Gem5-X
Full System Manual

[*]EMBEDDED SYSTEMS LABORATORY,
SWISS FEDERAL INSTITUTE OF TECHNOLOGY, LAUSANNE (EPFL)

[‡]SCHOOL OF ENGINEERING AND MANAGEMENT VAUD (HEIG-VD),
UNIVERSITY OF APPLIED SCIENCES WESTERN SWITZERLAND (HES-SO)

[**]DEPARTMENT OF COMPUTER ARCHITECTURE,
COMPLUTENSE UNIVERSITY OF MADRID

YASIR QURESHI[*], WILLIAM SIMON[*], MARINA ZAPATER[*‡], KATZALIN OLCOZ[**], AND DAVID ATIENZA[*]

July 2020

Contents

1 Executive Summary

1.1 Abstract

The gem5 architectural simulator is well established and widely used in both the industry and academia. Based on gem5, we present we present gem5-X (*"a gem5-based full-system simulator with architectural eXtensions*"), a simulation framework that enables fast profiling and architectural exploration and optimization for system level architectural innovations. Gem5-X provides out-of-the-box simulation of ARM based systems with full Linux stack, along with several architectural extensions like ISA extensions, clustering, heterogeneous many-core simulation and HBM2 memory model. Several enhanced features have also been added, like advanced check-pointing, workload automation (WA) and gperf profiler support. In this technical manual, we provide guidelines on how to use various architecture features and support enhancements of gem5-X. More information on downloading and source code for gem5-X can be found at https://esl.epfl.ch/gem5-x

1.2 Release Information

Version	Date	Changes
v2.0	August 2021	Core clustering, heterogeneous cores and SPM support added in Gem5-X.

1.3 Collaboration and Contact Information

The maintainers of this project can be contacted via email at {yasir.qureshi, william.simon, marina.zapater, david.atienza}@epfl.ch and katzalin@ucm.es.

Because the scope of this project is very large, we are always interested in potential collaboration efforts to develop new features and keep gem5-X updated to gem5 master. For inquiries, source code, and additional information, please contact one of the aforementioned emails.

2 Running gem5-X Full System (FS) Mode with ARMv8 and Linux

In this chapter we describe how to configure and run our ARMv8 64-bit FS simulation in gem5-X .

2.1 Necessary Files

Because our model is run in FS mode with a full Linux environment, we need several major system components. This includes,

- A bootloader
- A kernel binary, e.g., vmlinux
- A disk image
- A device tree binary

All of the aforementioned components must be compatible with the ARMv8.

2.1.1 Full System Files

Once you register for gem5-X at https://esl.epfl.ch/gem5-x, you will receive an email with a link to all the system files, except for the device tree. The file downloaded is named ***full_system_images.tar.gz***. This contains the disk image, bootloader and kernel binary. Follow the instructions below to set it up

```
1 tar -zxvf full_system_images.tar.gz
```

The files are as follows:

- Bootloader is under **[path_to_full_system_images]/binaries/**
- Kernels (vmlimux and vmlinux_wa) are at **[path_to_full_system_images]/binaries/**
- Disk image (gem5_ubuntu16.img) can be found at **[path_to_full_system_images]/disks/**

We now need to setup the path to *full_system_images*, so that the files under it can be used and recognized by gem5-X during FS simulation.

```
1 cd <path_to_gem5-X>
2 ./apply-patch.sh <PATH_TO_FULL_SYSTEM_IMAGES>
```

Alternatively, you can also do

```
1 export M5_PATH=<PATH_TO_FULL_SYSTEM_IMAGES>
```

The full system files are now setup and ready to be used in FS mode.

2.1.2 Device Tree

The device tree files are under

```
<path_to_gem5-X>/system/arm/dt
```

If running on an Ubuntu-based host system , the following prerequisites need to be installed before generating the device tree binaries.

```
sudo apt-get install  gcc-arm-linux-gnueabihf gcc-aarch64-linux-gnu
sudo apt-get install device-tree-compiler
```

To generate the device tree binary files,

```
cd <path_to_gem5-X>
make -C system/arm/dt
```

2.2 Quick-Start Guide

In this brief start-up guide, we will guide you through the basic steps to running your first full system (FS) simualtion with gem5-X . This guide assumes you have already the bootloader, device tree, kernel file, and disk image setup as described in the previous sections.

