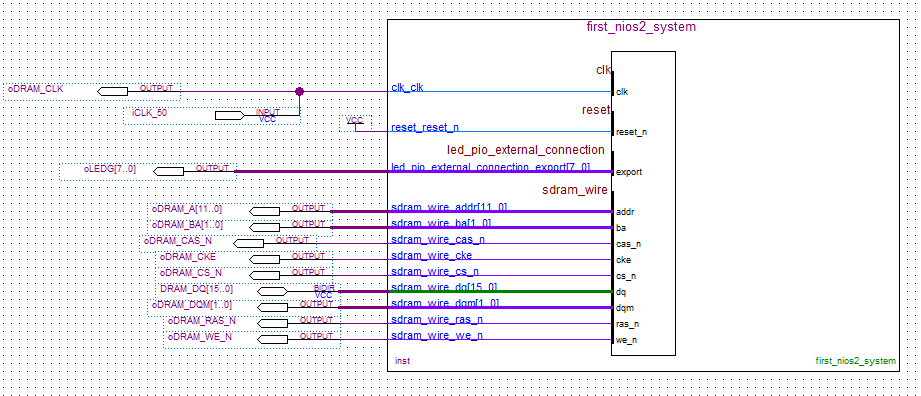
**Part I**

**Setting Up NIOS II**

Nios II – is a 32-bit embedded softcore microprocessor (a microprocessor than can be implemented by logic synthesis on an FPGA)

Setting up the NIOS II system, was a relatively trivial task –following the tutorial correctly was all that was required.

We first performed this on the DE2-70 board, and then later on the DE0 board. As we were one of the first groups to do this – we found there was an issue with the compiling of Quartus projects on the lab computers – they have to be compiled on the H:/ drive rather than the mapped \\ic.ac.uk\homes\ drive. Once this issue was overcome we proceeded on to the next problem.



The system was then tested using count binary and the hello\_world example files set out in the NIOS II Eclipse software.

**Determinant Square Matrix in Software**

The determinant of a 3x3 Matrix performed in software was done on the NIOS II. We used the LU Decomposition method to find the determinant of the matrix. In practice the amount of computer time used to calculate the speed of a program is proportional to the arithmetic and storage operations used to calculate the result. Using this measure we can calculate the LU decomposition of an NXN matrix to be . However a recursive method for finding the determinant of an NxN matrix is O(n!) . (K Eriksson – Computational Differential Equations 1996)

float determinant**(**float **\***matrix**,** int dimension**){**

int i**,** j**,** p**;**

float a**,** result**;**

float **\***m**;**

// Let us copy the matrix first

m **=** **(**float **\*)** malloc**(** **sizeof(**float**)\***dimension**\***dimension **);**

memcpy**(**m**,** matrix**,** **sizeof(**float**)\***dimension**\***dimension **);**

// First step: perform LU Decomposition using Doolittle's Method

// This algorithm will return, in the same matrix, a lower unit triangular matrix

// (i.e. diagonals one)

// and an upper trangular matrix

// https://vismor.com/documents/network\_analysis/matrix\_algorithms/S4.SS2.php

**for** **(**i **=** 0**;** i **<** dimension**;** i**++){**

**for** **(**j **=** 0**;** j **<** i**;** j**++){**

a **=** getAt**(**m**,** i**,** j**,** dimension**);**

**for** **(**p **=** 0**;** p **<** j**;** p**++){**

a **-=** getAt**(**m**,** i**,** p**,** dimension**)** **\*** getAt**(**m**,** p**,** j**,** dimension**);**

**}**

putAt**(**m**,** i**,** j**,** dimension**,** a**/**getAt**(**m**,** j**,** j**,** dimension**));**

**}**

**for** **(**j **=** i**;** j **<** dimension**;** j**++){**

a **=** getAt**(**m**,** i**,** j**,** dimension**);**

**for** **(**p **=** 0**;** p **<** i**;** p**++){**

a **-=** getAt**(**m**,** i**,** p**,** dimension**)** **\*** getAt**(**m**,** p**,** j**,** dimension**);**

**}**

putAt**(**m**,** i**,** j**,** dimension**,** a**);**

**}**

**}**

// Second step is to find the determinant.

