Appendices \label{sec:appendices}

# Expanded Views of the Hardware Layout[[1]](#footnote-1) \label{sec:appendices:layout}

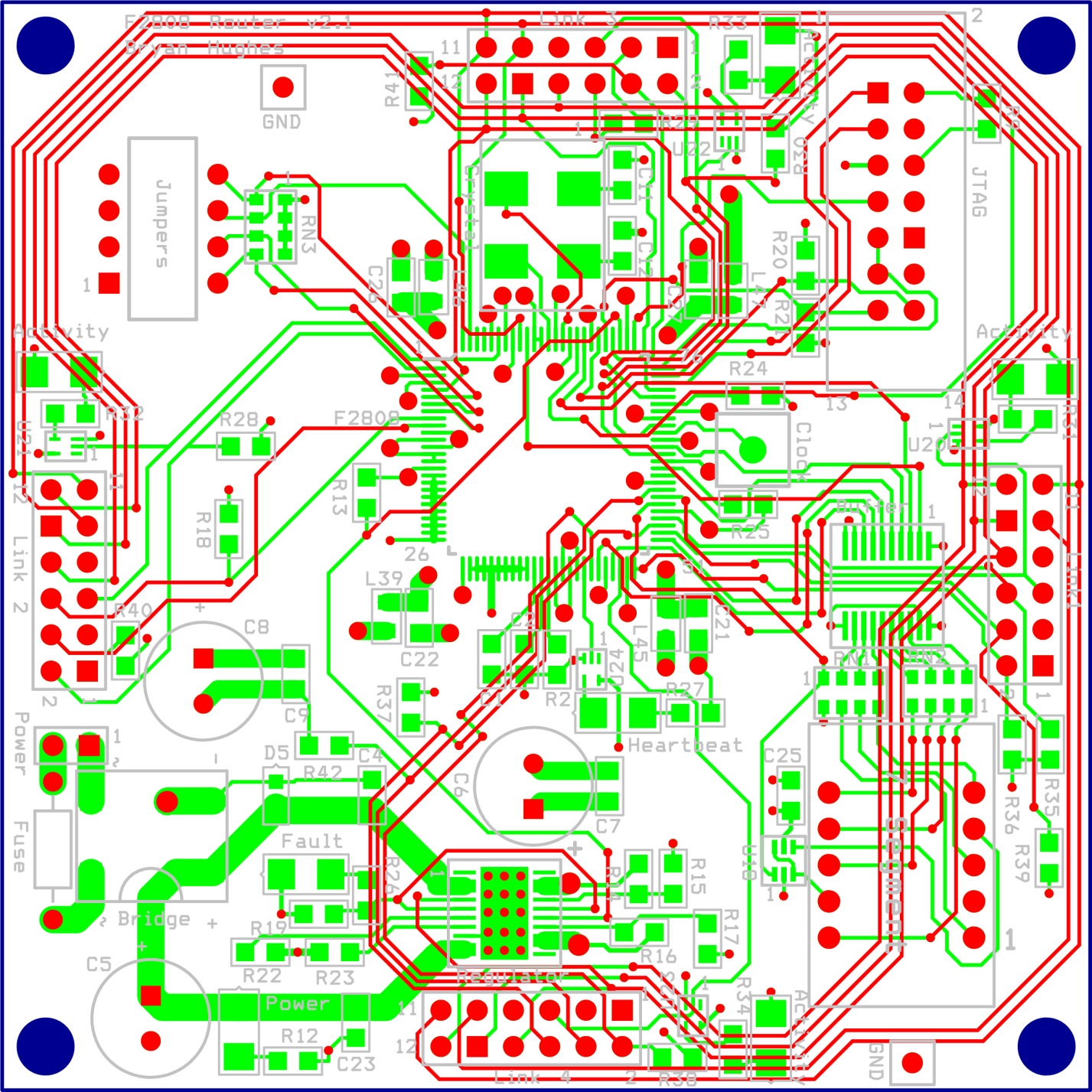


Figure x: Non-plane layers  
\label{fig:appendices:layout\_no\_power}

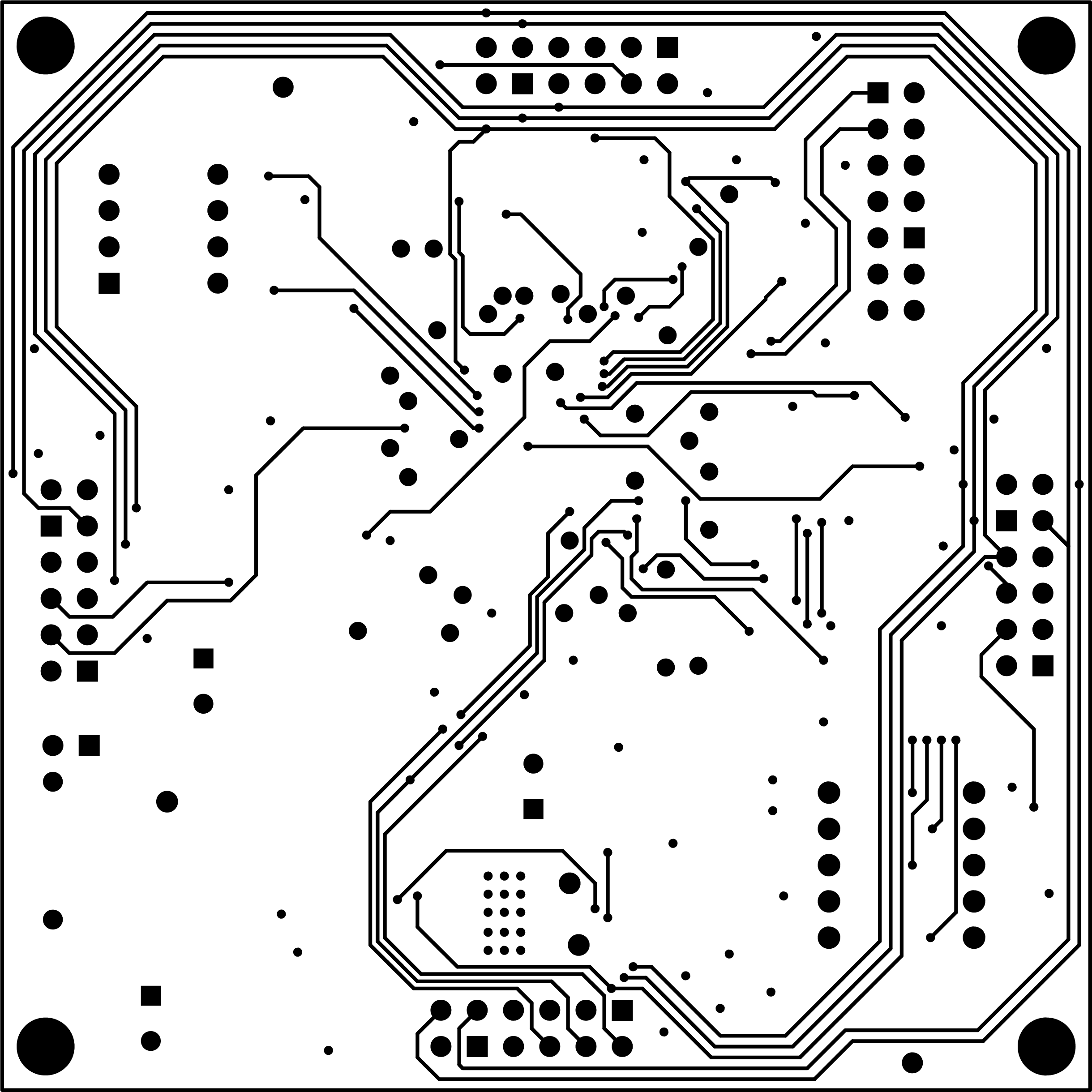


Figure x: Bottom layer  
\label{fig:appendices:layout\_bottom\_layer}

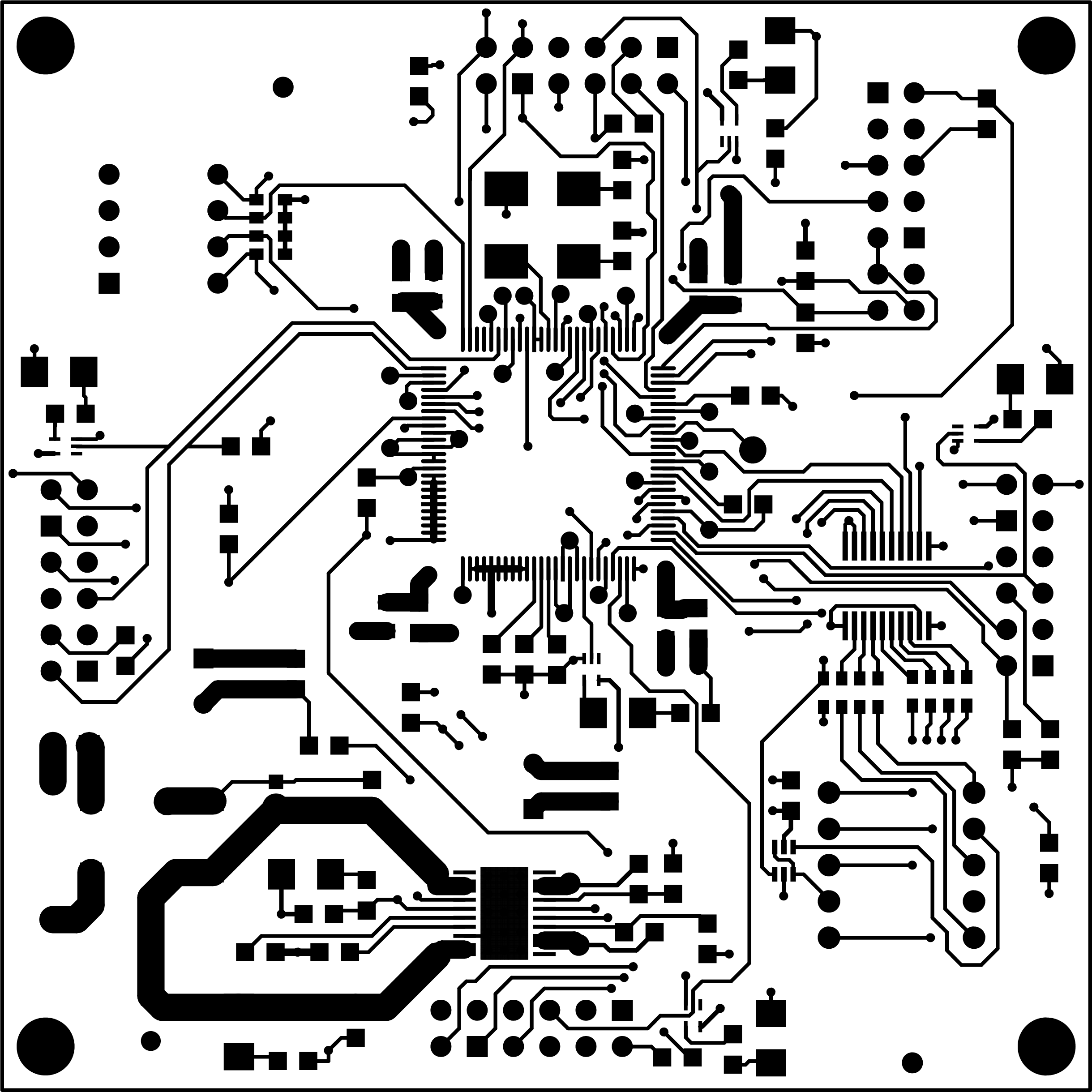


Figure x: Top layer  
\label{fig:appendices:layout\_top\_layer}

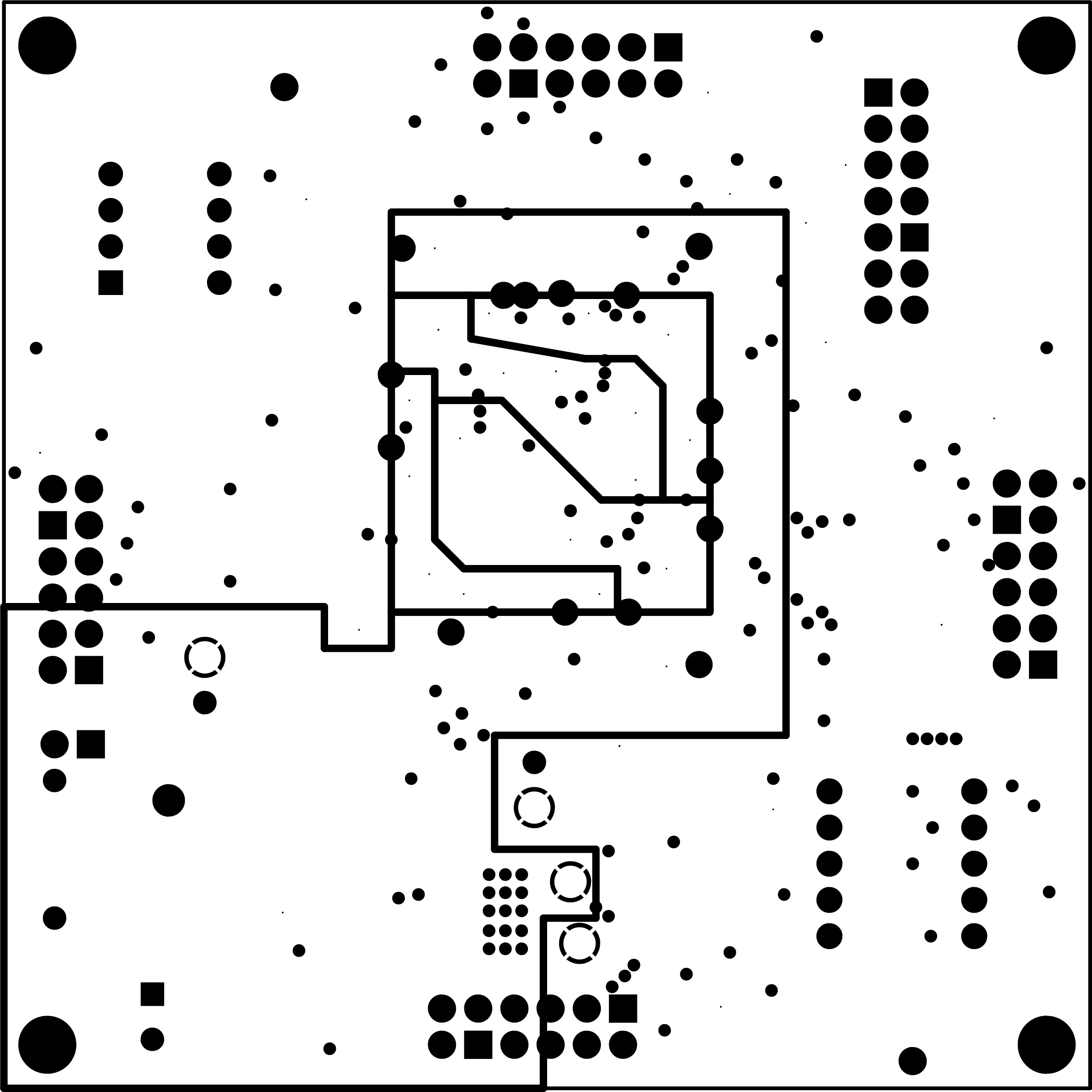


Figure x: Power plane  
\label{fig:appendices:layout\_power}

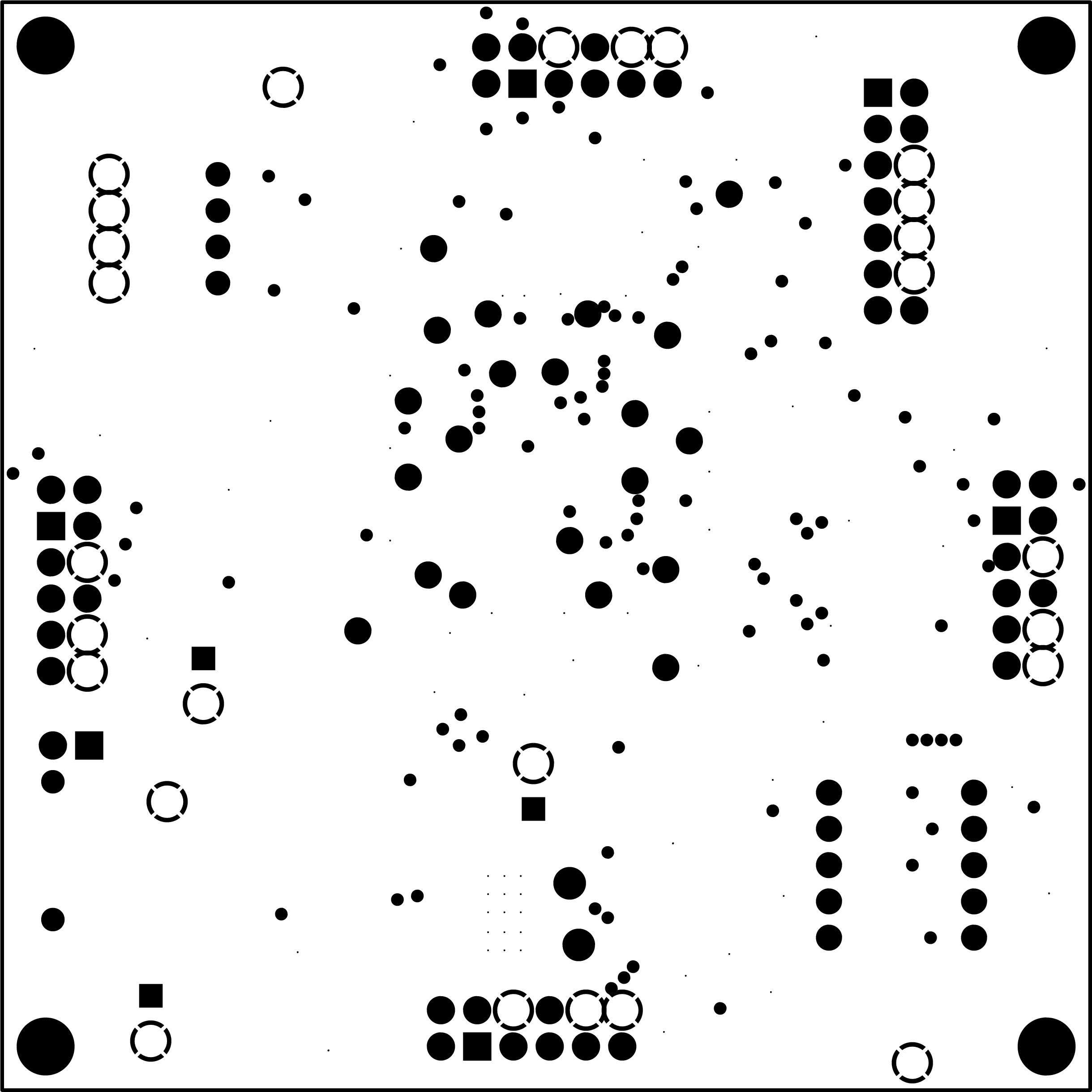


Figure x: Ground plane  
\label{fig:appendices:layout\_ground}

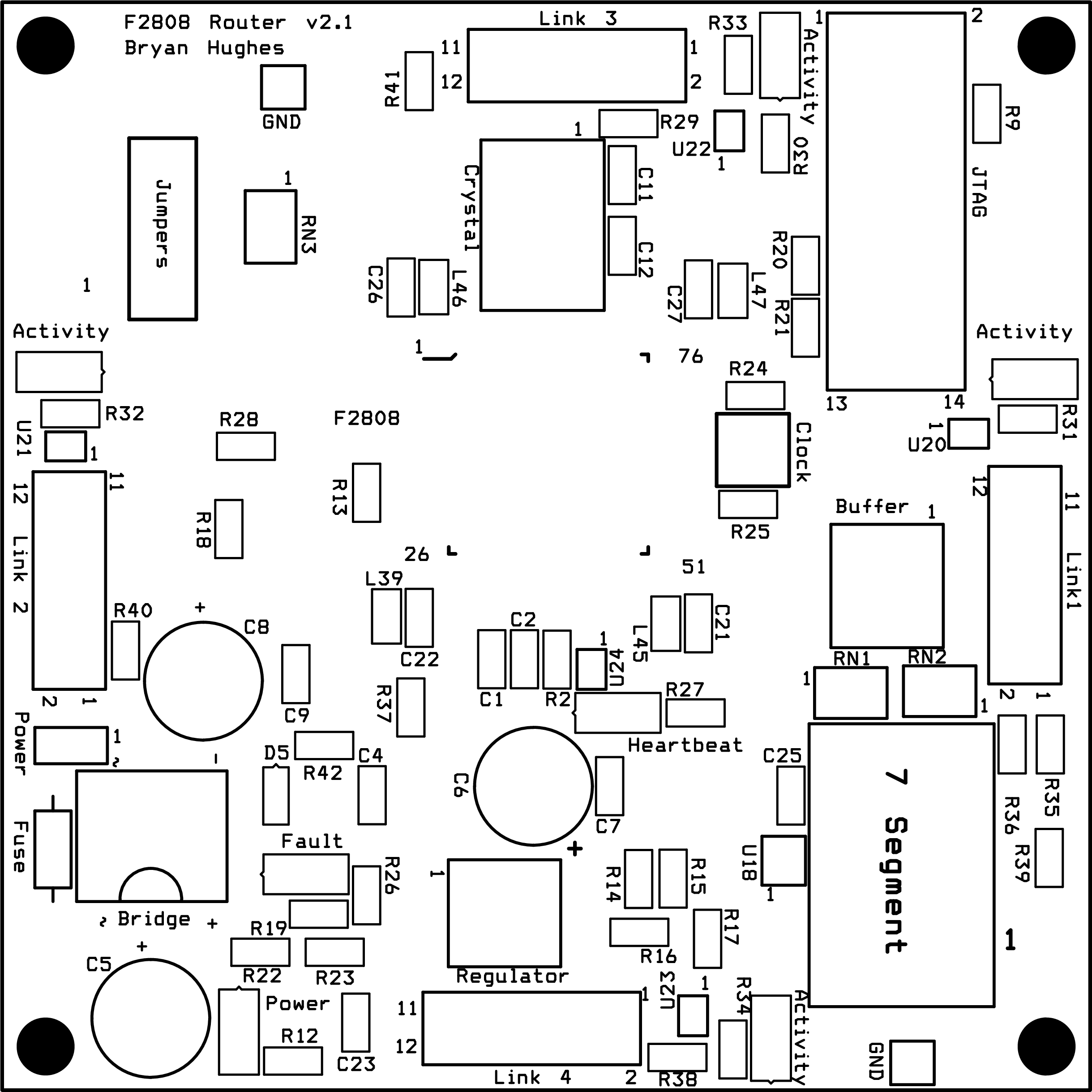


Figure x: Silkscreen layer  
\label{fig:appendices:layout\_silkscreen\_layer}

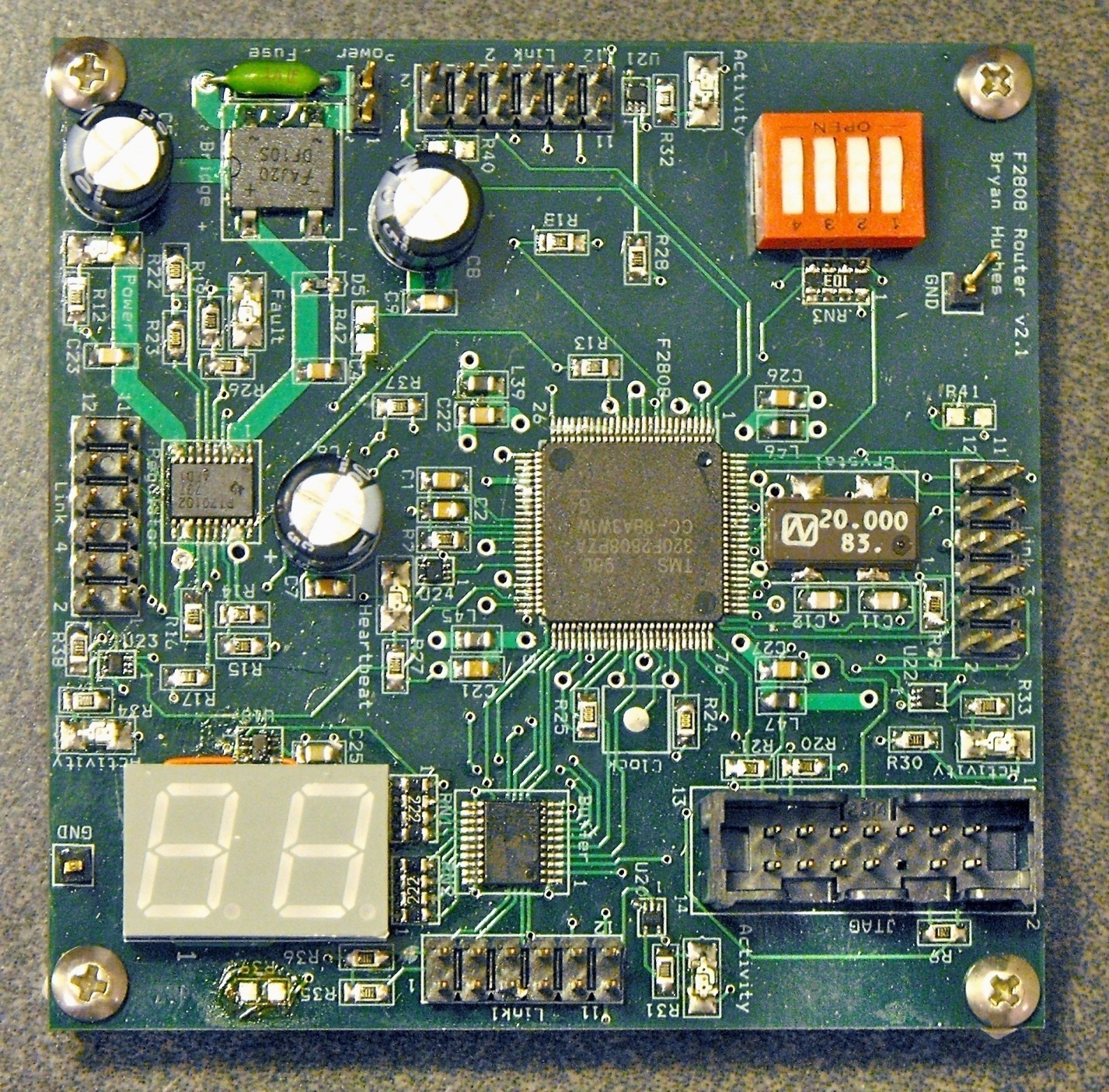


Figure x: Top of the board  
\label{fig:appendices:board\_top}

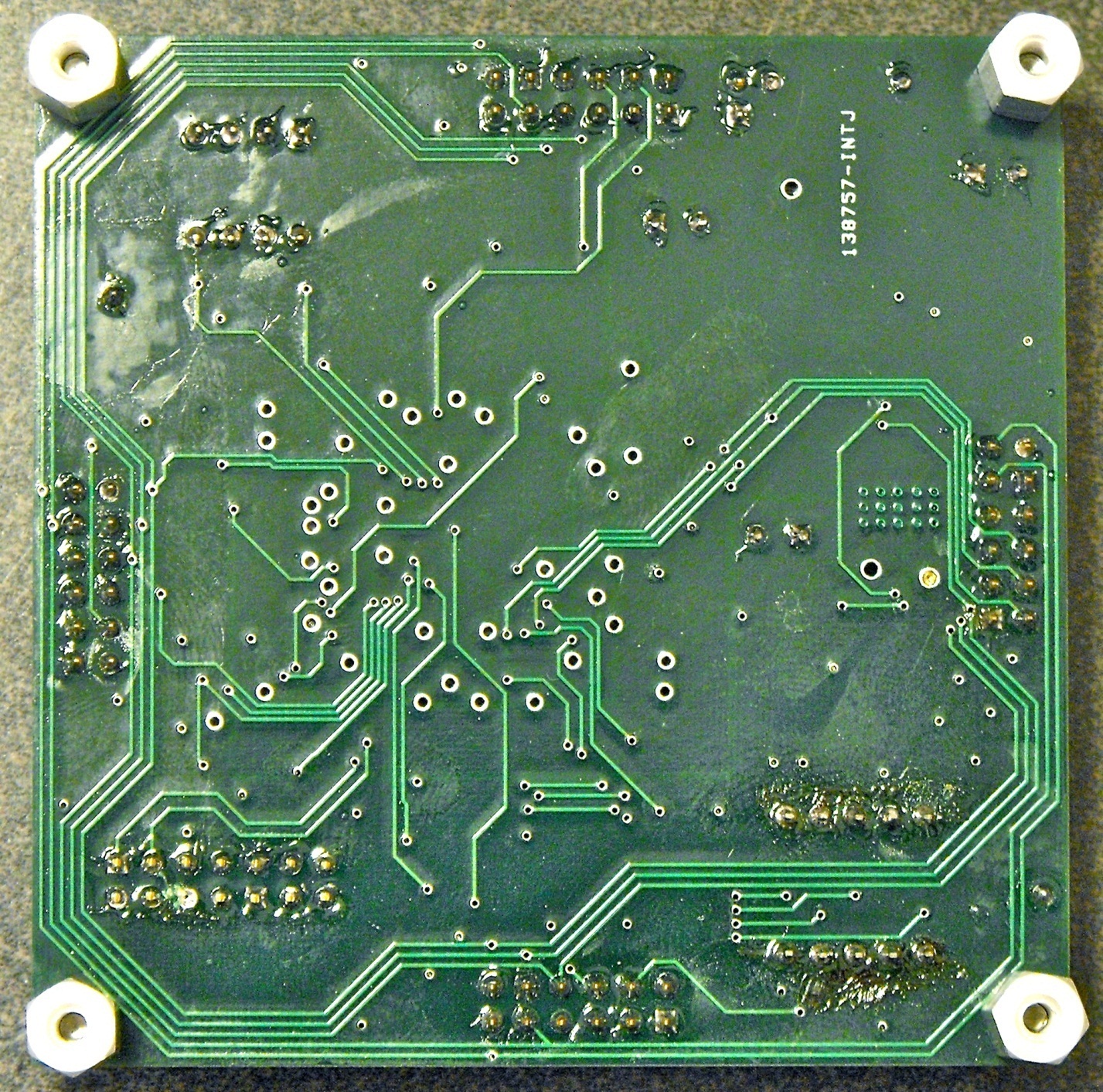


Figure x: Bottom of the board  
\label{fig:appendices:board\_bottom}

