Universidad Zaragoza

Trabajo Fin de Grado

Diseño de una CPU básica para la ejecución de programas BPF

Autor

Fernando Lahoz Bernad

Director

José Luis Briz Velasco

Dept. Informática e Ing. de Sistemas, Univ. Zaragoza

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2024

AGRADECIMIENTOS

En primer lugar, quiero comenzar dando las gracias a los profesores, en especial a aquellos de las asignaturas del itinerario de Ingeniería de Computadores, que me han enseñado las bases y han afianzado los conocimientos en el campo a lo largo del grado para que este proyecto haya sido posible.

Muchas gracias a Javier Resano, Pablo Ibáñez, Darío Suárez, Jesús Alastruey y Rubén Gran, de los que he aprendido todo lo necesario sobre el diseño de procesadores. También agradezco al profesor Denis Navarro su ayuda para introducirme el entorno de desarrollo de Vivado y la comunicación de componentes.

Y por último agradecer a José Luis Briz, quien ha dirigido este proyecto y propuso la idea original, dándome la oportunidad de trabajar en un campo en desarrollo, muy cerca de su equipo de investigación.

Este trabajo ha sido parcialmente financiado por el Programa de Becas de colaboración en departamentos universitarios convocadas por el Ministerio de Educación y Formación Profesional Curso académico 2023/24, por el proyecto MCIN/AEI/10.13039/501100011033 (PID2022-136454NB-C22), y por el grupo T58_23R - Gobierno de Aragón.

RESUMEN

La tecnología BPF surgió como una alternativa flexible a los filtros de red y programas de depuración basados en módulos o alteraciones al código fuente del kernel de Linux [1]. Se trata de un de una ISA pensada para ejecutar virtualmente el *bytecode* de un programa por medio de un *JIT compiler*. Su uso se ha extendido más allá del filtrado de paquetes como una forma de ganar ancho de banda y reducir la latencia introducida por el Sistema Operativo, hasta el punto de aparecer tarjetas de red capaces de ejecutar programas BPF previamente traducidos a arquitecturas ya establecidas en la industria. Esta idea hace que los programas se independicen del planificador del kernel, obteniendo un *jitter* mínimo apto para redes TSN [2].

En este contexto y en el ámbito de investigación en TSN se decide comprobar la viabilidad de un procesador que ejecute directamente el repertorio de instrucciones extendido de BPF a partir de un primer diseño para FPGA. Este proyecto comprende las fases de análisis, diseño, implementación y pruebas de un core BPF que sirva de acelerador para un procesador de propósito general en un sistema *baremetal*, permitiendo cargar programas BPF con su debido contexto, controlar el comienzo y fin de la ejecución e interactuar por medio de mapas accedidos desde memoria compartida, de la misma forma que se hace en un entorno GNU/Linux.

Este TFG se divide en dos partes. La primera incluye el desarrollo de un procesador segmentado multietapa capaz de ejecutar el repertorio de instrucciones eBPF, especificando su ruta de datos, unidades de control, componentes, interfaces de memoria y sistemas adicionales en el lenguaje de descripción de hardware VHDL, además de la ampliación de un proyecto de compilador de ensamblador eBPF. La segunda consiste en el diseño de un protocolo de comunicación con el core como periférico, planteando una distribución del espacio de memoria y los registros de control y aportando una solución original al problema de transacciones atómicas con buses de ancho reducido. Además, abarca la implementación en VHDL de un sistema de entrada y salida que implemente dicho protocolo para el procesador junto al desarrollo de una biblioteca en C que abstraiga la interacción con el periférico.

El resultado del trabajo es un SoC sintetizable para FPGA que contiene el procesador BPF acoplado a un procesador MicroBlaze [3], comunicados por un bus AXI Lite [4] accesible desde memoria.

ABSTRACT

The BPF technology emerged as a flexible alternative to network filtering and debugging based on modules or direct instrumentation of the Linux kernel source code. It is specified as an ISA, designed to virtually execute the bytecode of a program through JIT compiling. Current BPF usage reaches well beyond packet filtering, as a way to gain bandwidth and reduce the latency introduced by the Operating System, with newly developed NIC which are capable of executing BPF programs previously translated to architectures already established in the industry. This idea makes the programs independent on the kernel scheduler, minimizing *jitter* and turning them into suitable tools for TSN citebpf-tsn-building-blocks systems.

In this context, and within the scope of research in the field, this TFG test the feasibility of a processor that directly executes the BPF ISA on first a design targeting FPGA synthesis. This project encompasses the analysis, design, implementation and testing stages of a BPF core that serves as an accelerator for a general purpose processor in a *baremetal* system, allowing to load BPF programs with their proper context, to control execution start and termination, also providing interaction through BPF *map* structures accessed from shared memory, as performed in common GNU/Linux environments.

This project is divided into two parts. The first one includes the development of a multi-stage segmented processor, capable of executing the eBPF ISA, specifying in VHDL its data path, control units, components, memory interfaces and additional systems. Additionally, this part provides an extension of a eBPF assembly compiler project. The second part consists of the design of a communication protocol with the core considered a peripheral, proposing a memory organization, including control registers and providing an original solution to the problem of atomic transactions on buses with reduced datawidth. It also covers the VHDL design of an input and output system that implements this protocol for the processor, along with the development of a C library that provides an interface to interact with the peripheral.

The outcome and principal deliverable of this TFG is a synthesizable SoC for FPGA which contains a BPF processor coupled to a MicroBlaze softcore, inter-communicated by a memory-accessible AXI Lite bus.

Índice

Capítulo 1

Introducción

1.1. Motivación y contexto

Berkeley Packet Filtering (BPF) es una tecnología que permite ejecutar programas a distintos niveles dentro de la pila de protocolos del kernel de Linux. Consiste en una Arquitectura de Lenguaje Máquina (ALMA) (ISA) interpretada por un compilador Just-In-Time (JIT) con el objetivo de ofrecer una forma segura y flexible de acceder a un contexto privilegiado que, de otra forma, requeriría recompilar el kernel o confiar en un módulo [1].

En los últimos años se ha visto su potencial más allá del simple filtrado de paquetes para ganar ancho de banda y reducir la latencia introducida por el Sistema Operativo. Como consecuencia, surge el concepto *BPF Hardware Offload* que busca reducir la carga del procesador, delegando a la Tarjeta de Interfaz de Red (NIC) la ejecución el programa.

Este proyecto pretende explorar las opciones de diseño que aparecen a la hora de *desvirtualizar* la arquitectura BPF.

1.2. Objetivos y alcance

1.2.1. Objetivos generales

Mediante la realización de este Trabajo de Fin de Grado (TFG) se persigue, por una parte, la consolidación y ampliación de conocimientos y competencias relacionados con arquitectura y organización de computadores y sistemas empotrados adquiridos durante la titulación. Por otra, se busca un acercamiento a la realidad y práctica investigadora e industrial en el campo.

1.2.2. Objetivos específicos

– Estudio de la tecnología BPF, interfaces, aplicaciones e ISA.

– Diseño de un núcleo BPF básico.

– Diseño de un núcleo BPF integrable en un sistema empotrado.

1.2.3. Alcance

– Implementación en VHDL de un procesador BPF básico autónomo.

– Implementación en VHDL de un procesador BPF integrado con un procesador *softcore* en un sistema *baremetal* sobre una Field-Programmable Gate Array (FPGA) Xilinx Kintek 7©.

El código fuente desarrollado para este proyecto está a disponible en `https://github.com/uz-gaz/bpf_1`.

1.3. Contribuciones

La consecución de los objetivos anteriores ha generado las siguientes contribuciones:

– Diseño original de un procesador BPF

– Solución original al diseño del subsistema de memoria de un procesador BPF que soporta primitivas de acceso atómico.

– Solución al problema de compartición de memoria con un procesador de propósito general sobre una FPGA.

– Modificación y corrección de un programa ensamblador para soportar el repertorio de instrucciones BPF más reciente.

Este TFG contribuye a las metas 9.2 / 9.2.1; 9.4 / 9.4.1; 9.5 / 9.5.2 de los Objetivos de Desarrollo Sostenible.

1.4. Metodología y entorno de trabajo

El código de este proyecto se ha desarrollado haciendo uso del editor Visual Studio Code [5], con ayuda de la extensión Modern VHDL [6], con Git [7] como herramienta de control de versiones.

Para la simulación del código VHDL del procesador se ha utilizado GHDL [8]. GTKWave [9] se ha empleado para visualizar los cronogramas con la evolución de las señales en cada simulación. Al ser programas de línea de comandos el proceso de simulación se ha automatizado con scripts de Bash [10]. Durante el tiempo que se ha

trabajado en un sistema operativo Windows se ha utilizado MSYS2 [11] para emular un entorno similar a Linux.

Con el paquete de software de Vivado 2024.1 [12] se ha podido evaluar el procesador, integrarlo con un *softcore*, y desarrollar programas en C que permiten verificar su funcionamiento.

Para elaborar esta memoria se ha utilizado Overleaf [13] como editor colaborativo de LaTeX. Los diagramas de este documento son de elaboración propia, creados con la aplicación web Diagrams.net [14].

La metodología y entorno específico de las partes experimentales se describen en las Secs. 3.3, 4.3 y 4.4.

1.5. Planificación

Este proyecto se ha planificado en dos partes: la primera dedicada al diseño e implementación del procesador BPF y la segunda enfocada en su integración como periférico en un sistema *baremetal*. Las partes están pensadas para dedicarles un mes de desarrollo a cada una. El trabajo es progresivo, con fases de análisis, diseño, implementación y pruebas en cada uno de los sistemas desarrollados, y concluye con la elaboración de esta memoria. En el Anexo A aparece el diagrama de Gantt que muestra la distribución del trabajo a lo largo de las semanas.

1.6. Estructura del documento

Esta memoria de TFG se estructura como sigue. El Cap.2 introduce los conceptos necesarios para el seguimientos de la memoria y sitúa el proyecto dentro del estado actual de la tecnología BPF. El Cap. 3 está enfocado en el diseño de un procesador BPF, explicando el funcionamiento interno de la ruta de datos implementada y los componentes que la conforman. El Cap. 4 trata sobre la creación de un acelerador a partir de este procesador BPF, la especificación de un protocolo de interacción basado en registros y su conexión con un procesador de propósito general dentro de un SoC sintetizable en FPGA. Finalmente, el Cap. 5 recoge conclusiones y líneas futuras.

Capítulo 2

Fundamentos

2.1. Ruta de datos segmentada

El diseño de este procesador se basa en la arquitectura segmentada en cinco etapas original del MIPS. Se considera necesario para el seguimiento de esta memoria entender los siguientes conceptos básicos, para los que remitimos p.ej. a [15] y [16]:

- Riesgos de datos, estructurales y de control.
- Anticipación de operandos.
- Consolidación de instrucciones.

2.2. BPF

2.2.1. Programas BPF y usos

Los programas del sistema Berkeley Packet Filtering (BPF) son objetos binarios cargados por un usuario con privilegios limitados para actuar en distintos puntos de la pila de protocolos de red del kernel, llamados *hooks*. Dependiendo del tipo de programa especificado, este se puede ejecutar al recibir un paquete en un *socket* – SOCKET_FILTER–, al aplicar disciplinas de colas de Linux (*queue discplines*, `qdisc`)[1] – SCHED_CLS– o antes incluso de llegar al dominio del kernel – XDP–. La gran variedad de *hooks* permiten utilizar programas BPF no solo para filtrado de paquetes, sino como balanceador de carga, protección contra ataques de denegación de servicio, o acelerador de servicios [17]. También es posible usarlos para depurar funcionalidades del kernel.

[1] Es el momento en el que se cargan las *qdiscs* que se hayan configurado mediante el comando `tc` de Linux

2.2.2. Mapas

Los mapas representan regiones de memoria compartida entre los programas BPF y de usuario. Se accede a sus elementos mediante funciones auxiliares, explicadas con mayor profundidad en la Sec. 3.2.5. Los mapas también pueden ser de varios tipos, según se comporten como *arrays* de elementos contiguos, tablas *hash*, o estructuras más complicadas, como *array* de *sockets* u otros mapas.

Los mapas perduran más que los programas. Si el usuario elimina un mapa mientras hay un programa cargado que lo referencia, el kernel no lo elimina por completo hasta que el programa, a su vez, se descargue. El binario de un programa que use un mapa precisa ser modificado en tiempo de carga para reubicar las referencias al mapa.

El usuario maneja programas y mapas mediante comandos, utilizando la llamada al sistema `bpf()` [18].

Remitimos a la documentación oficial de BPF para Linux [19][20] para ampliar información sobre los diferentes tipos de programas y mapas.

2.2.3. Hardware Offload

El objetivo del *Hardware Offload* es incrementar el rendimiento, eximiendo al kernel, y en consecuencia al procesador principal, de ejecutar programas BPF en los *hooks* más bajos. Las NIC con soporte *offload* actúan como aceleradores. Permiten la carga de programas BPF a través de recursos como XDP y `qdisc` para su ejecución local, con lo que escapan a la planificación del kernel. Los mapas también pueden ser configurados como locales a las NIC para permitir el acceso rápido desde estos programas. Con *hardware offload* es posible programar moldeadores de tráfico (*traffic shapers)* y encaminadores flexibles y ágiles de modificar, con una latencia y *jitter* mínimos [2].

2.2.4. Arquitectura de Lenguaje Máquina

eBPF es el sucesor de lo que actualmente se conoce como *Classic BPF*[2]. Originalmente, los programas BPF utilizaban dos registros y un conjunto de instrucciones de 32 bits. La versión extendida amplía el número de registros a 10 de propósito general y un *Frame Pointer (FP)* de solo lectura. Las instrucciones están codificadas en 64 bits, que incluyen dos registros – fuente y destino –, un *offset* y un inmediato, además de la clase de la instrucción – salto, aritmética o memoria – y un código de operación (Fig. 2.1). El Anexo B contiene una descripción en pseudocódigo de las instrucciones consideradas en este proyecto. La información detallada sobre la ISA se encuentra en [21].

[2]Siguiendo una práctica habitual, este documento utiliza BPF para referirse a la versión extendida. Para la versión clásica se recomienda utilizar el acrónimo específico cBPF

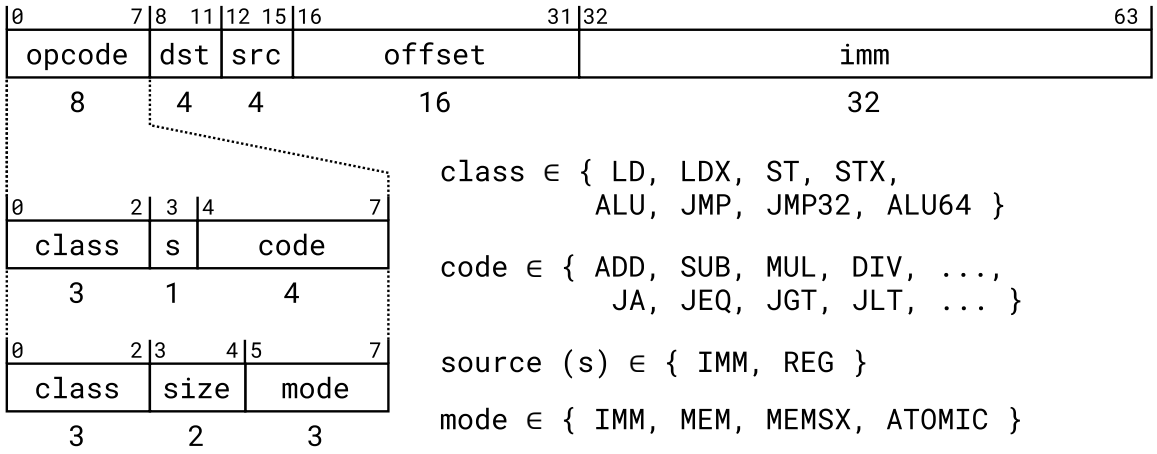


Figura 2.1: Formato de una instrucción BPF

2.3. Metodología de depuración y *testing*

Los *testbenchs* son el principal método de depuración en VHDL. Son entidades específicas para simulación, que encapsulan a la entidad a probar y controlan el valor de las señales con eventos de tiempo. Uno de los *testbenchs* del proyecto está presente en el Anexo G.2 a modo de ejemplo.

VHDL permite automatizar las comprobaciones, lo que es especialmente útil en pruebas de caja negra. Cuando lo que se pretende verificar es el comportamiento interno, se trabaja con un visor de ondas para observar el valor de las señales en cada instante de tiempo. El Anexo G.3 incluye ejemplos de visualización de señales.

Los tests desarrollados para este proyecto se describen junto a sus objetivos en las Sec. 3.3 y 4.3.

2.4. Trabajos relacionados

El *Hardware Offload* es una práctica en auge. Empresas como *Corigine* y *Netronome* tienen líneas de desarrollo de *SmartNIC*, cuyos drivers para Network Flow Processor (NFP) ya han sido integrados en el kernel de Linux. Sus procesadores no implementan la arquitectura BPF, sino que se basan en arquitecturas comerciales de uso extendido. En su lugar cargan los programas ya traducidos en la memoria de instrucciones. Este proyecto se basa en los documentos técnicos publicados por *Corigine* [22] para planificar la disposición de la RAM dentro del periférico. Estos documentos han sido necesarios para entender el acceso a mapas locales a la NIC.

La ruta de datos del procesador diseñada en este trabajo se inspira en los *engines* de la plataforma *eBPFlow* [23]. Consiste en una implementación sobre una *NetFPGA* – una FPGA especial para desarrollar dispositivos de red– de un NFP especializado para

programas BPF, con un procesador con ruta de datos segmentada en cinco etapas. Cabe mencionar que esa implementación necesita de software propio tanto para compilar los programas BPF como para cargarlos en la FPGA. Este proyecto intenta acercarse a esa idea de desarrollar un procesador específico para la ISA BPF, pero más acorde con la forma de interacción propia de Linux.

Capítulo 3

Diseño de un procesador BPF básico

El siguiente capítulo resume los detalles de diseño de un procesador BPF. El núcleo (core) es capaz de ejecutar el repertorio completo de instrucciones eBPF, a excepción de aquellas de legado, las codificadas en 128 bits para acceso directo a mapas, y las llamadas a función (esto se justifica en la Sec. 3.2.5). También cuenta con la capacidad de generar excepciones y gestionar señales externas para detener y reanudar la ejecución.

3.1. Ruta de datos

La Fig. 3.1 contiene un diagrama completo de la ruta de datos diseñada, sobre la que se recomienda apoyar la lectura. En el Anexo C se ha añadido la descripción RTL elaborada con las acciones tomadas por cada instrucción.

3.1.1. Control de PC

El espacio de direccionamiento de las instrucciones BPF es de 64 bits. En cada ciclo, PC apunta a la siguiente instrucción dentro de la memoria de instrucciones y se incrementa en 1 para avanzar la ejecución. La palabra leída se carga en el registro IR y se decodifica al siguiente ciclo.

La ISA BPF especifica salto no retardado. Las instrucciones de salto obtienen el offset y el inmediato en la etapa de decodificación (ID), y calculan el valor potencial de PC para salto tomado sumando uno de los dos a $PC + 1$. El valor decodificado debe entenderse como el número de instrucciones saltadas: `ja 1` evita la ejecución de la siguiente instrucción, `ja 0` es en efecto una NOP[1], y `ja -1` es un bucle infinito. Para saltos incondicionales de 32 bits (`jal`) se utiliza el inmediato. Para el resto se emplea el offset.

[1]Se entiende como NOP una instrucción sin efectos; no escribe en el banco de registros ni tiene efectos colaterales.

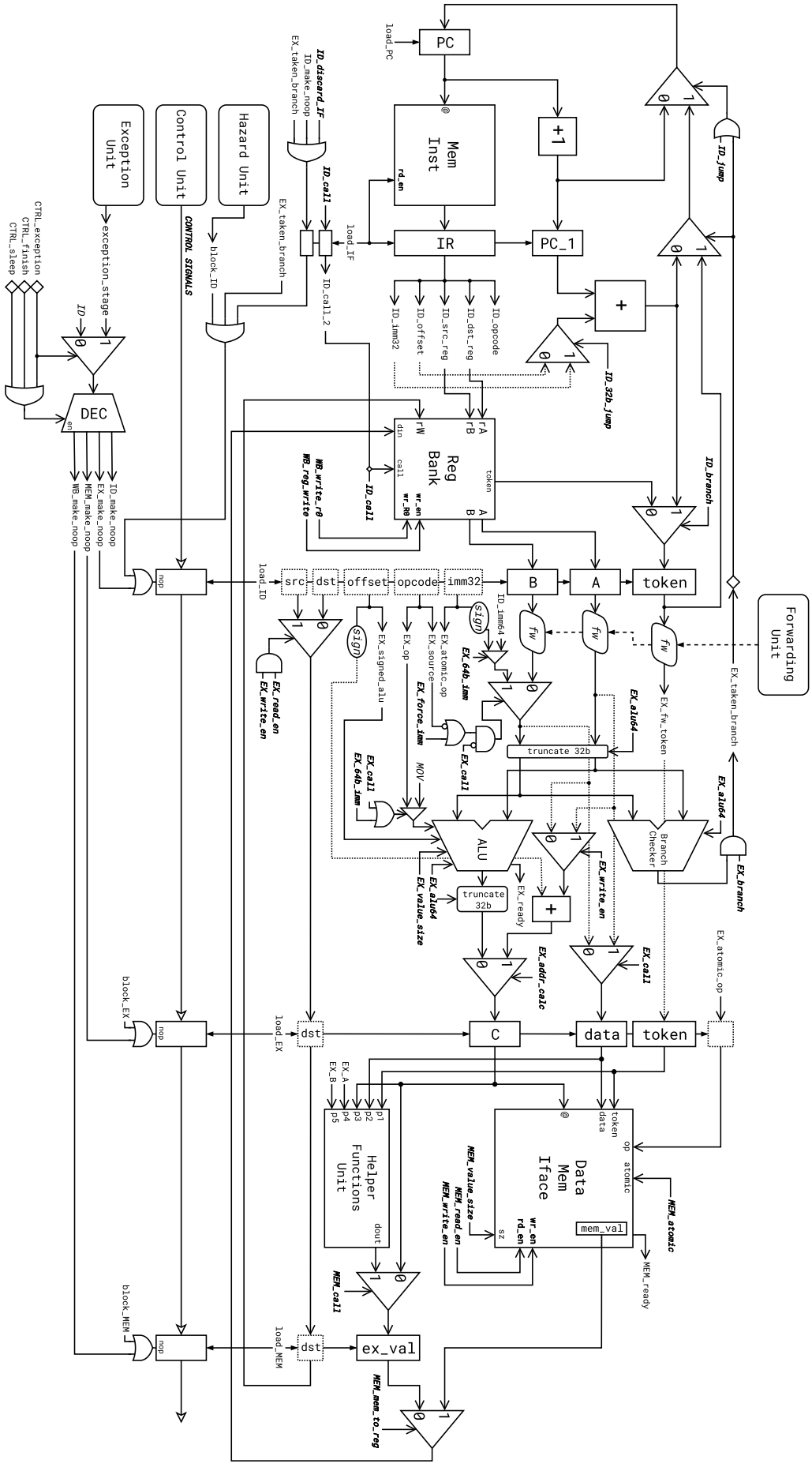


Figura 3.1: Esquema completo de la ruta de datos diseñada

La dirección definitiva de salto no se conoce hasta la etapa ID, por lo que es necesario descartar la que actualmente está siendo leída en IF. Eso se consigue activando y guardando la señal de control `ID_discard_IF` en el banco de etapa ID para que la aplique al siguiente ciclo y anule las señales de control. Los saltos condicionales se evalúan (tomados o no) en la etapa EX, por lo que en caso de cumplir la condición de salto, además de descartar IF, deben descartar la instrucción en ID (señal `EX_taken_branch`).

El registro IR no contiene una instrucción válida al comenzar la ejecución del programa. Para evitar que se ejecute código con comportamiento indefinido la señal `ID_discard` del banco de etapa ID se resetea a activa.

3.1.2. Búsqueda de operandos

Las instrucciones pueden utilizar como operandos los valores de los registros `A`, `B` o `token`[2]. Para evitar paradas por dependencias productor-consumidor, tanto la ALU como el resto de bloques que los utilizan pueden leer el valor anticipado desde los bancos de etapa MEM y WB, si así lo requieren. El control de los multiplexores de anticipación lo controla la Forwarding Unit (FU). Aún así, hay casos como el de una productora en etapa MEM seguida de consumidora, que obligan a detener la instrucción en etapa ID. Ese tipo de acciones las controla una Hazard Unit (HU). En los Anexos D.3 y D.2 se detalla el funcionamiento de ambas unidades.

Las operaciones aritméticas y los saltos condicionales leen siempre el registro destino desde `A`, pero el valor fuente puede venir tanto de `B` como del inmediato (extendido de signo a 64 bits). Estas instrucciones tienen versiones de 32 bits. En estos casos se truncan los 32 bits más significativos de las entradas de la ALU y el Branch Checker (BC), así como la salida de la ALU.

Las instrucciones de acceso a memoria calculan la dirección sumando el offset (con extensión de signo) al registro fuente, en el caso de un *store* u operaciones aritméticas, o al registro destino, en el caso de un *load*. El dato a escribir en memoria se puede obtener del registro fuente o del inmediato, según la instrucción sea `stx` o `st`.

Existe además la instrucción `ld64`, capaz de cargar un inmediato de 64 bits codificado en dos instrucciones. Para implementarla se ha reutilizado el camino de la ALU inyectando una operación `mov`.

Las funciones auxiliares se ejecutan en fase MEM, pero también reciben parte de sus operandos del banco de etapa EX, con el fin de reutilizar recursos ya disponibles. Lo hace con una doble fase de lectura:

[2]El `token` es un tercer parámetro implícito en la instrucción atómica `cmpxchg`. Normalmente es `r0`. También sirve como valor de salida del procesador, y para agilizar la lectura de los parámetros de funciones auxiliares (en cuyo caso es `r1`).

1. La primera vez carga `r1`, `r2` y `r3` en `token`(EX), `A` y `B` respectivamente. En la etapa EX se encarga de trasladar esos valores a `token`(MEM), `data` y `C`, obtenidos mediante anticipación si fuera necesario.

2. La segunda vez carga `r4`, `r5` en `A` y `B`. Como la FU no asegura que el valor anticipado permanezca intacto durante más de un ciclo, los parámetros `p4` y `p5` siempre son leídos desde el banco de registros[3] (asegurado por la HU).

En la Tab. C.5 hay una descripción RTL de este mismo proceso que puede facilitar la comprensión.

3.1.3. Escritura del resultado

El resultado generado por las instrucciones se guarda en el banco de registros en la etapa WB. El valor a escribir se obtiene de la salida de la interfaz de memoria, de la Helper Function Unit (HFU) o del dato generado en EX, dependiendo de las señales de control.

Los componentes de la RAM incluyen registros internos para optimizar el tiempo de ciclo. Por este motivo, el dato leído de memoria se almacena en un registro interno de la interfaz de memoria de datos (`mem_val`) en lugar de en el banco de etapa WB. Dicho interfaz asegura que el valor en `mem_val` está disponible al ciclo siguiente, junto con el resto de valores de WB. Si el dato leído de memoria se guardase en el banco de etapa se generaría una latencia adicional de un ciclo.

Por lo general, el registro escrito corresponde al registro codificado como destino. Sin embargo, las operaciones atómicas que tienen activado el flag `FETCH` escriben su resultado en el registro fuente. Por ello, se multiplexa el registro destino en etapa EX. Las funciones auxiliares y la instrucción `cmpxchg` escriben en `r0` de forma implícita, por lo que se ha habilitado en el banco de registros una señal específica para escribir en `r0`.

3.1.4. Control de los bancos de etapa

Una instrucción en una etapa concreta consiste en los datos que transporta y las señales de control que la hacen efectiva en esa etapa y las siguientes. Para que la información avance, los bancos de etapa tienen habitualmente su señal *load* activa, salvo en ciertas circunstancias.

Por lo general, las etapas diseñadas se ejecutan en un ciclo (i.e., la instrucción permanece un ciclo en esa etapa y progresa a la siguiente). Sin embargo, ha sido preciso

[3]Si una función auxiliar requiere más de tres parámetros siempre se pueden preparar `p4-p5` antes que `p1-p3` y se elimina el ciclo de detención.

diseñar etapas multiciclo para implementar algunas operaciones que reutilizan componentes de una etapa dada. Dado que diseñamos una microarquitectura en orden, una etapa multiciclo supone detener las instrucciones de las etapas anteriores. Por ello, las unidades funcionales de las etapas multiciclo presentan una señal de salida *ready* que detiene las instrucciones siguientes si está inactiva. La señal se activa de nuevo en el último ciclo de uso de la unidad funcional que corresponda.

