Projektpraktikum: Methode zur Validierung des Stage-2 Plastizitäts-Prozessors

Tobias Nonnenmacher
Betreuer: Simon Friedmann

18.4. - 18.5.2011

Abstract

This report is about a test-method for a new embedded processor (S2PP), that implements the Power Instruction Set Architecture (ISA) 2.06.
In order to verify the data of the testsystem, both of the test- and a reference-system the results are stored each in continous arrays that are compared in the end. It is important, that the testfunction is the same on both systems.
For simple integer-storage and a small prime number-test this method could be verified.

Zusammenfassung

Im vorliegenden Praktikumsbericht wird eine Methode beschrieben, die die Funktionsweise eines neu entwickelten, eingebetteten Prozessors auf Basis der Power ISA 2.06 testen soll. Dabei werden sowohl auf dem Testsystem als auch auf dem Referenzsystem Ergebnisse in einem zusammenhängenden Speicherarray abgelegt und am Ende auf Übereinstimmung überprüft. Es wird dabei auf jedem System dieselbe Testfunktion implememtiert.
Für das einfache Ablegen von Variablen und einen kleinen Primzahltest konnte diese Methode verifiziert werden.

Inhaltsverzeichnis

1 Einleitung

In der Electronic Vision(s)-Arbeitsgruppe am Kirchhoff-Institut für Physik an der Universität Heidelberg wird mit Hilfe analoger Schaltkreise das Verhalten biologischer Neurone simuliert. Als technologische Grundlage für die Modellierung neuronaler Netze dienen hier VLSI- Netzwerke in analogen Netzwerkchips[1].
Für weiterführende Informationen sei hier an dieser Stelle auf die Projekthomepage der Arbeitsgruppe verwiesen[2].
Zur Implementierung von Plastizität[3] in der neuromorphen Hardware wird hier ein Prozessor entwickelt, der Stage-2 Plasticity Processor (S2PP).
Seine Aufgabe soll am Ende darin bestehen synaptische Gewichte und andere Parameter interaktiv zu manipulieren und so ein entsprechendes Lernverhalten zu simulieren.
Grundlage für die Architektur des Prozessors ist die Power ISA 2.06[4] (Instruction Set Architecture).

1.1 Zielsetzung

Ziel des vorliegenden Projektes war es eine Möglichkeit zu finden, mit der man die Funktion komplexerer Programme auf dem S2PP testen und das Ergebnis mit dem eines Kontrollsystems vergleichen und damit validieren kann. Diese Möglichkeit sollte hierbei vorsehen, dass mit einem einzigen C-Programm, das sowohl auf dem Referenzsystem als auch auf dem S2PP ausgeführt wird, ein Datenabbild erzeugt wird, anhand dessen direkt ein Vergleich durchgeführt werden kann (siehe Abb.1).
Dabei erfolgt die Simulation des S2PP zyklentreu im Simulationsprogramm ModelSim SE 10.0 basierend auf den Designfiles in SystemVerilog.

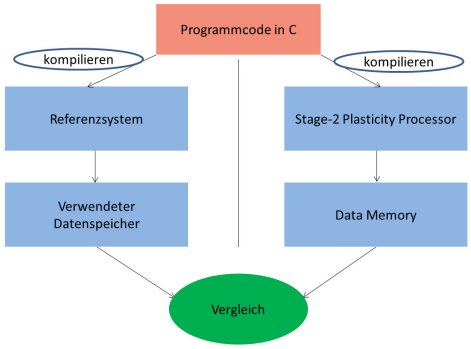

Abbildung 1: Projektziel

[1] http://www.kip.uni-heidelberg.de/cms/vision/projects/facets/neuromorphic_hardware/ [6.6.2011]
[2] http://www.kip.uni-heidelberg.de/cms/groups/vision/
[3] Schemmel, J et al.:Implementing Synaptic Plasticity in a VLSI Spiking Neural Network Model, Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN 2006), 1-6, IEEE Press (2006)
[4] http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf [6.6.2011]

1.2 Vorüberlegungen

Für die Umsetzung dieses Ziels zur Speicherabbilderstellung wurden zu Beginn mehrere Möglichkeiten betrachtet:

