



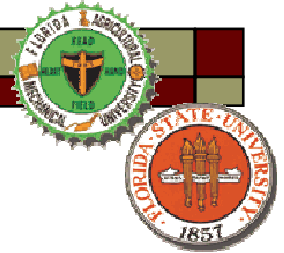
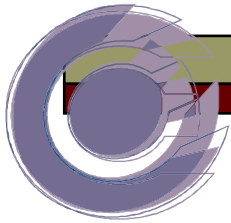
FAMU-FSU
College of Engineering



Space-Efficient Simulation of Quantum Computers

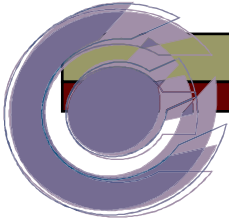
47th ACM Southeast Conference, Clemson, SC
March 19-21, 2009 (Session F3, Systems)

**Michael P. Frank¹, Uwe H. Meyer-Baese¹,
Irinel Chiroescu², Liviu Oniciuc¹, Robert A. van Engelen³**
¹Dept. of Elec. & Comp. Eng., FAMU-FSU College of Engineering
²National High Magnetic Field Laboratory, Florida State University
³Department of Computer Science, Florida State University



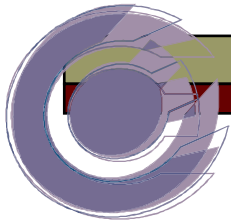
Abstract (for reference)

Traditional algorithms for simulating quantum computers on classical ones require an exponentially large amount of memory, and so typically cannot simulate general quantum circuits with more than about 30 or so qubits on a typical PC-scale platform with only a few gigabytes of main memory. However, more memory-efficient simulations are possible, requiring only polynomial or even linear space in the size of the quantum circuit being simulated. In this paper, we describe one such technique, which was recently implemented at FSU in the form of a C++ program called SEQCSim, which we releasing publicly. We also discuss the potential benefits of this simulation in quantum computing research and education, and outline some possible directions for further progress.



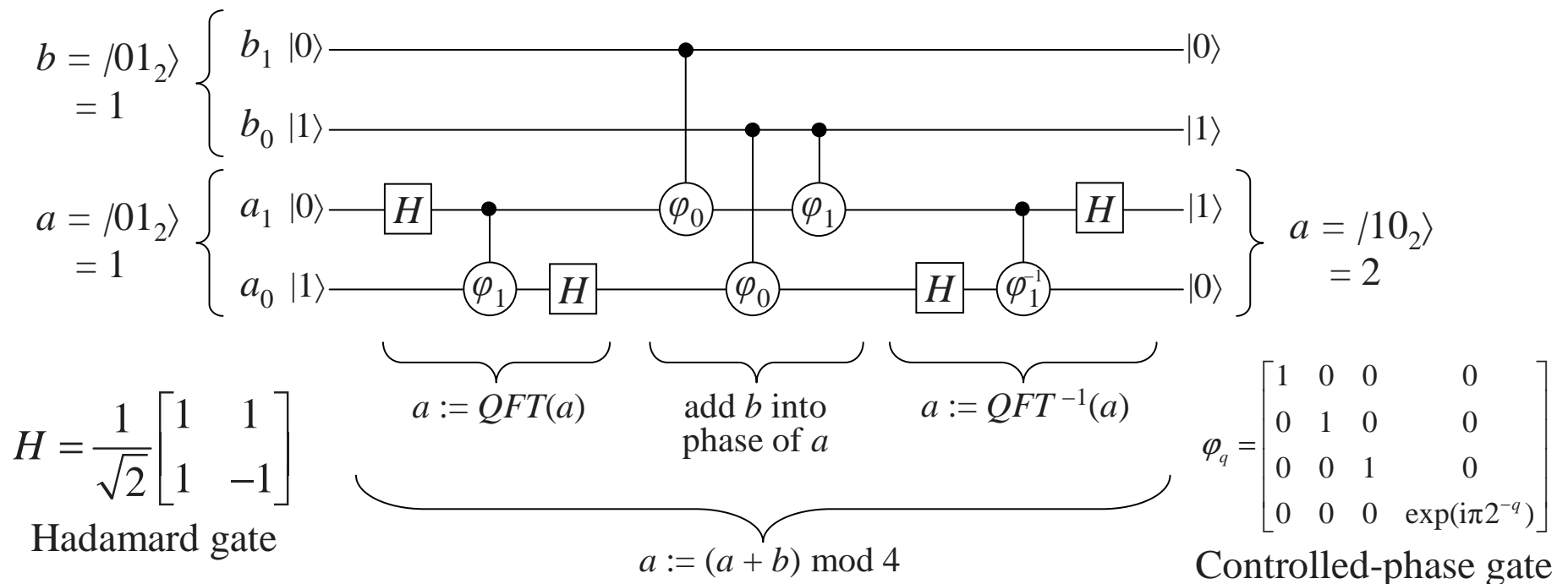
What is a Quantum Computer?

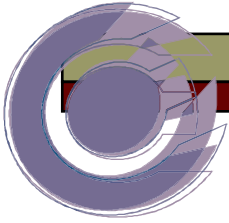
- A new, more powerful fundamental paradigm for computing within the laws of physics.
 - Apparently exponentially faster on some problems.
- Some key differences between Classical vs. Quantum Computation:
 - State representations:
 - **Classical:** A sequence of n bit values, $w \in \mathbf{B}^n$, where $\mathbf{B} = \{0,1\}$.
 - **Quantum:** A function $\Psi \in \mathbf{H}$, where $\mathbf{H} = \mathbf{B}^n \rightarrow \mathbf{C}$, mapping classical states to complex numbers (“amplitudes”).
 - Logic operators (“gates”):
 - **Classical:** A function from several bits to one bit, $g:\mathbf{B}^k \rightarrow \mathbf{B}$
 - **Quantum:** A unitary (invertible, length-preserving) linear transformation $U:\mathbf{S} \rightarrow \mathbf{S}$, where $\mathbf{S} = \mathbf{B}^k \rightarrow \mathbf{C}$.
 - Measurement of computation results:
 - **Classical:** Measured value is exactly determined by machine state.
 - **Quantum:** Probability of measuring state as being w is $\propto |\Psi(w)|^2$.