Detener una etapa implica lo siguiente:

1. Los bancos de las etapas anteriores mantienen sus valores (se inhibe su escritura).
2. Las etapas posteriores avanzan de la forma natural.
3. En consecuencia, el banco de entrada a la etapa bloqueante se mantiene mientras que el de salida se escribe con las señales de control anuladas.

Si coincidiera que hay dos etapas detenidas a la vez, lo anterior solo se aplica a la más avanzada (ver Fig. 3.2).

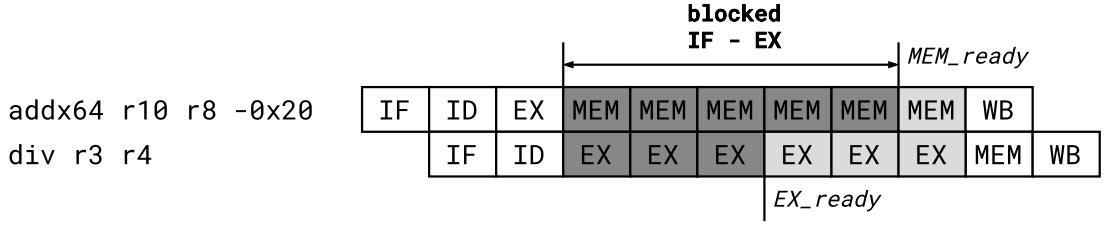
3.1.5. Gestión de excepciones y detención del procesador

Los programas BPF no generan excepciones. La ISA define la división entera de modo que devuelve un cero si el denominador (registro fuente) es cero, sin producir excepción. El verificador de Linux comprueba durante la carga del programa los accesos a direcciones no permitidas. No obstante, la ejecución de código con comportamiento indefinido puede resultar en un estado irrecuperable del procesador. Para evitarlo, se decidió añadir una Exception Unit (EU) que interactúe con el sistema de detención en los siguientes casos:

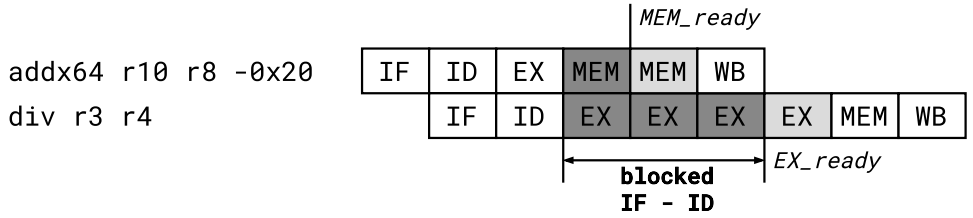
1. Decodificación de una instrucción ilegal o no implementada.
2. Lecturas y escrituras en registros inexistentes, o escrituras `r10` (solo lectura).
3. Acceso a zonas de memoria restringidas.
4. Llamadas a funciones auxiliares inexistentes.

El funcionamiento de la EU está descrito con más detalle en el Anexo D.4.

La detención del procesador puede ocurrir por excepción, señal externa para dormir al procesador o ejecución de la instrucción `exit`. Cuando se da una de estas situaciones se elige la etapa bloqueante, y se activa una señal que detiene el avance de todas las anteriores, permitiendo consolidar a las posteriores. Se toma como etapa bloqueante bien la que ha generado la excepción, o bien la etapa ID en el caso de fin de programa



(a) La etapa MEM dura más que la etapa EX



(b) La etapa EX dura más que la etapa MEM

Figura 3.2: Situaciones con etapas multiciclo ejecutadas simultáneamente

o procesador dormido. Si una etapa posterior genera excepción mientras el procesador está en proceso de detención, se convierte en la nueva etapa bloqueante, reemplazando a la anterior.

Consolidar todas las instrucciones anteriores a la que ha generado excepción tiene como objetivo establecer un comportamiento bien definido (estado en orden del procesador) frente a excepciones. Además, permite que la instrucción `exit` solo tenga que llegar a etapa ID para actuar. Solo cuando han consolidado todas las instrucciones que debían, se puede activar la señal de salida que indica la excepción, fin de programa o que el procesador está dormido. En ese momento, el procesador puede reiniciar la ejecución, por medio de la señal de *reset*, o reanudarla donde la dejó, en caso de estar dormido.

3.2. Descripción de componentes

3.2.1. Banco de registros

Este componente consiste en un *array* de registros de 64 bits con dos puertos de lectura (`A` y `B`), y uno de escritura (`W`). Se ha diseñado específicamente para adaptarse a las necesidades de la ruta de datos BPF. Así, permite en todo momento leer el registro `r0` como *token* y dato de salida del procesador, activa la escritura en `r0` a través de una señal dedicada (`wr_R0`), y presenta dos entradas adicionales para extraer los parámetros de las funciones auxiliares de acuerdo al protocolo explicado en la Sec. 3.1.2.

La lectura de `A` y `B` es asíncrona, mientras que la escritura es síncrona con el flanco de bajada del reloj. Todos los registros del procesador se escriben en flanco de subida,

a excepción del banco de registros que lo hace en flanco de bajada. De esta forma, la operación de escritura de la etapa WB y la lectura de operandos de la etapa ID se pueden llevar a cabo en el mismo ciclo. De otra forma, sería necesario un banco de etapa adicional desde el que anticipar el dato de posibles consumidoras.

3.2.2. Verificador de saltos (Branch Checker)

Este bloque se encarga de comparar los operandos `A` y `B` para determinar si un salto es tomado o no. BPF cuenta con 9 operaciones de comparación, que se pueden implementar reutilizando la salida de tres comparadores con las operaciones $=$, $<$ y $\&$ (Anexo E.1). Se calculan los resultados de todas las comparaciones con lógica adicional, y al final se elige la que corresponde según el código de operación.

3.2.3. Unidad Aritmética Lógica (ALU)

BPF tiene 17 tipos de operaciones aritméticas. Cada una de ellas se calcula en paralelo a partir de los operandos `A` y `B` y el resultado final es multiplexado (como en el BC). Las funciones y operadores aritméticos de biblioteca en VHDL suelen ser aptos para su síntesis en FPGA. No obstante, las operaciones división y módulo son más complejas y requieren utilizar un componente dedicado.

Se ha hecho uso de un divisor multiciclo de Xilinx para números naturales de 64 bits, generado a partir de un Divider Generator LogiCORE™ IP [4] . El mismo divisor genera como resultados el cociente y el resto, y se puede utilizar para calcular divisiones de naturales (`div, mod, div32, mod32`) o enteros (`sdiv, smod, sdiv32, smod32`). El Anexo E.2 describe el trabajo que ha requerido su integración en la ALU.

3.2.4. Interfaz de memoria de datos

El objetivo de este componente es controlar las operaciones atómicas y de acceso a memoria, y abstraer a la ruta de datos de los dispositivos de memoria empleados. También comprueba la dirección de memoria introducida y notifica excepciones a la EU si corresponde a un espacio de memoria inaccesible.

La memoria empleada es BRAM, generada a partir de un Block Memory Generator LogiCORE™ IP, que permite lecturas y escrituras en un ciclo (a frencuencias habituales de FPGA, 50-400 MHz), por lo que loads y stores tienen latencia 1 [5]. Las operaciones atómicas se implementan en dos fases: leer y modificar-escribir. Una máquina de estados

[4] El entorno de Vivado permite incluir bloques combinacionales prediseñados y configurables – *Intelectual Properties (IP)* – dentro del código VHDL.

[5] Cuando se accede a memoria compartida la latencia total de la instrucción puede variar por el tiempo que se tarda en adquirir control del bus de memoria.

se encarga de controlar los ciclos de lectura y sobre-escritura y ganar acceso al bus de memoria. En el proceso intervienen selectores de bytes, máscaras de escritura y multiplexación, que se explican con más detalle en el Anexo E.3.

3.2.5. Unidad de funciones auxiliares (HFU)

La ISA BPF contempla dos tipos de llamadas a función: de usuario y auxiliares. Las funciones de usuario son las habituales, las que se incluyen directamente en el código del programa. Las funciones auxiliares son un conjunto limitado de operaciones ajenas al código (ejecutadas por el kernel, módulos del procesador, aceleradores...) que permiten ampliar las funcionalidades del conjunto de instrucciones de BPF.

La HFU es la unidad encargada de ejecutar las funciones auxiliares o controlar su invocación. Está diseñada como una máquina de estados que avanza en paralelo junto con las etapas del procesador. De este modo, solo hay que modificar la HFU para añadir soporte a nuevas funciones, sin tocar la ruta de datos. En esta versión solo se implementa la función `bpf_map_lookup_elem`, que es la mínima utilidad necesaria para interactuar con los mapas (la gestión de mapas se explica en la Sec. 4.2.4).

La HFU permite consultar el número de operandos de una función a partir de su id y comunica una excepción a la EU si la función no está implementada. También permite generar errores de ejecución. Obtiene sus operandos en la etapa MEM, pero también puede ejecutar acciones en EX siempre que no los necesite.

La versión final de este procesador no implementa llamadas a funciones de usuario, porque ni el verificador del kernel las permite ni el *backend* LLVM las compila correctamente[6]. Versiones más actuales del kernel permiten su uso bajo determinadas condiciones [24], por lo que se ha considerado incluirlas en posibles proyectos futuros con este procesador. En el Anexo F se detallan las consideraciones necesarias para añadirlas a la ruta de datos.

3.3. Metodología de pruebas

Los componentes de la ruta de datos se traducen en entidades dentro del código VHDL. Para cada una de las entidades se ha elaborado un *testbench* con varios tests unitarios. El procesador en su conjunto se ha evaluado con tests de integración.

Los tests unitarios son pruebas de caja negra que comprueban automáticamente la corrección de las salidas generadas por las combinaciones de entradas más significativas.

[6]El código fuente de un programa BPF suele requerir marcar las funciones como *inline* para poder compilar.

Los tests de integración consisten en un conjunto de programas BPF precargados en una memoria de instrucciones simulada para ser ejecutados por el procesador en conjunto. Son pruebas de caja blanca, en los que se verifica manualmente el valor de las señales en cada ciclo de ejecución. A continuación, se detalla los programas de prueba elaborados:

- **Test de decodificación:** Su objetivo es comprobar que el procesador reconoce correctamente todas las instrucciones del repertorio y genera y propaga las señales de control correspondientes a cada una, sin generar excepciones (Anexo G.1).
- **Test de memoria:** Verifica el comportamiento de todas las operaciones de memoria y atómicas, incluida la detención de etapas multiciclo y la anticipación de operandos en casos problemáticos (Anexo G.2).
- **Test de ALU:** Traslada las pruebas unitarias de la ALU a código BPF a un entorno con anticipación. Verifica exhaustivamente las operaciones de división y módulo ante las señales externas `reset` y `sleep` (Anexo G.3).
- **Test de control:** Contiene todas las instrucciones de salto para evaluar el control de PC, el descarte de instrucciones y el fin de ejecución (también por excepciones) (Anexo G.4).
- **Test de HFU:** Asume una HFU de pruebas, con funciones con diferente número de operandos para comprobar detenciones, anticipación y generación de errores (Anexo G.5).

Componentes como el divisor y las BRAM no están disponibles para simulación fuera del entorno de Vivado, por lo que ha sido necesario crear entidades (no sintetizables[7]) que emulen su comportamiento.

3.3.1. Ensamblador

Generalmente, los programas BPF se escriben en C, haciendo uso de la biblioteca libbpf [25] para facilitar al programador escribir código de usuario y BPF que interactúen entre sí de forma coherente. Para los programas de prueba interesa poder escribir las instrucciones específicas y obtener su traducción en binario. Los compiladores ofrecen intrínsecos, pero añaden instrucciones adicionales (si no se optimiza), pueden alteran el orden (con optimización) y resultan poco prácticos para escribir un programa completo.

[7] En VHDL es muy sencillo crear de memorias RAM y divisores para simulación, no así para síntesis en FPGA.

La solución es utilizar un compilador de lenguaje ensamblador. A falta de ensambladores consolidados se utilizó un proyecto de código libre disponible en github [26], pero presentaba *bugs* y no abarca el repertorio de instrucciones más reciente, por lo que ha sido necesario parchearlo. El resultado es un ensamblador capaz de traducir a binario el repertorio completo presente en el Anexo B.

Capítulo 4

Integración de un procesador BPF en un sistema *baremetal*

Este capítulo trata las consideraciones tomadas para interactuar con el core como periférico acelerador desde un procesador de propósito general. El resultado es un SoC sintetizable para FPGA que contiene el procesador BPF acoplado a un procesador MicroBlaze, comunicados por un bus AXI Lite accesible desde memoria.

4.1. Procesador BPF como periférico

4.1.1. MicroBlaze y protocolo AXI

MicroBlaze es un microprocesador *software* (*softcore*) sintetizable para FPGA [3]. En la configuración elegida, su interfaz de memoria consiste en dos buses de acceso a memoria local, que usa para alojar y ejecutar programas compilados, y un interfaz AXI [4] primario[1] con el que es capaz de acceder a otros chips de memoria y comunicarse con periféricos mapeados en el espacio de direcciones. Por lo tanto, conectar el core BPF con el MicroBlaze requiere añadir una interfaz AXI secundaria.

Los buses AXI funcionan con un protocolo semisíncrono, en el que ambos dispositivos utilizan la misma frecuencia de reloj. Las transacciones implican un envío de datos en ambas direcciones (primario pregunta y secundario responde). Cada envío requiere que el emisor notifique la validez de los datos y que el receptor informe si está listo para recibirlos (el orden de señalización no se tiene en cuenta).

En particular, el bus de memoria para este MicroBlaze es de 32 bits. Las direcciones son truncadas a la dirección múltiplo de 4 más baja (se lee toda la palabra) y permite escrituras a nivel de byte mediante una máscara. Implementaciones más recientes permiten usar un bus de 64 bits. No obstante, se ha elegido el de 32 bits por

[1]Los buses AXI utilizan terminología maestro-esclavo para referirse a los nodos. En este documento se ha optado por los calificativos primario y secundario para diferenciar nodo activo y pasivo respectivamente.

compatibilidad con la mayoría de placas de desarrollo con FPGA[2]. En la Sec. 4.2.3 se explican los problemas que conlleva esa decisión.

4.1.2. Módulo de entrada-salida y memoria

El procesador BPF es solo una parte de un periférico que lo integra junto a un IOMM. Este IOMM controla los chips de memoria empleados por el procesador, una unidad de mapas y la interfaz de conexión con el exterior (en este caso el bus AXI). Se comunica con el core BPF a través interfaces (como la vista en la Sec. 3.2.4) que abstraen los protocolos específicos. Gracias a la existencia de este módulo el mismo procesador puede ser utilizado en combinación con otro IOMM, permitiendo que el periférico utilice un bus diferente, tenga más memoria o soporte más de un core.

La Fig. 4.1 muestra el esquema del IOMM desarrollado. Este cuenta con varios módulos de memoria. Cada uno de ellos es un chip diferente de BRAM, de manera que el procesador pueda leer instrucciones a la vez que realiza operaciones de memoria. Con esto también se logra que el acceso a memoria no compartida no precise arbitraje.

La memoria de instrucciones tiene capacidad para 4096 instrucciones (32 KB) porque es el tamaño máximo para cualquier programa BPF en Linux.

La memoria no compartida contiene el espacio de pila del programa – 512 Bytes, es el tamaño de pila de los programas BPF en Linux – y una región de 2 KB adicionales para el contexto. Por ejemplo, un programa XDP accede al paquete con un puntero recibido a través de r1. Esta región extra sirve para albergar dicho paquete[3].

El tamaño dedicado a la memoria compartida es de 28 KB, para completar un espacio de memoria en el dispositivo de 64 KB (en la Sec. 4.2 se muestra su distribución).

4.2. Interacción con el procesador

El MicroBlaze puede interactuar con el core BPF por medio de lecturas y escrituras a las direcciones de memoria donde se encuentra mapeado este último. La Fig. 4.2 muestra la distribución escogida para las zonas de memoria y las direcciones en las que se ha decidido situar los registros de control. A las direcciones mostradas se les debe sumar la dirección base del periférico, que varía dependiendo de la configuración del bus AXI. El Anexo E.5 detalla el diseño del controlador AXI implementado en este proyecto.

[2]Algunas placas incluyen un microprocesador de propósito general cuyo uso se prefiere al de un *softcore* y cuyo bus AXI no puede ser configurado, como es el caso de los SoC de la serie Zynq.

[3]El contexto es dependiente del tipo de programa BPF y no está limitado a paquetes

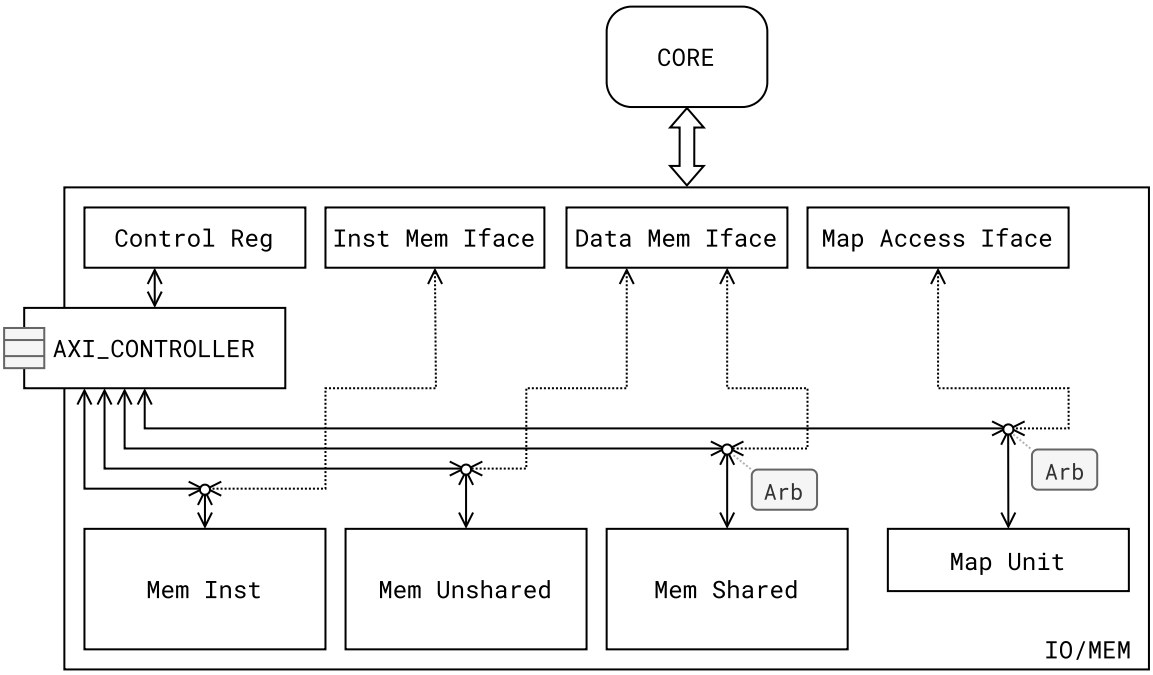


Figura 4.1: Esquema simplificado del módulo de entrada-salida y memoria del periférico

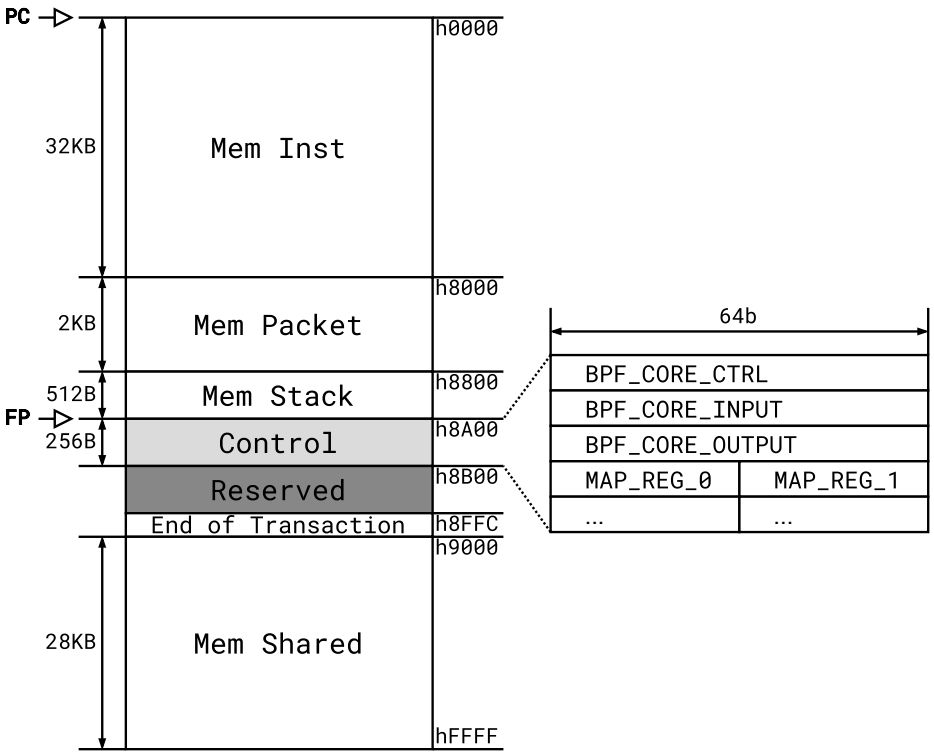


Figura 4.2: Espacio de direcciones del periférico

4.2.1. Registros de control

Los registros de control del procesador permiten controlar su arranque, parada y reinicio, además de poder cargar valores en el banco de registros y leer la salida del programa. Las señales se explican en detalle en la Tab. 4.1.

Tabla 4.1: Descripción de los bits dentro del registro de control `BPF_CORE_CTRL`

Bit	**Señal**	**Acceso**	**Descripción**	**Valor en reset**
3:0	`reg_dst`	r/w	Número de registro sobre el que escribir. Valores mayores a 10 no tienen ningún efecto.	0
4	`reg_write`	r/w	Cuando esta señal está activa el registro `reg_dst` queda escrito con el valor presente en `BPF_CORE_INPUT`.	0
5	`sleep`	r/w	Cuando esta señal está activa el procesador inicia la detención del pipeline. Cuando es desactivada la ejecución se retoma con normalidad desde la instrucción en la que se quedó detenido. **Advertencia:** No se debe considerar que el procesador está detenido con solo activar la señal. Para ello hay que comprobar el estado de la señal `sleeping`.	1
6	`sleeping`	r	El procesador activa esta señal para indicar que el pipeline ha quedado detenido. Siempre está desactivada en caso de que `sleep` también lo esté. **Advertencia:** El programa en curso puede terminar o generar excepciones mientras el procesador está en proceso de detención, en cuyo caso esta señal no se activa.	0
7	`exception`	r	El procesador activa esta señal para indicar que ha ocurrido una excepción. En ese caso, `BPF_CORE_OUTPUT` contiene el número de instrucción que provocó la excepción.	0
8	`finish`	r	El procesador activa esta señal para indicar que el programa ha finalizado satisfactoriamente. En ese caso, `BPF_CORE_OUTPUT` contiene el valor devuelto por el programa (`r0`).	0
9	`reset`	r/w	Cuando esta señal está activa reinicia el estado del procesador, `PC` apunta a la primera instrucción y el valor de los registros queda en 0. Los valores en memoria no se ven alterados.	1

4.2.2. Escritura del programa y arranque

Con las herramientas disponibles, el proceso de cargar y ejecutar un programa se resume en:

1. Dormir al procesador y activar reset.

2. Escribir el programa en la memoria de instrucciones.

3. Desactivar reset.

4. Escribir el valor inicial del FP en `r10`.

5. Despertar al procesador.

El resultado de la ejecución se obtiene por encuesta. El MicroBlaze tiene soporte a interrupciones, de modo que notificar el fin de programa mediante interrupción sería la opción más sensata, al menos de cara a introducir soporte en un sistema operativo. No obstante, se ha decido simplificar el protocolo para un primer acercamiento.

4.2.3. Accesos concurrentes a memoria compartida

Cuando se escribe en memoria no compartida desde la interfaz AXI el procesador BPF debe estar detenido, de lo contrario la operación no producirá ningún cambio. En el caso de leer en esta zona de memoria con el procesador en marcha, no se asegura que el dato leído sea correcto.

Si el acceso es a memoria compartida, entra en juego el modelo de memoria. BPF no tiene definido un modelo de memoria. Más bien es dependiente del modelo de memoria de la plataforma en la que se ejecuta [27]. El procesador BPF diseñado en este TFG garantiza que las instrucciones de memoria son atómicas en memoria compartida, por medio de un árbitro que limita el acceso al bus y permite bloquearlo hasta completar la operación. De esta forma la interacción con los elementos de un mapa se reduce a operaciones de memoria y no requiere incluir nuevas funciones auxiliares. Sin embargo, esta garantía no se puede aplicar a los accesos externos, debido a que el bus AXI es de 32 bits y las operaciones de 64 bits requieren dos transacciones.

Una forma de permitir accesos atómicos desde el exterior sería hacer visible un semáforo desde el bus AXI que permita bloquear las operaciones de memoria del core BPF hasta completar las transacciones necesarias. El problema de esta alternativa es que el tiempo de ejecución de los programas pasa a depender en gran medida de la carga de la CPU – justamente lo que se pretende evitar con el *Hardware Offload*.

La solución escogida no implica bloqueos y proporciona más independencia al procesador BPF:

– Lecturas y escrituras menores a 32 bits se ejecutan en una única transacción.

– Cuando se lee un dato de 32 bits, el IOMM almacena la dirección y carga la palabra de 64 bits en el que está contenido en un búfer. Si la siguiente transacción es la lectura de la otra mitad de la palabra (cuando la dirección guardada y la actual únicamente se diferencian en el tercer bit menos significativo), el dato es leído desde el búfer, en lugar de volver a acceder a memoria. Cualquier transacción distinta a esta invalida el búfer.

– Cuando se escribe un dato de 32 bits, el IOMM almacena la dirección y lo guarda en un búfer sin escribirlo en memoria. Si la siguiente transacción es la escritura de la otra mitad de la palabra, el dato guardado y el actual se escriben en una única operación. Cualquier transacción distinta a esta vacía el búfer y obliga a escribir el dato en memoria antes de continuar con la transacción actual.

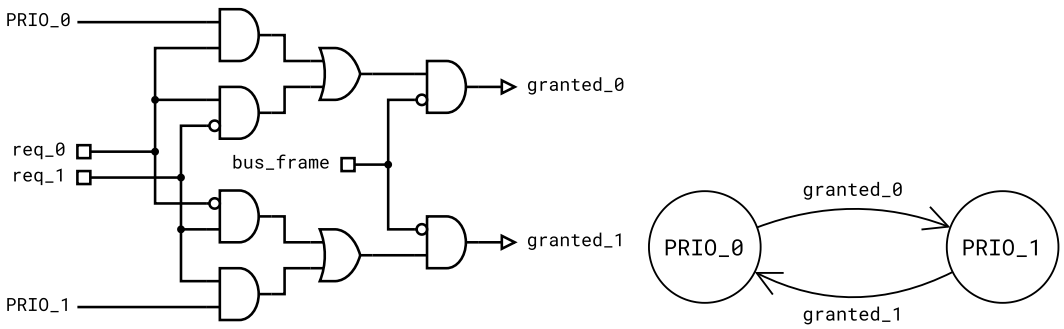
– Se puede forzar la invalidación o el vaciado del búfer escribiendo cualquier valor en la dirección anterior a la dirección base de la memoria compartida. (*End of Transaction*). Esto evita problemas cuando dos transacciones distintas de 32 bits se confunden con una de 64 bits emulada.

El arbitraje implicado utiliza un sistema con prioridad LRU, con el fin de evitar inanición. Su lógica interna se puede observar en la Fig. 4.3.

4.2.4. Gestión de mapas

Los mapas de BPF dejan libre a la implementación el tipo de mapas soportados, así como el límite en el tamaño de la clave y el valor de cada elemento. La unidad de mapas del IOMM soporta mapas de tipo *array* de hasta 64 bits de clave y/o valor. Actúa como una base de datos con información sobre los mapas cargados. Cuando la HFU requiere calcular la dirección de un elemento cualquiera en un mapa, consulta a la unidad de mapas el identificador del mapa y esta le devuelve un registro con la información descrita en la Tab. 4.2. El Anexo E.4 muestra los detalles de implementación de una HFU que permite la búsqueda de elementos a partir de dicha información.

Estos registros se pueden escribir desde el bus AXI. La gestión del espacio de memoria ante la creación y destrucción de nuevos mapas es responsabilidad del software, que se debe encargar de preparar los registros con la información correcta. El Anexo H.1 contiene el código fuente de un ejemplo de biblioteca elaborada en este proyecto para acceder al periférico, e incluye funciones para configurar hasta dos mapas.



