

**ĐẠI HỌC BÁCH KHOA HÀ NỘI  
TRƯỜNG ĐIỆN – ĐIỆN TỬ**

\*\*\*\*\*



**BÁO CÁO ĐỒ ÁN III**

**Đề tài : Tìm hiểu các hệ thống mô phỏng (simulators)  
cho các hệ thống máy tính**

Giảng viên hướng dẫn: TS Nguyễn Thanh Bình

Mã lớp : 736242

Sinh viên thực hiện: Lê Anh Đức

MSSV: 20192773

Hà Nội, tháng 1 năm 2024

# MỤC LỤC

CHƯƠNG 1 . GIỚI THIỆU .....	6
1.1 Định nghĩa.....	6
1.2 Hệ thống mô phỏng.....	7
1.3 Mục tiêu đặt ra .....	10
CHƯƠNG 2 . PHẦN MỀM MULTI2SIM .....	11
2.1 Giới thiệu về Multi2sim.....	11
2.2 Các nội dung chính của phần mềm Multi2sim .....	11
2.3 Cài đặt Multi2sim trên hệ điều hành Ubuntu.....	12
2.4 Chạy multi2sim bằng docker .....	13
2.4.1 Docker.....	13
2.4.2 Multi2sim by Docker .....	14
2.5 Mô phỏng hệ thống GPU-CPU không đồng nhất trên Multi2sim .....	15
2.5.1 Hệ thống GPU-CPU không đồng nhất.....	15
2.5.2 Phương pháp mô phỏng .....	17
2.5.3 Bản thực thi đầu tiên .....	20
CHƯƠNG 3 . PHẦN MỀM GEM5 .....	25
3.1 Giới thiệu .....	25
3.2 Cài đặt Gem5 trên hệ điều hành Ubuntu.....	26
3.3 Các kiểu Binary và các tệp mô hình trong gem5 .....	27
3.3.1 Các kiểu binary .....	27
3.3.2 Các tệp mô hình .....	28
3.4 Xây dựng tập lệnh cấu hình đơn giản .....	29
4.4 Thông tin của các bản mô phỏng .....	35
CHƯƠNG 4 . ỨNG DỤNG VÀ THỰC NGHIỆM.....	41
4.1 Mô phỏng Multi2Sim.....	41
4.2 Mô phỏng Gem5 .....	50
CHƯƠNG 5 . KẾT LUẬN .....	50
CHƯƠNG 6 . TÀI LIỆU THAM KHẢO .....	50

## DANH MỤC HÌNH ẢNH

Hình 1-1 Mô hình mô phỏng .....	7
Hình 1-2 Sơ đồ phương pháp mô phỏng.....	8
Hình 1-3 Verification & Validation .....	9
Hình 2-1 Multi2Sim.....	11
Hình 2-2 Docker .....	13
Hình 2-3 Khởi chạy Multi2Sim trên bash của Docker .....	15
Hình 2-4 m2s test-args .....	21
Hình 2-5 m2s --x86-debug-isa isa test-args .....	22
Hình 2-6 File isa.....	23
Hình 2-7 File syscall .....	24
Hình 3-1 Gem5 .....	25
Hình 3-2 Cấu trúc file gem5 .....	27
Hình 3-3 Build gem5 bằng scons.....	27
Hình 3-4 Các tệp mô hình.....	28
Hình 3-5 Hệ thống mô phỏng đơn giản .....	33
Hình 3-6 Build bản mô phỏng gem5 đơn giản.....	35
Hình 3-7 Config.ini 1 .....	36
Hình 3-8 Config.ini 2 .....	37
Hình 3-9 Config.json 1 .....	38
Hình 3-10 Config.json 2 .....	39
Hình 3-11 Stats.txt .....	40
Hình 4-1 m2s test-threads 4 .....	43
Hình 4-2 m2s --x86-sim detailed test-threads 4.....	44
Hình 4-3 File report_1 .....	46
Hình 4-4 File report_2 .....	50

## DANH MỤC KÍ TỰ

CPU : Central Processing Unit

GPU : Graphics Processing Unit

ARM : Advanced RISC Machine

RCS : Run Control Scripts

FS : FileSystem

SE : System Emulator

DANH MỤC BẢNG BIỂU

# CHƯƠNG 1 . GIỚI THIỆU

## 1.1 Định nghĩa

Mô phỏng (Simulation) là sự bắt chước hoạt động của một quy trình hoặc hệ thống trong thế giới thực theo thời gian. Nó là một mô hình hoạt hình bắt chước hoạt động của một hệ thống hiện có hoặc được đề xuất.

Mô phỏng là một mô hình máy tính trong đó các thử nghiệm có thể được tiến hành, tạo ra mức độ hoàn thiện cao hơn so với thử nghiệm thông thường. Các mô hình mô phỏng có thể hỗ trợ cả các thí nghiệm phức tạp và đơn giản, đồng thời chúng có thể được sử dụng với hầu hết mọi quy trình xã hội.

Mô phỏng hệ thống là hoạt động của một mô hình theo thời gian hoặc không gian, giúp phân tích hiệu suất của hệ thống hiện có hoặc hệ thống được đề xuất. Nói cách khác, mô phỏng là quá trình sử dụng mô hình để nghiên cứu hiệu suất của hệ thống. Đó là hành động sử dụng mô hình để mô phỏng.

Định nghĩa : Mô phỏng có thể được định nghĩa rộng rãi là một kỹ thuật nghiên cứu các hệ thống động lực trong thế giới thực bằng cách bắt chước hành vi của chúng bằng mô hình toán học của hệ thống được triển khai trên máy tính kỹ thuật số.

Sơ lược lịch sử :

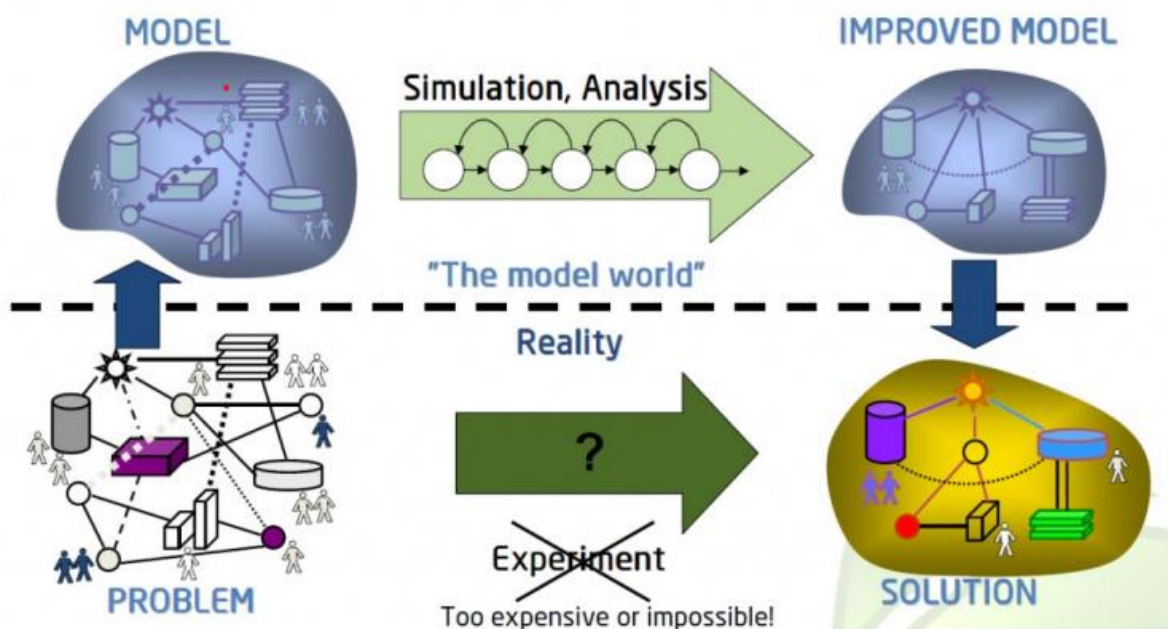
- 1940: Một phương pháp có tên ‘Monte Carlo’ được phát triển bởi các nhà nghiên cứu (John von Neumann, Stanislaw Ulan, Edward Teller, Herman Kahn) và các nhà vật lý làm việc trong dự án Manhattan để nghiên cứu sự tán xạ neutron.
- 1960: Ngôn ngữ mô phỏng có mục đích đặc biệt đầu tiên được phát triển, chẳng hạn như SIMSCRIPT của Harry Markowitz tại RAND Corporation.
- 1970: Trong thời kỳ này, nghiên cứu về toán học được bắt đầu cơ sở của mô phỏng.
- 1980: Trong thời kỳ này, phần mềm mô phỏng dựa trên PC, giao diện người dùng đồ họa và lập trình hướng đối tượng đã được phát triển.
- 1990: Trong giai đoạn này, mô phỏng dựa trên web, đồ họa hoạt hình lạ mắt, tối ưu hóa dựa trên mô phỏng, phương pháp Monte Carlo chuỗi Markov đã được phát triển.

Mô phỏng trong sản xuất đề cập đến một tập hợp rộng lớn các ứng dụng dựa trên máy tính để bắt chước hành vi của các hệ thống sản xuất.

Mô phỏng nhằm mục đích nghiên cứu mô hình của hệ thống thế giới thực này bằng cách đánh giá bằng số bằng phần mềm. Mô phỏng được thực hiện để đánh giá hiệu suất của một hệ thống, sản phẩm hoặc quy trình trước khi nó được xây dựng hoặc triển khai về mặt vật lý.

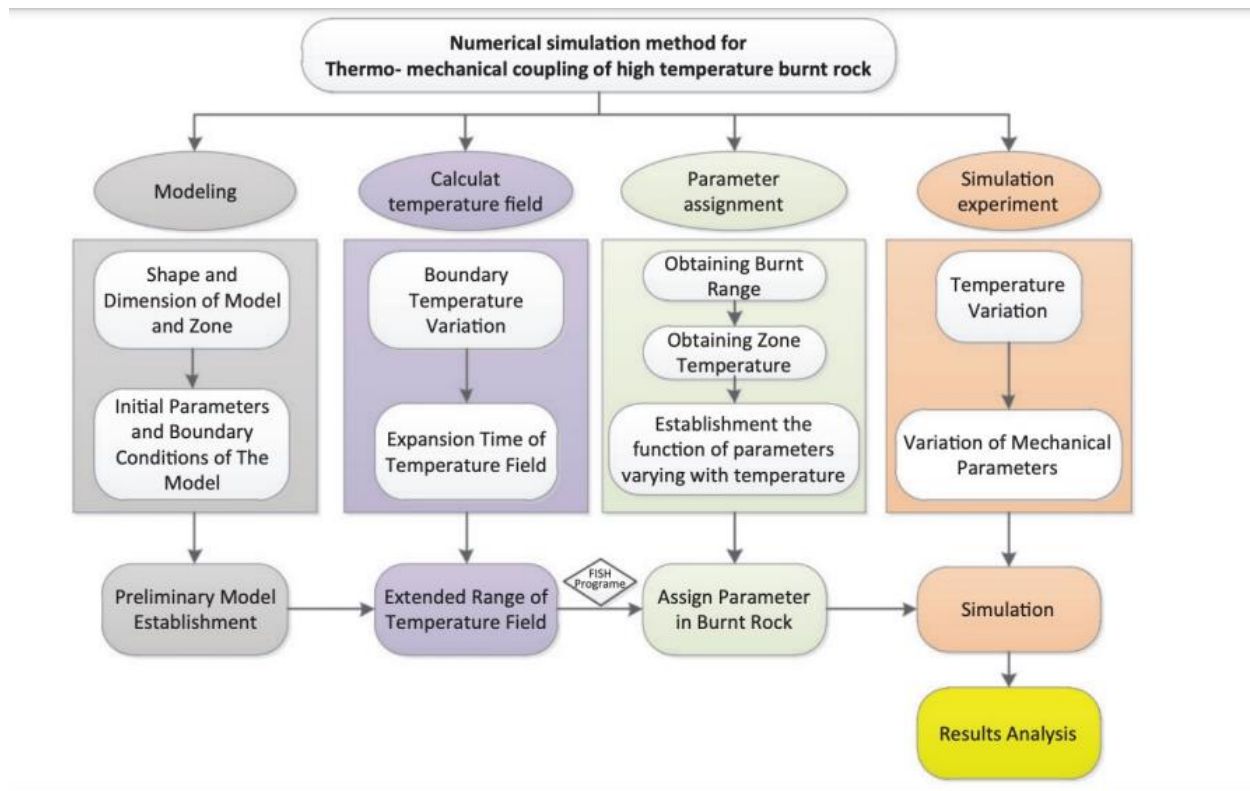
## 1.2 Hệ thống mô phỏng

Mô hình mô phỏng (Simulation modeling) được sử dụng để giúp các nhà thiết kế và kỹ sư hiểu được liệu trong những điều kiện nào và theo cách nào một bộ phận có thể bị hỏng cũng như tải trọng mà nó có thể chịu được. Mô hình mô phỏng là một ứng dụng kỹ thuật trong đó chúng ta hầu như có thể thấy bất kỳ hệ thống nào ở dạng toán học có thể được chuyển đổi thành mô hình. Nó là quá trình tạo và phân tích nguyên mẫu kỹ thuật số của mô hình vật lý để dự đoán hiệu suất của nó trong thế giới thực.



