

1 Installazione e setup dei simulatori per Windows e macOS

1.1 Premessa fondamentale: architetture e compatibilità

Prima di procedere, è essenziale capire che questi simulatori non sono normali applicazioni installabili con un semplice clic. Entrambi sono nati in ambito accademico per Linux, ma hanno esigenze molto diverse legate alla loro "età".

Questi due simulatori architetturali (Gem5 e MarssX86) nascono e vengono sviluppati nativamente per sistemi Unix-like.

- **macOS** è un sistema certificato Unix. "Parla la stessa lingua" di Linux, quindi può eseguire molti di questi programmi nativamente (direttamente dal terminale), con il solo aiuto di alcune librerie esterne.
- **Windows** ha un'architettura completamente diversa, per cui non può eseguire nativamente questi programmi. Per poterli eseguire è quindi necessario utilizzare delle soluzioni di **virtualizzazione o sottosistemi** per creare l'ambiente Unix necessario.

Bisogna anche considerare che, anche avendo un sistema Linux (o Unix), c'è il problema delle librerie software necessarie per compilare il codice (il "Time Gap").

- **Gem5** è un software **moderno e adattivo**, che viene continuamente sviluppato e aggiornato. Funziona senza problemi sui sistemi operativi moderni: su macOS lo si compila direttamente mentre su Windows si usa WSL con un'installazione Linux recente (Ubuntu 20.04/22.04).
- **MarssX86** è un progetto "legacy" (molto vecchio, fermo ad 2012 circa). Richiede compilatori (GCC vecchi) e librerie che sono state rimosse dai sistemi operativi moderni da anni. Né macOS moderno ne WSL possono eseguirlo facilmente, dunque è necessario creare una **Macchina virtuale** (con VirtualBox) che simuli un computer del 2014 con installato **Ubuntu** 14.04.

1.2 Installazione e setup del simulatore Gem5

Per quanto riguarda l'installazione del simulatore Gem5, la soluzione varia leggermente a seconda del sistema operativo utilizzato.

1.2.1 Installazione e setup di Gem5 su Windows

Per installare il simulatore su Windows la soluzione consigliata è **WSL** (Windows Subsystem for Linux). Questo sottosistema permette di avere un terminale Ubuntu vero e proprio integrato in Windows. Dunque, l'installazione si compone di diversi passaggi:

- **Passo 1: attivazione di WSL**
 1. Aprire la **PowerShell** come amministratore;
 2. Digitare `wsl --install`, premere invio e attendere il completamento del download;
 3. Al termine, seguire le istruzioni a schermo per creare *username* e *password* Ubuntu.

Una volta fatto ciò, la situazione iniziale nel terminale dovrebbe essere quella mostrata in figura 1, ed eseguendo il comando `pwd` sarà possibile vedere il proprio posizionamento all'interno del sottosistema WSL.

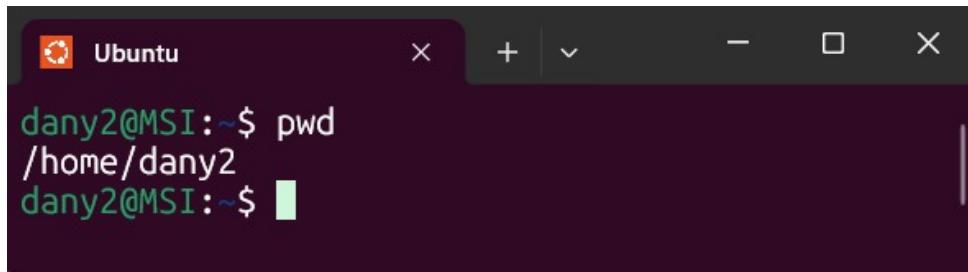


Figure 1: Schermata iniziale del sottosistema WSL

- **Passo 2: installazione delle dipendenze**

Nel terminale di Ubuntu appena aperto incollare questi comandi uno alla volta:

```
//Aggiorna i pacchetti
sudo apt update && sudo apt upgrade -y
```

```
//Installta compilatori, Python e librerie necessarie a Gem5
sudo apt install build-essential git m4 scons zlib1g zlib1g-dev
    libprotobuf-dev protobuf-compiler libprotoc-dev libgoogle-perftools-
    dev python3-dev python-is-python3 libboost-all-dev pkg-config libpng
    -dev libhdf5-dev libcapstone-dev -y
```

