



# Hacking the Cell Architecture

Rodrigo Rubira Branco (BSDaemon)  
Oger Systems Principal Security Researcher  
Scanit R&D Lab  
rodrigo \***noSPAM**\* scanit dot net  
bsddaemon \***noSPAM**\* risesecurity dot org

# Agenda



In-depth technical analysis of the cell architecture

Programming the cell architecture

- Running a program inside the spu
- Using DMA in spu

Security issues in Cell

- Memory protection inside spu
- DMA

Exploiting a vulnerable program running inside the spu

Injectin our code – shellcode development and issues

What else can we do?

Acknowledges

# Try it!



I'll try to not be boring!

This presentation have been splitted in three portions:

- Cell Theory – Inside the Cell Architecture Internals
- Cell Programming – Giving an overview of the programming resources that we will use
- Cell Samples – Those are the live samples prepared for this presentation

# What is Cell?



The so called 'Cell Broadband Engine Architecture (TM)'

It's a powerful, new and really improved computer architecture

Used in the Playstation 3 and also in big blade Machines

# Cell History



SONY

- IBM, SCEI/Sony, Toshiba Alliance formed in 2000
- Design Center opened in March 2001
  - Based in Austin, Texas
- Single CellBE operational Spring 2004
- 2-way SMP operational Summer 2004
- February 7, 2005: First technical disclosures
- October 6, 2005: Mercury announces Cell Blade
- November 9, 2005: Open source SDK & simulator published
- November 14, 2005: Mercury announces Turismo Cell offering
- February 8, 2006: IBM announced Cell Blade
- July 17, 2006: SDK 1.1 available

TOSHIBA

IBM®





# Cell Workarounds



- Power Wall
  - Limits in CMOS technology
  - Hard limit to acceptable system power
- Memory Wall
  - Processor frequency vs. DRAM memory latency
- Frequency Wall
  - Diminishing returns from deeper pipelines

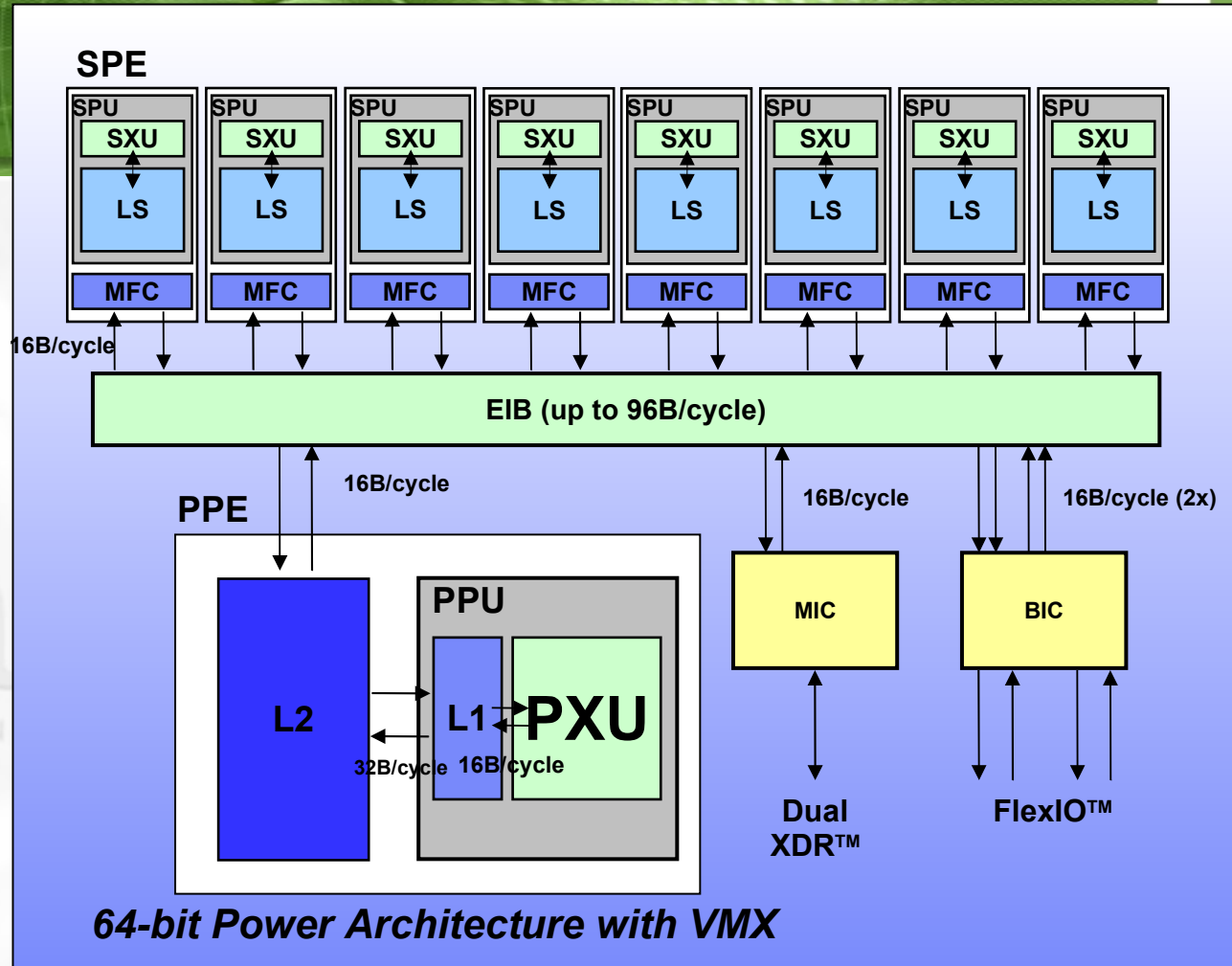
# Cell Basic Design Concept



- Compatibility with 64b Power Architecture™
  - Builds on and leverages IBM investment and community
- Increased efficiency and performance
  - Attacks on the “Power Wall”
    - **Non Homogenous Coherent Multiprocessor**
    - **High design frequency @ a low operating voltage with advanced power management**
  - Attacks on the “Memory Wall”
    - **Streaming DMA architecture**
    - **3-level Memory Model: Main Storage, Local Storage, Register Files**
  - Attacks on the “Frequency Wall”
    - **Non Homogenous Coherent Multiprocessor**
    - **Highly optimized implementation**
    - **Large shared register files and software controlled branching to allow deeper pipelines**
- Interface between user and networked world
  - Image rich information, virtual reality
  - Flexibility and security
  - Multi-OS support, including RTOS / non-RTOS

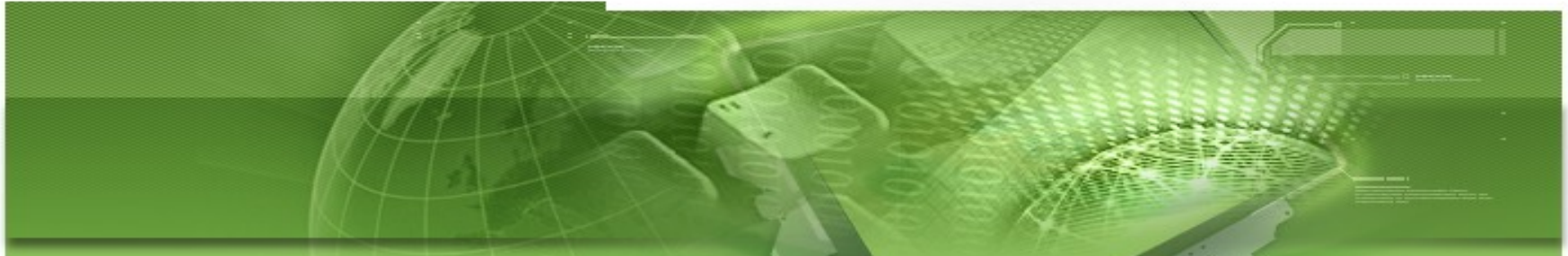
# Cell Architecture Components

- Heterogeneous multi-core system architecture
  - Power Processor Element for control tasks
  - Synergistic Processor Elements for data-intensive processing
- Synergistic Processor Element (SPE) consists of
  - Synergistic Processor Unit (SPU)
  - **Synergistic Memory Flow Control (MFC)**
    - Data movement and synchronization
    - Interface to high-performance Element Interconnect Bus