Hình 1-1 Mô hình mô phỏng

Các phương pháp mô phỏng (Simulation method) liên quan đến việc tạo ra các mô hình, tìm hiểu hành vi của các mô hình bằng thử nghiệm và đánh giá mức độ mà hành vi của các mô hình cung cấp một giải thích hợp lý về hành vi của các hệ thống 'tự nhiên' được quan sát.



Hình 1-2 Sơ đồ phương pháp mô phỏng

Các phương pháp mô phỏng dường như vừa tự nhiên vừa cần thiết để xác định xem số liệu thống kê kiểm tra nghiên cứu sự kiện có được xác định rõ hay không. Mô phỏng làm cho mô hình trở nên sống động và cho thấy một đối tượng hoặc hiện tượng cụ thể sẽ hoạt động như thế nào.

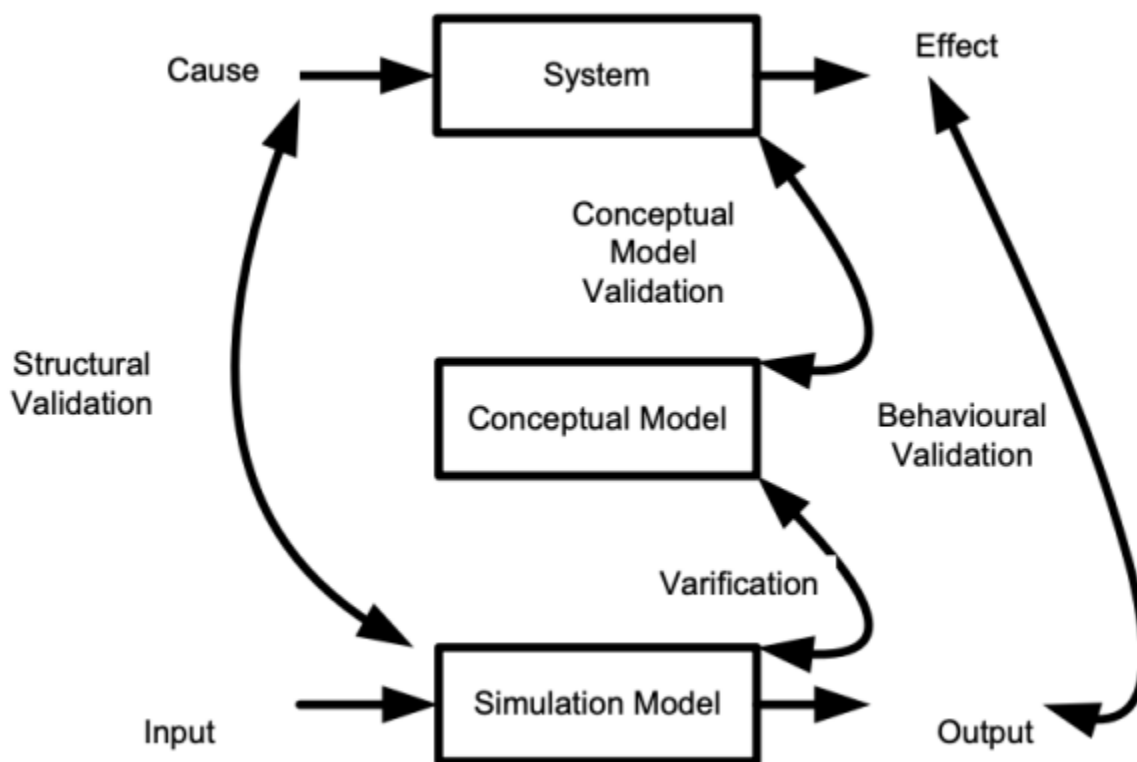
#### Phát triển mô hình mô phỏng

- Bước 1: Xác định vấn đề của hệ thống hiện có hoặc đặt ra các yêu cầu của hệ thống đề xuất.
- Bước 2: Thiết kế vấn đề trong khi quan tâm đến các yếu tố và hạn chế của hệ thống hiện có.
- Bước 3 : Thu thập và bắt đầu xử lý dữ liệu hệ thống, quan sát hiệu suất và kết quả của nó.
- Bước 4 : Phát triển mô hình bằng sơ đồ mạng và xác minh mô hình bằng các kỹ thuật xác minh khác nhau.
- Bước 5: Xác thực mô hình bằng cách so sánh hiệu suất của nó trong các điều kiện khác nhau với hệ thống thực.



- Bước 6 : Tạo tài liệu về mô hình để sử dụng trong tương lai, bao gồm các mục tiêu, giả định, biến đầu vào và hiệu suất một cách chi tiết.
- Bước 7: Lựa chọn thiết kế thí nghiệm phù hợp theo yêu cầu.
- Bước 8: Đưa ra điều kiện thí nghiệm trên mô hình và quan sát kết quả.

Xác thực (Validation) là quá trình so sánh hai kết quả. Trong quá trình này, chúng ta cần so sánh cách biểu diễn của mô hình khái niệm với hệ thống thực. Nếu so sánh là đúng thì nó hợp lệ, nếu không thì không hợp lệ.



*Hình 1-3 Verification & Validation*

Xác minh (Verification) là quá trình so sánh hai hoặc nhiều kết quả để đảm bảo tính chính xác của nó. Trong quá trình này, chúng tôi phải so sánh việc triển khai mô hình và dữ liệu liên quan của nó với mô tả khái niệm và thông số kỹ thuật của nhà phát triển.

Phân loại mô hình mô phỏng : Mô phỏng tĩnh (Static simulations) và mô phỏng động (Dynamic simulations).

- Mô hình mô phỏng tĩnh được giải thích bằng một ví dụ về khái niệm quy luật cung-cầu và cách nó được mô phỏng theo quan điểm của các mô hình thị trường tuyến tính và phi tuyến tính. Các mô hình mô phỏng chỉ biểu diễn hệ thống tại một thời điểm cụ thể được gọi là tĩnh. Loại mô phỏng này thường được gọi là mô phỏng Monte Carlo. Mô phỏng tĩnh là mô hình mô phỏng không có lịch sử bên trong của cả giá trị đầu ra và đầu vào đã được áp dụng trước đó. Nó cũng đại diện cho một mô hình trong đó thời gian không phải là một yếu tố. Loại mô phỏng này thường có một số hàm ( $f$ ) được tạo từ đầu vào ( $u$ ). Mỗi đầu ra trong loại mô phỏng này phụ thuộc vào giá trị của hàm ( $f$ ) và đầu vào ( $u$ ).
- Các mô hình mô phỏng động thể hiện các hệ thống khi chúng phát triển theo thời gian. Mô phỏng cửa hàng bánh rán trong giờ làm việc là một ví dụ về mô hình động. Mô hình mô phỏng động được giải thích bằng một ví dụ về mô hình bánh xe ô tô trong đó dao động của bánh xe được coi là động với các hệ số giảm chấn khác nhau và sau đó, sự khác biệt giữa các mô hình này sẽ được thể hiện bằng các biểu đồ và kết quả có liên quan. Trong mô phỏng động, một chương trình máy tính được sử dụng để xác định hành vi khác nhau của hệ thống tại các thời điểm khác nhau hoặc trong các tình huống khác nhau. Kiểu mô phỏng này duy trì bộ nhớ trong bao gồm các đầu vào, biến bên trong và đầu ra trước đó. Cũng nên nhớ rằng tất cả các biến trong mô phỏng động đều được biểu thị dưới dạng hàm của thời gian.

Mô phỏng động có thể được phân loại thành rời rạc hoặc liên tục.

- Các mô hình mô phỏng rời rạc sao cho các biến quan tâm chỉ thay đổi tại một tập hợp các điểm rời rạc theo thời gian.
- Mô hình mô phỏng liên tục sao cho các biến quan tâm thay đổi liên tục theo thời gian.

### 1.3 Mục tiêu đặt ra

Mục tiêu đặt ra là tìm hiểu hai phần mềm mô phỏng hàng đầu : Multi2sim và Gem5

## CHƯƠNG 2 . PHẦN MỀM MULTI2SIM

### 2.1 Giới thiệu về Multi2sim

Các kiến trúc sư máy tính không ngừng tìm cách thiết kế Bộ xử lý trung tâm (CPU) và Bộ xử lý đồ họa (GPU) nhanh hơn, mạnh hơn và tiết kiệm điện hơn. Nhưng việc sản xuất một thiết kế mới là một quá trình lâu dài và tốn kém, phụ thuộc vào việc xác minh tính chính xác và khả thi trước đó.

Multi2Sim là trình mô phỏng CPU và GPU, được sử dụng để kiểm tra và xác nhận các thiết kế phần cứng mới trước khi chúng được sản xuất thực tế. Bằng cách chạy một bộ điểm chuẩn tiêu chuẩn trên Multi2Sim, kiến trúc sư máy tính có thể xác minh xem thiết kế thay thế được đề xuất có chính xác hay không và hiệu suất tương đối của nó so với các thiết kế hiện có là bao nhiêu.



*Hình 2-1 Multi2Sim*

Multi2Sim là một khung mô phỏng cho tính toán không đồng nhất CPU-GPU được viết bằng C. Nó bao gồm mô hình cho CPU siêu vô hướng, đa luồng và đa lõi, cũng như kiến trúc GPU.

### 2.2 Các nội dung chính của phần mềm Multi2sim

- The x86 CPU Model
- The MIPS-32 CPU Model
- The ARM CPU model
- OpenCL Execution
- The AMD Evergreen GPU Model
- The AMD Southern Islands GPU Model

- The NVIDIA Fermi GPU Model
- The Memory Hierarchy
- Interconnection Networks
- M2S-Visual: The Multi2Sim Visualization Tool
- M2S-Cluster: Launching Massive Simulations
- Multi2C: The Kernel Compiler
- 

## 2.3 Cài đặt Multi2sim trên hệ điều hành Ubuntu

Để cài đặt Multi2Sim, ta có thể tải xuống gói nguồn giả lập từ trang chủ tại [www.multi2sim.org](http://www.multi2sim.org). Gói này là một tệp tar nén, có thể được giải nén và biên dịch bằng cách sử dụng các lệnh sau, thay thế <version> bằng giá trị thích hợp, phiên bản mới nhất là 5-0:

```
$ tar -xzf multi2sim-<version>.tar.gz
$ cd multi2sim-<version>
$ ./configure
$ make
```

Nếu tất cả các phụ thuộc vào thư viện đều được thỏa mãn và quá trình biên dịch thành công, dòng lệnh Multi2Sim chính công cụ này được tìm thấy tại multi2sim-<version>/src/m2s.

Các lệnh cài đặt trình mô phỏng là

```
$ sudo make install
```

Multi2Sim có một bộ tùy chọn dòng lệnh phong phú cho phép mô phỏng bộ xử lý phức tạp thiết kế và cấu hình chương trình khách. Nhưng nó cũng cung cấp một giao diện dòng lệnh đơn giản để khởi chạy mô phỏng thử nghiệm ban đầu. Cú pháp dòng lệnh chung là

```
$ m2s [<options>] [<program> [<args>]]
```

- ❖ Chuỗi <options> đại diện cho một tập hợp (có thể trống) các tùy chọn dòng lệnh xác định loại mô phỏng sẽ được thực hiện, cũng như mô hình bộ xử lý sẽ được sử dụng. Tất cả các tùy chọn bắt đầu bằng một cú đúp dấu gạch ngang (—) và tùy ý nhận một hoặc nhiều đối số.

Một danh sách các tùy chọn dòng lệnh có thể được thu được bằng cách chạy lệnh

```
$ m2s --help
```

Sau khi tùy chọn dòng lệnh cuối cùng được phân tích cú pháp, m2s có thể nhận thêm các chuỗi khác. Chuỗi đầu tiên theo sau tùy chọn cuối cùng được hiểu là chương trình

khách thực thi để chạy trên Multi2Sim. Tất cả còn lại chuỗi được hiểu là đối số cho chương trình khách.