(a) Lógica combinacional del árbitro (b) Estados del árbitro

Figura 4.3: Árbitro con prioridad LRU

Tabla 4.2: Descripción de los valores contenidos en un registro de control de mapa (`MAP_REG_X`)

Bit	**Señal**	**Acceso**	**Descripción**	**Valor en reset**
`11:0`	`base_ptr`	r/w	Dirección de la palabra de 64 bits a partir de la cual está alojado el mapa en memoria. La dirección es local al dispositivo y omite los 3 bits menos significativos.	`0`
`13:12`	`key_size`	r/w	Tamaño de las claves, representado por el logaritmo en base 2 del tamaño en bytes.	`0`
`15:14`	`value_size`	r/w	Tamaño de los valores, representado por el logaritmo en base 2 del tamaño en bytes.	`0`
`30:16`	`max_entries`	r/w	Número máximo de entradas que puede contener el mapa.	`0`
`31`	`valid`	r/w	Esta señal debe estar activa para indicar que el mapa está correctamente cargado.	`0`

4.3. Metodología de pruebas

En esta fase del desarrollo únicamente se han realizado pruebas de integración. A los programas de pruebas de la Sec. 3.3 se han añadido dos nuevos:

- **Test de IOMM:** Su objetivo es comprobar el correcto funcionamiento del IOMM ante todo tipo de transacciones AXI. Comprueba la carga de un programa (Anexo G.6), la activación del procesador, el acceso concurrente a memoria compartida con solicitudes de arbitraje que coinciden en un mismo ciclo y el estado del búfer ante transacciones de distintos tamaños.

- **Test de mapas:** Verifica el valor devuelto por las llamadas a la función `bpf_lookup_elem`. Desde el bus AXI se carga un mapa en la HFU y, con el core en marcha, se modifica el tamaño de los elementos del mapa y se altera su id (Anexo G.7).

Ambas pruebas requieren *testbenchs* específicos para generar las transacciones del bus. Los programas que no necesitan interacción externa se pueden seguir verificando usando un *testbench* de carga de programa (Anexo G.8).

4.4. Prueba de concepto

Alcanzada esta etapa del desarrollo, el periférico está listo para ser probado desde el MicroBlaze. El circuito a sintetizar se muestra en la Fig 4.4. Como componentes auxiliares se han incluido un LogiCORE™ IP Clocking Wizard, que divide o multiplica la frecuencia de reloj, una memoria local al MicroBlaze para contener el ejecutable y una salida de UART para depuración conectada mediante una LogiCORE™ IP AXI UART Lite interface.

La prueba de concepto consiste en un programa en C que carga un programa BPF previamente compilado a la memoria de instrucciones, lo ejecuta y aporta retroalimentación a través de la UART. Un ejemplo de programa y de biblioteca de interacción con el acelerador están disponibles en el Anexo H.

Desde el entorno de Vivado es posible ejecutar una simulación del diseño utilizando un *testbench* capaz de emular una terminal conectada a la UART, utilizando el paquete `textio` de la biblioteca estándar de VHDL.

La síntesis en FPGA queda fuera de el alcance de este proyecto. No obstante, se proporciona la vista de implementación del diseño en una Xilinx Kintek 7© (Fig. 4.5) con el fin de distinguir visualmente el uso de recursos de cada componente. En ella se observa claramente que el componente que hace uso de más celdas lógicas es la ALU, donde el divisor representa más del 80 % del área utilizada.

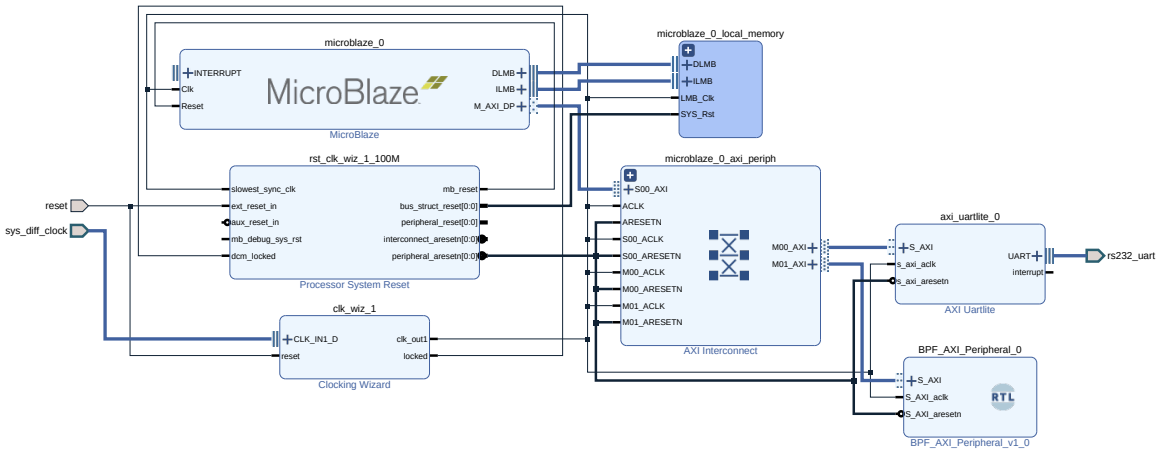
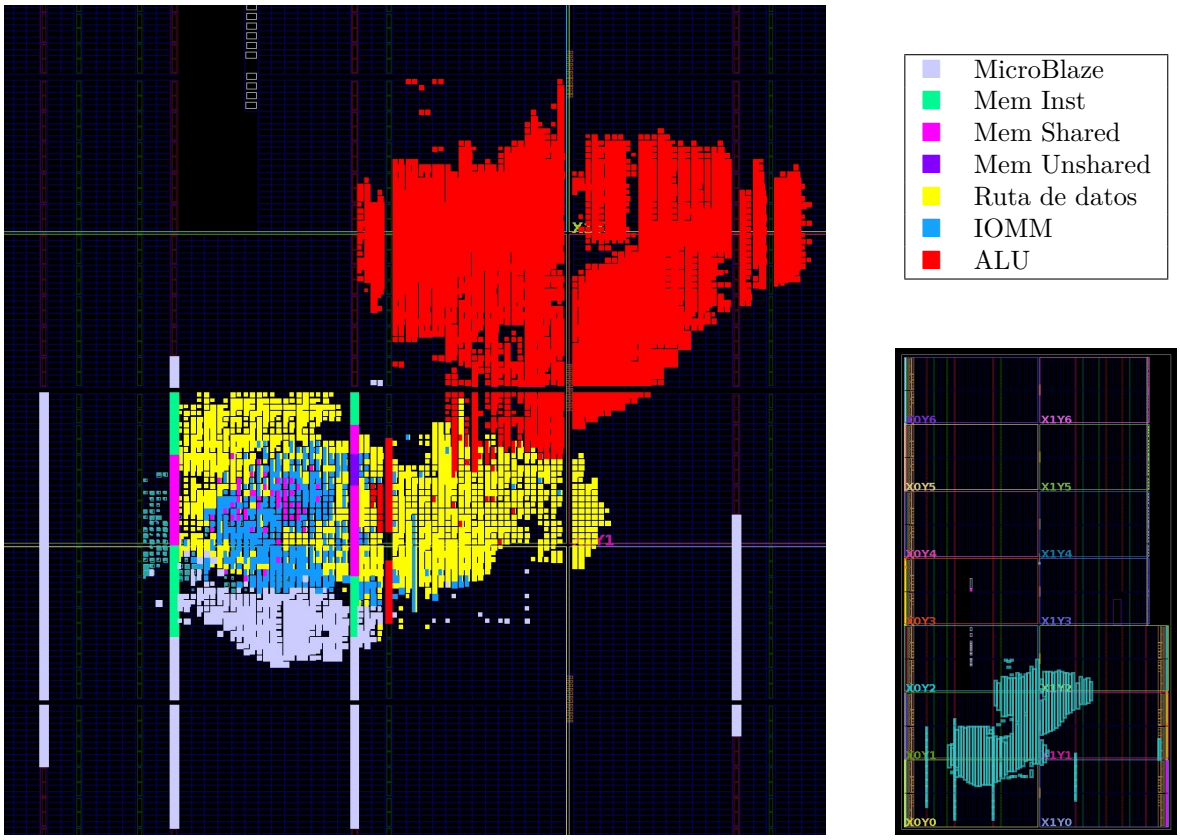


Figura 4.4: Diagrama de bloques generado por Vivado del circuito a sintetizar



(a) Identificación de componentes en el diseño implementado (b) Vista alejada

Figura 4.5: Vista de dispositivo del diseño implementado

Capítulo 5

Conclusiones y líneas abiertas

5.1. Diseño de un microprocesador para la ISA BPF

El esfuerzo de diseño realizado confirma que la ISA BPF ha sido concebida para su ejecución a través de un compilador JIT, y no a través de un hardware específico, aunque se trata en todo caso de una ISA tipo RISC muy simple. Un ejemplo entre otros es que `load` y `atomic` cambien el uso de los registros, que prioriza la semántica a cambio de complicar la decodificación. En cualquier caso, esto no representa una complicación real ni compromete el tiempo de ciclo, más sensible a la implementación del divisor, multiplicador y subsistema de acceso a memoria. Por otra parte, el uso de dos registros facilitan la gestión de mecanismos como la anticipación de operandos.

El interfaz de memoria ha estado especialmente marcado por el enfoque en el concepto *offload*. Para poder descargar al procesador principal tiene que diseñarse necesariamente con carácter de *acelerador* o *coprocesador*, lo que significa que la memoria o bien se comparte o bien requiere transferencias anfitrión - dispositivo, con los problemas y limitaciones que ello implica. En el caso de BPF es preciso además acceder a una región que permita la compartición de mapas con el procesador principal. Este carácter de acelerador / coprocesador lleva implícita la necesidad de diseñar interfaces de memoria que permitan la conexión con buses estándar, en este caso AXI, dado el contexto y el entorno de desarrollo.

BPF es una arquitectura abierta a cambios, que ha obligado a seleccionar las capacidades del procesador a partir de un conjunto de instrucciones bien establecido, descartando funcionalidades de legado y otros aspectos por definir en pos de un procesador lo más sencillo posible. Por otro lado, la existencia de funciones auxiliares y el enfoque que se les ha dado en el proyecto aportan una forma escalable de ampliar las funcionalidades del procesador.

Todos estos aspectos nos han llevado al diseño de una organización del procesador y la interfaz de memoria que consideramos original, al menos como punto de partida

de futuras optimizaciones.

5.2. Líneas abiertas

Este TFG supone un punto de partida novedoso que abre varias vías de continuación, entre las que consideramos más interesantes a corto o medio plazo las siguientes:

- Continuar la versión sintetizable en FPGA, buscando especialmente la reducción del tiempo de ciclo.
- Sabemos que se han diseñado bajo demanda NIC con soporte IEEE 802.1As para sincronización temporal, que incluyen una FPGA para implementar mecanismos TSN. Sería interesante poder integrar la versión sintetizable en uno de estos dispositivos para probar sus posibilidades para filtrado de paquetes.
- Ampliar el soporte para tipos adicionales de mapas.
- Mejorar la carga de programas mediante DMA y añadir soporte a interrupciones.
- Realizar una selección de programas / filtros BPF y estudiar sus características para determinar el posible beneficio de incorporar mecanismos de ocultación de latencia, predicción de saltos o explotación del paralelismo a nivel de instrucción.

Bibliografía

[1] "eBPF," https://ebpf.io/.

[2] F. Fejes, P. Antal, and M. Kerekes, "The TSN Building Blocks in Linux," *arXiv e-prints*, p. arXiv:2211.14138, Nov. 2022. doi: 10.48550/arXiv.2211.14138

[3] "MicroBlaze processor reference guide (UG984)," https://docs.amd.com/r/en-US/ug984-vivado-microblaze-ref.

[4] "AMBA AXI and ACE protocol specifcation." https://documentation-service.arm.com/static/5f915b62f86e16515cdc3b1c.

[5] "Visual Studio Code," https://code.visualstudio.com/.

[6] "Modern VHDL support for Visual Studio Code," https://marketplace.visualstudio.com/items?itemName=rjyoung.vscode-modern-vhdl-support.

[7] "Git: Free and open source distributed version control system," https://git-scm.com.

[8] "GHDL: free and open-source analyzer, compiler, simulator and (experimental) synthesizer for VHDL," https://ghdl.github.io/ghdl/.

[9] "GTKWave is a fully featured GTK+ based wave viewer," https://gtkwave.sourceforge.net/.

[10] "Bash is the GNU project's shell — the Bourne Again Shell," https://www.gnu.org/software/bash/.

[11] "MSYS2: Software distribution and building platform for Windows," https://www.msys2.org/.

[12] "Vivado is the design software for AMD adaptive SoCs and FPGAs," https://www.xilinx.com/products/design-tools/vivado.html.

[13] "Overleaf, the online LaTeX editor," https://www.overleaf.com.

[14] “dagrams.net: security-first diagramming for teams,” https://www.diagrams.net/.

[15] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128122757

[16] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017. ISBN 0128119055

[17] F. Shahinfar, S. Miano, G. Siracusano, R. Bifulco, A. Panda, and G. Antichi, “Automatic kernel offload using BPF,” in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, ser. HOTOS ’23. New York, NY, USA: Association for Computing Machinery, 2023. doi: 10.1145/3593856.3595888. ISBN 9798400701955 p. 143–149. [Online]. Available: https://doi.org/10.1145/3593856.3595888

[18] “bpf - perform a command on an extended BPF map or program,” https://man7.org/linux/man-pages/man2/bpf.2.html.

[19] “Program types (Linux),” https://ebpf-docs.dylanreimerink.nl/linux/program-type/.

[20] “Map types (Linux),” https://ebpf-docs.dylanreimerink.nl/linux/map-type/.

[21] “BPF Instruction Set Architecture (ISA),” https://www.kernel.org/doc/html/latest/bpf/standardization/instruction-set.html.

[22] “Corigine eBPF technical papers,” https://www.corigine.com/technologylist-24.html.

[23] R. D. G. Pacífico, L. F. S. Duarte, L. F. M. Vieira, B. Raghavan, J. A. M. Nacif, and M. A. M. Vieira, “eBPFlow: A hardware/software platform to seamlessly offload network functions leveraging eBPF,” *IEEE/ACM Transactions on Networking*, vol. 32, no. 2, pp. 1319–1332, 2024. doi: 10.1109/TNET.2023.3318251

[24] “BPF architecture - BPF to BPF calls,” https://docs.cilium.io/en/stable/bpf/architecture/#bpf-to-bpf-calls.

[25] “libbpf overview,” https://docs.kernel.org/bpf/libbpf/libbpf_overview.html.

[26] “eBPF bytecode assembler and compiler,” https://github.com/emilmasoumi/ebpf-assembler.

[27] J. Corbet, "Concurrency management in BPF," *LWN.net*, 2019. [Online]. Available: https://lwn.net/Articles/779120/

[28] "Classic BPF vs eBPF," https://www.kernel.org/doc/html/latest/bpf/classic_vs_extended.html.

[29] M. K. Alexei Starovoitov, Joe Stringer, "eBPF syscall," https://docs.kernel.org/userspace-api/ebpf/syscall.html.

[30] "Network Flow Processor (NFP) kernel driver," https://docs.kernel.org/networking/device_drivers/ethernet/netronome/nfp.html.

[31] "GitHub," https://github.com/.

[32] "IEEE standard for information technology - Portable Operating System Interface (POSIX(R))," *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pp. 1–3874, 2008. doi: 10.1109/IEEESTD.2008.4694976

Siglas

ALMA Arquitectura de Lenguaje Máquina. 1

ALU Arithmetic Logic Unit. 11, 15, 17, 26, 39, 68–71, VII

AXI Advanced eXtensible Interface. 19, 20, 23–25, 39, 75, III, IV, VI, VII

BC Branch Checker. 11, 15, 67, VII

BPF Berkeley Packet Filtering. 1–3, 5–9, 13–17, 19, 20, 23, 24, 26, 29, 30, 39, 43, 47, 49, 67, 68, 77, 100, III, IV, VI

BRAM Block Read Only Memory. 15, 17, 20, 71, 73

cBPF Classic Berkeley Packet Filter. 6

CPU Central Processing Unit. 23

CU Control Unit. 59–61, VII

DMA Direct Memory Access. 30

eBPF Extended Berkeley Packet Filter. 6, 9, III, IV

EU Exception Unit. 13, 15, 16, 65, 71, 74, VII

EX Execution (stage). 11, 12, 14, 16, 39, 61, 63, 74, 78, 105, 106

FC Frame Counter. 77

FP Frame Pointer. 6, 23, 77

FPGA Field-Programmable Gate Array. 2, 3, 7, 8, 15, 17, 19, 20, 26, 30, III, IV

FU Forwarding Unit. 11, 12, 63, VII

HFU Helper Function Unit. 12, 16, 17, 24, 25, 39, 43, 55, 74, 93, VI, VII

HU Hazard Unit. 11, 12, 59, 61, 63, 74, 77, VII

ID Instruction Decode (stage). 9, 11, 13–15, 59, 61, 63, 74, 78, 105

IF Instruction Fetch (stage). 11, 105

IOMM Input-Output and Memory Module. 20, 24, 25, 43, 98, 105

IP Intelectual Property. 15, 39, 68

IR Instruction Register. 9, 11, 65

ISA Instruction Set Architecture. 1, 6, 8, 9, 13, 16, 29, III, IV

JIT Just-In-Time. 1, 29, III, IV

LRU Least Recently Used. 24, 25, 39

MEM Memory Access (stage). 11, 12, 14, 16, 41, 61, 63, 71, 74

MIPS Microprocessor without Interlocked Pipeline Stages. 5

NFP Network Flow Processor. 7

NIC Network Interface Card. 1, 6, 7, 30, IV

PC Program Counter. 9, 17, 77

RAM Read Only Memory. 7, 12, 17

RTL Register Transfer Language. 9, 12, 41, 42, 55–58, 77, 78, VI

SFU Stack Frame Unit. 77, 78

SoC System on Chip. 3, 19, 20, III, IV

SP Stack Pointer. 39, 77, 78

TFG Trabajo de Fin de Grado. 1–3, 23, 30, 47, III, IV

TSN Time-Sensitive Networking. 30, 47, III, IV

UART Universal Asynchronous Receiver-Transmitter. 26

VHDL Very High Description Language. 2, 7, 15–17, 26, 68, III, IV

WB Write Back (stage). 11, 12, 15, 41, 63, 71

XDP eXpress Data Packet. 5, 6, 20

Lista de figuras

Lista de tablas

Lista de listados de código

Anexos

Anexos A

Dedicación

El diagrama de de Gantt con la distribución del trabajo realizado se muestra en la Fig. A.1. El tiempo dedicado al desarrollo de las partes del proyecto – procesador BPF e integración en un sistema *baremetal* – corresponde a aproximadamente un tercio del tiempo total de trabajo para cada una, dejando el último tercio dedicado a la elaboración de esta memoria. La Tab. A.1 muestra el tiempo total trabajado.

Este TFG se apoya en un trabajo previo de investigación acerca de la infraestructura que permite el uso de BPF dentro del kernel de Linux. Dicho trabajo formaba parte del trabajo de asignatura Laboratorio de Sistemas Empotrados, y solamente se enfocaba en analizar la arquitectura BPF, su implementación dentro del kernel de Linux y sus posibles aplicaciones dentro del campo de trabajo sobre TSN. Adicionalmente, establecía un diseño primitivo de la ruta de datos del procesador BPF, sin concretar el diseño de los componentes y omitiendo gran parte de las funcionalidades. Este trabajo previo está reflejado en la primera columna de tiempo del diagrama de Gantt (Fig. A.1) y no está contabilizado en la horas de dedicación (Tab. A.1).

Tabla A.1: Horas de dedicación al proyecto

Categoría	**Procesador BPF**	**Integración**
Análisis	10h	15h
Diseño	10h	17h
Implementación	47h	50h
Pruebas	24h	28h

Categoría	**Global**
Análisis	25h
Diseño	27h
Implementación	97h
Pruebas	52h
Documento	52h
Anexos	38h
TOTAL	**291h**

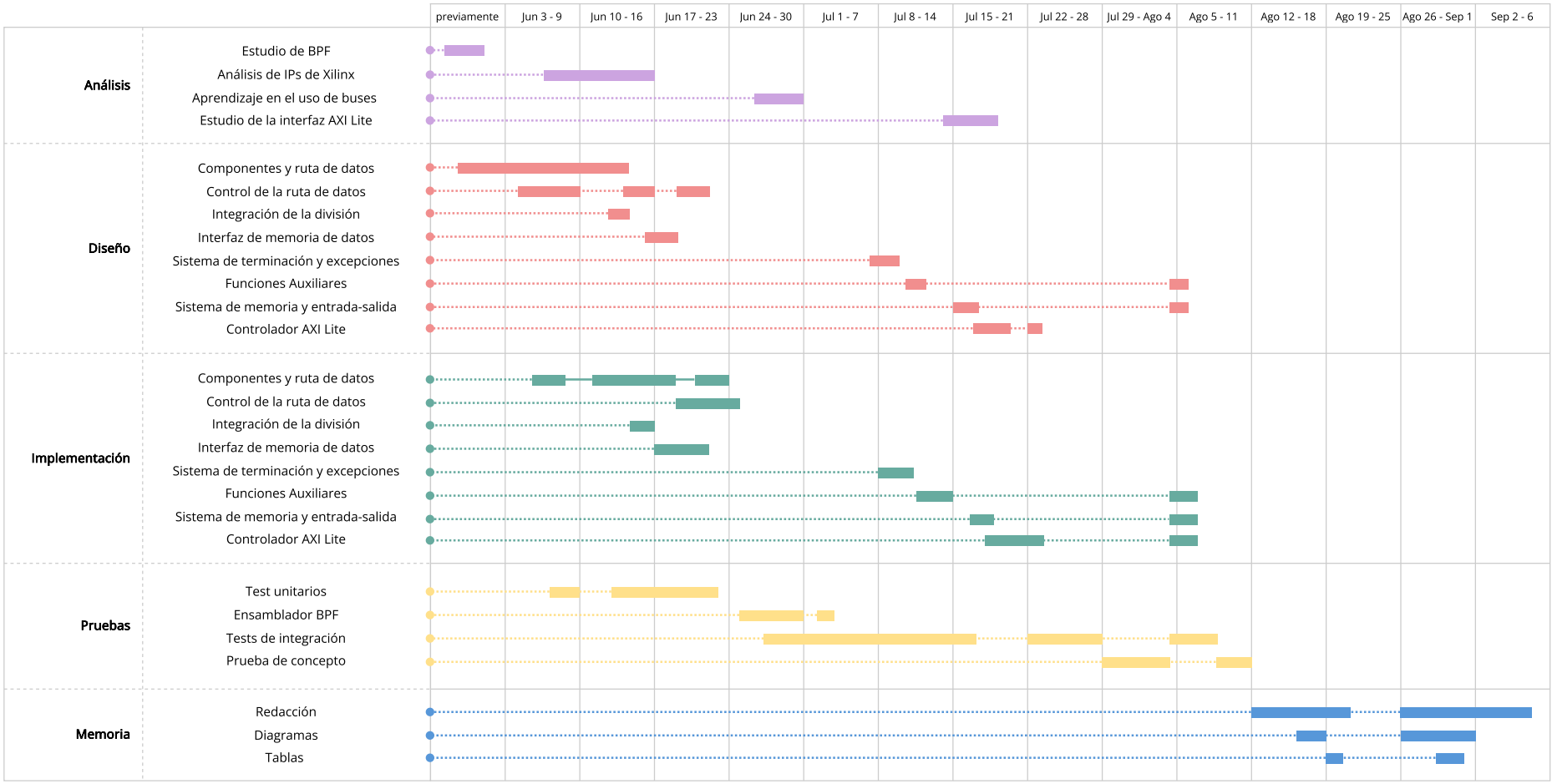


Figura A.1: Diagrama de de Gantt con la distribución del trabajo realizado a lo largo de las semanas de trabajo

Anexos B

Conjunto de Instrucciones eBPF

Las tablas mostradas a continuación contienen una descripción en pseudocódigo de las instrucciones BPF reconocidas por el procesador. Los nombres de las instrucciones son los reconocidos por el ensamblador utilizado para traducir los programas de pruebas. No existe una norma establecida sobre las abreviaturas de las instrucciones BPF por lo que podrían variar de cara a usar otro ensamblador.

Las instrucciones de 32 bits utilizan únicamente los 32 bits menos significativos de los registros fuente y usan un inmediato sin extensión de signo. Si además la operación es aritmética, los 32 bits más significativos del registro destino quedan truncados, salvo si la operación es `mov32sx8` o `mov32sx16`.

