Reconfigurable Computing Applications for High Perfomance Technical Computing

Troy Benjegerdes, Sean Stanek

Problem Background

- Apply Reconfigurable Computing to High-end Supercomputing applications
- Large, complex legacy codes
- 64 bit floating point
- Multi-processor parallelism

Goals/Objectives

- Examine GAMESS computational chemistry code as example code
  - Computational chemistry is a 'hard' problem to deal with.. if a methodology to accelerate this problem for HPC can be developed, it may be applicable to most HPC type codes
  - (FFT, image processing, etc are relatively 'easy')
- Profile the target application
- Develop a projection on possible speedups
- Target a subrouting for execution on an FPGA

Initial Assumptions

- Primarily double precision floating point math (precision is critical!!)
- Chemistry codes use many transcendental functions (exp, pow, sqrt)
- Could reasonably offload at least one transcendental function into an FPGA
  - Initial estimates are that one function will use up most of currently available FPGA's
- FPGA area continues to increase
  - Signifigant advantages as fpga area grows

Experimental Setup

- Get a profile of calls to transcendental functions vs calls to floating point multiply/add
- Problem: transcendental functions are in libm, floating point is in hardware
- Solution: Use software floating point!
  - Not particularly trivial
  - Requires rebuilding GCC, a c library (in this case uCLibc, http://www.uclibc.org), Fortran libg2c libraries, and convincing all these parts to output profileing data