# Cell Processor Components



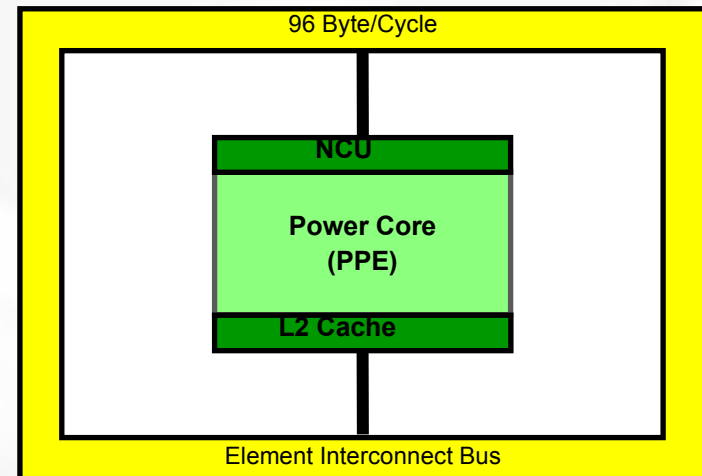
## Power Processor Element (PPE):

- General purpose, 64-bit RISC processor (PowerPC AS 2.0.2)
- 2-Way hardware multithreaded
- L1 : 32KB I ; 32KB D
- L2 : 512KB
- Coherent load / store
- VMX-32
- Realtime Controls
  - Locking L2 Cache & TLB
  - Software / hardware managed TLB
  - Bandwidth / Resource Reservation
  - Mediated Interrupts

## Element Interconnect Bus (EIB):

- Four 16 byte data rings supporting multiple simultaneous transfers per ring
- 96Bytes/cycle peak bandwidth
- Over 100+ simultaneous bus transactions
- Each EIB bus data port supports 25.6Gbytes/sec (assuming 3.5 Ghz core frequency) in each direction

*In the Beginning  
– the solitary Power Processor*



*Custom Designed  
– for high frequency, space,  
and power efficiency*

# Cell Processor Components

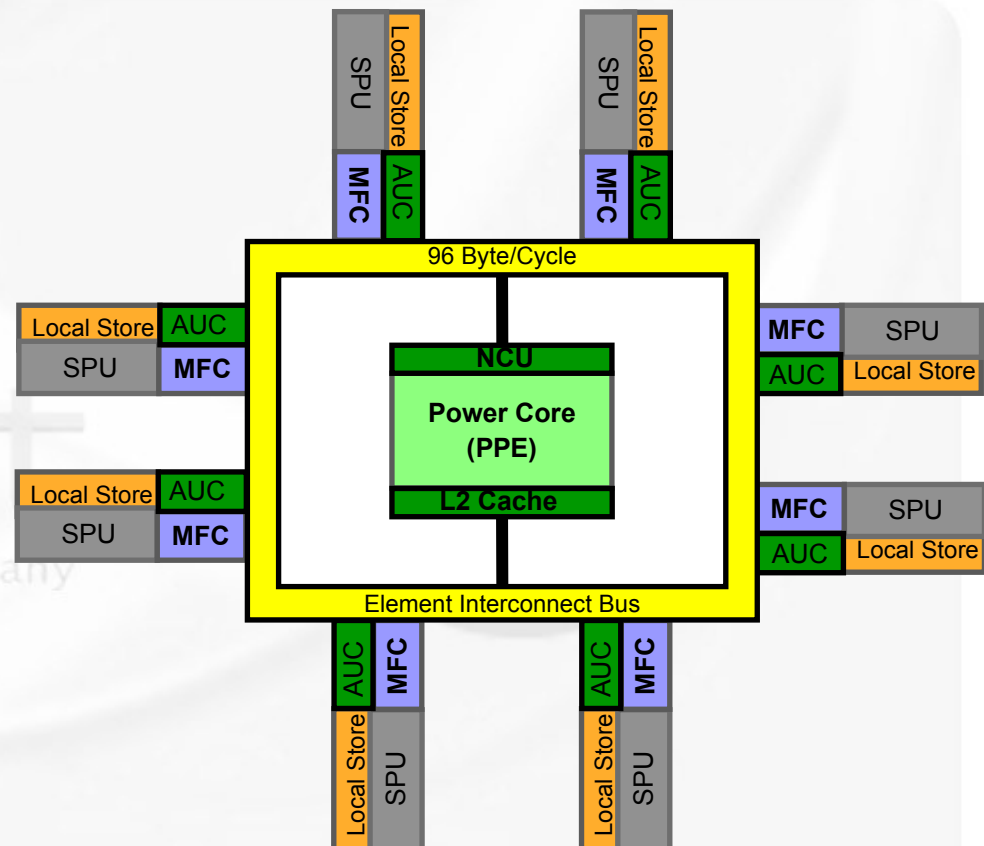


## Synergistic Processor Element (SPE):

- Provides the computational performance
- Simple RISC User Mode Architecture
  - Dual issue VMX-like
  - Graphics SP-Float
  - IEEE DP-Float
- Dedicated resources: unified 128x128-bit RF, 256KB Local Store
- Dedicated DMA engine: Up to 16 outstanding requests

## Memory Management & Mapping

- SPE Local Store aliased into PPE system memory
- MFC/MMU controls / protects SPE DMA accesses
  - Compatible with PowerPC Virtual Memory Architecture
  - SW controllable using PPE MMIO
- DMA 1,2,4,8,16,128 -> 16Kbyte transfers for I/O access
- Two queues for DMA commands: Proxy & SPU



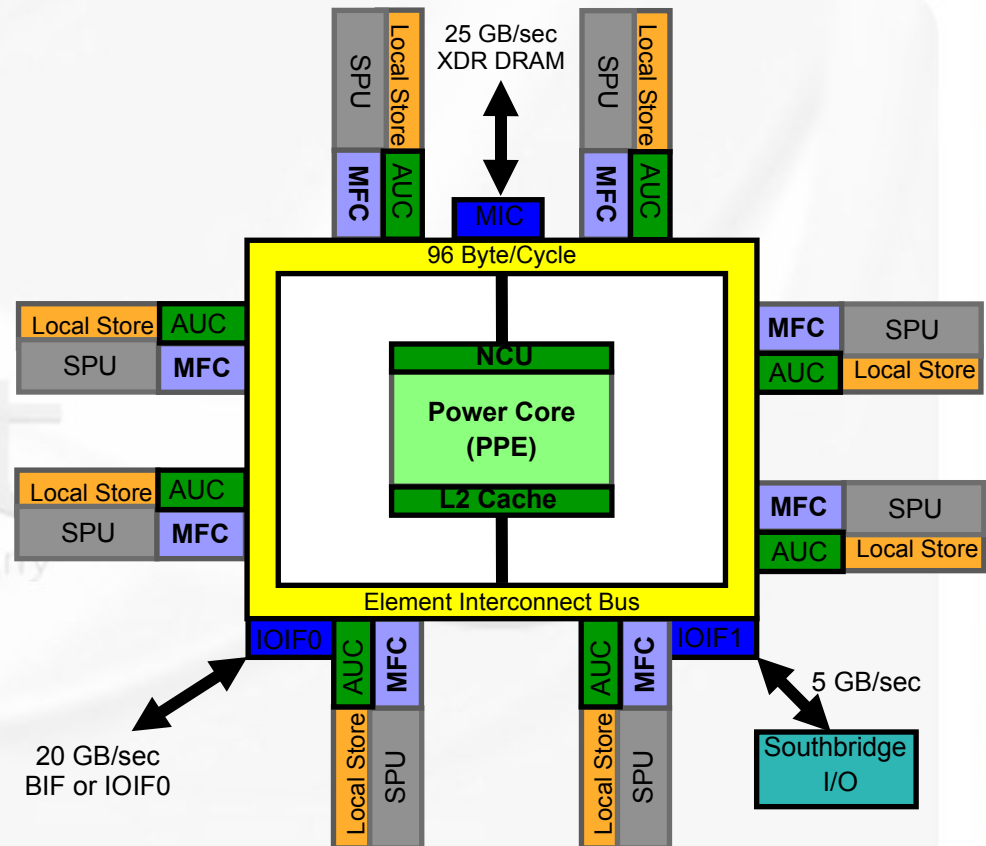
# Cell Processor Components

## Broadband Interface Controller (BIC):

- Provides a wide connection to external devices
- Two configurable interfaces (60GB/s @ 5Gbps)
  - Configurable number of bytes
  - Coherent (BIF) and / or I/O (IOIFx) protocols
- Supports two virtual channels per interface
- Supports multiple system configurations

