An SPU Reference Model for Simulation, Random Test Generation and Verification

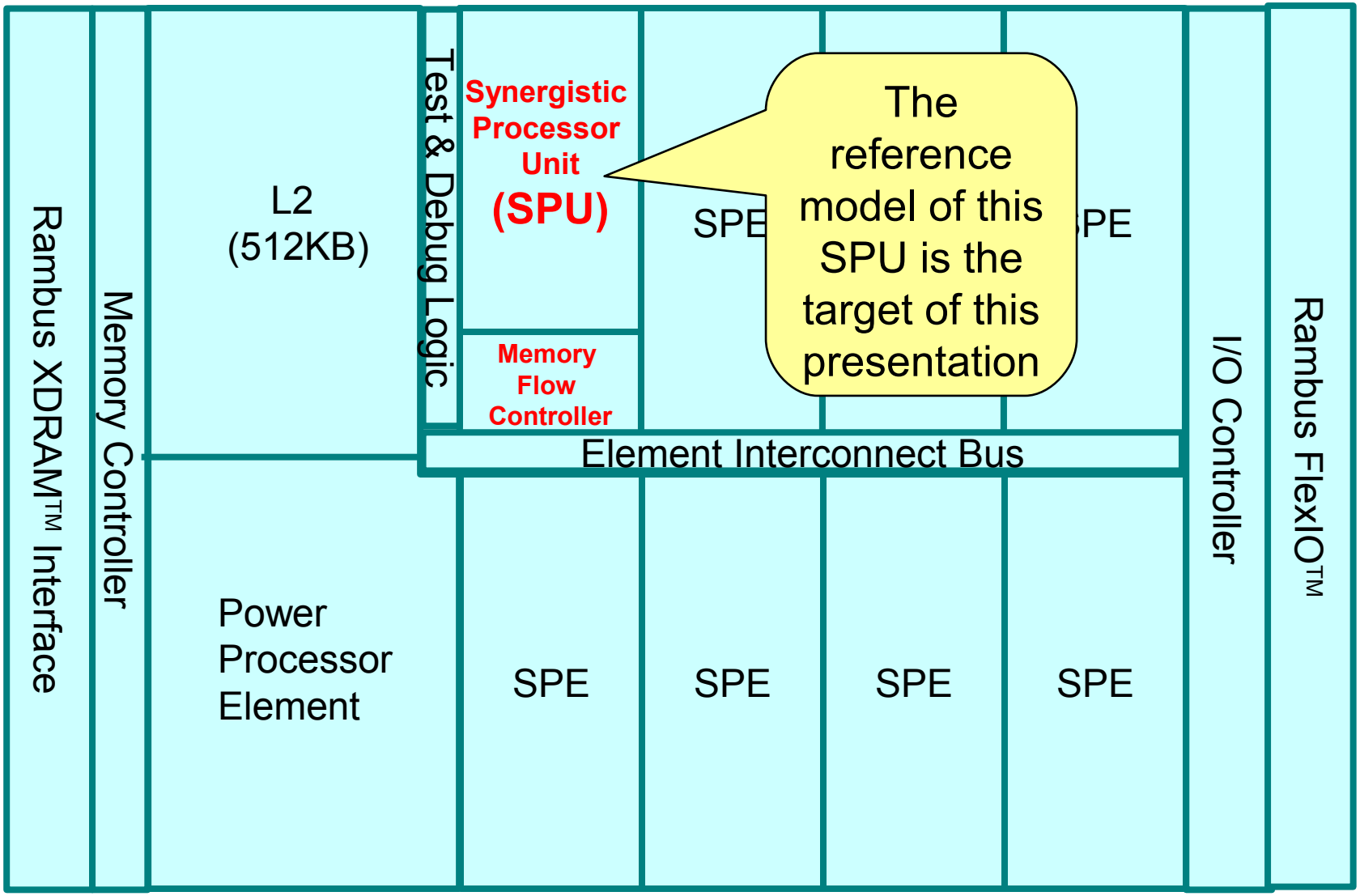
Yukio Watanabe, Balazs Sallay[1], Brad Michael[1],
Daniel Brokenshire[1], Gavin Meil[1], Hazim Shafi[1], Daisuke Hiraoka[2]

Toshiba Corporation
[1]IBM
[2]Sony Computer Entertainment Inc.

