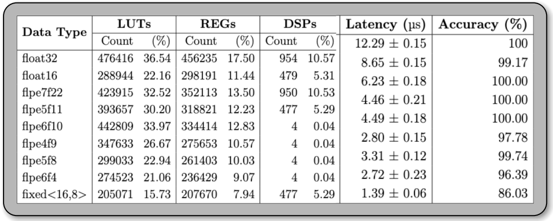
R&D: Analysis

- Latency and throughput analysis
- occupancy analysis
- Energy efficiency analysis
- Comparison with other architectures
- Numerical precision analysis
- Data type and/ or instruction set Analysis



R&D: Analysis

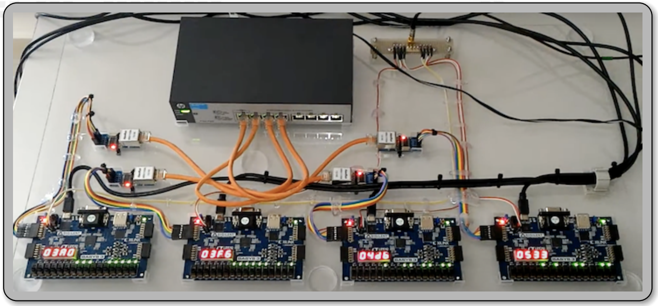
- Latency and throughput analysis
- occupancy analysis
- Energy efficiency analysis
- Comparison with other architectures
- Numerical precision analysis
- Data type and/ or instruction set Analysis



Data Type	LUTs		REGs		DSPs		Latency (µs)	Accuracy (%)
	Count	(%)	Count	(%)	Count	(%)		
float32	476416	36.54	456235	17.50	954	10.57	12.29 ± 0.15	100
float16	288944	22.16	298191	11.44	479	5.31	8.65 ± 0.15	99.17
flpe7f22	423915	32.52	352113	13.50	950	10.53	6.23 ± 0.18	100.00
flpe5f11	393657	30.20	318821	12.23	477	5.29	4.46 ± 0.21	100.00
flpe6f10	442809	33.97	334414	12.83	4	0.04	4.49 ± 0.18	100.00
flpe4f9	347633	26.67	275653	10.57	4	0.04	2.80 ± 0.15	97.78
flpe5f8	299033	22.94	261403	10.03	4	0.04	3.31 ± 0.12	99.74
flpe6f4	274523	21.06	236429	9.07	4	0.04	2.72 ± 0.23	96.39
fixed<16,8>	205071	15.73	207670	7.94	477	5.29	1.39 ± 0.06	86.03

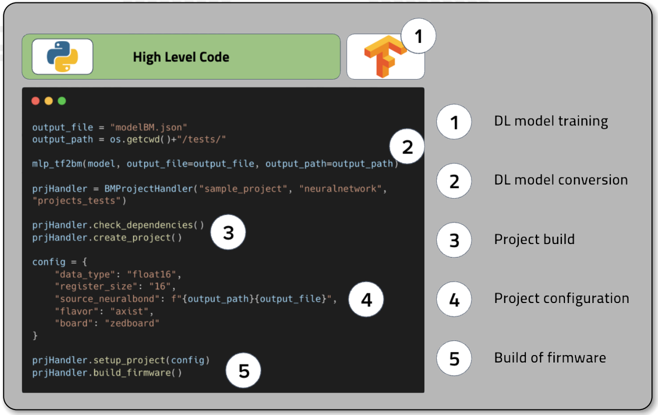
R&D: Problems

- SoC, edge and low power computing
- Machine Learning (inference)



R&D: Problems

- SoC, edge and low power computing
- Machine Learning (inference)



```
output_file = "modelBM.json"
output_path = os.getcwd()+"/tests/"

mlp_tf2bm(model, output_file=output_file, output_path=output_path)

prjHandler = BMProjectHandler("sample_project", "neuralnetwork",
"projects_tests")

prjHandler.check_dependencies()
prjHandler.create_project()

config = {
    "data_type": "float16",
    "register_size": "16",
    "source_neuralbond": f"{output_path}{output_file}",
    "flavor": "axist",
    "board": "zedboard"
}

prjHandler.setup_project(config)
prjHandler.build_firmware()
```

Quantum Computing Simulation with FPGA

We started experimenting with quantum computing. Our main interested is using FPGA to simulate quantum computers.

The goal is to experiment with classical/quantum hybrid computing backed by the CPU/FPGA hardware.

The work plan goes on 4 main directions:

- Learning and experimenting with reference quantum tools and establishing a testing framework to validate and compare the results of different quantum simulators. Activity 1
- BondMachine based quantum simulator. Activity 2
- HLS based quantum simulator. Activity 3
- Symbolic Quantum Operator FPGA based simulator. Activity 4

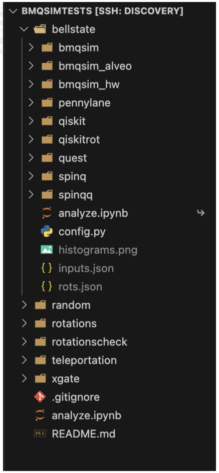
Validation

Activity 1

To test the correctness of the quantum simulator we are developing, we need to compare the results of the simulation with the results of a well-known quantum simulators.
We set up a validation framework in the bmqsimtests repository, at the url: `https://github.com/BondMachineHQ/bmqsimtests`

The repository is organized in two levels of directories. The first level is the quantum circuit to simulate, the second level is the specific simulatur to use. A Jupiter notebook is provided to run the tests and compare the results.