2.2.1 Prerequisites

You will need to set up the gem5-X environment in order to compile and run the gem5-X binary using the scons (SConstruct) builder. If running on an Ubuntu-based host system, you can use the following command to get all the required libraries:

```
    sudo apt install build-essential git m4 scons zlib1g \
    zlib1g-dev libprotobuf-dev protobuf-compiler libprotoc-dev \
    libgoogle-perftools-dev python-dev python-six python \
    libboost-all-dev swig
```

2.2.2 Building the gem5 Binary

Once the above is done, you will need to build a ARM gem5 binary. You can create multiple builds including .fast, .opt, and .debug. If you are only concerned about running experiments, it is recommended to only create gem5.fast. However, if you need to debug anything or want to generate traces, you will need to build gem5.opt or gem5.debug. Do this with the following:

```
    cd <path_to_gem5-X>/
    scons build/ARM/gem5.{fast, opt, debug}
```

Additionally, if you would like to speed up the compilation process, you can use the option ”-jN” on the scons build line where N is the number of threads you want to assign for compilation.

2.2.3 Running Your FS Simulation

Once the build process is complete you can launch your simulation in the following way

```
cd <path_to_gem5-X>/

./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<full_path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_1cpu.dtb \
-n 1 \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--mem-type=DDR4_2400_4x16 \
--mem-ranks=4 \
--mem-size=4GB \
--sys-clock=1600MHz \
```

At this point you should be able to connect to your running gem5 instance in another terminal with,

```
telnet localhost 3456
```

Alternatively, you can also build the terminal program provided with gem5-X and use it

```
cd <path_to_gem5-X>/util/term/
make
m5term 127.0.0.1 3456
```

Upon connecting to your gem5 instance, you should be able to the kernel dmesg, followed finally by a login and a terminal in the gem5-X FS mode

3 Support Enhancements of Gem5-X

In this chapter we will look into the following support enhancements we have added in gem5-X:

- Enhanced checkpointing
- gperf profiler
- File sharing between gem5-X and host system using 9P over Virtio
- Modifying disk image using QEMU

3.1 Enhanced Checkpointing

The boot process during the FS simulation in gem5-X automatically takes a checkpoint when the boot and login is complete and we get the terminal. Since the boot is in SimpleAtomic CPU model, the timing information is not there in the simulation. We can now switch to an accurate in-order or our-of-order (OoO) CPU model with all the timing information.

If your simulation is still running after the boot, you can exit it using the following command in the connected terminal,

```
m5 exit
```

Now we can use the checkpoint, that was automatically taken after boot and login, and switch to an accurate CPU model.

```
./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<full_path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_1cpu.dtb \
-n 1 \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--mem-type=DDR4_2400_4x16 \
--mem-ranks=4 \
--mem-size=4GB \
--sys-clock=1600MHz \
-r 1 \
--cpu-type={MinorCPU, DerivO3CPU}
```

The number after *–r* is the checkpoint number. In this case we are resuming from the first checkpoint. The CPU type can be *MinorCPU* for in-order core or *DerivO3CPU* for OoO cores.

Sometimes it is feasible to take a checkpoint using SimpleAtomic CPU model just before your region-of-interest (ROI) and then switch to an accurate in-order or OoO CPU. You can do this in either the command prompt or a script using the following command:

```
m5 checkpoint
```

If you are in a C/C++ program, you can use the following call within the program,

```
system("m5 checkpoint");
```

3.2 Gperf Profiler

Profiling capabilities within FS, by installing the gperf profiler on the disk image. The gperf statistical profiler developed by Google provides profiling capabilities on gem5-X itself with minimal overhead, enabling the identification of application bottlenecks and exploration of the effectiveness of architectural modifications and extensions.

To enable profiling using gperf when running a program, follow the instructions below;

```
LD_PRELOAD=/usr/lib/libprofiler.so.0 CPUPROFILE=<FILE_TO_SAVE_PROFILING>
CPUFREQUENCY=1000 <program>
```

This will launch the program to be profiled with profiling data being saved to file mentioned in *CPUPROFILE* parameter.