Tabla B.1: Instrucciones aritméticas de 64 bits

Ensamblador	**Pseudocódigo**
`add dst imm`	`dst += imm`
`add dst src`	`dst += src`
`sub dst imm`	`dst -= imm`
`sub dst src`	`dst -= src`
`mul dst imm`	`dst *= imm`
`mul dst src`	`dst *= src`
`div dst imm`	`dst /= imm [unsigned]`
`div dst src`	`dst /= src [unsigned]`
`sdiv dst imm`	`dst /= imm [signed]`
`sdiv dst src`	`dst /= src [signed]`
`mod dst imm`	`dst %= imm [unsigned]`
`mod dst src`	`dst %= src [unsigned]`
`smod dst imm`	`dst %= imm [signed]`
`smod dst src`	`dst %= src [signed]`
`or dst imm`	`dst |= imm`
`or dst src`	`dst |= src`
`and dst imm`	`dst &= imm`
`and dst src`	`dst &= src`
`lsh dst imm`	`dst <<= imm`

lsh dst src	dst <<= src
rsh dst imm	dst >>= imm [logical]
rsh dst src	dst >>= src [logical]
arsh dst imm	dst >>= imm [arithmetic]
arsh dst src	dst >>= src [arithmetic]
neg dst	dst = dst
xor dst imm	dst ^= imm
xor dst src	dst ^= src
mov dst imm	dst = imm
mov dst src	dst = src
movsx8 dst imm	dst = *(int8_t *) imm
movsx8 dst src	dst = *(int8_t *) src
movsx16 dst imm	dst = *(int16_t *) imm
movsx16 dst src	dst = *(int16_t *) src
movsx32 dst imm	dst = *(int32_t *) imm
movsx32 dst src	dst = *(int32_t *) src

Tabla B.2: Instrucciones aritméticas de 32 bits

Ensamblador	**Pseudocódigo**
add32 dst imm	dst += imm
add32 dst src	dst += src
sub32 dst imm	dst -= imm
sub32 dst src	dst -= src
mul32 dst imm	dst *= imm
mul32 dst src	dst *= src
div32 dst imm	dst /= imm [unsigned]
div32 dst src	dst /= src [unsigned]
sdiv32 dst imm	dst /= imm [signed]
sdiv32 dst src	dst /= src [signed]
mod32 dst imm	dst %= imm [unsigned]
mod32 dst src	dst %= src [unsigned]
smod32 dst imm	dst %= imm [signed]
smod32 dst src	dst %= src [signed]
or32 dst imm	dst |= imm
or32 dst src	dst |= src
and32 dst imm	dst &= imm
and32 dst src	dst &= src
lsh32 dst imm	dst <<= imm
lsh32 dst src	dst <<= src
rsh32 dst imm	dst >>= imm [logical]
rsh32 dst src	dst >>= src [logical]
arsh32 dst imm	dst >>= imm [arithmetic]
arsh32 dst src	dst >>= src [arithmetic]
neg32 dst	dst = dst
xor32 dst imm	dst ^= imm
xor32 dst src	dst ^= src

`mov32 dst imm`	`dst = imm`
`mov32 dst src`	`dst = src`
`mov32sx8 dst imm`	`dst = *(int8_t *) imm`
`mov32sx8 dst src`	`dst = *(int8_t *) src`
`mov32sx16 dst imm`	`dst = *(int16_t *) imm`
`mov32sx16 dst src`	`dst = *(int16_t *) src`

Tabla B.3: Instrucciones de *Byteswap*

Ensamblador	**Pseudocódigo**
`le16 dst`	`dst = host_to_little_endian16(dst)`
`le32 dst`	`dst = host_to_little_endian32(dst)`
`le64 dst`	`dst = host_to_little_endian64(dst)`
`be16 dst`	`dst = host_to_big_endian16(dst)`
`be32 dst`	`dst = host_to_big_endian32(dst)`
`be64 dst`	`dst = host_to_big_endian64(dst)`
`bswap16 dst`	`dst = byte_swap16(dst)`
`bswap32 dst`	`dst = byte_swap32(dst)`
`bswap64 dst`	`dst = byte_swap64(dst)`

Tabla B.4: Instrucciones atómicas

Ensamblador	**Pseudocódigo**
`addx32 dst src off`	`*(uint32_t *) (dst + off16) += src`
`addx64 dst src off`	`*(uint64_t *) (dst + off16) += src`
`andx32 dst src off`	`*(uint32_t *) (dst + off16) &= src`
`andx64 dst src off`	`*(uint64_t *) (dst + off16) &= src`
`orx32 dst src off`	`*(uint32_t *) (dst + off16) |= src`
`orx64 dst src off`	`*(uint64_t *) (dst + off16) |= src`
`xorx32 dst src off`	`*(uint32_t *) (dst + off16) ^= src`
`xorx64 dst src off`	`*(uint64_t *) (dst + off16) ^= src`
`addfx32 dst src off`	`src = atomic_fetch_add32(dst + off16, src)`
`addfx64 dst src off`	`src = atomic_fetch_add64(dst + off16, src)`
`andfx32 dst src off`	`src = atomic_fetch_and32(dst + off16, src)`
`andfx64 dst src off`	`src = atomic_fetch_and64(dst + off16, src)`
`orfx32 dst src off`	`src = atomic_fetch_or32(dst + off16, src)`
`orfx64 dst src off`	`src = atomic_fetch_or64(dst + off16, src)`
`xorfx32 dst src off`	`src = atomic_fetch_xor32(dst + off16, src)`
`xorfx64 dst src off`	`src = atomic_fetch_xor64(dst + off16, src)`
`xchgx32 dst src off`	`src = atomic_xchg32(dst + off16, src)`
`xchgx64 dst src off`	`src = atomic_xchg64(dst + off16, src)`
`cmpxchgx32 dst src off`	`r0 = atomic_cmpxchg32(dst + off16, r0, src)`
`cmpxchgx64 dst src off`	`r0 = atomic_cmpxchg64(dst + off16, r0, src)`

Tabla B.5: Instrucciones de memoria

Ensamblador	Pseudocódigo
ld64 dst imm	dst = imm [64b immediate]
ldx8 dst src off	dst = *(uint8_t *) (src + off)
ldx16 dst src off	dst = *(uint16_t *) (src + off)
ldx32 dst src off	dst = *(uint32_t *) (src + off)
ldx64 dst src off	dst = *(uint64_t *) (src + off)
ldxs8 dst src off	dst = *(int8_t *) (src + off)
ldxs16 dst src off	dst = *(int64_t *) (src + off)
ldxs32 dst src off	dst = *(int32_t *) (src + off)
st8 dst off imm	*(uint8_t *) (dst + off) = imm
st16 dst off imm	*(uint16_t *) (dst + off) = imm
st32 dst off imm	*(uint32_t *) (dst + off) = imm
st64 dst off imm	*(uint64_t *) (dst + off) = imm
stx8 dst src off	*(uint8_t *) (dst + off) = src
stx16 dst src off	*(uint16_t *) (dst + off) = src
stx32 dst src off	*(uint32_t *) (dst + off) = src
stx64 dst src off	*(uint64_t *) (dst + off) = src
stxx8 dst src off	*(uint8_t *) (dst + off16) += src
stxx16 dst src off	*(uint16_t *) (dst + off16) += src
stxx32 dst src off	*(uint32_t *) (dst + off16) += src
stxx64 dst src off	*(uint64_t *) (dst + off16) += src

Tabla B.6: Instrucciones de salto de 64 bits

Ensamblador	Pseudocódigo
ja off	PC += off ; Jump Always
jeq dst imm off	PC += off if dst == imm
jeq dst src off	PC += off if dst == src
jgt dst imm off	PC += off if dst > imm
jgt dst src off	PC += off if dst > src
jge dst imm off	PC += off if dst >= imm
jge dst src off	PC += off if dst >= src
jlt dst imm off	PC += off if dst < imm
jlt dst src off	PC += off if dst < src
jle dst imm off	PC += off if dst <= imm
jle dst src off	PC += off if dst <= src
jset dst imm off	PC += off if dst & imm
jset dst src off	PC += off if dst & src
jne dst imm off	PC += off if dst != imm
jne dst src off	PC += off if dst != src
jsgt dst imm off	PC += off if dst > imm [signed]
jsgt dst src off	PC += off if dst > src [signed]
jsge dst imm off	PC += off if dst >= imm [signed]
jsge dst src off	PC += off if dst >= src [signed]
jslt dst imm off	PC += off if dst < imm [signed]
jslt dst src off	PC += off if dst < src [signed]

`jsle dst imm off`	`PC += off if dst <= imm  [signed]`
`jsle dst src off`	`PC += off if dst <= src  [signed]`
`call imm`	`r0 = f(r1, r2, ..., r5); Function call`
`rel  imm`	`r0 = f(r1, r2, ..., r5); Relative function call`
`exit`	`return r0; Return from function or exit program`

Tabla B.7: Instrucciones de salto de 32 bits

Ensamblador	**Pseudocódigo**
`jal  imm`	`PC += imm ; Jump Always (Long Offset)`
`jeq32  dst imm off`	`PC += off if dst == imm`
`jeq32  dst src off`	`PC += off if dst == src`
`jgt32  dst imm off`	`PC += off if dst >  imm`
`jgt32  dst src off`	`PC += off if dst >  src`
`jge32  dst imm off`	`PC += off if dst >= imm`
`jge32  dst src off`	`PC += off if dst >= src`
`jlt32  dst imm off`	`PC += off if dst <  imm`
`jlt32  dst src off`	`PC += off if dst <  src`
`jle32  dst imm off`	`PC += off if dst <= imm`
`jle32  dst src off`	`PC += off if dst <= src`
`jset32 dst imm off`	`PC += off if dst &  imm`
`jset32 dst src off`	`PC += off if dst &  src`
`jne32  dst imm off`	`PC += off if dst != imm`
`jne32  dst src off`	`PC += off if dst != src`
`jsgt32 dst imm off`	`PC += off if dst >  imm  [signed]`
`jsgt32 dst src off`	`PC += off if dst >  src  [signed]`
`jsge32 dst imm off`	`PC += off if dst >= imm  [signed]`
`jsge32 dst src off`	`PC += off if dst >= src  [signed]`
`jslt32 dst imm off`	`PC += off if dst <  imm  [signed]`
`jslt32 dst src off`	`PC += off if dst <  src  [signed]`
`jsle32 dst imm off`	`PC += off if dst <= imm  [signed]`
`jsle32 dst src off`	`PC += off if dst <= src  [signed]`

Anexos C

Descripción RTL de instrucciones en la ruta de datos

Las tablas mostradas a continuación contienen una descripción algorítmica del camino seguido por los valores a través de los bancos de etapa. Las instrucciones están categorizadas según su clase y código. La columna *other* indica la condición que debe cumplir una instrucción para seguir ese camino, con prioridad de selección de arriba hacia abajo. La Tab. C.1 describe los operandos utilizados. A estos operandos se han añadido las macros auxiliares de la Tab. C.2 con el fin de simplificar el esquema.

Cuando se escribe `X <op> Y` se pretende indicar que el valor asignado es el resultado de la operación aritmética o lógica (trabajando en modo de 32 o 64 bits) aplicada a `X` e `Y`. La etiqueta `<size>` hace referencia a la señal de control `value_size`.

Tabla C.1: Descripción de operandos RTL

Operador	**Descripción**
`<-`	Asignación de valor a un registro.
`:=`	Asignación de un alias para un valor.
`[]`	Acceso indexado a un contenedor.
`##`	Efecto colateral sobre otra etapa.
`if/else`	Acción o valor condicional.
`atomic`	Acción atómica.

Tabla C.2: Descripción de macros auxiliares para RTL

Macro	**Descripción**
`Concat(low, high)`	Concatena los bits de ambas palabras.
`SX(x, size)`	Devuelve x extendido de signo (a 64 bits, si no se indica tamaño).
`Tr(x, size)`	Devuelve el valor de x truncado al tamaño indicado.
`fw(operand)`	Indica que el operando pasa por el sistema de anticipación (*forwarding*).
`HFU_Set_Code(id)`	Notifica a la HFU la id de la función que se va a ejecutar.
`HFU(p1..p5)`	Devuelve el valor de la función ejecutada por la HFU.
`Byte_Swap(x, width)`	Devuelve el valor de x con los *width* bytes menos significativos cambiados de *endian*.
`Write(&dst, x, size)`	Escribe en *dst* el valor de x los según el tamaño de palabra.

Tabla C.3: Descripción RTL de instrucciones atómicas, acceso a memoria y carga de inmediato

Class	Code	Other	IF	ID	EX	MEM	WB
LD	IMM		IR <- Mem_Inst[PC], PC <- PC + 1	PC <- PC + 1, *##DISCARD IF*	C <- Concat(ID_imm32, EX_imm32)	ex_val <- C	Reg_Bank[dst] <- ex_val
LDX	MEM			A <- Reg_Bank[dst]	C <- fw(A) + SX(offset)	mem_val <- Mem_Data[C]	Reg_Bank[dst] <- mem_val
	MEMSX			A <- Reg_Bank[dst]	C <- fw(A) + SX(offset)	mem_val <- SX(Mem_Data[C], <size>)	Reg_Bank[dst] <- mem_val
ST	MEM			A <- Reg_Bank[dst]	C <- fw(A) + SX(offset), data <- SX(EX_imm32)	Write(Mem_Data[C], data, <size>)	
STX	MEM			A <- Reg_Bank[dst], B <- Reg_Bank[src]	C <- fw(A) + SX(offset), data <- fw(B)	Write(Mem_Data[C], data, <size>)	
	ATOMIC	¬FETCH		A <- Reg_Bank[dst], B <- Reg_Bank[src]	C <- fw(A) + SX(offset), data <- fw(B)	result := Mem_Data[C] <op>data, Write(Mem_Data[C], result, <size>)	

		FETCH		A <- Reg_Bank[dst],
				B <- Reg_Bank[src]
					C <- fw(A) + SX(offset),
					data <- fw(B)
						atomic {
						mem_val <- Tr(Mem_Data[C], <size>),
						result := Mem_Data[C] <op>data,
						Write(Mem_Data[C], result, <size>)
						}
							Reg_Bank[dst] <- mem_val

		CMPXCHG		EX_token <- Reg_Bank[r0],
				A <- Reg_Bank[dst],
				B <- Reg_Bank[src]
					MEM_token <- fw(EX_token),
					C <- fw(A) + SX(offset),
					data <- fw(B)
						atomic {
						mem_val <- Tr(Mem_Data[C], <size>),
						if (MEM_token = Mem_Data[C]) {
						Write(Mem_Data[C], data, <size>)
						}
						}
							Reg_Bank[r0] <- mem_val

Tabla C.4: Descripción RTL de instrucciones aritméticas

Class	Code	Other	IF	ID	EX	MEM	WB
ALU	END	TO_LE	IR <- Mem_Inst[PC], PC <- PC + 1	*##DISCARD ID*			
	END	TO_BE		A <- Reg_Bank[dst]	width := SX(EX_imm32), C <- Byte_Swap(A, width)	ex_val = C	Reg_Bank[dst] <- ex_val
	MOV			B <- Reg_Bank[src]	op_B := if (source = REG) fw(B) else SX(EX_imm32), C <- SX(op_B, <size>)	ex_val = C	Reg_Bank[dst] <- ex_val
	default			A <- Reg_Bank[dst], B <- Reg_Bank[src]	op_B := if (source = REG) fw(B) else SX(EX_imm32), C <- fw(A) <op32> op_B	ex_val = C	Reg_Bank[dst] <- ex_val
ALU64	END			A <- Reg_Bank[dst]	width := SX(EX_imm32), C <- Byte_Swap(A, width)	ex_val = C	Reg_Bank[dst] <- ex_val
	MOV			B <- Reg_Bank[src]	op_B := if (source = REG) fw(B) else SX(EX_imm32), C <- SX(op_B, <size>)	ex_val = C	Reg_Bank[dst] <- ex_val
	default			A <- Reg_Bank[dst], B <- Reg_Bank[src]	op_B := if (source = REG) fw(B) else SX(EX_imm32), C <- fw(A) <op64> op_B	ex_val = C	Reg_Bank[dst] <- ex_val

Tabla C.5: Descripción RTL de instrucciones de salto

Class	Code	Other	IF	ID	EX	MEM	WB

JMP	CALL	src = r0	IR <- Mem_Inst[PC],
			PC <- PC + 1
				// call ID stage 1
				EX_token <- Reg_Bank[r1],
				A <- Reg_Bank[r2],
				B <- Reg_Bank[r3],
				
				HFU_Set_Code(ID_imm32),
				
				PC <- PC + 1,
				##DISCARD IF
					// call ID stage 2
					A <- Reg_Bank[r4]
					B <- Reg_Bank[r5]
					
					// call EX stage
					MEM_token <- fw(EX_token)
					data <- fw(A)
					C <- fw(B)
						ex_val <- HFU(token,data,C,A,B)	Reg_Bank[r0] <- ex_val

	EXIT						
	JA			PC <- PC + 1 + offset, *##DISCARD IF*			

	default			A <- Reg_Bank[dst],
				B <- Reg_Bank[src],
				PC_taken <- PC + 1 + offset
					op_B := if (source = REG) fw(B)
					else SX(EX_imm32),
					
					if (fw(A) <op64> op_B) {
					PC <- PC_taken,
					##DISCARD IF,
					##DISCARD ID
					}		

JMP32	JA			PC <- PC + 1 + imm32, *##DISCARD IF*			

	default			A <- Reg_Bank[dst],
				B <- Reg_Bank[src],
				PC_taken <- PC + offset
					op_B := if (source = REG) fw(B)
					else SX(EX_imm32),
					
					if (fw(A) <op32> op_B) {
					PC <- PC_taken,
					##DISCARD IF,
					##DISCARD ID
					}		

Anexos D

Control de la ruta de datos

D.1. Unidad de Control (CU)

La CU es la encargada de generar las señales de control durante la etapa ID. Las Tab. D.1 y D.2 contienen el valor de estas señales para cada tipo de instrucción. Adicionalmente, indica los registros que van a ser consumidos para informar a la HU, lógica presente en la Tabla D.3.

Tabla D.1: Señales de control por tipo de instrucción (I)

Class	Code	Other	jump	branch	32b_jump	call	64b_imm	alu64	alu_en	addr_calc	write_en	read_en	atomic
LD	IMM		0	0	-	0	1	1	0	0	0	0	0
LDX	MEM		0	0	-	0	0	1	0	1	0	1	0
	MEMSX		0	0	-	0	0	1	0	1	0	1	0
ST	MEM		0	0	-	0	0	1	0	1	1	0	0
STX	MEM		0	0	-	0	0	1	0	1	1	0	0
	ATOMIC	¬FETCH	0	0	-	0	0	1	0	1	1	0	1
		FETCH	0	0	-	0	0	1	0	1	1	1	1
		CMPXCHG	0	0	-	0	0	1	0	1	1	1	1
ALU	END	TO_LE	0	0	-	0	0	-	-	0	0	0	0
	END	TO_BE	0	0	-	0	0	1	1	0	0	0	0
	MOV		0	0	-	0	0	0	1	0	0	0	0
	default		0	0	-	0	0	0	1	0	0	0	0
ALU64	END		0	0	-	0	0	1	1	0	0	0	0
	MOV		0	0	-	0	0	1	1	0	0	0	0
	default		0	0	-	0	0	1	1	0	0	0	0
JMP	CALL	src=r0	0	0	-	1	0	1	0	0	0	0	0
	EXIT		0	0	-	0	0	-	0	0	0	0	0
	JA		1	0	0	0	0	-	0	0	0	0	0
	default		0	1	0	0	0	1	0	0	0	0	0
JMP32	JA		1	0	1	0	0	-	0	0	0	0	0
	default		0	1	0	0	0	0	0	0	0	0	0

Tabla D.2: Señales de control por tipo de instrucción (II)

Class	Code	Other	force_imm	sign_ext	value_size	mem_to_reg	reg_write	write_r0	discard_IF	finish
LD	IMM		1	0	64b	0	1	0	1	0
LDX	MEM		-	0	opcode.size	1	1	0	0	0
	MEMSX		-	1	opcode.size	1	1	0	0	0
ST	MEM		1	-	opcode.size	-	0	0	0	0
STX	MEM		0	-	opcode.size	-	0	0	0	0
	ATOMIC	¬FETCH	0	-	opcode.size	-	0	0	0	0
		FETCH	0	0	opcode.size	1	1	0	0	0
		CMPXCHG	0	0	opcode.size	1	0	1	0	0
ALU	END	TO_LE	0	0	-	0	0	0	0	0
	END	TO_BE	1	0	-	0	1	0	0	0
	MOV		0	1	code(offset)[1]	0	1	0	0	0
	default		0	0	-	0	1	0	0	0
ALU64	END		1	0	-	0	1	0	0	0
	MOV		0	1	code(offset)	0	1	0	0	0
	default		0	0	-	0	1	0	0	0
JMP	CALL	src=r0	-	-	64b	0	0	1	1	0
	EXIT		-	-	-	-	0	0	0	1
	JA		-	-	-	-	0	0	0	0
	default		0	-	-	-	0	0	0	0
JMP32	JA		-	-	-	-	0	0	0	0
	default		0	-	-	-	0	0	0	0

Tabla D.3: Señales que indican registro consumido

Class	Code	Other	DST	SRC	Token
LD	IMM		0	0	0
LDX	MEM		1	1	0
	MEMSX		1	1	0
ST	MEM		1	0	0
STX	MEM		1	1	0
	ATOMIC	¬FETCH	1	1	0
		FETCH	1	1	0
		CMPXCHG	1	1	1
ALU	END	TO_LE	1	0	0
	END	TO_BE	1	0	0
	MOV		0	1	0
	default	s = REG and ¬NEG	1	1	0
		else	1	0	0
ALU64	END		1	0	0
	MOV		0	1	0
	default	s = REG and ¬NEG	1	1	0
		else	1	0	0
JMP	CALL	src = r0	0	0	0
	EXIT		0	0	0
	JA		0	0	0
	default	s = REG	1	1	0
		else	1	0	0
JMP32	JA		0	0	0
	default	s = REG	1	1	0
		else	1	0	0

[1]El tamaño en bytes de la palabra para las instrucciones MOV se codifica como un número dentro del offset (8, 16 o 32). La señal de control value_size utiliza una codificación en 2 bits, por lo que la CU debe codificarlo.

D.2. Unidad de Riesgos (HU)

La HU funciona como detector de riesgos de datos para provocar que la instrucción decodificada quede detenida hasta poder obtener sus operandos correctamente. Los riesgos de control y estructurales son notificados propiamente por los componentes que los provocan y quedan fuera de la responsabilidad de la HU.

Los riesgos que debe detectar son los siguientes:

- Token (`r0`), registro fuente o registro destino no están listos para anticipación. Esto ocurre cuando el dato se produce en etapa MEM y la consumidora es la siguiente instrucción. Para cuando la consumidora está en etapa EX el valor aún no habrá sido producido, por lo que es necesario detenerla un ciclo en etapa ID.

- Parámetros 1, 2 o 3 de una función auxiliar no están listos para anticipación (causa análoga).

- Parámetros 4 o 5 de una función auxiliar no pueden ser obtenidos directamente del banco de registros. Estos valores no pasan por el sistema de anticipación y deben estar disponibles para leer del banco de registros cuando la instrucción `call` alcanza la etapa EX. Este caso se da independientemente de que el dato se produzca en EX o MEM.

La Tab. D.4 muestra las condiciones para cada riesgo en base a señales del procesador. Las señales `ID_use_dst`, `ID_use_src`, `ID_use_r0` las produce la CU, como se ve en el anexo D.1. Además, se sabe que una instrucción es productora en etapa MEM porque escribe el dato en `mem_val` (activa la señal de control `EX_mem_to_reg`) o porque es una llamada a función auxiliar (`EX_call`).

Tabla D.4: Condiciones de detención por riesgo de datos

Hazard description	**Hazard condition**										
	Is EX producer		Is ID consumer							Is result ready on MEM	
r0 not ready	EX_reg_write	and	EX_dst = 0	and	ID_use_r0					and	EX_mem_producer
	EX_write_r0	and			ID_use_r0					and	EX_mem_producer
dst not ready	EX_reg_write	and	EX_dst = ID_dst	and	ID_use_dst					and	EX_mem_producer
	EX_write_r0	and	ID_dst = 0	and	ID_use_dst					and	EX_mem_producer
src not ready	EX_reg_write	and	EX_dst = ID_src	and	ID_use_src					and	EX_mem_producer
	EX_write_r0	and	ID_src = 0	and	ID_use_src					and	EX_mem_producer
r1, r2 or r3 not ready	EX_reg_write	and	EX_dst $\in$ (0, 3]	and	EX_dst $\leq$ ID_num_params	and	ID_call	and	EX_mem_producer		
r4 or r5 not ready	EX_reg_write	and	EX_dst $\in$ (3, 5]	and	EX_dst $\leq$ ID_num_params	and	ID_call				

D.3. Unidad de Anticipación (FU)

La FU se encarga de proveer a la etapa EX de los operandos necesarios en caso de que el valor requerido no estuviera disponible para lectura en ID. A diferencia de la HU, la FU ignora si realmente se utiliza el valor, pero asegura que en caso de usarlo se proporciona el valor correcto. Las Tab. D.5 y D.6 contienen la lógica para determinar si los bancos de etapa MEM o WB contienen un posible valor de los operandos. El origen de lectura de cada operando se decide con la lógica de las Tab. D.7, D.8 y D.9. Cuando más de una condición aparece asignada a una señal significa que la activa con independencia del resto de condiciones (puerta *or* lógica).

Tabla D.5: Condiciones de posible anticipación desde banco MEM

Signal	**Activation condition**					
	Is MEM producer		Is EX consumer			
A_from_MEM	MEM_reg_write	and	MEM_dst = 2	and	EX_call	
	MEM_reg_write	and	MEM_dst = EX_dst	and	not EX_call	
	MEM_write_r0	and	EX_dst = 0	and	not EX_call	
B_from_MEM	MEM_reg_write	and	MEM_dst = 3	and	EX_call	
	MEM_reg_write	and	MEM_dst = EX_src	and	not EX_call	
	MEM_write_r0	and	EX_src = 0	and	not EX_call	
token_from_MEM	MEM_reg_write	and	MEM_dst = 1	and	EX_call	
	MEM_reg_write	and	MEM_dst = 0	and	not EX_call	
	MEM_write_r0	and			not EX_call	

Tabla D.6: Condiciones de posible anticipación desde banco WB

Signal	**Activation condition**					
	Is WB producer		Is EX consumer			
A_from_WB	WB_reg_write	and	WB_dst = 2	and	EX_call	
	WB_reg_write	and	WB_dst = EX_dst	and	not EX_call	
	WB_write_r0	and	EX_dst = 0	and	not EX_call	
B_from_WB	WB_reg_write	and	WB_dst = 3	and	EX_call	
	WB_reg_write	and	WB_dst = EX_src	and	not EX_call	
	WB_write_r0	and	EX_src = 0	and	not EX_call	
token_from_WB	WB_reg_write	and	WB_dst = 1	and	EX_call	
	WB_reg_write	and	WB_dst = 0	and	not EX_call	
	WB_write_r0	and			not EX_call	

Tabla D.7: Fuente de lectura anticipada del operando A

fw_A_from	A_from_MEM	A_from_WB	WB_mem_to_reg
FROM_MEM_C	1	-[2]	-
FROM_WB_EX_VAL	0	1	0
FROM_WB_MEM_VAL	0	1	1
NO_FW	0	0	-

Tabla D.8: Fuente de lectura anticipada del operando B

fw_B_from	B_from_MEM	B_from_WB	WB_mem_to_reg
FROM_MEM_C	1	-	-
FROM_WB_EX_VAL	0	1	0
FROM_WB_MEM_VAL	0	1	1
NO_FW	0	0	-

Tabla D.9: Fuente de lectura anticipada del token

fw_token_from	token_from_MEM	token_from_WB	WB_mem_to_reg
FROM_MEM_C	1	-	-
FROM_WB_EX_VAL	0	1	0
FROM_WB_MEM_VAL	0	1	1
NO_FW	0	0	-

[2]Se debe priorizar el valor del registro más reciente, por lo que se ignora si se usa en una etapa posterior.

D.4. Unidad de Excepciones (EU)

La EU se encarga de centralizar la gestión de errores en el procesador. Los componentes externos pueden indicarle errores en una etapa determinada. A su vez, se asegura de que no se ejecute ninguna instrucción ilegal o no implementada. La Tab. D.10 muestra los distintos valores que puede tener IR para ser considerada instrucción válida. La tabla se apoya en las condiciones auxiliares que se muestran a continuación:

```
writable ≔ reg < 10
readable ≔ reg ≤ 10

atomic_op ≔ imm[7:4] ∈ {ADD, OR, AND, XOR} and imm[..] = 0
atomic_fetch ≔ imm[0] = 1 and imm[7:4] ∈ {ADD, OR, AND, XOR, XCHG}
                and imm[..] = 0
atomic_cmpxchg ≔ imm[0] = 1 and imm[7:4] = CMPXCHG and imm[..] = 0

regular_op ≔ code ∈ {ADD, SUB, MUL, DIV, MOD, OR,
                     AND, LSH, RSH, NEG, XOR, ARSH}
signed_op ≔ code ∈ {DIV, MOD}
branch_op ≔ code ∈ {JEQ, JGT, JGE, JSET, JNE, JSGT, JSGE,
                    JLT, JLE, JSLT, JSLE}

mov_size ≔ ((offset[3] = 1 xor offset[4] = 1) xor offset[5] = 1)
                and offset[..] = 0
end_size_ok ≔ ((imm[4] = 1 xor imm[5] = 1) xor imm[6] = 1)
                and imm[..] = 0
```

Tabla D.10: Codificaciones de instrucción válidas

Valid combinations								
Class	Source	Inst. Code	Mem. Size	Mem. Mode	Dst	Src	Offset	Immediate
LD	-	-	64b	IMM	writable	0	0	-
LDX	-	-	-	MEM	writable	readable	-	0
LDX	-	-	-	MEMSX	writable	readable	-	0
ST	-	-	-	MEM	readable	readable	-	-
STX	-	-	-	MEM	readable	readable	-	0
STX	-	-	32b/64b	ATOMIC	readable	readable	-	atomic_op
STX	-	-	32b/64b	ATOMIC	readable	writable	-	atomic_fetch
STX	-	-	32b/64b	ATOMIC	readable	readable	-	atomic_cmpxchg
ALU	-	regular_op	-	-	writable	readable	0	-
ALU	-	signed_op	-	-	writable	readable	1	-
ALU	-	MOV	-	-	writable	readable	mov_size_ok	-
ALU	-	END	-	-	writable	readable	0	end_size_ok
ALU64	-	regular_op	-	-	writable	readable	0	-
ALU64	-	signed_op	-	-	writable	readable	1	-
ALU64	-	MOV	-	-	writable	readable	mov_size_ok	-
ALU64	0	END	-	-	writable	readable	0	end_size_ok
JMP	0	JA	-	-	0	0	-	0
JMP	-	branch_op	-	-	readable	readable	-	-
JMP	0	EXIT	-	-	0	0	0	0
JMP	0	CALL	-	-	0	0[3]	0	-
JMP32	0	JA	-	-	0	0	0	-
JMP32	-	branch_op	-	-	readable	readable	-	-

[3] El registro fuente indica si una instrucción `CALL` es una invocación a función de usuario ($src = 1$) o a una función auxiliar ($src = 0$).

Anexos E

Detalles de implementación por componente

E.1. Verificador de Saltos (BC)

La implementación del Verificador de Saltos genera una señal por cada condición de salto posible en BPF. Hay 3 condiciones básicas que emplean componentes dedicadas (=, > y &). El resto son derivadas de ellas según la lógica siguiente:

```
same_sign := (op_64b and A(63) = B(63)) or
               (not op_64b and A(31) = B(31))
eq := A = B
ne := not eq

lt := A < B
le := lt or eq

gt := not lt and not eq
ge := not lt

slt := (same_sign and lt) or (not same_sign and gt)
sle := (same_sign and le) or (not same_sign and ge)

sgt := (same_sign and gt) or (not same_sign and lt)
sge := (same_sign and ge) or (not same_sign and le)

set := (A & B) != 0
```

El resultado es multiplexado a partir de todas las condiciones, utilizando el código de operación como selector.

E.2. Unidad Aritmética Lógica (ALU)

La implementación de cada una de las operaciones aritméticas de BPF en VHDL es trivial usando la biblioteca estándar, salvo para la división y el módulo, que son las únicas cuya función no es sintetizable como circuito combinacional. Por ello, la ALU precisa de un divisor multiciclo. Los divisores que provee Xilinx como IP permiten elegir el tipo y tamaño de los operandos. La configuración elegida es un divisor de naturales de 64 bits, con salidas para el cociente y el resto (la alternativa es un resultado fraccionario). Con él se pueden implementar todas las versiones de la operación de división y módulo que ofrece BPF.