Outline

- Overview of the Cell Broadband Engine and the SPU
- SPU reference model
- Applications using the SPU reference model
  - Simulator
  - Random test case generator
  - Verification environment
- Summary

The Cell Broadband Engine

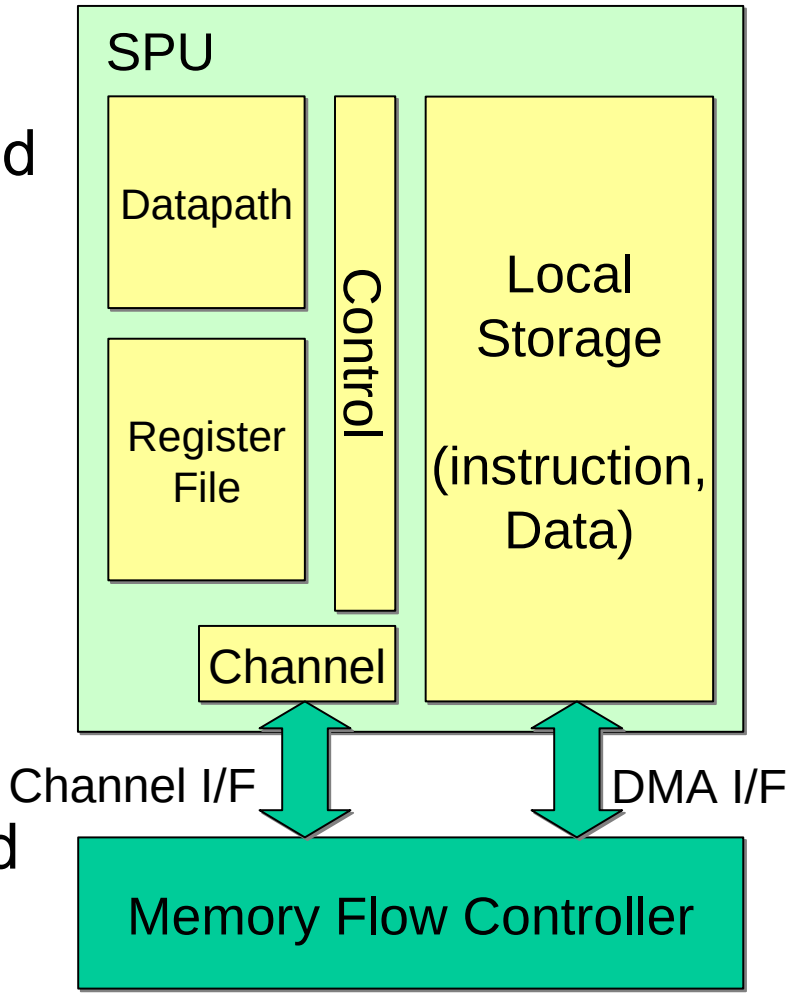


SPU --- Synergistic Processor Unit

- Brand new architecture for multimedia applications
- Instruction Set Architecture (ISA) was defined considering
  - ISA efficiency for multimedia applications
  - Area efficiency
  - Physical timing
  - Power efficiency

SPU Functions

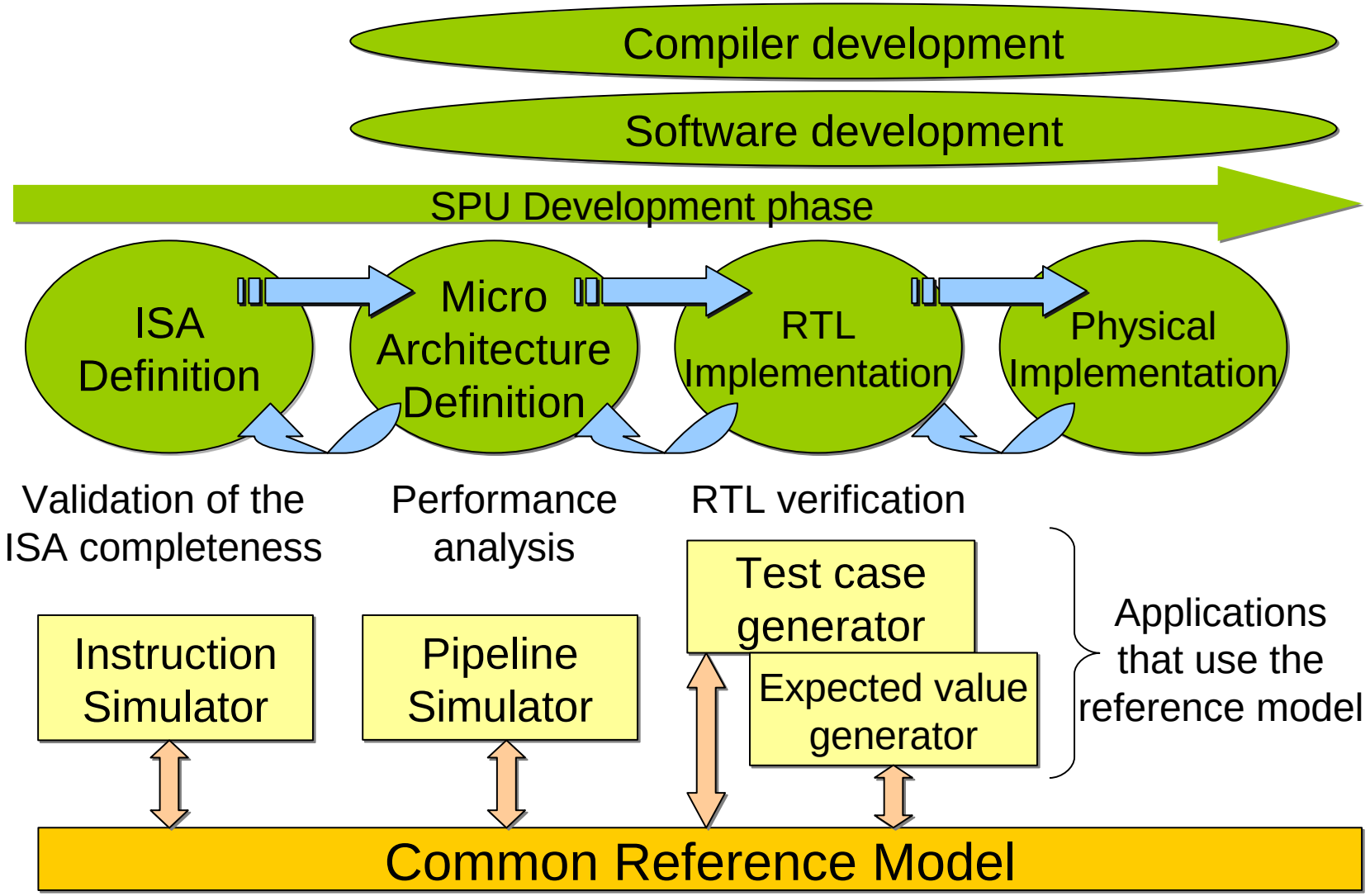
- Instruction execution
  - Instructions are fetched from the local storage and executed
  - Load/Store instructions can access only the local storage
- External transactions
  - Channel I/F is used to communicate with external devices
  - DMA transfers the data between the local storage and the external memory through DMA I/F