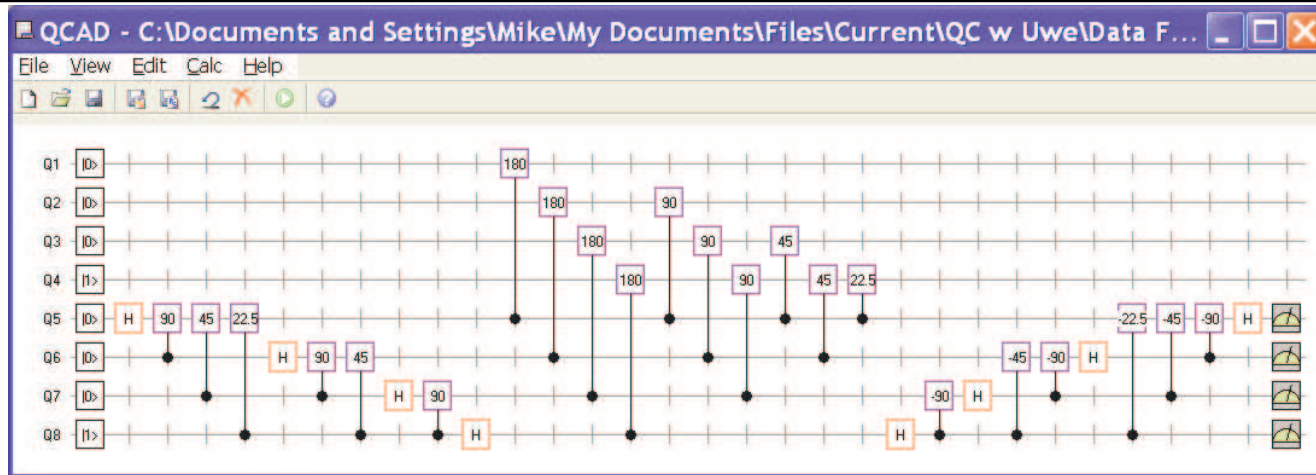
A Simple Quantum Circuit: Draper Adder

Uses the quantum Fourier transform (QFT) and its inverse QFT^{-1} to add two 2-bit input integers in a temporary phase-based representation. Here it is computing $1 + 1 = 2$.



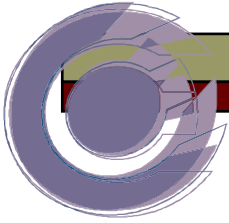


A Larger Draper Adder (2×4 bits)



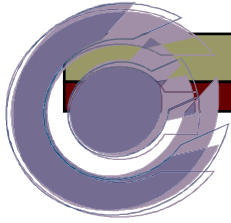
QCAD tool, by Hiroshi Watanabe, University of Tokyo, available from
<http://apollon.cc.u-tokyo.ac.jp/~watanabe/qcad/index.html>

- Some advantages of the Draper adder:
 - Minimal quantum space usage: Requires no ancilla bits for carries.
 - A good simple, but nontrivial example of a quantum algorithm.
- A disadvantage of the Draper adder:
 - Slow; requires $\Theta(n^2)$ gates for an n -bit add!
 - Unlikely to be used in practice, unless qubits are very expensive.



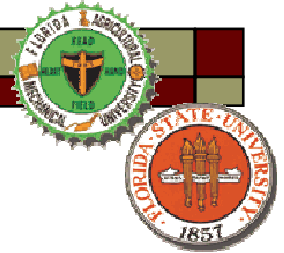
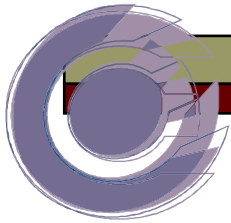
Some Potential Applications of Quantum Computers

- If quantum computers of substantial size are built, known quantum algorithms can be applied to obtain:
 - Polynomial-time cryptanalysis of popular public-key cryptosystems (*e.g.*, RSA).
 - Polynomial-time simulations of quantum-mechanical physical systems.
 - Square-root speedups of simple unstructured searches of computed oracle functions.
 - And not a whole lot else!
- A much wider variety of interesting & useful quantum algorithms is needed,
 - But new quantum algorithms are difficult to develop.
 - Need flexible, capable simulation tools for design validation.



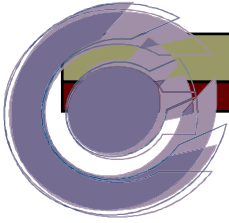
A Problem with Nearly All Existing Quantum Computer Simulators

- They require *exponential space* as the number of bits in the simulated computer increases.
 - **Why:** They update a *state vector* explicitly representing the full wavefunction $\Psi: \mathbf{B}^n \rightarrow \mathbf{C}$.
 - This vector contains 2^n complex numbers
 - 1 for each possible configuration of the machine's n bits
 - If the available memory holds 1G (2^{30}) numbers,
 - We can only simulate <30-bit quantum computers!
 - The large space usage also imposes a significant slowdown to access main memory or disk.



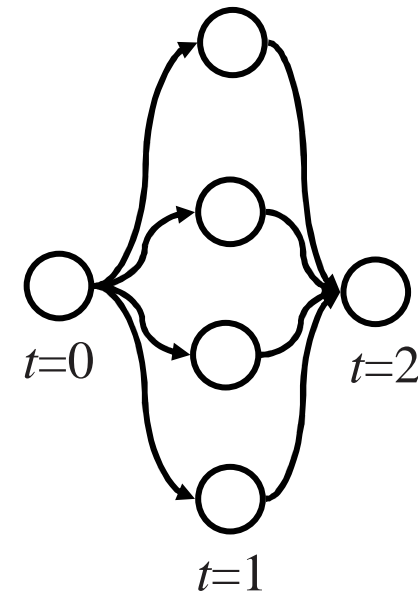
A Way to Solve This Problem

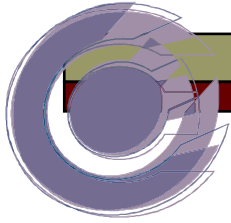
- We can reformulate quantum mechanics in an equivalent framework *without state vectors*.
 - Feynman (1942): Any desired amplitude can be computed using a *path integral* expression summing over possible *classical* trajectories.
 - Bohm (1952): Can maintain a *classical* state that evolves under the influence of only wavefunction amplitudes in the immediate neighborhood.
- The only real requirement is to obtain the right probability of arriving at each final state!