// Because the lower triangle is a unit triangular matrix

// the determinant is simply a product of all the upper triangle diagonal

// which in this case is exactly the diagonal of m

result **=** 1**;**

**for** **(**i **=** 0**;** i **<** dimension**;** i**++)**

result **\*=** getAt**(**m**,** i**,** i**,** dimension**);**

free**(**m**);**

**return** result**;**

**}**

// Based on i and j, and a float pointer, get the value at row i column j

float getAt**(**float **\***m**,** int i**,** int j**,** int dimension**){**

**return** **\*(**m **+** i**\***dimension **+** j**);**

**}**

// Based on i and j, and a float pointer, put the value at row i column j

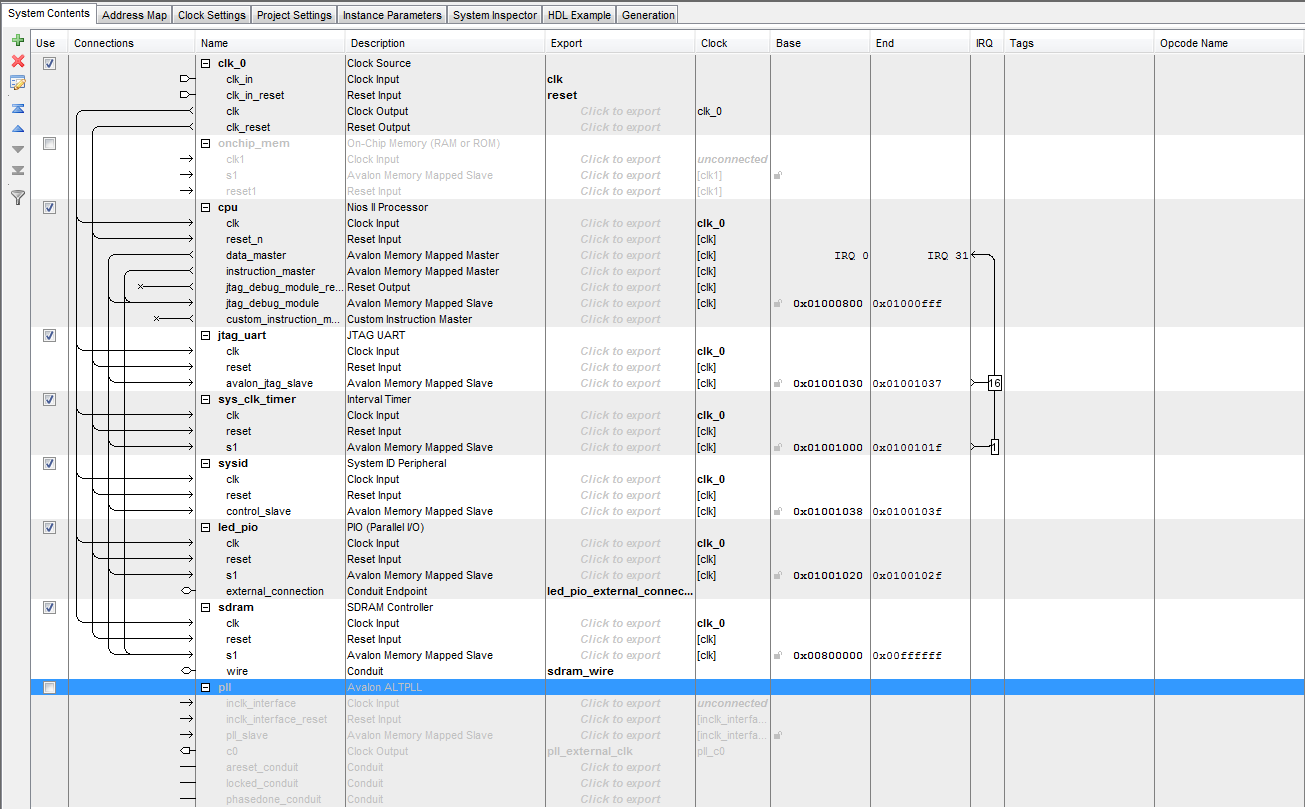
void putAt**(**float **\***m**,** int i**,** int j**,** int dimension**,** float value**){**

**\*(**m **+** i**\***dimension **+** j**)** **=** value**;**

**}**

Calculation of the 3X3 matrix took - 0.009 for 10 iterations.