- Emulation eines Power-Prozessors
- Erstellen eines kompletten Speicherabbildes durch Debugging auf einfachem Desktop-Referenzsystem
- Ausgabe des Datenspeichers über den Programmcode

Die Emulation eines Power-Prozessors war zu Beginn die naheliegendste Methode. Dabei sollte ein vergleichbares funktionelles Verhalten mit dem Speicherumgang wie das des S2PP in Software emuliert werden.
Allerdings ergab sich hier bei näherer Betrachtung das Problem einen passenden Emulator zu finden, der dieselbe ISA simuliert.
Infrage käme hier beispielsweise der QEMU[5]. Darüber hinaus besteht eine Hauptschwierigkeit darin ein komplett emuliertes Betriebssystem mit Power EABI[6] aufzusetzen, weshalb diese Möglichkeit schließlich ersteinmal zurückgestellt wurde.
Bei der zweiten Möglichkeit war vorgesehen, durch ein Debugging-Programm (z.B. gdb[7]) nach dem Programmablauf ein Abbild des kompletten Hauptspeichers sowohl von Referenz- (z.B. x86-Architektur) als auch vom Testsystem(S2PP) zu erstellen und diese beiden Abbilder dann zu vergleichen. Hier ergab sich neben dem Problem der sich stark unterscheidenden ISA zwischen Desktop-Prozessor und S2PP, die sehr verschiedene Kompilierungen bewirkt, auch noch die Tatsache, dass die gängigen Desktop-Prozessoren die Daten im Little Endian Format ablegen, im Gegensatz zum Power-Prozessor, der im Big Endian Format codiert.
Als dritte Möglichkeit hatten wir uns überlegt, den verwendeten Datenspeicher des Programms direkt im Code schon ausgeben zu lassen.
Diese Methode soll nun im Folgenden vorgestellt werden.

2 Realisierung

Die grundsätzliche Idee für die Umsetzung der Möglichkeit einer Ausgabe des Datenspeichers über Programmcode war es, dass in einem Array die Daten eines bestimmten Typs nacheinander in einer selbst gewählten Reihenfolge abgelegt werden können.

2.1 Speicherplatzreservierung und Auslesen im Referenzsystem

Um die Daten möglichst nahe an der Hardware in einen Speicher zu schreiben, bzw. aus diesem auszulesen, sollte die Reservierung eines dynamischen Speicherbereichs erfolgen,

[5] http://wiki.qemu.org/Main_Page
[6] IBM Microelectronics: PowerPC Embedded Processors Application Note, Version 1.0, 21.Sept.1998
[7] http://www.gnu.org/software/gdb/

der direkt auf den entsprechenden Hauptspeicherbereich abgebildet ist und somit nicht nur im virtuellen, sondern auch im physikalischen Adressraum zusammenhängend dargestellt ist.
In diesen Speicher sollten alle relevanten Ergebnisse und Zwischenergebnisse aus dem Programmcode direkt abgelegt werden.
Dafür hatte sich der *malloc()*-Befehl aus der stdlib.h[8] angeboten, der als Rückgabewert einen Zeiger auf den ersten Wert des reservierten Bereichs hat (Bsp. Listing 1).

```
1 //a ist hier die Groesse des Speichers in Byte
2 int a, i;
3 a = 1024;
4
5 //mem als Zeiger auf Integer zur 0.Adresse des reservierten Speichers
6 int * mem = malloc(a);
7
8 //fuellen des reservierten Speichers
9 for(i = 0; i<256; i++) mem[i] = i;
```

Listing 1: *malloc()*-Beispielcode im Referenzsystem

Das byteweise Auslesen der Speicherplätze wurde durch einen weiteren Zeiger auf *unsigned char* zur selben Speicheradresse wie der Rückgabeadresswert des Zeigers beim *malloc()*-Befehl ermöglicht. Allerdings ist hier an dieser Stelle das zuvor schon erwähnte Problem der Variablencodierung im Big Endian oder Little Endian Format weiterhin erhalten, sodass für den Auslesevorgang bei 32-bit Integervariablen immer die entsprechende Gruppe von 4 Bytes in umgekehrter Reihenfolge ausgelesen werden muss: Byte3→Byte2→Byte1→Byte0→Byte7→...
Abbildung 2 gibt nochmal einen Überblick über das Verfahren zur Speicherreservierung im Referenzsystem.

