Procesadores modernos con tan solo unos Clicks

Modern processors with just a few clicks

Lilian Bossuet[†1] and Carlos Andres Lara-Nino[*2]

[†]*Université Jean Monnet Saint-Étienne, CNRS, Laboratoire Hubert Curien UMR 5516, F-42023 SAINT-ETIENNE 42000, France*

[1]lilian.bossuet@univ-st-etienne.fr

[*]*Universitat Rovira i Virgili, Departament d'Enginyeria, Informàtica i Mathemàtiques TARRAGONA 43003, Spain*

[2]carlos.lara@fundacio.urv.cat

***Resumen*—Algunos programas educativos en áreas de ingeniería y tecnologías a menudo tienen dificultades para proporcionar a sus estudiantes los instrumentos necesarios para realizar trabajos experimentales. Esto puede deberse a dificultades económicas o simplemente por falta de oportunidades para adquirir suficientes dispositivos. El diseño asistido por computadora podría adoptarse para modelar un gran número de dispositivos con pequeños costos de operación. En este artículo se describe el uso del simulador gem5 como una herramienta que puede ser de utilidad para estudiantes de grado y posgrado en las áreas de ingeniería e informática. Revisaremos el reto que representa para los estudiantes el tener acceso a placas de prototipado modernas y describiremos cómo el uso de un simulador puede resolver este problema hasta cierto punto. Finalmente, ilustraremos paso a paso el uso del simulador gem5 para emular un microprocesador moderno.**

Palabras clave: ARM; RISC-V; gem5; emulación de microprocesadores.

***Abstract*— Modern educational programs in engineering and technology often struggle to provide their students with the necessary resources to perform experimental work. Either due to economic difficulties or simply due to a lack of opportunities for acquiring enough devices. Computer aided design could be adopted to model a large number of devices with small operational costs. This article describes the use of the gem5 simulator as a tool that can be useful for undergraduate and postgraduate students in the areas of engineering and computer science. We will review the challenge that accessing modern prototyping boards represents for students, and describe how the use of a simulator can solve this problem to some extent. Finally, we will illustrate step by step the use of the gem5 simulator to emulate a modern microprocessor system.**

Keywords: ARM; RISC-V; gem5; emulation of microprocessors.

I. Introducción

El desarrollo de la educación en las áreas de ingeniería y computación va de la mano con el uso de plataformas de prototipado que permiten al alumno efectuar actividades prácticas y conducir experimentos para enriquecer su formación profesional. Desafortunadamente, muchos estudiantes carecen de acceso a estos dispositivos. Dada la realidad socio-económica de latinoamérica, no es una expectativa razonable pensar que todos los alumnos podrían adquirir una tarjeta de pruebas sólo porque una asignatura la requiere. Si la institución educativa tiene la posibilidad de prestar estas tecnologías a sus estudiantes, este servicio generalmente viene con restricciones horarias y está sujeto a la disponibilidad del material. Sin mencionar que para un programa académico promedio, resultará complicado adquirir distintas versiones de la misma tecnología. Por ejemplo, se tendrá acceso a tarjetas Arduino con un procesador AVR® como el *PIC* de elección, sistemas que son raramente utilizados en el sector industrial o en tareas de investigación.

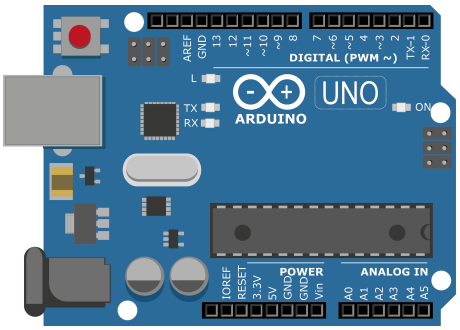


Figura 1: Las tarjetas de prototipado Arduino generalmente incluyen un microcontrolador AVR®. Se pueden programar desde una computadora usando un lenguaje estilo C.

Aunado a esto, quienes hayan tenido la experiencia de gestionar la adquisición de sistemas de procesamiento modernos en cantidades moderadas (unos 12 dispositivos) estarán al tanto del impacto de la escasez global de semiconductores. Hay largas listas de espera con tiempos de entrega que van de meses a años. Para un centro de formación esto se traduciría en al menos una promoción que no tendrá acceso a la tecnología recientemente adquirida. Para un alumno esto resultará en tener que buscar los dispositivos en mercados alternos, usualmente con sobreprecios.

Sin duda, existirá un grupo significativo de centros de educación superior con ofertas de formación en las áreas de ingenierías y computación en los cuales los alumnos no cuentan con acceso a plataformas de pruebas para desarrollar las actividades prácticas deseables. Desafortunadamente, un alto número de estudiantes llegará a optar por estas alternativas de formación, ya sea por limitantes económicos o por la proximidad a sus lugares de origen.

En todos estos casos, es imperativo identificar alternativas que permitan un acceso a la tecnología con bajo costo y alta disponibilidad. Que sean flexibles y puedan adaptarse a los avances en la investigación y desarrollo para brindar a los usuarios la oportunidad de generar experiencia con tecnologías de punta. Esto sin duda se verá reflejado en mejores oportunidades laborales y un mercado más laboral más competitivo.