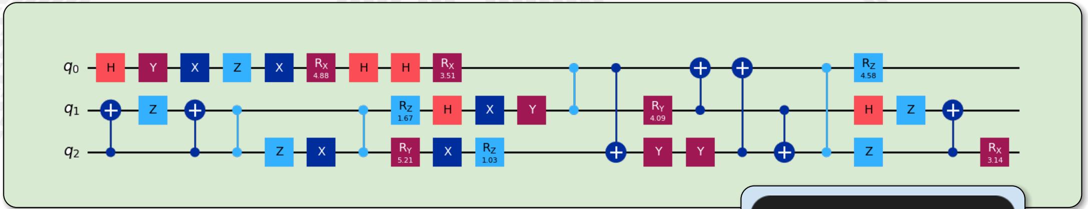
the readme.md file contains the instructions to run the tests and describe the two layer directory structure of the tests.

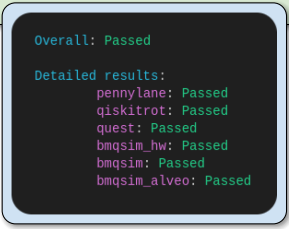


```
BMQSIMTESTS [SSH: DISCOVERY]
  bellstate
    bmqsim
    bmqsim_alveo
    bmqsim_hw
    pennylane
    qiskit
    qiskitrot
    quest
    spinq
    spinqq
    analyze.ipynb
    config.py
    histograms.png
    inputs.json
    rots.json
  random
  rotations
  rotationscheck
  teleportation
  xgate
  .gitignore
  analyze.ipynb
  README.md
```

Validation

Activity 1





```
Overall: Passed

Detailed results:
        pennylane: Passed
        qiskitrot: Passed
        quest: Passed
        bmqsim_hw: Passed
        bmqsim: Passed
        bmqsim_alveo: Passed
```

the validation is done by comparing the results of the simulation with the results of the same quantum circuit simulated by a well-known quantum simulator. randomizing both the quantum circuit and the input state.

BMQsim

Activity 2

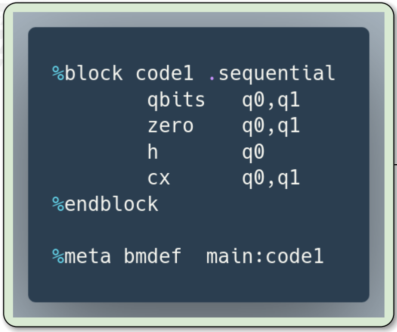
With all the capabilities of the BondMachine in terms of parallelism and speed, of customizability of the instruction set and the numerical precision, it is a natural question to ask whether the BondMachine could be used to simulate quantum computers.

A quantum computer simulator called bmqsim has been developed and is available within the BondMachine project.

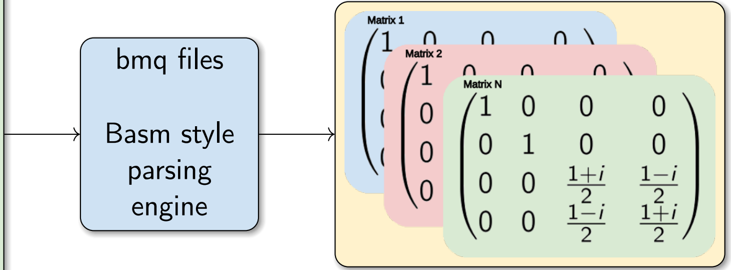
Quantum Circuit

Activity 2

The first ingredient for bmqsim is a quantum circuit. The quantum circuit is a sequence of quantum gates represented by a sequence of matrices. the "program" is a .bmq file that contains code similar to the Qasm code.



```
%block code1 .sequential
       qbits   q0,q1
       zero    q0,q1
       h       q0
       cx      q0,q1
%endblock

%meta bmdef  main:code1
```



$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} \\ 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} \end{pmatrix}$$

Independently of the backend, bmqsim translates the .bmq file into N matrices.

Backends

Activity 2

bmqsim may use different backends to operate. different backends create different hardware to simulate the same quantum circuit. Moreover, each backend may have different flavors to further fine-tune the HDL.

Software Simulation

Hardcoded matrices sequence

Hardcoded matrices sequence (HLS)

Loadable matrices sequence

Partially implemented

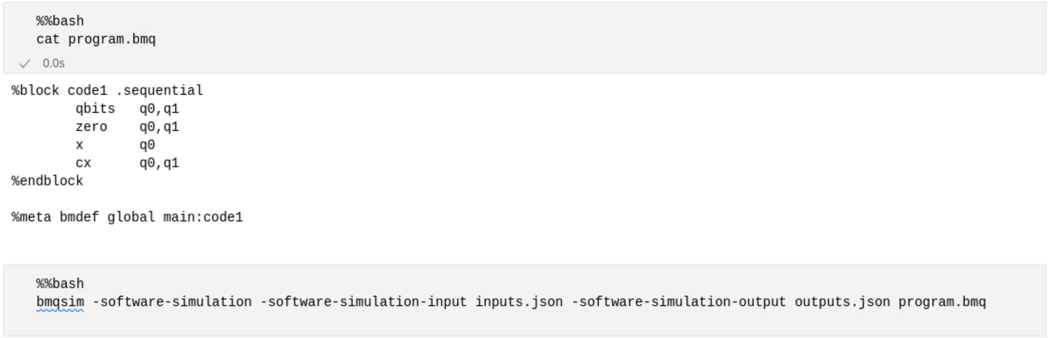
Full hardware deploy

Partially implemented

A command line option allows to choose the backend to use.

