Spiking neural network simulator

Xin Jin (jinxa@cs.man.ac.uk)
Supervisor: Steve Furber

1. Overview and Background

This project is a part of an EPSRC supported project - SpiNNaker - with the aim to build a chip multiprocessor (CMP) combined with its external SDRAM for real-time neural network simulation. Each chip in the SpiNNaker project contains 20 identical ARM968 processing subsystems each of which is called a fascicle processor and is responsible for modeling a number of neurons with associated inputs and outputs. The processors on a CMP share access to the SDRAM using a self-timed packet-switched Network-on-Chip (NoC) [1][4]. Neurons modeled by fascicle processors connect to and receive inputs from others in the same or different fascicles by a certain rule in terms of the specific algorithm adopted, to model a spiking neural network.

This project mainly looks at implementing the spiking neural network model and its algorithm developed by Izhikevich [2], on ARM968 system, in order to mimic pulse-coupled neural networks mechanism of information processing by the brain. The Izhikevich's model can reproduce a rich number of neuro-computational features with comparatively efficient implementation. In his model, neurons are receiving synaptic weights as spiking signals from the fired neurons connected, and then update their states by the Izhikevich Equation in terms of the synaptic weights received. This progress will repeat in real-time with a certain frequency. As a result, a neuro-computational pattern for each neuron will be generated. Similarly, in this project, we will simulate the progress above on the event driven model used in SpiNNaker and will finally generate very closed patterns as Izhikevich gave in his paper[3]. This allows us to evaluate the performance and efficiency as well as the feasibility of adopting a specific neuron updating algorithm in such a system (a ARM968 system with event driven model). The result of the simulation can be used as a reference for the SpiNNaker hardware design and may also be useful for carrying out further research on spiking neural network. We will develop a set of codes running on the ARM968 simulator to process neuron inputs and output, neuron state update and so on. These codes can simulate the activities of a small-scale neural network (about 1000 neurons) on one ARM968 processing subsystem.

The principle of such a spiking neural network system we are using on ARM968 is illustrated in Figure 1.1. All of the information for neuron connections, each of which is 4 bytes and is called a synaptic word, is stored in the external SDRAM. Each synaptic word is organized in a format like below:

|--4b-delay-|0|-11b-index-|-16b-weight-|.

Where "4b delay" represents the delay time (ms) of the input, "11b index" is the unique neuron index number in a fascicle, "16b weight" is the essential value which should be added to the right inputs buffer found in terms of "4b delay" in the correct neuron found by the "11b index".

Each neuron has its own data structure:

```
Neuron [0..n-1]{
    struct NeuronState;
    short bin [0..15];
}
```

It comprises the parameters of a neuron and 16 signed short neuron input buffers. The input buffers are actually utilized for implementing the time delay.

Every neuron will update its state every millisecond according to the algorithm we are using. If any of the neurons in the system fires, a 32-bits routing key (the neuron's unique identifying number), comprising its fascicle index number and an offset of the fired neuron in this fascicle, will be sent out. When it arrives as an input at one of the fascicles which have any of the neurons connecting to this fired neuron, that fascicle will start to process it. In the example displayed in Figure 1.1, the fascicle number of this input is X, and the offset is 0x30. The fascicle will look up the mapping table kept in its I-TCM to find the memory block in the SDRAM corresponding to this input, and then add the offset 0x30 to the address. So the result of the start memory address of the block will be 0x100430. Supposing the block size is 48 bytes comprising 12 synaptic words (indicating that 12 neurons in this fascicle connect to the fired neuron), the memory space of 0x100430 – 0x100460 will be the memory block we want.

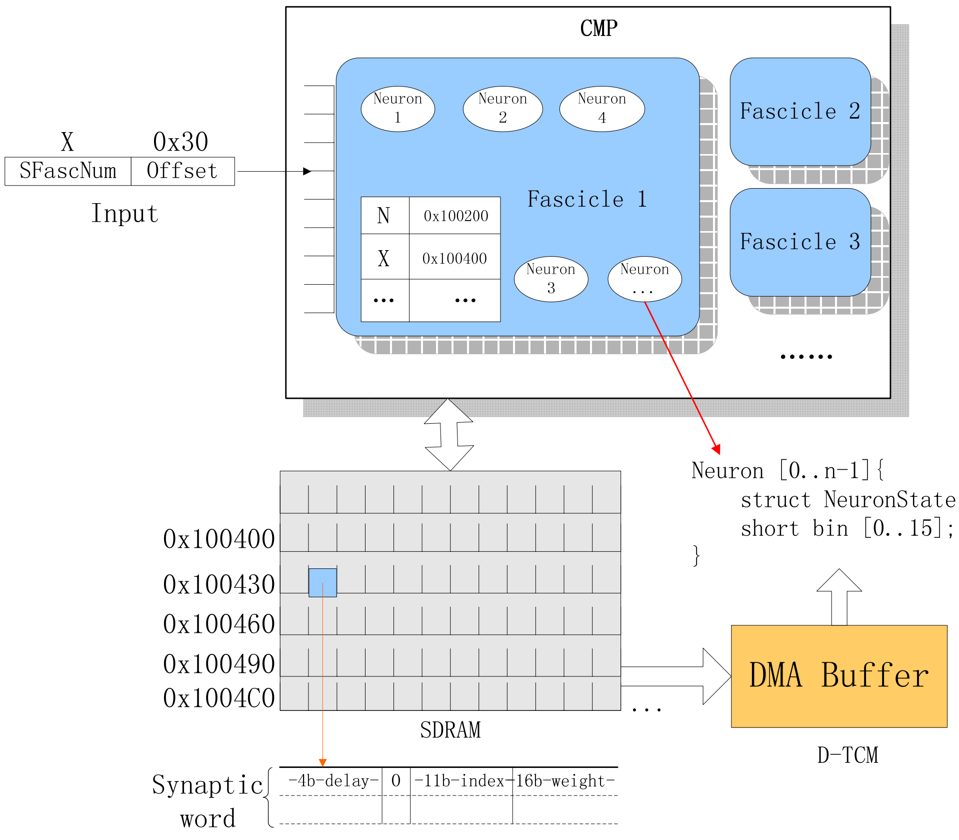

Figure 1.1 Principle of the system

After that, a DMA transfer is started to copy this memory block from the SDRAM to the DMA buffer located in the D-TCM. After the DMA operation completes, another routine will start to update the input buffer of each neuron according to the content of each synaptic word in the DMA buffer.