To view the data, we first convert it to .pdf file and then write it to the host machine as follows;

```
1 google-pprof --pdf <FILE_WITH_PROFILING_DATA> > <FILENAME>.pdf
2 m5 writefile <FILENAME>.pdf
```

The file *FILENAME.pdf* will now be available to be viewed in the host system under path passed to –d parameter when launching gem5-X simulation

```
-d /path/to/your/output/directory
```

3.3 9P over Virtio

We utilize the 9P protocol developed by Bell Labsover a virtio device driver to allow fast modification of files without modifying the root file system in gem5-X. While this feature is available in vanilla gem5, it is not enabled by default and has no kernel support. Both of these features are provided in gem5-X. Once Linux is booted, a folder on the host machine can be mounted within gem5 to access files on the host system. Without 9P mounting, every time a program is modified, we need to reload the disk image required for FS simulation and reboot Linux. In gem5, this process can take up to 20-30 minutes, a bottleneck that gem5-X eliminates. To use 9p over Virtio, follow the instructions below:

- First we need to install DIOD

  ```
  sudo apt-get install diod
  ```

- After installation, check where DIOD is installed by typing "which diod". This path should be updated in the file src/dev/virtio/VirtIO9P.py at line 62. Then re-compile gem5-X using scons command as usual.

```
cd <path_to_gem5-X>/
scons build/ARM/gem5.{fast, opt, debug}
```

- Use kernel "vmlinux_wa", during the gem5 simulation. This file is provided with gem5-X under full_system_images/binaries
- Use the following additional parameter when launching the simulation

  ```
  workload-automation-vio=<FULL_PATH_TO_SHARED_FOLDER_ON_HOST_SYSTEM>
  ```

- Once the system is booted, run the following in gem5 terminal

  ```
  mount.sh <FULL_PATH_TO_SHARED_FOLDER_ON_HOST_SYSTEM>
  ```

- Now any file under the "SHARED_FOLDER_ON_HOST_SYSTEM" appears in the /mnt directory in gem5 simulation.

3.4 Modifying disk image using QEMU

To run experiments an application and benchmarks in gem5-X, they need to be on the disk image. To to this we need to update and modify the disk image with the applications.

QEMU is used to modify the disk image. If running on an Ubuntu-based host system , the following prerequisites need to be installed.

```
sudo apt-get install qemu qemu-user qemu-system qemu-user-static
```

To mount the image

```
cd <PATH_TO_FULL_SYSTEM_IMAGES>/disks/
mkdir local_mnt
sudo mount -o loop,offset=$((2048*512)) gem5_ubuntu16.img local_mnt
sudo mount -o bind /proc local_mnt/proc
sudo mount -o bind /dev local_mnt/dev
sudo mount -o bind /dev/pts local_mnt/dev/pts
sudo mount -o bind /sys local_mnt/sys
```

Now we *chroot* into the image emulating using QEMU

```
cd local_mnt/
sudo chroot ./
```

At this point we are in the ARMv8 disk image and can now compile or download applications within the image. Since it is a Ubuntu 16.04 image, you can run the following first, before installing any new packages on it,

```
apt-get update
```

When the disk image has been update with the applications or benchmarks, we can exit it and unmount the image.

```
exit
cd ..
sudo umount local_mnt/proc
sudo umount local_mnt/dev/pts
```

```
5 sudo umount local_mnt/dev
6 sudo umount local_mnt/sys
7 sudo umount local_mnt
```

The modified image is now ready to be used for gem5-X simulation with new applications or benchmarks

4 ARMv8 ISA Extension

This guide describes how to extend the ARMv8 ISA, using as an example the creation of a custom “ADD1” instruction, which does exactly the same than the ARM ADD instruction, but using one of the unallocated opcodes.