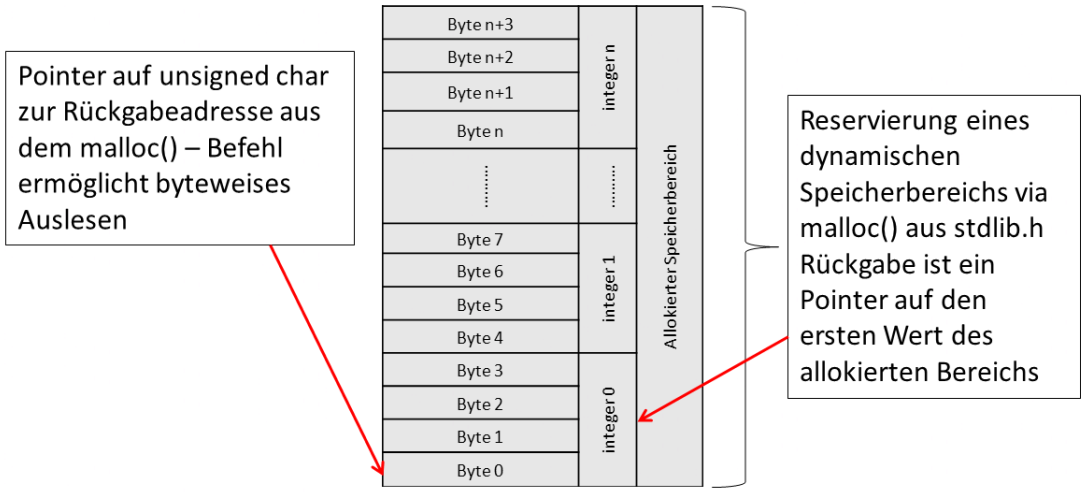

Abbildung 2: Speicherreservierung im Referenzsystem

[8] http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf, Kapitel 7.20.3.3 [6.6.2011]

2.2 Speicherplatzbelegung und Auslesen im S2PP

Auf dem eingebetteten S2PP braucht man im Gegensatz zum Referenzsystem keine dynamische Speicherplatzreservierung in diesem Sinne zu machen, sondern kann direkt über entsprechende Zeiger auf bestimmten Speicheradressen Lese- und Schreibzugriffe machen, da die Speicherverwaltung hier durch kein Betriebssystem reguliert wird.
Der dem Beispiel aus Listing 1 entsprechende Code ist in Listing 2 dargestellt.

```
1 int * mem;
2
3 //mem ist Zeiger auf Integer zur Adresse 0x10
4 mem = 0x10;
5
6 //Testfunktion
7 for(i = 0; i<256; i++) mem[i] = i;
```

Listing 2: Beispielcode in der S2PP-Simulation

Nach Ausführung eines Programms wird aus der Simulation schon direkt eine Datei ausgegeben, die dem Speicherabbild des S2PP am Programmende entspricht. Dabei ist der Stack, der beispielsweise Laufvariablen aus Funktionen enthält (momentan) am Ende des Speicherabbildes angeordnet, sodass man mit einem Zeiger auf die Startadresse von *0x10* einen eher maximalen Abstand gewählt hat und gegenseitige Beeinflussung erstmal für kleinere Programme ausgeschlossen werden kann. Probleme bezüglich gegenseitigem Störverhalten werden an dieser Stelle erst bei rekursiven Funktionen höherer Ordnung, die einen sehr großen Stack benötigen und bei großen abgelegten Datenmengen erwartet. Die momentan simulierte Speichergröße incl. Stack beträgt 4kB.