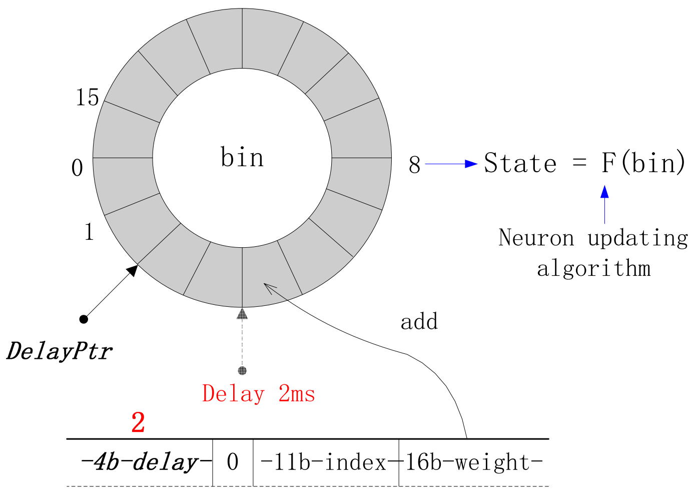

Figure 1.2 Input buffers

As illustrated in Figure 1.2, each neuron has a 16 half-words (32-byte) input buffer ring holding the "short bin [0...15]" array in its data structure. A serial of digital numbers indicating the strength of the firing signals (electrical current "I") is stored in this array in the order of the time they fired. There is a time relative pointer "DelayPtr" pointing to the current position in the buffer ring which indicates the current system time. If the value of "4b-delay" is 2 (ms), the "16b weight" will be added to the buffer ring 2 half words after the "DelayPtr". Within every one millisecond the neuron state will be updated using the current "I" (one half-word) loaded from the position of "DelayPtr" under the rule of the algorithm we are using in the system.

2. Approach of Implementation

2.1 Scheduler

There are three main tasks in each processor, with priority levels 1, 2, 3 in order as below:

1. Update neuron bin, which is to update neuron inputs buffers according to the weights in the synaptic words.
2. New input processing, which is responsible for processing all inputs from the neurons fired, finding the correct synaptic weight address in the SDRAM and starting DMA to copy them into the D-TCM buffers.
3. Update neuron states, which is to update the states of the neurons in one fascicle using the neuron updating algorithm (the Izhikevich Equation[2] in this context).

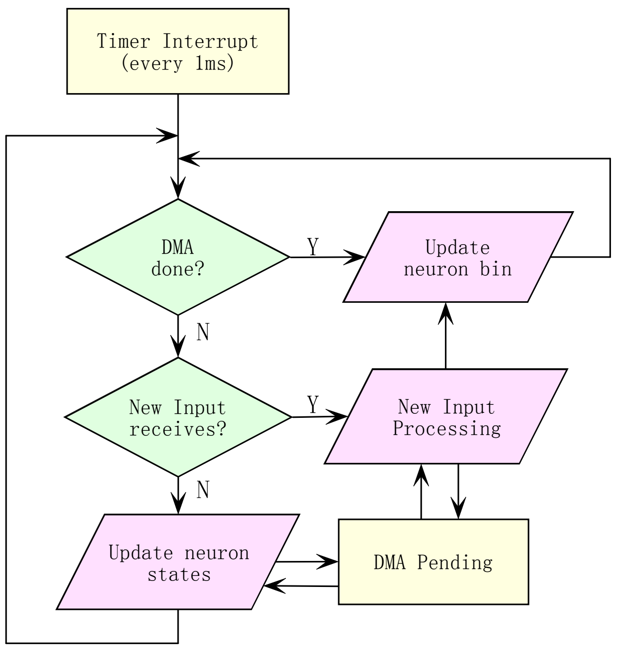

Figure 2.1 System Scheduler

Figure 2.1 indicates the approach of the system scheduling in each processor (we still haven't finally decide how to schedule the tasks. This is the scheduler we are currently using in this example). As you can see, whenever the processor has completed a task it goes back to the beginning to see what to do next.

A timer will generate an interrupt every millisecond to indicate the start of the next time unit. If any neuron or input runs out of time in the previous millisecond, the processor will continue processing it before the next round is allowed to start. Figure 2.2 illustrates the strategy to solve the "time out".