# Dynamic Memory Implementation \label{sec:appendices:dynamic\_memory}

Data transfers at the network and API level require the use of dynamic memory because messages can be of varying length. Texas Instruments provides a dynamic memory manager, but the version available for the project appeared to be dysfunctional. As an alternative, a custom memory manager was written that is described in this section.

The memory manager has a dedicated memory pool from which it allocates. The memory manager monitors allocations through a look-up table. The look-up table stores the base address and memory length of each allocation. Both allocation and de-allocation are supported. When an allocation is requested, the memory manager uses the lowest available memory address with enough free space, including in between previous allocations. This helps to combat fragmentation, but if there are enough allocations, fragmentation would become an issue. Fortunately, there are very few allocations in the toolkit, so fragmentation is not an issue.

# Follow-up Monitor \label{sec:appendices:follow\_up\_monitor}

Several parts of the toolkit require a follow-up action be executed at a later time, such as finding neighbors, data timeouts, etc. All of these cases use the ``follow-up monitor.'' The follow-up monitor provides capabilities for code to register a follow-up item to be run at a later date. A follow-up item consists of a callback function, an optional callback function argument, the follow-up rate, and whether this is a one shot follow-up, or should be called regularly. The follow-up monitor uses a single period function that manages the elapsed time of the follow-up items. Whenever it's time to run a follow-up item, the follow-up monitor calls the callback function. Follow-up items are added and (optionally) removed at run-time.

# Test Code \label{sec:appendices:test\_code}

The following code is shared between all nodes. A compile-time directive is defined to differentiate between the root and router. If the value IS\_ROUTER is defined, then the node is a non-root router. If the value IS\_ROOT is defined, then the node is the root. Both values cannot be defined at the same time. The test to run is defined by defining one of the compile-time directives TEST\_SPI, TEST\_PROTOCOL, or TEST\_MPI, which tests the physical/data link layer, network layer, and API respectively. Only one test can be defined at a time.