## Memory Interface Controller (MIC):

- Dual XDR™ controller (25.6GB/s @ 3.2Gbps)
- ECC support
- Suspend to DRAM support



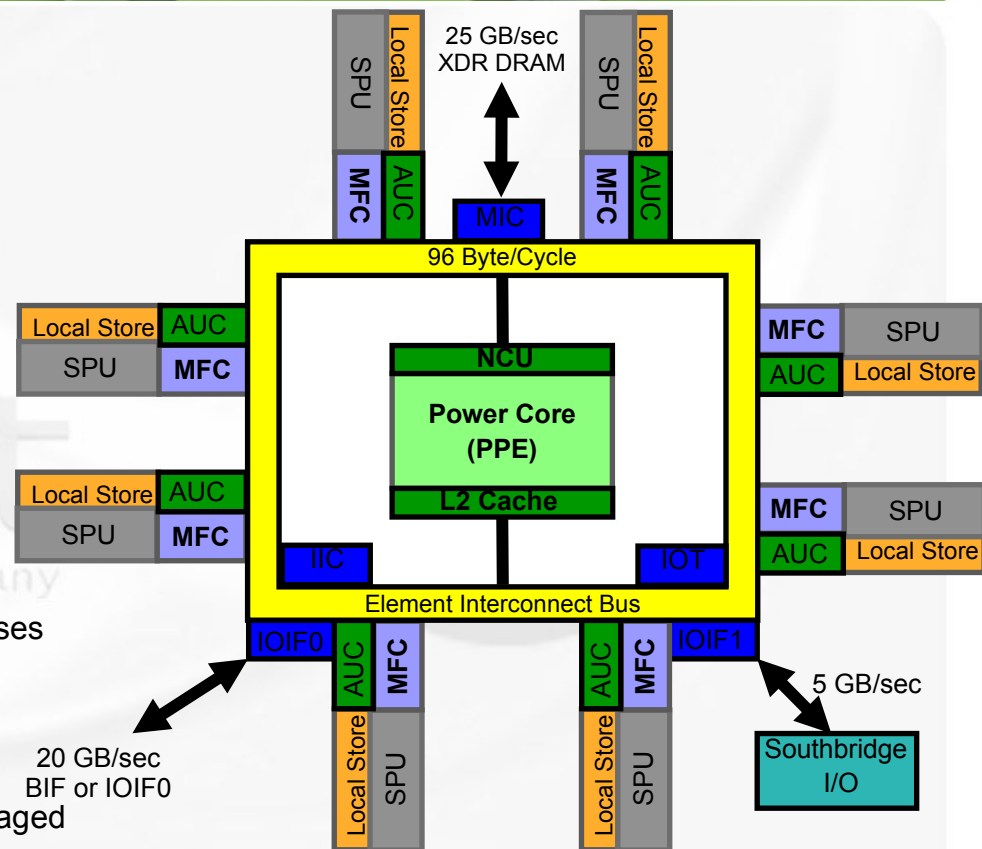
# Cell Processor Components

## Internal Interrupt Controller (IIC)

- Handles SPE Interrupts
- Handles External Interrupts
  - From Coherent Interconnect
  - From IOIF0 or IOIF1
- Interrupt Priority Level Control
- Interrupt Generation Ports for IPI
- Duplicated for each PPE hardware thread

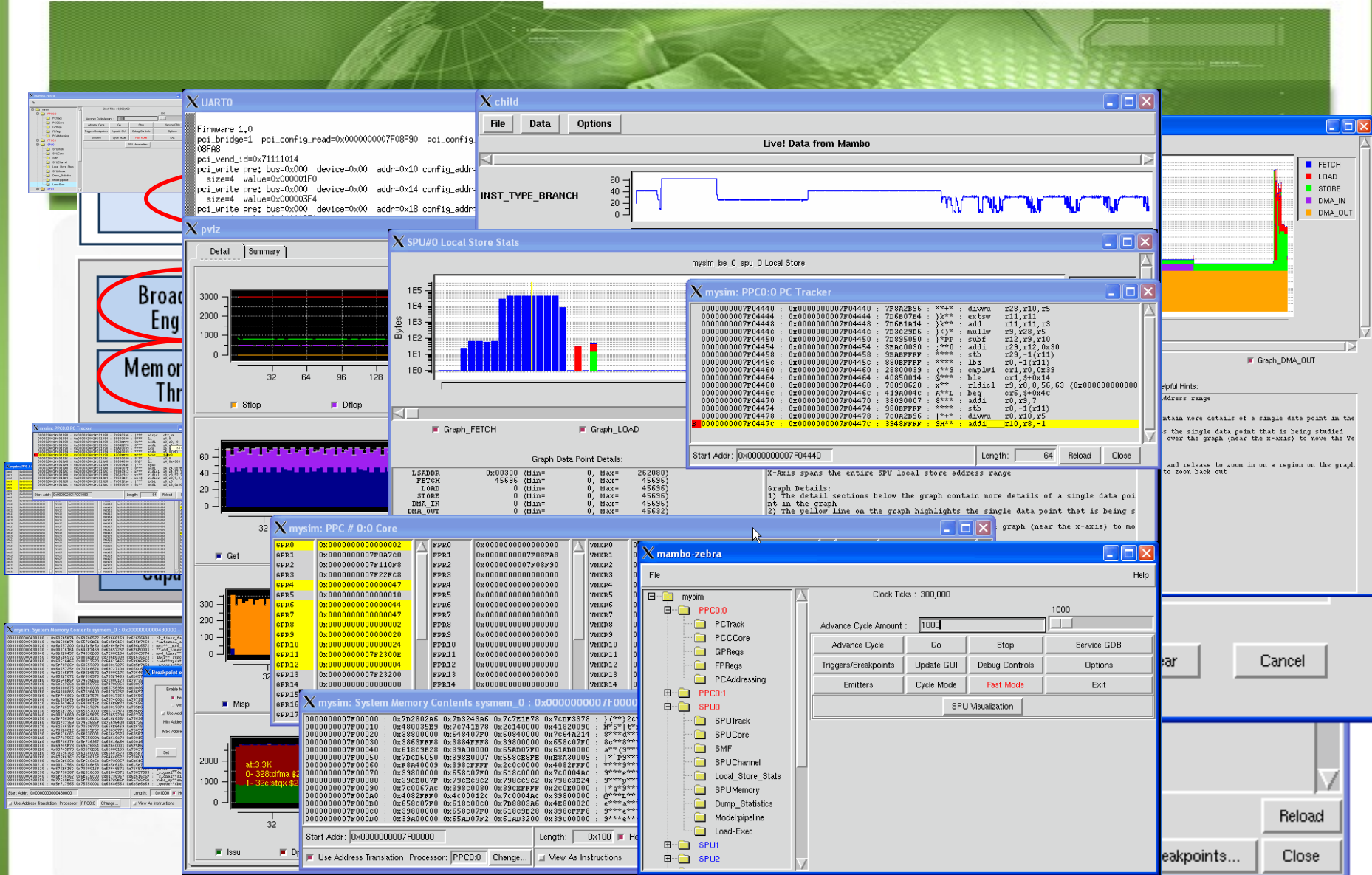
## I/O Bus Master Translation (IOT)