Backend: Software Simulation

In here, the quantum gates are simulated by the CPU. This is the slowest backend, but it useful for circuit design, debugging and testing. An example:



```
%%bash
cat program.bmq
```

✓ 0.0s

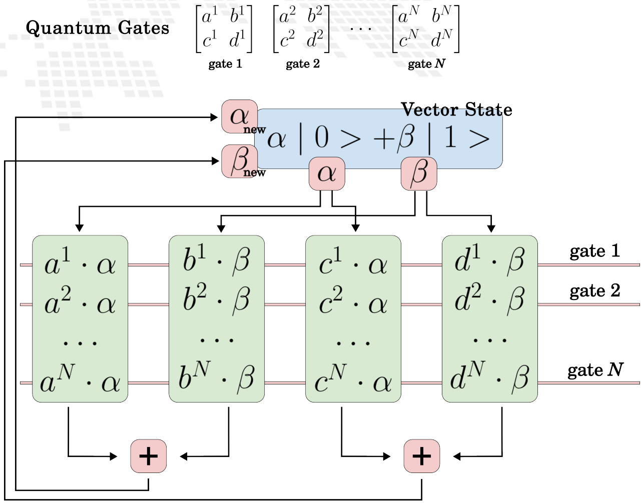
```
%block code1 .sequential
        qbits   q0,q1
        zero    q0,q1
        x       q0
        cx      q0,q1
%endblock

%meta bmdef global main:code1
```

```
%%bash
bmqsim -software-simulation -software-simulation-input inputs.json -software-simulation-output outputs.json program.bmq
```

Backend: Hardcoded matrices sequence

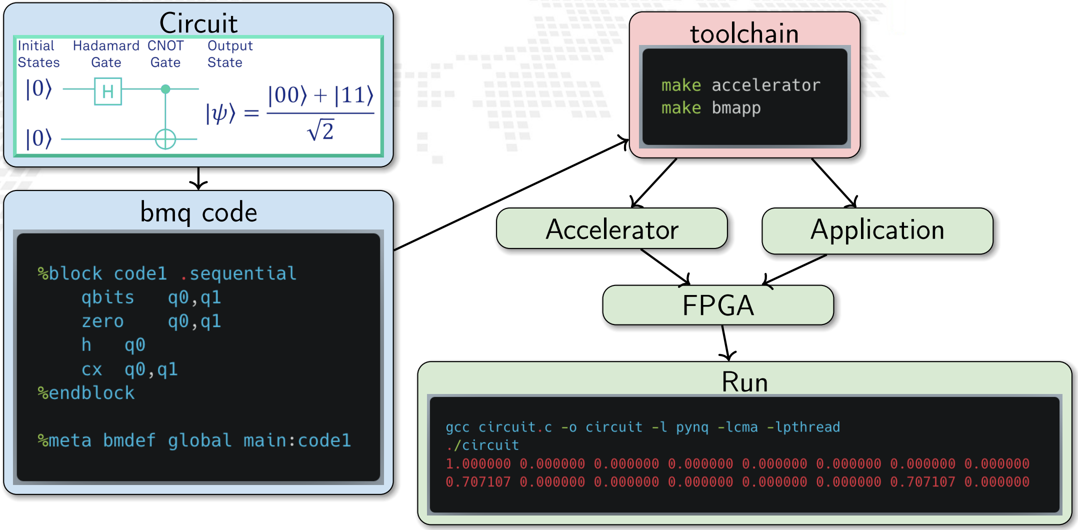
This backend creates a hardware that for each state of the quantum register, it applies the sequence of matrices.

For each matrix operation a dedicated processor is used. Within the processor, the matrix elements of all the gates are hardcoded.



Quantum Gates
$\begin{bmatrix} a^1 & b^1 \\ c^1 & d^1 \end{bmatrix}$ gate 1
$\begin{bmatrix} a^2 & b^2 \\ c^2 & d^2 \end{bmatrix}$ gate 2
$\cdots$
$\begin{bmatrix} a^N & b^N \\ c^N & d^N \end{bmatrix}$ gate N
Vector State
α_{new}
β_{new}
$\alpha \mid 0 > + \beta \mid 1 >$
α
β
$a^1 \cdot \alpha$, $a^2 \cdot \alpha$, $\ldots$, $a^N \cdot \alpha$
$b^1 \cdot \beta$, $b^2 \cdot \beta$, $\ldots$, $b^N \cdot \beta$
$c^1 \cdot \alpha$, $c^2 \cdot \alpha$, $\ldots$, $c^N \cdot \alpha$
$d^1 \cdot \beta$, $d^2 \cdot \beta$, $\ldots$, $d^N \cdot \beta$
gate 1
gate 2
gate N
+
+

Bell state example



$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

```
%block code1 .sequential
    qbits   q0,q1
    zero    q0,q1
    h   q0
    cx  q0,q1
%endblock

%meta bmdef global main:code1
```

```
make accelerator
make bmapp
```

```
gcc circuit.c -o circuit -l pynq -lcma -lpthread
./circuit
1.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000 0.000000
0.707107 0.000000 0.000000 0.000000 0.000000 0.000000 0.707107 0.000000
```

Backend: Hardcoded matrices sequence

Pros and Cons

Pros:

- The matrices elements of the gates are already inside each processor. There no movement of big matrices.
- Fast

Cons:

- The circuit is fixed. to use a different circuit hardware has to be re-synthesized.
- Matrices are fully expanded. This may lead to a big hardware.
- Sparse matrices uses hardware anyway.

Backend: Loadable matrices sequence

Similar to the previous backend, but the matrices are loaded from the final application command line. This allows to change the matrices without recompiling the hardware.

To do so a small boot loader is needed on every processor. And a protocol to load the matrices elements from the final application.

Pros:

- The matrices elements of the gates are already inside each processor. There no movement of big matrices.
- Fast
- The circuit is fixed, but a new circuit can be injected by the final application.

Cons:

- Matrices are fully expanded. This may lead to a big hardware.
- Sparse matrices uses hardware anyway.

Backend: Full hardware deploy

In this backend, the quantum circuit is synthesized in full hardware. Instead of having a state that is updated by each gate, only the relevant parts of the state are updated. Keeping track of the entalgment of the qubits and the sparce nature of the matrices.

Pros:

- Fast
- Less resources used with respect to the previous backends

Cons:

- The circuit is fixed and cannot be changed.

Backend: HLS Hardcoded matrices sequence

Activity 3

This backend is similar to the BM hardcoded matrices sequence backend, but it uses the HLS toolchain to create the hardware instead of the BM toolchain.