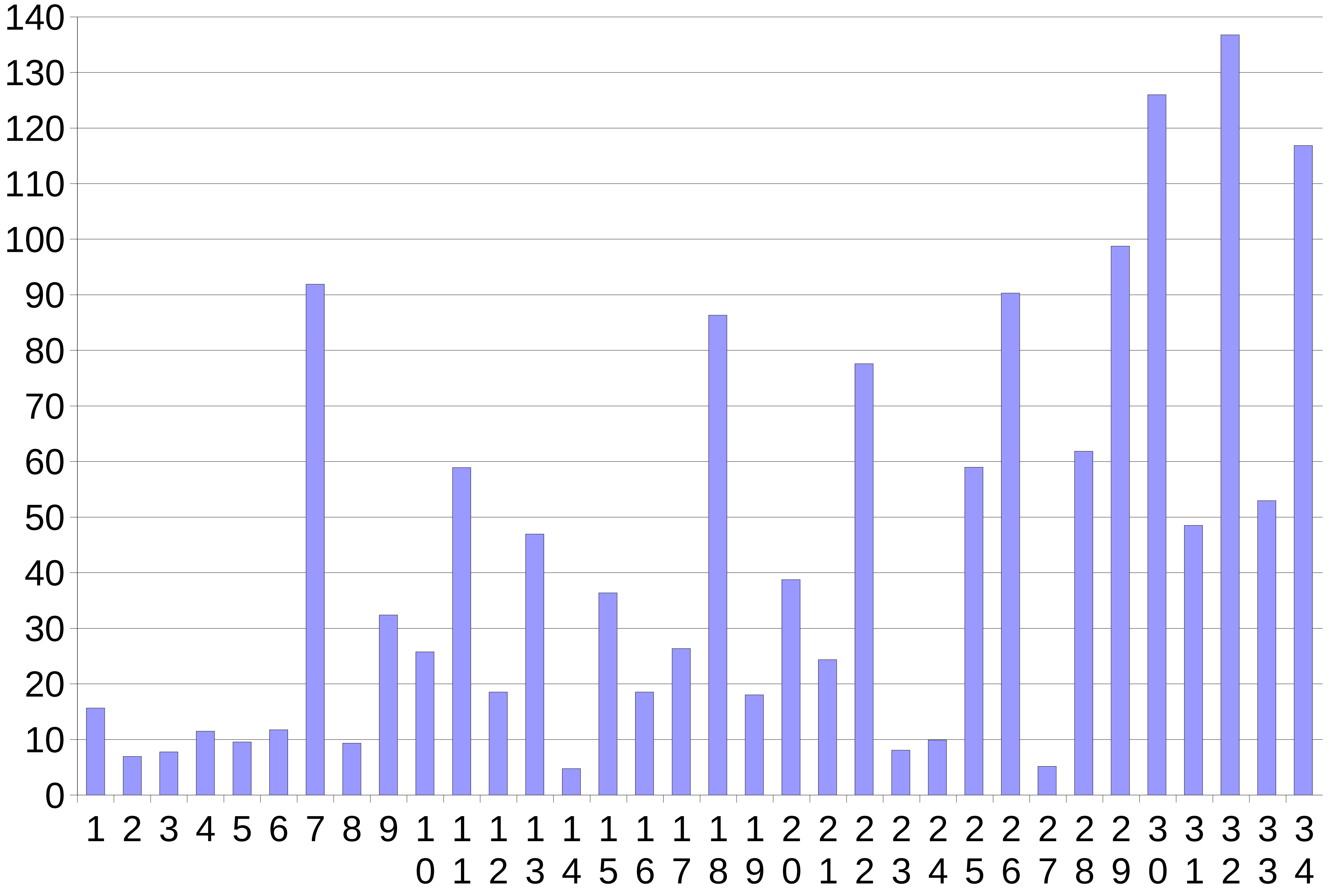
Profileing GAMESS

- System used: Debian Linux, 667mhz PPC7455 (g4) CPU
- Time to build GAMESS + uCLibc + libg2c measured in hours
- 34 example simulations provided with GAMESS, testing various code paths and type of simulations

