VNIVERSITAT DE VALÈNCIA
Escola Tècnica Superior d'Enginyeria
Departament d´Informàtica

Laboratorio de Arquitecturas Avanzadas con Cell y PlayStation 3

Fernando Pardo

Ampliación de Arquitectura de Computadores

Introducción

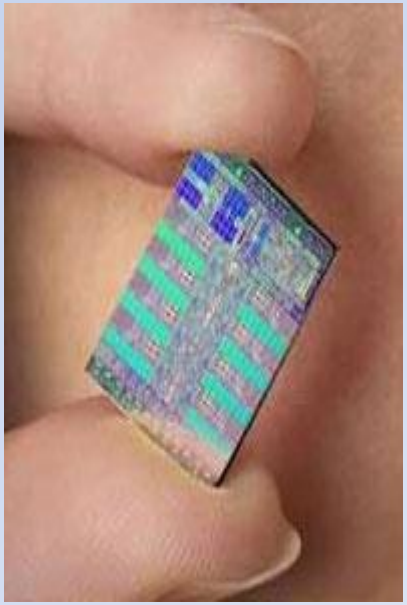
- Se presenta la experiencia de utilización del procesador Cell (PlayStation 3) para la docencia de arquitecturas avanzadas.
- Arquitectura y ventajas del procesador Cell.
- Clúster de PS3 para las prácticas de laboratorio.
- Programación de las PS3.

Procesador Cell

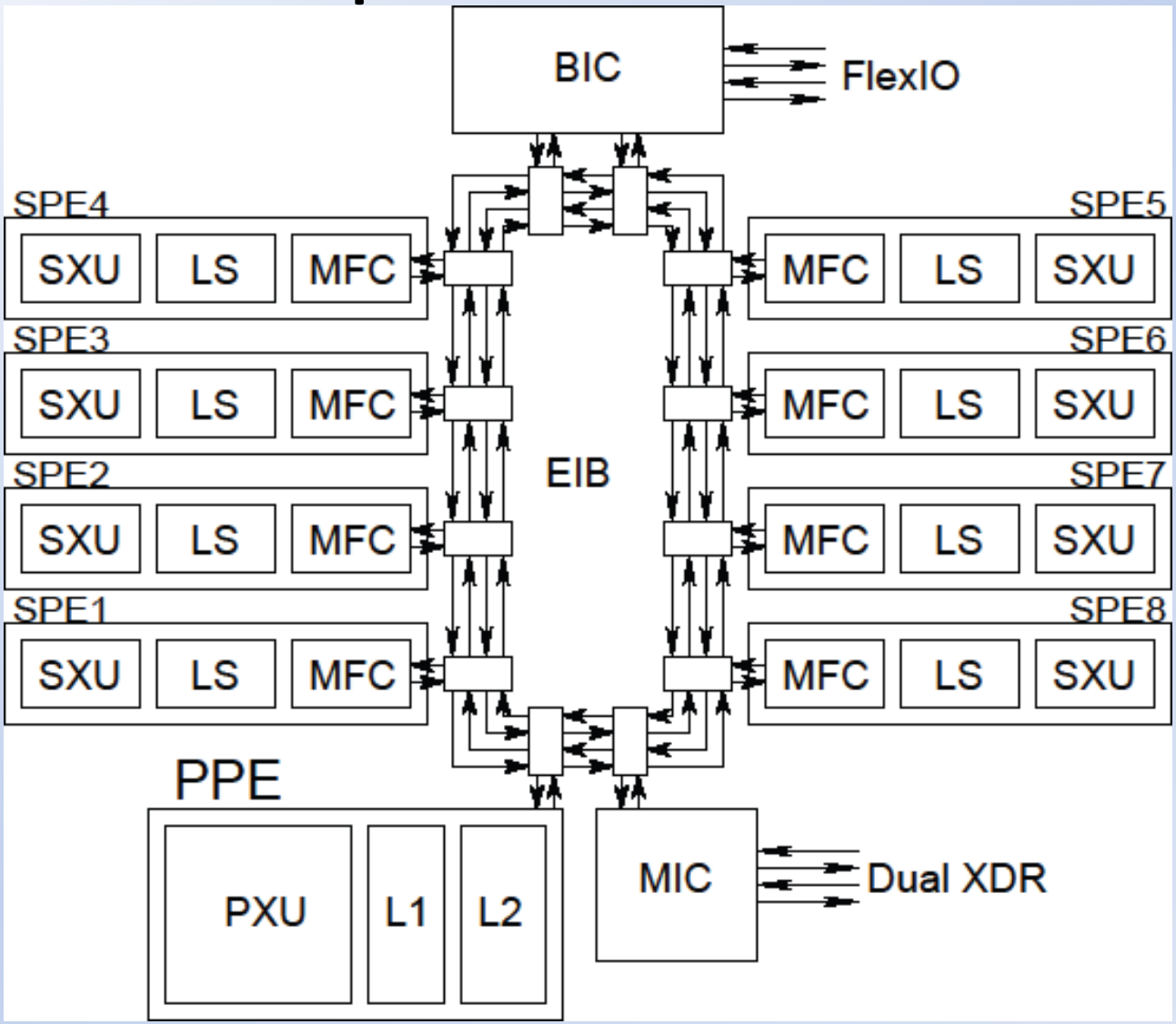
- El procesador Cell es el resultado de la unión entre IBM, Sony y Toshiba para crear un procesador capaz de hacer frente a las fuertes cargas de trabajo de cálculo intensivo de aplicaciones con contenido multimedia.
- Está formado por una arquitectura IBM PowerPC y múltiples unidades de cálculo vectorial de tipo SIMD (una instrucción-múltiples datos).