#include"System.h"

Uint16 receivedResponse[3] = {true,true,true};

Uint16 protocolTestPacketIDs[3];

#pragma CODE\_SECTION(TestService,"FlashCode");

void TestService(void)

{

// Wait for the system to finish initializing itself

SEM\_pend(&TestServiceSem,SYS\_FOREVER);

// Run the selected test

#if TEST == TEST\_SPI

SPITest();

#elif TEST == TEST\_PROTOCOL

//ProtocolTest1();

ProtocolTest2();

#elif TEST == TEST\_MPI

MPITest();

#endif

}

#pragma CODE\_SECTION(SPITest,"FlashCode");

void SPITest(void)

{

#if defined(IS\_ROUTER)

Uint16 flit[NUM\_CHARACTERS\_IN\_FLIT] = {

0xCA5F, 0xCA5F, 0xCA5F, 0xCA5F, 0xCA5F,

0xCA5F, 0xCA5F, 0xCA5F, 0xCA5F};

Uint32 numTransmitsInGroup, numGroups = 0, numTransmits = 0;

TSK\_sleep(2000);

while(numTransmits < 500000)

{

numTransmitsInGroup = 0;

while(numTransmitsInGroup < 5000)

{

if(gpioDataRegisters.GPBDAT.bit.SWITCH1)

{

while(globals.processing.outboundFlitQueFull[PORTA]);

EnqueOutboundFlit(flit,PORTA);

}

if(gpioDataRegisters.GPBDAT.bit.SWITCH2)

{

while(globals.processing.outboundFlitQueFull[PORTB]);