- Translates Bus Addresses to System Real Addresses
- Two Level Translation
  - I/O Segments (256 MB)
  - I/O Pages (4K, 64K, 1M, 16M byte)
- I/O Device Identifier per page for LPAR
- IOST and IOPT Cache – hardware / software managed





# Debugging Cell





# Linux on Cell



- Provided as patched to the 2.6.15 PPC64 Kernel
  - Added heterogeneous lwp/thread model
    - SPE thread API created (similar to pthreads library)
    - User mode direct and indirect SPE access models
    - Full pre-emptive SPE context management
    - spe\_ptrace() added for gdb support
    - spe\_schedule() for thread to physical SPE assignment
      - currently FIFO – run to completion
  - SPE threads share address space with parent PPE process (through DMA)
    - Demand paging for SPE accesses
    - Shared hardware page table with PPE

# Linux on Cell



- PPE proxy thread allocated for each SPE thread to:
  - Provide a single namespace for both PPE and SPE threads
  - Assist in SPE initiated C99 and POSIX-1 library services
- SPE Error, Event and Signal handling directed to parent PPE thread
- SPE elf objects wrapped into PPE shared objects with extended gld
- All patches for Cell in architecture dependent layer (subtree of PPC64)
- The Play3 contains:
  - 8 SPU
  - 1 reserved for redundancy
  - 1 used as hypervisor when using a Custom OS (our case)



# Extensions to Linux

PPC32 Apps.

Cell32 Workloads

Cell64 Workloads

PPC64 Apps.

Programming Models Offered: RPC, Device Subsystem, Direct/Indirect Access  
Hetergenous Threads -- Single SPU, SPU Groups, Shared Memory

SPE Management Runtime  
Library (64-bit)

std. PPC32  
elf interp

SPE Object Loader  
Services

std. PPC64  
elf interp

32-bit GNU Libs (glibc,etc)

ILP32 Processes

64-bit GNU Libs (glibc)

LP64 Processes

System Call Interface

exec Loader

File System  
Framework

Device  
Framework

Network  
Framework

Streams  
Framework

SPU Management  
Framework

Misc format bin  
SPU Object  
Loader Extension

Privileged  
Kernel  
Extensions

SPUFS  
Filesystem

SPU Allocation, Scheduling  
& Dispatch Extension

64-bit Linux Kernel

Cell BE Architecture Specific Code

Multi-large page, SPE event & fault handling, IIC & IOMMU support

Firmware / Hypervisor

Cell Reference System Hardware

# Extensions to Linux

PPC32 Apps.

Cell32 Workloads

Cell64 Workloads

PPC64 Apps.

Programming Models Offered: RPC, Device Subsystem, D...  
Hetergenous Threads -- Single SPU, SPU Groups, Shared

SPE Management Runtime  
Library (32-bit)

SPE M...

std. PPC32  
elf interp

SPE Object Loa  
Services

32-bit GNU Libs (glibc,etc)

ILP32 Processes

System Call Interface

exec Loader

File System  
Framework

Device  
Framework

Network  
Framework

Misc format bin  
SPU Object  
Loader Extension

Privileged  
Kernel  
Extensions

- SPUFS Filesystem /spu/thread#/  
• open, read, write, close
- mem – access to local storage
- regs – access to 128 – 128 bit registers
- mbox – spe->ppe mailbox
- plix - spe-> ppe interrupt mailbox
- mbox\_stat - obtain mailbox status
- signal1 – signal notification 1 access
- signal2 – signal notification 2 access
- signalx\_type – signal type to OR or overwrite
- npc - read/write SPE next program counter
- Fpcr – spe floating point control/status register
- Decr – spe decrementer
- decr\_status – spe decrementer status
- spu\_tag\_mask – access tag query mask
- Event\_mask – access spe event\_mask
- Srr0 – access spe state restore register 0

64-bit Linux Kernel

Cell BE Architecture Specific Code

Multi-large page, SPE event & fault handling, IIC & IOMMU support

Firmware / Hypervisor

Cell Reference System Hardware

# Extensions to Linux

PPC32 Apps.

Cell32 Workloads

Cell64 Workloads

PPC64 Apps.

Programming Models Offered: RPC, Device Subsystem, Direct/Indirect Access  
Hetergenous Threads -- Single SPU, SPU Groups, Shared Memory

SPE Management Runtime  
Library (32-bit)

SPE Management Runtime  
Library (64-bit)

std. PPC32  
elf interp

SPE Object Loader  
Services

std. PPC64  
elf interp

32-bit GNU Libs (glibc,etc)

64-bit GNU Libs (glibc)

ILP32 Processes

LP64 Processes

System Call Interface

exec Loader

File System  
Framework

Device  
Framework

Network  
Framework

Misc format bin  
SPU Object  
Loader Extension

Privileged  
Kernel  
Extensions

- SPUFS Filesystem /spu/thread#/
  - open, mmap, close
- mem – problem state access to Local Storage
- signal1 – direct application access to Signal 1
- signal2 – direct application access to Signal 2
- cntl – direct application access to SPE controls, DMA Queues, mailboxes

64-bit Linux Kernel

Cell BE Architecture Specific Code

Multi-large page, SPE event & fault handling, IIC & IOMMU support

Firmware / Hypervisor

Cell Reference System Hardware



# Extensions to Linux

PPC32 Apps.

Cell32 Workloads

Cell64 Workloads

PPC64 Apps.

Programming Models Offered: RPC, Device Subsystem, Direct/Indirect Access  
Heterogenous Threads -- Single SPU, SPU Groups, Shared Memory

SPE Management Runtime

- SPE Task Control System Calls
- Sys\_spu\_create\_thread – allocates an spe task/context and creates a directory in spuifs
- Sys\_spu\_run – activates an SPE task.context on a physical SPE and blocks in the kernel as a proxy thread to handle SPE events, mmu faults and errors

SPE Management Runtime  
Library (64-bit)

loader

std. PPC64  
elf interp

64-bit GNU Libs (glibc)