**SDRAM**

****

3x3 Matrix SDRAM 100 iterations 0.077 s

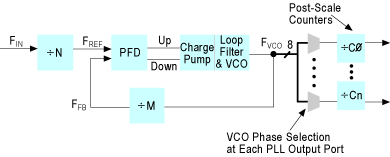
6x6 Matrix SDRAM 100 iterations 0.537 s

8x8 Matrix SDRAM 100 iterations 1.245 s

10x10 Matrix SDRAM 100 iterations 2.494 s

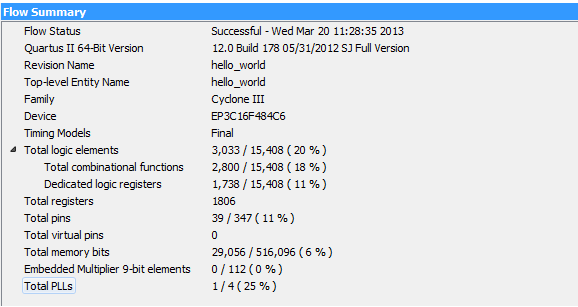
20x20 Matrix SDRAM 100 iterations 20.343 s

As the SDRAM is very sensitive to clock variations we added a PLL with a time-shift of 2.55ns. A PLL is a phase-locked loop, this is a closed loop feedback system based on the difference in phase between the clock input and a clock signal of a controlled oscillator.



N – a pre-scale counter, PFD – phase frequency detection, VCO voltage controlled oscillator, M – a feedback counter, N – a pre-scale counter and C - post-scale counters.

Image Source http://www.altera.com/support/devices/pll\_clock/images/fig\_07\_00\_PLL\_block.gif



From this image we can see that on the DE0 board – the NIOS2 and SDRAM with a PLL takes up 20% of the logic elements and 6% of the total memory.

**Embedded Multipliers**

Embedded Multipliers are configured as either one 18 x 18 multiplier or two 9 x 9 Multipliers, the Cyclone III chip on the DE0 board has 112 Embedded 19x19 multipliers.

Times and Sizes

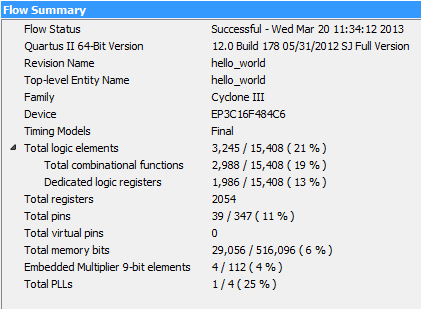
3x3 Matrix SDRAM With embedded multipliers 100 iterations 0.044s

6x6 Matrix SDRAM With embedded multipliers 100 iterations 0.283s

8x8 Matrix SDRAM With embedded multipliers 100 iterations 0.658s

10x10 Matrix SDRAM With embedded multipliers 100 iterations 1.278s

20x20 Matrix SDRAM With embedded multipliers 100 iterations 10.669s



Using Embedded Multipliers uses 1% (212) more logic elements on the board, and uses an additional 4 Embedded Multipliers. It uses 248 more registers.

**LUT – based Multipliers**

In addition to embedded multipliers, there are also look-up tables on the cyclone II chip created from M9K memory blocks. Look-up tables are implemented using very small amounts of RAM, and are used to implement combinational logic – for example an N-input LUT can implement any Boolean function of N inputs.