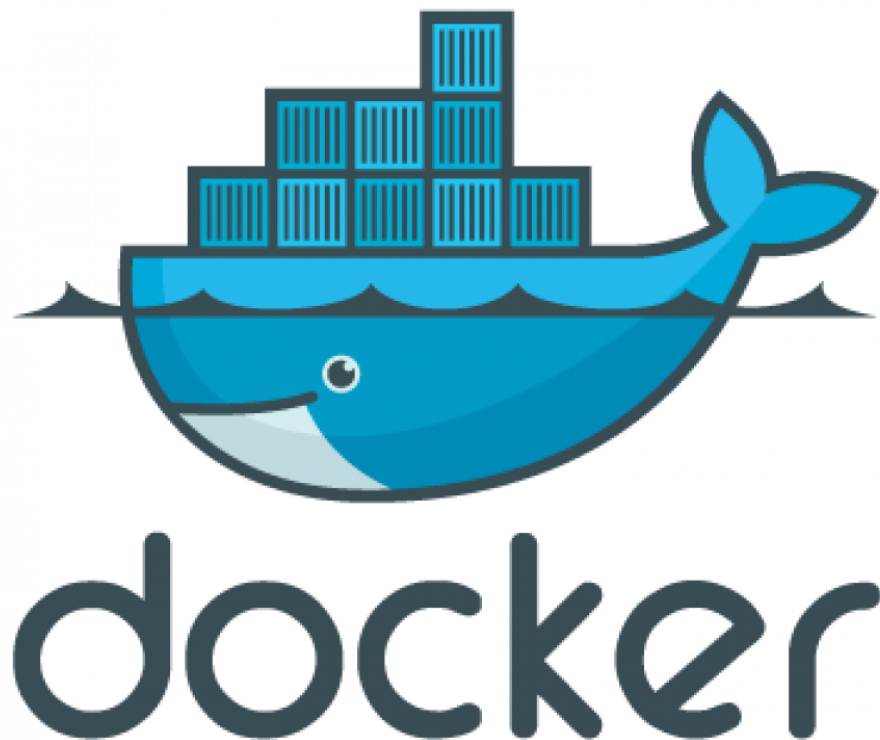
## 2.4 Chạy multi2sim bằng docker

### 2.4.1 Docker

Docker là một nền tảng mở để phát triển, vận chuyển và chạy ứng dụng.

Docker cho phép chúng ta tách ứng dụng của mình khỏi cơ sở hạ tầng để bạn có thể triển khai phần mềm một cách nhanh chóng. Với Docker, bạn có thể quản lý cơ sở hạ tầng của mình theo cách bạn quản lý ứng dụng của mình.

Bằng cách tận dụng các phương pháp của Docker để vận chuyển, kiểm thử và triển khai mã một cách nhanh chóng, bạn có thể giảm đáng kể khoảng thời gian giữa việc viết mã và chạy nó trong môi trường sản xuất.



*Hình 2-2 Docker*

Docker không chỉ là một nền tảng để phát triển, vận chuyển và chạy ứng dụng mà còn là một công cụ mạnh mẽ giúp tạo và quản lý các môi trường containerized. Containerization là một phương pháp tiên tiến cho việc đóng gói và triển khai ứng dụng, giúp đơn giản hóa quá trình phát triển và chuyển giao phần mềm.

Các đặc điểm quan trọng của Docker bao gồm:

- **Đóng Gói Đồng Nhất:** Docker cho phép bạn đóng gói tất cả các phần của ứng dụng và các phụ thuộc của nó vào một container duy nhất. Điều này bao gồm cả mã nguồn, thư viện, môi trường thực thi và các cài đặt hệ thống khác.
- **Di Động và Linh Hoạt:** Container Docker có thể chạy trên bất kỳ máy chủ nào hỗ trợ Docker, giúp đơn giản hóa quá trình di chuyển ứng dụng giữa các môi trường phát triển, kiểm thử và sản xuất.
- **Tiết Kiệm Tài Nguyên:** Containers chia sẻ hạ tầng hệ điều hành với máy chủ chủ, giảm thiểu sự lãng phí tài nguyên so với máy ảo truyền thống.
- **Quản Lý Tự Động:** Docker cung cấp các công cụ quản lý như Docker Compose để tự động hóa quy trình triển khai, mở rộng và cập nhật ứng dụng trong môi trường containerized.
- **Tương Thích Đa Nền Tảng:** Docker hỗ trợ nhiều nền tảng, bao gồm Linux, Windows và macOS, làm cho việc phát triển và triển khai ứng dụng trở nên dễ dàng trên nhiều môi trường.

Docker không chỉ là một công cụ mà còn là một quy trình làm việc, một triết lý giúp đẩy nhanh quá trình phát triển phần mềm và tăng cường tính di động và đồng nhất của ứng dụng.

#### 2.4.2 Multi2sim by Docker

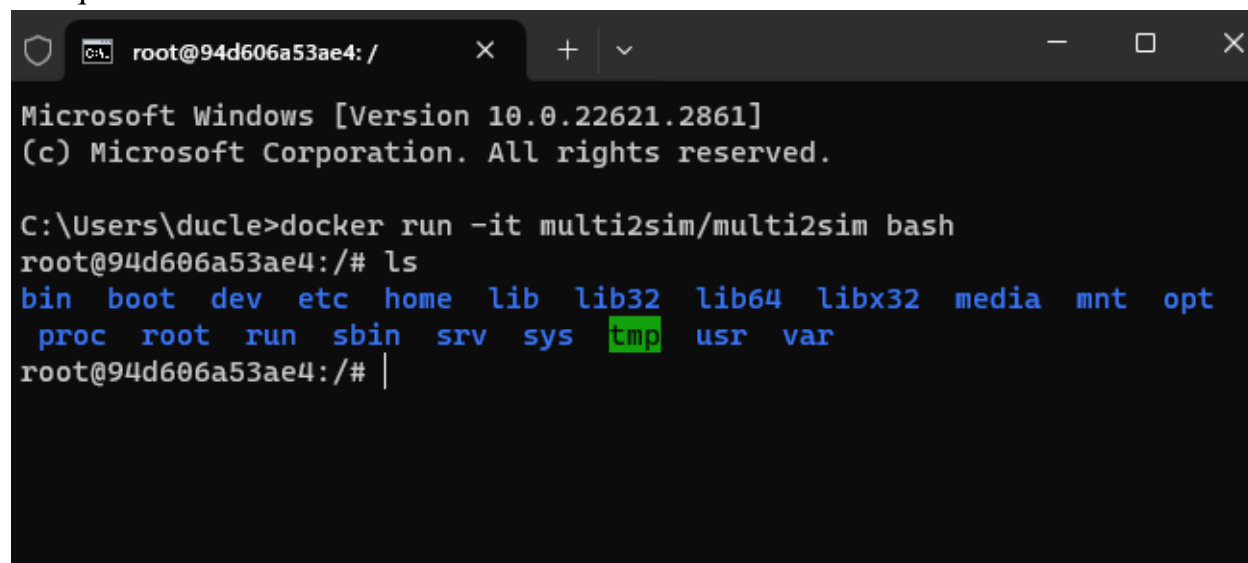
Sau khi cài docker trên máy tính , ta mở thiết bị đầu cuối (terminal) để lấy mã nguồn Multi2sim về bằng lệnh :

```
docker pull multi2sim/multi2sim
```

Sau đó khởi chạy container và chuyển sang bash của multi2sim bằng lệnh :

```
docker run -it multi2sim/multi2sim bash
```

Kết quả :



```
Microsoft Windows [Version 10.0.22621.2861]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ducle>docker run -it multi2sim/multi2sim bash
root@94d606a53ae4:/# ls
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt
proc  root  run  sbin  srv  sys  tmp  usr  var
root@94d606a53ae4:/# |
```

*Hình 2-3 Khởi chạy Multi2Sim trên bash của Docker*

## 2.5 Mô phỏng hệ thống GPU-CPU không đồng nhất trên Multi2sim

### 2.5.1 Hệ thống GPU-CPU không đồng nhất

Một hệ thống CPU-GPU không đồng nhất thường đề cập đến một môi trường tính toán kết hợp khả năng của cả bộ xử lý trung tâm (CPU) và bộ xử lý đồ họa (GPU). Mỗi loại bộ xử lý này được tối ưu hóa cho các nhiệm vụ khác nhau, và khi kết hợp, chúng có thể cung cấp hiệu suất cao hơn cho một loạt ứng dụng khác nhau.

Một số tính chất quan trọng của hệ thống CPU-GPU không đồng nhất :

a, Song Song Công Việc vs. Song Song Dữ Liệu:

- Song Song Công Việc (CPU): CPU thích hợp cho các nhiệm vụ đòi hỏi quyết định phức tạp, nhánh và thực thi tuần tự. Chúng xuất sắc trong việc xử lý nhiều nhiệm vụ cùng một lúc và thường xử lý tính toán chung.
- Song Song Dữ Liệu (GPU): GPU được tối ưu hóa cho xử lý đồng thời trên tập dữ liệu lớn. Chúng bao gồm nhiều lõi có thể thực hiện cùng một phép toán trên các phần khác nhau của dữ liệu đồng thời. Điều này giúp GPU hiệu quả cho các nhiệm vụ

vụ có thể được thực hiện song song như làm đồ họa, mô phỏng khoa học và học máy.

**b, Vai Trò Bổ Sung:**

- CPU: Quản lý toàn bộ hệ thống, thực thi nhiệm vụ tuần tự, xử lý các thao tác vào/ra và quản lý luồng điều khiển phức tạp.
- GPU: Tăng tốc nhiệm vụ có thể song song hóa bằng cách thực hiện tính toán trên nhiều yếu tố dữ liệu cùng một lúc, phù hợp cho các nhiệm vụ như xử lý hình ảnh, mô phỏng khoa học và học máy.

**c, Mô Hình Lập Trình:**

- Mô Hình Lập Trình CPU: Thường theo mô hình lập trình thông thường (ví dụ: C, C++, Java).
- Mô Hình Lập Trình GPU: Thường sử dụng ngôn ngữ hoặc giao diện lập trình đặc biệt, chẳng hạn như CUDA (Compute Unified Device Architecture) cho GPU của NVIDIA hoặc OpenCL (Open Computing Language) để hỗ trợ nhiều nền tảng.

**d, Cấu Trúc Bộ Nhớ:**

- Cấu Trúc Bộ Nhớ CPU: Thường có cấu trúc bộ nhớ phân cấp với bộ nhớ đệm L1, L2, và đôi khi là L3, cùng với quyền truy cập vào bộ nhớ chính.
- Cấu Trúc Bộ Nhớ GPU: Ngoài các bộ nhớ đệm riêng, GPU thường có bộ nhớ toàn cục và bộ nhớ chia sẻ được tối ưu hóa cho mô hình truy cập song song.

**e, Nền Tảng Tính Toán Khác Nhau:**

- APUs (Accelerated Processing Units): Kết hợp CPU và GPU trên cùng một vi chip, cho phép tích hợp chặt chẽ và có thể cải thiện truyền dữ liệu giữa chúng.
- GPU Tích Hợp: Một số CPU đi kèm với GPU tích hợp trên cùng một chip, chia sẻ bộ nhớ hệ thống.

**f, Truyền Dữ Liệu và Giao Tiếp:**

- Giao tiếp hiệu quả giữa CPU và GPU là quan trọng để đạt được hiệu suất cao.
- Việc truyền dữ liệu giữa bộ nhớ CPU và GPU có thể tạo ra chi phí và việc giảm thiểu chi phí này là quan trọng để tối ưu hiệu suất.

**g, Chuyển Giao Nhiệm Vụ:**

- Chuyển Giao Nhiệm Vụ CPU: Một số nhiệm vụ có thể được chuyển giao từ CPU sang GPU để thực hiện song song, cải thiện hiệu suất tổng thể của hệ thống.
- Cân Bằng Tải: Cân bằng công việc giữa CPU và GPU là quan trọng để tối đa hóa sự sử dụng và tránh tình trạng chậm trễ.

**h, Hiệu Suất Năng Lượng:**

- Hệ thống không đồng nhất nhằm cân bằng hiệu suất và hiệu quả năng lượng, tận dụng những ưu điểm của mỗi bộ xử lý cho các nhiệm vụ phù hợp.



#### ❖ Các Trường Hợp Sử Dụng:

Tính toán không đồng nhất đặc biệt hữu ích cho các ứng dụng có sự kết hợp của công việc tuần tự và công việc có thể song song hóa, như mô phỏng khoa học, biên tập video và học máy.