EnqueOutboundFlit(flit,PORTB);

}

if(gpioDataRegisters.GPADAT.bit.SWITCH3)

{

while(globals.processing.outboundFlitQueFull[PORTC]);

EnqueOutboundFlit(flit,PORTC);

}

if(gpioDataRegisters.GPADAT.bit.SWITCH4)

{

while(globals.processing.outboundFlitQueFull[PORTD]);

EnqueOutboundFlit(flit,PORTD);

}

numTransmits++;

numTransmitsInGroup++;

}

numGroups++;

SetSevenSegmentDisplay(numGroups);

}

SEM\_pend(&TestServiceSem,SYS\_FOREVER);

#endif

}

#pragma CODE\_SECTION(ProtocolTest1,"FlashCode");

void ProtocolTest1(void)

{

#if defined(IS\_ROUTER)

#if PROFILE\_TEST\_SERVICE == true

double executionTime;

Uint32 startTime;

#endif

struct Packet newPacket;

Uint16 x[100], i, j;

for(i = 0; i < 100; i++)

x[i] = i;

TSK\_sleep(3000);

if(gpioDataRegisters.GPBDAT.bit.SWITCH1)

{

for(i = 0; i < 100; i++)

{

for(j = 0; j < 50; j++)

{

InitializePacket(&newPacket, PACKET\_ID\_UNDEFINED);

// Create the data transfer packet

newPacket.a.communicationType = COMM\_TYPE\_UNICAST;

newPacket.transmissionInfo.destination = ROOT\_ADDRESS;

newPacket.b.command = COMMAND\_DATA\_TRANSFER;

newPacket.b.packetSequenceStep = SEQUENCE\_DATA\_TRANSFER\_REQUEST\_TRANSFER;

// Set the data

newPacket.dataBuffer = x;

newPacket.dataBufferInfo.dataBufferLength = 100;

#if PROFILE\_TEST\_SERVICE == true

startTime = timer0Registers.TIM.all;

#endif

SendDataPacket(&newPacket);

while(globals.statistics.packet.numDataTransfersSucceeded + globals.statistics.packet.numDataTransfersExpired +

globals.statistics.packet.numDataTransfersFailed < globals.statistics.packet.numDataTransfersSetup);

#if PROFILE\_TEST\_SERVICE == true

executionTime = TimeDifference(startTime, timer0Registers.TIM.all);

globals.statistics.profiling.avgTestServiceTime =

(globals.statistics.profiling.avgTestServiceTime \* globals.statistics.profiling.numTestServiceSamples) + executionTime;

globals.statistics.profiling.numTestServiceSamples++;

globals.statistics.profiling.avgTestServiceTime /= globals.statistics.profiling.numTestServiceSamples;

if(executionTime < globals.statistics.profiling.minTestServiceTime)

globals.statistics.profiling.minTestServiceTime = executionTime;

else if(executionTime > globals.statistics.profiling.maxTestServiceTime)

globals.statistics.profiling.maxTestServiceTime = executionTime;

#endif

}

SetSevenSegmentDisplay(i);

}

}

#endif

}

#pragma CODE\_SECTION(ProtocolTest2,"FlashCode");

void ProtocolTest2(void)

{

#if defined(IS\_ROOT)

struct Packet newPacket;

Uint16 x[100], i, j;

#if PROFILE\_TEST\_SERVICE == true

double executionTime;

Uint32 startTime;

#endif

for(i = 0; i < 100; i++)

x[i] = i;

TSK\_sleep(3000);

for(i = 0; i < 100; i++)

{

for(j = 0; j < 50; j++)

{

InitializePacket(&newPacket, PACKET\_ID\_UNDEFINED);

// Create the data transfer packet

newPacket.a.communicationType = COMM\_TYPE\_UNICAST;

newPacket.transmissionInfo.destination = 2;

newPacket.b.command = COMMAND\_DATA\_TRANSFER;

newPacket.b.packetSequenceStep = SEQUENCE\_DATA\_TRANSFER\_REQUEST\_TRANSFER;

// Set the data

newPacket.dataBuffer = x;

newPacket.dataBufferInfo.dataBufferLength = 10;

SendDataPacket(&newPacket);

InitializePacket(&newPacket, PACKET\_ID\_UNDEFINED);

// Create the data transfer packet

newPacket.a.communicationType = COMM\_TYPE\_UNICAST;

newPacket.transmissionInfo.destination = 3;

newPacket.b.command = COMMAND\_DATA\_TRANSFER;

newPacket.b.packetSequenceStep = SEQUENCE\_DATA\_TRANSFER\_REQUEST\_TRANSFER;

// Set the data

newPacket.dataBuffer = x;

newPacket.dataBufferInfo.dataBufferLength = 10;

SendDataPacket(&newPacket);

InitializePacket(&newPacket, PACKET\_ID\_UNDEFINED);

// Create the data transfer packet

newPacket.a.communicationType = COMM\_TYPE\_UNICAST;

newPacket.transmissionInfo.destination = 4; /\*(gpioDataRegisters.GPBDAT.bit.SWITCH1 << 3) + (gpioDataRegisters.GPBDAT.bit.SWITCH2 << 2) +

(gpioDataRegisters.GPADAT.bit.SWITCH3 << 1) + (gpioDataRegisters.GPADAT.bit.SWITCH4);\*/

newPacket.b.command = COMMAND\_DATA\_TRANSFER;

newPacket.b.packetSequenceStep = SEQUENCE\_DATA\_TRANSFER\_REQUEST\_TRANSFER;

// Set the data

newPacket.dataBuffer = x;

newPacket.dataBufferInfo.dataBufferLength = 10;

#if PROFILE\_TEST\_SERVICE == true

startTime = timer0Registers.TIM.all;

#endif

#if PROFILE\_LATENCY == true

globals.statistics.profiling.latencyStartTime = timer0Registers.TIM.all;

#endif

SendDataPacket(&newPacket);

while(globals.statistics.packet.numDataTransfersSucceeded + globals.statistics.packet.numDataTransfersExpired +

globals.statistics.packet.numDataTransfersFailed < globals.statistics.packet.numDataTransfersSetup);

#if PROFILE\_TEST\_SERVICE == true

executionTime = TimeDifference(startTime, timer0Registers.TIM.all);

globals.statistics.profiling.avgTestServiceTime =

(globals.statistics.profiling.avgTestServiceTime \* globals.statistics.profiling.numTestServiceSamples) + executionTime;

globals.statistics.profiling.numTestServiceSamples++;

globals.statistics.profiling.avgTestServiceTime /= globals.statistics.profiling.numTestServiceSamples;

if(executionTime < globals.statistics.profiling.minTestServiceTime)

globals.statistics.profiling.minTestServiceTime = executionTime;

else if(executionTime > globals.statistics.profiling.maxTestServiceTime)

globals.statistics.profiling.maxTestServiceTime = executionTime;

#endif

}

SetSevenSegmentDisplay(i);

}

#endif

}

#ifdef \_\_cplusplus

#pragma DATA\_SECTION("MpiData")

#else

#pragma DATA\_SECTION(xglobal,"MpiData");

#endif

int xglobal[700];

#ifdef \_\_cplusplus

#pragma DATA\_SECTION("MpiData")

#else

#pragma DATA\_SECTION(yglobal,"MpiData");

#endif

int yglobal[700];

#ifdef \_\_cplusplus

#pragma DATA\_SECTION("MpiData")

#else

#pragma DATA\_SECTION(resultglobal,"MpiData");

#endif

int resultglobal[1399];

#pragma CODE\_SECTION(MPITest,"FlashCode");

void MPITest(void)

{

int rank, numNodes;

Uint16 sendBuf[1], \*receiveBuf;

Uint16 i, j;

MPI\_Status status;

Uint32 startTime;

double elapsedTime;

SEM\_pend(&TestServiceSem,SYS\_FOREVER);

// Generate the x data

for(i = 0; i < 14; i++)

{

for(j = 0; j < 25; j++)

xglobal[i \* 50 + j] = (j / 4);

for(j = 0; j < 25; j++)

xglobal[i \* 50 + j + 25] = 6 - (j / 4);

}

// Generate the y data

for(i = 0; i < 28; i++)

{

for(j = 0; j < 25; j++)

yglobal[i \* 25 + j] = j / 2;

}

// Initialize MPI

MPI\_Init(NULL,NULL);

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

globals.processing.sevenSegmentUpperDigit = SEVENSEG\_1DASH;

// Distribute the number of nodes to use

if(rank == 0)

{

numNodes = (gpioDataRegisters.GPBDAT.bit.SWITCH1 << 3) + (gpioDataRegisters.GPBDAT.bit.SWITCH2 << 2) +