The matrices are hardcoded in the HLS (C++) code. The HLS pragmas are used to create the hardware. The HLS code is then compiled with the Vitis HLS toolchain.

Pros:

- There is no processor abstraction, the hardware is lighter that the BM hardcoded matrices sequence backend.

Cons:

- The circuit is fixed. to use a different circuit hardware has to be re-synthesized.
- Without the processor abstraction, the hardware is less flexible. Classical/quantum hybrid computing is not possible.

Applications

Alongside the FPGA hardware, bmqsim can create the end application that can be used to simulate quantum circuits.

Three types of applications are available:

- Jupiter Notebook using the PYNQ framework
- Standalone C application using pynq-api
- C++/OpenCL application

The application are tailored to the specific board,circuit and backend used.

Symbolic Quantum Operator approach

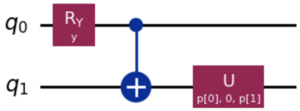
Activity 4

An alternative approach is to use symbolic quantum operators.

qiskit-symb

Quantum Circuit

```
y = Parameter('y')
p = ParameterVector('p', length=2)

pqc = QuantumCircuit(2)
pqc.ry(y, 0)
pqc.cx(0, 1)
pqc.u(p[0], 0, p[1], 1)
```

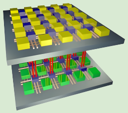


Symbolic expression (Sympy)

$$\left[\ \cos\left(\frac{p[0]}{2}\right)\cos\left(\frac{y}{2}\right) \quad -e^{1.0ip[1]}\sin\left(\frac{p[0]}{2}\right)\sin\left(\frac{y}{2}\right) \quad \sin\left(\frac{p[0]}{2}\right)\cos\left(\frac{y}{2}\right) \quad e^{1.0ip[1]}\sin\left(\frac{y}{2}\right)\cos\left(\frac{p[0]}{2}\right)\ \right]$$

flexpy

BM/HLS



Example from qiskit-symb - https://github.com/SimoneGasperini/qiskit-symb.git

Flexpy (FPGA Logic from EXpressions)

Activity 4

We created a project called flexpy to convert symbolic mathematical expressions into FPGA logic.

The symbolic expressions are created using the sympy library. flexpy can parse the expression and create either BASM (the BondMachine assembly language) or C/C++ code with HLS pragmas.

The generated code can be used to create an hardware accelerator for the given expression using the BondMachine or HLS toolchain.

The approach is much more general, it can be used not only for quantum computing, but also for any other application that can be expressed as a symbolic expression.

flexpy repository: https://github.com/BondMachineHQ/flexpy.git

Flexpy example

Activity 4

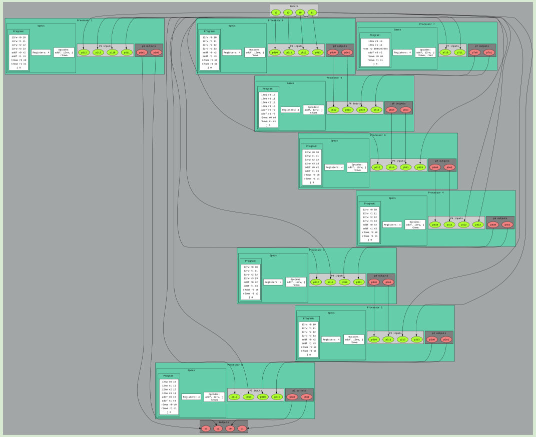
Symbolic expression

```
x = sp.Symbol('x', real=False)
y = sp.Symbol('y', real=False)
with sp.evaluate(False):
        spExpr = sp.Array([( x + 5 ) +
                 ( y + x ) + x + x +
                 x + x + x, x + y])
```

toolchain

```
make bondmachine
make ...
make show
```

Accelerator



Flexpy test units

Activity 4

Some mathematical operations (for example the cosine) are flequently used in quantum computing.

These operations can either be implemented in hardware (HDL) or in software (in assembly in the BM case). Whatever the case, implementing these operations is a complex task.

flexpy can be used to generate test units to check the correctness of the implementation of these operations.

Moreover, the test units can be used to check also the errors introduced by using reduced precision data types and operations.

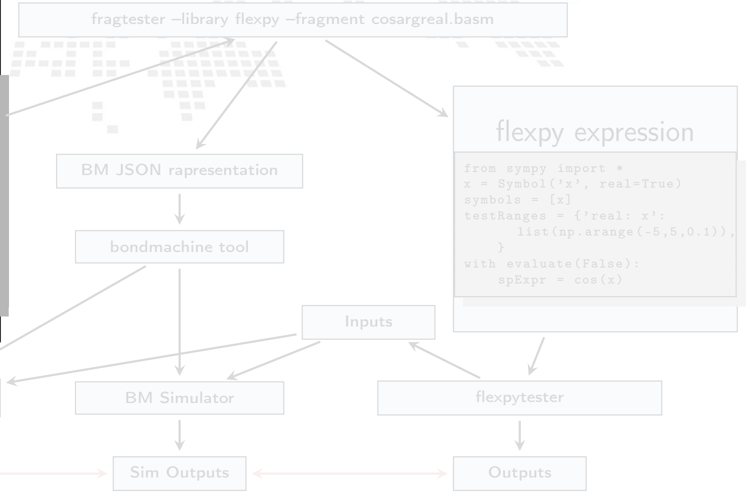
flexpytester repository: https://github.com/BondMachineHQ/flexpytester.git
Symbolic tests repository: https://github.com/BondMachineHQ/bmsymtests.git