La Fig. E.1 muestra la ruta de datos correspondiente al camino de la división y el módulo dentro de la ALU, diseñada para soportar operaciones con o sin signo, de 32 o 64 bits y tratar divisiones entre 0 como caso aparte. Si un operando es negativo y se ejecuta la división natural, se trata como si fuera positivo, pero se niega en caso de ser una división entera. El resultado de una división entera únicamente se niega si los signos de los operandos son distintos. La operación módulo en BPF está definida igual que en el lenguaje C, por lo aprovecha la misma lógica que la división para negar valores.

Este procesador no tiene lanzamiento concurrente de instrucciones a fase de ejecución, por lo que el divisor siempre va a estar disponible cuando llega una nueva operación. El autómata de la Fig. E.2 asume esto. Sin embargo, durante la configuración del divisor se puede establecer una latencia de iniciación, pensada para ahorrar recursos. A cambio, el procesador ya no puede asumir que el divisor está siempre disponible. Como el sistema de anticipación de operandos solo garantiza que el valor anticipado es correcto durante el primer ciclo que se usa, la ALU debe guardar los operandos en

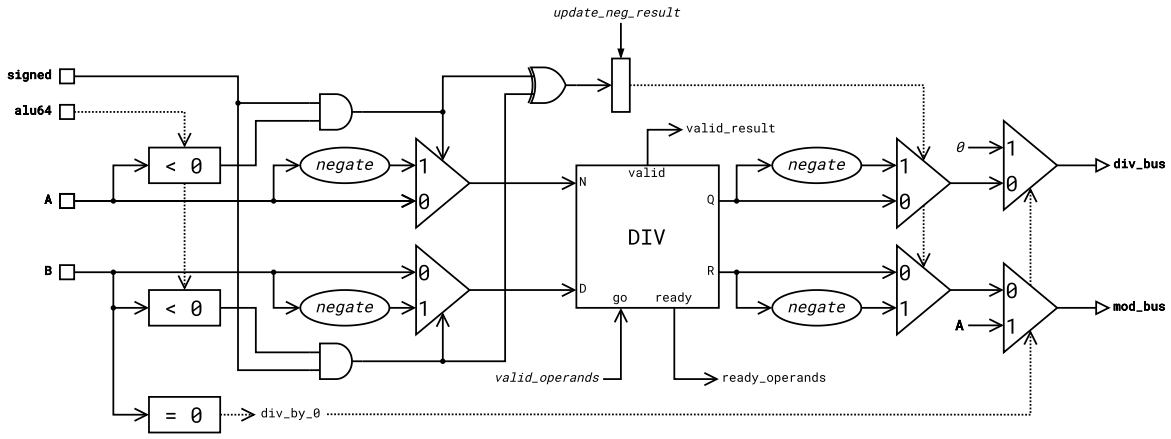


Figura E.1: Camino de la ALU diseñado para la ejecución de divisiones. `DIV` es una representación simplificada de un Divider Generator LogiCORE™ IP

caso de que el divisor no esté disponible. Este comportamiento más complejo es el se ha incluido en el core y su autómata se puede ver en la Fig. E.3.

Algo que se puede observar en los autómatas es el estado dedicado para reset. Su objetivo es mantener activa la señal reset del divisor durante al menos dos ciclos, tal y como requiere.

Otra decisión de diseño de la ALU fue utilizar el multiplicador en un ciclo que se genera por defecto. Utilizar un multiplicador secuencial permitiría reducir el tiempo de ciclo en el caso de que la multiplicación fuera el camino crítico. Como no es posible identificar este camino crítico desde la fase de diseño, en esta primera versión se ha decidido dejar el multiplicador por defecto.

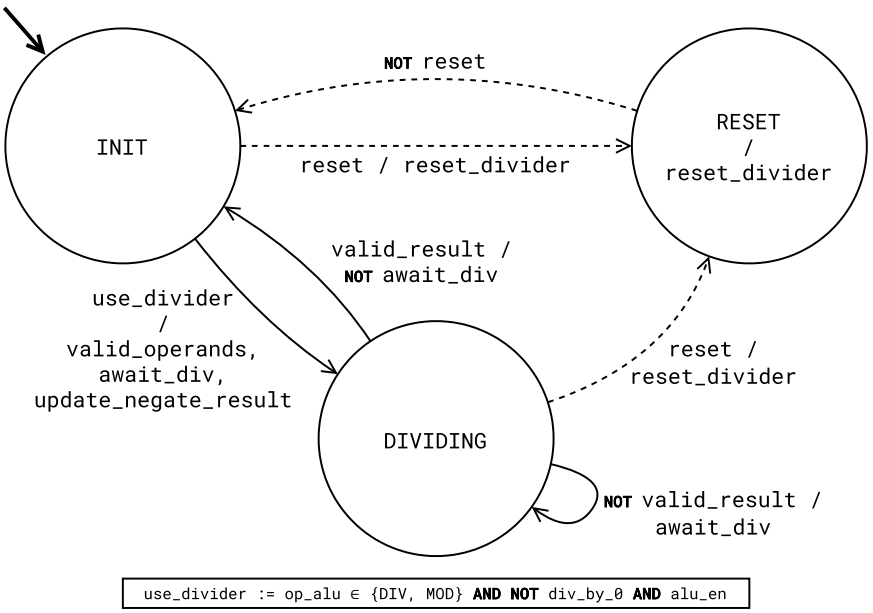


Figura E.2: Autómata de la ALU con soporte a división sin latencia de iniciación

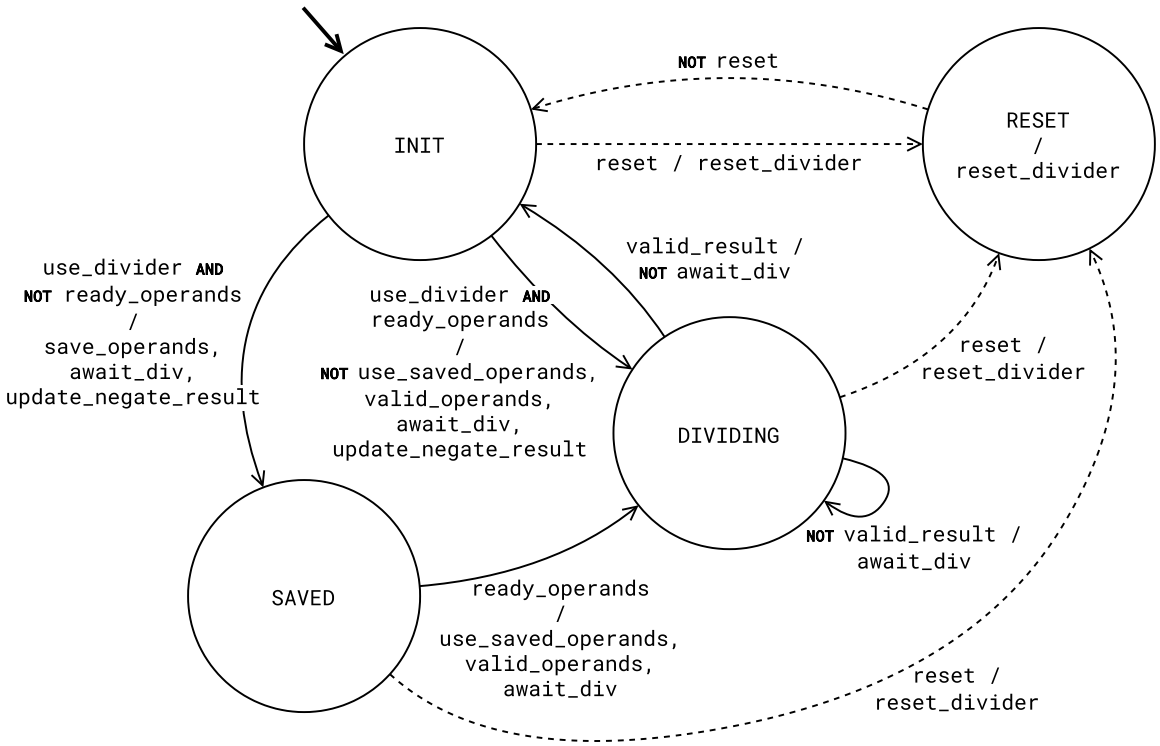


Figura E.3: Autómata de la ALU con soporte a división con latencia de iniciación

E.3. Interfaz de memoria de datos

Este componente tiene la responsabilidad de llevar a cabo todas las operaciones que requieren acceso a memoria, incluidas las operaciones atómicas, a la vez que abstrae la interacción con los bloques de memoria. La ruta de datos que se observa en la Fig. E.4 y su correspondiente autómata de control (Fig. E.5) están diseñados para interactuar con módulos de BRAM direccionables a palabras de 64 bits. Aunque en el esquema se vea un módulo de BRAM, en realidad este módulo no está presente dentro del componente, tal y como ha aparecido en la Fig 4.1. El bloque ha sido añadido para representar que las señales conectadas corresponden a las que interactúan con el bus de memoria.

La BRAM permite lecturas y escrituras con latencia de un ciclo, por lo que solamente las operaciones atómicas son multiciclo. Los accesos a memoria compartida también pueden durar más de un ciclo si no se obtiene del árbitro durante el primer ciclo.

Las lecturas son siempre de 8 bytes y alineadas, por lo que se usa el tamaño del valor y los bits menos significativos de la dirección para seleccionar el valor de salida (bloque `Byte Select`). Esta operación se produce en la etapa WB (por la latencia de un ciclo), por lo que es necesario almacenar previamente los datos de selección (`sel_info`).

Las escrituras también son alineadas, pero se permite elegir cuáles de los 8 bytes van a ser escritos, mediante una máscara. Las escrituras de palabras menores a 64 bits no alineadas requieren corregir la posición de los bytes (bloque `Byte Xchg`). Además, es preciso generar una máscara de escritura a partir de los datos de selección (bloque `Size to Mask`).

Las operaciones atómicas se realizan en dos fases: lectura y modificación-escritura. El autómata de control se encarga de bloquear las etapas anteriores generando la señal `ready`, de mantener el acceso al bus de memoria con la señal `bus_frame` si se está accediendo a memoria compartida y de multiplexar el dato de entrada a la BRAM (`store_modified`). La ruta de datos incluye una ALU de 64 bits con las operaciones `add`, `or`, `and` y `xor`. Cuando la instrucción es de 32 bits se debe alinear `data` con el valor leído de memoria. Para implementar la instrucción `xchg`, la ALU permite el paso `data`. En el caso de `cmpxchg`, la ALU sirve como multiplexor entre `data` y el dato de memoria, activado con el resultado de comparar el dato de memoria con `token`.

La interfaz de memoria de datos también se encarga de comprobar si las direcciones corresponden al espacio de memoria compartido o no compartido, con el fin de notificarlo al autómata (señal `shared`). Si la dirección no corresponde a ninguno de ellos, pasa a estado de error, donde notifica a la EU que la etapa MEM ha generado excepción.

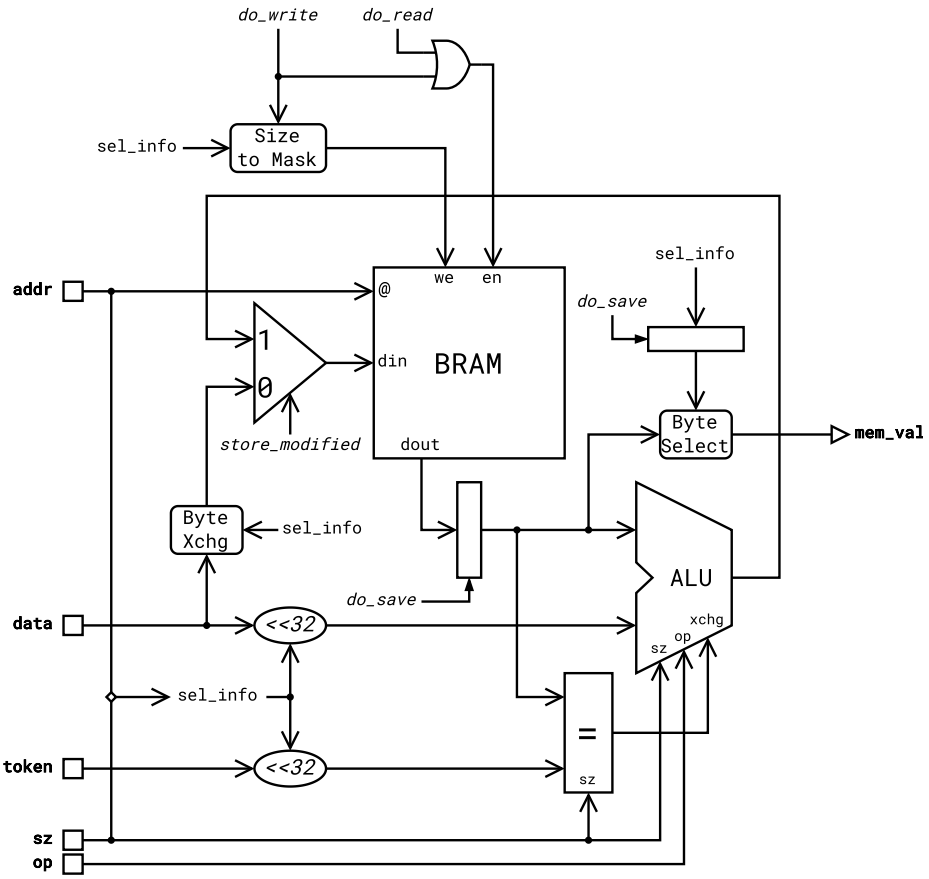


Figura E.4: Ruta de datos de la interfaz de memoria de datos

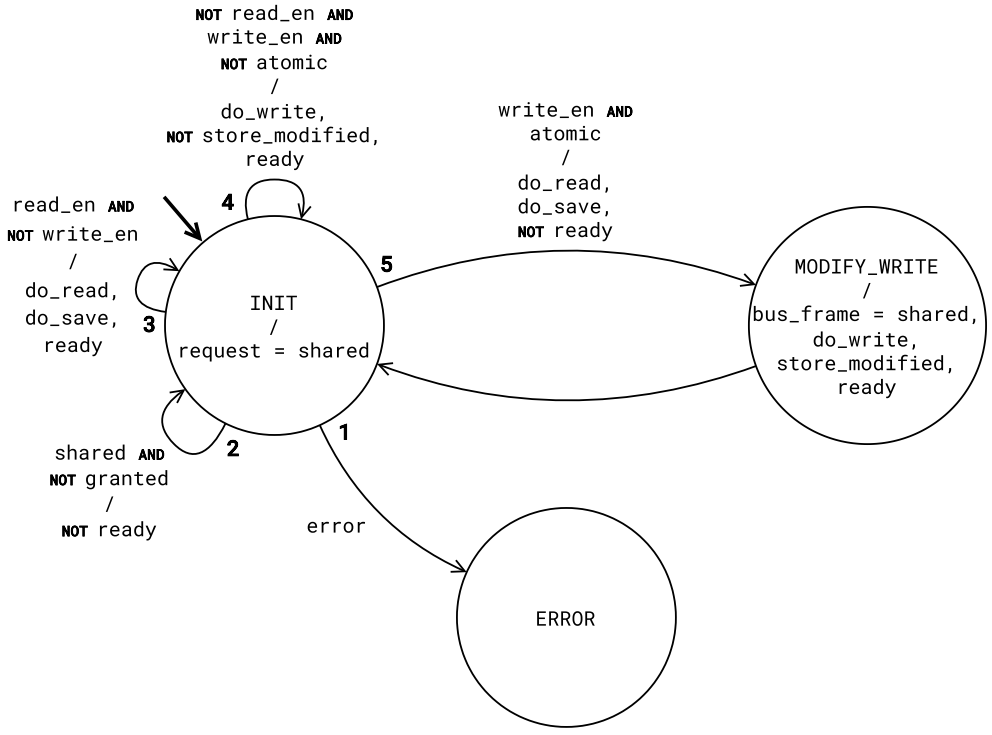


Figura E.5: Autómata de control de la interfaz de memoria de datos

El esquema de la ruta de datos (Fig E.4) muestra un registro a la salida de la BRAM. Se trata de un registro retardado, que guarda el dato de entrada un ciclo después de que su señal *load* esté activa. Su salida es el valor de entrada si *load* está activa, o contenido guardado en caso contrario.

E.4. Unidad de funciones auxiliares (HFU)

La HFU es un componente que permite implementar funciones auxiliares mediante circuitos secuenciales que avanzan en sincronía a las etapas de la ruta de datos del procesador. Utiliza la etapa ID para decodificar el id de la función, a partir del cual se decide el siguiente estado. En caso de que no exista la función indicada, avisa a la EU de error. También permite la consulta del número de parámetros que recibe una función, para informar de su consumo a la HU.

Esta versión únicamente implementa la función `bpf_lookup_elem`, que devuelve un puntero al valor del elemento buscado a partir de la id del mapa y la clave del elemento. La HFU ejecuta esta función por completo en la etapa MEM, en dos fases: consulta de los datos del mapa en la unidad de mapas y cálculo de la dirección del elemento. El autómata de la Fig. E.6 muestra el avance de la función y su sincronización con el avance de las etapas ID y EX.

El cálculo de la dirección del elemento se realiza a partir de los datos del registro de mapa, descritos en la Tab. 4.2, aplicando el siguiente cálculo:

```
elem_ptr = (base_ptr * 8) + (truncate(key, key_size) << value_size)
```

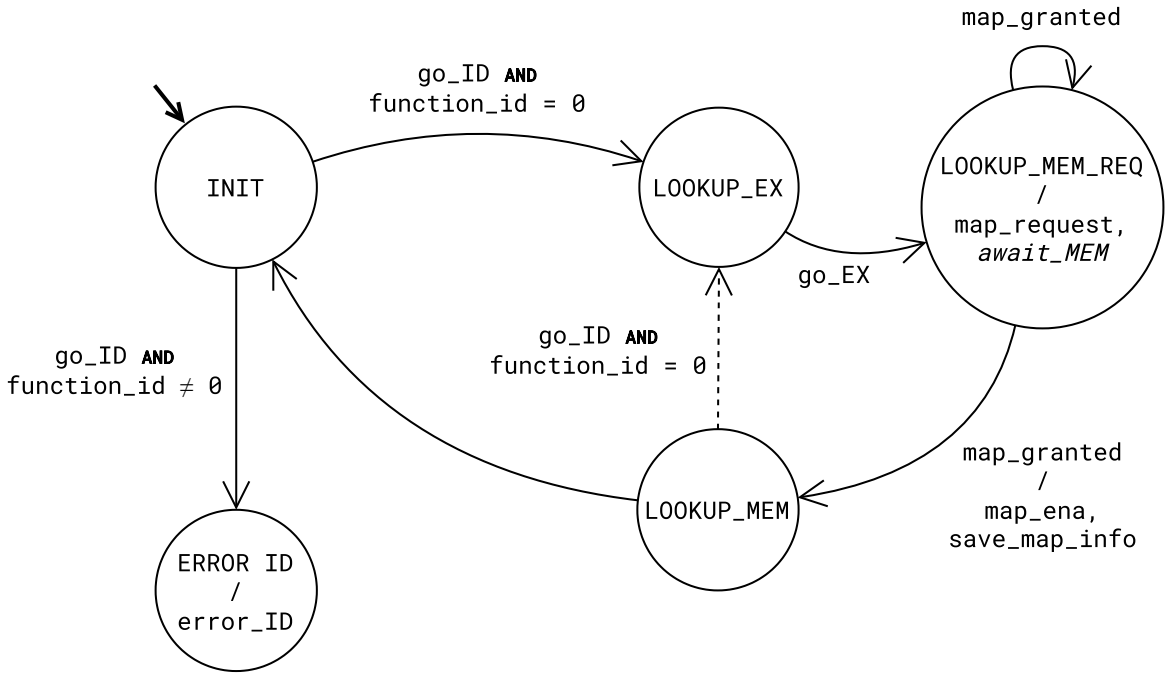


Figura E.6: Autómata de la HFU implementada con soporte para `bpf_lookup_elem`

E.5. Controlador AXI

Este componente implementa una interfaz AXI Lite secundaria para conectar el periférico con el exterior y permitir las transacciones de memoria y control explicadas en la Sec. 4.2. Para comprender en detalle las señales que conforman la interfaz AXI Lite se remite al Cap. B1 de la especificación del protocolo AXI [4].

Para la mayoría de las transacciones, el controlador funciona como decodificador de la dirección, permitiendo leer o escribir en el bloque correspondiente. Sin embargo, la inclusión de transacciones atómicas de 64 bits en un bus de 32 bits complica el diseño. Su comportamiento está determinado por dos autómatas: uno principal para traducir las transacciones en señales de control (Fig. E.7) y otro auxiliar para controlar el estado del búfer de memoria compartida (Fig. E.8).

Los estados del búfer representan la última operación sobre memoria compartida que se realizó desde el bus: `EMPTY`, si no ha afectado al búfer; `STORE_0/1`, si se trata de una escritura; y `LOAD_0/1`, si se trata de una lectura. El cero y el uno que complementan al estado señalan que la palabra accedida son los 4 bytes más o menos significativos. Los estados del búfer permiten al autómata principal determinar si es preciso escribir el contenido del búfer de escritura en memoria (trasladándose a los estados de *flush*). También sirven para saber si una lectura debe acceder a memoria o extraer el dato del búfer de lectura. El autómata auxiliar se encarga de actualizar el estado del búfer, pero solo lo hace en los ciclos que el autómata principal lo indica con la señal `update_buffer_state` (`UBS`).

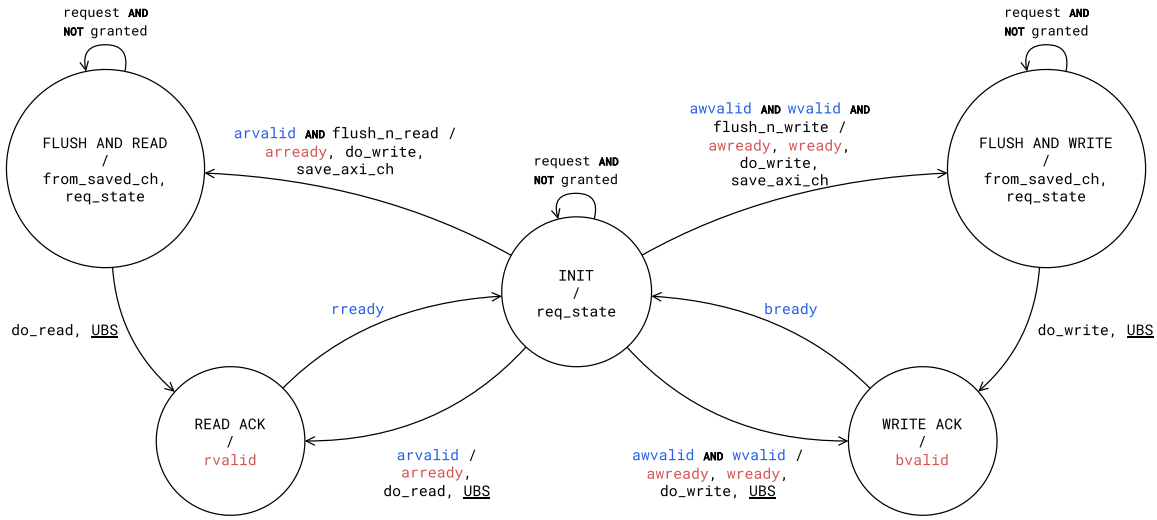


Figura E.7: Autómata de control de la interfaz AXI Lite

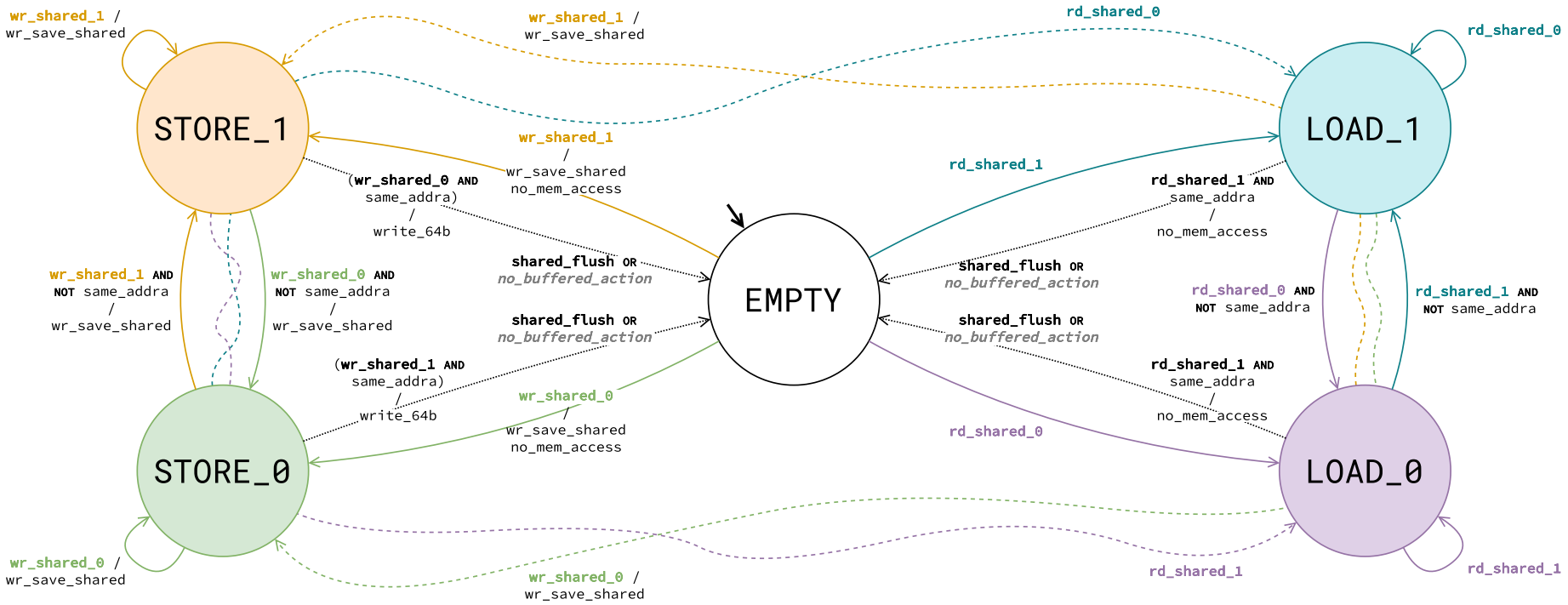


Figura E.8: Autómata de control del búfer de memoria

Anexos F

Funciones definidas por el usuario

Como se explica en la Sec. 3.2.5, el procesador BPF no implementa las llamadas a funciones de usuario, aunque aparecen en el anexo B como instrucción `rel`. Este anexo explica las consideraciones de diseño que se deben tomar para incluirlas en la ruta de datos actual.

Las llamadas a funciones son saltos relativos a PC que guardan la información del FP y el PC de retorno ($PC + 1$), con el fin de crear un nuevo marco de pila. Como BPF limita a 8 los marcos de pila activos [24], ambos valores podrían ser guardados en un pequeño banco de 8 registros, la Stack Frame Unit (SFU), indexada por la cantidad actual de marcos activos (FC). PC se guarda codificado en 12 bits, porque los programas tienen un tamaño máximo permitido de 4096 instrucciones, y para el FP se usan 6 bits, aprovechando que la pila siempre tiene un tamaño de 512 bytes y está alineada a 8 bytes (64 posibles posiciones de FP).

La arquitectura de BPF no tiene un Stack Pointer (SP) visible al programador, por lo que el marco de pila debe ser deducido a partir de las escrituras en el espacio de pila. Se considera que el SP cambia cuando se escribe en una dirección más baja (BPF usa una pila descendente). Si el programa intentase leer de zonas todavía no escritas incurre en comportamiento indefinido, lo que no está permitido por el verificador de Linux, por lo que es seguro utilizar este método.

Para soportar este tipo de llamadas, es preciso añadir comportamiento especial a la instrucción `exit` en caso de estar dentro de una función (cuando el FC es mayor a cero). Esta instrucción debe ser capaz de recuperar el estado anterior a la llamada a función. La HU también debe ser modificada para detectar riesgos de datos producidos por alteraciones en el SP.

La Tab. F.1 contiene una posible descripción RTL del comportamiento de las instrucciones a lo largo de las etapas. Se han añadido las Fig. F.1 y F.2 como borradores de lo que podría ser la infraestructura a añadir.