#### ❖ Thách Thức:

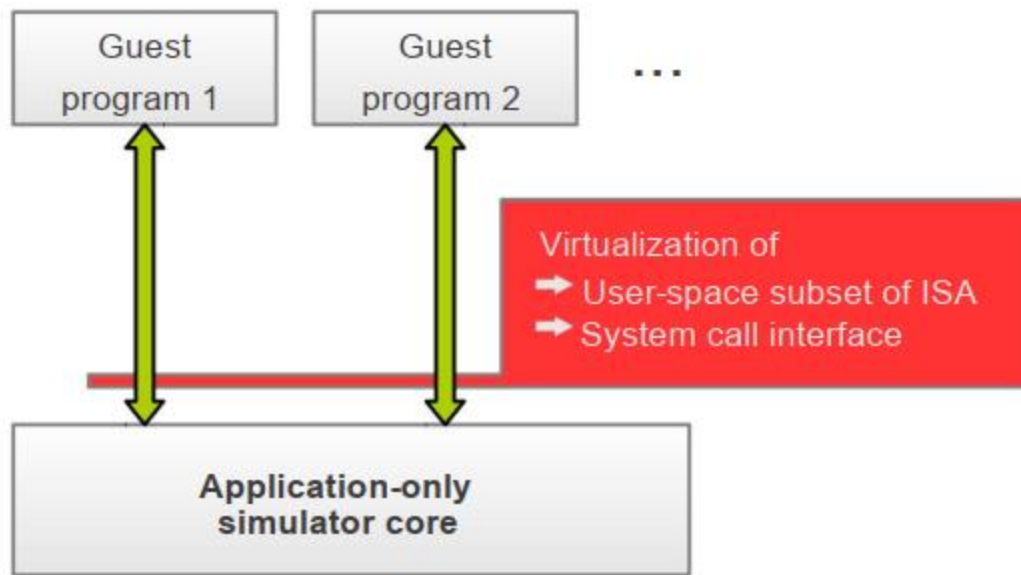
- Quản lý sự phức tạp của việc lập trình cho hệ thống không đồng nhất.
- Xử lý dữ liệu chuyển đổi giữa CPU và GPU một cách hiệu quả.
- Đảm bảo cân bằng công việc và tránh tình trạng chiếm trọn một bộ xử lý.

Hiểu rõ các khía cạnh lý thuyết của hệ thống CPU-GPU không đồng nhất là quan trọng để thiết kế và tối ưu hóa hiệu suất ứng dụng một cách hiệu quả. Điều này đòi hỏi xem xét cẩn thận về đặc tính của công việc, kiến trúc của CPU và GPU, và các cơ chế giao tiếp giữa chúng.

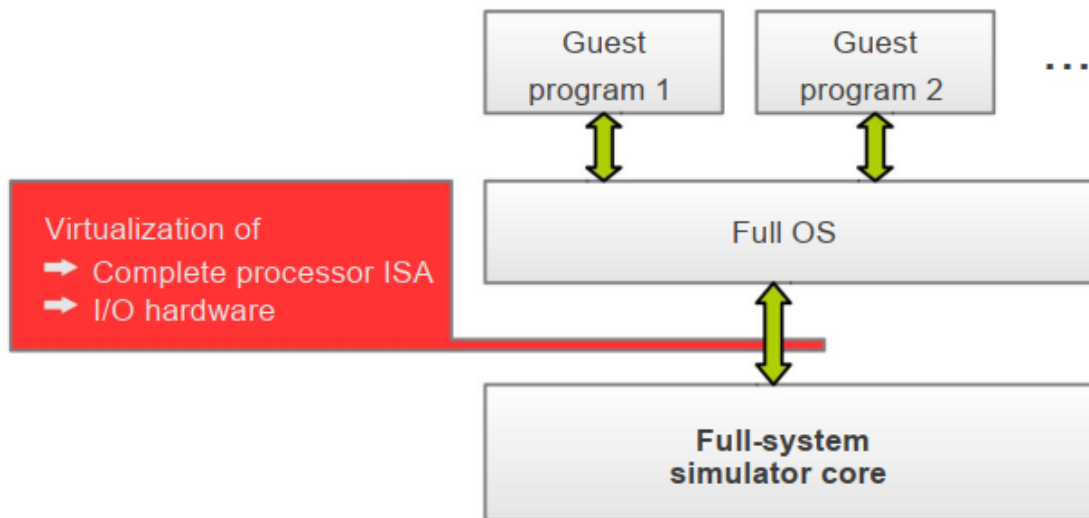
### 2.5.2 Phương pháp mô phỏng

Có 2 loại phương pháp mô phỏng gồm : Application-only(Mô phỏng chỉ ứng dụng) và Full System (Mô phỏng toàn hệ thống)

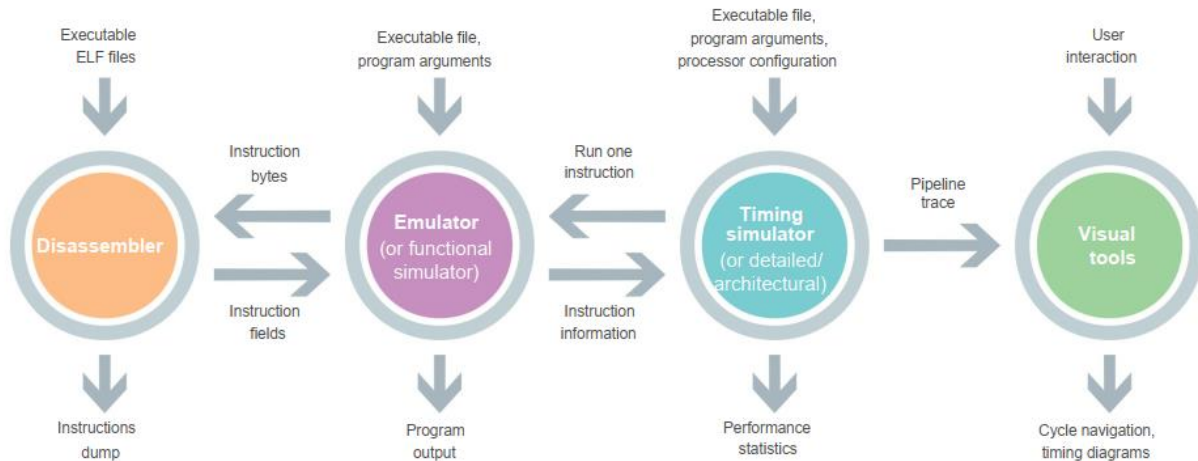
Application-Only Simulation là một phương pháp mô phỏng trong đó chỉ có ứng dụng hoặc tải công việc cụ thể được mô phỏng, mà không mô hình chi tiết về kiến trúc phần cứng hoặc các chi tiết của hệ thống. Nói một cách đơn giản, trọng tâm của nó là giả lập hành vi và hiệu suất của một ứng dụng cụ thể trong môi trường ảo mà không xem xét chi tiết phức tạp của toàn bộ hệ thống máy tính.



Full-system simulation : là một phương pháp mô phỏng mà trong đó toàn bộ hệ thống máy tính, bao gồm cả phần cứng và phần mềm, được mô phỏng và tái tạo trong một môi trường ảo. Điều này bao gồm cả kiến trúc CPU, bộ nhớ, tác vụ hệ điều hành, và các thành phần khác của hệ thống.



Quá trình mô phỏng gồm 4 giai đoạn :



## Triển khai cho cấu hình X86

### ❖ Disassembler :

- Triển khai một bộ giải mã hướng dẫn hiệu quả dựa trên các bảng tra cứu.
- Khi được sử dụng như một công cụ độc lập, kết quả được cung cấp với định dạng chính xác như trình dịch mã x86 của GNU để kiểm tra tự động

### ❖ Emulator

- Program Loading : Phân tích tập lệnh , khởi tạo ngăn xếp, thanh ghi
- Emulation Loop : Mô phỏng các lệnh X86 , các lệnh gọi hệ thống

### ❖ Timing simulator

- Superscalar Processor : Xử lý siêu vô hướng
- Multithreaded and Multicore Processors : Xử lý đa luồng và đa lõi
- Benchmark Support : Hỗ trợ điểm chuẩn SPEC , SPLASH-2 ...

### ❖ Visual tools

- Công cụ trực quan hóa sơ đồ

Hỗ trợ kiến trúc hiện đại :

	Disasm.	Emulation	Timing simulation, simulation	Graphics pipelines
ARM	X	X	-	-
MIPS	X	X	-	-
x86	X	X	X	X
AMD Evergreen	X	X	X	X
AMD Southern Islands	X	X	X	X
NVIDIA Fermi	X	X	In progress	-
NVIDIA Kepler	X	In progress	-	-

Hình 2-4 Hỗ trợ các kiến trúc

### 2.5.3 Bản thực thi đầu tiên

Tạo 1 chương trình cơ bản trong nguồn của dự án multi2sim

```

/tutorial/demo1/test-args.c

1  #include <stdio.h>
2
3  int main(int argc, char **argv)
4  {
5      int i;
6
7      printf("Number of arguments: %d\n", argc);
8      for (i = 0; i < argc; i++)
9          printf("\targv[%d] = %s\n", i, argv[i]);
10
11     return 0;
12 }
13
14
15
RAM 3.32 GB CPU 1.00% Signed in

```

“argc” : Biến này lưu trữ số lượng đối số được truyền vào chương trình, bao gồm cả tên chương trình itself.

“argv” : Một mảng con trỏ (pointer array) chứa các chuỗi đại diện cho các đối số dòng lệnh.

Khởi chạy file test-args.c để biên dịch nguồn (Compile C source) :

```
gcc test-args.c -o test-args
```

Khởi chạy test-args trên Multi2Sim :

```
m2s test-args  
m2s test-args 2>err
```

Lệnh “m2s test-args 2>err “: Lệnh này chạy chương trình test-args và mọi thông báo lỗi do chương trình tạo ra sẽ được chuyển hướng đến một tệp có tên err. 2> được sử dụng để chuyển hướng stderr.

```
# m2s test-args  
  
; Multi2Sim 5.0 - A Simulation Framework for CPU-GPU Heterogeneous Computing  
; Please use command 'm2s --help' for a list of command-line options.  
; Simulation alpha-numeric ID: hpp1m  
  
Number of arguments: 1  
    argv[0] = test-args  
  
;  
; Simulation Statistics Summary  
;  
  
[ General ]  
RealTime = 0.03 [s]  
SimEnd = ContextsFinished  
  
[ x86 ]  
RealTime = 0.02 [s]  
Instructions = 93736  
InstructionsPerSecond = 3811336  
  
# []  
  
RAM 3.28 GB CPU 0.00% Signed in
```

*Hình 2-5 m2s test-args*

[ General ]: Phần này cung cấp thông tin tổng quan về mô phỏng.

- RealTime : Thời gian thực hiện mô phỏng là 0.03 giây.

- KếtThúcMôPhỏng: Mô phỏng kết thúc khi tất cả các ngữ cảnh (contexts) hoàn thành.

[ x86 ]: Phần này chứa thông tin liên quan đến kiến trúc x86 (đối với CPU chạy mã lệnh x86).

- RealTime : Thời gian thực hiện trên kiến trúc x86 là 0.02 giây.
- Instructions : Tổng số lệnh thực thi là 93736.
- InstructionsPerSecond : Tốc độ thực thi lệnh trung bình là 3811336 lệnh mỗi giây.