Times and sizes –

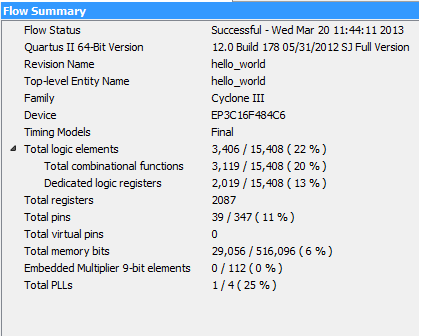
3x3 Matrix SDRAM with LUT based Multipliers 0.045s

6x6 Matrix SDRAM with LUT based Multipliers 100 iterations 0.286s

8x8 Matrix SDRAM with LUT based Multipliers 100 iterations 0.666s

10x10 Matrix SDRAM with LUT based Multipliers 100 iterations 1.293s

20x20 Matrix SDRAM with LUT based Multipliers 100 iterations 10.72s

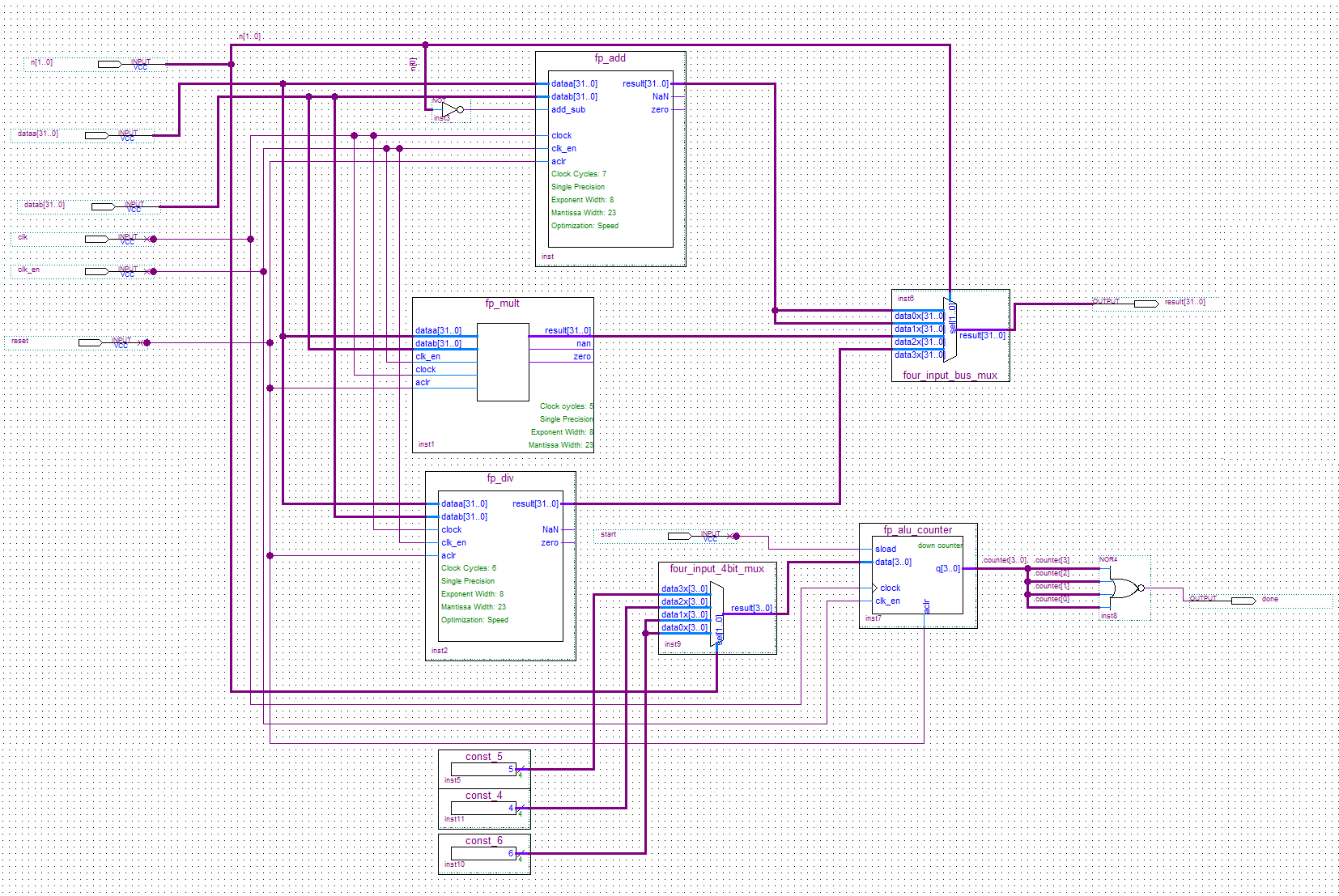


Using LUTs uses 2% more logic elements than the SDRAM alone, 281 more registers.