II. Una computadora dentro de otra

En mayor o menor medida, se puede constatar que el acceso en latinoamérica a equipos de cómputo se ha incrementado a lo largo de los años. Estos sistemas de cómputo distan mucho de los microcontroladores requeridos en la formación de ingenierías y computación. Primero, su tamaño y especificaciones difieren: la cantidad de memoria disponible será superior en una computadora, está tendrá uno o varios procesadores más potentes, y el sistema de alimentación es generalmente constante. Podríamos llegar a la conclusión que una computadora (procesador) será generalmente superior a una tarjeta de prototipado (microprocesador). Pero al mismo tiempo, no es sencillo programar los procesadores de una computadora como lo haríamos con una tarjeta Arduino, o tener el mismo grado de acceso a sus periféricos. ¿Cómo hacer entonces para utilizar los equipos de cómputo con mayor disponibilidad para *emular* el comportamiento de un microprocesador que es difícil de conseguir?

Figura 2: Si consideramos que una computadora es un procesador grande, es lógico pensar que sería posible modelar un procesador más pequeño dentro de este sistema. El procesador más pequeño puede entonces contener sus propias aplicaciones o programas.

En las ciencias de la computación existe la noción de *máquina virtual*, que se aplica para el caso donde utilizamos un programa de computadora que va a emular otra computadora dentro del equipo. Esto puede ser de utilidad para tener acceso a un sistema operativo distinto sin necesidad de modificar el equipo, o para probar la ejecución de algunas aplicaciones en un entorno controlado.

Una de las herramientas más ampliamente utilizada para virtualizar microprocesadores es *qemu* [1], que toma su nombre del inglés "**q**uick **emu**lator." Este programa de computadora permite emular el procesador de un sistema mediante la *traducción binaria dinámica* y proporciona múltiples modelos de hardware que permiten ejecutar una variedad de sistemas operativos o también aplicaciones en *bare metal* (sin sistema operativo).

Sin embargo, un entorno virtual permite simplemente replicar el comportamiento *lógico* del sistema que deseamos emular. En el mejor de los casos, va a trasladar todas las instrucciones del entorno virtual en código máquina que puede ser interpretado por el hardware. En contraste, si se desea estudiar la cantidad de accesos a memoria, los fallos en la cache, o incluso la disipación de potencia de una plataforma, será necesario buscar otras alternativas.

III. Emulación de hardware

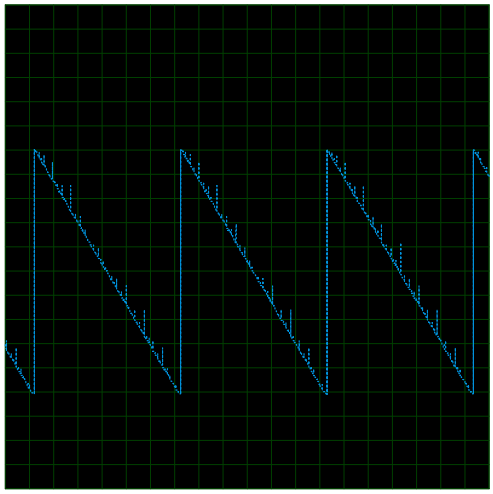
Al igual que hay emuladores del comportamiento lógico de una computadora, existe software dedicado a modelar la respuesta de un circuito electrónico, sea este analógico o digital. El más popular es sin duda SPICE [2] (del inglés "**S**imulation **P**rogram with **I**ntegrated **C**ircuit **E**mphasis"). Esta herramienta permite, entre otras cosas, verificar que la operación de un circuito analógicos sea correcta y predecir su funcionamiento. Desde su introducción en 1989, el éxito de este simulador se debe en gran medida a que es *Open Source*, esto quiere decir que el código fuente está disponible para todos, además que su uso es gratuito para actividades no lucrativas. Su desventaja principal es la complejidad de uso, ya que es necesario contar con los modelos matemáticos precisos de todos los componentes del sistema a analizar. Esto hace que el estudio de sistemas complejos, como un procesador moderno, sea prácticamente imposible.

A pesar de la complejidad de los sistemas digitales, se tiene la ventaja de que su comportamiento está definido por ciclos de reloj. Evidentemente, para modelar fenómenos térmicos o eléctricos se necesitará una resolución menor, pero en general se considera que un modelo *cycle acurate* (preciso a nivel de ciclos) es aceptable. Para estas tareas se tienen herramientas como KTechLab que son de código abierto y gratuitas y que permiten analizar circuitos digitales complejos.

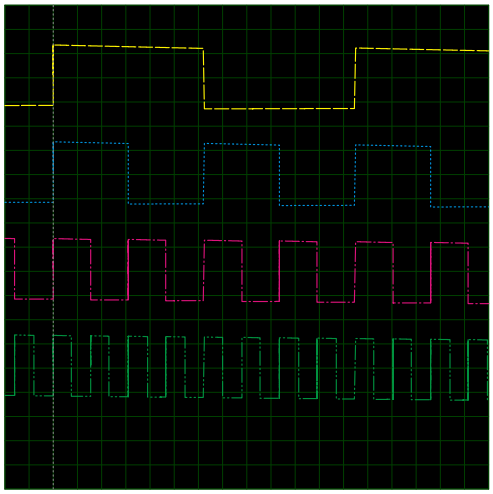
En la industria, encontramos soluciones más potentes como *Proteus Design Suite*®, que permiten estudiar el funcionamiento de componentes analógicos y digitales, así como microprocesadores con códigos internos modificables. La principal limitación para su uso en educación es su costo elevado. Es posible solicitar una licencia temporal para estudiantes, pero esta no permite guardar el estado del proyecto. Otra limitación de esta herramienta es que los modelos de los componentes son limitados y crear u obtener nuevos modelos resulta difícil si hemos de respetar la licencia de uso. Además, ¿qué pasa cuando es de interés estudiar rutinas de procesamiento más complicadas como un sistema operativo completo?