Tabla F.1: Posibles acciones RTL para soportar funciones de usuario

Class	Code	Other	IF	ID	EX	MEM	WB
JMP	CALL	src = 1	IR <- Mem_Inst[PC], PC <- PC + 1	PC <- PC + 1 + imm32, SFU(FC + 1).ret_PC <- PC + 1, SFU(FC + 1).saved_FP <- Reg_Bank[r10], FC <- FC + 1, ##DISCARD IF[1]	C <- SP,	ex_val <- C	Reg_Bank[r10] <- ex_val
	EXIT	FC > 0	IR <- Mem_Inst[PC], PC <- PC + 1	PC <- SFU(FC).ret_PC, A <- Reg_Bank[r10], ##DISCARD IF	SP <- fw(A), C <- SFU(FC).saved_FP, FC <- FC - 1,	ex_val <- C	Reg_Bank[r10] <- ex_val

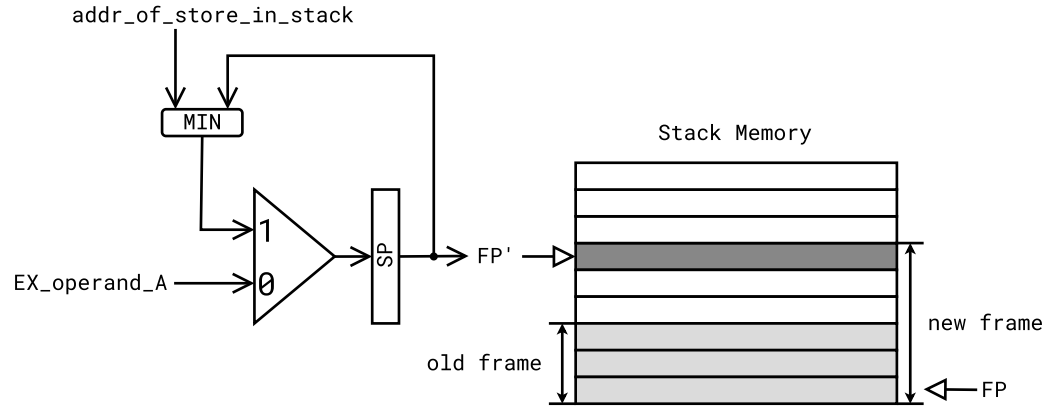


Figura F.1: Infraestructura de *shadow* SP

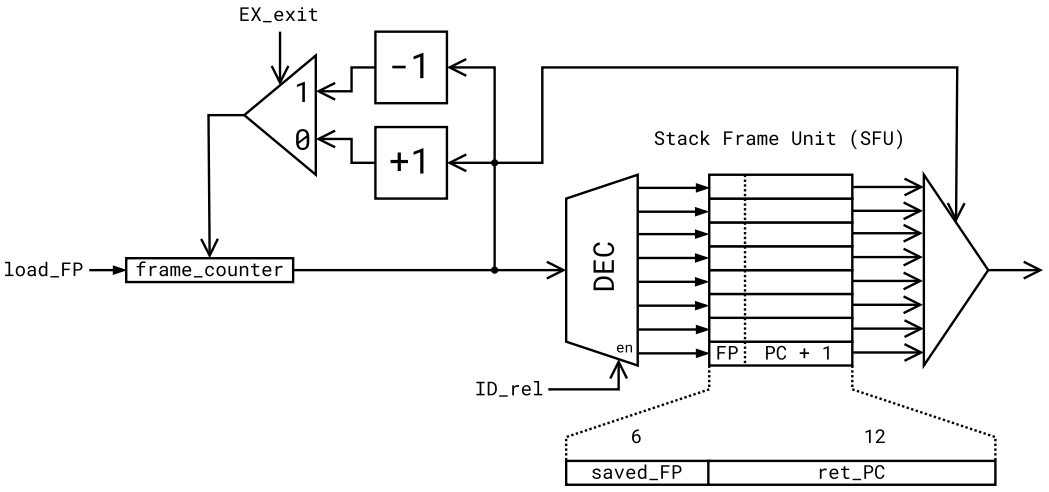


Figura F.2: Sistema de guardado y recuperación del marco de pila

[1]Descartar la siguiente instrucción permite usar la SFU tanto en la etapa ID como EX.

Anexos G

Pruebas de integración

G.1. Programas de prueba

Listado G.1: Test de decodificación (`test_decode.s`)

```
; 64 bit ALU instructions
add  r6 64
add  r6 r4
sub  r6 64
sub  r6 r4
mul  r6 64
mul  r6 r4
div  r6 64
div  r6 r4
sdiv  r6 64
sdiv  r6 r4
or   r6 64
or   r6 r4
and  r6 64
and  r6 r4
lsh  r6 64
lsh  r6 r4
rsh  r6 64
rsh  r6 r4
neg  r6
mod  r6 64
mod  r6 r4
smod  r6 64
smod  r6 r4
xor  r6 64
xor  r6 r4
mov  r6 64
mov  r6 r4
movsx8  r6 64
movsx8  r6 r4
movsx16  r6 64
movsx16  r6 r4
movsx32  r6 64
movsx32  r6 r4
arsh r6 64
arsh r6 r4

; 32 bit ALU instructions
add32  r3 32
add32  r3 r2
sub32  r3 32
sub32  r3 r2
mul32  r3 32
mul32  r3 r2
div32  r3 32
div32  r3 r2
sdiv32  r3 32
sdiv32  r3 r2
```

```
or32   r3 32
or32   r3 r2
and32  r3 32
and32  r3 r2
lsh32  r3 32
lsh32  r3 r2
rsh32  r3 32
rsh32  r3 r2
neg32  r3
mod32  r3 32
mod32  r3 r2
smod32  r3 32
smod32  r3 r2
xor32  r3 32
xor32  r3 r2
mov32  r3 32
mov32  r3 r2
mov32sx8  r3 32
mov32sx8  r3 r2
mov32sx16  r3 32
mov32sx16  r3 r2
arsh32 r3 32
arsh32 r3 r2

; Byteswap instructions
le16 r7
le32 r7
le64 r7
be16 r7
be32 r7
be64 r7
bswap16 r7
bswap32 r7
bswap64 r7

; Atomic operations
addx32     r1 r5 54
addx64     r1 r5 54
andx32     r1 r5 54
andx64     r1 r5 54
orx32      r1 r5 54
orx64      r1 r5 54
xorx32     r1 r5 54
xorx64     r1 r5 54
addfx32    r1 r5 54
addfx64    r1 r5 54
andfx32    r1 r5 54
andfx64    r1 r5 54
orfx32     r1 r5 54
orfx64     r1 r5 54
xorfx32    r1 r5 54
xorfx64    r1 r5 54
xchgx32    r1 r5 54
xchgx64    r1 r5 54
cmpxchgx32 r1 r5 54
cmpxchgx64 r1 r5 54

; Memory instructions
ld64    r7 -10
ldx8    r7 r9 101
ldx16   r7 r9 101
ldx32   r7 r9 101
ldx64   r7 r9 101
ldxs8   r7 r9 101
ldxs16  r7 r9 101
ldxs32  r7 r9 101
st8     r7 101 80
st16    r7 101 80
st32    r7 101 80
st64    r7 101 80
stx8    r7 r9 101
stx16   r7 r9 101
stx32   r7 r9 101
```

```
stx64   r7 r9 101

; 64 bit Jump instructions
ja  L1; Uses offset
L1: jeq  r1 12 L2
L2: jeq  r1 r10 L3
L3: jgt  r1 12 L4
L4: jgt  r1 r10 L5
L5: jge  r1 12 L6
L6: jge  r1 r10 L7
L7: jlt  r1 12 L8
L8: jlt  r1 r10 L9
L9: jle  r1 12 L10
L10: jle  r1 r10 L11
L11: jset r1 12 L12
L12: jset r1 r10 L13
L13: jne  r1 12 L14
L14: jne  r1 r10 L15
L15: jsgt r1 12 L16
L16: jsgt r1 r10 L17
L17: jsge r1 12 L18
L18: jsge r1 r10 L19
L19: jslt r1 12 L20
L20: jslt r1 r10 L21
L21: jsle r1 12 L22
L22: jsle r1 r10 L23
L23: call 12
;rel  12 ; User defined function (Not supported in this processor)

; 32 bit Jump instructions
jal L24 ; Uses immediate
L24: jeq32  r1 12 L25
L25: jeq32  r1 r10 L26
L26: jgt32  r1 12 L27
L27: jgt32  r1 r10 L28
L28: jge32  r1 12 L29
L29: jge32  r1 r10 L30
L30: jlt32  r1 12 L31
L31: jlt32  r1 r10 L32
L32: jle32  r1 12 L33
L33: jle32  r1 r10 L34
L34: jset32 r1 12 L35
L35: jset32 r1 r10 L36
L36: jne32  r1 12 L37
L37: jne32  r1 r10 L38
L38: jsgt32 r1 12 L39
L39: jsgt32 r1 r10 L40
L40: jsge32 r1 12 L41
L41: jsge32 r1 r10 L42
L42: jslt32 r1 12 L43
L43: jslt32 r1 r10 L44
L44: jsle32 r1 12 L45
L45: jsle32 r1 r10 L46
L46:

exit
```

Listado G.2: Test de memoria (`test_mem.s`)

```
; == Test STORE + LOAD ==

        st64 r10 -0x00 327 ;
        ldx64 r0 r10 -0x00 ; r0 = 327


        st32 r10 -0x08 4123;
        ldx32 r1 r10 -0x08 ; r1 = 4123

        st32 r10 -0x0C 543 ;
        ldx32 r2 r10 -0x0C ; r2 = 543


        st16 r10 -0x10 999 ;
```

```
        ldx16 r3 r10 -0x10 ; r3 = 999

        st16 r10 -0x12 66  ;
        ldx16 r4 r10 -0x12 ; r4 = 66

        st16 r10 -0x14 3232;
        ldx16 r5 r10 -0x14 ; r5 = 3232

        st16 r10 -0x16 756 ;
        ldx16 r6 r10 -0x16 ; r6 = 756


        st8 r10 -0x18 11  ;
        ldx8 r0 r10 -0x18 ; r0 = 11

        st8 r10 -0x19 22  ;
        ldx8 r1 r10 -0x19 ; r1 = 22

        st8 r10 -0x1A 33  ;
        ldx8 r2 r10 -0x1A ; r2 = 33

        st8 r10 -0x1B 44  ;
        ldx8 r3 r10 -0x1B ; r3 = 44

        st8 r10 -0x1C 55  ;
        ldx8 r4 r10 -0x1C ; r4 = 55

        st8 r10 -0x1D 66  ;
        ldx8 r5 r10 -0x1D ; r5 = 66

        st8 r10 -0x1E 77  ;
        ldx8 r6 r10 -0x1E ; r6 = 77

        st8 r10 -0x1F 88  ;
        ldx8 r7 r10 -0x1F ; r7 = 88

; == Test STORE + LOADSX ==

        ld64 r9 -327       ;
        stx64 r10 r9 -0x00 ;
        ldx64 r0 r10 -0x00 ; r0 = -327


        ld64 r9 -4123       ;
        stx32 r10 r9 -0x08  ;
        ldxs32 r1 r10 -0x08 ; r1 = -4123

        ld64 r9 -543        ;
        stx32 r10 r9 -0x0C  ;
        ldxs32 r2 r10 -0x0C ; r2 = -543


        ld64 r9 -999        ;
        stx16 r10 r9 -0x10  ;
        ldxs16 r3 r10 -0x10 ; r3 = -999

        ld64 r9 -66         ;
        stx16 r10 r9 -0x12  ;
        ldxs16 r4 r10 -0x12 ; r4 = -66

        ld64 r9 -3232       ;
        stx16 r10 r9 -0x14  ;
        ldxs16 r5 r10 -0x14 ; r5 = -3232

        ld64 r9 -756        ;
        stx16 r10 r9 -0x16  ;
        ldxs16 r6 r10 -0x16 ; r6 = -756


        ld64 r9 -11       ;
        stx8 r10 r9 -0x18  ;
        ldxs8 r0 r10 -0x18 ; r0 = -11

```

```
        ld64 r9 -22         ;
        stx8 r10 r9 -0x19   ;
        ldxs8 r1 r10 -0x19 ; r1 = -22

        ld64 r9 -33         ;
        stx8 r10 r9 -0x1A   ;
        ldxs8 r2 r10 -0x1A ; r2 = -33

        ld64 r9 -44         ;
        stx8 r10 r9 -0x1B   ;
        ldxs8 r3 r10 -0x1B ; r3 = -44

        ld64 r9 -55         ;
        stx8 r10 r9 -0x1C   ;
        ldxs8 r4 r10 -0x1C ; r4 = -55

        ld64 r9 -66         ;
        stx8 r10 r9 -0x1D   ;
        ldxs8 r5 r10 -0x1D ; r5 = -66

        ld64 r9 -77         ;
        stx8 r10 r9 -0x1E   ;
        ldxs8 r6 r10 -0x1E ; r6 = -77

        ld64 r9 -88         ;
        stx8 r10 r9 -0x1F   ;
        ldxs8 r7 r10 -0x1F ; r7 = -88

; == Test STORE + ADD + LOAD ==

        ld64 r8 -24          ;
        st64 r10 -0x20 3     ;
        addx64 r10 r8 -0x20 ;
        ldx64 r0 r10 -0x20   ; r0 = -21

        st32 r10 -0x28 3     ;
        addx32 r10 r8 -0x28 ;
        ldx32 r1 r10 -0x28   ; r1 = 0xFFFFFFFFFFFFFFEB

        st32 r10 -0x2C 3     ;
        addx32 r10 r8 -0x2C ;
        ldxs32 r2 r10 -0x2C ; r2 = -21


        ld64 r3 -53           ;
        st64 r10 -0x30 66     ;
        addfx64 r10 r3 -0x30 ; r3 = 66
        ldx64 r6 r10 -0x30    ; r6 = 13

        ld64 r4 -53           ;
        st32 r10 -0x38 64     ;
        addfx32 r10 r4 -0x38 ; r4 = 64
        ldx32 r7 r10 -0x38    ; r7 = 11

        ld64 r5 -53           ;
        st32 r10 -0x3C 62     ;
        addfx32 r10 r5 -0x3C ; r5 = 62
        ldxs32 r8 r10 -0x3C   ; r8 = 9

; == Test STORE + OR|AND|XOR + LOAD ==

        ; Same operands as in test_alu

        ld64 r9 72057851736457312  ;
        ld64 r8 562950893143040    ;
        stx64 r10 r8 -0x40         ;
        orx64 r10 r9 -0x40         ;
        ldx64 r0 r10 -0x40     ; r0 = 72620802629403744

        ld64 r9 72057851736457312  ;
        ld64 r8 72620544931070976  ;
        stx64 r10 r8 -0x50         ;
        andfx64 r10 r9 -0x50       ; r9 = 72620544931070976
```

```
        ldx64 r0 r10 -0x50    ; r0 = 72057594038124544

        ld64 r9 72057851736457312  ;
        ld64 r8 72620544931070976  ;
        stx64 r10 r8 -0x60         ;
        xorfx64 r10 r9 -0x60       ; r9 = 72620544931070976
        ldx64 r0 r10 -0x60         ; r0 = 563208591279200

; == Test STORE + XCHG|CMPXCHG + LOAD ==

        st64 r10 -0xA0 3     ;
        mov r9 24            ;
        xchgx64 r10 r9 -0xA0 ; r9 = 3
        ldx64 r8 r10 -0xA0   ; r8 = 24
        mov r1 r9 ; (!) wait for xchgx64 to finish MEM stage,
                  ;     then consume r9 from MEM-WB buffer


        st64 r10 -0xA0 7        ;
        mov r9 35               ;
        mov r0 7                ; Equal -> exchange
        cmpxchgx64 r10 r9 -0xA0 ; r0 = 7  (!) get r0 from MEM
        mov r2 r0 ; (!) stop 1 cycle and wait for cmpxchgx64 to finish
                  ;     MEM stage, then consume r0 from MEM-WB buffer
        ldx64 r8 r10 -0xA0      ; r8 = 35

        st64 r10 -0xA0 -32      ;
        mov r0 -34              ; Not Equal -> not exchange
        mov r9 981              ;
        cmpxchgx64 r10 r9 -0xA0 ; r0 = -32   (!) get r0 from WB
        ldx64 r8 r10 -0xA0      ; r8 = -32
        add r0 r1 ; (!) wait for cmpxchgx64 to finish MEM stage, then
                  ;     consume r0 from MEM-WB buffer -> r0 = -32 + 3 = -29

        exit
```

Listado G.3: Test de ALU (`test_alu.s`)

```
; == Test ADD ==

        mov r0 24 ; A
        mov r1 5  ; B
        add r0 r1 ; r0 = 29

        mov r2 -1 ; A
        add r2 1  ; r2 = 0

        mov r3 -67    ; A
        add r3 -2131  ; r3 = -2198

        mov r0 24 ; A
        mov r1 5  ; B
        add32 r0 r1 ; r0 = 29

        mov r2 -1 ; A
        add32 r2 1  ; r2 = 0

        mov r3 -67    ; A
        add32 r3 -2131  ; r3 = 00000000FFFFF76A (-2198)

; == Test SUB ==

        mov r0 24 ; A
        mov r1 5  ; B
        sub r0 r1 ; r0 = 19

        mov r2 -1 ; A
        sub r2 1  ; r2 = -2

        mov r3 -67    ; A
        sub r3 -2131  ; r3 = 2064

        mov r0 24 ; A
```

```
        mov r1 5  ; B
        sub32 r0 r1 ; r0 = 19

        mov r2 -1 ; A
        sub32 r2 1  ; r2 = 00000000FFFFFFFE (-2)

        mov r3 -67    ; A
        sub32 r3 -2131  ; r3 = 2064

; == Test MUL ==

        mov r0 24 ; A
        mov r1 5  ; B
        mul r0 r1 ; r0 = 120

        mov r2 -1 ; A
        mul r2 1  ; r2 = -1

        mov r3 -67    ; A
        mul r3 -2131  ; r3 = 142777

        mov r0 24   ; A
        mov r1 5    ; B
        mul32 r0 r1 ; r0 = 120

        mov r2 -2   ; A
        mul32 r2 1  ; r2 = 00000000FFFFFFFE (-2)

        mov r3 -67      ; A
        mul32 r3 -2131  ; r3 = 142777

; == Test DIV ==

        mov r0 100 ; A
        mov r1 7   ; B
        div r0 r1  ; r0 = 14

        mov r0 100 ; A
        mov r1 -7  ; B
        div r0 r1 ; r0 = 0

        mov r0 -100 ; A
        mov r1 7    ; B
        div r0 r1   ; r0 = 2 635 249 153 387 078 788

        mov r0 -100 ; A
        mov r1 -7   ; B
        div r0 r1   ; r0 = 0

        mov r0 100 ; A
        mov r1 0   ; B
        div r0 r1  ; r0 = 0

        mov r0 -100 ; A
        mov r1 0    ; B
        div r0 r1   ; r0 = 0


        mov r0 100 ; A
        sdiv r0 7  ; r0 = 14

        mov r0 100 ; A
        sdiv r0 -7 ; r0 = -14

        mov r0 -100 ; A
        sdiv r0 7   ; r0 = -14

        mov r0 -100 ; A
        sdiv r0 -7  ; r0 = 14

        mov r0 100 ; A
        sdiv r0 0  ; r0 = 0

```

```
        mov r0 -100 ; A
        sdiv r0 0   ; r0 = 0


        mov r0 100  ; A
        mov r1 7    ; B
        div32 r0 r1 ; r0 = 14

        mov r0 100  ; A
        mov r1 -7   ; B
        div32 r0 r1 ; r0 = 0

        mov r0 -100 ; A
        mov r1 7    ; B
        div32 r0 r1 ; r0 = 613 566 742

        mov r0 -100 ; A
        mov r1 -7   ; B
        div32 r0 r1 ; r0 = 0

        mov r0 100  ; A
        mov r1 0    ; B
        div32 r0 r1 ; r0 = 0

        mov r0 -100 ; A
        mov r1 0    ; B
        div32 r0 r1 ; r0 = 0


        mov r0 100  ; A
        sdiv32 r0 7 ; r0 = 14

        mov r0 100   ; A
        sdiv32 r0 -7 ; r0 = 00000000FFFFFFF2

        mov r0 -100 ; A
        sdiv32 r0 7 ; r0 = 00000000FFFFFFF2

        mov r0 -100  ; A
        sdiv32 r0 -7 ; r0 = 14

        mov r0 100  ; A
        sdiv32 r0 0 ; r0 = 0

        mov r0 -100 ; A
        sdiv32 r0 0 ; r0 = 0

; == Test MOD ==

        mov r0 100 ; A
        mov r1 7   ; B
        mod r0 r1  ; r0 = 2

        mov r0 100 ; A
        mov r1 -7  ; B
        mod r0 r1  ; r0 = 100

        mov r0 -100 ; A
        mov r1 7    ; B
        mod r0 r1   ; r0 = 0

        mov r0 -100 ; A
        mov r1 -7   ; B
        mod r0 r1   ; r0 = -100 (full dividend, as div equals 0)

        mov r0 100 ; A
        mov r1 0   ; B
        mod r0 r1  ; r0 = 100

        mov r0 -100 ; A
        mov r1 0    ; B
        mod r0 r1   ; r0 = -100

```

```
182
183         mov r0 100 ; A
184         smod r0 7 ; r0 = 2
185
186         mov r0 100 ; A
187         smod r0 -7 ; r0 = -2
188
189         mov r0 -100 ; A
190         smod r0 7   ; r0 = -2
191
192         mov r0 -100 ; A
193         smod r0 -7  ; r0 = 2
194
195         mov r0 100 ; A
196         smod r0 0  ; r0 = 100
197
198         mov r0 -100 ; A
199         smod r0 0   ; r0 = -100
200
201
202         mov r0 100  ; A
203         mov r1 7    ; B
204         mod32 r0 r1 ; r0 = 2
205
206         mov r0 100  ; A
207         mov r1 -7   ; B
208         mod32 r0 r1 ; r0 = 100
209
210         mov r0 -100 ; A
211         mov r1 7    ; B
212         mod32 r0 r1 ; r0 = 2
213
214         mov r0 -100 ; A
215         mov r1 -7   ; B
216         mod32 r0 r1 ; r0 = 00000000FFFFFF9C
217
218         mov r0 100  ; A
219         mov r1 0    ; B
220         mod32 r0 r1 ; r0 = 100
221
222         mov r0 -100 ; A
223         mov r1 0    ; B
224         mod32 r0 r1 ; r0 = 00000000FFFFFF9C
225
226
227         mov r0 100  ; A
228         smod32 r0 7 ; r0 = 2
229
230         mov r0 100   ; A
231         smod32 r0 -7 ; r0 = 00000000FFFFFFFE
232
233         mov r0 -100 ; A
234         smod32 r0 7 ; r0 = 00000000FFFFFFFE
235
236         mov r0 -100  ; A
237         smod32 r0 -7 ; r0 = 2
238
239         mov r0 100  ; A
240         smod32 r0 0 ; r0 = 100
241
242         mov r0 -100 ; A
243         smod32 r0 0 ; r0 = 00000000FFFFFF9C    (PC + 1: 163)
244
245
246 ; == Test OR | AND | XOR ==
247
248     ;     00000001000000000000000000011110000000000000000111100000001100000
249     ;  or 00000000000000100000000000000000000111000000000110000010000000000
250     ; ---------------------------------------------------------------------
251     ;     00000001000000100000000000011110000111000000000111100001000110000
252
253         ld64 r9 72057851736457312
254         ld64 r8 562950893143040
```

```
        or r9 r8             ; r9 = 72620802629403744


    ;     0000000100000000000000000001111000000000000000011110000000001100000
    ; and 0000000100000001000000000000000000011100000000011000001000000000000
    ; ---------------------------------------------------------------------------
    ;     0000000100000000000000000000000000000000000001100000000000000000

        ld64 r8 72057851736457312 ;
        ld64 r9 72620544931070976 ;
        and r9 r8            ; r9 = 72057594038124544


    ;     0000000100000000000000000001111000000000000000011110000000001100000
    ; xor 0000000100000001000000000000000000011100000000011000001000000000000
    ; ---------------------------------------------------------------------------
    ;     0000000000000001000000000001111000011100000000100100001000110000

        ld64 r9 72057851736457312 ;
        ld64 r8 72620544931070976 ;
        xor r9 r8            ; r9 = 563208591279200


; == Test LSH | RSH ==

        mov r0 100  ; A
        lsh r0 2    ; r0 = 400

        mov r0 800  ; A
        rsh r0 1027 ; r0 = (800 >> 3) = 100


; == Test ARSH ==

        ld64 r0 0x00000FF0000000FF  ; A
        arsh r0 8    ; r0 = 0x0000000FF0000000

        ld64 r0 0x80000FF0000000FF  ; A
        arsh r0 8    ; r0 = 0xFF80000FF0000000


        ld64 r0 0x00000FF00000FFFF  ; A
        arsh32 r0 8    ; r0 = 0x000000000000000FF

        ld64 r0 0x80000FF0800000FF  ; A
        arsh32 r0 8    ; r0 = 0x00000000FF800000


; == Test MOVSX ==

        movsx8  r6 0xFD         ; r6 = -3
        movsx8  r6 0x7D         ; r6 = 125
        movsx16  r6 0xFFFD      ; r6 = -3
        movsx16  r6 0x7FFD      ; r6 = 32 765
        movsx32  r6 0xFFFFFFFD  ; r6 = -3
        movsx32  r6 0x700000FD  ; r6 = 1 879 048 445

        mov32sx8  r6 0xFD         ; r6 = 00000000FFFFFFFD
        mov32sx8  r6 0x7D         ; r6 = 125
        mov32sx16  r6 0xFFFD      ; r6 = 00000000FFFFFFFD
        mov32sx16  r6 0x7FFD      ; r6 = 32 762


; == Test END ==

        ld64 r7 0x0102030405060708 ;
        bswap16 r7 ; r7 = 0000000000000807

        ld64 r7 0x0102030405060708 ;
        bswap32 r7 ; r7 = 0000000008070605

        ld64 r7 0x0102030405060708 ;
        bswap64 r7 ; r7 = 0807060504030201
```

```
328
329
330         ld64 r7 0x0102030405060708 ;
331         be16 r7 ; r7 = 0000000000000807
332
333         ld64 r7 0x0102030405060708 ;
334         be32 r7 ; r7 = 0000000008070605
335
336         ld64 r7 0x0102030405060708 ;
337         be64 r7 ; r7 = 0807060504030201
338
339         ld64 r7 0x0102030405060708 ;
340         le16 r7 ; do nothing
341         le32 r7 ; do nothing
342         le64 r7 ; do nothing
343
344         exit
```

Listado G.4: Test de control (`test_branch.s`)

```
1  ; == Test Jump Always ==
2  L0:
3          ja  L1 ; Jump Always
4          mov r0 -1
5          mov r1 -1
6          mov r2 -1
7
8  L1:
9          jal  L2 ; Jump Always (Long)
10         mov r0 -1
11         mov r1 -1
12         mov r2 -1
13
14 ; == Test EQ ==
15 L2:
16         mov r8 23
17         jeq r8 24 L2 ; Not taken (if fail infinite loop)
18         mov r9 23
19         jeq r8 r9 L3 ; Taken
20         mov r0 -1
21         mov r1 -1
22         mov r2 -1
23
24 ; == Test NE ==
25 L3:
26         mov r8 23
27         jne r8 23 L3 ; Not taken
28         mov r9 24
29         jne r8 r9 L4 ; Taken
30         mov r0 -1
31         mov r1 -1
32         mov r2 -1
33
34 ; == Test GT ==
35 L4:
36         mov r8 23
37         jgt r8 23 L4 ; Not taken
38
39         mov r8 14
40         mov r9 15
41         jgt r8 r9 L4 ; Not taken
42
43         mov r8 16
44         mov r9 7
45         jgt r8 r9 L5 ; Taken
46         mov r0 -1
47         mov r1 -1
48         mov r2 -1
49
50 ; == Test GE ==
51 L5:
52         mov r8 23
53         jge r8 23 L6 ; Taken
```

```
        mov r0 -1
        mov r1 -1
        mov r2 -1
L6:
        mov r8 14
        mov r9 15
        jge r8 r9 L6 ; Not taken

        mov r8 16
        mov r9 7
        jge r8 r9 L7 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1

; == Test SGT ==
L7:
        mov r8 23
        jsgt r8 23 L7 ; Not taken

        mov r8 14
        mov r9 15
        jsgt r8 r9 L7 ; Not taken

        mov r8 16
        mov r9 7
        jsgt r8 r9 L8 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1

L8:
        mov r8 -23
        mov r9 -17
        jsgt r8 r9 L8 ; Not taken

        mov r8 -42
        jsgt r8 -52 L9 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1


; == Test SGE ==
L9:
        mov r8 23
        jsge r8 23 L10 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1
L10:
        mov r8 14
        mov r9 15
        jsge r8 r9 L10 ; Not taken

        mov r8 16
        mov r9 7
        jsge r8 r9 L11 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1

L11:
        mov r8 -23
        mov r9 -17
        jsge r8 r9 L11 ; Not taken

        mov r8 -42
        jsge r8 -52 L12 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1