A Complexity Theorist's View of Feynman's Path Integral

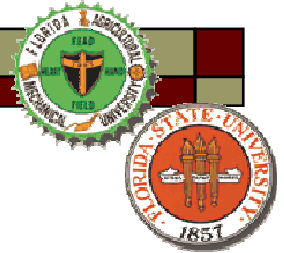
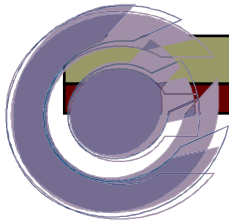
- Consider any computation with a wide dataflow graph (uses more space than time)
 - The graph at right uses 4 variables at time $t=1$, but only takes 2 steps.
- We can make the algorithm more space-efficient by recomputing intermediate variables dynamically when needed, instead of storing them.
- Bernstein & Vazirani, 1993: Can apply this generic tradeoff to simulating quantum computers (duh).





SEQCSim: The Space-Efficient Quantum Computer Simulator

- Core idea was conceived circa 2002 at UF.
 - Adding Bohm updates to Feynman recursion.
 - Avoids having to enumerate all possible final states.
- A working C++ software prototype was developed and demonstrated at FSU in 2007.
 - Future versions of the simulator will have a more expressive programming interface.
- A performance-optimized FPGA-based implementation is currently being developed.



SEQCSim Input Files for 2×2-Bit Draper Adder

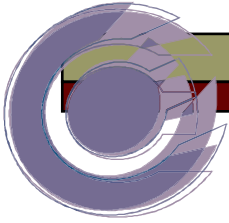
```
qconfig.txt format version 1
bits: 4      Declare registers
named bitarray: a[2] @ 0
named bitarray: b[2] @ 2
```

```
qinput.txt format version 1
a = 1      Input values to add
b = 1
```

```
qoperators.txt format version 1
operators: 4
operator #: 0
name: H
size: 1 bits
matrix:
(0.7071067812 + i*0)(0.7071067812 + i*0)
(0.7071067812 + i*0)(-0.7071067812 + i*0)
operator #: 1
name: cZ      Gate
size: 2 bits  definitions
matrix:
(1 + i*0) (0 + i*0) (0 + i*0) (0 + i*0)
(0 + i*0) (1 + i*0) (0 + i*0) (0 + i*0)
(0 + i*0) (0 + i*0) (1 + i*0) (0 + i*0)
(0 + i*0) (0 + i*0) (0 + i*0) (-1 + i*0)
... (two additional operators elided for brevity)
```

Quantum circuit (gate application sequence)

```
qopseq.txt format version 1
operations: 9
operation #0: apply unary operator H to bits a[1]
operation #1: apply binary operator cPiOver2 to bits a[1], a[0]
operation #2: apply unary operator H to bits a[0]
operation #3: apply binary operator cZ to bits b[1], a[1]
operation #4: apply binary operator cZ to bits b[0], a[0]
operation #5: apply binary operator cPiOver2 to bits b[0], a[1]
operation #6: apply unary operator H to bits a[0]
operation #7: apply binary operator inv_cPiOver2 to bits a[1], a[0]
operation #8: apply unary operator H to bits a[1]
```



SEQCSim Core Algorithm

// Bohm-inspired iterative state updating.

procedure SEQCSim::run():

curState := *inputState*; **// Current basis state**

curAmp := 1; **// Current amplitude**

for *PC* =: 0 to #gates, **// Current gate index**

(w.r.t. gate[*PC*] operator and its operands,)

for each neighbor *nbri* of *curState*,

if *nbri* = *curState*, *amp*[*nbri*] := *curAmp*;

else *amp*[*nbri*] := calcAmp(*nbri*);

amp[] := opMatrix * *amp*[]; **// Matrix prod.**

// Calculate probabilities as normalized

// squares of amplitudes.

prob[] := normSqr(*amp*[]);

// Pick a successor of the current state.

i := pickFromDist(*prob*[]);

curState := *nbri*; *curAmp* := *amp*[*nbri*].

// Feynman-inspired recursive

// amplitude-calculation procedure.

function SEQCSim::calcAmp(Neighbor *nbr*):

curState := *nbr*;

if *PC*=0 return (*curState* = *inputState*) ? 1 : 0;

(w.r.t. gate[*PC*-1] operator and its operands,)

for each predecessor *predi* of *curState*,

PC := *PC* - 1;

amp[*predi*] = calcAmp(*predi*);

PC := *PC* + 1;

amp[] := opMatrix * *amp*[];

return *amp*[*curState*];