Những con số này cung cấp cái nhìn tổng quan về thời gian và hiệu suất của mô phỏng

Tiếp theo , hiển thị dấu vết của hướng dẫn x86 và cập nhật cho các vị trí thanh ghi/bộ nhớ bằng lệnh :

```
m2s --x86-debug-isa isa test-args

# m2s --x86-debug-isa isa test-args

; Multi2Sim 5.0 - A Simulation Framework for CPU-GPU Heterogeneous Computing
; Please use command 'm2s --help' for a list of command-line options.
; Simulation alpha-numeric ID: IvnyQ


Number of arguments: 1
    argv[0] = test-args

;
; Simulation Statistics Summary
;

[ General ]
RealTime = 1.77 [s]
SimEnd = ContextsFinished

[ x86 ]
RealTime = 1.77 [s]
Instructions = 93736
InstructionsPerSecond = 53046

# []
```

RAM 3.39 GB CPU 0.00%  Signed in

Hình 2-6 m2s --x86-debug-isa isa test-args

Nó sẽ tạo ra 1 file isa :

```
docker exec -it 45dd3309dbf x + v
1000      0 10d0: mov     eax,esp (2 bytes)
1000      1 10d2: call   4790 (5 bytes)
1000      2 4790: push  ebp (1 bytes)
1000      3 4791: mov     ebp,esp (2 bytes)
1000      4 4793: push  edi (1 bytes)
1000      5 4794: push  esi (1 bytes)
1000      6 4795: push  ebx (1 bytes)
1000      7 4796: sub     esp,0x5c (3 bytes)
1000      8 4799: call   18768 (5 bytes)
1000      9 18768: mov     ebx,DWORD PTR [esp] (3 bytes) [0xbffff8c]=0x479e
1000     10 1876b: ret     (1 bytes)
1000     11 479e: add     ebx,0x1c862 (6 bytes)
1000     12 47a4: mov     DWORD PTR [ebp-0x44],eax (3 bytes) [0xbffffb4] <- 0xbffff000
1000     13 47a7: rdtsc   (2 bytes)
1000     14 47a9: mov     DWORD PTR [ebx-0x2f0],eax (6 bytes) [0x20d10] <- 0x64c48126
1000     15 47af: mov     eax,DWORD PTR [ebx-0xd0] (6 bytes) [0x20f30]=0xe
1000     16 47b5: mov     esi,0x6fffffff (5 bytes)
1000     17 47ba: mov     DWORD PTR [ebx-0x2ec],edx (6 bytes) [0x20d14] <- 0x25a
1000     18 47c0: lea     edx,[ebx-0xd0] (6 bytes)
1000     19 47c6: mov     edi,edx (2 bytes)
1000     20 47c8: sub     edi,DWORD PTR [ebx] (6 bytes) [0x21000]=0x20f30
1000     21 47ce: test    eax,eax (2 bytes)
1000     22 47d0: mov     DWORD PTR [ebx+0x564],edx (6 bytes) [0x21564] <- 0x20f30
1000     23 47d6: mov     DWORD PTR [ebx+0x55c],edi (6 bytes) [0x2155c] <- 0x0
1000     24 47dc: jne     47f9 (2 bytes)
1000     25 47f9: cmp     eax,0x21 (3 bytes)
1000     26 47fc: jbe     47e9 (2 bytes)
1000     27 47e9: mov     DWORD PTR [ebx+eax*4+0x57c],edx (7 bytes) [0x215b4] <- 0x20f30
1000     28 47f0: add     edx,0x8 (3 bytes)
1000     29 47f3: mov     eax,DWORD PTR [edx] (2 bytes) [0x20f38]=0x4
1000     30 47f5: test    eax,eax (2 bytes)
1000     31 47f7: je      4828 (2 bytes)
1000     32 47f9: cmp     eax,0x21 (3 bytes)
1000     33 47fc: jbe     47e9 (2 bytes)
```

Hình 2-7 File isa

Qua file isa , mã được biểu diễn với định dạng : Address Instruction (Size bytes) [Memory Access] <- Value.

Hiện thị dấu vết các cuộc gọi hệ thống Linux, với các thông số qua lệnh :

```
m2s --x86-debug-syscall syscall test-args
```

Tạo ra file syscall chứa thông tin :

```
docker exec -it 45dd3309dbf X + v
[x86 context 1000] System call 'brk' (code 45, inst 1062)
  newbrk = 0x0 (previous brk was 0x804b000)
  ret = (134524928, 0x804b000)
[x86 context 1000] System call 'newuname' (code 122, inst 2155)
  putsname=0xbffefcaa
  sysname='Linux', nodename='Multi2Sim'
  release='3.1.9-1.fc16.i686#1 Fri Jan 13 16:37:42 UTC 2012', version='i686'
  machine='', domainname=''
  ret = (0, 0x0)
[x86 context 1000] System call 'access' (code 33, inst 3113)
  file_name='/etc/ld.so.nohwcap', mode=0x0
  full_path='/etc/ld.so.nohwcap'
  mode={}
  ret = (-2, 0xffffffff), errno = ENOENT)
[x86 context 1000] System call 'mmap2' (code 192, inst 5002)
  addr=0x0, len=4096, prot=0x3, flags=0x22, guest_fd=-1, offset=0x0
  prot={PROT_READ|PROT_WRITE}, flags={MAP_PRIVATE|MAP_ANONYMOUS}
  ret = (-1208287232, 0xb7fb0000)
[x86 context 1000] System call 'access' (code 33, inst 5573)
  file_name='/etc/ld.so.preload', mode=0x4
  full_path='/etc/ld.so.preload'
  mode={R_OK}
  ret = (-2, 0xffffffff), errno = ENOENT)
[x86 context 1000] System call 'open' (code 5, inst 6347)
  filename='/etc/ld.so.cache' flags=0x80000, mode=0x0
  fullpath='/etc/ld.so.cache'
  flags={524288}
  File opened:
    guest_fd=3
    host_fd=4
  ret = (3, 0x3)
[x86 context 1000] System call 'fstat64' (code 197, inst 6369)
  fd=3, statbuf_ptr=0xbffef8f0
  host_fd=4
```

Hình 2-8 File syscall



## CHƯƠNG 3 . PHẦN MỀM GEM5

### 3.1 Giới thiệu

Gem5 là một nền tảng mô phỏng hệ thống máy tính điều khiển sự kiện rời rạc theo mô-đun. Đó có nghĩa là các thành phần của gem5 có thể được sắp xếp lại, tham số hóa, mở rộng hoặc thay thế dễ dàng để phù hợp với nhu cầu của bạn. Nó mô phỏng thời gian trôi qua như một chuỗi các sự kiện rời rạc. Mục đích sử dụng của nó là mô phỏng một hoặc nhiều hệ thống máy tính theo nhiều cách khác nhau.

Nó không chỉ là một trình mô phỏng; đó là một nền tảng mô phỏng cho phép bạn sử dụng bao nhiêu thành phần có sẵn của nó tùy thích để xây dựng hệ thống mô phỏng của riêng mình.



*Hình 3-1 Gem5*

Gem5 được viết chủ yếu bằng C++ và python và hầu hết các thành phần được cung cấp theo giấy phép kiểu BSD. Nó có thể mô phỏng một hệ thống hoàn chỉnh với các thiết bị và hệ điều hành ở chế độ toàn hệ thống (chế độ FS) hoặc các chương trình chỉ dành cho không gian người dùng trong đó các dịch vụ hệ thống được trình mô phỏng cung cấp trực tiếp ở chế độ mô phỏng tòa nhà (chế độ SE). Có nhiều mức hỗ trợ khác nhau để thực thi các tệp nhị phân Alpha, ARM, MIPS, Power, SPARC, RISC-V và 64 bit x86 trên các mẫu CPU, bao gồm hai mô hình CPI đơn giản, một mô hình không theo thứ tự và một mô hình theo thứ tự. Một hệ thống bộ nhớ có thể được xây dựng linh hoạt từ bộ đệm và thanh ngang hoặc trình mô phỏng Ruby để cung cấp mô hình hệ thống bộ nhớ linh hoạt hơn nữa.

### 3.2 Cài đặt Gem5 trên hệ điều hành Ubuntu

Các yêu cầu để có thể cài đặt gem 5:

- ❖ Git : Dự án gem5 sử dụng Git để kiểm soát phiên bản.  
Lệnh : `sudo apt install git`
- ❖ Gcc 7+  
Lệnh : `sudo apt install build-essential`
- ❖ SCons 3.0+ : gem5 sử dụng SCons làm môi trường xây dựng. SCons giống như tạo trên steroid và sử dụng tập lệnh Python cho tất cả các khía cạnh của quá trình xây dựng. Điều này cho phép một hệ thống xây dựng rất linh hoạt (nếu chậm).  
Lệnh : `sudo apt install scons`
- ❖ Python 3.6+  
Lệnh : `sudo apt install python3-dev`
- ❖ Protobuf 2.1+ : Bộ đệm giao thức là một cơ chế mở rộng trung lập về ngôn ngữ, nền tảng để tuần tự hóa dữ liệu có cấu trúc. Trong gem5, thư viện protobuf được sử dụng để tạo và phát lại dấu vết.  
Lệnh : `sudo apt install libprotobuf-dev protobuf-compiler libgoogle-perftools-dev`
- ❖ Boost : Thư viện Boost là một tập hợp các thư viện C++ có mục đích chung. Đó là sự phụ thuộc cần thiết nếu muốn sử dụng triển khai SystemC.  
Lệnh : `sudo apt install libboost-all-dev`

Thay đổi thư mục đến nơi muốn tải xuống nguồn gem5. Sau đó, để sao chép kho lưu trữ, sử dụng lệnh git clone :

- `git clone https://github.com/gem5/gem5`

Xây dựng bản dựng gem 5 đầu tiên :

Để xây dựng gem5, chúng ta sẽ sử dụng SCons. SCons sử dụng tệp SConstruct (gem5/SConstruct) để thiết lập một số biến, sau đó sử dụng tệp SConscript trong mọi thư mục con để tìm và biên dịch tất cả nguồn gem5.

SCons tự động tạo thư mục gem5/build khi được thực thi lần đầu tiên. Trong thư mục này, ta sẽ tìm thấy các tệp được tạo bởi SCons, trình biên dịch, v.v. Sẽ có một thư mục riêng cho từng bộ tùy chọn (ISA và giao thức kết hợp bộ đệm) sử dụng để biên dịch gem5.

```

anhduc@AnhDuc12:~/gem5$ ls
CODE-OF-CONDUCT.md  MAINTAINERS.yaml  TESTING.md  ext  requirements.txt  util
CONTRIBUTING.md    README.md          build       include  site_scons
COPYING             RELEASE-NOTES.md  build_opts  m5out  src
KCONFIG.md          SConsopts         build_tools optional-requirements.txt system
LICENSE             SConstruct        configs     pyproject.toml  tests

anhduc@AnhDuc12:~/gem5$ cd build
anhduc@AnhDuc12:~/gem5/build$ ls
X86
anhduc@AnhDuc12:~/gem5/build$

```

Hình 3-2 Cấu trúc file gem5

Lệnh thực hiện :

- python3 `which scons` build/X86/gem5.opt -j9

```

Checking for C header file linux/if_tun.h... (cached) yes
Checking size of struct kvm_xsave ... (cached) yes
Checking whether __i386__ is declared... (cached) no
Checking whether __x86_64__ is declared... (cached) yes
Checking for compiler -Wno-self-assign-overloaded support... (cached) yes
Checking for linker -Wno-free-nonheap-object support... (cached) yes
scons: done reading SConscript files.
scons: Building targets ...
[VER TAGS] -> X86/sim/tags.cc
scons: 'build/X86/gem5.opt' is up to date.
scons: done building targets.

```

Hình 3-3 Build gem5 bằng scons

### 3.3 Các kiểu Binary và các tệp mô hình trong gem5

#### 3.3.1 Các kiểu binary

##### ❖ debug

Được xây dựng mà không có bất kỳ tối ưu hóa nào và có ký tự debug. Binary này hữu ích khi sử dụng bộ gỡ lỗi để theo dõi biến mà bạn cần xem xét được tối ưu hóa trong phiên bản 'opt' của gem5. Chạy với chế độ debug là chậm so với các binary khác

##### ❖ opt

Binary này được xây dựng với hầu hết các tối ưu hóa được kích hoạt (ví dụ: -O3), nhưng vẫn bao gồm các ký tự debug. Binary này nhanh hơn đáng kể so với phiên bản debug, nhưng vẫn chứa đủ thông tin debug để có thể giải quyết hầu hết các vấn đề khi debug.

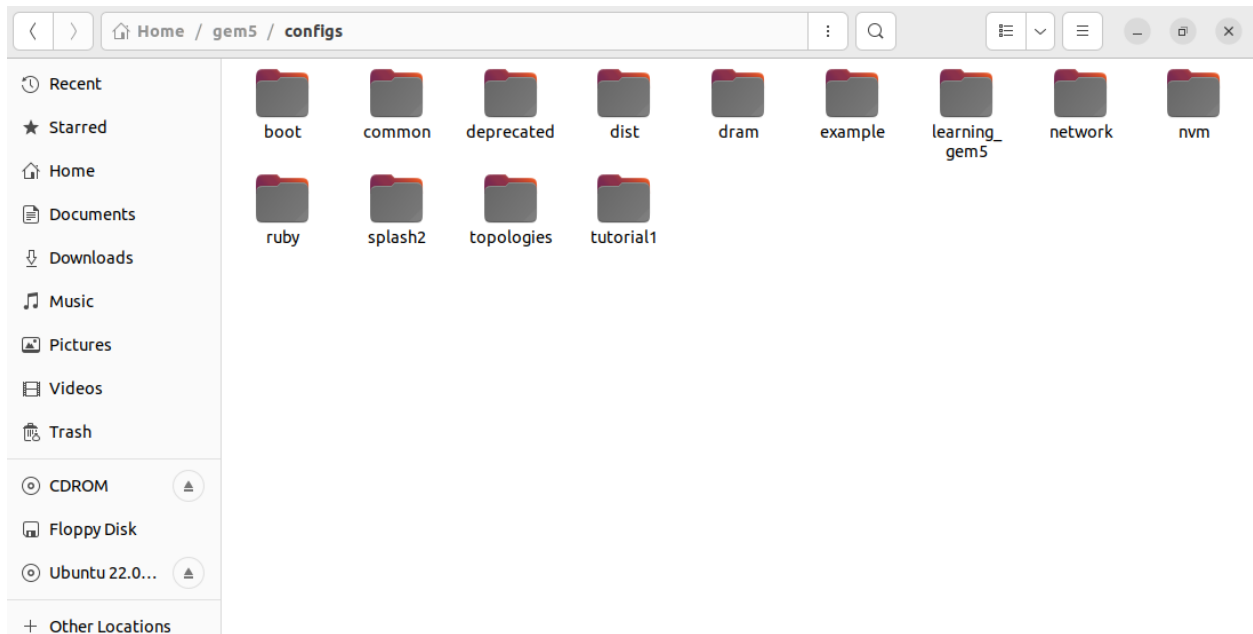
##### ❖ fast

Được xây dựng với tất cả các tối ưu hóa được kích hoạt (bao gồm cả tối ưu hóa thời gian liên kết trên các nền tảng hỗ trợ) và không có ký tự debug. Hơn nữa, mọi khai báo

assert đều đã bị loại bỏ, nhưng các panics và fatals vẫn được bao gồm. 'fast' là binary có hiệu suất cao nhất và nhỏ gọn hơn nhiều so với 'opt'. Tuy nhiên, 'fast' chỉ thích hợp khi bạn cảm thấy rằng khả năng code của bạn không có lỗi lớn.