Note: facendo copia-incolla dei comandi potrebbero verificarsi dei problemi dovuti ai "-" letti dal terminale di Ubuntu come caratteri speciali, in tal caso è consigliato riscriverli a mano.

- **Passo 3: scaricare e compilare Gem5**

Prima di **scaricare** Gem5 è consigliato creare una cartella adatta a poterlo contenere.

Quest'operazione viene svolta principalmente perché in futuro dovremo scaricare anche il simulatore architetturale **MarssX86**: così facendo avremo entrambi i simulatori in una cartella apposita.

Senza spostarsi dalla posizione iniziale /home/utente si va a creare la cartella:

```
mkdir dev_local // Crea la cartella dev_local
cd dev_local // Ci si sposta dentro dev_local per scaricare gem5
```

Una volta posizionati in /home/utente/dev_local è possibile utilizzare il comando `git clone` per scaricare il simulatore gem5 dal repository ufficiale

```
//Clonare il repository del progetto nel proprio workspace
git clone https://github.com/gem5/gem5.git
```

Quando la clonazione sarà completata, all'interno di `dev_local` sarà presente un'altra cartella nominata "gem5" al cui interno saranno presenti tutti i file necessari per la compilazione del simulatore.

Prima di far partire la compilazione è necessario fare alcune **premesse importanti**.

Utilizzare un calcolatore con 16GB o meno di memoria RAM non basterà: Gem5 è enorme e usa template C++ complessi, ogni core che compila può occupare anche 2-3 GB di RAM. Inoltre, considerando che Windows utilizza parte di questa RAM, la situazione si complica ulteriormente.

Una possibile soluzione a questo problema, nel caso in cui non si disponesse di un calcolatore

con la quantità di RAM necessaria, è quella di "truccare" WSL per dargli più memoria usando il disco fisso come se fosse RAM (operazione di Swap della memoria):

1. Aprire il blocco note di windows;
2. Incollare esattamente queste righe:

```
[wsl2]
memory=10GB
swap=16GB
```

In questo modo stiamo dando a WSL massimo 10GB di RAM vera, ma aggiungiamo 16GB di "memoria lenta" su disco per evitare che la compilazione vada in crash.

3. Salvare il file con nome `.wslconfig` (assicurandosi che non ci sia `.txt` alla fine!) nella propria cartella utente principale: `C:\Utenti\NomeUtente\.wslconfig`
4. Aprire la PowerShell (sempre in modalità amministratore) e spegnere WSL per applicare la modifica con il comando `wsl --shutdown`

Una volta fatte queste modifiche, è finalmente possibile far partire la compilazione: ri-aprendo il terminale Ubuntu e posizionandosi in `/home/NomeUtente/dev_local/gem5` deve essere scritto il seguente comando:

```
scons build/ARM/gem5.opt -j 1
```

Tramite di esso la compilazione viene fatta partire utilizzando un solo core della cpu. Nonostante l'utilizzo dello "Swap" con la memoria del disco, la compilazione può essere talmente pesante che utilizzando più di un core per la compilazione si rischierebbe di saturare la RAM mandando in crash la compilazione. È **assolutamente "normale"** che i tempi di compilazione varino da 1h a 2h in base alla cpu che si ha a disposizione.

Note: non appena la compilazione del simulatore ARM termina, per incominciare la compilazione del simulatore RISC-V basterà reinviare il comando `scons` qui sopra, sostituendo ARM con RISCV.