Complete C++ console app has
24 source files, total size 115 KB

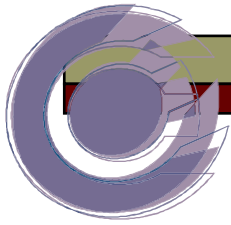
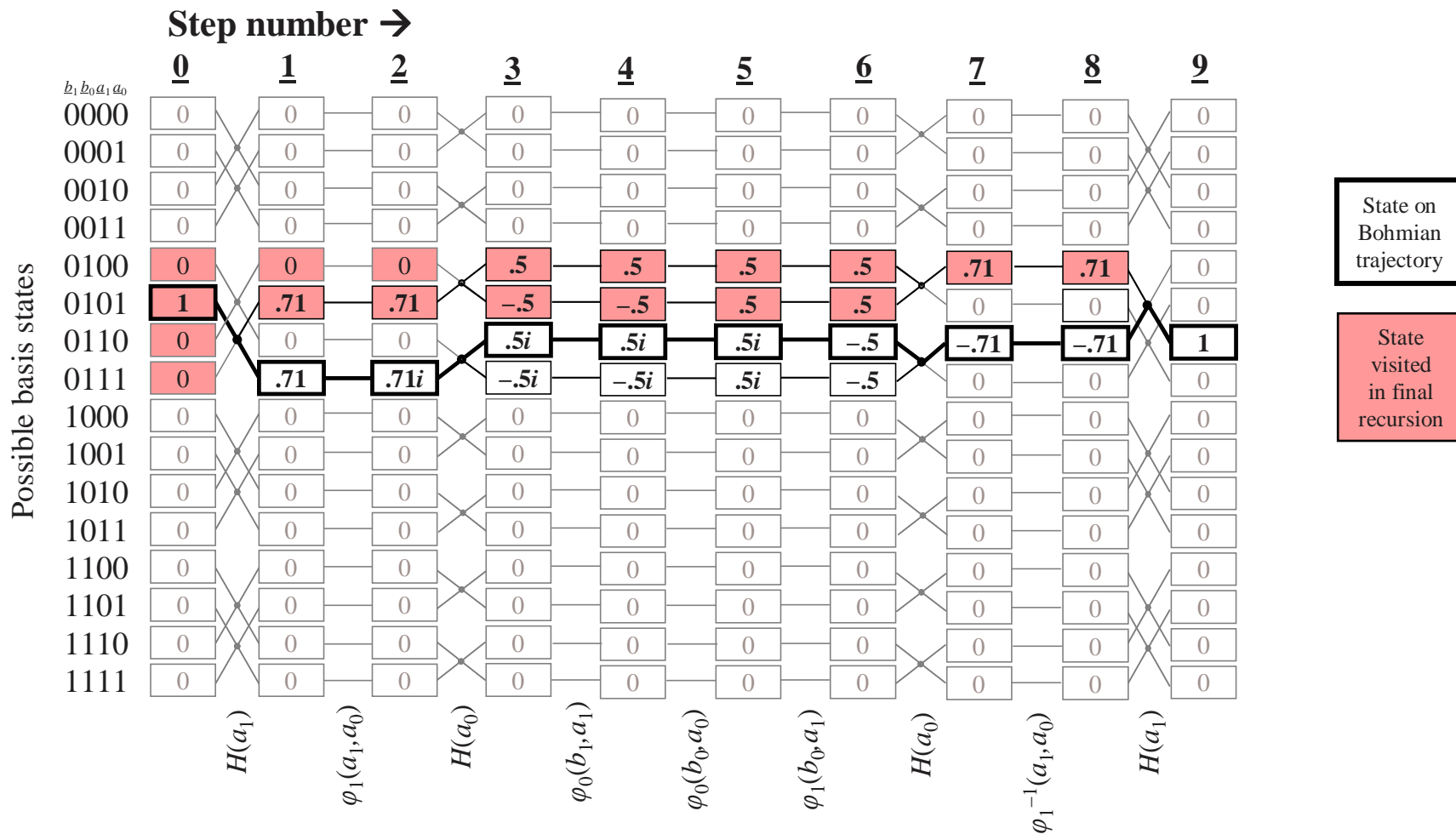
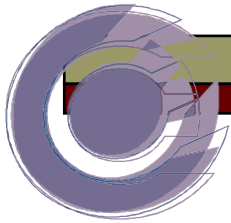


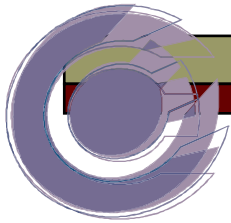
Illustration of SEQCSim Operation on 2×2-Bit Draper Adder





Complexity Analysis

- Defining the following parameters:
 - $a = \text{const.} = \text{max. arity of quantum gates}$
 - $s = \text{width (\# of qubits) in simulated circuit}$
 - $t = \text{time (\# of operations) in simulated circuit}$
 - $k (< t) = \# \text{ of } \textit{nontrivial} \text{ operations in sim'd circ.}$
- For a moderately well-optimized implementation of SEQCSim, we can have
 - Space complexity: $O(s + t)$
 - Time complexity: $O(s + t \cdot 2^{ak})$



SEQCSim Output on 2×2-Bit Draper Adder

Welcome to SEQCSIM, the Space-Efficient Quantum Computer SIMulator.

(C++ console version)

By Michael P. Frank, Uwe Meyer-Baese, Irinel Chiorescu, and Liviu Oniciuc.

Copyright (C) 2008 Florida State University Board of Trustees.

All rights reserved.

SEQCSim::run(): Initial state is 3->0101<-0 (4 bits) ==> $(1 + i*0)$.

SEQCSim::Bohm_step_forwards(): (tPC=0)

The new current state is 3->0111<-0 (4 bits) ==> $(0.707107 + i*0)$.

SEQCSim::Bohm_step_forwards(): (tPC=1)

The new current state is 3->0111<-0 (4 bits) ==> $(0 + i*0.707107)$.

... (5 intermediate steps elided for brevity) ...

SEQCSim::Bohm_step_forwards(): (tPC=7)

The new current state is 3->0110<-0 (4 bits) ==> $(-0.707107 + i*0)$.

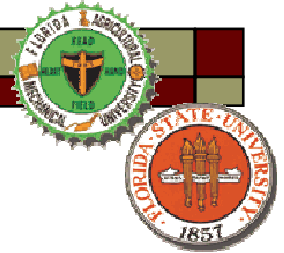
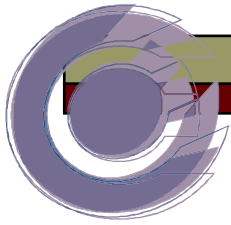
SEQCSim::Bohm_step_forwards(): (tPC=8)

The new current state is 3->0110<-0 (4 bits) ==> $(1 + i*0)$.

SEQCSim::done(): The PC value 9 is \geq the number of operations 9.