Outline

- Overview of the Cell Broadband Engine and the SPU
- SPU reference model
- Applications using the SPU reference model
  - Simulator
  - Random test case generator
  - Verification environment
- Summary

SPU Development Phases and How the Reference Model was Used



SPU Reference Model Overview

- SPU reference model is a C library that executes SPU instructions
- The SPU reference model is commonly used by various applications
  - Instruction/Pipeline/System level simulators
  - Random test case generator
  - On-the-fly expected value generator in the verification environment
- ISA changes due to various feedback had to be implemented only in the reference model

Reference Model Interface

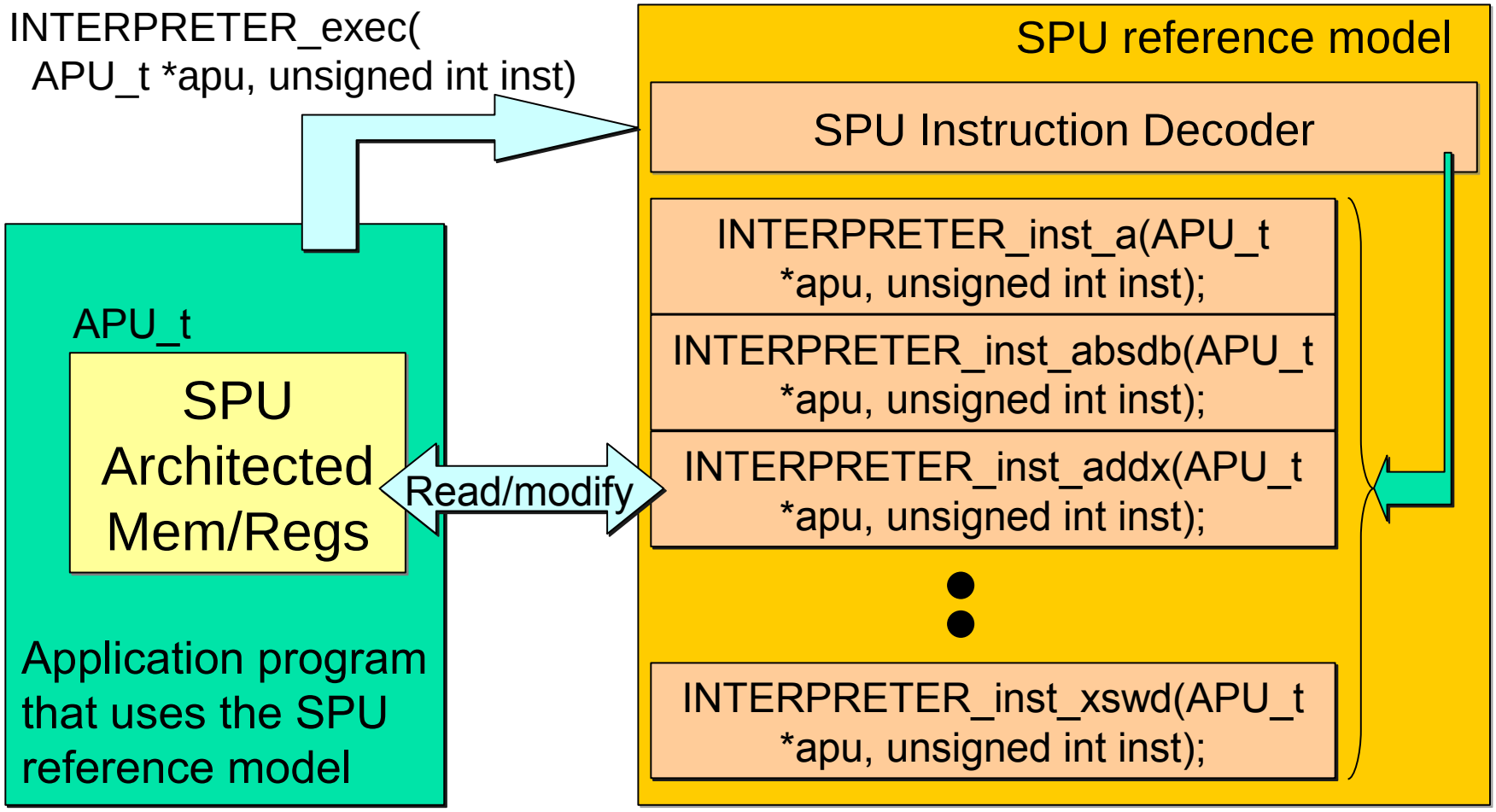
INTERPRETER_exec(APU_t *apu, unsigned int inst);