LP64 Processes

exec Loader

File System  
Framework

Device  
Framework

Network  
Framework

Streams  
Framework

SPU Management  
Framework

Misc format bin  
SPU Object  
Loader Extension

Privileged  
Kernel  
Extensions

SPUFS  
Filesystem

SPU Allocation, Scheduling  
& Dispatch Extension

64-bit Linux Kernel

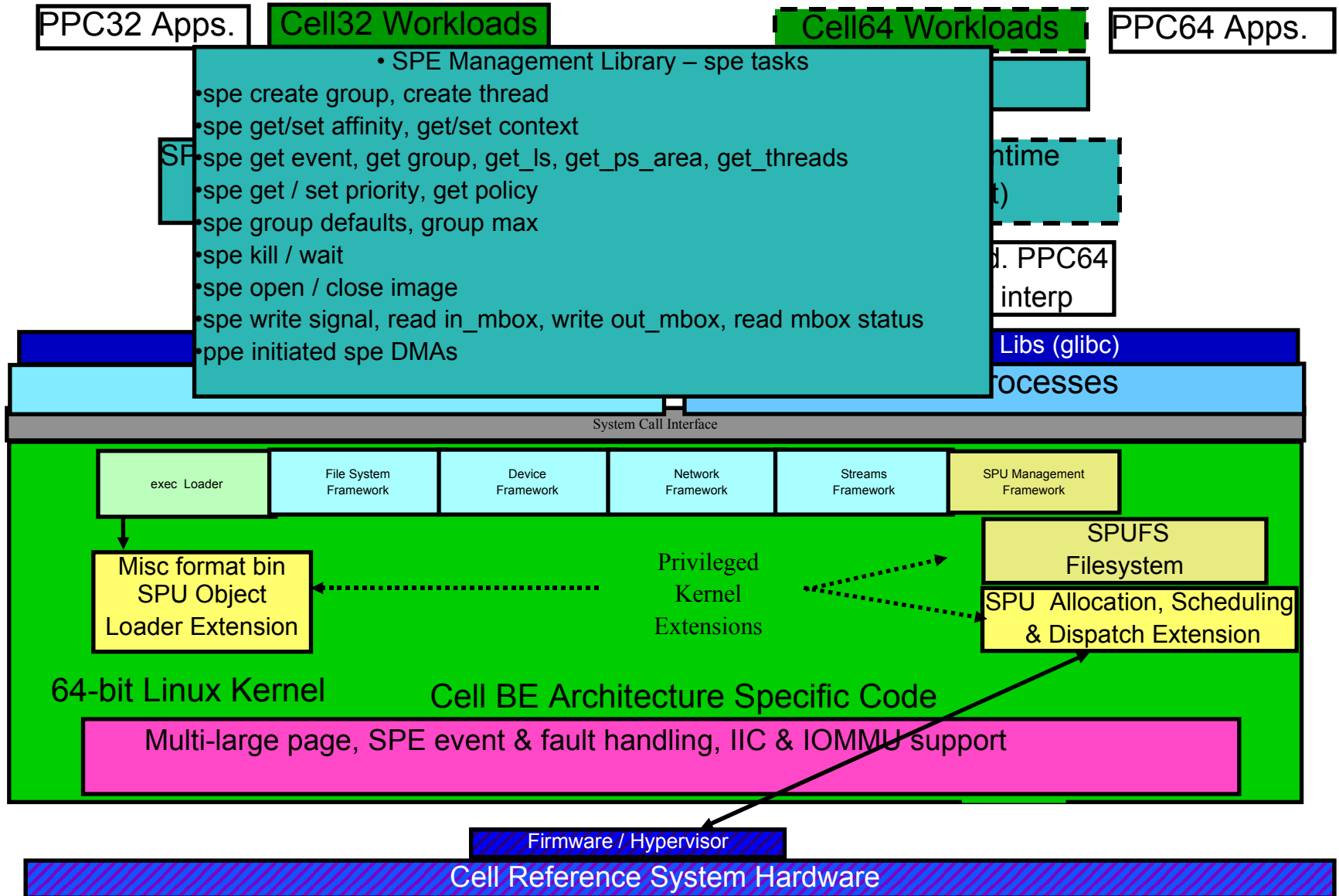
Cell BE Architecture Specific Code

Multi-large page, SPE event & fault handling, IIC & IOMMU support

Firmware / Hypervisor

Cell Reference System Hardware

# Extensions to Linux





**Hello word PPU/SPU**

scanit  
The security company

# "Hello World!" – SPE



- SPE Program

```
#include <stdio.h>

int main()
{
    printf("Hello world!\n");
    return 0;
}
```

- SPE Makefile

```
PROGRAM_spu    := hello_spu
LIBRARY_embed  := hello_spu.a
IMPORTS        = $(SDKLIB_spu)/libc.a
include $(TOP)/make.footer
```

# "Hello World!" – PPE



- PPU program

```
#include <stdio.h>
#include <libspe.h>
extern spe_program_handle_t hello_spu;
int main(void)
{
    int speid, status;
    speid = spe_create_thread (0, &hello_spu, NULL, NULL, -1, 0);
    spe_wait(speid, &status, 1);
    return 0;
}
```

- PPU Makefile

```
DIRS = spu
PROGRAM_ppu = hello_ppu
IMPORTS = ../spu/hello_spu.a -lspe
include $(TOP)/make.footer
```



# PPE and SPE Synergistic Programming



## PPE Code

```
#include <stdio.h>
#include <libspe.h>
extern spe_program_handle_t hello_spu;
int main(void)
{
    int speid, status;
    speid = spe_create_thread (0, &hello_spu, NULL, NULL, -1, 0);
    spe_wait(speid, &status, 1);
    return 0;
}
```

## SPE Code

```
#include <stdio.h>
#include <cbe_mfc.h>
#include <spu_mfcio.h>

int main(unsigned long long speid, unsigned long long argp, unsigned long long envp)
{
    printf("Hello world!\n");
    return 0;
}
```



# Backup Slide – “Hello World”

Applications Places Desktop 6:38 PM

FedoraCore5-CellBroadband - VMware Server Console

File Edit View Host VM Tabs Help

Shut Down Suspend Start Up Restart Snapshot Revert Full Screen Quick Switch Summary Console

Inventory

- Slackware Tests
- FedoraCore5-CellBroadband

localhost.localdomain FedoraCore5-CellBroadband

Browse and run installed applications

```
root@none hello_spu# ls
Makefile hello_spu hello_spu.c hello_spu.d hello_spu.o
root@none hello_spu# ./hello_spu
Hello World!
root@none hello_spu#
```

root@none:~/HITB/hello\_spu

File Edit View Terminal Tabs Help

```
4152880258182: (3787522324): [root@none] hello_spu# ./hello_spu4154180000000:
[0:0]: (PC:0xC0000000000032590) : 127.9 Kilo-Inst/Sec : 260000.0 Kilo-Cycles/Sec
4154180000000: [0:1]: (PC:0xC0000000000032590) : 126.2 Kilo-Inst/Sec
4156720000000: [0:0]: (PC:0xC0000000000032590) : 62.9 Kilo-Inst/Sec : 254000.0 Kilo-Cycles/Sec
4156720000000: [0:1]: (PC:0xC00000000000C358) : 240.4 Kilo-Inst/Sec
4156721734409: (3793299561): Hello World!
4156722135955: (3793683830): [root@none] hello_spu# 4156740000000: [0:0]: (PC:0xC0000000000032590) : 31.9 Kilo-Inst/Sec : 2500.0 Kilo-Cycles/Sec
4156740000000: [0:1]: (PC:0xC0000000000032590) : 240.6 Kilo-Inst/Sec
4160100000000: [0:0]: (PC:0xC0000000000032590) : 146.6 Kilo-Inst/Sec : 672000.0 Kilo-Cycles/Sec
4160100000000: [0:1]: (PC:0xC0000000000032590) : 96.1 Kilo-Inst/Sec
4162800000000: [0:0]: (PC:0xC0000000000032590) : 113.5 Kilo-Inst/Sec : 540000.0 Kilo-Cycles/Sec
4162800000000: [0:1]: (PC:0xC0000000000032590) : 75.1 Kilo-Inst/Sec
```

Your version of VMware Tools is out of date.

Local host

# Library Calls from SPU



- When the SPU needs to do any standard library calls, like printf or exit, it has to call back to the main thread
  - Using the stop-and-signal assembly instruction with standardized argument value
  - That value is returned from the ioctl call and the user thread must react to that. This means copying the arguments from the SPE Local Store, execute the library call and calling the ioctl again

**stop u14** - Stop and signal. Execution is stopped, the current address is written to the SPU NPC register, the value u14 is written to the SPU status register, and an interrupt is sent to the PowerPC® Processor Unit (PPU).