(gpioDataRegisters.GPADAT.bit.SWITCH3 << 1) + (gpioDataRegisters.GPADAT.bit.SWITCH4);

sendBuf[0] = numNodes;

for(i = 1; i <= 6; i++)

MPI\_Send(sendBuf,1,MPI\_SHORT,i,0,MPI\_COMM\_WORLD);

}

else

{

MPI\_Recv((void\*\*)(&receiveBuf),1,MPI\_SHORT,0,0,MPI\_COMM\_WORLD,&status);

numNodes = receiveBuf[0];

MemFree(receiveBuf);

}

MPI\_Barrier(MPI\_COMM\_WORLD);

// If this node is participating in the convolution

if(rank < numNodes)

{

globals.processing.sevenSegmentUpperDigit = SEVENSEG\_2DASH;

startTime = timer0Registers.TIM.all;

Convolution(xglobal,yglobal,100,100,resultglobal,numNodes);

elapsedTime = TimeDifference(startTime, timer0Registers.TIM.all);

// Finalize MPI

if(rank == 0)

{

globals.processing.sevenSegmentLowerDigit = SEVENSEG\_FINAL;

globals.processing.sevenSegmentUpperDigit = SEVENSEG\_FINAL;

}

else

globals.processing.sevenSegmentUpperDigit = SEVENSEG\_3DASH;

}

else

{

globals.processing.sevenSegmentUpperDigit = SEVENSEG\_FINAL;

}

SEM\_pend(&TestServiceSem,SYS\_FOREVER);

}

#pragma CODE\_SECTION(Convolution,"FlashCode");

void Convolution(int\* x, int\* y, int xLength, int yLength, int\* result, int numNodesToUse)

{

int n, m, resultLength = xLength + yLength - 1, rank, transfersIncomplete, flag;

int \*nodeResults[MAX\_NUM\_NODES];

int receivedResults[MAX\_NUM\_NODES] = {false,false,false,false,false,false,false,false,false};

MPI\_Status status;

MPI\_Request recvRequests[MAX\_NUM\_NODES];

// Initialize data

MPI\_Comm\_rank(MPI\_COMM\_WORLD,&rank);

for(n = 0; n < resultLength; n++)

result[n] = 0;

// x[n] conv y[n] = sum(x[m] \* y[n-m] over m)

if(rank == 0)

{

// Convolve the signals

for(n = 0; n < resultLength / numNodesToUse; n++)

{

result[n] = 0;

for(m = n - yLength; m < n + yLength; m++)

{

if((m >= 0 && m < xLength) && (n - m >= 0 && n - m < yLength))

result[n] += x[m] \* y[n - m];

}

}

}

else

{

// Convolve the signals

for(n = 0; n < (rank \* resultLength) / numNodesToUse; n++)

result[n] = 0;

for(n = (rank \* resultLength) / numNodesToUse; n < ((rank + 1) \* resultLength) / numNodesToUse; n++)

{

result[n] = 0;

for(m = n - yLength; m < n + yLength; m++)

{

if((m >= 0 && m < xLength) && (n - m >= 0 && n - m < yLength))

result[n] += x[m] \* y[n - m];

}

}

for(n = ((rank + 1) \* resultLength) / numNodesToUse; n < resultLength; n++)

result[n] = 0;

}

if(numNodesToUse > 1)

{

// Collect the data

if(rank == 0)

{

// Initiate the receive requests

for(n = 1; n < numNodesToUse; n++)

MPI\_Irecv((void\*\*)(&(nodeResults[n])),resultLength,MPI\_SHORT,n,0,MPI\_COMM\_WORLD,&(recvRequests[n]));

// Test the status of the receives

transfersIncomplete = true;

while(transfersIncomplete)

{

transfersIncomplete = false;

for(n = 1; n < numNodesToUse; n++)

{

if(!receivedResults[n])

{

// Test the status of the receive from node of rank n

MPI\_Test(&(recvRequests[n]),&flag,&status);

// If the transfer is complete, save the results

if(flag)

{

// Save the results

receivedResults[n] = true;

for(m = (n \* resultLength) / numNodesToUse; m < ((n + 1) \* resultLength) / numNodesToUse; m++)

result[m] = nodeResults[n][m - (n \* resultLength) / numNodesToUse];

MemFree((Uint16\*)(nodeResults[n]));

}

else

transfersIncomplete = true;

}

}

}

}

else

{

MPI\_Send(&(result[(rank \* resultLength) / numNodesToUse]),

((rank + 1) \* resultLength) / numNodesToUse - (rank \* resultLength) / numNodesToUse,

MPI\_SHORT,0,0,MPI\_COMM\_WORLD);

}

}

}

# Expanded Convolution Results \label{sec:appendices:convolution}

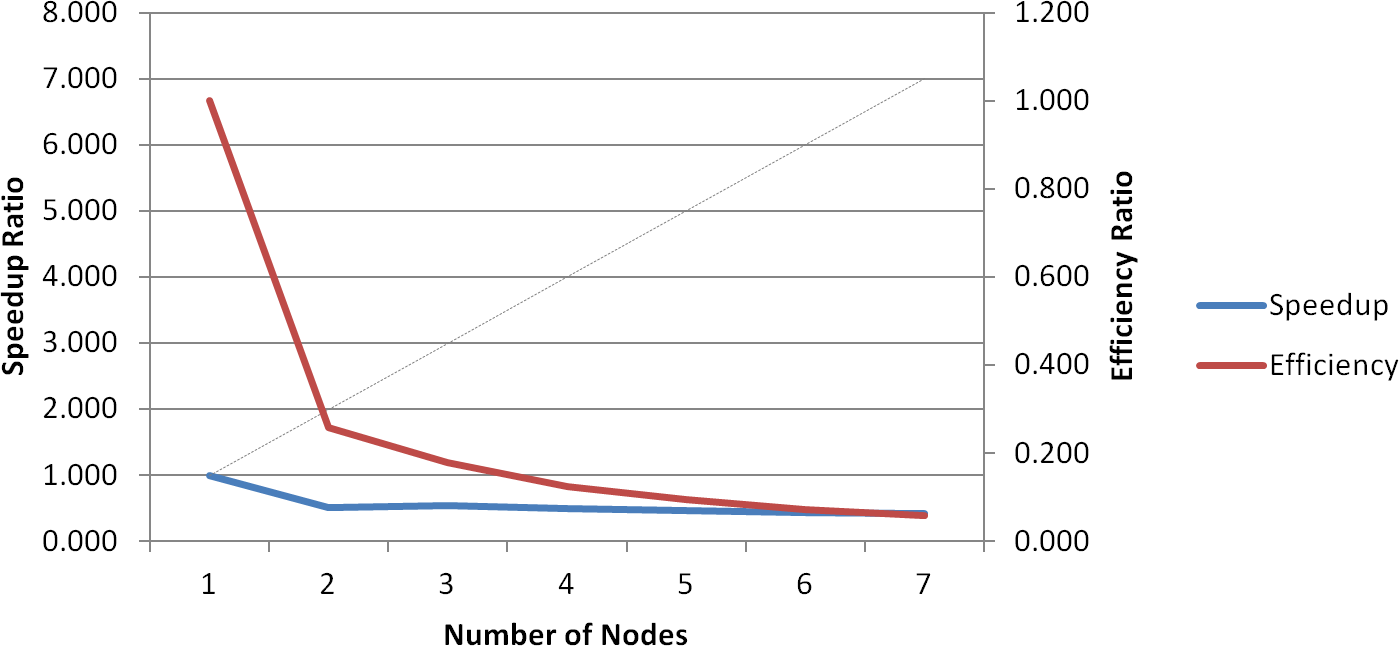


Figure x: Speedup and efficiency versus number of nodes (100 byte data payload)  
\label{fig:appendices:convolution\_results\_100}