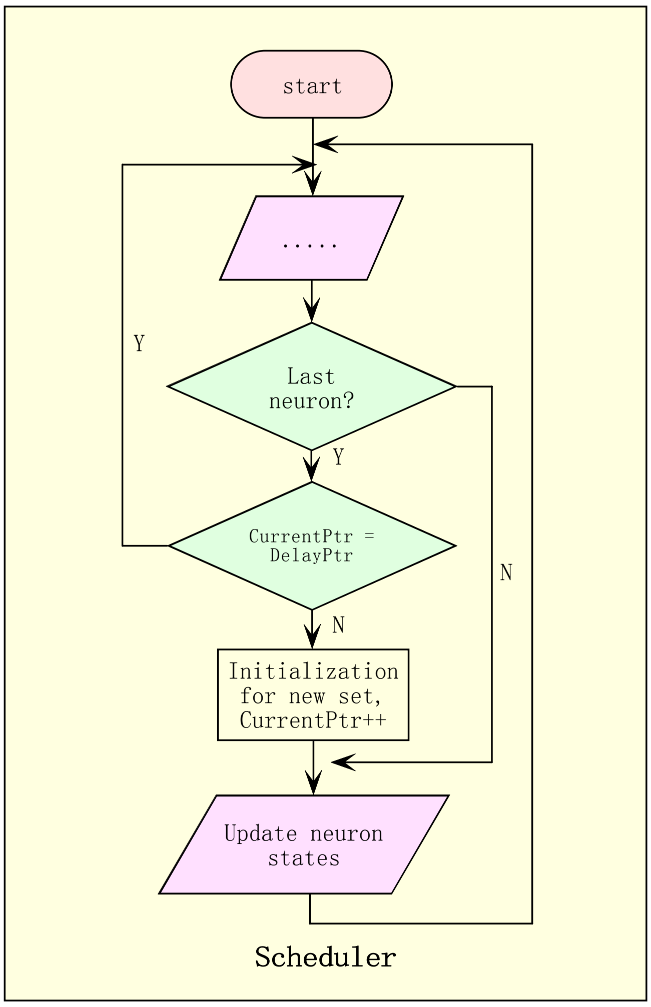

Figure 2.2 Strategy for time out

“Timer Interrupt” is a timer interrupt service routine which will be executed every millisecond mainly accumulating the variable “DelayPtr” as well as doing some other jobs. As indicated in Figure 2.2, the system will keep on processing the inputs and updating the neurons left if they run out of time in one millisecond. The new round will only start after all of the previous jobs have been finished. When all of the neurons have been updated, the processor will compare the value CurrentPtr to DelayPtr. If not equal, which means the system is time out, it starts a new round. Otherwise, it will go to process the inputs and wait for the next timer interrupt.

As in Figure 2.1, the “New Input Processing” task may branch to the “Update neuron bin” task without going back to the scheduler. So there are two entries to access the “Update neuron bin” task. Supposing we have two DMA buffers: Buffer A and Buffer B, the DMA status is checked immediately in the “New Input Processing” task after a DMA transfer is started, say, from the SDRAM to the Buffer A. If any DMA complete signal is detected, which means Buffer B has just been filled, it will branch to the “Update neuron bin” task and pass the address of Buffer B. Otherwise, it will return to the scheduler and check the DMA status again, if any DMA complete signal detected at this point, it branches to “Update neuron bin” task and passes the address of the Buffer A..

If the DMA is not ready on the point we are going to start a DMA transfer in “New Input Processing”, it will go to “DMA Pending“ which checks the DMA status every time coming back from “Update neuron states”, and returns to “New Input Processing” once the DMA is ready.

Codes:

```
StarSim
          LDR     r7,  =CommsBuf            ;neuron commPtr
          LDR     r8,  =0x4000000           ;commRx
          LDR     r9,  =DTCMBuf1            ;DMALoc
          LDR     r5,  =DTCMBuf2
          EOR     r5,  r5,  r9
          STR     r5,  DMALocBits

          LDR    r10,  =NeuronData          ;&NeuronData[n]
          LDR    r11,  =0x0                 ;DelayPtr

SCHD      LDR     r1,  =IRQRawStatus
          LDR     r1,  [r1]
          TST     r1,  #DMA_INTbit
          MOVNE   r1,  r9
          BNE     Synaptic_process

NoDMA     LDR     r6,  =CommsBuf
          CMP     r7,  r6
          BNE     newCommsInput

NoInput   LDR     r3,  =NeuronData+NeuronNum*NeuronSize
          CMP     r3,  r10                 ;last neuron?
          BNE     Goon                     ;if not, go on
          LDR     r0,  DelayPtr
          CMP    r11,  r0                  ;r11 = current DelayPtr
          BEQ     Finish                   ;if yes, finish
          ADD    r11,  r11,  #0x2          ;if not, r11+2, and start new round
          CMP    r11,  #0x20
          MOVEQ  r11,  #0x0
          LDR     r8,  =0x4000000
          LDR    r10,  =NeuronData

Goon      BL      IzkEquation
Finish    B       SCHD           END
```

The scheduler will begin from “StarSim” when the system starts. In this part, some frequently used variables are kept in the registers for the purpose of getting better performance.

r7 -- new input buffer address; r8 -- 4 bytes’ new input; r9 -- DMA buffer address in D-TCM
r10 -- address of neuron parameters; r11 – Current Delay pointer

2.2 Update neuron bin

This routine is responsible for processing a set of synaptic data items in the D-TCM buffer which have already been copied from the SDRAM. The neuron data structures are as below:

```
struct Neurons
{
    struct  NeuronState
    {
        signed short Param_v;              //v
        signed short Param_u;              //u
        signed short Param_a;              //ab
        signed short Param_b;              //-a
        signed short Param_c;              //c
        signed short Param_d;              //d
    }NeuronStates;
    signed short   bin[16];                //delayed input buffer
};
```

The codes of task "update neuron bin":

```
; DTCMBuf1 is passing use r1
Synaptic_process
        ;clear dma interrupt
        ldr    r3,  =DMAIntClr
        str    r3,  [r3]

        LDR    r3,  =NeuronData
        ADD    r3,  r3,  #NDelayOffset
        MOV   r12,  #NeuronSize
        ADD    r2,  r1,  #BlkSize
L1      LDR    r0,[r1],  #4
;NB schedule instruction that doesn't use r0!
        BIC    r5,  r0,  #0xf0000000
        SMLABT r6, r12,  r5,  r3
;NB schedule instruction that doesn't use r6!
L2      ADD    r4,  r0,  r11, lsl #27              ;offset from Ptr, not #28
        LSR    r4,  r4,  #27
        LDRH   r5, [r6, r4]!
        ADD    r5,  r5,  r0

L3      STRH   r5, [r6]
        CMP    r1,  r2                             ;finished?
        BLT    L1
        B      SCHD
        END
```

2.3 New input processing

The "New Input Processing" routine arranges inputs to a fascicle by the source fascicle ID, finds out the memory address of the synaptic words belonging to this input in the SDRAM, and then starts a DMA transfer which copies them to the D-TCM buffer.

```
struct SFascicle
{
    int SFascAddr;                   //address to match
    int SFascMask;                   //bits to ignore
    int *SFascPtr;                   //Sdram address of fasc start
    struct SFascicle *NextSFasc;     //go here if >
};
```

In the current system, there are only two DMA buffers – DTCMBuf1 & DTCMBuf2, to save

D-TCM space usage as well as simplify the programming. But whenever a DMA is complete, we should process it as soon as possible to clean up the DMA buffer.

In the case when several new inputs arriving at the same time, we use another buffer – CommsBuf, which is used just like a stack, to buffer the new inputs.