• **Passo 4: test di funzionamento**

Per testare che il simulatore sia stato installato correttamente è possibile lanciare un comando di prova già incluso al suo interno.

Sempre posizionati in `/home/NomeUtente/dev_local/gem5`, lanciare questo comando:

```
./build/ARM/gem5.opt configs/deprecated/example/se.py --cmd=tests/test-progs/hello/bin/arm/linux/hello
```

```
dany2@MSI:~/dev_local/gem5$ ./build/ARM/gem5.opt configs/deprecated/example/se.py --cmd=tests/test-progs/hello/bin/arm/linux/hello
gem5 Simulator System. https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 25.1.0.0
gem5 compiled Jan 14 2026 01:11:53
gem5 started Jan 14 2026 01:41:36
gem5 executing on MSI, pid 576
command line: ./build/ARM/gem5.opt configs/deprecated/example/se.py --cmd=tests/test-progs/hello/bin/arm/linux/hello

warn: The se.py script is deprecated. It will be removed in future releases of gem5.
Global frequency set at 100000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
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 deprecated.
system.remote_gdb: Listening for connections on port 7000
**** REAL SIMULATION ****
Hello world!
Exiting @ tick 2920000 because exiting with last active thread context
Simulated exit code not 0! Exit code is 13
```

Figure 2: test di simulazione per Gem5

L'esecuzione di questo comando comporterà la creazione, all'interno della cartella "m5out" situata all'interno della cartella "gem5", di **due file fondamentali**:

- **"stats.txt"**: Contiene tutte le statistiche (cicli, cache miss, istruzioni eseguite, ecc.). È il file di testo enorme che si dovrà analizzare dopo l'esecuzione vera e propria del workload;
- **"config.ini"**: Contiene la descrizione dettagliata dell'hardware appena simulato (utile per verificare se è stata davvero usata la CPU che si voleva).

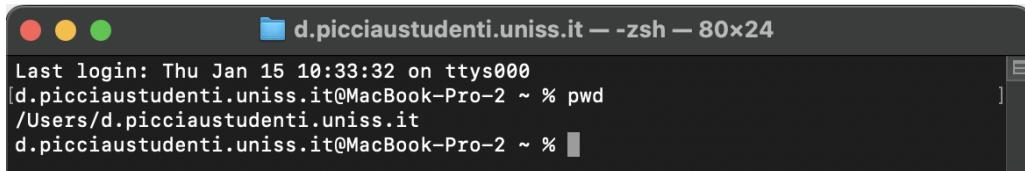
1.2.2 Installazione e setup di Gem5 su macOS

Sebbene sia teoricamente possibile compilare nativamente su macOS, la gestione delle dipendenze (in particolare Python e Protobuf) risulta spesso instabile a causa degli aggiornamenti di sistema Apple.

Per garantire la **massima riproducibilità** e stabilità, la soluzione raccomandata per macOS è l'utilizzo di un container **Docker** basato su Ubuntu 22.04. Come per WSL, questo approccio permette di avere un ambiente di sviluppo basato su Linux nativo.

- **Passo 1: preparazione del container**

1. Scaricare e installare Docker desktop dal sito ufficiale <https://www.docker.com/>;
2. È consigliato aprire il terminale posizionandosi nella cartella del proprio apple account come mostrato in figura 3;



```
d.picciaustudenti.uniss.it — zsh — 80x24
Last login: Thu Jan 15 10:33:32 on ttys000
[d.picciaustudenti.uniss.it@MacBook-Pro-2 ~ % pwd
/Users/d.picciaustudenti.uniss.it
d.picciaustudenti.uniss.it@MacBook-Pro-2 ~ %
```

Figure 3: posizionamento iniziale per l'installazione di docker

3. Creare il proprio workspace che conterrà il simulatore Gem5 e in futuro MarssX86.

```
mkdir dev_local
cd dev_local
```