IV. Un simulador para hardware y software

Las limitaciones mencionadas en las previas secciones no aplican únicamente al caso de los estudiantes, sino que son problemas frecuentes alrededor del mundo. Es así que algunos investigadores se han dado a la tarea de dar respuesta a las preguntas planteadas mediante la creación de un sistema de simulación completo, gratuito y de código abierto, que



(a) Estudio de una señal analógica



(b) Comportamiento de un circuito digital

Figura 3: Para analizar una magnitud como el voltaje o la temperatura del circuito es necesario usar un periodo de simulación acorde a la variación de la señal. Por el contrario, para estudiar el comportamiento de un circuito digital es posible efectuar un análisis a la frecuencia del procesador.

permite la emulación de sistemas lógicos complejos, pero también hace posible modelar el hardware del sistema.

Al combinar una iniciativa de la Universidad de Wisconsin llamada GEMS [3], que viene de "**G**eneral **E**xecution-driven **M**ultiprocessor **S**imulator," con un proyecto de la Universidad de Michigan llamado M5 [4] (ambas instituciones de los EEUU) fue posible crear un simulador ahora conocido como *gem5* [5].

Figura 4: El simulador gem5 es gratuito y de código abierto.

Este sistema ofrece soporte nativo para la simulación de procesadores con ISAs de tipo Alpha, ARM, SPARC, MIPS, POWER, RISC-V y x86. Es posible lanzar una simulación reducida del sistema conocida como *system-call emulation* (SE) o una simulación completa conocida como *full system simulation* (FS) que permite interactuar con el procesador y obtener estadísticas adicionales (disponible para Alpha, ARM, SPARC, RISC-V y x86). En particular, los modelos para ARM y RISC-V proporcionan herramientas interesantes para el estudio de los microcontroladores utilizados en aplicaciones de vanguardia.

Existen ciertas limitantes principales para el uso de *gem5* en las tareas de educación. Primero, el software existe solo para sistemas operativos basados en Linux. Según datos recopilados por *statcounter*[1], para abril de 2022 casi el 75 % de las conexiones a internet provenientes de computadoras de escritorio y portátiles usaba Windows, mientras que apenas un 2.5 % usaba sistemas operativos basados en Linux. No se tienen estadísticas sólo para América latina, pero se puede esperar que el porcentaje de computadoras que usan Linux será menor. Mientras que es posible emular Ununtu o Debian en una máquina virtual, esto no es recomendado para lanzar una simulación de *gem5*.

Lo que nos lleva a la segunda limitante: la complejidad de procesamiento. El tiempo requerido para completar una simulación tipo FS dependerá en gran medida de la arquitectura a emular y la carga de procesamiento. Por ejemplo, para una computadora portátil con un procesador i7 y 32 GB de RAM, emular Ubuntu en un procesador Cortex A-57 tomará algunos minutos; note que estas especificaciones estarán muy por encima de los sistemas de cómputo normalmente disponibles para un estudiante. Para solventar esta limitante, *gem5* permite hacer uso de *checkpoints* y *regions of interest* que son instantáneas tomadas de la simulación que hacen necesario tener que ejecutar las tareas más pesadas sólo una vez y retomar el procesamiento en el punto de interés. Por ejemplo, si deseamos modificar sólo el archivo binario que será ejecutado en el microprocesador no sería práctico re-simular todo el proceso de arranque del sistema.

La tercera característica que podríamos considerar como una limitante es la inclinada pendiente en la curva de aprendizaje para trabajar con *gem5*. El principal lenguaje del simulador es Python, lo que facilita su uso en cierta medida. No obstante, un usuario avanzado requerirá conocimientos puntuales de sistemas basados en Linux, programación en C/C++, uso de compiladores cruzados, arquitectura de computadoras, sistemas de memoria, e incluso sistemas de control de versiones (*git*). Sin embargo, todo depende del cristal con que se mire. Estos conocimientos son necesarios para crear un modelo de simulación que una vez definido

[1] https://statcounter.com/

puede ser utilizado por un amplio número de usuarios. A su vez, trabajar en el diseño de estos modelos aportaría una amplia gama de conocimientos para los estudiantes.

V. Un ejemplo de uso

Para esta Sección supondremos que el equipo de trabajo cuenta con todas las dependencias necesarias[2]. También supondremos que el usuario cuenta con permisos de administrador para ejecutar algunas tareas. El ejemplo se ha creado en una computadora portátil con Ubuntu 20.04 y la versión v22.0.0.2 de *gem5*[3].

