University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture

Juan del Cuvillo
Weirong Zhu
Ziang Hu
Guang Gao

CAPSL Technical Memo 62

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

This paper reports our experience and lessons learned in the design, implementation and experimentation of an instruction-set level simulator for the IBM Cyclops-64 (or C64 for short) architecture. This simulation tool, named Functionally Accurate Simulation Toolset (FAST), is designed for the purpose of architecture design verification as well as early system and application software development and testing. FAST has been in use by the C64 architecture team, system software developers and application scientists. We report some preliminary results and illustrate, through case studies, how the FAST toolchain performs in terms of its design objectives as well as where it should be improved in the future.

Contents

List of Figures

List of Tables

1 Introduction

It is increasingly clear that the huge number of transistors that can be put on a chip (now is reaching 1 billion and continues to grow) can no longer be effectively utilized by traditional microprocessor technology that only integrates a single processor on a chip. A new generation of technology is emerging by integrating a large number of tightly-coupled simple processor cores on a chip empowered by parallel system software technology that will coordinate these processors toward a scalable solution.

This paper reports our experience and lessons learned in the design, implementation and experimentation of an instruction-set level simulator for the IBM Cyclops-64 architecture that integrates on a single chip up to 150 processing cores, an equal number of SRAM memory banks and 75 floating point units. This simulation tool, named Functionally Accurate Simulator Toolset (FAST), is designed for the following goals (1) architecture design verification; (2) early system software development and testing; (3) early application software development and testing. For our purposes, a cycle accurate (rather than function accurate) simulator would be too slow for a system consisting of one or more fully-populated C64 chips. Currently, FAST efficiently handles C64 systems consisting of either a single processing core, a C64 chip fully populated or a system built out of several nodes connected with a 3D mesh.

We present several important aspects of the FAST simulator and highlight the tradeoffs faced during its design and implementation. Some design decisions are made based on the unique features of the C64 architecture. For instance, C64 employs no data caches. Instead, on-chip memories are organized in two levels — global interleaved memory banks that are uniformly addressable, and scratch memories that are local to individual processing cores.

FAST has been in use by the C64 architecture team, system software developers and application scientists. We report some preliminary results and illustrate, through case studies, how FAST performs in terms of its design objectives as well as where it should be improved in the future.

2 Cyclops64 chip architecture

The Cyclops-64 (C64) is the latest version of the Cyclops cellular architecture designed to serve as a dedicated petaflop compute engine for running high performance applications [10]. A C64 supercomputer is attached — through a number of Gigabit Ethernet links — to a host system. The host system provides a familiar computing environment to application software developers and end users.

A C64 is built out of tens of thousands of C64 processing nodes arranged in a 3D-mesh network. Each processing node consists of a C64 chip, external DRAM, and a small amount of external interface logic. A C64 chip employs a multiprocessor-on-a-chip design with a large number of hardware thread units, half as many floating point units, embedded memory, an interface to the off-chip DDR SDRAM memory and bidirectional inter-chip routing ports, see

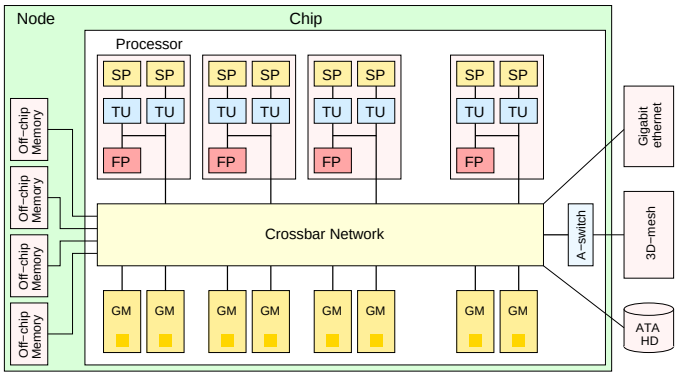


Figure 1: Cyclops-64 node

Figure 1. A C64 chip has 75 processors, each with two thread units, a floating-point unit and two SRAM memory banks of 32KB each. A 32KB instruction cache, not shown in the figure, is shared among five processors. The C64 chip has no data cache. Instead a portion of each SRAM bank can be configured as scratchpad memory (SP). The remaining sections of SRAM together form the global memory (GM) that is uniformly addressable from all thread units. On-chip resources are connected to a 96-port crossbar network, which sustains all the intra-chip traffic communication and provides access to the routing ports that connect each C64 chip to its nearest neighbors in the 3D-mesh network. The intra-chip network also facilitates access to special devices such as the Gigabit Ethernet port and the serial ATA disk drive attached to each C64 node.

The C64 architecture represents a major departure from mainstream microprocessor design in several aspects. The C64 chip integrates processing logic, embedded memory and communication hardware in the same piece of silicon. However, it provides no resource virtualization mechanisms. For instance, execution is non preemptive and there is no hardware virtual memory manager. The former means a single application can run at a given time on a set of C64 nodes. Additionally, the OS will not interrupt the user program running on the thread units unless the user explicitly specifies preemption or an exception occurs. The latter means the three-level memory hierarchy of the C64 chip is visible by the programmer. From the processing core standpoint, a thread unit is a simple 64-bit, single issue, in-order RISC processor with a small instruction set architecture (60 instruction groups) operating at a moderate clock rate (500MHz). Nonetheless, it incorporates efficient support for thread level execution. For instance, a thread can stop executing instructions for a number of cycles or indefinitely; and when asleep it can be woken up by another thread through a hardware interrupt. Additionally, the integration of processing logic and memory is further leveraged with a rich set of hardware supported in-memory atomic instructions. Unlike similar instructions on common off-the-shelf microprocessors, atomic instructions in the C64 only block the memory bank where they operate upon while the remaining banks proceed servicing other requests. This functionality provides a higher memory bandwidth.