Componentes del Cell

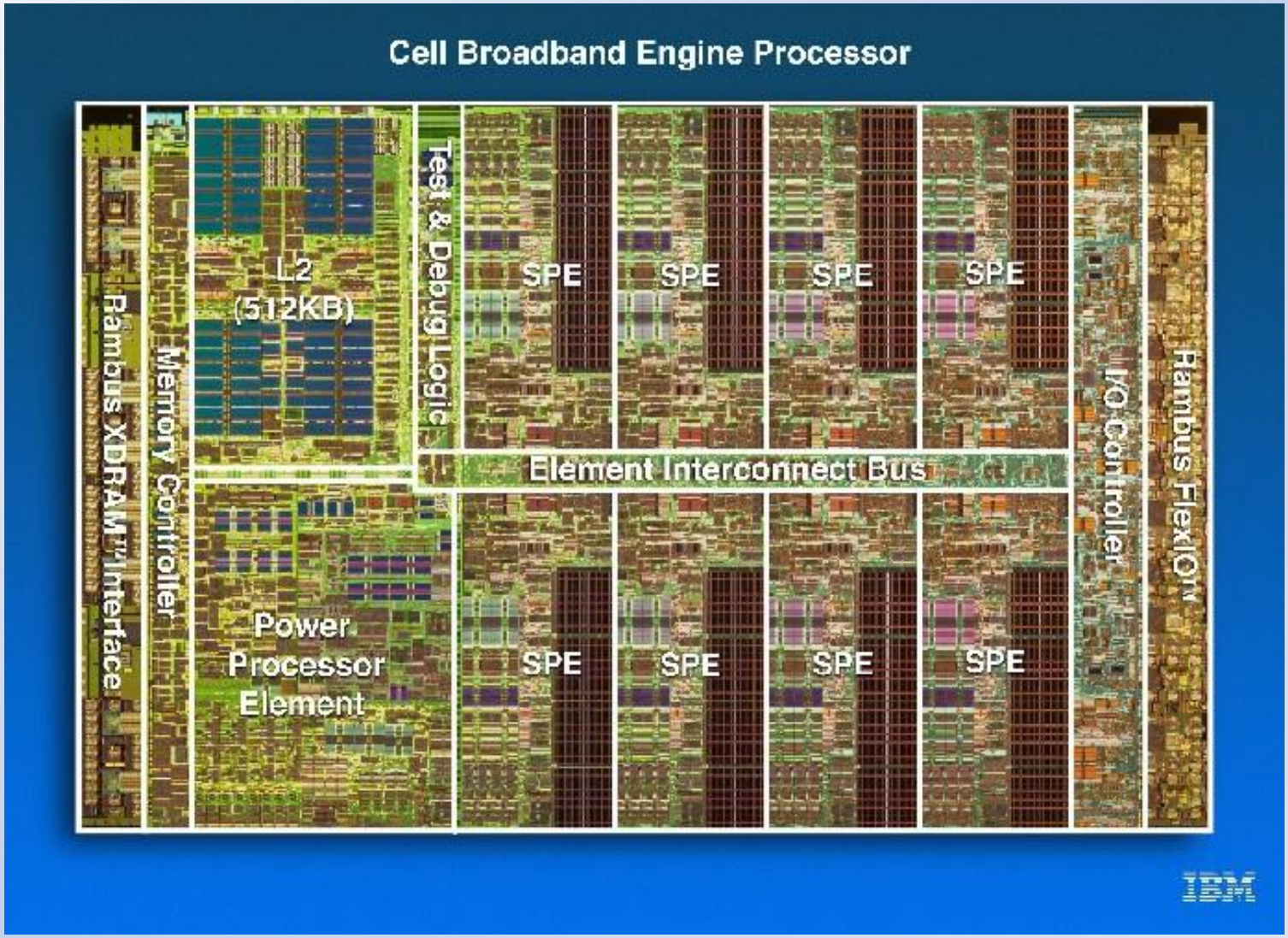
- 1 Procesador típico basado en la arquitectura PowerPC de IBM (PowerPC Processing Element, PPE).
- 8 Procesadores especiales (Synergistic Processor Element, SPE).
- 1 Bus de Interconexión (Element Interconnect Bus, EIB).
- Interfaz exterior:
  - Memoria: módulo BIC -> Flex IO.
  - Periféricos: módulo MIC -> Dual XDR.