### 3.3.2 Các tệp mô hình

Các tệp mô hình của gem 5 nằm trong thư mục configs/. Mỗi thư mục được mô tả ngắn gọn dưới đây:



Hình 3-4 Các tệp mô hình

#### - boot/

Đây là các tệp RCS được sử dụng ở chế độ toàn hệ thống. Các tệp này được trình mô phỏng tải sau khi Linux khởi động và được shell thực thi. Hầu hết chúng được sử dụng để kiểm soát điểm chuẩn khi chạy ở chế độ toàn hệ thống. Một số là các hàm tiện ích, như `hack_back_ckpt.rcS`. Những tệp tin này được đề cập sâu hơn trong chương về mô phỏng toàn bộ hệ thống.

#### - common/

Thư mục này chứa một số tập lệnh và hàm trợ giúp để tạo các hệ thống mô phỏng. Ví dụ: `Caches.py` tương tự như các tệp `caches.py` và `caches_opts.py` được tạo trong các chương trước.

- `Options.py` chứa nhiều tùy chọn có thể được đặt trên dòng lệnh. Giống như số lượng CPU, đồng hồ hệ thống và nhiều thứ khác.
- `CacheConfig.py` chứa các tùy chọn và chức năng để thiết lập tham số bộ đệm cho hệ thống bộ nhớ cổ điển.

- MemConfig.py cung cấp một số chức năng trợ giúp để thiết lập hệ thống bộ nhớ.
- FSConfig.py chứa các chức năng cần thiết để thiết lập mô phỏng toàn bộ hệ thống cho nhiều loại hệ thống khác nhau. Mô phỏng toàn bộ hệ thống sẽ được thảo luận thêm trong chương riêng của nó.
- Simulator.py chứa nhiều hàm trợ giúp để thiết lập và chạy gem5. Rất nhiều mã chứa trong tệp này quản lý việc lưu và khôi phục các điểm kiểm tra. Các tệp cấu hình ví dụ bên dưới trong ví dụ/ sử dụng các hàm trong tệp này để thực hiện mô phỏng gem5. Tệp này khá phức tạp nhưng nó cũng cho phép rất linh hoạt trong cách chạy mô phỏng.

- dram/

Chứa các tập lệnh để kiểm tra DRAM.

- example/

Thư mục này chứa một số tập lệnh cấu hình gem5 mẫu có thể được sử dụng ngay để chạy gem5. Cụ thể, se.py và fs.py khá hữu ích. Thông tin thêm về các tập tin này có thể được tìm thấy trong phần tiếp theo. Ngoài ra còn có một số script cấu hình tiện ích khác trong thư mục này.

- learning\_gem5/

Thư mục này chứa tất cả các tập lệnh cấu hình gem5 được tìm thấy trong sách learning\_gem5.

- network/

Thư mục này chứa các tập lệnh cấu hình cho mạng HeteroGarnet.

- nvm/

Thư mục này chứa các tập lệnh mẫu sử dụng giao diện NVM.

- ruby/

Thư mục này chứa các tập lệnh cấu hình cho Ruby và các giao thức kết hợp bộ đệm đi kèm của nó. Thông tin chi tiết có thể được tìm thấy trong chương về Ruby.

- splash2/

Thư mục này chứa các tập lệnh để chạy bộ điểm chuẩn Splash2 với một số tùy chọn để định cấu hình hệ thống mô phỏng.

- topologies/

Thư mục này chứa việc triển khai các cấu trúc liên kết có thể được sử dụng khi tạo hệ thống phân cấp bộ đệm Ruby. Thông tin chi tiết có thể được tìm thấy trong chương về Ruby.

### 3.4 Xây dựng tập lệnh cấu hình đơn giản

(Các file có thể xem ở link github trong phần tài liệu tham khảo)

Kịch bản cấu hình sẽ mô hình hóa một hệ thống rất đơn giản. Chúng ta sẽ chỉ có một lõi CPU đơn giản. Lõi CPU này sẽ được kết nối với bus bộ nhớ toàn hệ thống. Và chúng ta sẽ có một kênh bộ nhớ DDR3 duy nhất, cũng được kết nối với bus bộ nhớ.

- Tạo 1 tập tin cấu hình

```
mkdir configs/tutorial/part1/  
touch configs/tutorial/part1/simple.py
```

- Nhập thư viện m5 và tất cả SimObjects

```
import m5  
from m5.objects import *
```

- Tạo SimObject đầu tiên:

Hệ thống mà chúng ta sắp mô phỏng. Đối tượng “System” sẽ là cha mẹ của tất cả các đối tượng khác trong hệ thống mô phỏng. Đối tượng “System” chứa nhiều thông tin chức năng (không phải mức thời gian), như phạm vi bộ nhớ vật lý, miền đồng hồ gốc, miền điện áp gốc, hạt nhân (trong mô phỏng toàn hệ thống), v.v. Để tạo SimObject hệ thống, chúng ta chỉ cần khởi tạo nó giống như một lớp python bình thường.

```
system = System()
```

Bây giờ ta đã có một tham chiếu đến hệ thống sẽ mô phỏng, hãy đặt đồng hồ trên hệ thống. Đầu tiên chúng ta phải tạo một miền đồng hồ. Sau đó chúng ta có thể đặt tần số đồng hồ trên miền đó. Việc đặt tham số trên SimObject hoàn toàn giống với việc đặt các thành viên của một đối tượng trong python, vì vậy, chẳng hạn, chúng ta có thể chỉ cần đặt đồng hồ thành 1 GHz. Cuối cùng, chúng ta phải xác định miền điện áp cho miền đồng hồ này. Vì hiện tại chúng ta không quan tâm đến nguồn điện hệ thống nên chúng ta sẽ chỉ sử dụng các tùy chọn mặc định cho miền điện áp.

```
system.clk_domain = SrcClockDomain()  
system.clk_domain.clock = '1GHz'  
system.clk_domain.voltage_domain = VoltageDomain()
```

Sau khi có một hệ thống, hãy thiết lập cách bộ nhớ sẽ được mô phỏng. Chúng ta sẽ sử dụng chế độ thời gian cho mô phỏng bộ nhớ. Sẽ gần như luôn luôn sử dụng chế độ thời

gian cho mô phỏng bộ nhớ, trừ những trường hợp đặc biệt như fast-forwarding và khôi phục từ một điểm kiểm tra. Chúng ta cũng sẽ thiết lập một phạm vi bộ nhớ duy nhất có kích thước 512 MB, một hệ thống rất nhỏ. Trong các tập lệnh cấu hình python, mỗi khi cần kích thước, bạn có thể chỉ định kích thước đó bằng cách sử dụng ngôn ngữ và đơn vị thông thường như '512MB'. Tương tự, với thời gian, bạn có thể sử dụng đơn vị thời gian (ví dụ: '5ns'). Những giá trị này sẽ tự động được chuyển đổi sang một biểu diễn chung.

```
system.mem_mode = 'timing'  
system.mem_ranges = [AddrRange('512MB')]
```

Bây giờ chúng ta có thể tạo ra một CPU. Chúng ta sẽ bắt đầu với CPU dựa trên thời gian đơn giản nhất trong gem5 cho X86 ISA, *X86TimingSimpleCPU*. Mẫu CPU này thực thi từng lệnh trong một chu kỳ xung nhịp duy nhất để thực thi, ngoại trừ các yêu cầu bộ nhớ truyền qua hệ thống bộ nhớ. Để tạo CPU, chỉ cần khởi tạo đối tượng:

```
system.cpu = X86TimingSimpleCPU()
```

Tiếp theo, sẽ tạo bus bộ nhớ toàn hệ thống:

```
system.membus = SystemXBar()
```

Bây giờ ta đã có bus bộ nhớ, hãy kết nối các cổng bộ đệm trên CPU với nó. Trong trường hợp này, do hệ thống mà chúng ta muốn mô phỏng không có bất kỳ bộ nhớ đệm nào nên chúng ta sẽ kết nối trực tiếp các cổng I-cache và D-cache với membus. Trong hệ thống này, không có bộ nhớ đệm.

```
system.cpu.icache_port = system.membus.cpu_side_ports  
system.cpu.dcache_port = system.membus.cpu_side_ports
```

Để kết nối các thành phần hệ thống bộ nhớ với nhau, gem5 sử dụng tính năng trừu tượng hóa cổng. Mỗi đối tượng bộ nhớ có thể có hai loại cổng, *cổng yêu cầu* (request port) và *cổng phản hồi* (response port). Yêu cầu được gửi từ cổng yêu cầu đến cổng phản hồi và phản hồi được gửi từ cổng phản hồi đến cổng yêu cầu. Khi kết nối cổng, bạn phải kết nối cổng yêu cầu với cổng phản hồi.

Trong xây dựng này, icache\_port của CPU là một cổng yêu cầu, và cpu\_side của bộ nhớ cache là một cổng phản hồi. Cổng yêu cầu và cổng phản hồi có thể ở bất kỳ bên nào của

dấu '=' và kết nối sẽ được thiết lập. Sau khi thiết lập kết nối, người gửi yêu cầu có thể gửi các yêu cầu đến người đáp ứng. Có rất nhiều kỹ thuật phía sau để thiết lập kết nối, chi tiết đó không quan trọng đối với hầu hết người dùng.