Table 1: Simulation parameters

Component	# of units	Params./unit
Threads	150	single in-order issue, 500MHz
FPUs	75	floating point/MAC, divide/square root
I-cache	15	32KB
SRAM (on-chip)	150	32KB
DRAM (off-chip)	4	256MB
Crossbar	1	96 ports, 4GB/s port
A-switch	1	6 ports, 4GB/s port

3 FAST design and implementation

FAST is an execution-driven, binary-compatible simulator of a multichip multithreaded C64 system. It accurately reproduces the functional behavior and count of hardware components such thread units, on-chip and off-chip memory banks, and the 3D-mesh network, see Table 1. The actual number of simulated chips is limited by practical reasons, since the memory corresponding to all the chips need to be allocated in the host machine.

FAST has been developed following a modular approach, such that additional features could be easily incorporated into the existing design. To help the architecture team with the verification of the C64 chip design, the simulator executes instructions (3.1), models the architecture exceptions (3.2), reproduces the C64 memory map (3.3) and produces histograms of the instruction mix as well as detailed traces of all instructions executed (3.4). For the purpose of early system and application software design and evaluation, in addition FAST accounts for memory and interconnect contention (3.5, 3.6), supports intra-chip communication through the A-switch device (3.7) and incorporates debugging facilities (3.8). Finally, an overview of the simulator internals is provided (3.9).

3.1 Instruction execution

FAST simulates the four-stage pipeline employed in the C64 architecture, see Figure 2.

At the first stage of the pipeline, an instruction (see Table 2) is fetched from the program instruction buffer (PIB) and decoded. FAST may account for the access to the PIB and subsequent delay if the instruction has to be read from the instruction cache or memory, if a miss should occur. Whenever the branch prediction is incorrect, execution in a thread unit stalls for three cycles while the pipeline is flushed. However, FAST does not reflect the operation of the branch predictor and regards all conditional branches as correctly predicted.

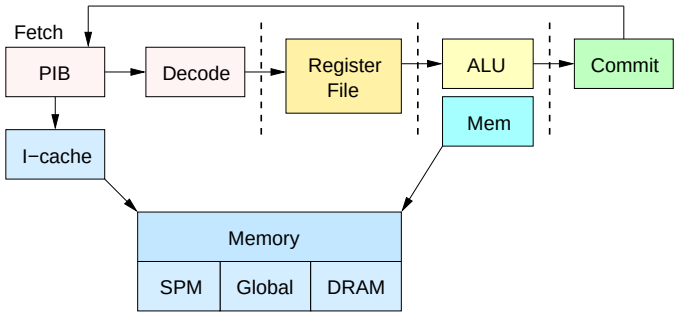


Figure 2: Four-stage instruction pipeline

Table 2: Instruction set summary

Core Integer and Branch	**Floating Point**
Load, Store	Add, Subtract
Load, Store Multiple	Multiply, Divide
Add, Subtract [Immediate]	Multiply and Add
Multiply, Divide	Conversions
Compare [Immediate]	Square Root
Trap on Condition [Immediate]	
Logic [Immediate]	
Shift [Immediate]	
Shift left 16 then OR immediate	
Insert, Extract	
Move if Condition	
Branch on Condition	
Branch and Link	
Exotic	**Control**
Bit Gather (permute bits)	All Stop
Count Leading Zeros	I-Cache Invalidate
Count Population	Move From/To SPR
Load then Op	Return from Interrupt
Multiply and Accumulate	Sleep
	Supervisor Call

In the second pipeline stage, the instruction input operands are read from the register file. For all the C64 instructions, except the floating multiply and add (FMA), one or two register operands are read in one cycle. FMA instructions have three input operands, hence an extra cycle may be required to read the third operand since the register file has two read ports.

Table 3: Instruction timing

Instruction type	x	d
Branches	2	0
Count population	1	1
Integer multiplication	1	5
Integer divide, remainder	1	33
Floating add, mult. and conv.	1	5
Floating mult. and add	1	10
Floating divide double	1	30
Floating square root double	1	56
Floating mult. and accumulate	1	5
Memory operation (local SRAM)	1	2
Memory operation (global SRAM)	1	20
Memory operation (off-chip DRAM)	1	36
All other operations	1	0

In the third stage the instruction is executed. RISC-like instructions such as integer, floating-point, branch and memory operations are modeled based on execution times expressed by x/d pairs, where x is the execution time in the ALU, and d represents the delay before the result of the instruction becomes available. Instruction timing reported in Table 3 is based on information provided by the C64 chip designer team. For instance, integer division is said to take one cycle in the ALU but a subsequent instruction will not be able to use the result until 33 cycles later. During this delay, execution of independent instructions can proceed normally. However, if the result of an instruction is to be used by another instruction before it is available, the pipeline will stall. It is the compiler and programmer responsibility to cover these delays as much as possible, with the appropriate instruction scheduling.

The result is finally committed in the fourth stage if no exception is generated. Otherwise, a context switch causes execution to continue from the address specified by the interrupt vector. When the results are to be put away, conflicts may occur, since the register file has two write ports. However, these events are not expected to happen frequently and FAST does not account for them.

In terms of instruction execution, FAST allows thread units to fetch, decode and execute instructions independently, following the sequence of events dictated by each thread's instruction stream. However, care need to be taken for some special instructions. The sleep instruction, the wakeup signal, the inter-thread interrupt, etc., all imply a synchronization between threads. For instance, a thread unit, while asleep, does not execute any instruction. During this time the simulator will not update its clock counter. When a wakeup signal is received, the clock counter is set to that of the remote thread that executed a store in the wakeup memory area (plus some delay). To handle these synchronizations, threads shall commit instructions once

the simulated chip clock reaches the time point at which the instruction is executed by the thread. In other words, although instructions are executed asynchronously they are committed in a synchronized fashion.

3.2 Exception handling

Exceptions are thread-specific events. Some are caused by instructions and trigger what we call synchronous interrupts that cannot be disabled. For instance, an attempt to execute an instruction with an invalid opcode generates an illegal interrupt. Others, known as asynchronous, are caused by events such as a timer alarm and can be disabled. While disabled, only the first exception of each type generated by a sequence of events is held pending; subsequent ones are lost. Throughout the instruction's execution, multiple exceptions of both classes may occur. FAST checks for exceptions at the end of the execution stage. Before the results are put away, if one or more enabled exception exists, FAST generates an interrupt according to the priority order specified by the architecture.