Both of the new input buffers and DMA buffers are located in the D-TCM.

```
BufDataStart  SPACE  CommsBufSize+BlkSize*2

                MAP    BufDataStart  ;see option.s

CommsBuf  #  CommsBufSize
DTCMBuf1  #  BlkSize
DTCMBuf2  #  BlkSize
```

Codes of “new input processing”:

```
newCommsInput
         LDR      r0,  [r7, #-4]!                 ;r0 input(routing key) received
         LDR      r5,  =SourceFascicle
L1       MOV      r1,  r5
L2       LDMIA   r1!,  {r2-r5}
         ANDS     r6,  r0, r3                     ;mask out bits
         ;BEQ     error                           ;sentinel found
         CMP      r6,  r2                         ;this fascicle?
         BGT      L1
         BLT      L2
; assert: we have a match
         BIC      r6,  r0, r3                     ;offset within fasc
         ADD      r1,  r6, r4                     ;r1 start data address of the fired neuron
; r6 -> DRAM row to load to DTCM by DMA...

         LDR      r2,  =DMAstat                   ;check DMA status
         LDR      r2,  [r2]
         CMP      r2,  #0                         ;is it free?
         STMFD   sp!,  {r1}
         BLNE     DMAPending                      ;wait for DMA ready
         LDMFD   sp!, {r1}

;r0 dma register start addr  r1 = dma from r2 = dmato r3= dmasize r4=dmacon

         LDR      r0,  =DMAfrom                   ;dma register start addr
         LDR      r6,  DMALocBits
         EOR      r9,  r9, r6
         MOV      r2,  r9
         LDR      r3,  =BlkSize
         LDR      r4,  =0x1

         STMIA   r0!, {r1-r4}                     ;start DMA

         LDR      r1,  =IRQRawStatus              ;any previous DMA complete?
         LDR      r1,  [r1]
         TST      r1,  #DMA_INTbit                ;use the other DMA buffer
         BEQ      SCHD

         LDR      r6,  DMALocBits
         EOR      r1,  r9,  r6
```

```
        B        Synaptic_process
        END
```

2.4 Update neuron states

2.4.1 Description of the Equation and its parameters

The Izhikevich Equation has been adopted as the main algorithm for updating neuron states in the system.
The original form of the Equation is :

v =v + 0.04*v*v + 5*v + 140 + I – u …………...(1)
u = u + a*(b*v - u)……………………………….(2)
if v >= 30 mV, then v = c; u = u + d………..(3).

Where v and u are dimensionless variables, and a, b, c, and d are dimensionless parameters. v represents the membrane potential of the neuron and u represents a membrane recovery variable. Synaptic currents or injected dc-currents are delivered via the variable I.

Different choices of the parameters a, b, c, d result in different neuron firing patterns. In the software simulator, the ratio of excitatory to inhibitory neurons is 4 to 1 and parameters are generated randomly according to the following [3]:

Each excitatory cell:
$(a_i, b_i) = (0.02, 0.2)$ and $(c_i, d_i) = (-65,8) + (15,-6)\, r_i^2$
Each inhibitory cell:
$(a_i, b_i) = (0.02, 0.25) + (0.08, -0.05)\, r_i$ and $(c_i, d_i) = (-65,2)$

Where r_i is a random variable uniformly distributed on the interval [0, 1].
So, $0.02< a_i <0.1$; $0.2< b_i <0.25$

In the equation, the minimum value of variable v will be around -80, and the maximum will be around 380 (when its previous value was 30mV and depending on the value of I as well).

2.4.2 The choice of scaling factor

In order to represent the variables as integers with acceptable accuracy, we need a parameter to amplify the original variables and parameters which will be helpful to get a better precision.
The value range of Signed Short we can use is -32767 ~ +32767.

If we choose a scaling factor p, the value can be decided by:

$$|v^* p| < 32767 \qquad (-80<v<380)$$

since the variable v has both the smallest and the biggest value in the Equation. According to this,

$$p < 86$$

So it seems that we should choose 64 as the value of p. Is 64 the limit? The answer is no, because we can extend 16 bit to 32 bit within the processing procedure. Actually 256 is the limit according to:

$$|-80|*p<32767$$

-80 will be one of the results stored in the neuron structure which has a Signed Short data in it..

But 256 is still too small for the parameters a, b and constant 0.04 in the Equation if we implement it in the form [u = u + a*(b*v - u)]. What's more, if we want to implement it by using the form [u = ab*v –a*u + u] (which will execute more efficiently), p = 256 is far away from the requirement, because 0.004<ab<0.025. The solution to this issue is to adopt two different scaling factors: p1 & p2.

$$p1 = 256, p2 >= 1024$$

One issue concerning the choice of p1 & p2 is that it is not a simple case of the bigger the better. The option varies with the parameters a & b and implementation (such as the choice between u = u + a*(b*v - u) or u = ab*v –a*u + u) . Different combinations of p1 & p2 make a difference as well. But generally, a bigger scaling factor will be better.

Mathematic analasys:

v*p1 = {v*p1[(0.04p2* v*p1)/p2+ 6*p1]}/p1 + 140p1 – u*p1 + I*p1 ………(1)

u*p1 = [(- a*p2) *u*p1]/p2 +u*p1+ [(ab*p2)*v*p1]/p2 ……………………..(2)

where, p1 = 256; p2 = 65536

Programming Hint:

if when p2 = 65536, an equation such as

(0.04p2* v*p1)/p2+ 6*p1

can be implemented by one instruction – "SMLAWB"

2.4.3 Implementation by assembler code running on the Armulator

Following the discussion of scaling factors p1 & p2 in section 2.4.2, and taking programming and executing efficiency into consideration, we choose

$$p1 = 256, \quad p2 = 65536$$

The equations:

v = v*(0.04*v + 6) + 140 + I – u …………...(1)

u = ab*v –a*u + u……………………..…….(2)