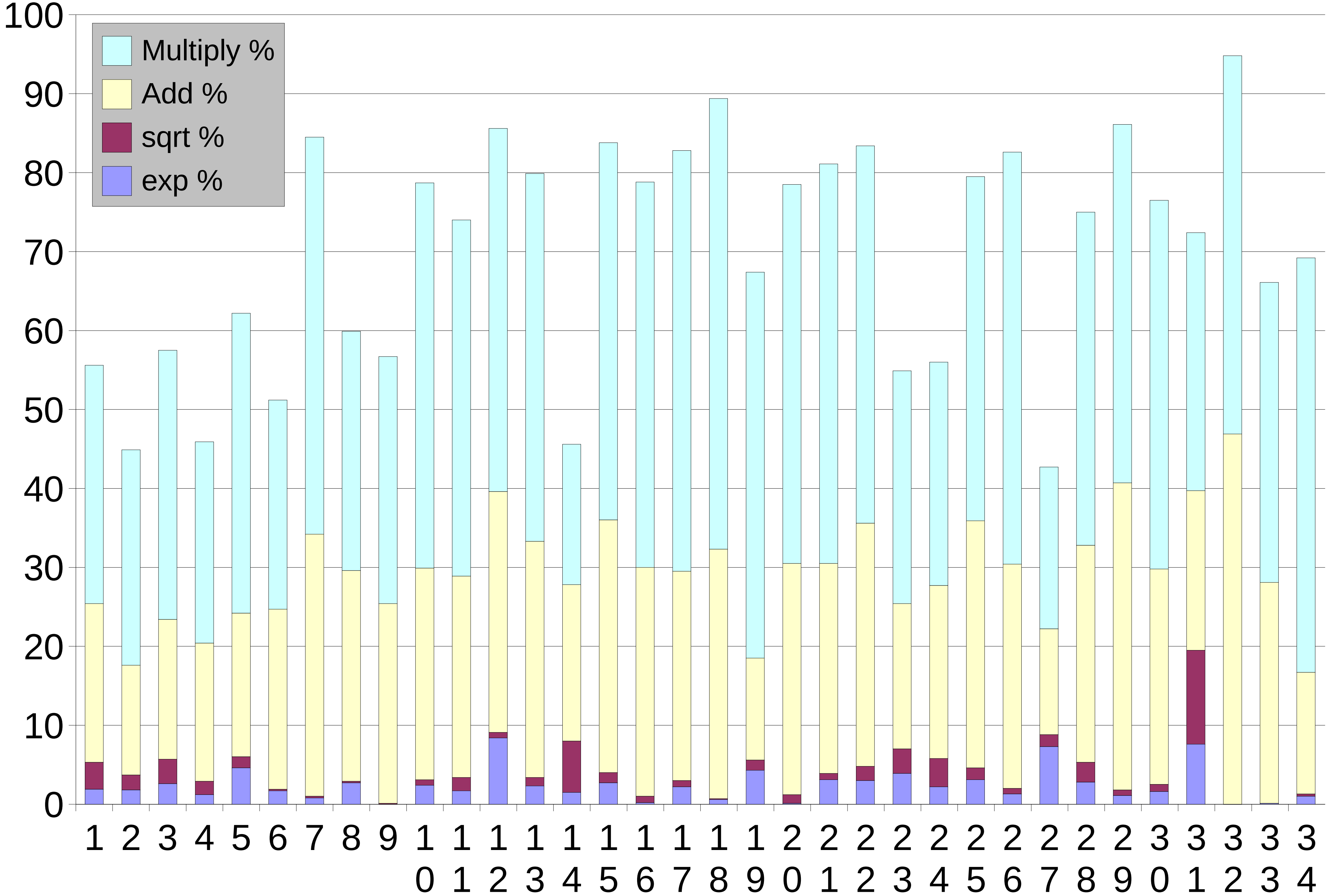
Achievements

- Profiling results
  - Nothing stuck out except MAIN, and software floating point subroutines
  - VERY dependant on type of run
- projections of exp() performance on fpga
  - matrix-multiply shows power and total GFLOPS advantage over Pentium IV on Virtex2Pro