3.3 Segmented memory space

The C64 chip hardware supports a shared address space model: all on-chip SRAM and off-chip DRAM banks are addressable from all thread units/processors within a chip. That is, all threads see a single shared address space.

Architecturally, each thread unit has an associated 32KB SRAM bank. Each memory bank can be partitioned (configured) into two sections: one called "global" (or "interleaved") section, the other "local" (or "scratchpad") section. All such global sections together form the (on-chip) global memory in an interleaved fashion that is free of holes and uniformly addressable from all thread units. Although scratchpad memory, global memory and off-chip DRAM memory are addressable from any thread within the chip, the access is not uniform. Besides having different latencies, these three memories have a separate address space, resulting in a three level hierarchy. Furthermore, there is no virtual memory manager in the C64 architecture, hence this memory hierarchy is directly exposed to the programmer.

The FAST simulator accurately reproduces the C64 memory map by implementing the above mentioned non-uniform shared address space. It also includes the address upper limit special purpose registers (AULx) that define the highest existing location in scratchpad memory, global memory and DRAM memory, respectively. Nonetheless, all memory-specific parameters such as the number of banks, size of each bank, latency, and bandwidth are easily configurable. Additionally, it considers three protection boundary special purpose registers (PBx). These registers define regions in scratchpad, interleaved and DRAM memory that can only be written in supervisor state, which effectively provide a basic mechanism to protect the kernel against malign user code.

3.4 Execution trace and instruction statistics

Given the appropriate command line option, the toolset generates the execution trace of a program. There are two mechanisms to select the instructions that are to be stored in the trace. The user can either specify the time interval (in clock cycles) for which the program execution is to be traced, or enclose the instructions to be output to the trace within TraceOn/TraceOff macros. These macros access unarchitected special purpose registers, i.e. SPRs that control the simulator's functionality but are not present in the C64 chip design. The output consisting of a text file per active thread on the C64 system, contains detailed information such as clock cycle, instruction executed, source and target register values, address of the memory location touched by the instruction, if applicable, and specific information regarding events that could have delayed the execution of the instruction (contention in the crossbar network, operand not available yet, etc).

FAST may also collect instruction statistics over an execution interval and produce histograms of the instruction mix. Similarly to the procedure available for tracing, the user can specify an interval in clock cycles or use StatsOn/StatsOff macros to start/stop collecting statistics, respectively. A combined report for each node as well as individual reports for all active threads are generated.

3.5 Program instruction buffer and instruction cache

FAST models the latency for fetching instructions from the program instruction buffer (PIB) and the instruction cache (I-cache). Each thread unit has a PIB, which is like a small cache, having two lines of 16 instructions each. A PIB line is aligned on a quadword (4 instructions) boundary. On a C64 chip, there are 16 32KB SRAM I-caches, each shared by five processors (i.e. 10 thread units). Each I-cache is 8-way set associative, with 16 instructions per cache line. Cache lines are aligned on 64-byte boundaries.

During the instruction fetch stage, if the instruction is found in the PIB, it is regarded as a hit and incurs in no delay. If a miss occurs, 16 instructions aligned on a 16-byte boundary are transferred from the I-cache. Because instructions are aligned differently in the PIB and I-cache, one PIB miss may affect two cache lines. Hence, one PIB miss may result in two I-cache misses. On a miss, the I-cache sends a request to memory. The delay for this operation is provided by the memory and interconnection contention module, section 3.6. When data arrives from memory, it takes 8 additonal cycles to store each line into the corresponding cache set, 16 cycles in total.

FAST also accounts for the contention on the I-cache that happens whenever concurrent accesses are performed by any of the 5 processors that share each cache. At each cycle, only one request can be served by the I-cache. Moreover, while the I-cache is handling a miss, it can still serve cache hits. However, it can not attend any other cache misses.

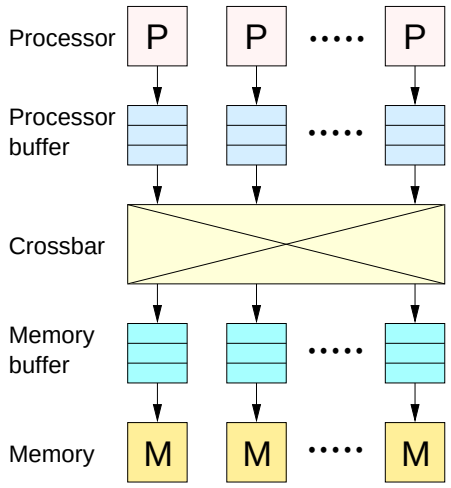


Figure 3: Interconnection to the on-chip crossbar

3.6 Memory and interconnect contention

One of the latest additions to the FAST simulator is a module that accounts for the contention in the crossbar network and in the memory system.

Figure 3 illustrates the data path between processors and memory banks on a C64 chip. Every memory instruction executed on a processor results in a network packet delivered by the crossbar network to the appropriate memory bank (global SRAM or off-chip DRAM). For load operations, the memory replies with another packet containing the data retrieved from memory.

FAST models the following sources of contention: (1) Packets issued by threads on the same processor are queued on a 7-slot FIFO (processor buffer) until they are retrieved by the crossbar. If a thread issues a memory operation when the FIFO is full, the pipeline will stall until space is available; (2) The crossbar retrieves packets from the input ports and delivers packets to the output ports, one per cycle. If at the same cycle, two packets are to be delivered to the same output port, the crossbar blocks one of them arbitrarily; (3) Between the crossbar and each memory bank there is another 7-slot FIFO (memory buffer) where packets are held until processed by the memory. Whenever this buffer becomes full, the crossbar stops delivering packets to this destination. At the same time, it stops retrieving packet from any input that tries to send packets to the blocked output port; (4) Memory latencies are also taken into account. SRAM memory banks can perform a load or store operation every cycle, i.e., 4GB/s per bank. Whereas DRAM memory can sustain a much lower bandwidth. DRAM memory consists of four banks and each bank is subdivided into four subbanks. Subbanks can service requests simultaneously, one every 32 cycles. While a memory subbank is in service, an incoming request is held pending in the memory buffer. Therefore, the DRAM bandwidth is 2GB/s for single loads and stores. For multiple transfers, using load multiple (LDM) and store multiple (STM) instructions, the DRAM bandwidth is 16GB/s instead.