Arquitectura Cell



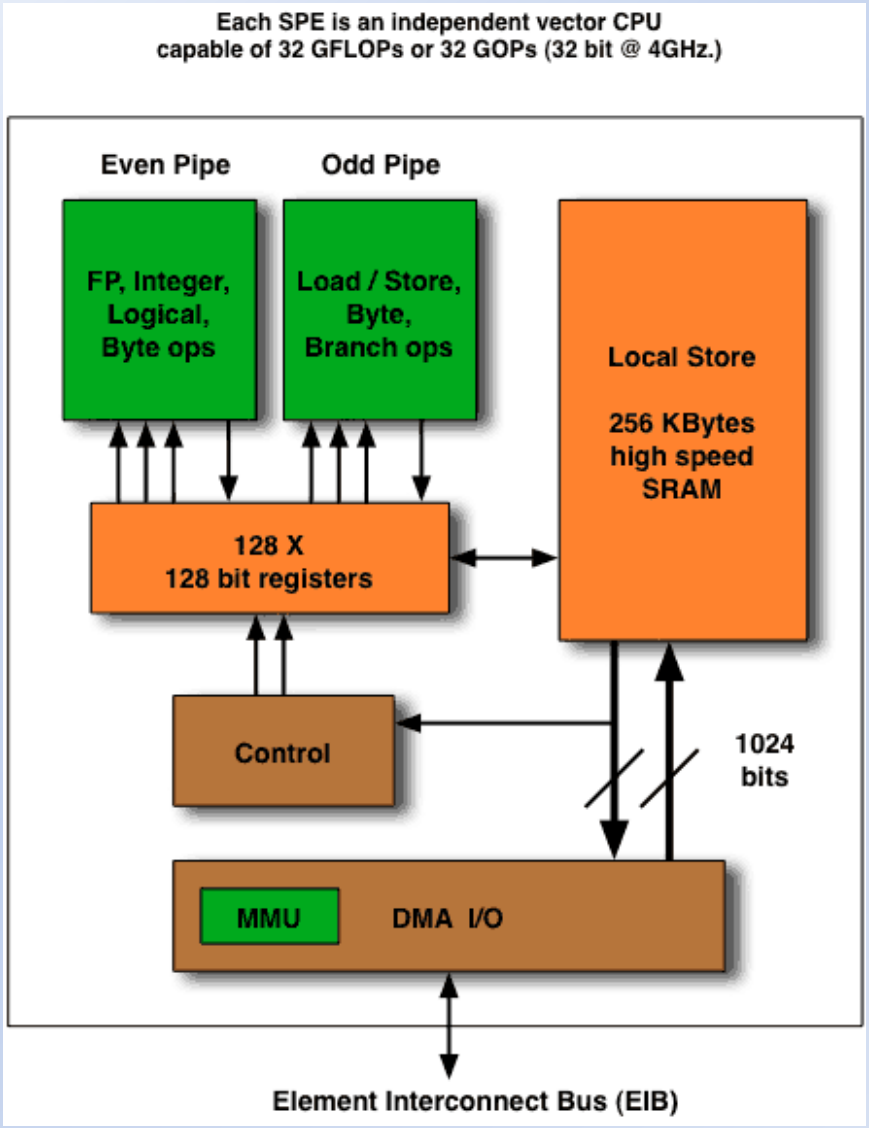
Arquitectura Cell



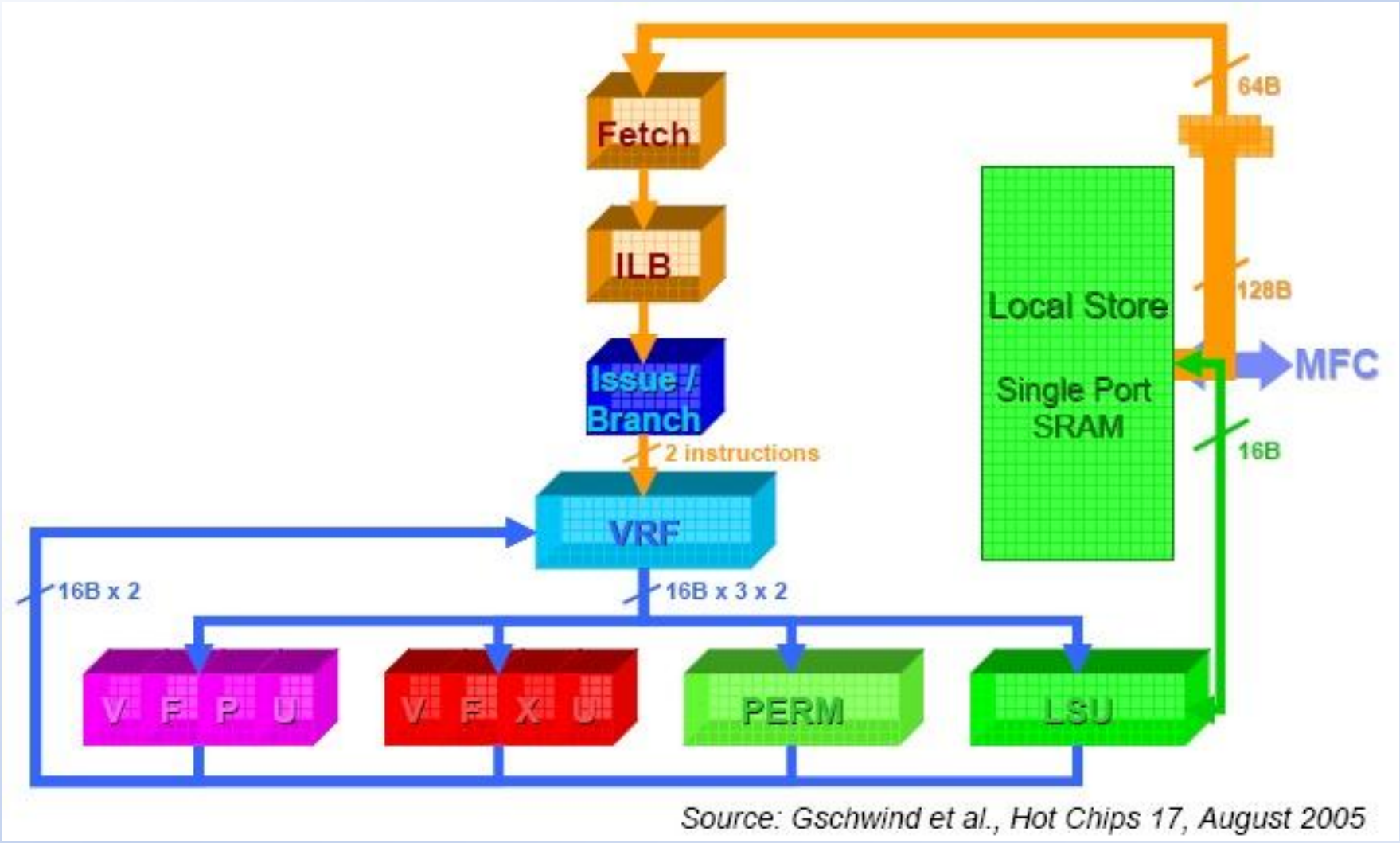
Arquitecturas en el Cell

- Procesado vectorial SIMD:
  - Cada SPE tiene una unidad SIMD a nivel de instrucción.
  - Los 8 SPE se pueden programar para realizar la misma tarea como si fuera un computador matricial.
- Multicomputador:
  - Los SPE forman un "mini-multicomputador" pues cada uno tiene su memoria local y proceso independiente. De trata de un SoC.
- Encauzamiento o Stream Processing:
  - La red de interconexión en anillo permite el encauzado de los SPE entre sí.