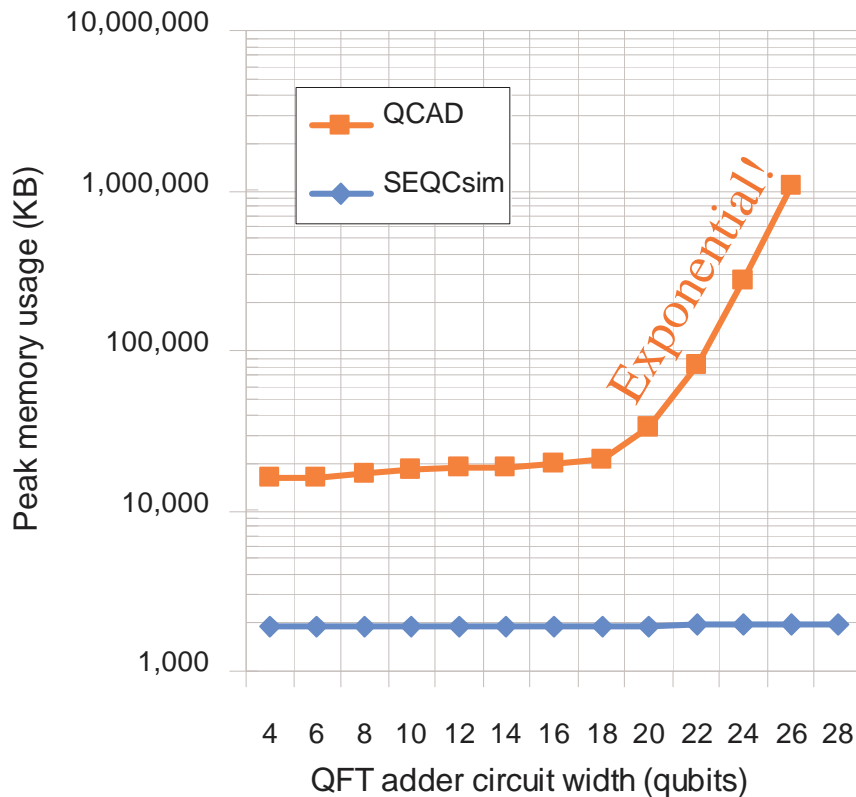
We are done!

$1+1=2=10_2$



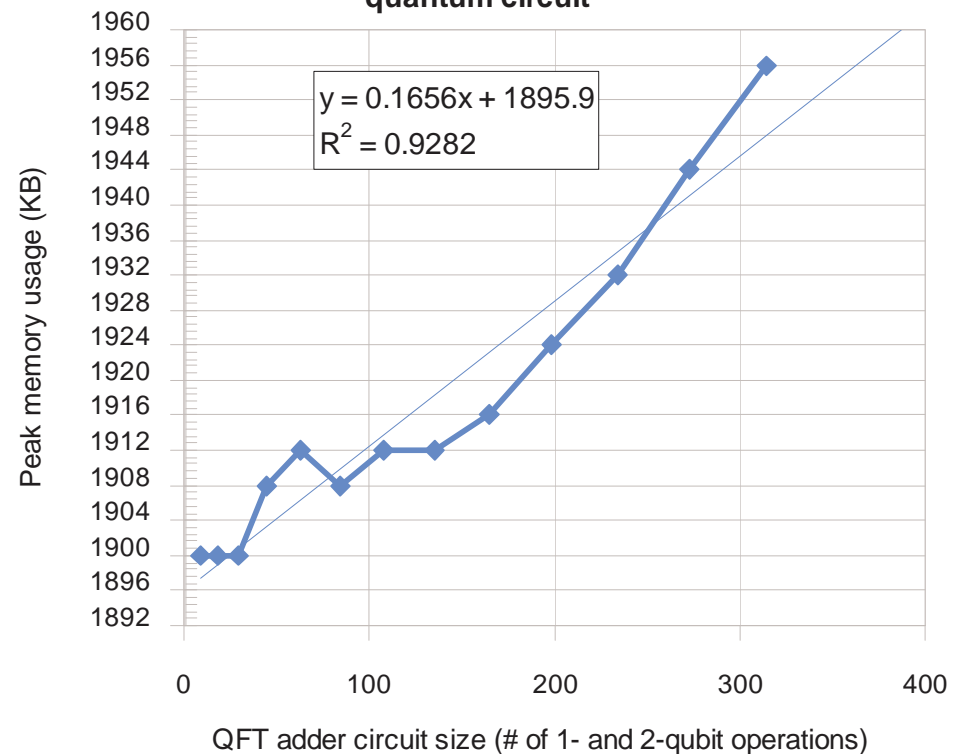
Empirical Measurements of Space Complexity

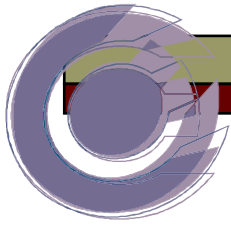
QCAD vs. SEQCsim memory usage



(Note: QCAD crashed on the 28-bit circuit, due to insufficient memory available on the test PC.)