Se propone la simulación completa de un chip ARMv8-A, que ha sido utilizado en una amplia gama de sistemas desde smartphones (Huawei Mate 9) hasta tarjetas de prototipado (Hikey 960). Dentro del chip existe un *cluster* con varios núcleos Cortex-A57/A73 y un *cluster* con varios núcleos Cortex-A53 siguiendo una arquitectura big.LITTLE. El principal atractivo de este tipo de sistemas es su eficiencia energética al cambiar la carga de procesamiento de un cluster a otro según la demanda. En este microprocesador vamos a cargar el sistema operativo Ubuntu 18.04 y desde allí ejecutar una aplicación en C.

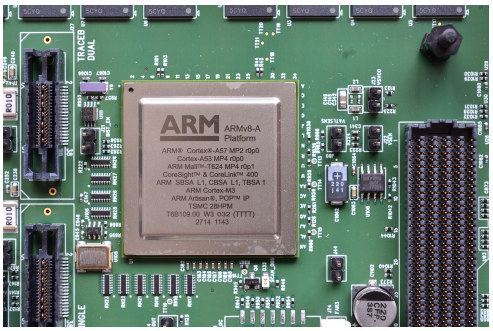


Figura 5: Un chip de la familia ARMv8-A. Las tarjetas de prototipado equipadas con estos microprocesadores tienen un costo en el mercado que va de los 200 a 250 USD.

El primer paso a realizar es obtener una copia del código del simulador, algo que es tan sencillo como abrir una terminal y ejecutar el comando:

```
git clone https://gem5.googlesource.com/public/gem5
```

Una vez que se ha finalizado la descarga, se debe entrar al directorio creado y compilar el simulador. Esto se puede llevar a cabo con las instrucciones:

```
cd gem5
scons build/ARM/gem5.opt -j $(nproc)
```

Note que en este caso vamos a compilar el simulador para trabajar con arquitecturas ARM. Si se desea usar una ISA diferente se deberá modificar la segunda instrucción, por ejemplo `scons build/X86/gem5.opt` creará los componentes necesarios para simular sistemas X86. Adicionalmente, usaremos la extensión *opt*, pero igualmente se podría lanzar `gem5.debug` o `gem5.fast` según el tipo de optimización deseado.

El proceso para compilar el simulador es largo y pesado. Afortunadamente sólo es necesario efectuar una vez esta tarea. Cuando este ha terminado, procederemos a obtener algunos archivos binarios para conducir la simulación. Estos se pueden descargar usando:

```
mkdir common; cd common
wget http://dist.gem5.org/dist/v22-0/arm/aarch-system
    -20220707.tar.bz2
tar -xf aarch-system-20220707.tar.bz2
wget http://dist.gem5.org/dist/v22-0/arm/disks/ubuntu
    -18.04-arm64-docker.img.bz2
bzip2 -d ubuntu-18.04-arm64-docker.img.bz2
cd -
```

Dentro del directorio `gem5/common/` ahora encontraremos binarios para el proceso de *boot*, imágenes del kernel de Linux, y discos con sistemas de archivos para el sistema operativo. En particular haremos uso de los archivos `gem5/common/binaries/boot.arm64`, `gem5/common/binaries/vmlinux.arm64`, y `gem5/common/ubuntu-18.04-arm64-docker.img`. Note que todos corresponden a la arquitectura AArch64 que se encuentra en los procesadores Cortex A53/A57.

En este punto ya es posible lanzar una simulación básica del sistema para verificar que todo ha sido compilado correctamente. Para ello se hará uso de un *script* de simulación escrito en lenguaje Python. Dentro del directorio `gem5/configs/example/arm` se puede encontrar el archivo `fs_bigLITTLE.py` que permite simular una arquitectura big.LITTLE genérica. En la terminal, dentro del directorio `gem5/` se puede lanzar la simulación usando el comando:

```
./build/ARM/gem5.opt \
./configs/example/arm/fs_bigLITTLE.py \
--caches \
--bootloader=./common/binaries/boot.arm64 \
--kernel=./common/binaries/vmlinux.arm64 \
--disk=./common/ubuntu-18.04-arm64-docker.img
```

Para interactuar con el microprocesador simulado, en otra ventana de la terminal será necesario compilar la terminal virtual de *gem5* y conectarse con el simulador. Para ello basta utilizar los siguientes comandos:

```
cd gem5/util/term
make
m5term 3456
```

Note que el puerto utilizado es el mismo indicado en la primera terminal donde se lanzó la simulación, generalmente aparecerá como:

```
system.terminal: Listening for connections on port 3456
```

Si todo ha marchado bien, la segunda terminal deberá indicar la bienvenida a Ubuntu y el *login* automático de la sesión de administrador. Una vez dentro de la simulación, esta puede finalizarse usando el comando:

```
m5 exit
```

Este es una caso muy básico donde se usan esencialmente todos los parámetros de una simulación genérica de la arquitectura. En las secciones siguientes vamos a crear el modelo de simulación para el ejemplo propuesto, para ello se creará un nuevo *script* de simulación para aplicar los cambios necesarios:

```
cd configs/example/arm/
cp fs_bigLITTLE.py Hikey960.py
gedit Hikey960.py &
cd -
```

[2]https://www.gem5.org/documentation/learning_gem5/part1/building/
[3]Commit 1d03f6de941520860c673b5f7954c82a46e8b191

V-A. *Ajuste dinámico de frecuencias y voltajes*

Una característica muy importante de los microprocesadores modernos es su capacidad para modificar su frecuencia y voltaje de operación en función de la demanda de procesamiento. Esta estrategia es conocida como DVFS, del inglés "**D**ynamic **V**oltage and **F**requency **S**caling."