2.3 Implementierung

Auf dem S2PP wurde die zu testende Funktion in der Haupfunktion mit dem Zeiger *mem* als Übergabevariable aufgerufen (vgl. Testfunktion in Listing 2, bzw. Code im Anhang). Die Testfunktion selbst wurde im Header eingebunden.
Im Code für das Referenzsystem wurde der Aufruf der Testfunktion auf dieselbe Art durchgeführt. Allerdings musste hier bei der Initialisierung der Hauptfunktion noch die Speicherreservierung gemacht und die entsprechenden Zeiger gesetzt werden(vgl. Listing 3, bzw. Anhang). Für die Ausgabe des Speicherarrays wird eine Datei angelegt, in der die 32-bit Integervariablen in Big-Endian-Codierung abgelegt werden.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include "testfunktion.h"
4
5 int main(){
6   FILE * pFILE;
7   int* mem = malloc(4*size);
8   unsigned char* t;
9   t = mem;
10
11  int i, j;
```

```
12
13    //zu testende Funktion
14    test_funktion(mem);
15
16    pFILE = fopen ("endianneu.txt","w");
17
18    for(i = 3; i < 4*size; i = i+4){
19      for(j = i; j > i - 4; j--){
20        if(t[j] < 0x10) fprintf(pFILE, "0%x\n", t[j]);
21        else fprintf(pFILE, "%x\n", t[j]);
22      }
23    }
24
25    fclose (pFILE);
26    free(mem);
27    compare();
28    return 0;
29  }
```

Listing 3: Hauptfunktion im Referenzsystem

Mit *compare()* am Ende des Codes wurde ein Vergleichsparser als void-Funktion aufgerufen, der die Speicherdatei der Simulation und die zuvor angelegte Datei des Referenzsystems ausliest und dabei direkt miteinander vergleicht.
Es wird jeweils das entsprechende Byte aus den Dateien beider Systeme ausgegeben, auf die der Pointer gerade zeigt, falls hierbei ein Fehler auftritt wird dies gleich an dieser Stelle mit angegeben; bei kompletter Übereinstimmung der verglichenen Bereiche erhält man am Ende eine *match*-Ausgabe. Für die Datei aus der Simulation ist dabei momentan eine maximale Kommentarlänge von 1024 Bytes vorgesehen, die beim Parsen zu Beginn übersprungen wird.
Die Größe des zu vergleichenden Bereiches wird durch eine globale Variable *size* festgelegt, da weder auf dem Referenzsystem mit dem *malloc()*-Befehl, noch auf der Prozessorsimulation eine vorherige Initialisierung des allokierten Bereichs durchgeführt wird, sodass ein Vergleich über den beschriebenen Sektor hinaus auf jeden Fall zu Fehlermeldungen führen würde.
Der komplette verwendete Code für das Referenzsystems ist im Anhang angefügt.

3 Testprozedur

Die Übersicht zur Durchführung eines Testdurchlaufs ist in Abbildung 3 dargestellt.

Nach der Programmierung eines Testprogramms, in dem der zu belegende Speicher über *size* festgelegt wird, kann das Programm für das Power-System kompiliert werden. Die kompilierte Datei wird nun in der Simulation ausgeführt, wobei hier allerdings darauf geachtet werden muss, dass das Programm auch terminiert. Ein nicht abgeschlossener Programmablauf führt zu einem Speicherabbild in der Simulation, bei dem im zu vergleichenden Bereich weniger Variablen abgelegt sind als im Referenzsystem, in dem das

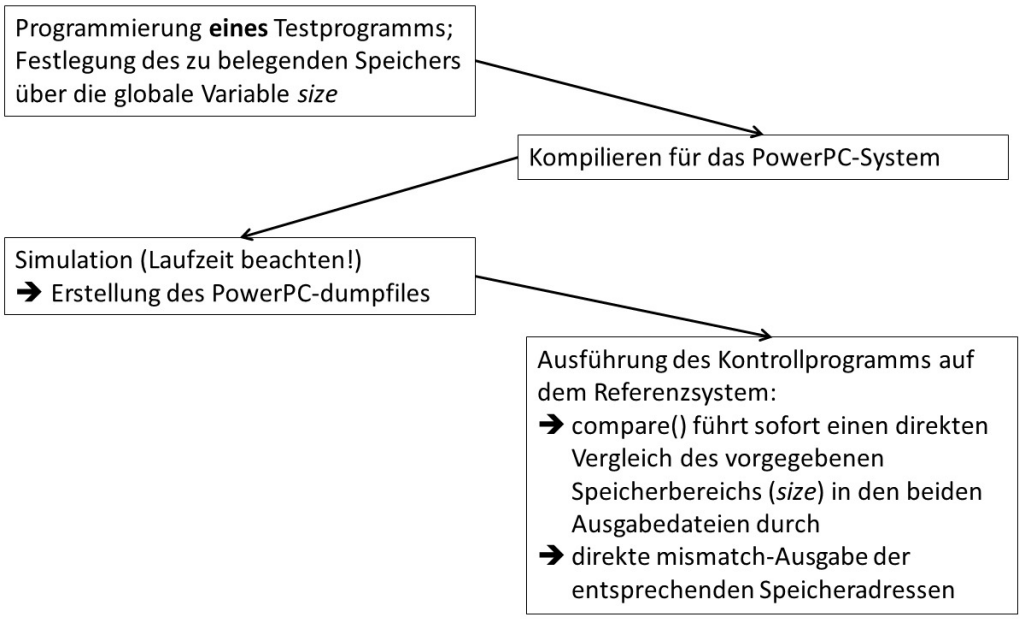