3.7 A-switch device

In FAST, there are two optional modes for simulating the A-switch: message accurate and packet accurate simulation. The former is faster but less accurate, since it copies the whole message directly to the destination node. The latter models all of the hardware mechanisms involved in transferring packets double word by double word through the 3D-mesh network. However, this model is still under testing. Mainly, because it does not account for the interaction between the A-switch and the crossbar network. In other words, reading from and writing to memory while sending or receiving messages do not generate the corresponding packets in the crossbar. Therefore, performance estimations obtained with FAST for multichip simulations should be regarded as less accurate.

3.8 Debugger

FAST integrates a user-friendly assembly-level debugger. In debugging mode, there are commands to set a breakpoint, continue with the execution after a breakpoint, single-step the execution, inspect and modify the values of registers or memory, etc. Although useful, this method is tedious. To eliminate the hazard of mapping statements in the source code to assembly instructions and vice versa, a source level debugger is a necessary tool. The GNU debugger, GDB, has been partly ported to the C64 architecture.

3.9 Simulator internals

The simulated C64 system starts running when one of the three main simulator functions is called. To maximize performance, each function specifically handles a C64 system consisting of a single processing core, a C64 chip fully populated, or a system built out of several nodes. Therefore, the decision is simply based on the system configuration.

In multinode simulations, the main function starts with a loop that iterates over all the active threads on all the nodes. Each thread unit attempts to execute an instruction. For a new instruction, calls to routines that take care of instruction fetch, instruction decode, reading the input operands from the register file, and instruction execution, are invoked. If the thread unit is asleep, stalled waiting for an operand or due to a resource hazard, or waiting to commit the previous instruction, it does nothing but return.

Back in the main function, the chip clock is moved forward, just enough to allow one thread unit, at least, to commit the current instruction. Once the clock is updated, the crossbar and memory banks proceed to flush packets and memory operations that are to be performed by this time.

Then a second loop iterates over all the threads, regardless of their status. First, thread units check whether an exception occurred, and if it did, the corresponding interrupt is serviced with the appropriate context switch. If no interrupt was triggered, they try to commit the last instruction. At this stage, threads compare the chip clock with their own internal clock. When

the execution on the chip reaches the time step at which a thread can commit an instruction, the results are put away. Otherwise, the thread waits.

Finally, after the status of the A-switch is updated, execution returns to the beginning of the main loop. The process is repeated until thread units on every node execute the ALLSTOP instruction in supervisor state.

To simplify the communication among components of the simulator, the representation of the simulated C64 system is kept in a single multilevel data structure. At the chip level, it contains information regarding thread units, floating point units, on-chip SRAM and off-chip DRAM memories, I-caches, crossbar model, and A-switch. At the thread level, it accounts for general, special purpose and accumulator registers, in addition to timing information as to when the value stored in a general purpose register will be available, the last decoded instruction, program counter, exception flags, thread status, and a third-level data structure with statistics counters.

3.10 Source file descriptions

The following list describes the functionality of the code files of the FAST simulator.

- args.[c,h]: Defines and parses command line options.
- aswitch.[c,h]: Simulation of the A-switch device.
- cb.[c,h]: Defines the data structure and methods for managing the circular buffer, the memory area maintained by the A-switch to store incoming messages.
- cc.[c,h]: Utility routine that compares two operands according to any of the 40 different condition codes.
- common.[c,h]: A miscellaneous set of simple functions called from various places in the simulator.
- debug.[c,h]: Set of routines to read symbol information from the program executable.
- disasm.[c,h]: Definition of macros and functions to disassemble an instruction.
- elf.[c,h]: Load a 64-bit ELF binary file into the simulator's memory.
- exception.[c,h]: Routines to account for exceptions and to perform the context switch once an interrupt is to be processed.
- fast.[c,h]: A set of function to have the command line parsed, the Cyclops-64 machine initialized, the target program loaded before control is given to the main simulator function in machine.
- icache.[c,h]: Functions to implement the I-cache and calculate the delay caused when the pipeline stalls on a PIB or/and I-cache miss.

- kernel.[c,h]: Holds routines to load and set up the C64 microkernel. The TNT runtime system library is split into a microkernel and a user library. The former executes in the supervisor level while the latter is linked with the user's program and executes in the user level.

- machine.[c,h]: Besides holding the main simulator function, this file contains code to manage all architectural resources and defines data structures to hold the status of thread units, chips, etc.

- machine.def.h: Contains a list of macros that define each instruction, including the actions to carry out during the read operands, execution and commit stages of the instruction pipeline. Similarly to the SimpleScalar DEF file, this file defines the C64 instruction set.

- memmap.[c,h]: Support for the three-level memory hierarchy. It provides functions to check the address and generate an interrupt if appropriate, determine the memory bank and delay for a memory access.

- memory.[c,h]: Contains functions to initialize, read from and write to the target memory. Scratchpad memory, interleaved memory and DRAM memory are implemented as a flat space that is accessed based on the translated address provided by memmap.

- memqueue.[c,h]: Contains functions to model contention in FIFO and the crossbar network, and to account for the bandwidth of SRAM and DRAM memory banks.

- spr.[c,h]: Utilities to handle instructions that move from/to special purpose registers. SPRs that are not in the actual architecture are used as the interface between the simulator and the host machine to support system calls in the Cyclops-64 system.

- stats.[c,h]: Holds routines used to produce statistic reports.

- trace.[c,h]: Contains the functions that output the program execution trace to a file.

- types.h: Defines the data types used to declare architecture-specific data structures such as registers, the program counter, etc.

4 FAST usage

To run a program foo.bin in the simulator, simply enter on the command line:

```
cyclops64-linux-elf-sim [simulator_options] foo.bin [program_args]
```

The command line can include `simulator_options`, which apply to the simulator only. Options placed after the program name are passed as arguments to the program.

4.1 Command line options

The simulator will accept the following command line options:

–**aswitch or -a** Perform cycle accurate simulation of the A-switch.