In order to extend the ISA of any architecture in Gem5, some unallocated opcodes need to exist. There are several opportunities to extend the ARMv8 ISA in gem5-X as there exist a lot of unallocated opcodes. The complete ARMv8 ISA can be found at:

https://static.docs.arm.com/ddi0487/ca/DDI0487C_a_armv8_arm.pdf

4.1 Adding a new custom instruction to the ARMv8 ISA

The first step towards extending the ISA is to find an unallocated opcode. If you have a look at section “C4.1 A64 instruction set encoding” of the ARM ISA manual, you will find an unallocated opcode field in Table C4-1. We will use this unallocated field to add our custom instruction. To add it in gem5-X

1. Go to the ARM ISA folder

   ```
   cd src/arch/arm/isa/
   ```

2. Go to the Formats folder

   ```
   cd formats
   ```

3. Open the file aarch64.isa, which contains the top-level decoder functionality

   ```
   vim aarch64.isa
   ```

4. Go to the end of the file and look for the function “def format Aarch64()”. This is where the top-level decoding is done according to Table C4-1. You will see that when bit[27]=0 and bit [28]=0, there is a call to “Unknown64()” function, as there is no instruction allocated for this opcode. This is where we will add our instruction. You can remove the line where there is a return of Unknown4() and add the following, where our instruction will be decoded to “decodeCusDataProcImm” function.

   ```
   // bit 28:27=00
   return decodeCusDataProcImm(machInst);
   ```

5. Now, we need to add this new function. To do it, add it in the beginning of the file under

   ```
   output header {{
   namespace Aarch64
   {
   ```

6. Now, to add the ADD functionality to this new function, we implement it in a similar way as is done for “AddXImm” function in the file. We also name our new instruction as Add1XImm. Please refer to the modified aarch64.isa file that can be found here under the *modified_files* folder.

7. The above was to add the instruction into the decoding path. To have the actual functionality implemented as ADD, we need to modify the following:

```
cd ../insts/          (full path is gem5/src/arch/arm/isa/insts)
vim data64.isa
```

8. Search for the line containing the original ADD instruction as

```
buildDataInst("add", "Dest64 = resTemp = Op164 + secOp;", "add")
```

9. Below this we will add our custom ADD instruction as

```
buildDataInst("add1", "Dest64 = resTemp = Op164 + secOp;", "add1",
overrideOpClass="CusAluOp")
```

10. Note that have assigned this instruction an OpClass of CusAluOp. To enable this, we had to add the overrideOpClass parameter to different functions in the file. Please see the attached data64.isa file here under the *modified_files* folder to see how we did it.

4.2 Handling the new ADD1 instruction

The new ADD1 instruction is now added to the ARM ISA. But here is no functional unit in the CPU to handle this new instruction. The OpClass parameter tells which functional unit will execute a given instruction. For the original ADD instruction, IntAlu executes it. For our new ADD1 instruction, we mentioned above that the CusAlu functional unit will implement it. To do it we need to add the functional units in the CPU models in gem5-X.

1.
```
cd ../../../../cpu/ (full path gem5/src/cpu)
vim FuncUnit.py
```

2. We need to add the CusAlu unit under (Please refer to attached FuncUnit.py file here under the *modified_files* folder)

```
class OpClass(Enum):
  vals class OpClass(Enum):
  vals = [
```

3. Open the file op_class.hh and add the following to the end

```
static const OpClass CusAluOp = Enums::CusAlu;
```

4. Add the functional unit to the Minor CPU model.

```
cd minor/
vim MinorCPU.py
```

See the attached MinorCPU.py file for details

5. Add the functional unit to O3 CPU model

```
cd ../o3
```

Edit the following files. (See the attached files here under the *modified_files* folder for details)

- FUPool.py
- FuncUnitConfig.py

6. Add the stats information in simple cpu

```
cd ../simple/
```

Edit the following files. (Attached here under the *modified_files* folder)

- base.cc
- exec_context.hh

7.
```
cd ../../proto/      (Full path gem5/src/proto)
```

8. Edit the file inst.proto and add following to the enum InstType

```
CusAlu = 34;
```

9. You can now compile gem5 using

```
scons build/ARM/gem5.fast
```

4.3 Testing that the ISA extension works

To test that the ARM64 ISA extension works, we have also written a test program in C++ using inline assembly to call the new instruction and test it. The test program is also attached here under the *add1_test* folder. (The program adds a constant 3 to the value being passed)

1. Run Gem5 in SE (System Emulation) mode using the test program. The output should look like this:

```
C[0] = 13
C[1] = 23
C[2] = 33
C[3] = 43
C[4] = 53
C[5] = 63
C[6] = 73
C[7] = 83
C[8] = 93
C[9] = 103
```

The program should also work perfectly fine in full-system mode as we are using in-line assembly to call the newly added instruction.