Abbildung 3: Testprozedur

Programm terminiert hat. Dies führt zu mismatch-Ausgaben beim späteren Vergleich. Nach erfolgreicher Erstellung des Abbilds aus der Simulation kann das Programm für das Referenzsystem kompiliert und darauf ausgeführt werden. Der Vergleichsparser sorgt dann am Ende für die direkte Ausgabe der entsprechenden Fehlerstellen.
Bisher wurden mit dieser Methode nur weniger komplizierte Programme, wie das Ablegen von Integer-Variablen oder ein einfacher Primzahltest (Programmcode s. Anhang) durchgeführt. Diese Tests zeigten aber bisher allesamt Übereinstimmung für die Ergebnisse auf beiden Systemen an.

4 Diskussion

Es ist klar, dass in Zukunft kompliziertere Testprogramme auf dem Prozessor laufen sollen, sodass einer der Kritikpunkte sicherlich die Tatsache ist, dass hier im vorliegenden Code nur Integervariablen vorgesehen sind. Komplexere Programme enthalten allerdings noch andere Datentypen, die auch berücksichtigt werden müssen. Die einfachste Lösung für dieses Problem wäre für diese Datentypen eigene Speicherbereiche zu reservieren, die dann gesondert verglichen werden.
Bei der Vorstellung der Methode wurde darauf verwiesen, dass mit dem Vergleich der (Zwischen-)ergebnisse natürlich nur das Ergebnis selbst verglichen wird. Bei einer Fehlersuche – für den Fall einer mismatch-Ausgabe – ist es jedoch sinnvoller zusätzlich noch den Stack vergleichen zu können, sodass die Grundidee der Simulation eines Power-Prozessors nochmals genauer überprüft wird.
Da für weitere Tests aufwändigere Programme getestet werden sollen, die auf der Simu-

lation relativ lange Testzeiten in Anspruch nehmen können, ist vorgesehen diese dann auf FPGAs oder auf einem in Fertigung befindlichen Prototyp ASIC zu testen.
Hierfür waren Programme nahe am künftigen Einsatz des Prozessors zur Plastizität und möglicherweise ein Performance-Benchmark angedacht.

5 Abkürzungsverzeichnis

ASIC application-specific integrated circuit

EABI embedded-application binary interface

FPGA Field Programmable Gate Array

ISA Instruction Set Architecture

QEMU Quick Emulator

S2PP Stage-2 Plasticity Processor

VLSI Very Large Scale Integration Network

6 Literaturverzeichnis

[1] http://www.kip.uni-heidelberg.de/cms/vision/projects/facets/neuromorphic_hardware/ [6.6.2011]
[2] http://www.kip.uni-heidelberg.de/cms/groups/vision/
[3] Schemmel, J et al.:Implementing Synaptic Plasticity in a VLSI Spiking Neural Network Model, Proceedings of the 2006 International Joint Conference on Neural Networks (IJCNN 2006), 1-6, IEEE Press (2006)
[4] http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf [6.6.2011]
[5] http://wiki.qemu.org/Main_Page
[6] IBM Microelectronics: PowerPC Embedded Processors Application Note, Version 1.0, 21.Sept.1998, https://www-01.ibm.com/chips/techlib/techlib.nsf/techdocs/852569B20050FF77852569970071B0D6/$file/eabi_app.pdf [7.6.2011]
[7] http://www.gnu.org/software/gdb/
[8] http://www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf, Kapitel 7.20.3.3 [6.6.2011]

7 Anhang

7.1 Code für den Stage-2 Plasticity Processor