–**bw** Account for the bandwidth limitation of the SRAM and DRAM memory banks. Automatically sets –crossbar option.

–**crossbar** Model contention in the crossbar switch.

–**icache** Model the instruction cache and program instruction buffer. Automatically sets –crossbar and –bw.

–**kernel** Run the program in supervisor state.

–**kernel-name=filename** Specifies the name of the kernel file. Automatically sets –kernel.

–**memory=s,i,d or -m** Specifies a memory configuration, with given sizes for the scratchpad per thread (KB), interleaved SRAM(KB), and DRAM(MB), respectively.

–**node=x,y,z or -n** Nodes in the system. For example, -n=2,3,4 represents a 24-node system arranged in a 2×3×4 3D mesh.

–**output or -o** Send stdout and stderr to a file (one file per node).

–**processor=# or -p** Number of processors per chip.

–**polling-threads=#** Number of polling threads.

–**quiet or -q** Do not produce any output (default).

–**stats=# or -s** Gather statistics starting at cycle #. -s alone means no statistics are gathered unless this feature is turned on by the program.

–**spmd** Start execution of user program in SPMD mode (default).

–**no-spmd or –spsd** Start program execution running only one user thread.

–**summary** Output an execution summary.

–**trace=# or -t** Trace execution starting at cycle #. -t alone means no trace is generated unless this feature is turned on by the program.

–**user** Run program in user mode (default).

–**user-threads=#** Reserve a number of user threads. These threads will not participate in the SPMD execution.

–**verbose or -v** Produce verbose output. Multiple -v (-vv for instance) increases verbosity. This option controls the amount of information dumped to stdout and written into the trace files.

–**help** Display a help list.

–**usage** Display a short usage message.

4.2 Kernel execution

By default, the simulator loads and executes a microkernel (included as part of the software distribution), which is responsible for the execution of the user program starting from main. If needed, a different kernel file can be specified using the option `--kernel-name`.

In some instances, a program needs to be run in supervisor state, i.e., without loading and executing the kernel first. This can be done by providing the option `--kernel`. In our regression testsuite, for instance, the interrupt handlers are tested using this procedure.

While using the microkernel, the user needs to be aware that at least one thread is reserved for kernel operations. Since the Cyclops-64 architecture does not support preemption, the runtime system can only control execution on thread units remotely, by means of the inter-thread interrupt. Therefore, having a thread unit in supervisor state ensures that a chip can be brought to a predetermined status in the event that a program stops responding.

4.3 SPMD execution

After initialization, the microkernel automatically spawns all available threads, which then start the execution of the user program from main. This is called execution in SPMD mode, and it is the default in the simulator. The user still has some control over the total number of threads spawned by the kernel. For instance, the simulator can be told to leave a number of thread units idle so that the user program can assign them a function afterwards. This is done with option `--user-threads`. Additionally, when inter-chip communication is expected to be intensive, the user can instruct the communication library to have a certain number of threads polling the A-switch. This is done with option `--polling-threads`. The reason for having two separate options is that the latter leaves threads in supervisor state, which is a requirement to operate the A-switch, whereas the former results in threads switching to problem or user state.

Finally, execution in SPMD mode can be disabled with the option `--no-spmd`. In this scenario, the microkernel spawns a single thread, so the user program needs to handle the remaining threads itself. In this mode, the option `--polling-threads` still applies.

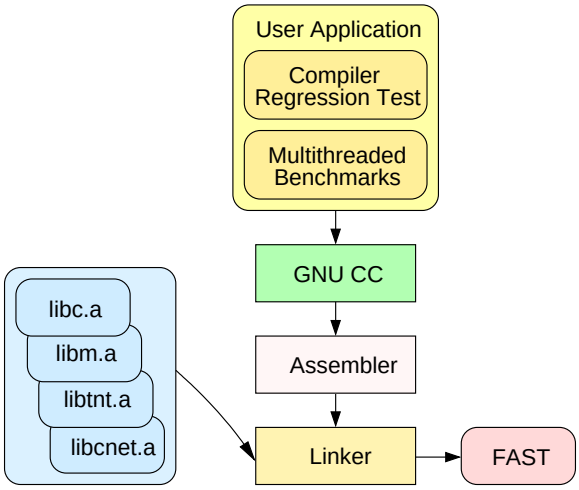


Figure 4: Cyclops-64 software toolchain

5 Experience

The goal of FAST is three-fold: FAST is designed for the purpose of architect design verification (section 5.1). As part of the C64 toolchain, FAST provides the basic platform for early system software development and testing (section 5.2). FAST has been in use by other users for application development and testing. Although not cycle accurate, the timing information provided by the simulator has proven to be useful for performance estimation and application tuning as well (section 5.3).

5.1 Design verification

For the purpose of architecture design verification, the execution trace generated by FAST is compared to the output of the VHDL simulator that reproduces the C64 at a gate level. Initial verification of the C64 design was carried out following this procedure with a set of short programs intended to test the C64 instruction set architecture [11].

The Cyclops-E is another cellular architecture design, target to the embedded market. The first hardware implementation of a single-chip Embedded Cyclops system was accomplished with DIMES, an FPGA-based multiprocessor emulation system [21]. Concurrently with the development of DIMES an earlier version of the FAST simulator, known as CeDIMES, was also implemented [9]. Since this simulation tool is also binary-compatible, once the hardware emulation system was brought up, design verification started immediately. A test suite consisting of more than 200 programs specifically designed to test the Cyclops-E ISA were run on the actual hardware platform and the results were compared to those produced by the simulator. The initial testing revealed a few bugs in the chip design, which were fixed by the chip architect.