Pointer to the C structure entity which represents all the SPU architected memory/registers.

A 32bit instruction to be executed in the SPU reference model.

```
/* SPU Architected Memory/Register Resource Structure */
typedef struct apu_t {
  REG reg[128];                /*128b 128entry register file*/
  char memory[256*1024]; /*256KB local storage */
  int pc;                      /* Program counter */
  int spu_status;              /* status register */
                .
                .
} APU_t;
```

SPU Reference Model Structure



By splitting off the Mem/Regs structure from the reference model, the reference model became simple and versatile.

Outline

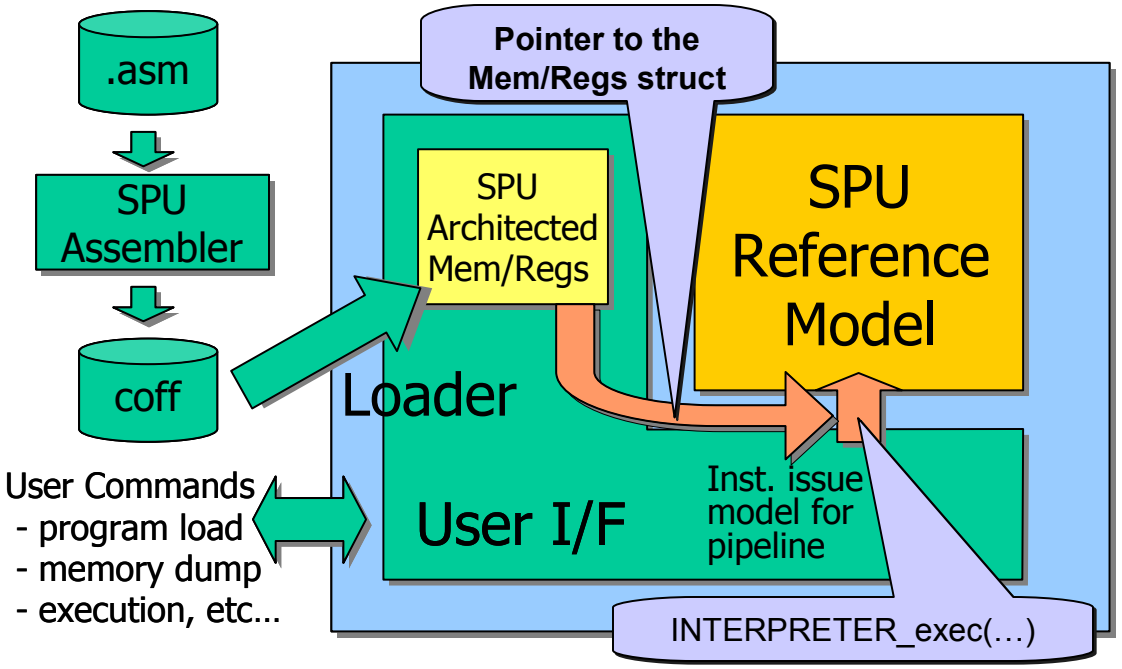
- Overview of the Cell Broadband Engine and the SPU
- SPU reference model
- Applications using the SPU reference model
  - Simulator
  - Random test case generator
  - Verification environment
- Summary

Simulators Using the Reference Model

- Three kinds of simulators were implemented
  - Instruction level
    - ISA validation
    - Software development as a fast simulator
  - Cycle accurate pipeline level
    - Performance analysis of the micro architecture
    - Software/Compiler optimization
  - System level including all the Cell Broadband Engine elements such as PPE, SPU and Memory Flow Controller
    - System software development

Pipeline Level Simulator Implementation

- SPU program binary image is generated by assembler or C/C++ compiler.
- Binary image is loaded into the SPU local storage and PC is set.
- Simulation is executed by user commands.