- **Passo 2: installazione delle dipendenze**

1. All'interno di /Users/NomeUtente/dev_local, installare tutte le dipendenze necessarie per la creazione del container di Ubuntu 22.04:

```
docker pull ghcr.io/gem5/ubuntu-24.04_all-dependencies:v25-1
```

2. Una volta che il workspace è stato creato è possibile far partire il container di Ubuntu (sempre stando posizionati in /Users/NomeUtente/dev_local):

```
docker run --volume /Users/NomeUtente/dev_local/gem5:/gem5 --rm -it
ghcr.io/gem5/ubuntu-24.04_all-dependencies:v24-0
```

Il completamento di questo comando dovrebbe aprire una riga di comando di questo tipo, ad indicare che siamo all'interno del container Ubuntu.

```
root@1ablef08f2f:/#
```

- **Passo 3: scaricare e compilare Gem5**

All'interno del container Ubuntu appena creato, è possibile **scaricare** il repository ufficiale tramite il comando `git clone`:

```
root@1ab1ef08f2f:/# git clone https://github.com/gem5/gem5.git
```

Verrà creata una cartella "gem5" al cui interno saranno al cui interno saranno presenti tutti i file necessari per la compilazione del simulatore.

Come specificato al "**Passo 3**" del capitolo 1.2.1 la compilazione richiede una grande quantità di RAM per cui è richiesto un calcolatore con 16 o più GB di memoria RAM (solo per la compilazione di Gem5!). Dopo alcuni test da noi effettuati è risultato che la compilazione su un calcolatore con 8GB di RAM potrebbe impiegare anche mezza giornata!

Proprio per questo motivo, anche disponendo di 16GB di memoria RAM è necessario andare nella GUI di Docker impostazioni → resources → advanced, e da qui cambiare le seguenti impostazioni:

- **Memory limit**: impostiamo la capacità massima di memoria RAM che Docker può utilizzare. È consigliato impostarlo quasi al massimo, (ad esempio, con 16 GB di RAM a disposizione impostarlo a 14/15) per permettere al sistema operativo (macOS) di "sopravvivere"
- **Swap**: Invece, tramite questa impostazione, indichiamo quanto spazio di "memoria lenta" prendere dal disco nel caso in cui la RAM vada in saturazione a causa della pesantezza di compilazione.

Una volta che il repository è stato clonato ci si sposta all'interno della cartella "gem5" per far partire la **compilazione del simulatore**, utilizzando 1 o al massimo 2 core per evitare che la compilazione vada in crash a causa della saturazione della RAM a disposizione.

```
// Mi sposto all'interno della cartella gem5
root@1ab1ef08f2f:/# cd gem5

// Compilazione ARM
root@1ab1ef08f2f:/gem5# scons build/ARM/gem5.opt -j1
```

Come per Windows, anche la compilazione su macOS può impiegare da 1h a 2h, ed è "normale" vista la mole di file da scaricare.

Note: non appena la compilazione del simulatore ARM termina, per incominciare la compilazione del simulatore RISC-V basterà reinviare il comando scons qui sopra, sostituendo ARM con RISCV.

Passo 4: test di funzionamento

Per testare che il simulatore sia stato installato correttamente è possibile lanciare un comando di prova già incluso al suo interno.

Sempre posizionati in root@1ab1ef08f2f:/gem5#, lanciare questo comando:

```
./build/ARM/gem5.opt configs/deprecated/example/se.py --cmd=tests/test-
progs/hello/bin/arm/linux/hello
```

L'esecuzione di questo comando comporterà la creazione, all'interno della cartella "m5out" situata all'interno della cartella "gem5", di **due file fondamentali**:

- **"stats.txt"**: Contiene tutte le statistiche (cicli, cache miss, istruzioni eseguite, ecc.). È il file di testo enorme che si dovrà analizzare dopo l'esecuzione vera e propria del workload;
- **"config.ini"**: Contiene la descrizione dettagliata dell'hardware appena simulato (utile per verificare se è stata davvero usata la CPU che si voleva).