From the data and the graph it can be seen that the SDRAM alone is the slowest for all Matrix dimensions and Embedded Multipliers are the fastest. However there are only a few tens of milliseconds difference over 100 iterations between the LUT and Embedded Multipliers, this will make a minimal difference over 1 iteration – however if we were to want to do one million iterations of a 20x20 matrix would make a 510 second difference between using LUTs and Embedded Multipliers, so in terms of speed it makes sense to use Embedded Multipliers.

However on the DE0 board there are only 112 Embedded multipliers, but 15000 Lookup tables – so if a larger project was undertaken then it would be sensible to put the most time sensitive calculations onto Embedded Multipliers and the rest on to the Lookup tables.

**External Floating Point Hardware Accelerator**

****

To create the floating point custom instruction hardware, the megawizard function blocks were used. The exponent and mantissa widths were set to 8 bits and 23bits respectively for all blocks. The latency for the floating point blocks were 5, 6 and 7 cycles for the floating point multiply, divide and add/subtract block respectively. As we are not trying to overclock the NIOS2 CPU clock to over 50MHz choosing the lowest latency will not cause any problems. The lowest latency also uses the smallest size on the FPGA so for our requirements this seems to be the optimal design choice – smallest size and fastest operation. However if we had wanted to increase the frequency of the clock we would have had to choose a higher latency to allow for more pipelines states within the Floating point unit to shorten the critical path.

As an extension to this Floating Point ALU a bypass operation was added in for 0, 1, NAN and infinity as they are neutral, or absorbent, elements in addition and multiplication, according to the floating point standard. This extension was implemented in the Verilog file fp\_alu\_improved.v which can be seen in the **appendix below** .

3x3 Floating Point 100 iterations 0.016s

6x6 Floating Point 100 iterations 0.035s

8x8 Floating Point 100 iterations 0.058s

10x10 Floating Point 100 iterations 0.09s

20x20 Floating Point 100 iterations 0.489s

Using Hardware Floating Point Instructions

float determinant**(**float **\***matrix**,** int dimension**){**

int i**,** j**,** p**;**

float a**,** result**;**

float **\***m**;**

// Let us copy the matrix first

m **=** **(**float **\*)** malloc**(** **sizeof(**float**)\***dimension**\***dimension **);**

memcpy**(**m**,** matrix**,** **sizeof(**float**)\***dimension**\***dimension **);**

// First step: perform LU Decomposition using Doolittle's Method

// This algorithm will return, in the same matrix, a lower unit triangular matrix

// (i.e. diagonals one)

// and an upper trangular matrix

// https://vismor.com/documents/network\_analysis/matrix\_algorithms/S4.SS2.php

**for** **(**i **=** 0**;** i **<** dimension**;** i**++){**

**for** **(**j **=** 0**;** j **<** i**;** j**++){**

a **=** getAt**(**m**,** i**,** j**,** dimension**);**

**for** **(**p **=** 0**;** p **<** j**;** p**++){**

a **=** fp\_sub**(**a**,** fp\_mul**(** getAt**(**m**,** i**,** p**,** dimension**),** getAt**(**m**,** p**,** j**,** dimension**))** **);**

**}**

putAt**(**m**,** i**,** j**,** dimension**,** a**/**getAt**(**m**,** j**,** j**,** dimension**));**

**}**

**for** **(**j **=** i**;** j **<** dimension**;** j**++){**

a **=** getAt**(**m**,** i**,** j**,** dimension**);**

**for** **(**p **=** 0**;** p **<** i**;** p**++){**

a **=** fp\_sub**(**a**,** fp\_mul**(** getAt**(**m**,** i**,** p**,** dimension**)** **,** getAt**(**m**,** p**,** j**,** dimension**)));**

**}**

putAt**(**m**,** i**,** j**,** dimension**,** a**);**

**}**

**}**

// Second step is to find the determinant.