```

```
; == Test LT ==
L12:
        mov r8 23
        jlt r8 23 L12 ; Not taken

        mov r8 14
        mov r9 15
        jlt r8 r9 L13 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1

L13:
        mov r8 16
        mov r9 7
        jlt r8 r9 L13 ; Not taken

; == Test LE ==
L14:
        mov r8 23
        jle r8 23 L15 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1
L15:
        mov r8 14
        mov r9 15
        jle r8 r9 L16 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1
L16:
        mov r8 16
        mov r9 7
        jle r8 r9 L16 ; Not taken

; == Test SLT ==
L17:
        mov r8 23
        jslt r8 23 L18 ; Not taken

        mov r8 14
        mov r9 15
        jslt r8 r9 L18 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1
L18:
        mov r8 16
        mov r9 7
        jslt r8 r9 L18 ; Not taken


L19:
        mov r8 -23
        mov r9 -17
        jslt r8 r9 L20 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1
L20:
        mov r8 -42
        jslt r8 -52 L20 ; Not taken

; == Test SLE ==
L21:
        mov r8 23
        jsle r8 23 L22 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1
L22:
        mov r8 14
```

```
        mov r9 15
        jsle r8 r9 L23 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1
L23:
        mov r8 16
        mov r9 7
        jsle r8 r9 L23 ; Not taken

L24:
        mov r8 -23
        mov r9 -17
        jslt r8 r9 L25 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1
L25:
        mov r8 -42
        jslt r8 -52 L25 ; Not taken

        ja L26


; == END OF TEST ==
L_end:
        exit
        mov r0 -2
        mov r1 -2
        mov r2 -2

; == SET ==
L26:
        mov r8 23
        jset r8 1 L27 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1
L27:
        mov r8 8
        mov r9 4
        jset r8 r9 L27; Not taken

        mov r8 -1
        mov r9 75
        jset r8 r9 L28; ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1


; == 32 bit cases ==

; == Test SGT ==
L28:
        mov32 r8 -23
        mov32 r9 -17
        jsgt32 r8 r9 L28 ; Not taken

        mov32 r8 -42
        jsgt32 r8 -52 L29 ; Taken
        mov r0 -1
        mov r1 -1
        mov r2 -1


; == Test SGE ==
L29:
        mov32 r8 -23
        mov32 r9 -17
        jsge r8 r9 L29 ; Not taken

        mov32 r8 -42
```

```
273         jsge32 r8 -52 L30 ; Taken
274         mov r0 -1
275         mov r1 -1
276         mov r2 -1
277
278 ; == Test SLT ==
279 L30:
280         mov32 r8 -23
281         mov32 r9 -17
282         jslt32 r8 r9 L31 ; Taken
283         mov r0 -1
284         mov r1 -1
285         mov r2 -1
286 L31:
287         mov32 r8 -42
288         jslt32 r8 -52 L31 ; Not taken
289
290 ; == Test SLE ==
291 L32:
292         mov32 r8 -23
293         mov32 r9 -17
294         jslt32 r8 r9 L33 ; Taken
295         mov r0 -1
296         mov r1 -1
297         mov r2 -1
298 L33:
299         mov32 r8 -42
300         jslt32 r8 -52 L33 ; Not taken
301
302
303
304 ; == some edge cases ==
305
306         ; taken branch after taken branch
307         jeq r0 r0 L34
308         jeq r1 r1 L_error
309
310 L34:
311         ; unconditional jump after taken branch
312         jeq r2 r2 L35
313         jeq r3 r3 L_error
314
315 L35:
316         ja L_end
317
318 L_error:
319         mov r0 -1
320         mov r1 -1
321         mov r2 -1
322
323         ; Program ends at tag L_end
324         ; If this is reached exception will be generated
325         call -1
```

Listado G.5: Test de HFU (`test_call.s`)

```
1  ; Warning! This test only works with "hfu/Test_HFU.vhd" instead of actual
2  ; HFU entity. In order to test it using "launch-core-testbench.sh" it
3  ; has to be executed using --hfu=Test_HFU option
4
5          mov r1 0x000A0000
6          mov r2 0x0000B000
7          mov r3 0x00000C00
8          mov r4 0x000000D0
9          mov r5 0x0000000F
10
11         call 0 ; does not receive parameters -> r0 = -1
12
13         call 1 ; receives 1 parameter -> r0 = r1 = 0x000A0000
14
15         mov r8 16
16         div r8 4 ; Test if stays on INIT_S state
17         call 2 ; receives 2 parameters -> r0 = r1|r2 = 0x000AB000
```

```
18
19          mov r8 16
20          st8 r10 0x00 2 ; Test if stays on FUNCTION3_EX_S
21                         ; tate while st8 blockes pipeline
22          call 3 ; receives 3 parameters -> r0 = r1|r2|r3 = 0x000ABC00
23
24          call 4 ; receives 4 parameters -> r0 = r1|r2|r3|r4 = 0x000ABCD0
25
26          call 5 ; receives 5 parameters -> r0 = r1|r2|r3|r4|r5 = 0x000ABCDF
27
28
29  ;-- 1 parameter dependency ---------------------------------------------------
30          mov r1 0x00010000
31          call 1 ; r0 = 0x00010000 ; get r1 from MEM, don't stop
32          mov r8 r0 ; get r0 from WB, don't stop
33          mov r9 r0 ; get r0 from RegBank, don't stop
34
35          st64 r10 0x00 0x00020000
36          ldx64 r1 r10 0x00
37          call 1 ; r0 = 0x00020000
38                 ; get r1 from WB, stop on ID stage while ldx64 is on MEM
39
40          mov r1 0x00030000
41          st64 r10 0x00 0xFFFFFFFF
42          ldx64 r2 r10 0x00
43          call 1 ; r0 = 0x00030000
44                 ; get r1 from RegBank, don't stop because of r2
45
46  ;-- 2 parameters dependency --------------------------------------------------
47          mov r2 0x0000F000
48
49          mov r1 0x00010000
50          call 2 ; r0 = 0x0001F000
51                 ; get r1 from MEM, don't stop
52          mov r8 r0 ; get r0 from WB, don't stop
53          mov r9 r0 ; get r0 from RegBank, don't stop
54
55          st64 r10 0x00 0x00020000
56          ldx64 r1 r10 0x00
57          call 2 ; r0 = 0x0002F000
58                 ; get r1 from WB, stop on ID stage while ldx64 is on MEM
59
60          mov r1 0x00030000
61          st64 r10 0x00 0xFFFFFFFF
62          ldx64 r3 r10 0x00
63          call 2 ; r0 = 0x0003F000
64                 ; get r1 from RegBank, don't stop because of r3
65
66
67          mov r1 0x000F0000
68
69          mov r2 0x00001000
70          call 2 ; r0 = 0x000F1000
71                 ; get r2 from MEM, don't stop
72          mov r8 r0 ; get r0 from WB, don't stop
73          mov r9 r0 ; get r0 from RegBank, don't stop
74
75          st64 r10 0x00 0x00002000
76          ldx64 r2 r10 0x00
77          call 2 ; r0 = 0x000F2000
78                 ; get r2 from WB, stop on ID stage while ldx64 is on MEM
79
80          mov r2 0x00003000
81          st64 r10 0x00 0xFFFFFFFF
82          ldx64 r3 r10 0x00
83          call 2 ; r0 = 0x000F3000
84                 ; get r2 from RegBank, don't stop because of r3
85
86
87  ;-- 3 parameters dependency --------------------------------------------------
88          mov r2 0x0000F000
89          mov r3 0x00000F00
90
```

```
91          mov r1 0x00010000
92          call 3 ; r0 = 0x0001FF00
93                 ; get r1 from MEM, don't stop
94          mov r8 r0 ; get r0 from WB, don't stop
95          mov r9 r0 ; get r0 from RegBank, don't stop
96
97          st64 r10 0x00 0x00020000
98          ldx64 r1 r10 0x00
99          call 3 ; r0 = 0x0002FF00
100                ; get r1 from WB, stop on ID stage while ldx64 is on MEM
101
102         mov r1 0x00030000
103         st64 r10 0x00 0xFFFFFFFF
104         ldx64 r4 r10 0x00
105         call 3 ; r0 = 0x0003FF00
106                ; get r1 from RegBank, don't stop because of r4
107
108
109         mov r1 0x000F0000
110         mov r3 0x00000F00
111
112         mov r2 0x00001000
113         call 3 ; r0 = 0x000F1F00
114                ; get r2 from MEM, don't stop
115         mov r8 r0 ; get r0 from WB, don't stop
116         mov r9 r0 ; get r0 from RegBank, don't stop
117
118         st64 r10 0x00 0x00002000
119         ldx64 r2 r10 0x00
120         call 3 ; r0 = 0x000F2F00
121                ; get r2 from WB, stop on ID stage while ldx64 is on MEM
122
123         mov r2 0x00003000
124         st64 r10 0x00 0xFFFFFFFF
125         ldx64 r4 r10 0x00
126         call 3 ; r0 = 0x000F3F00
127                ; get r2 from RegBank, don't stop because of r4
128
129
130         mov r1 0x000F0000
131         mov r2 0x0000F000
132
133         mov r3 0x00000100
134         call 3 ; r0 = 0x000FF100
135                ; get r3 from MEM, don't stop
136         mov r8 r0 ; get r0 from WB, don't stop
137         mov r9 r0 ; get r0 from RegBank, don't stop
138
139         st64 r10 0x00 0x00000200
140         ldx64 r3 r10 0x00
141         call 3 ; r0 = 0x000FF200
142                ; get r3 from WB, stop on ID stage while ldx64 is on MEM
143
144         mov r3 0x00000300
145         st64 r10 0x00 0xFFFFFFFF
146         ldx64 r4 r10 0x00
147         call 3 ; r0 = 0x000FF300
148                ; get r3 from RegBank, don't stop because of r4
149
150 ;-- 4 parameters dependency ----------------------------------------------------
151         mov r2 0x0000F000
152         mov r3 0x00000F00
153         mov r4 0x000000F0
154
155         mov r1 0x00010000
156         call 4 ; r0 = 0x0001FFF0
157                ; get r1 from MEM, don't stop
158         mov r8 r0 ; get r0 from WB, don't stop
159         mov r9 r0 ; get r0 from RegBank, don't stop
160
161         st64 r10 0x00 0x00020000
162         ldx64 r1 r10 0x00
163         call 4 ; r0 = 0x0002FFF0
```

```
164                 ; get r1 from WB, stop on ID stage while ldx64 is on MEM
165
166         mov r1 0x00030000
167         st64 r10 0x00 0xFFFFFFFF
168         ldx64 r5 r10 0x00
169         call 4 ; r0 = 0x0003FFF0
170                ; get r1 from RegBank, don't stop because of r5
171
172
173         mov r1 0x000F0000
174         mov r3 0x00000F00
175         mov r4 0x000000F0
176
177         mov r2 0x00001000
178         call 4 ; r0 = 0x000F1FF0
179                ; get r2 from MEM, don't stop
180         mov r8 r0 ; get r0 from WB, don't stop
181         mov r9 r0 ; get r0 from RegBank, don't stop
182
183         st64 r10 0x00 0x00002000
184         ldx64 r2 r10 0x00
185         call 4 ; r0 = 0x000F2FF0
186                ; get r2 from WB, stop on ID stage while ldx64 is on MEM
187
188         mov r2 0x00003000
189         st64 r10 0x00 0xFFFFFFFF
190         ldx64 r5 r10 0x00
191         call 4 ; r0 = 0x000F3FF0
192                ; get r2 from RegBank, don't stop because of r5
193
194
195         mov r1 0x000F0000
196         mov r2 0x0000F000
197         mov r4 0x000000F0
198
199         mov r3 0x00000100
200         call 4 ; r0 = 0x000FF1F0
201                ; get r3 from MEM, don't stop
202         mov r8 r0 ; get r0 from WB, don't stop
203         mov r9 r0 ; get r0 from RegBank, don't stop
204
205         st64 r10 0x00 0x00000200
206         ldx64 r3 r10 0x00
207         call 4 ; r0 = 0x000FF2F0
208                ; get r3 from WB, stop on ID stage while ldx64 is on MEM
209
210         mov r3 0x00000300
211         st64 r10 0x00 0xFFFFFFFF
212         ldx64 r5 r10 0x00
213         call 4 ; r0 = 0x000FF3F0
214                ; get r3 from RegBank, don't stop because of r5
215
216
217         mov r1 0x000F0000
218         mov r2 0x0000F000
219         mov r3 0x00000F00
220
221         mov r4 0x00000010
222         call 4 ; r0 = 0x000FFF10
223                ; get r4 from RegBank, stop while mov is on EX stage
224         mov r8 r0 ; get r0 from WB, don't stop
225         mov r9 r0 ; get r0 from RegBank, don't stop
226
227         st64 r10 0x00 0x00000020
228         ldx64 r4 r10 0x00
229         call 4 ; r0 = 0x000FFF20
230                ; get r4 from WB, stop on ID stage while ldx64 is on MEM
231
232         mov r4 0x00000030
233         st64 r10 0x00 0xFFFFFFFF
234         ldx64 r5 r10 0x00
235         call 4 ; r0 = 0x000FFF30
236                ; get r4 from RegBank, don't stop because of r5
```

```
237
238 ;-- 5 parameters dependency ----------------------------------------------------
239         mov r2 0x0000F000
240         mov r3 0x00000F00
241         mov r4 0x000000F0
242         mov r5 0x0000000F
243
244         mov r1 0x00010000
245         call 5 ; r0 = 0x0001FFF0
246                ; get r1 from MEM, don't stop
247         mov r8 r0 ; get r0 from WB, don't stop
248         mov r9 r0 ; get r0 from RegBank, don't stop
249
250         st64 r10 0x00 0x00020000
251         ldx64 r1 r10 0x00
252         call 5 ; r0 = 0x0002FFFF
253                ; get r1 from WB, stop on ID stage while ldx64 is on MEM
254
255         mov r1 0x00030000
256         st64 r10 0x00 0xFFFFFFFF
257         ldx64 r6 r10 0x00
258         call 5 ; r0 = 0x0003FFFF
259                ; get r1 from RegBank, don't stop because of r6
260
261
262         mov r1 0x000F0000
263         mov r3 0x00000F00
264         mov r4 0x000000F0
265         mov r5 0x0000000F
266
267         mov r2 0x00001000
268         call 5 ; r0 = 0x000F1FFF
269                ; get r2 from MEM, don't stop
270         mov r8 r0 ; get r0 from WB, don't stop
271         mov r9 r0 ; get r0 from RegBank, don't stop
272
273         st64 r10 0x00 0x00002000
274         ldx64 r2 r10 0x00
275         call 5 ; r0 = 0x000F2FFF
276                ; get r2 from WB, stop on ID stage while ldx64 is on MEM
277
278         mov r2 0x00003000
279         st64 r10 0x00 0xFFFFFFFF
280         ldx64 r6 r10 0x00
281         call 5 ; r0 = 0x000F3FFF
282                ; get r2 from RegBank, don't stop because of r6
283
284
285         mov r1 0x000F0000
286         mov r2 0x0000F000
287         mov r4 0x000000F0
288         mov r5 0x0000000F
289
290         mov r3 0x00000100
291         call 5 ; r0 = 0x000FF1FF
292                ; get r3 from MEM, don't stop
293         mov r8 r0 ; get r0 from WB, don't stop
294         mov r9 r0 ; get r0 from RegBank, don't stop
295
296         st64 r10 0x00 0x00000200
297         ldx64 r3 r10 0x00
298         call 5 ; r0 = 0x000FF2FF
299                ; get r3 from WB, stop on ID stage while ldx64 is on MEM
300
301         mov r3 0x00000300
302         st64 r10 0x00 0xFFFFFFFF
303         ldx64 r6 r10 0x00
304         call 5 ; r0 = 0x000FF3FF
305                ; get r3 from RegBank, don't stop because of r6
306
307
308         mov r1 0x000F0000
309         mov r2 0x0000F000
```

```
310         mov r3 0x00000F00
311         mov r5 0x0000000F
312
313         mov r4 0x00000010
314         call 5 ; r0 = 0x000FFF1F
315                ; get r4 from RegBank, stop while mov is on EX stage
316         mov r8 r0 ; get r0 from WB, don't stop
317         mov r9 r0 ; get r0 from RegBank, don't stop
318
319         st64 r10 0x00 0x00000020
320         ldx64 r4 r10 0x00
321         call 5 ; r0 = 0x000FFF2F
322                ; get r4 from WB, stop on ID stage while ldx64 is on MEM
323
324         mov r4 0x00000030
325         st64 r10 0x00 0xFFFFFFFF
326         ldx64 r6 r10 0x00
327         call 5 ; r0 = 0x000FFF3F
328                ; get r4 from RegBank, don't stop because of r6
329
330
331         mov r1 0x000F0000
332         mov r2 0x0000F000
333         mov r3 0x00000F00
334         mov r4 0x000000F0
335
336         mov r5 0x00000001
337         call 5 ; r0 = 0x000FFFF1
338                ; get r5 from RegBank, stop while mov is on EX stage
339         mov r8 r0 ; get r0 from WB, don't stop
340         mov r9 r0 ; get r0 from RegBank, don't stop
341
342         st64 r10 0x00 0x00000002
343         ldx64 r5 r10 0x00
344         call 5 ; r0 = 0x000FFFF2
345                ; get r5 from WB, stop on ID stage while ldx64 is on MEM
346
347         mov r5 0x00000003
348         st64 r10 0x00 0xFFFFFFFF
349         ldx64 r6 r10 0x00
350         call 5 ; r0 = 0x000FFFF3
351                ; get r5 from RegBank, don't stop because of r6
352
353
354         call -1 ; generates exception in MEM stage
```

Listado G.6: Test de IOMM (`test_io_mem.s`)

```
1  ; This program copies content from BPF_MEM_PACKET_BASE (h8000) up to
2  ; BPF_FRAME_POINTER (h89F8) into shared memory (h9000).
3
4  ; It requires being tested using test_io_mem.vhd testbench, which includes:
5
6
7  ;       Control test:
8  ;           - Write control signals
9  ;           - Read control signals, input and output
10
11 ;       Concurrency test:
12 ;           - Concurrent 1 step write
13 ;           - Concurrent 1 step read
14
15 ;           - Concurrent flush and write
16 ;           - Concurrent flush and read
17
18 ;       Buffered read test:
19 ;           - Read lower 32b + higher 32b of same addr
20 ;           - Read higher 32b + lower 32b of same addr
21 ;           - Read lower 32b + lower 32b of same addr
22 ;           - Read higher 32b + higher 32b of same addr
23
24 ;           - Read lower 32b + higher 32b of different addr
25 ;           - Read higher 32b + lower 32b of different addr
```

```
;           - Read lower 32b + lower 32b of different addr
;           - Read higher 32b + higher 32b of different addr

;       Buffered write test:
;           - Write lower 32b + higher 32b of same addr
;           - Write higher 32b + lower 32b of same addr
;           - Write lower 32b + lower 32b of same addr
;           - Write higher 32b + higher 32b of same addr

;           - Write lower 32b + higher 32b of different addr
;           - Write higher 32b + lower 32b of different addr
;           - Write lower 32b + lower 32b of different addr
;           - Write higher 32b + higher 32b of different addr

;       Forced flushed test:
;           - Write 32b + FLUSH
;           - Write <32b + FLUSH
;           - Read lower 32b + FLUSH + Read higher 32b (same addr)


        mov r0 0x8000

L1:     ldx64 r1 r0 0x0000
        stx64 r0 r1 0x1000

        add r0 8
        jlt r0 0x8A00 L1


        mov r0 0x9000
        ld64 r2 0x0001000000010000

L2:     addx64 r0 r2 0x0000 ; Add both halves

        add r0 8
        jlt r0 0x9A00 L2

        exit
```

Listado G.7: Test de mapas (`test_map.s`)

```
; This test requires being tested using test_map.vhd testbench

        ; Map with 32-bit values
        mov r9 0
loop:
        mov r1 1    ; map id  <- 1
        mov r2 r9   ; map key <- 0..100
        call 0      ; lookup_elem(1, i);

        jeq r0 0 null

        ldx32 r3 r0 0x00
        ldx32 r4 r0 0x04 ; next elem

        ja loop_guard
null:
        mov r3 -1
        mov r4 -1

loop_guard:
        add r9 1
        jlt r9 100 loop

        mov r1 1    ;
        mov r2 100  ; ! out of bounds
        call 0      ; lookup_elem(0, 100) -> NULL

        mov r1 0    ; ! map 0 should not exist
        mov r2 45   ;
        call 0      ; lookup_elem(0, 45) -> NULL

;-- Dummy Loop to change map settings --;
```

```
        mov r9 0
dummy_loop_0:
        le64 r7 ; NOOP
        le64 r7 ; NOOP
        add r9 1
        jlt r9 10 dummy_loop_0
;----------------------;

        ; Map with 8-bit values
        mov r1 0    ; ! map 0 should have replaced map 1 (same data)
        mov r2 45   ;
        call 0      ; lookup_elem(0, 45)

        ldx32 r3 r0 0x00
        ldx32 r4 r0 0x04 ; next elem

;-- Dummy Loop to change map settings --;
        mov r9 0
dummy_loop_1:
        le64 r7 ; NOOP
        le64 r7 ; NOOP
        add r9 1
        jlt r9 10 dummy_loop_1
;----------------------;

        ; Map with 16-bit values
        mov r1 0    ;
        mov r2 45   ;
        call 0      ; lookup_elem(0, 45)

        ldx32 r3 r0 0x00
        ldx32 r4 r0 0x04 ; next elem

;-- Dummy Loop to change map settings --;
        mov r9 0
dummy_loop_2:
        le64 r7 ; NOOP
        le64 r7 ; NOOP
        add r9 1
        jlt r9 10 dummy_loop_2
;----------------------;

        ; Map with 64-bit values
        mov r1 0    ;
        mov r2 45   ;
        call 0      ; lookup_elem(0, 45)

        ldx64 r3 r0 0x00
        ldx64 r4 r0 0x04 ; next elem


        mov r1 1    ; ! map 1 should not exist
        mov r2 66   ;
        call 0      ; lookup_elem(0, 66) -> NULL

        ; END WITH EXCEPTION: derreferencing NULL pointer
        ldx32 r3 r0 0x00
```

G.2. Ejemplo de *testbench*

Listado G.8: *Testbench* que permite simular la carga y ejecución de un programa BPF (`program_test.vhd`)

```
--------------------------------------------------------------------------------
-- Project Name: A basic processor core for running BPF programs
-- Author:       Fernando Lahoz Bernad
--
-- Description:  Testbench for testing loading of a BPF program and it's
--               execution.
--------------------------------------------------------------------------------

```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

use work.bpf.all;

entity BPF_Peripheral_Testbench is
end BPF_Peripheral_Testbench;

architecture Behavioral of BPF_Peripheral_Testbench is

    component BPF_AXI_Peripheral is
        port (
            -- AXI slave interface
            S_AXI_aclk : in std_logic;
            S_AXI_aresetn : in std_logic; -- This Signal is Active LOW

            S_AXI_awaddr : in std_logic_vector(31 downto 0);
            S_AXI_awprot : in std_logic_vector(2 downto 0);
            S_AXI_awvalid : in std_logic;
            S_AXI_awready : out std_logic;

            S_AXI_wdata : in std_logic_vector(31 downto 0);
            S_AXI_wstrb : in std_logic_vector(3 downto 0);
            S_AXI_wvalid : in std_logic;
            S_AXI_wready : out std_logic;

            S_AXI_bresp : out std_logic_vector(1 downto 0);
            S_AXI_bvalid : out std_logic;
            S_AXI_bready : in std_logic;

            S_AXI_araddr : in std_logic_vector(31 downto 0);
            S_AXI_arprot : in std_logic_vector(2 downto 0);
            S_AXI_arvalid : in std_logic;
            S_AXI_arready : out std_logic;

            S_AXI_rdata : out std_logic_vector(31 downto 0);
            S_AXI_rresp : out std_logic_vector(1 downto 0);
            S_AXI_rvalid : out std_logic;
            S_AXI_rready : in std_logic
        );
    end component;

    -- RAM from which instructions are being loaded
    component Inst_RAM is
        port (
            clk : in std_logic;
            addr : in std_logic_vector (11 downto 0);
            input : in std_logic_vector (63 downto 0);
            write_en : in std_logic;
            read_en : in std_logic;
            output : out std_logic_vector (63 downto 0)
        );
    end component;

    function vec32(input_vector: std_logic_vector) return std_logic_vector is
        constant target_length : integer := 32;
        variable output_vector : std_logic_vector(target_length-1 downto 0);
        variable input_length  : integer := input_vector'length;
    begin
        if input_length < target_length then
            -- Zero-extend the input vector to 32 bits
            output_vector := (others => '0');
            output_vector(input_length-1 downto 0) := input_vector;
        else
            -- Truncate or keep the input vector to 32 bits
            output_vector := input_vector(target_length-1 downto 0);
        end if;
        return output_vector;
    end function;

    constant CLK_PERIOD : time := 10 ns;
```

```
82      constant NUM_CLKS : natural := 10000;
83
84      signal clk, reset : std_logic;
85      signal cycles : natural := 0;
86
87      signal instruction : std_logic_vector(63 downto 0);
88      signal inst_addr : std_logic_vector(11 downto 0);
89
90      signal S_AXI_awaddr : std_logic_vector(31 downto 0);
91      signal S_AXI_awprot : std_logic_vector(2 downto 0);
92      signal S_AXI_awvalid : std_logic;
93      signal S_AXI_awready : std_logic;
94
95      signal S_AXI_wdata : std_logic_vector(31 downto 0);
96      signal S_AXI_wstrb : std_logic_vector(3 downto 0);
97      signal S_AXI_wvalid : std_logic;
98      signal S_AXI_wready : std_logic;
99
100     signal S_AXI_bresp : std_logic_vector(1 downto 0);
101     signal S_AXI_bvalid : std_logic;
102     signal S_AXI_bready : std_logic;
103
104     signal S_AXI_araddr: std_logic_vector(31 downto 0);
105     signal S_AXI_arprot : std_logic_vector(2 downto 0);
106     signal S_AXI_arvalid : std_logic;
107     signal S_AXI_arready : std_logic;
108
109     signal S_AXI_rdata : std_logic_vector(31 downto 0);
110     signal S_AXI_rresp : std_logic_vector(1 downto 0);
111     signal S_AXI_rvalid : std_logic;
112     signal S_AXI_rready : std_logic;
113
114 begin
115
116     tested_unit: BPF_AXI_Peripheral
117         port map (
118             S_AXI_aclk => clk,
119             S_AXI_aresetn => not reset,
120
121             S_AXI_awaddr => S_AXI_awaddr,
122             S_AXI_awprot => S_AXI_awprot,
123             S_AXI_awvalid => S_AXI_awvalid,
124             S_AXI_awready => S_AXI_awready,
125
126             S_AXI_wdata => S_AXI_wdata,
127             S_AXI_wstrb => S_AXI_wstrb,
128             S_AXI_wvalid => S_AXI_wvalid,
129             S_AXI_wready => S_AXI_wready,
130
131             S_AXI_bresp => S_AXI_bresp,
132             S_AXI_bvalid => S_AXI_bvalid,
133             S_AXI_bready => S_AXI_bready,
134
135             S_AXI_araddr => S_AXI_araddr,
136             S_AXI_arprot => S_AXI_arprot,
137             S_AXI_arvalid => S_AXI_arvalid,
138             S_AXI_arready => S_AXI_arready,
139
140             S_AXI_rdata => S_AXI_rdata,
141             S_AXI_rresp => S_AXI_rresp,
142             S_AXI_rvalid => S_AXI_rvalid,
143             S_AXI_rready => S_AXI_rready
144         );
145
146     program_buffer: Inst_RAM
147         port map (
148             clk => clk,
149             addr => inst_addr,
150             input => (63 downto 0 => '0'),
151             write_en => '0',
152             read_en => '1',
153             output => instruction
154         );
```

