Recapitulare Arhitectura SIMD
Pentru cei care nu isi mai amintesc ce este o arhitectura SIMD (*din laboraotrul despre Taxonomia Flynn*): Arhitecturile SIMD (Single Instruction stream, Multiple Data stream) au capacitatea de a manipula date vectoriale si matriciale in timpi mici. Un exemplu de arhitectura SIMD este arhitectura CELL.

Utilizarile cele mai frecvente sunt in cercetarea cancerului si prevenirea meteo. Puterea acesor masini devine evidenta cand dimensiunea vectorului de prelucrat este echivalenta cu numarul de procesoare aritmetice. In acest caz adunarea si multiplicarea elementelor vectorului de intrare se poate face simultan. Toate arhitecturile SIMD contin o unitate de comanda si mai multe unitati de prelucrare. Comenzile date de unitatea de comanda se executa pentru toate unitatile de prelucrare active.

2. Recapitulare Lucrul cu tipul vector in Cell/B.E.
O recapitulare *din laboratorul 8 IDE Eclipse si SDK 3.0.*
O prezentare a tipurilor de date vector a fost facuta in laboratorul 7. Mai jos este prezentat un tabel cu cateva dintre aceste functii pentru vectori insotite de explicatii:

Tabelul 1. Intrisincs SPU cu mapare unu-la-unu pe Vector/SIMD Multimedia Extension

SPU Intrinsic	Vector/SIMD Multimedia Extension Intrinsic	For Data Types
spu_add	vec_add	vector operands only, no scalar operands
spu_and	vec_and	vector operands only, no scalar operands
spu_andc	vec_andc	all
spu_avg	vec_avg	all
spu_cmpeq	vec_cmpeq	vector operands only, no scalar operands
spu_cmpgt	vec_cmpgt	vector operands only, no scalar operands
spu_convtf	vec_ctf	limited scale range (5 bits)
spu_convts	vec_cts	limited scale range (5 bits)
spu_convtu	vec_ctu	limited scale range (5 bits)
spu_extract	vec_extract	all
spu_genc	vec_addc	all
spu_insert	vec_insert	all
spu_madd	vec_madd	float only
spu_mulhh	vec_mule	all
spu_mulo	vec_mulo	halfword vector operands only, no scalar operands

spu_nmsub	vec_nmsub	float only
spu_nor	vec_nor	all
spu_or	vec_or	vector operands only, no scalar operands
spu_promote	vec_promote	all
spu_re	vec_re	all
spu_rl	vec_rl	vector operands only, no scalar operands
spu_rsqrte	vec_rsqrte	all
spu_sel	vec_sel	all
spu_splats	vec_splats	all
spu_sub	vec_sub	vector operands only, no scalar operands
spu_genb	vec_subc	vector operands only, no scalar operands
spu_xor	vec_xor	vector operands only, no scalar operands

- vec = spu_splats(scal) - replica un scalar in fiecare element al unui vector
  ex: vec1111 = spu_splats((float)1)
- vec_float = spu_convtf(vec_int, scale) - converteste un vector de int intr-un vector de float
- vec = spu_add(vec_a, vec_b) - adunare de vectori element cu element
- vec = spu_sub(vec_a, vec_b) - scadere de vectori element cu element
- vec = spu_mul(vec_a, vec_b) - inmultire de vectori element cu element (produs scalar)
- vec = spu_madd(vec_a, vec_b, vec_c) - multiply (vec_a cu vec_b) si add (produsul se aduna cu vec_c);
- vec = spu_nmadd(vec_a, vec_b, vec_c) - (multiply & add) negat
- vec = spu_msub(vec_a, vec_b, vec_c) - analog madd, dar cu sub in loc de add
- vec = spu_nmsub(vec_a, vec_b, vec_c) - analog nmsub, dar cu sub in loc de add
- vec = spu_shuffle(vec_a, vec_b, vec_perm) - vec este rezultatul unui amestec (shuffle) controlat intre vec_a si vec_b; vec_perm specifica ce octeti din vec_a si din vec_b se vor afla in vectorul rezultat vec.

Tipuri de date de tip vector:
- vector [unsigned] {char, short, float, double}
ex: "vector float", "vector signed short", "vector unsigned int", ...
- Numarul de elemente din fiecare astfel de vector depinde de tipul elementelor. Trebuie tinut cont ca indiferent de tip, un vector are 128 biti. El contine astfel 4 * int, 4 * float, 8 * short, 16 * char ...
- Se poate face cast intre diferite tipuri vector
- Vectorii sunt aliniati la stanga in blocuri de dimensiunea quadword (16 octeti)

Pointeri la vectori :
- Ex: "vector float *p"
- p+1 e pointer spre urmatorul vector (16B) dupa vectorul la care refera p
- Se poate face cast din pointeri la scalari si din pointeri la tipuri vector

3. Rezumat Mailboxes

Cutiile postale sunt folosite pentru a transmite mesaje scurte intre PPU si SPU sau chiar intre SPU si SPU. Au mai fost prezentate si in laboratorul precedent, o sa rezum aici intr-un tabel functiile principale si scrierea lor. Denumirea casutelor este facuta din punctul de vedere al SPU-ului, de aceea spu_out_mbox o sa fie de la SPU catre PPU si citit din PPU.