5 High Bandwidth Memory v2 (HBM2)

High Bandwidth Memeory (HBM) is based on 3D stacked DRAM banks made possible due to Through Silicon Vias (TSVs) achieving a high bandwidth of up to 307.2 GB/s. To implement the functional behavior of the HBM2 memory model in gem5-X, we extend the DRAM controller model of gem5 according to the architectural details of HBM2. To have 8-channels with memory interleaving, we initialized 8 DRAM controllers, each 128 bits wide. We connect all 8 DRAM controllers to a 1024-bit wide system bus, that connects to the cache hierarchy.

To use 8-channel HBM2 in gem5-X full system simulation, with appropriate bus widths throughout the system all the way to the caches, use the following command:

```
cd <path_to_gem5-X>/

./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<full_path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_1cpu.dtb \
-n 1 \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--l2bus-width=128 \
--membus-width=128 \
--mem-type=HBM2_2000_4H_1x128 \
--mem-ranks=1 \
--mem-channels=8 \
--mem-size=4GB \
--sys-clock=1600MHz \
```

No separate software support is required to use HBM2 in FS mode, and hence we are able to boot the Ubuntu Linux distribution using HBM2.

The HBM2 memory model can be found in the following file

```
<full_path_to_gem5-X>/src/mem/DRAMCtrl.py
```

6 Core Clustering

Core clustering enables group of compute cores to have their own shared cache, which can be last level cache (LLC), separate from other cores in the system. This reduces the shared resources between different compute clusters in the system to just cross bar interconnect and memory. In addition, clustering is also used when different type of cores are used in system. Same core types are clustered together with their own LLCs. This enables to have a heterogeneous system.

Cluster in now supported in gem5-X. To have different core clusters in gem5-X, use the following command:

```
./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_<NUM_CORES>cpu.dtb \
-n <NUM_OF_CORES> \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--l2_cluster_size=<NUM_OF_CORE_PER_CLUSTER> \
--mem-type=DDR4_2400_4x16 \
--mem-channels=4 \
--mem-ranks=4 \
--mem-size=4GB \
--sys-clock=1600MHz
```

This command will simulate a system with core clusters. Each cluster will have number of cores defined in *–l2_cluster_size* parameter. The number of cores defined by *-n* parameter should be divisible by the *–l2_cluster_size*. Dividing *n* by *l2_cluster_size*, gives the number of clusters in the system. Each cluster will have its own L2 (LLC) cache.

7 Heterogeneous Cores

Heterogeneity enables different workloads with vartying performance and energy constraints to be allocated to different core types in the system. Gem5-X supports both in-order and OoO cores in the same system. Different core types are distributed into different clusters.

To use heterogeneity in gem5-X, first the system is launched to boot up the linux to reach the region-of-interest (ROI), with the following command:

```
./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_<NUM_CORES>cpu.dtb \
-n <NUM_OF_CORES> \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--l2_cluster_size=<NUM_OF_CORE_PER_CLUSTER> \
--cluster_size_1=4 \
--mem-type=DDR4_2400_4x16 \
--mem-channels=4 \
--mem-ranks=4 \
--mem-size=4GB \
--sys-clock=1600MHz
```

This command will simulate a system with core clusters. The parameter *–cluster_size_1* defines the size of the 1st cluster of type 1. This should be the same as *–l2_cluster_size*. All the cores in the remaining clusters will be of type 2. For instance, if number of cores is defined to be 16, and both *–l2_cluster_size* and *–cluster_size_1* are set to 4, this implies to have 4 clusters in the system, each with 4 cores. The first cluster will have cores of type 1 and the remaining three clusters will have cores of types 2.