are implemented by the code listed below:

```
IzkEquation

       MOV       r0,  r10
       LDMFD    r0!,  {r2-r4}

;;v = v*(0.04*v+6)+140+I-u;
       LDR      r12,  =const1
       LDR       r1,  =const2
       SMLAWB    r1,  r12, r2, r1             ;r1 = 0.04*v+6

       LSL       r1,  #(0x10-FixedP1)
       LDR      r12,  =const3
       SMLAWB    r1,  r1,  r2,  r12           ;r1 = v*(0.04*v+6)+140

       MOV       r5,  #0x1<<(FixedP1+InhibitI)  ;inhibits I

       LDR       r6,  =NeuronInh
       LDR       r6,  [r6]
       CMP      r10,  r6
       ADDCC     r5,  r5,  r5
       ADDCC     r5,  r5,  r5, lsr #1           ;excitatory I = 3*inhibit I

       LDRH      r6,  [r0, r11]                 ;r6 = I
       STRH      r5,  [r0, r11]

       LSL     r6,  #0x10                        ;in case of I is minus

       ADD     r1,  r1,  r6, ASR #0x10
       SUB     r1,  r1,  r2, ASR #0x10           ;r1 = v

;; u = -a*u+u+ab*v;

       SMLATT r12,  r2,  r3,  r2
       ASR    r12,  #0x10
       SMLAWB  r2,  r1,  r3,  r12                ;r2 = u;

;;if (v > 30)?
       LDR    r12,  =threshold
       CMP    r12,  r1
       BGE    TEnd1

       ADD     r2,  r4,  r2,  LSL #0x10          ;if v>30
       STR     r2,  [r10]

       STR     r8,  [r7], #4
       B       L4

TEnd1  STRH    r1,  [r10]                        ;store v&u
       STRH    r2,  [r10,#0x2]

L4     ADD     r8,  r8, #BlkSize
       ADD    r10,  r10, #NeuronSize
       MOV     pc,  lr

       END
```

2.5 Timer Interrupt Service

The timer in the Amulator is used to produce an interrupt every millisecond. In the timer interrupt service routine, operations such as accumulating the variable “DelayPtr”, checking the total running time of the system, and preparations for the next millisecond are carried out.
The codes for these functions are below:

```
Timers
        STMFD sp!,  {r0,  r1}
L1      LDR       r0,  ExecTime
        ADD       r0,  r0,  #0x1
        STR       r0,  ExecTime
        CMP       r0,  #TimeLimits
        BNE       L2
        BL        ProEnd
        B         .

L2      LDR       r0,  DelayPtr
```

```
        ADD      r0,  r0,  #2
        CMP      r0,  #0x20                       ;DelayPtr = 34?
        MOVEQ    r0,  #0x0
        STR      r0,  DelayPtr

        ;clear timer 1 interrupt
        ldr      r0,  =TCLR1
        ldr      r1,  =0x10
        str      r1,  [r0]

        LDMFD   sp!,  {r0,  r1}
        SUBS   pc,  lr,  #4
        END
```

2.6 DMA Pending

“DMA Pending” is called when the DMA is already busy and a new DMA operation cannot be started by the “New Input Processing” task. The program checks the DMA status very frequently and goes back to the “New Input Processing” immediately the DMA is ready. The codes for these functions are below:

```
DMAPending
         STMFD  sp!,  {lr}
loop
         LDR     r3,  =NeuronData+NeuronNum*NeuronSize
         CMP     r3,  r10                        ;last neuron?
         BNE     goon                            ;if not, go on
         LDR     r0,  DelayPtr
         CMP    r11,  r0                         ;r11 = current DelayPtr
         BEQ     L1                              ;if yes, finish
         ADD    r11,  r11, #0x2                  ;if not, r11+2, and start new round
         CMP    r11,  #0x20
         MOVEQ  r11,  #0x0
         LDR     r8,  =0x4000000
         LDR    r10,  =NeuronData

goon     BL      IzkEquation
L1       LDR     r2,  =DMAstat                   ;test DMA status
         LDR     r2,  [r2]
         CMP     r2,  #0
         BNE     loop
         LDMFD  sp!,  {pc}
         END
```

2.7 System initialization

In order to start the system, a set of neuron data must be initialized including parameters for all neurons and synaptic weights for all connections. This part is implemented in C code.

Parameters:

```
void Neurons_Init(void)
{
    int i,j;
    float ra, rb;
    //excitatory neurons
    for (i=0; i<InStartNum; i++ )
    {
```

```
    NeuronData[i].NeuronStates.Param_v = (signed short)(ExciteV*FixedP1);
    NeuronData[i].NeuronStates.Param_u = (signed
short)(ExciteU*NeuronData[0].NeuronStates.Param_v);
    NeuronData[i].NeuronStates.Param_a = (signed short)(ExciteA*FixedP2);           //a = ab
    NeuronData[i].NeuronStates.Param_b = (signed short)(ExciteB*FixedP2);           //b = -a
    NeuronData[i].NeuronStates.Param_c = (signed
short)((ExciteC+ExciteCr*(rand()/(RAND_MAX+1.0))*(rand()/(RAND_MAX+1.0)))*FixedP1); //c
    NeuronData[i].NeuronStates.Param_d = (signed
short)((ExciteD+ExciteDr*(rand()/(RAND_MAX+1.0))*(rand()/(RAND_MAX+1.0)))*FixedP1); //d

    for (j=0; j<16; j++)
        {
         NeuronData[i].bin[j]= (signed short)(ExciteI*(rand()/(RAND_MAX+1.0))*FixedP1);
                                                                         //I currency
        }
    }

    //inhibitory neurons
    for (i=InStartNum; i<NeuronNum; i++ )
    {
        ra = InhibitA+InhibitAr*(rand()/(RAND_MAX+1.0));
        rb = InhibitB+InhibitBr*(rand()/(RAND_MAX+1.0));

        NeuronData[i].NeuronStates.Param_v = (signed short)(InhibitV*FixedP1);
        NeuronData[i].NeuronStates.Param_u = (signed
short)(InhibitU*NeuronData[0].NeuronStates.Param_v);
        NeuronData[i].NeuronStates.Param_a = (signed short)(ra*rb*FixedP2);    //a = ab
        NeuronData[i].NeuronStates.Param_b = (signed short)(-ra*FixedP2);      //b = -a
        NeuronData[i].NeuronStates.Param_c = (signed short)(InhibitC*FixedP1); //c
        NeuronData[i].NeuronStates.Param_d = (signed short)(InhibitD*FixedP1); //d
     for (j=0; j<16; j++)
     {
        NeuronData[i].bin[j]= (signed short)(InhibitI*(rand()/(RAND_MAX+1.0))*FixedP1);
     }
   }
}
```