Flexpy test units

Activity 4

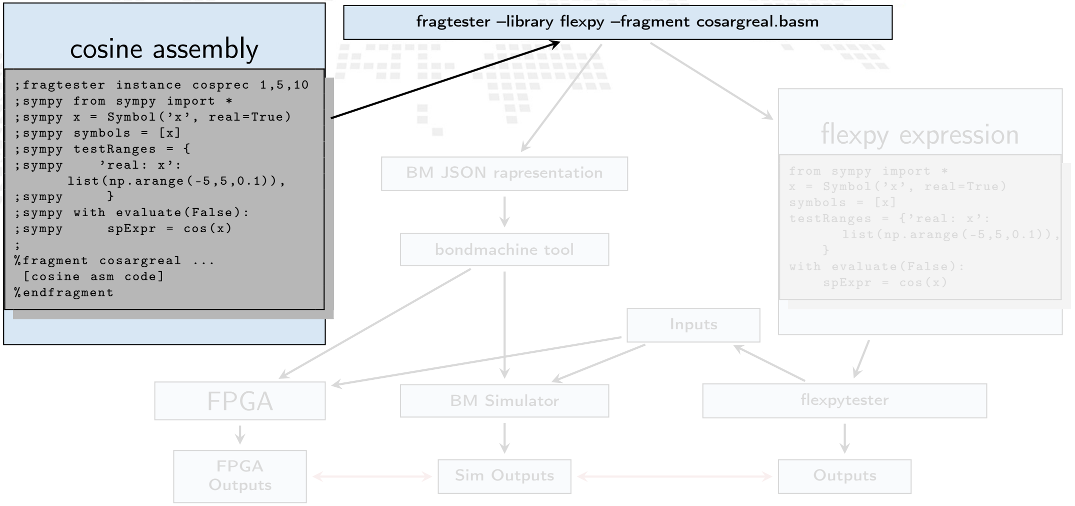
cosine assembly

```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy    'real: x':
      list(np.arange(-5,5,0.1)),
;sympy     }
;sympy with evaluate(False):
;sympy     spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```



Flexpy test units

Activity 4

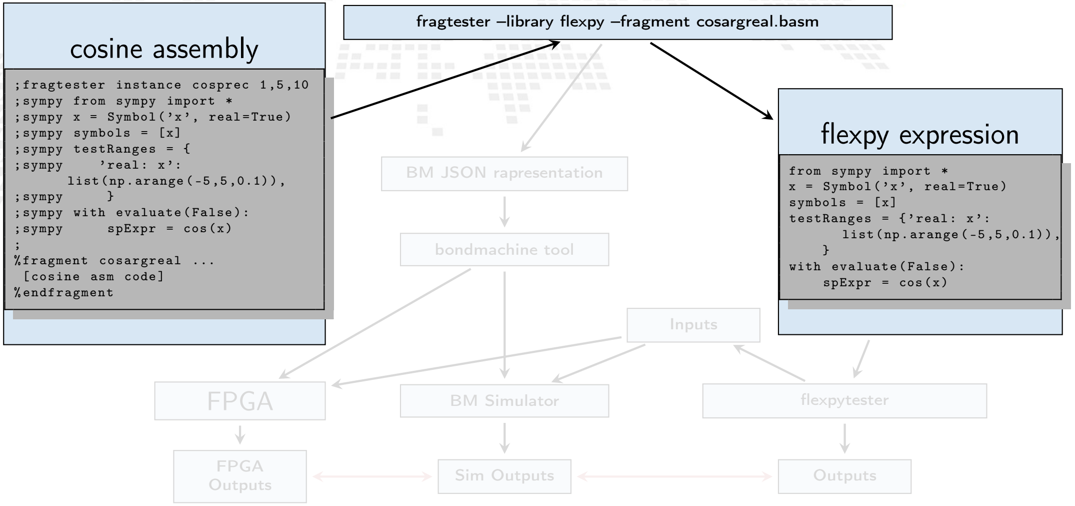


```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy    'real: x':
      list(np.arange(-5,5,0.1)),
;sympy     }
;sympy with evaluate(False):
;sympy     spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```

```
from sympy import *
x = Symbol('x', real=True)
symbols = [x]
testRanges = {'real: x':
      list(np.arange(-5,5,0.1)),
    }
with evaluate(False):
    spExpr = cos(x)
```

Flexpy test units

Activity 4

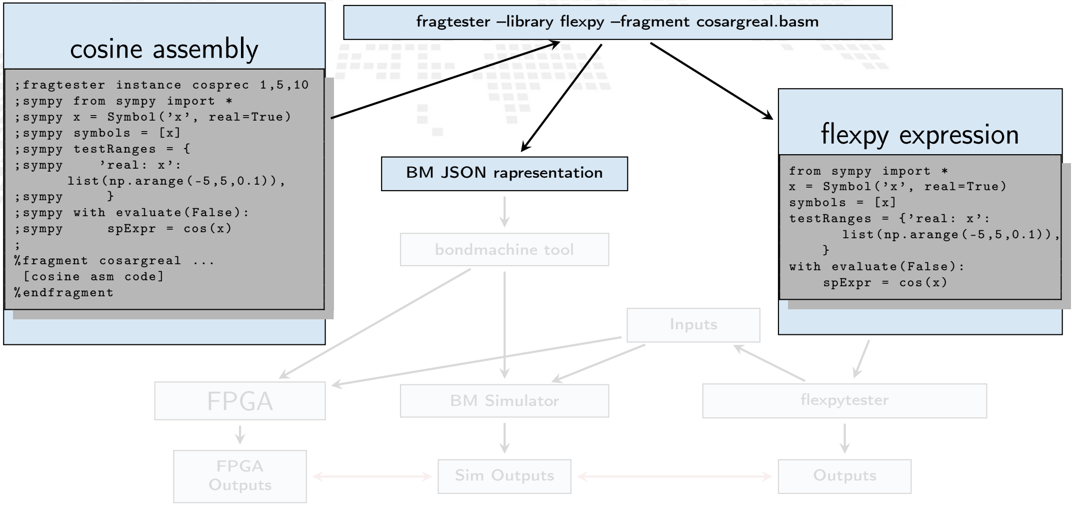


```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy     'real: x':
      list(np.arange(-5,5,0.1)),
;sympy      }
;sympy with evaluate(False):
;sympy      spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```