Linear growth of SEQCsim memory usage with size of quantum circuit

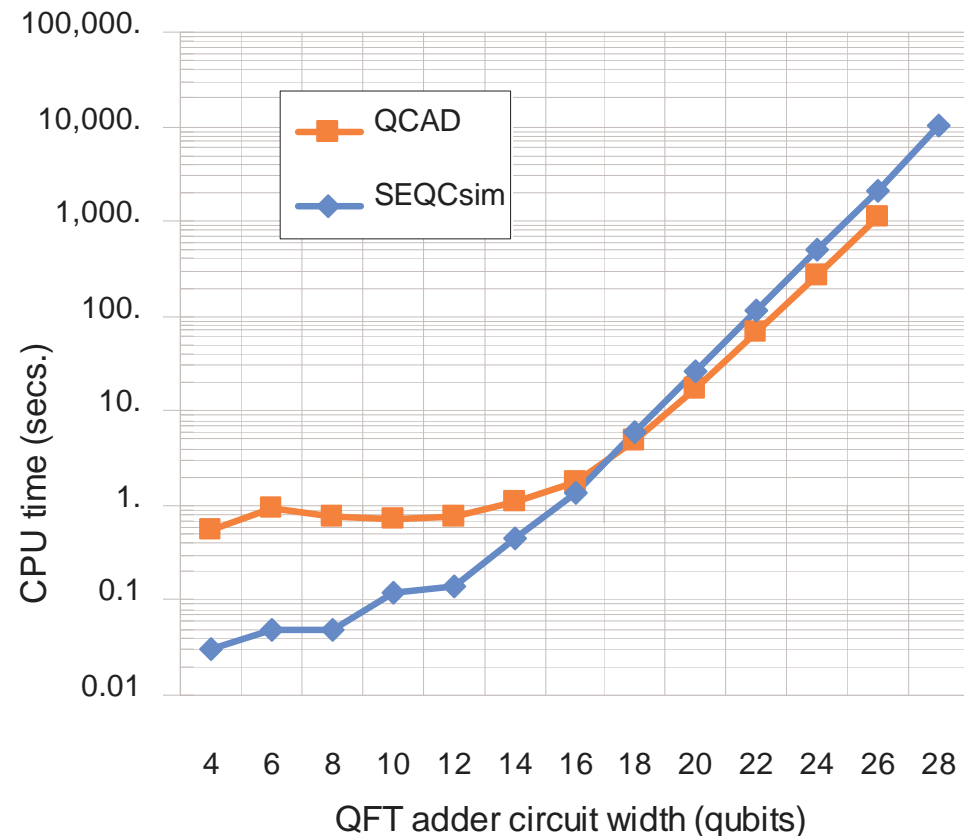


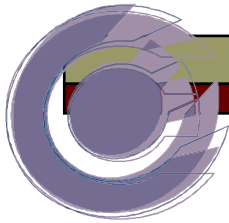


Empirical Measurements of CPU Time Utilization

- SEQCSim is $10\times$ faster than QCAD on small circuits
 - This is probably largely because QCAD has a GUI and SEQCSim doesn't.
- SEQCSim is $\sim 2\times$ slower than QCAD on large circuits.
 - But there is much room for improvement.
 - Take better advantage of available memory.
 - Reimplement in special-purpose hardware

QCAD vs. SEQCsim CPU time usage





FPGA Tools (1 of 5): Altera SOPC Builder

Altera SOPC Builder - nios_system.sopc (C:\AlteraDE2\DE2_DMA_tutorial\nios_system.sopc)

File Edit Module System View Tools Nios II Help

System Contents System Generation

Altera SOPC Builder

- Nios II Processor
- Bridges and Adapters
- Interface Protocols
 - ASi
 - Ethernet
 - High Speed
 - PCI
 - Serial
 - Avalon-ST JTAG I
 - Avalon-ST Serial I
 - JTAG UART
 - SPI (3 Wire Serial)
 - UART (RS-232 Se
- Legacy Components
- Memories and Memory Contro
 - DMA
 - Flash
 - On-Chip
 - Avalon-ST Dual C
 - Avalon-ST Multi-C
 - Avalon-ST Round

Target

Device Family: Cyclone II

Clock Settings

Name	Source	MHz
clk_1	External	50.0

Add Remove

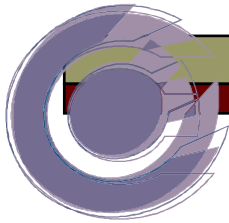
Use	Conne...	Module Name	Description	Clock	Base	End	IRQ
<input checked="" type="checkbox"/>		cpu_0	Nios II Processor				
		instruction_master	Avalon Memory Mapped Master	clk_1			
		data_master	Avalon Memory Mapped Master	clk_1			
		jtag_debug_module	Avalon Memory Mapped Slave	clk_1	0x01002800	0x01002fff	IRQ 0
<input checked="" type="checkbox"/>		onchip_memory2_0	On-Chip Memory (RAM or ROM)				
		s1	Avalon Memory Mapped Slave	clk_1	0x01001000	0x01001fff	
<input checked="" type="checkbox"/>		Switches	PIO (Parallel I/O)				
		s1	Avalon Memory Mapped Slave	clk_1	0x01003020	0x0100302f	
<input checked="" type="checkbox"/>		LEDs	PIO (Parallel I/O)				
		s1	Avalon Memory Mapped Slave	clk_1	0x01003030	0x0100303f	
<input checked="" type="checkbox"/>		jtag_uart_0	JTAG UART				
		avalon_jtag_slave	Avalon Memory Mapped Slave	clk_1	0x01003040	0x0100304f	
<input checked="" type="checkbox"/>		sdram_0	SDRAM Controller				
		s1	Avalon Memory Mapped Slave	clk_1	0x00800000	0x00ffffff	
<input checked="" type="checkbox"/>		sys_clk_timer	Interval Timer				
		s1	Avalon Memory Mapped Slave	clk_1	0x01003000	0x0100301f	

New... Edit... Add...

Remove Edit... Move Up Move Down Address Map... Filter...

Warning: **Switches**: PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.

Exit Help Prev Next Generate



FPGA Tools (2 of 5): NIOS II Soft-Core Configuration

Nios II Processor - cpu_0

Settings

Core Nios II > Caches and Memory Interfaces > Advanced Features > MMU and MPU Settings > JTAG Debug Module > Custom Instructions

Core Nios II

Select a Nios II core:

	<input type="radio"/> Nios II/e	<input type="radio"/> Nios II/s	<input checked="" type="radio"/> Nios II/f
Nios II Selector Guide Family: Cyclone II f _{system} : 50.0 MHz cpuicd: 0	RISC 32-bit	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide	RISC 32-bit Instruction Cache Branch Prediction Hardware Multiply Hardware Divide Barrel Shifter Data Cache Dynamic Branch Prediction
Performance at 50.0 MHz	Up to 5 DMIPS	Up to 25 DMIPS	Up to 51 DMIPS
Logic Usage	600-700 LEs	1200-1400 LEs	1400-1800 LEs
Memory Usage	Two M4Ks (or equiv.)	Two M4Ks + cache	Three M4Ks + cache

Hardware Multiply: ☐ Hardware Divide

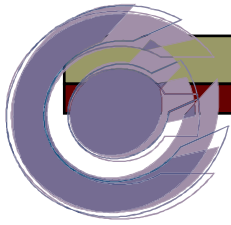
Reset Vector: Memory: Offset: 0x00800000