Synaptic weights:

```
//SnapticWord |--4b-delay-|0|-11b-index-|-16b-weight-|
void SnapticWord(void)
{
    int j,k;

     srand( (unsigned)time( NULL ) );

    for (k=0; k<InStartNum; k++)
    {
         for (j=0;j<BlkSize/4; j++)
         {

           Synapse[k][j] = (int)(16.0*rand()/(RAND_MAX+1.0))<<28 | (int)(rand()%NeuronNum)<<16
| (int)((ExSpikI*rand()/(RAND_MAX+1.0))*FixedP1);
          }
    }
    for (k=InStartNum; k<NeuronNum; k++)
    {
         for (j=0;j<BlkSize/4; j++)
         {
           Synapse[k][j] = (int)(16.0*rand()/(RAND_MAX+1.0))<<28 | (int)(rand()%NeuronNum)<<16
| (int)((InSpikI*rand()/(RAND_MAX+1.0))*FixedP1)&0xffff;
          }
    }
}
```

3. System Performance:

We model a system with a 200MHz ARM968 core, a 100MHz AHB bus, a 64k D-TCM and a 100MHz SDRAM, on the Armulator, using the IDE Realview 2.2.

1. Update neuron bin.
   This takes 110 ns/ weight (assuming that the neuron parameters are in the D-TCM) and 240 ns/ weight on average (if the neuron parameters are in the SDRAM)
2. New input processing.
   This takes 280ns to process an input from a fired neuron and start a DMA transfer (if the DMA is free).
3. Update neuron states.
   One neuron update using the Izhikevich Equation takes 240 ns (reset I to constant after updating, if reset to a random value, it will take 330ns) when the neuron parameters (which are V, U, a, b, c, d and I corresponding to the Equation) are in the D-TCM, and it takes about 660 ns to do the same job when the parameters are in the SDRAM (100 MHZ).

4. Performance analysis:

4.1 Firing frequency allowed

For 1000 neurons with 10% connectivity, x represents the maxima number of neurons fired in 1 millisecond (the fire rating), when all of the debugging functions are turned off.

110*100*x + 280*x + 240*1000 + y = 1000 000 (Nanosecond)

y represents time delays (nanosecond) caused by scheduler, timer interrupt, DMA pending subroutine

x equals to about 60.

Notes: there is no operation to search the fascicle binary tree in this project and writing the DMA register takes tens of nanoseconds, the delay of the DMA operation is also not very accurate at the moment.

4.2 D-TCM usage:

X represents the maxima numbers of neuron can be allocated in the D-TCM.

768(heap) + 512(stack) + 4*100 (CommsBuf) + 44*x(neuron parameters) + 2*BlkSize(DMABuf) + about 40 (global variable) + 56 (VectorTable) <= 64k (Byte)

X is about 1400.

5. Simulating results

Figure 5.1, 5.2, 5.3 presents the firing patterns of an excitatory neuron in a system of 1000 neuron with 10% connectivity, the ratio of excitatory neurons to inhibitory neurons is 4 to 1 [3], initial and reset values of I:

I for excitatory neuron is random 0 - 6 (mV)

I for inhibitory neuron is random 0 - 2 (mV)

the value of synaptic connection weights are:

excitatory: random 0 - 1.0 (mV) inhibitory: random -2.0 - 0 (mV)

Computes for 1 second, the number of firings received by this neuron is 613.

In the figures below, the x coordinate represents time (ms), the y coordinate is its original value multiplied by the scaling factor p1 (256).