El primer paso será agregar opciones al *script* para para poder *pasar* los parámetros al simulador desde la línea de comandos. En la función `addOptions()` se van a agregar tres argumentos y se editarán dos que ya existen:

```
def addOptions(parser):
    [...]

    parser.add_argument("--big-cpu-clock", nargs="+",
         default="2GHz")
    parser.add_argument("--little-cpu-clock", nargs="+",
          default="1GHz")
    parser.add_argument("--big-cpu-voltage", nargs="+",
         default="1.0V")
    parser.add_argument("--little-cpu-voltage", nargs="+
         ", default="1.0V")
    parser.add_argument("--dvfs", action="store_true")

    return parser
```

En seguida se editará el modelo de los clusters para enlazar los valores de voltaje proporcionados a cada procesador:

```
def build(options):
    [...]

        system.bigCluster = big_model(system,
                              options.big_cpus,
                              options.big_cpu_clock,
                              options.big_cpu_voltage)

    [...]

        system.littleCluster = little_model(system,
                                 options.little_cpus,
                                 options.little_cpu_clock,
                                 options.
                                      little_cpu_voltage)

    [...]
```

También será necesario activar el módulo para DVFS del simulador e indicar que use los dominios de reloj asignados a cada cluster:

```
def build(options):
    [...]

    if options.dvfs:
        system.dvfs_handler.domains = [
                    system.bigCluster.clk_domain,
                    system.littleCluster.clk_domain]
        system.dvfs_handler.enable = options.dvfs

    return root
```

Luego será necesario editar el archivo `gem5/configs/example/arm/devices.py` para asignar un ID válido a cada dominio de reloj. Note que este archivo también especifica los valores para las memorias cache utilizadas en la arquitectura a simular, por ende para obtener un modelo más preciso del sistema sería necesario ajustar estos valores.

```
class CpuCluster(SubSystem):
   [...]

   self.clk_domain = SrcClockDomain(clock=cpu_clock,
                     voltage_domain=self.voltage_domain,
                     domain_id=system.numCpuClusters())
   [...]
```

La arquitectura de una computadora está declarada en archivos de extensión *dtb* (*devicetree blob*) que son generados a partir de un archivo *dts* (*devicetree source*). Esta especifcicación es usada por el kernel para instalar los drivers necesarios que permitirán al sistema operativo controlar el dispositivo. *gem5* cuenta con una herramienta que permite generar automáticamente un archivo *dtb* para cada simulación. No obstante, esta utilidad no permite enlazar correctamente los dominios de reloj para múltiples clusters. De no usar un archivo *dtb* adaptado para el modelo, el segundo cluster no sería conectado correctamente al controlador de DVFS.

Afortunadamente, el código fuente del simulador también incluye algunas plantillas para generar archivos *dtb* par algunas arquitecturas familiares. Aún así, será necesario editar estas fuentes para obtener la especificación definitiva de la arquitectura a simular. En el ejemplo propuesto buscamos un sistema big.LITTLE con cuatro núcleos en ambos clusters mientras que las plantillas genéricas sólo ofrecen hasta dos núcleos en el cluster principal.

Primero es necesario editar el archivo `gem5/system/arm/dt/Makefile` para agregar un nuevo objeto a la lista de compilación:

```
TARGETS=\
  [...]

  armv8_gem5_v1_big_little_4_4.dtb \

  [...]
```

Después se debe editar el archivo `gem5/system/arm/dt/armv8_big_little.dts` para incluir la configuración deseada:

```
  [...]

  #define _4_4 3

  [...]

  #elif CONF_CPUS == _4_4
    CPU(0,0x0)
    CPU(1,0x1)
    CPU(2,0x2)
    CPU(3,0x3)
    CPU(4,0x104)
    CPU(5,0x105)
    CPU(6,0x106)
    CPU(7,0x107)
    cpu-map {
      cluster0 {
        core0 { cpu = <&CPU0>; };
        core1 { cpu = <&CPU1>; };
        core2 { cpu = <&CPU2>; };
        core3 { cpu = <&CPU3>; };
      };
      cluster1 {
        core0 { cpu = <&CPU4>; };
        core1 { cpu = <&CPU5>; };
        core2 { cpu = <&CPU6>; };
        core3 { cpu = <&CPU7>; };
      };
    };

  [...]
```

Finalmente, usar el comando `make` dentro del directorio `gem5/system/arm/dt/` producirá el archivo `armv8_gem5_v1_big_little_4_4.dtb` que será utilizado en la simulación.