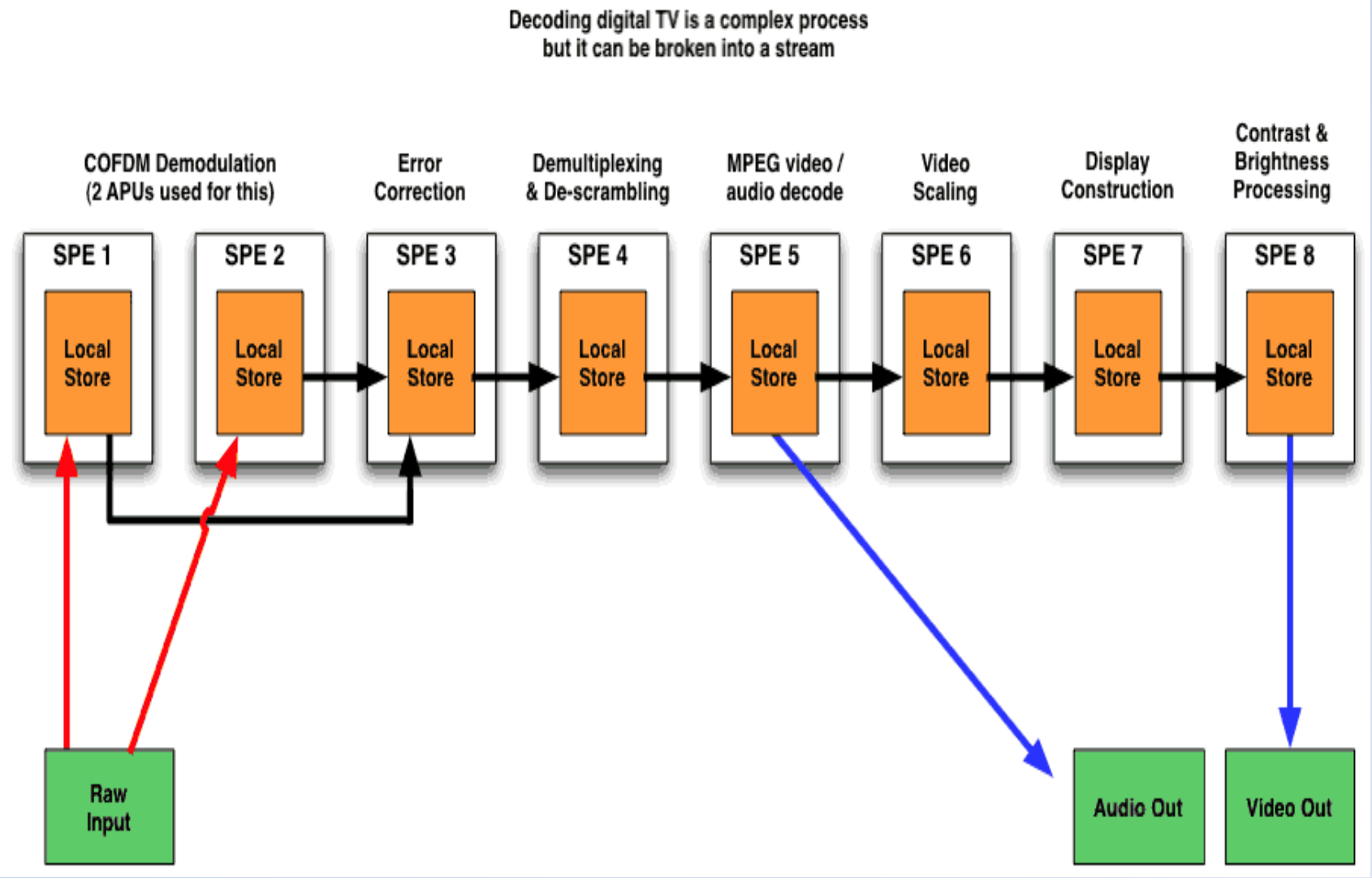
Arquitectura de un SPE del Cell



Arquitectura de un SPE del Cell



Cell stream processing



Clúster de PlayStation 3

- 13 PlayStation3.
- 13 procesadores Cell.
- Cada Cell tiene 1 PowerPC y 6 SPE.
- Yellow Dog Linux.
- Interconexión Ethernet 100 Mb.

Programación del Cell: compilación

```
OPT = -W -Wall
SPUFLAG = $(OPT)
PPUFLAG = -D_REENTRANT $(OPT) -m64

LD = -lspe2 -lpthread

prog: spe.c ppe.c

# Compilacion del codigo del SPE
    spu-gcc spe.c $(SPUFLAG) -o spe

# Exportacion del codigo del SPE en la variable spe_code
    ppu-embedspu spe_code spe spe.o

# Compilacion del codigo del PPE
    gcc ppe.c spe.o $(PPUFLAG) $(LD) -o prog
```

Programación del Cell

- Creación de un contexto:
  - El contexto contiene el programa que ejecuta cada SPE.
  - El PPE asigna un hilo a cada contexto.
- Ejecución:
  - El PPE inicia los programas de cada SPE.
- Transferencia de datos:
  - Iniciadas por el PPE.
  - Iniciadas por el SPE.

Programación del Cell: vector.h

```
#define MAX_SPE 6
#define LON_VECTOR 3600  /* que sea multiplo
  de 120 para alineamiento, más de 3600 no
  vale pues ocupa mas de 16k. */

/* La longitud de la estructura debe ser
  multiplo de 16 bytes, rellenar con "extras"
  para que tenga la longitud adecuada. */
typedef struct {unsigned long long op1;
  unsigned long long res; int lon; int