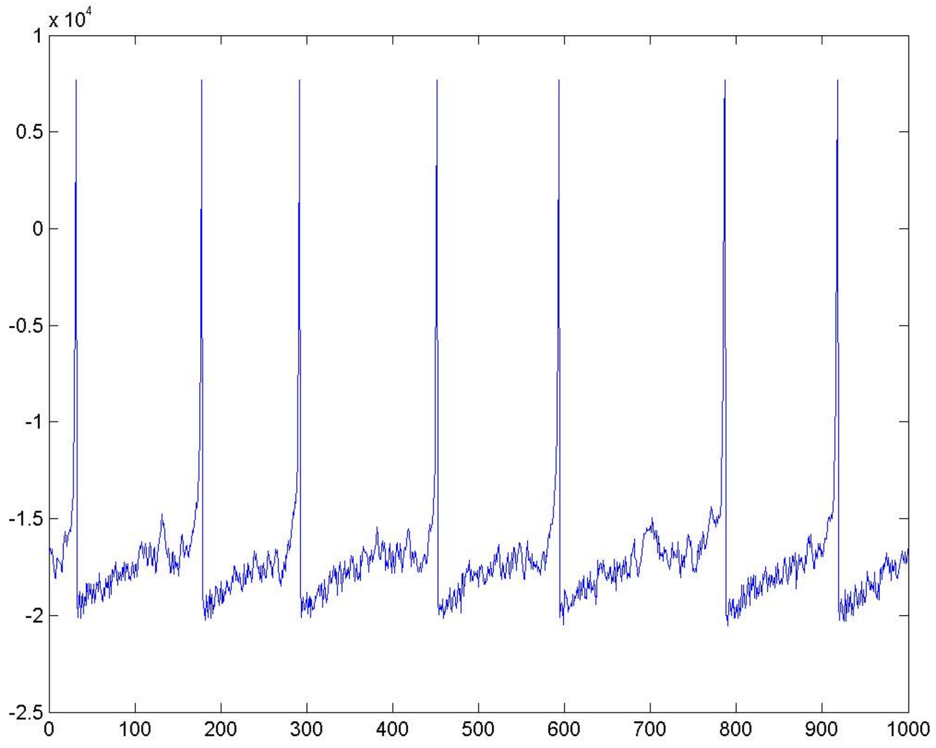

Figure 5.1 the membrane potential variable v of an excitatory neuron

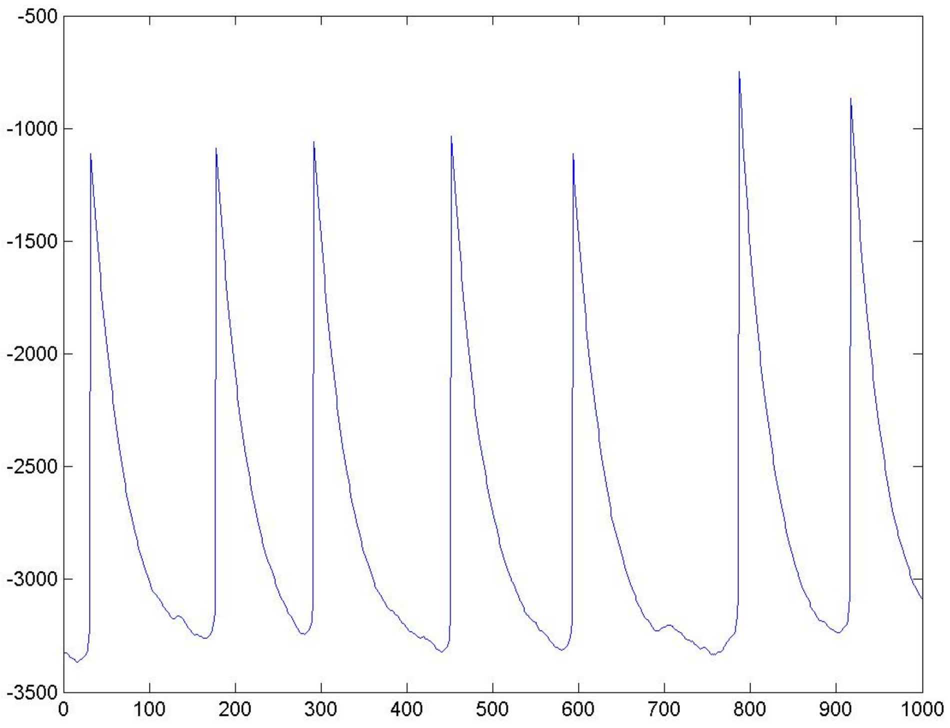

Figure 5.2 the membrane recovery variable u of an excitatory neuron

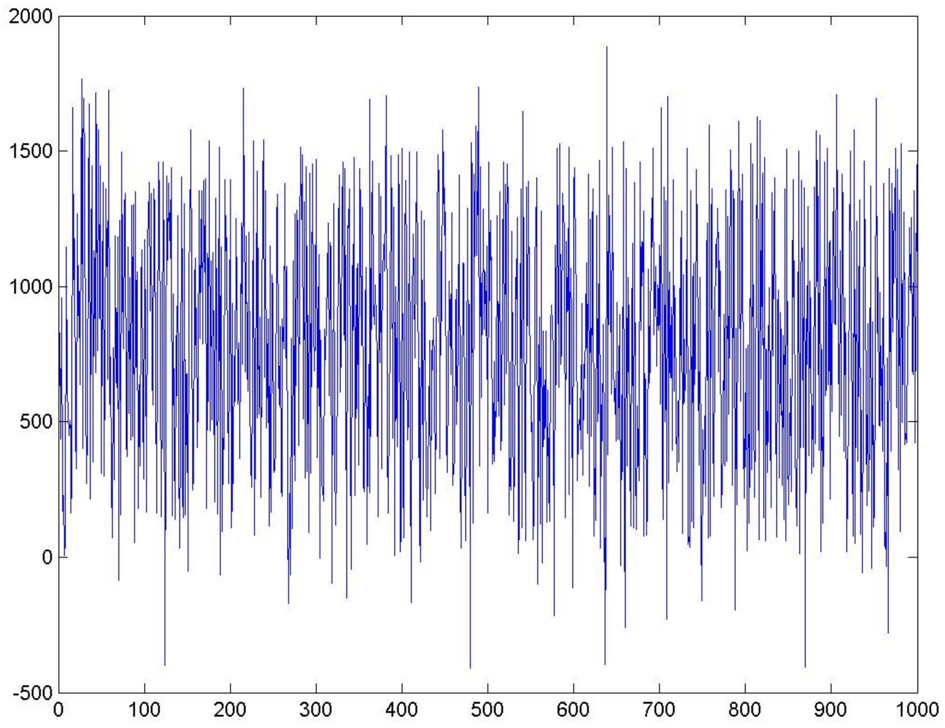

Figure 5.3 the current variable I of an excitatory neuron

The firing patterns of an inhibitory neuron in the same system as above are displayed in figures

5.4, 5.5 5.6.