Tiếp theo, chúng ta cần kết nối một số cổng khác để đảm bảo rằng hệ thống của chúng ta sẽ hoạt động chính xác. Chúng ta cần tạo bộ điều khiển I/O trên CPU và kết nối nó với bus bộ nhớ. Ngoài ra, chúng ta cần kết nối một cổng đặc biệt trong hệ thống với membus. Cổng này là cổng chỉ có chức năng cho phép hệ thống đọc và ghi bộ nhớ.

```
system.cpu.createInterruptController()
system.cpu.interrupts[0].pio = system.membus.mem_side_ports
system.cpu.interrupts[0].int_requestor = system.membus.cpu_side_ports
system.cpu.interrupts[0].int_responder = system.membus.mem_side_ports

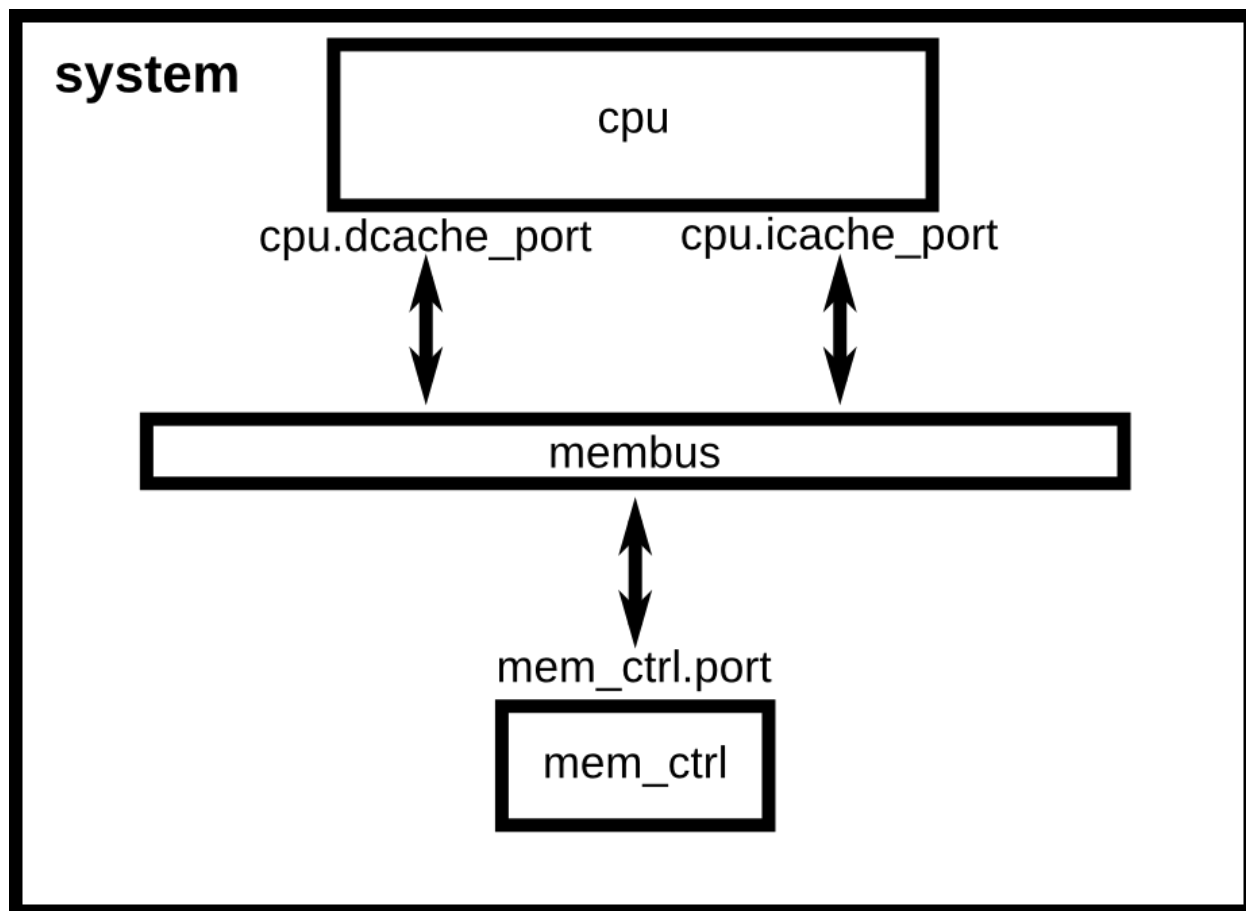
system.system_port = system.membus.cpu_side_ports
```

Tiếp theo, chúng ta cần tạo bộ điều khiển bộ nhớ và kết nối nó với membus. Đối với hệ thống này, ta sẽ sử dụng bộ điều khiển DDR3 đơn giản và nó sẽ chịu trách nhiệm về toàn bộ phạm vi bộ nhớ trong hệ thống.

```
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports
```

Sau những kết nối cuối cùng đó, ta đã hoàn thành việc khởi tạo hệ thống mô phỏng của mình. Hệ thống trên sẽ giống như hình bên dưới.





*Hình 3-5 Hệ thống mô phỏng đơn giản*

Gem5 có thể chạy ở hai chế độ khác nhau được gọi là chế độ “mô phỏng tòa nhà (syscall emulation)” và “toàn bộ hệ thống (full system)” hoặc chế độ SE và FS. Ở chế độ toàn hệ thống, gem5 mô phỏng toàn bộ hệ thống phần cứng và chạy kernel chưa sửa đổi. Chế độ toàn hệ thống tương tự như chạy máy ảo.

Mặt khác, chế độ mô phỏng Syscall không mô phỏng tất cả các thiết bị trong hệ thống và tập trung vào việc mô phỏng CPU và hệ thống bộ nhớ. Mô phỏng Syscall dễ cấu hình hơn nhiều vì không bắt buộc phải khởi tạo tất cả các thiết bị phần cứng cần thiết trong hệ thống thực. Tuy nhiên, mô phỏng syscall chỉ mô phỏng các cuộc gọi hệ thống Linux và do đó chỉ mô hình hóa mã chế độ người dùng.

Nếu không cần lập mô hình hệ điều hành cho các câu hỏi nghiên cứu của mình và muốn có hiệu suất cao hơn, nên sử dụng chế độ SE. Tuy nhiên, nếu cần mô hình hóa hệ thống có

độ chính xác cao hoặc tương tác với hệ điều hành như các bước đi trong bảng trang là quan trọng thì nên sử dụng chế độ FS.

Tiếp theo, ta phải tạo quy trình (một SimObject khác). Sau đó, đặt lệnh quy trình thành lệnh muốn chạy. Đây là một danh sách tương tự như argv, với tệp thực thi ở vị trí đầu tiên và các đối số của tệp thực thi ở phần còn lại của danh sách. Sau đó, thiết lập CPU để sử dụng quy trình làm khối lượng công việc và cuối cùng tạo bối cảnh thực thi chức năng trong CPU.

```
binary = 'tests/test-progs/hello/bin/x86/linux/hello'
```

```
# for gem5 V21 and beyond
```

```
system.workload = SEWorkload.init_compatible(binary)
```

```
process = Process()
```

```
process.cmd = [binary]
```

```
system.cpu.workload = process
```

```
system.cpu.createThreads()
```

Điều cuối cùng cần làm là khởi tạo hệ thống và bắt đầu thực thi. Đầu tiên chúng ta tạo Root đối tượng. Sau đó, khởi tạo mô phỏng. Quá trình khởi tạo sẽ đi qua tất cả các SimObject mà ta đã tạo trong Python và tạo ra các C++ SimObject tương đương.

```
root = Root(full_system = False, system = system)
```

```
m5.instantiate()
```

Cuối cùng, chúng ta có thể bắt đầu mô phỏng thực tế. Hiện tại, gem5 hiện đang sử dụng print các hàm kiểu 3 của Python, do đó print không còn là một câu lệnh nữa và phải được gọi là một hàm.

```
print("Beginning simulation!")
```

```
exit_event = m5.simulate()
```

Và sau khi mô phỏng kết thúc, ta có thể kiểm tra trạng thái của hệ thống.

```
print('Exiting @ tick {} because {}'.format(m5.curTick(), exit_event.getCause()))
```

Chạy chương trình bằng lệnh : `build/X86/gem5.opt configs/tutorial1/simple.py`

Kết quả :

```
anhduc@AnhDuc12:~/gem5$ vi configs/tutorial1/simple.py
anhduc@AnhDuc12:~/gem5$ build/X86/gem5.opt configs/tutorial1/simple.py
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 23.0.1.0
gem5 compiled Dec 18 2023 17:55:34
gem5 started Dec 25 2023 00:31:05
gem5 executing on AnhDuc12, pid 2327
command line: build/X86/gem5.opt configs/tutorial1/simple.py

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and p
df.
src/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes) does not
match the address range assigned (512 Mbytes)
src/base/statistics.hh:279: warn: One of the stats is a legacy stat. Legacy stat
is a stat that does not belong to any statistics::Group. Legacy stat is depreca
ted.
system.remote_gdb: Listening for connections on port 7000
Beginning simulation!
src/sim/simulate.cc:194: info: Entering event queue @ 0. Starting simulation...

Hello world!
Exiting @ tick 454646000 because exiting with last active thread context
anhduc@AnhDuc12:~/gem5$
```

*Hình 3-6 Build bản mô phỏng gem5 đơn giản*

#### 4.4 Thông tin của các bản mô phỏng

(Các file có thể xem ở trong link github trong phần tài liệu tham khảo )

Sau khi mô phỏng chúng ta sẽ thu được một số kết quả nằm trong thư mục “m5out” gồm

3 file : `config.ini`, `config.json` và `stats.txt`

- `config.ini` : Chứa danh sách mọi `SimObject` được tạo cho mô phỏng và các giá trị cho các tham số của nó.
- `config.json` : Gõn `config.init` nhưng ở dạng json
- `stats.txt` : Một bản trình bày văn bản của tất cả số liệu thống kê gem5 đã đăng ký cho mô phỏng.

Kết quả dựa theo bản mô phỏng đã dựng ở phần 4.3 :

`Config.ini`

```

[root]
type=Root
children=system
eventq_index=0
full_system=false
sim_quantum=0
time_sync_enable=false
time_sync_period=1000000000000
time_sync_spin_threshold=100000000

[system]
type=System
children=clk_domain cpu dvfs_handler mem_ctrl membus workload
auto_unlink_shared_backstore=false
cache_line_size=64
eventq_index=0
exit_on_work_items=false
init_param=0
m5ops_base=0
mem_mode=timing
mem_ranges=0:536870912
memories=system.mem_ctrl.dram
mmap_using_noreserve=false
multi_thread=false
num_work_ids=16
readfile=
redirect_paths=
shadow_rom_ranges=
shared_backstore=
symbolfile=
thermal_components=
thermal_model=None
-- INSERT --

```

2,1

[Top](#)

Hình 3-7 Config.ini 1

```
anhduc@AnhDuc12: ~/gem5/ X + v - □ X

[system.clk_domain]
type=SrcClockDomain
children=voltage_domain
clock=1000
domain_id=-1
eventq_index=0
init_perf_level=0
voltage_domain=system.clk_domain.voltage_domain

[system.clk_domain.voltage_domain]
type=VoltageDomain
eventq_index=0
voltage=1.0

[system.cpu]
type=BaseTimingSimpleCPU
children=decoder interrupts isa mmu power_state tracer workload
branchPred=Null
checker=Null
clk_domain=system.clk_domain
cpu_id=-1
decoder=system.cpu.decoder
do_checkpoint_insts=true
do_statistics_insts=true
eventq_index=0
function_trace=false
function_trace_start=0
interrupts=system.cpu.interrupts
isa=system.cpu.isa
max_insts_all_threads=0
max_insts_any_thread=0
-- INSERT --
```

68,1 10%

Hình 3-8 Config.ini 2

Config.json

```
{
  "type": "Root",
  "cxx_class": "gem5::Root",
  "name": null,
  "path": "root",
  "eventq_index": 0,
  "full_system": false,
  "sim_quantum": 0,
  "time_sync_enable": false,
  "time_sync_period": 1000000000000,
  "time_sync_spin_threshold": 100000000,
  "system": {
    "type": "System",
    "cxx_class": "gem5::System",
    "name": "system",
    "path": "system",
    "auto_unlink_shared_backstore": false,
    "cache_line_size": 64,
    "eventq_index": 0,
    "exit_on_work_items": false,
    "init_param": 0,
    "m5ops_base": 0,
    "mem_mode": "timing",
    "mem_ranges": [
      "0:536870912"
    ],
    "memories": [
      "system.mem_ctrl.dram"
    ],
    "mmap_using_noreserve": false,
    "multi_thread": false,
    "num_work_ids": 16,

```

1,1 [Top](#)

Hình 3-9 Config.json 1

```
anhduc@AnhDuc12: ~/gem5/ × + ▾ - □ ×
},
"clk_domain": {
  "type": "SrcClockDomain",
  "cxx_class": "gem5::SrcClockDomain",
  "name": "clk_domain",
  "path": "system.clk_domain",
  "clock": [
    1000
  ],
  "domain_id": -1,
  "eventq_index": 0,
  "init_perf_level": 0,
  "voltage_domain": {
    "type": "VoltageDomain",
    "cxx_class": "gem5::VoltageDomain",
    "name": "voltage_domain",
    "path": "system.clk_domain.voltage_domain",
    "eventq_index": 0,
    "voltage": [
      1.0
    ]
  }
},
"cpu": {
  "type": "BaseTimingSimpleCPU",
  "cxx_class": "gem5::TimingSimpleCPU",
  "name": "cpu",
  "path": "system.cpu",
  "branchPred": null,
  "checker": null,
  "clk_domain": "system.clk_domain",
  "cpu_id": -1,
}
-- INSERT -- 81,1 10%
```

Hình 3-10 Config.json 2

Stats.txt

```

----- Begin Simulation Statistics -----
simSeconds          0.000455          # Number of seconds simulated (Second)
simTicks            454646000          # Number of ticks simulated (Tick)
finalTick           454646000          # Number of ticks from beginning of simulation (restored
from checkpoints and never reset) (Tick)
simFreq             1000000000000      # The number of ticks per simulated second ((Tick/Second
))
hostSeconds         0.06              # Real time elapsed on the host (Second)
hostTickRate        7946454079        # The number of ticks simulated per host second (ticks/s)
((Tick/Second))
hostMemory          637948            # Number of bytes of host memory used (Byte)
simInsts            5701              # Number of instructions simulated (Count)
simOps              10302             # Number of ops (including micro ops) simulated (Count)
hostInstRate        99507             # Simulator instruction rate (inst/s) ((Count/Second))
hostOpRate          179795            # Simulator op (including micro ops) rate (op/s) ((Count/
Second))
system.clk_domain.clock      1000      # Clock period in ticks (Tick)
system.clk_domain.voltage_domain.voltage 1      # Voltage in Volts (Volt)
system.cpu.numCycles         454646    # Number of cpu cycles simulated (Cycle)
system.cpu.cpi               79.567028 # CPI: cycles per instruction (core level) ((Cycle/Count)
)
system.cpu.ipc               0.012568   # IPC: instructions per cycle (core level) ((Count/Cycle)
)
system.cpu.numWorkItemsStarted      0      # Number of work items this cpu started (Count)
system.cpu.numWorkItemsCompleted    0      # Number of work items this cpu completed (Count)
system.cpu.commitStats0.numInsts     5714  # Number of instructions committed (thread level) (Count)
system.cpu.commitStats0.numOps       10315 # Number of ops (including micro ops) committed (thread l
evel) (Count)
system.cpu.commitStats0.numInstsNotNOP 0      # Number of instructions committed excluding NOPs or pref
etches (Count)
system.cpu.commitStats0.numOpsNotNOP  0      # Number of Ops (including micro ops) Simulated (Count)
"stats.txt" 499L, 64125B              1,0-1      Top

```

Hình 3-11 Stats.txt



## CHƯƠNG 4 . ỨNG DỤNG VÀ THỰC NGHIỆM

### 4.1 Mô phỏng Multi2Sim

(Các file có thể xem tại link github ở mục tài liệu tham khảo)

Chương trình C sử dụng thư viện “pthread” để tạo và quản lý nhiều luồng (threads).