```
from sympy import *
x = Symbol('x', real=True)
symbols = [x]
testRanges = {'real: x':
      list(np.arange(-5,5,0.1)),
    }
with evaluate(False):
    spExpr = cos(x)
```

Flexpy test units

Activity 4



cosine assembly

```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy     'real: x':
      list(np.arange(-5,5,0.1)),
;sympy      }
;sympy with evaluate(False):
;sympy      spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```

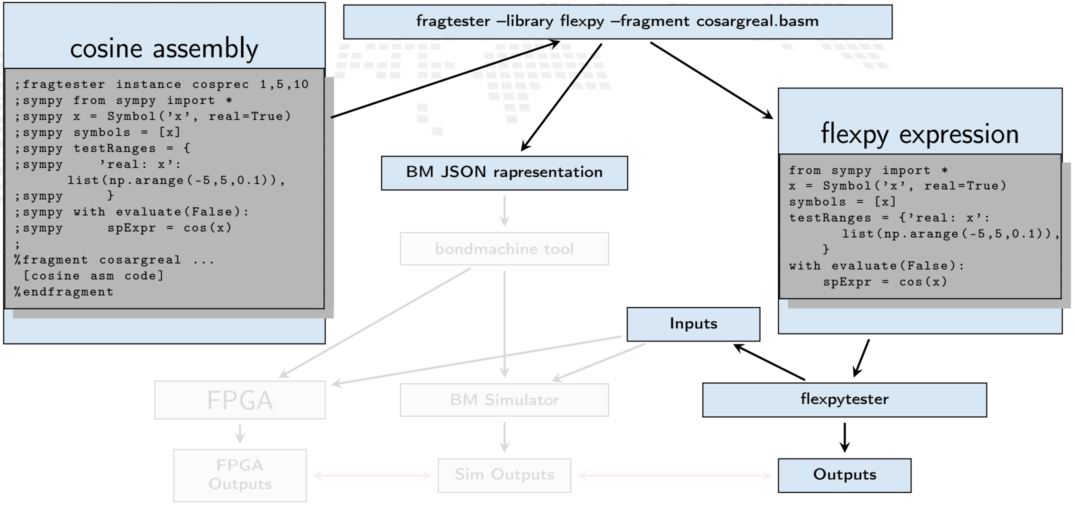
```
fragtester –library flexpy –fragment cosargreal.basm
```

flexpy expression

```
from sympy import *
x = Symbol('x', real=True)
symbols = [x]
testRanges = {'real: x':
      list(np.arange(-5,5,0.1)),
    }
with evaluate(False):
    spExpr = cos(x)
```

Flexpy test units

Activity 4



cosine assembly

```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy     'real: x':
      list(np.arange(-5,5,0.1)),
;sympy     }
;sympy with evaluate(False):
;sympy     spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```

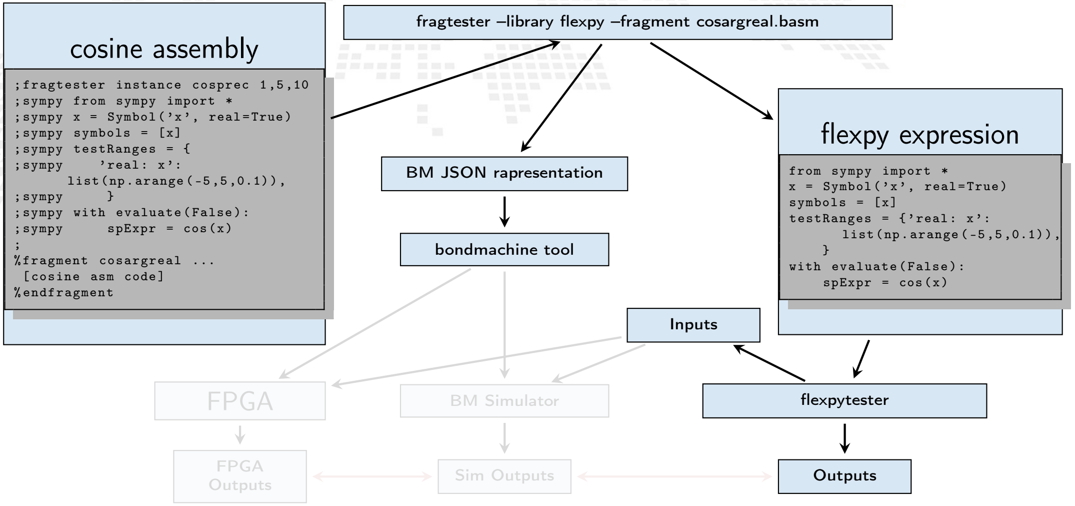
```
fragtester –library flexpy –fragment cosargreal.basm
```

flexpy expression

```
from sympy import *
x = Symbol('x', real=True)
symbols = [x]
testRanges = {'real: x':
      list(np.arange(-5,5,0.1)),
    }
with evaluate(False):
    spExpr = cos(x)
```

Flexpy test units

Activity 4

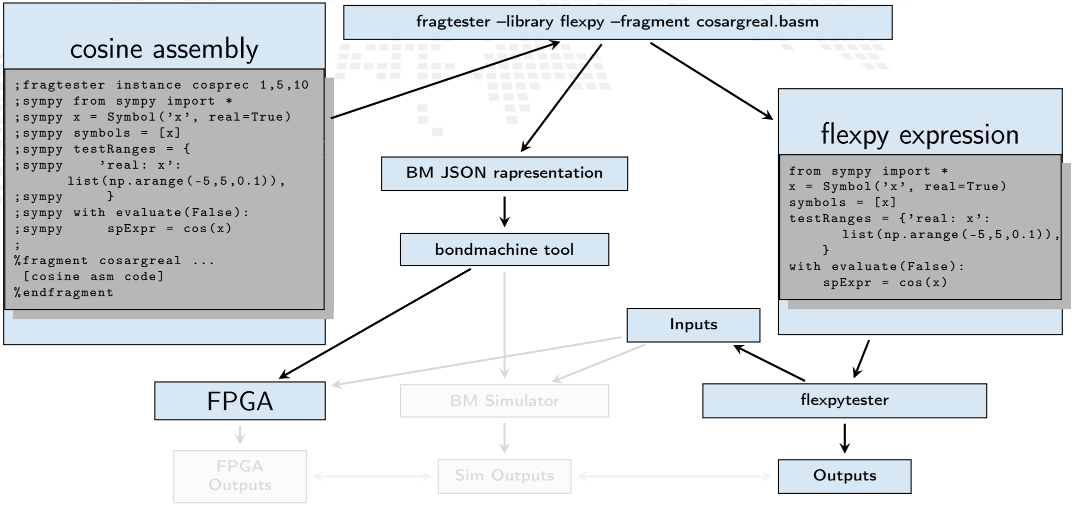