- Instruction fetch, branch prediction and issue control logics are simulated and an instruction is executed only when the instruction is committed.

Simulation Result Example

Running Eratosthenes' sieve program

```
command > pipeline
changed to pipeline simulator mode
command > run
result of sieve program
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103
107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211
223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331
337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449
457 461 463 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587
593 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701 709
719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827 829 839 853
857 859 863 877 881 883 887 907 911 919 929 937 941 947 953 967 971 977 983 991
997
STOPPED PC = 0xc0, SYSCALL_halt

HALT ... program finished successfully
instruction count: 80769

BREAKPOINT
Interpreter stopped at
000000c4:00000000(stop )
```

Simulation Result Example (cont.)

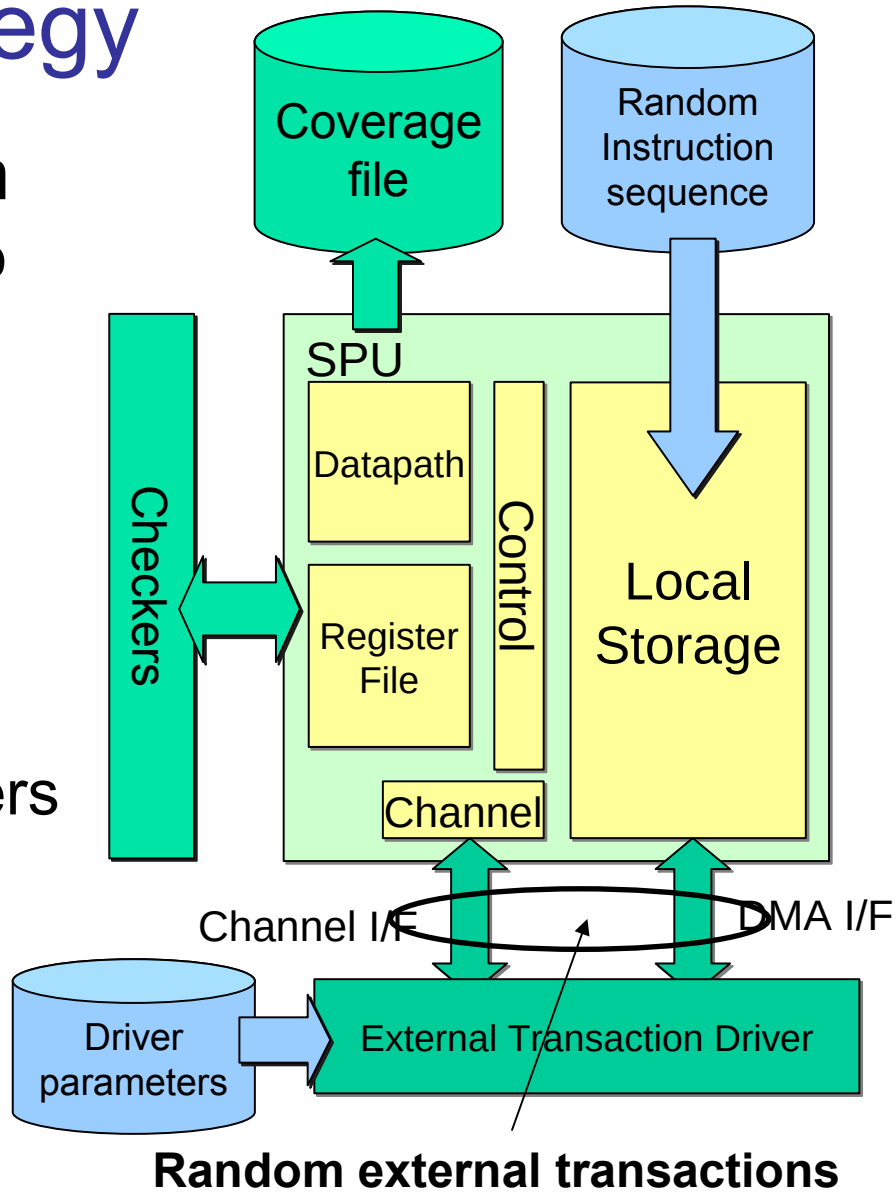
```
command > stat

***
Total Cycle count                372828
Total Instruction count          80769
Total CPI                        4.62
***
Performance Cycle count          372828
Performance Instruction count    80769 (80769)
Performance CPI                  4.62 (4.62)

Branch instructions              18103
Branch taken                     9883
Branch not taken                 8220

Hint instructions                0
Hint hit                         0

Contention at LS between Load/Store and Prefetch 0
```

Simulation Result Example (cont.)