Para probar que se ha habilitado el mecanismo de DVFS de manera correcta, ahora se puede lanzar una simulación más completa del sistema:

```
./build/ARM/gem5.opt \
./configs/example/arm/Hikey960.py \
--big-cpus 4 \
--little-cpus 4 \
--caches \
```

```
--bootloader=./common/binaries/boot.arm64 \
--kernel=./common/binaries/vmlinux.arm64 \
--disk=./common/ubuntu-18.04-arm64-docker.img \
--dtb=./system/arm/dt/armv8_gem5_v1_big_little_4_4.dtb \
--dvfs \
--big-cpu-clock 2362MHz 2112MHz 1805MHz 1421MHz 903MHz \
--little-cpu-clock 1402MHz 999MHz 533MHz \
--big-cpu-voltage 1.0V 0.99V 0.98V 0.97V 0.96V \
--little-cpu-voltage 0.87V 0.86V 0.85V
```

Para usar DVFS en *gem5* es necesario que el número de valores de frecuencia para un cluster sea el mismo que el número de valores de voltaje, y que ambos estén ordenados de manera decreciente. Note que es posible especificar los diferentes puntos de operación en voltaje y frecuencia del procesador sin necesidad de modificar el modelo, esto es gracias al uso de argumentos en el *script* de simulación.

V-B. Modelando el consumo de energía del sistema

Por defecto, *gem5* genera algunas estadísticas de la simulación en el archivo `gem5/m5out/stats.txt`. Es posible usar este sistema para evaluar distintas métricas del procesador, por ejemplo la cantidad de accesos o fallos en alguna de las memorias cache. Se puede también crear un modelo del consumo de energía. Para ello es necesario editar el *script* de la simulación. Primero será necesario definir un modelo de potencia para cada componente de interés. En este ejemplo revisaremos cómo proceder en el caso de los CPU, pero es posible también agregar modelos para las memorias u otros componentes.

En *gem5* un modelo de potencia extiende la clase `PowerModel` e implementa cuatro clases que a su vez extienden la clase `MathExprPowerModel` correspondientes con los estados de *ON*, *CLK_GATED*, *SRAM_RETENTION* y *OFF*. Cada modelo debe tener estos cuatro comportamientos, aunque es posible indicar que todos usen el mismo modelo:

```
[...]

class CpuPowerOn(MathExprPowerModel):
    def __init__(self, cpu_path, **kwargs):
        super(CpuPowerOn, self).__init__(**kwargs)
        self.dyn = "voltage * 2 * {}.ipc".format(
            cpu_path)
        self.st = "4 * temp"

class CpuPowerOff(MathExprPowerModel):
    dyn = "0"
    st = "0"

class CpuPowerModel(PowerModel):
    def __init__(self, cpu_path, **kwargs):
        super(CpuPowerModel, self).__init__(**kwargs)
        self.pm = [
            CpuPowerOn(cpu_path),       # ON
            CpuPowerOff(),              # CLK_GATED
            CpuPowerOff(),              # SRAM_RETENTION
            CpuPowerOff(),              # OFF
        ]

[...]
```

Aquí también es conveniente usar argumentos de la línea de comandos para controlar la adquisición de estas estadísticas, se agregará un argumento que permita habilitar la captura y otro que determine la tasa de adquisición:

```
def addOptions(parser):
    [...]

    parser.add_argument("--power-models", action="
        store_true")
    parser.add_argument("--stat-freq", type=float,
        default=1.0E-3)
    return parser
```

Finalmente, para agregar los modelos a la simulación e indicar la tasa de adquisición a utilizar es necesario editar el método principal del *script* de simulación:

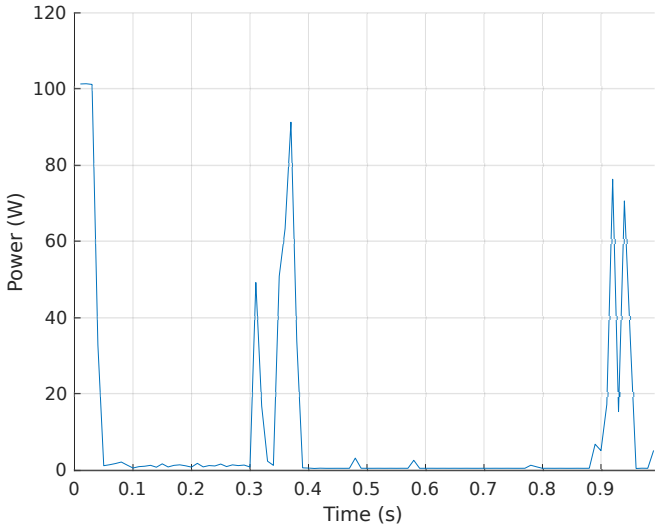
```
def main():
  [...]

  if options.power_models:
     if options.cpu_type == "atomic":
         m5.fatal("The power models require the 'timing'
               CPUs.")
     for cpu in root.system.descendants():
             if not isinstance(cpu, m5.objects.BaseCPU):
                 continue
             cpu.power_state.default_state = "ON"
             cpu.power_model = CpuPowerModel(cpu.path())

  instantiate(options)

  m5.stats.periodicStatDump(m5.ticks.fromSeconds(options
      .stat_freq))

    [...]
```

Note que para estimar modelos del consumo de potencia es necesario usar un tipo de CPU llamado *timing*. En *gem5* existen CPUs de tipo *timing* o *atomic*, que a su vez pueden usarse para construir otros modelos. En esencia, los CPU de tipo *atomic* usan accesos atómicos a memoria mientas que los CPU de tipo *timing*, como su nombre lo indica, tienen un comportamiento que emula periodos de espera y latencias. Es decir, se aproximan mejor a un comportamiento real. Por defecto, el *scipt* base utilizado toma CPUs de tipo *timing*, pero esto también se puede cambiar usando argumentos en la línea de comandos, por ejemplo, se puede usar CPUs equivalentes a los de un chip Exynos:

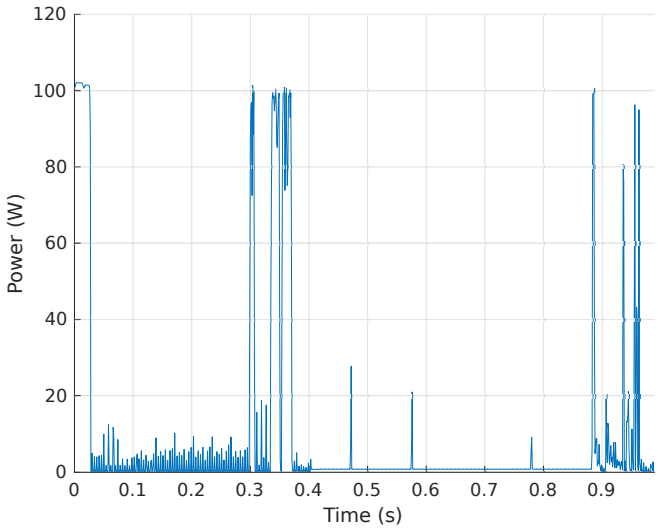
```
./build/ARM/gem5.opt \
./configs/example/arm/Hikey960.py \
--big-cpus 4 \
--little-cpus 4 \
--cpu-type=exynos \
--caches \
--dtb=./system/arm/dt/armv8_gem5_v1_big_little_4_4.dtb \
--bootloader=./common/binaries/boot.arm64 \
--kernel=./common/binaries/vmlinux.arm64 \
--disk=./common/ubuntu-18.04-arm64-docker.img \
--dvfs \
--big-cpu-clock 2362MHz 2112MHz 1805MHz 1421MHz 903MHz \
--little-cpu-clock 1402MHz 999MHz 533MHz \
--big-cpu-voltage 1.0V 0.99V 0.98V 0.97V 0.96V \
--little-cpu-voltage 0.87V 0.86V 0.85V \
--power-models \
--stat-freq 1.0E-3
```

Una vez que se analiza el contenido del archivo de estadísticas es posible identificar aquellas relacionadas con los modelos de potencia. Para este ejemplo se tendrá la potencia estática y dinámica para cada uno de los ocho núcleos. La frecuencia de volcado está dada por el argumento `--stat-freq`. Al especificar una tasa de 1.0E-2 esto equivale a calcular los modelos 100 veces por segundo simulado, mientras que una tasa de 1.0E-3 será equivalente a calcular estos datos cada milisegundo simulado. Evidentemente, incrementar la frecuencia llevará a tener un modelo más preciso, pero también incrementará el costo de la simulación de forma significativa.

Note que los modelos de potencia utilizados corresponden con el cálculo de la potencia dinámica y estática del microprocesador. En el modelo propuesto el primero de estos valores corresponde a dos veces el voltaje del núcleo multiplicado por el número de instrucciones por ciclo. Para



(a) "Timing" CPU a una tasa de 1E-2



(b) "Exynos" CPU a una tasa de 1E-3

Figura 6: Gráficas de la disipación de potencia en el CPU0 del cluster "big." En este caso solo se muestra la disipación durante el arranque del sistema simulado.

el segundo se toma cuatro veces la temperatura del procesador. Existen tres valores globales que se pueden utilizar en estos modelos: el voltaje (*voltage*), la temperatura (*temp*) y el periodo del procesador (*clock_period*). Fuera de estos, para hacer uso de otras estadísticas se debe proporcionar el nombre completo del objeto, por ejemplo la siguiente expresión va a representar el número de instrucciones por ciclo para el procesador inmediatamente asociado con el modelo:

```
"{}.ipc".format(cpu_path)
```

Este tipo de asignación dinámica permitirá poder utilizar un modelo genérico que se aplicará a todos los objetos de tipo CPU o que tienen relación con un CPU.

V-C. Aplicaciones y checkpoints

Una vez que se tiene una simulación aproximada de la arquitectura de interés, el siguiente paso será poder modificar las aplicaciones que se desea ejecutar dentro del sistema. En el ejemplo propuesto se usa Ubuntu como sistema operativo, y éste usa el sistema de archivos contenido dentro de un disco. Por ende, una forma sencilla de proceder es primero crear las aplicaciones, posteriormente compilarlas, y finalmente cargarlas en el archivo de disco.

Para crear una aplicación o programa es posible escribirla en prácticamente cualquier lenguaje de programación y después usar una serie de interpretes y compiladores que permitirán crear el código objeto para la arquitectura que deberá ejecutar este software. En este caso se partirá de una especificación en lenguaje C, que es popularmente utilizado en procesos de formación por su eficiencia y disponibilidad:

```
#include <stdio.h>

int main()
{
    printf("Hello World from gem5!\n");
    return 0;
}
```

Normalmente, el siguiente paso sería compilar el programa y ejecutarlo. Sin embargo, se debe tener en cuenta que al utilizar un compilador instalado en el ordenador de trabajo, este creará un código para la arquitectura de dicho ordenador. Por ende, es necesario utilizar un compilador cruzado o *cross-compiler* que permitirá generar un archivo ejecutable por la arquitectura de interés, en este caso ARM.