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define MAX_THREADS 32

void *child_fn(void *arg)
{
    int idx = (int) arg;
    printf("I'm thread %d\n", idx);
    fflush(stdout);
    return NULL;
}

int main(int argc, char **argv)
{
    int nthreads, i;
    pthread_t child[MAX_THREADS];

    /* Syntax */
    if (argc < 2) {
        fprintf(stderr, "syntax: test-threads <nthreads>\n");
        exit(1);
    }

    /* Get number of threads */
    nthreads = atoi(argv[1]);
    if (nthreads < 1 || nthreads > MAX_THREADS) {
        fprintf(stderr, "error: nthreads must be > 0 and < %d\n", MAX_THREADS);
        exit(1);
    }

    /* Run */
    for (i = 1; i < nthreads; i++)
        pthread_create(&child[i], NULL, child_fn, (int *) i);
    child_fn((int *) 0);
}
```

```
for (i = 1; i < nthreads; i++)  
    pthread_join(child[i], NULL);  
return 0;  
}
```

\* Compile sources

```
gcc -m32 test-threads.c -o test-threads -lpthread
```

\* Chạy trên Multi2Sim với số luồng = 4 :

```
m2s test-threads 4
```

```
# gcc -m32 test-threads.c -o test-threads -lpthread
# m2s test-threads 4

; Multi2Sim 5.0 - A Simulation Framework for CPU-GPU Heterogeneous Computing
; Please use command 'm2s --help' for a list of command-line options.
; Simulation alpha-numeric ID: ONdYq


I'm thread 0
I'm thread 3
I'm thread 1
I'm thread 2

;
; Simulation Statistics Summary
;

[ General ]
RealTime = 0.05 [s]
SimEnd = ContextsFinished

[ x86 ]
RealTime = 0.05 [s]
Instructions = 238735
InstructionsPerSecond = 4455258

# []
```

RAM 3.50 GB CPU 0.00%  Signed in

*Hình 4-1 m2s test-threads 4*

\* Chạy mô phỏng thời gian trên Multi2Sim

```
m2s --x86-sim detailed test-threads 4
```

```

;
; Simulation Statistics Summary
;

[ General ]
RealTime = 12.50 [s]
SimEnd = ContextsFinished
SimTime = 1497810.00 [ns]
Frequency = 1000 [MHz]
Cycles = 1497811

[ x86 ]
RealTime = 12.49 [s]
Instructions = 518664
InstructionsPerSecond = 41529
SimTime = 1497811.00 [ns]
Frequency = 1000 [MHz]
Cycles = 1497811
CyclesPerSecond = 119930
FastForwardInstructions = 0
CommittedInstructions = 225348
CommittedInstructionsPerCycle = 0.1505
CommittedMicroInstructions = 379142
CommittedMicroInstructionsPerCycle = 0.2531
BranchPredictionAccuracy = 0.8451

```

*Hình 4-2 m2s --x86-sim detailed test-threads 4*

#### Tổng Quan:

- RealTime: Thời gian thực hiện mô phỏng, 12.50 giây.
- SimEnd: Kết thúc mô phỏng khi tất cả ngữ cảnh đã hoàn thành.
- SimTime: Thời gian được mô phỏng, 1497810.00 ns.
- Frequency: Tần số mô phỏng, 1000 MHz.
- Cycles: Tổng số chu kỳ mô phỏng, 1497811.

#### Kiến Trúc x86:

- RealTime: Thời gian thực hiện mô phỏng x86, 12.49 giây.
- Instructions: Tổng số lệnh thực thi, 518664.
- InstructionsPerSecond: Tốc độ thực hiện lệnh mỗi giây, 41529.

- SimTime: Thời gian mô phỏng x86, 1497811.00 ns.
- Frequency: Tần số mô phỏng x86, 1000 MHz.
- Cycles: Tổng số chu kỳ mô phỏng x86, 1497811.
- CyclesPerSecond: Tốc độ chu kỳ mỗi giây mô phỏng x86, 119930.
- FastForwardInstructions: Số lệnh được chuyển tiếp nhanh (nếu có), 0.
- CommittedInstructions: Số lệnh đã thực hiện, 225348.
- CommittedInstructionsPerCycle: Tốc độ thực hiện lệnh mỗi chu kỳ, 0.1505.
- CommittedMicroInstructions: Số micro lệnh đã thực hiện, 379142.
- CommittedMicroInstructionsPerCycle: Tốc độ thực hiện micro lệnh mỗi chu kỳ, 0.2531.
- BranchPredictionAccuracy: Độ chính xác của dự đoán nhảy, 0.8451.

Những thông số này cung cấp thông tin chi tiết về hiệu suất và hành vi của chương trình được mô phỏng trên kiến trúc x86

Chạy lệnh :

```
m2s --x86-sim detailed --x86-report report test-threads
```

Tạo ra file “report” chứa thông tin chi tiết của kết quả mô phỏng x86

```
docker exec -it 45dd3309dbf X + v
;
; CPU Configuration
;

[ Config.General ]
Frequency = 1000
Cores = 1
Threads = 1
FastForward = 0
ContextQuantum = 100000
ThreadQuantum = 1000
ThreadSwitchPenalty = 0
RecoverKind = Writeback
RecoverPenalty = 0

[ Config.Pipeline ]
FetchKind = TimeSlice
DecodeWidth = 4
DispatchKind = TimeSlice
DispatchWidth = 4
IssueKind = TimeSlice
IssueWidth = 4
CommitKind = Shared
CommitWidth = 4
OccupancyStats = False

[ Config.Queues ]
FetchQueueSize = 64
UopQueueSize = 32
RobKind = Private
RobSize = 64
IqKind = Private
IqSize = 40
LsqKind = Private
"report" 931 lines, 20565 characters
```

Hình 4-3 File report\_1

Khởi tạo 1 file “x86-config” :

```
[ General ]
```

Cores = 2

Tạo cho nó số lõi cores = 2

Khởi chạy lệnh :

```
m2s --x86-sim detailed --x86-config x86-config --x86-report report test-threads
```

Sẽ có sự thay đổi một số thông về báo cáo trong file “report” vì cores được thay đổi.

Giờ ta sẽ thay đổi cho hệ thống mô phỏng : Hình dạng bộ nhớ đệm , mô-đun bộ nhớ đệm , bộ nhớ chính , mạng và cấu hình mô phỏng

Tạo file “mem-config” chứa các thông tin thay đổi :

```
[CacheGeometry geo-l1]
Sets = 128
Assoc = 2
BlockSize = 256
Latency = 2
Policy = LRU
Ports = 2

[CacheGeometry geo-l2]
Sets = 512
Assoc = 4
BlockSize = 256
Latency = 20
Policy = LRU
Ports = 4

[Module mod-l1-0]
Type = Cache
Geometry = geo-l1
LowNetwork = net-l1-l2
LowModules = mod-l2-0 mod-l2-1

[Module mod-l1-1]
Type = Cache
Geometry = geo-l1
LowNetwork = net-l1-l2
LowModules = mod-l2-0 mod-l2-1

[Module mod-l1-2]
Type = Cache
```

Geometry = geo-l1  
LowNetwork = net-l1-l2  
LowModules = mod-l2-0 mod-l2-1

[Module mod-l2-0]  
Type = Cache  
Geometry = geo-l2  
HighNetwork = net-l1-l2  
LowNetwork = net-l2-mm  
LowModules = mod-mm  
AddressRange = BOUNDS 0x00000000 0x7FFFFFFF

[Module mod-l2-1]  
Type = Cache  
Geometry = geo-l2  
HighNetwork = net-l1-l2  
LowNetwork = net-l2-mm  
LowModules = mod-mm  
AddressRange = BOUNDS 0x80000000 0xFFFFFFFF

[Module mod-mm]  
Type = MainMemory  
BlockSize = 256  
Latency = 200  
HighNetwork = net-l2-mm

[Network net-l2-mm]  
DefaultInputBufferSize = 1024  
DefaultOutputBufferSize = 1024  
DefaultBandwidth = 256

[Network net-l1-l2]  
DefaultInputBufferSize = 1024  
DefaultOutputBufferSize = 1024  
DefaultBandwidth = 256

[Entry core-0]  
Arch = x86  
Core = 0  
Thread = 0  
DataModule = mod-l1-0  
InstModule = mod-l1-0



```
[Entry core-1]
Arch = x86
Core = 1
Thread = 0
DataModule = mod-l1-1
InstModule = mod-l1-1
```

```
[Entry core-2]
Arch = x86
Core = 2
Thread = 0
DataModule = mod-l1-2
InstModule = mod-l1-2
```

Chỉnh lại “file x86-config” :

```
[ General ]
Cores = 3
Threads = 1
```

Chạy chương trình trên Multi2Sim :

```
m2s --x86-sim detailed \--x86-config x86-config \--mem-config mem-config \--mem-report report \test-threads 3
```

Kết quả file “report” hiện thị báo cáo về bộ nhớ

```
docker exec -it 45dd3309dbf X + v
; Report for caches, TLBs, and main memory
;   Accesses - Total number of accesses - Reads, Writes, and NCWrites (non-coherent)
;   Hits, Misses - Accesses resulting in hits/misses
;   HitRatio - Hits divided by accesses
;   Evictions - Invalidated or replaced cache blocks
;   Retries - For L1 caches, accesses that were retried
;   ReadRetries, WriteRetries, NCWriteRetries - Read/Write retried accesses
;   Reads, Writes, NCWrites - Total read/write accesses
;   BlockingReads, BlockingWrites, BlockingNCWrites - Reads/writes coming from lower-level cache
;   NonBlockingReads, NonBlockingWrites, NonBlockingNCWrites - Coming from upper-level cache

[ mod-l1-0 ]

Sets = 128
Ways = 2
ReplacementPolicy = LRU
WritePolicy = WriteBack
BlockSize = 256
DataLatency = 2
Ports = 2

Accesses = 92644
CoalescedAccesses = 34802
RetriedAccesses = 39
Evictions = 1535
Hits = 90578
Misses = 2066
HitRatio = 0.9777

Reads = 73772
CoalescedReads = 26820
ReadHits = 72064
ReadMisses = 1708
"report" 727 lines, 15484 characters
```

Hình 4-4 File report\_2

## 4.2 Mô phỏng Gem5

## CHƯƠNG 5 . KẾT LUẬN

## CHƯƠNG 6 . TÀI LIỆU THAM KHẢO

1. <http://www.multi2sim.org/>
2. <https://www.gem5.org/documentation/>
3. <https://github.com/gem5/gem5>
4. <https://github.com/Multi2Sim/multi2sim>
5. ASPLOS 2014 Title Multi2Sim 4.2 — A Compilation and Simulation Framework for Heterogeneous Computing Authors March 2014
6. <https://www.youtube.com/watch?v=fD3hhNnfL6k&list=PL5b3yU2NFowPDqaUaIbta0gsk2UVvAqJP>
7. <https://hub.docker.com/r/multi2sim/multi2sim>

8. <https://transport.itu.edu.tr/docs/librariesprovider99/dersnotlari/dersnotlarimak537e/notlar/lecture-14---system-simulation.pdf?sfvrsn=2#:~:text=Simulation%20of%20a%20system%20is,using%20a%20model%20for%20simulation.>
9. Các file code và lệnh nằm trong [https://github.com/AnhDuc0312/simulation\\_system](https://github.com/AnhDuc0312/simulation_system)