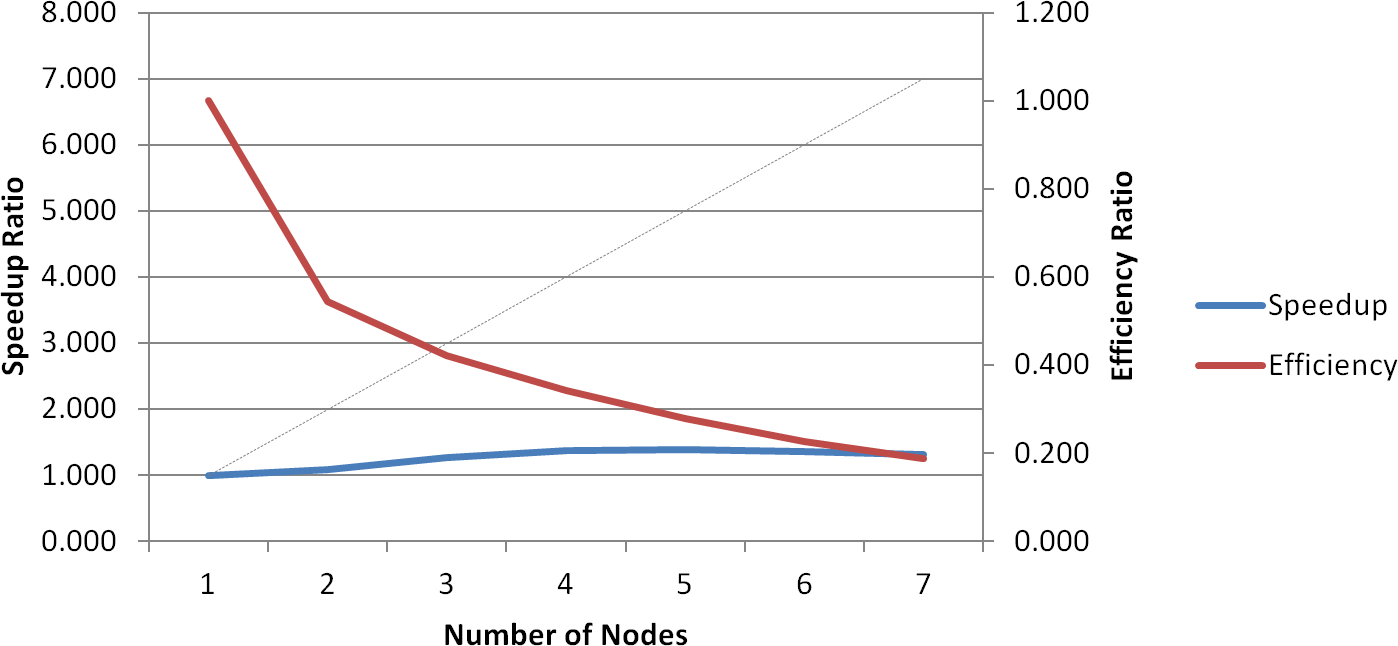


Figure x: Speedup and efficiency versus number of nodes (200 byte data payload)  
\label{fig:appendices:convolution\_results\_200}

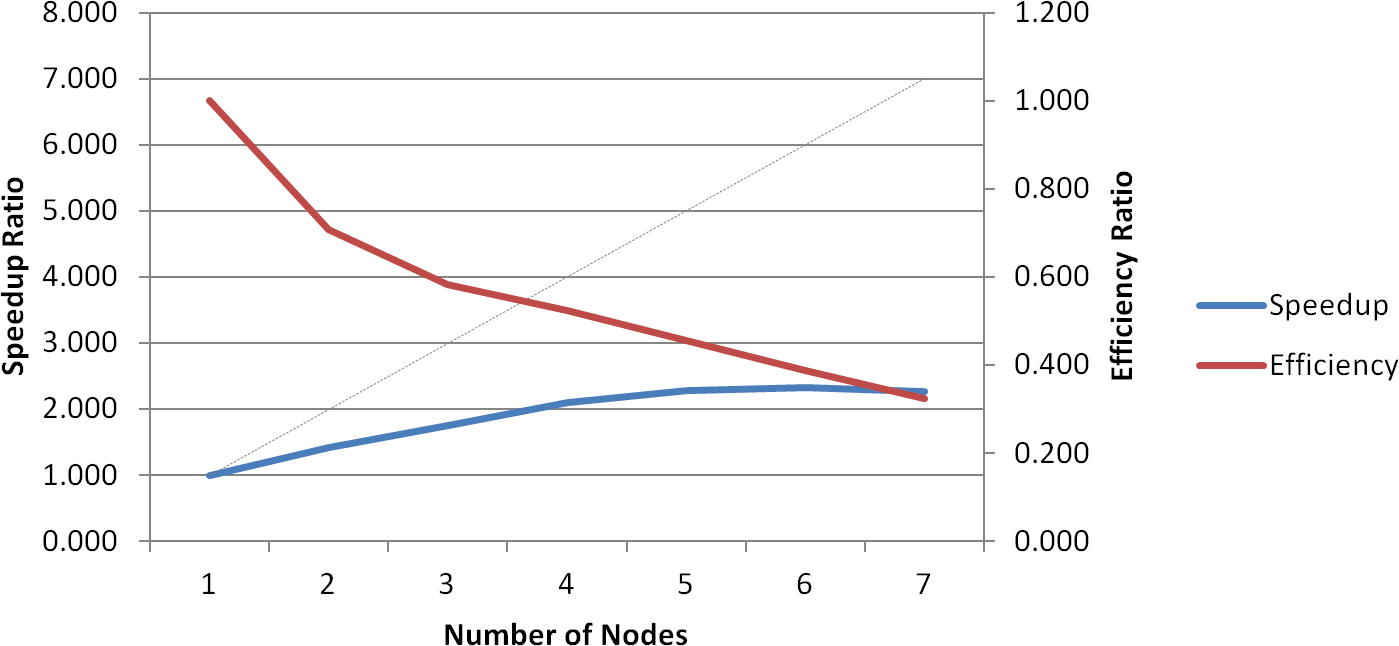


Figure x: Speedup and efficiency versus number of nodes (300 byte data payload)  
\label{fig:appendices:convolution\_results\_300}

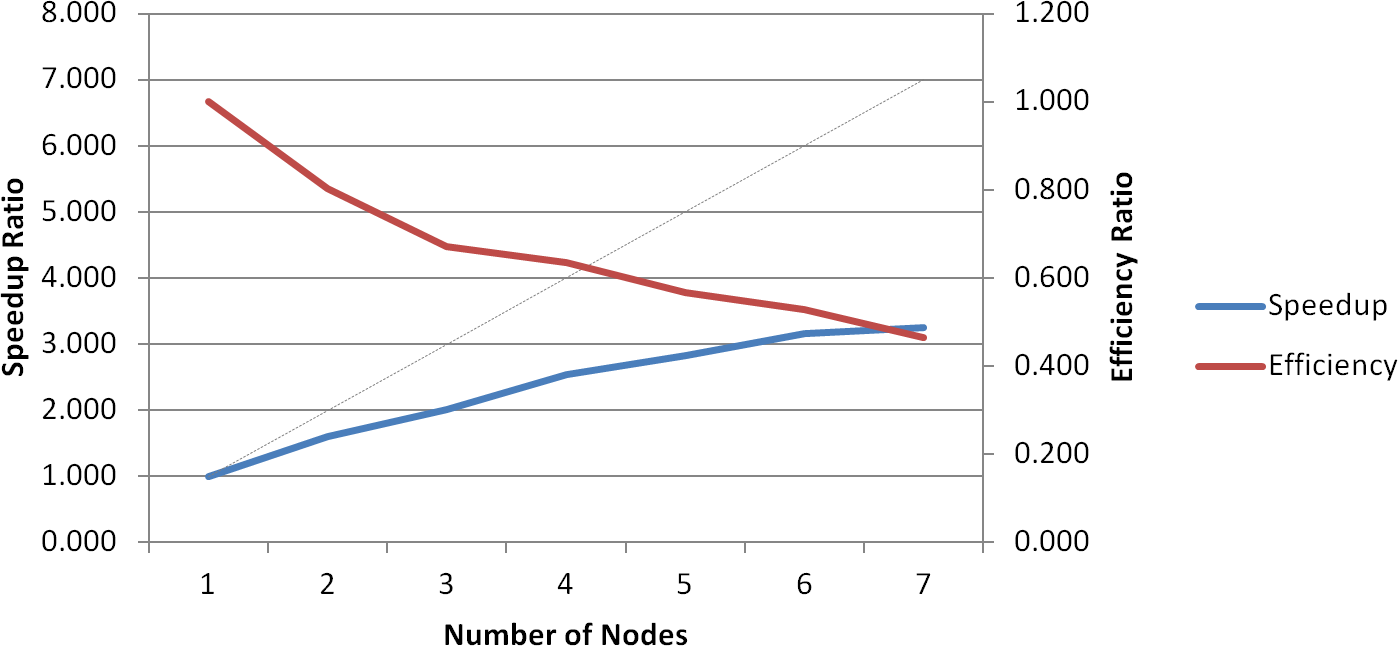


Figure x: Speedup and efficiency versus number of nodes (400 byte data payload)  
\label{fig:appendices:convolution\_results\_400}