1.3 Installazione e setup del simulatore MarssX86

Come detto precedentemente al capitolo 1.1, MarssX86 è troppo vecchio per funzionare su WSL moderlo o su macOS. Esistono due strade che risolvono il problema, le quali funzionano sia per Windows che per macOS:

1.3.1 Docker

L'utilizzo di Docker è da prendere in considerazione nel caso in cui si voglia una maggiore rapidità, si è comodi con la riga di comando e si vuole tenere il pc pulito. Infatti, permette di scaricare un immagine di Ubuntu 14.04 e lavorarci dentro da terminale senza installare un intero sistema operativo.

- **Passo 1: Installare Docker**

1. Windows/macOS: scaricare e installare Docker desktop dal sito ufficiale <https://www.docker.com/>;
2. Avviarlo e assicurarsi che sia attivo (controllare l'icona della balena nella barra delle applicazioni).

- **Passo 2: Creare l'ambiente (il Dockerfile)**

Creare una cartella sul computer chiamata "*marss_project*" e al suo interno creare un file di testo chiamato "*Dockerfile*" in cui andrà incollato al suo interno il seguente contenuto:

```
// Si usa una versione vecchia di Ubuntu compatibile con MarssX86
FROM ubuntu:14.04

// Aggiornare i repository per trovare i vecchi pacchetti
RUN apt-get update && \
    apt-get install -y build-essential git scons zlib1g-dev python g++
    vim

// Impostare la cartella di lavoro
WORKDIR /root/marss

// Comando di default quando si entra nel container
CMD ["/bin/bash"]
```

- **Passo 3: costruire e avviare**

Aprire il terminale (o PowerShell) nella cartella dove è stato inserito il Dockerfile.

1. **Costruzione dell'immagine** tramite il comando:

```
docker build -t marss_env .
```

2. **Ingresso nel contenitore** tramite il comando:

```
docker run -it --name marss_container marss_env
```

Una volta dentro, il terminale "penserà" di essere nel 2014 garantendo la possibilità di scaricare e compilare MarssX86 li dentro

1.3.2 VirtualBox

Invece, la VirtualBox è consigliata se si preferisce avere un'interfaccia grafica, quindi vedere le finestre e usare un editor di testo grafico dentro la macchina virtuale, perché permette di avere un'esperienza più simile all'utilizzo di un computer fisico.

- **Passo 1: scaricare e installare il software**

1. Scaricare e installare **VirtualBox** al seguente indirizzo <https://www.virtualbox.org/>;
2. Scaricare la ISO di Ubuntu 14.04 LTS (Trusty Tahr) al seguente indirizzo <https://releases.ubuntu.com/14.04/>

- **Passo 2: creazione della macchina virtuale**

1. Aprire la virtualbox e andare su "nuova";
2. Nome: "*MarssVM*", Tipo: "*Linux*", Versione: *Ubuntu (64-bit)*;
3. È necessario assegnare almeno 4GB di RAM e 2 – 4 processori, fondamentali per la velocità di compilazione.
4. Creare un disco fisico di almeno 30GB;
5. Avviare la macchina, selezionare la ISO scaricata precedentemente e installare Ubuntu come se fosse un pc vero.

- **Passo 3: setup interno**

Una volta che Ubuntu è stato installato nella virtual machine, aprire il terminale interno e lanciare i seguenti comandi, uno alla volta:

1. Questo comando serve per scaricare la lista aggiornata di tutto il software disponibile.

```
sudo apt-get update
```

2. Serve per installare i **compilatori di C++** (build-essential e g++) che traducono il codice in linguaggio macchina, il **gestore dei file** (scons) che lancia in automatico tutti i file necessari, **git** per scaricare il codice da internet e **python** perché molti script di configurazione di MarssX86 sono scritti in quel linguaggio.

```
sudo apt-get install build-essential git scons zlib1g-dev python g++
```