Single cycle	80769 (21.7%)
Dual cycle	0 (0.0%)
Nop cycle	0 (0.0%)
Stall due to branch miss	175079 (47.0%)
Stall due to prefetch miss	0 (0.0%)
Stall due to dependency	116971 (31.4%)
Stall due to fp resource conflict	0 (0.0%)
Stall due to waiting for hint target	0 (0.0%)
Stall due to dp pipeline	0 (0.0%)
Channel stall cycle	0 (0.0%)
SPU Initialization cycle	9 (0.0%)
Total cycle	372828 (100.0%)

SPU Verification Strategy

- Coverage driven verification
  - Verification items that have to be tested are described as coverage events
  - Random test cases are run in the simulation to hit all the coverage items
    - Random instruction sequence
    - Random external transactions
  - During the simulation, checkers check that the SPU logic is working properly
  - Coverage files are generated as the result of simulations

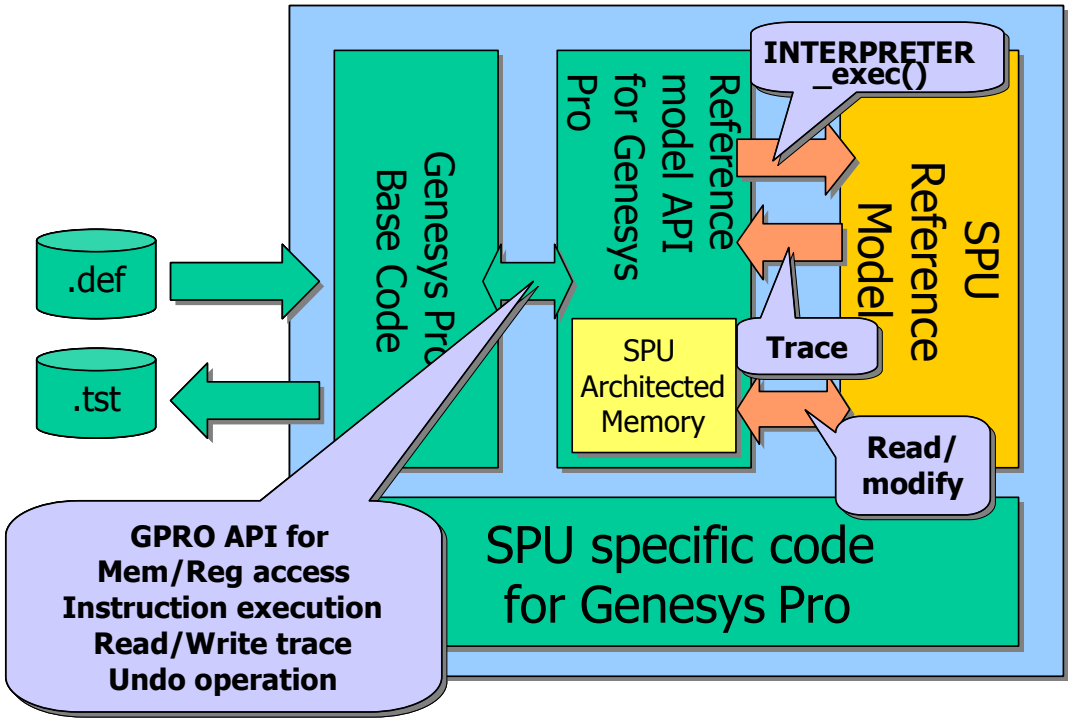


Key Points in Coverage Driven Verification

- Good-quality random test cases
  - Directed random test case generator is necessary to hit coverage events of focus area
- Robustness of the verification environment to run random test cases
  - Instruction execution and external transactions can interact each other
    - Exceptional cases handling
    - Recovery from error states
  - Correct expected values must be generated even with asynchronous interactions

Random Test Case Generator

- Genesys Pro (GPRO) which is an IBM tool to generate random test cases was applied to SPU
- GPRO reads a rule definition file (.def file) as an input and generate a random instruction sequence as .tst file.
- Example of the rule
  - Register dependencies on previous instructions
  - Massive load/store instructions using same address



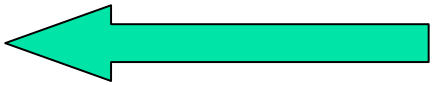
- Reference model is used to generate read/write trace information and to check that the generated instruction sequence obeys the rule given as .def

Interaction between Instruction Execution and External Transactions

Instruction sequence

```
        xor $0, $0, $0        // $0<=0x0;
        stqa $0, 0x00000 // LS[0x0]<=$0
           sequence 0
                 •
                 •
        lqa $1, 0x00000   // $1<=LS[0x0]
        brz $1, br_taken   // go to br_taken if $1==0
br_not_taken:
           sequence 1
                 •
                 •
br_taken:
           sequence 2
                 •
                 •
```