Existe una amplia variedad de compiladores para ARM[4] que pueden ser utilizados según el entorno de trabajo de la plataforma. Para el ejemplo propuesto se utilizará la herramienta `aarch64-none-linux-gnu-gcc` que permite compilar un código C para una arquitectura AArch64 con Linux. Su uso es bastante intuitivo:

```
wget https://developer.arm.com/-/media/Files/downloads/
    gnu-a/10.3-2021.07/binrel/gcc-arm-10.3-2021.07-
    x86_64-aarch64-none-linux-gnu.tar.xz
tar -xf gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-
    gnu.tar.xz
mv gcc-arm-10.3-2021.07-x86_64-aarch64-none-linux-gnu
    toolchain
./toolchain/bin/aarch64-none-linux-gnu-gcc hello.c -o
    hello
```

En seguida se cargará la aplicación en formato binario en el sistema de archivos de Ubuntu a emular. Para ello es necesario *montar* el disco y copiar el archivo. El proceso es un poco complejo, pero repetitivo. Una vez que se comprende la secuencia de pasos la tarea resulta sencilla:

```
#por simplicidad asumiremos que el archivo 'hello'
    existe en gem5/common/
cd common
mkdir disk_mnt
sudo mount -o loop,offset=65536 ubuntu-18.04-arm64-
    docker.img disk_mnt
sudo cp hello disk_mnt/home/
sudo umount disk_mnt
```

Note que en este paso se requiere tener permisos de administrador para realizar las operaciones de montado y copia al disco que se va a emular. La aplicación a ejecutar estará disponible en el directorio `/home`.

Evidentemente, si el objetivo principal de un experimento es editar el archivo binario que se ha creado, no será del todo atractivo el tener que simular todo el arranque del sistema operativo con cada modificación. Para ello se puede hacer uso de la instrucción `m5 checkpoint` dentro de la simulación. Posteriormente se puede hacer uso del

[4]https://developer.arm.com/downloads

argumento `--restore-from` para especificar la ruta del punto de restauración deseado.

```
./build/ARM/gem5.opt \
./configs/example/arm/Hikey960.py \
--big-cpus 4 \
--little-cpus 4 \
--cpu-type=timing \
--caches \
--dtb=./system/arm/dt/
    armv8_gem5_v1_big_little_4_4.dtb \
--bootloader=./common/binaries/boot.arm64 \
--kernel=./common/binaries/vmlinux.arm64 \
--disk=./common/ubuntu-18.04-arm64-docker.img \
--dvfs \
--big-cpu-clock 2362MHz 2112MHz 1805MHz 1421MHz
    903MHz \
--little-cpu-clock 1402MHz 999MHz 533MHz \
--big-cpu-voltage 1.0V 0.99V 0.98V 0.97V 0.96V \
--little-cpu-voltage 0.87V 0.86V 0.85V \
--power-models \
--stat-freq 1.0E-3 \
--restore-from=./m5out/cpt.1044789997495
```

Y con esto concluimos la demostración del uso de *gem5* para emular un SoC moderno con algunas características prácticas. Los archivos fuente generados o editados se pueden encontrar en https://github.com/CarlosAndresLARA/hikey960-gem5.

VI. Conclusión

En este artículo hemos analizado uno de los retos que se pueden encontrar en los procesos de formación de estudiantes de educación superior y posgrado. Hablamos de la dificultad por tener acceso a sistemas de evaluación y prototipado modernos que les permitan adquirir experiencia con las nuevas tecnologías utilizadas en la industria o la investigación.

Hemos descrito diferentes herramientas comúnmente empleadas para el estudio de sistemas digitales y enumerado los diversos problemas que existen con las soluciones conocidas. Llegando a la conclusión de que emular un sistema de computo requiere una reproducción de su comportamiento lógico como también la estimación de diversos indicadores sobre su operación. El simulador gem5 ofrece ambas opciones.

Esta herramienta es un software de código libre y de distribución gratuita que puede ser empleada para emular sistemas de cómputo modernos como los que se basan en procesadores ARM y RISC-V. Permite verificar la operación no solo de los programas que se han de ejecutar sobre el sistema simulado y proporciona datos adicionales que pueden ser de interés para el análisis de un chip.

No obstante, existen tres retos principales para poder considerar a gem5 como una solución de aplicación practica, en particular para el caso de latinoamérica. Primero, la complexidad del software obliga a utilizar sistemas de computo potentes que a menudo no serian disponibles para los estudiantes de recursos limitados. Segundo, gem5 solo funciona para sistemas Linux que no son ampliamente conocidos. Y finalmente, tiene una curva de aprendizaje muy lenta. Nuestro trabajo trata de abordar este último desafío.

Reconocimientos

Los autores reconocen el financiamiento de la *Agence Nationale de la Recherche (ANR)*—Francia—a través del proyecto ARCHI-SEC (ANR-19-CE39-0008).

Referencias

[1] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ser. ATEC '05. USA: USENIX Association, 2005, p. 41.

[2] A. Vladimirescu, *The Spice Book*. USA: John Wiley & Sons, Inc., 1994.

[3] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, p. 92–99, nov 2005.

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," *IEEE Micro*, vol. 26, no. 4, p. 52–60, jul 2006.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011.