  extras[3];} argumento;
```

Programación del Cell: ppe.c

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <libspe2.h>
#include <sys/times.h>
#include "vector.h"

extern spe_program_handle_t spe_code;

void * thread(void * data)
{
      unsigned int entry;
      spe_context_ptr_t ctx;
      argumento *datos = (argumento *)data;

      /* Inicializacion del contexto */
      ctx = spe_context_create(0,NULL);
      entry = SPE_DEFAULT_ENTRY;
      spe_program_load(ctx,&spe_code);

      /* Ejecuta el contexto sobre un SPE, el hilo se bloquea hasta
      el fin de la ejecucion */
      spe_context_run(ctx,&entry,0,(void *)datos, NULL,NULL);
      spe_context_destroy(ctx);
      return (void *) 0;
}
```

```
argumento datosHilo[MAX_SPE]
__attribute__((aligned(16)));
int operando1[LON_VECTOR] __attribute__((aligned(16)));
int resultado[LON_VECTOR] __attribute__((aligned(16)));

/*lista de los identificadores de pthreads*/
pthread_t threads[MAX_SPE];

if (argc < 2)
{
        printf("Se requiere al menos un parámetro con el
número de SPEs a usar.\n");
        return -1;
}
nbspe = atoi(argv[1]);
if (nbspe>6)
{
        printf("Error: numero maximo de SPE disponibles =
6\n");
        return -1;
}
```

```
int main(int argc, char ** argv)
{
      int i,nbspe;
      struct tms tmsaux;
      clock_t ini,fin;
```

Programación del Cell: ppe.c