External transactions



DMA transfer to local storage
LS[0x0] <= Non zero value

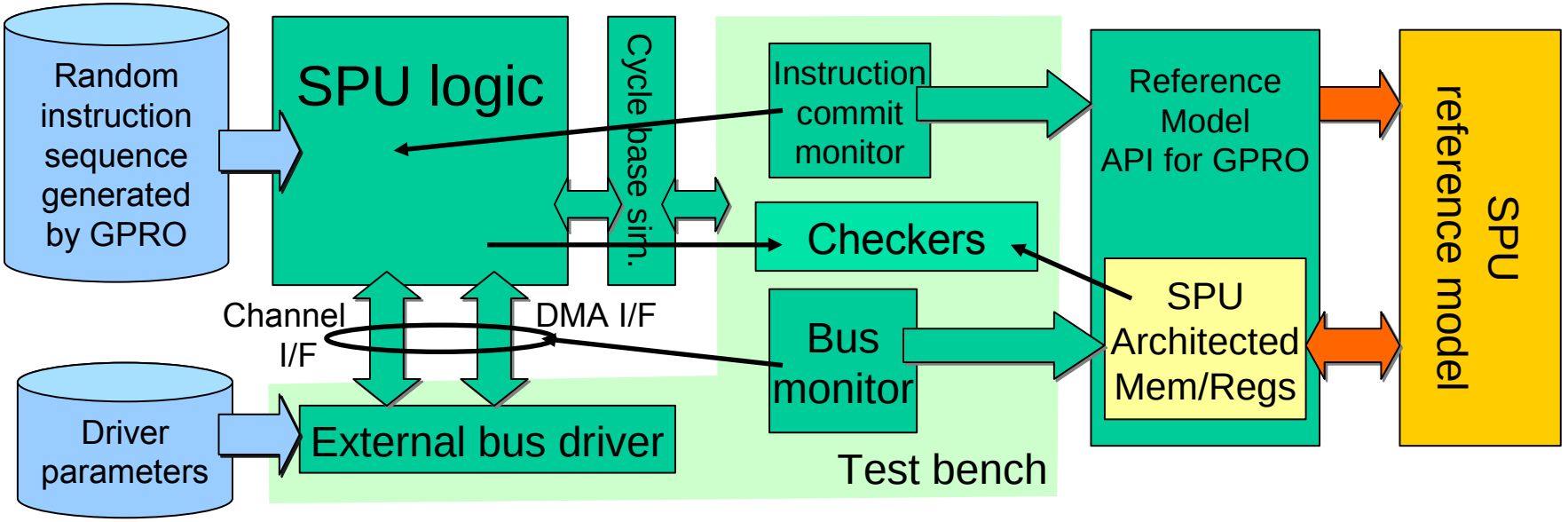
If there is no DMA transactions, sequence 2 is executed.

If there is a DMA transactions between the stqa and lqa instructions, sequence 1 is executed.

-> Instruction sequence is unpredictable

Interaction between Instruction Execution and External Transactions : Solution

- *SPU reference model* is integrated in the verification environment
- *Bus monitor* synchronizes SPU architected memory/registers with the logic when external transactions occur
- *Instruction commit monitor* checks if an instruction is committed in the logic, and execute an instruction in the reference model



Outline

- Overview of the Cell Broadband Engine and the SPU
- SPU reference model
- Applications using the SPU reference model
  - Simulator
  - Random test case generator
  - Verification environment
- Summary

Summary

- The reference model is used for
  - Simulator
    - ISA definition and validation
    - Defining the micro architecture of the SPU by changing the instruction execution latency
    - Performance evaluation
    - Software development and optimization
    - Compiler development
  - Random test case generator
    - Good quality test case generation for coverage driven verification
  - Expected value generator in the verification environment
    - Since expected values can be generated on the fly, any combination of random instruction sequence and random external transactions can be simulated properly avoiding unpredictability

Summary (cont.)

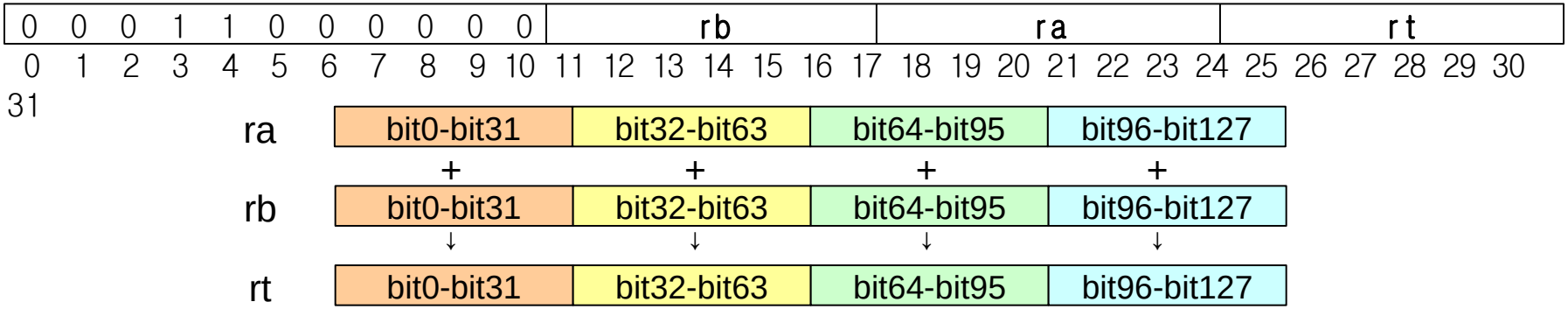
- The same reference model is used for each application
  - Only the single code had to be maintained as the definition of instructions.
  - Reduced the burden to keep up with the ISA changes or bug fixes for each application developer and reduced the likelihood of mistakes in the implementation
- By including the reference model in the verification environment, asynchronous external transactions are treated properly and various corner cases are easily covered and this improved the verification quality.
  - Only one bug was found in the first silicon which was a mistake of the specification of the asynchronous interrupt, though the SPU was a novel processor.