5.2 System software development

5.2.1 Toolchain

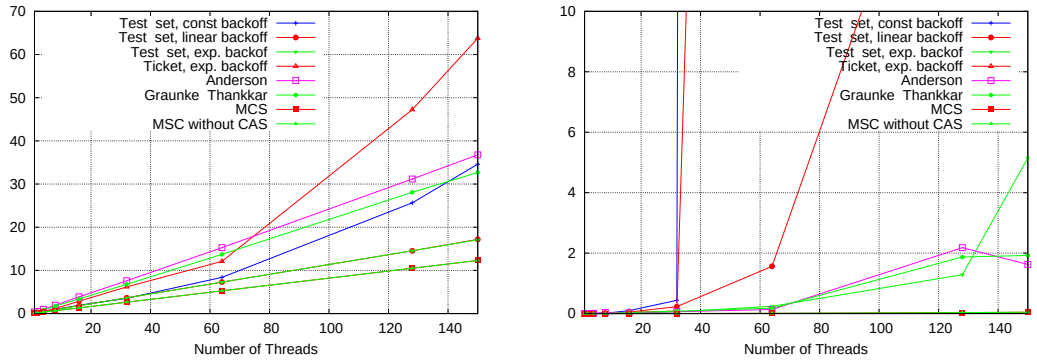
FAST is part of the software toolchain available for application development on the C64 platform, see Figure 4. Programs written in C or Fortran are compiled using a porting of the GCC-3.2.3 suite. The assembler and linker, which are based on binutils-2.11.2, along with the necessary libraries, produce a 64-bit ELF executable that can then be loaded into FAST and executed. The C standard and math libraries are based in newlib-1.10.0. In addition, we wrote the TNT runtime system and the CNET communication libraries to manage hardware resources such as the thread units and the A-switch, respectively.

5.2.2 Thread library

We reported our work in the design of a thread model for C64 that maps directly to the architecture assisted by a native thread runtime library, called TNT (or TiNy Threads) [8]. In the development, debugging and evaluation of the TNT library, FAST's capability to accurately simulate a large number of hardware threads with practical time has proven to be useful.

5.2.3 Spin lock

For a thread library, it is important that all components are efficiently implemented. In multithreaded environments, especially for architectures like C64 with 150 threads on a chip, spin lock, as an indirect synchronization mechanism is known to be a key factor for scalability. For this reason, we conducted a study on spin lock algorithms on the C64 architecture. We implemented eight programs based on well known spin lock algorithms: three based on test-and-set, one on tickets, two on array queue, and two on list queue [15]. All programs consist of a short critical section (a single variable update) enclosed within calls to procedures that acquire and release a lock following the corresponding algorithm. The entire process (lock, critical section, unlock) is repeated a thousand times as part of a loop body. We run the programs on FAST with memory contention enabled and measure the execution time as well as the overhead due to contention in the crossbar. As expected, the results show list-based queuing locks are the most efficient algorithms, see Figure 5(a). On C64, contrarily to most shared memory multiprocessors, array-based queuing lock methods do not perform well, because there is no data cache. In other words, accesses to the array queue are as expensive as any memory operation seen in test-and-set based implementations. Indeed, test-and-set based algorithms with linear and exponential backoff perform better. Not surprisingly, list queue locks generate the least amount of memory traffic on the crossbar, since threads spin locally on their own scratchpad memory, see Figure 5(b). That means they would interfere least with the normal execution of a program if it had additional memory accesses. As a result of this experience, the implementation of mutexes in the TNT library is based on list queue locks.



(a) Execution time (in ms)

(b) Execution delay (in ms)

Figure 5: Spin lock programs

5.2.4 Communication library

With the A-switch module, FAST can be used to simulate C64 multichip system configurations. With this feature, we developed a communication library implemented as several layers, each accessible through its own interface. At the lowest level, the packet transfer layer accesses the A-switch directly, hiding all hardware details from layers above. A second layer, built on top of the packet transfer layer, provides user-level remote memory read and write, inter-chip synchronization primitives and remote procedure call mechanisms. Finally, we are in the process of porting the SHMEM library [17] to the C64 architecture based on the two previous layers.

5.3 Application development and evaluation

To demonstrate FAST is functionally accurate, stable and hence, useful for software development and performance estimation, we write several benchmarks programs to confirm that the trends predicted by the simulator match to what the C64 architecture is capable of.

5.3.1 GUPS

Table Toy, which is also called Random Access benchmark, is an important benchmark included in the *HPC Challenge Benchmark Suite* [1]. It uses a metric known as GUPS (Giga Updates Per Second) to evaluate the random access capabilities of the memory system. In the context of this experience, we use Table Toy to verify that FAST reflects the C64 memory system accurately.

The kernel operations of Table Toy can be summarized as follows:

```
1    tmp1 = stable[j];     (load)
2    tmp2 = table[i];      (load)
3    val = tmp2 xor tmp1;  (xor)
4    table[i] = val;       (store)
```

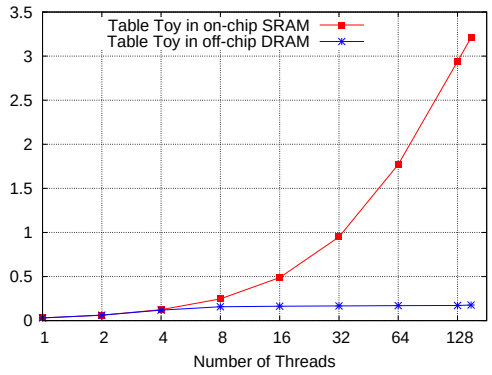
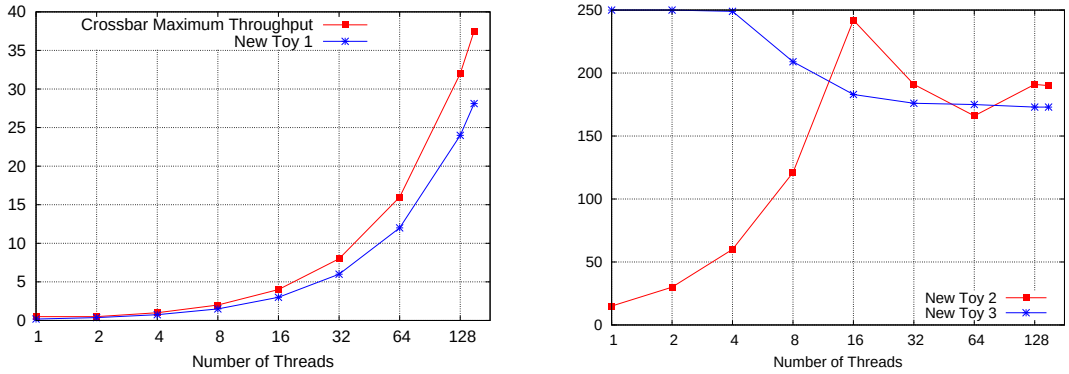


Figure 6: Table Toy on a C64 node (in GUPS)

The i, j are the pseudo random locations chosen for *table* and *stable*. Ideally, thread units should access different locations of the *table* to avoid conflicts. The *table* can be placed either in the on-chip SRAM or the off-chip DRAM, and is accessed by all thread units. The substitution table (*stable*) is allocated in the thread's scratchpad memory. The key point is that the last three operations (load, xor, and store) must be atomic in a multithreaded execution.