```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy    'real: x':
      list(np.arange(-5,5,0.1)),
;sympy     }
;sympy with evaluate(False):
;sympy     spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```

```
from sympy import *
x = Symbol('x', real=True)
symbols = [x]
testRanges = {'real: x':
      list(np.arange(-5,5,0.1)),
    }
with evaluate(False):
    spExpr = cos(x)
```

Flexpy test units

Activity 4

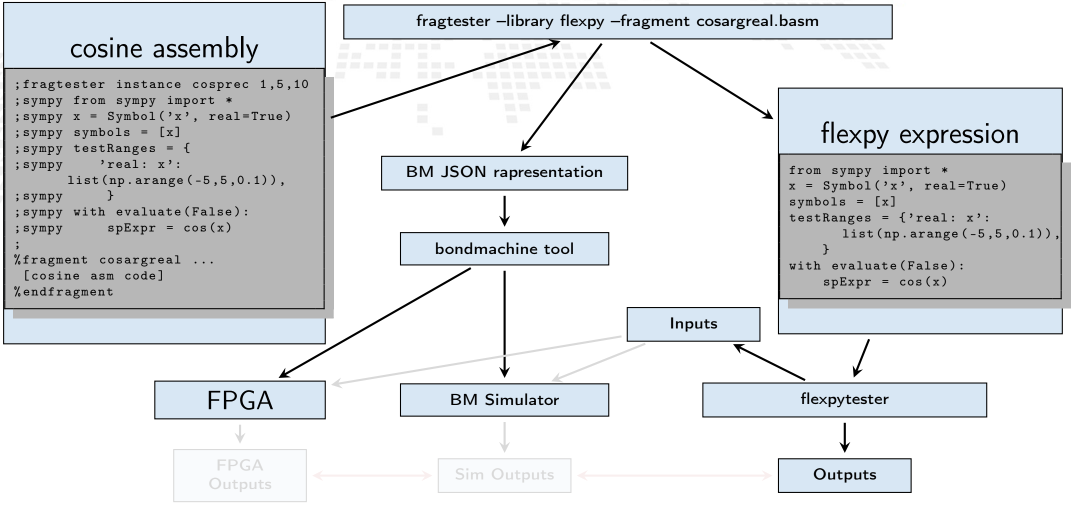


```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy    'real: x':
      list(np.arange(-5,5,0.1)),
;sympy     }
;sympy with evaluate(False):
;sympy     spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```

```
from sympy import *
x = Symbol('x', real=True)
symbols = [x]
testRanges = {'real: x':
      list(np.arange(-5,5,0.1)),
    }
with evaluate(False):
    spExpr = cos(x)
```

Flexpy test units

Activity 4



cosine assembly

```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy    'real: x':
      list(np.arange(-5,5,0.1)),
;sympy     }
;sympy with evaluate(False):
;sympy     spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```

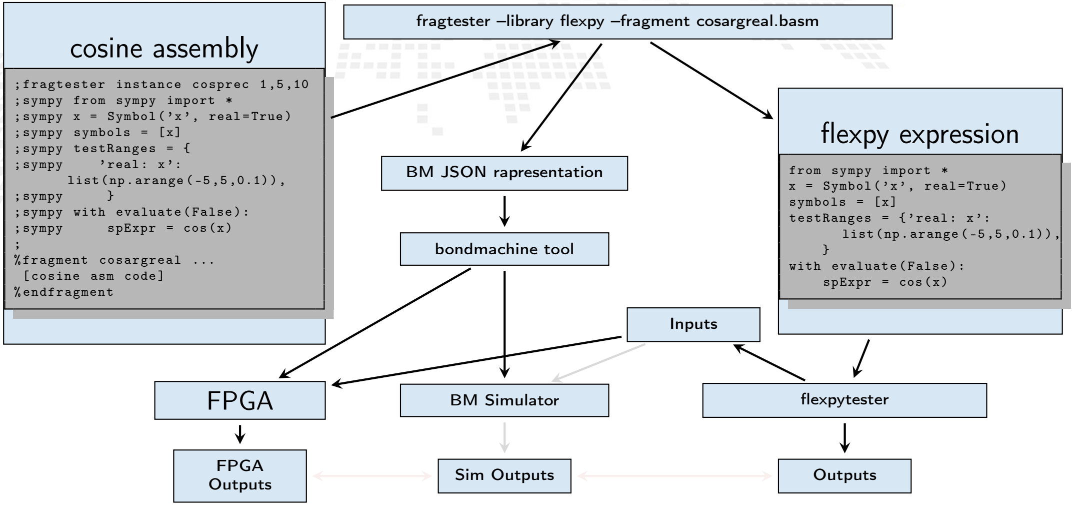
```
fragtester –library flexpy –fragment cosargreal.basm
```

flexpy expression

```
from sympy import *
x = Symbol('x', real=True)
symbols = [x]
testRanges = {'real: x':
      list(np.arange(-5,5,0.1)),
    }
with evaluate(False):
    spExpr = cos(x)
```