Exception Vector: Memory: Offset: 0x00800020

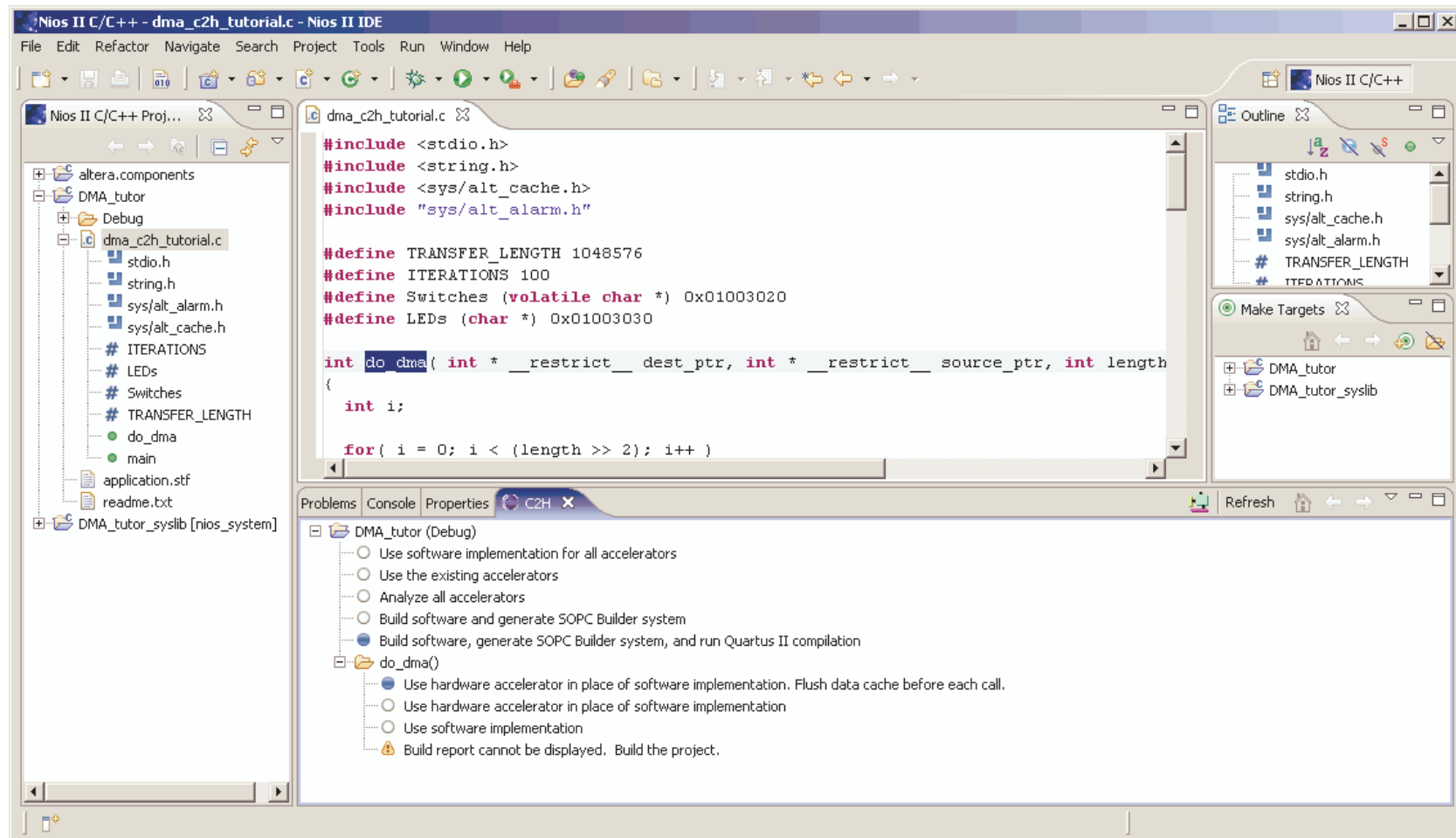
☐ Include MMU
Only include the MMU when using an operating system that explicitly supports an MMU
Fast TLB Miss Exception Vector: Memory: Offset:

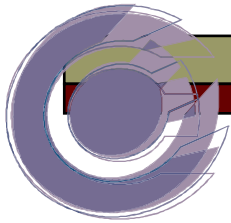
☐ Include MPU

Cancel < Back Next > Finish

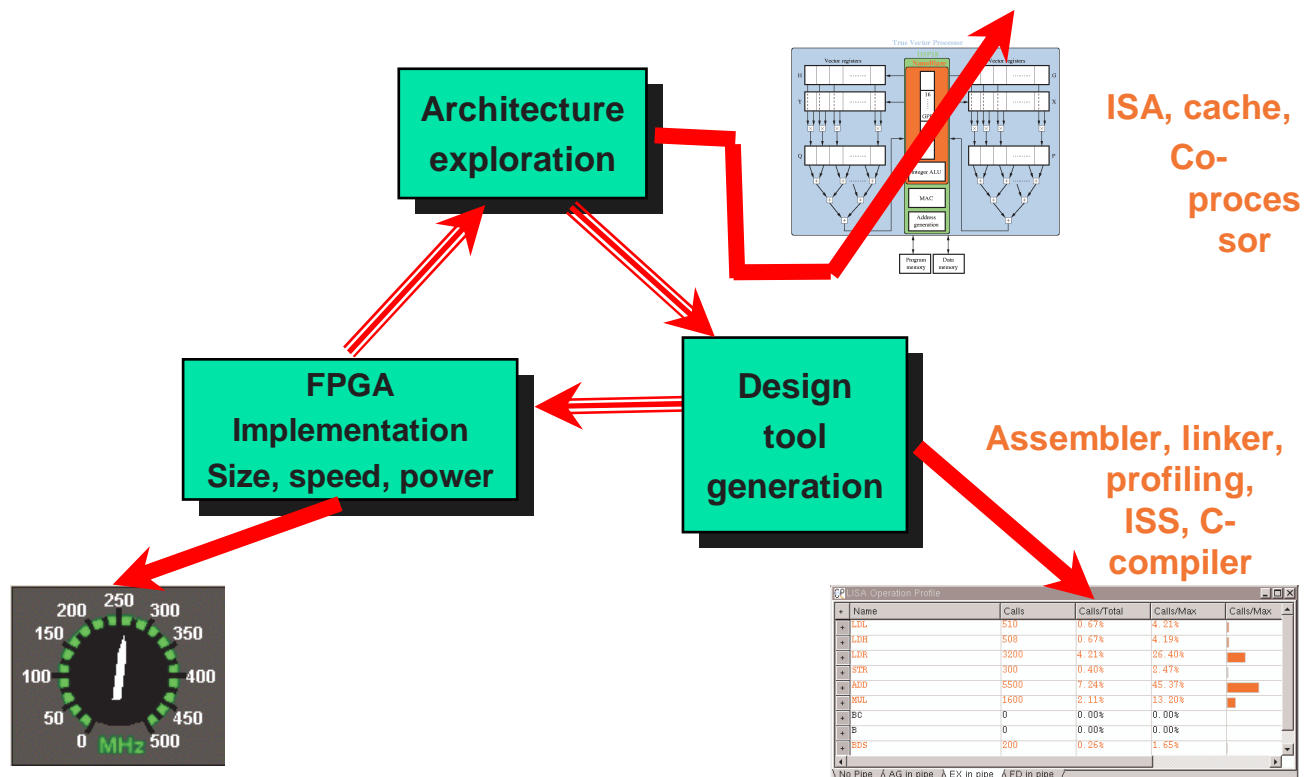


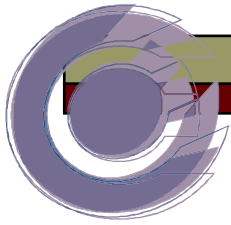
FPGA Tools (3 of 5): Custom Hardware Generation with C2H





FPGA Tools (4 of 5): LISA Processor Design Cycle





FPGA Tools (5 of 5): LISA Development Tools

Processor Debugger: /vlsi/home/meyerbaese/models/dsp18/app/mactest/mactest.out

File Program Debug View Profiling Windows Extras Help

Symbol Set Image Symbols Goto Symbol Goto Address 0x00000000 32K Application:

File: "mactest.asm"

```
[00000000]: NOP
[00000001]: NOP

; R[2] has pointer to coefficients operands
[00000002]: LDH R[2], #(_h0 & 0xff)
[00000003]: LDH R[2], #(_h0 >> 8)
; R[3] is pointer to x data array
[00000004]: LDH R[3], #(_x0 & 0xff)
[00000005]: LDH R[3], #(_x0 >> 8)
[00000006]: NOP
[00000007]: NOP

; R[4]=R[4]+SR[3]++ * SR[2]++
[00000008]: MAC R[4], R[3], R[2]
[00000009]: MAC R[4], R[3], R[2]
[0000000a]: MAC R[4], R[3], R[2]

; Test program
[0000000b]: NOP
[0000000c]: NOP
[0000000d]: NOP
[0000000e]: NOP
```

Disassembler

Symbols	Address	Instruction	Disassembly
	[00000001]	00000	NOP
	[00000002]	01200	LDH R[2], #0
	[00000003]	02200	LDH R[2], #0
	[00000004]	01306	LDH R[3], #6
	[00000005]	02300	LDH R[3], #0
	[00000006]	00000	NOP
	[00000007]	00000	NOP
	[00000008]	1b324	MAC R[4], R[3], R[2]
	[00000009]	1b324	MAC R[4], R[3], R[2]
	[0000000a]	1b324	MAC R[4], R[3], R[2]
	[0000000b]	00000	NOP
	[0000000c]	00000	NOP
	[0000000d]	00000	NOP

Memory monitor

Address	0	1	2	3	4	5	6	7
0000000000	00001	00002	00003	00004	00005	00006	00007	00008
0000000008	00040	00080	00160	00320	00640	01280	02560	05120
0000000016	00000	00000	00000	00000	00000	00000	00000	00000
0000000024	00000	00000	00000	00000	00000	00000	00000	00000
0000000032	00000	00000	00000	00000	00000	00000	00000	00000
0000000040	00000	00000	00000	00000	00000	00000	00000	00000
0000000048	00000	00000	00000	00000	00000	00000	00000	00000
0000000056	00000	00000	00000	00000	00000	00000	00000	00000
0000000064	00000	00000	00000	00000	00000	00000	00000	00000
0000000072	00000	00000	00000	00000	00000	00000	00000	00000
0000000080	00000	00000	00000	00000	00000	00000	00000	00000
0000000088	00000	00000	00000	00000	00000	00000	00000	00000

Profiler

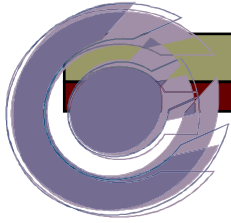
Name	Calls	Calls/Total
NOP	7	8.86%
decode	14	17.72%
B_type	0	0.00%
D_type	0	0.00%
M_type	3	3.80%
I_type	5	5.06%
R_type	0	0.00%
direct_addressing	0	0.00%
indirect_addressing	0	0.00%
indirect2_addressing	3	3.80%

regs

Name	Value
EPC	12
APC	13
FPC	14
BPC	0
BPC_valid	0
R[0]	0
R[1]	0
R[2]	3
R[3]	9
R[4]	170
R[5]	0
R[6]	0
R[7]	0
R[8]	0
R[9]	0
R[10]	0
R[11]	0
R[12]	0
R[13]	0
R[14]	0
R[15]	0

Nothing to do
Nothing to do
Nothing to do
000A * 0001 + 0000 = 000A
0014 * 0002 + 000A = 0032
0028 * 0003 + 0032 = 00AA
Nothing to do
Nothing to do
stdout stderr

Source: mactest.asm 56 Step: 14 Simulation Mode: JIT-CCS (c) CoWare LISATek Version 2005.2.1 Linux -- February, 2006



Conclusion & Future Work

- We have implemented in C++ and validated a working prototype of a quantum computer simulator that uses only linear space.
 - This tool can be useful to help students & researchers validate quantum algorithms.
 - Online resources at <http://www.eng.fsu.edu/~mpf/SEQCSim>
 - Contact michael.patrick.frank@gmail.com for source code
 - A future version will provide a more expressive quantum programming language based on C++.
- We are also designing an FPGA-based hardware implementation to boost simulator performance.
 - This approach is made much more feasible by the extreme memory-efficiency of our algorithm.