## FedoraCore5-CellBroadband - VMware Server Console



File Edit View Host VM Tabs Help



Shut Down



Suspend



Start Up



Restart



Snapshot



Revert



Full Screen



Quick Switch



Summary



Console

Inventory

Slackware Tests

FedoraCore5-CellBroadband

localhost.localdomain x FedoraCore5-CellBroadband x

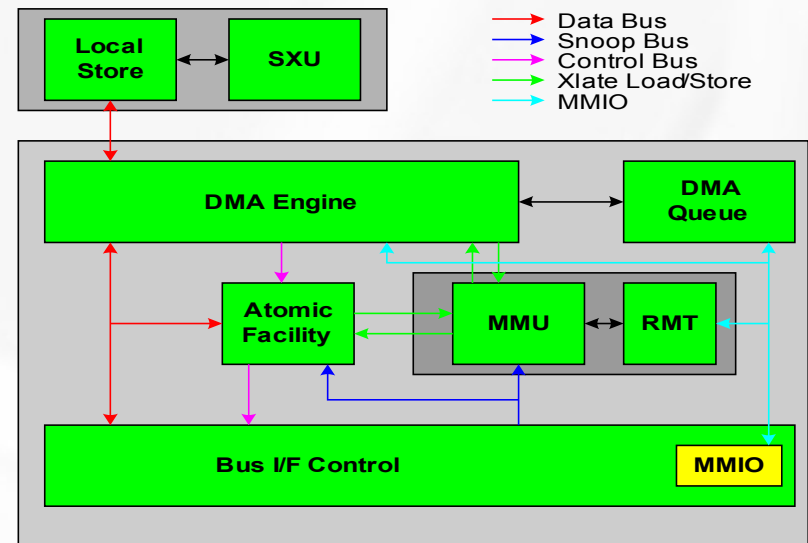
root@(none):~/HITB/hello\_spu

```
[root@(none) hello_spu]# spu-gdb ./hello_spu
GNU gdb 6.3
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=powerpc64-unknown-linux-gnu --target=spu"...
(gdb) disassemble main
Dump of assembler code for function main:
0x00000170 <main+0>:   ila    $3,0x210 <_.rodata>
0x00000174 <main+4>:   stqd   $0,16($1)
0x00000178 <main+8>:   nop    $127
0x0000017c <main+12>:  stqd   $1,-32($1)
0x00000180 <main+16>:  ai     $1,$1,-32
0x00000184 <main+20>:  brsl   $0,0x1a0 <puts> # 1a0
0x00000188 <main+24>:  ai     $1,$1,32      # 20
0x0000018c <main+28>:  fsmbi  $3,0
0x00000190 <main+32>:  lqd    $0,16($1)
0x00000194 <main+36>:  bi     $0
0x00000198 <main+40>:  stop
0x0000019c <main+44>:  stop
End of assembler dump.
(gdb) █
```

# Cell Primary Communication Mechanisms



- **DMA transfers, mailbox messages,** and signal-notification
- All three are implemented and controlled by the SPE's MFC



Mechanism	Description
DMA transfers	Used to move data and instructions between main storage and an LS. SPEs rely on asynchronous DMA transfers to hide memory latency and transfer overhead by moving information in parallel with SPU computation.
Mailboxes	Used for control communication between an SPE and the PPE or other devices. Mailboxes hold 32-bit messages. Each SPE has two mailboxes for sending messages and one mailbox for receiving messages.
Signal notification	Used for control communication from the PPE or other devices. Signal notification (also called <i>signaling</i> ) uses 32-bit registers that can be configured for one-sender-to-one-receiver signalling or many-senders-to-one-receiver signalling.



# MFC (Memory Flow Control) Commands



- Main mechanism for SPUs to
  - access main storage
  - maintain **synchronization** with other processors and devices in the system
- Can be issued either SPU via its MFC by PPE or other device, as follows:
  - Code running on the SPU issues an MFC command by executing a series of writes and/or reads using **channel instructions**
  - Code running on the PPE or other devices issues an MFC command by performing a series of stores and/or loads to **memory-mapped I/O** (MMIO) registers in the MFC
- MFC commands are queued in one of two independent MFC command queues:
  - MFC SPU Command Queue — For channel-initiated commands by the associated SPU
  - MFC Proxy Command Queue — For MMIO-initiated commands by the PPE or other device



# DMA Commands



- MFC commands that transfer data are referred to as DMA commands
- Transfer direction for DMA commands referenced from the SPE
  - Into an SPE (from main storage to local store) → **get**
  - Out of an SPE (from local store to main storage) → **put**

# DMA GET/PUT Commands



- DMA get from main memory into local store

```
(void) mfc_get( volatile void *ls, uint64_t ea, uint32_t size,  
               uint32_t tag, uint32_t tid, uint32_t rid)
```

- DMA put into main memory from local store

```
(void) mfc_put(volatile void *ls, uint64_t ea, uint32_t size,  
               uint32_t tag, uint32_t tid, uint32_t rid)
```

- To ensure order of DMA request execution:

- mfc\_putf : **fenced** (all commands executed before within the same tag group must finish first, later ones could be before)
- mfc\_putb : **barrier** (the barrier command and all commands issued thereafter are not executed until all previously issued commands in the same tag group have been performed)

# DMA Resources



- DMA transfers
  - transfer sizes can be 1, 2, 4, 8, and  $n \times 16$  bytes ( $n$  integer)
  - maximum is 16KB per DMA transfer
  - 128B alignment is preferable
- DMA command queues per SPU
  - 16-element queue for SPU-initiated requests
  - 8-element queue for PPE-initiated requests
  - SPU-initiated DMA is always preferable
- DMA tags
  - each DMA command is tagged with a 5-bit identifier
  - same identifier can be used for multiple commands
  - tags used for polling status or waiting on completion of DMA commands
- DMA lists
  - a single DMA command can cause execution of a list of transfer requests (in LS)
  - lists implement scatter-gather functions
  - a list can contain up to 2K transfer requests





# SPE2SPE DMA



- Address in the other SPE's local store is represented as a 32-bit effective address (global address)
- SPE issuing the DMA command needs a pointer to the other SPE's local store as a 32-bit effective address (global address)

- PPE code can obtain effective address of an SPE's local store:

```
#include <libspe.h>
speid_t speid;
void *spe_ls_addr;
spe_ls_addr = spe_get_ls(speid);
```

- Effective address of an SPE's local store can then be made available to other SPEs (e.g. via DMA or mailbox)

# Simple DMA Demo



This is just a simple program to show how to send/receive information using DMA

Will be expanded in next sections

scanit  
The security company



# Backup Slide - Simple DMA Demo

Applications Places Desktop 8:30 PM

FedoraCore5-CellBroadband - VMware Server Console

File Edit View Host VM Tabs Help

Shut Down Suspend Start Up Restart Snapshot Revert Full Screen Quick Switch Summary Console

Inventory

- Slackware Tests
- FedoraCore5-CellBroadband

localhost.localdomain FedoraCore5-CellBroadband

```
root@(none):~/HITB/mbox1
[root@(none) mbox1]# ls
Makefile common.h hello hello.c hello.d hello.o spu
[root@(none) mbox1]# vi hello.c
[root@(none) mbox1]# ./hello
SPE 0 byte offset for buffer is 2400
SPE 0 has LS + offset of f7fa1400
SPE 0 received buffer "Good morning!"
New modified buffer is Guten Morgen!
[root@(none) mbox1]#
```

8872461964177: (7284190957): SPE 0 byte offset for buffer is 2400  
8872474617464: (7296723803): SPE 0 has LS + offset of f7fa1400  
8872474739872: (7296955693): SPE 0 received buffer "Good morning!"  
8872475179863: (7297377267): [root@(none) mbox1]# 8872480000000: [0:0]: (PC:0xC00000000032590) : 491.4 Kilo-Inst/Sec : 666.7 Kilo-Cycles/Sec  
8872480000000: [0:1]: (PC:0xC000000000032590) : 11.2 Kilo-Inst/Sec  
8878880000000: [0:0]: (PC:0xC000000000032590) : 199.7 Kilo-Inst/Sec : 1280000.0 Kilo-Cycles/Sec  
8878880000000: [0:1]: (PC:0xC000000000032590) : 254.2 Kilo-Inst/Sec

Your version of VMware Tools is out of date.