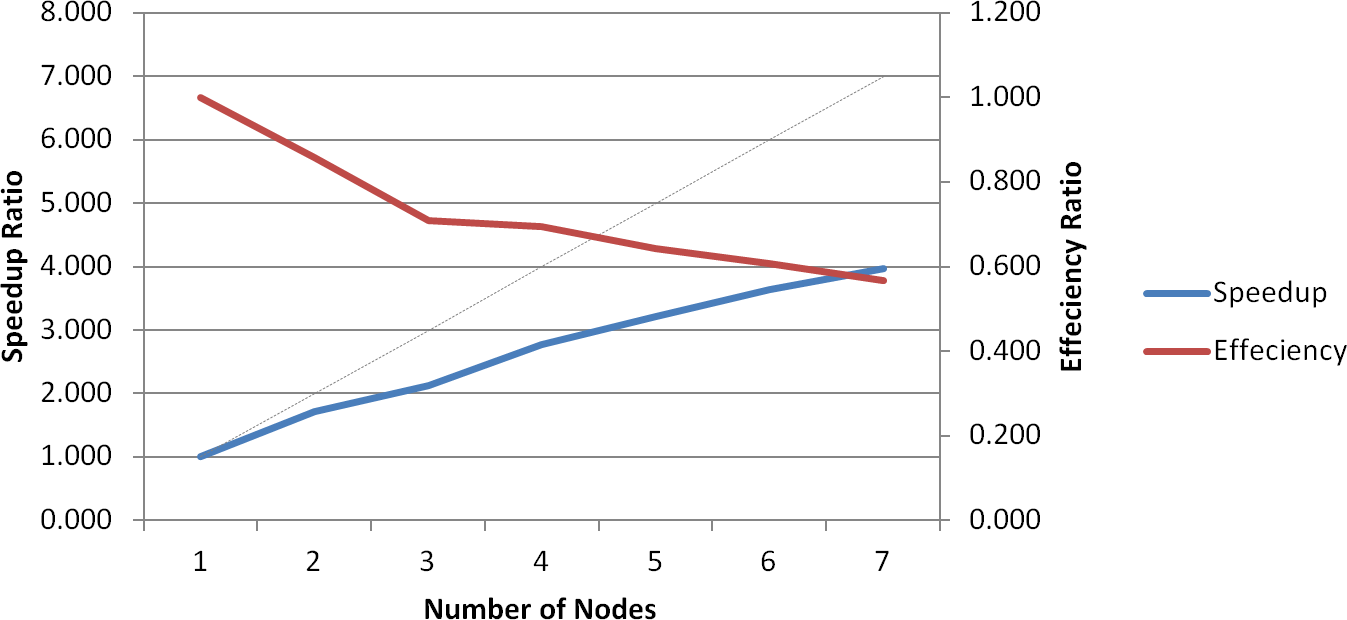


Figure x: Speedup and efficiency versus number of nodes (500 byte data payload)  
\label{fig:appendices:convolution\_results\_500}

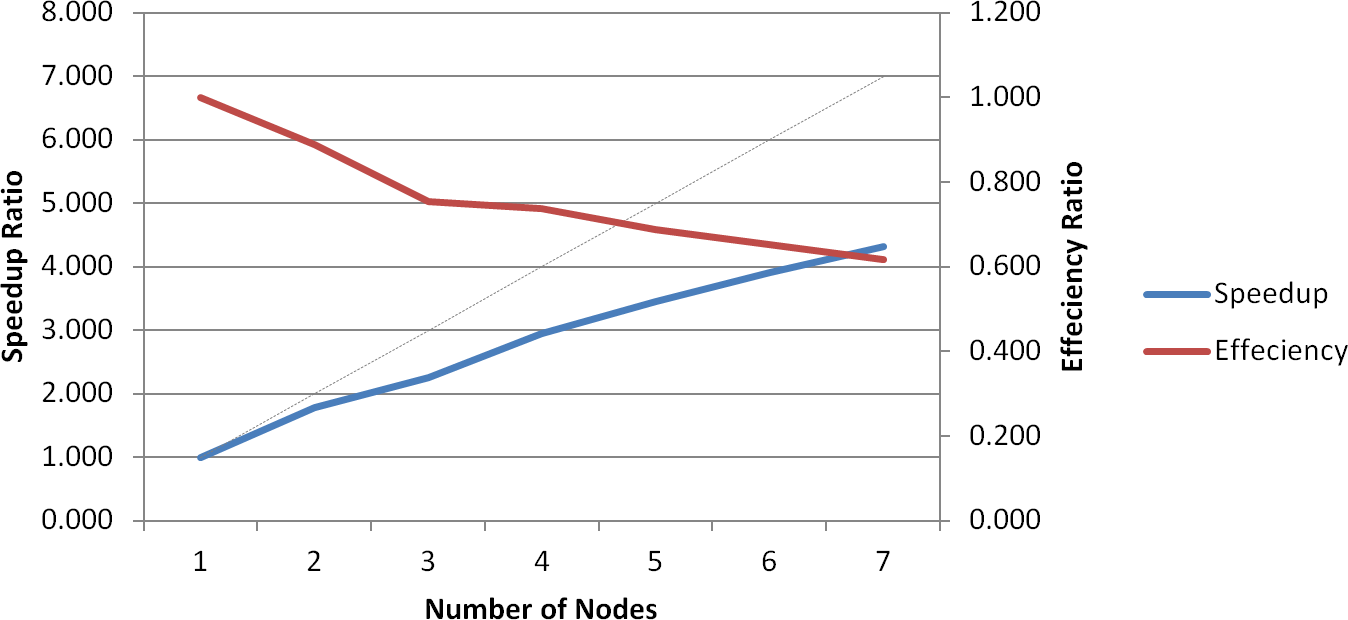


Figure x: Speedup and efficiency versus number of nodes (600 byte data payload)  
\label{fig:appendices:convolution\_results\_600}

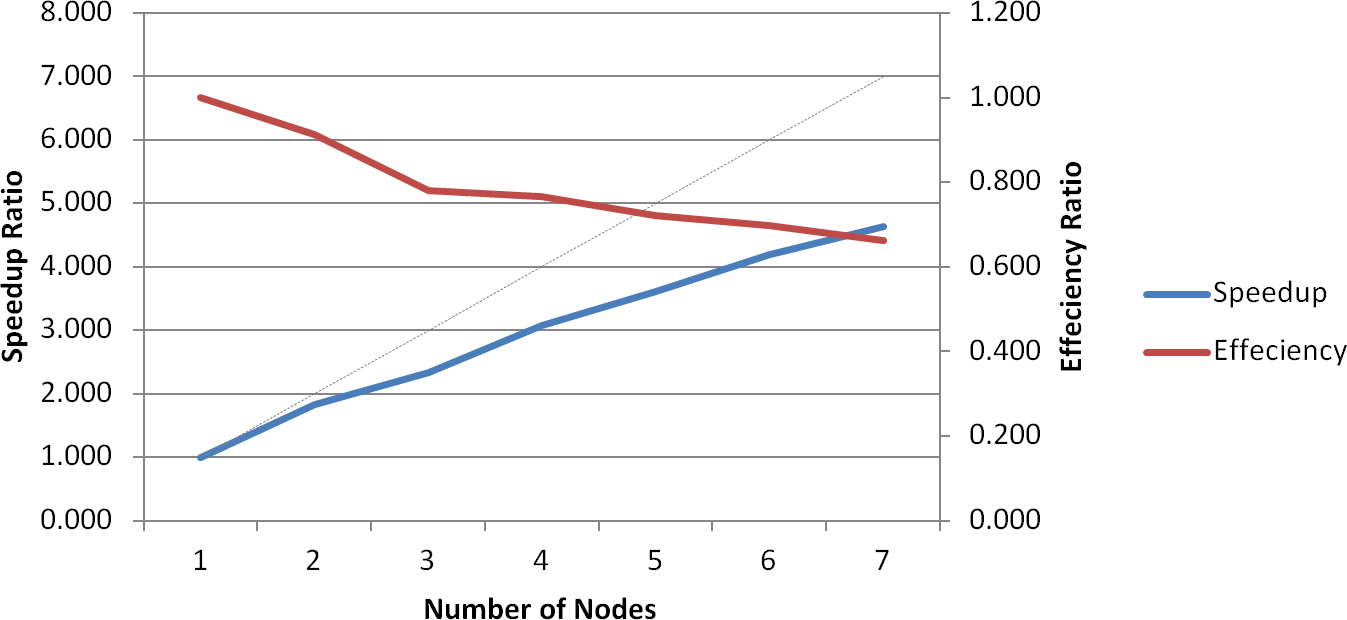


Figure x: Speedup and efficiency versus number of nodes (700 byte data payload)  
\label{fig:appendices:convolution\_results\_700}

1. Boards are 3'' by 3'' [↑](#footnote-ref-1)