Figure 6 shows the GUPS obtained on a C64 node with up to 150 thread running in parallel. By taking advantage of C64's *xor_m* in-memory atomic instruction (xor to memory), we guarantee the atomicity needed while all data dependences are removed from the kernel loop. Therefore, the number of memory updates is actually the number of *xor_m* instructions issued. If the updates are performed in SRAM, the curve scales well as the number of threads increases, due to the large on-chip memory bandwidth. On the other hand, the off-chip DRAM memory bandwidth is limited. Consequently, the DRAM curve flattens when the number of threads exceeds 16. In both cases, the maximum achievable memory bandwidth is not reached. It appears that the pseudo random numbers generated in Table Toy result in several threads accessing a memory bank at the same time. Hence, the bandwidth limitation is not due to the crossbar network but to conflicts accessing the memory banks.

To prove our hypothesis, we write three separate microbenchmarks with a deterministic access pattern to the memory banks. In our first microbenchmark, New Toy 1, each thread issues 3 store operations every 8 cycles. In addition, each thread targets one SRAM bank only. Therefore, a processor issues 6 store operations every 8 cycles to the on-chip SRAM memory. This represents 75% of the peak throughput of the crossbar which is indeed achieved because there are no conflicts as the memory bank addressed by each thread unit is different, see Figure 7(a). The other microbenchmarks test the off-chip DRAM memory subsystem in different ways. In New Toy 2, each thread targets one of the 16 DRAM subbanks based on the thread identifier. Therefore, threads 0 and 16 access subbank 0, threads 1 and 17 access subbank 1, and so forth. A DRAM subbank can only service one request every 32 cycles, this is 15 MUPS, but all 16 subbanks can service requests in parallel. Figure 7(b) confirms the expected result. As long as no more than 16 threads are active, the DRAM throughput increases linearly, at a rate of 15 MUPS per thread, up to 250 MUPS. In New Toy 3, every thread executes 16



(a) GUPS from New Toy 1

(b) MUPS from New Toy 2 and 3)

Figure 7: New Toys on a C64 node

consecutive stores every 22 cycles and each store targets one of the DRAM subbanks. That means a thread can issue operations to memory faster than the memory can handle. Figure 7(b) shows that for a small number of active threads, contention can be tolerated, and the crossbar and DRAM memory system deliver the peak throughput, 250 MUPS. Finally, as contention increases, performance drops.

5.3.2 Matrix-matrix-multiply

As an example of what an application developer can expect to learn using the FAST toolset, we hereby report a tuning experience using the matrix-matrix-multiply program for a problem size of 1024×1024. Throughout this exercise we use the simulator's accurate time counter, the histograms file with the instruction mix and the execution trace to determine the cause of delays and/or bottlenecks that may prevent the program from achieving higher performance.

Our baseline is a straightforward sequential code with the matrices stored in DRAM. The program that is compiled with -O3, achieves 16.7 MFLOPS. From the trace file, we found that the main reasons for the low performance are the poor data re-usage and the long latency to access DRAM. In order to improve the performance, we unroll the two outermost loops 4 times each and manually prefetch data and re-schedule the instructions with the hints from the trace files generated by FAST. In the resulting code, data is fed to the floating point unit in a pipelined manner such that all load latencies are hidden. This implementation achieves 216.1 MFLOPS, a speedup of 13 compared to the baseline version. We also parallelized our tuned MxM program to make use of multiple thread units. As shown in Figure 8, the curve of the parallel version scales almost linearly up to 32 threads and then flats out because of the bandwidth limitation. Afterwards, it even drops when memory contention becomes too high. We believe higher performance can be achieved by employing other techniques. However this is not the purpose of this experiment.

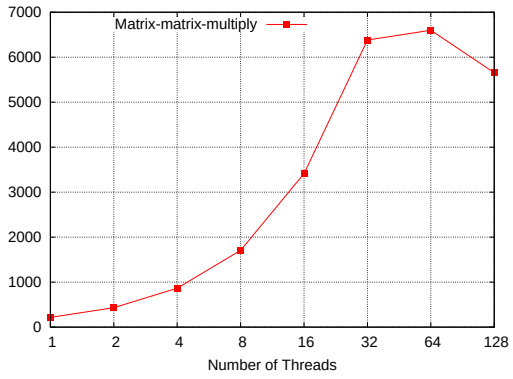


Figure 8: MFLOPS for the matrix-matrix-multiply with prefetching on a C64 node

5.3.3 Multi-chip benchmarks

To verify the correctness (not the accuracy) of FAST's multichip simulation and the communication libraries, we developed an assorted set of multichip multithreaded benchmarks. It includes implementations of matrix-matrix-multiply, 1D Laplace solver, heat conduction and Sobel edge detection.

6 Related work

To analyze and understand the impact of various architectural parameters and components as well as study the application performance and get detailed statistics, both academia and industry developed a number of simulators. Simulation frameworks for microarchitecture research and design exploration, such as SimpleScalar [7, 3], Microlib [18], Liberty [23], RSIM [12] and Turantdot [16], concentrate on accurately modeling the architecture design and normally they are cycle accurate. FAST is a functional simulator since cycle accurate simulation would be too slow for a system consisting of one or more C64 chips. There are also full system simulators capable of running commercial workloads on an unmodified operating system, such as SIMOS [20], Simics [13] and PearPC [4]. A C64 compute engine is attached to a host system. The host system provides a familiar computing environment to application developers. FAST only simulates the compute engine running a custom microkernel, whereas conventional OS services are provided by the native OS running on the host system.