Flexpy test units

Activity 4

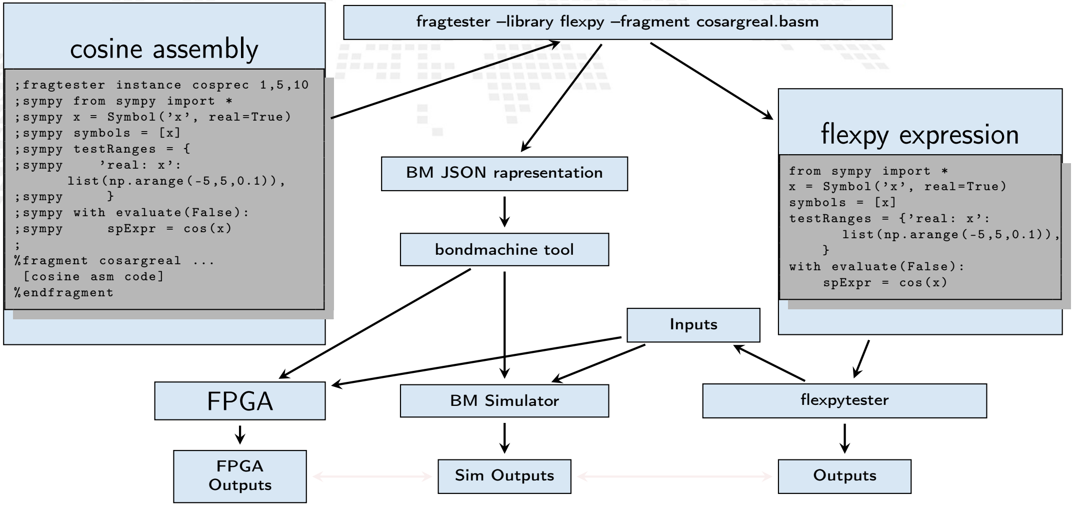


```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy    'real: x':
       list(np.arange(-5,5,0.1)),
;sympy     }
;sympy with evaluate(False):
;sympy     spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```

```
from sympy import *
x = Symbol('x', real=True)
symbols = [x]
testRanges = {'real: x':
      list(np.arange(-5,5,0.1)),
    }
with evaluate(False):
    spExpr = cos(x)
```

Flexpy test units

Activity 4



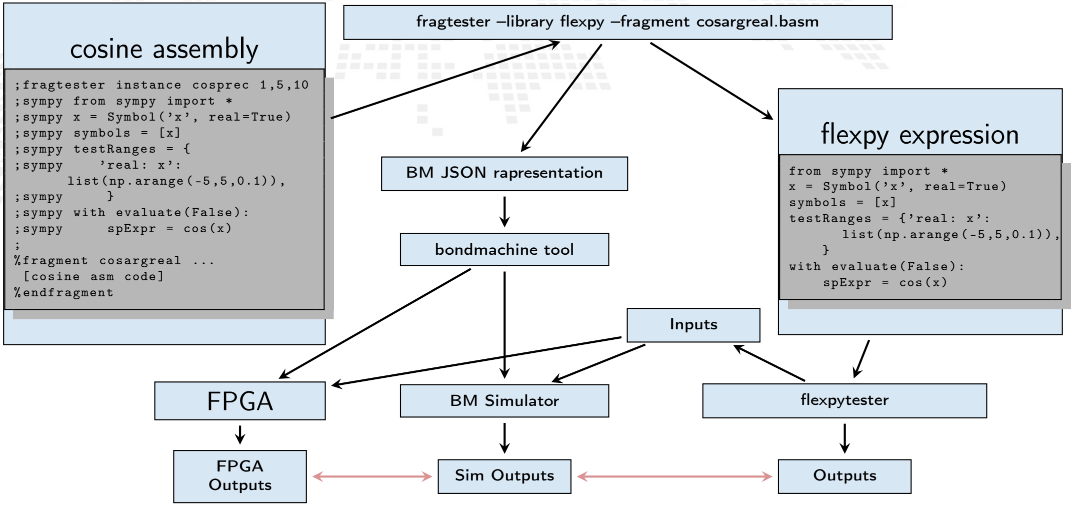
```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy    'real: x':
      list(np.arange(-5,5,0.1)),
;sympy     }
;sympy with evaluate(False):
;sympy     spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```

```
fragtester –library flexpy –fragment cosargreal.basm
```

```
from sympy import *
x = Symbol('x', real=True)
symbols = [x]
testRanges = {'real: x':
      list(np.arange(-5,5,0.1)),
    }
with evaluate(False):
    spExpr = cos(x)
```

Flexpy test units

Activity 4



```
;fragtester instance cosprec 1,5,10
;sympy from sympy import *
;sympy x = Symbol('x', real=True)
;sympy symbols = [x]
;sympy testRanges = {
;sympy     'real: x':
      list(np.arange(-5,5,0.1)),
;sympy     }
;sympy with evaluate(False):
;sympy     spExpr = cos(x)
;
%fragment cosargreal ...
 [cosine asm code]
%endfragment
```

```
from sympy import *
x = Symbol('x', real=True)
symbols = [x]
testRanges = {'real: x':
      list(np.arange(-5,5,0.1)),
    }
with evaluate(False):
    spExpr = cos(x)
```

Conclusions, Ongoing Work and Future Work

We enabled the simulation of quantum circuits using FPGA.

Ongoing work:

- The inclusion of a parametric quantum circuit in the bmqsim framework.
- The development of all the instructions needed to simulate quantum circuits in the flexpy framework.
- The development of a HLS backend for the flexpy framework.

Goals:

- Operate useful quantum circuits on FPGA and apply all the techniques we developed to test and optimize the computation in terms of latency, throughput, power consumption and numerical precision.
- Use the HPC bubbles with FPGA accelerators to run (not only) quantum circuits on multi-FPGA systems.