Local host

# DMA Demo Between SPUs



This is just a simple program to show how to send/receive information using DMA between SPUs.

scanit  
The security company

# Backup Slide - Between SPU's

Applications Places Desktop 8:34 PM

FedoraCore5-CellBroadband - VMware Server Console

File Edit View Host VM Tabs Help

Shut Down Suspend Start Up Restart Snapshot Revert Full Screen Quick Switch Summary Console

Inventory

- Slackware Tests
- FedoraCore5-CellBroadband

localhost.localdomain x FedoraCore5-CellBroadband x

```
root@(none):~/HITB/spe2spe
[root@(none) spe2spe]# ls
Makefile common.h hello.c hello.d hello.o spu
[root@(none) spe2spe]# ./hello
SPE 0 byte offset for buffer is 2400
SPE 0 has LS + offset of f7fa1400
SPE 1 byte offset for buffer is 2380
SPE 1 has LS + offset of f7761380
SPE 0 received buffer "Good morning!"
Last SPE has received buffer of Guten Morgen!
New modified buffer is Bon jour!
[root@(none) spe2spe]#
```

9068634618204: (7415039141): SPE 0 received buffer "Good morning!"  
9068634649983: (7415087885): Last SPE has received buffer of Guten Morgen!  
9068634774749: (7415444811): New modified buffer is Bon jour!  
9068635349652: (7416029232): [root@(none) spe2spe]# 9068640000000: [0:0]: (PC:0xC000000000032590) : 479.2 Kilo-Inst/Sec : 645.2 Kilo-Cycles/Sec  
9068640000000: [0:1]: (PC:0xC000000000032590) : 12.8 Kilo-Inst/Sec  
9075300000000: [0:0]: (PC:0xC000000000032590) : 206.2 Kilo-Inst/Sec : 1332000.0 Kilo-Cycles/Sec  
9075300000000: [0:1]: (PC:0xC000000000032590) : 267.8 Kilo-Inst/Sec



# **Debugging the SPE** **What is going on?**

scanit  
The security company



- **SPU\_INFO=1**
  - Implemented within libspe runtime library
  - When loading SPE ELF executable, prints message  
Loading SPE program : NNN  
SPU LS Entry Addr : NNN
  - Before starting up new SPE thread, prints message  
Starting SPE thread 0x..., to attach debugger  
use: spu-gdb -p NNN
- **SPU\_DEBUG\_START=1**
  - *Includes everything done by SPU\_INFO=1*
  - *Waits until debugger is attached (or signal received)*





# Debug HINT



Since each SPU register can hold multiple fixed (or floating) point values of different sizes, GDB offers to us a data structure that can be accessed with different formats:

```
(gdb) ptype $r70
type = union __gdb_builtin_type_vec128 {
    int128_t uint128;
    float v4_float[4];
    int32_t v4_int32[4];
    int16_t v8_int16[8];
    int8_t v16_int8[16];
}
```

So, specifying the field in the data structure, we can update it:

```
(gdb) p $r70.uint128
$1 = 0x00018ff000018ff000018ff000018ff0
(gdb) set $r70.v4_int32[2]=0xdeadbeef
(gdb) p $r70.uint128
$2 = 0x00018ff000018ff0deadbeef00018ff0
```





# Exploiting Software Vulnerabilities

scanit  
The security company

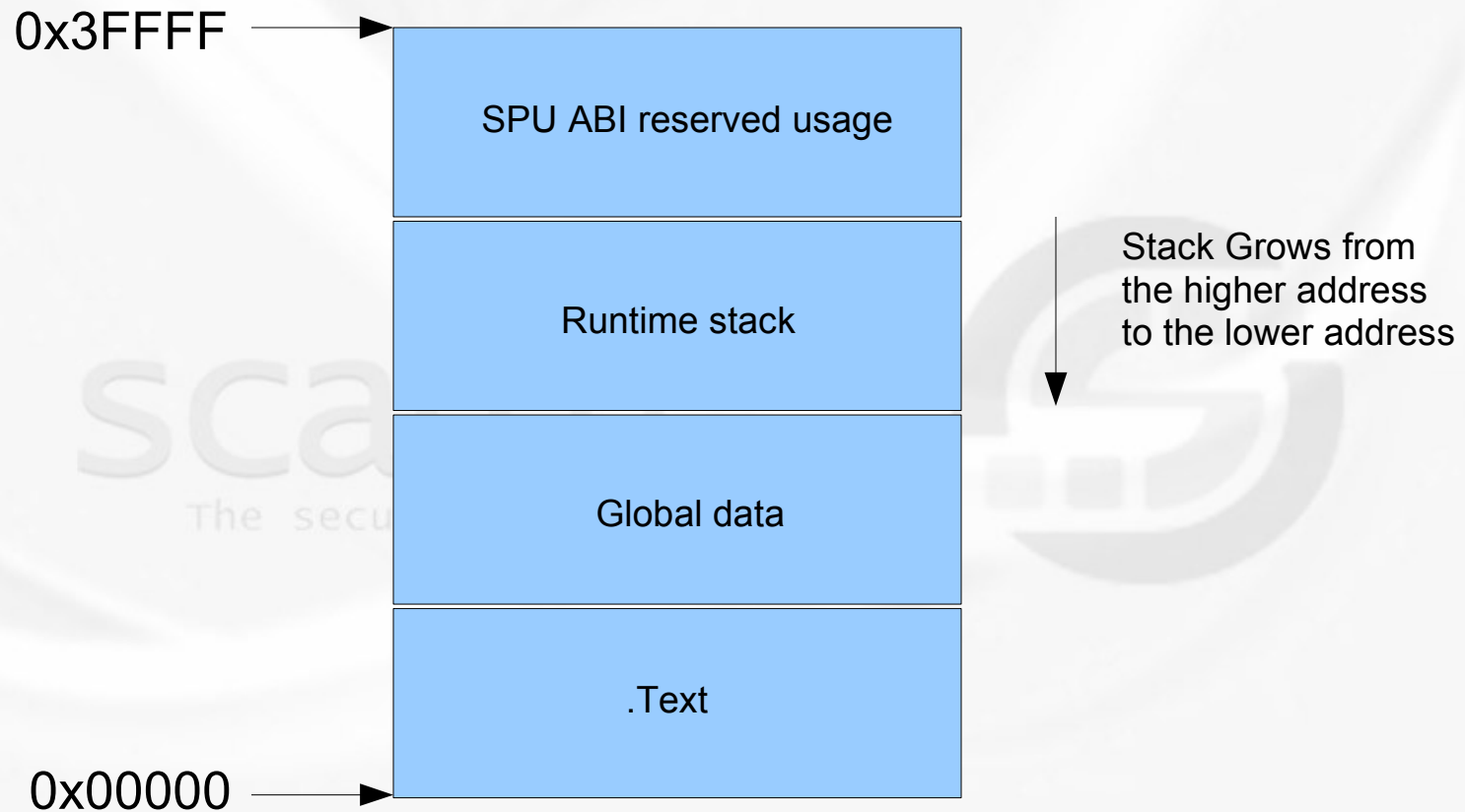


# Memory Overflows



“The SPU Local Store has no memory protection, and memory access wraps from the end of Local Store back to the beginning. **An SPU program is free to write anywhere in Local Store including its own instruction space.** A common problem in SPU programming is the corruption of the SPU program text when the stack area overflows into the program area. This problem typically does not become apparent until some later point in the program execution when the program attempts to execute code in area that was corrupted, which typically results in an illegal instruction exception. Even with a debugger it can be difficult to track down this type of problem because the cause and effect can occur far apart in the program execution. Adding printf's just moves the failure point around. “

# SPU Memory Layout



# Exploit Sample



Exploiting problems in the DMA communications between PPU-SPU and SPU-SPU.

The attacker can take complete control over the application running in the SPU... They can later force one SPU to exploit others, using the SPU-SPU communications.

For sure, it's not a problem in the architecture, but with programmers if they don't write secure code.