The END

Supplementary Slides

Example of an Instruction Execution Routine

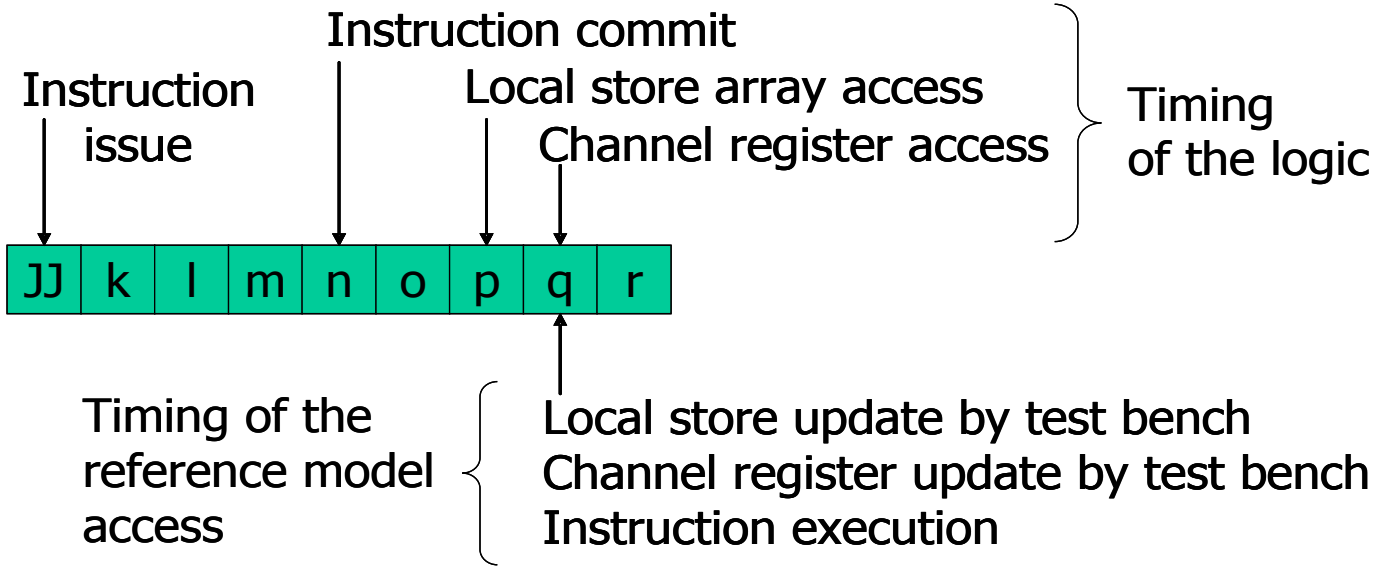
a rt,ra,rb




```
/* add word: a rt,ra,rb */
void INTERPRETER_inst_a(APU_t *apu, unsigned int inst)
{
        RTW(0) = RAW(0) + RBW(0);
        RTW(1) = RAW(1) + RBW(1);
        RTW(2) = RAW(2) + RBW(2);
        RTW(3) = RAW(3) + RBW(3);
}
```

```
#define RA(x)   (((unsigned int)(x) >>  7) & 0x7f)
#define RAW(x)  apu->reg[RA(inst)].word[(x)]
```

Timing to Access the Reference Model



- When an external transaction happens, a interface monitor detects it and update the SPU architected memory/registers structure at ‘q’ pipeline stage.
- When a monitor detects an instruction commit in the logic at ‘n’ pipeline stage, the instruction is executed in the reference model 3 cycle later at the ‘q’ pipeline stage
- When the local storage contents are compared between logic and the reference model, value at the ‘p’ stage of logic is compared with the value of the ‘q’ stage of the reference model

Extension to Generate Trace Information

- The information which memory/register resources were read and which memory/register resources were written when an instruction was executed was required
- A C++ class was defined to access SPU memory/register resources
- Macros were redefined to use the class

  #define RAW(x) (SPU_Slice(refmodel, “V”, RA(inst), (x)<<5, ((x)<<5)+31))
- Trace information is automatically added by operator overloading without changing reference model description

RTW(0) = RAW(0) + RBW(0);

When SPU_Slice class is at the left side of ‘=‘, write trace is added

When SPU_Slice class is accessed as an integer value, read trace is added

Simulator performance

	Instruction level (instructions/sec)	Pipeline level (cycles/sec)
Sample1 (No fpu instructions)	16.83M	1.83M
Sample2 (Lots of fpu instructions)	3.38M	1.63M

Platform : RHL3 with Opteron 248(2.2GHz)