Softfloat vs Hardfloat run time ratio

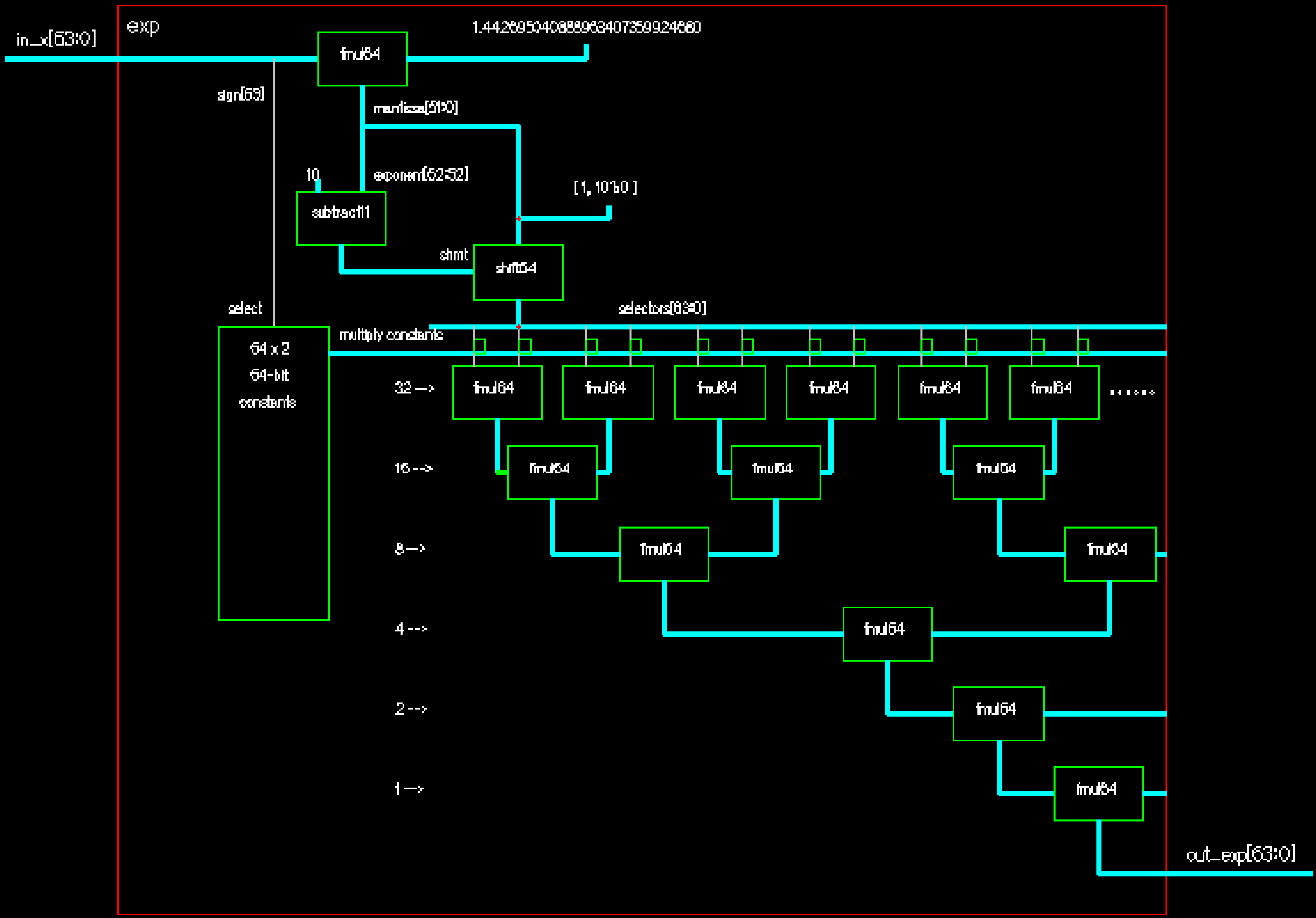


GAMESS w/softfloat profile results, % of time in function



Implementation

- Can be implemented on modern large FPGAs
- $\exp(x) = e^x = 2^{x*\log_2 e}$
- Generic algorithm tweaked and optimized for hardware
  - Sixty-four 64-bit floating-point multipliers
  - Integer adder and variable shifter
  - 128-entry array of 64-bit floating-point numbers
- 7 pipeline stages of multipliers
  - Tree-style convolution
- 1 pipeline stage for add & shift



in_x[63:0]
exp
fmul64
1.4426950408889634073599246810
sign[63]
mantissa[51:0]
10
exponent[62:52]
subtract11
[1, 10'b0]
shamt
shift64
select
selectors[63:0]
multiply constants
64 x 2
64-bit
constants
32 -->
16 -->
8 -->
4 -->
2 -->
1 -->
fmul64
out_exp[63:0]

Space Analysis

- Dillon Engineering 64-bit floating-point multiply cores take 783 slices per multiply
  - Roughly 50,000 slices are needed just for the multiply logic
  - Other logic is less space significant
- Largest Virtex II Pro part has 55,616 slices
- For smaller FPGAs, the algorithm could be modified to reuse the multipliers at the cost of speed

Time Analysis

- Multiply stages take between 8-23 clocks
- Overall latency of ~57-162 clocks
- Theoretical throughput of 1 result per clock
- Clock speeds reaching 130MHz - 200MHz
  - 2.0 GHz Opteron @ ~37M results/sec (54clk/op)
  - 550 MHz Athlon @ 7.6M results/sec (72clk/op)
  - 1.2 GHz Power4 @ 5.3M results/sec (226clk/op)
  - 667 MHz G4 7455 @ 2.7M results/sec (247clk/op)
  - 400 MHz G4 @ 1.8M results/sec (222clk/op)

Problems

- Figuring out how to integrate the FPGA with software
  - Memory bandwidth (200M * 64 bits = 12.8Gbps)
  - Software must have a need for this many ops
- Design could be extended to include more complex operations with less need for data
- Quick operations with low latency might be difficult

Future work

- Run tightly-coupled on a VirtexII-Pro
  - Use PPC-softfloat build, but replace _muldf3, _adddf3, exp, and sqrt with fpga-memory mapped operations?
  - Needs some compiler/tool magic to deal with pipelineing issues
- Verify correct operation with pipelined-C code simulation of exp() on fpga
- Examine Feasability of re-codeing portions to allow deeply-pipelined exp and sqrt functions