Recently a new generation of simulators capable of simulating SMT and CMP architectures have been developed: SMTSIM [22], SESC [19], GEMS [14], M5 [5] and Mambo [6]. It would appear the latter simulation frameworks as well as extensions of SimpleScalar, Simics, SimOS and Turantdot are normally used to simulate 2/4/8 way SMT/CMP processors under multiprogramming, thread level speculation, and commercial workloads[1]. FAST is designed to simulate and model a CMP system consisting of several C64 nodes, each with up to 150 processing

[1]Based on papers published in HPCA from 2000 to 2005.

cores. However, the C64 architecture is designed for the purpose of running massively parallel applications, which deal with the complexity of scientific and engineering multithreading workloads.

Probably, the closest related work to FAST is the Cyclops-32 simulator. These simulators are as similar as the architectures they simulate. However, there are significant differences as well. For instance, FAST detects dependences and conflicts as instructions are executed. Therefore, it directly produces performance estimates. On the other hand, the C32 simulator does not have timing information. Performance estimates are generated by two other performance tools (a cache simulator and a trace analyzer) that post-process the execution trace produced by the simulator [2].

7 Summary

This paper presents FAST, a functionally accurate simulation toolset for the IBM Cyclops-64 architecture that is fast, flexible and efficient. To the best of our knowledge, it is the only simulation tool capable of simulating multichip multithreaded cellular architectures with reasonable accuracy and practical speed. We report some preliminary results and illustrate, through case studies, how the FAST tool chain accomplishes its purpose of architecture design verification as well as early system and application software development and testing.

As future work, we plan to increase the amount of profile information FAST produces, including text and data symbols, and to incorporate integer counters to facilitate the performance analysis of multithreaded programs.

Acknowledgments

We acknowledge support from IBM, in particular, Monty Denneau and Henry Warren. We thank ETI for support of this work. We also acknowledge our government sponsors. Finally, we also thank many CAPSL members for helpful discussions.

References

[1] HPC challenge benchmark. URL http://icl.cs.utk.edu/hpcc.

[2] G. S. Almási, C. Caşcaval, J. G. Castaños, M. Denneau, W. Donath, M. Eleftheriou, M. Giampapa, H. Ho, D. Lieber, J. E. Moreira, D. Newns, M. Snir, and H. S. Warren, Jr. Demonstrating the scalability of a molecular dynamics application on a petaflops computer. *International Journal of Parallel Programming*, 30(4):317–351, August 2002.

[3] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[4] S. Biallas. PearPC - PowerPC architecture emulator, May 2004. URL http://pearpc.sourceforge.net/.

[5] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. *Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, pages 36–43, Anaheim, California, February 9, 2003. Held in conjunction with the 9th International Symposium on High-Performance Computer Architecture.

[6] P. Bohrer, J. Peterson, M. Elnozahy, R. Rajamony, A. Gheith, R. Rockhold, C. Lefurgy, H. Shafi, T. Nakra, R. Simpson, E. Speight, K. Sudeep, E. V. Hensbergen, and L. Zhang. Mambo: A full system simulator for the PowerPC architecture. *ACM SIGMETRICS Performance Evaluation Review*, 31(4):8–12, March 2004.

[7] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin at Madison, Madison, Wisconsin, June 1997.

[8] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao. TiNy Threads: A thread virtual machine for the Cyclops64 cellular architecture. *Fifth Workshop on Massively Parallel Processing*, page 265, Denver, Colorado, April 8, 2005. Held in conjunction with the 19th International Parallel and Distributed Processing Symposium.

[9] J. del Cuvillo, R. Klosiewicz, and Y. Zhang. A software development kit for DIMES. CAPSL Technical Note 10 Revised, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, March 2005.

[10] M. Denneau. Computing at the speed of life: The BlueGene/Cyclops supercomputer. CITI Distinguished Lecture Series, Rice University, Huston, Texas, September 25, 2005.

[11] M. Denneau. Personal communication, February 2005.

[12] C. J. Hughes, V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: Simulating shared-memory multiprocessors with ILP processors. *Computer*, 35(2):40–49, February 2002.

[13] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[14] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. Submitted to Computer Architecture News.

[15] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[16] J. Moreno, M. Moudgill, J. Wellman, P. Bose, and L. Trevillyan. Trace-driven performance exploration of a PowerPC 601 OLTP workload on wide superscalar processors. In *First Workshop on Computer Architecture Evaluation using Commercial Workloads*, Las Vegas, Nevada, February 1, 1998. Held in conjunction with the 4th International Symposium on High-Performance Computer Architecture.

[17] K. Parzyszek, J. Nieplocha, and R. A. Kendall. General portable SHMEM library for high performance computing. In *Proceedings of SC2000: High Performance Networking and Computing*, Dallas, Texas, November 4–10, 2000. URL http://www.supercomp.org/sc2000/proceedings/.

[18] D. G. Pérez, G. Mouchard, and O. Temam. Microlib: A case for the quantitative comparison of micro-architecture mechanisms. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 43–54, Portland, Oregon, December 4–8, 2004.

[19] J. Renau. SESC, 2004. URL http://sesc.sourceforge.net.

[20] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using the SimOS machine simulator to study complex computer systems. *ACM Transactions on Modeling and Computer Simulation*, 7(1):78–103, January 1997.

[21] H. Sakane, L. Yakay, V. Karna, C. Leung, and G. R.Gao. DIMES: An iterative emulation platform for multiprocessor-system-on-chip designs. In *Proceedings of the IEEE International Conference on Field-Programmble Technology*, pages 244–251, Tokio, Japan, December 15–17, 2003.

[22] D. M. Tullsen. Simulation and modeling of a simultaneous multithreading processor. In *Proceedings of the 22nd Annual Computer Measurement Group Conference*, pages 384–393, San Diego, California, December 10–13, 1996.

[23] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural exploration with Liberty. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 271–282, Istanbul, Turkey, November18–22, 2002.