```
#include "testfunktion.h"

int start() {

  int* mem;
  mem = 0x10;

  //zu testende funktion
  test_funktion(mem);

return 0;
}
```

matrix.c

7.2 Code für das Referenzsystem

```
#include <stdio.h>
#include <stdlib.h>

#include "testfunktion.h"

//die maximale kommentarlaenge in den dateien betraegt 1024byte

void compare(void){

  int i, indikator;
  char *dump = malloc(1024 + 4*size);
  char *endian = malloc(1024);


  FILE *pFiledump;
  FILE *pFileendian;

  pFiledump = fopen ("/afs/kip.uni-heidelberg.de/user/tnonne/s2pp-source/
      dump_c_matrix.mem","r");
  pFileendian = fopen ("endianneu.txt","r");

  i = 0;
  while(i < 0x10){
    i++;

    fscanf (pFiledump, "%s", dump);

    //ausblendung der kommentare
    if((dump[0] == '/') && (dump[1] == '/')) {
      while(fgetc(pFiledump) != '\n');
      i--;
```

```
31      }
32
33      else {
34        printf("%d, %s \n",i, dump);
35      }
36    }
37
38    indikator = 1;
39    for(i = 0; i < (4*size); i++){
40      fscanf(pFiledump, "%s", dump);
41      fscanf(pFileendian, "%s", endian);
42
43      if(strcmp(dump, endian) != 0){
44        printf("%d mismatch in ", i);
45        indikator = 0;
46      }
47
48      printf("%d, %s, %s \n", i, dump, endian);
49    }
50
51    fclose (pFiledump);
52    fclose (pFileendian);
53
54    free(endian);
55    free(dump);
56
57    if(indikator == 1) printf("match \n");
58  }
59
60
61  int main(){
62
63    FILE * pFILE;
64    int* mem = malloc(4*size);
65    unsigned char* t;
66
67    t = mem;
68
69    int i, j;
70
71
72    //zu testende Funktion
73    test_funktion(mem);
74
75    pFILE = fopen ("endianneu.txt","w");
76
77    for(i = 3; i < 4*size; i = i+4){
78
79      for(j = i; j > i - 4; j--){
80
81        if(t[j] < 0x10){
```

```
82            //printf("%d, %x, %p, 0%x, %d \n", j, mem[i/4], (mem + (i/4)), t[j
                  ], t[j]);
83            fprintf(pFILE, "0%x\n", t[j]);
84          }
85
86          else{
87            //printf("%d, %x, %p, %x, %d \n", j, mem[i/4], (mem + (i/4)), t[j
                  ], t[j]);
88            fprintf(pFILE, "%x\n", t[j]);
89          }
90        }
91    }
92
93    fclose (pFILE);
94    free(mem);
95    compare();
96
97    return 0;
98 }
```

endianneu.c

7.3 Testprogramm

```
1  //anzahl der ints mit denen der allokierte speicher befuellt werden soll
2  int size = 24+200;
3
4  void test_funktion(int* mem){
5
6    int i, j, x;
7    j = 0;
8
9    for(x = 3; x < 100; x++){
10
11     i = 2;
12
13     while(x%i != 0){
14       i++;
15     }
16
17     if(x == i){
18       mem[j] = x;
19
20       j++;
21     }
22   }
23
24   //Fuellung des Arrays mit ints
25   for(i=24; i < 225; i++) {
26     mem[i] = i;
27   }
28 }
```

testfunktion.h