```
/* Creación de los vectores a operar */
for (i=0;i<LON_VECTOR;i++)
{
     operando1[i]=i;
}

/* Creación de los argumentos de los hilos */
// ATENCION: LON_VECTOR debe ser múltiplo de 4, 5 y 6
     (multiplo de 120)
for(i=0;i<nbspe;i++)
{
   int trozo; // Los trozos deben ser múltiplos de 16 bytes
     (enteros multiplos de 4)
   trozo=LON_VECTOR/nbspe;
   datosHilo[i].op1=(unsigned long long)&(operando1[i*trozo]);
   datosHilo[i].res=(unsigned long long)&(resultado[i*trozo]);
   datosHilo[i].lon=trozo;
}

ini=times(&tmsaux);   // Inicio del tiempo de ejecucion
```

```
/* Creacion de los hilos */
for(i=0;i<nbspe;i++)
{
  /*Iniciar un hilo por cada SPE */
  if (pthread_create(threads+i,NULL,thread,&(datosHilo[i])) !=
     0)
  {
     perror("Thread create.");
     return -1;
  }
}

/* Esperar el fin de cada hilo, la funcion pthread_join espera
     el fin del thread, y recoge el valor devuelto por este en
     el segundo argumento */
for(i=0;i<nbspe;i++)
{
  pthread_join(threads[i],NULL);
}

fin=times(&tmsaux);  // Final del tiempo de ejecución
```

Programación del Cell: ppe.c

```
/* Muestra resultados */

// comprueba resultado
for (i=0;i<LON_VECTOR;i++)
{
  if (operando1[i]-1!=resultado[i])
    printf("%3d -> %3d\n",operando1[i],resultado[i]);
}

printf("\nTiempo: %ld\n",   fin-ini);

printf("\nFin del programa.\n");

return 0;
```

Programación del Cell: spe.c

```
#include <stdio.h>
#include <spu_mfcio.h>
#include "vector.h"

int main(unsigned long long spe_id, unsigned long long arg, unsigned long long env)
{
int i,j;
uint32_t tag;
argumento dato __attribute__((aligned(16)));
int op1[LON_VECTOR] __attribute__((aligned(16)));
int res[LON_VECTOR] __attribute__((aligned(16)));


/* Trae los parametros de la mem principal (arg) al SPE (dato)*/

 if((tag=mfc_tag_reserve())==MFC_TAG_INVALID) // reserva una etiqueta de transaccion
 {
   printf("SPE: ERROR - No se puede reservar un tag de trasaccion.\n");
   return 1;
 }
 mfc_get(&dato,arg,sizeof(argumento),tag,0,0);
 mfc_write_tag_mask(1<<tag);
 mfc_read_tag_status_all();
```

Programación del Cell: spe.c

```
/* Trae los parametros de la mem principal (dato.op1) al SPE (op1)*/
mfc_get(op1,dato.op1,sizeof(int)*dato.lon,tag,0,0);
mfc_write_tag_mask(1<<tag);
mfc_read_tag_status_all();

/* Operaciones a realizar */
// bucle para hacer tiempo
for (j=0;j<200000;j++)
{
  for (i=0;i<dato.lon;i++)
  {
     res[i]=op1[i]-1;
  }
}

/* Copiar el resultado local de res en la memoria principal a la direccion dato.res*/
mfc_put(res,dato.res,dato.lon*sizeof(int),tag,0,0);
mfc_write_tag_mask(1<<tag);
mfc_read_tag_status_all();

mfc_tag_release(tag); // libera la etiqueta de transaccion

return 0;
```

Programación SIMD del SPE

- Los SPE tienen una unidad de ejecución matricial.
- Las instrucciones SIMD operan sobre datos fijos de 128 bits, lo que equivale a vectores de 2 dobles, 4 flotantes o enteros, 8 enteros cortos, etc.
- Se utiliza el tipo especial "vector" para especificar el tipo de datos sobre el que aplicar las instrucciones.
- Se puede utilizar directamente instrucciones SIMD de la biblioteca o dejar que el compilador las aplique.

Experimentos de laboratorio

- Sesión de introducción:
  - Operación simple con un vector largo.
  - Reparto de la tarea entre varios SPE.
  - Utilización de instrucciones SIMD en cada SPE.