Descriere	PPU		SPU
Se scoate primul mesaj din coada daca exista.	**int spe_out_mbox_read (spe_context_ptr_t spe, unsigned int *mbox_data,**	**int spe_out_intr_mbox_read (spe_context_ptr_t spe,**	**int spe_in_mbox_write (spe_context_ptr_t spe, unsigned int *mbox_data,**

	int count)	**unsigned int *mbox_data, int count, unsigned int behavior)**	**int count, unsigned int behavior)**
Se intoarce numarul de mesaje care asteapta in coada.	**int spe_out_mbox_status (spe_context_ptr_t spe)**	**int spe_out_intr_mbox_status (spe_context_ptr_t spe)**	**int spe_in_mbox_status (spe_context_ptr_t spe)**

Se pot folosi si semnale pentru comunicarea in SPU-uri, pentru mai multe detalii puteti consulta pagina http://cellbe.editthis.info/wiki/SPE_to_SPE_signaling.

4. Double Buffering

Supranumit si „ping-pong buffering", aceasta metoda isi propune sa optimizeze timpul petrecut asteptand transferurile sa se termine.

Modelul de lucru al metodei de double buffering:

1. SPU cere un prim transfer DMA `GET` pentru a lua o parte din problema in bufferul #1.
2. SPU cere un transfer DMA `GET` pentru a lua o parte din problema in bufferul #2.
3. SPU asteapta ca transferul pentru bufferul #1 sa se termine.
4. SPU proceseaza bufferul #1.
5. SPU cere un transfer DMA `PUT` pentru a transmite bufferul #1 apoi cere un transfer DMA `GETB` care sa se execute *dupa* `PUT` pentru a reumple bufferul #1 cu urmatoarea bucata de memorie din problema.
6. SPU asteapta ca transferul pentru bufferul #2 sa se termine.
7. SPU proceseaza bufferul #2.
8. SPU cere un transfer DMA `PUT` pentru a transmite bufferul #2 apoi cere un transfer DMA `GETB` care sa se execute *dupa* `PUT` pentru a reumple bufferul #2 cu urmatoarea bucata de memorie din problema.
9. Se repeta de la pasul 3 pana cand nu mai exista date.
10. Asteapta ca si ultimele transferuri sa se termine.

Exemplu

Acest exemplu face adunarea a doi vectori intr-un al treilea. SPU se foloseste de double buffering pentru transferuri de la PPU la SPU. Rezltatul este trimis inapoi tot prin DMA.

array_add.h

```
#ifndef __array_add_h__

#define __array_add_h__

#define ARRAY_SIZE 5120

#define NUM 512

#define BUF_SIZE_BYTES (256*4)

#define BUF_SIZE (256)

typedef struct _control_block

{

unsigned int a;

unsigned int b;

unsigned int c;
```

```
unsigned int size;

unsigned char pad[112];

} control_block;

#endif
```

PPU

```
#include <stdio.h>

#include <stdlib.h>

#include <libspe2.h>

#include <errno.h>

#include "array_add.h"

#define MY_ALIGN(_my_var_def_, _my_al_) _my_var_def_
__attribute__((__aligned__(_my_al_)))

MY_ALIGN(float array_a[ARRAY_SIZE],128);

MY_ALIGN(float array_b[ARRAY_SIZE],128);

MY_ALIGN(float array_c[ARRAY_SIZE],128);

MY_ALIGN(control_block cb, 128);

extern spe_program_handle_t array_add_spu;

int status;

void init_array() {

int i;

for(i=0;i<ARRAY_SIZE; i++) {

array_a[i] = (float)(i * NUM);

array_b[i] = (float)((i * NUM)*2);

array_c[i] = 0.0f;

}
```

```
}

int main(void){

int i;

init_array();

/* complete control block*/

cb.a = (unsigned int) &array_a[0];

cb.b = (unsigned int) &array_b[0];

cb.c = (unsigned int) &array_c[0];

cb.size = ARRAY_SIZE;

spe_context_ptr_t spe_context = spe_context_create( 0, NULL );

spe_program_load( spe_context, &array_add_spu );

unsigned int runflags = 0;

unsigned int entry = SPE_DEFAULT_ENTRY;

spe_context_run( spe_context, &entry,
runflags, &cb, NULL, NULL );

spe_context_destroy( spe_context );

printf("Array Addition completes. Verifying results...\n");

for (i=0; i<ARRAY_SIZE; i++) {

if (array_c[i] != (float)((i * NUM)*3)) {

printf("ERROR in array addition\n");

return -1;

}

}

printf("Correct!\n");
```

```
return 0;

}
```