```
155
156     CLK_PROC: process
157     begin
158         if (cycles /= NUM_CLKS) then
159             clk <= '0';
160             wait for CLK_PERIOD / 2;
161             clk <= '1';
162             cycles <= cycles + 1;
163             wait for CLK_PERIOD / 2;
164         else
165             wait;
166         end if;
167     end process;
168
169     ------------------------------------------------------------------------
170
171     TEST_PROC: process
172     begin
173         reset <= '1';
174
175         S_AXI_awaddr <= x"00000000";
176         S_AXI_awvalid <= '0';
177
178         S_AXI_wdata <= x"00000000";
179         S_AXI_wstrb <= "0000";
180         S_AXI_wvalid <= '0';
181
182         S_AXI_araddr <= x"00000000";
183         S_AXI_arvalid <= '0';
184
185         -- Always ready
186         S_AXI_bready <= '1';
187         S_AXI_rready <= '1';
188
189         -- Unused
190         S_AXI_arprot <= "000";
191         S_AXI_awprot <= "000";
192
193         wait on clk until clk = '1';
194         reset <= '0';
195
196         -- Step 1: load program --
197         inst_addr <= x"000";
198         wait for CLK_PERIOD / 8;
199         while instruction /= (63 downto 0 => '0') loop
200             -- lower 32 bit
201             S_AXI_awaddr <= BPF_MEM_INST_BASE_U32 + (inst_addr & "000");
202             S_AXI_awvalid <= '1';
203
204             S_AXI_wdata <= instruction(31 downto 0);
205             S_AXI_wstrb <= "1111";
206             S_AXI_wvalid <= '1';
207
208             wait until S_AXI_awready = '1' and S_AXI_wready = '1';
209             wait on clk until clk = '1';
210
211             S_AXI_awvalid <= '0';
212             S_AXI_wvalid <= '0';
213
214             wait until S_AXI_bvalid = '1';
215             wait on clk until clk = '1';
216
217             -- higher 32 bit
218             S_AXI_awaddr <= BPF_MEM_INST_BASE_U32 + (inst_addr & "100");
219             S_AXI_awvalid <= '1';
220
221             S_AXI_wdata <= instruction(63 downto 32);
222             S_AXI_wstrb <= "1111";
223             S_AXI_wvalid <= '1';
224
225             wait until S_AXI_awready = '1' and S_AXI_wready = '1';
226             wait on clk until clk = '1';
227
```

```
228             S_AXI_awvalid <= '0';
229             S_AXI_wvalid <= '0';
230 
231             wait until S_AXI_bvalid = '1';
232             wait on clk until clk = '1';
233 
234             inst_addr <= inst_addr + "01";
235             wait for CLK_PERIOD / 8;
236 
237         end loop;
238 
239         -- Step 2: load frame pointer (FP) --
240 
241         S_AXI_awaddr <= BPF_CORE_CTRL_U32;
242         S_AXI_awvalid <= '1';
243 
244         S_AXI_wdata <= vec32("000011" & "1010"); -- reset = 0; sleep = 1; write FP
245         S_AXI_wstrb <= "1111";
246         S_AXI_wvalid <= '1';
247 
248         wait until S_AXI_awready = '1' and S_AXI_wready = '1';
249         wait on clk until clk = '1';
250 
251         S_AXI_awvalid <= '0';
252         S_AXI_wvalid <= '0';
253 
254         wait until S_AXI_bvalid = '1';
255         wait on clk until clk = '1';
256 
257         S_AXI_awaddr <= BPF_CORE_INPUT_U32;
258         S_AXI_awvalid <= '1';
259 
260         S_AXI_wdata <= vec32(BPF_FRAME_POINTER);
261         S_AXI_wstrb <= "1111";
262         S_AXI_wvalid <= '1';
263 
264         wait until S_AXI_awready = '1' and S_AXI_wready = '1';
265         wait on clk until clk = '1';
266 
267         S_AXI_awvalid <= '0';
268         S_AXI_wvalid <= '0';
269 
270         wait until S_AXI_bvalid = '1';
271         wait on clk until clk = '1';
272 
273         -- Step 3: execute --
274 
275         S_AXI_awaddr <= BPF_CORE_CTRL_U32;
276         S_AXI_awvalid <= '1';
277 
278         S_AXI_wdata <= vec32("000000" & "0000"); -- reset = 0; sleep = 0; don't write
279         S_AXI_wstrb <= "1111";
280         S_AXI_wvalid <= '1';
281 
282         wait until S_AXI_awready = '1' and S_AXI_wready = '1';
283         wait on clk until clk = '1';
284 
285         S_AXI_awvalid <= '0';
286         S_AXI_wvalid <= '0';
287 
288         wait until S_AXI_bvalid = '1';
289         wait on clk until clk = '1';
290 
291         -- -- -- -- --
292 
293         wait;
294 
295     end process;
296 
297 end Behavioral;
```

G.3. Ejemplos de visualización de señales

Los tests de integración se comprueban manualmente observando el valor de las señales con un visor de ondas (p. ej. GTKWave). A continuación se muestran algunos ejemplos de comportamientos del procesador que deben ser verificados ante ciertas situaciones:

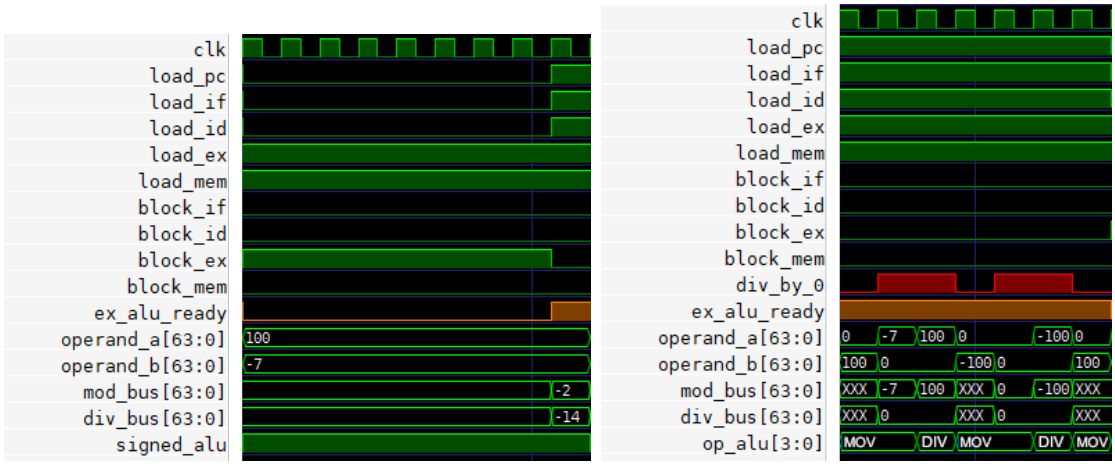
- La división es una operación multiciclo. En etapa EX debe bloquear el avance de etapas anteriores y permitir el de las posteriores, tal y como muestra la Fig. G.1a. No obstante, si la división es entre cero deja de ser multiciclo, como se ve en la Fig. G.1b.

- Como se menciona en la Sec. 3.1.1, los saltos incondicionales descartan la instrucción en etapa IF y los condicionales tomados también descartan la etapa ID. La Fig. G.2 corresponde al comienzo del test de control, donde aparece este comportamiento de los saltos y se puede observar el primer descarte obligatorio producido después de reiniciar el core.

- En caso de generarse una excepción el procesador devuelve el número de instrucción que la provocó. El programa del test de mapas está pensado para acabar en excepción leyendo de un puntero nulo. En la Fig. G.3 se puede comprobar que la instrucción causante es el último *load*.

- Una parte del test de IOMM tiene por objetivo asegurar el acceso en exclusión mutua al bus de memoria compartida. Un árbitro se encarga de otorgar acceso y decidir en caso de solicitudes simultáneas, priorizando al solicitante menos reciente para evitar inanición. En la Fig. G.4 se ve un caso de coincidencia de solicitudes y varios casos de cambios en la prioridad del árbitro.

- La Fig. G.5 muestra el proceso de carga del programa a la memoria de instrucciones y el arranque del core siguiendo el proceso explicado en la Sec. 4.2.2.



(a) División normal (multiciclo) (b) División entre cero

Figura G.1: Posibles etapas EX de una división

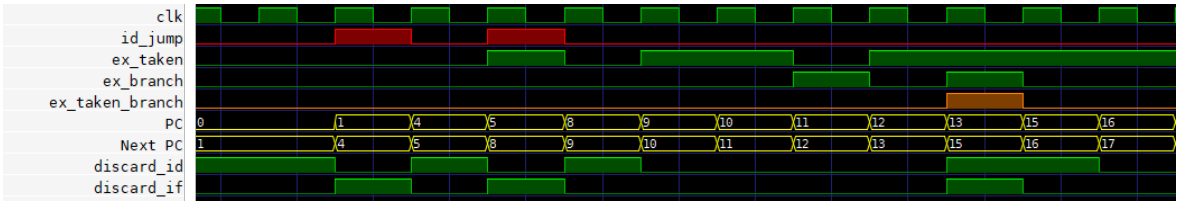


Figura G.2: Comparación en el número de etapas descartadas entre saltos incondicionales (*jump*) y condicionales(*branch*)

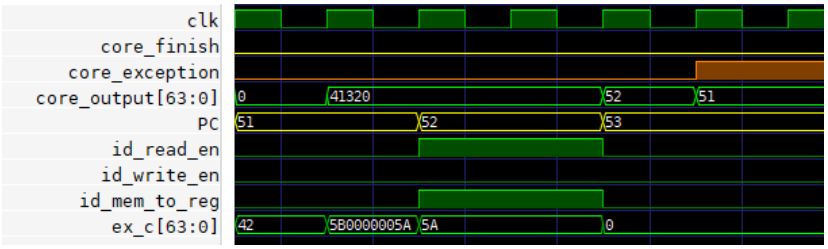


Figura G.3: Fin de ejecución por excepción en el último *load*

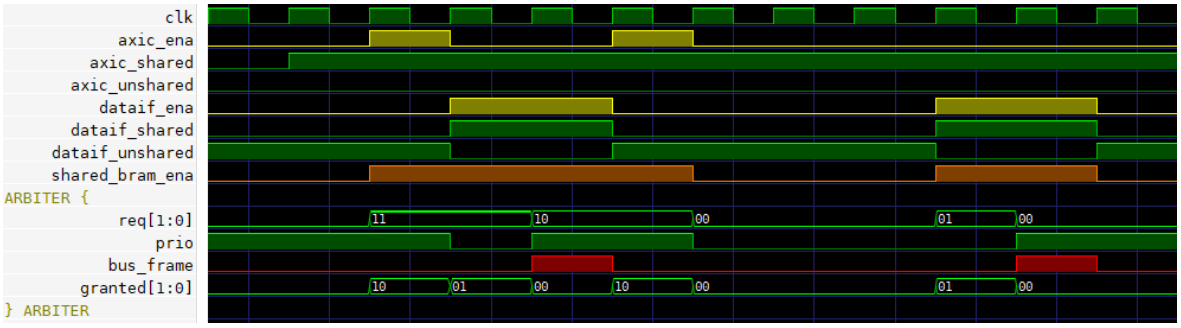


Figura G.4: Coincidencia en la petición de acceso al bus de memoria compartida y cambios de prioridad

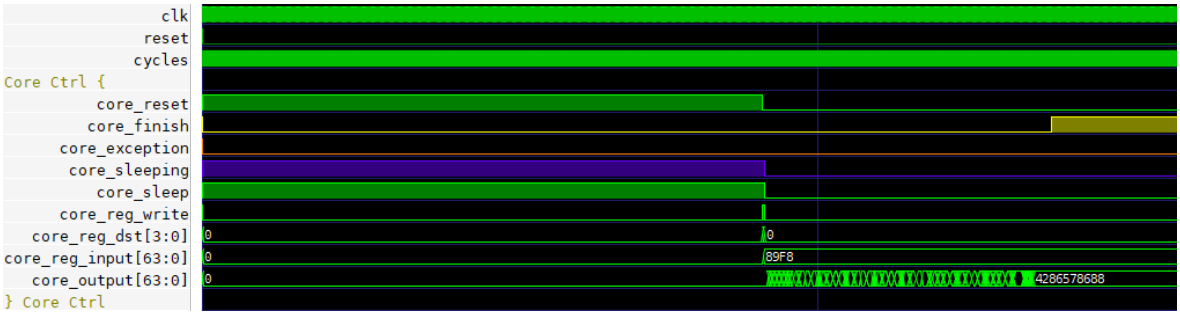


Figura G.5: Fases de carga, ejecución y fin de programa visualizadas con las señales del registro de control

Anexos H

Código de demostración

H.1. Biblioteca de interacción con el periférico

Listado H.1: Biblioteca para interactuar con el procesador BPF (`ebpf_lib.c`).

```c
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include <sys/_types.h>

/**
 * "platform.h" is automatically generated by Xilinx IDE from exported hardware.
 * It contains XPAR_BPF_AXI_PERIPHERAL_0_BASEADDR.
 */
#include "platform.h"

#include "xparameters.h"


// ---------------------------------------------------------------------------- //
//    AXI bus access                                                            //
// ---------------------------------------------------------------------------- //

uint8_t bpf_axi_read_8b(size_t addr)
{
    return *(volatile uint8_t *) (XPAR_BPF_AXI_PERIPHERAL_0_BASEADDR + addr);
}

uint16_t bpf_axi_read_16b(size_t addr)
{
    return *(volatile uint16_t *) (XPAR_BPF_AXI_PERIPHERAL_0_BASEADDR + addr);
}

uint32_t bpf_axi_read_32b(size_t addr)
{
    return *(volatile uint32_t *) (XPAR_BPF_AXI_PERIPHERAL_0_BASEADDR + addr);
}

uint64_t bpf_axi_read_64b(size_t addr)
{
    // This is will trigger two AXI transactions
    return *(volatile uint64_t *) (XPAR_BPF_AXI_PERIPHERAL_0_BASEADDR + addr);
}

void bpf_axi_write_8b(size_t addr, uint8_t value)
{
    *(volatile uint8_t *) (XPAR_BPF_AXI_PERIPHERAL_0_BASEADDR + addr) = value;
}

void bpf_axi_write_16b(size_t addr, uint16_t value)
{
    *(volatile uint16_t *) (XPAR_BPF_AXI_PERIPHERAL_0_BASEADDR + addr) = value;
}
```

```
49
50  void bpf_axi_write_32b(size_t addr, uint32_t value)
51  {
52      *(volatile uint32_t *) (XPAR_BPF_AXI_PERIPHERAL_0_BASEADDR + addr) = value;
53  }
54
55  void bpf_axi_write_64b(size_t addr, uint64_t value)
56  {
57      // This is will trigger two AXI transactions
58      *(volatile uint64_t *) (XPAR_BPF_AXI_PERIPHERAL_0_BASEADDR + addr) = value;
59  }
60
61
62  // ------------------------------------------------------------------------ //
63  //    Core control and program loading                                      //
64  // ------------------------------------------------------------------------ //
65
66  // Address space constants
67  enum bpf_peripheral_address
68  {
69      BPF_MEM_INST_BASE   = 0x0000,
70      BPF_MEM_PACKET_BASE = 0x8000,
71      BPF_MEM_STACK_BASE  = 0x8800,
72      BPF_MEM_SHARED_BASE = 0x9000,
73
74      // End Of Transaction
75      BPF_MEM_SHARED_FLUSH_BUFFER = 0x8FFF,
76
77      BPF_CORE_CTRL   = 0x8A00,
78      BPF_CORE_INPUT  = 0x8A08,
79      BPF_CORE_OUTPUT = 0x8A10,
80
81      BPF_MAP_BASE    = 0x8A18,
82
83      BPF_FRAME_POINTER = 0x89F8
84  };
85
86
87  struct bpf_core_ctrl_reg
88  {
89      unsigned reg_dst : 4;
90      unsigned reg_write : 1;
91      unsigned sleep : 1;
92      unsigned sleeping : 1;
93      unsigned exception : 1;
94      unsigned finish : 1;
95      unsigned reset : 1;
96      uint64_t __padding : 54;
97  };
98
99  /**
100  * @brief Reads control register from BPF_CORE_CTRL.
101  */
102 struct bpf_core_ctrl_reg bpf_core_get_ctrl_reg()
103 {
104     struct bpf_core_ctrl_reg ctrl_reg;
105     uint64_t value = bpf_axi_read_64b(BPF_CORE_CTRL);
106     memcpy(&ctrl_reg, &value, sizeof(uint64_t));
107     return ctrl_reg;
108 }
109
110 /**
111  * @brief Writes control register on BPF_CORE_CTRL.
112  */
113 void bpf_core_set_ctrl_reg(struct bpf_core_ctrl_reg ctrl_reg)
114 {
115     uint64_t value;
116     memcpy(&value, &ctrl_reg, sizeof(uint64_t));
117     bpf_axi_write_64b(BPF_CORE_CTRL, value);
118 }
119
120
121
```

```c
typedef uint64_t bpf_instruction_t;

/**
 * @brief Loads a program into instruction memory.
 *
 * @param program Instruction buffer.
 * @param length Size of buffer.
 *
 * @return Zero for a successful call. On error, -1 is returned.
 */
int bpf_load_program(bpf_instruction_t* program, int length)
{
    struct bpf_core_ctrl_reg ctrl_reg = bpf_core_get_ctrl_reg();
    if (!(ctrl_reg.finish || ctrl_reg.exception || ctrl_reg.sleeping))
        return -1;

    for (int i = 0; i < length; ++i)
        bpf_axi_write_64b(BPF_MEM_INST_BASE + 8*i, program[i]);

    return 0;
}

/**
 * @brief Resets processor state, sets initial parameters and
 *        starts program execution.
 */
void bpf_start_program()
{
    struct bpf_core_ctrl_reg ctrl_reg;

    // Start over
    ctrl_reg.reset = 1;
    bpf_core_set_ctrl_reg(ctrl_reg);

    // Load frame pointer
    ctrl_reg.reset = 0;
    ctrl_reg.sleep = 1;
    ctrl_reg.reg_write = 1;
    ctrl_reg.reg_dst = 10;
    bpf_core_set_ctrl_reg(ctrl_reg);
    bpf_axi_write_64b(BPF_CORE_INPUT, BPF_FRAME_POINTER);

    // Start execution
    ctrl_reg.sleep = 0;
    ctrl_reg.reg_write = 0;
    bpf_core_set_ctrl_reg(ctrl_reg);
}

/**
 * @brief Sets 'sleep' flag and awaits for processor to stop.
 */
void bpf_sleep_program()
{
    struct bpf_core_ctrl_reg ctrl_reg = bpf_core_get_ctrl_reg();
    if (ctrl_reg.finish || ctrl_reg.exception || ctrl_reg.sleeping)
        return;

    ctrl_reg.reset = 0;
    ctrl_reg.sleep = 1;
    ctrl_reg.reg_write = 0;
    bpf_core_set_ctrl_reg(ctrl_reg);

    // Await until processor stops
    while (!(ctrl_reg.finish || ctrl_reg.exception || ctrl_reg.sleeping))
        ctrl_reg = bpf_core_get_ctrl_reg();
}

/**
 * @brief Sets 'sleep' flag for processor to resume execution.
 *        Does nothing if processor is already awaken.
 *        Fails when trying to resume a completed program.
 *
 * @return Zero for a successful call. On error, -1 is returned.
```

```
 */
int bpf_awake_program()
{
    struct bpf_core_ctrl_reg ctrl_reg = bpf_core_get_ctrl_reg();
    if (ctrl_reg.finish || ctrl_reg.exception)
        return -1;

    ctrl_reg.sleep = 0;
    bpf_core_set_ctrl_reg(ctrl_reg);

    return 0;
}


enum bpf_program_end_cause
{
    BPF_FINISH = 0,
    BPF_EXCEPTION = -1
};

/**
 * @brief Blocks execution until the BPF program finishes.
 *
 * @return 'BPF_FINISH' or 'BPF_EXCEPTION', depending on
 *         what caused termination.
 */
int bpf_await_program()
{
    struct bpf_core_ctrl_reg ctrl_reg = bpf_core_get_ctrl_reg();
    while (!(ctrl_reg.finish || ctrl_reg.exception || ctrl_reg.sleeping))
        ctrl_reg = bpf_core_get_ctrl_reg();

    if (ctrl_reg.finish)
        return BPF_FINISH;
    else
        return BPF_EXCEPTION;
}

/**
 * @brief Reads program result. In case of exception, it returns the PC value
 *        of the instruction that generated it.
 */
uint64_t bpf_core_get_program_result()
{
    return bpf_axi_read_64b(BPF_CORE_OUTPUT);
}


// ---------------------------------------------------------------------------- //
//    BPF maps management                                                       //
// ---------------------------------------------------------------------------- //

enum bpf_map_type
{
    BPF_MAP_TYPE_ARRAY
};

struct bpf_map_entry
{
    unsigned base_ptr : 12;
    unsigned key_size : 2;
    unsigned val_size : 2;
    unsigned max_entries : 15;
    unsigned valid : 1;
};

// Returns map entry associated with that ID. Should not be called by user.
struct bpf_map_entry _bpf_map_get_entry(size_t id)
{
    struct bpf_map_entry map_reg;
    uint32_t value = bpf_axi_read_32b(BPF_MAP_BASE + (4 * id));
    memcpy(&map_reg, &value, sizeof(uint32_t));
    return map_reg;
```

```
}

// Sets map entry associated with that ID. Should not be called by user.
void _bpf_map_set_entry(struct bpf_map_entry map_reg, size_t id)
{
    uint32_t value;
    memcpy(&value, &map_reg, sizeof(uint32_t));
    bpf_axi_write_32b(BPF_MAP_BASE, value + (4 * id));
}

// With only two maps, allocator can be implemented as a state machine.
enum bpf_map_alloc_state
{
    BPF_MAP_ALLOC_STATE_HALF_TOP,
    BPF_MAP_ALLOC_STATE_HALF_BOT,
    BPF_MAP_ALLOC_STATE_FULL,
    BPF_MAP_ALLOC_STATE_EMPTY
};

const unsigned BPF_MAP_MAX_SIZE = 3584 * 8; // bytes
unsigned bpf_map_next_ptr_v = 0;
enum bpf_map_alloc_state bpf_map_alloc_state_v = 0;

// Compacts size in bytes to its base 2 logarithm.
unsigned _bpf_compact_size(unsigned sz)
{
    switch (sz) {
        case 8:  return 0;
        case 16: return 1;
        case 32: return 2;
        case 64: return 3;
        default: return 0;
    }
}

// Creates a bit mask from a compacted size.
uint64_t _bpf_mask_from_size(unsigned sz)
{
    switch (sz) {
        case 0: return 0x00000000000000FF;
        case 1: return 0x000000000000FFFF;
        case 2: return 0x00000000FFFFFFFF;
        case 3: return 0xFFFFFFFFFFFFFFFF;
        default: return 0;
    }
}

/**
 * @brief Loads a map into the map unit with the specified configuration.
 *        Fails if it is not possible to create a new map.
 *
 * @param type Must be BPF_MAP_TYPE_ARRAY.
 * @param key_size Size in bytes of element key.
 * @param val_size Size in bytes of element value.
 * @param max_entries Max number of entries.
 *
 * @return ID of the generated map for a successful call.
 *         On error, -1 is returned.
 */
int bpf_create_map(enum bpf_map_type type, unsigned key_size,
        unsigned val_size, unsigned max_entries)
{
    struct bpf_map_entry map_reg;
    unsigned aux_next_ptr, id;
    map_reg.base_ptr = bpf_map_next_ptr_v / 8;

    if (type != BPF_MAP_TYPE_ARRAY)
        return -1;

    // next base pointer, aligned to 8 bytes
    aux_next_ptr = bpf_map_next_ptr_v + (val_size * max_entries / 8) * 8;
    if (aux_next_ptr > BPF_MAP_MAX_SIZE)
        return -1;
```

```
341
342     switch (bpf_map_alloc_state_v) {
343     case BPF_MAP_ALLOC_STATE_EMPTY:
344
345         id = 0;
346         bpf_map_next_ptr_v = aux_next_ptr;
347         if (bpf_map_next_ptr_v == BPF_MAP_MAX_SIZE)
348             bpf_map_alloc_state_v = BPF_MAP_ALLOC_STATE_FULL;
349         else
350             bpf_map_alloc_state_v = BPF_MAP_ALLOC_STATE_HALF_TOP;
351
352         break;
353     case BPF_MAP_ALLOC_STATE_HALF_TOP:
354
355         id = 1;
356         bpf_map_next_ptr_v = aux_next_ptr;
357         bpf_map_alloc_state_v = BPF_MAP_ALLOC_STATE_FULL;
358
359         break;
360     case BPF_MAP_ALLOC_STATE_HALF_BOT:
361
362         id = 0;
363         bpf_map_next_ptr_v = aux_next_ptr;
364         bpf_map_alloc_state_v = BPF_MAP_ALLOC_STATE_FULL;
365
366         break;
367     case BPF_MAP_ALLOC_STATE_FULL:
368         return -1;
369     default:
370         return -1;
371     }
372
373     map_reg.valid = 1;
374     map_reg.key_size = _bpf_compact_size(key_size);
375     map_reg.val_size = _bpf_compact_size(val_size);
376     map_reg.max_entries = max_entries;
377     _bpf_map_set_entry(map_reg, id);
378
379     return id;
380 }
381
382
383 /**
384  * @brief Removes a map from the map unit by its ID.
385  *        Fails if map does not exist.
386  *
387  * @param id Targeted map.
388  *
389  * @return Zero for a successful call. On error, -1 is returned.
390  */
391 int bpf_delete_map(unsigned id)
392 {
393     struct bpf_map_entry map_reg;
394
395     if (id >= 2)
396         return -1;
397
398     switch (bpf_map_alloc_state_v) {
399     case BPF_MAP_ALLOC_STATE_EMPTY:
400         return -1;
401     case BPF_MAP_ALLOC_STATE_HALF_TOP:
402
403         if (id != 0)
404             return -1;
405
406         bpf_map_next_ptr_v = 0;
407         bpf_map_alloc_state_v = BPF_MAP_ALLOC_STATE_EMPTY;
408
409         break;
410     case BPF_MAP_ALLOC_STATE_HALF_BOT:
411
412         if (id != 1)
413             return -1;
```

```
414
415         bpf_map_next_ptr_v = 0;
416         bpf_map_alloc_state_v = BPF_MAP_ALLOC_STATE_EMPTY;
417
418         break;
419     case BPF_MAP_ALLOC_STATE_FULL:
420
421         map_reg = _bpf_map_get_entry(id);
422
423         if (id == 1)
424         {
425             bpf_map_next_ptr_v = map_reg.base_ptr * 8;
426             bpf_map_alloc_state_v = BPF_MAP_ALLOC_STATE_HALF_TOP;
427         }
428         else // id == 0
429         {
430             bpf_map_next_ptr_v = 0;
431             bpf_map_alloc_state_v = BPF_MAP_ALLOC_STATE_HALF_BOT;
432         }
433         break;
434     default:
435         return -1;
436     }
437
438     map_reg.valid = 0;
439     _bpf_map_set_entry(map_reg, id);
440
441     return 0;
442 }
443
444 /**
445  * @brief Looks for an element inside a map.
446  *
447  * @param id Targeted map.
448  * @param key Search key for an element.
449  *
450  * @return Memory address of searched element.
451  *         When element is not found, NULL is returned.
452  */
453 void *bpf_lookup_elem(unsigned id, uint64_t key)
454 {
455     struct bpf_map_entry map_reg;
456     uint64_t key_masked;
457     size_t elem_ptr;
458
459     if (id >= 2)
460         return NULL;
461
462     map_reg = _bpf_map_get_entry(id);
463
464     if (!map_reg.valid)
465         return NULL;
466
467     key_masked = key & _bpf_mask_from_size(map_reg.key_size);
468
469     if (key_masked >= map_reg.max_entries)
470         return NULL;
471
472     elem_ptr = (map_reg.base_ptr * 8) + (key_masked << map_reg.val_size);
473
474     return (void *) (XPAR_BPF_AXI_PERIPHERAL_0_BASEADDR + elem_ptr);
475 }
```

H.2. Ejemplo de programa

Listado H.2: Ejemplo de programa principal (`main.c`).

```c
#include <stddef.h>
#include <stdint.h>
#include <string.h>
#include <stdbool.h>
#include <sys/_intsup.h>

#include "xil_printf.h"

#include "ebpf_lib.h"

#define PRINT_ENABLE 1

#define PRINT(fmt, ...) \
{ \
    if (PRINT_ENABLE) \
        xil_printf(fmt, ##__VA_ARGS__); \
}

bpf_instruction_t program[] =
    {
        #include "programs/test_mem.txt"
    };

int main()
{
    int end_cause;
    uint64_t result;

    init_platform();

    PRINT("Loading program...\n\r");

    if (bpf_load_program(program, sizeof(program)/sizeof(bpf_instruction_t)) < 0)
    {
        PRINT("FAIL\n\r");
        goto exit_point;
    }

    PRINT("OK\n\r");
    PRINT("Start of execution\n\r");

    bpf_start_program();

    end_cause = bpf_await_program();
    result = bpf_core_get_program_result();

    if (end_cause == BPF_FINISH)
    {
        PRINT("Program finished with result: %llx\n\r", result);
    }
    else {
        PRINT("Program threw an exception at PC = %llu\n\r", result);
    }

exit_point:
    cleanup_platform();
    return 0;
}
```