# Backup Slide - Exploit Sample

Applications Places Desktop 10:02 PM

FedoraCore5-CellBroadband - VMware Server Console

File Edit View Host VM Tabs Help

Shut Down Suspend Start Up Restart Snapshot Revert Full Screen Quick Switch Summary Console

Inventory

- Slackware Tests
- FedoraCore5-CellBroadband

localhost.localdomain x FedoraCore5-CellBroadband x

Applications Places System

root@(none):~/HITB/exploit

```
0x3ff48: stop
0x3ff4c: stop
0x3ff50: stop
0x3ff54: stop
0x3ff58: stop
0x3ff5c: stop
0x3ff60: stop
0x3ff64: stop
0x3ff68: stop
0x3ff6c: stop
0x3ff70: stop
0x3ff74: stop
0x3ff78: stop
0x3ff7c: stop
0x3ff80: stop
0x3ff84: stop
0x3ff88: stop
0x3ff8c: stop
---Type <return> to continue, or q <re
0x3ff90: stop
0x3ff94: stop
0x3ff98: stop
0x3ff9c: stop
0x3ffa0: stop
0x3ffa4: stop
0x3ffa8: stop
0x3ffac: stop
0x3ffb0: stop
0x3ffb4: stop
0x3ffb8: stop
0x3ffbc: stop
0x3ffc0: ilhu $65,33410
0x3ffc4: ilhu $65,33410
0x3ffc8: ilhu $65,33410
0x3ffcc: ilhu $65,33410
0x3ffd0: ilhu $65,33410
```

root@(none):~/HITB/exploit

File Edit View Terminal Tabs Help

```
11692760000000: [0:0]: (PC:0xC000000000032590) : 0.2 Kilo-Ins
11692760000000: [0:1]: (PC:0xC000000000032590) : 0.7 Kilo-Ins
11692780000000: [0:0]: (PC:0xC000000000032590) : 0.1 Kilo-Ins
11692780000000: [0:1]: (PC:0xC000000000032590) : 0.2 Kilo-Ins
11692800000000: [0:0]: (PC:0xC000000000032590) : 0.2 Kilo-Ins
11692800000000: [0:1]: (PC:0xC000000000032590) : 0.1 Kilo-Ins
11692820000000: [0:0]: (PC:0xC000000000032590) : 0.1 Kilo-Ins
11692820000000: [0:1]: (PC:0xC000000000032590) : 0.5 Kilo-Ins
11692840000000: [0:0]: (PC:0xC000000000032590) : 0.2 Kilo-Ins
11692840000000: [0:1]: (PC:0xC000000000032590) : 0.2 Kilo-Ins
11692860000000: [0:0]: (PC:0xC000000000032590) : 0.3 Kilo-Ins
11692860000000: [0:1]: (PC:0xC000000000032590) : 0.3 Kilo-Ins
11692880000000: [0:0]: (PC:0xC000000000032590) : 0.1 Kilo-Ins
11692880000000: [0:1]: (PC:0xC000000000032590) : 0.2 Kilo-Ins
11692900000000: [0:0]: (PC:0xC000000000032590) : 0.3 Kilo-Ins
11692900000000: [0:1]: (PC:0xC000000000032590) : 0.1 Kilo-Ins

11692903477267: (10379741891): Program received signal SIGINT, Int
11692903634477: (10379884661): 0x000002d4 in memcpy ()
11692903662796: (10379912827): (gdb) 11695260000000: [0:0]: (PC:0x
00.0 Kilo-Cycles/Sec
11695260000000: [0:1]: (PC:0xC000000000032590) : 93.7 Kilo-Ins
x/11702600000000: [0:0]: (PC:0xC000000000032590) : 239.4 Kilo-I
```



# Scenario



```
main(int argc, char **argv)
{
    char buffer[10];
    char buffer2[10];

    memset(buffer, 0, 10);
    memset(buffer2, 0, 10);

    strcpy(buffer, "AAAABBBB");
    strncpy(buffer2, argv[1], 10);

    printf("\n Tamanho1: %d - %s\n", strlen(buffer), buffer);
    printf("\n Tamanho2: %d - %s\n", strlen(buffer2), buffer2);
}

[root@localhost ~]# ./a.out rodrigo123

Tamanho1: 8 - AAAABBBB

Tamanho2: 18 - rodrigo123AAAABBBB
[root@localhost ~]#
```

# Scenario



If a code like this are been used to receive a value from user and then pass it through DMA, but the static value are used to define the buffer lenght, the concatenation will force an overflow.

This can be used in a DMA communication, permitting the attacker to take control of a SPU.

Also, between SPUs this could occur.

# Easy to find?



The simulator has a feature that monitors selected addresses or regions of Local Store for read or write accesses. This feature can identify stack overflow conditions.

Invoked in the simulator command window as follows:

```
enable_stack_checking [spu_number] [spu_executable_filename]
```

This procedure uses the nm system utility to determine the area of Local Store that will contain program code and creates trigger functions to trap writes by the SPU into this region.

Note: The simulator's method of detecting stack overflow only looks for stack overflow into the text and static data segments and thus does not detect stack overflows into the heap.

The same approach used by this TCL function can be used to monitor other portions and structures.

# What else can we do?

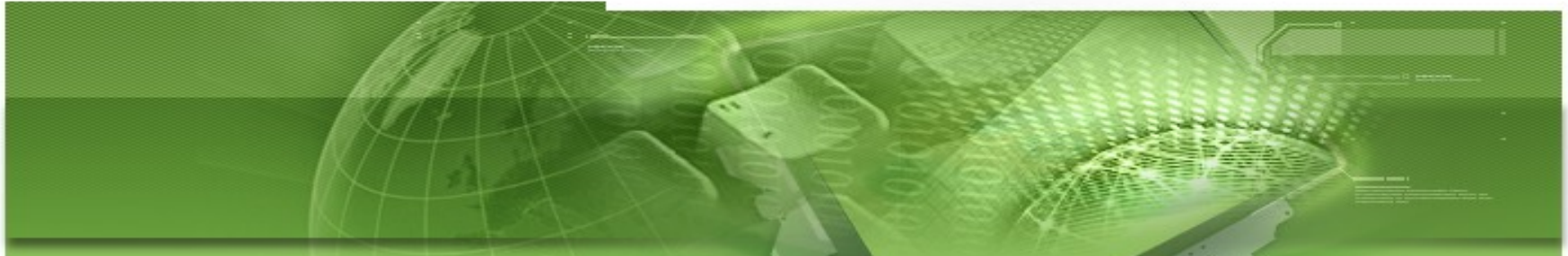


- Nick Breese presented his CrackStation project in Blackhat this year
  - Since he sent it to me before the conference, I prepared this presentation without redundancies...
  - It used the SIMD capabilities and big registers provided by the architecture to crack passwords ;)
- IBM Researchers released a study about the usage of the Cell SPU as a Garbage Collector Co-processor
- I have not tried to see if there is a JTAG-enabled interface in those Cell machines (blade and PS3) to try the RISCWatch
- Since the SPU access are controlled by the PPE, the idea of run system integrity protections inside de SPE cannot be done
  - **Hint: The SMM manipulation library have been released in Phrack #65 (this month - April/2008)**





# Acknowledges



- I would like to tks to Scanit and Oger Systems for give me a PS3 to prepair this presentation
- I used lots of copy+paste from IBM training materials in this presentation, so tks to the authors! - Pay attention to copyrights
- I also need to tks to my research partners at Rise Security and Filipe Balestra
- Special tks to Andre Detsch, the Cell Kernel Guru for sharing with me his experiences and insights
- Also, special tks to the HITB organizers for trust me (again!)
- Let's party folks!





# End! Really is?



# scanit

The security company

THANK YOU

Questions & Maybe,  
Answers

# References



- Cell resource center at developerWorks
  - <http://www-128.ibm.com/developerworks/power/cell/>
- Cell developer's corner at power.org
  - <http://www.power.org/resources/devcorner/cellcorner/>
- The cell project at IBM Research
  - <http://www.research.ibm.com/cell/>
- The Cell BE at IBM alphaWorks
  - <http://www.alphaworks.ibm.com/topics/cell>
- IBM Power Architecture
  - <http://www-03.ibm.com/chips/power/>
- Cell BE documentation at IBM Microelectronics
  - [http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Cell\\_Broadband\\_EngineCell](http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_EngineCell)
- Linux info at the Barcelona Supercomputing Center website
  - <http://www.bsc.es/projects/deepcomputing/linuxoncell>