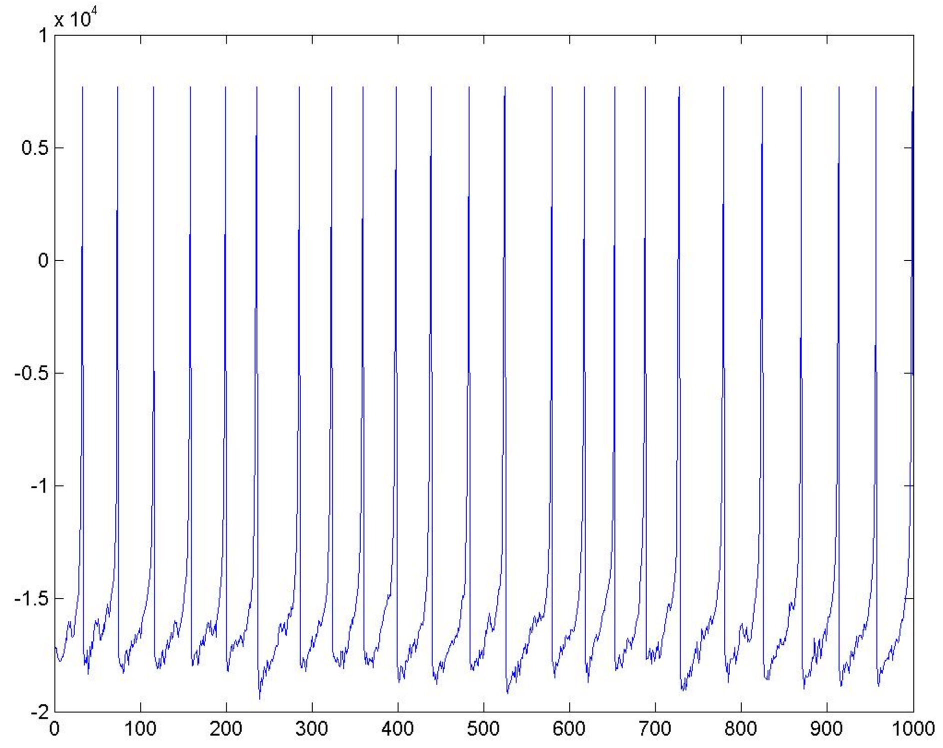

Figure 5.4 the membrane potential variable v of an inhibitory neuron

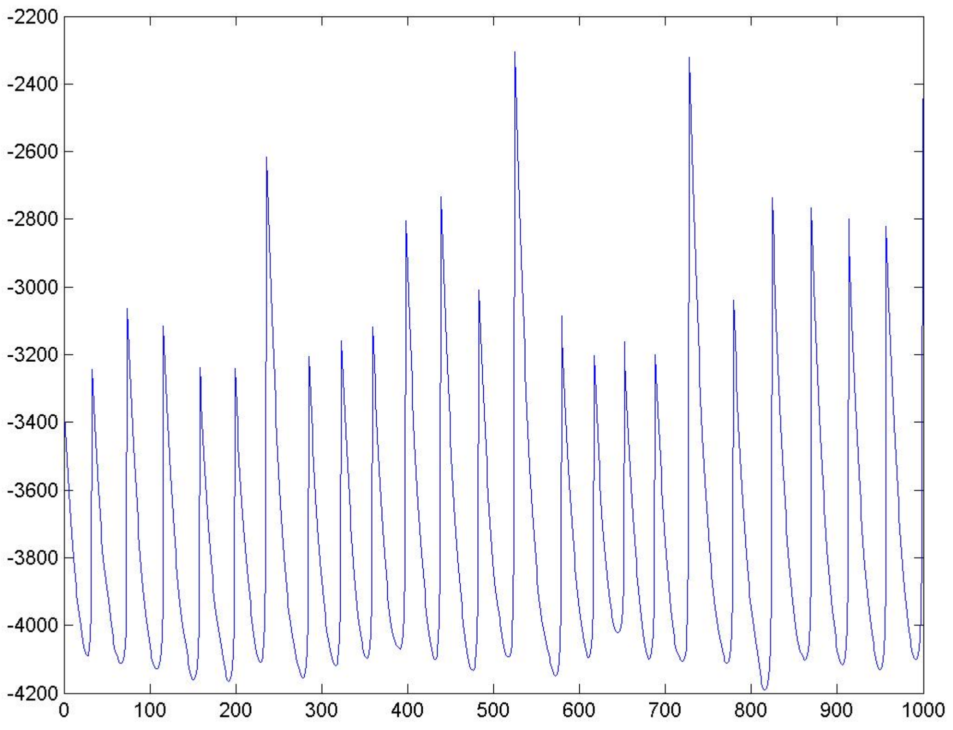

Figure 5.5 the membrane recovery variable u of an inhibitory neuron

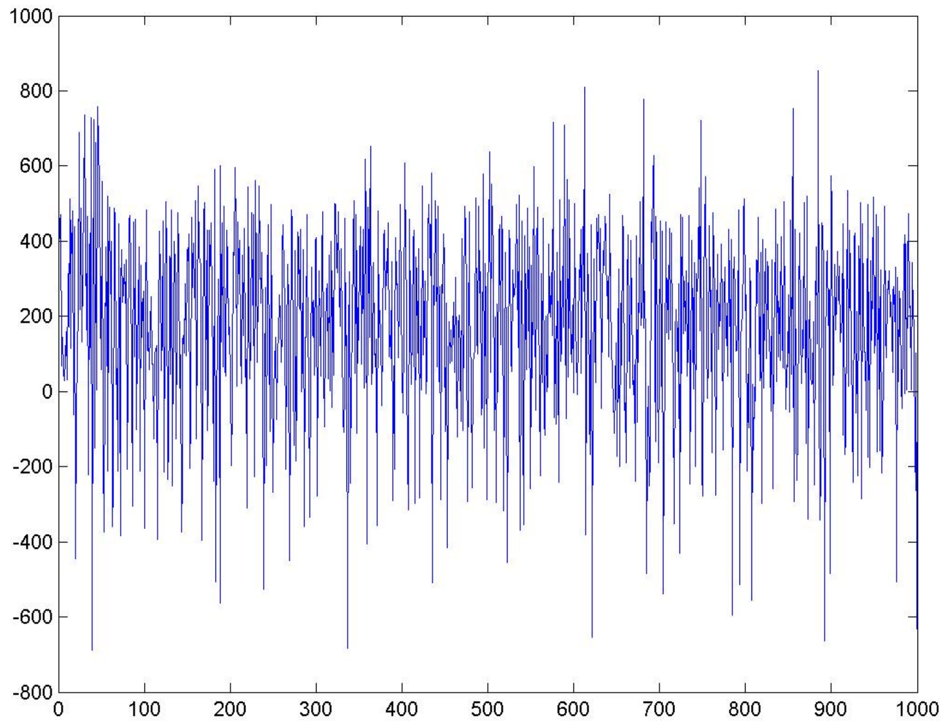

Figure 5.6 the current variable I of an inhibitory neuron

References:

[1] S. B. Furber, S. Temple and A. D. Brown. “High-Performance Computing for Systems of Spiking Neurons”. The AISB’06 workshop on GC5: Architecture of Brain and Mind, Bristol, 3-4 April 2006.

[2] Eugene M. Izhikevich “Which Model to Use for Cortical Spiking Neurons”, IEEE Trans. Neural Networks, vol. 15, no. 5, Sep. 2004

[3] Eugene M. Izhikevich, “Simple Model of Spiking Neurons”, IEEE Trans. Neural Networks, vol. 14, pp. 1569-1572, Nov. 2003.

[4] S. B. Furber, S. Temple and A. D. Brown. “On-chip and Inter-Chip Networks for Modelling Large-Scale Neural Systems”. Proc. ISCAS’06, Kos, May 2006.

[5] Eugene M. Izhikevich, “Neural excitability, spiking and bursting,” Int. J. Bifurcation Chaos, vol. 10, pp. 1171-1266, 2000.

[6] ARM Ltd. ARM968E-S Technical Reference Manual. DDI 0311C, 2004. http://www.arm.com/products/CPUs/ARM968E-S.html