Once the ROI is reached, take a checkpoint using ”m5 checkpoint” command. Then one can resume from the checkpoint with the desired core types for each cluster. For the above code type 1 cores are set to be in-order and type 2 to be OoO, as in the following command:

```
./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_<NUM_CORES>cpu.dtb \
```

```
 9 -n <NUM_OF_CORES> \
10 --disk-image=gem5_ubuntu16.img  \
11 --caches  \
12 --l2cache  \
13 --l1i_size=32kB \
14 --l1d_size=32kB  \
15 --l2_size=1MB  \
16 --l2_assoc=2  \
17 --l2_cluster_size=<NUM_OF_CORE_PER_CLUSTER> \
18 --cluster_size_1=4 \
19 --mem-type=DDR4_2400_4x16 \
20 --mem-channels=4 \
21 --mem-ranks=4 \
22 --mem-size=4GB \
23 --sys-clock=1600MHz \
24 -r 1 \
25 --cpu-type=MinorCPU \
26 --cpu-type_2=DerivO3CPU \
```

8 Scratchpad Memory (SPM)

Scratchpad Memories (SPMs) are software programmable memories at the same level as L1 cache, but controlled by the user. Gem5-X supports SPMs, which are both local and shared between two consecutive cores.

To use SPMs gem5-X, the following command can be used:

```
./build/ARM/gem5.{fast, opt, debug} \
--remote-gdb-port=0 \
-d /path/to/your/output/directory \
configs/example/fs.py \
--cpu-clock=1GHz \
--kernel=vmlinux \
--machine-type=VExpress_GEM5_V1 \
--dtb-file=<path_to_gem5-X>/system/arm/dt/armv8_gem5_v1_<NUM_CORES>cpu.dtb \
-n <NUM_OF_CORES> \
--disk-image=gem5_ubuntu16.img \
--caches \
--l2cache \
--l1i_size=32kB \
--l1d_size=32kB \
--l2_size=1MB \
--l2_assoc=2 \
--mem-type=DDR4_2400_4x16 \
--mem-ranks=4 \
--mem-size=4GB \
--sys-clock=1600MHz \
--spm \
--d_spm_size=128kB
```

The *–spm* commands enabes SPM in gem5-Xand *–d_spm_size* defines the SPM size, which is set to 128KB in the above example. The SPMs can be accessed by two consecutive cores. For instance, SPM0 is accessible by core0 and core1, SPM1 by core1 and core2, SPM2 by core2 and core3 and so on.

Since this is a FS mode of gem5-X, to use SPM, they need to be mapped using mmapp, as in the following code:

```
void * spm_mem_alloc (uint64_t mem_size, uint64_t mem_address)
{

        uint64_t alloc_mem_size, page_mask, page_size;
        void *mem_pointer;
        void *virt_addr;

        page_size = sysconf(_SC_PAGESIZE);
        alloc_mem_size = (((mem_size / page_size) + 1) * page_size);
        page_mask = (page_size - 1);

        int mem_dev = open("/dev/mem", O_RDWR | O_SYNC);
        if(mem_dev == -1)
```

```
14          {
15                  perror("Cannot open /dev/mem \n");
16                  //return -1;
17
18          }
19
20          mem_pointer = mmap(NULL,
21                          alloc_mem_size,
22                          PROT_READ | PROT_WRITE,
23                          MAP_SHARED,
24                          mem_dev,
25                          (mem_address & ~page_mask)
26          );
27
28          if(mem_pointer == MAP_FAILED)
29          {
30              perror("Cannot MAP \n");
31              //return -1;
32          }
33
34          printf("Memory Mapped \n");
35          virt_addr = (mem_pointer + (mem_address & page_mask));
36
37
38      return virt_addr;
39  }
```

The above core snippet returns a virtual pointer in SPM in FS mode. The parameter *uint64_t mem_size* is used to define the size of memory allocated within SPM. The parameter *uint64_t mem_address* defines the memory address of the SPM in physical memory space. So for SPM0 this should be at an offset after the main memory and I/O devices in gem5-X. So for instance of the main memory size is 4GB, the offset for SPM0 should be 4GB+2GB(I/O devices memory space), i.e. 6GB=6442450944. SPM1 should be at an offset defined by main-memory size + I/O devices + SPM0_size.