// Because the lower triangle is a unit triangular matrix

// the determinant is simply a product of all the upper triangle diagonal

// which in this case is exactly the diagonal of m

result **=** 1**;**

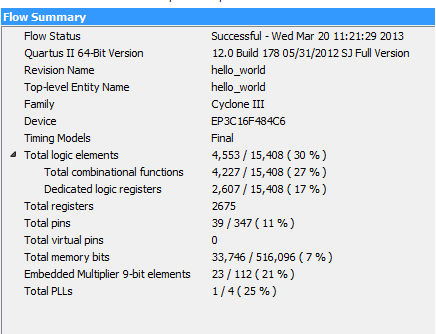
**for** **(**i **=** 0**;** i **<** dimension**;** i**++)**

result **=** fp\_mul**(**result**,** getAt**(**m**,** i**,** i**,** dimension**));**

free**(**m**);**

**return** result**;**

**}**



Adding a floating point Arithmetic and Logic unit to calculate the determinant of the matrix has increased the number of logic elements by 9% of the total available. The total memory usage increased by 1% and the number of embedded multipliers used has increased by 19 elements (17%).

**External Determinant Hardware Accelerator**

**Algorithm Selection**

The determinant hardware accelerator was implemented in Verilog. The algorithm used in the hardware determinant is the LU decomposition with Doolittle algorithm. (The pseudo-code used can be found here <https://vismor.com/documents/network_analysis/matrix_algorithms/S4.SS2.php>) The reason for this using this algorithm is stated above – the recursive (and probably simplest to implement) is O(n!) and whereas LU decomposition is O(n3)

**Hardware Design**

Due to the nature of the LU Decomposition modifying the values of the matrix in situ for further computation later on there is no way to simply cache any particular column or row of the matrix we used the RAM to cache the entire matrix, which is instantiated using an M9K memory block.

**Hardware Implementation**

The Verilog implementation uses a state machine consisting of 5 states.

The first state (state 0) is the idle state where the hardware is ready to compute. If invoked with a non-zero address via the Custom Instruction slave interface, the hardware will proceed to the second state and return “99” to the processor. If the dimension of the matrix is less than one it will return “-1” to the processor to indicate its idle status.

The second state (state 1) is the read from SDRAM state where data is read from the word addressed SDRAM addressed (so the next word is found by adding 4 to the address – 4 bytes to a word). There is just one cycle latency in reading from SDRAM, this is achieved by using the Avalon slave which co-ordinates the reading data from SDRAM – to request a new address from RAM we wait for the waitRequest signal to go low (a sign that Avalon Slave has received our previous request) and the read signal to be high. We can then increment to the next address. To receive the data from the address we wait for the readdatavalid signal to be high and the ramwritedone signal must remain low. The reason for separate request and receive states as there is could be an indeterminate amount of time between requesting and receiving so it will be more efficient these two issues. Due to the nature of the Avalon bus we therefore know that the request and then receive will be sent in the correct order. When this data is read from SDRAM it is moved to RAM generated by an M9K memory block.

The third state (state 2) – is the Doolittle’s LU Decomposition algorithm state, there are many stages due to the

Instead of swapping the contents of memory locations when row swapping is required we instead swap the memory addresses of the data, which is faster than memory swaps.

The fourth state (state 3) – is the Diagonal Multiplication state – where diagonals of the matrix are multiplied.

The fifth and final state (state 4) is the result output state – the done values are set high and the result is passed to the Avalon slave.

For small dimension sizes (less than 4X4), it may be faster to do the matrix calculations in software if you had a data caches as NIOS could then perform single cycle reads which will be significantly faster than reading from SDRAM in the hardware determinant implementation. However once we get past these small sizes of MATRIX the hardware determinant implementation will always be faster, providing we have enough memory to perform the calculations. 4 M9Ks are enough to store one 32x 32 matrix, and 16M9Ks are required to store one 64X64 matrix. It can be seen that as the matrix size doubles the number of M9Ks required squares. This is gets close to the limit of space that available on the DE0 board.

**Possible extensions to the project**

**Multiple NIOS II on the FPGA – should in theory be able to get 4 NIOS II on the FPGA.**