SPU

```
#include <stdio.h>

#include <spu_mfcio.h>

#include "../array_add.h"

#define MY_ALIGN(_my_var_def_, _my_al_) _my_var_def_

__attribute__((__aligned__(_my_al_)))

MY_ALIGN(control_block cb, 128);

MY_ALIGN(float dataBuf[BUF_SIZE * 6], 128);

float *aData[2], *bData[2], *cData[2];

int main(unsigned long long speid, unsigned long long argp, unsi
gned long long envp){

int i, j,size;

unsigned int src_tag, dest_tag, src_mask, dest_mask;

float *aSrcPtr, *aLocalPtr, *bSrcPtr, *bLocalPtr, *cSrcPtr, *cLo
calPtr;

/* Setup data buffers */

aData[0] = dataBuf; aData[1] = dataBuf + BUF_SIZE;

bData[0] = dataBuf + BUF_SIZE*2;
bData[1] = dataBuf + BUF_SIZE*3;

cData[0] = dataBuf + BUF_SIZE*4;
cData[1] = dataBuf + BUF_SIZE*5;

/* setup src DMAs to use tag 31, dest DMAs to use tag 30*/

src_tag = 31; src_mask = 1<<src_tag;

dest_tag = 30; dest_mask = 1<<dest_tag;
```

```
/* First: DMA our control block into local store */

mfc_get(&cb, argp, sizeof(cb), src_tag, 0, 0);

mfc_write_tag_mask(src_mask);

mfc_read_tag_status_all();

/* obtain information from control block */

size = cb.size/BUF_SIZE;

aSrcPtr = (float *)cb.a;

bSrcPtr = (float *)cb.b;

cSrcPtr = (float *)cb.c;

/* Now we can kick off our first DMAs. */

/* This will load the first section of our A and B data arrays
into the local store */

mfc_get((void *) aData[0], (unsigned int)aSrcPtr, (BUF_SIZE*size
of(float)), src_tag, 0, 0);

aSrcPtr += BUF_SIZE;

mfc_get((void *) bData[0], (unsigned int)bSrcPtr, (BUF_SIZE*size
of(float)), src_tag, 0, 0);

bSrcPtr += BUF_SIZE;

for (j=0; j<size; j++) {

mfc_write_tag_mask(src_mask);

mfc_read_tag_status_all();

/* Now that we have some data to work on, we can kick off
transfering the */

/* next block of data over while we work on the first block */

mfc_get((void *) aData[(j+1)&1], (unsigned int)aSrcPtr, (BUF_SIZ
E*sizeof(float)), src_tag, 0, 0);
```

```
aSrcPtr += BUF_SIZE;

mfc_get((void *) bData[(j+1)&1], (unsigned int)bSrcPtr, (BUF_SIZ
E*sizeof(float)), src_tag, 0, 0);

bSrcPtr += BUF_SIZE;

aLocalPtr = aData[j&1];

bLocalPtr = bData[j&1];

cLocalPtr = cData[j&1];

/*Perform the data add*/

for(i=0; i<BUF_SIZE; i++, aLocalPtr++, bLocalPtr++,
cLocalPtr++) {

*cLocalPtr = *aLocalPtr + *bLocalPtr;

}

mfc_write_tag_mask(dest_mask);

mfc_read_tag_status_all();

/* Finally, DMA computed array back to main memory*/

mfc_put((void *) cData[j&1], (unsigned int)cSrcPtr, (BUF_SIZE*si
zeof(float)), dest_tag, 0, 0);

cSrcPtr += BUF_SIZE;

}

mfc_write_tag_mask(dest_mask);

mfc_read_tag_status_all();

return 0;

}
```

5. HANDS - ON

Creati un program pentru arhitectura cell care scaneaza mailuri pentru a identifica amenintari teroriste.

1) Creati un Proiect cu 4 Spu-uri si un Ppu care are ca efect ca fiecare SPu spune Hello World si id-ul sau.
2) Modificati proiectul in asa fel incat PPU-ul sa transfere prin dma fiecarui spu un sir de caractere de 256 de caractere, diferite pe care acestea le afiseaza.
3) Modificati proiectul in asa fel incat Spu-urile sa caute cuvintul “BOMB” in sirul primit ca parametru si sa afiseze

cand il gasesc.
4) Modificati proiectul in asa fel incat Spu-urile sa transmita printr-un mesaj PPU-ului indicele la care au gasit “BOMB” si sa astepte un raspuns de la acesta “ok” si sa contiune cautare. Cand au terminat de cautat intorc -1. Transferul de mesaje este busy-waiting.
5) Modificati proiectul in asa fel incat Spu-urile sa comunice cu PPU-ul prin evenimente.
6) Modificati proiectul in asa fel incat daca un Spu gaseste “BOMB” atunci sa trimita un mesaj celorlalte Spu-uri si ele sa caute de la capat ore sub forma XX:XX (pentru simplificare cautati 12:00 ca nu trebuie sa parsati siruri)